**OOP Interview Questions with Solutions**

Q.1 Abstraction

Abstraction is the process of hiding the implementation details and showing only the essential features of the object to the user. It helps in managing complexity by providing a simpler and more organized view.

**Code Example:**

abstract class Vehicle {

abstract void startEngine(); // Abstract method (no implementation)

void stopEngine() {

System.out.println("Engine is off."); // Concrete method

}

}

class Car extends Vehicle {

@Override

void startEngine() {

System.out.println("Car engine started."); // Implementation details are hidden

}

}

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | Feature |  | | --- | --- | | | Interface |  | | --- | --- | | | Abstract Class | | --- | |  | |
| Purpose | |  |  | | --- | --- | | Defines a contract for what a class can do. A class can implement multiple interfaces. |  | | |  | | --- | | Provides a common base for related classes. A class can inherit from only one abstract class. | |  | |
| Methods | |  |  | | --- | --- | | All methods are implicitly `public` and `abstract`. Can have default, static, and private methods from Java 8 onwards. |  | | |  | | --- | | Can have abstract methods (without implementation) and concrete methods (with implementation). | |  | |
| |  |  | | --- | --- | | Variables |  | | |  |  | | --- | --- | | Variables are implicitly `public`, `static`, and `final`. |  | | |  | | --- | | Can have instance variables, static variables, and final variables. | |  | |
| |  |  | | --- | --- | | Inheritance |  | | |  |  | | --- | --- | | A class uses the `implements` keyword to implement an interface. |  | | A class uses the `extends` keyword to inherit from an abstract class. |

Q.2 Interface & Abstract class

Q.3 polymorphism

Polymorphism is the ability of an object to take on many forms. It allows objects of different classes to be treated as objects of a common superclass. The most common types are compile-time (method overloading) and runtime (method overriding).

**Code Example (Runtime Polymorphism - Method Overriding):**

class Animal {

public void makeSound() {

System.out.println("The animal makes a sound.");

}

}

class Cat extends Animal {

@Override

public void makeSound() {

System.out.println("The cat meows.");

}

}

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("The dog barks.");

}

}

public class Main {

public static void main(String[] args) {

Animal myCat = new Cat(); // Animal reference, Cat object

Animal myDog = new Dog(); // Animal reference, Dog object

myCat.makeSound(); // Outputs: The cat meows.

myDog.makeSound(); // Outputs: The dog barks.

}

Q.4 Method overriding

Method overridingis a feature that allows a subclass to provide a specific implementation of a method that is already provided by its superclass. The method in the subclass must have the same name, parameters, and return type as the superclass method.

**Code Example:**

class Parent {

void show() {

System.out.println("Parent's show() method.");

}

}

class Child extends Parent {

@Override

void show() {

System.out.println("Child's show() method (overridden).");

}

}

Q.5 “IS-A” vs “HAS-A”

“IS-A” (Inheritance)\*\* signifies that an object is a type of another. For example, a `Car` IS-A `Vehicle`.

“HAS-A” (Composition)\*\* signifies that a class has an instance of another class. For example, a `Car` HAS-A `Engine`.

**Code Example:**

// IS-A relationship (Inheritance)

class Vehicle {}

class Car extends Vehicle {} // Car IS-A Vehicle

// HAS-A relationship (Composition)

class Engine {}

class Car {

private Engine engine; // Car HAS-A Engine

public Car() {

this.engine = new Engine();

}

}

Q.6 Use of Inheritance

Inheritance is used for:

* \*\*Code Reusability:\*\* It allows a subclass to reuse the fields and methods of the superclass.
* \*\*Polymorphism:\*\* It is a prerequisite for achieving runtime polymorphism (method overriding).
* \*\*Logical Organization:\*\* It helps to create a clear and logical hierarchy of classes.

**Code Example:**

class Shape {

String color;

void draw() {

System.out.println("Drawing a " + color + " shape.");

}

}

class Circle extends Shape {

int radius;

void calculateArea() {

System.out.println("Area of circle: " + (3.14 \* radius \* radius));

}

}

Q.7 Dynamic Binding

Dynamic binding\*\*, also known as late binding or runtime polymorphism, refers to the process of linking a method call to its code at runtime. It is the core concept behind method overriding, where the specific method to be executed is determined based on the actual object type, not the reference type.

**Code Example:**

class A {

void display() { System.out.println("Class A"); }

}

class B extends A {

@Override

void display() { System.out.println("Class B"); }

}

// In main method:

A obj = new B(); // A is the reference type, B is the object type

obj.display(); // The method from B is called at runtime

Q.8 Constructor chaining

Constructor chaining is the process of calling one constructor from another within the same class (`this()`) or from a parent class (`super()`).

**Code Example:**

class Parent {

Parent() {

System.out.println("Parent's constructor.");

}

}

class Child extends Parent {

Child() {

super(); // Calls the parent's constructor

System.out.println("Child's default constructor.");

}

Child(int x) {

this(); // Calls the default constructor of the same class

System.out.println("Child's parameterized constructor.");

}

}

Q.9 Implementation Of Encapsulation

Encapsulation bundles data (variables) and methods that operate on the data into a single unit (a class). It is implemented by making variables `private` and providing `public` getter and setter methods for controlled access.

**Code Example:**

class BankAccount {

private double balance; // Data is private

public double getBalance() { // Public getter

return balance;

}

public void deposit(double amount) { // Public method for controlled access

if (amount > 0) {

this.balance += amount;

System.out.println("Deposit successful.");

}

}

}

Q.10 super keyword

The `super` keyword refers to the immediate parent class object. It's used to access a parent's overridden method, access a parent's field, or invoke a parent's constructor.

**Code Example:**

class Parent {

String message = "Hello from Parent.";

void display() {

System.out.println("Parent's method.");

}

}

class Child extends Parent {

String message = "Hello from Child.";

void display() {

super.display(); // Calls parent's method

System.out.println("Child's method.");

System.out.println("Parent message: " + super.message); // Accesses parent's field

}

}