One stop solution – Java , SpringBoot , SQL , Testing , Git

Let's dive deep into **"1. Core Java Fundamentals"**. This section covers the absolute basics of Java that every developer, regardless of experience level, must know inside out.

---

1. Core Java Fundamentals: The Building Blocks of Java

This section forms the foundation of all Java development. A strong grasp here demonstrates your understanding of how Java works at a fundamental level.

1.1. JVM, JRE, JDK

These three acronyms are often confused but are distinct and crucial for understanding Java's "Write Once, Run Anywhere" philosophy.

- **JVM (Java Virtual Machine):**
    - **What it is:** The heart of Java's platform independence. It's an **abstract machine** that provides a runtime environment in which Java bytecode can be executed.
    - **Function:** It takes the .class (bytecode) files generated by the Java compiler and converts them into machine-specific code that the underlying operating system (Windows, macOS, Linux, etc.) can understand and execute.
    - **Key Responsibilities:**
        - **Loading:** Loads .class files.
        - **Verification:** Verifies the bytecode for security and correctness.
        - **Execution:** Executes the bytecode.
        - **Runtime Environment:** Provides the memory areas (Heap, Stack, Method Area, PC Registers, Native Method Stacks) necessary for program execution.
        - **Garbage Collection:** Manages memory automatically.
        - **JIT (Just-In-Time) Compiler:** A component of the JVM that compiles frequently executed bytecode segments into native machine code at runtime for performance optimization.
    - **Platform Dependent:** While Java source code is platform-independent, the JVM itself is platform-dependent. You need a specific JVM implementation for each operating system (e.g., a Windows JVM, a Linux JVM).
- **JRE (Java Runtime Environment):**
    - **What it is:** A software package that provides the minimum requirements for **running** a Java application. It contains the **JVM** along with the necessary **Java class libraries** (like java.lang, java.util, java.io, etc.) and supporting files.
    - **Function:** If you only want to run Java applications (not develop them), the JRE is sufficient. It contains everything needed to execute compiled Java code.

    - **Contents:** JVM + Java Standard Library (classes).
- **JDK (Java Development Kit):**
    - **What it is:** A complete software development environment for **developing, compiling, and running** Java applications.
    - **Function:** It includes everything in the JRE (JVM + Java Standard Library) PLUS development tools like:
        - **javac (Java Compiler):** Translates Java source code (.java files) into bytecode (.class files).
        - **java (Java Launcher):** Used to launch Java applications.
        - **javadoc:** Generates API documentation from source code comments.
        - **jar:** Archives Java classes and resources into a single JAR file.
        - **Debugger, Profiler, etc.**
    - **Contents:** JRE + Development Tools.
    - **When to use:** If you are a Java developer, you need the JDK.
- **"Write Once, Run Anywhere" Principle:** This principle is achieved because the Java compiler (javac) converts your .java source code into platform-agnostic **bytecode** (.class files). This bytecode can then be executed by any JVM, regardless of the underlying operating system. The JVM acts as a translator, making the bytecode executable on the specific machine.

1.2. Data Types

Java is a strongly typed language, meaning every variable must be declared with a data type.

- **Primitive Data Types:**
    - Store direct values.
    - Are fixed in size and performance.
    - **Integer types:**
        - byte: 8-bit signed integer (-128 to 127)
        - short: 16-bit signed integer (-32,768 to 32,767)
        - int: 32-bit signed integer (most commonly used)
        - long: 64-bit signed integer (for very large numbers, append 'L' or 'l')
    - **Floating-point types:**
        - float: 32-bit single-precision (append 'F' or 'f')
        - double: 64-bit double-precision (default for decimal numbers)
    - **Character type:**

        - char: 16-bit Unicode character (can hold any character, including special symbols)
    - **Boolean type:**
        - boolean: Represents true or false.

- **Object (Reference) Data Types:**
    - Do not store values directly but store **references** (memory addresses) to objects.
    - Examples: String, Array, Scanner, Integer, custom class objects (e.g., MyClass).
    - Default value is null.
    - Occupy space in the Heap memory.

1.3. Variables

Variables are containers for storing data values.

- **Local Variables:**
    - Declared inside a method, constructor, or block.
    - Accessible only within that method, constructor, or block.
    - Must be initialized before use. No default value.
    - Stored on the **Stack** memory.
- **Instance Variables (Non-static Fields):**
    - Declared inside a class but outside any method, constructor, or block.
    - Belong to an **instance** (object) of the class.
    - Each object has its own copy of instance variables.
    - Default values are assigned if not explicitly initialized (e.g., 0 for numeric, false for boolean, null for objects).
    - Stored on the **Heap** memory.
- **Static Variables (Class Variables):**
    - Declared inside a class but outside any method, constructor, or block, using the static keyword.
    - Belong to the **class** itself, not to any specific object.
    - Only one copy exists for the entire class, shared by all instances.
    - Can be accessed directly using the class name (e.g., ClassName.staticVariable).
    - Default values are assigned.
    - Stored in the **Method Area (part of Heap in modern JVMs)**.
- **final Variables:**

- A variable declared final can be assigned a value only once. Its value cannot be changed after initialization.
- For primitive types, final means the value itself is constant.
- For reference types, final means the *reference* itself cannot be changed to point to another object. The *contents* of the object it points to can still be modified (unless the object itself is immutable, like String).

1.4. Operators

Symbols that perform operations on variables and values.

- **Arithmetic Operators:** +, -, *, /, % (modulus)
- **Relational Operators:** == (equality), != (not equal), <, >, <=, >= (return boolean)
- **Logical Operators:** && (logical AND), || (logical OR), ! (logical NOT)
- **Bitwise Operators:** &, |, ^, ~, <<, >>, >>> (operate on individual bits)
- **Assignment Operators:** =, +=, -=, *=, /=, %=, etc.
- **Unary Operators:** +, -, ++ (increment), -- (decrement)
- **Ternary Operator (Conditional Operator):** condition ? valueIfTrue : valueIfFalse; (shorthand for if-else)
- **instanceof Operator:** Checks if an object is an instance of a particular class or interface.
- **Operator Precedence and Associativity:** Important for evaluating expressions correctly. (e.g., multiplication before addition).

1.5. Control Flow Statements

Statements that control the order in which instructions are executed.

- **Conditional Statements:**
  - **if-else:** Executes a block of code if a condition is true, otherwise executes another block.
  - **if-else if-else:** Multiple conditions.
  - **switch:** Allows a variable to be tested for equality against a list of values. Good for multiple choice scenarios. (break is crucial to prevent fall-through).
- **Looping Statements:**
  - **for loop:** Executes a block of code a specific number of times. Ideal when you know the number of iterations.
  - **Enhanced for loop (for-each):** Iterates over arrays or collections without explicitly managing indices. Read-only access.
  - **while loop:** Executes a block of code repeatedly as long as a condition is true. The condition is checked *before* each iteration.

- **do-while loop:** Executes a block of code at least once, and then repeatedly as long as a condition is true. The condition is checked *after* the first iteration.

- **Branching Statements:**
  - **break:** Terminates the nearest enclosing loop or switch statement.
  - **continue:** Skips the current iteration of a loop and proceeds to the next iteration.
  - **return:** Exits the current method and optionally returns a value.

1.6. Strings

In Java, String is a class, not a primitive type. Strings are sequences of characters.

- **String (Immutable):**
  - String objects are immutable, meaning once created, their content cannot be changed.
  - Any operation that appears to modify a String (e.g., concatenation) actually creates a *new* String object.
  - This immutability makes String objects thread-safe and suitable for use as keys in HashMaps.
  - String Pool: Java maintains a "String Pool" in the Heap to optimize memory usage. When you create a String literal (e.g., String s = "hello";), the JVM first checks if "hello" already exists in the pool. If it does, it returns a reference to the existing object; otherwise, it creates a new one and puts it in the pool.
  - Using new String("hello") always creates a new object in the Heap, even if "hello" is in the String Pool.

- **StringBuilder (Mutable, Not Thread-Safe):**
  - Used when you need to perform many modifications to a string.
  - Provides methods like append(), insert(), delete(), reverse().
  - More efficient for string manipulation than String concatenation because it avoids creating many intermediate String objects.
  - **Not thread-safe:** If multiple threads modify a StringBuilder concurrently, it can lead to inconsistent results.

- **StringBuffer (Mutable, Thread-Safe):**
  - Similar to StringBuilder in functionality (mutable).
  - **Thread-safe:** All its public methods are synchronized. This makes it suitable for multi-threaded environments, but it comes with a performance overhead compared to StringBuilder.
  - **When to use:** In a single-threaded environment, StringBuilder is preferred for performance. In a multi-threaded environment where multiple threads might modify the same string, StringBuffer is safer.

1.7. Arrays

A container object that holds a fixed number of values of a single data type.

- **Declaration:** dataType[] arrayName; or dataType arrayName[];
- **Initialization:**
    - int[] numbers = new int[5]; (creates an array of 5 integers, initialized to 0)
    - String[] names = {"Alice", "Bob", "Charlie"}; (initializes with values)
- **Accessing Elements:** Using index (0-based): arrayName[index].
- **length property:** arrayName.length gives the size of the array.
- **Multidimensional Arrays:** Arrays of arrays (e.g., int[][] matrix = new int[3][3];).
- **Limitations:** Fixed size once created. Cannot grow or shrink dynamically.

1.8. Wrapper Classes

Java's primitive data types are not objects. Wrapper classes provide object representations for primitive types.

- **Purpose:**
    - To allow primitives to be treated as objects (e.g., when used in Java Collections Framework, which only stores objects).
    - To provide utility methods (e.g., Integer.parseInt(), Double.toString()).
    - To support null values (primitives cannot be null).
- **Examples:**
    - byte -> Byte
    - short -> Short
    - int -> Integer
    - long -> Long
    - float -> Float
    - double -> Double
    - char -> Character
    - boolean -> Boolean
- **Autoboxing and Unboxing (Java 5+):**
    - **Autoboxing:** Automatic conversion of a primitive type to its corresponding wrapper class object.
        - Integer i = 10; (automatically converts int 10 to Integer object)

- **Unboxing:** Automatic conversion of a wrapper class object to its corresponding primitive type.
    - int j = i; (automatically converts Integer object i to int primitive j)
- This simplifies code by allowing you to mix primitives and wrapper objects in many contexts.

1.9. Exception Handling

A robust way to handle runtime errors in a program. Prevents program termination and allows for graceful recovery.

- **try block:** Contains the code that might throw an exception.
- **catch block:** Catches specific types of exceptions thrown in the try block. You can have multiple catch blocks for different exception types (most specific first).
- **finally block:** Always executes, regardless of whether an exception occurred or was caught. Used for cleanup code (e.g., closing resources like file streams, database connections).
- **throw keyword:** Used to explicitly throw an exception from your code.
- **throws keyword:** Used in a method signature to declare that a method might throw one or more specified checked exceptions. The caller of the method must then either handle or re-declare these exceptions.
- **Checked vs. Unchecked Exceptions:**
    - **Checked Exceptions:**
        - Must be handled explicitly by the programmer (either by a try-catch block or by declaring throws in the method signature).
        - Occur at **compile time**.
        - Examples: IOException, SQLException, ClassNotFoundException.
        - They typically represent external problems that your application can anticipate and recover from.
    - **Unchecked Exceptions (Runtime Exceptions):**
        - Do not need to be handled explicitly.
        - Occur at **runtime**.
        - Examples: NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException.
        - They typically represent programming errors that should be fixed rather than caught (e.g., dividing by zero, accessing an invalid array index).
        - Error (e.g., OutOfMemoryError, StackOverflowError) are also unchecked and typically indicate severe problems that the application cannot recover from.
- **Custom Exceptions:** You can create your own exception classes by extending Exception (for checked) or RuntimeException (for unchecked).

1.10. Access Modifiers

Keywords that set the accessibility (visibility) of classes, fields, methods, and constructors.

- **public:** Accessible from anywhere (within the same class, same package, subclasses, and different packages). Most permissive.
- **private:** Accessible only within the **same class**. Most restrictive. Used for encapsulation.
- **protected:** Accessible within the same package and by subclasses (even if in different packages).
- **Default (No keyword):** Accessible only within the **same package**. Often called "package-private".

