1. Is Java Platform Independent? If Yes, How?

Yes, Java is platform-independent because its code is **compiled into bytecode** using the **Java compiler (`javac`)**, which can be executed on any platform with a **Java Virtual Machine (JVM)**.

This platform independence is achieved through the following process:

- Source code (`.java`) → Compiled bytecode (`.class`) → JVM interprets bytecode on any operating system.
- The key factor here is the JVM, which acts as a translator between bytecode and the host machine.

---

2. Difference Between JVM, JRE, and JDK

Component	Description	Purpose
JVM (Java Virtual Machine)	Abstract machine that runs bytecode	Executes Java bytecode
JRE (Java Runtime Environment)	Includes JVM + core libraries	Allows running Java applications
JDK (Java Development Kit)	JRE + development tools (compiler, debugger)	For developing, compiling, and running Java programs

---

3. What Is a ClassLoader? What Are Its Types?

A **ClassLoader** in Java is a part of the JVM responsible for **loading classes** dynamically during runtime. It converts the bytecode into machine-specific instructions.

Types of ClassLoaders in Java

1. **Bootstrap ClassLoader**:
    - Loads the core Java classes from the `rt.jar` file (`java.lang` , `java.util` , etc.).
2. **Extension ClassLoader**:
    - Loads classes from the `jre/lib/ext` directory.

3. **System (or Application) ClassLoader**:
    - Loads classes from the application's classpath (`CLASSPATH` environment variable).

How ClassLoader Works

Class loading follows a **delegation model**, where each ClassLoader delegates loading to its parent before attempting to load the class itself.

---

4. How Is `String` Immutable in Java?

A `String` is **immutable** in Java because once created, its value cannot be changed.

Why is it Immutable?

1. **Security**: Prevents accidental or malicious changes when strings are used for security keys or file paths.
2. **String Pooling**: Immutable strings can be shared in the **String Pool**, reducing memory usage.
3. **Thread Safety**: Since strings can't be modified, they are inherently thread-safe.

How It Works Internally

```
String str1 = "Hello";
String str2 = "Hello";
str1 = str1.concat(" World");
System.out.println(str1);  // "Hello World"
System.out.println(str2);  // "Hello"
```

Here, `str1.concat(" World")` creates a **new String object** instead of modifying the existing one.

---

5. What Is the Java String Pool?

The **String Pool** is a special memory area in the JVM that stores **literal strings** to optimize memory usage and improve performance.

How It Works

1. Strings created with literals (`String s1 = "Hello";`) are automatically placed in the String Pool.
2. When a new literal is encountered, the JVM checks if the same string already exists in the pool. If yes, it reuses the existing object.
3. Strings created with `new` (`String s2 = new String("Hello");`) are stored in the heap, not the pool.

Example

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = new String("Hello");

System.out.println(s1 == s2);  // true (points to same pool object)
System.out.println(s1 == s3);  // false (different heap object)
```

1. Differences Between `String` and `StringBuffer`

Feature	`String`	`StringBuffer`
Mutability	Immutable	Mutable
Thread Safety	Thread-safe (Immutable objects are inherently thread-safe)	Thread-safe (synchronized methods)
Performance	Slower for frequent changes as it creates new objects for modifications	Faster than `String` for modifications due to mutability
Use Case	Best for read-only operations	Best when multiple updates to the string are needed

Example:

```
String str = "Hello";
str = str.concat(" World");  // Creates a new object
System.out.println(str);  // Hello World

StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");  // Updates the same object
System.out.println(sb);  // Hello World
```

2. Differences Between `StringBuffer` and `StringBuilder`

Feature	`StringBuffer`	`StringBuilder`
Mutability	Mutable	Mutable
Thread Safety	Thread-safe (synchronized methods)	Not thread-safe
Performance	Slower due to synchronization	Faster due to lack of synchronization
Use Case	Best for multi-threaded environments	Best for single-threaded environments

---

3. Which Should Be Preferred: `StringBuffer` or `StringBuilder`?

- **Prefer** `StringBuilder` when working in **single-threaded** environments for better performance.
- **Use** `StringBuffer` in **multi-threaded** applications where thread safety is required.

---

4. How to Create an Immutable Class in Java?

To create an immutable class in Java, follow these guidelines:

1. **Declare the class as** `final` to prevent subclassing.
2. **Make all fields private and final.**
3. **Do not provide setter methods.**
4. **Initialize all fields through a constructor.**
5. **Ensure deep copies of mutable objects are returned in getters.**

Example:

