1. **What is cloning and what are the requirements for cloning in Java?**

Ans:

Cloning in Java refers to the process of creating a **duplicate object** with the same state as the original object. This is typically done using the clone() method provided by the **Object** class.

To enable cloning, a class must:

**Implement the Cloneable interface** (marker interface).

**Override the clone() method** from the Object class.

1. **What are the different types of cloning in Java?**

Ans:

There are **two types** of cloning in Java:

**Shallow Cloning**: This creates a new object but **does not copy nested objects deeply**. References of nested objects are **shared** between the original and cloned objects. It is achieved using super.clone().

**Deep Cloning**: This creates a **fully independent copy** of the original object, including all nested objects. It requires manually cloning all mutable reference fields.

1. **Why do we need to implement Cloneable interface for cloning?**

Ans:

The Cloneable interface is a **marker interface** (it has no methods). If a class does **not** implement Cloneable and calls clone(), it throws **CloneNotSupportedException**. Implementing this interface tells the JVM that the class allows **field-by-field copying**.

1. **Can we override clone() method without implementing Cloneable?**

Ans:

Yes, we can override clone(), but if the class does not implement Cloneable, calling super.clone() will throw **CloneNotSupportedException**.

1. **What are the alternatives to cloning in Java?**

Ans:

Cloning has some limitations (e.g., needing Cloneable, not being flexible). Alternatives include:

**Copy Constructor**: Create a new object manually copying fields.

**Serialization and Deserialization**: Convert the object to a byte stream and back.

**Using ObjectMapper from Jackson** (for deep copying JSON objects).

1. **What is the difference between clone() and a copy constructor?**

Ans:

The clone() method requires implementing Cloneable, while a **copy constructor** does not. Clone does **not call constructors**, whereas a copy constructor **explicitly calls a constructor** for creating a new object. Deep copying is easier to implement using copy constructors.

Example of a Copy Constructor:

class Person {

String name;

Person(Person p) { // Copy constructor

this.name = p.name;

}

}

1. **Does Object class implement Cloneable interface?**

Ans:

No. Why ?

Java founders did not make Object class implements Cloneable. Had they made Object class implement Cloneable , each and every class in Java would have become Cloneable ( eligible for cloning). This is what they ( java founders) did not want. They wanted to give a choice to the programmer. If programmer would like to clone the object , then he can implement the class with Cloneable otherwise not.

1. **Why "clone()" method is protected in java.lang.Object?**

Ans:

if clone() method were public client code would invoke developer code's clone() method which developer might not agree ( he doesn't want client to invoke clone() method of his class).

The advantage of clone() method being protected is that by default ( if developer does not override clone() method of Object class), client code cannot invoke clone() of developer's class. Here if developer would to like to allow client to invoke clone() on his class, he can override it with "public" accessibility.

1. **Marker interface does not contain any contract i.e. abstract method, then what's the actual use of marker interface in Java?**

Ans:

A **marker interface** is an interface **without methods** that serves as a **tag** to indicate special behavior.

**Java uses marker interfaces** like Cloneable, Serializable, and Remote to allow certain operations only for classes that opt-in.

**They work using instanceof or reflection**, allowing Java to check if a class implements them before applying specific behaviour i.e. whether serializing object/s of the class or cloning object/s of the class.

1. **What is Java Reflection API?**

Ans:  
Java **Reflection API** is a feature that allows inspection and modification of classes, methods, fields, and constructors **at runtime**. It is part of the java.lang.reflect package.

1. **With reflection we come to know what are the methods ,constructors,fields etc. are there in a particular class. But these details we can also find out by referring to docs API. So what is the exact benefit of Reflection API?**

Ans:

We can check methods, constructors, and fields from Java Docs, so why use **Reflection**? The key benefit of Reflection is that it allows us to **inspect and manipulate classes, methods, and objects at runtime**, even if we don’t know them at compile-time.

