Technical Interview Preparation Guide

Entry-Level

---

1. Object-Oriented Programming (OOP) Principles

1.1 What are the four pillars of OOP?

Encapsulation: Bundling data and methods that operate on that data within a single unit (class), hiding internal implementation details.

- *Example*: A `BankAccount` class that keeps the balance private and provides public methods like `deposit()` and `withdraw()` to modify it safely.

Inheritance: Mechanism where a new class inherits properties and behaviors from an existing class.

- *Example*: A `Vehicle` parent class with properties like `speed` and `color`, and child classes like `Car` and `Motorcycle` that inherit these and add specific features.

Polymorphism: Ability of objects to take multiple forms. The same method name can behave differently based on the object calling it.

- *Example*: A `Shape` interface with a `calculateArea()` method. `Circle`, `Rectangle`, and `Triangle` classes implement this method differently.

Abstraction: Hiding complex implementation details and showing only essential features.

- *Example*: When you drive a car, you use the steering wheel and pedals (abstraction) without needing to know how the engine works internally.

1.2 What is a Class?

A blueprint or template for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of that type will have.

1.3 What is an Object?

An instance of a class. It's a concrete entity created from the class blueprint, with actual values for the attributes defined in the class.

---

2. SOLID Principles

2.1 Single Responsibility Principle (SRP)

A class should have only one reason to change, meaning it should have only one job or responsibility.

Example: Instead of having a `User` class that handles user data, email sending, and database operations, split it into:

- `User` (data model)
- `EmailService` (sending emails)
- `UserRepository` (database operations)

Real Project Example: In an e-commerce application, separate `OrderProcessor` (handles order logic) from `PaymentGateway` (handles payment processing) from `InventoryManager` (manages stock).

2.2 Open/Closed Principle (OCP)

Software entities should be open for extension but closed for modification.

Example: Instead of modifying a `PaymentProcessor` class every time you add a new payment method, create a `PaymentMethod` interface and implement it with `CreditCardPayment`, `PayPalPayment`, etc.

```
interface PaymentMethod {
    void processPayment(double amount);
}

class CreditCardPayment implements PaymentMethod {
    public void processPayment(double amount) { /* ... */ }
}

class PayPalPayment implements PaymentMethod {
    public void processPayment(double amount) { /* ... */ }
}
```

2.3 Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of a subclass without breaking the application.

Example: If you have a `Bird` class with a `fly()` method, a `Penguin` subclass violates LSP because penguins can't fly. Better design: create separate `FlyingBird` and `FlightlessBird` classes.

2.4 Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they don't use.

Example: Instead of one large `Worker` interface with methods `work()`, `eat()`, `sleep()`, create smaller interfaces:

- `Workable` with `work()`
- `Eatable` with `eat()`
- `Sleepable` with `sleep()`

Then classes implement only what they need.

2.5 Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Example: Instead of `OrderService` directly creating a `MySQLDatabase` object, inject a `Database` interface:

```
class OrderService {
    private Database database;

    public OrderService(Database database) {
        this.database = database; // Dependency injection
    }
}
```

---

3. Design Patterns

3.1 Creational Patterns

Singleton: Ensures a class has only one instance and provides a global access point.

- *Example*: Database connection pool, logger, configuration manager

```
public class DatabaseConnection {
    private static DatabaseConnection instance;

    private DatabaseConnection() {}

    public static DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }
}
```

Factory: Creates objects without specifying the exact class to create.

- *Example*: Creating different types of notifications (Email, SMS, Push) based on user preference

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

public class NotificationFactory {
    public static Notification createNotification(String type) {
        switch(type) {
            case "EMAIL": return new EmailNotification();
            case "SMS": return new SMSNotification();
            default: throw new IllegalArgumentException();
        }
    }
}
```

Builder: Constructs complex objects step by step.

- *Example*: Building a `House` object with optional features like garage, pool, garden

```
House house = new House.Builder()
    .setRooms(4)
    .setGarage(true)
    .setPool(true)
    .build();
```

3.2 Structural Patterns

Adapter: Allows incompatible interfaces to work together.

- *Example*: Adapting a legacy payment system to work with a modern interface

```
// Legacy system
class OldPaymentGateway {
    void makePayment(String amount) { /* ... */ }
}

// Modern interface
interface PaymentProcessor {
    void processPayment(double amount);
}

// Adapter
class PaymentAdapter implements PaymentProcessor {
    private OldPaymentGateway oldGateway;

    public void processPayment(double amount) {
        oldGateway.makePayment(String.valueOf(amount));
    }
}
```

Decorator: Adds new functionality to objects dynamically.

- *Example*: Adding features to a coffee order (milk, sugar, whipped cream) without modifying the base `Coffee` class

Facade: Provides a simplified interface to a complex subsystem.

- *Example*: A `HomeTheaterFacade` that simplifies operations like turning on TV, DVD player, and sound system with one `watchMovie()` method

3.3 Behavioral Patterns

Observer: Defines a one-to-many dependency where when one object changes state, all dependents are notified.

- *Example*: Newsletter subscription system where subscribers are notified when new content is published

```
interface Observer {
    void update(String news);
}

class NewsAgency {
    private List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers(String news) {
        for (Observer observer : observers) {
            observer.update(news);
        }
    }
}
```

Strategy: Defines a family of algorithms and makes them interchangeable.

- *Example*: Different sorting strategies (QuickSort, MergeSort, BubbleSort) that can be swapped at runtime

Template Method: Defines the skeleton of an algorithm, letting subclasses override specific steps.

- *Example*: A `DataProcessor` abstract class with steps: `readData()`, `processData()`, `saveData()`, where subclasses provide specific implementations

---

4. Additional Software Engineering Principles

4.1 DRY (Don't Repeat Yourself)

Avoid code duplication. Extract common logic into reusable functions or classes.

Bad Example:

