JAVA Interview Questions:

1. What is an Object?
   An Object is an instance of a class. An Object has a state and behavior. It is a real-world entity.
   Ex: A dog is an Object.
   State: Color, name etc Behavior : Barking, walking etc

2. Ways to create an Object in Java?

a. Using new keyword

```
Class createClass{
        String name = "Class creation";
        Public static void main(String[] args)
        {
                createClass obj = new createClass();
                System.out.println(obj.name);

        }
    }
```

b. Using Factory Methods

```
Calendar calendar = Calendar.getInstance();
```

c. Using the clone() method

```
ClassName obj1 = new ClassName();
ClassName obj2 = (ClassName) obj1.clone(); etc,..
```

3. What are the methods the Object class has in Java?
Object class is present in java.lang package. Every class in Java is directly or indirectly derived from the Object class. If a class does not extend any other class then it is a direct child class of Object and if extends another class then it is indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of the inheritance hierarchy in any Java Program.

The Object class provides multiple methods which are as follows:

- tostring() method
- hashCode() method
- equals(Object obj) method
- finalize() method

- getClass() method
- clone() method
- wait(), notify() notifyAll() methods

4. What is the finalize() method in Java and it's return type?
The finalize() method in Java is a method provided by the Object class, and it serves as a way for an object to perform cleanup operations just before it is garbage collected. Garbage collection is the process by which Java's automatic memory management system reclaims memory occupied by objects that are no longer reachable and in use by the program.
Here's how the finalize() method works:

- When an object becomes eligible for garbage collection (i.e., there are no more references to it from any part of the program), the garbage collector will enqueue it for finalization. The finalize() method is not called immediately when the object becomes eligible for garbage collection. Instead, the object is added to a special queue for finalization.
- At some point, the garbage collector will run a finalization thread that will call the finalize() method on objects in the finalization queue. This means that the finalize() method is typically called in an asynchronous manner by the garbage collector.
- The finalize() method allows you to perform any necessary cleanup operations, such as releasing system resources (e.g., closing files or network connections), before the object is reclaimed by the garbage collector. The return type is : Void

```
public class MyResource {

  // Constructor and other class members

  // Your custom finalize method

  @Override

  protected void finalize() throws Throwable {

    try {

      // Perform cleanup operations here, e.g., close a file or network connection.

    } finally {

      super.finalize();

    }

  }

}
```

5. What is Encapsulation ?

Encapsulation is one of the fundamental principles of Object - oriented programming. It is a concept of bundling of data (attributes) and the methods (functions) that operate on that data into a single unit, often referred to as a class. Encapsulation restricts direct access to some of an object’s components and prevents the accidental modification of data.

Key aspects of encapsulation in Java:

a. Private Access Modifiers:

   Encapsulation is often achieved by declaring the data fields of a class as private, which means they can only be accessed from within the class itself. This restricts direct access from outside the class.

```
public class Person {

   private String name;

   private int age;

   // Getters and setters to access and modify the private fields

}
```

b. Getters and Setters:

   To allow controlled access to the private data fields, you can provide public methods called "getters" and "setters." Getters are used to retrieve the values of the private fields, and setters are used to modify them.

```
public class Person {

   private String name;

   private int age;

   public String getName() {

      return name;

   }

   public void setName(String name) {

      this.name = name;

   }

  public int getAge() {        return age;}
```

```
    public void setAge(int age) {
        this.age = age;
    }
}
```

```
public class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if (name != null && !name.isEmpty()) {
            this.name = name;
        } else {
            System.out.println("Name cannot be empty.");
        }
    }

    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        } else {
            System.out.println("Age cannot be negative.");
        }
    }
}

public class PersonExample {
    public static void main(String[] args) {
        // Create a Person object
        Person person = new Person();

        // Attempt to set the name and age
        person.setName("John");
        person.setAge(30);

        // Display the person's details
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());

        // Try to set invalid values
        person.setName(""); // This will print "Name cannot be empty."
        person.setAge(-5);  // This will print "Age cannot be negative."
        // Display the person's updated details

        System.out.println("Name: " + person.getName()); // Name remains "John"

        System.out.println("Age: " + person.getAge());   // Age remains 30

    }

}
```

Data Hiding: Encapsulation allows you to hide the internal details and implementation of a class from the outside world. Clients of the class interact with the object's public interface (methods), and they don't need to know how the data is stored or manipulated internally.

Validation and Control: With encapsulation, you can add logic and validation checks within the setter methods to control the values that are allowed to be set for the private fields. This helps maintain the integrity of the object's state.

