

JAVA

INTERVIEW

QUESTIONS & ANSWERS

**300+ interview questions and
Answers**



A K R A Y

Disclaimer

Copyright © 2025 by AK Ray

This book has been created with the sole purpose of **learning and knowledge enhancement**. The questions, examples, and explanations included are designed to help developers **improve their skills, broaden their understanding, and prepare for interviews** in a structured manner.

 However, this book does **not guarantee**:

- Job placement or employment of any kind.
- Success in interviews, assessments, or examinations.
- Any direct or indirect outcome in professional opportunities.

All content is provided “**as is**” for **educational purposes only**. While every effort has been made to ensure accuracy and clarity, the author and publisher shall not be held responsible for any errors, omissions, or outcomes resulting from the use of this material.

By using this book, you acknowledge that **continuous learning, hands-on practice, and real-world experience** are the true enablers of professional success.

© [2025] [AK Ray]. All rights reserved. No part of this publication may be reproduced or transmitted in any form without prior written permission from the author.

First Edition

About the Author

Amitesh is a highly accomplished software engineer with over 12 years of experience in core software development, specializing in product-based organizations across India and abroad. He is currently a **Lead Software Development Engineer (Lead-SDE) at M2P Fintech**.

Holding a *certification in System Design and Architecture* from the University of Alberta, Amitesh is also pursuing an *Advanced Certificate Program in Applied AI & ML* from **IIT Madras**, an intensive 11-month program.

His **technical expertise** spans **Java, Spring Boot, Microservices, AWS, Kafka, Python, AI, and Machine Learning**, making him a sought-after mentor and industry expert.

A passionate contributor to the developer community, Amitesh has **authored 15+ books** on Java, Spring Boot, and Microservices, empowering developers with in-depth knowledge. With a keen focus on technical excellence, he has conducted **over 700+ technical interviews** in the past 6 years, helping organizations identify top engineering talent.

Through his books, mentorship, and professional engagements, Amitesh remains dedicated to sharing his knowledge and expertise, shaping the future of software development.

Warm regards,

A handwritten signature in black ink, appearing to read 'Amitesh'.

A Message to My Fellow Developers

Dear Developers,

Over the past few years, I've had the privilege of conducting **700+ interviews** — and being a candidate in many as well. Through those experiences, I've seen a clear pattern: the best engineers are not just the ones who know answers, but the ones who truly understand **why** and **how** things work.

That's what inspired this book.

Here, I've curated **300+ real-world interview questions and answers** drawn directly from **hands-on experience with Java, Spring Boot, Microservices, Kafka, Security, and System Design**. These are not theoretical questions — they come from real engineering discussions, design sessions, and production challenges.

Some of you may have read my previous eBook, which contained only the questions. This edition takes it further — with **complete, structured answers**, practical explanations, and **scenario-based discussions** to help you think like an engineer, not just an interviewee.

Whether you're just starting your journey or have years of experience, my goal is to help you **build confidence, strengthen fundamentals, and approach interviews with a problem-solving mindset**.

Remember — interviews are checkpoints, not destinations. Keep learning, keep experimenting, and never stop exploring. Growth in tech is a journey, and every question you master brings you one step closer to mastery.

Keep coding. Keep growing.

Q1: I have a list of 10K employee data. I want to perform some data transformation and filtration, remove duplicates, and get data in sorted order. How do you handle it?

Answer:

When dealing with large datasets like 10K employees, the key is **efficient processing without consuming too much memory**. In Java, the **Streams API** is perfect for transformation, filtering, and sorting.

1. **Filtering:** Use `filter()` to select only the employees that match your criteria.
2. **Removing duplicates:** Use `distinct()`, which internally relies on properly implemented `equals()` and `hashCode()` in your `Employee` class.
3. **Sorting:** Use `sorted()` with a `Comparator`.

Example:

```
List<Employee> result = employees.stream()
    .filter(e -> e.getSalary() > 100_000)
    .distinct()
    .sorted(Comparator.comparing(Employee::getName))
    .collect(Collectors.toList());
```

For **performance**, you can use **parallel streams** for large lists, but make sure your transformations are thread-safe. This approach is clean, readable, and maintainable.

Q2: I have a list of Employees. How to avoid duplicates?

Answer:

Avoiding duplicates depends on **how you define “duplicate”**. Usually, it is based on a unique identifier like `employeeId`.

1. **Using Set:**

```
Set<Employee> uniqueEmployees = new HashSet<>(employeeList);
```

2. **Using Streams API:**

```
List<Employee> unique = employeeList.stream()
    .distinct()
    .collect(Collectors.toList());
```

Important: Always override `equals()` and `hashCode()` in your Employee class, ideally using a unique field like `employeeId`. This ensures Java can correctly identify duplicates.

Q3: Assume you are making a REST API call from one service to another service. How will you manage data security, and what are possible ways to make remote calls?

Answer:

When calling a remote service, **security and reliability** are critical.

1. Data Security:

- Use **HTTPS** to encrypt data in transit.
- Authenticate using **OAuth2/JWT tokens** or **API keys**.
- Use **mutual TLS** if services are internal and sensitive.

2. Remote Call Options:

- **Spring RestTemplate** (blocking)
- **WebClient** (reactive, non-blocking)
- **Feign Client** (declarative HTTP client)

3. Best Practices:

- Set **timeouts** for connection and read.
- Implement **retry policies** with exponential backoff.
- Use **circuit breaker** to prevent cascading failures (Resilience4j).

Example with WebClient:

```
WebClient.create()
    .get()
    .uri("https://service/api/employees")
    .header("Authorization", "Bearer token")
    .retrieve()
    .bodyToMono(Employee.class);
```

Q4: You are calling one service from another. What if the remote service goes down or is not responding?

Answer:

Remote service failures are common in microservices. To handle this:

1. **Circuit Breaker Pattern:** Stop calling a failing service temporarily to avoid overloading it.
2. **Fallback Methods:** Return default or cached data when the remote service fails.
3. **Timeouts and Retries:** Configure retries with exponential backoff to avoid hammering the remote service.
4. **Asynchronous Calls:** Use `CompletableFuture` or reactive streams to avoid blocking your service threads.

Example using Resilience4j:

```
@CircuitBreaker(name = "employeeService", fallbackMethod = "fallback")
public Employee getEmployee(Long id) {
    return restTemplate.getForObject("https://service/api/" + id, Employee.class);
}

public Employee fallback(Long id, Throwable t) {
    return new Employee(id, "Unknown", 0);
}
```

Q5: I am getting millions of requests per second through a REST API. How to handle it safely to avoid data loss?

Answer:

Handling high traffic requires **scalability, resilience, and safe processing**. Here's how to approach it:

1. **Load Balancing:**
 - Use a **load balancer** (Nginx, HAProxy, or cloud ALB) to distribute requests across multiple service instances.

- Horizontal scaling ensures no single instance is overwhelmed.
- 2. **Asynchronous Processing:**
 - Do not process everything synchronously. Push incoming requests to **message queues** (Kafka, RabbitMQ).
 - Consumers can process requests at their own pace, preventing overload.
- 3. **Rate Limiting & Backpressure:**
 - Use **rate limiting** to prevent abuse (Bucket4j, API Gateway).
 - Apply **backpressure** strategies in streaming data pipelines.
- 4. **Idempotent APIs:**
 - Ensure API calls can safely retry without causing duplicates (use unique request IDs).
- 5. **Monitoring & Alerts:**
 - Monitor **throughput, error rate, and queue lag** to proactively detect overload.

Example: Push API requests to Kafka:

```
kafkaTemplate.send("employee-topic", requestId, employeeData);
```

This approach **prevents data loss, scales efficiently, and ensures reliability**.

Q6: How can you enable distributed tracing in your microservice architecture?

Answer:

Distributed tracing helps **track a request across multiple microservices**, which is crucial for debugging and performance analysis.

1. **Use a Tracing Framework:**
 - **OpenTelemetry** (vendor-neutral, supports Java)
 - **Jaeger** or **Zipkin** for visualization
2. **Propagate Trace IDs:**
 - Each request gets a **unique trace ID**.
 - Pass the trace ID in HTTP headers (**X-Trace-Id**) across services.
3. **Integrate with Logging:**
 - Use **MDC (Mapped Diagnostic Context)** to include trace IDs in logs.

Example:

```
MDC.put("traceId", traceId);  
logger.info("Processing employee request");
```

4. **Visualization:**

- Collect traces in Jaeger/Zipkin to visualize request paths and latency.

Benefits: Quickly identify bottlenecks, failed calls, and performance issues across services.

Q7: Suppose you are going to design your microservice architecture. What are the common cross-cutting concerns you will implement and how? What tools will you use?

Answer:

Cross-cutting concerns are **features that affect all services**. Common ones include:

1. **Logging & Monitoring:**

- SLF4J + Logback for logging
- ELK stack (Elasticsearch, Logstash, Kibana) or Grafana + Prometheus for metrics

2. **Security:**

- OAuth2/JWT for authentication
- mTLS for service-to-service encryption

3. **Rate Limiting & Throttling:**

- Implement via API Gateway or Resilience4j

4. **Configuration Management:**

- Use Spring Cloud Config Server or Kubernetes ConfigMaps/Secrets

5. **Distributed Tracing:**

- OpenTelemetry, Jaeger, Zipkin

6. **Exception Handling:**

- Standardized error responses
- Global exception handling in Spring Boot (`@ControllerAdvice`)

Example: Centralized logging + distributed tracing ensures you can **debug and monitor all services in one place**.

Q8: Suppose you have 20+ microservices. How do you maintain runtime parameters like DB details and other global configurations? What is the best way to handle such use-cases?

Answer:

Managing runtime parameters for many services requires **centralization and dynamic configuration**:

1. **Config Server:**

- Use Spring Cloud Config Server to store configurations in Git/DB.

- Services fetch configs at startup or dynamically via refresh endpoints.
- 2. **Kubernetes ConfigMap & Secrets:**
 - Store environment-specific configurations (non-sensitive in ConfigMap, secrets in Secret).
 - Mount as environment variables or files.
- 3. **Best Practice:**
 - **Externalize all configurations** (DB URLs, API keys, thresholds).
 - Avoid hardcoding in code or service images.

Benefits:

- Changes can be applied **without redeploying services**.
 - Supports **multi-environment consistency** (dev, QA, prod).
-

Q9: What is a Config Server and ConfigMap, and why do we use them in microservices?

Answer:

In a microservice architecture, each service has **different environment-specific configurations** like DB URLs, API keys, or feature flags. Managing these individually can be messy and error-prone.

1. **Config Server:**
 - A centralized service (like **Spring Cloud Config Server**) that provides configuration to multiple services.
 - Configurations can be stored in **Git, DB, or file system**.
 - Supports **dynamic refresh** without restarting services.
2. **ConfigMap (Kubernetes):**
 - A Kubernetes object to store **non-sensitive configuration** as key-value pairs.
 - Can be mounted into pods as **environment variables or files**.

Why use them:

- Decouples configuration from code.
 - Makes updates easy across environments.
 - Reduces risk of misconfigurations in production.
-

Q10: Filter Employee data where Name starts with “A”, CITY=”Bangalore”, SALARY>100k, sort data by Employee Name in DESC order, avoid duplicate data, and in the final response collect only Name, Salary, and Designation (Employee class has more than 10 fields). How would you do it?

Answer:

This can be achieved efficiently using **Java Streams API**:

```
List<Map<String, Object>> result = employees.stream()
    .filter(e -> e.getName().startsWith("A"))
    .filter(e -> "Bangalore".equals(e.getCity()))
    .filter(e -> e.getSalary() > 100_000)
    .distinct()
    .sorted(Comparator.comparing(Employee::getName).reversed())
    .map(e -> Map.of(
        "name", e.getName(),
        "salary", e.getSalary(),
        "designation", e.getDesignation()
    ))
    .collect(Collectors.toList());
```

Key points:

- `distinct()` removes duplicates.
- `Comparator.comparing().reversed()` sorts in descending order.
- Mapping to `Map` keeps only required fields to **reduce memory footprint**.

Q11: You have a high-traffic payment processing system. Multiple threads are updating the same transaction record. How would you ensure atomicity and prevent double spending?

Answer:

Atomicity is critical in **financial systems**.

1. **Database Transactions:**
 - Use **ACID transactions** to ensure updates are atomic.
 - Choose **isolation level** carefully (e.g., Serializable or Repeatable Read).
2. **Locking:**
 - **Optimistic locking:** Add a `version` field and check before updating.
 - **Pessimistic locking:** Lock the row during processing to prevent conflicts.
3. **Distributed Transactions:**
 - If updates span multiple services, use **SAGA pattern** or **Two-Phase Commit**.

Example (Optimistic Locking):

```
@Version
private Long version;
```

This prevents two threads from updating the same transaction simultaneously.

Q12: Your application processes millions of messages from Kafka. How would you design a thread pool strategy using `ExecutorService` to avoid thread starvation and memory leaks? Do you think `ExecutorService` is best to use or is there a better way?

Answer:

Processing high-volume Kafka messages requires **controlled concurrency**:

1. **Thread Pool Design:**

- Use **bounded `ExecutorService`** to limit threads.
- Avoid unbounded queues which can cause **OOM**.
- Set proper **`corePoolSize`, `maxPoolSize`, `queueSize`**.

2. **Better Approaches:**

- **Reactive streams (Project Reactor, Akka Streams)** handle backpressure and async processing efficiently.
- Kafka consumer concurrency can also be tuned instead of spawning many threads manually.

Example:

```
ExecutorService executor = new ThreadPoolExecutor(
    10, 50, 60L, TimeUnit.SECONDS,
    new LinkedBlockingQueue<>(1000)
);
```

Q13: In production, you see frequent **OutOfMemoryError: Java Heap Space. What real-time tools and Java coding practices would you use to identify and fix memory leaks?**

Answer:

1. **Tools:**

- **VisualVM, JConsole, Java Flight Recorder (JFR), YourKit** for memory profiling.
- Heap dumps to identify objects consuming memory.

2. **Coding Practices:**

- Avoid **static collections** growing indefinitely.

- Close **resources properly** (try-with-resources).
 - Use **weak references** or caching frameworks like Caffeine with eviction policies.
 - Monitor memory usage in production with **Prometheus/Grafana**.
-

Q14: In production, your app frequently exhausts DB connections. How would you tune HikariCP for 100 active connections?

Answer:

HikariCP is a high-performance JDBC connection pool. Tune as follows:

```
spring.datasource.hikari.maximum-pool-size=100
spring.datasource.hikari.connection-timeout=30000
spring.datasource.hikari.idle-timeout=600000
spring.datasource.hikari.max-lifetime=1800000
```

Tips:

- Ensure your database can handle 100 concurrent connections.
 - Monitor **active connections**, **waiting threads**, and **connection usage**.
 - Adjust pool size based on **query latency** and **throughput**.
-

Q15: In a loan approval workflow, if one step fails, you need to rollback gracefully and log details. How would you structure your custom exceptions?

Answer:

1. **Custom Exception Hierarchy:**
 - Base exception: `LoanProcessingException`
 - Step-specific exceptions: `CreditCheckException`, `DocumentValidationException`
2. **Transactional Handling:**
 - Wrap workflow in `@Transactional` for automatic rollback.
3. **Logging:**
 - Include **step name, input data, and failure reason** in logs.

Example:

```
public class LoanProcessingException extends RuntimeException {  
    public LoanProcessingException(String message) {  
        super(message);  
    }  
}
```

This makes it easier to **debug and audit failures**.

Q16: Your Java service updates a DB and also publishes to Kafka. If DB commit succeeds but Kafka fails, how would you handle exactly-once consistency?

Answer:

Use **Outbox Pattern** or **Kafka Transactions**:

1. **Outbox Pattern:**
 - Write DB update and Kafka event in the same transaction.
 - A separate process reads the outbox table and publishes events to Kafka.
2. **Kafka Transactional Producer:**
 - Begin Kafka transaction, update DB, send Kafka message, commit both atomically.

This ensures **no message loss and exactly-once delivery**.

Q17: You see GC pauses in a low-latency trading system. How would you tune the Garbage Collector (G1/ZGC) for minimum stop-the-world pauses?

Answer:

1. **G1 GC tuning:**
 - Set `-XX:MaxGCPauseMillis=10` to minimize pause time.
 - Increase heap to reduce frequency of GC cycles.
2. **ZGC (Java 21/25):**
 - ZGC is low-latency and performs most GC concurrently.
 - Monitor **allocation rate**, reduce **short-lived objects**.
3. **Monitoring:**
 - Use **JFR or GC logs** to adjust heap size and pause goals.

Goal: **consistent low latency** with minimal stop-the-world interruptions.

Q18: What are the possible GC types in the latest Java version (Java 21 or Java 25)?

Answer:

1. **Serial GC:** Simple, single-threaded, suitable for small apps.
2. **Parallel GC:** Multi-threaded, good for throughput.
3. **G1 GC:** Balanced throughput and low pause, default in modern Java.
4. **ZGC:** Low-latency, concurrent GC for very large heaps.
5. **Shenandoah GC:** Low-pause, concurrent GC alternative to ZGC.
6. **Epsilon GC:** No-op collector for testing, does not reclaim memory.

Choose GC based on latency vs throughput requirements.

Q19: In microservices, tracing a single request across multiple services is difficult. How would you implement correlation IDs in logs? Have you ever implemented distributed logging? If yes, explain.

Answer:

1. **Correlation IDs:**
 - Generate a unique ID for each request (**UUID**).
 - Pass it via HTTP headers (**X-Correlation-Id**) to all services.
 - Include it in logs using **MDC**:

```
MDC.put("correlationId", requestId);  
logger.info("Processing request");
```

2. **Distributed Logging:**
 - Use **ELK stack** (Elasticsearch, Logstash, Kibana) or **Grafana Loki**.

- All services push logs to a centralized system.
- Search by correlation ID to trace full request flow.

Benefits: **debugging and monitoring becomes much easier** in microservices.

Q20: Do you participate in code reviews? If yes, what are the best practices for code review? How can you ensure your code is secure? How to avoid code smells?

Answer:

1. **Code Review Best Practices:**
 - Check **readability, maintainability, and correctness**.
 - Encourage **peer discussion** and knowledge sharing.
 - Review **unit tests and integration tests** coverage.
2. **Code Security:**
 - Validate input and sanitize outputs.
 - Avoid **hardcoded secrets**.
 - Follow OWASP best practices for web apps.
3. **Avoid Code Smells:**
 - Avoid **duplicated logic**, long methods, tight coupling.
 - Use **SOLID principles** to improve design.
 - Refactor continuously to keep code clean.

My FEBs[5-15 yrs]

1. Tell me about yourself (brief)

Speak-it-out short version (30–45 sec):

I'm Amitesh Kumar Ray, a Lead Software Engineer with ten years of experience building and designing Java-based systems. I specialize in backend systems, Spring Boot, microservices, and scalable architectures. I lead a team of engineers, design systems end-to-end (HLD and LLD), perform code reviews and security checks, and spend around half my time writing code and mentoring developers. I've also authored multiple technical books and run tech sessions for teams.

Longer, book-style explanation (what to include and why):

When recruiters or interviewers ask “Tell me about yourself,” they want a short story that shows who you are professionally, what you do now, and what you want next. Start with a one-line headline (role + years + domain). Then add two or three specific accomplishments that show impact (for example: “I led the migration of a monolithic payments system to microservices, which improved release velocity and reduced incidents”). Close by saying what you're looking for next—this aligns you to the role you're talking about.

Why this format works:

- It is concise and measurable.
 - It gives enough context for follow-ups.
 - It highlights leadership, technical skill, and results.
-

2. Explain your last project and your role there

Short version to speak:

I led a migration project that split a large monolithic order-processing system into domain-focused microservices (Order, Payment, Inventory, Notification). I was the technical lead: I defined the architecture (HLD/LLD), oversaw implementation and CI/CD, handled security and observability, mentored developers, and contributed significant code for critical flows like payment and data migration.

Detailed explanation (book style):

Project goal and motivation:

The monolith had become hard to change: deployments were risky, peak load caused outages, and feature delivery slowed. The goal was to split the monolith so each domain could be developed, deployed, and scaled independently, while preserving data correctness and minimizing customer impact.

Architecture choices:

- **Microservices by bounded context:** Order Service, Payment Service, Inventory Service, User Service, Notification Service. Each service owns its data (database-per-service).
- **Communication:** A mixture of synchronous REST for request/response and asynchronous messaging (Kafka) for events and decoupled workflows.
- **Authentication:** Keycloak for centralized OAuth2/OpenID Connect.
- **Observability:** OpenTelemetry for tracing, Prometheus for metrics, and ELK/EFK stack for logs.
- **Deployment:** Docker containers orchestrated with Kubernetes. CI/CD pipelines deploy to test/staging/prod with canary releases.

My responsibilities:

- **High-level design (HLD):** Defined service boundaries, contracts, data ownership, and integration patterns.
- **Low-level design (LLD):** Specified API shapes, database schemas, event topic layouts, and retry/idempotency strategies.
- **Implementation & reviews:** Wrote critical modules (e.g., payment workflow), performed code and security reviews, and ensured testing coverage.
- **Migration strategy:** Planned and executed data migration with minimal downtime using dual writes and Change Data Capture (CDC) to keep new services in sync.
- **Monitoring & incident response:** Setup dashboards and SLOs, established runbooks for incidents.

Outcome:

- Deploy time reduced from days to hours.
- Incident Mean Time To Repair (MTTR) decreased significantly.
- System scaled better under load and allowed teams to deliver features independently.

Lessons learned:

- Data migration is the hardest part; invest time in planning and safe rollout mechanisms.
- Observability from day one avoids many debugging headaches.
- Start small and split services incrementally—not all at once.

3. Explain the Architecture of your current application

Short overview:

The application uses a microservices architecture behind an API Gateway. Services are mostly Spring Boot applications, stateless, containerized, and orchestrated using Kubernetes. There's an authentication provider (Keycloak), an event bus (Kafka) for asynchronous communication, and different databases depending on the service needs (MySQL, Redis, Cassandra). Observability (metrics, tracing, logs) and CI/CD are first-class citizens.

Detailed architecture breakdown (book style):

1. Edge / API Gateway

- The gateway handles TLS termination, routing, rate limiting, authentication checks, and sometimes response aggregation. It hides internal topology from clients and provides centralized policies.

2. Authentication & Authorization

- Keycloak is used as the identity provider. User tokens (JWT) are validated by services or at the gateway. For machine-to-machine communication, we use OAuth2 client credentials and mTLS for stronger trust.

3. Microservices

- Services are domain-focused and own their data. Each service exposes REST endpoints and can publish/subscribe to Kafka topics. Services are stateless so they can scale horizontally.

4. Data stores

- **Transactional:** MySQL for strong consistency operations (orders, payments).
- **Cache / fast-access store:** Redis for session or frequently accessed data.
- **Analytics / events:** Cassandra or object storage for append-only logs and events.
- Each service manages its own schema (database-per-service) to avoid coupling.

5. Message Bus / Event Streaming

- Kafka used for publish/subscribe; it decouples services and implements eventual consistency. Topics are designed by business domain.

6. Resilience & Fault Tolerance

- Circuit Breakers, Retries with exponential backoff, and Bulkheads using Resilience4j. These patterns prevent cascading failures.

7. Observability

- **Tracing:** OpenTelemetry/Jaeger for distributed tracing.
- **Metrics:** Prometheus + Grafana with SLOs & alerts.
- **Logging:** Structured logs sent to ELK/EFK with correlation IDs.

8. CI/CD & Deployments

- Pipelines build artifacts (Docker images), run tests, scan for vulnerabilities, and promote images through environments. Deployments use canary or blue/green knobs to reduce risk.

9. Security & Secrets

- Secrets in HashiCorp Vault or Kubernetes secrets. TLS for service communication and secure storage for credentials.

Why these choices:

- They give independent scaling, faster deployments, and fault isolation. The trade-off is higher operational complexity—so automation, observability, and good practices are necessary.
-

4. Which Java version are you using? Tell me some features of that Java version

(Tailor the first line to your actual project version — below assumes Java 21 LTS. If you use another version, replace the version and pick relevant features.)

Short practical answer:

We use Java 21 (LTS). Important features we use are Virtual Threads (Project Loom), structured concurrency APIs, records and pattern matching enhancements, sealed classes, and continued GC improvements (G1, Shenandoah, ZGC options).

Detailed explanation of key features and how they help:

1. Virtual Threads (Project Loom)

- Virtual threads are lightweight threads managed by the JVM, not the OS. They make writing synchronous code that scales easier because you can have millions of virtual threads without OS thread overhead. Use-case: I/O-heavy server tasks, batch consumers. They let you write simple code (blocking style) which performs like an async system.

2. Structured Concurrency

- An API that helps group tasks and manage them together. It simplifies error handling and cleanup across concurrently running tasks.

3. Records and Pattern Matching

- Records provide a concise way to declare data-carrying classes without boilerplate. Pattern matching for instance-of and switch is cleaner and reduces error-prone casting, making code more readable.

4. Sealed Classes

- Allow controlled class hierarchies: only a fixed set of classes can extend a given sealed class. Useful for modeling domain events or DTO types where exhaustive handling is desirable.

5. Garbage Collector Improvements

- JVM continues to improve pause times and throughput across G1, Shenandoah, and ZGC. These collectors reduce GC pause times and help meet latency SLAs for services.

How to present this in an interview or book:

Explain which features you used, why you chose them, and show a small code snippet—e.g., how using records simplified DTOs, or how virtual threads simplified a patch that previously used a callback-heavy thread pool.

5. What is the latest Java version? Tell me some features from that

How to answer in interviews / in a book:

Java releases frequently. If you are not sure about the very latest release on the interview day, say what you use (e.g., Java 21 LTS) and state that newer versions continue incremental improvements (e.g., Project Loom refinements, Panama/foreign function API, serialization enhancements, language conveniences). It's perfectly fine to show you work on LTS versions in production and are aware of the direction of the platform.

Key features that often appear in recent non-LTS releases (examples):

- Continued improvements to virtual threads and structured concurrency.
- Enhancements to foreign function access (Panama) enabling safer and faster native calls.
- Performance and GC tuning improvements.
- Small language improvements: pattern matching refinements, compact number APIs, better string handling.

Advice:

If asked for the literal “latest” during an interview, clarify whether they mean the latest LTS or the most recent release. It's fine to say: “My production work targets LTS like Java 21; I follow newer releases and evaluate beneficial features before adoption.”

6. Have you ever worked on performance tuning or memory management? Describe a use-case

Short answer:

Yes. Example: I fixed production `OutOfMemoryError` (OOM) and high tail latency in an order-processing service by analyzing heap dumps, GC logs, and traces, then applied code fixes, improved serialization, and tuned JVM and GC settings.

Detailed case study (step-by-step):

1. Problem observed:

- During peak hours, service experienced OOMs and long pause times. Throughput dropped and customer operations were delayed.

2. Measurement & diagnosis:

- **Monitoring:** Prometheus showed CPU and memory patterns.
- **Tracing:** Jaeger indicated some requests were creating many objects.
- **GC logs:** `-Xlog:gc*` revealed long pause times and frequent full GCs.
- **Heap dumps:** Captured via `jmap` and analyzed with Eclipse MAT; found a large retention from in-memory buffers and unused caches.

3. Root causes found:

- Hot code path created many short-lived objects (temporary strings, wrappers).
- Inefficient JSON parsing created intermediate objects.
- A JNI-based library had a memory leak when used under special conditions.

4. Actions taken:

- **Code refactoring:** Reused buffers, switched to Jackson streaming API to avoid intermediate objects, and optimized loops to not create objects every iteration.
- **Library fix:** Replaced the JNI-based library or applied upgrade/patch.
- **GC tuning:** Chosen a GC that matched the latency profile (e.g., Shenandoah/ZGC for low pause times) and tuned heap ratios and pause targets. Adjusted `-Xms/-Xmx`, survivor spaces, and occupancy thresholds.
- **DB & pooling tuning:** Tuned HikariCP size and timeouts to avoid thread-blocking and connection exhaustion.

5. Outcome:

- OOMs stopped. p95 latency reduced dramatically (example: from ~800ms to <150ms). GC pauses decreased and the system scaled with peak load.

General principles:

- Always measure before tuning.
- Prefer code-level fixes (reduce allocation rate) over aggressive JVM flags.

- Heap dumps, GC log analysis, and profiling are your best tools.
-

7. What is GC? How does it work? Have you tuned GC in your application?

Short, high-level answer:

GC (Garbage Collection) is the JVM process that automatically reclaims memory from objects that are no longer reachable. It reduces developer burden of manual memory management. The JVM uses generations and specialized collectors to balance throughput and latency. Yes, I have tuned GC by selecting collectors and adjusting flags to meet latency and throughput goals.

Full explanation (simple language):

1. Why Garbage Collection exists:

- In Java you do not manually free memory (no `free()` or `delete()` like C). The JVM tracks object references and periodically reclaims memory for unreachable objects. This prevents many common bugs but introduces the need to manage GC behavior to meet performance goals.

2. Generational hypothesis:

- Most objects die young. JVM splits the heap into **Young** (Eden + Survivor spaces) and **Old** generations. Objects are created in Eden; survivors move to survivor spaces and eventually promoted to Old if they live long enough.

3. Types of GC events:

- **Minor GC:** Cleans the Young generation. It's frequent and usually cheap.
- **Major/Full GC:** Cleans the Old generation (and sometimes entire heap). It's heavier and can pause application threads.

4. Different collectors and when to use them:

- **Serial GC:** Simple, for single-threaded or small applications.
- **Parallel GC:** Focuses on throughput using multiple threads; it stops application threads during collection.
- **G1 GC (Garbage-First):** Region-based; balances latency and throughput and is a good general-purpose choice.
- **Shenandoah / ZGC:** Low-latency collectors with concurrent compaction designed for low pause times, suitable for latency-sensitive services.

5. Common GC tuning levers:

- **Heap sizing:** `-Xms` (initial) and `-Xmx` (max) sizes. Too small causes frequent GC; too large can cause OS swapping.
- **Choose collector:** `-XX:+UseG1GC`, `-XX:+UseZGC`, `-XX:+UseShenandoahGC`.
- **Pause targets & ergonomics:** `-XX:MaxGCPauseMillis=200` (goal setting), `-XX:InitiatingHeapOccupancyPercent` for G1, etc.
- **GC logging:** `-Xlog:gc*` to capture GC events and analyze them.

6. Tuning approach I follow:

- **Measure:** Use GC logs and tools.
 - **Fix code first:** Reduce allocation rates by reusing objects and using streaming parsers.
 - **Adjust GC:** Select appropriate collector and tune heap and related thresholds.
 - **Repeat:** Test under load and iterate.
-

8. What design patterns have you used? List some and where they fit

Short list (common patterns I use):

- **Creational:** Singleton, Factory, Builder.
- **Structural:** Adapter, Decorator, Facade.
- **Behavioral:** Strategy, Observer, Command, Iterator.
- **Enterprise / distributed:** Repository, DAO, Circuit Breaker, Retry, Saga, CQRS, Event Sourcing.

Detailed explanation (why & where to use them):

1. Factory / Abstract Factory

- Use when object creation depends on runtime choices (for example, instantiate different storage handlers depending on configuration).

2. Builder

- Use when constructing complex objects (many constructor parameters). It improves readability and avoids telescoping constructors.

3. **Singleton**

- Use sparingly — good for config loaders or resources that must be unique, but be careful with testing and global state.

4. **Adapter**

- Useful when integrating with third-party libraries that have incompatible interfaces. Adapter translates one interface to another.

5. **Decorator**

- Attach extra responsibilities dynamically (e.g., adding logging or caching around a service).

6. **Strategy**

- Useful for swapping algorithms dynamically. Example: different pricing strategies based on customer type.

7. **Observer / Event-driven**

- For decoupled communication — publish/subscribe style systems (Kafka consumers and producers are similar patterns).

8. **Repository / DAO**

- Abstract the data access logic from business logic. Easier to test and replace persistence.

9. **Circuit Breaker, Retry, Bulkhead**

- Resilience patterns for networked/distributed systems. Circuit breaker trips on failures, bulkheads limit fault domains.

10. **Saga (Choreography & Orchestration)**

- For distributed transactions across services. Saga ensures either all steps succeed or compensations run to revert partial work.

11. **CQRS & Event Sourcing**

- For systems requiring separate read and write models and strong auditability.

How to present in a book/interview:

Explain where you used a pattern, show a small code example, and discuss trade-offs. Avoid “pattern for the sake of pattern” — always show real usage.

9. What are Microservices? Why should I go for them? Benefits

Short definition:

Microservices are a way of designing applications as a suite of small, independently deployable services, each running in its own process and communicating through lightweight mechanisms (usually HTTP/REST or messaging).

Why use microservices (benefits):

1. Independent development & deployment

- Teams can work on services separately. One team's release does not force other teams to release at the same time.

2. Scalability

- You can scale only the services that need more resources. For example, scale the payment service during sale events without scaling the whole app.

3. Resilience

- Failures are often limited to a single service, so they don't necessarily bring down the entire application.

4. Technology heterogeneity

- Teams can choose the best tool for the job (language, database) for each service.

5. Clear ownership and domain separation

- Each service maps to a business capability.

When not to use microservices (trade-offs):

- They increase **operational complexity**: deployment, monitoring, distributed tracing, and more complex testing pipelines.
- **Distributed systems problems**: network latency, partial failures, data consistency issues.
- **Higher costs**: more infrastructure and orchestration overhead.

Advice:

Start as a modular monolith—strongly modularized but deployed as one unit. Once complexity and team size grow, consider carefully splitting into microservices.

10. Difference between Monolith and Microservices; trade-offs

Short comparison:

- **Monolith:** Single deployable app containing all modules.
 - Pros: Simpler to develop, test, and deploy; easy to debug locally.
 - Cons: Hard to scale parts independently; long deploy cycles as app grows.
- **Microservices:** Many small services, each deployable separately.
 - Pros: Independent deploys, scalable, team autonomy.
 - Cons: Operational overhead, distributed system complexity.

