Q1. What is an abstract class? Why cannot an abstract class be instantiated directly? Can an abstract class have a constructor? If yes, what access modifier is typically used for it?

An abstract class is a class that represents an incomplete abstraction and may contain abstract methods that must be implemented by subclasses. It cannot be instantiated directly because it may define behaviors without providing full implementations. An abstract class can have a constructor, which is used to initialize common state when a concrete subclass is created; this constructor is typically declared as protected to prevent direct instantiation while allowing subclass access.

Q2. What types of methods and member variables can an abstract class contain? List at least 4 types. Is it correct to say "an abstract class can only contain abstract methods"? Explain.

An abstract class can contain abstract methods, concrete (fully implemented) methods, instance variables, static variables, final variables, and static or instance initialization blocks. Therefore, it is incorrect to say that an abstract class can only contain abstract methods, because it can also provide shared implementation and state for its subclasses.

Q3. What are the default modifiers for methods and variables in an interface? Why can't an interface have a constructor?

In an interface, methods are public by default, and variables are implicitly public, static, and final. An interface cannot have a constructor because it is not intended to represent an object with state or a lifecycle; instead, it defines a contract that implementing classes must follow.

Q4. Can a class extend multiple classes in Java? Can a class implement multiple interfaces? Explain the syntax for each case.

A class cannot extend multiple classes in Java, but it can implement multiple interfaces. The syntax uses the extends keyword for inheriting from a single class and the implements keyword followed by a comma-separated list of interfaces for multiple interface implementations.

Q5. Explain the relationship between interfaces: can an interface extend another interface? If yes, what keyword is used? Can an interface extend multiple interfaces?

An interface can extend another interface using the extends keyword, and it can extend multiple interfaces at the same time. This allows one interface to combine multiple contracts into a larger abstraction without implementation inheritance.

Q6. Compare abstract classes and interfaces from the following three perspectives: (1) Design intent (what it represents), (2) Multiple inheritance support, (3) Constructors and member variables.

From a design intent perspective, an abstract class represents an “is-a” relationship with shared behavior and state, while an interface represents a capability or role. Regarding multiple inheritance, abstract classes do not support it, but interfaces do. In terms of constructors and member variables, abstract classes can have constructors and instance variables, whereas interfaces cannot have constructors and only allow constants.

Q7. Why does Java not allow multiple inheritance of classes but allows multiple implementation of interfaces? How does this design avoid the Diamond Problem?

Java disallows multiple inheritance of classes to avoid ambiguity caused by conflicting implementations, known as the Diamond Problem. It allows multiple interface implementation because interfaces traditionally do not contain state and, even with default methods, conflicts must be explicitly resolved by the implementing class, thus eliminating ambiguity.

Q8. Explain the difference between Aggregation and Composition. Focus on the difference in object lifetime management and provide one practical example for each.

Aggregation represents a “has-a” relationship where the contained object can exist independently of the container, such as a department aggregating employees who can exist without the department. Composition is a stronger form of ownership where the contained object’s lifecycle is tied to the container, such as a house composed of rooms that do not exist independently of the house.

Q9. In design patterns and SOLID principles, there is an important principle: "Favor composition over inheritance." Explain why composition is generally more flexible than inheritance.

Composition is generally more flexible than inheritance because it allows behavior to be changed at runtime by combining objects rather than relying on a fixed class hierarchy. It also reduces tight coupling and avoids the rigidity and fragility that can arise from deep inheritance structures.

Q10. Describe the three core characteristics of the Singleton design pattern. Why is the Static Inner Class implementation of Singleton thread-safe and efficient? How does this approach achieve lazy initialization?

The Singleton pattern ensures that a class has exactly one instance, provides a global access point to that instance, and controls the instance’s creation. The static inner class implementation is thread-safe and efficient because the inner class is not loaded until it is accessed, which guarantees lazy initialization, and class loading in Java is inherently thread-safe without requiring explicit synchronization.