Core Java Interview Questions

Interface vs Abstract class

Interface: An interface is a reference type in Java, similar to a class, containing only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields or constructors.

Usage: It defines a contract that other classes must follow. An interface specifies what a class must do, but not how it does it.

Inheritance: A class can implement multiple interfaces, providing a way to achieve multiple inheritance-like behavior.

Members: An interface cannot have instance fields. All variables in an interface are implicitly public, static, and final.

Access Modifiers: Methods in an interface are implicitly public. From Java 9 onwards, interfaces can also have private methods.

Constructor: Interfaces cannot have constructors because they cannot be instantiated directly.

State: Interfaces cannot maintain state. They are purely abstract contracts that other classes must adhere to.

Abstract Class: An abstract class is a class that cannot be instantiated on its own and may contain abstract methods (methods without a body) as well as concrete methods (methods with a body).

Usage: It is used when you want to provide a common base class with some shared code (concrete methods) and some methods that must be implemented by subclasses (abstract methods).

Inheritance: A class can extend only one abstract class because Java does not support multiple inheritance for classes.

Members: An abstract class can have fields (member variables) and constructors. It can also have method implementations.

Access Modifiers: Abstract class methods can have any visibility (public, protected, private).

Constructor: An abstract class can have constructors, which can be called during the instantiation of subclasses.

State: Can have state (fields) and can maintain state information.

Summary:

- Use an abstract class when you want to share code among several closely related classes, and when you want to provide a base class that contains default behavior.

- Use an interface when you want to define a contract for what a class can do, but not how it does it, and when you want to take advantage of multiple inheritance.

Composition vs Aggregation vs Association?

Association:

Association is a relationship where all objects have their lifecycle and there is no ownership. It represents a "uses-a" relationship.

Usage: It is used when two classes need to communicate with each other but do not depend on each other.

Example: A teacher teaches a student. Both Teacher and Student objects can exist independently.

Aggregation:

Aggregation is a specialized form of Association where all objects have their lifecycle, but there is ownership. However, the child can exist independently of the parent.

Usage: It represents a "has-a" relationship, where the child can belong to multiple parents.

Example: A library has books. Books can exist independently of the library and can be associated with other libraries.

Composition:

Composition is a stronger form of Aggregation. It implies ownership and a strict lifecycle dependency between the parent and child objects. When the parent object is destroyed, the child objects are also destroyed.

Usage: It represents a "part-of" relationship, where the child cannot exist independently of the parent.

Example: A house has rooms. Rooms do not exist independently of a house. If the house is destroyed, the rooms are destroyed as well.

Summary:

Lifecycle Dependency:
Association: Independent lifecycle.
Aggregation: Independent lifecycle but ownership exists.
Composition: Dependent lifecycle; child objects cannot exist without the parent.

Ownership:
Association: No ownership.
Aggregation: Weak ownership.
Composition: Strong ownership.

Example:
Association: Teacher and Student.
Aggregation: Library and Books.
Composition: House and Rooms.

Overloading vs Overriding

Method Overloading:

Method overloading allows a class to have more than one method with the same name, but different parameters (different type, number, or both).

Usage: It is used to increase the readability of the program by defining multiple methods with the same name but different behaviors based on different parameters.

Compile-Time: It is resolved at compile-time.

Return Type: The return type can be the same or different, but overloading is based on the method signature (method name + parameter list).

Inheritance: Method overloading can occur within a single class and can also be applied to a method inherited from a superclass.

Method Overriding: Method overriding allows a subclass to provide a specific implementation of a method already defined in its superclass.

Usage: It is used to achieve runtime polymorphism and to modify the behavior of a method inherited from the superclass.

Run-Time: It is resolved at runtime.

Return Type: The return type must be the same or a subtype of the return type declared in the original method in the superclass (covariant return type).

Inheritance: Method overriding involves two classes that have an inheritance relationship (parent-child).

Annotations: The @Override annotation is used to indicate that a method is intended to override a method in the superclass.

Access Modifiers: The access level of the overriding method cannot be more restrictive than the overridden method.

Summary:

Purpose:
Overloading: To have multiple methods with the same name but different parameters to perform similar but distinct tasks.
Overriding: To change the implementation of an inherited method to provide specific behavior in a subclass.

Method Signature:
Overloading: Methods must have different parameter lists.
Overriding: Methods must have the same parameter list.

Binding:
Overloading: Compile-time (static binding).
Overriding: Run-time (dynamic binding).

Inheritance:
Overloading: Can occur within a single class or between superclass and subclass.
Overriding: Always occurs between superclass and subclass.

Return Type:
Overloading: Can be different.
Overriding: Must be the same or a subtype (covariant return type).

Can private method or static methods be overridden in Java?

Private Methods:

Scope: Private methods are only accessible within their declared class. They are not visible to subclasses.