Detailed trade-offs to consider (book style):

1. **Complexity**
 - Monolith has lower operational complexity initially. Microservices require sophisticated infra: service discovery, observability, deployment automation.
2. **Scaling**
 - Monolith scales as a whole (wasteful). Microservices scale specific parts.
3. **Development speed**
 - Small teams can move fast in microservices without stepping on each other's toes. But coordination and cross-cutting concerns can slow things.
4. **Testing & debugging**
 - Monolith is easier for end-to-end testing locally. Microservices need robust integration and contract testing.
5. **Data consistency**
 - Monolith easily uses transactions across modules. Microservices need patterns like Saga or event-driven designs for distributed transactions.

When to choose what:

- Start monolith if the product is early-stage and the team is small. Move to microservices when team size, scalability needs, and delivery speed justify the overhead.

11. Have you worked on Spring Boot? Advantages over Spring

Short practical answer:

Yes. Spring Boot is an opinionated framework built on Spring that simplifies configuration and startup. It gives auto-configuration, embedded servers, starters, and production-ready features with minimal boilerplate.

Detailed explanation and why it matters:

1. **Auto-configuration**
 - Spring Boot automatically configures components (data source, web server, security) based on dependencies and properties. This saves time and reduces configuration errors.
2. **Starters**
 - “Starters” are curated dependencies (e.g., [spring-boot-starter-web](#)) that bundle what you need for common use cases.
3. **Embedded server**
 - Run a Spring Boot app as a standalone JAR (`java -jar app.jar`) with embedded Tomcat/Jetty/Undertow. No external application server needed.
4. **Actuator**
 - Production endpoints (health, metrics, environment, info) for observability right out-of-the-box.
5. **Convention over configuration**
 - Opinionated defaults let developers focus on business logic.

Spring (core) vs Spring Boot:

- Spring provides the foundation (dependency injection, AOP, etc.). Spring Boot makes it easy to create production-ready Spring applications quickly.

How we use it in microservices:

- Spring Boot services start quickly, expose standardized endpoints, integrate well with Spring Cloud components (config, discovery), and pair with Actuator for monitoring.

12. How do microservices communicate? What are the possible ways?

High-level answer:

Microservices communicate either synchronously (HTTP/REST, gRPC) or asynchronously (message brokers like Kafka, RabbitMQ). Each approach has trade-offs in latency, complexity, and coupling.

Detailed breakdown and when to use each:

1. **Synchronous communications**
 - **REST/HTTP (JSON):** Human-friendly, widely supported, good for public APIs. Simpler to implement.
 - **gRPC:** Binary protocol, faster and lower overhead, supports streaming. Best for internal, low-latency communication.
2. **Considerations:** Synchronous calls couple services at runtime and can increase latency. Use circuit breakers and timeouts.

3. **Asynchronous communications**
 - **Message brokers (Kafka, RabbitMQ):** Decouples services; one service publishes events and others consume them. Suited for high-throughput and eventual consistency patterns.
 - **Event-driven architecture:** Useful for workflows where immediate consistency isn't required.
4. **Considerations:** Asynchronous systems result in eventual consistency and need robust message schema management and error handling.
5. **Other methods**
 - **GraphQL:** Query aggregator between front-end and many backend services; good for client-specific needs.
 - **Change Data Capture (CDC):** Tools like Debezium capture DB changes and publish events. Great for decoupling legacy databases.

Best practices:

- Always design idempotent APIs for retries.
 - Implement timeouts and circuit breakers for synchronous calls.
 - Use schema evolution practices for message formats (Avro/Protobuf) and topic governance for asynchronous flows.
-

13. How do you handle system-to-system integration? How do you take care of security? Explain OAuth2 grant types

Short answer:

Use strong authentication for machines (client credentials with OAuth2), secure transport (TLS/mTLS), API gateway for policy enforcement, token validation, and least-privilege access. OAuth2 grant types relevant to system-to-system communication are Client Credentials and occasionally JWT Bearer.

Detailed explanation (best practices + OAuth2 grant types):

1. **Authentication & trust between systems**
 - **Client Credentials (OAuth2):** Best option for service-to-service authentication. A service requests an access token from the identity provider using its client id and secret, then uses the token to call downstream services.
 - **Mutual TLS (mTLS):** Provide two-way TLS so both sides authenticate each other using certificates—helps even if tokens are compromised.
2. **API Gateway & Token validation**
 - Validate tokens at the gateway to avoid expensive token introspection for every service. Services should still verify JWT signatures locally if possible.
3. **OAuth2 grant types (short descriptions & use cases):**

- **Authorization Code (with PKCE):** For web apps or mobile apps where a user logs in and gives consent. PKCE secures public clients.
- **Client Credentials:** For machine-to-machine services without a user context. Best for service integrations.
- **Resource Owner Password Credentials:** Legacy, insecure, not recommended.
- **Implicit:** Deprecated — use Authorization Code + PKCE instead.
- **Refresh Token:** Allows renewing access tokens without re-authenticating the user. Use with secure storage.

4. Other considerations:

- **Scopes & roles:** Define granular scopes and roles to limit access.
- **Token lifetimes:** Keep access tokens short-lived. Use refresh tokens with care.
- **Secrets management:** Use a secure vault and rotate credentials.
- **Auditing & logging:** Log token issuance and use for audit trails.

14. What are the possible ways to call remote APIs using REST? Have you used third-party clients?

Short answer:

Common ways to call REST APIs in Java: make direct HTTP calls using low-level clients (Apache HttpClient, OkHttp), use Spring's WebClient (non-blocking), RestTemplate (blocking, legacy), or use declarative clients like OpenFeign. For resilient calls, wrap them with libraries like Resilience4j.

Detailed explanation (tools & best practices):

1. **RestTemplate**
 - Synchronous and blocking. Easy to use but considered legacy in favor of WebClient for new apps.
2. **WebClient (Spring WebFlux)**
 - Non-blocking and reactive. Good for high-concurrency outbound call patterns.
3. **OpenFeign**
 - Declarative REST client; you define an interface and Feign generates the client. Works well with service discovery.
4. **Low-level libraries (OkHttp, Apache HttpClient)**
 - Use when you need fine-grained control of connection pooling, timeouts, and custom TLS settings.
5. **Resiliency & client-side patterns**
 - **Retries with exponential backoff, circuit breaker, bulkhead** to protect services from slow downstreams. Use Resilience4j for these patterns.
6. **Third-party SDKs**
 - When vendors provide SDKs (AWS SDK, Stripe SDK), prefer them as they are optimized and handle authentication and retries.
7. **Best practices:**
 - Configure connection pools and timeouts properly.

- Set request and connection timeouts; never use infinite timeouts.
 - Use idempotency keys when retrying POST operations that could be repeated.
 - Use structured logging for outbound requests and correlate them with trace IDs.
-

15. What is REST? How does it work?

Short definition:

REST (Representational State Transfer) is an architectural style for distributed systems that uses standard HTTP methods to access and manipulate resources identified by URIs. It emphasizes statelessness, cacheability, and a uniform interface.

Detailed explanation with practical guidance:

1. **Core principles:**
 - **Resources:** Everything you expose is a resource (an entity such as an order or a user). Each resource is identified by a URL.
 - **HTTP verbs:** Use GET (read), POST (create), PUT (update or replace), PATCH (partial update), DELETE (remove).
 - **Stateless:** Each request contains enough info to serve it. No server-side session that changes request handling.
 - **Representation:** Clients interact with resource representations (e.g., JSON).
 - **Cacheability:** Responses should indicate if they can be cached using headers (Cache-Control, ETag).
 2. **How requests flow (simple):**
 - Client sends a request to resource URI.
 - Server authenticates/authorizes, validates input, performs action, and returns a representation and HTTP status code.
 3. **Design tips for good REST APIs:**
 - Keep URLs resource-oriented: `/orders`, `/orders/{id}/items`.
 - Use consistent status codes (200 OK, 201 Created, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 409 Conflict, 500 Internal Server Error).
 - Provide meaningful error bodies.
 - Use pagination for lists, and filters for queries.
 - Document APIs with OpenAPI/Swagger.
 4. **When REST may not be ideal:**
 - When you need highly efficient, low-latency internal communication (gRPC may be better).
 - When a client needs complex queries aggregated from many services (GraphQL might help).
-

16. How do you handle API security in your application? How to do it in Spring Boot 6.x+?

Short action plan:

Use OAuth2/OpenID Connect for authentication and authorization (Keycloak or cloud providers), validate tokens at gateway and/or resource servers, enforce scopes/roles, enable TLS, perform input validation, and integrate SAST/DAST into CI.

Concrete steps and a simple Spring Security 6.x approach:**1. Use OAuth2 Resource Server**

Configure your Spring Boot service as an OAuth2 resource server that validates JWTs:

```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
        )
        .oauth2ResourceServer(oauth2 -> oauth2.jwt());
    return http.build();
}
```

- Set `spring.security.oauth2.resourceserver.jwt.jwk-set-uri` to your identity provider's JWS URI.
 - 2. Gateway responsibilities
 - Validate tokens at gateway for external traffic. Downstream services still verify tokens locally to avoid trusting gateway only.
 - 3. Scopes and roles
 - Map JWT claims to authorities and use `@PreAuthorize` to enforce method-level security.
 - 4. Best practices:
 - Use HTTPS and secure cookie practices.
 - Short-lived access tokens and limited scopes.
 - Store refresh tokens securely (avoid long-lived tokens for public clients).
 - Use secure secrets storage (Vault/K8s secrets) and rotate keys.
 - 5. Additional protections in Spring Boot:
 - Use input validation with `@Valid`, DTOs, and avoid binding raw request structures directly to entities.
 - Add rate limiting and throttling for malicious clients.
 - Use security headers (CSP, X-Frame-Options) and enable CSRF protection for browser-based flows when needed.
-

17. How do you handle exception handling in your application? Best practices & end-to-end flow

Short answer:

Use a consistent error model, centralize mapping of exceptions to HTTP responses (e.g., `@ControllerAdvice` in Spring), include correlation IDs in errors and logs, and avoid leaking internal details to clients.

Detailed end-to-end flow and best practices:

1. **Domain exceptions:**
 - Throw meaningful, domain-specific exceptions in service layers (e.g., `OrderNotFoundException`, `PaymentFailedException`). Do not swallow exceptions silently.
2. **Controller advice / global handler:**
 - Use `@ControllerAdvice` to map exceptions to common error DTOs. This ensures clients receive consistent error formats and HTTP status codes.
3. **Error response model:**
 - Provide a structured error object with fields like:
 - `timestamp`
 - `traceId` (correlation id)
 - `status` (HTTP code)
 - `errorCode` (business-specific code)
 - `message` (friendly message)
 - `details` (optional, internal code or link to docs)

Example:

```
{
  "timestamp": "2025-10-26T10:00:00Z",
  "traceId": "abcd-1234",
  "status": 404,
  "errorCode": "ORDER_NOT_FOUND",
  "message": "Order 12345 not found"
}
```

4. **Correlation/Tracing:**
 - Generate and propagate a correlation ID for every request across services (X-Request-Id header). Include it in logs and error responses so debugging is easier.
5. **Logging & monitoring:**
 - Log exception stack traces and contextual data on server side with trace id. Create metrics for error types and set alerts for spikes.
6. **Graceful degradation:**
 - For partial failures, return helpful fallback responses when possible (e.g., “partial data available”) and mark clearly.
7. **Security:**

- Never return sensitive internal information (stack traces, SQL, PII) in client-visible error messages.
 - 8. **Testing exceptions:**
 - Write integration and contract tests for error scenarios. Ensure client-side logic can distinguish between retryable and non-retryable errors.
-

18. Have you worked on SQL tuning? Best practices

Short answer:

Yes. SQL tuning involves analyzing execution plans, adding appropriate indexes, avoiding inefficient joins, and tuning database configuration. ORM usage must be careful to avoid N+1 problems.

Detailed guidance and best practices:

1. **Analyze query plans:**
 - Use **EXPLAIN** or **EXPLAIN ANALYZE** to find full table scans, slow joins, or expensive sorts. Query plans show what the DB is doing.
2. **Indexing:**
 - Add indexes on columns used in WHERE, JOIN, ORDER BY. Use composite indexes where appropriate and in the right column order. Avoid over-indexing as it slows writes.
3. **Avoid N+1 queries:**
 - In ORM frameworks (JPA/Hibernate), use **JOIN FETCH** or **EntityGraph** to fetch related entities in a single query when needed.
4. **Use pagination and cursors:**
 - Use keyset pagination (a.k.a. cursor) for large data sets instead of OFFSET which becomes slow on big tables.
5. **Denormalization & materialized views:**
 - For read-heavy systems, maintain denormalized tables or materialized views to avoid repeated expensive joins.
6. **Connection pooling & DB tuning:**
 - Tune HikariCP pool sizes based on DB capacity. Monitor wait times and connection usage.
7. **Partitioning & archiving:**
 - Partition tables by date or logical key for very large tables and archive old data.
8. **Monitoring:**
 - Enable slow query logs and monitor DB metrics (locks, IO, CPU, connections).
9. **Query rewriting:**
 - Sometimes rewriting queries or adding hints can produce better execution plans.

Example fix:

Replace repeated selects inside a loop (N+1) with a single join or batched select — often yields huge performance wins.

19. Have you handled security in your application? Best approach to avoid security bugs

Short answer:

Yes. Adopt a secure development lifecycle including threat modeling, secure coding practices, SAST/DAST scans, dependency scanning, secrets management, least privilege, and runtime protections.

Detailed checklist and steps:

1. **Design & threat modeling:**
 - At the design phase, identify attack surfaces and data sensitivity. Decide which data must be encrypted and which operations need stronger authorization.
2. **Secure coding standards:**
 - Input validation, output encoding, and avoiding string concatenation for SQL. Use parameterized queries and prepared statements.
3. **Dependency and vulnerability management:**
 - Use tools like Dependabot/Snyk to scan dependencies for known vulnerabilities and patch promptly.
4. **Static and dynamic analysis:**
 - Integrate SAST in CI to catch code smells and common security mistakes. Run DAST to find runtime problems.
5. **Authentication & authorization:**
 - Implement OAuth2/OIDC, strong session management, least privilege for service accounts, and RBAC.
6. **Encryption & secrets:**
 - Use TLS everywhere. Store secrets in vaults and rotate them. Do not hardcode credentials.
7. **Runtime protections:**
 - Rate limiting, WAF, mTLS, and network segmentation. Monitor logs for anomalies and set alerts.
8. **Penetration testing & audits:**
 - Regular pen-tests and audits to find issues that automated tools miss.
9. **Education & review culture:**
 - Security checklists in code review and developer training help prevent common errors.

Result:

Security should be part of the development lifecycle, not an afterthought. Integrate checks into CI/CD to catch problems early.

20. Do you participate in code review? Best practices you follow

Short answer:

Yes. Good code reviews are respectful, focused on design and correctness, and balance code quality with delivery speed.

Best practices:

1. **Keep pull requests small**
 - Smaller changes are reviewed faster and more thoroughly.
2. **Automate checks first**
 - Run linters, unit tests, static analysis, and formatting in CI so reviewers focus on logic, design, and correctness.
3. **Focus on important things**
 - Security, correctness, and maintainability first; nitpicks and style issues can often be auto-fixed.
4. **Be constructive and specific**
 - Explain *why* something should change, not just that it should. Offer alternatives.
5. **Ask clarifying questions**
 - If intent is unclear, ask rather than guess. Author explains tricky decisions.
6. **Respect time**
 - Review promptly and set SLAs for review times to keep work flowing.
7. **Use templates & checklists**
 - PR templates help ensure essential information is included (what changed, why, tests, rollout plan).
8. **Ensure tests & documentation**
 - Require tests for behavioral changes and update docs or API contracts.

Cultural tips:

- Encourage shared code ownership. Keep reviews educational and avoid gatekeeping.
-

21. Have you worked on CI/CD? Best practices

Short answer:

Yes. CI/CD pipelines automate build, test, and deploy steps. Best practices include pipeline-as-code, fast feedback loops, quality gates, artifact immutability, and safe deployment strategies like canary or blue/green.

Detailed best-practice list:

1. **Pipeline as code**
 - Store pipeline definitions with the repo so changes are versioned and reviewable.
2. **Fail fast and early**
 - Run unit tests and static checks early. Fast feedback prevents wasted effort.
3. **Keep builds deterministic**
 - Use artifact registries and lock dependencies for reproducible builds.
4. **Quality gates**
 - Block merges on failing tests, security issues, or low coverage.
5. **Artifact immutability**
 - Build once and deploy the same artifact across environments.
6. **Progressive deployment**
 - Use canary, blue/green, or feature flags to reduce risk during deployment.
7. **Automated rollbacks and monitoring**
 - Rollback on critical metrics or failed smoke tests.
8. **Security in pipeline**
 - Do not expose secrets in logs. Use encrypted secrets and rotate credentials.
9. **Parallelize and cache**
 - Make pipelines fast by parallel steps and caching dependencies.
10. **Testing strategy**
 - Unit tests, integration tests, contract tests, and end-to-end tests. Automate as many as possible.

Tools and practices:

- GitHub Actions, GitLab CI, Jenkins, CircleCI for CI.
 - ArgoCD, Flux for GitOps and deployments.
 - Use vulnerability and license scanning during pipeline.
-

22. What is the current Java version? Tell me some features

The latest Java release as of 2025 is **Java SE 25 (JDK 25)**, officially released in **September 2025**. It is a **Long-Term Support (LTS)** version, offering improved performance, language features, and developer productivity.

Key Features of Java 25:

- 1. Primitive Types in Patterns and Switch Statements**
 - Enables pattern matching directly with primitive types.
 - Simplifies code and improves type safety when working with both objects and primitives.
 - 2. Compact Object Headers (JEP 469)**
 - Reduces memory overhead by using smaller object headers.
 - Results in better memory efficiency and faster execution in large-scale applications.
 - 3. Structured Concurrency (JEP 482)**
 - Introduces a new API to manage multiple concurrent tasks as a single unit.
 - Makes multithreaded programming simpler and safer.
 - 4. Vector API Enhancements (JEP 459)**
 - Improves support for vectorized computations, enhancing performance in numerical and AI-related workloads.
 - 5. Scoped Values (JEP 464)**
 - Provides a safer alternative to thread-local variables for sharing immutable data within threads.
 - Improves reliability in concurrent programming.
 - 6. JFR (Java Flight Recorder) CPU-Time Profiling on Linux**
 - Adds precise CPU-time profiling, helping developers diagnose and optimize performance bottlenecks.
-

Summary:

Java 25 continues the modernization of the platform with focus on **performance**, **developer productivity**, and **safe concurrency**. Being an LTS release, it's ideal for enterprise adoption and long-term projects.

Please get the simplified COPY for JAVA 25 :

https://topmate.io/amitesh_kumar_ray/1751888?utm_source=linkedin&utm_campaign=ss&utm_medium=product

23. Best way to handle transactions in a distributed system

Short answer:

Prefer Saga patterns (choreography or orchestration) and eventual consistency over distributed two-phase commit (2PC) in most microservice systems. Use idempotency and the outbox pattern for reliable event publishing.

Detailed explanation & patterns:

1. **Why 2PC is often avoided**
 - Two-phase commit locks resources across services, doesn't scale well, and can become a bottleneck.
2. **Saga pattern**
 - Break a global transaction into a series of local transactions. Each step has a corresponding compensating action if a future step fails.
3. **Two styles:**
 - **Choreography:** Services emit events and other services react. Simpler but can be harder to monitor.
 - **Orchestration:** A central orchestrator coordinates the steps and handles failures; easier to observe but centralizes logic.
4. **Outbox pattern**
 - Ensure event publishing is atomic with the local transaction by writing events to an outbox table in the same DB transaction, then a background process publishes those events to the message bus. This avoids inconsistencies between DB and event bus.
5. **Idempotency & retries**
 - Use idempotency keys for requests so retries don't cause duplicate side effects.
6. **Monitoring & manual reconciliation**
 - Build tools to find and fix long-running or failed sagas; provide dashboards and dead-letter handling.

When to consider 2PC:

- Only for special cases where absolute synchronous atomicity across systems is required and infrastructure supports it. Mostly, prefer Sagas and compensate.
-

24. What are non-functional requirements (NFRs)? Explain with real-time use-cases

Short definition:

NFRs describe how the system behaves — its qualities like performance, availability, security, scalability, maintainability, and compliance. They are as important as functional requirements.

Detailed list with real examples:

1. **Performance / Latency**
 - Example: Payment authorization must respond within 200ms (p95). This drives choices like in-memory caches, low-latency GC, and fewer network hops.
2. **Scalability**
 - Example: Platform must handle 10x seasonal traffic. This leads to autoscaling strategies and stateless services.
3. **Availability**
 - Example: SLAs requiring 99.99% uptime meaning multi-region deployments and failover plans.
4. **Security**
 - Example: GDPR compliance requires encryption at rest and strict access controls. This affects data storage and retention design.
5. **Maintainability**
 - Example: Objective to onboard new engineers in 2 weeks leads to clean code, standards, tests, and good documentation.
6. **Recoverability**
 - Example: Required RTO (Recovery Time Objective) of 1 hour and RPO (Recovery Point Objective) of 15 minutes. This necessitates backups, replication, and automation.
7. **Observability**
 - Example: Must provide traces and logs to explain latency spikes; choose OpenTelemetry and centralized logging.
8. **Testability & Deployability**
 - Example: Automate smoke tests and canary analysis for safe releases.

How to capture NFRs:

- Make them explicit in requirements docs, and translate them into measurable SLOs (Service Level Objectives) and tests.

Why they are critical:

NFRs influence architecture and tech choices more than functional requirements. Ignoring them leads to systems that work but fail under real conditions.

25. Where do you want to see yourself after 5 years? Execution plans

Short, clear answer to give in interviews:

In five years, I see myself as a senior architect or engineering leader who shapes large-scale systems and mentors other engineers. I want to be responsible for strategic architectural decisions and help engineer teams build resilient, high-quality systems.

Concrete plan (how you'll get there):

1. **Technical depth and breadth**
 - Continue building expertise in distributed systems, performance engineering, and cloud-native patterns. Learn advanced topics like event-driven data platforms, stream processing, or large-scale storage systems.
2. **Leadership and communication**
 - Lead larger cross-team initiatives, deliver architecture reviews, and improve stakeholder communication. Mentor more engineers and give technical talks.
3. **Delivery & impact**
 - Lead at least two major migrations or platform projects that materially improve company metrics (throughput, cost, MTTR).
4. **Education and public presence**
 - Take courses in software architecture or leadership. Publish blog articles or a book chapter to share lessons learned.
5. **Measure progress**
 - Track measurable outcomes: reduced deployment time, better SLO attainment, lower cost per transaction, and successful mentorship outcomes.

How to present this:

Be honest but ambitious. Show that you have a roadmap with measurable milestones and that the role you're applying for aligns with those goals.

Java Latest [just Good to HAVE]

1 What are Virtual Threads and how do they improve concurrency compared to Platform Threads?

In traditional Java, every thread created by the JVM is backed by an *operating system thread*. These are known as **platform threads**. Each platform thread requires its own memory stack (typically 1 MB or

more) and involves an expensive context switch whenever multiple threads share CPU time. As a result, if your application tries to create tens of thousands of concurrent threads (for example, to handle many incoming requests), you'll soon hit memory and scheduling limits.

To solve this, Java introduced **Virtual Threads** under *Project Loom* — first previewed in Java 19 and officially stabilized in **Java 21**.

A **Virtual Thread** is a lightweight thread managed by the **JVM**, not by the operating system. It still uses the familiar `java.lang.Thread` API, so developers don't have to change their code drastically.

* How Virtual Threads Improve Concurrency

- **Lightweight:**
You can create millions of virtual threads because they consume very little memory. Unlike platform threads, they don't each have a dedicated OS stack.
- **No Thread Pool Needed:**
You don't have to manage complex thread pools anymore. Each task can run in its own virtual thread without worrying about performance overhead.
- **Non-blocking I/O Integration:**
When a virtual thread waits on I/O (for example, a database or network call), it simply *yields* — freeing up the underlying platform thread to do other work. The virtual thread is resumed automatically once data is ready.

Example:

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {  
    IntStream.range(0, 10_000).forEach(i ->  
        executor.submit(() -> {  
            Thread.sleep(1000);  
            System.out.println("Task " + i);  
        }))  
    );  
}
```

Here, 10,000 tasks run concurrently with minimal overhead — something impossible with traditional threads.

In short: Virtual threads bring *scalability of reactive programming* with *simplicity of blocking code*. They make concurrency natural and efficient again.

2 Explain Record Patterns in Java 21 and how they simplify destructuring.

In Java 21, **Record Patterns** make it easier to **extract data** from record objects in a clean, readable way — this is called *destructuring*.

Earlier, when you had a record like:

```
record Employee(String name, int age, String department) {}
```

To get its fields, you had to manually call getters:

```
if (obj instanceof Employee emp) {  
    String name = emp.name();  
    int age = emp.age();  
}
```

Now, with **Record Patterns**, you can destructure directly inside the `instanceof` or `switch` pattern:

```
if (obj instanceof Employee(String name, int age, String department)) {  
    System.out.println(name + " works in " + department);  
}
```

* Benefits:

- **Cleaner pattern matching:** No need to explicitly call methods.
- **More readable code:** Especially useful in switch expressions.
- **Safer destructuring:** Compile-time type checking ensures correctness.

This feature fits beautifully with Java's ongoing evolution toward pattern matching and declarative style programming.

3 What are String Templates? How can multiline Strings help you simplify your code?

String Templates (introduced as a preview in Java 21 and evolving in Java 25) make string interpolation easier and safer. They replace the messy `String.format()` or concatenation with a more elegant syntax.

* Traditional approach:

```
String name = "Amitesh";  
String message = "Hello, " + name + "! Welcome to Java.";
```

✓ With String Templates:

```
String name = "Amitesh";  
String message = STR."Hello, \{name}! Welcome to Java.";
```

Here, the **STR** processor evaluates the embedded expressions and builds the final string efficiently and safely.

* Multiline Strings (Text Blocks):

Introduced in Java 15, **text blocks** simplify writing long or multiline strings (like JSON, SQL, or HTML).

Example:

```
String json = ""  
{  
    "name": "Amitesh",  
    "language": "Java"  
}  
"";
```

No more escaping quotes or `\n`!

Combined with string templates, text blocks make constructing structured data (like SQL or HTML) **clean, readable, and less error-prone**.

4 In Java 21, how does a record differ from a standard class in terms of boilerplate and immutability?

A **record** in Java is a *special kind of class* designed to hold immutable data. It's part of Java's move toward data-oriented programming.

Standard Class Example:

```
public class Employee {  
    private final String name;
```

```

private final int age;

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

public String name() { return name; }
public int age() { return age; }
}

```

✓ Record Equivalent:

```
public record Employee(String name, int age) {}
```

The compiler automatically generates:

- `private final` fields
- Constructor
- Accessors (`name()`, `age()`)
- `toString()`, `equals()`, `hashCode()`

* Key Differences:

Aspect	Standard Class	Record
Boilerplate	Manual	Auto-generated
Immutability	Optional	Implicit
Inheritance	Allowed	Cannot extend another class
Purpose	Behavior-centric	Data-centric

Records are ideal for DTOs, configuration objects, and immutable models — reducing 70–80% of your boilerplate code.

5 What are Sequenced Collections in Java 21, and how do they differ from regular collections?

Before Java 21, interfaces like `List`, `Set`, and `Map` had different ways to handle ordering. There was no uniform way to access elements by *position*.

Sequenced Collections were introduced in **Java 21** to fix this inconsistency.

They introduce new interfaces:

- `SequencedCollection`
- `SequencedSet`
- `SequencedMap`

* What's new?

They define a **consistent set of methods**:

`first()`, `last()`, `reversed()`, `addFirst()`, `addLast()`, `removeFirst()`, `removeLast()`

Example:

```
SequencedCollection<String> names = new ArrayList<>(List.of("A", "B", "C"));
names.addFirst("Z");
System.out.println(names.first()); // Z
```

* Why it matters:

- Brings **consistency** across all collections.
 - Simplifies code that manipulates order (no more custom logic for `LinkedList` or `LinkedHashMap`).
 - Improves developer experience when working with both lists and sets.
-

6 What are the Security Enhancements in Java 25?

Java 25 (released September 2025) continues improving the JVM's **security posture**, especially for enterprise and cloud environments.

* Key Enhancements:

1. **Enhanced Key and Certificate APIs:**
Stronger cryptography, extended support for modern elliptic-curve algorithms, and improved keystore handling.
2. **Improved TLS 1.3 Configuration:**
Easier configuration for mutual TLS and stricter defaults to prevent insecure cipher fallback.
3. **Stronger Sandbox & Module Boundaries:**
Security Manager deprecation now replaced by improved modular access controls at compile time.

4. **Better Secure Randomness:**
Faster, more reliable random number generation suitable for cryptographic operations.
5. **Improved JAR Signing:**
Updated signature algorithms and timestamp verification for code authenticity.

Overall, Java 25 emphasizes **secure-by-default configuration** — developers get safer cryptography, network communication, and code loading without extra effort.

How is Memory and GC Improved in Java 25?

Java 25 brings major updates to **Garbage Collection (GC)** and memory efficiency. The focus is *reduced pause times*, *faster allocations*, and *better generational handling*.

Key GC Enhancements:

1. **Generational ZGC:**
ZGC is now generational, meaning it separates short-lived and long-lived objects — improving efficiency dramatically.
2. **Generational Shenandoah:**
Similar improvement for Shenandoah GC — generational segregation reduces write barriers and speeds up reclamation.
3. **Improved Memory Footprint:**
Better heap compaction and more efficient metadata storage.
4. **Faster Warm-up:**
New JIT and runtime optimizations mean large applications reach optimal performance more quickly after startup.
5. **Simplified GC Tuning:**
Java 25 introduces better default heuristics, meaning less manual tuning is needed in production.



Example:

```
java -XX:+UseZGC -XX:+ZGenerational
```

This enables the new generational ZGC, combining ultra-low pause times with memory efficiency — ideal for high-load microservices.

8 What are some common developer-specific features in Java 25?

Java 25 introduces many *developer-centric* improvements — making code more expressive and easier to maintain.

* Notable Developer Features:

1. **String Templates (Standardized):**
No need for concatenation or formatting APIs — just embed variables directly.
2. **Unnamed Variables & Patterns:**
Use `_` for unused variables in patterns or lambda expressions — cleaner, clearer code.
3. **Enhanced `switch` pattern matching:**
More exhaustive and type-safe pattern handling.
4. **Improved Scoped Values:**
ThreadLocal replacement that's lightweight and safe for structured concurrency.
5. **Refined Virtual Thread APIs:**
Even better integration with Executor services and debugging tools.
6. **Improved Profiling APIs:**
Developers get deeper runtime metrics and heap analysis tools.

These features focus on **developer productivity, readability, and safety** — reflecting Java's continuing evolution toward clarity and simplicity.

9 If you are using Spring Boot, you can use Lombok — but in Java 17 we have Records. How do they differ?

Both **Lombok** and **Records** aim to reduce boilerplate code, but they achieve this differently.

Feature	Lombok	Record
Type	Library (annotation-based)	Language feature
Mutability	Can be mutable	Always immutable
Requires dependency	Yes (<code>@Data</code> , <code>@Getter</code> , etc.)	No
Works on older Java	Yes (Java 8+)	Java 14+
Adds methods	Yes (e.g., setters, builders)	Only accessors

* Example:

```
// Lombok
@Data
class Employee {
    private String name;
    private int age;
}

// Record
public record Employee(String name, int age) {}
```

If your data is **immutable** and you're using **Java 17 or above**, prefer **Records** — they're safer, lighter, and don't require extra libraries. Lombok still helps when you need **mutable objects** or builders.

10 How do Sealed Classes improve control over inheritance?

Sealed Classes let you explicitly declare **which other classes can extend or implement** them.

Before sealed classes, any class could extend another if it wasn't **final**. That often led to uncontrolled inheritance hierarchies and potential misuse.

Example:

```
public sealed class Shape permits Circle, Rectangle {}

final class Circle extends Shape {}
```

```
final class Rectangle extends Shape {}
```

Here, only `Circle` and `Rectangle` can extend `Shape`. No other class can.

* Why it matters:

- **Tighter control:** Prevents unwanted subclassing.
- **Exhaustive pattern matching:** The compiler knows all subclasses, enabling safer `switch` handling.
- **Improved security:** Prevents extension from untrusted modules.

In short: Sealed classes combine the flexibility of inheritance with the safety of `final` classes — giving you clear, maintainable hierarchies.

Java

1) What is cloning and why do we go for it?

Answer

What is cloning?

Cloning is creating a copy of an existing object so the new object has the same state (field values). In Java, cloning is commonly associated with the `clone()` method from `java.lang.Object`. A class that supports the `clone()` method usually implements the `Cloneable` marker interface and overrides `clone()` to make a field-by-field copy.

Kinds of cloning

- **Shallow copy:** copies primitive fields and references to objects — referenced objects are *shared* between original and clone. Default `Object.clone()` performs a shallow copy.
- **Deep copy:** creates copies of the mutable objects referred to by the original object so the clone has independent copies — typically this requires custom code.

Why clone? When is cloning useful?

- **Performance:** when constructing a new object from scratch is costly, a clone can be cheaper (but measure; cloning may not always be faster).
- **Prototype pattern:** create new objects by copying a prototypical instance.
- **Snapshot/undo:** quickly take a snapshot of an object's state for rollback.
- **Cache pre-populated objects:** prepare a ready-to-go object and clone it when a new instance is required.

Pitfalls & caveats

- `Cloneable` is a marker interface without a contract method — `Object.clone()` throws `CloneNotSupportedException` if a class doesn't implement it.
- Default `clone()` is shallow — you often need to override to implement deep copy.
- Cloning can be fragile with complex object graphs (circular references, final fields).
- Alternatives often preferred: copy constructors, static factory (e.g., `MyType.copyOf(other)`), or serialization/JSON mapping for deep copy.

Example (shallow override & deep copy hint)