```
// Calculating discount in multiple places
double price1 = basePrice * 0.9;
double price2 = otherPrice * 0.9;
```

Good Example:

```
double calculateDiscount(double price) {
    return price * 0.9;
}
```

4.2 GRASP (General Responsibility Assignment Software Patterns)

Information Expert: Assign responsibility to the class that has the information needed to fulfill it.

- *Example*: `Order` class calculates its own total since it has access to order items

Creator: Class A should create instances of Class B if A contains, aggregates, or closely uses B.

- *Example*: `ShoppingCart` creates `Order` objects

Controller: Assign responsibility for handling system events to a controller class.

- *Example*: `OrderController` handles HTTP requests for order operations

Low Coupling: Minimize dependencies between classes.

- *Example*: Use interfaces instead of concrete classes

High Cohesion: Keep related functionality together in a class.

- *Example*: All user authentication logic in `AuthenticationService`

---

5. Java Fundamentals

5.1 String Handling

Q: Difference between `String a = "test"` and `new String("test")` ?

A: `String a = "test"` uses the string pool (string interning). If "test" already exists in the pool, `a` will reference that existing string. `new String("test")` always creates a new object in heap memory, even if "test" exists in the pool.

```
String s1 = "hello";
String s2 = "hello";        // s1 == s2 is true (same reference)
String s3 = new String("hello");  // s1 == s3 is false (different objects)
```

Q: When to use StringBuilder?

A: Use `StringBuilder` for multiple string concatenations, especially in loops. String is immutable, so each concatenation creates a new object, which is inefficient.

```
// Bad - creates many String objects
String result = "";
for (int i = 0; i < 1000; i++) {
    result += i;
}

// Good - modifies one StringBuilder object
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    sb.append(i);
}
String result = sb.toString();
```

5.2 Hash and Equals Contract

Q: What is the contract between hashCode() and equals()?

A:

1. If two objects are equal according to `equals()`, they must have the same `hashCode()`
2. If two objects have the same `hashCode()`, they are NOT necessarily equal
3. If you override `equals()`, you must override `hashCode()`

Q: If two objects have the same hash are they equal? If two objects are equal, are their hashes the same?

A:

- Same hash → NOT necessarily equal (hash collision)
- Equal objects → MUST have the same hash (contract requirement)

```
class Person {
    String name;
    int age;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && Objects.equals(name, person.name);
    }

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

5.3 Exceptions

Q: What are exceptions in Java?

A: Events that disrupt normal program flow. They are objects that represent error conditions.

Q: How many types are there?

A:

1. **Checked Exceptions**: Must be caught or declared (e.g., `IOException`, `SQLException`)
2. **Unchecked Exceptions**: Runtime exceptions (e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`)
3. **Errors**: Serious problems that applications shouldn't catch (e.g., `OutOfMemoryError`, `StackOverflowError`)

Q: How to handle exceptions?

A:

```
try {
    // Code that might throw exception
    int result = 10 / 0;
} catch (ArithmeticException e) {
    // Handle specific exception
    System.out.println("Cannot divide by zero");
} catch (Exception e) {
    // Handle general exception
    System.out.println("An error occurred");
} finally {
    // Always executes (cleanup code)
    System.out.println("Cleanup");
}
```

5.4 Final vs Finally vs Finalize

final:

- Variable: Cannot be reassigned
- Method: Cannot be overridden
- Class: Cannot be extended

finally:

- Block that executes after try-catch, regardless of whether an exception occurred
- Used for cleanup (closing files, connections)

finalize:

- Deprecated method called by garbage collector before object is destroyed
- Not recommended; use try-with-resources or explicit cleanup instead

```
final int MAX = 100;           // Constant
final class Utility { }        // Cannot be extended

try {
    // code
} finally {
    // cleanup
}

// finalize() - deprecated, don't use
```

---

6. REST APIs

6.1 What is REST?

REST (Representational State Transfer) is an architectural style for designing networked applications. It uses HTTP methods to perform CRUD operations on resources identified by URLs.

Key Principles:

- Stateless: Each request contains all necessary information
- Client-Server: Separation of concerns
- Cacheable: Responses can be cached
- Uniform Interface: Consistent way to interact with resources

6.2 HTTP Verbs

- **GET**: Retrieve a resource (read-only, safe, idempotent)
- **POST**: Create a new resource (not idempotent)
- **PUT**: Update/replace an entire resource (idempotent)

- **PATCH**: Partially update a resource (idempotent)
- **DELETE**: Remove a resource (idempotent)
- **HEAD**: Like GET but returns only headers
- **OPTIONS**: Describes communication options for resource

6.3 Authentication vs Authorization

Authentication: Verifying who you are (identity)

- *Example*: Logging in with username and password

Authorization: Verifying what you're allowed to do (permissions)

- *Example*: Admin can delete users, regular user cannot

6.4 PUT vs POST vs PATCH

POST:

- Creates a new resource
- Server typically assigns the ID
- `POST /api/users` → Creates a new user
- Not idempotent (multiple calls create multiple resources)

PUT:

- Updates/replaces entire resource
- Client specifies the ID
- `PUT /api/users/123` → Replaces user 123
- Idempotent (multiple identical calls have same effect as one)

PATCH:

- Partially updates a resource
- Only modifies specified fields
- `PATCH /api/users/123` → Updates specific fields of user 123
- Idempotent

6.5 Idempotency

An operation is idempotent if calling it multiple times produces the same result as calling it once.