```
final class ImmutablePerson {
    private final String name;
    private final int age;

    public ImmutablePerson(String name, int age) {
        this.name = name;
```

```
        this.age = age;
    }

    public String getName() {
        return name;
    }

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

Here, no external changes can modify the internal state of `ImmutablePerson`.

---

5. What Is the Wrapper Class in Java? Why Do We Need Them?

Definition:

A **wrapper class** is a class that encapsulates primitive data types into objects.

Why Do We Need Wrapper Classes?

1. **Collection Framework Compatibility**: Collections like `List`, `Set`, and `Map` store objects, not primitives.
2. **Autoboxing and Unboxing**: Wrappers allow automatic conversion between primitives and objects (`int` ↔ `Integer`).
3. **Utility Methods**: Wrappers provide useful methods, such as `Integer.parseInt()` and `Double.valueOf()`.
4. **Null Handling**: Unlike primitives, wrapper classes can store `null`.

Examples of Wrapper Classes:

Primitive	Wrapper Class
`int`	`Integer`
`char`	`Character`

Primitive	Wrapper Class
`float`	`Float`
`boolean`	`Boolean`

Example Usage:

```
List<Integer> list = new ArrayList<>();
list.add(10);  // Autoboxing
int num = list.get(0);  // Unboxing
```

1. What Is a Static Variable?

A **static variable** in Java is a class-level variable shared among all instances of the class. It belongs to the class rather than any individual instance.

Key Characteristics:

- Declared using the keyword `static` .
- Initialized only once, at class loading time.
- Shared by all objects of the class.

Example:

```
class Counter {
    static int count = 0;

    public void increment() {
        count++;
    }

    public static void main(String[] args) {
        Counter obj1 = new Counter();
        Counter obj2 = new Counter();
        obj1.increment();
        obj2.increment();
```

```
        System.out.println("Count: " + Counter.count);  // Output: 2
    }
}
```

---

2. Difference Between Instance Variable and Class Variable

Aspect	Instance Variable	Class Variable (Static Variable)
Belongs To	An individual object	Class
Memory Location	Stored in heap memory	Stored in method area
Initialization	Initialized when object is created	Initialized when class is loaded
Access	Requires an object	Accessed via class name or object
Sharing	Not shared among objects	Shared among all objects

---

3. What Is the `transient` Keyword?

The `transient` keyword is used to indicate that a field should **not be serialized** during object serialization.

This is useful when certain data is sensitive or temporary and doesn't need to be saved.

Example:

```
import java.io.*;

class Person implements Serializable {
    String name;
    transient int age;  // Will not be serialized

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```
public class TestTransient {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        Person person = new Person("John", 30);
        FileOutputStream fos = new FileOutputStream("person.dat");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(person);
        oos.close();

        FileInputStream fis = new FileInputStream("person.dat");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Person deserializedPerson = (Person) ois.readObject();
        ois.close();

        System.out.println("Name: " + deserializedPerson.name);  // John
        System.out.println("Age: " + deserializedPerson.age);    // 0
    }
}
```

---

4. What Are Access Specifiers and Their Types?

Access specifiers define the **scope** or **visibility** of a class, method, or variable.

Types of Access Specifiers:

Specifier	Scope in Class	Scope in Package	Scope in Subclass	Scope Outside Package
Public	Yes	Yes	Yes	Yes
Private	Yes	No	No	No
Protected	Yes	Yes	Yes	No
Default (no keyword)	Yes	Yes	No	No

---

5. What Is a Marker Interface? Why Do We Need It?

A **marker interface** is an interface with **no methods or fields**. It is used to indicate metadata or a special property of a class to the JVM or frameworks.