```
public class Person implements Cloneable {
    private String name;
    private Address address; // mutable
    public Person(String name, Address address) {
        this.name = name; this.address = address;
    }

    @Override
    public Person clone() {
        try {
            Person p = (Person) super.clone(); // shallow copy
            p.address = address.clone();      // deep copy of mutable field
            return p;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError(e);
        }
    }
}
```

Mentor tip: prefer explicit copy constructors or factory methods for clarity and safer deep copy semantics unless you have a strong reason to use `clone()`.

2) What is serialization, why do we use it? What is the use of `serialVersionUID`?

Answer

What is serialization?

Serialization converts an object into a sequence of bytes (a byte stream) so it can be persisted (to disk, database) or transmitted (over a network). Deserialization reverses the process reconstructing the object from the byte stream.

Standard Java mechanism:

`java.io.Serializable` marks a class as serializable. Java's built-in object serialization (via `ObjectOutputStream` / `ObjectInputStream`) encodes the object graph, types, fields, and references.

Why use serialization?

- **Persistence:** store object state to a file or DB (quick prototyping).
- **Distributed systems:** send objects between JVMs (RMI historically used Java serialization).
- **Caching / session replication:** stores object state to share across nodes.
- **Message passing** in simple systems (but modern systems favor JSON/Protobuf/Avro).

serialVersionUID — what and why

- `serialVersionUID` is a `private static final long` which serves as a version identifier for a serializable class.
- During deserialization, JVM compares the `serialVersionUID` in the stream with the one in the current class. If they differ, `InvalidClassException` is thrown.
- If you don't explicitly declare it, the compiler generates one algorithmically based on class structure — that generated value changes if you change the class, which can cause deserialization to fail even for compatible changes.
- **Best practice:** declare `private static final long serialVersionUID = <number>L;` explicitly.

Example

```
public class Employee implements Serializable {  
    private static final long serialVersionUID = 1L;  
    private String name;  
    private int id;  
}
```

Advanced notes

- `transient` fields are not serialized (good for sensitive data or derived fields).
- `Externalizable` gives full control over serialization logic (implement `writeExternal` and `readExternal`).
- For long-term or cross-language compatibility, use safer formats (JSON, Protobuf) instead of Java serialization.
- Be aware of **security** issues — native Java serialization has been a source of remote code execution vulnerabilities in the wild; many systems avoid it entirely.

Mentor tip: reserve Java serialization for internal, controlled contexts. For cross-platform or external-facing APIs, prefer explicit schema-based formats (Protobuf/Avro/JSON).

3) Can I keep an **Employee** object or any custom object as **HashMap** key?

Answer

Yes — but with conditions. Java **HashMap** depends on **hashCode()** and **equals()** of keys:

- **hashCode()** determines which bucket the key belongs to.
- **equals()** is used to find the key among entries in that bucket.

Requirements to use custom object as key

1. **Override **equals()** and **hashCode()** consistently** — if two keys are logically equal, they must produce the same hash code.
2. **Prefer immutability** for fields used in **equals/hashCode** (see next question).
3. If the key is mutable and its fields (used for equality) change after insertion, map lookup will break.

Example

```
public class Employee {
    private final int id;
    private final String name;

    public Employee(int id, String name) { this.id = id; this.name = name; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Employee e = (Employee) o;
        return id == e.id;
    }

    @Override
    public int hashCode() { return Integer.hashCode(id); }
}
```

Pitfall: Using mutable fields (like **salary**) in equality will make the key unstable. Avoid that.

Mentor tip: use immutable key objects (or primitive/wrapper keys like **Integer**, **String**) unless you have a good reason otherwise.

4) Why do we keep hashmap keys as immutable?

Answer

Because a **HashMap** relies on the key's **hashCode()** and **equals()** values remaining stable while the key is in the map. If you mutate a key's state (the parts used to compute **hashCode/equals**) after insertion:

- The key may hash to a different bucket than originally inserted.
- The map will not find the entry by lookup (**map.get(key)**), leading to lost entries or leakage.
- Removing such keys becomes impossible because **remove** uses **hashCode** and **equals** to find the entry.

Illustrative failure

```
Map<Person, String> map = new HashMap<>();
Person p = new Person("Alice", 30);
map.put(p, "value");
p.setName("Bob"); // if name used in hashCode/equals, map corruption occurs
map.get(p); // returns null
```

Therefore: prefer immutable keys or ensure fields used in equality are not changed while the key is stored in hash structures.

Mentor tip: if you must use a mutable object, you can use a stable surrogate key (id) as the map key and keep the mutable object as value.

5) How do we achieve immutability in Java? How to write your own immutable class?

Answer

Immutability means after construction, the object's state cannot change. Immutable objects are safe to share among threads and simpler to reason about.

Step-by-step recipe to create an immutable class

1. Declare the class **final** (optional but prevents subclassing which could break immutability).
2. Make all fields **private final**.
3. Initialize all fields in the constructor.
4. Do not provide setters.
5. If a field is mutable (e.g., **Date**, arrays, collections), perform deep defensive copies in the constructor and when returning them (getters).
6. Ensure **this** reference is not leaked during construction (don't call overridable methods from constructor).

Example (correct immutable class)

```
public final class Person {
    private final String name;
    private final int age;
    private final List<String> tags;

    public Person(String name, int age, List<String> tags) {
        this.name = name;
        this.age = age;
        // defensive copy and make unmodifiable
        this.tags = Collections.unmodifiableList(new ArrayList<>(tags));
    }

    public String getName() { return name; }
    public int getAge() { return age; }
    public List<String> getTags() { return tags; } // safe: unmodifiable list
}
```

Deep immutability vs shallow

- *Shallow immutability* — object reference fields are final but refer to mutable objects.
- *Deep immutability* — all reachable state is immutable (harder to achieve).

Advantages

- Thread-safety by design.
- Simpler equals/hashCode semantics.
- Safer as `HashMap` keys / cache keys.

Mentor tip: Prefer immutable value objects (DTOs, keys). Use `record` (Java 16+) for simple immutable data carriers — they give you boilerplate-free immutability.

6) Why do we use generics with collection objects?

Answer

Generics add **compile-time type checking** and eliminate many runtime cast problems. Before generics (pre-Java 5), collections stored `Object` and required explicit casting — error-prone and unsafe.

Benefits

- **Type Safety:** prevents insertion of wrong types into collections (compile-time error).
- **No explicit casting on retrieval:** cleaner and safer code.
- **Self-documenting APIs:** `List<String>` says what the list contains.
- **Reusable code:** collections and utility methods can operate on different types.

Example

```
List<String> names = new ArrayList<>();
names.add("Alice");
// names.add(1); // compile-time error
String first = names.get(0); // no cast needed
```

Generics advanced concepts

- **Type erasure:** generics are implemented with erasure; runtime does not retain generic type parameters.
- **Wildcards (?):** *? extends T* (producer), *? super T* (consumer) — PECS rule: *Producer Extends, Consumer Super*.
- **Bounded type parameters:** e.g., *<T extends Number>* to restrict allowable types.

```
import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // names.add(123); // ❌ Compile-time error: only Strings allowed

        System.out.println("List of Names:");
        for (String name : names) {
            System.out.println(name);
        }

        String first = names.get(0); // ✅ No cast needed
        System.out.println("First name: " + first);
    }
}
```

Output

```
List of Names:
Alice
Bob
Charlie
First name: Alice
```

Mentor tip: prefer generics everywhere for collections and utility APIs; be mindful of wildcard variance when designing APIs.

```
import java.util.*;

// Generic class example
class Box<T> {
    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}

// Generic method example
class Util {
    public static <T> void printList(List<T> list) {
        for (T item : list) {
            System.out.println(item);
        }
    }
}

public class GenericsExample {
    public static void main(String[] args) {
        // Example 1: Using generics with a collection
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        // names.add(1); // ❌ Compile-time error — type safety enforced
        String first = names.get(0); // ✅ No casting needed
        System.out.println("First name: " + first);

        // Example 2: Using a generic class
        Box<Integer> intBox = new Box<>();
        intBox.set(100);
        System.out.println("Box contains: " + intBox.get());

        Box<String> strBox = new Box<>();
        strBox.set("Hello Generics");
        System.out.println("Box contains: " + strBox.get());

        // Example 3: Using a generic method
        List<Integer> numbers = Arrays.asList(10, 20, 30);
        System.out.println("Printing names:");
        Util.printList(names);
    }
}
```

```
        System.out.println("Printing numbers:");
        Util.printList(numbers);
    }
}
```

Explanation:

1. **Generic Collection:**
 - `List<String>` ensures only `String` objects can be added.
 - Eliminates the need for casting when retrieving elements.
2. **Generic Class:**
 - `Box<T>` can hold any type (`String`, `Integer`, etc.) without rewriting the class.
3. **Generic Method:**
 - `<T>` before the return type defines a method that can work with any type.

```
First name: Alice
Box contains: 100
Box contains: Hello Generics
Printing names:
Alice
Bob
Printing numbers:
10
20
30
```

7) Can you override the static method?

Answer

No — you cannot override static methods. Static methods are associated with the class, not the instance, so method overriding (which is runtime polymorphism) does not apply. You can *hide* a static method in a subclass by declaring a static method with the same signature; calls are resolved at compile time based on the reference type.

Example

```
class A { static void hello() { System.out.println("A"); } }
class B extends A { static void hello() { System.out.println("B"); } }

A a = new B();
a.hello(); // prints "A" — static method resolved by reference type A.
```

Mentor tip: static method hiding is confusing; avoid depending on this behavior — prefer instance methods for polymorphic behavior.

8) How do you utilize overloading concepts while implementing APIs?

Answer

Method overloading: same method name, different parameter lists. Overloading is used to provide multiple convenient ways to call the same logical operation — it's a way to emulate optional parameters and supply different entry points without creating multiple method names.

Practical API design uses

- **Convenience overloads:** e.g., `void send(Message m)` and `void send(Message m, boolean encrypt)`.

Default arguments pattern using overloads: provide a simple method that calls the full method:

```
void connect(String host) {
    connect(host, 80);
}

void connect(String host, int port) { ... }
```

- **Different input types:** allow inputs as `String`, `Path`, or `InputStream` for the same operation.
- **Fluent builder + overload combination:** where overloads create common defaults and builders provide full control.
- **Backward compatibility:** add new overloads instead of changing signatures of older methods.

Caution

- Avoid ambiguous overloads (same erasure under generics).
- Avoid too many overloads that make the API confusing — prefer a builder when there are many optional parameters.

Mentor tip: for APIs with many optional parameters, prefer a builder pattern; use overloads only for a small set of common convenience calls.

9) How do you handle exceptions in child class?

Answer

If you are **overriding** a method in a subclass, you must follow Java's exception rules:

- **Checked exceptions:** the overriding method **cannot throw broader checked exceptions** than the parent method. It may:
 - Throw the same checked exception(s),
 - Throw a subclass of the parent's checked exception(s),
 - Or throw no checked exceptions at all.
- **Unchecked exceptions (RuntimeException and its subclasses):** can be thrown freely — the compiler does not enforce restrictions.
- **Checked vs unchecked design:** use checked exceptions for recoverable conditions; use unchecked for programming errors.

Example

```
class Parent {  
    void process() throws IOException { ... }  
}  
  
class Child extends Parent {  
    // Allowed: narrower exception  
    @Override  
    void process() throws FileNotFoundException { ... }  
    // Not allowed: broader exception (e.g., Exception)  
}
```

Handling exceptions in a subclass (practical patterns)

- **Wrap and rethrow** checked exceptions into unchecked if the subclass cannot meaningfully recover.
- **Add context:** catch low-level exceptions and throw a domain-specific exception with more context (exception chaining via `new MyException("msg", cause)`).
- **Avoid swallowing** exceptions (don't catch and do nothing).

Mentor tip: document exceptions thrown by overridden methods and provide clear semantics about recoverability.

10) What is method reference and variable inference in Java?

Answer

Method references (Java 8) are shorthand for lambdas that simply call an existing method. They make code more compact and readable. Forms:

1. `ClassName::staticMethod` — static method reference.
2. `instance::instanceMethod` — instance method on a particular object.
3. `ClassName::instanceMethod` — instance method where the instance is the first parameter (used by functional interfaces).
4. `ClassName::new` — constructor reference.

Examples

```
// lambda
list.forEach(x -> System.out.println(x));

// method reference
list.forEach(System.out::println);

// constructor reference
Supplier<List<String>> s = ArrayList::new;
```

Variable inference (**var**)

Introduced in Java 10. You declare local variables using **var** and the compiler infers the type from the initializer. It is not dynamic typing — the type is fixed at compile time.

Example

```
var map = new HashMap<String, Integer>(); // map is HashMap<String,Integer> at compile-time
```

Benefits

- Reduces verbosity, especially for generic type-heavy declarations.
- Improves readability when the concrete type is obvious.

Drawbacks

- Overuse can obscure types and make code harder to read.
- Not allowed for fields, method parameters, or return types — only local variables with initializers.

Mentor tip: use `var` when it improves clarity (e.g., long generic types), and avoid it when it hides important type information.

11) Can you create your own exception class? How do you manage exception handling in your application?

Answer

Yes — **create custom exceptions** to model domain errors cleanly.

How to create

- For recoverable, checked exceptions extend `Exception`.
- For programming errors or unrecoverable conditions extend `RuntimeException`.
- Always provide constructors:
 - `Exception(String message)`
 - `Exception(String message, Throwable cause)`
 - `Exception(Throwable cause)`

Example

```
public class UserNotFoundException extends RuntimeException {  
    public UserNotFoundException(String message) { super(message); }  
    public UserNotFoundException(String message, Throwable cause) { super(message, cause); }  
}
```

Exception handling strategy (application-level best practices)

1. **Design exceptions by semantics** — domain-specific exceptions (e.g., `PaymentFailedException`) provide better context than generic ones.
2. **Checked vs unchecked:**
 - Use **checked** exceptions for recoverable, caller-handlable conditions.
 - Use **unchecked** for programming errors or when the cost of forcing handling outweighs benefits.
3. **Wrap low-level exceptions** into domain exceptions with context (`throw new MyDomainException("failed processing order " + id, e)`).
4. **Centralized handling:**
 - For web apps, use a global exception handler (`@ControllerAdvice` in Spring).

- For batch jobs, trap top-level exceptions and handle logging/alerting and graceful shutdown.
- 5. **Logging:**
 - Log exceptions with sufficient context (IDs, parameters), but avoid logging sensitive data.
- 6. **Avoid swallowing exceptions** (don't catch and ignore).
- 7. **Use exception chaining** to preserve root cause (`new Exception("context", cause)`).
- 8. **Fail fast:** validate inputs early and throw informative exceptions.

Mentor tip: make your exceptions expressive and document their usage in API contract; prefer narrow, meaningful exception types rather than `RuntimeException` everywhere.

12) Which version of Java are you using in your application? Tell me some common features?

Answer (mentor-style guidance for interviews / answers you can use)

What to say in an interview (honest and practical):

- “Our team currently uses an LTS Java: typically Java 17 or Java 21 depending on the project. We prefer an LTS because it has long-term support and stability; for new greenfield projects we consider the latest LTS (e.g., Java 21 or Java 25 after evaluating new features).”

Common features available in modern Java LTS versions (11, 17, 21)

- **Local-variable type inference (`var`)** (Java 10) — reduces verbosity.
- **Records** (Java 16) — compact immutable data carriers.
- **Pattern matching for `instanceof` and `switch`** (various JEPs) — simpler type checks.
- **Sealed classes** (Java 17) — control inheritance for domain modeling.
- **Text Blocks** (Java 13+) — easier multiline strings.
- **Enhanced `switch` / `switch` expressions** — more concise.
- **Stream API and lambda expressions** (Java 8) — functional-style programming.
- **Foreign Function & Memory API, Project Panama** (recent versions) — easier native interoperability.
- **Virtual threads (Project Loom)** — lightweight concurrency model (preview / stabilized incrementally).
- **String templates, record patterns, value types (Project Valhalla)** — evolving across recent releases.

(If asked in an interview, name the LTS you actually use and list 4–6 features you use daily, e.g., records, streams, Optionals, new Date/Time API.)

13) What are the enhancements in Java Date/Time APIs in Java 8?

Answer

Problems with old `java.util.Date` and `Calendar`

- Mutable, not thread-safe, confusing API (month index starts at 0), poor timezone handling.

Java 8 Date-Time (`java.time`) improvements

- **Immutable and thread-safe classes:** `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`.
- **Clear separation of concerns:**
 - `LocalDate` — date without time or timezone.
 - `LocalTime` — time-of-day without date.
 - `LocalDateTime` — date + time without timezone.
 - `ZonedDateTime` — date + time + timezone.
 - `Instant` — machine time (timestamp) in UTC.
- **Better timezone handling:** `ZoneId`, `ZoneOffset`.
- **Human-readable durations:** `Duration` for time-based amounts, `Period` for date-based amounts.
- **Fluent API:** `LocalDate.now().plusDays(2).with(TemporalAdjusters.firstDayOfMonth())`.
- **Parsing/formatting:** `DateTimeFormatter` is immutable and thread-safe.
- **Interoperability:** conversion utilities to/from legacy `Date/Calendar`.

Example

```
LocalDate dob = LocalDate.of(1990, Month.JANUARY, 1);
ZonedDateTime meeting = ZonedDateTime.of(2025, 10, 26, 10, 0, 0, 0, ZoneId.of("Asia/Kolkata"));
Duration d = Duration.between(Instant.now(), Instant.parse("2025-12-31T00:00:00Z"));
```

Mentor tip: always use `java.time` for new code. For reading/writing older APIs, use conversion utilities but move to `java.time` internally.

14) What is `hashCode` and `equals` contract and how do you use it in your application? Tell me some use-cases.

Answer

Contract summary

- If two objects are equal according to `equals(Object)`, then calling `hashCode()` on each must produce the same integer result.
- If two objects are unequal according to `equals()`, their `hashCode()` may be the same (collision possible), but good hash functions minimize collisions.

- `equals()` must be:
 - Reflexive: `x.equals(x)` is true.
 - Symmetric: `x.equals(y)` implies `y.equals(x)`.
 - Transitive: `x.equals(y)` and `y.equals(z)` implies `x.equals(z)`.
 - Consistent: repeated calls return same result if objects unchanged.
 - `x.equals(null)` returns false.

Why it matters

- `HashMap`, `HashSet`, and `Hashtable` use `hashCode()` to find the right bucket and `equals()` to find an exact match in that bucket. Violating contract breaks collection behavior.

How to implement

- Use the same fields in both `equals()` and `hashCode()`.
- For performance and consistency, prefer immutable fields for fields used in equality.
- Use `Objects.equals()` and `Objects.hash(...)` or IDE-generated implementations or Lombok/record helpers.

Example

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Employee)) return false;
    Employee e = (Employee) o;
    return id == e.id && Objects.equals(email, e.email);
}

@Override
public int hashCode() {
    return Objects.hash(id, email);
}
```

Use cases

- **Keys in `HashMap`**: employee lookup by a key object.
- **Elements in `HashSet`**: uniqueness semantics for domain objects.
- **Caching**: correct cache key equality.
- **Collections operations**: removing, contains checks rely on contract.

Mentor tip: Pay extra attention to `equals/hashCode` in classes used as keys or set elements. Using generated code or `record` classes reduces bugs.

15) What is `String` and how does it differ from `StringBuffer` and `StringBuilder`? Use-cases?

Answer

`String`

- Immutable sequence of characters.
- Safe to share across threads.
- Any modification creates a new `String` object.

`StringBuffer`

- Mutable sequence of characters.
- **Synchronized** — thread-safe for concurrent access.
- Slightly slower due to synchronization overhead.

`StringBuilder`

- Mutable sequence of characters.
- **Not synchronized** — not thread-safe but faster in single-threaded scenarios.
- Introduced in Java 5 as a high-performance alternative to `StringBuffer` when thread-safety not required.

Which to use when

- `String`: use for keys, constants, when immutability is desirable.
 - `StringBuilder`: use for local string concatenation in single-threaded contexts — preferred for building long strings (e.g., loops).
 - `StringBuffer`: use when you need thread-safe mutation across threads (rarely required; better patterns exist).
-

Feature	String	StringBuffer	StringBuilder
Mutability	Immutable	Mutable	Mutable
Thread Safety	Not thread-safe	Thread-safe (synchronized)	Not thread-safe
Performance	Slower for many modifications	Slower than StringBuilder (due to sync)	Faster (no sync overhead)
Introduced in	Java 1.0	Java 1.0	Java 1.5
Use Case	Constant text / keys	Multi-threaded text modification	Single-threaded text modification

Example

```
StringBuilder sb = new StringBuilder();
for (String s : words) sb.append(s).append(' ');
String result = sb.toString();
```

Example 1: String (Immutable)

```
public class StringExample {
    public static void main(String[] args) {
        String str = "Hello";
        str.concat(" World"); // creates new String, original
        // unchanged
        System.out.println(str); // Output: Hello

        str = str.concat(" Java"); // reassign to new String
        System.out.println(str); // Output: Hello Java
    }
}
```

- ✓ Each modification creates a **new object**.
Useful for **constants**, **keys**, and **read-only data**.
-

Example 2: StringBuffer (Mutable and Thread-Safe)

```
public class StringBufferExample {  
    public static void main(String[] args) {  
        StringBuffer sb = new StringBuffer("Hello");  
        sb.append(" World");  
        sb.append(" Java");  
        System.out.println(sb); // Output: Hello World Java  
    }  
}
```

- ✓ **Modifies the same object** in memory.
 - ✓ **Thread-safe** due to synchronization.
- Use in **multi-threaded environments** when multiple threads modify the same string.
-

Example 3: StringBuilder (Mutable and Fast)

```
public class StringBuilderExample {  
    public static void main(String[] args) {  
        StringBuilder sb = new StringBuilder("Hello");  
        sb.append(" World");  
        sb.append(" Java");  
        System.out.println(sb); // Output: Hello World Java  
    }  
}
```

- ✓ **Faster** than **StringBuffer** because it's **not synchronized**.
Use in **single-threaded** operations where performance is important (e.g., loops or concatenations).
-

Performance Comparison Example

```
public class StringPerformanceTest {
    public static void main(String[] args) {
        long startTime, endTime;

        // Using String
        startTime = System.nanoTime();
        String str = "Java";
        for (int i = 0; i < 10000; i++) {
            str = str + " Programming";
        }
        endTime = System.nanoTime();
        System.out.println("String time: " + (endTime - startTime) + " ns");

        // Using StringBuilder
        startTime = System.nanoTime();
        StringBuilder sb = new StringBuilder("Java");
        for (int i = 0; i < 10000; i++) {
            sb.append(" Programming");
        }
        endTime = System.nanoTime();
        System.out.println("StringBuilder time: " + (endTime - startTime) + " ns");
    }
}
```

Output (approx):

```
String time: 120000000 ns
StringBuilder time: 500000 ns
```

➡ **StringBuilder** is **much faster** for repeated concatenations.

16) What are the possible ways to create DTO/POJO in Java?

Answer

Common ways to create DTOs / POJOs:

1. Manual POJO

- Plain class with private fields, public getters/setters, **equals**, **hashCode**, **toString**.
- Full control, but lots of boilerplate.

2. Lombok

- Use annotations like `@Data`, `@Value`, `@Builder`, `@AllArgsConstructor`, `@NoArgsConstructor`.
 - Reduces boilerplate. But adds a compile-time dependency and requires tooling support.
3. **Records** (Java 16+)
- `public record User(int id, String name) {}` — compiler generates immutable fields, `equals`, `hashCode`, `toString`.
 - Great for immutable DTOs.
4. **Auto-generated code / tools**
- IDE generation, codegen from schemas (OpenAPI codegen, Protobuf/Avro codegen), MapStruct for mapping.
5. **Framework/proxy-driven approaches**
- Some frameworks synthesize DTOs (less common).

Which to choose

- **Records**: for simple, immutable DTOs.
- **Lombok**: if you need builders, mutable DTOs, or to avoid updating Java version.
- **Manual**: when you require complete control (e.g., JPA entities often require no-arg constructor and mutable fields).

Mentor tip: prefer `record` for simple immutable data holders; use Lombok for advanced boilerplate reduction but be prepared to explain generated behavior in interviews.

17) What is Record in Java? Opinion: Lombok, Records, and classic DTOs with setters/getters — which to use?

Answer

What is `record`?

A `record` is a special kind of class (Java 16+) designed to be a compact, immutable data carrier. The compiler automatically provides:

- final fields,
- canonical constructor,
- accessors (not named `getX` but `x()`),
- `equals()`, `hashCode()`, and `toString()` implementations.

Example

```
public record Employee(int id, String name) {}  
// usage: Employee e = new Employee(1, "Alice"); e.name() returns "Alice"
```

Pros

- Eliminates boilerplate for value objects.
- Immutable by default — good for safe concurrency, hash keys.

- Clear semantic: this type is a pure data carrier.

Cons

- Not suitable for JPA entities (JPA expects no-arg constructors, mutable fields).
- Less flexible if you need complex behaviors or large mutable states.
- Cannot extend other classes (implicitly final).

Lombok vs Records vs classic DTO

- **Records**: use when you want **immutable** lightweight DTOs and can use Java 16+. Great for API responses, internal DTOs, map keys, etc.
- **Lombok**: use when you need:
 - Builders ([@Builder](#)),
 - Compatibility with older Java versions,
 - Mutable DTOs or special constructor patterns,
 - JPA-supporting entities (Lombok can help but use with caution).
- **Classic DTOs (setters/getters)**: use when:
 - You must support frameworks requiring mutable beans (JPA),
 - You need field-level injection/deserialization that expects mutability.

Mentor recommendation

- Prefer [record](#) for read-only DTOs and value objects.
 - Use Lombok to reduce boilerplate for mutable DTOs or when working with older Java versions; ensure team is comfortable with Lombok.
 - For entities with lifecycle and persistence (JPA), use classic mutable POJOs (or carefully structured record wrappers) because JPA's lifecycle often expects mutability.
-

18) What is the latest version of Java? Tell me some features from it.

Answer

(Up-to-date note — I checked the current release information.) The latest Java SE platform release (as of October 2025) is **JDK 25 (Java SE 25)** released in September 2025 — and update releases (e.g., 25.0.1) have followed. JDK 25 is the most recent LTS release in the JDK train as of this date. ([Oracle](#))

Key features introduced in recent Java (across JDK 17–25 era) — these are some of the items you can mention:

- **Record patterns & pattern matching enhancements** — more concise destructuring and conditional logic.
- **Sealed classes** — restrict which classes can extend/implement a type.
- **Virtual threads (Project Loom)** — lightweight threads for massive concurrency (stabilizing in recent releases).

- **Foreign Function & Memory API (Project Panama)** — safer, high-performance interop with native code.
- **String templates & text blocks improvements** (some features previewed/adjusted across releases).
- **Value types / Project Valhalla developments** — improved memory layout for value classes (progress across releases).
- **Scoped values & structured concurrency** — API improvements for concurrent programming ergonomics.
- **Language and library refinements** — small language features that improve patterns and developer ergonomics.

Mentor tip: if asked which version you use in work, name the LTS release your company uses (e.g., 17, 21, or 25) and highlight 3–4 features from it that you actually use (for example, records, sealed classes, pattern matching, and virtual threads if you’ve experimented with them).

19) Which version of Java do you like most and why? Why do we say Java 8 is the baseline for functional programming approach?

Answer

Mentor-style guidance on answering this

- Many engineers say **Java 8** is the most influential version because it introduced the core functional programming features that changed how Java is written: **lambda expressions, the Stream API, method references, Optional, and the new Date/Time API**. These features enabled a functional-style, declarative way to process collections and led to clearer, more compact code.
- In practical production work, **LTS releases** like Java 11, 17, or newer LTS (e.g., 21 or 25) are often preferred for stability. Pick whichever your organization uses and highlight features you rely on.

Why Java 8 is considered the baseline for functional programming

- **Lambdas** — allowed passing behavior (functions) as parameters.
- **Functional interfaces** (**Function**, **Predicate**, **Consumer**, **Supplier**) — standardized function shapes.
- **Streams** — enabled fluent, declarative pipelines (map/filter/reduce) over collections.
- **Parallel streams** — allowed concise parallelism for collection processing.
- **Optional** — encouraged explicit handling of absent values, reducing NPEs.
- These constructs together made functional programming idioms practical in Java.

Mentor tip: in interviews, acknowledge Java 8’s importance, but also mention the practical value of recent language features (records, pattern matching, virtual threads).

20) Tell me features of Java 8, how Lambda and Stream help us write better and simplified code.

Answer

Key Java 8 features

- **Lambda expressions** — concise anonymous functions.
- **Method references** — shorthand for lambdas that call existing methods.
- **Functional interfaces** — single abstract method interfaces (e.g., [Supplier](#), [Function](#), [Consumer](#), [Predicate](#)).
- **Stream API** — a rich set of operations for processing collections in a declarative way.
- **Optional** — container object to represent optional values.
- **New Date-Time API ([java.time](#))** — immutable, thread-safe date/time classes.
- **Default and static methods in interfaces** — allow adding methods to interfaces without breaking implementers.

How Lambdas & Streams simplify code

Less boilerplate: replace anonymous inner classes with lightweight lambdas.

```
// before:
list.forEach(new Consumer<String>() { public void accept(String s) { System.out.println(s); }});

// after:
list.forEach(s -> System.out.println(s));

// or method reference:
list.forEach(System.out::println);
```

Declarative processing: operation chain expresses *what* is done rather than *how*.

```
List<String> result = people.stream()
    .map(Person::getName)
    .filter(name -> name.startsWith("A"))
    .distinct()
    .sorted()
    .collect(Collectors.toList());
```

- **Parallelism:** switch to [parallelStream\(\)](#) to express parallel processing with minimal code change (but be aware of thread-safety).
- **Composability:** small, reusable function objects ([Predicate](#), [Function](#)) can be composed.

Pitfalls

- Overuse of parallel streams can degrade performance if operations are not CPU-bound or are I/O bound.
- Lambda expressions can obscure intent if used to write very complex inline logic — favor small, named methods for complex logic.

Mentor tip: use lambdas/streams to express pipeline logic clearly; keep functions small and testable, and profile before switching to parallelism.

21) What is the default method and why was it introduced? Where to use this? Use-cases.

Answer

Default methods (Java 8) are methods in interfaces with a default implementation using the **default** keyword. They allow adding new methods to an interface without breaking existing implementations (backward compatibility).

Why introduced

- To evolve interfaces (especially JDK interfaces like **Collection**) without forcing all implementers to implement new methods.
- To support the addition of behaviour to interfaces safely.

Example

```
public interface Logger {  
    void log(String message);  
    default void logError(String message) {  
        log("ERROR: " + message);  
    }  
}
```

Use-cases

- **Interface evolution:** add new helper methods to widely implemented interfaces without breaking code.
- **Optional behaviors:** provide default implementations that can be overridden by implementers.
- **Mix-in behavior:** provide reusable behavior for multiple classes to implement.

Diamond problem

- If a class implements two interfaces that define the same default method, the class must override the method or explicitly choose which default to call — resolves ambiguity intentionally.

Mentor tip: use default methods sparingly and thoughtfully. They're powerful for library designers but can complicate interface hierarchies if overused.

22) What is **Optional**? What does it do? How does it help to avoid **NullPointerException**?

Answer

Optional<T> is a container object used to represent the presence or absence of a value. It's not intended to replace all uses of **null**, but to make absent values explicit in APIs.

Key methods

- **Optional.of(value)** — non-null value (throws **NullPointerException** for null).
- **Optional.ofNullable(value)** — may contain null.
- **optional.isPresent()** / **optional.isEmpty()**
- **optional.orElse(default)** / **orElseGet(supplier)** / **orElseThrow()**
- **optional.map(...)** / **flatMap(...)** / **filter(...)**

How it helps avoid NPEs

- It forces the caller to confront the possibility of no value — rather than returning **null** silently.
- Encourages methods like **map()** and **orElse()** to handle absence in a composable way.

Example

```
Optional<User> userOpt = repo.findById(id);  
String email = userOpt.map(User::getEmail).orElse("not-provided@example.com");
```

Caveats

- Don't use **Optional** for fields in entities or serialized fields — it's intended primarily for return types and streams.
- Avoid **Optional.get()** without checking **isPresent()** (similar to calling **get()** on a maybe-empty container).

Mentor tip: use **Optional** in public API return types to communicate optionality explicitly and reduce casual **null** usages.

23) What is variable inference? Can you tell me its benefits and drawbacks?

Answer

Variable inference via `var` (Java 10+) lets the compiler infer the type of a local variable from its initializer. It is *static* type inference — the type is known at compile time, not runtime.

Example

```
var list = new ArrayList<String>(); // inferred type: ArrayList<String>
```

Benefits

- **Reduces verbosity**, especially for generic types (e.g., `Map<String, List<MyType>>>`).
- **Improves readability** when the right-hand side already communicates the type.
- Cleaner code when dealing with long generic types.

Drawbacks

- **Obscures type** if initializer is not clear (e.g., `var result = doSomething()` — what is result's type?).
- **Not allowed** for uninitialized locals, fields, method parameters, or return types.
- **Can hide subtle bugs** (e.g., diamond inference to raw types) if misused.

Rules

- `var` requires an initializer.
- It's purely for local variables (not method signatures, fields, or catch parameters).

Mentor tip: use `var` where the type is obvious (e.g., `var stream = list.stream()`) and avoid it where it reduces clarity.

24) What is Sealed classes/Interface in Java and where to use it?

Answer

Sealed types (Java 17+) allow a class or interface to **restrict which other classes or interfaces may extend or implement it** using the `permits` clause. It's a tool to model closed hierarchies explicitly.

Syntax

```
public sealed interface Shape permits Circle, Rectangle {}  
public final class Circle implements Shape {}  
public final class Rectangle implements Shape {}
```

Benefits

- **Stronger modeling** for domain hierarchies (you explicitly list permitted subclasses).
- Helps with exhaustive `switch` statements (pattern matching benefits).
- Improves security and reasoning — the compiler knows all possible subclasses.

