Diffrences in java

☑ 1. OOP vs POP (Object-Oriented Programming vs Procedural-Oriented Programming)

Feature	OOP (Object-Oriented Programming)	POP (Procedural-Oriented Programming)
Basic Concept	Organizes code around objects (data + methods).	Organizes code around procedures or functions.
Modularity	Code is modular due to encapsulation and classes.	Less modular, functions and data are separate.
Data Handling	Emphasizes data security using encapsulation.	Data is exposed and shared between functions.
Code Reusability	Achieved through inheritance and polymorphism.	Limited; code reuse is done via function calls or copy-paste.
Examples in Java	Class, Object, Inheritance, Polymorphism.	Main method with functions and sequential logic.
Maintainability	Easier to maintain and scale.	Becomes complex as size increases.
Example Languages	Java, C++, Python (with classes).	C, Pascal, Basic.

☑ 2. Abstraction vs Encapsulation

Feature	Abstraction	Encapsulation
Definition	Hides implementation details and shows only essential info.	Binds data and methods into a single unit (class).
Focus	Focuses on **what** an object does.	Focuses on **how** data is protected and managed.
Achieved Using	Abstract classes and interfaces.	Access modifiers (private, public, protected).
Access Control	Not necessarily controlling access directly.	Strictly controls access to internal state.
Example in Java	`abstract class` or `interface` with unimplemented methods.	Using `private` variables and `public` getters/setters.
Purpose	Reduce complexity by hiding implementation.	Safeguard internal state of object.

☑ 3. Inheritance vs Polymorphism

Feature	Inheritance	Polymorphism
Definition	Allows one class to inherit properties/methods from another.	Allows methods to behave differently based on the object.
Purpose	Code reuse and method overriding.	Flexibility and dynamic behavior.
Types	Single, Multilevel, Hierarchical (Java doesn't support multiple).	Compile-time and Run-time.

Java Example	`class B extends A`	Overriding or overloading a method.
Relation	Is a mechanism (is-a relationship).	Is a behavior (many forms).
Use Case	Sharing common logic across related classes.	Executing behavior specific to an object type at runtime.

☑ 4. Overloading vs Overriding

Feature	Overloading	Overriding
Definition	Defining multiple methods with same name but different parameters.	Redefining a superclass method in a subclass.
Method Signature	Must differ in parameter type, number, or order.	Must be same as superclass method.
Class Relationship	Happens within the same class.	Happens across superclass and subclass.
Polymorphism Type	Compile-time polymorphism.	Run-time polymorphism.
Java Support	Yes (method overloading, constructor overloading).	Yes (method overriding using `@Override`).
Use Case	Increase method flexibility.	Customize superclass behavior.

☑ 5. Compile-time vs Run-time Polymorphism

Feature	Compile-time Polymorphism	Run-time Polymorphism
Also Known As	Method Overloading	Method Overriding
Binding Time	Resolved during compile time.	Resolved during run time.
How it's Achieved	By method overloading or operator overloading.	By method overriding with inheritance.
Performance	Faster (no need to decide method at runtime).	Slightly slower due to dynamic method resolution.
Flexibility	Less flexible.	More flexible and extensible.
Example in Java	`void sum(int a, int b)` and `void sum(double a, double b)`	Superclass reference pointing to subclass object.
Java Mechanism	Compiler decides which method to invoke.	JVM decides at runtime which method to invoke.

☑ 1. Interface vs Abstract Class

Feature	Interface	Abstract Class
Purpose	Defines a contract that implementing classes must follow.	Serves as a base class for subclasses with shared code.
Keyword	`interface`	`abstract class`

Method Implementation	All methods are abstract by default (Java 7); Java 8+ allows `default` and `static` methods.	Can have both abstract and concrete methods.
Inheritance	A class can implement multiple interfaces.	A class can extend only one abstract class.
Constructors	Cannot have constructors.	Can have constructors.
Access Modifiers	Methods are implicitly `public abstract`.	Can use any access modifier (`public`, `protected`, etc.).
Use Case	When multiple unrelated classes need to share method signatures.	When you want to share common code and define a common base.

☑ 2. `final` VS `finally` VS `finalize`

Keyword	`final`	`finally`	`finalize()`
Type	Modifier (applies to variables, methods, and classes).	Block (used with `try-catch`).	Method (inherited from `Object` class).
Purpose	Prevents modification (e.g., inheritance or reassignment).	Ensures code runs after try-catch, regardless of exception.	Used for cleanup before object is garbage collected.
Usage Example	`final int x = 10;` `final class MyClass {}`	`try { ... }` `finally { ... }`	`protected void finalize()` `throws Throwable {}`
Can be Overridden?	No, methods or classes marked `final` can't be overridden/extended.	Not applicable.	Can be overridden in custom classes (though deprecated).
Modern Use	Frequently used.	Common in resource management.	Rarely used, deprecated in Java 9+.

