TOP 50 CORE JAVA INTERVIEW QUESTION ANSWERS

ULTIMATE GUIDE TO CRACK YOUR NEXT INTERVIEW

BY NILANCHALA PANIGRAHY X.COM/ASKNILAN

Question 1: Explain the difference between JDK, JRE, and JVM.

JVM (Java Virtual Machine) is the runtime engine that executes Java bytecode. It is platform-dependent, meaning different OS/hardware needs different JVM. JVM also provides memory management and security

JRE stands for Java Runtime Environment. It provides JVM and core libraries to run Java applications. JRE is required for running compiled Java programs

JDK stands for Java Development Kit. It contains tools like compiler, debugger, and libraries for the development of Java programs. You need JDK for developing Java applications

Question 2: What are the different types of memory areas allocated by JVM?

In Java, JVM allocates memory to different processes, methods, and objects. Some of the memory areas allocated by JVM are:

ClassLoader:
The ClassLoader is responsible for loading Java class files at runtime into the JVM. It doesn't technically allocate a specific memory area, but it plays a crucial role in loading classes into memory.

Method Area (Class Area):
This area stores per-class structures such as the runtime constant pool, field and method data, and the code for methods (bytecode). It's shared among all threads and is created when the JVM starts. The Method Area may also include metadata information about classes and interfaces.

Heap:
The Heap is the runtime data area where all Java objects and arrays are allocated. It's shared among all threads. Heap is managed by Java's garbage collection mechanism. The GC frees up unused memory periodically.

Stack:
Each thread maintains its own **Stack** for storing **frames**. Each frame corresponds to a method invocation and holds local variables, partial results, and return addresses. The stack is crucial for method invocation and stores temporary data such as **method arguments and local variables**. Unlike the heap, memory in the stack is freed as soon as the thread completes its method execution.

Program Counter (PC) Register:
This memory area contains the address of the Java virtual machine instruction that is currently being executed.

Native Method Stack:
This area is specifically reserved for storing the state of native method calls. The native methods are typically written in C or C++.

Question 3: How does ClassLoader work in Java?

In Java, ClassLoader is a class that is used to load files from their physical locations e.g. Filesystem, Network in JVM. There are three main types of ClassLoaders in Java.

1. Bootstrap ClassLoader: This is the first ClassLoader. It loads classes from the jar file.
2. Extension ClassLoader: It loads class files from `jre/lib/ext` location.
3. Application ClassLoader: This ClassLoader depends on **CLASSPATH** to find the location of class files. If you specify your jars in CLASSPATH, then this ClassLoader will load them.

Question 4: What is the difference between `byte` and `char` data types in Java?

Both byte and char are numeric data types in Java. They are used to represent numbers in a specific range. The major difference between them is that a byte can store raw binary data where as a char stores characters or text data.

- Byte values range from -128 to 127.
- A byte is made of 8 bits. But a char is made of 16 bits. So it is equivalent to 2 bytes.

Question 5: Why do we need a constructor in Java?

A constructor is essential for initializing objects in Java. It is used for passing initial values and allocating resources before the object is used. Here are some facts about Java constructors

- Constructors are called automatically when you create an object using a new operator
- The class name and the constructor name should be the same otherwise compiler treats that as a method.
- A constructor can not have any return type, not even Void.

- You can declare overload constructors in Java.

Usually, a constructor is required in almost every Class. If no explicit constructor is declared, the Java compiler provides a no-argument default constructor for each class.

Question 6: Is it possible to use this() and super() both in the same constructor?

No, Java does not allow using both `super()` and `this()` in the same constructor. Both super() and this() must be the first statement in a constructor.

Question 7: What is the difference between the comparison done by the equals() method and the == operator?

The `equals()` method in Java is used for comparing the contents of two string objects. It returns **true** when the two strings have the same value. The == operator compares the references of two string objects.

In the following example, `equals()` returns true as the two string objects have the same values. However, the == operator returns false as both string objects `str1` and `str2` are referencing different objects.

```
public class EqualsTest {

    public static void main(String args[]) {
        String str1 = new String("Hello World");
        String str2 = new String("Hello World");

        //Reference comparison
        System.out.println(s1 == s2);

        //Content comparison
        System.out.println(s1.equals(s2));

        // integer-type
        System.out.println(10 == 10);

        // char-type
        System.out.println('a' == 'a');
    }

}
```

Output

```
false
true
true
true
```

Question 8: What is the output of the following program?

```
public class Test {

    public void print(int a, long b) {
        System.out.println("Method 1");
    }

    public void print(long a, int b) {
        System.out.println("Method 2");
    }

    public static void main(String[] args) {
        Test obj = new Test();
        obj.print(5, 10);
    }

}
```

The above program will output:

```
Method 1
```

This is because Java will automatically promote smaller numeric types to larger numeric types. In this case, the integer literal 5 can be automatically promoted to either an `int` or a `long`, and the integer literal `10` can also be promoted to either an `int` or a `long`.