- Useful when you want controlled extension — either by you or a set of trusted implementations.

Use-cases

- Algebraic data types (ADTs) and sum types modeling.
- Domain models where you want to tightly control variants (e.g., `PaymentMethod = Card, Paypal, WireTransfer`).
- Libraries where you want to permit extension only to specific internal subclasses.

Mentor tip: combine sealed types with `record` subclasses to get compact, safe ADT-like structures.

25) How do you handle exceptions in your application? Tell me some best practices for it.

Answer

High-level strategy

- **Catch at the right level** — low-level code should throw or wrap exceptions; high-level code should handle them and present user-friendly messages or retry logic.
- **Have a centralized exception policy** — for web apps use a global handler, for batch use a top-level catch that reports and continues/aborts appropriately.

Best practices

1. **Use meaningful exceptions:** domain-specific exceptions give callers actionable information.
2. **Wrap low-level exceptions with higher-level context:** `throw new OrderProcessingException("Order 123 failed", e)`.
3. **Don't swallow exceptions:** always log, rethrow, or handle properly.
4. **Use checked vs unchecked judiciously:** prefer checked for recoverable business errors that callers should handle.
5. **Log with context:** include correlation IDs, user IDs, trace IDs — but avoid logging sensitive data (PII).
6. **Fail fast where appropriate:** detect invalid inputs early and throw `IllegalArgumentException` or `NullPointerException` with clear messages.
7. **Retry/Idempotency:** implement retry logic for transient errors and ensure idempotency for retried operations.
8. **Circuit breakers / bulkheads:** for distributed systems, use resilient patterns to avoid cascading failures.
9. **Use monitoring and alerting:** exceptions should surface to monitoring and not just logs.
10. **User-friendly error responses:** convert exceptions into well-formed API error payloads (status codes, error codes, messages).
11. **Test exception scenarios:** write unit/integration tests for error paths and simulate partial failures.

Mentor tip: treat exceptions as part of your API contract — document what callers should expect and how to react.

26) What is OOPS and tell me about its concepts?

Answer

OOP (Object-Oriented Programming) is a programming paradigm centered on objects — encapsulating state and behavior.

Core concepts

1. **Encapsulation:** bundling data (fields) and methods; control access via visibility (**private**, **protected**, **public**). Encapsulation enforces invariants and hides internal representation.
2. **Abstraction:** expose essential behavior and hide implementation details (via interfaces and abstract classes).
3. **Inheritance:** reuse code by deriving classes from base classes (**extends**) — models an “is-a” relationship.
4. **Polymorphism:** objects of different classes can be treated uniformly through a base type, with method calls resolved at runtime (late binding) — supports flexible designs.

Supporting concepts

- **Composition:** “has-a” relationships — favor composition over inheritance for flexibility.
- **SOLID principles:** Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion help design maintainable OOP systems.

Mentor tip: use OOP where domain objects and behavior map naturally to classes; prefer composition for flexible designs and limit deep inheritance hierarchies.

27) What is the IS-A and HAS-A relationship? How have you used it in your application? Give examples.

Answer

IS-A (Inheritance)

- Represents a subtype relationship. Example: **Car IS-A Vehicle** (if **Car extends Vehicle**).
- Use when subclass **is** a specialized form of the parent.

HAS-A (Composition/Aggregation)

- Represents ownership/containment. Example: **Car HAS-A Engine** (Car has an Engine field).
- More flexible: you can replace/reuse components and mock them in tests.

Practical usage

- **IS-A:** `class AdminUser extends User` — when AdminUser truly is a specialized User and obeys Liskov Substitution.
- **HAS-A:** `class Order { private PaymentProcessor processor; }` — Order delegates payment behavior to a PaymentProcessor implementation.

Mentor tip: prefer composition (HAS-A) over inheritance (IS-A) unless you're modeling a true subtype; composition favors loose coupling and testability.

28) What is Abstraction? Why do we go for it? Use-cases and examples from applications.

Answer

Abstraction is the practice of exposing only essential features while hiding underlying complexity. It's implemented via **interfaces** and **abstract classes** in Java.

Why use abstraction

- **Reduce complexity** for consumers of APIs.
- **Encourage loose coupling:** code depends on abstractions, not concrete implementations.
- **Enable substitution:** swap implementations without changing clients (DI, testing with mocks).
- **Encapsulate implementation details** and provide stable contracts.

Use-cases

- **Service interfaces:** `PaymentService` interface with implementations `StripePaymentService`, `PaypalPaymentService`.
- **Repository pattern:** `UserRepository` hides data store choice (DB, in-memory).
- **UI abstraction:** `Renderer` interface for different output formats (HTML/PDF/JSON).

Example

```
public interface PaymentService {
    PaymentResult process(PaymentRequest req);
}
// Implementation
public class StripePaymentService implements PaymentService { ... }
```

Mentor tip: design code to depend on interfaces; it simplifies testing and reduces coupling.

29) What is late binding and early binding? Why do we go for it? How do you handle exceptions in case of overriding?

Answer

Early binding (static binding)

- Method to be invoked is determined at compile time. Occurs for overloaded methods and static methods.
- Typically faster because the call target is known at compile time.

Late binding (dynamic binding / runtime binding)

- The method to invoke is decided at runtime based on the object's actual type. Happens with instance method overriding (polymorphism).
- Enables flexible behavior: call `obj.doWork()` and actual runtime class's method runs.

Why use late binding

- To achieve polymorphism and flexible extensibility: you call methods on abstractions and let subclasses provide behavior.

Exceptions & overriding

- When overriding a method, you **cannot throw broader checked exceptions** than the overridden method; you may throw the same, narrower checked exceptions, or none. (Seen also in Q9.)
- Unchecked exceptions are not checked by the compiler and can be thrown freely.

Example

```
class Parent { void run() throws IOException {} }  
  
class Child extends Parent { @Override void run() throws FileNotFoundException {} } // ok
```

Mentor tip: design API methods to declare exceptions that callers can reasonably handle; use unchecked exceptions for programming errors.

30) What is a functional interface? Why is it required and list some pre-defined functional interfaces in Java.

Answer

Functional interface: An interface with exactly one abstract method. It can have default and static methods but only one abstract method qualifies it as a functional interface. Functional interfaces are the targets for lambda expressions and method references.

Why required

- Lambdas need a single method “shape” to implement — functional interfaces define that contract.
- Improves clarity & reusability by naming common function shapes.

Pre-defined functional interfaces (java.util.function)

- **Supplier<T>** — supplies results (T get()).
- **Consumer<T>** — accepts an argument and returns nothing (void accept(T)).
- **Function<T,R>** — maps from T to R (R apply(T)).
- **Predicate<T>** — boolean-valued function (boolean test(T)).
- **BiFunction<T,U,R>** — two-arg function.
- **UnaryOperator<T>/BinaryOperator<T>** — specializations returning the same type.
- **Runnable** (from core) – no-arg void; **Callable<V>** returns V and may throw.

Custom functional interfaces

- You can annotate with **@FunctionalInterface** for clarity and to get a compile-time error if more than one abstract method is added.

Example

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

Mentor tip: prefer standard **java.util.function** interfaces when possible for interoperability with existing APIs.

31) What is generics and why is it required? Do you see any enhancements in the latest Java version?

Answer

What is generics?

A mechanism that allows types (classes and interfaces) to be parameterized with type arguments (e.g., **List<String>**). It enables writing general-purpose, type-safe code.

Why required

- **Type safety** at compile time.
- **DRY code**: reusable classes/methods without casting.
- **Expressiveness**: method contracts express the types involved.

Core features

- Type parameters (<T>)
- Bounded types (<T extends Number>)
- Wildcards (?, ? extends T, ? super T)
- Generic methods (static <T> T identity(T t))

Enhancements in recent Java versions

- **Improved type inference:** the compiler is better at inferring generic types in complex expressions (various JEPs).
- **Pattern matching + records:** better combination with generics in type-based logic.
- The latest experimental features in the language and jdk aim to make generics easier to use, but the core generics model (with type erasure) remains fundamental.

Mentor tip: be comfortable with PECS (**P**roducer **e**xtends, **C**onsumer **s**uper) and type inference rules. For API design, prefer generics over **Object** returns.

32) What is the contract between `hashCode(..)` and `equals(..)` method?

Answer

This was already covered earlier — concise restatement:

The contract

- If `a.equals(b)` returns **true** then `a.hashCode() == b.hashCode()` must be true.
- If `a.equals(b)` returns **false**, there is no requirement about hash codes — collisions are allowed.
- `hashCode()` must be consistent across executions while the object's state used in `equals` remains unchanged.

Consequences

- When overriding `equals()`, **always** override `hashCode()` too.
- Use immutable fields for equality to avoid breakage in hash containers.

Mentor tip: explain with an example during interviews and mention `Objects.equals`, `Objects.hash`, and `record` (which auto-generates both) as recommended tools.

Collections API

1. What are the design patterns you have recently used in your project

In most real-world projects, we apply design patterns naturally once we start organizing code for flexibility and testability. Some of the most commonly used patterns are Singleton, Factory, Builder, Strategy, Observer, and DAO (Data Access Object).

The **Singleton pattern** ensures that only one instance of a class exists throughout the application. For example, an application-wide configuration class can be implemented as a Singleton so that every part of the system refers to the same configuration.

```
public final class AppConfig {
    private static final AppConfig INSTANCE = new AppConfig();
    private Properties props;

    private AppConfig() {
        props = new Properties();
        props.setProperty("url", "http://example.com");
    }

    public static AppConfig getInstance() {
        return INSTANCE;
    }

    public String get(String key) {
        return props.getProperty(key);
    }
}
```

The **Factory pattern** helps when object creation logic is complex or dependent on input. For example, when sending different types of notifications (email, SMS, push), a factory method can decide which implementation to return.

```
interface Notification {
    void send(String message);
}
```

```

class EmailNotification implements Notification {
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }
}

class NotificationFactory {
    public static Notification create(String type) {
        if (type.equalsIgnoreCase("email")) return new EmailNotification();
        throw new IllegalArgumentException("Unknown type");
    }
}

```

The **Builder pattern** is useful when constructing objects with many optional parameters, like a **User** object with multiple attributes.

```

class User {
    private String name, email;
    private int age;

    public static class Builder {
        private String name, email;
        private int age;

        public Builder name(String n) { this.name = n; return this; }
        public Builder email(String e) { this.email = e; return this; }
        public Builder age(int a) { this.age = a; return this; }
        public User build() { return new User(this); }
    }

    private User(Builder b) {
        this.name = b.name; this.email = b.email; this.age = b.age;
    }
}

```

Strategy pattern lets you change algorithms dynamically at runtime. For instance, different pricing strategies can be applied to a shopping cart without changing the main logic.

```

interface PricingStrategy {
    double getPrice(double price);
}

```

```
class DiscountStrategy implements PricingStrategy {
    public double getPrice(double price) { return price * 0.9; }
}

class ShoppingCart {
    private PricingStrategy strategy;
    public void setStrategy(PricingStrategy s) { this.strategy = s; }
    public double checkout(double price) { return strategy.getPrice(price); }
}
```

Observer pattern is useful when one event triggers multiple actions. For example, when an order is placed, multiple systems (email, SMS, analytics) may need to respond.

Finally, **DAO pattern** separates database access logic from business logic, improving maintainability and testability.

2. What is the Java Collections Framework, and what are its main interfaces?

The Java Collections Framework (JCF) is a unified architecture for storing and manipulating groups of data. It provides interfaces, classes, and algorithms to manage data efficiently. It's like a digital toolbox that contains ready-made data structures.

The main interfaces are:

- **Collection**: The root of most collection types. It defines basic operations like [add](#), [remove](#), [size](#), and [iterator](#).
- **List**: An ordered collection that allows duplicate elements. Common implementations are [ArrayList](#), [LinkedList](#), and [Vector](#).
- **Set**: A collection that does not allow duplicates. Implementations include [HashSet](#), [LinkedHashSet](#), and [TreeSet](#).
- **Queue**: A collection for holding elements before processing, usually in FIFO order. Examples are [LinkedList](#) and [PriorityQueue](#).
- **Map**: A collection of key-value pairs where keys are unique. Common implementations include [HashMap](#), [TreeMap](#), and [LinkedHashMap](#).

Example:

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");

Set<Integer> numbers = new HashSet<>();
numbers.add(10);
numbers.add(20);
```

```
Map<Integer, String> students = new HashMap<>();
students.put(1, "John");
students.put(2, "Mary");
```

3. Explain the difference between ArrayList and LinkedList. When would you use one over the other?

ArrayList and **LinkedList** both implement the **List** interface but have different internal data structures.

- **ArrayList** uses a dynamic array. Access by index is very fast ($O(1)$), but inserting or removing elements in the middle is slow because other elements must be shifted.
- **LinkedList** uses a doubly linked list where each node holds data and references to the previous and next nodes. Insertions and deletions are faster ($O(1)$) if you already have the node reference, but random access is slow ($O(n)$) because you must traverse the list.

Use **ArrayList** when you need fast access and few insertions/deletions. Use **LinkedList** when you frequently add or remove elements from the ends.

Example:

```
List<String> arrayList = new ArrayList<>();
arrayList.add("A");
arrayList.add("B");
System.out.println(arrayList.get(1)); // Fast access

List<String> linkedList = new LinkedList<>();
linkedList.add("A");
linkedList.add("B");
linkedList.remove("A"); // Fast removal
```

4. How does a HashMap work? What are its key properties?

A **HashMap** stores data in key-value pairs and uses a hash table to achieve fast access. When you store a value, it calls **hashCode()** on the key to find the correct bucket (array index). If two keys map to the same bucket (collision), they are stored in a linked list or a tree structure inside that bucket.

Steps for **put(key, value)**:

1. Compute the hash code of the key.
2. Find the correct bucket.
3. If the bucket is empty, store the entry there.
4. If not, compare keys using **equals()** to check if it already exists.

5. If the key exists, replace its value; otherwise, add a new entry.

Example:

```
Map<String, Integer> map = new HashMap<>();
map.put("apple", 1);
map.put("banana", 2);
System.out.println(map.get("apple"));
```

Key properties:

- Average time complexity for get/put: $O(1)$
- Allows one **null** key and multiple **null** values
- Not thread-safe
- Order is not guaranteed

5. What are the differences between HashMap and TreeMap?

HashMap stores entries in an unordered way, while **TreeMap** stores them in sorted order based on their keys.

Feature	HashMap	TreeMap
Order	Unordered	Sorted
Performance	$O(1)$ average	$O(\log n)$
Null keys	Allows one	Does not allow
Backed by	Hash table	Red-black tree

Example:

```
Map<String, Integer> hashMap = new HashMap<>();
hashMap.put("B", 2);
hashMap.put("A", 1);

Map<String, Integer> treeMap = new TreeMap<>(hashMap);
System.out.println(treeMap); // Sorted order: {A=1, B=2}
```

Use **HashMap** for fast lookups. Use **TreeMap** when you need sorted keys or range queries.

6. What is the significance of ConcurrentHashMap? How does it differ from HashMap?

`ConcurrentHashMap` is a thread-safe version of `HashMap` that allows concurrent access without locking the entire map.

In earlier Java versions, `ConcurrentHashMap` divided the map into segments, each with its own lock. Modern versions use finer-grained mechanisms and non-blocking algorithms.

Key differences:

- `HashMap` is not thread-safe.
- `ConcurrentHashMap` allows concurrent read/write operations.
- It does not allow `null` keys or values.
- Iterators are weakly consistent — they don't throw exceptions if the map changes during iteration.

Example:

```
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
map.put("a", 1);
map.putIfAbsent("b", 2);
map.computeIfPresent("a", (k, v) -> v + 1);
```

7. How does LinkedHashMap maintain insertion order? What are its advantages over HashMap?

`LinkedHashMap` extends `HashMap` but adds a doubly linked list to maintain the insertion order of elements. When you iterate through it, elements appear in the order they were inserted.

It can also be configured for access order (recently accessed entries appear last), which makes it ideal for implementing LRU caches.

Example of an LRU cache:

```
LinkedHashMap<Integer, String> cache = new LinkedHashMap<>(16, 0.75f, true) {
    protected boolean removeEldestEntry(Map.Entry<Integer, String> eldest) {
        return size() > 3;
    }
};
```

Advantages:

- Predictable iteration order.
- Easy to implement caches.
- Slightly slower than `HashMap` due to linked structure.

8. What is the difference between HashSet and TreeSet? When would you use each?

`HashSet` and `TreeSet` both prevent duplicate elements, but `HashSet` is unordered and faster, while `TreeSet` keeps elements sorted.

Feature	HashSet	TreeSet
Order	Unordered	Sorted
Speed	O(1)	O(log n)
Null	Allows one	Does not allow
Backed by	HashMap	TreeMap

Example:

```
Set<Integer> hashSet = new HashSet<>(Arrays.asList(3, 1, 2));
System.out.println(hashSet); // Unordered

Set<Integer> treeSet = new TreeSet<>(hashSet);
System.out.println(treeSet); // [1, 2, 3]
```

Use `HashSet` when performance matters, and `TreeSet` when order matters.

9. How does PriorityQueue work, and what are its typical use cases?

`PriorityQueue` is a queue where elements are ordered by priority rather than insertion order. Internally, it's implemented using a binary heap.

When you insert an element, it is placed at the correct position based on its priority. The smallest (or highest-priority) element is always at the head.

Example:

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(5);
pq.add(1);
pq.add(3);
System.out.println(pq.poll()); // 1
```

Use cases include scheduling tasks, graph algorithms like Dijkstra, and event simulation system

10. Explain the concept of Hashtable. How does it differ from HashMap?

Hashtable is an older synchronized map introduced before the Java Collections Framework. Every method is synchronized, which means only one thread can access it at a time.

Key differences:

- Thread-safe but slower.
- Does not allow **null** keys or values.
- Considered legacy — replaced by **HashMap** and **ConcurrentHashMap**.

Example:

```
Hashtable<String, Integer> table = new Hashtable<>();  
table.put("A", 1);
```

11. What is the role of the Collection interface in Java? How does it differ from List, Set, and Queue?

Collection is the root interface for most data structures. It defines basic operations such as **add**, **remove**, **size**, and **iterator**.

List, **Set**, and **Queue** extend it:

- **List** is ordered and allows duplicates.
 - **Set** does not allow duplicates.
 - **Queue** is used for processing elements in order (usually FIFO).
-

12. How does the Iterator interface work? What is the difference between Iterator and ListIterator?

Iterator provides a way to traverse a collection without knowing its structure. It supports `hasNext()`, `next()`, and `remove()`.

ListIterator extends **Iterator** and adds more features:

- Can move both forward and backward.
- Allows adding or replacing elements while iterating.

Example:

```
List<String> list = new ArrayList<>(List.of("A", "B", "C"));
ListIterator<String> it = list.listIterator();
while (it.hasNext()) {
    String s = it.next();
    if (s.equals("B")) it.set("Beta");
}
```

13. What are Collections.synchronizedList, Collections.synchronizedMap, and Collections.synchronizedSet used for?

These methods wrap normal collections in a synchronized version to make them thread-safe.

Example:

```
List<String> list = Collections.synchronizedList(new ArrayList<>());
synchronized (list) {
    for (String item : list) {
        System.out.println(item);
    }
}
```

They provide simple synchronization but can become a bottleneck under high concurrency.

14. Describe how the List interface works and its common implementations.

The **List** interface represents an ordered collection that allows duplicates and positional access.

Common implementations:

- **ArrayList** – dynamic array, fast random access.
 - **LinkedList** – doubly linked list, fast insert/remove.
 - **Vector** – synchronized, legacy.
 - **CopyOnWriteArrayList** – thread-safe, uses copy-on-write.
-

15. How does the Set interface handle duplicate elements? What are its common implementations?

A **Set** ensures all elements are unique. Uniqueness is determined using `hashCode()` and `equals()`.

Common implementations:

- **HashSet** – unordered, fast operations.
- **LinkedHashSet** – maintains insertion order.
- **TreeSet** – keeps elements sorted.

Example:

```
Set<String> names = new HashSet<>();
names.add("Alice");
names.add("Alice");
System.out.println(names.size()); // 1
```

16. What is the difference between ArrayList and Vector?

Both **ArrayList** and **Vector** are dynamic arrays in Java that grow as you add elements. However, there are some key differences in how they handle synchronization, growth, and performance.

- **Synchronization:**
Vector is synchronized — meaning only one thread can access it at a time. This makes it thread-safe but slower.
ArrayList is **not synchronized**, which makes it faster but not safe for concurrent modification.
- **Growth:**
When a **Vector** needs to grow, it doubles its capacity (100% increase).
ArrayList grows by 50% of its size.
- **Legacy:**
Vector is a legacy class that was part of older Java versions. Today, developers prefer **ArrayList** for single-threaded applications and use concurrent collections (like **CopyOnWriteArrayList**) for multi-threaded ones.

Example:

```
ArrayList<String> arrayList = new ArrayList<>();
arrayList.add("One");
arrayList.add("Two");

Vector<String> vector = new Vector<>();
vector.add("One");
vector.add("Two");
```

In modern practice, you'd almost always use `ArrayList` unless you explicitly need thread safety — and even then, you'd likely wrap it using `Collections.synchronizedList()` or use `CopyOnWriteArrayList`.

17. What is the difference between `HashMap` and `Hashtable`?

`HashMap` and `Hashtable` both store key-value pairs but differ in synchronization and null-handling.

- **Synchronization:**
`Hashtable` is synchronized, so it's thread-safe but slower.
`HashMap` is not synchronized but faster for single-threaded use.
- **Nulls:**
`Hashtable` does **not** allow `null` keys or values.
`HashMap` allows one `null` key and multiple `null` values.
- **Legacy:**
`Hashtable` is a legacy class, replaced by `HashMap` and `ConcurrentHashMap` for modern use.

Example:

```
Hashtable<Integer, String> table = new Hashtable<>();  
table.put(1, "Apple"); // Works fine  
// table.put(null, "Banana"); // Throws NullPointerException  
  
HashMap<Integer, String> map = new HashMap<>();  
map.put(null, "Banana"); // Works fine
```

In modern Java, `Hashtable` is rarely used. `HashMap` (or `ConcurrentHashMap` for threads) is the preferred choice.

18. What are `Comparable` and `Comparator` interfaces? How are they used?

Both are used to define sorting rules for objects, but they serve slightly different purposes.

- **Comparable:** defines the **natural order** of objects.
The class itself implements the interface and defines the `compareTo()` method.
- **Comparator:** defines a **custom order** outside the class, useful when you want to sort in different ways.

Example using Comparable:

```
class Student implements Comparable<Student> {
    int id;
    String name;

    Student(int id, String name) { this.id = id; this.name = name; }

    public int compareTo(Student other) {
        return this.id - other.id; // Sort by ID
    }
}
```

Example using Comparator:

```
List<Student> list = new ArrayList<>();
list.add(new Student(2, "Alice"));
list.add(new Student(1, "Bob"));

list.sort(Comparator.comparing(s -> s.name)); // Sort by name
```

In short:

- Use **Comparable** when sorting is natural and consistent with the object.
 - Use **Comparator** when sorting rules might change.
-

19. What is the difference between fail-fast and fail-safe iterators?

A **fail-fast iterator** throws a [ConcurrentModificationException](#) if the collection is modified while iterating (except through the iterator's own methods).

A **fail-safe iterator** does not throw an exception — it works on a copy of the collection.

- **Fail-fast collections:** [ArrayList](#), [HashMap](#), [HashSet](#)
- **Fail-safe collections:** [ConcurrentHashMap](#), [CopyOnWriteArrayList](#)

Example (fail-fast):

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");

for (String s : list) {
    list.add("C"); // Throws ConcurrentModificationException
}
```

Example (fail-safe):

```
CopyOnWriteArrayList<String> safeList = new CopyOnWriteArrayList<>();
safeList.add("A");
for (String s : safeList) {
    safeList.add("B"); // No exception
}
```

20. What is the difference between shallow copy and deep copy in collections?

- **Shallow copy:** copies references of objects, not the actual objects. Both copies point to the same objects in memory.
- **Deep copy:** creates new objects with the same data, so changes in one do not affect the other.

Example:

```
List<String> list1 = new ArrayList<>();
list1.add("A");

List<String> shallowCopy = new ArrayList<>(list1);
list1.add("B");

System.out.println(shallowCopy); // ["A"], separate list (safe for immutables)
```

For mutable objects, deep copying may require cloning or serialization.

21. What is the difference between Enumeration and Iterator?

Enumeration is an older interface used with legacy classes like **Vector** and **Hashtable**.

Iterator is part of the modern Collections Framework and allows safer and more flexible traversal.

Feature	Enumeration	Iterator
Legacy	Yes	Modern
Remove elements	No	Yes
Methods	<code>hasMoreElements()</code> , <code>nextElement()</code>	<code>hasNext()</code> , <code>next()</code> , <code>remove()</code>

Example:

```
Vector<String> v = new Vector<>();
v.add("A");
v.add("B");
```

```
Enumeration<String> e = v.elements();
while (e.hasMoreElements()) {
    System.out.println(e.nextElement());
}
```

22. What is the difference between Iterator and Spliterator?

Spliterator (Split + Iterator) was introduced in Java 8 to support parallel iteration, especially for streams.

- **Iterator** traverses sequentially.
- **Spliterator** can divide the collection into parts for parallel processing.

Example:

```
List<String> list = List.of("A", "B", "C", "D");
Spliterator<String> sp = list.spliterator();
sp.trySplit().forEachRemaining(System.out::println);
```

This helps in parallelizing tasks, especially in large datasets.

23. What is the difference between Arrays and Collections?

- **Arrays** are fixed in size, can hold both primitives and objects.
- **Collections** are dynamic, can grow or shrink, and only hold objects.

Example:

```
int[] arr = {1, 2, 3}; // Fixed size
List<Integer> list = new ArrayList<>(List.of(1, 2, 3)); // Dynamic
list.add(4);
```

Collections also provide algorithms like sorting, searching, and synchronization, while arrays do not.

24. What is the difference between List and Set?

- **List** allows duplicates and maintains order.
- **Set** does not allow duplicates and has no guaranteed order (unless `LinkedHashSet` or `TreeSet` is used).

Example:

```
List<Integer> list = new ArrayList<>(List.of(1, 2, 2));
System.out.println(list); // [1, 2, 2]

Set<Integer> set = new HashSet<>(list);
System.out.println(set); // [1, 2]
```

Use a **List** when order and duplicates matter, and a **Set** when uniqueness is required.

25. What is the difference between Comparable and Comparator (recap simplified)?

Comparable defines a default sorting order within the class.

Comparator defines external sorting logic you can apply dynamically.

26. What is the difference between HashSet and LinkedHashSet?

Both prevent duplicates, but:

- **HashSet** is unordered.
- **LinkedHashSet** preserves the insertion order using a linked list.

Example:

```
Set<String> set1 = new HashSet<>(List.of("B", "A", "C"));
System.out.println(set1); // Random order

Set<String> set2 = new LinkedHashSet<>(List.of("B", "A", "C"));
System.out.println(set2); // [B, A, C]
```

Use **LinkedHashSet** when you care about the order of insertion.

27. What is the difference between Collection and Collections?

- **Collection** (singular) is an interface.
- **Collections** (plural) is a utility class with static helper methods.

Example:

```
List<Integer> list = new ArrayList<>(List.of(3, 1, 2));  
Collections.sort(list);  
Collections.reverse(list);
```

28. How can you make a collection read-only?

Use the `Collections.unmodifiable` methods.

Example:

```
List<String> list = new ArrayList<>(List.of("A", "B"));  
List<String> readOnly = Collections.unmodifiableList(list);  
// readOnly.add("C"); // Throws UnsupportedOperationException
```

This is useful for ensuring data safety when sharing collections.

29. What is the difference between deep cloning and shallow cloning in HashMap?

When cloning a `HashMap`:

- **Shallow clone** copies only references of values.
- **Deep clone** copies actual value objects too.

Example:

```
HashMap<Integer, Student> map1 = new HashMap<>();  
map1.put(1, new Student(1, "John"));  
  
HashMap<Integer, Student> shallow = (HashMap<Integer, Student>) map1.clone();
```

If you modify the `Student` object in one map, it affects both because both point to the same reference.

30. What is the difference between Iterator and for-each loop?

The **for-each loop** is a simpler way to use an iterator. However, you can't remove elements safely during iteration using it.

Example:

```
for (String s : list) {  
    // Can't remove directly  
}  
  
Iterator<String> it = list.iterator();  
while (it.hasNext()) {  
    if (it.next().equals("A")) it.remove(); // Safe  
}
```

31. What is the difference between HashMap and ConcurrentHashMap (recap simplified)?

ConcurrentHashMap allows multiple threads to access it safely without locking the entire map, while **HashMap** is not thread-safe and may lead to data corruption if used concurrently.

32. How does TreeMap maintain order?

TreeMap maintains a **sorted order** of keys using a **Red-Black tree**. You can define a custom comparator if you want different ordering.

Example:

```
TreeMap<Integer, String> map = new TreeMap<>();  
map.put(3, "C");  
map.put(1, "A");  
map.put(2, "B");  
System.out.println(map); // {1=A, 2=B, 3=C}
```

33. How do you sort a List of custom objects?

Use either **Comparable** or **Comparator**.

Example:

```
class Employee {
    int id; String name;
    Employee(int id, String name) { this.id = id; this.name = name; }
}

List<Employee> list = new ArrayList<>(List.of(new Employee(2, "Bob"), new Employee(1, "Alice")));
list.sort(Comparator.comparing(e -> e.name));
```

34. What is the difference between HashMap and LinkedHashMap?

HashMap is unordered.

LinkedHashMap preserves the order of insertion (or access order, if configured).

35. How can you remove duplicate elements from a List?

Convert the list to a **Set**.

Example:

```
List<Integer> list = List.of(1, 2, 2, 3);
List<Integer> unique = new ArrayList<>(new HashSet<>(list));
System.out.println(unique);
```

36. How can you synchronize a Collection manually?

Wrap it using **Collections.synchronizedList**, **synchronizedSet**, or **synchronizedMap**.

Example:

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
```

37. How does CopyOnWriteArrayList work?

CopyOnWriteArrayList creates a **new copy** of the list whenever it is modified (add, set, remove). This ensures thread safety without locking for reads.

It's ideal for read-heavy, write-light use cases — for example, caching or listener lists.

Example:

```
CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();  
list.add("A");  
for (String s : list) {  
    list.add("B"); // Safe, as new copy is created  
}
```

Java Concurrency

1. Why **CompletableFuture** over **Future**?

Concept (short):

Future represents a pending result of an asynchronous computation. **CompletableFuture** builds on **Future** and adds non-blocking composition, callbacks, exception handling, and easy chaining of async steps.

Explanation and details:

Future is useful when you submit a task to an **ExecutorService** and want to retrieve the result later. But **Future** is fairly limited:

- You usually block with `future.get()` to wait for the result (blocking thread).
- No easy way to chain async tasks or run continuations when the result arrives.
- Exception handling and combining multiple futures is awkward.

CompletableFuture solves these problems:

- It supports **non-blocking callbacks**: `thenApply`, `thenAccept`, `thenCompose`, `thenCombine`.
- You can **compose** futures (chain one async operation after another) using `thenCompose`.
- You can **combine** independent futures (`thenCombine`, `allOf`, `anyOf`).
- It has built-in **exception handling** (`exceptionally`, `handle`).
- It can be **completed manually** from external code (useful for event-driven or callback adapters).
- Works well with functional style and `CompletableFuture.supplyAsync`.

Analogy:

If a **Future** is a letter sitting in a mailbox, **CompletableFuture** is a smart mail system that lets you register a callback to be called when the letter arrives, send the letter to another department for processing, or combine two letters into a single report — without blocking a worker waiting by the mailbox.

Code example:

```
// Using Future (blocking)
ExecutorService ex = Executors.newFixedThreadPool(2);
Future<Integer> f = ex.submit(() -> {
    Thread.sleep(500);
    return 42;
});
Integer result = f.get(); // blocks

// Using CompletableFuture (non-blocking, composable)
```

```
CompletableFuture<Integer> cf = CompletableFuture.supplyAsync(() -> {
    sleep(500);
    return 42;
}, ex);

cf.thenApply(i -> i * 2)
    .thenAccept(i -> System.out.println("Result: " + i))
    .exceptionally(exn -> { exn.printStackTrace(); return null; });

// combine two futures
CompletableFuture<Integer> a = CompletableFuture.supplyAsync(() -> 2);
CompletableFuture<Integer> b = CompletableFuture.supplyAsync(() -> 3);
a.thenCombine(b, Integer::sum).thenAccept(sum -> System.out.println("Sum: " + sum));
```

When to use which:

Use **Future** for very simple fire-and-get tasks. Use **CompletableFuture** when you need composition, non-blocking pipelines, better error handling, or integration with reactive flows.

2. What is the difference between concurrency and parallelism?

Concept (short):

Concurrency is about managing multiple tasks in overlapping time periods (interleaving), while **parallelism** is actually performing multiple tasks at the same time using multiple CPU cores.

Explanation and details:

- Concurrency is a design property — systems structured to handle multiple tasks at once (e.g., UI thread that handles events while background tasks run). Concurrency can be achieved on a single core by time-slicing (switching between tasks).
- Parallelism is about performance — running multiple tasks simultaneously to speed up computation, which requires multiple CPU cores or processors.

Analogy:

Imagine a single cook juggling several dishes on one stove (concurrency — tasks interleaved). Two cooks working side-by-side, each on a dish, is parallelism.

Why it matters in Java:

- **Thread** and **ExecutorService** give you concurrency.
- To get **real parallelism**, the JVM needs multiple hardware cores and the code must be structured to exploit them (parallel streams, ForkJoinPool, multiple threads doing CPU-bound work).
- I/O-bound concurrent tasks (network calls) benefit from concurrency even without parallelism since threads spend time waiting.