1.11. this and super Keywords

- **this keyword:**
    - Refers to the **current object** (the object on which the method is invoked).
    - **Uses:**
        - To differentiate between instance variables and local variables if they have the same name (this.variable = variable;).
        - To invoke the current class's constructor from another constructor in the same class (this(); or this(args);).
        - To pass the current object as an argument to a method.
        - To return the current object from a method.
- **super keyword:**
    - Refers to the **immediate parent class object**.
    - **Uses:**
        - To call the parent class's constructor (super(); or super(args); must be the first statement in a child class constructor).
        - To invoke the parent class's method if it's overridden in the child class (super.methodName();).
        - To access parent class's instance variables if they are hidden by a child class's variables of the same name (super.variableName;).

1.12. static Keyword

A non-access modifier applicable to fields, methods, blocks, and nested classes. It makes a member belong to the **class** itself rather than to any specific instance.

- **static variable:**
    - Also known as a **class variable**.
    - Shared by all instances of the class. Only one copy exists.

    - Initialized once when the class is loaded.
    - Can be accessed directly by the class name (e.g., ClassName.staticVariable).
- **static method:**
    - Belongs to the class, not an object.
    - Can be called directly using the class name (e.g., ClassName.staticMethod();).
    - **Cannot access non-static (instance) variables or non-static methods directly** because they belong to an object which might not exist when the static method is called.
    - **Can only access other static members** (variables and methods).
    - Cannot use this or super keywords.
    - **Can a static method be overridden? No.** Method overriding is a runtime polymorphism concept that applies to instance methods associated with objects. Static methods are resolved at compile time based on the reference type, not the actual object type. This is called "method hiding" if a subclass defines a static method with the same signature.
- **static block:**
    - Used to initialize static variables.
    - Executes only once when the class is loaded into memory.
    - Can have multiple static blocks; they execute in the order they appear.
- **static nested class (Nested Static Class):**
    - A static class defined inside another class.
    - Does not require an outer class instance to be created.
    - Can access only static members of the outer class.

1.13. final Keyword

A non-access modifier used to restrict or prevent changes.

- **final variable:**
    - The value of a final variable can be assigned only once. It becomes a constant.
    - Must be initialized at the time of declaration, in a constructor (for instance final variables), or in a static block (for static final variables).
    - For primitive types, the value itself is constant.
    - For reference types, the reference cannot be changed to point to another object, but the object's internal state (if mutable) can still be modified.
- **final method:**
    - A final method cannot be overridden by subclasses.

    - Used when you want to ensure that a method's implementation remains consistent across all subclasses.

- **final class:**
    - A final class cannot be subclassed (inherited from).
    - Used to prevent inheritance and ensure security or immutability (e.g., String class is final).

1.14. Garbage Collection

Automatic memory management in Java.

- **What it is:** A process that automatically frees up memory occupied by objects that are no longer reachable (referenced) by the running program.
- **JVM Component:** It's a daemon thread running in the background of the JVM.
- **Advantages:**
    - Reduces memory leaks.
    - Frees developers from manual memory management (like malloc()/free() in C++).
- **How it works (Simplified):**
    - Identifies unreachable objects.
    - Reclaims the memory occupied by them.
- **System.gc() / Runtime.getRuntime().gc():** These methods are **hints** to the JVM to run the garbage collector, but there's no guarantee it will run immediately. JVM decides when to run GC based on its algorithms and memory needs.
- **Memory Areas (Heap is where objects live):**
    - **Heap:** Where all objects (instance variables, arrays) are allocated. It's divided into generations (Young Generation, Old Generation).
    - **Stack:** Where method calls, local variables, and primitive data types are stored. Each thread has its own stack.
    - **Method Area (PermGen/Metaspace):** Stores class structures, method data, static variables, etc.
    - **PC Registers:** Stores the address of the next instruction to be executed for each thread.
    - **Native Method Stacks:** For native methods written in other languages (C/C++).
- **Garbage Collection Algorithms (Basic understanding):**
    - **Mark and Sweep:** Marks reachable objects, then sweeps away unreachable ones.
    - **Copying:** Copies live objects from one memory area to another, leaving dead objects behind.

- **Mark-Compact:** Marks live objects, then compacts them to one end of the heap, reducing fragmentation.
- **Generational GC:** Divides the heap into generations (Young, Old) assuming most objects die young. Different algorithms are used for different generations (e.g., Copying for Young, Mark-Compact for Old). Common GCs include Serial, Parallel, CMS, G1, ZGC, Shenandoah.

1.15. Serialization and Deserialization

- **Serialization:**
  - The process of converting an object's state into a byte stream.
  - This byte stream can then be stored in a file, transmitted across a network, or stored in a database.
  - Allows objects to persist or be transferred.
  - To make an object serializable, its class must implement the java.io.Serializable interface (a marker interface, no methods to implement).
  - ObjectOutputStream is used for serialization.
- **Deserialization:**
  - The reverse process: converting a byte stream back into a Java object in memory.
  - ObjectInputStream is used for deserialization.
- **transient keyword:**
  - If you don't want a particular field of an object to be serialized, declare it as transient.
  - When an object is deserialized, transient fields will be initialized to their default values (e.g., 0 for int, null for objects).

1.16. Generics (Java 5+)

- **Purpose:** To provide type safety at compile time and eliminate the need for casting, reducing runtime errors (ClassCastException).
- **Example without Generics:** List list = new ArrayList(); list.add("hello"); list.add(123); String s = (String) list.get(0); Integer i = (Integer) list.get(1); (Here, you have to cast and there's no compile-time check if you add wrong types).
- **Example with Generics:** List<String> list = new ArrayList<>(); list.add("hello"); // list.add(123); // Compile-time error String s = list.get(0); (No casting needed, type safety).
- **Type Erasure:**
  - Generics are primarily a **compile-time** feature in Java.
  - During compilation, generic type information is "erased" and replaced with raw types (e.g., List<String> becomes List).

    - This ensures backward compatibility with older Java versions that didn't support generics.
    - Because of type erasure, you cannot:
        - Create instances of type parameters (e.g., new T()).
        - Use instanceof with type parameters (e.g., obj instanceof T).
        - Create arrays of type parameters (e.g., new T[size]).
- **Wildcards (?):** Used in generic code to relax the restrictions on a variable's type parameter.
    - **Unbounded Wildcard (<?>):** Represents any type. List<?> can hold a list of any type.
    - **Upper Bounded Wildcard (<? extends T>):** Represents T or any subclass of T. You can **read** from such a list, but you cannot **add** elements (except null) because you don't know the exact type. (PECS principle: Producer extends).
    - **Lower Bounded Wildcard (<? super T>):** Represents T or any superclass of T. You can **add** elements of type T or its subtypes to such a list, but you can only read objects as Object type. (PECS principle: Consumer super).

1.17. Java 8 Features (Crucial!)

These significantly changed how Java code is written and are almost always a topic of discussion.

- **Lambda Expressions:**
    - **What they are:** A concise way to represent an anonymous function (a function without a name).
    - **Syntax:** (parameters) -> expression or (parameters) -> { statements; }
    - **Use Case:** Primarily used to implement **functional interfaces**. They simplify code for callbacks and functional programming constructs.
    - **Example:** Collections.sort(list, (s1, s2) -> s1.compareTo(s2));
- **Functional Interfaces:**
    - An interface with exactly **one abstract method**.
    - Annotated with @FunctionalInterface (optional, but good practice).
    - Examples: Runnable, Callable, Comparator, Consumer, Predicate, Function, Supplier.
    - Lambda expressions are instances of functional interfaces.
- **Stream API:**
    - **What it is:** A sequence of elements that supports sequential and parallel aggregate operations.
    - **Declarative Style:** Allows you to express what you want to do (e.g., filter, map, collect) rather than how (iterating manually).

- **Non-terminal (Intermediate) Operations:** Return a new stream. They are lazy and don't process the data until a terminal operation is called. Examples: filter(), map(), flatMap(), sorted(), distinct(), peek().
- **Terminal Operations:** Produce a result or a side-effect, and close the stream. Examples: forEach(), collect(), reduce(), count(), min(), max(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny().
- **Source -> Zero or more Intermediate Operations -> One Terminal Operation**
- **No modification of original data source.** Streams are designed to be immutable.

- **Default Methods in Interfaces:**
  - **What they are:** Methods with an implementation provided directly within an interface, prefixed with the default keyword.
  - **Purpose:** Allows adding new methods to existing interfaces without breaking backward compatibility for classes that implement that interface.
  - **Example:** interface MyInterface { default void myDefaultMethod() { System.out.println("Default implementation"); } }
- **Optional Class:**
  - **What it is:** A container object that may or may not contain a non-null value.
  - **Purpose:** To provide a clear way to represent the absence of a value, helping to avoid NullPointerException (a common runtime error).
  - **Methods:** isPresent(), isEmpty(), get(), orElse(), orElseThrow(), ifPresent().
  - **Example:** Optional<String> name = Optional.ofNullable(getName()); name.ifPresent(System.out::println); String result = name.orElse("Default Name");
- **Date and Time API (java.time package):**
  - **Replacement for:** The old java.util.Date and java.util.Calendar classes (which had issues with thread-safety, mutability, and design flaws).
  - **Key Classes:**
    - LocalDate: Date without time and timezone (e.g., 2025-07-13).
    - LocalTime: Time without date and timezone (e.g., 13:02:53).
    - LocalDateTime: Date and time without timezone (e.g., 2025-07-13T13:02:53).
    - Instant: A point in time on the timeline (machine-readable, often used for timestamps).
    - ZonedDateTime: Date, time, and timezone.
    - Duration: Represents a quantity of time in seconds and nanoseconds (e.g., difference between two Instants).

        - Period: Represents a quantity of time in years, months, and days (e.g., difference between two LocalDates).
    - **Features:** Immutable, thread-safe, clear API.

This detailed breakdown of "Core Java Fundamentals" should give you a very strong foundation for your interview. Remember to not just know definitions but also **how and when to use these concepts** and be able to **provide small code examples**.

You got it! Let's dive deep into **"2. Object-Oriented Programming (OOP)"**. This is arguably the most fundamental paradigm for Java and understanding its principles is crucial for designing robust, maintainable, and scalable applications.

---

2. Object-Oriented Programming (OOP): The Foundation of Java Design

OOP is a programming paradigm based on the concept of "objects," which can contain data and code: data in the form of fields (attributes or properties), and code in the form of procedures (methods). The primary goal of OOP is to increase the flexibility and maintainability of programs.

2.1. Four Pillars of OOPs

These are the core principles that define object-oriented programming. You should be able to explain each thoroughly, provide real-world analogies, and demonstrate them with code.

a. Encapsulation

- **Concept:** The binding of data (attributes/fields) and the methods that operate on that data into a single unit (a class), and keeping them safe from outside interference and misuse. It's about "data hiding" or "information hiding."
- **How it's achieved in Java:**
    - **Declaring instance variables as private:** This prevents direct access from outside the class.
    - **Providing public getter and setter methods:** These "accessor" and "mutator" methods are the controlled ways to read and modify the private data. You can add validation logic within setters.
- **Benefits:**
    - **Data Security:** Protects data from unauthorized access.
    - **Flexibility/Maintainability:** Allows changes to the internal implementation of a class without affecting external code that uses it (as long as the public interface, i.e., getters/setters, remains consistent).
    - **Modularity:** Makes classes self-contained and easier to reuse.
    - **Control over Data:** You can enforce business rules or validation logic when data is set.
- **Example:**

Java

```
class BankAccount {
    private String accountNumber; // Encapsulated data
    private double balance;       // Encapsulated data

    public BankAccount(String accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        if (initialBalance >= 0) { // Validation in constructor
            this.balance = initialBalance;
        } else {
            this.balance = 0; // Or throw an exception
        }
    }

    // Public getter for account number (read-only)
    public String getAccountNumber() {
        return accountNumber;
    }

    // Public getter for balance
    public double getBalance() {
        return balance;
    }

    // Public setter/mutator for deposit with validation
    public void deposit(double amount) {
        if (amount > 0) {
            this.balance += amount;
            System.out.println("Deposited " + amount + ". New balance: " + this.balance);
        } else {
            System.out.println("Deposit amount must be positive.");
```

```
        }
    }

    // Public setter/mutator for withdrawal with validation
    public void withdraw(double amount) {
        if (amount > 0 && this.balance >= amount) {
            this.balance -= amount;
            System.out.println("Withdrew " + amount + ". New balance: " + this.balance);
        } else {
            System.out.println("Invalid withdrawal amount or insufficient balance.");
        }
    }
}

public class EncapsulationDemo {
    public static void main(String[] args) {
        BankAccount myAccount = new BankAccount("123456789", 1000.0);
        // Cannot directly access myAccount.balance = -500; due to private access

        System.out.println("Account Number: " + myAccount.getAccountNumber());
        System.out.println("Current Balance: " + myAccount.getBalance());

        myAccount.deposit(500);
        myAccount.withdraw(200);
        myAccount.withdraw(2000); // Will fail due to validation
    }
}
```

b. Inheritance

- **Concept:** The mechanism by which one class acquires the properties (fields) and behaviors (methods) of another class. It represents an "is-a" relationship (e.g., "A Car *is-a* Vehicle").
- **Keywords:**