Inheritance: Since private methods are not inherited by subclasses, they cannot be overridden.

Purpose: Private methods are meant for internal use within a class and are not intended to be modified by subclasses.

Static Methods:

Binding: Static methods are resolved at compile-time (static binding) rather than at runtime (dynamic binding).

Scope: Static methods belong to the class, not to instances of the class. They can be called on the class itself.

Inheritance: Although static methods can be inherited, they are not polymorphic. Thus, they cannot be overridden in the traditional sense, but they can be hidden.

Summary:

Private Methods: Cannot be overridden because they are not accessible or visible to subclasses.

Static Methods: Cannot be overridden in the traditional sense; they can be hidden by defining a method with the same signature in the subclass. Static method calls are resolved at compile-time based on the type of the reference, not the actual object.

Why is multiple inheritance not supported

1. **Diamond Problem:**

The Diamond Problem is a specific issue that arises in multiple inheritance when a class inherits from two classes that both inherit from a common superclass. This can lead to ambiguity and conflicts about which method or property to inherit from the common ancestor.

2. Simplicity and Maintainability:

Java was designed to be a simple and readable language. Allowing multiple inheritance would increase the complexity of the language, making it more difficult to learn, use, and maintain. By avoiding multiple inheritance, Java maintains a simpler and cleaner class hierarchy.

3. Interface Inheritance:

Java provides a way to achieve some benefits of multiple inheritance through interfaces. A class can implement multiple interfaces, which allows for a form of multiple inheritance without the issues related to the Diamond Problem. Interfaces define methods that must be implemented by the class, ensuring that there is no ambiguity about which method to use.

4. Code Reusability:

Java encourages composition over inheritance. Instead of inheriting from multiple classes, Java promotes the use of composition to achieve code reuse. This means that a class can have instances of other classes and delegate functionality to these instances.

Summary:

Avoids Ambiguity: Prevents the Diamond Problem and related ambiguities in the method or property inheritance.

Simplifies Design: Keeps the language simple and the class hierarchy clear.

Uses Interfaces: Achieves multiple inheritance of behavior through interfaces without the complexities of multiple inheritance of implementation.

Promotes Composition: Encourages composition over inheritance for code reuse, leading to more flexible and maintainable designs.

How String pool works?

How String Constant Pool Works:

String Literals: When you create a String using double quotes (e.g., String s1 = "Hello";), the JVM checks the String Constant Pool to see if an identical String already exists.

Reusing Strings: If an identical String exists in the pool, the reference to that existing String is returned, rather than creating a new instance.

New Strings: If the String does not exist in the pool, a new String object is created and added to the pool.

new Keyword: When you create a String using the new keyword (e.g., String s3 = new String("Hello");), a new String object is created in the heap memory, and it will not refer to any String in the pool even if an identical String exists there. To ensure that the new String is added to the pool, you can use the intern() method.

Benefits of String Constant Pool

Memory Efficiency: By reusing existing String instances, the String Constant Pool helps to save memory. This is especially useful when there are many identical String literals in the code.

Performance: Reusing String objects can improve performance because it reduces the overhead of creating and garbage collecting new String instances.

Summary:

String Literals: Automatically placed in the String Constant Pool.

new Keyword: Creates a new String object in the heap memory.

intern() Method: Ensures that the String is added to the pool and returns the reference from the pool if it exists.

Memory Efficiency and Performance: The String Constant Pool helps in conserving memory and enhancing performance by reusing existing String instances.

What are some Object class methods?

Key Methods of the Object Class:

public boolean equals(Object obj):
Purpose: Determines whether two objects are equal.
Default Implementation: Checks if the references point to the same object.
Commonly Overridden: Often overridden in classes to provide meaningful equality logic based on the content of the objects.

public int hashCode():
Purpose: Returns a hash code value for the object.
Default Implementation: Provides a unique integer based on the object's memory address.
Commonly Overridden: Often overridden to maintain the general contract of hashCode and equals.

public String toString():
Purpose: Returns a string representation of the object.
Default Implementation: Returns a string that includes the class name and the object's memory address.
Commonly Overridden: Often overridden to provide a meaningful string representation of the object's content.

protected Object clone() throws CloneNotSupportedException:
Purpose: Creates and returns a copy (clone) of the object.
Default Implementation: Creates a shallow copy of the object.
Commonly Overridden: Classes that need to support cloning must implement the Cloneable interface and override this method to provide the cloning logic.

public final void wait() throws InterruptedException:
Purpose: Causes the current thread to wait until another thread invokes notify() or notifyAll() on the same object.
Usage: Used in synchronization to coordinate actions between threads.

public final void wait(long timeout) throws InterruptedException:
Purpose: Causes the current thread to wait until either another thread invokes notify() or notifyAll(), or a specified amount of time has passed.

public final void wait(long timeout, int nanos) throws InterruptedException:
Purpose: Similar to the previous wait method but with additional precision for nanoseconds.

public final void notify():
Purpose: Wakes up a single thread that is waiting on this object's monitor.
Usage: Used in synchronization to coordinate actions between threads.

public final void notifyAll():
Purpose: Wakes up all threads that are waiting on this object's monitor.
Usage: Used in synchronization to coordinate actions between threads.

protected void finalize() throws Throwable:
Purpose: Called by the garbage collector before the object is garbage collected.
Usage: Used to perform cleanup operations before the object is removed from memory (not recommended for general use).