Implications:

- For CPU-bound tasks, parallelism gives speedup (subject to Amdahl's Law).
 - For I/O-bound tasks, concurrency (asynchronous IO or many threads) helps throughput without necessarily needing many cores.
-

3. What are possible ways in Java to make your API call asynchronous?

Concept (short):

You can make API calls async using threads, thread pools, **CompletableFuture**, asynchronous HTTP clients, reactive frameworks, or virtual threads.

Detailed options with examples:

ExecutorService with **Callable** / **Future**

Submit tasks to a pool and get a **Future** that completes later.

```
ExecutorService ex = Executors.newFixedThreadPool(10);
Future<Response> f = ex.submit(() -> httpClient.callSync());
// later: f.get();
```

CompletableFuture.supplyAsync

Non-blocking composition and callbacks.

```
CompletableFuture.supplyAsync(() -> httpClient.callSync(), ex)
    .thenAccept(response -> handle(response));
```

Asynchronous HTTP client (e.g., Java **HttpClient** in java.net.http or Netty)

Use the async API that returns **CompletableFuture<HttpResponse>**.

```
HttpClient client = HttpClient.newHttpClient();
client.sendAsync(request, BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

Reactive programming (Project Reactor / RxJava)

Return **Mono/Flux** (or **Observable**) for asynchronous, back-pressured streams.

```
Mono<String> mono = webClient.get().uri("/api").retrieve().bodyToMono(String.class);
mono.subscribe(System.out::println);
```

Callback-based approach

Provide a callback interface to be invoked when response arrives (older style).

Virtual threads (Project Loom)

Launch lots of lightweight threads and perform blocking calls; the runtime makes this cheap.

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {  
    executor.submit(() -> httpClient.callSync());  
}
```

When to use which:

- For simple tasks and composition: **CompletableFuture**.
 - For high-throughput non-blocking IO: reactive frameworks or async HTTP clients.
 - For simple scaling and blocking code: virtual threads simplify code while staying synchronous-looking.
-

4. What is the difference between **synchronized** and **volatile** in Java?

Concept (short):

synchronized ensures mutual exclusion and establishes happens-before relationships (both read and write visibility + atomicity for a block/method). **volatile** ensures visibility of writes to a variable across threads but does **not** provide atomic compound actions.

Detailed explanation:

- **volatile** variable guarantees that reads always see the latest write from any thread (visibility). It also prevents certain reordering optimizations. But **volatile** does **not** make compound operations atomic. For example, **volatile int x; x++;** is not atomic.
- **synchronized** (method or block) provides mutual exclusion — only one thread may hold the monitor at a time — and also provides visibility: when a thread releases the monitor, changes made inside the synchronized block are visible to the next thread that acquires the same monitor.

Examples:

```
// volatile example (visibility only)  
volatile boolean flag = false;  
Thread t1 = new Thread(() -> { while (!flag) {} System.out.println("seen"); });  
Thread t2 = new Thread(() -> { flag = true; });  
t1.start(); t2.start();  
  
// synchronized example (atomicity + visibility)
```

```
int counter = 0;
synchronized (this) { counter++; } // safely increment
```

When to use:

- Use **volatile** for simple flags or single-variable state where only visibility is required.
- Use **synchronized** when you need atomicity for compound operations (check-then-act or increments).

Pitfalls:

- **volatile** is not a substitute for locks when you need mutually exclusive access.
- Overusing **synchronized** can reduce concurrency by serializing operations.

5. What are the differences between **synchronized** and **Lock** in Java?

Concept (short):

Both provide mutual exclusion, but **Lock** (from `java.util.concurrent.locks`) offers more flexibility and features compared to Java's built-in **synchronized** (monitor): timed waits, interruptible waits, fairness settings, `tryLock`, and multiple condition variables.

Detailed explanation:

- **synchronized** is simple and built-in: **synchronized(this) { ... }** or **synchronized method**. It automatically releases lock on exit (including exceptions).
- **ReentrantLock** (an implementation of **Lock**) allows:
 - **lock()**, **unlock()** with explicit control.
 - **tryLock()** to try acquiring without blocking.
 - **tryLock(long timeout, TimeUnit unit)** to wait a limited time.
 - **lockInterruptibly()** to allow interruption while waiting.
 - **newCondition()** to obtain a **Condition** (multiple wait-sets vs single monitor wait-set in **synchronized**).
 - optional fairness policy (first-come-first-served).

Example using **ReentrantLock:**

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try {
    // critical section
} finally {
    lock.unlock();
}
```

When to prefer **Lock**:

- Need timed or interruptible lock acquisition.
- Need multiple condition variables (**Condition**) for complex coordination.
- Want a non-blocking attempt to acquire lock (**tryLock**).
- Want fairness policy (but note fairness may reduce throughput).

When to prefer **synchronized**:

- Simpler, less error-prone for basic mutual exclusion.
- JVM optimizations (biased locking, etc.) usually make **synchronized** very efficient.

Pitfalls:

- With **Lock**, you must ensure **unlock()** in **finally** — it's easy to forget and cause deadlocks.
 - Avoid mixing **synchronized** and the same **Lock** on the same data — stick to one mechanism for clarity.
-

6. **ExecutorService** runs tasks in Async or in Parallel?

Concept (short):

ExecutorService runs tasks asynchronously (they may start later) and may run them in parallel depending on the thread pool configuration and hardware. Asynchrony \neq parallelism; **ExecutorService** provides concurrency; real parallelism requires multiple threads and multiple CPU cores.

Detailed explanation:

- Submitting a **Runnable/Callable** to an **ExecutorService** schedules the task to be run asynchronously — the submitting thread does not block unless it calls **get()** on a **Future**.
- If the **ExecutorService** has multiple worker threads (e.g., **newFixedThreadPool(n)**), multiple tasks can run **in parallel** if the JVM has multiple cores.
- If you use a single-threaded executor (**newSingleThreadExecutor()**), tasks are asynchronous relative to the submitter but **not parallel** with each other (they run serially on the single worker thread).

Examples:

```
ExecutorService pool = Executors.newFixedThreadPool(4); // up to 4 tasks can run concurrently
pool.submit(() -> doWork()); // asynchronous, possibly parallel
```

Summary:

ExecutorService always gives you **asynchronous execution**; whether it provides **parallel execution** depends on the pool size and available CPU cores.

7. How does the **Future** interface work, and what methods does it provide?

Concept (short):

Future<V> represents the result of an asynchronous computation. It provides methods to check if computation is done, to wait for result, or cancel it.

Methods and explanation:

- **boolean cancel(boolean mayInterruptIfRunning)** — attempts to cancel execution.
- **boolean isCancelled()** — true if task was cancelled.
- **boolean isDone()** — true if computation completed or was cancelled.
- **V get()** — blocks until computation completes and returns result (throws exceptions).
- **V get(long timeout, TimeUnit unit)** — blocks up to timeout and throws **TimeoutException** if not done.

Example:

```
ExecutorService ex = Executors.newSingleThreadExecutor();
Future<Integer> f = ex.submit(() -> {
    Thread.sleep(1000);
    return 10;
});

if (!f.isDone()) System.out.println("Still running");
Integer val = f.get(); // blocks
ex.shutdown();
```

Limitations:

- **get()** is blocking — it ties up a thread if used in a blocking manner.
 - No composition or callbacks — **CompletableFuture** provides a richer API.
-

8. What is the difference between **Runnable** and **Callable**?

Concept (short):

Runnable is a task that does not return a result and cannot throw checked exceptions; **Callable<V>** returns a result and may throw checked exceptions.

Details:

- **Runnable**: `void run()` – used with `Thread` or `ExecutorService.submit(Runnable)`.
- **Callable<V>**: `V call()` – submitted to `ExecutorService` and returns a `Future<V>`.

Examples:

```
Runnable r = () -> System.out.println("run");  
Callable<Integer> c = () -> { return expensiveCalculation(); };
```

```
ExecutorService ex = Executors.newFixedThreadPool(2);  
ex.submit(r); // returns Future<?>; get() returns null on success  
Future<Integer> f = ex.submit(c); // get returns Integer result
```

When to use which:

Use **Runnable** for side-effect tasks that don't produce a result. Use **Callable** when you need a computed result or to propagate checked exceptions.

9. How do you prevent deadlocks in Java?

Concept (short):

Deadlocks occur when two or more threads hold locks and wait for locks held by each other. Prevent by ordering locks, using timeouts, minimizing synchronized regions, or using **tryLock**.

Detailed strategies:

Lock ordering: Always acquire locks in the same global order across threads. If every thread locks resources A then B, deadlocks cannot occur.

```
// Acquire locks in order: lock1 then lock2  
  
synchronized (lock1) {  
    synchronized (lock2) { ... }  
}
```

TryLock with timeout: Use `ReentrantLock.tryLock(timeout, unit)` to avoid waiting forever; handle failure to acquire by backing off.

```
if (lock.tryLock(1, TimeUnit.SECONDS)) {  
    try { ... } finally { lock.unlock(); }  
} else {  
    // fallback or retry  
}
```

1. **Lock striping / reduce lock granularity:** Avoid holding locks across blocking IO or long computations. Keep critical sections small.
2. **Avoid nested locks when possible:** Use concurrent collections (ConcurrentHashMap) to avoid explicit locks.
3. **Use a single lock for multiple resources:** Simpler but reduces concurrency.
4. **Deadlock detection and recovery:** In advanced systems, detect circular waits and recover by killing or rolling back tasks.

Example of a potential deadlock:

Thread A: locks **a** then **b**. Thread B: locks **b** then **a**. If they interleave, deadlock occurs. Solve by locking both in consistent order.

Pitfall:

Using different lock ordering in different methods or third-party libraries can create deadlocks; document and enforce lock order.

10. Describe the role of **CountDownLatch** in synchronization.

Concept (short):

CountDownLatch is a one-shot synchronization aid that allows one or more threads to wait until a set of operations in other threads complete.

Detailed explanation:

CountDownLatch is initialized with a count (e.g., number of worker threads). Each worker calls **countDown()** when done. Threads waiting for all workers call **await()** which blocks until the count reaches zero.

Common uses:

- Main thread waits for several services to finish startup.
- Testing: wait until background tasks complete before asserting results.
- Starting line for race tests: use a latch with count 1 to start all threads simultaneously (see **CyclicBarrier** for more flexibility).

Example:

```
CountDownLatch latch = new CountDownLatch(3);
for (int i = 0; i < 3; i++) {
    new Thread() -> {
        doWork();
        latch.countDown();
    }.start();
}
latch.await(); // waits until all three have called countDown
```

Important:

Latch is one-time use — once reaches zero it cannot be reset. For repeated cycles, use [CyclicBarrier](#) or [Semaphore](#).

11. What is the purpose of [CyclicBarrier](#), and how is it used?

Concept (short):

[CyclicBarrier](#) is used to make a group of threads wait for each other at a common barrier point; unlike [CountDownLatch](#), it can be reused (cyclic).

Detailed explanation:

You initialize a [CyclicBarrier](#) with a number of participating threads and optionally a barrier action (Runnable) that runs once all parties reach the barrier. Each thread calls [await\(\)](#) and blocks until the required number have called it; then all are released simultaneously.

Use cases:

- Coordinating phases in parallel algorithms (e.g., iterative solvers).
- Synchronize start of a next step after all threads finish current step.

Example:

```
int parties = 3;
CyclicBarrier barrier = new CyclicBarrier(parties, () -> System.out.println("All reached barrier"));

for (int i = 0; i < parties; i++) {
    new Thread() -> {
        doPhase1();
        barrier.await(); // waits till all arrive
        doPhase2();
    }.start();
}
```

Notes:

- If a thread is interrupted or times out while waiting, the barrier is broken.
 - **CyclicBarrier** is reusable across iterations — good for looped parallel phases.
-

12. Explain the **ReentrantLock** class and its features compared to using **synchronized** blocks.

Concept (short):

ReentrantLock is an explicit lock offering the usual mutual exclusion plus advanced features: **tryLock**, timed lock, interruptible lock, fairness policy, and multiple **Condition** objects.

Features and examples:

- **Reentrancy**: same thread can lock multiple times; must unlock the same number of times.
- **tryLock()**: attempt to acquire without blocking.
- **tryLock(timeout, unit)**: wait up to timeout.
- **lockInterruptibly()**: allow interruption while waiting.
- **Fairness**: can be created with a fairness policy to reduce starvation.
- **Condition objects**: multiple wait-sets associated with the same lock for fine-grained wait/notify semantics.

```
ReentrantLock lock = new ReentrantLock(true); // fair
lock.lock();
try {
    // critical section
} finally {
    lock.unlock();
}
```

Advantages over **synchronized**:

- More flexible control (tryLock, timed waits).
- Multiple **Conditions** instead of a single monitor wait-set.
- Explicit unlocking (good for complex flows).

Disadvantages:

- More boilerplate; forgot **unlock()** -> risk of deadlock.
 - **synchronized** is simpler and has JVM optimizations.
-

13. What is the **ForkJoinPool**, and how does it differ from **ThreadPoolExecutor**?

Concept (short):

ForkJoinPool is a thread pool optimized for divide-and-conquer (fork-join) tasks that can be recursively split into subtasks. It uses work-stealing to balance load across threads. **ThreadPoolExecutor** is a general-purpose pool for independent tasks.

Detailed explanation:

- **ForkJoinPool** works best with **ForkJoinTask** subclasses (like **RecursiveTask** and **RecursiveAction**) for algorithms that split tasks into subtasks.
- **Work-stealing**: idle worker threads steal tasks from busy workers' dequeues, improving throughput and load balance.
- **ThreadPoolExecutor** handles independent tasks submitted by clients; no built-in work-stealing (though Java 8+ introduced **Executors.newWorkStealingPool()** backed by **ForkJoinPool**).

When to use which:

- Use **ForkJoinPool** for recursive, CPU-bound tasks (parallel sort, parallel sum).
- Use **ThreadPoolExecutor** for handling incoming independent requests, I/O-bound tasks, or fixed pools with bounded queues.

Example (ForkJoin recursive sum):

```
class SumTask extends RecursiveTask<Long> {
    private final long[] arr; int lo, hi;
    static final int THRESHOLD = 1000;
    SumTask(long[] a, int lo, int hi) { this.arr = a; this.lo = lo; this.hi = hi; }
    protected Long compute() {
        if (hi - lo <= THRESHOLD) {
            long s=0; for(int i=lo;i<hi;i++) s+=arr[i]; return s;
        }
        int mid = (lo+hi)/2;
        SumTask left = new SumTask(arr, lo, mid);
        SumTask right = new SumTask(arr, mid, hi);
        left.fork();
        long r = right.compute();
        long l = left.join();
        return l + r;
    }
}
```

```
ForkJoinPool pool = new ForkJoinPool();
long total = pool.invoke(new SumTask(array, 0, array.length));
```

14. How does the **ForkJoinPool** work, and what types of tasks is it designed for?

Concept (short):

ForkJoinPool is designed for **divide-and-conquer** problems. It splits a big task into smaller subtasks, runs them in parallel, and combines results. It uses work-stealing for efficiency.

Detailed working:

- A task forks subtasks using `fork()` and waits for them using `join()`.
- Each worker thread maintains a deque (double-ended queue). Workers push/pop tasks from the head; thieves steal from the tail (older tasks) to reduce contention.
- This design keeps locality and reduces contention.

Types of tasks:

- CPU-bound, recursive tasks that can be broken into many small independent tasks (e.g., parallel sort, matrix operations, recursive search).
- Not ideal for tasks blocking on I/O (unless using `ManagedBlocker`) — blocking reduces available worker threads.

Best practices:

- Choose a reasonable threshold to avoid too much overhead (task splitting cost vs work size).
- Avoid blocking operations inside tasks; if blocking is necessary, use `ForkJoinPool.ManagedBlocker`.

15. What is the **ThreadLocal** class, and how does it work?

Concept (short):

`ThreadLocal<T>` provides a separate instance of a variable for each thread — each thread sees its own value.

Detailed explanation:

- `ThreadLocal` maintains a map **per thread** (backed by `Thread.threadLocals`) keyed by the `ThreadLocal` instance.
- When a thread calls `get()` for the first time, `initialValue()` may be used to create a value. Subsequent `get()` returns the thread-specific value.

- Useful for per-thread state that would otherwise be passed through method parameters (e.g., user context, non-thread-safe objects like `SimpleDateFormat`).

Example:

```
private static final ThreadLocal<SimpleDateFormat> TL = ThreadLocal.withInitial(() -> new
SimpleDateFormat("yyyy-MM-dd"));

String s = TL.get().format(new Date());
```

Important notes:

- Clean up using `remove()` especially in thread pools where threads are reused to avoid stale data causing memory leaks.
 - Avoid overuse; prefer immutable data passed explicitly when possible.
-

16. Explain how you can use `CompletableFuture` to handle exceptions.

Concept (short):

`CompletableFuture` provides methods like `exceptionally`, `handle`, and `whenComplete` to react to exceptions and recover from failures without blocking.

Detailed usage:

- `exceptionally(Function<Throwable, T>)` returns a fallback value if computation throws.
- `handle(BiFunction<T, Throwable, R>)` is invoked whether result or exception occurs; it receives both result and exception.
- `whenComplete(BiConsumer<T, Throwable>)` executes a callback for side effects; it doesn't change the result unless you throw.

Examples:

```
CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> {
    if (Math.random() > 0.5) throw new RuntimeException("fail");
    return "ok";
});

cf.exceptionally(ex -> {
    System.out.println("Recovered from " + ex);
    return "default";
});
```

```
}).thenAccept(System.out::println);

// handle example (transform both success and failure)
cf.handle((res, ex) -> {
    if (ex != null) return "fallback";
    else return res.toUpperCase();
}).thenAccept(System.out::println);
```

Best practice:

Handle exceptions at a boundary of your async chain; avoid swallowing exceptions silently.

17. What is Virtual Threads in Java and how does it differ from Normal Threads?

Concept (short):

Virtual threads (Project Loom) are lightweight user-mode threads that make it cheap to create and manage large numbers of threads. Platform (OS) threads (a.k.a. platform or "carrier" threads) are heavier and limited.

Detailed explanation:

- **Platform threads** are backed by OS threads (one-to-one). Creating thousands is expensive and may exhaust OS resources.
- **Virtual threads** are scheduled by the JVM on a small pool of carrier threads. They are cheap to create, and the OS thread count remains small.
- From code perspective, virtual threads behave like regular threads — you can call blocking APIs (e.g., JDBC), and the runtime will unmount the virtual thread while blocking occurs and remount it later — enabling large concurrency with simpler imperative code.

Example:

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    for (int i=0; i<10000; i++) {
        executor.submit(() -> {
            // blocking operations are fine:
            String r = blockingHttpCall();
            System.out.println(r);
        });
    }
} // executor shutdown
```

Benefits:

- Simplifies writing async code — you can use straightforward blocking APIs but achieve scalability.
- Reduces complexity compared to callback-based or reactive code.

Limitations and considerations:

- Some older libraries may not be compatible if they rely on thread-local assumptions; workarounds exist.
 - Virtual threads are designed primarily for IO-bound tasks where traditional threads blocked most of the time.
-

18. What is structured concurrency? How does it work? Tell me some use-cases where you can use it.

Concept (short):

Structured concurrency treats a group of concurrent tasks as a unit with a clear lifecycle; it makes concurrency easier to reason about by ensuring tasks are scoped and exceptions are managed collectively.

Detailed explanation:

- Structured concurrency principles: spawn tasks within a scope; the scope waits for children to finish before exiting; exceptions in children are propagated in a controlled way.
- It avoids orphaned tasks and makes reasoning about resources easier — when you exit a method, all child tasks have completed or been cancelled.

Example pattern (pseudo-Java using structured concurrency APIs from Loom proposals):

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {  
    Future<String> a = scope.fork() -> callA();  
    Future<String> b = scope.fork() -> callB();  
    scope.join(); // wait for all or failure  
    scope.throwIfFailed(); // propagate  
    String ra = a.resultNow();  
    String rb = b.resultNow();  
}
```

Use-cases:

- Parallel calls to multiple microservices where you want to wait for all or stop on first failure.
- Any short-lived parallel computations with clear scope (web request handling, batch jobs).
- Easier error handling and cancellation compared to manually managing many futures.

Benefits:

- Makes life-cycle of threads explicit and local.
 - Simplifies cancellation and error propagation.
-

19. How can you enable parallelism in Java?

Concept (short):

Enable parallelism by using multiple threads and structures that exploit multiple CPU cores: `ExecutorService` with multiple threads, `ForkJoinPool`, parallel streams (`parallelStream()`), or by using `CompletableFuture` with a custom executor or `newWorkStealingPool`.

Methods:

- `Executors.newFixedThreadPool(n)` with $n > 1$ and CPU cores to run tasks concurrently.
- `ForkJoinPool` for divide-and-conquer tasks.
- **Parallel Streams:** `list.parallelStream().map(...).collect(...)`.
- **Work-stealing pools:** `Executors.newWorkStealingPool()`.
- `CompletableFuture` with custom `Executor` to perform many async tasks.

Important:

Parallelism yields benefit for CPU-bound tasks. For IO-bound tasks, adding more threads helps concurrency, but not raw CPU parallelism.

Example (parallel stream):

```
long sum = LongStream.range(0, 1_000_000).parallel().reduce(0, Long::sum);
```

Cautions:

- Beware shared mutable state — use thread-safe or immutable operations.
 - Parallel streams use the common `ForkJoinPool` by default — be careful in server apps; you might supply a custom pool.
-

20. Explain the Java Memory Model (JMM) and how it affects concurrency.

Concept (short):

The Java Memory Model (JMM) defines rules for visibility, ordering, and atomicity of actions across threads. It tells when writes by one thread become visible to reads in another.

Key ideas:

- **Happens-before:** a relation that, if established between two actions, guarantees visibility and ordering.
 - e.g., `synchronized` unlock → lock establishes happens-before; `volatile` write → read establishes visibility; thread start/join have happens-before rules.
- Without proper synchronization, threads may see stale values or reordered operations.
- The JMM allows compilers and CPUs to reorder instructions unless prevented by memory barriers (synchronization primitives).

Example pitfalls:

- Without `volatile` or synchronization, setting a boolean flag in one thread may not be seen by another thread due to caching/reordering.
- Double-checked locking requires `volatile` for the instance reference.

Example (safe publish using `volatile`):

```
class Holder {
    private static volatile Singleton instance;
    static Singleton get() {
        if (instance == null) {
            synchronized(Holder.class) {
                if (instance == null) instance = new Singleton();
            }
        }
        return instance;
    }
}
```

Practical guidance:

- Use high-level constructs (`synchronized`, locks, concurrent collections) rather than relying on subtle JMM behavior.
- Use `volatile` for simple flags and properly documented patterns.

21. What are atomic operations in Java (`AtomicInteger`, CAS)? Why are they faster than synchronization?

Concept (short):

Atomic classes (e.g., `AtomicInteger`) provide lock-free, thread-safe operations using hardware-supported compare-and-swap (CAS). They are often faster than locking because they avoid monitor contention and context switches.

Detailed explanation:

- CAS operates by comparing a memory location to an expected value and swapping it with a new value if they are equal — this is done atomically by the CPU.
- `AtomicInteger.incrementAndGet()` uses CAS loop internally to implement atomic increment.

- Lock-free algorithms give better throughput under low to moderate contention. Under extremely high contention, CAS loops may spin and cost CPU cycles.

Example:

```
AtomicInteger counter = new AtomicInteger(0);  
counter.incrementAndGet(); // atomic
```

Why faster than `synchronized`:

- `synchronized` involves acquiring and releasing a monitor which can cause OS-level blocking and context switching when contention occurs.
- CAS avoids blocking — threads spin and retry, which is cheaper when contention is low and operations are short.

When to use:

- Use atomic variables for simple counters, states, or references.
- Use higher-level locks for more complex invariants or multi-variable updates (unless using specialized lock-free algorithms).

22. How would you design a high-performance concurrent system in Java (e.g., using `CompletableFuture`, parallel streams, or reactive programming with Project Reactor)?

Concept (short):

Designing a high-performance concurrent system requires choosing the right concurrency model for the workload (CPU-bound vs I/O-bound), minimizing contention, using non-blocking or asynchronous I/O where appropriate, and correctly sizing thread pools.

Design principles and steps:

1. **Understand workload:**
 - CPU-bound: use parallelism (`ForkJoinPool`, parallel streams) distributing tasks across cores.
 - I/O-bound: use asynchronous I/O or many light-weight threads (virtual threads) to keep CPU busy.
2. **Use non-blocking IO and async APIs where practical:**
 - For network services, prefer non-blocking clients or reactive stacks (Netty + Reactor) to handle many connections with few threads.
3. **Use appropriate thread pools:**
 - For CPU-bound, set pool size near number of CPU cores.
 - For I/O-bound, larger pools or virtual threads are acceptable.
 - Use bounded queues to prevent OOM if producers are faster than consumers.
4. **Avoid shared mutable state and minimize lock contention:**

- Favor immutable data, thread-local state, and partitioned data structures (sharding) to reduce synchronization.
- 5. **Use `CompletableFuture` for composing async tasks:**
 - Compose operations non-blockingly: `supplyAsync()`, `thenCompose()`, `allOf()` for parallel calls, `exceptionally()` for errors.
- 6. **Reactive streams for backpressure:**
 - If you need to handle streaming data and apply backpressure, use Project Reactor or RxJava. They allow flow control so producers don't overwhelm consumers.
- 7. **Measure and profile:**
 - Use benchmarks, thread dumps, CPU and latency profiling. Tune pool sizes and data structures accordingly.
- 8. **Graceful degradation & circuit breakers:**
 - Use timeouts, retries with backoff, bulkheads and circuit breakers to isolate failures and prevent cascading overload.
- 9. **Example architecture approaches:**

Web request with parallel service calls (`CompletableFuture`):

```
CompletableFuture<User> userF = supplyAsync(() -> callUserService(id));
CompletableFuture<Account> acctF = supplyAsync(() -> callAccountService(id));
CompletableFuture<Void> combined = userF.thenCombine(acctF, (u, a) -> combine(u, a))
    .thenAccept(result -> respond(result));
```

Reactive approach (Project Reactor):

```
Mono<User> user = webClient.get().uri("/user").retrieve().bodyToMono(User.class);
Mono<Account> account = webClient.get().uri("/account").retrieve().bodyToMono(Account.class);
Mono.zip(user, account).map(tuple -> merge(tuple.getT1(), tuple.getT2()))
    .subscribe(this::sendResponse);
```

- - 10. **Resilience and observability:**
 - Add metrics, logging, distributed tracing, and circuit breakers (Resilience4j).
 - Use health checks and graceful shutdown.
-

Good to HAVE !!

1. What is the difference between Process and Thread?

A **process** is an independent program in execution with its own memory space.

A **thread** is a lightweight sub-unit of a process that shares the same memory but can run independently.

Detailed explanation:

When you open a new application on your computer (like Chrome or IntelliJ), the operating system creates a **process**.

Each process has its own separate **address space**, meaning one process cannot directly access variables of another process (unless through special inter-process communication like sockets or shared memory).

Inside each process, there can be multiple **threads** — these are smaller execution units that share the same heap memory (objects, static variables) but have their own **stack** (method calls and local variables).

Analogy:

*Think of a **process** as a company.*

*Inside the company, different **threads** are employees working together, sharing the same office resources (heap memory).*

Each employee has their own notebook (stack memory) for temporary notes.

Employees (threads) can coordinate to complete tasks faster, but if they mess up synchronization, they might step on each other's work!

Differences summary:

Feature	Process	Thread
Memory	Each has its own memory space	Shares memory within same process
Communication	Needs IPC (Inter-Process Communication)	Easier — share objects directly
Creation cost	Heavy (needs OS resources)	Light (less memory and faster to create)
Failure	Crash of one process doesn't affect others	A thread crash can affect the whole process
Use case	Isolated programs	Parallel tasks within one program

Example (in Java terms):

When you run your Java application, the JVM is a **process**.
Inside it, you might create multiple **threads** to handle tasks like:

```
public class Example {  
    public static void main(String[] args) {  
        Thread t1 = new Thread() -> System.out.println("Task 1");  
        Thread t2 = new Thread() -> System.out.println("Task 2");  
        t1.start();  
        t2.start();  
    }  
}
```

Here, the JVM process runs multiple threads concurrently — all sharing the same heap (objects, classes).

2. Explain the difference between synchronized method vs synchronized block in Java.

Concept (short):

Both are used to ensure **mutual exclusion** (only one thread executes a section at a time), but they differ in **scope** and **flexibility**.

Synchronized method:

- When you declare a method as **synchronized**, the entire method is locked.
For **instance methods**, the lock is on the **current object** (**this**).
- For **static methods**, the lock is on the **class object** (**ClassName.class**).

```
public synchronized void increment() {  
    count++;  
}
```

This means **only one thread** can execute **increment()** on a particular instance at a time.

Synchronized block:

- Gives **fine-grained control**: you can lock only a **specific part** of code or use a **custom lock object**.
- Reduces contention by keeping lock scope small.

```
public void increment() {  
    synchronized (this) {  
        count++;  
    }  
}
```

Or using a specific lock object:

```
private final Object lock = new Object();  
  
public void increment() {  
    synchronized (lock) {  
        count++;  
    }  
}
```

Differences summary:

Feature	Synchronized Method	Synchronized Block
Scope	Entire method	Specific code section
Lock	Implicitly <code>this</code> or class object	Can choose custom lock
Performance	May block threads longer	More efficient (less time under lock)
Flexibility	Limited	Flexible (choose what to lock)

Analogy:

If a “synchronized method” is like locking an entire room before doing a small task, a “synchronized block” is like locking just the drawer you need — faster and safer.

Best practice:

Always synchronize only the **critical section** (the part accessing shared mutable data). This maximizes concurrency.

3. What are Deadlock, Livelock, and Starvation? Give examples.

Concept (short):

- **Deadlock** – Threads are waiting forever for each other’s locks.
 - **Livelock** – Threads keep reacting to each other’s changes, but no progress is made.
 - **Starvation** – A thread never gets CPU time or resource due to unfair scheduling.
-

1. Deadlock

Definition: Two or more threads are blocked forever, each waiting for a resource held by another.

Example:

```
class DeadlockExample {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    void method1() {
        synchronized (lock1) {
            System.out.println("Thread 1: Holding lock1...");
            sleep(100);
            synchronized (lock2) {
                System.out.println("Thread 1: Got lock2!");
            }
        }
    }

    void method2() {
        synchronized (lock2) {
            System.out.println("Thread 2: Holding lock2...");
            sleep(100);
            synchronized (lock1) {
                System.out.println("Thread 2: Got lock1!");
            }
        }
    }

    void sleep(int ms) { try { Thread.sleep(ms); } catch (InterruptedException e) {} }

    public static void main(String[] args) {
        DeadlockExample ex = new DeadlockExample();
        new Thread(ex::method1).start();
        new Thread(ex::method2).start();
    }
}
```

Both threads hold one lock and wait for the other — **deadlock**.

Prevention:

Always acquire locks in the **same order**.

2. Livelock

Definition: Threads are not blocked but keep changing state in response to each other, making no progress.

Analogy:

Two people walking toward each other in a hallway both step aside in the same direction repeatedly — “after you”, “no, after you” — forever.

Example (conceptual):

Two threads keep retrying because they detect the other one also changed something, leading to continuous retries.

Prevention:

Add random delays or backoff to break symmetry.

3. Starvation

Definition: A thread never gets a chance to run because others always take the CPU or locks first.

Example:

If one thread always acquires a lock because of unfair scheduling, the other is starved.

```
ReentrantLock lock = new ReentrantLock(false); // unfair
```

If a high-priority thread constantly acquires CPU, a low-priority one may starve.

Solution:

Use **fair locks** (`new ReentrantLock(true)`) or proper scheduling.

Summary Table:

Summary Table:

Problem	Cause	Symptom	Solution
Deadlock	Circular lock waiting	Threads blocked forever	Lock ordering, tryLock
Livelock	Threads react endlessly	Active but no progress	Random delay/backoff
Starvation	Unfair scheduling	Some threads never run	Fair locks or priorities

4. Difference between `wait()`, `sleep()`, and `join()` in Java multithreading.

Concept (short):

- `wait()` → Releases lock, waits for notify signal (used for thread communication).
- `sleep()` → Pauses thread for some time but **does not release** lock.
- `join()` → Makes one thread wait until another thread finishes.

`sleep(long millis)`

- Static method in `Thread` class.
- Temporarily pauses thread execution.
- Does **not** release any locks.

```
synchronized(this) {  
    Thread.sleep(1000); // lock still held  
}
```

`wait()`

- Called on an object, not on a thread.
- Thread must own the object's lock (`synchronized`).
- Releases the lock and waits for `notify()` or `notifyAll()`.

```
synchronized(lock) {  
    lock.wait(); // releases lock  
}
```

Example of wait/notify:

```
synchronized(lock) {  
    while(!ready) lock.wait();  
}
```

Another thread:

```
synchronized(lock) {  
    ready = true;  
    lock.notify();  
}
```

join()

- Used to make one thread wait until another finishes execution.

```
Thread t = new Thread() -> work();  
t.start();  
t.join(); // main thread waits for t to finish
```

Summary Table:

Method	Where used	Releases lock	Purpose
wait()	Object class	✓ Yes	Thread communication
sleep()	Thread class	✗ No	Pause execution for time
join()	Thread class	✗ No	Wait for another thread to finish

Analogy:

- **sleep()** is like saying “I’m resting for 1 second.”
 - **wait()** is like saying “I’ll stop until someone wakes me up.”
 - **join()** is like saying “I’ll wait for my friend to finish their work before continuing.”
-

5. How does the **volatile** keyword work? When should you use it?

Concept (short):

volatile ensures **visibility** of variable updates across threads — a read always sees the latest write. It also prevents certain instruction reorderings.

Detailed explanation:

- Normally, threads may cache variables locally (in registers or CPU cache). Without synchronization, one thread's update may not be visible to others.
- Declaring a variable **volatile** tells the JVM:
 1. Always read/write from **main memory**, not from thread-local cache.
 2. Prevent certain reordering optimizations around volatile reads/writes.