    - extends: Used by a class to inherit from another class.
    - implements: Used by a class to implement an interface.
- **Types of Inheritance (in terms of class hierarchy):**
    - **Single Inheritance:** A class inherits from only one direct parent class. (Java supports this for classes).
    - **Multilevel Inheritance:** A class inherits from a class, which in turn inherits from another class (e.g., A -> B -> C).
    - **Hierarchical Inheritance:** Multiple classes inherit from a single parent class.
    - **Multiple Inheritance:** A class inheriting from multiple direct parent classes. **Java does NOT support multiple inheritance for classes** to avoid the "diamond problem" (ambiguity if parent classes have methods with same signature).
    - **Hybrid Inheritance:** A mix of two or more types of inheritance.
- **Benefits:**
    - **Code Reusability:** Child classes don't need to rewrite code already present in the parent.
    - **Extensibility:** New functionalities can be added by extending existing classes.
    - **Polymorphism:** A key enabler for runtime polymorphism.
- **Example:**

Java

```
class Vehicle { // Parent class (Superclass)
    String brand;

    public Vehicle(String brand) {
        this.brand = brand;
    }

    public void honk() {
        System.out.println("Vehicle sound!");
    }

    public void displayBrand() {
        System.out.println("Brand: " + brand);
    }
```

```
}

class Car extends Vehicle { // Child class (Subclass) inherits from Vehicle
    String model;

    public Car(String brand, String model) {
        super(brand); // Calls the parent class constructor
        this.model = model;
    }

    // Method Overriding (runtime polymorphism)
    @Override
    public void honk() {
        System.out.println("Car horn!");
    }

    public void drive() {
        System.out.println(brand + " " + model + " is driving.");
    }
}

class ElectricCar extends Car { // Multilevel inheritance
    boolean autonomous;

    public ElectricCar(String brand, String model, boolean autonomous) {
        super(brand, model);
        this.autonomous = autonomous;
    }

    @Override
    public void honk() {
```

```
        System.out.println("Silent electric car hum!");
    }

    public void charge() {
        System.out.println(brand + " " + model + " is charging.");
    }
}

public class InheritanceDemo {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Camry");
        myCar.displayBrand();
        myCar.honk();  // Calls Car's honk() due to overriding
        myCar.drive();

        ElectricCar tesla = new ElectricCar("Tesla", "Model S", true);
        tesla.displayBrand(); // Inherited from Vehicle
        tesla.honk();        // Calls ElectricCar's honk()
        tesla.charge();
    }
}
```

c. Polymorphism

- **Concept:** "Many forms." It refers to the ability of an object to take on many forms or the ability of a variable to refer to objects of different types at different times. In Java, it primarily means that a single interface can be used for different underlying forms (data types).
- **Types of Polymorphism in Java:**
    1. **Compile-time Polymorphism (Static Polymorphism / Method Overloading):**
        - **Concept:** Occurs when there are multiple methods in the *same class* with the *same name* but *different method signatures* (different number of parameters, different types of parameters, or different order of parameters).
        - The compiler decides which method to call based on the arguments passed at compile time.
        - **Example:**

Java

```
class Calculator {
   public int add(int a, int b) {
      return a + b;
   }
   public double add(double a, double b) { // Overloaded method
      return a + b;
   }
   public int add(int a, int b, int c) { // Overloaded method
      return a + b + c;
   }
}
// Usage:
Calculator calc = new Calculator();
calc.add(5, 10);      // Calls int version
calc.add(5.0, 10.0);  // Calls double version
calc.add(1, 2, 3);    // Calls three-int version
```

2. **Runtime Polymorphism (Dynamic Polymorphism / Method Overriding):**
    - **Concept:** Occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The method signature (name, return type, parameters) must be exactly the same.
    - The decision of which method to call is made at *runtime* based on the actual object type, not the reference type. This is also known as "Dynamic Method Dispatch."
    - **Rules for Overriding:**
        - Method must have the same name, same parameters, and same return type.
        - Access modifier cannot be more restrictive than the overridden method.
        - final or static methods cannot be overridden.
        - Constructors cannot be overridden.
    - **Example (refer to Vehicle and Car example in Inheritance):**

Java

Vehicle v1 = new Vehicle("Generic");

Vehicle v2 = new Car("Honda", "Civic"); // Polymorphic reference

Vehicle v3 = new ElectricCar("BMW", "iX", false); // Polymorphic reference

v1.honk(); // Output: Vehicle sound!

v2.honk(); // Output: Car horn! (Runtime decision based on actual object Car)

v3.honk(); // Output: Silent electric car hum! (Runtime decision based on ElectricCar)

- **Benefits:**
    - **Flexibility:** A single interface (method name) can be used to perform different actions depending on the object.
    - **Code Reusability:** Write generic code that can work with objects of different types, as long as they share a common supertype.
    - **Maintainability:** Easier to add new types without modifying existing code, just extend the base class and override.

d. Abstraction

- **Concept:** The process of hiding the implementation details and showing only the essential features of an object to the outside world. It's about "what" an object does, not "how" it does it.
- **How it's achieved in Java:**
    - **Abstract Classes:**
        - A class declared with the abstract keyword.
        - Cannot be instantiated (you cannot create objects of an abstract class).
        - Can have both **abstract methods** (methods without an implementation, marked abstract) and **concrete (non-abstract) methods**.
        - Can have constructors (which are called by subclass constructors via super()).
        - Can have instance variables and static variables.
        - A subclass of an abstract class must either implement all its abstract methods or also be declared abstract.
        - Represents a partial implementation, serving as a base for specific subclasses.
    - **Interfaces:**
        - A blueprint of a class. All methods in an interface are implicitly public abstract (before Java 8).

- Can only declare constants (implicitly public static final) and abstract methods.
- Cannot have constructors.
- Before Java 8, interfaces could *only* have abstract methods. From Java 8, they can have default and static methods with implementations. From Java 9, they can also have private methods.
- A class implements an interface using the implements keyword and must provide implementations for all its abstract methods (unless the implementing class is abstract itself).
- Represents a contract: any class implementing the interface guarantees to provide the specified behaviors.
- Java supports multiple inheritance of *interfaces* (a class can implement multiple interfaces).

- **Abstract Class vs. Interface (Key Differences):**

Feature	Abstract Class	Interface
Keyword	abstract class	interface
Instantiation	Cannot be instantiated directly	Cannot be instantiated directly
Abstract Methods	Can have zero or more	All methods were implicitly public abstract before Java 8.
		From Java 8, can have default and static methods.
		From Java 9, can have private methods.
Concrete Methods	Can have concrete methods (with body)	Only default and static methods (and private in Java 9) can have bodies.
Variables	Can have instance and static variables	Only public static final variables (constants)
Constructors	Can have constructors	Cannot have constructors

Feature	Abstract Class	Interface
Inheritance	A class extends one abstract class.	A class implements multiple interfaces.
Purpose	Defines a template for subclasses with some common implementation. Good for "is-a" relationships where some common behavior is shared.	Defines a contract for behavior. Good for "can-do" relationships or specifying common functionalities across unrelated classes.

- **When to Use Which:**
    - **Abstract Class:** Use when you want to provide a base class with some common functionality implemented, but also enforce that subclasses implement specific methods. Ideal when you have a strong "is-a" relationship and want to share code.
    - **Interface:** Use when you want to define a contract that multiple unrelated classes can adhere to. Ideal for defining capabilities or behaviors that can be shared across different class hierarchies.
- **Example (Abstraction):**

Java

```
// Abstract Class
abstract class Shape {
   String name;

   public Shape(String name) {
      this.name = name;
   }

   // Abstract method - must be implemented by subclasses
   public abstract double calculateArea();

   // Concrete method - has an implementation
   public void displayInfo() {
      System.out.println("This is a " + name + " shape.");
   }
```

```
}

class Circle extends Shape {
    double radius;

    public Circle(String name, double radius) {
        super(name);
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    double length;
    double width;

    public Rectangle(String name, double length, double width) {
        super(name);
        this.length = length;
        this.width = width;
    }

    @Override
    public double calculateArea() {
        return length * width;
    }
}
```

```
// Interface
interface Drawable {
    void draw(); // Abstract method (implicitly public abstract)
    default void setColor(String color) { // Default method (Java 8+)
        System.out.println("Setting color to " + color);
    }
}

class Triangle implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a triangle.");
    }
}

public class AbstractionDemo {
    public static void main(String[] args) {
        // Shape myShape = new Shape("Generic"); // Compile-time error: Cannot instantiate abstract
class

        Circle circle = new Circle("My Circle", 5.0);
        circle.displayInfo();
        System.out.println("Circle Area: " + circle.calculateArea());

        Rectangle rectangle = new Rectangle("My Rectangle", 4.0, 6.0);
        rectangle.displayInfo();
        System.out.println("Rectangle Area: " + rectangle.calculateArea());

        Triangle triangle = new Triangle();
        triangle.draw();
```

```
        triangle.setColor("Blue"); // Using default method from interface
    }
}
```

2.2. Constructors

- **Concept:** A special type of method used to initialize objects. It's invoked automatically when an object of a class is created using the new keyword.
- **Rules:**
    - Must have the **same name as the class**.
    - Has **no return type** (not even void).
    - Cannot be static, final, abstract, or synchronized.
- **Types of Constructors:**
    - **Default Constructor:** If you don't define any constructor, Java automatically provides a public, no-argument default constructor. It initializes instance variables to their default values.
    - **No-argument Constructor:** A constructor explicitly defined by you that takes no arguments.
    - **Parameterized Constructor:** A constructor that takes arguments. Used to initialize instance variables with specific values provided at object creation.
- **Constructor Overloading:** A class can have multiple constructors with the same name but different parameter lists (just like method overloading).
- **Constructor Chaining:**
    - Using this(): To call another constructor of the *same class*.
    - Using super(): To call a constructor of the *parent class*. super() or super(args) must be the first statement in a child class constructor.
- **Example:**

Java

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

    // No-argument constructor
    public Dog() {
        this("Buddy", 3); // Calls the parameterized constructor
```

```
        System.out.println("Dog object created with default values.");
    }

    // Parameterized constructor
    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("Dog object created: " + name + ", " + age + " years old.");
    }

    // Another overloaded constructor
    public Dog(String name) {
        this(name, 0); // Calls the parameterized constructor with default age
        System.out.println("Dog object created with name only.");
    }

    public void bark() {
        System.out.println(name + " says Woof!");
    }
}

public class ConstructorDemo {
    public static void main(String[] args) {
        Dog dog1 = new Dog(); // Calls no-arg constructor
        dog1.bark();

        Dog dog2 = new Dog("Max", 5); // Calls parameterized constructor
        dog2.bark();

        Dog dog3 = new Dog("Lucy"); // Calls overloaded constructor
        dog3.bark();
```

```
    }
}
```

2.3. equals() and hashCode()

These methods (defined in java.lang.Object) are crucial when dealing with collections, especially HashMap, HashSet, and Hashtable.

- **equals() Method:**
    - **Purpose:** Used to compare two objects for **equality of content** (value equality), not just reference equality (==).
    - **Default Implementation:** In the Object class, equals() behaves like == (compares memory addresses).
    - **Overriding:** You should override equals() whenever you want to define what it means for two objects of your class to be logically equal, based on their content.
    - **Contract (Important!):** If you override equals(), you *must* also override hashCode() to maintain the general contract for the Object.hashCode() method.
- **hashCode() Method:**
    - **Purpose:** Returns an integer hash code value for the object. This hash code is primarily used by hash-based collections (HashMap, HashSet, Hashtable) to determine where to store and retrieve objects efficiently.
    - **Default Implementation:** In Object class, hashCode() typically returns a unique integer for each object based on its memory address.
    - **Contract:**
        1. If two objects are equals() according to the equals() method, then calling the hashCode() method on each of the two objects *must* produce the same integer result.
        2. If two objects are *not* equals(), their hash codes *do not have to be different*. However, different hash codes for unequal objects can improve the performance of hash tables.
        3. The hashCode() result must be consistent: if the object's state (used in equals() comparisons) is not modified, hashCode() must return the same value.
- **Why override both?**
    - If you override equals() but not hashCode(), and you put objects of your class into a HashSet or use them as keys in a HashMap:
        - HashSet might store two "equal" objects at different locations because their default hashCode() values (based on memory address) would be different, leading to duplicate "equal" objects in a set.

        - HashMap might not be able to retrieve a value you stored because the key you use for retrieval (which is equals() to the original key) hashes to a different bucket.
    - HashMap first checks hashCode() to narrow down the search to a specific bucket, then uses equals() to find the exact object within that bucket. If hashCode() values differ for equal objects, the second equals() check will never happen.

- **Example (Overriding equals() and hashCode()):**

Java