Examples of Marker Interfaces:

- `Serializable` : Indicates that objects can be serialized.
- `Cloneable` : Indicates that objects can be cloned.

Why We Need Marker Interfaces:

- To enable specific behaviors like object serialization (`Serializable`).
- Simplifies type checking at runtime.

---

6. Differences Between Abstract Class and Interface

Aspect	Abstract Class	Interface
Methods	Can have both abstract and non-abstract methods	All methods are implicitly abstract (prior to Java 8)
Fields	Can have instance variables	Can only have constants (`public static final`)
Multiple Inheritance	Not supported	Supported
Constructors	Can have constructors	Cannot have constructors
Access Modifiers	Can be private, protected, or public	Methods are public by default

Example Abstract Class:

```
abstract class Animal {
    abstract void sound();

    void eat() {
        System.out.println("Eating...");
    }
}
```

Example Interface:

```
interface Flyable {
    void fly();
}
```

1. What Do You Mean by Encapsulation?

Encapsulation is the process of **hiding the internal details of a class** and only exposing a controlled interface to the outside world. This is achieved by:

- Declaring instance variables as **private**
- Providing **public getter and setter methods** to access and modify the variables

Benefits of Encapsulation:

- **Data hiding:** Prevents unauthorized access
- **Improved maintainability:** Internal code changes don't affect external classes
- **Increased flexibility:** Allows validation in setters

Example:

```
class Person {
    private String name;  // Encapsulated field

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if (name != null && !name.trim().isEmpty()) {
            this.name = name;
        } else {
            System.out.println("Invalid name");
        }
    }
```

```
}

public class TestEncapsulation {
    public static void main(String[] args) {
        Person person = new Person();
        person.setName("John");
        System.out.println("Name: " + person.getName());
    }
}
```

---

2. What Is Polymorphism?

Polymorphism is the ability of an object to take **multiple forms**. In Java, it is mainly achieved through **method overloading** and **method overriding**.

Types of Polymorphism:

1. **Compile-time (static) polymorphism:** Achieved via method overloading
2. **Runtime (dynamic) polymorphism:** Achieved via method overriding

Example of Compile-time Polymorphism:

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
}

public class TestPolymorphism {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
```

```
        System.out.println(calc.add(2, 3));        // 5
        System.out.println(calc.add(2, 3, 4));     // 9
    }
}
```

Example of Runtime Polymorphism:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

public class TestPolymorphism {
    public static void main(String[] args) {
        Animal animal = new Dog();  // Upcasting
        animal.sound();  // Dog barks
    }
}
```

---

3. What Is Inheritance?

Inheritance is the mechanism where one class **acquires properties and behaviors** from another class.

- The **parent class (superclass)** provides common functionality.
- The **child class (subclass)** inherits and can extend or override the parent class methods.

Types of Inheritance:

1. Single inheritance
2. Multilevel inheritance
3. Hierarchical inheritance

   (Note: Java does not support multiple inheritance with classes to avoid ambiguity but supports it with interfaces.)

Example:

```
class Vehicle {
    void start() {
        System.out.println("Vehicle is starting");
    }
}

class Car extends Vehicle {
    void drive() {
        System.out.println("Car is driving");
    }
}

public class TestInheritance {
    public static void main(String[] args) {
        Car car = new Car();
        car.start();  // Inherited from Vehicle
        car.drive();  // Defined in Car
    }
}
```

Benefits:

- Code reusability
- Faster development
- Easy maintenance

---

4. What Is Abstraction?

Abstraction is the process of **hiding the implementation details** while exposing only the essential functionalities to the user.

Achieved Through:

1. **Abstract classes:** Partially implemented classes
2. **Interfaces:** Fully abstract types

Benefits:

- Increases code maintainability
- Reduces code complexity
- Provides flexibility

Example Using Abstract Class:

```
abstract class Animal {
    abstract void sound();  // Abstract method

    void eat() {  // Concrete method
        System.out.println("This animal eats food");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

public class TestAbstraction {
    public static void main(String[] args) {
        Animal animal = new Cat();
        animal.sound();  // Cat meows
        animal.eat();    // This animal eats food
```

```
    }
}
```