- **Idempotent**: GET, PUT, PATCH, DELETE
- **Not Idempotent**: POST

Example:

- `DELETE /api/users/123` - First call deletes user, subsequent calls do nothing (user already deleted)
- `POST /api/users` - Each call creates a new user with a different ID

6.6 Headers

HTTP headers provide metadata about the request or response.

Common Request Headers:

- `Content-Type` : Format of request body (e.g., `application/json`)
- `Authorization` : Credentials for authentication (e.g., `Bearer <token>`)
- `Accept` : Expected response format
- `User-Agent` : Client application information

Common Response Headers:

- `Content-Type` : Format of response body
- `Content-Length` : Size of response body
- `Cache-Control` : Caching directives
- `Location` : URL of newly created resource (with POST)

6.7 What is JWT and How Does It Work?

JWT (JSON Web Token) is a compact, self-contained way to securely transmit information between parties as a JSON object.

Structure: `header.payload.signature`

1. **Header**: Token type and hashing algorithm

   ```
   {"alg": "HS256", "typ": "JWT"}
   ```

2. **Payload**: Claims (user data, expiration)

   ```
   {"sub": "123", "name": "John", "exp": 1516239022}
   ```

3. **Signature**: Verifies token hasn't been tampered with

```
HMACSHA256(base64(header) + "." + base64(payload), secret)
```

How It Works:

1. User logs in with credentials
2. Server validates and creates JWT
3. Client stores JWT (usually in localStorage)
4. Client includes JWT in `Authorization` header for subsequent requests
5. Server verifies JWT signature and extracts user information

6.8 Do All Requests Return Something?

Not necessarily. HTTP status codes indicate the result:

- **200 OK**: Request successful with response body
- **201 Created**: Resource created successfully (often includes `Location` header)
- **204 No Content**: Request successful but no response body (common with DELETE)
- **404 Not Found**: Resource doesn't exist
- **500 Internal Server Error**: Server-side error

Example: `DELETE /api/users/123` might return 204 (no body) or 200 with a confirmation message.

---

7. Spring Boot

7.1 How Do Requests Get to the Controller?

1. **Request arrives** at the server (e.g., `GET /api/users/123`)
2. **DispatcherServlet** (front controller) receives it
3. **HandlerMapping** finds the appropriate controller method based on URL and HTTP method
4. **Controller** method executes
5. **Response** is returned through DispatcherServlet

```
Request → DispatcherServlet → HandlerMapping → Controller → Response
```

7.2 How Do You Annotate a Controller?

```
@RestController  // Combines @Controller + @ResponseBody
@RequestMapping("/api/users")
public class UserController {

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        return userService.findById(id);
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.save(user);
    }

    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User user) {
        return userService.update(id, user);
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        userService.delete(id);
    }
}
```

7.3 Request Param vs Path Variable

@PathVariable: Extracts values from the URL path

```
@GetMapping("/users/{id}")
public User getUser(@PathVariable Long id) { }
// URL: /users/123
```

@RequestParam: Extracts query parameters

```
@GetMapping("/users")
public List<User> getUsers(@RequestParam String name,
                           @RequestParam(required = false) Integer age) { }
// URL: /users?name=John&age=30
```

7.4 What is a Body in a Request?

The data sent with POST, PUT, or PATCH requests, typically in JSON format.

```
@PostMapping("/users")
public User createUser(@RequestBody User user) {
    // user object is deserialized from JSON in request body
    return userService.save(user);
}
```

Request:

```
POST /api/users
Content-Type: application/json

{
  "name": "John Doe",
  "email": "john@example.com",
  "age": 30
}
```

7.5 Bean Scopes

Singleton (default): One instance per Spring container

- Used for stateless beans
- Shared across all requests

Prototype: New instance every time bean is requested

- Used for stateful beans

Request: New instance per HTTP request (web applications)

Session: New instance per HTTP session (web applications)

Application: One instance per ServletContext

```
@Component
@Scope("prototype")
public class PrototypeBean { }
```

7.6 What is Dependency Injection?

A design pattern where objects receive their dependencies from external sources rather than creating them internally.

Benefits:

- Loose coupling
- Easier testing (can inject mocks)
- Better maintainability

Types in Spring:

1. **Constructor Injection** (recommended):

```
@Service
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

2. **Setter Injection**:

```
@Service
public class UserService {
    private UserRepository userRepository;

    @Autowired
    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

3. **Field Injection** (not recommended):

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;
}
```

---

8. Angular

8.1 What is TypeScript?

TypeScript is a superset of JavaScript that adds static typing and other features. It compiles to plain JavaScript.

Benefits:

- Type safety catches errors at compile time
- Better IDE support (autocomplete, refactoring)
- Improved code documentation
- Object-oriented features (interfaces, generics)

```
// JavaScript
function add(a, b) {
    return a + b;
}

// TypeScript
function add(a: number, b: number): number {
    return a + b;
}
```

8.2 What is Angular?

Angular is a TypeScript-based framework for building single-page applications (SPAs). It provides:

- Component-based architecture
- Two-way data binding
- Dependency injection
- Routing
- Form handling
- HTTP client

8.3 Difference Between a Library and a Framework

Library: You call the library's code when you need it

- *Example*: React, Lodash, Moment.js
- You control the flow

Framework: The framework calls your code

- *Example*: Angular, Spring Boot, Django
- Framework controls the flow (Inversion of Control)

Metaphor: Library is like going to a furniture store (you pick what you need). Framework is like buying a house (structure is already there, you fill in the details).

8.4 File Structure in Angular