```
class Employee {
   int id;
   String name;

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

   // Overriding equals()
   @Override
   public boolean equals(Object o) {
      if (this == o) return true; // Same object reference
      if (o == null || getClass() != o.getClass()) return false; // Null check, type check

      Employee employee = (Employee) o; // Downcast
      return id == employee.id && // Compare 'id' for equality
           name.equals(employee.name); // Compare 'name' for equality
   }

   // Overriding hashCode() - essential if equals() is overridden
   @Override
   public int hashCode() {
      int result = id;
```

```
        result = 31 * result + name.hashCode(); // Prime number (31) for better distribution
        return result;
    }

    @Override
    public String toString() {
        return "Employee{id=" + id + ", name='" + name + "'}";
    }
}

public class EqualsHashCodeDemo {
    public static void main(String[] args) {
        Employee emp1 = new Employee(1, "Alice");
        Employee emp2 = new Employee(1, "Alice"); // Logically equal to emp1
        Employee emp3 = new Employee(2, "Bob");

        System.out.println("emp1 == emp2: " + (emp1 == emp2)); // false (different objects)
        System.out.println("emp1.equals(emp2): " + emp1.equals(emp2)); // true (after override)

        System.out.println("Hash Code of emp1: " + emp1.hashCode());
        System.out.println("Hash Code of emp2: " + emp2.hashCode());
        System.out.println("Hash Code of emp3: " + emp3.hashCode());

        // Demonstrate in HashSet
        java.util.Set<Employee> employeeSet = new java.util.HashSet<>();
        employeeSet.add(emp1);
        employeeSet.add(emp2); // This will NOT be added if equals/hashCode are correctly overridden

        System.out.println("Size of employeeSet: " + employeeSet.size()); // Should be 1
        System.out.println("Set contains emp1: " + employeeSet.contains(emp1)); // true
```

```
        System.out.println("Set contains emp2: " + employeeSet.contains(emp2)); // true (finds emp1 via hash & equals)
    }
}
```

2.4. Composition vs. Inheritance

Two fundamental ways to establish relationships between classes.

- **Inheritance (is-a relationship):**
    - As discussed, one class acquires properties and behaviors of another.
    - **Example:** Car extends Vehicle (A Car *is-a* Vehicle).
    - **Pros:** Code reuse, facilitates polymorphism.
    - **Cons:** Tight coupling between parent and child classes, can lead to rigid hierarchies, "fragile base class problem" (changes in base class can break subclasses).
- **Composition (has-a relationship):**
    - A class contains an object of another class as one of its fields (instance variables).
    - Instead of inheriting, you use an instance of another class to get its functionality.
    - **Example:** Car has an Engine (A Car *has-a* Engine).
    - **Pros:**
        - **Loose Coupling:** Classes are less dependent on each other.
        - **Flexibility:** You can change the component class at runtime or easily swap implementations.
        - **High Reusability:** The component class can be reused independently.
        - **Better Maintainability:** Changes in one class are less likely to impact others.
    - **Cons:** Can sometimes involve more initial setup (delegating calls).
- **When to Prefer Composition over Inheritance:**
    - **"Favor composition over inheritance"** is a common design principle.
    - Use **inheritance** when there's a strong, clear "is-a" relationship and you want to leverage polymorphism and common base class implementations.
    - Use **composition** when a class *has* or *uses* another class's functionality, but isn't necessarily a specialized version of it. It offers greater flexibility and less coupling.
- **Example:**

Java

```
// Composition Example: Car has an Engine

class Engine {
```

```
    public void start() {
        System.out.println("Engine started.");
    }
    public void stop() {
        System.out.println("Engine stopped.");
    }
}

class VehicleComposition {
    private Engine engine; // Vehicle has an Engine

    public VehicleComposition() {
        this.engine = new Engine(); // Engine is part of Vehicle
    }

    public void startVehicle() {
        engine.start(); // Delegating the call
        System.out.println("Vehicle started.");
    }

    public void stopVehicle() {
        engine.stop();
        System.out.println("Vehicle stopped.");
    }
}

public class CompositionDemo {
    public static void main(String[] args) {
        VehicleComposition myVehicle = new VehicleComposition();
        myVehicle.startVehicle();
        myVehicle.stopVehicle();
```

```
    }
}
```

2.5. Design Patterns (Basic knowledge for 3 years experience)

While a full deep dive into all design patterns isn't expected, knowing a few common ones and their purpose demonstrates good design principles.

a. Singleton Pattern

- **Purpose:** Ensures that a class has only one instance and provides a global point of access to that instance.
- **Use Cases:** Logging, configuration management, thread pools, database connection pools, caching.
- **Implementation (Common ways):**
    1. **Lazy Initialization (Thread-safe with synchronized method/block):**

Java

```
class Logger {
    private static Logger instance; // Lazily initialized
    private Logger() {} // Private constructor to prevent external instantiation

    public static synchronized Logger getInstance() { // Thread-safe method
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }
    public void log(String message) {
        System.out.println("Log: " + message);
    }
}
```

    2. **Eager Initialization (Thread-safe, simplest):**

Java

```
class Configuration {
    private static Configuration instance = new Configuration(); // Instantiated at class load time
    private Configuration() {}
```

```
    public static Configuration getInstance() {
        return instance;
    }
    public void loadConfig() {
        System.out.println("Configuration loaded.");
    }
}
```

3. **Double-Checked Locking (More performant thread-safe lazy):**

```
Java
class ConnectionPool {
    private static volatile ConnectionPool instance; // volatile for visibility
    private ConnectionPool() {}

    public static ConnectionPool getInstance() {
        if (instance == null) { // First check: no need to synchronize if already created
            synchronized (ConnectionPool.class) { // Synchronize on class object
                if (instance == null) { // Second check: only create if still null
                    instance = new ConnectionPool();
                }
            }
        }
        return instance;
    }
    public void getConnection() { /* ... */ }
}
```

4. **Static Inner Class (Bill Pugh Singleton - most common and recommended):** This is the most widely accepted approach as it combines lazy loading with thread safety without explicit synchronization.

```
Java
class MySingleton {
    private MySingleton() {}
```

```
    // Static inner class. Not loaded until getInstance() is called.
    private static class SingletonHelper {
        private static final MySingleton INSTANCE = new MySingleton();
    }

    public static MySingleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
    public void doSomething() {
        System.out.println("Singleton is doing something.");
    }
}
```

5. **Enum Singleton (Most robust, handles serialization and reflection attacks):**

Java

```
public enum EnumSingleton {
    INSTANCE; // The single instance

    public void doSomething() {
        System.out.println("Enum Singleton is doing something.");
    }
}
// Usage: EnumSingleton.INSTANCE.doSomething();
```

b. Factory Method Pattern

- **Purpose:** Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. It's about "deferring instantiation to subclasses."
- **Use Cases:** When a class cannot anticipate the type of objects it needs to create, or when a class wants its subclasses to specify the objects to be created.[1]
- **Example:**[2]

Java

```
// Product Interface
```

```
interface Notification {
    void notifyUser();
}

// Concrete Products
class EmailNotification implements Notification {
    @Override
    public void notifyUser() {
        System.out.println("Sending an email notification.");
    }
}

class SMSNotification implements Notification {
    @Override
    public void notifyUser() {
        System.out.println("Sending an SMS notification.");
    }
}

// Creator Abstract Class (or Interface) with Factory Method
abstract class NotificationFactory {
    public abstract Notification createNotification();
}

// Concrete Creators
class EmailNotificationFactory extends NotificationFactory {
    @Override
    public Notification createNotification() {
        return new EmailNotification();
    }
}
```

```
class SMSNotificationFactory extends NotificationFactory {
    @Override
    public Notification createNotification() {
        return new SMSNotification();
    }
}

public class FactoryMethodDemo {
    public static void main(String[] args) {
        NotificationFactory emailFactory = new EmailNotificationFactory();
        Notification email = emailFactory.createNotification();
        email.notifyUser();

        NotificationFactory smsFactory = new SMSNotificationFactory();
        Notification sms = smsFactory.createNotification();
        sms.notifyUser();
    }
}
```

c. Builder Pattern

- **Purpose:** Constructs a complex object step by step. It separates the construction of a complex object from its representation, so the same construction process can create different representations.
- **Use Cases:** When an object has many optional parameters, or when its construction involves multiple steps or requires various configurations. Avoids "telescoping constructors."
- **Example:**

Java

```
class Pizza {
    private String dough;
    private String sauce;
    private String cheese;
    private boolean pepperoni;
```

```
    private boolean mushrooms;

    // Private constructor - only accessible by the Builder
    private Pizza(PizzaBuilder builder) {
        this.dough = builder.dough;
        this.sauce = builder.sauce;
        this.cheese = builder.cheese;
        this.pepperoni = builder.pepperoni;
        this.mushrooms = builder.mushrooms;
    }

    @Override
    public String toString() {
        return "Pizza [dough=" + dough + ", sauce=" + sauce + ", cheese=" + cheese +
                ", pepperoni=" + pepperoni + ", mushrooms=" + mushrooms + "]";
    }

    // Static nested Builder class
    public static class PizzaBuilder {
        private String dough;
        private String sauce;
        private String cheese;
        private boolean pepperoni = false; // Default values
        private boolean mushrooms = false; // Default values

        public PizzaBuilder(String dough, String sauce, String cheese) {
            this.dough = dough;
            this.sauce = sauce;
            this.cheese = cheese;
        }
```

```
        public PizzaBuilder addPepperoni(boolean pepperoni) {
            this.pepperoni = pepperoni;
            return this; // Return builder for method chaining
        }

        public PizzaBuilder addMushrooms(boolean mushrooms) {
            this.mushrooms = mushrooms;
            return this;
        }

        public Pizza build() {
            return new Pizza(this); // Create Pizza object
        }
    }
}

public class BuilderPatternDemo {
    public static void main(String[] args) {
        Pizza veggiePizza = new Pizza.PizzaBuilder("Thin Crust", "Tomato", "Mozzarella")
                                .addMushrooms(true)
                                .build();
        System.out.println(veggiePizza);

        Pizza meatLovers = new Pizza.PizzaBuilder("Thick Crust", "BBQ", "Cheddar")
                                .addPepperoni(true)
                                .build();
        System.out.println(meatLovers);
    }
}
```

d. Observer Pattern

- **Purpose:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Use Cases:** Event handling systems, real-time data updates (e.g., stock tickers), UI components reacting to model changes.
- **Components:**
    - **Subject (Observable):** The object that maintains a list of its dependents (observers) and notifies them of state changes.
    - **Observer:** The object that wants to be notified of changes in the subject.
- **Example (Conceptual, using Java's built-in Observer/Observable is deprecated but good for understanding; often custom interfaces are used):**

Java

```
import java.util.ArrayList;
import java.util.List;

// Subject interface
interface Subject {
   void addObserver(Observer o);
   void removeObserver(Observer o);
   void notifyObservers(String message);
}

// Concrete Subject
class NewsAgency implements Subject {
   private List<Observer> observers = new ArrayList<>();
   private String news;

   public void setNews(String news) {
      this.news = news;
      notifyObservers(news); // Notify all subscribed observers
   }

   @Override
```

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

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}

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

// Concrete Observers
class NewsChannel implements Observer {
    private String channelName;

    public NewsChannel(String channelName) {
        this.channelName = channelName;
    }

    @Override
```

```
    public void update(String news) {
        System.out.println(channelName + " received news: " + news);
    }
}

public class ObserverPatternDemo {
    public static void main(String[] args) {
        NewsAgency agency = new NewsAgency();

        NewsChannel channel1 = new NewsChannel("Channel Alpha");
        NewsChannel channel2 = new NewsChannel("Channel Beta");
        NewsChannel channel3 = new NewsChannel("Channel Gamma");

        agency.addObserver(channel1);
        agency.addObserver(channel2);

        agency.setNews("Breaking: New tech company launched!");
        // Output:
        // Channel Alpha received news: Breaking: New tech company launched!
        // Channel Beta received news: Breaking: New tech company launched!

        agency.removeObserver(channel1);
        agency.addObserver(channel3);

        agency.setNews("Market trends indicate growth.");
        // Output:
        // Channel Beta received news: Market trends indicate growth.
        // Channel Gamma received news: Market trends indicate growth.
    }
}
```

Mastering these OOP concepts is fundamental. Be ready to explain them theoretically, draw diagrams if needed, and write simple code demonstrating each principle. Good luck!