Question 9: How to create an infinite loop in Java?

Infinite loops are those loops that run infinitely without any breaking conditions. Some examples of consciously declaring an infinite loop are:

Using For Loop:

```
for (;;)
{
   // Business logic
   // Any break logic
}
```

Using while loop:

```
while(true){
   // Business logic
   // Any break logic
}
```

Using do-while loop:

```
do{
   // Business logic
   // Any break logic
} while(true);
```

Question 10: Explain the Diamond Problem. How does Java overcome this problem?

The Diamond Problem is a situation where a class inherits from two classes that have a common ancestor. This creates ambiguity in the path through which a method or a member variable is inherited because there are 2 possible paths to follow up the inheritance hierarchy to the common ancestor.

Java does not support multiple inheritance for classes, meaning a class cannot extend more than one class. However, it supports multiple inheritance of types through the interface.

Question 11: What is a `default` method in the interface? Can Java Interface contain method implementation?

Java 8 has introduced the concept of default methods (Also known as Extension Methods) which allow the interfaces to have methods with implementation.

This provides the ability for interfaces to define new methods without breaking the implementation for all classes that implement the interface.

You can do this by using the **default** keyword.

```
interface Vehicle {
    String speedUp();
    String slowDown();

    default String alarmOn() {
        return "Turning the alarm on.";
    }

    default String alarmOff() {
        return "Turning the alarm off.";
    }
}
```

To learn more about default methods for interfaces, visit this link.

Question 12: Explain how Java deals with the diamond problem due to the default method in the interface.

With the introduction of Default methods in interfaces in Java 8, the potential for a diamond problem arose. Default methods allow an interface to provide a default implementation for methods, which means a class can inherit concrete methods from multiple interfaces.

There are two ways to address this issue in Java:

1. Explicit Implementation:
If a class implements multiple interfaces that define the same default method, the class must explicitly provide an implementation for that method to resolve the conflict.

2. Inheritance Rule:
If a class inherits a method from a superclass, it takes precedence over any default methods provided by interfaces.

```
interface Interface1 {
    default void hello() {
        System.out.println("Hello from Interface1");
    }
}

interface Interface2 {
    default void hello() {
        System.out.println("Hello from Interface2");
    }
}
```

```java
public class Child implements Interface1, Interface2 {
    @Override
    public void hello() {
        System.out.println("inside Child class hello method");

        // Calling Interface1's default method
        Interface1.super.hello();
    }

    public static void main(String[] args) {
        Child obj = new Child();

        // This will call the Child class hello method
        obj.hello();
    }
}
```

Question 13: What is Functional Interfaces in Java?

A functional interface is the foundation of functional programming that allows developers to pass and execute behavior as arguments. Functional interfaces must contain exactly one abstract method declaration, generally used as a lambda expression or a method reference.

Since default methods are not abstract you're free to add default methods to your functional interface. We can use arbitrary interfaces as lambda expressions as long as the interface only contains one abstract method.

To ensure your interface meets the requirements, you can optionally add the `@FunctionalInterface` annotation. The compiler is aware of this annotation and throws a compiler error as you try to add a second abstract method declaration to the interface.

```java
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}
```

This can be used as

```java
Converter<String, Integer> converter= (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted);
```

Question 14: Can Abstract Class in Java have constructors?

Yes, abstract classes in Java can indeed have constructors.

Although it might initially appear counterintuitive since abstract classes cannot be directly instantiated, these constructors serve a crucial purpose by initializing the fields and can be invoked during the instantiation of their concrete subclasses.

Here is an example of an abstract class with constructors:

```
abstract class MyAbstractClass {

    public int a;
    public int b;

    public MyAbstractClass(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public void print() {
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
}

public class AbstractDemo extends MyAbstractClass {

    public AbstractDemo(int x, int y) {
        super(x, y);

    }

    public static void main(String[] args) {
        AbstractDemo obj = new AbstractDemo(5, 10);
        obj.print();
    }

}
```

When a concrete subclass of an abstract class is instantiated, the constructor of the abstract class is called first, either implicitly or through the use of `super()` by passing arguments.

This process is known as constructor chaining and ensures that the initialization behavior defined in the abstract class's constructor is executed.

Question 15: Why is the Main Method Static in Java?

The `static` keyword in Java can be used for classes, members, or methods. Static variables and methods are associated with the class, not specific instances, so all instances share a copy.

