**Java Theoretical**

Q1 - Why Java instead of other programming languages.

I prefer Java because it's platform-independent, object-oriented, and has strong community support. It’s great for building scalable, secure applications, and its vast libraries and frameworks make development faster and more efficient.

* Platform independence means the code can run on any operating system without changes. This saves time, increases flexibility.
* Object-oriented programming makes code more modular, reusable, and easier to maintain. It helps break complex problems into smaller objects, each with clear roles, which improves readability and scalability.
* **Scalable** means the ability of a system or application to handle **increased load or growth**—like more users, data, or transactions—**without slowing down or crashing**.

A **scalable application** can grow in terms of:

* **Users** (more traffic)
* **Data** (larger databases)
* **Features** (adding new functions without breaking the old ones)

**Frameworks:**

* Spring
* Spring Boot
* Hibernate
* Struts

**\*Libraries:**

* **java.lang**  
  Fundamental classes like String, Math, Object, Thread, etc.
* **java.util**  
  Collections framework (List, Set, Map), Date & Time utilities, Random, etc.
* **java.io**  
  Input/output classes for reading and writing data (File, InputStream, OutputStream).

Q2- Java is said to be *platform-independent*. Can you explain**why** and how Java is platform-independent?

* Java is platform-independent because of the concept of **bytecode** and the **JVM**. When we compile a Java program, it is not directly converted into machine code. Instead, it is converted into an intermediate form called *bytecode*. This bytecode is the same on all systems. Then, the JVM on each operating system translates this bytecode into the native machine code of that OS. That’s why the same Java program can run on Windows, Linux, or Mac without modification — we just need a JVM for that platform.

Q3- Explain the difference between **JDK, JRE, and JVM.**

* **JDK (Java Development Kit):** It is a complete package that provides tools to develop, compile, and debug Java programs. It includes the JRE and development tools like the compiler (javac).
* **JRE (Java Runtime Environment):** It is the environment required to run Java programs. It contains the JVM + libraries and classes but does not have development tools.
* **JVM (Java Virtual Machine):** It is an abstract machine that executes Java bytecode line by line. It makes Java platform-independent by converting bytecode into machine code according to the underlying OS.

Q4- What are the main features of Java ?

* **Robust :** Java has strong memory management, exception handling, and type checking mechanisms, which make programs less error-prone and more reliable.
* **Secure :** Java provides a secure environment by using bytecode verification, runtime security checks, and the sandbox model, which restricts access to system resources.
* **Multithreaded :** Java supports multithreading, allowing multiple threads to run concurrently, which helps in better CPU utilization and smoother program execution.

Q5- What is the difference between == and .equals() in Java? When would you use each?

* **==operator:**  
  It checks if two reference variables point to the **same object** in memory. In other words, it compares the **memory addresses** of the objects.
* **.equals()method:**  
  It is used to compare the **contents or values** of two objects to see if they are logically equal. This method can be overridden by classes to define what equality means for their objects (like in String, Integer, etc.).

**Example**

String s1 = new String("hello");

String s2 = new String("hello");

System.out.println(s1 == s2); // false, because s1 and s2 are different objects

System.out.println(s1.equals(s2)); // true, because their contents are the same.

Q6 - What are the access modifiers in Java? Can you name them and explain their scope?

* **public:** The member is accessible from **anywhere** in the program, across all packages.
* **private:** The member is accessible **only within the same class**. It cannot be accessed from outside the class.
* **protected:**  
  The member is accessible within the **same package** and also in **subclasses** (even if they are in different packages).
* **Default (no modifier):**  
  When no access modifier is specified, it is called **package-private** or default access. The member is accessible only within the **same package**.

Q7 - What is the difference between an abstract class and an interface in Java? When would you use each?

* **Abstract Class:**
* Can have **both abstract and non-abstract methods** (methods with or without implementation).
* Can have **constructors** and **instance variables**.
* Supports **inheritance** (a class can only extend one abstract class).
* Used when classes share a **common base** with some default behavior.

**Example-**

abstract class Animal {

abstract void sound();

}

class Dog extends Animal {

void sound() {

System.out.println("Dog barks");

}

}

* **Interface:**
* An **interface** in Java is like a **contract** or **rulebook** that a class agrees to follow. It contains only method **declarations**—no actual code or logic inside the methods.
* When a class **implements** an interface, it must provide the **actual code** for all the methods declared in that interface.
* Interfaces are mainly used when we want **different classes to follow the same set of rules**, even if they’re not related. It also helps Java support **multiple inheritance**, since a class can implement **more than one interface**.

**Example-**

interface Flyable {

void fly();

}

class Bird implements Flyable {

public void fly() {

System.out.println("Bird flies");

}

}

Q8- What is the difference between ArrayList and LinkedList in Java? When would you use one over the other?