Alright, let's deep dive into **"3. Collections Framework"**. This is an absolutely critical topic for any Java developer, especially one with 3 years of experience. You'll use these classes daily, and interviewers often ask about their internal workings, performance characteristics, and thread-safety aspects.

---

3. Collections Framework: Managing Groups of Objects

The Java Collections Framework (JCF) is a set of interfaces and classes that represent groups of objects as a single unit. It provides a unified architecture for storing and manipulating collections of objects.

3.1. Hierarchy

Understanding the hierarchy is key to understanding the relationships and shared behaviors among different collection types.

- **Iterable<E> (Top-level Interface):**
    - The root of the collection hierarchy.
    - Defines the iterator() method, which returns an Iterator<E>.
    - Enables the use of the enhanced for-each loop.
- **Collection<E> (Root Interface of all Collections):**
    - Extends Iterable<E>.
    - Represents a group of objects.
    - Defines common behaviors for all collections: add(), remove(), contains(), size(), isEmpty(), toArray(), clear(), etc.
    - **Sub-interfaces:** List, Set, Queue.
- **List<E> (Interface):**
    - Extends Collection<E>.
    - Represents an **ordered collection** (elements have a specific sequence).
    - Allows **duplicate** elements.
    - Elements can be accessed by their integer index (0-based).
    - Provides methods like get(int index), set(int index, E element), add(int index, E element), remove(int index), indexOf(), lastIndexOf(), subList().
    - **Common Implementations:** ArrayList, LinkedList, Vector (legacy), Stack (legacy).
- **Set<E> (Interface):**

- Extends Collection<E>.
- Represents a collection that contains **no duplicate elements**.
- Does **not guarantee order** (except for LinkedHashSet which maintains insertion order, and TreeSet which maintains sorted order).
- The add() method returns true if the element was added (i.e., it was not a duplicate) and false otherwise.
- **Common Implementations:** HashSet, LinkedHashSet, TreeSet.

- **Queue<E> (Interface):**
  - Extends Collection<E>.
  - Represents a collection designed for holding elements prior to processing.
  - Typically orders elements in a **FIFO (First-In, First-Out)** manner, but other orderings are possible (e.g., PriorityQueue).
  - Provides specific methods for adding (offer), removing (poll, remove), and inspecting (peek, element) elements from the "head" of the queue.
  - **Common Implementations:** LinkedList (implements Queue), PriorityQueue, ArrayDeque.
- **Map<K, V> (Interface):**
  - **Does NOT extend Collection<E>**. It's a separate hierarchy.
  - Represents a collection of **key-value pairs**.
  - Each key is **unique**; a key maps to at most one value.
  - Keys and values can be null (depending on implementation).
  - Provides methods like put(K key, V value), get(Object key), remove(Object key), containsKey(), containsValue(), keySet(), values(), entrySet().
  - **Common Implementations:** HashMap, LinkedHashMap, TreeMap, Hashtable (legacy).

3.2. List Implementations

- **ArrayList:**
  - **Internal Data Structure:** Resizable array.
  - **Order:** Maintains insertion order.
  - **Duplicates:** Allows duplicates.
  - **Nulls:** Allows null elements.
  - **Thread-Safety:** Not thread-safe.
  - **Performance:**

        - **get(index):** O(1) - constant time, very fast due to direct index access.
        - **add(element) (at end):** Amortized O(1) - usually fast, but can be O(N) if resizing is needed (doubles its capacity).
        - **add(index, element) (in middle):** O(N) - requires shifting elements.
        - **remove(index):** O(N) - requires shifting elements.
        - **Iteration:** Fast using for loop or enhanced for-each loop.
    - **Use Case:** When you need frequent random access to elements and fewer insertions/deletions in the middle.
- **LinkedList:**
    - **Internal Data Structure:** Doubly-linked list.
    - **Order:** Maintains insertion order.
    - **Duplicates:** Allows duplicates.
    - **Nulls:** Allows null elements.
    - **Thread-Safety:** Not thread-safe.
    - **Performance:**
        - **get(index):** O(N) - requires traversing from head or tail.
        - **add(element) (at end/start):** O(1).
        - **add(index, element) (in middle):** O(N) to find the index, then O(1) for insertion. Better than ArrayList if the current node is already known.
        - **remove(index):** O(N) to find the index, then O(1) for removal.
        - **Iteration:** Slower than ArrayList for direct index access, but fast using an Iterator.
    - **Use Case:** When you need frequent insertions and deletions at the ends or in the middle, and less frequent random access. Also implements Queue and Deque interfaces, making it suitable for queues, stacks, and double-ended queues.
- **Vector (Legacy):**
    - **Internal Data Structure:** Resizable array, similar to ArrayList.
    - **Thread-Safety: Synchronized** (thread-safe). All its methods are synchronized, which comes with performance overhead.
    - **Performance:** Similar to ArrayList but slower due to synchronization. Capacity increments by doubling by default.
    - **Use Case:** Rarely used in modern Java code unless thread-safety is strictly required for *all* operations on a single list instance, and Collections.synchronizedList() or CopyOnWriteArrayList are not viable. Generally, ArrayList with explicit synchronization or concurrent collections are preferred.

- **Stack (Legacy):**
    - **Internal Data Structure:** Extends Vector.
    - **Concept:** LIFO (Last-In, First-Out) data structure.
    - **Methods:** push() (add), pop() (remove and return top), peek() (return top without removing), empty(), search().
    - **Thread-Safety:** Synchronized (due to Vector inheritance).
    - **Use Case:** Should be avoided. For LIFO stack functionality, use ArrayDeque which is more efficient and provides better performance, or LinkedList as a Deque.

3.3. Set Implementations

- **HashSet:**
    - **Internal Data Structure:** Uses a HashMap internally to store its elements (elements are stored as keys in the underlying HashMap, with a dummy value).
    - **Order: No guaranteed order**. Order can change over time.
    - **Duplicates:** Stores only unique elements.
    - **Nulls:** Allows one null element.
    - **Thread-Safety:** Not thread-safe.
    - **Performance:**
        - **add(), remove(), contains():** O(1) on average (constant time), assuming good hash function distribution. In worst case (many collisions), can degrade to O(N).
    - **Important:** Relies on hashCode() and equals() methods of the stored objects for uniqueness. If you override equals(), you *must* override hashCode().
    - **Use Case:** When you need to store unique elements and the order doesn't matter, and you need fast lookups.
- **LinkedHashSet:**
    - **Internal Data Structure:** Uses a LinkedHashMap internally. Combines hashing with a doubly-linked list.
    - **Order:** Maintains **insertion order** (the order in which elements were added).
    - **Duplicates:** Stores only unique elements.
    - **Nulls:** Allows one null element.
    - **Thread-Safety:** Not thread-safe.
    - **Performance:** Slightly slower than HashSet for insertions and deletions due to maintaining the linked list, but still O(1) on average. Iteration is faster than HashSet because it follows the insertion order link.

- **Use Case:** When you need unique elements and also want to preserve the order of insertion.

- **TreeSet:**
  - **Internal Data Structure:** Uses a TreeMap internally (backed by a Red-Black tree, a self-balancing binary search tree).
  - **Order:** Stores elements in **sorted (natural) order** or by a custom Comparator.
  - **Duplicates:** Stores only unique elements.
  - **Nulls:** Does **NOT** allow null elements (throws NullPointerException on add if the set is not empty or on first element if Comparator is used that cannot handle null).
  - **Thread-Safety:** Not thread-safe.
  - **Performance:**
    - **add(), remove(), contains():** O(log N) - logarithmic time, efficient for large sets.
  - **Important:** Elements must implement Comparable interface (for natural ordering) or you must provide a Comparator in the TreeSet constructor.
  - **Use Case:** When you need to store unique elements in a sorted order.

3.4. Map Implementations

- **HashMap:**
  - **Internal Data Structure:** Array of linked lists (or Red-Black trees in Java 8+ for large buckets).
  - **Order: No guaranteed order**. Order can change over time.
  - **Keys/Values:** Allows one null key and multiple null values.
  - **Thread-Safety:** Not thread-safe.
  - **Performance:**
    - **put(), get(), remove():** O(1) on average (constant time), assuming good hash function for keys. In worst case (many collisions), can degrade to O(N).
  - **Important:** Relies on hashCode() and equals() methods of the **keys** for uniqueness and retrieval.
  - **Use Case:** Most commonly used Map implementation when order is not important and you need fast key-based lookups.

- **LinkedHashMap:**
  - **Internal Data Structure:** HashMap + doubly-linked list running through its entries.
  - **Order:** Maintains **insertion order** by default. Can also be configured for **access order** (elements move to the end of the list when accessed, useful for LRU caches).

    - **Keys/Values:** Allows one null key and multiple null values.
    - **Thread-Safety:** Not thread-safe.
    - **Performance:** Slightly slower than HashMap for additions and deletions due to maintaining the linked list, but still O(1) on average. Iteration is faster than HashMap because it follows the insertion/access order.
    - **Use Case:** When you need a map that preserves the order of key-value pair insertion (or access).
- **TreeMap:**
    - **Internal Data Structure:** Red-Black tree.
    - **Order:** Stores entries in **sorted order based on the keys** (natural order or custom Comparator).
    - **Keys/Values:** Does **NOT** allow null keys (throws NullPointerException). Allows multiple null values.
    - **Thread-Safety:** Not thread-safe.
    - **Performance:**
        - **put(), get(), remove():** O(log N).
    - **Important:** Keys must implement Comparable or you must provide a Comparator.
    - **Use Case:** When you need a map where keys are stored in a sorted order, or when you need to perform range queries (e.g., subMap()).
- **Hashtable (Legacy):**
    - **Internal Data Structure:** Array of linked lists, similar to HashMap.
    - **Order:** No guaranteed order.
    - **Keys/Values:** Does **NOT** allow null keys or null values. Throws NullPointerException.
    - **Thread-Safety: Synchronized** (thread-safe). All its methods are synchronized.
    - **Performance:** Slower than HashMap due to synchronization overhead.
    - **Use Case:** Largely superseded by ConcurrentHashMap for concurrent access or HashMap with external synchronization if needed.

3.5. Concurrency in Collections

Thread-safety is a major concern when collections are accessed by multiple threads.

- **Problem with non-thread-safe collections (ArrayList, HashMap, etc.) in multi-threaded environments:**
    - **Race Conditions:** Multiple threads trying to modify the collection simultaneously can lead to inconsistent state.

- **ConcurrentModificationException:** If one thread iterates over a collection while another thread modifies it (add/remove), the iterator can become invalid, leading to this runtime exception.

- **Solutions for Thread-Safe Collections:**

  1. **Collections.synchronizedList(), Collections.synchronizedSet(), Collections.synchronizedMap():**

     - **How it works:** These are static factory methods that return a synchronized (wrapper) view of a non-synchronized collection. They internally wrap the methods of the underlying collection with synchronized blocks.
     - **Example:** List<String> synchronizedList = Collections.synchronizedList(new ArrayList<>());
     - **Drawbacks:**
       - **Performance:** Can be slow due to single lock for the entire collection (high contention).
       - **External Synchronization for Iteration:** You *still* need to manually synchronize externally when iterating over these collections to prevent ConcurrentModificationException.

Java

```
synchronized (synchronizedList) {
   for (String item : synchronizedList) {
      // ...
   }
}
```

       - Generally not recommended for high concurrency scenarios.

  2. **java.util.concurrent package (Concurrent Collections):**

     - Provides highly optimized, scalable, and thread-safe alternatives. These are generally preferred for concurrent applications.
     - **ConcurrentHashMap:**
       - **How it works:** Uses a more fine-grained locking mechanism (segment locking or node-level locking depending on Java version) instead of a single global lock. This allows multiple threads to read and write concurrently to different parts of the map without blocking each other.
       - **Performance:** Much better concurrency than Hashtable or Collections.synchronizedMap().
       - **Nulls:** Does not allow null keys or values.