Why is wrapper class required?

1. Object Manipulation:
Collections Framework: The Java Collections Framework (like ArrayList, HashSet, HashMap, etc.) works with objects. Since primitive data types are not objects, they cannot be used directly with collections. Wrapper classes allow primitives to be converted into objects so that they can be stored in collections.

2. Autoboxing and Unboxing:
Convenience: Wrapper classes facilitate autoboxing (automatic conversion of a primitive type to its corresponding wrapper class) and unboxing (automatic conversion of a wrapper class to its corresponding primitive type). This feature was introduced in Java 5 to make the use of primitive types and wrapper classes seamless.

3. Utility Methods:
Conversion and Parsing: Wrapper classes provide utility methods to convert strings to their respective primitive types and vice versa. They also provide constants and utility methods for various operations.

4. Default Values in Collections:
Nullability: Wrapper classes can represent a null value, while primitive types cannot. This is useful when you need to represent the absence of a value in collections or other data structures.

5. Immutable and Thread-safe:
Immutable Objects: Wrapper classes create immutable objects, meaning their state cannot be changed once created. This immutability is useful in multithreading environments to ensure that objects cannot be modified by multiple threads simultaneously.

6. Using with Generics:
Generics: Java's generics only work with objects, not primitive types. Wrapper classes allow primitive types to be used with generics.

Can main() method be overloaded?

Yes, the main() method in Java can be overloaded like any other method. However, the JVM will only call the standard main method with the signature public static void main(String[] args) to start the program. Other overloaded versions of the main method will not be called automatically by the JVM but can be called manually from within the standard main method or from other parts of the code.

- Standard main Method: This is the entry point of the program, which the JVM calls when the program starts.

- Overloaded main Methods: These are additional methods with the same name but different parameter lists. They can be called from within the standard main method or from other parts of the code.

Summary:

- The main method can be overloaded like any other method in Java.
- The JVM only looks for the standard main method with the signature public static void main(String[] args) to start the program.
- Overloaded main methods will not be called automatically by the JVM but can be called explicitly from within the standard main method or other parts of the code.
- Overloading the main method can be useful for testing purposes or to demonstrate method overloading concepts.

Why are Strings immutable?

Strings in Java are immutable, meaning that once a String object is created, it cannot be changed. There are several important reasons for this design decision, which contribute to the performance, security, and overall robustness of the Java language.

1. Thread Safety:
Immutability: Immutable objects are inherently thread-safe because their state cannot be changed after they are created. This means that multiple threads can access and use String objects concurrently without any synchronization, leading to simpler and safer concurrent programming.

2. Security:
Security: Strings are frequently used to represent sensitive data such as usernames, passwords, and configuration settings. Making String objects immutable helps ensure that such data cannot be inadvertently or maliciously modified once it is created.

3. String Pooling:
Memory Efficiency: Java uses a string constant pool to store unique String objects. When a string literal is used, the JVM checks the pool to see if an identical string already exists. If it does, the existing string is reused, saving memory. This pooling mechanism relies on the immutability of strings to ensure that the shared strings do not change.

4. Caching and Performance:
Caching: Because strings are immutable, they can safely cache their hash code. This makes repeated access to the hash code very fast, which is particularly useful in hash-based collections like HashMap and HashSet.

5. Design Simplicity:
Simplicity: Immutability simplifies the design and use of strings. Since strings cannot change, you don't need to worry about the side effects of modifying them. This leads to fewer bugs and more predictable code behavior.

6. Integrity and Security in Reflection and Class Loading:
Reflection and Class Loading: Strings are used in various critical Java APIs such as reflection, class loading, and other system-level operations. Immutability ensures that strings used in these contexts cannot be tampered with, preserving system integrity and security.

Summary:

- Thread Safety: Immutability makes strings inherently thread-safe.
- Security: Immutable strings ensure sensitive data cannot be modified.
- String Pooling: Immutability allows for efficient reuse of String objects.
- Caching and Performance: Immutable strings can cache their hash code for better performance.
- Design Simplicity: Immutable strings lead to simpler and more predictable code.
- Integrity and Security in Reflection and Class Loading: Immutability ensures the reliability of critical system operations involving strings.

Can Abstract class have main method?