6. What is the difference between local and global variable?

Local Variables:

- **Scope**: Local variables are declared within a specific block of code, such as a method, constructor, or a code block enclosed by curly braces {}. They are only accessible within the block where they are declared.
- **Lifetime**: Local variables have a limited lifetime. They are created when the block of code is entered and are destroyed when the block is exited.
- **Visibility**: Local variables are not visible outside of the block where they are declared. They are only accessible within the method or block in which they are defined.

Instance (Global) Variables:

- **Scope**: Instance variables are declared within a class but outside of any method, constructor, or code block. They are associated with an instance of the class (i.e., an object) and can be accessed throughout the class's methods and constructors.
- **Lifetime**: Instance variables live as long as the object they are associated with. They are created when an object is instantiated and exist until the object is no longer referenced and is eligible for garbage collection.
- **Visibility**: Instance variables are visible and accessible throughout the class in which they are declared, as well as from outside the class if they are declared with appropriate access modifiers (e.g., public, protected, or package-private) and via an object of the class.

  In easy words, local variables are method level variables and global variables are class level variables.

7. Can a class be declared private?

In Java, a top-level class (a class that is not an inner class or nested within another class) cannot be declared as private. The access modifiers that can be applied to top-level classes in Java are:

- public: A public class can be accessed from anywhere in the code, including other packages.
- Default (package-private): If no access modifier is specified, the class is considered "package-private" or having default access. It can only be accessed by other classes within the same package.
- abstract, final, strictfp: These modifiers can be applied in conjunction with one of the access modifiers mentioned above (e.g., public abstract class, abstract class, etc.), but they do not change the class's access level.

Inner classes, also known as nested classes, can have different access modifiers, including private. Inner classes are classes defined within another class, and they can have access modifiers to control their visibility within the containing class.

```
public class OuterClass {
   private class InnerPrivateClass {
      // Class members
   }
}
```

8. Can a class be final in Java?

Yes, a class in Java can be declared as final. When a class is declared as final, it means that the class cannot be subclassed or extended. In other words, you cannot create a new class that inherits from a final class.

9. Can a class be final public in Java?
Yes, a class in Java can be declared both as final and public at the same time. This combination means that the class is publicly accessible and cannot be subclassed.

10. Can a class be final public static in Java?
No, in Java, a class cannot be declared as both final and static at the same time. The final and static modifiers serve different purposes and have incompatible meanings when applied to classes.

11. What is a static variable in Java?
In Java, a static variable, also known as a class variable, is a variable that belongs to a class rather than to instances (objects) of that class. It's declared using the static keyword and is shared among all instances of the class. Static variables are stored in memory once, and their values are common to all objects of the class.
Here are some key points about static variables:

- **Shared Among Instances**: Static variables are common to all instances of a class. Any change made to a static variable is reflected in all objects of the class.

- **Initialization**: Static variables are initialized only once when the class is loaded, and they maintain their values throughout the program's execution.
- **Accessing**: You can access static variables using the class name, such as ClassName.staticVariable.

Real-life use cases for static variables include:

- **Counters**: Counting the number of instances of a class is a common use of static variables. For example, you might use a static variable to count the number of customers, transactions, or products in a system.
- **Configuration Settings**: Storing configuration settings that should be common to all instances of a class, like database connection details or application settings.
- Constants: Creating class-level constants. For example, you might define constants like Math.PI in the java.lang.Math class.

12. what is the use of declaring a variable as static final in Java?

Declaring a variable as static final in Java is a common practice when you want to create constants that are associated with a class and shared among all instances of the class. These constants are effectively class-level constants and are often used to define values that should not change during the execution of the program. Here's the use of declaring a variable as static final:

- Constants: static final variables are used to define constants that have the following characteristics:
- They are shared among all instances of the class.
- They are constant (cannot be modified) once assigned a value.
- They are accessed using the class name, making it clear that they are constants associated with the class.

```
public class Geometry {

   public static final double PI = 3.141592653589793;

   public static double calculateCircleArea(double radius) {

       return PI * radius * radius;

   }

}
```

13. Can there be public static final abstract variables in Java?

In Java, a variable cannot be declared as both abstract and final at the same time. The abstract modifier is used with abstract classes and methods, indicating that the method or class must be implemented in a subclass, and it cannot have a concrete (final) implementation in the current class.
However, you can have public static final variables in Java, and they are typically used to define constants that should not be modified.

14. Can local variables be static in Java?

The static modifier is used for class-level variables (static variables) and methods (static methods), not for local variables.

15. What is the concept of method overriding in Java?

In Java, method overriding allows a subclass to provide a specific implementation for a method that is already defined in its superclass. The overridden method in the subclass should have the same method signature (name, return type, and parameters) as the method in the superclass.

In Java, when you override a method in a subclass, the overriding method must not have a more restrictive access modifier than the overridden method in the superclass.