* **ArrayList :**
* Internally uses a **dynamic array** to store elements.
* Provides **fast random access** (get() operation is O(1)) because elements are stored in contiguous memory.
* Adding or removing elements at the **end** is fast (amortized O(1)), but adding or removing elements in the **middle or beginning** requires shifting elements, which is slower (O(n)).
* Better when you need **fast access** and mostly add/remove at the end.
* **LinkedList :**
* Internally uses a **doubly linked list** data structure.
* Does **not provide fast random access** (get() operation is O(n)) because it has to traverse nodes sequentially.
* Adding or removing elements anywhere in the list is **fast** (O(1)) if you already have a reference to the node.
* Better when you need **frequent insertions and deletions** in the middle or at the beginning of the list.

Q9-What is Exception Handling in Java? Can you explain the difference between checked and unchecked exceptions?

* **Exception Handling** in Java is a mechanism to handle runtime errors so that the normal flow of the program is maintained. It helps in catching and managing errors gracefully instead of abruptly terminating the program.
* Java provides keywords like try, catch, finally, throw, and throws to handle exceptions.

| **Feature** | **Checked Exceptions** | **Unchecked Exceptions** |
| --- | --- | --- |
| Checked by Compiler? | Yes, compiler forces you to handle them. | No, compiler does not check. |
| When occur? | At compile time or during file I/O, etc. | At runtime, usually programming errors (like divide by zero). |
| Examples | IOException, SQLException | NullPointerException, ArithmeticException |

* **Checked exceptions** must be either caught using a try-catch block or declared with the throws keyword.
* **Unchecked exceptions** are runtime exceptions that usually indicate programming bugs and don’t require explicit handling.
* The **try block** contains the code that might cause an exception.
* The **catch block** handles the exception if it happens.
* The **finally block** contains code that runs **always**, whether an exception occurs or not, usually to clean up resources.

**Example-**

try {

int result = 10 / 0; // This causes ArithmeticException

}

catch (ArithmeticException e) {

System.out.println("Cannot divide by zero");

}

finally {

System.out.println("This always runs");

}

Q 10- What is the difference between final, finally, and finalize in Java?

**1. final (Keyword):**

* Used to declare **constants**, **prevent method overriding**, or **inheritance**.
* Can be applied to **variables**, **methods**, and **classes**.

🔹 **Examples:**

final int x = 10 ; // variable can't be changed

final void show() {} // method can't be overridden

final class Test {} // class can't be extended

**2. finally (Block):**

* Used in **exception handling**.
* The finally block **always executes**, whether an exception is handled or not.
* Used to **close resources** like files, database connections, etc.

🔹 **Example:**

try {

int a = 5 / 0;

}

catch (ArithmeticException e)

{

System.out.println("Exception caught");

}

finally {

System.out.println("This will always execute");

}

**3. finalize() (Method):**

* It's a **method** in the Object class.
* Called by the **garbage collector** before an object is destroyed.
* Rarely used now (and deprecated in newer Java versions).

🔹 **Example:**

protected void finalize() throws Throwable

{

System.out.println("Object is being garbage collected");

}

Q11- What is the difference between String, StringBuilder, and StringBuffer in Java?

**1. String**

* **Immutable**: Once created, the value of a String cannot be changed.
* Every modification creates a **new object** in memory.
* Slower when doing many modifications.

🔹 **Example:**

String s = "Hello";

s = s + " World"; // creates a new String object

**2. StringBuilder**

* **Mutable**: Can be modified without creating new objects.
* **Not thread-safe** (not synchronized), but faster than StringBuffer.
* Best for **single-threaded** applications where performance matters.

🔹 **Example:**

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World"); // modifies the same object

**3. StringBuffer**

* **Mutable** like StringBuilder.
* **Thread-safe** (synchronized), so it's safe to use in **multi-threaded** environments.
* Slightly slower than StringBuilder because of synchronization.

🔹 **Example:**

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World"); // thread-safe modification

Q12- What are constructors in Java? Can you explain the types of constructors with examples?

* A **constructor** is a special method that is used to **initialize objects**. It has the same name as the class and **does not have a return type**, not even void.

**Types of Constructors in Java:**

1. **Default Constructor**

* Created automatically by the compiler if no constructor is defined.
* Takes **no parameters**.

🔹 **Example:**

class Student {

Student() {

System.out.println("Default constructor called");

}

}

1. **Parameterized Constructor**

* Takes **arguments** to initialize objects with specific values.

🔹 **Example:**

class Student {

String name;

Student(String n) {

name = n;

}

}

1. **Copy Constructor *(Not built-in, but can be created manually)***

* Used to create a new object by copying values from another object.