```
volatile boolean flag = false;
```

```
Thread t1 = new Thread(() -> {  
    while (!flag) {} // loops until flag becomes true  
});  
  
Thread t2 = new Thread(() -> flag = true);
```

Without **volatile**, thread **t1** might never see **flag = true**.

When to use:

- For simple state flags or status indicators.
- For **one-writer-many-reader** scenarios where atomicity isn't required.
- Do **not** use for compound operations (like **count++**).

Not safe example:

```
volatile int count = 0;  
count++; // NOT atomic
```

Use `AtomicInteger` for atomic increments.

In short:

`volatile` fixes visibility issues, not atomicity.

6. Difference between `ThreadPoolExecutor` and `ForkJoinPool`.

Concept (short):

Both are thread pools, but they serve different purposes.

- `ThreadPoolExecutor` → General-purpose pool for independent tasks.
 - `ForkJoinPool` → Designed for divide-and-conquer tasks using work-stealing.
-

Detailed explanation:

ThreadPoolExecutor

- Executes independent tasks.
- Configurable: core threads, max threads, queue, rejection policy.
- Tasks submitted via `submit()` or `execute()`.
- Each worker thread pulls tasks from a shared queue.

```
ExecutorService pool = Executors.newFixedThreadPool(4);  
pool.submit(() -> work());
```

Used in web servers, background processing, etc.

ForkJoinPool

- Executes tasks that can **split** into subtasks (RecursiveTask / RecursiveAction).
- Uses **work-stealing**: idle threads steal work from busy ones.
- Built for parallel computation and divide-and-conquer.

```
ForkJoinPool pool = new ForkJoinPool();  
pool.invoke(new SumTask(array, 0, array.length));
```

Key differences:

Feature	ThreadPoolExecutor	ForkJoinPool
Task Type	Independent	Divide & Conquer
Queue	Shared BlockingQueue	Per-thread deque (work-stealing)
Best for	I/O or independent tasks	CPU-bound recursive tasks
Subtask handling	Manual	Recursive (<code>fork()</code> / <code>join()</code>)
Example	Web requests	Parallel array sum

Analogy:

`ThreadPoolExecutor` is like a factory line where workers pull jobs from a common queue.

`ForkJoinPool` is like a team of workers who can split their tasks into subtasks and help each other when idle.

7. How would you implement a Producer-Consumer problem using `BlockingQueue`?

The **Producer-Consumer problem** models a scenario where producers generate data and consumers process it.

`BlockingQueue` makes this easy by handling synchronization internally — producers wait when the queue is full; consumers wait when it's empty.

In older Java versions, producer-consumer required manual `wait()/notify()` logic.
With `BlockingQueue`, all synchronization is handled automatically.

Example:

```
import java.util.concurrent.*;

public class ProducerConsumer {
    private static final BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);

    static class Producer implements Runnable {
        public void run() {
            try {
                for (int i = 1; i <= 10; i++) {
                    System.out.println("Produced: " + i);
                    queue.put(i); // blocks if full
                    Thread.sleep(100);
                }
                queue.put(-1); // poison pill
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    static class Consumer implements Runnable {
        public void run() {
            try {
                while (true) {
                    int val = queue.take(); // blocks if empty
                    if (val == -1) break; // poison pill stops consumer
                    System.out.println("Consumed: " + val);
                    Thread.sleep(150);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    public static void main(String[] args) {
        Thread p = new Thread(new Producer());
        Thread c = new Thread(new Consumer());
        p.start();
        c.start();
    }
}
```

Explanation of how it works:

- The `ArrayBlockingQueue` of size 5 ensures only 5 items can be stored at once.
- If the queue is full, `put()` blocks producer until space is available.
- If the queue is empty, `take()` blocks consumer until something is produced.
- No explicit synchronization or locks are required — all handled internally.

Benefits:

- No manual wait/notify logic.
- Safe and efficient.
- Scales easily with multiple producers and consumers.

Analogy:

Imagine a conveyor belt (queue) with limited capacity. Producers place boxes on it; consumers take boxes off.

If the belt is full, producers wait; if it's empty, consumers wait. The `BlockingQueue` manages this coordination automatically.

JAVA Streams

1. Why should I use Streams?

Streams were introduced in Java 8 to bring a **functional programming** style to data processing. Instead of writing **loops**, you can describe *what* you want to do with your data rather than *how* to do it.

Traditional approach (imperative):

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<String> upperNames = new ArrayList<>();
for (String name : names) {
    upperNames.add(name.toUpperCase());
}
System.out.println(upperNames);
```

Using Streams (declarative):

```
List<String> upperNames = names.stream()
    .map(String::toUpperCase)
    .toList();
System.out.println(upperNames);
```

Why Streams are better:

- **Readable and expressive:** You describe data transformations, not loops.
- **Less boilerplate:** No explicit iteration or temporary collections.
- **Parallelism support:** Easily switch from `.stream()` to `.parallelStream()` for multicore processing.
- **Chaining operations:** Map, filter, sort, collect — all in a fluent way.

Analogy:

Think of Streams like a **data pipeline**. You pour data in from one end (the source), transform it as it flows (filters, maps, sorts), and collect it at the other end (collector).

2. How to handle exceptions in Streams?

Streams are elegant, but checked exceptions can make them tricky — you can't just throw checked exceptions (like `IOException`) inside lambda expressions.

Example problem:

```
List<String> files = List.of("file1.txt", "file2.txt");

files.stream()
    .map(file -> Files.readString(Path.of(file))) // Compile error: IOException
    .forEach(System.out::println);
```

✓ Common ways to handle exceptions:

1. Wrap checked exceptions in try-catch inside the lambda

```
files.stream()
    .map(file -> {
        try {
            return Files.readString(Path.of(file));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    })
    .forEach(System.out::println);
```

2. Create a utility method (reusable wrapper)

```
@FunctionalInterface
interface CheckedFunction<T, R> {
    R apply(T t) throws Exception;
}

static <T, R> Function<T, R> wrap(CheckedFunction<T, R> func) {
    return t -> {
        try {
            return func.apply(t);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    };
}

files.stream()
```

```
.map(wrap(file -> Files.readString(Path.of(file))))  
.forEach(System.out::println);
```

Why this is useful:

It keeps your stream clean and readable while handling checked exceptions gracefully.

3. How does parallel Stream work if your system has 1 CPU or 4 CPUs?

Parallel streams divide data into **multiple chunks** and process them on **different threads** (from ForkJoinPool).

Case 1: System has 1 CPU

- Only one thread can run at a time.
- Parallel stream will still split tasks, but threads will compete for CPU.
- Result: **No performance gain**, maybe even slower (due to thread overhead).

Case 2: System has 4 CPUs

- Tasks are divided among 4 threads (by default).
 - Each thread processes a portion of data in parallel.
 - Results are merged at the end.
-

Example:

```
List<Integer> numbers = IntStream.rangeClosed(1, 10).boxed().toList();  
  
numbers.parallelStream()  
    .forEach(n -> System.out.println(n + " processed by " + Thread.currentThread().getName()));
```

You'll see multiple thread names like `ForkJoinPool.commonPool-worker-1`, `worker-2`, etc.

Summary Table:

CPUs	Parallel Stream Behavior	Performance
1	Sequential execution (no gain)	Possibly slower
2+	Real parallel execution	Faster if CPU-bound
Many	Good speedup	But merging overhead can offset gains

Tip:

Parallel streams are great for **CPU-heavy** tasks, not **I/O** tasks.

Tip:

Parallel streams are great for **CPU-heavy** tasks, not **I/O** tasks.

4. What is a Functional Interface? Can I keep default methods in it?

A **Functional Interface** is an interface that has **exactly one abstract method** — it can have any number of **default** or **static** methods.

Example:

```
@FunctionalInterface
interface Calculator {
    int add(int a, int b);

    default int multiply(int a, int b) {
        return a * b;
    }

    static void show() {
        System.out.println("Functional Interface example");
    }
}
```

Explanation:

- The **@FunctionalInterface** annotation ensures that only one abstract method is present.
- **Default methods** are allowed since they have a body.
- These interfaces are the foundation of **lambda expressions** and **method references**.

Example of usage:

```
Calculator calc = (a, b) -> a + b;  
System.out.println(calc.add(5, 3)); // Output: 8
```

5. List some most commonly used functional interfaces you have used.

Here are some you'll use daily from `java.util.function` package:

Interface	Abstract Method	Purpose	Example
<code>Function<T, R></code>	<code>R apply(T t)</code>	Transform a value	<code>map(String::length)</code>
<code>Predicate<T></code>	<code>boolean test(T t)</code>	Filter based on condition	<code>filter(x -> x > 5)</code>
<code>Consumer<T></code>	<code>void accept(T t)</code>	Perform action	<code>forEach(System.out::println)</code>
<code>Supplier<T></code>	<code>T get()</code>	Provide values	<code>() -> Math.random()</code>
<code>BiFunction<T, U, R></code>	<code>R apply(T t, U u)</code>	Function with 2 inputs	<code>(a, b) -> a + b</code>
<code>UnaryOperator<T></code>	<code>T apply(T t)</code>	Operates on same type	<code>x -> x * 2</code>
<code>BinaryOperator<T></code>	<code>T apply(T a, T b)</code>	Combine two same-type values	<code>Integer::sum</code>

6. What is terminal and intermediate operation in Stream API?

Streams are composed of **intermediate** and **terminal** operations.

Intermediate operations

- Transform the stream.
- **Lazy** — they don't execute until a terminal operation is called.
- Return another Stream.

Examples:

```
filter(), map(), flatMap(), sorted(), distinct(), peek()
```

Terminal operations

- Trigger the processing of the stream.
- Return a non-stream result (like a collection, value, or void).

Examples:

```
collect(), forEach(), reduce(), count(), findFirst()
```

Example:

```
List<String> names = List.of("Alex", "Bob", "Anna");

long count = names.stream()
    .filter(n -> n.startsWith("A")) // intermediate
    .map(String::toUpperCase)      // intermediate
    .count();                      // terminal

System.out.println(count); // 2
```

Flow:

- Stream created
 - Filter and map defined (lazy)
 - `count()` triggers the full pipeline
-

7. What is FlatMap?

`flatMap()` is used to **flatten nested structures** — like converting `Stream<List<T>>` into `Stream<T>`.

It's commonly used when each element of a stream is a collection itself.

Example:

```
List<List<String>> names = List.of(
    List.of("A", "B"),
    List.of("C", "D")
);

names.stream()
    .flatMap(List::stream)
    .forEach(System.out::print);
```

Output:

ABCD

Without `flatMap`, you'd get a stream of lists instead of strings.

8. How flatMap differs from map?

Feature	<code>map()</code>	<code>flatMap()</code>
Output	Transforms elements	Transforms and flattens
Input -> Output	One-to-one	One-to-many
Result type	Stream of Stream	Stream of elements

Example:

```
List<String> words = List.of("Hello", "World");

// map
words.stream()
    .map(word -> word.split(""))
    .forEach(arr -> System.out.println(Arrays.toString(arr)));

// flatMap
words.stream()
    .flatMap(word -> Arrays.stream(word.split("")))
    .forEach(System.out::print);
```

map output:

```
[H, e, l, l, o]
[W, o, r, l, d]
```

flatMap output:

```
HelloWorld
```

9. Flatten list of lists or list of arrays using Stream API

List of Lists

```
List<List<Integer>> numbers = List.of(
    List.of(1, 2),
    List.of(3, 4)
);

List<Integer> flat = numbers.stream()
    .flatMap(List::stream)
    .toList();

System.out.println(flat); // [1, 2, 3, 4]
```

List of Arrays

```
List<String[]> data = List.of(
    new String[]{"A", "B"},
    new String[]{"C", "D"}
);

List<String> flatList = data.stream()
    .flatMap(Arrays::stream)
    .toList();

System.out.println(flatList); // [A, B, C, D]
```

10. What is method reference and how can you use it for your methods?

A **method reference** is a shorthand for writing a lambda expression that simply calls an existing method.

Example:

```
List<String> names = List.of("Alice", "Bob", "Charlie");

// Lambda
names.forEach(name -> System.out.println(name));

// Method reference
names.forEach(System.out::println);
```

Types of method references:

Type	Syntax	Example
Static method	<code>Class::method</code>	<code>Math::max</code>
Instance method of object	<code>instance::method</code>	<code>System.out::println</code>
Instance method of class	<code>Class::method</code>	<code>String::toUpperCase</code>
Constructor reference	<code>Class::new</code>	<code>Employee::new</code>

Example of constructor reference:

```
List<String> names = List.of("Alex", "Bob");
List<Employee> employees = names.stream()
    .map(Employee::new)
    .toList();
```

11. You have List, filter Employee if name starts from “A” from given city and sorted in desc order by city.

```
List<Employee> result = employees.stream()
    .filter(e -> e.getName().startsWith("A"))
    .filter(e -> e.getCity().equalsIgnoreCase("Delhi"))
    .sorted(Comparator.comparing(Employee::getCity).reversed())
    .toList();
```

12. Find the topper from each city

Assume Employee has fields: `name`, `city`, `marks`.

```
Map<String, Optional<Employee>>> topperByCity =
    employees.stream()
        .collect(Collectors.groupingBy(
            Employee::getCity,
            Collectors.maxBy(Comparator.comparingInt(Employee::getMarks))
        ));
```

Output Example:

```
{Delhi=Optional[Employee {name='Amit', marks=98}],
Mumbai=Optional[Employee {name='Arun', marks=95}]}
```

13. Can we use Stream twice? How to print some message during Stream processing?

No, **Streams are single-use**.

Once a terminal operation (like `collect()` or `forEach()`) is called, the stream is closed.

Example (wrong):

```
Stream<String> s = Stream.of("A", "B");
s.forEach(System.out::println);
s.forEach(System.out::println); // Exception: stream has already been operated upon
```

If you need to reuse, create a new stream from the source collection again.

Printing messages during processing:

Use `peek()` to debug intermediate steps.

```
List<String> result = Stream.of("A", "B", "C")
    .peek(s -> System.out.println("Before map: " + s))
    .map(String::toLowerCase)
    .peek(s -> System.out.println("After map: " + s))
    .toList();
```

14. I have a class `Employee` and `Address` is embedded inside; group `Employee` data by city

```
class Employee {
    String name;
    Address address;
}
class Address {
    String city;
}
```

Grouping by city:

```
Map<String, List<Employee>> groupByCity =
    employees.stream()
        .collect(Collectors.groupingBy(e -> e.getAddress().getCity()));
```

15. I have Employee object, group employees by city, keep sorted order by city and I want to see only name and city as a result.

You can use `TreeMap` (for sorting) and map data to a new DTO.

```
record EmployeeView(String name, String city) {}

Map<String, List<EmployeeView>> result =
    employees.stream()
        .collect(Collectors.groupingBy(
            Employee::getCity,
            TreeMap::new, // keeps sorted order
            Collectors.mapping(
                e -> new EmployeeView(e.getName(), e.getCity()),
                Collectors.toList()
            )
        ));

result.forEach((city, list) -> System.out.println(city + " -> " + list));
```

Output:

Bangalore -> [EmployeeView[name=Arun, city=Bangalore]]
Delhi -> [EmployeeView[name=Amit, city=Delhi]]

✓ End of Streams & Functional Programming Chapter

In this section, we covered:

- Why streams exist and how they work
 - How to handle exceptions and parallelism
 - Core functional interfaces
 - Advanced Stream techniques — flatMap, grouping, mapping, method references
 - Real-life problems solved elegantly with Streams
-

Springboot

1 What are the most common annotations you have used in your application

In a Spring Boot application, annotations are like **instructions to the Spring framework** — they tell Spring what to create, how to inject dependencies, how to handle HTTP requests, and so on.

Here are some **commonly used annotations**:

- `@SpringBootApplication` – entry point of the app (includes `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`).
- `@RestController` – marks a class as a REST API controller.
- `@RequestMapping`, `@GetMapping`, `@PostMapping` – define HTTP endpoints.
- `@Autowired` – injects dependencies.
- `@Service`, `@Repository`, `@Component` – mark beans for component scanning.
- `@Configuration` – defines a class containing bean definitions.
- `@Bean` – manually define a bean.
- `@Value` – injects values from `application.properties`.
- `@Transactional` – marks a method or class for transaction management.
- `@ExceptionHandler` and `@ControllerAdvice` – handle exceptions globally.
- `@Slf4j` – from Lombok for logging.
- `@Data`, `@Builder`, `@Getter`, `@Setter`, `@ToString` – Lombok annotations to reduce boilerplate.

Example:

```
@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    @Autowired
    private EmployeeService service;

    @GetMapping("/{id}")
    public ResponseEntity<Employee> getEmployee(@PathVariable Long id) {
```

```
        return ResponseEntity.ok(service.findById(id));
    }
}
```

2 How does the bean lifecycle work? Explain all

Think of Spring beans like **living creatures** that are born, live, and die within the Spring Container.

The lifecycle:

1. **Instantiation** – Spring creates an object of the bean.
2. **Populate Properties** – dependencies are injected (via [@Autowired](#) or XML).
3. **BeanNameAware, BeanFactoryAware** – Spring gives context if the bean implements these interfaces.
4. **Before Initialization** – [@PostConstruct](#) or [BeanPostProcessor](#) runs.
5. **Initialization** – any init method specified with [@Bean\(initMethod=...\)](#) runs.
6. **Ready for use** – Bean is now part of the container.
7. **Before destruction** – [@PreDestroy](#) or [DisposableBean](#) cleanup runs before context closes.

Example:

```
@Component
public class MyBean {

    @PostConstruct
    public void init() {
        System.out.println("Bean initialized!");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Bean destroyed!");
    }
}
```

3 What is a profile and how does it help?

Profiles in Spring are like having different settings for different environments (dev, test, prod).

You can mark beans with [@Profile\("dev"\)](#) or set active profiles in [application.properties](#).

spring.profiles.active=dev

```
@Profile("dev")
@Configuration
public class DevConfig {
    @Bean
    public DataSource dataSource() {
        // dev datasource
    }
}
```

This helps you **avoid hardcoding environment-specific settings**.

4 How to keep database creds in spring application

Sensitive information like database credentials should never be in plain code.

Options:

- Keep in `application.properties` or `application.yml`.
- Use environment variables.
- Use external config servers (like **Spring Cloud Config**).
- Use **Vault** (HashiCorp Vault or AWS Secrets Manager) for sensitive secrets.

```
spring.datasource.username=${DB_USER}
spring.datasource.password=${DB_PASSWORD}
```

5 You have 4 environments for application, how do you manage configurations for each environment

You can create **different property files** for each environment:

```
application-dev.properties
application-test.properties
application-uat.properties
application-prod.properties
```

Then, in your main `application.properties`, specify:

spring.profiles.active=dev

In CI/CD pipelines, this value can be injected dynamically.

6 Why does the Rest API return result in JSON only? What if I want XML?

By default, Spring Boot uses **Jackson** for JSON serialization.

If you want XML, add the dependency:

```
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

And use:

```
@GetMapping(value = "/employee", produces = MediaType.APPLICATION_XML_VALUE)
```

Spring will automatically convert your object to XML.

7 What are the conditional annotations in Spring and how does it help?

Conditional annotations help you **load beans only when certain conditions are met**.

Examples:

- `@ConditionalOnProperty`
- `@ConditionalOnMissingBean`
- `@ConditionalOnClass`
- `@ConditionalOnExpression`

Example:

```
@ConditionalOnProperty(name = "feature.new.enabled", havingValue = "true")
@Bean
public FeatureService featureService() {
    return new FeatureService();
}
```

Use case: feature toggles, environment-specific beans, or optional modules.

8 How do you map your request data to entities while saving to DB? What kind of mapper are you using?

Usually, the incoming data from API (DTO) is mapped to the Entity.

You can use:

- Manual mapping
- `ModelMapper`
- `MapStruct` (compile-time safe, preferred)

Example using **MapStruct**:

```
@Mapper(componentModel = "spring")
public interface EmployeeMapper {
    Employee toEntity(EmployeeDTO dto);
    EmployeeDTO toDto(Employee entity);
}
```

This keeps your code clean and avoids boilerplate.

9 How do you handle exceptions in Spring Boot?

You can handle exceptions using:

- `@ExceptionHandler` (local to controller)
- `@ControllerAdvice` (global)

Example:

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(EmployeeNotFoundException.class)
    public ResponseEntity<String> handleNotFound(EmployeeNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
    }
}
```

10 How do you create your own exception? and how do you handle global exception handling?

Custom exception:

```
public class EmployeeNotFoundException extends RuntimeException {  
    public EmployeeNotFoundException(String message) {  
        super(message);  
    }  
}
```

Handle it globally with `@ControllerAdvice` as shown above.

11 How does autoconfiguration help developers?

Spring Boot's **AutoConfiguration** automatically configures beans you are likely to need, based on dependencies in the classpath.

For example, if you have `spring-boot-starter-web`, it configures:

- Tomcat server
- DispatcherServlet
- Jackson mapper

You can disable it using:

`@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)`

It reduces manual configuration drastically.

12 What if I want to perform some action right after the spring container is initialized?

You can use:

- `CommandLineRunner`
- `ApplicationRunner`
- `@EventListener(ApplicationReadyEvent.class)`

Example:

```
@Component  
public class StartupTask implements CommandLineRunner {  
    public void run(String... args) {  
        System.out.println("App started! Performing post-init actions...");  
    }  
}
```

Use cases: preloading data, initializing caches, or starting scheduled jobs.

13 What is @Qualifier and how does it help?

When you have **multiple beans of the same type**, `@Qualifier` helps you specify which one to inject.

```
@Service
@Qualifier("emailService")
public class EmailNotificationService implements NotificationService { }

@Service
@Qualifier("smsService")
public class SmsNotificationService implements NotificationService { }

@Autowired
@Qualifier("emailService")
private NotificationService notificationService;
```

14 Can you tell me how your request reaches your Rest Controller?

Flow:

1. HTTP request → Tomcat (embedded server)
 2. Tomcat passes it to **DispatcherServlet**
 3. `DispatcherServlet` looks up handler mapping
 4. Request goes to correct **Controller method**
 5. Controller returns data → `HttpMessageConverter` converts to JSON/XML
 6. Response sent back
-

15 Tell me some Spring Cloud components that you have used in your application.

Common Spring Cloud components:

- **Eureka Server** → Service registry
- **Spring Cloud Config** → Centralized configuration
- **Zuul / Spring Cloud Gateway** → API Gateway
- **Hystrix / Resilience4j** → Circuit breaker
- **Feign Client** → Declarative REST client
- **Sleuth + Zipkin** → Distributed tracing

Example: using Feign for interservice communication.

```
@FeignClient(name = "employee-service")
public interface EmployeeClient {
```

```
@GetMapping("/api/employees/{id}")
Employee getEmployee(@PathVariable Long id);
}
```

16 How to achieve security using Spring Boot?

Use **Spring Security**:

- Add dependency `spring-boot-starter-security`
- Configure `SecurityFilterChain`
- Use JWT for token-based authentication

```
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf().disable()
        .authorizeHttpRequests()
        .requestMatchers("/admin/**").hasRole("ADMIN")
        .anyRequest().authenticated()
        .and().formLogin();
    return http.build();
}
```

17 How to secure data over the network

- Use **HTTPS** with SSL/TLS certificates.
- Encrypt sensitive data (e.g., AES).
- Use JWT with proper expiration.
- Secure API keys and secrets.
- Use OAuth2 for authentication.

18 Using Stream API fetch emp record based on their department: `<emp_dept, list<emp>>`

```
Map<String, List<Employee>> byDept = employees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment));
```

19 What is BlackDuck?

BlackDuck is a security tool used to scan your application for:

- Open-source vulnerabilities

- License compliance
- Dependency risks

It's often used in enterprise CI/CD pipelines.

20 Can we have a prototype bean inside a singleton bean in Spring Boot?

Singleton bean is created once, prototype bean is created each time requested.
You need to use **ObjectFactory** or **Provider** to ensure new instance:

```
@Component
public class SingletonBean {
    @Autowired
    private ObjectFactory<PrototypeBean> factory;

    public void usePrototype() {
        PrototypeBean p = factory.getObject();
        p.doSomething();
    }
}
```

21 Type of vulnerability faced in project

Common ones:

- SQL Injection
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- Insecure Direct Object References (IDOR)
- Sensitive data exposure

Fixes include validation, sanitization, parameterized queries, and HTTPS.

22 Non-functional requirements you will ask before starting any project

Non-functional requirements define **how** the system behaves:

- Performance (response time, load capacity)

- Scalability
 - Security
 - Logging & Monitoring
 - Reliability
 - Maintainability
 - Availability
 - Compliance (GDPR, PCI DSS)
-

23 How do you handle exceptions in your Spring Boot app?

- Local using `@ExceptionHandler`
 - Global using `@ControllerAdvice`
 - For async processes: handle via `CompletableFuture.exceptionally()` or global error handler.
-

24 How to implement global exception handling in Spring Boot microservice?

Create a `@ControllerAdvice` class with `@ExceptionHandler` methods.
You can also use `ResponseEntityExceptionHandler` to customize responses.

25 How to handle exceptions using AspectJ?

Use AOP (Aspect-Oriented Programming):

```
@Aspect
@Component
public class ExceptionAspect {

    @AfterThrowing(pointcut = "execution(* com.app.service.*.*(..))", throwing = "ex")
    public void logException(Exception ex) {
        log.error("Exception occurred: ", ex);
    }
}
```

This helps separate error handling logic from business logic.

26 How to enable distributed logging for your microservices? What are the tools you have ever used?

Use **Spring Cloud Sleuth + Zipkin** or **ELK Stack** (Elasticsearch, Logstash, Kibana).

Flow:

- Each microservice logs with a trace ID.
- Sleuth attaches trace IDs automatically.
- Zipkin or ELK visualizes request flow.

Tools:

- Sleuth
 - Zipkin
 - ELK Stack
 - Prometheus + Grafana for metrics
-

1) What is the best way to call an API from another service? What steps do you take while integrating to external/internal services? Security steps?

Short answer: Use a robust HTTP client (or a declarative client like Feign), keep calls resilient (timeouts, retries, circuit breakers), validate inputs/outputs, instrument/log traces, and secure communication (TLS, auth tokens) and credentials.

Detailed checklist / story

Calling another service is a common, mundane action — but when you do it at scale you must treat it like a first-class subsystem. Think reliability, performance, security, and observability.

Steps & best practices:

1. Choose the right client

- Internal services: use a declarative client (Feign) or `RestTemplate` / `WebClient` (reactive). For blocking calls, `RestTemplate` (legacy) or `HttpClient`; for reactive, Spring WebFlux `WebClient`.
- External APIs: prefer typed clients and wrapper classes so you can adapt to API changes safely.

2. Timeouts

- Configure connect, read and write timeouts. Never rely on defaults. Example using `HttpClient` or `WebClient`.

3. Retries & Backoff

- Retries for transient errors with exponential backoff and jitter (avoid thundering herd). Use libraries like Resilience4j.

4. Circuit Breaker

- Open/close circuits to avoid repeated calls to failing services (Resilience4j/Hystrix-like patterns).

5. Bulkhead / Rate-limiting

- Partition resources to avoid cascading failures. Limit concurrency to third-party APIs.

6. Input & Output Validation

- Validate outgoing request parameters and sanitize incoming responses.

7. Authentication

- Use TLS (HTTPS) always. Use tokens (OAuth2 JWTs, mTLS) for auth. Avoid sending credentials in query params.

8. Secrets management

- Don't hardcode creds. Use Vault, AWS Secrets Manager, KMS, or Spring Cloud Config with encrypted properties.

9. Observability

- Trace requests end-to-end (Sleuth, OpenTelemetry + Jaeger/Zipkin). Log request IDs and errors.

10. Idempotency & Retries

- For non-idempotent ops, use idempotency keys to prevent duplicate processing.

11. Graceful degradation

- Provide fallback behavior if external service fails.

12. API contracts

- Use swagger/openapi, validate responses, version your APIs.

Example (WebClient with timeout and simple retry):

```
WebClient client = WebClient.builder()
    .baseUrl("https://api.example.com")
    .clientConnector(new ReactorClientHttpConnector(HttpClient.create()
        .responseTimeout(Duration.ofSeconds(5))))
    .build();

Mono<String> resp = client.get().uri("/items/{id}", id)
    .retrieve()
    .bodyToMono(String.class)
    .retryWhen(Retry.backoff(3, Duration.ofMillis(200)).jitter(0.5));
```

Security checklist: TLS, OAuth2 or mTLS, secrets in vault, minimal privileges, input validation, logging without secrets.

2) How have you leveraged OAuth2 in your application?

Short answer: Used OAuth2 for token-based authentication/authorization. We mainly used Authorization Code flow with JWT access tokens for user-centric APIs and Client Credentials for service-to-service calls. We used Spring Security OAuth2 and an identity provider (Keycloak / Auth0 / Okta) in front of services.

Detailed story & implementation points:

- **Use cases implemented:**
 - Web UI login via Authorization Code (OIDC) — user logs in at IdP, receives ID token + access token.
 - Microservice-to-microservice calls: Client Credentials grant with short-lived JWTs.
 - Token introspection or public-key verification for resource servers.
- **Key pieces:**
 - **Authorization Server / IdP:** Keycloak (open-source), or managed (Okta/Auth0).
 - **Resource Server:** Spring Boot services validate JWTs or introspect tokens.
 - **Client:** Web app or service that requests tokens.
- **Spring Boot example:**

Client config for Resource Server:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jwk-set-uri: https://idp.example.com/.well-known/jwks.json
```

- Controller secured by scopes / roles using `@PreAuthorize("hasAuthority('SCOPE_orders:read')")`.
- **Why OAuth2:**
 - Delegated authorization, tokens, revocation, standard flows for web/mobile/service apps.

3) How are you getting auth token from auth server and how does it get refreshed?

Short answer: Use the appropriate grant to obtain tokens; refresh with Refresh Token grant (for flows that provide refresh tokens). Clients store refresh tokens securely (or use token rotation). For client credentials (machine-to-machine), just request new tokens when needed.

Detailed flow examples:

- **Authorization Code (user login)**

- Client redirects user to IdP consent/login page.
- IdP returns authorization code.
- Client exchanges code for **access_token** and **refresh_token**.
- Client stores refresh token securely (e.g., httpOnly secure cookie or server-side session).
- When access token is near expiry, client uses refresh token to call **/token** with **grant_type=refresh_token** to obtain a new access token (and sometimes new refresh token).

- **Client Credentials (service-to-service)**

- Client requests token with client id/secret. No refresh token usually; simply request a new access token when expired.

Example token refresh (pseudo):

POST /oauth/token

Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token&refresh_token=<<refresh>>&client_id=app&client_secret=...

Security notes:

- Refresh tokens must be stored securely; consider rotating and revoking refresh tokens on suspicious activity.
- Use short-lived access tokens.

4) Explain the grant types with use cases you've used

Quick list and uses we used:

- **Authorization Code (with PKCE for mobile/SPA)** — used for user login flows. Best for confidential clients and public clients with PKCE. We used it for web UIs (React) + Spring backend.
- **Client Credentials** — machine-to-machine authentication (microservice-to-microservice). This is common for backend jobs / services.
- **Password (Resource Owner Password Credentials)** — deprecated/strongly discouraged. We avoided it.
- **Implicit** — deprecated; replaced by Authorization Code with PKCE for SPAs.

- **Refresh Token** — not a grant per se but a way to renew access tokens. Used for long-lived user sessions.
- **Device Code** — when devices have limited input (e.g., TV). Rare in our apps.
- **JWT Bearer Grant** — when exchanging external JWTs for local tokens.

Example use case mapping:

- Web UI (user): Authorization Code + PKCE
- Mobile app: Authorization Code + PKCE
- Microservice backend: Client Credentials
- Long live sessions: Refresh tokens with rotation

5) How does the resource server verify your token? Does it hit the auth server for each request? How to optimize?

Short answer: Resource servers validate JWT signature locally using the IdP's public keys (JWK). They do **not** need to call the auth server on each request. Use token validation + caching of JWKs and optional introspection for opaque tokens.

Detailed explanation:

- **JWT tokens:** Resource server verifies signature (public key), checks claims (iss, aud, exp, nbf). This is local verification and inexpensive.
- **Opaque tokens:** Resource server calls token introspection ([/introspect](#)) on IdP to check validity. This is network-bound and slower.
- **Optimization:**
 - Prefer JWTs for offline verification.
 - Cache JWKs locally with refresh policy (IdP publishes JWK set URL).
 - For introspection-heavy cases, use local caching for introspection results with short TTL.
 - Validate token expiry and audience locally to avoid unnecessary introspection.

Spring Boot example:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jwk-set-uri: https://idp.example.com/.well-known/jwks.json
```

Spring will fetch JWKs and validate signatures locally. JWKs are cached and refreshed periodically.

Security tradeoff: Local JWT verification avoids network calls, but revocation is harder. Use short expiry + token revocation lists or introspection for critical revocation scenarios.

6) How to avoid users from using most common passwords in Spring Boot? Any libraries / options?

Short answer: Use a password policy checker that rejects common or leaked passwords. Use open-source lists (HaveIBeenPwned) or libraries (OWASP Password Policy, zxcvbn) and enforce complexity and blacklist checks.

Practical approaches:

1. **Blacklist common passwords**
 - Keep a list of banned passwords (e.g., top 10k). Check on registration and password change.
2. **Check against leaked password databases**
 - Use **HaveIBeenPwned** API (k-Anonymity model) to check if password has appeared in breaches (send sha1 prefix only).
 - Use [hip](#) client libraries.
3. **Use password strength libraries**
 - **zxcvbn** (Dropbox) has Java ports; measures strength rather than rules.
4. **Enforce complexity & length**
 - Minimum length (≥ 12), disallow common patterns, require passphrases.
5. **Rate-limit password attempts**
 - Prevent brute force.
6. **Password hashing**
 - Use bcrypt, Argon2, or PBKDF2 with good parameters.

Example snippet (using HIBP k-Anonymity):

- Compute SHA1 of password, send first 5 hex chars to the HIBP API, check response for suffix count.