The main method is always static because it belongs to the class, not an object. If not static, it's available to all objects, which isn't acceptable to the JVM. The JVM calls the main method based on the class name, not the object.

Java programs must have only one main method as execution starts from it.

Question 16: Explain Java Streams.

A `java.util.Stream` represents a sequence of elements on which one or more operations can be performed. Stream operations are either intermediate or terminal.

While terminal operations return a result of a certain type, intermediate operations return the stream itself so you can chain multiple method calls in a row.

Streams are created on a source, e.g. a `java.util.Collection` like lists or sets (maps are not supported). Stream operations can either be executed sequentially or parallelly. Streams can be created either by
calling `Collection.stream()` or `Collection.parallelStream()` method.

The following sections explain the most common stream operations.

```
List<String> listItems = new ArrayList<>();
listItems.add("item1");
listItems.add("item3");
listItems.add("item2");

listItems
    .stream()
    .filter((s) -> s.startsWith("item3"))
    .forEach(System.out::println);
```

Question 17: How would you differentiate between a `String`, `StringBuffer`, and a `StringBuilder`?

A `String` is immutable, whereas both the `StringBuilder` and `StringBuffer` are mutable. In string, the String pool serves as the storage area. For StringBuilder and StringBuffer, memory is allocated in the heap memory storage area.

It is quite slow to work with a String. However, StringBuilder is the fastest in performing operations. The speed of a StringBuffer is more than a String and less than a StringBuilder. (For example, appending a character is fastest in StringBuilder and very slow in String because a new memory is required for the new String with appended character.)

```
// String
String first = "Stacktips";
String second = new String("StackTips");

// StringBuffer
StringBuffer third = new StringBuffer("StackTips");

// StringBuilder
StringBuilder fourth = new StringBuilder("StackTips");
```

Question 18: Explain the output of the following program related to the `equals()` method of `StringBuilder`.

```
public class Demo {

    public static void main(String[] args) {

        StringBuilder sb1 = new StringBuilder("hello");
        StringBuilder sb2 = new StringBuilder("hello");

        if(sb1.equals(sb2)) {
            System.out.println("Equal");
        } else {
            System.out.println("Not Equal");
        }

    }
}
```

The output of the given Java program will be `Not Equal`.

This is because the `equals()` method in the `StringBuilder` class does not override the `equals()` method from the `Object` class. In the `Object` class, the `equals()` method checks for reference equality, meaning it returns true if and only if both references point to the same object in memory.

Since ***sb1*** and ***sb2*** are references to two different StringBuilder objects even though they contain the same sequence of characters, the `equals()` method inherited from the Object class will return false.

Question 19: What is method overriding? What restrictions are in place on method overriding?

When a class defines a method using the same name, same return type, and same argument list as that of a method in its superclass, it is said to be overridden. When the method is invoked for an object, it is the overridden method that is called and not the method definition from the superclass.

Following are some of the restrictions placed on method overriding:

- Overridden methods must have the same name, argument list, and return type.
- The overriding method may not limit the access of the method it overrides. Methods may be overridden to be more public, not more private.
- The overriding method may not throw any exceptions that may not be thrown by the overridden method.

Question 20: What is the Difference Between an Inner Class and a Sub-Class?

An Inner class is a class that is nested within another class. An Inner class has access rights to the class which is nesting it and it can access all variables and methods defined in the outer class.

A sub-class is a class that inherits from another class called a superclass. Sub-class can access all public and protected methods and fields of its superclass.

Question 21: What are the various access specifiers used for class definition in Java?

Access specifiers are the keywords used before a class name that defines the access scope. The types of access specifiers for classes are:

- **Public:** Class, Method, and Field are accessible from anywhere.
- **Protected:** Method, Field can be accessed from the same class to which they belong or from the sub-classes, and the class of the same package, but not from outside.
- **Default:** Method, Field, and class can be accessed only from the same package and not from outside of its native package.
- **Private:** Method, Field can be accessed from the same class to which they belong.

Question 22: Explain encapsulation in OOPS

Encapsulation is a concept in Object Oriented Programming (OOP) that combines properties and methods in a single unit.

Encapsulation helps programmers follow a modular approach to software development as each object has its own set of methods and variables and serves its functions independently of other objects. Encapsulation also serves the data hiding purpose.

Question 23: What is a Singleton class? Give a practical example of when should you use Singleton class.

Singleton design pattern belongs to the creational family of patterns that governs the instantiation process. This pattern ensures at most one instance of a class is ever created through the application lifecycle.