        - **Iteration:** Iterators are weakly consistent (fail-safe). They reflect the state of the map at the time the iterator was created and will not throw ConcurrentModificationException if the map is modified concurrently.
        - **Use Case:** Primary choice for thread-safe maps in high-concurrency environments.
    - **CopyOnWriteArrayList and CopyOnWriteArraySet:**
        - **How they work:** When a modifying operation (add, set, remove) is performed, a **new copy** of the underlying array is created. Reads operate on the old, unmodified array.
        - **Performance:** Excellent for concurrent **read** operations. Very poor for frequent **write** operations due to the overhead of copying the entire array.
        - **Use Case:** When reads vastly outnumber writes. E.g., maintaining a list of listeners in an event system. Iterators also fail-safe.
    - **BlockingQueue Interfaces (e.g., ArrayBlockingQueue, LinkedBlockingQueue):**
        - **Concept:** Queues that support operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available when storing an element.
        - **Use Case:** Producer-Consumer patterns, thread pools.
- **Fail-Fast vs. Fail-Safe Iterators:**
    - **Fail-Fast Iterators:**
        - Implemented by most non-concurrent collections (ArrayList, HashMap, HashSet, etc.).
        - They try to detect concurrent modifications (modifications to the underlying collection by any other means than the iterator's own remove() method) and throw a ConcurrentModificationException immediately.
        - They are designed to "fail fast" to indicate potential bugs in multi-threaded code.
    - **Fail-Safe Iterators:**
        - Implemented by concurrent collections (ConcurrentHashMap, CopyOnWriteArrayList, etc.).
        - They operate on a clone or snapshot of the collection at the time the iterator was created.
        - They **do not throw ConcurrentModificationException** if the original collection is modified concurrently.

        - The trade-off is that they might not reflect the very latest state of the collection.

3.6. Comparable vs. Comparator

These interfaces are used for custom sorting of objects.

- **Comparable<T> (Natural Ordering):**
    - **Interface:** public interface Comparable<T> { int compareTo(T o); }
    - **Purpose:** Defines the **natural or default sorting order** for objects of a class.
    - **Implementation:** The class whose objects you want to sort must implement this interface.
    - **compareTo() method:**
        - Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
        - obj1.compareTo(obj2):
            - < 0 if obj1 comes before obj2
            - == 0 if obj1 is equal to obj2
            - > 0 if obj1 comes after obj2
    - **Usage:** Used by Collections.sort(List) for lists, and TreeSet/TreeMap constructors.
- **Comparator<T> (Custom/External Ordering):**
    - **Interface:** public interface Comparator<T> { int compare(T o1, T o2); boolean equals(Object obj); // Default method from Object }
    - **Purpose:** Defines an **alternative or custom sorting order** for objects. Useful when:
        - You don't want to modify the original class.
        - You need multiple ways to sort the same objects (e.g., sort Employee by name, id, salary).
        - The objects do not implement Comparable.
    - **Implementation:** You create a separate class that implements Comparator or use a lambda expression (Java 8+).
    - **compare() method:**
        - Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.
        - comparator.compare(obj1, obj2): Same return value logic as compareTo().
    - **Usage:** Used by Collections.sort(List, Comparator), Arrays.sort(array, Comparator), and TreeSet/TreeMap constructors that take a Comparator argument.

- **Java 8+:** Often implemented using lambda expressions for conciseness: (o1, o2) -> o1.getName().compareTo(o2.getName())

- **Example:**

Java

```
import java.util.*;


class Person implements Comparable<Person> { // Implements Comparable for natural ordering by age
    String name;
    int age;


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


    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age); // Natural order: sort by age
    }


    @Override
    public String toString() {
        return "Person{" + "name='" + name + '\'' + ", age=" + age + '}';
    }
}


public class SortingDemo {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Charlie", 25));
```

```
        people.add(new Person("Bob", 35));

        // 1. Sort using natural ordering (Comparable)
        Collections.sort(people); // Requires Person to implement Comparable
        System.out.println("Sorted by Age (Natural Order):");
        people.forEach(System.out::println);
        // Output: Charlie (25), Alice (30), Bob (35)

        // 2. Sort using a custom Comparator (by Name)
        Comparator<Person> nameComparator = (p1, p2) -> p1.name.compareTo(p2.name);
        Collections.sort(people, nameComparator);
        System.out.println("\nSorted by Name (Custom Comparator):");
        people.forEach(System.out::println);
        // Output: Alice (30), Bob (35), Charlie (25)

        // 3. Using a TreeSet (natural order)
        TreeSet<Person> sortedPeopleByAge = new TreeSet<>();
        sortedPeopleByAge.add(new Person("David", 40));
        sortedPeopleByAge.add(new Person("Eve", 20));
        sortedPeopleByAge.add(new Person("Frank", 30));
        System.out.println("\nTreeSet (Sorted by Age):");
        sortedPeopleByAge.forEach(System.out::println);

        // 4. Using a TreeMap (custom comparator for keys)
        Map<Person, String> personMap = new TreeMap<>(nameComparator); // TreeMap uses
Comparator
        personMap.put(new Person("Zara", 28), "Developer");
        personMap.put(new Person("Yara", 32), "Manager");
        personMap.put(new Person("Xavier", 25), "Analyst");
        System.out.println("\nTreeMap (Sorted by Name):");
        personMap.forEach((person, role) -> System.out.println(person + " -> " + role));
```

```
    }
}
```

The Collections Framework is vast, but focusing on these core interfaces and their most common implementations, their performance characteristics, and thread-safety concerns will make you well-prepared for interview questions. Be ready to discuss specific scenarios for when to use ArrayList vs. LinkedList, HashMap vs. TreeMap, and HashMap vs. ConcurrentHashMap.

Got it. Let's delve into **"4. Multithreading and Concurrency"** in detail. This is a crucial topic for a 3-year experienced Java developer, as it directly relates to building high-performance, responsive, and scalable applications. Interviewers will often ask about thread creation, synchronization, common concurrency problems, and the java.util.concurrent package.

---

4. Multithreading and Concurrency: Building Concurrent Applications

Multithreading allows a program to execute multiple parts of its code concurrently within a single process. Each independent part of execution is called a thread.

Concurrency is the ability of different parts of a program or system to execute independently or in parallel.

4.1. Thread Basics

- **What is a Thread?**
    - The smallest unit of execution within a process.[1]
    - A single process can have multiple threads.[2]
    - Threads share the same memory space (heap) of the process, but each has its own call stack and PC (Program Counter) register.[3]
    - Lightweight compared to processes (faster context switching).[4]
- **Advantages of Multithreading:**
    - **Responsiveness:** UI applications remain responsive while long-running tasks execute in the background.
    - **Resource Utilization:** Efficiently utilizes CPU (especially multi-core processors) by allowing tasks to run in parallel.[5]
    - **Faster Execution:** Completes tasks quicker by dividing them among multiple threads.[6]
    - **Better Resource Sharing:** Threads within the same process can easily share data.[7]
- **Disadvantages of Multithreading:**
    - **Complexity:** Harder to design, debug, and test concurrent applications due to race conditions, deadlocks, etc.

- **Overhead:** Context switching between threads incurs some overhead.
- **Resource Contention:** Multiple threads accessing shared resources can lead to data corruption or inconsistent states.[8]

- **Creating Threads:** In Java, there are two primary ways to create and run threads:[9]
    1. **Extending the Thread Class:**
        - Create a class that extends java.lang.Thread.
        - Override the run() method with the code that the thread will execute.
        - Create an instance of your class and call its start() method (which in turn calls run() in a new thread).
        - **Limitation:** Java does not support multiple inheritance, so your class cannot extend any other class if you choose this approach.

Java

```
class MyThread extends Thread {
   @Override
   public void run() {
      System.out.println("Thread extending Thread class is running: " +
Thread.currentThread().getName());
   }
}


public class ThreadCreationDemo {
   public static void main(String[] args) {
      MyThread thread1 = new MyThread();
      thread1.start(); // Starts a new thread, calls run()
   }
}
```

    2. **Implementing the Runnable Interface:** (Recommended approach)
        - Create a class that implements java.lang.Runnable.
        - Override the run() method.
        - Create an instance of your Runnable class.
        - Create a Thread object, passing your Runnable instance to its constructor.
        - Call the start() method on the Thread object.

- **Advantage:** Your class can still extend another class while implementing Runnable, offering more flexibility. This separates the task (what to run) from the thread (how to run it).

Java

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Thread implementing Runnable is running: " +
Thread.currentThread().getName());
    }
}


public class RunnableCreationDemo {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread2 = new Thread(myRunnable, "WorkerThread-1"); // Give thread a name
        thread2.start();
    }
}
```

3. **Implementing Callable and Future (Java 5+):**
    - **Purpose:** Runnable's run() method cannot return a value or throw checked exceptions. Callable solves this.
    - Callable's call() method returns a result and can throw exceptions.[10]
    - Future represents the result of an asynchronous computation.[11] It provides methods to check if the computation is complete, wait for its completion, and retrieve the result.[12]
    - Requires an ExecutorService to execute Callable tasks.[13]

Java

```
import java.util.concurrent.*;


class MyCallable implements Callable<Integer> {
    private int number;
```

```
    public MyCallable(int number) {
        this.number = number;
    }

    @Override
    public Integer call() throws Exception {
        System.out.println("Callable task " + number + " running in thread: " +
Thread.currentThread().getName());
        Thread.sleep(1000); // Simulate work
        return number * number;
    }
}

public class CallableFutureDemo {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executor = Executors.newFixedThreadPool(2); // Create a thread pool

        Future<Integer> future1 = executor.submit(new MyCallable(5));
        Future<Integer> future2 = executor.submit(new MyCallable(10));

        System.out.println("Result of task 1: " + future1.get()); // get() blocks until result is available
        System.out.println("Result of task 2: " + future2.get());

        executor.shutdown(); // Shut down the executor
    }
}
```

- **Thread Lifecycle:** A thread goes through various states from its creation to its termination.
    - **NEW:** A thread that has been created but not yet started (i.e., start() method not called).
    - **RUNNABLE:** A thread that is executing or ready to execute. It's waiting for its turn on the CPU. It includes both "Running" and "Ready" states.

- **BLOCKED:** A thread that is waiting for a monitor lock (e.g., waiting to enter a synchronized block/method) to access a shared resource.[14]
- **WAITING:** A thread that is indefinitely waiting for another thread to perform a particular action (e.g., calling Object.wait(), Thread.join(), LockSupport.park()). It will not return to the runnable state until another thread explicitly wakes it up.
- **TIMED_WAITING:** A thread that is waiting for another thread to perform an action for a specified waiting time (e.g., Thread.sleep(long millis), Object.wait(long millis), Thread.join(long millis), LockSupport.parkNanos(), LockSupport.parkUntil()). It will return to the runnable state after the time expires or if woken up.
- **TERMINATED:** A thread that has finished its execution (its run() method has completed or thrown an uncaught exception).

4.2. Synchronization

The process of controlling access to shared resources by multiple threads to prevent race conditions and ensure data consistency.[15]

- **Race Condition:** Occurs when multiple threads try to access and modify a shared resource simultaneously, and the final outcome depends on the unpredictable order of execution of these threads.[16]
- **How to achieve Synchronization:**
    1. **synchronized Keyword:**
        - **synchronized method:**
            - When applied to an instance method, it acquires the lock on the **object** itself (i.e., this object) before executing the method. Only one synchronized method of an object can be executed at a time by any thread.
            - When applied to a static method, it acquires the lock on the **class** object (ClassName.class). Only one static synchronized method of a class can be executed at a time by any thread.
            - Java
            - class Counter {
            - private int count = 0;
            - public synchronized void increment() { // Synchronized method
            - count++;
            - }
            - public synchronized int getCount() {
            - return count;
            - }

- }

- **synchronized block:**
  - Acquires a lock on a specified object (the monitor).
  - Allows for more fine-grained control than synchronized methods. You can synchronize only the critical section of code.
  - Java
  - class SharedResource {
  - private int data = 0;
  - private final Object lock = new Object(); // A dedicated lock object
  -
  - public void increment() {
  - synchronized (lock) { // Synchronized block on 'lock' object
  - data++;
  - }
  - }
  - public void updateData(int newValue) {
  - synchronized (this) { // Synchronized on 'this' object
  - this.data = newValue;
  - }
  - }
  - }
- **Key points about synchronized:**
  - Re-entrant: A thread that has acquired a lock can re-enter any synchronized block or method that is guarded by the same lock.
  - Intrinsic Lock (Monitor Lock): Every Java object has an intrinsic lock associated with it.

2. **volatile Keyword:**
   - **Purpose:** Ensures visibility of changes to variables across threads. It prevents compilers/CPUs from caching variables locally and ensures that any write to a volatile variable happens before any subsequent read of that variable by another thread.

- **Does NOT guarantee atomicity:** volatile only ensures visibility, not that operations like i++ (which are read-modify-write) are atomic. For atomicity, you need synchronization or Atomic classes.
- **Use Case:** When one thread writes to a variable and other threads only read that variable, and you need to ensure readers see the latest value. (e.g., a flag to stop a thread).
- Java
- ```
  class VolatileExample {
      private volatile boolean running = true;

      public void start() {
          new Thread(() -> {
              while (running) {
                  // Do work
              }
              System.out.println("Thread stopped.");
          }).start();
      }

      public void stop() {
          running = false; // Change will be immediately visible to other thread
      }
  }
  ```

3. **wait(), notify(), notifyAll() Methods:**
    - **From Object class:** These methods are used for **inter-thread communication** and must be called from within a synchronized block/method, on the object whose intrinsic lock is held.
    - **wait():** Causes the current thread to release the lock on the object and go into a WAITING or TIMED_WAITING state until another thread invokes notify() or notifyAll() on the same object, or the specified timeout expires.
    - **notify():** Wakes up a single thread that is waiting on this object's monitor.[17] Which thread gets woken up is non-deterministic.
    - **notifyAll():** Wakes up all threads that are waiting on this object's monitor.
    - **Typical Use Case:** Producer-Consumer problem.
  ```