Libraries:

- [zxcvbn4j](#) (zxcvbn port)
 - OWASP [passay](#) for policies
 - Use HIBP for breach checks
-

7) Explain the Auth flow in Spring Boot — calling an API, how request proceeds & response returned

High-level flow (user logs in and calls protected API):

1. User login (Auth flow)

- For web UI: server redirects to IdP (Authorization Code).
- User authenticates at IdP (credentials, MFA), IdP returns code.
- Client exchanges code for tokens (access token + optional refresh + id token).
- Client stores access token (secure storage).

2. Call protected API

- Client sends request with **Authorization: Bearer <access_token>**.
- Request hits API Gateway / Spring Boot resource server.
- **Spring Security filter chain** intercepts the request, extracts token.
- If JWT: server validates signature & claims locally; sets **SecurityContext** with **Authentication**.
- If opaque token: server introspects (or cache + introspect).
- Controller is invoked; security annotations (**@PreAuthorize**) check authorities.
- Controller returns response; token remains valid until expiry.

3. Token refresh

- If access token expired, client uses refresh token to get new access token (server-side on web apps or secured refresh endpoint).

Key Spring pieces:

- **OncePerRequestFilter** to parse token
- **JwtDecoder** to validate tokens
- **SecurityContextHolder** to hold authenticated principal

8) How does Spring Boot remember you if you call APIs a second time? Does it do all auth steps again?

Short answer: No. Once a request is authenticated, Spring populates the **SecurityContext** for that request. Across requests, the client typically sends the same access token (stateless); the server verifies it for each request (signature & claims check) — a quick operation. Session-based logins use server session store (cookies) to remember.

Details:

- **Stateless (JWT) approach:** Each incoming request includes the token; server validates token per request (no server-side session needed). Signature validation and claim checks happen each time.

- **Stateful (Session) approach:** Spring Security creates an HTTP session. The user is stored in the server session and identified by cookie. Future requests are matched to the session; no token validation needed.
 - For JWTs, validation is fast — server does not perform the entire OAuth handshake again.
-

9) How can you implement your own auth provider?

Short answer: Implement `AuthenticationProvider` and register it in Spring Security configuration. Use it when you have custom logic (e.g., DB-based, LDAP, external API).

Example outline:

```
public class CustomAuthProvider implements AuthenticationProvider {
    @Autowired private UserService userService;

    @Override
    public Authentication authenticate(Authentication auth) throws AuthenticationException {
        String username = auth.getName();
        String password = auth.getCredentials().toString();
        UserDetails user = userService.loadUserByUsername(username);
        if (passwordEncoder.matches(password, user.getPassword())) {
            return new UsernamePasswordAuthenticationToken(user, password, user.getAuthorities());
        }
        throw new BadCredentialsException("Bad credentials");
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return UsernamePasswordAuthenticationToken.class.isAssignableFrom(authentication);
    }
}
```

Register in SecurityConfig:

```
http.authenticationProvider(customAuthProvider());
```

10) What auth server have you used? How do you validate users? If I want resources protected, how do you do it?

Common auth servers used:

- **Keycloak** (self-hosted, feature-rich)
- **Okta / Auth0** (managed)
- **AWS Cognito** (cloud)
- Custom OAuth2 servers using Spring Authorization Server (newer option)

User validation approaches:

- **Users stored in IdP** (Keycloak user DB) — IdP handles authentication.
- **External user store:** IdP federates with LDAP or custom user DB.
- **MFA:** integrated via IdP.

Protecting resources:

- Configure Spring Resource Server to validate JWTs (JWK URL).
 - Apply method-level security (`@PreAuthorize`) or URL-based security rules in `SecurityFilterChain`.
 - Use scopes/roles for authorization.
-

11) Difference between Authentication and Authorization in Spring Security

- **Authentication** = verifying identity (who you are). In Spring: `AuthenticationManager` authenticates credentials and yields an `Authentication` object with principal and authorities.
 - **Authorization** = access control (what you're allowed to do). In Spring: `@PreAuthorize`, `hasRole()`, or security filter chain determine access to resources based on `Authentication`'s authorities/roles.
-

12) Role of `UserDetailsService` in Spring Security

`UserDetailsService` is how Spring loads user data during authentication. Implement `loadUserByUsername(String username)` to return `UserDetails` with username, password, and authorities.

Example:

```
@Service
public class MyUserDetailsService implements UserDetailsService {
    @Autowired UserRepository repo;
    public UserDetails loadUserByUsername(String username) {
        User u = repo.findByUsername(username);
        return new org.springframework.security.core.userdetails.User(u.getUsername(), u.getPassword(),
            mapRoles(u));
    }
}
```

13) How to implement custom authentication in Spring Security?

- Implement `UserDetailsService` and `AuthenticationProvider` or configure `DaoAuthenticationProvider` with your `UserDetailsService`.
- Configure password encoding (`BCryptPasswordEncoder`).
- Register provider in `SecurityFilterChain`.

14) What is a **SecurityContext**, and how does it work in Spring Security?

SecurityContext holds the **Authentication** object (principal + authorities) for the current execution. Spring stores it in **SecurityContextHolder**. By default, **SecurityContext** is stored in a **ThreadLocal** for the request. For web apps, **SecurityContextPersistenceFilter** manages loading/saving the context to session or request.

15) How do you secure a REST API using Spring Security?

Steps:

1. Add **spring-boot-starter-security**.
2. Configure **SecurityFilterChain** — disable CSRF for stateless APIs.
3. Configure resource server JWT validation or session/cookie auth.
4. Use **@PreAuthorize/@RolesAllowed** for method-level checks.
5. Use HTTPS, rate-limiting, input validation, logging.
6. Use CORS policy to restrict origins.

Basic JWT Resource Server config:

```
http.csrf().disable()
    .authorizeHttpRequests().requestMatchers("/public/**").permitAll()
    .anyRequest().authenticated()
    .and().oauth2ResourceServer().jwt();
```

16) Differences between Role-based and Permission-based access control

- **Role-based (RBAC):** Assign roles (e.g., **ROLE_ADMIN**) to users; permissions are implied by role. Simpler, coarser-grained.
- **Permission-based (ABAC or fine-grained):** Check specific permissions/attributes (e.g., **invoice:read**, **invoice:write**, or attribute conditions). More flexible and granular.

Spring supports both: **hasRole("ADMIN")** vs **hasAuthority("invoice:read")** or custom **@PreAuthorize** using expressions.

17) How does Spring Security support OAuth2 and OpenID Connect?

- Spring Security offers **oauth2Login()** for OIDC login, **oauth2ResourceServer()** for resource servers, and **spring-security-oauth2-client** to configure clients (Feign + OAuth2).
- OIDC adds **id_token** for user identity and standard claims. You configure **ClientRegistration** for the IdP.

Example: **spring.security.oauth2.client.registration** in properties for clients.

18) Implement method-level security using `@PreAuthorize` and `@Secured`

- Enable method security: `@EnableMethodSecurity` (or old `@EnableGlobalMethodSecurity`).
- Use `@PreAuthorize("hasRole('ADMIN')")` or complex SpEL
- `@Secured("ROLE_ADMIN")` is simpler but less expressive.

Example:

```
@PreAuthorize("hasRole('ADMIN') or #id == principal.id")  
  
public void deleteUser(Long id) { ... }
```

19) Role of Spring Boot Actuator and use in microservices

Actuator provides production-ready endpoints: health, metrics, info, environment, thread dump. Useful with monitoring/alerts.

- Integrate with Prometheus/Grafana for metrics.
- Use `/health` for service discovery readiness/liveness probes.
- Secure actuator endpoints (expose only authenticated/admin).

Example:

management.endpoints.web.exposure.include: health,info,prometheus

20) Configure security in Spring Boot without default configuration

Spring Boot auto-configures security. To override:

- Provide your own `SecurityFilterChain` bean and disable `WebSecurityConfigurerAdapter`-style defaults.
- Example minimal:

```
@Bean  
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
    http.csrf().disable()  
        .authorizeRequests().anyRequest().authenticated()  
        .and().httpBasic();  
    return http.build();  
}
```

21) Spring Boot integration with external configuration sources (DB, cloud config)

- **Spring Cloud Config Server:** central configuration, reads from Git, vault, or DB and serves to clients.

- **Database-backed config:** implement custom `PropertySource` or use `JdbcTemplatePropertySource`.
- **Vault:** HashiCorp Vault + Spring Cloud Vault for secrets.
- **Environment variables** and Kubernetes ConfigMaps/Secrets also supported.

Example: client adds `spring-cloud-starter-config` and `bootstrap.yml` points to config server.

22) How does Spring Boot manage transactions? Propagation and isolation levels?

- Spring uses `@Transactional` to manage transactions via AOP.
- **Propagation:**
 - `REQUIRED` (default): use existing transaction or create new.
 - `REQUIRES_NEW`: create a new transaction, suspends existing.
 - `NESTED`: nested transaction (savepoints).
 - `SUPPORTS`, `NOT_SUPPORTED`, `MANDATORY`, `NEVER`.
- **Isolation levels:**
 - `DEFAULT`, `READ_UNCOMMITTED`, `READ_COMMITTED`, `REPEATABLE_READ`, `SERIALIZABLE`.

Example:

```
@Transactional(propagation = Propagation.REQUIRED, isolation = Isolation.READ_COMMITTED)
public void doWork() { ... }
```

Pitfalls: Avoid long transactions; keep transactional boundaries narrow.

23) Benefits & challenges of using Spring Boot with reactive stack (WebFlux) vs MVC

Benefits:

- Non-blocking, scalable for high concurrency with fewer threads.
- Efficient for I/O-bound workloads.

Challenges:

- Higher cognitive load — reactive programming model (Flux/Mono).
- Not all libraries are non-blocking (JDBC is blocking). Need R2DBC for reactive DB.
- Debugging and tracing differences.
- Migration from imperative code is non-trivial.

Use when: You have high concurrency and lots of I/O (APIs, streaming). For CPU-bound workloads MVC is simpler and often better.

24) Implement and configure a custom filter or interceptor

- **Filter:** servlet-level filter. Implement `javax.servlet.Filter` and register as bean or via `FilterRegistrationBean`.
- **Interceptor:** Spring MVC HandlerInterceptor for pre/post handle within Spring MVC.

Example Filter:

```
@Component
public class LoggingFilter implements Filter {
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) {
        // pre
        chain.doFilter(req, res);
        // post
    }
}
```

Example Interceptor:

```
public class AuthInterceptor implements HandlerInterceptor {
    public boolean preHandle(HttpServletRequest req, HttpServletResponse res, Object handler) {
        // check token
    }
}
```

Register interceptor in `WebMvcConfigurer#addInterceptors`.

25) Purpose of `@Conditional` annotations and how to create custom conditions

`@Conditional` family controls bean creation based on environment. Common ones:

- `@ConditionalOnProperty`
- `@ConditionalOnClass`
- `@ConditionalOnMissingBean`
- `@ConditionalOnExpression`

Create custom condition:

```
public class OnMyCondition implements Condition {
    public boolean matches(ConditionContext ctx, AnnotatedTypeMetadata meta) { ... }
}
```

```
@Conditional(OnMyCondition.class)
@Bean public MyBean myBean() { ... }
```

Use cases: enable features only in certain environments, optional integrations.

26) How do you document your REST API?

- Use **OpenAPI/Swagger** with [springdoc-openapi](#) or [springfox](#) (older).
- Annotate controllers with [@Operation](#), [@ApiResponse](#).
- Generate API UI & JSON spec: Swagger UI, ReDoc.

Example dependency: [org.springdoc:springdoc-openapi-ui](#) — adds [/swagger-ui.html](#).

27) How to handle input validation in Spring? Ways and best approach

Options:

- **Jakarta Bean Validation** ([@Valid](#), [@NotNull](#), [@Size](#)) with Hibernate Validator (preferred).
- Manual checks in code (not recommended as primary).
- Custom validators ([ConstraintValidator](#)) for complex rules.

Example:

```
public class UserDto {
    @NotBlank private String name;
    @Email private String email;
}
@PostMapping
public ResponseEntity<?> create(@Valid @RequestBody UserDto dto) { ... }
```

Use [@ControllerAdvice](#) to centralize [MethodArgumentNotValidException](#) handling.

Best practice: Use bean validation + custom validators for business rules; return structured errors.

28) How to enable security for Swagger docs?

- Secure Swagger endpoints via Spring Security configuration.
- Expose login endpoints to Swagger if needed; protect [/v3/api-docs](#) and [/swagger-ui/**](#).
- Optionally require API key or basic auth to view docs in non-dev environments.

Example:

```
http.authorizeRequests()
    .antMatchers("/swagger-ui/**", "/v3/api-docs/**").hasRole("DEV");
```

29) What is JWT?

JSON Web Token (JWT) is a compact, URL-safe token containing claims and a signature. Structure: `header.payload.signature`. Commonly used as bearer tokens.

- **header**: algorithm (e.g., RS256).
- **payload**: claims (`sub`, `exp`, `aud`, `scope`).
- **signature**: server signs header+payload with secret/private key.

Pros: stateless, can validate locally.

Cons: revocation and long-lived JWTs require careful handling.

30) What is OAuth2.0 and how have you used it?

Short: OAuth2 is a framework for delegated authorization. In our projects, we used it for SSO (with OIDC) and for service-to-service auth (Client Credentials). Implementation involved Keycloak/Okta as IdP and Spring Security clients & resource servers.

31) How have you handled security for service-to-service communication?

Approaches:

- **Client Credentials OAuth2**: each service has client id/secret and requests token to call other services.
- **mTLS** (mutual TLS): both sides authenticate via certificates.
- **JWT tokens**: services validate tokens locally.
- **Network segmentation & service mesh** (Istio) for mutual auth and policy enforcement.

Typical setup: Service A obtains token (client credentials) from IdP and calls Service B with `Authorization: Bearer`.

32) Grant Types in OAuth2 — explain use cases

- **Authorization Code (with PKCE)**: Web & mobile user login. (User-agent flow)
- **Implicit**: Deprecated (used for SPAs previously).
- **Client Credentials**: Service-to-service machine auth.
- **Resource Owner Password Credentials**: Deprecated (user provides username/password to client) — avoid.
- **Refresh Token**: Exchange refresh token to get new access token.

- **Device Code:** Devices with limited input.
-

33) How are you handling Authentication & Authorization in your application? Auth servers used?

Typical architecture used:

- **Auth server:** Keycloak (or Okta) for identity and OAuth2 flows.
 - **Gateways:** API Gateway validates tokens and performs authz checks when possible.
 - **Resource servers:** Validate JWTs and enforce method-level security.
 - **Auditing & logging:** Log user ID and trace IDs.
-

34) What is OIDC and what does it do? Have you used it? Explain the flow

OpenID Connect (OIDC) is an identity layer on top of OAuth2. It issues an `id_token` (JWT) that contains user identity claims.

Flow (Auth Code with OIDC):

1. Client redirects user to IdP with `scope=openid`.
2. IdP authenticates user.
3. IdP returns authorization code.
4. Client exchanges code for `id_token` and `access_token`.
5. `id_token` contains user info (sub, name, email). `access_token` used for resource access.

Used for SSO in our applications with Keycloak/Okta.

35) What is LDAP and how integrate with Auth server?

LDAP: Lightweight Directory Access Protocol — used for centralized user directories (Active Directory).

Integration options:

- IdP (Keycloak) connects to LDAP as user federation source.
- Spring Security can directly authenticate against LDAP using `LdapAuthenticationProvider`.
- Use LDAP for username/password + group membership.

Example (Spring LDAP config) is straightforward with `AuthenticationManagerBuilder ldapAuthentication()`.

36) How are you securing resources in microservice architecture? Possible ways in Spring Boot

Techniques:

- **JWT-based resource servers** validate tokens locally.
 - **API Gateway** enforces auth and limits — only forward valid requests.
 - **mTLS** for service-to-service authentication.
 - **Service mesh** (Istio) for mutual TLS and policies.
 - **Role and attribute-based access control** with fine-grained permissions.
 - **Rate limiting**, quotas, WAF, and network ACLs.
-

37) Ways to keep a logged-in user logged-in; session management for auth tokens

Common approaches:

- **JWT + refresh tokens**: short-lived access tokens, refresh tokens used to obtain new ones.
- **Session cookies**: server-managed sessions; cookie stored client-side (httpOnly, secure).
- **Token rotation & sliding sessions**: refresh tokens rotated on use; sliding expiry for active sessions.

Session store options: In-memory (not for scale), Redis-backed distributed session store, or stateless JWT.

Security considerations: use httpOnly secure cookies, CSRF protection for cookie-based sessions, rotate refresh tokens.

38) How to avoid accepting most commonly used passwords

Approaches used in practice:

1. **Blacklist Top Passwords**
 - Keep a list of common passwords (e.g., 100k list) and reject on signup.
2. **Check with HaveIBeenPwned** (k-Anonymity)
 - Use HIBP's password API safely (send SHA-1 prefix only) to know if password appeared in leaks.
3. **Strength measurement (zxcvbn)**
 - Estimate password strength; require passphrase or entropy threshold.
4. **Password complexity & minimum length**
 - Force ≥ 12 chars, no simple patterns.
5. **Disallow patterns**
 - Disallow **Test123**, **Password1**, or username in password.
6. **Rate limit & captcha**
 - Avoid automated attempts.
7. **Policy as feedback**
 - Inform users why a password is rejected and give suggestions.

Implementation pattern:

- On registration/password change:
 1. Validate length & simple checks.
 2. Check blacklists.
 3. Check HIBP.
 4. Return structured validation errors.

Libraries/tools: [zxcvbn4j](#), HIBP API, OWASP [passay](#).

Final Recommendations & Patterns (wrap-up)

- **Use standards** (OAuth2, OIDC, JWT) and delegated auth servers (Keycloak/Okta) unless you need extreme customization.
 - **Favor short-lived access tokens** + refresh tokens for user sessions.
 - **Protect secrets:** Vault or cloud secret managers.
 - **Local JWT validation** to avoid network latency; but consider revocation strategies.
 - **Service-to-service security:** Client Credentials + mTLS + service mesh for defense-in-depth.
 - **Security by design:** validate input, log safely (no secrets), apply least privilege, add monitoring and detection.
 - **User password hygiene:** combine blacklists, breach checks, and strength meters.
-

SpringData

1) How do you integrate your application with different databases?

Short answer:

Spring Boot provides auto-configuration for various databases (SQL & NoSQL). You just add the right driver dependency and configure the datasource properties in `application.yml` or `application.properties`.

Approach:

1. Add dependency

- MySQL → `mysql-connector-j`
- PostgreSQL → `org.postgresql:postgresql`
- MongoDB → `spring-boot-starter-data-mongodb`
- Redis → `spring-boot-starter-data-redis`

2. Configure properties

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: user
    password: pass
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
```

2. Entity + Repository layer

- Annotate entity classes with `@Entity` or `@Document` (for Mongo).
- Define repository interfaces extending `JpaRepository` or `MongoRepository`.

3. Multiple Databases

- Use separate `DataSource`, `EntityManagerFactory`, and `TransactionManager` beans for each DB.
- Mark repositories with `@EnableJpaRepositories(basePackages="...")` pointing to respective configs.

Example:

```
@Configuration
@EnableJpaRepositories(
    basePackages = "com.example.mysqlrepo",
    entityManagerFactoryRef = "mysqlEntityManagerFactory",
    transactionManagerRef = "mysqlTransactionManager"
)
public class MySQLConfig { ... }
```

2) How does **CrudRepository** differ from **JpaRepository**?

Feature	CrudRepository	JpaRepository
Base	Part of Spring Data Commons	Extends CrudRepository
Purpose	Basic CRUD operations	Adds JPA-specific features
Methods	save, findById, findAll, deleteById	Adds flush, batch operations, pagination, and sorting
Query Language	Generic	Supports JPQL and custom queries
Example	Good for simple NoSQL or generic stores	Standard for relational databases (JPA/Hibernate)

Use:

- Use **CrudRepository** for lightweight/simple data access.
- Use **JpaRepository** for production-level JPA access (preferred for SQL).

3) Hierarchy of Spring Data interfaces & creating your own Repository

Hierarchy:

Repository (marker)

```
└─ CrudRepository
    └─ PagingAndSortingRepository
        └─ JpaRepository
```

To create a custom repository:

Create a base interface:

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {  
    List<Employee> findByCity(String city);  
}
```

For custom logic:

```
public interface EmployeeCustomRepo {  
    void customUpdate(Long id, String city);  
}
```

```
@Repository  
public class EmployeeCustomRepoImpl implements EmployeeCustomRepo {  
    @PersistenceContext EntityManager em;  
    public void customUpdate(Long id, String city) {  
        em.createQuery("UPDATE Employee e SET e.city = :city WHERE e.id = :id")  
            .setParameter("city", city).setParameter("id", id).executeUpdate();  
    }  
}
```

Then combine:

```
public interface EmployeeRepository extends JpaRepository<Employee, Long>,  
EmployeeCustomRepo {}
```

1.

4) How do you write a query method in Spring Boot? Common rules

Derived query method:

Spring Data derives query logic from method names.

Example:

```
List<Employee> findByName(String name);  
List<Employee> findByCityAndSalaryGreaterThan(String city, double salary);
```

Rules:

- Start with `findBy`, `readBy`, `getBy`.
- Combine fields with `And` / `Or`.
- Use keywords like `Between`, `LessThan`, `GreaterThan`, `OrderBy`, `Like`, `In`.
- Support nested properties: `findByAddress_City(String city)`.

Examples:

- `findByDepartmentNameAndSalaryGreaterThan("IT", 50000)`
 - `findTop3ByOrderByMarksDesc()` → top 3 students
 - `existsByEmail(String email)` → boolean return
-

5) How to run SQL query in Spring Boot? Why use native queries?

Ways:

JPQL query (entity-based)

```
@Query("SELECT e FROM Employee e WHERE e.city = :city")
List<Employee> findByCity(@Param("city") String city);
```

Native query (SQL)

```
@Query(value = "SELECT * FROM employee WHERE city = :city", nativeQuery = true)
List<Employee> findByCityNative(@Param("city") String city);
```

Why native SQL?

- Need DB-specific features (e.g., JSON columns, window functions, stored procs).
- Performance-tuned complex queries.
- Access to non-entity tables or views.

Use native only when JPQL or Criteria API can't meet the requirement.

6) How to handle pagination and sorting (SQL & NoSQL)?

Spring Data provides `Pageable` and `Sort`.

JPA (MySQL) example:

```
Page<Employee> findByCity(String city, Pageable pageable);
```

Usage:

```
PageRequest page = PageRequest.of(0, 5, Sort.by("salary").descending());  
repository.findByCity("Delhi", page);
```

MongoDB example:

```
Page<Student> findByDepartment(String dept, Pageable pageable);
```

Internally:

Spring Data translates pagination/sort into **LIMIT/OFFSET** or cursor-based pagination depending on backend.

7) How do you handle auditing in Spring Data JPA? Why needed?

Why:

Track who created or modified an entity and when. Critical for traceability, debugging, and compliance.

Setup:

Enable auditing:

```
@EnableJpaAuditing  
@SpringBootApplication  
public class Application {}
```

Add fields in entity:

```
@Entity  
@EntityListeners(AuditingEntityListener.class)  
public class Employee {  
    @CreatedDate LocalDateTime createdDate;  
    @LastModifiedDate LocalDateTime updatedDate;  
    @CreatedBy String createdBy;  
    @LastModifiedBy String updatedBy;  
}
```

Provide auditor:

```
@Component
```

```
public class AuditorAwareImpl implements AuditorAware<String> {  
    public Optional<String> getCurrentAuditor() {  
        return Optional.ofNullable(SecurityContextHolder.getContext().getAuthentication().getName());  
    }  
}
```

8) How do you handle composite keys in Entity class?

Two approaches:

@EmbeddedId

```
@Embeddable  
public class EmployeeId implements Serializable {  
    private Long deptId;  
    private Long empNo;  
}
```

```
@Entity  
public class Employee {  
    @EmbeddedId  
    private EmployeeId id;  
}
```

@IdClass

```
@IdClass(EmployeeId.class)  
@Entity  
public class Employee {  
    @Id  
    private Long deptId;  
    @Id  
    private Long empNo;  
}
```

Use **@EmbeddedId** when you want reusable embedded key object.

9) Have you used Flyway? How does it help?

Flyway automates DB migrations — versioned scripts run in sequence to keep all environments consistent.

Benefits:

- Version-controlled DB changes.
- Auto migration on app startup.
- Rollback and history tracking.
- Integrates easily with CI/CD.

Setup:

- Add dependency: `org.flywaydb:flyway-core`
 - Add migration files: `src/main/resources/db/migration/V1__init.sql`
 - On startup, Flyway auto-runs missing scripts.
-

10) How to tune your join query?

Steps:

- Ensure join columns are indexed.
- Fetch only required columns (`select new Dto(...)`).
- Use `fetch join` for eager data without N+1 issue.
- Avoid cross joins; use proper ON conditions.
- Limit fetched rows using pagination.
- Profile using Hibernate logs and `EXPLAIN` plan.

Example:

```
@Query("SELECT e FROM Employee e JOIN FETCH e.department WHERE e.city = :city")
List<Employee> findByCityWithDept(String city);
```

11) How to make select query faster?

Techniques used:

1. Use proper indexes.
2. Avoid `SELECT *` – fetch only required fields.
3. Use projections (DTO or interface-based).
4. Enable second-level cache for frequently read entities.
5. Use pagination for large datasets.
6. Analyze queries with `EXPLAIN`.

Tune connection pool and batch fetch sizes:

```
spring.jpa.properties.hibernate.default_batch_fetch_size=20
```

12) What is sharding and partitioning?

- **Partitioning:** Split data within a single DB (e.g., by range, hash, list). Helps performance & manageability.
- **Sharding:** Distribute data across multiple databases or servers (horizontal scaling).

Benefits:

- Improved performance & scalability.
- Isolation and better throughput.

Example use: Multi-tenant systems or high-scale event stores.

13) Have you used indexing? Use cases?

Indexes speed up data retrieval at cost of write performance.

Use cases:

- Frequently queried columns (**WHERE**, **JOIN**, **ORDER BY**).
- Unique constraints (email, username).
- Composite indexes for multi-column queries.

Example:

```
@Entity
@Table(indexes = @Index(name="idx_emp_city", columnList="city"))
public class Employee { ... }
```

14) Get the 3rd topper student from table using JPA

```
@Query("SELECT s FROM Student s ORDER BY s.marks DESC LIMIT 1 OFFSET 2")
Student findThirdTopper(); // in Hibernate 6+ supports offset/limit
```

or using Pageable:

```
PageRequest page = PageRequest.of(2, 1, Sort.by("marks").descending());
Student third = repo.findAll(page).getContent().get(0);
```

15) Get last five and first five results

```
List<Student> findTop5ByOrderByIdAsc();  
List<Student> findTop5ByOrderByIdDesc();
```

Spring Data keywords **Top** and **First** handle limits automatically.

16) Group by employee name and get highest salary from each group

```
@Query("SELECT e.name, MAX(e.salary) FROM Employee e GROUP BY e.name")  
List<Object[]> findTopSalaryByEmployee();
```

For DTO projection:

```
@Query("SELECT new com.example.dto.EmpSalaryDto(e.name, MAX(e.salary)) FROM Employee e  
GROUP BY e.name")  
List<EmpSalaryDto> getHighestSalaryPerEmployee();
```

17) How do you design DB for your services? What if DB goes down?

Design approach:

- Normalize up to 3NF for OLTP systems.
- Use foreign keys and indexes wisely.
- For microservices → each service owns its DB (no cross-service DB).
- Add audit & soft delete columns (**is_active**, timestamps).
- For failover → use replicas and connection pool failover configs.

If DB down:

- Use **circuit breaker** to handle failures gracefully.
 - Implement **retry** and **fallback** mechanisms.
 - For critical systems → DB replication (master-slave) and read replicas.
 - Use cache to serve recent reads until DB recovers.
-

18) Why should I go for MongoDB?

Reasons:

- Schema-less, flexible JSON-like data.
- Handles hierarchical data easily.
- High scalability via sharding.

- Fast writes and reads for unstructured data.
- Ideal for event logging, IoT, and real-time analytics.

Use Mongo when:

- Schema changes frequently.
 - You need document-oriented storage.
 - You need horizontal scalability.
-

19) Aware of any columnar database? Why should we go for it?

Yes – e.g., **Amazon Redshift**, **ClickHouse**, **Apache Cassandra**, **HBase** (hybrid).
Columnar DBs store data by columns instead of rows.

Why:

- Faster for analytical queries (aggregations, OLAP).
 - Better compression (columnar data types are similar).
 - Ideal for BI, data warehouses.
-

20) Approaches to handle SQL exceptions in Spring Boot & CRUD best practices

Approaches:

- Use `@Transactional` to rollback on exceptions.
- Global exception handling via `@ControllerAdvice`.
- Catch and map `DataAccessException` (Spring wraps JDBC exceptions).
- Use custom exception hierarchy for API responses.

CRUD best practices:

- Validate inputs before DB calls.
 - Return proper HTTP status (`201 Created`, `404 Not Found`).
 - Use DTOs instead of exposing entities.
 - Use optimistic locking (`@Version`).
 - Paginate all list APIs.
-

21) How do you handle joins in Spring Data?

Options:

JPQL Join

```
@Query("SELECT e FROM Employee e JOIN e.department d WHERE d.name = :name")  
List<Employee> findByDeptName(String name);
```

Fetch Join – eager load related entity:

`@Query("SELECT e FROM Employee e JOIN FETCH e.department")`

Projection Join Result – map partial data to DTO.

22) Best way to design your data access layer?

Pattern:

- **Entity:** Domain objects (JPA or Document)
- **Repository layer:** Interfaces + custom queries
- **Service layer:** Business logic + transactions
- **DTO layer:** Input/Output models

Practices:

- Keep repositories thin.
 - No business logic inside repositories.
 - Use `@Transactional` in service layer.
 - Use DTO projection for performance.
-

23) What kind of cache are you using? Benefits?

Common choices:

- **Redis** (preferred for distributed)
- **Caffeine** or **Ehcache** (local)

Benefits:

- Reduces DB load.
- Faster response time.
- Supports TTL for stale data eviction.
- Ideal for frequently read static data (config, user profiles, etc.)

Example:

```
@Cacheable("employee")
public Employee findById(Long id) { ... }

@CacheEvict(value="employee", key="#id")
public void deleteById(Long id) { ... }
```

24) How to customize connection pool behavior in Spring Boot 3.3?

Spring Boot 3.3+ uses HikariCP by default.

Example configuration:

```
spring:
  datasource:
    hikari:
      maximum-pool-size: 20
      minimum-idle: 5
      idle-timeout: 30000
      max-lifetime: 1800000
      connection-timeout: 20000
```

You can also tune via:

```
@Bean
@ConfigurationProperties("spring.datasource.hikari")
HikariDataSource dataSource() { return new HikariDataSource(); }
```

Advanced tuning:

- Use `leakDetectionThreshold` for detecting connection leaks.
- Monitor with Micrometer metrics.

25) How to add a composite key in Spring Boot?

Already covered under **Composite keys** → use `@EmbeddedId` or `@IdClass`.
Spring Boot 3.3+ uses `jakarta.persistence.*` imports.

26) How to write custom queries based on field name (examples)?

```
List<Employee> findByNameStartingWith(String prefix);
List<Employee> findBySalaryBetween(double min, double max);
List<Employee> findByDepartmentIn(List<String> depts);
List<Employee> findByActiveTrue();
List<Employee> findByJoinDateAfter(LocalDate date);
List<Employee> findTop3ByOrderBySalaryDesc();
```

Rules:

- Use entity field names.
 - Combine multiple conditions using **And/Or**.
 - Use comparison keywords (**LessThan**, **GreaterThanOrEqualTo**).
-

27) What is N+1 problem in Hibernate?**Definition:**

When fetching a parent entity triggers multiple queries (1 for parent + N for each child).

Example:

```
List<Department> deps = repo.findAll();  
// then deps.getEmployees() triggers N queries
```

Solutions:**Use **JOIN FETCH****

```
@Query("SELECT d FROM Department d JOIN FETCH d.employees")
```

Use **@EntityGraph:**

```
@EntityGraph(attributePaths = "employees")  
List<Department> findAll();
```

Tune batch size:

```
spring.jpa.properties.hibernate.default_batch_fetch_size=20
```

28) Migrating from MySQL to MongoDB — code changes required**Replace dependency:**

```
<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-data-mongodb</artifactId>  
</dependency>
```

1. Remove JPA annotations (`@Entity`, `@Table`) → use `@Document`.
2. Replace `JpaRepository` → `MongoRepository`.
3. Update query methods (Mongo supports JSON-based queries and aggregation).
4. Remove SQL-related properties, add Mongo properties:

`spring.data.mongodb.uri: mongodb://localhost:27017/mydb`

5. Rewrite native queries to use aggregation pipelines.
6. No transactions by default (except in replica sets).

Note: Domain and repository logic remains similar — only persistence annotations and repository types change.

KAFKA

1) What is Apache Kafka, and what are its core components?

Imagine a **factory** where hundreds of sensors constantly produce data — temperature, speed, pressure. Instead of directly sending all this data to the monitoring system (which might overload it), we introduce a **message broker** in the middle — that's what **Apache Kafka** does.

Kafka is a **distributed event streaming platform** that lets you publish, store, and consume streams of records in real-time — just like a durable “event log” that different systems can read at their own pace.

Core components:

- **Producer:** Sends messages (events) to Kafka topics.
- **Topic:** A logical category (like a folder) where messages are stored.
- **Partition:** Each topic is split into smaller parts (partitions) for scalability.
- **Broker:** Kafka server that stores and serves data.
- **Consumer:** Reads messages from topics.
- **Consumer Group:** A set of consumers working together to consume messages in parallel.
- **ZooKeeper/KRaft (Controller):** Coordinates brokers and metadata (in latest Kafka, KRaft replaces ZooKeeper).