Following are some of the examples where Singleton classes are best suited:

- **Project Configuration:** A class that reads your project configuration can be made Singleton. By making this singleton, you are allowing global access for all classes in your application. If the project configs are stored in a file, it just reads once and holds on the application cache. You don’t have to read the file multiple times.
- **Application Log:** The logger will be used everywhere in your application. It must be initialized once and used everywhere

- **Analytics and Reporting:** If you are using some kind of data analysis tool like Google Analytics, you will notice that they are designed to be a singleton. It initializes once and is used everywhere for each user action.

For the step-by-step guide to implementing singleton class, check out this link.

Question 24: What is the difference between continue and break statement?

The `break` and `continue` are two important keywords used in the loops. When a break keyword is used in a loop, the loop is broken instantly while when the continue keyword is used, the current iteration is broken and the loop continues with the next iteration.

In the following example, the Loop is broken when the counter reaches 4.

```
for (counter = 0; counter< 10; counter++)
    system.out.println(counter);

    if (counter == 4) {
        break;
    }
}
```

In the following example when the counter reaches 4, the loop jumps to the next iteration and any statements after the continue keyword are skipped for the current iteration.

```
for (counter = 0; counter < 10; counter++)
    system.out.println(counter);
    if (counter == 4) {
        continue;
    }
}
```

Question 25: What is the difference between double and `float` variables in Java?

In Java, float takes 4 bytes in memory while **Double** takes 8 bytes in memory. **Float** is a single-precision floating-point decimal number while Double is a double-precision decimal number.

Question 26: What is the `final` Keyword in Java?

A variable declared with the final keyword is a constant in Java. Value can be assigned only once and after the assignment, the value of a constant can't be changed.

For example, a constant with the name **MAX_LIMIT** is declared and assigned value:

```
private final int MAX_LIMIT=100
```

When a method is declared as final, it can NOT be overridden by the subclasses. This method is faster than any other method because they are resolved at compile time.

When a class is declared as final, it cannot be inherited. Example String, Integer, and other wrapper classes.

Question 27: Can we declare a class as abstract without having any abstract method?

Yes, we can create an abstract class by using the abstract keywords before the class names, even if it doesn't have any abstract methods. However, if a class has even one abstract method, it must be declared as abstract. Otherwise, it will result in an error.

Question 28: What is the difference between an abstract class and an interface in Java?

The primary difference between an abstract class and an interface is that an interface can only possess a declaration of public static methods with no concrete implementation while an abstract class can have members with any access specifiers (public, private, etc) with or without a concrete implementation.

Another key difference in the use of abstract classes and interfaces is that a class that implements an interface must implement all the methods of the interface while a class that inherits from an abstract class doesn't require the implementation of all the methods of its super class.

A class can implement multiple interfaces but it can extend only one abstract class.

Question 28: Can we declare the main method of our class as private?

No, in Java, the main method cannot be declared as a private method. If the main method is declared as private, it will not get any compilation error but, it will get the following runtime error.

```
Error: Main method not found in class MainClass, please define the mai
n method as:
   public static void main(String[] args)
```

If you want the JVM to recognize and execute it the main method needs to be public and static.

Question 30: How can we execute any code even before the main method?

To execute your code before the main method, you can use a static initializer block. Any statements inside this static initializer block will be first loaded by JVM and executed once before the main() method.

```
public class MainClass {

    static {
        System.out.println("Executing static block");
    }

    public static void main(String[] args) {
        System.out.println("Executing main method.");
    }
}
```

Output:

```
Executing static block
Executing main method.
```

Question 31: What is a Serializable interface?

Serializable is a marker interface. When an object has to be transferred over a network (typically through RMI or EJB) or persist the state of an object to a file, the class needs to implement a Serializable interface.

Implementing this interface will allow the object converted into a byte stream and transferred over a network.

Question 32: What is the use of `serialVersionUID`?

During object serialization, the default Java serialization mechanism writes the metadata about the object, which includes the class name, field names, types, and superclass. This class definition is stored as a part of the serialized object.

This stored metadata enables the deserialization process to reconstitute the objects and map the stream data into the class attributes with the appropriate type every time an object is serialized the Java serialization mechanism automatically computes a hash value.

ObjectStreamClass's `computeSerialVersionUID()` method passes the class name, sorted member names, modifiers, and interfaces to the secure hash algorithm (SHA), which returns a hash value. The `serialVersionUID` is also called SUID.

So when the serialized object is retrieved, the JVM first evaluates the suid of the serialized class and compares the suid value with one of the objects. If the suid values match, then the object is said to be compatible with the class, and hence, it is de-serialized. If not, an **InvalidClassException** exception is thrown.

Changes to a serializable class can be compatible or incompatible.

Question 33: How does garbage collection work in Java?

