Object-Oriented Programming (OOP) Interview Guide for Fresh B.Tech Students

Introduction

This document provides a set of interview questions and answers on Object-Oriented Programming (OOP) concepts using Java, targeted at fresh B.Tech students. The aim is to evaluate their understanding of OOP principles with real-life examples and code snippets.

1. What is a Class?

Question: Can you explain what a class is in OOP?

Answer: A class is a blueprint for creating objects. It defines a datatype by bundling data (attributes) and methods (functions) that work on the data into a single unit.

Follow-up Question: Can you provide a real-life example?

Real-life Example: Consider a class `Car` which defines attributes like color, brand, and methods like `drive()`, `stop()`.

```
public class Car {
    String color;
    String brand;

    void drive() {
        System.out.println(brand + " is driving.");
    }

    void stop() {
        System.out.println(brand + " has stopped.");
    }
}
```

2. What is an Object?

Question: What is an object in OOP?

Answer: An object is an instance of a class. It represents a real-world entity with state and behavior defined by the class.

Follow-up Question: Can you create an object from the `Car` class and use its methods?

Real-life Example: A specific car like a red Toyota.

```java
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.color = "Red";
        myCar.brand = "Toyota";
        myCar.drive();
        myCar.stop();
    }
}
```

3. What is Encapsulation?

Question: What is encapsulation?

Answer: Encapsulation is the mechanism of wrapping data (variables) and code (methods) together as a single unit, restricting direct access to some components.

Follow-up Question: Why is encapsulation important?

Real-life Example: Think of a capsule that contains medicine inside it, preventing exposure to its contents.

```java
public class Person {
    private String name;
    private int age;

    // Getter and Setter methods
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

4. What is Polymorphism?

Question: What does polymorphism mean?

Answer: Polymorphism means "many shapes". It allows one interface to be used for a general class of actions, where the specific action is determined by the exact nature of the situation.

Follow-up Question: Can you demonstrate polymorphism with method overloading and overriding?

Real-life Example: A person can be a teacher, a father, and a friend in different contexts.

Method Overloading (Compile-Time Polymorphism):

```
public class MathUtils {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }
}
```

Method Overriding (Run-Time Polymorphism):

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.sound();
    }
}
```

5. What is Abstraction?

Question: Can you explain abstraction?

Answer: Abstraction is the concept of hiding complex implementation details and showing only the essential features of an object.

Follow-up Question: How do you achieve abstraction in Java?

Real-life Example: Using an ATM machine without knowing its internal mechanics.

```
abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a Circle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle();
        shape.draw();
    }
}
```

Follow-up Question: What is the difference between abstraction and encapsulation?

Answer: Abstraction focuses on hiding the complex implementation details and showing only the necessary features, while encapsulation is about bundling the data and methods that operate on the data within a single unit and restricting access to some of the object's components.

6. What is Inheritance?

Question: What is inheritance in OOP?

Answer: Inheritance is a mechanism where a new class is derived from an existing class, inheriting its attributes and methods.

Follow-up Question: Can you provide a real-life example?

Real-life Example: Consider a base class `Doctor` and derived classes `Cardiologist`, `Orthopedic`, `GeneralSurgeon`.

```
class Doctor {
    void treatPatient() {
        System.out.println("Treating patient");
    }
```

```
}

class Cardiologist extends Doctor {
    void treatHeart() {
        System.out.println("Treating heart");
    }
}

public class Main {
    public static void main(String[] args) {
        Cardiologist doc = new Cardiologist();
        doc.treatPatient();
        doc.treatHeart();
    }
}
```

7. What is Overloading?

Question: What is method overloading?

Answer: Overloading occurs when two or more methods in the same class have the same name but different parameters.

Follow-up Question: Can you show an example?

```
public class DisplayOverloading {
    public void display(int a) {
        System.out.println("Argument: " + a);
    }

    public void display(String a) {
        System.out.println("Argument: " + a);
    }

    public static void main(String[] args) {
        DisplayOverloading obj = new DisplayOverloading();
        obj.display(1);
        obj.display("Hello");
    }
}
```

8. What is Overriding?

Question: What is method overriding?

Answer: Overriding occurs when a derived class has a definition for one of the member functions of the base class, replacing the base function.

Follow-up Question: Can you show an example?

```
class Parent {
    void show() {
        System.out.println("Parent's show()");
    }
}

class Child extends Parent {
    void show() {
        System.out.println("Child's show()");
    }

    public static void main(String[] args) {
        Parent obj = new Child();
        obj.show();
    }
}
```

9. What is an Abstract Class?

Question: What is an abstract class?