Java

```
class Message {
    private String msg;
    private boolean isEmpty = true;

    public synchronized String take() { // Consumer method
        while (isEmpty) {
            try {
                wait(); // Wait if no message
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        isEmpty = true;
        notifyAll(); // Notify producer that it can put another message
        return msg;
    }

    public synchronized void put(String msg) { // Producer method
        while (!isEmpty) {
            try {
                wait(); // Wait if message is not consumed
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        this.msg = msg;
        isEmpty = false;
        notifyAll(); // Notify consumer that message is available
    }
}
```

4.3. Thread Pools (ExecutorService)

- **Problem:** Creating a new thread for every task is expensive in terms of resource consumption (memory, CPU cycles for context switching).
- **Solution:** Thread pools manage a pool of worker threads. Tasks are submitted to the pool, and the pool's threads execute them.
- **ExecutorService (Interface):** The main interface for executing tasks asynchronously.[18]
- **Executors (Utility Class):** Provides static factory methods to create common ExecutorService configurations.[19]
    - **newFixedThreadPool(int nThreads):** Creates a thread pool with a fixed number of threads.[20] If more tasks are submitted than threads, they wait in a queue.
    - **newCachedThreadPool():** Creates a thread pool that creates new threads as needed, but reuses existing ones when they become available.[21] Good for applications with many short-lived tasks.
    - **newSingleThreadExecutor():** Creates a single-threaded executor.[22] Ensures tasks are executed sequentially.
    - **newScheduledThreadPool(int corePoolSize):** Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.[23]
- **Lifecycle:** ExecutorService needs to be explicitly shut down using shutdown() (waits for current tasks to complete) or shutdownNow() (attempts to stop all running tasks immediately).[24]
- **Benefits:**
    - **Reduced Overhead:** Reuses threads, avoiding the overhead of creating/destroying threads.[25]
    - **Resource Management:** Limits the number of concurrent threads, preventing resource exhaustion.
    - **Improved Responsiveness:** Tasks are picked up faster from the queue.
    - **Separation of Concerns:** Decouples task submission from thread management.
- **Example (See CallableFutureDemo above for basic usage).**

4.4. Concurrency Utilities (java.util.concurrent)

This package introduced powerful, high-level concurrency constructs that are generally preferred over low-level wait()/notify() or synchronized for complex scenarios, due to their ease of use, robustness, and performance.[26]

- **Lock Interface (e.g., ReentrantLock):**
    - **Problem with synchronized:** Limited flexibility (must release lock in the same block, cannot interrupt a waiting thread, cannot try to acquire a lock without blocking, cannot acquire multiple locks in specific order).

- ReentrantLock: An explicit lock implementation that provides more flexibility than synchronized.
  - **lock():** Acquires the lock. Blocks if the lock is not available.
  - **unlock():** Releases the lock. **Must be called in a finally block** to ensure release even if exceptions occur.
  - **tryLock():** Attempts to acquire the lock without blocking.[27]
  - **tryLock(long timeout, TimeUnit unit):** Attempts to acquire the lock within a specified timeout.[28]
  - **lockInterruptibly():** Acquires the lock unless the current thread is interrupted.[29]
  - **Re-entrant:** A thread can acquire the same lock multiple times.[30]
- **Condition Interface:** Can be obtained from a Lock (lock.newCondition()).[31] It's an alternative to wait()/notify() that provides more granular control over thread waiting and notification (e.g., multiple wait sets for a single lock).
- **ReentrantReadWriteLock:** Allows multiple readers to access a resource concurrently, but only one writer at a time.[32] Ideal for read-heavy workloads.

- **CountDownLatch:**
  - **Purpose:** A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
  - **Mechanism:** Initialized with a count. Threads call countDown() when their task is complete.[33] A thread waiting on the latch calls await() and blocks until the count reaches zero.[34]
  - **Use Case:** Start multiple threads, do some concurrent work, and then wait for all of them to finish before proceeding.
- **CyclicBarrier:**
  - **Purpose:** A synchronization aid that allows a set of threads to wait for each other to reach a common barrier point.[35]
  - **Mechanism:** Initialized with a number of parties. Threads call await() when they reach the barrier. All threads block until all parties have called await().
  - **Reusability:** The barrier can be reused once all threads have passed it.[36]
  - **Use Case:** Useful for scenarios where a group of threads needs to perform a series of operations in stages, with all threads needing to complete a stage before moving to the next.
- **Semaphore:**
  - **Purpose:** Controls access to a limited number of resources.

    - **Mechanism:** Initialized with a number of permits. Threads acquire a permit (acquire()) to access the resource and release it (release()) when done.[37] If no permits are available, threads block.
    - **Use Case:** Limiting the number of concurrent database connections, limiting concurrent requests to a specific service.
- **BlockingQueue (e.g., ArrayBlockingQueue, LinkedBlockingQueue):**
    - **Purpose:** Queues that automatically handle synchronization when adding or removing elements.
    - **Blocking Operations:**
        - put(E e): Waits if the queue is full.
        - take(): Waits if the queue is empty.[38]
    - **Non-blocking variants:** offer(), poll().
    - **Use Case:** The quintessential solution for the Producer-Consumer problem. Producers add items, consumers take items.
- **Atomic Variables (e.g., AtomicInteger, AtomicLong, AtomicBoolean, AtomicReference):**
    - **Purpose:** Provide atomic operations on single variables without using explicit locks.
    - **Mechanism:** Use low-level, hardware-supported compare-and-swap (CAS) operations.
    - **Benefits:** More performant and scalable than using synchronized for simple variable updates.
    - **Use Case:** Atomic counters, flags, or references that need to be updated safely by multiple threads.

4.5. Deadlock

- **What it is:** A situation where two or more threads are blocked indefinitely, each waiting for the other to release a resource that it needs.[39]
- **Conditions for Deadlock (Coffman Conditions - all four must be met):**
    1. **Mutual Exclusion:** At least one resource must be held in a non-sharable mode (e.g., a lock).
    2. **Hold and Wait:** A thread holding at least one resource is waiting to acquire additional resources held by other threads.[40]
    3. **No Preemption:** Resources cannot be forcibly taken from a thread; they must be released voluntarily.[41]
    4. Circular Wait: A set of threads T1, T2, ..., Tn exists such that T1 is waiting for a resource held by T2, T2 is waiting for a resource held by T3, and so on, with Tn waiting for a resource held by T1.
- **Example (Classic Deadlock):**

Java

```
public class DeadlockExample {
    private static Object lock1 = new Object();
    private static Object lock2 = new Object();

    public static void main(String[] args) {
        // Thread 1: Tries to acquire lock1, then lock2
        new Thread(() -> {
            synchronized (lock1) {
                System.out.println("Thread 1: Acquired lock1");
                try { Thread.sleep(100); } catch (InterruptedException e) {}
                System.out.println("Thread 1: Waiting for lock2...");
                synchronized (lock2) { // Tries to acquire lock2 while holding lock1
                    System.out.println("Thread 1: Acquired lock2");
                }
            }
        }, "Thread-1").start();

        // Thread 2: Tries to acquire lock2, then lock1
        new Thread(() -> {
            synchronized (lock2) {
                System.out.println("Thread 2: Acquired lock2");
                try { Thread.sleep(100); } catch (InterruptedException e) {}
                System.out.println("Thread 2: Waiting for lock1...");
                synchronized (lock1) { // Tries to acquire lock1 while holding lock2
                    System.out.println("Thread 2: Acquired lock1");
                }
            }
        }, "Thread-2").start();
    }
}
```

- **How to Prevent Deadlock:**
    - **Break Circular Wait:** The most common strategy. Establish a consistent ordering of resource acquisition. If Thread 1 acquires Lock A then Lock B, Thread 2 should also acquire Lock A then Lock B, never Lock B then Lock A.
    - **Avoid Hold and Wait:** Acquire all necessary resources at once, or release all held resources if additional ones cannot be acquired.
    - **Allow Preemption:** If a thread cannot acquire a resource, it releases the ones it holds and retries.
    - **Avoid Mutual Exclusion (if possible):** Use concurrent data structures that don't require explicit locking (e.g., ConcurrentHashMap).
- **How to Detect Deadlock:**
    - **Thread Dumps:** Analyze thread dumps (jstack <pid>). They will show threads in BLOCKED state, often with waiting for monitor entry on specific locks and holding lock on others.
    - **Monitoring Tools:** Profilers or APM (Application Performance Monitoring) tools can detect deadlocks.[42]
- **How to Resolve Deadlock:** Usually involves fixing the code to prevent one of the four conditions from being met, most commonly by ensuring a consistent lock acquisition order.

Multithreading and Concurrency are vast and complex topics. For a 3-year experienced developer, the expectation is to not just know the basics, but also to understand the trade-offs of different synchronization mechanisms, when to use the java.util.concurrent utilities, and how to identify and prevent common concurrency issues like deadlocks. Practice writing concurrent code and debugging it to solidify your understanding.

Got it! Let's dive deep into **"5. Spring Boot & Microservices"**. This is a crucial area for a Java developer with 3 years of experience, as Spring Boot is the de-facto standard for building modern Java applications, especially in the context of microservices.

---

5. Spring Boot & Microservices: Modern Java Application Development

Spring Boot is an opinionated framework that simplifies the development of production-ready, stand-alone Spring applications.[1] It builds on top of the Spring Framework, making it easier to get Spring applications up and running with minimal configuration.[2] Microservices are an architectural style that structures an application as a collection of loosely coupled, independently deployable services.[3]

5.1. Spring Core Concepts

Spring Boot is built on the foundation of the Spring Framework.[4] Understanding these core concepts is vital.

a. IoC (Inversion of Control) Container and Dependency Injection (DI)

- **IoC (Inversion of Control):**

- **Concept:** Instead of the developer manually creating and managing objects and their dependencies, the Spring IoC container takes control of this process. The control of object creation and lifecycle is *inverted* from the application code to the framework.
- **How it works:** Spring scans your application, identifies components, creates their instances, and manages their lifecycle (creation, configuration, destruction).[5]
- **Benefits:** Reduces boilerplate code, promotes loose coupling, easier testing, and promotes modularity.[6]

- **Dependency Injection (DI):**
  - **Concept:** A specific implementation of IoC where the container "injects" (provides) the dependencies (collaborating objects) that an object needs, rather than the object creating or looking up its dependencies itself.[7]
  - **Analogy:** Instead of a class A creating an instance of B (tight coupling), A declares that it *needs* an instance of B, and Spring provides it.
  - **Types of Dependency Injection:**
    1. **Constructor Injection (Recommended):** Dependencies are provided through the class constructor.
       - **Pros:** Guarantees that required dependencies are available when the object is created (immutable dependencies), makes testing easier, clearer about required dependencies.[8]
       - **Example:**

Java

```
@Service
public class UserService {
    private final UserRepository userRepository; // Final for immutability

    @Autowired // Optional from Spring Boot 2.x if only one constructor
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    // ... methods using userRepository
}
```

    2. **Setter Injection:** Dependencies are provided through setter methods.
       - **Pros:** Allows for optional dependencies, more flexible for re-configuring objects after creation.[9]