Yes, an abstract class in Java can have a main method. In Java, the main method is just a static method, and no restrictions are preventing it from being included in an abstract class.

Summary:

- An abstract class can have a main method because the main method is static, and static methods can be included in abstract classes.
- The main method in an abstract class can be used to test the class or for any other purpose, just like in any other class.
- You can also have a main method in a concrete subclass, which can call the methods of the abstract class or implement its own functionality.

String, StringBuffer, StringBuilder

String:

Immutability: String objects are immutable, meaning once a String object is created, it cannot be changed. Any modification to a String results in the creation of a new String object.

Performance: Because of its immutability, performing many modifications to a String can be inefficient, as it involves creating multiple String objects.

Thread Safety: String objects are thread-safe because they are immutable.

StringBuffer:

Mutability: StringBuffer objects are mutable, meaning you can modify the contents of a StringBuffer without creating a new object.

Performance: It is more efficient than String when performing many modifications (like appending or inserting) because it doesn’t involve creating new objects with each modification.

Thread Safety: StringBuffer is synchronized, which means it is thread-safe but might be slower due to synchronization overhead.

StringBuilder:

- Mutability: Similar to StringBuffer, StringBuilder objects are mutable and allow modifications without creating new objects.
- Performance: Generally faster than StringBuffer because it is not synchronized, making it more suitable for single-threaded scenarios.
- Thread Safety: StringBuilder is not synchronized, so it is not thread-safe. It is best used in a single-threaded context where thread safety is not a concern.

equals vs ==

== Operator

- Purpose: Checks for reference equality.
- Usage: Determines if two references point to the same object in memory.
- Behavior: It does not compare the contents of the objects; it only checks if the two references are identical.

equals Method:

- Purpose: Checks for logical equality.
- Usage: Determines if two objects are logically equivalent based on the class's equals method implementation.
- Behavior: The equals method should be overridden in a class to provide meaningful comparison logic for the contents of the objects.

Key Points:

Primitive Types: For primitive types (like int, char, etc.), == is used to compare values directly. equals does not apply to primitives because it is a method and primitives don't have methods.

String: The String class overrides the equals method to compare the actual contents of the strings, so s1.equals(s2) compares the characters in the strings.

Custom Classes: When using custom classes, you should override the equals method to define how objects of that class are compared. If you do not override it, the default implementation (inherited from Object) will be used, which is equivalent to ==.

final, finally, finalize

final

- Purpose: Defines constants, and prevents method overriding, and inheritance.
- Usage:
- Final Variables: When a variable is declared as final, its value cannot be changed once it is initialized.
- Final Methods: When a method is declared as final, it cannot be overridden by subclasses.
- Final Classes: When a class is declared as final, it cannot be subclassed.

finally

- Purpose: Ensures that a block of code is executed after a try block, regardless of whether an exception is thrown or not.
- Usage: It is used in exception handling to execute code that must run whether or not an exception occurs, such as closing resources.

finalize

- Purpose: Allows an object to perform cleanup before it is reclaimed by the garbage collector.
- Usage: The finalize method is called by the garbage collector on an object when there are no more references to it. This is used to release resources or perform cleanup operations.
- Important Note: The use of finalize is generally discouraged in favor of other resource management techniques (like try-with-resources), as it is unpredictable and may not be called promptly.

Summary

- final: Used for constants, preventing method overriding, and preventing inheritance.
- finally: Used to ensure code execution after a try block, regardless of whether an exception occurs.
- finalize: A method that allows for cleanup before an object is garbage-collected, but it is generally considered unreliable and less preferred in modern Java programming.

What is marker interface?

In Java, a marker interface is a special type of interface that does not contain any methods or fields. Its primary purpose is to indicate that a class implementing this interface has a certain property or should be treated in a special way by the Java runtime or other libraries.

Purpose of Marker Interfaces

Marker interfaces are used to convey metadata or specific behavior characteristics to the Java compiler or runtime system. They can be used to:

- Indicate Special Behavior: Inform other parts of a system that the implementing class should be treated differently. For example, the Serializable interface is a marker interface used to indicate that objects of the class can be serialized.
- Provide Metadata: Serve as a way to attach metadata to classes. For instance, the Cloneable interface indicates that a class supports cloning.

How Marker Interfaces Work

Marker interfaces typically do not have any impact on their own. Instead, they are used in conjunction with reflection or specific library mechanisms that check if a class implements a particular marker interface to determine behavior.

For example, the Java ObjectOutputStream class uses reflection to check if a class implements the Serializable marker interface before attempting to serialize an object.

Modern Alternatives:

While marker interfaces are still used, modern Java often employs annotations as an alternative. Annotations can provide more flexibility and are generally preferred for adding metadata to classes, methods, or fields. For example, you might use an annotation like @Entity in Java Persistence API (JPA) to mark a class as an entity.

Thank You!!