Garbage collection in Java is an automatic memory management process that frees up memory occupied by objects that are no longer in use. It identifies which objects in memory are no longer reachable and periodically removes the unreachable objects from memory.

Garbage collection (GC) helps to prevent memory leaks and manage the limited memory available in Java applications. The typical GC process uses a **Mark-and-Sweep** algorithm.

Question 34: Explain the differences between checked and unchecked exceptions

Checked Exceptions:

- Checked Exceptions are exceptions that the JAVA compiler requires us to handle. We have to use the try, cache block to handle the exception or throw the exception up the call stack
- For example, **IOException**, **SQLException**, etc are examples of checked exceptions.
- All modern Java IDEs (Integrated Development Environments) will show these errors during the compilation time.

Unchecked exceptions:

- The Java compiler ignores the unchecked exceptions and we are not forced by the compiler to handle it. Unchecked exceptions are those exceptions that extend `RuntimeException`
- For example, **NullPointerException**, **IllegalArgumentException**, etc are examples of unchecked exceptions.
- Though the Java compiler does not force us, we can still choose to handle this exception to maintain the normal flow of the program.

Read more about exception handling in Java here.

Question 35: What is the purpose of the `transient` keyword in Java?

Serialization is the process of converting an object into a byte stream so it can be saved to a file or transferred over a network. Deserialization is the reverse process, where the byte stream is converted back into an object.

The transient keyword in Java is used to mark fields in a class that should not be serialized when the class is converted into a byte stream during serialization. When an object is serialized to write to a file or sent over a network, fields marked as transient are excluded from the serialization process.

In the following example, the password field is declared as transient meaning, it will not be serialized. When we deserialize the user object, notice that the password will be null.

```
public class User implements Serializable {
    private String username;
    private transient String password;

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
    @Override
    public String toString() {
        return "User{username='" + username + "', password='" + passwo
rd + "'}";
    }
}

public class TestTransient {
    public static void main(String[] args) {
        User user = new User("john_doe", "password123");

        // Serialization
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileO
utputStream("user.txt"))) {
            oos.writeObject(user);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialization
        try (ObjectInputStream ois = new ObjectInputStream(new FileInp
utStream("user.txt"))) {
            User deserializedUser = (User) ois.readObject();
            System.out.println(deserializedUser);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Question 36: What is an `Externalizable` interface in Java? How is it different from the `Serializable` interface?

The `Serializable` is a marker interface, but `Externalizable` is not a marker interface. Externalizable interface extends Serializable interface.

If a class implements a Serializable interface, the class is serialized automatically by default. However, the Externalizable interface requires explicit implementation of serialization logic. If provides more control over the object serialization and deserialization process using the following 2 methods:

- **writeExternal(ObjectOutput out):** This method specifies how the object's data should be written out during serialization.
- **readExternal(ObjectInput in)**: This method specifies how the object's data should be read back in during deserialization.

The default serialization mechanism saves all non-transient fields. However, in the case of Externalizable, you must explicitly write and read each field. A public no-arg constructor is needed when using an externalizable interface.

In Serialization, we need to define `serialVersionUID`. If it is not explicitly specified, it will be generated automatically based on all the class fields and methods.

```
class User implements Externalizable {
    private String username;
    private String password;

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public User() {
    }
    @Override
    public String toString() {
        return "User{username='" + username + "', password='" + passwo
rd + "'}";
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(username); // Serialize object
        out.writeObject(password);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, Class
NotFoundException {
        this.username = (String) in.readObject(); // Deserialize objec
t
        this.password = (String) in.readObject();

    }
}
```

Question 37: What is Reflection in Java? What are the uses of Java Reflection?

Reflection in Java provides the ability to inspect and manipulate the structure and behavior of classes, interfaces, fields, and methods at runtime.

1. Dynamic Instantiation
Reflection can be used to create an instance of classes dynamically without knowing the class at compile time. This is often used in frameworks like dependency injection containers.

```
Class<?> clazz = Class.forName("com.stacktips.User");
Object instance = clazz.getDeclaredConstructor().newInstance();
```

2. Inspecting class structure
Using reflection, you can inspect class metadata, such as the list of methods, fields, constructors, and annotations.

```
for (Method method:clazz.getDeclaredMethods()) {
    System.out.println("Method: " + method.getName());
}
```

3. Access private fields
Using reflection, we can access private fields and methods. It is useful for testing purposes or when you need access to a class's private fields for specialized tasks.