☑ 3. `throw` VS `throws`

Feature	`throw`	`throws`
Purpose	Used to explicitly throw an exception.	Declares that a method might throw an exception.
Placement	Inside method body.	In method signature.
Syntax	`throw new IOException("Error");`	`void readFile() throws IOException {}`
Can throw multiple exceptions?	No, only one exception at a time.	Yes, can declare multiple exceptions.
Used With	Instance of `Throwable` subclasses.	Used to indicate the caller must handle or declare.

☑ 4. Error vs Exception

Feature	Error	Exception
Definition	Serious issues that a program should not handle.	Issues that a program can handle or recover from.
Hierarchy	Subclass of `Throwable`.	Also subclass of `Throwable`.
Examples	`OutOfMemoryError`, `StackOverflowError`.	`IOException`, `NullPointerException`, `SQLException`.
Handling	Generally not caught using try-catch.	Usually handled using try-catch blocks.
Use Case	Represent JVM/system failures.	Represent logical or runtime problems in code.

☑ 5. Thread class vs Runnable interface

Feature	Thread Class	Runnable Interface
Definition	Represents a thread of execution.	Represents a task to be executed in a thread.
Inheritance	Extends `Thread` class.	Implements `Runnable` interface.
Multiple Inheritance	Not allowed (Java only supports single inheritance).	Allowed (can implement multiple interfaces).
Code Reuse	Less flexible; can't extend another class.	More flexible; can extend another class simultaneously.
Syntax Example	`class MyThread extends Thread { public void run() {...} }`	`class MyRunnable implements Runnable { public void run() {...} }`
Execution	`new MyThread().start();`	`new Thread(new MyRunnable()).start();`
Best Practice	Not preferred if your class needs to extend something else.	Preferred approach for better design and separation.

☑ 31. Stack vs Queue

Feature	Stack	Queue
Definition	Linear data structure that follows **LIFO** (Last In First Out).	Linear data structure that follows **FIFO** (First In First Out).
Insertion Method	`push(element)` – adds to the top.	`enqueue(element)` – adds to the rear.
Removal Method	`pop()` – removes the top element.	`dequeue()` – removes the front element.
Access	Only top element is accessible.	Only front and rear are accessible.
Java Implementation	`Stack<Integer> stack = new Stack<>();`	`Queue<Integer> queue = new LinkedList<>();`
Use Cases	Function calls (call stack), undo operations, expression evaluation.	Print queue, task scheduling, resource sharing.

Example	Stack: `[10, 20, 30] → pop() = 30`	Queue: `[10, 20, 30] → dequeue() = 10`

☑ 32. JVM vs JRE vs JDK

Feature	JVM (Java Virtual Machine)	JRE (Java Runtime Environment)	JDK (Java Development Kit)
Definition	Executes Java bytecode.	Provides environment to run Java applications.	Provides tools to develop and run Java programs.
Contains	Part of JRE and JDK.	Includes JVM + libraries + other files.	Includes JRE + compiler (`javac`) + development tools.
Use Case	Runs compiled `.class` files.	Runs Java applications on end-user systems.	Used by developers to write and compile Java code.
Tools Included	No tools (only runtime engine).	No compiler or debugger.	Includes tools like `javac`, `javadoc`, `jdb`, etc.
Required For	Running Java programs.	Running Java programs.	Developing and compiling Java programs.
Example	JVM loads and executes bytecode via `java` command.	JRE installed to run `.jar` files.	JDK used for compiling via `javac`.

☑ 33. Static Binding vs Dynamic Binding

Feature	Static Binding (Early Binding)	Dynamic Binding (Late Binding)
Binding Time	Resolved at **compile time**.	Resolved at **runtime**.
Based On	Reference type.	Object type.
Performance	Faster, as it happens at compile time.	Slightly slower due to runtime resolution.
Applicable To	Static methods, final methods, private methods, overloaded methods.	Overridden methods.
Polymorphism Type	Compile-time polymorphism.	Runtime polymorphism.
Example	`java class A { void show(int x) {...} }`	`java class A { void show() {} } class B extends A { void show() {} }`
Java Mechanism	Overloading, static methods.	Overriding, method invocation via superclass reference.