```
src/
├── app/
│   ├── components/
│   │   └── user/
│   │       ├── user.component.ts       // Logic
│   │       ├── user.component.html     // Template
│   │       ├── user.component.css      // Styles
│   │       └── user.component.spec.ts  // Tests
│   ├── services/
│   │   └── user.service.ts
│   ├── models/
│   │   └── user.model.ts
│   ├── app.component.ts
│   ├── app.module.ts                   // Root module
│   └── app-routing.module.ts           // Routes
├── assets/                             // Images, fonts
├── environments/                       // Config
└── index.html                          // Main HTML
```

8.5 Lifecycle Hooks in Angular

Hooks that allow you to tap into key moments in a component's lifecycle:

1. **ngOnChanges**: Called when input properties change
2. **ngOnInit**: Called once after first ngOnChanges (initialization logic)
3. **ngDoCheck**: Custom change detection
4. **ngAfterContentInit**: After content (ng-content) is initialized
5. **ngAfterContentChecked**: After content is checked
6. **ngAfterViewInit**: After view (template) is initialized
7. **ngAfterViewChecked**: After view is checked
8. **ngOnDestroy**: Cleanup before component is destroyed

```
export class UserComponent implements OnInit, OnDestroy {
    ngOnInit() {
        // Initialize data, subscribe to observables
    }

    ngOnDestroy() {
        // Unsubscribe, clear timers
    }
}
```

8.6 Difference Between ngOnInit and Constructor

Constructor:

- TypeScript feature, not Angular-specific
- Called when component class is instantiated
- Used for dependency injection
- Component's inputs are NOT available yet

ngOnInit:

- Angular lifecycle hook
- Called after first ngOnChanges
- Used for initialization logic
- Component's inputs ARE available

```
export class UserComponent implements OnInit {
    @Input() userId: string;

    constructor(private userService: UserService) {
        // userId is undefined here
    }

    ngOnInit() {
        // userId is available here
        this.userService.getUser(this.userId);
    }
}
```

8.7 Require vs Import

require:

- CommonJS syntax (Node.js)
- Runtime loading
- Dynamic: `const module = require(variableName)`

import:

- ES6 modules
- Compile-time loading
- Static: `import { Component } from '@angular/core'`
- Can be dynamic with `import()`: `const module = await import('./module.js')`

8.8 How to Handle Inputs?

1. Template-Driven Forms (simpler, less control):

```
<input [(ngModel)]="userName" />
```

2. Reactive Forms (recommended, more control):

```
export class UserComponent {
    userForm = new FormGroup({
        name: new FormControl('', Validators.required),
        email: new FormControl('', [Validators.required, Validators.email])
    });

    onSubmit() {
        if (this.userForm.valid) {
            console.log(this.userForm.value);
        }
    }
}
```

```
<form [formGroup]="userForm" (ngSubmit)="onSubmit()">
    <input formControlName="name" />
    <input formControlName="email" />
    <button type="submit">Submit</button>
</form>
```

8.9 RxJS Operators

RxJS is a library for reactive programming using Observables.

map: Transforms emitted values

```
this.http.get<User[]>('/api/users').pipe(
    map(users => users.filter(user => user.active))
);
```

filter: Filters emitted values

```
this.searchInput$.pipe(
    filter(term => term.length >= 3)
);
```

pipe: Chains operators

```
this.data$.pipe(
    filter(x => x > 0),
    map(x => x * 2)
);
```

subscribe: Consumes the observable

```
this.userService.getUsers().subscribe(
    users => console.log(users),        // next
    error => console.error(error),      // error
    () => console.log('completed')      // complete
);
```

8.10 Subject vs BehaviorSubject

Both are special Observables that can multicast to multiple subscribers.

Subject:

- No initial value
- New subscribers don't receive previous values
- Use for events

```
const subject = new Subject<string>();
subject.next('first');
subject.subscribe(val => console.log(val));  // Doesn't log 'first'
subject.next('second');                      // Logs 'second'
```

BehaviorSubject:

- Requires initial value
- New subscribers immediately receive the last emitted value
- Use for state management

```
const behaviorSubject = new BehaviorSubject<string>('initial');
behaviorSubject.next('first');
behaviorSubject.subscribe(val => console.log(val));  // Logs 'first'
behaviorSubject.next('second');                      // Logs 'second'
```

---

9. Databases

9.1 SQL vs NoSQL

SQL (Relational):

- Structured data with fixed schema
- Tables with rows and columns
- ACID compliant
- Uses SQL query language
- Vertical scaling
- *Examples*: PostgreSQL, MySQL, Oracle
- *Use cases*: Banking, e-commerce, applications requiring complex queries

NoSQL (Non-Relational):

- Flexible schema
- Document, key-value, graph, or column-family stores
- Eventually consistent (usually)
- Various query languages
- Horizontal scaling
- *Examples*: MongoDB, Redis, Cassandra, Neo4j
- *Use cases*: Big data, real-time applications, content management

9.2 Types of Keys

Primary Key:

- Uniquely identifies each record in a table
- Cannot be NULL
- Only one per table
- *Example*: `user_id` in Users table

Foreign Key:

- Creates a link between two tables
- References a primary key in another table
- Enforces referential integrity
- *Example*: `user_id` in Orders table references `user_id` in Users table

Candidate Key:

- Column(s) that could serve as a primary key
- Unique and non-null
- *Example*: In Users table, both `user_id` and `email` could be candidate keys

Composite Key:

- Primary key made of multiple columns
- *Example*: `(student_id, course_id)` in Enrollments table

9.3 What is a Foreign Key?

A column that creates a relationship between two tables by referencing the primary key of another table.