```
Field privateField = clazz.getDeclaredField("username"); privateField.
setAccessible(true);
Object value = privateField.get(instance);
```

Question 38: How can we access the `private` method of a class from outside the class?

Reflection can be used to access the `private` method of a class from outside the class. We use `getDeclaredMethod()` to get an instance of a private method. Then we mark this method accessible and finally invoke it.

Example:

```
public class Foo {
    private void printHello(String user) {
        System.out.println("Hello " + user + " !");
    }
}
```

```
class ReflectionDemo {
    public static void main(String[] args) throws Exception {
        Class<?> c = Class.forName("com.stacktips.reflection.Foo");
        Object obj = c.newInstance();
        Method method = c.getDeclaredMethod("printHello", String.class
);
        method.setAccessible(true);
        method.invoke(obj, "John");
    }
}
```

Question 39: How many objects does the following code create?

```
String s1="Hello World!";
String s2="Hello World!";
String s3="Hello World!";
```

The above code creates only one object. Since there is only one String Literal “HelloWorld” created, all the references point to the same object. This saves memory usage.

Question 40: How will you create an immutable class in Java?

In Java, we can declare a class final to make it immutable. Here are some of the key considerations to make while making a class immutable:

1. Declare the Class as `final` to prevent it from being subclassed. The final keyword in class declaration ensures that its immutability is not compromised by its subclass.
2. Make All Fields `private` and `final` to prevent direct access and as `final` to ensure they are assigned only once, at the time of object creation.
3. Do not include any setter methods for fields. This way, once the object is constructed, its state cannot be changed externally.
4. Initialize Fields using a copy constructor to prevent external modifications of mutable fields after the object is created.
5. For fields that hold mutable objects, return a deep copy instead of the actual object to prevent modifications from outside the class.
6. Avoid the `clone` method, instead return a deep copy of the object instead of a reference to the object itself.

Example:

```
public final class Person {
    private final String name;
    private final int age;
    private final Address address;

    public Person(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        this.address = new Address(address);  // copy constructor
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    // Return a deep copy of the mutable Address object
    public Address getAddress() {
        return new Address(this.address);
    }
}

// Mutable class
class Address {
    private String street;
    private String city;

    public Address(Address address) {
        this.street = address.street;
        this.city = address.city;
    }

    public Address(String street, String city) {
        this.street = street;
        this.city = city;
    }

    // Getters and setters
}
```

Question 41: How can we create a singleton class, ensuring it is thread-safe?

There are several ways we can create a thread-safe singleton class in Java.

1. Using Eager Initialization

In this method, the singleton instance is created at the time of loading the class. It's thread-safe by default because the instance is created before any thread accesses it.

```
class Singleton {
    private static Singleton INSTANCE = new Singleton();

    private Singleton() {
    }
    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

The downside to this is that the instance is created even if it's never used.

2. Lazy Initialization with Synchronized Method

In this approach, the instance is created lazily only when it's needed. Using the `synchronized` keyword makes this thread safe by allowing only one thread to access the `getInstance()` method at a time.

```
class Singleton {
    private static Singleton INSTANCE;

    private Singleton() {
    }
    public static synchronized Singleton getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Singleton();
        }
        return INSTANCE;
    }
}
```

The `synchronized` keyword can lead to performance issues.

3. Using Double-Checked Locking

This approach minimizes the performance cost of synchronization by only locking the critical section when the instance is `null`. We need to use the `volatile` keyword for the instance to ensure the visibility of changes to variables across threads.

```
class Singleton {
    private volatile static Singleton INSTANCE;

    private Singleton() {
    }
    public static synchronized Singleton getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Singleton();
        }
        return INSTANCE;
    }
}
```

4. Bill Pugh Singleton Design (Inner Static Helper Class

This is a highly efficient and widely recommended approach. The instance is created by a static inner helper class, which is only loaded when `getInstance()` is called.

```
class Singleton {

    private static class SingletonHelper {
        private static final Singleton INSTANCE = new Singleton();
    }

    private Singleton() {
    }

    public static Singleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

Question 41: List some of the key features introduced in Java 17.

Java 17 introduced several important features including:

- Sealed Classes- Restricts which classes can inherit from a sealed class
- Pattern Matching for switch: Allows more expressive and concise switch statements and reduces boilerplate code in type-checking scenarios
- Records: Provides a compact syntax for creating immutable data classes
- Text Blocks: Allows multi-line string literals without escape characters

- Improved Random Number Generators: Provides a new interface and implementations for random number generation

Question 42: Explain the concept of text blocks in Java. How do they improve code readability?

Text blocks in Java improve the way multiline strings are handled in code. Traditionally very difficult to read strings that span across multiple lines and require preserving formatting.

Java 17 introduced triple quotes ("""). Text inside triple quotes is called text blocks. The compiler automatically strips incidental indentation and preserves formatting.

