Chuwa - Java Backend 1215 Assignment

Delin "Angelina" Liang

15 Dec 2025

Question 1. 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?

ANSWER

What is an abstract class?
An abstract class is a class that **cannot be instantiated** and **contains abstract methods** (methods that have no implementation and must be implemented by the subclasses). It is defined using the `abstract` keyword in Java.

Why cannot an abstract class be instantiated directly?
Because it contains abstract methods that have no implementations, which is considered incomplete for JVM and cannot create usable objects based on it.

Can an abstract class have a constructor? If yes, what access modifier is typically used for it?
Yes, an abstract class can have everything else as same as a normal Java class has i.e. `constructor`, `static variables` and `methods`.
The constructor is typically declared as `protected` so as to keep the security of the data: only subclasses can invoke it, and external code cannot attempt to instantiate the abstract class

Question 2. 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.

ANSWER

An abstract class can have everything else as same as a normal Java class has.
Example

- **Constructors**
- **Static variables**
- **Instance variables** (fields)
- **Abstract methods** (without implementation)
- **Non-abstract/Normal methods** (at least in Java 8)

Therefore, we can see that an abstract class is not restricted to only having abstract methods, so it is NOT correct to say "an abstract class can only contain abstract methods."

Question 3. What are the default modifiers for methods and variables in an interface? Why can't an interface have a constructor?

ANSWER

Default modifiers:

- methods: `public`
- variables: `public static final` (which makes it a constant)

Interfaces cannot have constructors because they cannot be instantiated. They are designed for defining behaviors only and do not hold instance variables, so there is no need to initialize objects through a constructor

Question 4. Can a class extend multiple classes in Java? Can a class implement multiple interfaces? Explain the syntax for each case.

ANSWER

In Java, a class **cannot extend multiple classes**, but it **can implement multiple interfaces**.

Multiple inheritance - extend multiple classes: is not allowed because it leads to ambiguity issues (the *diamond problem*). Which mean you can only

```
class Child extends Parent1 {
}
```

but you cannot

```
class Child extends Parent1, Parent2 {
}
```

Multiple inheritance - implement multiple interfaces: is allowed because interfaces only define what a class should do (method signatures), not how (method implementations). It is the implementing class's responsibility to provide all implementations. So you can do either

```
class Child implements InterfaceA, InterfaceB, InterfaceC {
}
```

or

```
class Child extends Parent implements Interface1, Interface2 {

}
```

Question 5. Explain the relationship between interfaces: can an interface extend another interface? If yes, what keyword is used? Can an interface extend multiple interfaces?

ANSWER

An interface can extend from another interface. It uses the keyword `extends` to use another interface.
An interface may also extend multiple interfaces at the same time. The reasoning is similar to Question 4 – interfaces define behavior without maintaining instance state, so multiple inheritance does not introduce ambiguity.
An example would be

```
interface A {
    void a();
}

interface B {
    void b();
}

interface C extends A, B {
    void c();
}
```

Question 6. 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.

ANSWER

- **Design intent**
  - Abstract classes: are used when there is some common behavior that all subclasses must **share**
  - Interfaces: are used when there is a set of behaviors that all subclasses must **support**, but the implementation details can vary
- **Multiple inheritance support**
  - Abstract classes: being considered as normal classes, do NOT support multiple inheritance
  - Interfaces: do support multiple inheritance
- **Constructors and member variables**
  - Abstract classes: can have everything else as same as a normal Java class, i.e. `constructor`, `static variables` and `methods`
  - Interfaces: CANNOT have constructors and instance variables. All variables are implicitly `public static final` constants, and methods are `public abstract` by default (with optional default/static methods in Java 8+)

Question 7. Why does Java not allow multiple inheritance of classes but allows multiple implementation of interfaces? How does this design avoid the Diamond Problem?

ANSWER

Java does not allow multiple inheritance of classes because it can lead to ambiguity and conflicting method implementation, which is also known as the *Diamond Problem*: If multiple parent classes define different versions of the same method or contain overlapping fields, and worst case the `Override` keyword is missing, the subclass would not know which version to inherit from. To avoid these kinds of issues, Java chooses to forbid multiple inheritance

But then, in cases where we do need some kind of "multiple inheritance" , Java allows this by letting a class to implement multiple interfaces. This is because that interfaces do not contain instance states and provide only method signatures without concrete implementations. This and Java's nature of requiring subclasses to resolve ambiguity explicitly eliminate the conflicts described above. This enables safe "multiple inheritance"and avoids the *Diamond Problem*

Question 8. Explain the difference between Aggregation and Composition. Focus on the difference in object lifetime management and provide one practical example for each

ANSWER

Aggregation:

- represents a weak **"has-a"** relationship
- Lifetime rule: part is independent of the whole, that is, when the owner object is destroyed, the associated part can continue to exist
  - Example: in the following code

```
class Student {}
class University {
    private Student student;
}
```

represents a sample of aggregation: even after a `University` object is gone/does not exist, the `Student` object is still available, like this:

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

    Student s = new Student("Alice");

    {
        University u = new University(s);
    }

    s.someMethodCall();  // Student still exists!
}
```

Composition:

- represents a strong **"part-of"** relationship
- Lifetime rule: parts are fully dependent and will die together with the owner
  - Example: in the following code

```
class Room {}
class Hotel {
    private Room room = new Room();
}
```

represents a sample of composition: since the creation of `Room`s depends on `Hotel`s, if the `Hotel` object is gone/does not exist, the `Room` object will also be gone, like this:

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

    Room r;

    {
        Hotel h = new Hotel(101);
        r = h.getRoom();
    }

    // Room r is invalid!
}
```

Question 9. In design patterns and SOLID principles, there is an important principle: "Favor composition over inheritance."Explain why composition is generally more flexible than inheritance

ANSWER

Composition is generally more flexible than inheritance in the following aspects:

- **Composition allows runtime behavior changes:** Inheritance is a behavior happening during compile-time, which means an object's behavior is determined the moment it is instantiated and cannot inherit from a different class later. However, composition allows such behavior since it is determined at run-time, letting you to change the component object at runtime, making behaviors highly adaptable
- **Composition avoids the tight coupling caused by inheritance**: Since inheritance forces subclasses to depend on the entire implementation of the parent class, whenever you need to make changes to the parent (or in some cases, one accidentally added bad designs for the parent), the subclasses was forced to change accordingly. While for composition, since parts depends only on the interface of the component—making the system less fragile and easier to maintain
- **Composition does not suffer from inheritance problems**: as mentioned, composition only depends on the interfaces of the component, and do not need to worry about the internal designs of the parent, therefore, it does not have problems of "rigid hierarchies"like inheritance does
- **Composition supports the Open/Closed Principle (OCP) better**: composition allows behavior to be extended through object arrangement rather than modifying existing code (which is what inheritance does, making the codes cleaner and more reusable

Question 10. 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?

ANSWER

The three core characteristics of the Singleton design pattern are:

- **A single instance** of the class is guaranteed to exist
- **A private constructor** to prevent direct instantiation from outside
- **A public static method (global access point)** that provides controlled access to that single instance

The Static Inner Class implementation is thread-safe because the JVM ensures that class loading and initialization occur exactly once and are inherently synchronized. This avoids the need for explicit synchronization, making it more efficient than synchronized methods or double-checked locking

Lazy initialization in this case is achieved because the Singleton instance is created only when the inner class is first accessed via getInstance(). Since that is where our instance is hold, this means that the instance is not created until it is actually needed