Answer: An abstract class cannot be instantiated but can be subclassed. It can have abstract methods and non-abstract methods.

Follow-up Question: Can you provide an example?

```
abstract class Animal {
    abstract void sound();

    void sleep() {
        System.out.println("Animal is sleeping");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }

    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();
        dog.sleep();
    }
}
```

Follow-up Question: When would you use an abstract class over an interface?

Answer: Use an abstract class when you want to share code among several closely related classes. Use an interface when you want to specify that a class must implement certain methods, regardless of where the class is in the inheritance hierarchy.

10. What is an Interface?

Question: What is an interface in Java?

Answer: An interface is a reference type in Java, similar to a class, that can contain only abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Follow-up Question: Can you provide an example?

```
interface Animal {
    void eat();
    void travel();
}

class Mammal implements Animal {
    public void eat() {
        System.out.println("Mammal eats");
    }

    public void travel() {
        System.out.println("Mammal travels");
    }

    public static void main(String[] args) {
        Mammal m = new Mammal();
        m.eat();
        m.travel();
    }
}
```

Difference Between Abstract Class and Interface

Abstract Class:

1. **Implementation:** An abstract class can have both abstract methods (methods without a body) and concrete methods (methods with a body). This allows the abstract class to provide some default behavior.

```
  abstract class Animal {
      abstract void sound();
```

```
    void sleep() {
        System.out.println("Animal is sleeping");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
```

1. **Constructors:** Abstract classes can have constructors, which can be used to initialize fields of the class.

```
abstract class Vehicle {
    String type;

    Vehicle(String type) {
        this.type = type;
    }

    abstract void move();
}

class Car extends Vehicle {
    Car() {
        super("Car");
    }

    void move() {
        System.out.println("Car is moving");
    }
}
```

1. **Instance Variables:** Abstract classes can have instance variables.

```
abstract class Shape {
    String color;

    Shape(String color) {
        this.color = color;
    }

    abstract void draw();
}

class Circle extends Shape {
```

```
    Circle(String color) {
        super(color);
    }

    void draw() {
        System.out.println("Drawing a " + color + " circle");
    }
}
```

1. **Inheritance:** A class can inherit only one abstract class (single inheritance). This is because Java does not support multiple inheritance with classes.
2. **Access Modifiers:** Abstract class methods can have any visibility: public, protected, private.
3. **Use Case:** Use abstract classes when you have a base class that should provide some default behavior that subclasses can override or use as-is. It is suitable when there is a clear hierarchical relationship between classes.

Interface:

1. **Implementation:** Interfaces cannot have any implementation for methods (prior to Java 8). From Java 8 onwards, interfaces can have default and static methods with implementation.

```
interface Animal {
    void sound();

    default void sleep() {
        System.out.println("Animal is sleeping");
    }
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}
```

1. **Constructors:** Interfaces cannot have constructors. They cannot be used to create objects directly.
2. **Instance Variables:** Interfaces can only have constants (static final variables).

```
interface Constants {
    int MAX_SPEED = 120; // implicitly public, static, and final
}
```

1. **Inheritance:** A class can implement multiple interfaces (multiple inheritance). This is a way to overcome the limitation of single inheritance with classes.

```java
interface Animal {
    void eat();
}

interface Mammal {
    void walk();
}

class Dog implements Animal, Mammal {
    public void eat() {
        System.out.println("Dog eats");
    }

    public void walk() {
        System.out.println("Dog walks");
    }
}
```

1. **Access Modifiers:** Interface methods are implicitly public. All methods in an interface are abstract (prior to Java 8), and cannot have any other visibility.
2. **Use Case:** Use interfaces when you want to define a contract for what a class can do, without dictating how it should do it. They are ideal for defining capabilities that can be shared across different classes that do not share a parent-child relationship.

Summary Table:

Feature	Abstract Class	Interface
Method Implementation	Can have both abstract and concrete methods	Can have default and static methods (Java 8+)
Constructors	Can have constructors	Cannot have constructors
Instance Variables	Can have instance variables	Can only have static final variables
Inheritance	Supports single inheritance	Supports multiple inheritance
Access Modifiers	Methods can have any visibility	Methods are implicitly public
Use Case	Suitable for shared code among related classes	Suitable for defining capabilities across classes

Example Code for Abstract Class:

```java
abstract class Animal {
    abstract void sound();
```

```
    void sleep() {
        System.out.println("Animal is sleeping");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
```

Example Code for Interface:

```
interface Animal {
    void sound();

    default void sleep() {
        System.out.println("Animal is sleeping");
    }
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}
```

By understanding the differences between abstract classes and interfaces, you can make more informed decisions about which to use in your Java programs based on the design requirements and constraints.