1. **What are the main classes of Java Reflection API?**

Ans:

Java Reflection API consists of several key classes/interfaces in the java.lang.reflect package:

1. **Class<?>** - Represents a loaded class.
2. **Field** - Represents a class field (instance or static variable).
3. **Method** - Represents a method in a class.
4. **Constructor** - Represents a class constructor.
5. **Modifier** - Provides methods to check the access level (e.g., isPrivate(), isPublic()).
6. **Parameter** - Represents method parameters.
7. **How do you get the Class object of a Java class?**

Ans:  
There are **three ways** to obtain the Class object:

1. Using .class

Class<?> clazz1 = String.class;

2. Using getClass() method

String str = "Hello";

Class<?> clazz2 = str.getClass();

3. Using Class.forName()

Class<?> clazz3 = Class.forName("java.lang.String");

**Use Case:** Class.forName() is commonly used for **dynamic loading** of JDBC drivers and frameworks.

1. **How to invoke a private method using Reflection?**

Ans:

Use Method.invoke() along with setAccessible(true)

1. **Can Reflection break encapsulation?**

Ans:

Yes, reflection **can break encapsulation** by accessing private fields and methods.

1. **What are the real-world applications of Reflection API?**

Ans:  
Java Reflection is widely used in:

**Spring Framework** - Dependency Injection (DI).

**Hibernate** - ORM, mapping Java objects to database tables.

**JUnit & Mockito** - Unit testing, mocking private methods.

**Serialization** - Accessing private fields dynamically.

**JDBC** - Dynamic loading of database drivers.

1. **How do you create an immutable class in Java?**

Ans:

a) Don't provide "setter" methods — methods that modify fields

b) If the instance fields include references to mutable objects, don't allow those objects to be changed:

i.e. Don't provide methods that modify the mutable objects.

c) If the instance fields include references to mutable objects, don't allow those objects to be changed:

Don't share references to the mutable objects.

d) if necessary, create copies to avoid returning the originals in your methods

e) a class should be final

1. **Why finalize method is protected in java.lang.Object class?**

Ans:

The finalize() method in Java is declared as **protected** in java.lang.Object:

protected void finalize() throws Throwable { }

But why **protected** and not **public** or **private**?

Let's break it down:

**1. Preventing Direct Invocation from Outside Classes**

* If finalize() were **public**, any code could call it on an object, potentially causing **unexpected behavior** or **forcing cleanup** prematurely.
* Keeping it **protected** means only **subclasses** and the same package can override or access it, preventing **unintentional misuse**.

**Example: Allowed (Inside the Class or Subclass)**

class MyClass {

@Override

protected void finalize() throws Throwable {

System.out.println("Finalize called!");

}

}

**Example: Not Allowed (From Outside the Class)**

MyClass obj = new MyClass();

obj.finalize(); // Compilation error: 'finalize()' has protected access

**2. Designed for Overriding, Not Direct Calling**

* finalize() is meant to be **overridden** by subclasses for cleanup before garbage collection.
* If it were **private**, subclasses **couldn’t override it**, breaking the intended purpose.

**Example: Correct Usage (Overriding in a Subclass)**

class MyResource {

@Override

protected void finalize() throws Throwable {

System.out.println("Cleaning up resources...");

}

}

**3. Security and Encapsulation**

* By making finalize() **protected**, Java **limits access** to only **the class and its subclasses**, reducing security risks.
* A **public finalize()** would allow malicious code to force cleanup operations at the wrong time.

**4. Maintains Object-Oriented Principles**

* **Encapsulation**: Objects should **manage their own cleanup**, not let external code call finalize().
* **Inheritance**: Allows subclasses to implement custom finalization logic.

**Conclusion**

The finalize() method is **protected** in java.lang.Object because:  
Prevents unintended **external** calls.  
Allows **subclasses** to override it for cleanup.  
Avoids **security risks** and improper resource management.  
Follows **encapsulation** and **inheritance** principles.