🔹 **Example:**

class Student {

String name;

Student(String n) {

name = n;

}

Student(Student s) {

name = s.name;

}

}

Q 13- What is inheritance in Java? What are the different types, and how is it implemented?

* **Inheritance** in Java is a mechanism where one class (called a **subclass** or **child class**) can inherit the properties and behaviors (fields and methods) of another class (called a **superclass** or **parent class**).
* It helps in **code reusability** and makes programs easier to manage.

**Types of Inheritance in Java:**

1. **Single Inheritance:** One class inherits from one parent class.  
   🔹 Example: class Dog extends Animal
2. **Multilevel Inheritance:** A class inherits from a class which is already inherited from another class.

🔹 Example : class Puppy extends Dog extends Animal

1. **Hierarchical Inheritance:** Multiple classes inherit from a single parent class.

🔹 Example : Cat, Dog, Cow all inherit from Animal.

**Example-**

class Animal {

void eat() {

System.out.println("This animal eats food");

} }

class Dog extends Animal {

void bark() {

System.out.println("Dog barks");

}

}

Q14- What is polymorphism in Java? Explain its types with examples?

* **Polymorphism** means "**many forms**." In Java, it allows a single method or object to take multiple forms depending on how it’s used.
* **Method Overloading:**
* Happens **within the same class**.
* Same method name, but **different parameters** (number, type, or order).
* It is a type of **compile-time (static) polymorphism.**

**Example-**

class Math {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

}

* **Method Overriding:**
* Happens **between a superclass and a subclass**.
* Same method name, **same parameters**, but different implementation.
* It is a type of **run-time (dynamic) polymorphism**.

class Animal {

void sound() {

System.out.println("Animal makes sound");

}

}

class Dog extends Animal {

void sound() {

System.out.println("Dog barks");

}

}

Q15-Can you explain what a Java Package is? What are the benefits of using packages?

* In Java, a **package** is a **namespace** that organizes a set of related classes and interfaces. Think of it like a folder in your computer that groups related files together.
* **Benefits of Using Packages:**

1. **Organizes code:** Helps in logically grouping related classes and interfaces, making the project easier to manage.
2. **Avoids name conflicts:** Different packages can have classes with the same names without conflicts.
3. **Access control:** Packages provide access protection by controlling the visibility of classes, methods, and variables.
4. **Reusability:** Helps in modular programming by allowing classes to be reused across different projects.
5. **Easier maintenance:** Since related classes are grouped, it’s easier to update and maintain the code.

* **How to declare a package:** package com.example.myapp;

Q16 -What is garbage collection in Java? How does it work?

* Garbage collection in Java is an automatic process that frees up memory by removing objects no longer used by the program. It runs in the background and helps prevent memory leaks, so programmers don’t have to manually manage memory.

Q17-What is multithreading in Java? What are its advantages?

* Multithreading in Java is a process where multiple threads run simultaneously within a program, allowing tasks to be performed concurrently.

**Advantages of Multithreading:**

1. **Improved performance** by doing multiple tasks at the same time.
2. Better **resource utilization**, especially on multi-core processors.
3. Makes programs **more responsive**, for example, in user interfaces.
4. Helps in **parallel processing** for faster execution.

Q18-What is the difference between throw and throws in Java?

* Both throw and throws are related to exception handling but have different uses:
* **Throw**  is used to **actually throw an exception** explicitly in a method. It is followed by an instance of Throwable.

Example: throw new ArithmeticException("Divide by zero");

* **throws** is used in a **method declaration** to declare that the method might throw one or more exceptions, informing the caller to handle or declare them.

Example:

public void readFile() throws IOException {

// method code

}

**Summary:**

* throw is used to **throw** an exception.
* throws is used to **declare** exceptions that a method can throw.

Q19-What is the purpose of the static keyword in Java?

The static keyword in Java is used to **declare members (variables or methods) that belong to the class, rather than to any specific instance** of the class.

1. **Static variables** are shared among all objects of the class.
2. **Static methods** can be called without creating an object.
3. Used for **utility or helper methods** like Math.max().
4. Static blocks are used to initialize static variables.

**Example:**

class Example {

static int count = 0;

static void display() {

System.out.println("Count: " + count);

}

}

Q20- What are **functional interfaces** in Java? How are they related to **lambda expressions**?

* A **functional interface** is an interface with **only one method** to implement. It is used with **lambda expressions** to write shorter and easier code.
* **For example-**

@FunctionalInterface

interface Calculator {

int add(int a, int b);

}

* Lamda-

Calculator calc = (a, b) -> a + b;

System.out.println(calc.add(5, 3)); // Output: 8

Q21- What is extends in java?