```
CREATE TABLE Users (
    user_id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    user_id INT,
    amount DECIMAL,
    FOREIGN KEY (user_id) REFERENCES Users(user_id)
);
```

Referential Integrity: Ensures that foreign key values always point to existing records.

9.4 How to Handle Many-to-Many Relationships?

Use a junction/bridge/helper table that contains foreign keys to both tables.

Example: Students and Courses (a student can enroll in many courses, a course can have many students)

```
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE Courses (
    course_id INT PRIMARY KEY,
    title VARCHAR(100)
);

CREATE TABLE Enrollments (
    student_id INT,
    course_id INT,
    enrollment_date DATE,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES Students(student_id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);
```

9.5 What is an Index and How Does It Work?

An index is a data structure that improves query performance by allowing faster data retrieval.

How it works:

- Similar to a book index
- Stores pointers to data locations
- Typically uses B-tree or hash structures
- Trades storage space and write speed for read speed

```
CREATE INDEX idx_user_email ON Users(email);

-- This query will be faster with the index
SELECT * FROM Users WHERE email = 'john@example.com';
```

Trade-offs:

- ✅ Faster SELECT queries
- ❌ Slower INSERT, UPDATE, DELETE (index must be updated)
- ❌ Additional storage space

9.6 ACID Principles

Properties that guarantee database transactions are processed reliably:

Atomicity:

- Transaction is all-or-nothing
- If any part fails, entire transaction rolls back
- *Example*: Transferring money between accounts — both debit and credit must succeed

Consistency:

- Transaction brings database from one valid state to another
- All constraints and rules are maintained
- *Example*: Account balance cannot be negative after transaction

Isolation:

- Concurrent transactions don't interfere with each other
- Appears as if transactions execute serially
- *Example*: Two people buying the last item simultaneously — only one succeeds

Durability:

- Committed transactions are permanent
- Survive system failures
- *Example*: After payment confirmation, data persists even if server crashes

9.7 HAVING Clause

Used to filter groups after aggregation (with GROUP BY).

Difference from WHERE:

- **WHERE**: Filters rows before grouping
- **HAVING**: Filters groups after aggregation

```
 -- Find departments with more than 5 employees
SELECT department, COUNT(*) as employee_count
FROM Employees
GROUP BY department
HAVING COUNT(*) > 5;

-- Combined WHERE and HAVING
SELECT department, AVG(salary) as avg_salary
FROM Employees
WHERE status = 'active'           -- Filter rows first
GROUP BY department
HAVING AVG(salary) > 50000;       -- Filter groups after
```

---

10. Operating System Basics

10.1 Thread vs Process

Process:

- Independent execution unit
- Has its own memory space
- Heavyweight (expensive to create)
- Isolated from other processes
- Communication between processes is complex (IPC)
- *Example*: Running multiple applications (Chrome, VS Code, Spotify)

Thread:

- Lightweight execution unit within a process
- Shares memory with other threads in same process
- Cheaper to create and manage
- Can directly communicate via shared memory
- *Example*: Multiple tabs in a browser, background tasks in an application

Comparison:

```
Process: [Memory | Thread1 | Thread2 | Thread3]
Process: [Memory | Thread1 | Thread2]
```

10.2 Deadlock vs Livelock

Deadlock:

- Two or more threads wait for each other indefinitely
- System is stuck; no progress is made
- *Example*: Thread A holds Resource 1 and wants Resource 2. Thread B holds Resource 2 and wants Resource 1. Both wait forever.

```
// Deadlock example
Thread1: synchronized(lock1) { synchronized(lock2) { /* work */ } }
Thread2: synchronized(lock2) { synchronized(lock1) { /* work */ } }
```

Livelock:

- Threads keep changing state in response to each other but make no progress
- System appears active but accomplishes nothing
- *Example*: Two people in a hallway keep moving left and right trying to let the other pass, but they keep blocking each other

Key Difference: Deadlock = frozen, Livelock = busy but unproductive

10.3 How to Avoid Deadlock?

Four Coffman Conditions (all must be present for deadlock):

1. **Mutual Exclusion**: Resources cannot be shared
2. **Hold and Wait**: Thread holds resources while waiting for others
3. **No Preemption**: Resources cannot be forcibly taken
4. **Circular Wait**: Circular chain of threads waiting for resources

Prevention Strategies:

1. **Lock Ordering**: Always acquire locks in the same order

```
// Good: Consistent order
synchronized(lock1) {
    synchronized(lock2) {
        // work
    }
}
```

2. **Lock Timeout**: Use tryLock with timeout