```
String query = "SELECT e.employee_id, e.first_name, e.last_name, \n" +
               " d.department_name, j.job_title, \n" +
               " e.salary, e.hire_date \n" +
               "FROM employees e \n" +
               "JOIN departments d ON e.department_id = d.department_i
d \n" +
               "JOIN jobs j ON e.job_id = j.job_id \n" +
               "WHERE e.salary > 50000 \n" +
               " AND e.hire_date > '2020-01-01' \n" +
               " AND d.department_name IN ('Sales', 'Marketing', 'IT')
\n" +
               "ORDER BY e.salary DESC, e.last_name ASC";

// With text blocks
String query = """
        SELECT e.employee_id, e.first_name, e.last_name,
          d.department_name, j.job_title,
          e.salary, e.hire_date
        FROM employees e
        JOIN departments d ON e.department_id = d.department_id
        JOIN jobs j ON e.job_id = j.job_id
        WHERE e.salary > 50000
          AND e.hire_date > '2020-01-01'
          AND d.department_name IN ('Sales', 'Marketing', 'IT')
        ORDER BY e.salary DESC, e.last_name ASC
        """;
```

Question 43: What are the Records in Java?

Records is a new language feature introduced in Java 14 and finalized in Java 16. A record is a special type of class in Java that allows us to define classes that act as transparent carriers for immutable data. Records can be used to replace traditional POJOs, which are often verbose and require a lot of boilerplate code.

This is how we can define the records

```
public record Vehicle(String make, String model, int year) {}
```

The fields of a record are implicitly final, which means that once a record instance is created, the data it contains cannot be changed.

Records provide several built-in methods for common operations such as constructors, getters, `equals()`, `hashCode()`, and `toString()`. Like a regular class, methods inside the record can be extended to provide a custom implementation.

Records allow you to define a **Compact constructor** which omits the parameter list, assuming the same parameters as the record components. Within this constructor, you can include validation or normalization logic for the fields.

```
public record Vehicle(String make, String model, int year) {

    // Compact constructor
    public Vehicle {
        if (year < 1886) { // The first car was made in 1886
            throw new IllegalArgumentException("Invalid year");
        }
        make = make.trim();
        model = model.trim();
    }
}
```

Records are ideal for creating simple data-carrier classes, such as DTOs (Data Transfer Objects), value objects in domain-driven design, tuples, and more. Records are serializable by default, provided that all their components are serializable.

Question 44: Explain the concept of sealed classes in Java 17.

Sealed classes are a new language feature introduced in Java 17 as part of JEP 409. They provide a way to restrict the subclasses that can extend a class or implement an interface. This feature is useful to create more robust and maintainable code and to define a closed hierarchy of types.

Sealed Classes allow you to specify a limited set of subclasses that can extend a given class or implement an interface.

This is how we can declare a sealed class in Java:

```
public sealed abstract class Vehicle permits Car, Truck, Motorcycle {
    private final String make;
    private final String model;

    public Vehicle(String make, String model) {
        this.make = make;
        this.model = model;
    }

    public abstract void displayVehicleInfo();
}

public final class Car extends Vehicle {
    private final int seatingCapacity;

    public Car(String make, String model, int seatingCapacity) {
        super(make, model);
        this.seatingCapacity = seatingCapacity;
    }

    @Override
    public void displayVehicleInfo() {
        System.out.println("Car: " + getMake() + " " + getModel() + ",
Seating Capacity: " + seatingCapacity);
    }
}
```

The `permits` clause is used to specify the allowed subclasses for type Shape.

Since the compiler knows all the possible subtypes of a sealed class, it will prevent any other class except `Circle`, `Square`, or `Rectangle` from extending the Shape class.

Question 45: How Sealed classes are different from final classes?

Sealed classes and final classes serve different purposes in Java, although both are used to restrict inheritance.

- Final Classes in Java cannot be inherited at all. It can be used when a class should never be extended.
- Sealed Classes can be inherited, but only by a predefined set of classes. It is used when you want to allow inheritance, but only for specific classes.

Question 46: How does the pattern matching for `instanceof` work in Java 17?

Pattern Matching for `instanceof` in Java 17 enhances the `instanceof` operator and eliminates the need for type casting and checking.

Traditional approach:

```
if (obj instanceof Car) {
    Car str = (Car) obj;
    // Use car object here
}
```

With pattern matching for `instanceof`:

```
if (obj instanceof Car car) {
    // Use car object here
}
```

As you can notice, it makes the code more concise but also enhances readability and reduces the possibility of errors, such as incorrect casting.

Question 47: How does the Pattern matching for the switch work?

Pattern Matching for the switch is introduced as a preview feature in Java 17. This feature extends the switch expression and switch statement to allow pattern matching in case labels. Pattern Matching for switch aims to make code more readable and safe, especially when dealing with type testing and casting.

Please note, that the Pattern Matching for the switch is still in preview and not yet finalized.