Kafka is like a “distributed commit log” — you can replay data anytime, ensuring durability, fault tolerance, and high throughput.

2) What is Kafka Connect, and how is it used for integrating with external systems?

Kafka Connect acts like a **bridge** between Kafka and external data systems (databases, file systems, cloud storage, etc.).

Instead of writing custom code to move data between Kafka and a database, Kafka Connect provides **ready-made connectors**.

Example:

- **Source Connector:** Reads data from a database (like MySQL) and sends it to a Kafka topic.
- **Sink Connector:** Reads data from Kafka and writes it into an external system (like Elasticsearch).

Use case:

Suppose you want to replicate your MySQL data changes to Elasticsearch in real-time for searching. You use:

- **Debezium MySQL Source Connector** → to capture DB changes.
- **Elasticsearch Sink Connector** → to write those changes to ES.

So, Kafka Connect simplifies integration — **no need for custom producers or consumers.**

3) What is log compaction in Kafka, and when would you use it?

Story:

Imagine you store user profile updates in Kafka — name changes, address changes, etc. You don't care about every update forever; you just want the **latest state** of each user.

That's where **log compaction** helps.

In log compaction:

- Kafka retains only the **latest message for each key**.
- Older messages with the same key are removed during compaction.
- The topic behaves like a **key-value store** with the most recent data.

Use case:

- Maintaining the current balance per account.
- Latest configuration per user or device.

Example:

Key=User1 | Value=Name:John

Key=User1 | Value=Name:John Smith ← older record compacted

This ensures Kafka topics never grow infinitely while keeping the most recent data version.

4) Explain the difference between a topic, partition, and segment.

Think of it like a **book**:

- **Topic** = The entire book (e.g., "OrderEvents").
- **Partition** = Each chapter in the book. Topics can have many partitions.
- **Segment** = Each partition is split into smaller files (segments) on disk.

Purpose:

- **Partitions** help scale horizontally — multiple consumers can process data in parallel.
 - **Segments** help manage storage — old segments can be deleted or compacted efficiently.
-

5) What are Kafka consumer groups, and how do they affect message consumption?

A **consumer group** is a team of consumers that together consume messages from a topic.

Story:

If a topic has 3 partitions and 3 consumers in a group, each consumer will read from one partition.

If one consumer fails, another takes over its partition automatically.

Key idea:

- Each partition is read by only **one consumer** in the group at a time (ensuring no duplication).
- Multiple consumer groups can independently consume the same topic.

This allows **parallel processing** while maintaining **partition-level ordering**.

6) How does Kafka ensure message ordering?

Messages are ordered **within a partition**, not across partitions.

Example:

If you send user events with **key=userId**, all events of that user go to the same partition → they stay in order.

So, Kafka ensures ordering **per key (or per partition)**, not globally.

Best practice: Use keys wisely if your business logic depends on order (e.g., transactions, status updates).

7) How does Kafka ensure high throughput and scalability?

Kafka is built for performance:

- **Partitioning:** Distributes data load across brokers.
- **Batching:** Producers send messages in batches.
- **Sequential disk writes:** Uses append-only logs (no random I/O).
- **Zero-copy transfer:** Uses OS-level optimization for fast network transfers.
- **Replication:** Ensures fault tolerance across brokers.

These combined make Kafka handle **millions of messages per second** even on modest hardware.

8) What is a consumer group in Kafka?

A consumer group is a **named collection of consumers** that share the workload of consuming messages from a topic.

- Each consumer in the group reads from unique partitions.
- Multiple groups can consume the same topic independently.

Example:

- **GroupA** (analytics app)
 - **GroupB** (billing app)
- Both can be consumed from **order-topic** without interfering.
-

9) How do you achieve exactly-once semantics (EOS) in Kafka?

Exactly-once semantics (EOS) means every message is processed **once and only once** — even if retries or failures occur.

Kafka achieves this with:

1. **Idempotent producers:** Avoids duplicate writes during retries.
2. **Transactions:** Combines producer send + consumer offset commit atomically.
3. **Transactional APIs:** (**transactional.id**, **enable.idempotence=true**)

Example use case:

If a payment event is processed, you don't want to charge the customer twice — EOS ensures that.

10) What is Kafka's replication factor, and why is it important?

Replication factor = number of copies of each partition stored on different brokers.

If replication factor = 3:

- One broker is **leader**, two are **followers**.
- Followers replicate data from the leader.

If one broker fails, another follower becomes leader automatically.

This ensures **data durability** and **high availability**.

11) How does Kafka achieve fault tolerance?

Kafka replicates each partition across multiple brokers.

If the **leader broker** fails, a **follower replica** takes over as leader (through ZooKeeper or KRaft controller).

Clients automatically reconnect to the new leader, ensuring minimal disruption.

In short:

Replication + Leader Election = Fault Tolerance.

12) How do Kafka producers handle retries and failures?

Producers have configurations like:

- **retries**: how many times to retry sending.
- **acks**: acknowledgment setting:
 - **acks=0**: no wait for acknowledgment.
 - **acks=1**: leader acknowledgment only.
 - **acks=all**: waits for all in-sync replicas (safe but slower).

Idempotent producers prevent duplicates during retries.

Example config:

```
props.put("enable.idempotence", "true");  
props.put("acks", "all");  
props.put("retries", 5);
```

13) Explain Kafka's partitioning strategy and how it impacts performance.

Partitioning decides which partition a message goes to.

You can:

- Use **key-based partitioning** (default) — ensures order per key.
- Use **round-robin** — evenly distributes messages.
- Write **custom partitioner** for advanced routing.

Impact:

- More partitions → better parallelism.
- But too many → overhead in coordination and offset tracking.

Rule:

Start with a moderate number (e.g., 6–12) and scale based on throughput.

14) What is Kafka retention policy, and how does it work?

Kafka doesn't delete messages immediately after consumption.

Instead, it retains them based on **time** or **size**:

```
retention.ms=604800000 # 7 days  
retention.bytes=1073741824 # 1GB
```

After this limit, old log segments are deleted (or compacted if enabled).
Consumers can re-read data anytime until retention expires.

15) Describe Kafka's consumer offset management.

Every consumer tracks its **offset** — the position up to which it has read in each partition.

Kafka stores offsets in a special topic: `__consumer_offsets`.

Types:

- **Automatic commit:** Kafka commits offsets periodically.
- **Manual commit:** Application explicitly commits after successful processing (safer).

If a consumer restarts, it resumes from its last committed offset — no data loss.

16) How can Kafka handle back-pressure in real-time data processing?

When consumers are slower than producers:

- Kafka brokers queue up messages (disk-based, so safe).
- Consumers can **pause/resume** partitions temporarily.
- Stream frameworks like **Kafka Streams** or **Flink** auto-handle back-pressure using flow control.

Best practice:

- Monitor consumer lag.
 - Scale consumers horizontally when lag increases.
-

17) Explain exactly-once semantics (EOS) again (simplified).

$\text{EOS} = \text{No duplicates} + \text{No data loss} + \text{Atomic processing}$.

Kafka achieves it using:

- Idempotent producers.
- Transactions across send + commit offset.
- Consumer commits offset only after successful transaction.

This ensures one clean, atomic data pipeline.

18) How would you monitor and optimize Kafka performance in production?

Monitoring tools:

- **Prometheus + Grafana**
- **Confluent Control Center**
- **Kafka Manager**
- **Datadog**

Key metrics:

- Consumer lag
- Request latency
- Disk I/O
- Under-replicated partitions
- Broker CPU/memory

Optimization tips:

- Tune `batch.size`, `linger.ms`, `compression.type`.
 - Balance partition count per broker.
 - Increase replication for critical topics.
 - Keep message size small (<1 MB recommended).
-

19) Describe how Kafka handles leader election for partitions.

Each partition has:

- **One leader**
- **Multiple followers**

If leader broker fails:

- ZooKeeper/KRaft picks a new leader from the in-sync replicas (ISR).
- Clients automatically detect the new leader.

This ensures seamless failover.

20) What are the challenges of using Kafka in a multi-datacenter setup?

Challenges:

- **Replication lag** across data centers.
- **High latency** in cross-region replication.
- **Network partitioning** risks message duplication.
- **Consistency vs Availability** trade-offs.

Solutions:

- Use **MirrorMaker 2.0** for cross-cluster replication.
 - Apply **idempotent writes** and **deduplication**.
 - Keep local clusters for low-latency reads.
-

21) How does Kafka handle fault tolerance at the broker level?

If a broker fails:

- Its partitions' replicas on other brokers are promoted.
- Kafka clients retry automatically.
- When the broker rejoins, it syncs data and re-joins ISR.

No single broker failure can bring the system down.

22) How would you design a Kafka-based system to guarantee data consistency in event of failures?

Approach:

- Use **replication factor = 3**
- **acks=all** on producer
- **enable.idempotence=true**
- Consumer commits offset **after** successful processing
- Enable **EOS** transactions

This ensures both **durability** and **exactly-once** delivery — even during broker failures.

23) What are the possible ways to tune Kafka?

- **Producer configs:**
 - `batch.size`, `linger.ms`, `compression.type`, `acks`
- **Consumer configs:**
 - `fetch.min.bytes`, `max.poll.records`, `auto.offset.reset`
- **Broker configs:**
 - `num.network.threads`, `log.retention.ms`, `replica.fetch.max.bytes`
- **Disk tuning:**
 - Use SSDs, dedicate disks for logs.
- **Network tuning:**

- Enable compression, optimize MTU size.
-

24) How to ensure consumers don't consume duplicate data?

- Enable **idempotent producer**.
 - Commit offsets **only after** successful processing.
 - Use **transactions** if you need atomicity.
 - Use **unique keys** to deduplicate in downstream systems.
-

25) What are Kafka publishing strategies?

1. **Fire and Forget:** Fast, but no acknowledgment (**acks=0**).
2. **Synchronous send:** Waits for broker acknowledgment.
3. **Asynchronous send:** Sends in background using callbacks.

Best practice: **Async send + retries + idempotence**.

26) What are the strategies for partitioning?

- **Round-robin:** Evenly distributes data (no ordering).
 - **Key-based:** Consistent ordering for same key.
 - **Custom:** Implement your own logic (e.g., based on region or customer type).
-

27) Implement Kafka using multithreading

Create one thread per partition:

- Each thread uses its own consumer.
- Ensure thread safety by not sharing consumer instance.

Example:

```
ExecutorService executor = Executors.newFixedThreadPool(partitions);
for (int i = 0; i < partitions; i++) {
    executor.submit(new KafkaConsumerThread(topic, groupId, i));
}
```

28) What serializers are required? Can you customize?

Kafka needs **Serializer** (producer) and **Deserializer** (consumer).

Common types:

- `StringSerializer`
- `ByteArraySerializer`
- `JsonSerializer` (Spring Kafka)
- Custom serializer (e.g., Avro, Protobuf)

Custom Example:

```
public class EmployeeSerializer implements Serializer<Employee> {  
    public byte[] serialize(String topic, Employee data) {  
        return new ObjectMapper().writeValueAsBytes(data);  
    }  
}
```

29) What are Kafka quotas?

Kafka allows setting **quotas** to control producer/consumer throughput and prevent abuse.

```
kafka-configs.sh --alter --add-config 'producer_byte_rate=1048576' --entity-type clients --entity-name client1
```

Use for **multi-tenant environments** to avoid one client overloading brokers.

30) How to track data loss in Kafka?

- Monitor **consumer lag**.
 - Enable **delivery and commit logs**.
 - Use tools:
 - **Burrow**
 - **Confluent Control Center**
 - **Cruise Control**
 - Compare producer and consumer message counts periodically.
-

31) Why do you need Kafka? Best practices in design

You need Kafka when:

- You have **real-time data streams**.
- Systems must **communicate asynchronously**.
- You need **event replayability** or **audit logs**.

Best practices:

- Keep messages small.
 - Use schema registry (Avro/Protobuf).
 - Use keys wisely.
 - Monitor lag and partition distribution.
 - Enable idempotence and replication.
-

32) What if ZooKeeper went down? Can we avoid ZooKeeper? What's the difference with KRaft?

Old Kafka relied on **ZooKeeper** for metadata (brokers, topics).

From **Kafka 3.0+**, **KRaft (Kafka Raft mode)** replaces ZooKeeper.

KRaft benefits:

- No external dependency.
- Faster startup and leader election.
- Simpler deployment and monitoring.

So yes — new Kafka (KRaft mode) doesn't need ZooKeeper anymore.

33) How to make Kafka concurrent?

- Increase partitions → more parallelism.
 - Create multiple consumers in the same group → parallel read.
 - Use **multithreaded processing** after polling messages.
 - Ensure consumer per partition for ordered processing.
-

34) What is Kafka offset reset? How to customize?

When a consumer joins with no prior offset, Kafka checks:

`auto.offset.reset = earliest | latest | none`

- **earliest**: Start from beginning.
 - **latest**: Start from new messages only.
 - **none**: Throws error if no offset found.
-

35) Can multiple consumers poll from same partition? How does ordering work?

Within a **consumer group**, **NO** — only one consumer per partition.
Across different groups — **YES**, each group can independently consume.

Ordering is guaranteed **within each partition** only.

36) How does Kafka handle message durability?

- Data written to disk immediately (append-only logs).
- Replicated to multiple brokers.
- Acknowledged to producer only after successful replication (acks=all).

Kafka ensures data isn't lost even if a broker crashes.

37) What is Kafka's ISR (In-Sync Replicas), and how does it work?

ISR = Set of replicas that are fully caught up with the leader.

- Leader sends data to followers.
- Followers acknowledge.
- If a follower lags behind, it's removed from ISR.
- Only ISR members can become the next leader.

Ensures data consistency and safe failover.

38) What are common challenges in scaling a Kafka cluster, and how to handle them?

Challenges:

- Too many small partitions → overhead.
- Disk I/O bottlenecks.
- Network congestion.
- Uneven partition distribution.
- Consumer lag under heavy load.

Solutions:

- Monitor and rebalance partitions.
 - Use SSDs, increase broker count.
 - Tune producer/consumer batch sizes.
 - Implement autoscaling and back-pressure logic.
 - Use MirrorMaker for horizontal scaling.
-

System Design

1. Design an Amazon-like Order Management & Inventory System

Problem Statement

You need to design a simplified **Amazon-like system** where:

- Customers can **place orders** for products.
- The system must **check inventory** in real time before confirming.
- Each order must have a **unique, pattern-based order ID**, that encodes:
 - **Location**
 - **Zone**
 - **User type** (Prime, Regular, Guest, etc.)

The system should be **scalable, event-driven, and consistent**.

Solution Design

We'll follow **Domain-Driven Design (DDD)** and **Microservice Architecture**.

Core Microservices

Service	Responsibility
API Gateway	Entry point for customers and support team. Handles auth, routing, throttling.
Order Service	Handles order creation, validation, order ID generation, and persistence.
Inventory Service	Manages stock levels, reserves inventory during order placement.
Payment Service	Handles transaction flow between customer and system.
Notification Service	Sends order confirmation via email/SMS.

User Service	Stores user info, location, and type.
Support Dashboard (Internal)	Allows support team to view order status and inventory.

Order Flow (Step-by-Step)

1. **Customer places order** through web/mobile → **API Gateway**.
 2. **Order Service**:
 - Validates request.
 - Calls **User Service** to get user type and zone.
 - Generates **pattern-based Order ID**.
 - Publishes **OrderCreated** event to Kafka.
 3. **Inventory Service** consumes event:
 - Checks availability.
 - Reserves items (atomic operation).
 - Publishes **InventoryReserved** event.
 4. **Payment Service** consumes **InventoryReserved**:
 - Charges customer.
 - Publishes **PaymentSuccessful** event.
 5. **Order Service** updates status = *Confirmed*.
 6. **Notification Service** sends confirmation.
 7. **Support System** fetches from read replica or caches via **CQRS model** for faster access.
-

Order ID Generation Pattern

Format Example:

<RegionCode>-<ZoneCode>-<UserTypeCode>-<Timestamp>-<RandomSuffix>

Example:

IN-NE-P-20251026-AB12

- **IN** = India
- **NE** = NorthEast Zone
- **P** = Prime User
- **20251026** = Date
- **AB12** = Random 4-char hash

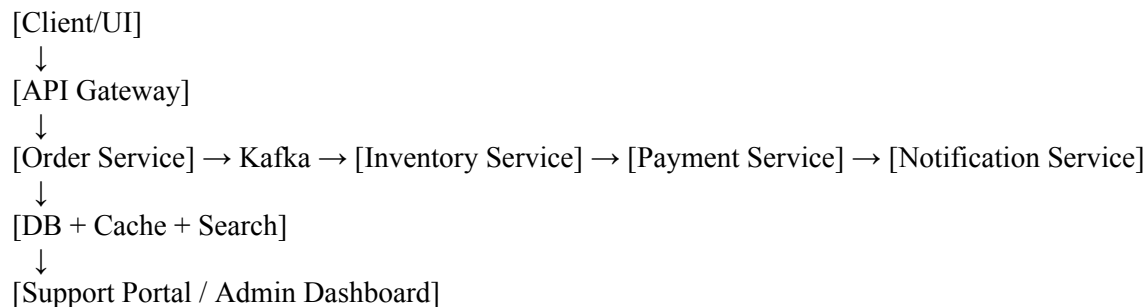
Implementation could use:

- **Snowflake ID** or **ULID** for distributed unique ID.
- Include **metadata prefix** using **user & zone info**.

Pseudo-code Example:

```
String generateOrderId(User user) {  
    String region = user.getRegionCode();  
    String zone = user.getZoneCode();  
    String type = user.isPrime() ? "P" : "R";  
    String timestamp = LocalDate.now().format(DateTimeFormatter.BASIC_ISO_DATE);  
    String random = RandomStringUtils.randomAlphanumeric(4).toUpperCase();  
    return String.format("%s-%s-%s-%s-%s", region, zone, type, timestamp, random);  
}
```

Architecture Overview



- **Asynchronous communication:** Kafka ensures decoupling and reliability.
 - **Databases:**
 - Orders: PostgreSQL (strong consistency)
 - Inventory: MongoDB/Redis (fast updates)
 - Analytics: Elasticsearch (for dashboards)
-

Best Practices

- ✓ Use **CQRS** (Command Query Responsibility Segregation) for fast reads.
- ✓ Use **Kafka** or **Pulsar** for event-driven decoupling.
- ✓ Enable **Idempotency keys** to prevent double orders.
- ✓ Secure APIs with OAuth2 and JWT.
- ✓ Use **Redis caching** for product & inventory lookups.

- ✓ Apply **Circuit Breaker (Resilience4j)** for fault tolerance.
 - ✓ Use **Observability stack**: ELK + Prometheus + Grafana.
-
-

2. Design a Phone System with User, Partner, and RBI Transactions

Problem Statement

You must design a **PhonePe/Paytm-like** system where:

- **User transactions** (P2P transfers) happen **instantly**.
- **Partner transactions** (e.g., merchants) are **settled every 15 days**.
- **RBI transactions** (bank settlements) are done on **scheduled configuration times** (e.g., nightly batch).

The system must ensure **consistency**, **security**, and **fault-tolerant transaction handling**.

Solution Design

Core Components

Service	Role
Transaction API Gateway	Accepts transactions, authenticates users.
User Transaction Service	Handles instant P2P transactions.
Partner Settlement Service	Aggregates and settles partner transactions every 15 days.
RBI Reconciliation Service	Transfers funds to/from RBI on scheduled times.
Ledger Service	Maintains double-entry accounting for every transaction.
Scheduler Service	Cron/Quartz-based scheduling for settlements.
Notification Service	Sends confirmations and reports.
Kafka/Event Bus	Ensures async, reliable message delivery between services.

⚙️ Transaction Flow

◆ User Transaction (Real-Time)

1. User sends money via app.
2. API Gateway → User Transaction Service.
3. Validates balance from Ledger Service.
4. Debits sender, credits receiver atomically.
5. Publishes **UserTxnCompleted** to Kafka.
6. Notification Service sends confirmation instantly.

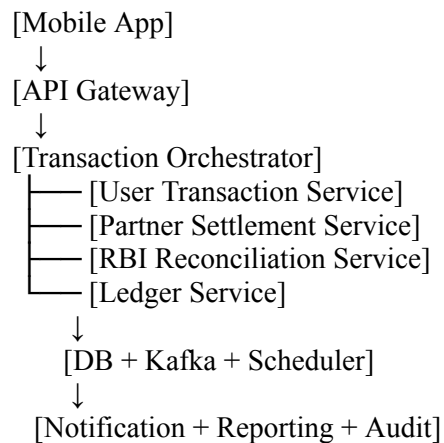
◆ Partner Transaction (T+15)

1. Partner transactions stored daily in **PartnerTxn DB**.
2. **Scheduler Service** runs a 15-day job.
3. Aggregates and triggers **Partner Settlement Service**.
4. Deducts platform fees, generates reports.
5. Publishes settlement confirmation to Kafka.

◆ RBI Transaction (Scheduled)

1. RBI Reconciliation Service uses **cron config** for schedule.
2. Transfers net amount via secure RBI API.
3. Updates Ledger and publishes **RBISettlementCompleted** event.

🗺️ Architecture Diagram (Text Form)



Data Model

Ledger Table:

TxnID	From	To	Amount	Type	Status	Timestamp
TXN001	U1	U2	500	USER_TXN	SUCCESS	2025-10-26

PartnerTxn Table:

PartnerID	TxnDate	Amount	Status
-----------	---------	--------	--------

RBI Table:

| BatchID | Amount | SettlementDate | Status |

Best Practices

- ✓ Use **event-driven** async processing (Kafka).
 - ✓ Implement **Saga Pattern** for distributed transactions (compensating actions on rollback).
 - ✓ Use **Retry Queues** and **Dead Letter Queues** for failed transactions.
 - ✓ Apply **encryption-at-rest** and **TLS-in-transit**.
 - ✓ Use **idempotent keys** for transaction requests.
 - ✓ Keep **audit trails** for every movement (immutable ledger).
 - ✓ Apply **rate limiting and throttling** at gateway level.
 - ✓ Monitor via **Prometheus + Grafana dashboards**.
-

Key Patterns Used

- **Event Sourcing + CQRS** → for ledger accuracy.
 - **Saga Pattern** → to manage distributed consistency.
 - **Scheduler Pattern** → for partner and RBI settlements.
 - **Outbox Pattern** → ensures event + DB transaction atomicity.
-

1. I want to design an Amazon-like service where you can place your order as a customer and I can check the inventory in the support system. how will you design the system? can you generate pattern specific random id for order id where order id should be justifying the order location, zone and user type.

Solution:

Problem Statement

Imagine you're asked to build a simplified **Amazon-like service**. Customers should be able to:

- Browse products,
- Place orders,
- Track their orders.

On the other side, support staff should be able to:

- View inventory in real-time,
- Check stock availability,
- Monitor order flow.

Each order must have a **unique and meaningful Order ID** that encodes:

- **Location** of the order,
- **Zone** of the warehouse or user region,
- **User type** (like Prime, Regular, or Guest).

Example:

IN-NW-P-20251026T153045-04-F9H2K3

Here:

- **IN** → Country (India)
- **NW** → Zone (North-West)
- **P** → User type (Prime)
- The rest → Timestamp and unique hash

So when someone sees the Order ID, they can immediately tell where it came from and what kind of user placed it.

You also need to ensure that —

- ✓ Orders are processed reliably,
- ✓ Inventory updates are consistent,
- ✓ System can scale to handle millions of orders,

- ✓ Support staff see data in near real time,
 - ✓ The architecture is **fault-tolerant**, **event-driven**, and **future-proof**.
-

The Story — Designing an Amazon-like Ordering System

Let's imagine a story:

It's Monday morning. You are the **lead architect** at a large e-commerce company that wants to become the next Amazon.

The CEO says:

“We want customers to buy seamlessly and support teams to see everything instantly. But we can't afford downtime or slow responses.”

You smile — because you know this is the perfect chance to design something elegant.

You start drawing on a whiteboard and break the problem into **independent services**.

Step 1: Identify the Core Services

You divide the system into **microservices**, each responsible for one business capability.

Service	Responsibility	Technology Example
API Gateway	Entry point for all requests (auth, routing, throttling)	Spring Cloud Gateway, Kong
User Service	Handles user details, zones, and types (Prime, Guest)	Spring Boot + MySQL
Catalog Service	Manages product details, pricing, availability	Spring Boot + MongoDB
Inventory Service	Keeps track of stock per warehouse, updates after orders	Spring Boot + PostgreSQL
Order Service	Handles order creation, updates, and ID generation	Spring Boot + PostgreSQL
Payment Service	Integrates with payment gateways to charge users	Spring Boot + Stripe SDK
Notification Service	Sends confirmation emails and SMS	Spring Boot + Kafka

Support Service (Read Model)	Provides a fast searchable view for support team	Spring Boot + Elasticsearch
Outbox & Event Stream	Kafka-based messaging system to sync all services asynchronously	Kafka or RabbitMQ
Audit Service	Stores event trails for order changes	Elastic + Logstash

Step 2: Understand the Flow (User Story)

Let's follow a customer named **Ananya** who lives in Bangalore (South Zone). She opens the app and orders a phone.

- Ananya places the order.**
 - Her request comes through the **API Gateway**.
 - The **Order Service** receives it, validates user and product details via **User Service** and **Catalog Service**.
- Generate the Order ID.**
 - The Order Service uses the user's region (IN), zone (SOUTH), and user type (P for Prime).
 - It creates an Order ID: **IN-S-P-20251026T100455-09-ULIDHASH**.
- Persist and Publish.**
 - The order is saved in the **Order database**.
 - An **OrderCreated** event is stored in an **Outbox table** and later published to **Kafka**.
- Inventory Check.**
 - Inventory Service** listens to **OrderCreated**.
 - If stock exists, it reserves the items and emits **InventoryReserved**.
 - Otherwise, it emits **InventoryFailed**.
- Payment Handling.**
 - Payment Service** consumes **InventoryReserved** events.
 - It processes the payment and emits **PaymentSucceeded** or **PaymentFailed**.
- Order Confirmation.**
 - The **Order Service** listens again.
 - On success, it marks the order as **CONFIRMED** and emits an **OrderConfirmed** event.
- Support Sync.**
 - The **Support Service** consumes events like **OrderCreated**, **InventoryReserved**, **OrderConfirmed**.
 - It updates its **Elasticsearch read model**, giving near-real-time visibility to the support team.
- Notification.**

- Finally, the **Notification Service** sends an SMS and email confirmation to Ananya:
“Your order IN-S-P-20251026T100455-09-ULIDHASH has been placed successfully!”

9. **Support Staff View.**

- A support agent can query by Order ID and instantly view the live status and remaining stock.
-

Step 3: The Core Components

1. API Gateway

- Validates tokens and routes requests to backend services.
- Adds trace IDs for observability.

2. Order Service

- Generates pattern-based Order IDs.
- Saves orders, writes events to Outbox.
- Publishes **OrderCreated**, **OrderConfirmed**, **OrderCancelled**.

3. Inventory Service

- Consumes events.
- Manages available vs reserved stock.
- Publishes reservation events.

4. Payment Service

- Handles transactions.
- Publishes payment success/failure.

5. Support Read Model

- Maintains a fast, denormalized view.
- Powers support dashboards and search.

6. Notification Service

- Sends user communications.

7. Kafka Event Bus

- Backbone for async communication between services.

8. Databases

- Order DB (Postgres) for transactions.
 - Inventory DB (Postgres/Redis).
 - Elasticsearch for support queries.
 - Redis for caching hot data.
-



Step 4: The Architecture Diagram (Mermaid)

```
flowchart LR
    subgraph Client
        U[User App]
        S[Support Dashboard]
    end

    subgraph Gateway
        G[API Gateway]
    end

    subgraph CoreServices
        O[Order Service]
        I[Inventory Service]
        P[Payment Service]
        N[Notification Service]
        Su[Support Service]
        C[Catalog Service]
        U1[User Service]
    end

    subgraph Data
        D1[(Order DB)]
        D2[(Inventory DB)]
        D3[(Elastic Search)]
        K[(Kafka Topics)]
    end

    U --> G
    G --> O
    O --> D1
    O --> K
    K --> I
    K --> P
    K --> Su
    K --> N
    I --> D2
    Su --> D3
```

S --> Su
O --> C
O --> U1

style O fill:#cce5ff,stroke:#004085,stroke-width:1px
style I fill:#d4edda,stroke:#155724,stroke-width:1px
style P fill:#f8d7da,stroke:#721c24,stroke-width:1px
style Su fill:#fff3cd,stroke:#856404,stroke-width:1px
style K fill:#f7f1e3,stroke:#856404,stroke-width:1px

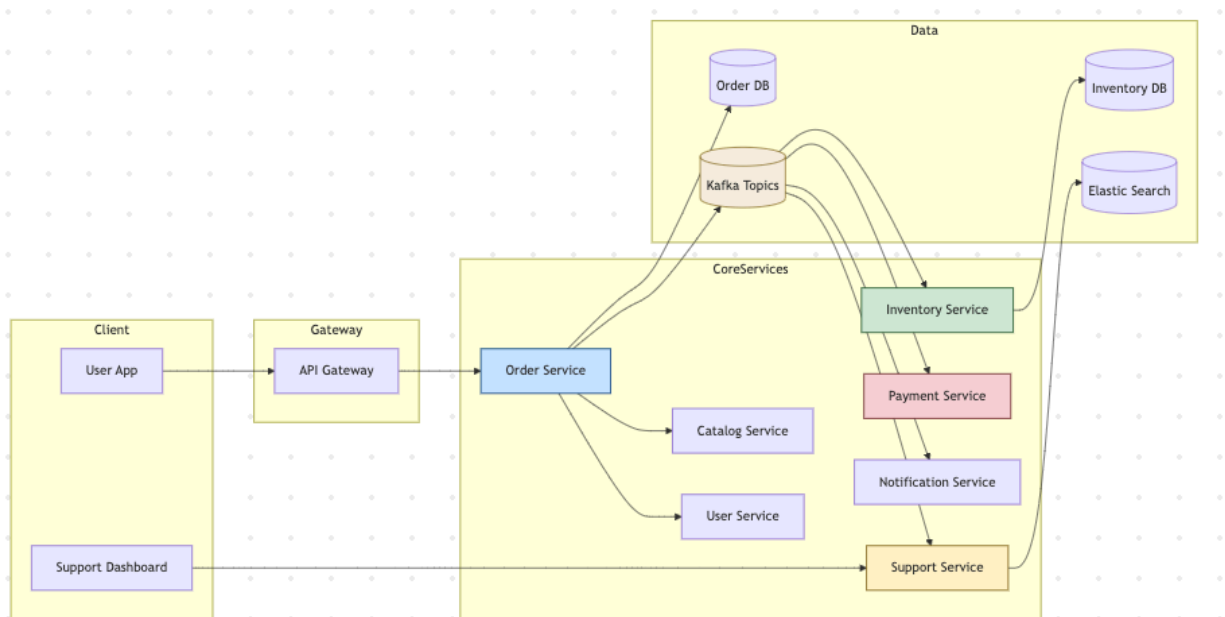
Go to Mermaid live editor and paste above content:

https://mermaid.live/edit#pako:eNqFk9GOojAUhl-FdG6cRA1VoMjFJrOwMcaNmmW92bIXHdoKGaSkIHFc47tvFRRmxjiEEL7-_U57GjiAWFAGPMAzsYsTIpXx81eUG_oqq-eNJEVi-FnKclUPnq41XpdMGk9F8bcdDHFYFYXQfkDK5FkQSZuU5TTKP5ScEsV2ZN_qU_y0ml2G74i-kCxx8jWNWdnaS7yUVG-pSTrbmuFZ_qp3L-T-RrrCK7Lf6vhGtsALoVKexkSIr8xIayulX8OfewTRTKxuZGtYX2A75Nb3Oa6RusFEPfqPoPvj516wOj32i4_ZGPc-5GRUqWxXo_IOOmme9ybE_5CjN-iSOPy8fNW1sZg8M2Y1jA9w7KG5RkC2KV5DfMzzLqw6kJYdWIRw6yuN6oprGoeN_jOqxfsu7CG18NT-4zpUZ5mmfcQx8zmvF8qKV6Y92CalunaDQ52KVWJB4u3rjlTGoxSsnVhLaNRtZdc9WY3KWoy6IRjL8wdbeNyyk4plfVtR3LvK_OLybikI2_NkEfbGRKgadkxfpgy-SWnBACtIUjoBK2ZRHw9CtlnFSZikCUH7VWkPyPENuLKUW1SYDHSVZqqgqq_90gJfrDbafod4IJX1S5At7kXAF4B_AGPBcOLWhDx4Km6ejb7oM98KBjD6EzgSbST4QcGx374N95TXOIJshxlbJc10Hm2Joc_wOzt2Zr

OR

<https://www.eraser.io/ai/mermaid-diagram-editor>

You can use any online diagram editor and you can generate it.



Step 5: Key Best Practices

1. **CQRS (Command-Query Responsibility Segregation):**
 - Writes happen in transactional DB.
 - Reads happen in Elasticsearch for speed.
 2. **Outbox Pattern:**
 - Ensures DB changes and events are atomic.
 3. **Event-Driven Communication (Kafka):**
 - Decouples services and supports scalability.
 4. **Idempotency Keys:**
 - Avoid duplicate order creation.
 5. **SAGA Pattern:**
 - Manages distributed transactions (Order → Inventory → Payment).
 6. **Structured Order IDs:**
 - Encodes metadata for quick debugging and tracking.
 7. **Monitoring and Logging:**
 - Use **Prometheus**, **Grafana**, and **ELK** for observability.
 8. **Fault Tolerance:**
 - Retries, DLQs (Dead Letter Queues), and fallback mechanisms.
-

Summary

You have now built an **Amazon-like scalable, fault-tolerant system** with:

- Pattern-based meaningful Order IDs
 - Event-driven microservices (Order, Inventory, Payment)
 - Real-time support dashboard using Elasticsearch
 - Consistent and decoupled data flow with Kafka
 - Future-ready patterns (Saga, Outbox, CQRS)
-