```
if (lock1.tryLock(1000, TimeUnit.MILLISECONDS)) {
    try {
        if (lock2.tryLock(1000, TimeUnit.MILLISECONDS)) {
            try {
                // work
            } finally {
                lock2.unlock();
            }
        }
    } finally {
        lock1.unlock();
    }
}
```

3. **Avoid Nested Locks**: Minimize situations where you need multiple locks
4. **Use Higher-Level Concurrency Utilities**: ExecutorService, ConcurrentHashMap, etc.

---

11. Data Structures & Algorithms

11.1 Common Data Structures

Array:

- Fixed size, contiguous memory
- O(1) access by index
- O(n) insertion/deletion (requires shifting)
- *Use case*: When size is known, frequent random access

ArrayList (Dynamic Array):

- Resizable array
- O(1) access, O(1) amortized append
- O(n) insertion/deletion in middle
- *Use case*: Flexible size, frequent access and append

LinkedList:

- Nodes with pointers to next (and previous for doubly-linked)
- O(1) insertion/deletion at ends
- O(n) access by index
- *Use case*: Frequent insertion/deletion, less frequent access

Stack:

- LIFO (Last In, First Out)
- O(1) push, pop, peek
- *Use case*: Function call stack, undo operations, expression evaluation

Queue:

- FIFO (First In, First Out)
- O(1) enqueue, dequeue
- *Use case*: Task scheduling, breadth-first search, buffering

HashMap:

- Key-value pairs
- O(1) average case insert, delete, lookup
- O(n) worst case (with collisions)
- *Use case*: Fast lookups, caching, counting frequencies

HashSet:

- Unique elements
- O(1) average case add, remove, contains
- *Use case*: Eliminating duplicates, membership testing

TreeMap/TreeSet:

- Sorted order (Red-Black Tree)
- O(log n) operations
- *Use case*: Ordered data, range queries

Heap (Priority Queue):

- Complete binary tree
- O(log n) insert, O(log n) delete-min/max
- O(1) get-min/max
- *Use case*: Priority scheduling, finding k largest/smallest elements

Graph:

- Nodes (vertices) and edges
- Adjacency list or adjacency matrix representation
- *Use case*: Social networks, maps, dependencies

Tree:

- Hierarchical structure
- Binary Tree, BST, AVL, B-Tree
- *Use case*: File systems, databases, expression parsing

11.2 Time Complexity (Big O Notation)

O(1) - Constant: Array access, HashMap lookup **O(log n)** - Logarithmic: Binary search, balanced tree operations **O(n)** - Linear: Linear search, array traversal **O(n log n)** - Log-linear: Merge sort, quick sort (average), heap sort **O(n²)** - Quadratic: Bubble sort, selection sort, nested loops **O(2ⁿ)** - Exponential: Recursive Fibonacci, subsets generation **O(n!)** - Factorial: Permutations

Space Complexity: Memory used by an algorithm

11.3 Common Algorithms

Sorting:

- **Bubble Sort**: O(n²) - Compare adjacent elements, swap if needed
- **Selection Sort**: O(n²) - Find minimum, swap with first unsorted
- **Insertion Sort**: O(n²) - Build sorted array one item at a time
- **Merge Sort**: O(n log n) - Divide and conquer, stable
- **Quick Sort**: O(n log n) average, O(n²) worst - Partition around pivot
- **Heap Sort**: O(n log n) - Use heap data structure

Searching:

- **Linear Search**: O(n) - Check each element
- **Binary Search**: O(log n) - Divide sorted array in half repeatedly

```
// Binary Search
int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

Graph Algorithms:

- **BFS (Breadth-First Search)**: Level-by-level traversal, uses queue
- **DFS (Depth-First Search)**: Explores as far as possible, uses stack/recursion
- **Dijkstra's**: Shortest path in weighted graph
- **Topological Sort**: Linear ordering of vertices (DAG)

Dynamic Programming:

- Break problem into overlapping subproblems
- Store results to avoid recomputation
- *Examples*: Fibonacci, longest common subsequence, knapsack problem

---

12. Networking & Protocols

12.1 Client-Server Model

Architecture where clients request resources/services and servers provide them.

Client:

- Initiates requests
- User-facing (browser, mobile app)
- Limited resources

Server:

- Responds to requests
- Centralized resources
- Serves multiple clients simultaneously

Flow:

```
Client → Request → Server
Client ← Response ← Server
```

Types:

- **2-Tier**: Client directly communicates with database server
- **3-Tier**: Client → Application Server → Database Server
- **N-Tier**: Multiple layers (presentation, business logic, data access, database)

12.2 HTTP Protocol

HTTP (HyperText Transfer Protocol) is an application-layer protocol for transferring hypermedia documents.

Characteristics:

- Stateless: Each request is independent
- Text-based protocol
- Request-response model
- Runs over TCP (port 80 for HTTP, 443 for HTTPS)

HTTP Request Structure:

```
GET /api/users HTTP/1.1
Host: example.com
Authorization: Bearer token123
Content-Type: application/json

{"name": "John"}
```

HTTP Response Structure:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 45

{"id": 1, "name": "John", "email": "john@example.com"}
```

Status Codes:

- **1xx**: Informational (100 Continue)
- **2xx**: Success (200 OK, 201 Created, 204 No Content)
- **3xx**: Redirection (301 Moved Permanently, 302 Found, 304 Not Modified)
- **4xx**: Client Error (400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found)
- **5xx**: Server Error (500 Internal Server Error, 503 Service Unavailable)

HTTP Methods: See REST section (6.2)

HTTPS: HTTP over TLS/SSL

- Encrypted communication
- Certificate-based authentication
- Protects against eavesdropping and tampering

12.3 TCP vs UDP

TCP (Transmission Control Protocol):

- Connection-oriented (handshake)
- Reliable delivery (acknowledgments, retransmission)
- Ordered delivery
- Flow control and congestion control
- Slower but guaranteed delivery
- *Use cases*: Web browsing (HTTP), email (SMTP), file transfer (FTP)

UDP (User Datagram Protocol):

- Connectionless
- No guarantees (best-effort delivery)
- No ordering
- No flow control
- Faster but may lose packets
- *Use cases*: Video streaming, gaming, DNS, VoIP

Three-Way Handshake (TCP):

```
Client → SYN → Server
Client ← SYN-ACK ← Server
Client → ACK → Server
[Connection Established]
```

12.4 DNS (Domain Name System)

Translates human-readable domain names to IP addresses.

Flow:

