1. **Can a class be derived (inherit) from two different independent classes?**

In Java, a class **cannot inherit** from two different **independent classes** because Java does not support multiple inheritance with classes. This restriction is in place to avoid the **Diamond Problem**, where ambiguity arises if two parent classes have methods with the same name. Instead, Java provides **interfaces** to achieve similar functionality, as a class can implement multiple interfaces.

1. **Can a class be derived (inherit) from a “final” class?**

No, a class cannot inherit from a final class in Java. The final keyword on a class means that it cannot be subclassed, ensuring that its implementation remains unchanged.

For example:

*final class FinalClass {*

*…*

*}*

*// This would cause a compilation error*

*class SubClass extends FinalClass {*

*…*

*}*

1. **What is polymorphism, how it works, and why is it needed?**

**Polymorphism** in Java refers to the ability of an object to take on multiple forms. It allows a single interface to represent different underlying forms (types).

**Types of Polymorphism:**

* **Compile-Time Polymorphism (Method Overloading)**: Achieved by defining multiple methods with the same name but different parameters in a class.
* **Run-Time Polymorphism (Method Overriding)**: Achieved when a subclass provides a specific implementation for a method already defined in its parent class.

**How it works:**

* **Method Overloading**: Determined at compile time based on the method signature.
* **Method Overriding**: The method to execute is determined at runtime based on the actual object type.

**Why is it needed:**

* **Code Reusability**: Polymorphism promotes reusability by allowing the same code to work with objects of different types.
* **Flexibility**: It enables developers to write generic code that can work with multiple object types.
* **Scalability**: It supports adding new functionality without changing the existing code.

Example of polymorphism:

*class Animal {*

*void sound() {*

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

*}*

*}*

*class Dog extends Animal {*

*@Override*

*void sound() {*

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

*}*

*}*

*public class Main {*

*public static void main(String[] args) {*

*Animal animal = new Dog(); // Polymorphism*

*animal.sound(); // Outputs: Dog barks*

*}*

*}*

1. **What is an inner (nested) class?**

An **inner class** is a class defined **within another class**. It is used to logically group classes that belong together and to increase encapsulation.

**Types of Inner Classes:**

* **Non-Static Inner Class** (**Member Inner Class**): Associated with an instance of the outer class.
* **Static Nested Class**: Can be instantiated without an outer class object.
* **Local Inner Class**: Defined inside a method or block.
* **Anonymous Inner Class**: A one-time use class without a name, usually used with interfaces or abstract classes.

Example:

*class OuterClass {*

*private String message = "Hello from Outer Class";*

*class InnerClass {*

*void display() {*

*System.out.println(message); // Access outer class's members*

*}*

*}*

*}*

*public class Main {*

*public static void main(String[] args) {*

*OuterClass outer = new OuterClass();*

*OuterClass.InnerClass inner = outer.new InnerClass();*

*inner.display();*

*}*

*}*

1. **What is an abstract class and why is it needed?**

An **abstract class** is a class declared with the abstract keyword. It **cannot be instantiated** and may contain both **abstract methods** (without implementation) and **concrete methods** (with implementation).

Why is it needed:

**Blueprint for Subclasses**: Provides a base for other classes to extend and define specific behaviors.

**Partial Implementation**: Allows defining some default behavior while enforcing the implementation of other behaviors by subclasses.

**Polymorphism**: Abstract classes enable polymorphism by allowing objects to be treated as instances of the parent abstract class.

Example:

*abstract class Animal {*

*abstract void sound(); // Abstract method*

*void sleep() { // Concrete method*

*System.out.println("Animal is sleeping");*

*}*

*}*

*class Dog extends Animal {*

*@Override*

*void sound() {*

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

*}*

*}*

1. **What is an “interface” class in Java?**

An **interface** in Java is a blueprint for a class that contains **abstract methods** (prior to Java 8) or **default methods** (from Java 8 onward). It is used to achieve **100% abstraction** (before Java 8) and **multiple inheritance**.

**Features:**

1. All methods are implicitly public and abstract (prior to Java 8).
2. Fields are public, static, and final by default.
3. A class can implement multiple interfaces, overcoming the limitation of single inheritance in classes.

**Why is it needed:**

* To define a contract for classes to implement.
* To achieve multiple inheritance.
* To decouple the implementation and definition of behavior.

Example:

*interface Animal {*

*void sound(); // Abstract method*

*}*

*class Dog implements Animal {*

*@Override*

*public void sound() {*

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

*}*

*}*

*public class Main {*

*public static void main(String[] args) {*

*Animal animal = new Dog();*

*animal.sound(); // Outputs: Dog barks*

*}*

*}*