```
switch (obj) {
    case String s -> System.out.println("It's a string: " + s);
    case Integer i -> System.out.println("It's an integer: " + i);
    case null -> System.out.println("It's null");
    default -> System.out.println("It's something else");
}
```

Question 48: Is it possible to use records with inheritance?

A record declaration does not have an extends clause, so it is not possible to explicitly declare a direct superclass type, even a Record. However, a record can implement interfaces, so you can use them polymorphically.

Refer to the Java 17 JLS 8.10 notes for more information.

Question 49: How does the new Random Number Generators API in Java 17 improve upon the previous implementation?

The new Random Number Generators API in Java 17 introduced a new `RandomGenerator` interface as the top-level interface for random number generators. It provides specialized interfaces like `SplittableRandomGenerator`, `JumpableRandomGenerator`, and `LeapableRandomGenerator` for different types of generators.

It includes algorithms like LXM, Xoroshiro128PlusPlus, and Xoshiro256PlusPlus, which are more modern and efficient than the older Linear Congruential Generator (LCG) used in the legacy `Random` class.

The new API interface offers better control over the sequence of random numbers generated and provides methods to "jump" or "leap" ahead in the sequence without generating intermediate values.

```
import java.util.random.RandomGenerator;
import java.util.random.RandomGeneratorFactory;

public class RandomExample {
    public static void main(String[] args) {
        RandomGenerator rng = RandomGeneratorFactory.of("Xoshiro256Plu
sPlus")
                .create();
        int randomInt = rng.nextInt();
        rng.doubles().limit(5).forEach(System.out::println);

        // Create a splittable generator for parallel processing
        RandomGenerator splitRng = RandomGeneratorFactory.of("L64X128M
ixRandom").create();
        RandomGenerator split1 = splitRng.split();
        RandomGenerator split2 = splitRng.split();

        System.out.println(split1.nextInt());
        System.out.println(split2.nextInt());
    }
}
```

Question 50: Explain the differences between the Supplier and Consumer interfaces.

The Supplier and Consumer interfaces are built-in functional interfaces in `java.util.function` package. There are several differences between the two as follows:

The `Supplier` interface does not accept an argument and produces a return value of any data type. It is useful when you want to generate values lazily or return a constant value on demand.

Example:

```
// Supplier that provides a constant string value
Supplier<String> greetingSupplier = () -> "Hello, World!";
System.out.println(greetingSupplier.get()); // Outputs: Hello, World!

// Supplier that provides the current system time
Supplier<Long> timeSupplier = System::currentTimeMillis;
System.out.println("Current Time: " + timeSupplier.get());
```

The `Consumer` interface accepts an argument of any data type and it does not return any value, It is useful when you need to act on an input, such as printing or logging values.

Example:

```
// Consumer that prints a string in uppercase
Consumer<String> printUpperCase = str -> System.out.println(str.toUppe
rCase());
printUpperCase.accept("hello");

// Consumer that logs the length of a string
Consumer<String> logLength = str -> System.out.println("Length: " + st
r.length());
logLength.accept("Java");
```

ABOUT THE AUTHOR

Hello, I am **Nilanchala Panigrahy**! I have over 16+ years of experience in system analysis, design, and implementation of enterprise apps. Over the past few years, I have worked on technology transformational projects for renowned organizations such as NESS, Volkswagen, SAP Labs, Transamerica, and AMEX.

In addition to my day job, I drive great satisfaction from mentoring aspiring developers and sharing my expertise through my blog www.stacktips.com. During my mentoring sessions, I frequently receive many requests for a comprehensive set of interview preparation questions. This book is a culmination of my accumulated knowledge and experiences shared with my mentees over the years.

You can find more information about me on my blog stacktips.com. If you'd like to connect with me directly, you can find me on X under the handle @asknilan.

Finally, I want to express my gratitude to all of you for choosing this book. I hope you find this guide useful in preparing for your next technical interview.

THANK YOU!

Thank you for reading this e-book. I hope you find all the necessary information here. Should you have any questions, or queries, or simply wish to engage in a conversation, please do not hesitate to contact me via hello@stacktips.com. I will promptly respond to your inquiries.

If you're not already a member, don't forget to head to www.stacktips.com and sign up for your free account. It is packed with tons of step-by-step tutorials, free courses, and goodies.

Finally, I would like to extend my best wishes for your upcoming interview. I am sure you're going to knock the next one.