```
User types "google.com"
→ Browser checks cache
→ OS checks cache
→ Query DNS resolver (ISP)
→ Query root DNS servers
→ Query TLD servers (.com)
→ Query authoritative nameserver
→ Returns IP address (142.250.185.46)
→ Browser connects to IP
```

12.5 Load Balancing

Distributes incoming traffic across multiple servers.

Algorithms:

- **Round Robin**: Rotate through servers sequentially
- **Least Connections**: Send to server with fewest active connections
- **IP Hash**: Use client IP to determine server
- **Weighted**: Assign different capacities to servers

Benefits:

- Improved availability (no single point of failure)
- Better performance (distributed load)
- Scalability (add more servers)

---

13. Spring & Dependency Injection

13.1 Spring Framework

Comprehensive framework for enterprise Java applications.

Core Features:

- Dependency Injection (IoC Container)
- Aspect-Oriented Programming (AOP)
- Data Access (JDBC, ORM)
- Transaction Management
- Spring MVC (web framework)
- Spring Boot (convention over configuration)

13.2 Inversion of Control (IoC)

Design principle where control flow is inverted — framework calls your code rather than your code calling libraries.

Traditional:

```
class UserService {
    private UserRepository repo = new UserRepository(); // Tight coupling
}
```

IoC with DI:

```
class UserService {
    private UserRepository repo;

    public UserService(UserRepository repo) { // Injected
        this.repo = repo;
    }
}
```

13.3 Spring Dependency Injection

@Component: Marks class as a Spring-managed bean **@Service**: Specialized @Component for service layer **@Repository**: Specialized @Component for data access layer **@Controller**: Specialized @Component for web controllers **@RestController**: @Controller + @ResponseBody

@Autowired: Tells Spring to inject dependency

```
@Service
public class UserService {
    private final UserRepository userRepository;
    private final EmailService emailService;

    // Constructor injection (recommended)
    public UserService(UserRepository userRepository,
                       EmailService emailService) {
        this.userRepository = userRepository;
        this.emailService = emailService;
    }

    public void registerUser(User user) {
        userRepository.save(user);
        emailService.sendWelcomeEmail(user);
    }
}
```

@Configuration: Defines configuration class with @Bean methods

```
@Configuration
public class AppConfig {
    @Bean
    public DataSource dataSource() {
        return new HikariDataSource();
    }
}
```

@Qualifier: Specifies which bean to inject when multiple candidates exist

```
@Service
public class PaymentService {
    private final PaymentGateway gateway;

    public PaymentService(@Qualifier("stripeGateway") PaymentGateway gateway) {
        this.gateway = gateway;
    }
}
```

13.4 Bean Lifecycle

1. Instantiation
2. Populate properties
3. setBeanName (if BeanNameAware)
4. setBeanFactory (if BeanFactoryAware)
5. setApplicationContext (if ApplicationContextAware)
6. @PostConstruct / afterPropertiesSet
7. Custom init method
8. Bean ready for use
9. @PreDestroy / destroy
10. Custom destroy method

```
@Component
public class MyBean {
    @PostConstruct
    public void init() {
        // Initialization logic
    }

    @PreDestroy
    public void cleanup() {
        // Cleanup logic
    }
}
```

---

14. System Design

14.1 What is System Design?

Process of defining architecture, components, modules, interfaces, and data for a system to satisfy specified requirements.

Key Considerations:

- Scalability
- Reliability
- Availability
- Performance
- Security
- Maintainability

14.2 Scalability

Vertical Scaling (Scale Up):

- Add more resources to existing server (CPU, RAM)
- Limited by hardware constraints
- Single point of failure
- *Example*: Upgrade from 16GB to 64GB RAM

Horizontal Scaling (Scale Out):

- Add more servers
- Unlimited scaling potential
- Requires load balancing
- *Example*: Add 5 more application servers

14.3 Caching

Storing frequently accessed data in faster storage layer.

Levels:

- **Browser Cache**: Store static assets locally
- **CDN**: Cache content geographically close to users
- **Application Cache**: In-memory cache (Redis, Memcached)
- **Database Cache**: Query result caching

Strategies:

- **Cache Aside**: Application checks cache first, loads from DB on miss
- **Write Through**: Write to cache and DB simultaneously
- **Write Behind**: Write to cache, asynchronously write to DB
- **Refresh Ahead**: Proactively refresh cache before expiration

Cache Invalidation:

- **TTL (Time To Live)**: Expire after set time
- **LRU (Least Recently Used)**: Remove least recently accessed items
- **Manual**: Explicitly invalidate on update

14.4 Database Design Considerations

Normalization: Reduce redundancy by organizing data into related tables

- **1NF**: Atomic values, no repeating groups
- **2NF**: No partial dependencies
- **3NF**: No transitive dependencies

Denormalization: Add redundancy for performance (read-heavy systems)

Sharding: Horizontal partitioning of data across multiple databases

- *Example*: Users 1-1000 on DB1, 1001-2000 on DB2

Replication:

- **Master-Slave**: Write to master, read from replicas
- **Master-Master**: Multiple writable nodes

14.5 Message Queues

Asynchronous communication between services.

Benefits:

- Decoupling
- Buffering (handle traffic spikes)
- Reliability (retry failed messages)
- Scalability

Examples: RabbitMQ, Apache Kafka, Amazon SQS

Pattern:

```
Producer → Queue → Consumer
         (Buffer)
```

14.6 Microservices vs Monolith

Monolith:

- Single codebase, single deployment
- Easier to develop initially
- Harder to scale and maintain as it grows

Microservices:

- Multiple independent services
- Each service owns its data
- Independently deployable and scalable
- More complex (distributed systems challenges)

---

15. Testing

15.1 Types of Testing

Unit Testing:

- Test individual components in isolation
- Fast, focused on single function/method
- Mock dependencies
- *Framework*: JUnit, Mockito (Java); Jest (JavaScript)