* The keyword extends is used for **inheritance**, which is a fundamental concept in object-oriented programming (OOP). It allows one class to inherit fields and methods from another class.

Q22- What is a constructor in Java? How is it different from a method?

* A constructor is a special method used to initialize objects. It has the same name as the class and **does not have a return type**, not even void. Unlike methods, it is called automatically when an object is created.

Q23- What are the types of constructors in Java?

1. **Default Constructor**

* A **default constructor** is a constructor **without any parameters**. It is either:
* **Implicitly provided by the compiler** if no constructor is defined in the class.
* Or, **explicitly defined** by the programmer.

**Example-**

public class Person {

String name; // instance variable

int age; // instance variable

}

public class Main {

public static void main(String[] args) {

Person p = new Person(); // Java uses default constructor here

System.out.println(p.name); // prints: null

System.out.println(p.age); // prints: 0

}

**}**

1. **Parameterized Constructor**

* A **parameterized constructor** is a constructor that **takes arguments** to initialize an object with **custom values**.

**Example-**

class Car {

String model;

Car(String m) { // Parameterized constructor

model = m;

}

public static void main(String[] args) {

Car c = new Car("Tesla");

System.out.println("Model: " + c.model); // O/P: Model: Tesla

}

}

Q24- Can a constructor call another constructor in the same class?

* Yes, using the this() keyword. Must be the **first statement** in the constructor.

**Example-**

public class Test {

Test() {

this(5);

}

Test(int x) {

System.out.println("Value: " + x);

}

}

Q25- What is the super keyword?

* The super keyword is used to **access things from the parent class** — like methods or constructors — from a child class.

**Example-**

class Animal {

void speak() {

System.out.println("Animal speaks");

}

}

class Dog extends Animal {

void speak() {

super.speak(); // Calls the method from Animal class

System.out.println("Dog barks");

}

}

Q26- Explain different types of variables in class ?

1. **Instance Variables (Non-static variables)**
   * Declared inside a class but outside any method, constructor, or block.
   * Belong to each object (instance) of the class.
   * Every object has its own copy.
   * Default values are assigned if not initialized (e.g., 0 for int, null for objects).

**Example:**

class Person {

String name; // instance variable

}

1. **Static Variables (Class variables)**

* Declared with the static keyword inside a class but outside methods.
* Shared among all instances of the class.
* Initialized only once when the class is loaded.

**Example:**

class Person {

static int population; // static variable

}

1. **Local Variables**
   * Declared inside a method, constructor, or block.
   * Only accessible within that method or block.
   * Must be initialized before use.
   * Not assigned default values.

**Example:**

void greet() {

int hour = 10; // local variable

}

1. **Parameters**
   * Variables passed to methods or constructors.
   * Only accessible within the method or constructor.

**Example:**

void setName(String name) { // name is a parametervariable

this.name = name;

}

Q27- What are the four pillars of OOP?

**1. Encapsulation** : Encapsulation is the concept of **hiding the internal details** of an object and only exposing what is necessary. It protects the internal state of an object from unintended or harmful changes.

**Real-world example:**  
Think of a **coffee machine**. You press a button to make coffee, but you don’t know (or need to know) how it grinds the beans or heats the water internally. The complexity is hidden — that's encapsulation.

**2. Abstraction :** Abstraction means **focusing on what an object does** instead of how it does it. It allows us to work with ideas or behaviors rather than concrete implementations.(hiding internal implementation )

**Real-world example:**  
A **TV remote** provides buttons for volume, power, and channels. You don’t know exactly how those commands are implemented inside the TV — you just use them.

**3. Inheritance** : Inheritance is the mechanism by which one class (**child**) can **acquire the properties and behaviors of another class** (**parent**). It promotes code reuse.

**Real-world example:**  
A **car** is a type of **vehicle**. The vehicle class can have general features like speed or fuel capacity, which are shared with the car, truck, or bike classes.

**4. Polymorphism :** Polymorphism allows objects of different classes to be treated as objects of a common superclass. It means **many forms** — the same function can behave differently depending on the object that invokes it.

**Real-world example:**  
The word **“draw”** means different things depending on context drawing with a pencil, drawing money from a bank, or drawing a gun. Same action, different results.

Q28- What is the difference between a class and an object?

* A **class** is a blueprint or template that defines attributes and behaviors, while an **object** is an instance of a class. For example, if Car is a class, then a specific car like myCar is an object of that class.

Q29- Difference between abstraction and encapsulation?

* **Abstraction** focuses on *hiding implementation details* and showing only essential features.
* **Encapsulation** is about *protecting data* by bundling it with methods that operate on it.
* Abstraction is more about design, while encapsulation is about implementation.
* Top of Form
* Bottom of Form