**Java Versions & Features**

* **Java 8 (2014):** Lambdas, Streams, Optional, Default methods, CompletableFuture.
* **Java 11 (2018):** var (local inference), HttpClient, String methods (isBlank, repeat, strip).
* **Java 17 (2021 LTS):** Sealed Classes, Switch Expressions, Pattern Matching instanceof, Text Blocks.
* **Java 21 (2023 LTS):** Virtual Threads (Loom), Pattern Matching for Switch, Sequenced Collections, Record Patterns.

**JDK, JRE, JVM**

* **JVM:** Runs bytecode, manages GC, JIT.
* **JRE:** JVM + libraries (runtime only).
* **JDK:** JRE + dev tools (javac, debugger, etc.).

**Memory & Runtime**

* **Heap:** Young (Eden + Survivor), Old (Tenured).
* **Metaspace (Java 8+):** Class metadata.
* **Stack:** Method frames, locals.
* **GC:** G1GC, ZGC, Parallel GC.
* **JIT vs Interpreter:** Interpreter = line-by-line, JIT = hotspot native compilation → faster.

**Key Language Enhancements**

* **Records (Java 16+):** Immutable data carrier, auto equals, hashCode, toString.
* record Employee(String name, int id) {}
* **Sealed Classes (Java 17):** Restrict inheritance (permits).
* public sealed class Shape permits Circle, Rectangle {}
* **Switch Expressions (Java 14+):** Concise & returns values.
* int r = switch(day){case MONDAY->6; default->0;};
* **Text Blocks (Java 15+):** Multi-line strings (""" ... """).

**Modules (Java 9+)**

* Encapsulation via module-info.java.
* Example:
* module com.myapp {
* requires java.sql;
* exports com.myapp.services;
* }
* Enables smaller runtime images (jlink).

**Latest (Java 21 Highlights)**

* **Virtual Threads:** Millions of lightweight threads.
* **Record Patterns:** Pattern deconstruction in switch/if.
* **Sequenced Collections:** Ordered list/set/map API.
* **GC & perf:** Further optimizations.

**Rules for Subclasses:**

* A subclass of a **sealed class** must declare itself as **final**, **sealed**, or **non-sealed**:
  1. **final** → cannot be extended further.
  2. **sealed** → can be extended, but restricts further again.
  3. **non-sealed** → removes restrictions (open extension).

**1. What are the 4 pillars of OOP, and how have you applied them in complex systems?**

**Answer:**

* **Encapsulation:** Hiding internal state via private fields + public APIs (e.g., DTOs, Entities). Used in microservices to ensure controlled exposure.
* **Inheritance:** Reusing behavior (abstract base classes for payment methods). Used cautiously to avoid tight coupling.
* **Polymorphism:** Common interface but different implementations (e.g., PaymentProcessor for CreditCard, Wallet, UPI). Used in strategy patterns.
* **Abstraction:** Hiding details via interfaces/abstract classes (e.g., repository interfaces for persistence).

**2. What’s the difference between composition and inheritance? Which do you prefer?**

**Answer:**

* **Inheritance:** “is-a” relationship. Risk of fragile hierarchies and tight coupling.
* **Composition:** “has-a” relationship, favors flexibility.
* **Best Practice:** Prefer **composition over inheritance**. For example, instead of class FlyingCar extends Car, use class Car { Engine engine; Flyable flyingFeature; }.

**3. Explain SOLID principles with practical Java examples.**

**Answer:**

* **S (Single Responsibility):** One class → one responsibility (InvoicePrinter vs InvoiceCalculator).
* **O (Open/Closed):** Classes open for extension, closed for modification (use strategy pattern).
* **L (Liskov Substitution):** Subtypes should replace base types without breaking (Square vs Rectangle pitfall).
* **I (Interface Segregation):** Avoid fat interfaces (split Printer into Scanner, Copier).
* **D (Dependency Inversion):** Depend on abstractions (PaymentService depends on PaymentGateway interface).

**4. What are design patterns you’ve applied, and why?**

**Answer:**

* **Creational:** Singleton (config service), Factory (object creation logic).
* **Structural:** Adapter (integrating external APIs), Decorator (logging, caching).
* **Behavioral:** Strategy (payment methods), Observer (event-driven systems), Command (undo operations).

**5. What is polymorphism, and how does Java achieve it at runtime?**

**Answer:**

* **Compile-time polymorphism:** Method overloading.
* **Runtime polymorphism:** Method overriding → achieved via **dynamic dispatch** (JVM picks method based on object’s runtime type, not reference type).

**6. How do abstract classes differ from interfaces? When would you use each?**

**Answer:**

* **Abstract class:** Can hold state + partial implementation. Single inheritance only.
* **Interface (Java 8+):** Pure abstraction, default/static methods allowed, multiple inheritance possible.
* **Use Case:**
  + Abstract class → common base behavior (e.g., HttpServlet).
  + Interface → contracts for unrelated classes (e.g., Comparable, Runnable).

**7. How do you implement immutability in OOP? Why is it important?**

**Answer:**

* Declare class final.
* Private final fields, no setters.
* Defensive copies of mutable objects.
* Example: String is immutable in Java → ensures thread safety, caching, and predictable behavior.

**8. What are common OOP pitfalls in large systems?**

**Answer:**

* **God Objects:** Too many responsibilities → violates SRP.
* **Deep inheritance chains:** Hard to maintain → use composition.
* **Overuse of design patterns:** Leads to complexity (pattern abuse).
* **Improper equals/hashCode:** Breaks collections (HashMap, Set).

**9. Explain Dependency Injection (DI) in OOP context.**

**Answer:**

* Instead of a class creating dependencies itself, dependencies are provided externally (constructor, setter, framework).
* Example with Spring:
* class OrderService {
* private final PaymentGateway gateway;
* @Autowired
* public OrderService(PaymentGateway gateway) { this.gateway = gateway; }
* }
* Improves testability, decoupling, and maintainability.

**10. How does OOP help with concurrency and thread safety?**

**Answer:**

* **Encapsulation:** Synchronize access to shared mutable state.
* **Immutability:** Eliminates need for locks (String, LocalDate).
* **Polymorphism:** Strategy pattern for concurrency policies (e.g., different ExecutorService implementations).