```
@Test
public void testCalculateDiscount() {
    PricingService service = new PricingService();
    double result = service.calculateDiscount(100, 0.1);
    assertEquals(90, result, 0.01);
}
```

Integration Testing:

- Test interaction between components
- Verify modules work together correctly
- May use real database or services
- *Framework*: Spring Test, TestContainers

```
@SpringBootTest
@AutoConfigureMockMvc
public class UserControllerIntegrationTest {
    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testCreateUser() throws Exception {
        mockMvc.perform(post("/api/users")
            .contentType(MediaType.APPLICATION_JSON)
            .content("{\"name\":\"John\"}"))
            .andExpect(status().isCreated());
    }
}
```

End-to-End (E2E) Testing:

- Test entire application flow from user perspective
- Simulates real user scenarios
- Slow but comprehensive
- *Framework*: Selenium, Cypress, Playwright

Acceptance Testing:

- Verify system meets business requirements
- Often written in business-readable format (BDD)
- *Framework*: Cucumber, SpecFlow

15.2 Black Box vs White Box Testing

Black Box:

- Test functionality without knowing internal structure
- Focus on inputs and outputs
- Tester doesn't need to know code
- *Techniques*: Equivalence partitioning, boundary value analysis

White Box:

- Test internal structures and logic
- Requires knowledge of code
- Aims for code coverage
- *Techniques*: Statement coverage, branch coverage, path coverage

15.3 Test-Driven Development (TDD)

Development approach where tests are written before code.

Cycle:

1. **Red**: Write a failing test
2. **Green**: Write minimum code to pass test
3. **Refactor**: Improve code while keeping tests green

Benefits:

- Better design (testable code)
- Confidence in changes
- Living documentation
- Fewer bugs

15.4 Mocking

Creating fake objects to simulate dependencies in unit tests.

```
@Test
public void testSendWelcomeEmail() {
    // Mock dependency
    EmailService emailService = Mockito.mock(EmailService.class);
    UserService userService = new UserService(emailService);

    User user = new User("John", "john@example.com");
    userService.registerUser(user);

    // Verify interaction
    Mockito.verify(emailService).sendEmail(
        eq("john@example.com"),
        eq("Welcome!"),
        anyString()
    );
}
```

15.5 Code Coverage

Measures how much code is executed during testing.

Metrics:

- **Line Coverage**: % of code lines executed
- **Branch Coverage**: % of decision branches executed
- **Method Coverage**: % of methods called

Tool: JaCoCo, Istanbul, Coverage.py

Note: 100% coverage doesn't guarantee bug-free code, but low coverage indicates insufficient testing.

15.6 Test Pyramid

```
      /\
     /E2E\        ← Few, slow, expensive
    /------\
   /  Integration\  ← Some, moderate speed
  /--------------\
 /   Unit Tests   \ ← Many, fast, cheap
/------------------\
```

Principle: Most tests should be fast unit tests, fewer integration tests, and even fewer E2E tests.

---

16. Additional Topics

16.1 Git Version Control

Common Commands:

```
git clone <url>              # Copy repository
git pull                     # Fetch and merge changes
git add <file>               # Stage changes
git commit -m "message"      # Commit changes
git push                     # Upload to remote
git branch <name>            # Create branch
git checkout <branch>        # Switch branch
git merge <branch>           # Merge branches
git rebase <branch>          # Reapply commits on top of another
```

Workflow:

1. Create feature branch from main
2. Make changes and commit
3. Push branch to remote
4. Create pull request
5. Code review
6. Merge to main

16.2 CI/CD

Continuous Integration: Automatically build and test code when changes are pushed

Continuous Deployment: Automatically deploy to production after passing tests

Pipeline Example:

```
Code Push → Build → Unit Tests → Integration Tests → Deploy to Staging → E2E Tests → Deploy to Production
```

Tools: Jenkins, GitLab CI, GitHub Actions, CircleCI

16.3 Docker Basics

Containerization platform that packages applications with dependencies.

Container: Lightweight, isolated environment

Image: Template for creating containers

Dockerfile: Instructions to build an image

```
FROM openjdk:17
COPY target/app.jar /app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Benefits:

- Consistent environments (dev, test, prod)
- Isolation
- Portability
- Efficient resource usage

16.4 API Design Best Practices

- Use nouns for resources (`/users` , not `/getUsers`)
- Use HTTP methods correctly (GET, POST, PUT, DELETE)
- Version your API (`/api/v1/users`)
- Use proper status codes
- Provide meaningful error messages
- Implement pagination for large datasets
- Use filtering and sorting (`/users?role=admin&sort=name`)
- Secure your API (authentication, authorization)
- Document with OpenAPI/Swagger

16.5 Security Basics

Common Vulnerabilities:

- **SQL Injection**: Use prepared statements/parameterized queries
- **XSS (Cross-Site Scripting)**: Sanitize user input, escape output
- **CSRF (Cross-Site Request Forgery)**: Use CSRF tokens
- **Authentication Issues**: Strong passwords, MFA, secure session management
- **Authorization Issues**: Principle of least privilege, role-based access control

Best Practices:

- HTTPS everywhere
- Input validation
- Secure password storage (bcrypt, Argon2)
- Regular security updates
- Logging and monitoring
- Rate limiting

---

17. Behavioral Interview Tips

- Use STAR method (Situation, Task, Action, Result)
- Prepare examples of challenges overcome
- Show learning and growth mindset
- Be honest about weaknesses and how you're improving
- Ask clarifying questions
- Think out loud during technical problems
- Admit when you don't know something