- **Cons:** Object might be in an inconsistent state before all setters are called. Not suitable for mandatory dependencies.
- **Example:**

Java

```
@Service
public class ProductService {
   private ProductRepository productRepository;

   @Autowired
   public void setProductRepository(ProductRepository productRepository) {
      this.productRepository = productRepository;
   }
   // ... methods using productRepository
}
```

3. Field Injection (Least Recommended but commonly seen): Dependencies are injected directly into fields using @Autowired.
    - **Pros:** Very concise, less code.[10]
    - **Cons:**
        - Makes objects harder to test without Spring context.
        - Breaks encapsulation (private fields are directly accessed by the framework).[11]
        - Makes dependencies less obvious.
        - Cannot declare fields as final.
    - **Example:**

Java

```
@Service
public class OrderService {
   @Autowired
   private OrderRepository orderRepository; // Field injection
   // ... methods using orderRepository
}
```

b. Bean Lifecycle

- **Bean:** An object that is instantiated, assembled, and managed by a Spring IoC container. They are the backbone of your application and form the objects that are managed by the Spring IoC container.[12]
- **Lifecycle Stages (Simplified):**
    1. **Instantiation:** The container creates an instance of the bean.[13]
    2. **Populate Properties:** Dependencies are injected (DI).[14]
    3. **Initialization:**
        - BeanPostProcessors are applied (e.g., @PostConstruct methods are called, InitializingBean's afterPropertiesSet() method is called).
        - Custom init-method is called (if configured).
    4. **Ready for Use:** The bean is ready for use by the application.
    5. **Destruction:**
        - PreDestroy methods are called, DisposableBean's destroy() method is called.[15]
        - Custom destroy-method is called (if configured).
- **Bean Scopes:** Define how Spring container manages instances of a bean.
    1. **singleton (Default):**
        - **Concept:** Only **one single instance** of the bean is created per Spring IoC container. This is the most common scope.
        - **Use Case:** Stateless beans (e.g., services, repositories, controllers), shared resources.
    2. **prototype:**
        - **Concept:** A **new instance** of the bean is created every time it is requested from the container.
        - **Use Case:** Stateful beans that need a fresh instance for each client.
    3. **Web-Aware Scopes (only in web applications):**
        - **request:** A new instance for each HTTP request.
        - **session:** A new instance for each HTTP session.
        - **application:** A new instance for the lifecycle of the ServletContext.
        - **websocket:** A new instance for the lifecycle of a WebSocket.

c. Annotations

Spring Boot heavily relies on annotations for configuration and defining components.[16]

- **@Component:** A generic stereotype annotation indicating that an annotated class is a "component" and should be managed by the Spring IoC container.[17] It's a general-purpose annotation for any Spring-managed component.
- **Stereotype Annotations (specialized @Component):**
    - **@Service:** Indicates that an annotated class is a "service" component in the business layer.[18] It provides better readability and can be used for specific cross-cutting concerns (e.g., transaction management).
    - **@Repository:** Indicates that an annotated class is a "repository" component in the data access layer.[19] It enables automatic exception translation from technology-specific exceptions (e.g., JDBC SQLException) to Spring's DataAccessException hierarchy.[20]
    - **@Controller:** Indicates that an annotated class is a "controller" component in the web layer, typically handling incoming web requests and returning responses (often views).[21]
    - **@RestController:** A convenience annotation that combines @Controller and @ResponseBody. [22]Used for building RESTful web services that return data directly (e.g., JSON/XML) rather than view names.
- **@Autowired:** Used for automatic dependency injection.[23] Spring automatically finds a matching bean by type and injects it.[24]
    - Can be used on constructors, setter methods, fields, and even method parameters.[25]
- **@Qualifier("beanName"):** Used with @Autowired when there are multiple beans of the same type.[26] It helps specify exactly which bean to inject by its name.
- **@Value("${property.name}"):** Used to inject values from properties files (application.properties, application.yml) or environment variables directly into fields or method parameters.
- **@Bean:**
    - Annotated on methods within a @Configuration class.
    - Indicates that a method produces a bean to be managed by the Spring IoC container. The method's return value will be registered as a Spring bean.
    - The method name typically becomes the bean name.[27]
- **@Configuration:**
    - Indicates that a class declares one or more @Bean methods.
    - Spring processes these classes to generate bean definitions and service requests for those beans at runtime.
    - Often used for externalizing bean definitions or for complex configurations that cannot be done with simple stereotype annotations.

5.2. Spring Boot

Spring Boot significantly streamlines Spring application development.[28]

- **What is Spring Boot?**
    - An extension of the Spring Framework that simplifies the creation of stand-alone, production-grade Spring applications.[29]
    - Aims to get you up and running with minimal fuss by providing sensible defaults.[30]
- **Advantages of Spring Boot:**
    - Auto-Configuration: Automatically configures your Spring application based on the dependencies present in your classpath. For example, if spring-boot-starter-web is present, it auto-configures Tomcat and Spring MVC. If spring-boot-starter-data-jpa and a database driver are present, it configures DataSource and EntityManager.
    - **Embedded Servers:** Comes with embedded Tomcat, Jetty, or Undertow servers.[31] This means you can run your application as a simple JAR file without needing to deploy it to an external web server.
    - Starter Dependencies: Provides "starter" POMs (e.g., spring-boot-starter-web, spring-boot-starter-data-jpa). These are convenient dependency descriptors that bring in all necessary transitive dependencies for a particular functionality, simplifying your pom.xml.
    - **Opinionated Defaults:** Provides sensible default configurations, reducing the need for explicit XML or Java-based configuration in many cases.[32]
    - **Production-Ready Features:** Includes features like externalized configuration, health checks, metrics, and tracing, making it easy to monitor and manage applications in production.[33]
    - **No XML Configuration:** Largely eliminates the need for verbose XML configurations; prefers annotation-based and JavaConfig.
- **Creating RESTful APIs with Spring Boot:**
    - Uses @RestController to define controllers that handle incoming HTTP requests and return JSON/XML responses.[34]
    - **@RequestMapping:** Maps HTTP requests to handler methods.[35]
        - Can be used at class level (base path) and method level.
        - @RequestMapping(value = "/users", method = RequestMethod.GET)
    - **Shorthand Annotations (Preferred):**
        - **@GetMapping:** For HTTP GET requests (read data).
        - **@PostMapping:** For HTTP POST requests (create data).
        - **@PutMapping:** For HTTP PUT requests (update/replace data).[36]
        - **@DeleteMapping:** For HTTP DELETE requests (delete data).[37]
        - **@PatchMapping:** For HTTP PATCH requests (partial update data).[38]

- **@PathVariable:** Binds a method parameter to a URI template variable (e.g., /users/{id}).[39]
- **@RequestParam:** Binds a method parameter to a web request parameter (query parameter, e.g., /users?name=Alice).[40]
- **@RequestBody:** Maps the HTTP request body to a domain object (e.g., converting JSON request body to a Java object).[41] Requires a JSON/XML processing library like Jackson.
- **@ResponseBody:** (Implicit in @RestController) Indicates that the return value of a method should be bound directly to the web response body.[42]
- **ResponseEntity<T>:** Allows you to control the entire HTTP response (status code, headers, body).[43]

- **Basic Exception Handling in Spring Boot:**
  - **@ResponseStatus:** Annotate custom exception classes with @ResponseStatus to automatically map them to specific HTTP status codes.[44]
  - **@ExceptionHandler:** Methods annotated with this in a @Controller or @RestController handle specific exceptions that occur within that controller.[45]
  - **@ControllerAdvice / @RestControllerAdvice:**
    - Global exception handlers. Methods in a class annotated with @ControllerAdvice (or @RestControllerAdvice) and @ExceptionHandler can handle exceptions from *any* controller in the application.
    - This centralizes exception handling logic.
- **application.properties/application.yml:**
  - Used for externalizing configuration.
  - Allows you to define application-specific properties (e.g., server port, database connection details, logging levels, custom properties).[46]
  - application.yml is preferred for its cleaner, more readable YAML syntax.
  - Spring Boot automatically loads these files. Profiles can be used (application-dev.properties, application-prod.yml) for environment-specific configurations.[47]

5.3. Spring Data JPA/Hibernate

Spring Data JPA simplifies data access layer development by providing abstractions over JPA (Java Persistence API) and Hibernate (a popular JPA implementation).[48]

- **ORM (Object-Relational Mapping):**
  - **Concept:** A technique that maps objects in an object-oriented programming language (like Java) to tables in a relational database. It abstracts away the complexities of JDBC and SQL.
  - **Benefit:** Allows developers to interact with the database using object-oriented paradigms, rather than writing raw SQL.

- **JPA (Java Persistence API):**
    - A **specification** (standard) for ORM in Java. It defines a set of interfaces and annotations.
    - Hibernate is one of the most popular **implementations** of the JPA specification.[49]
- **Core Annotations for Entities:**
    - **@Entity:** Marks a plain old Java object (POJO) as a JPA entity, meaning it maps to a table in the database.[50]
    - **@Table(name="table_name"):** (Optional) Specifies the name of the database table to which this entity is mapped.[51] If omitted, the class name is used.
    - **@Id:** Marks the primary key field of the entity.
    - **@GeneratedValue(strategy = GenerationType.IDENTITY):** Configures how the primary key value is generated. IDENTITY uses database auto-increment. Other strategies include AUTO, SEQUENCE, TABLE.
    - **@Column(name="column_name"):** (Optional) Specifies the database column name for a field.[52]
    - **@OneToMany, @ManyToOne, @OneToOne, @ManyToMany:** Annotations for defining relationships between entities.[53]
- **JpaRepository / CrudRepository:**
    - Spring Data JPA provides these interfaces that offer out-of-the-box CRUD (Create, Read, Update, Delete) operations and query methods without writing any implementation code.[54]
    - You just declare an interface extending JpaRepository
    - **CrudRepository:** Basic CRUD functionalities (save, findById, findAll, delete).[55]
    - **PagingAndSortingRepository:** Extends CrudRepository and adds methods for pagination and sorting.
    - **JpaRepository:** Extends PagingAndSortingRepository and adds JPA-specific features like flush(), saveAndFlush(), deleteInBatch(), and methods for batch operations. It also integrates with JPA EntityManager.
- **Derived Query Methods:** Spring Data JPA can automatically generate queries from method names in your repository interfaces.[56]
    - findByFirstName(String firstName): Generates SELECT * FROM table WHERE first_name = ?
    - findByAgeGreaterThan(int age)
    - findByEmailContainingIgnoreCase(String email)
    - findByNameAndAge(String name, int age)

    - You can also use @Query annotation for complex custom queries (JPQL or native SQL).

- **Lazy vs. Eager Loading:**
    - Applies to fetching strategies for associated entities (relationships like @OneToMany, @ManyToOne).
    - **Lazy Loading (Default for @OneToMany, @ManyToMany):**
        - Associated data is *not* loaded from the database immediately when the main entity is loaded.
        - It's loaded only when it's explicitly accessed (e.g., when you call a getter method on the collection).
        - **Pros:** Efficient, loads only what's needed.
        - **Cons:** Can lead to N+1 query problem if not managed properly (fetching child entities one by one in a loop).
    - **Eager Loading (Default for @ManyToOne, @OneToOne):**
        - Associated data is loaded from the database *immediately* along with the main entity.
        - **Pros:** Data is available instantly.
        - **Cons:** Can fetch unnecessary data, leading to performance issues if relationships are deep or data is large.
    - **Recommendation:** Prefer lazy loading and use explicit fetching strategies (e.g., JOIN FETCH in JPQL queries or @EntityGraph) when you know you need the associated data to avoid N+1 issues.

5.4. Microservices (Conceptual Understanding)

While "microservices" is an architectural style, Spring Boot is a primary tool for building them.[57]

- **What are Microservices?**
    - An architectural approach where a large application is built as a suite of small, independent services.[58]
    - Each service:
        - Runs in its own process.
        - Communicates with other services through lightweight mechanisms (e.g., REST APIs, message queues).[59]
        - Is independently deployable.
        - Is built around a specific business capability.
        - Can be developed, deployed, and scaled independently.
        - Often uses its own data store (decentralized data management).

- **Advantages:**
    - **Scalability:** Services can be scaled independently based on their load.[60]
    - **Resilience:** Failure in one service doesn't necessarily bring down the entire application.
    - **Technology Heterogeneity:** Different services can use different programming languages, databases, and frameworks best suited for their needs.
    - **Faster Development Cycles:** Smaller, autonomous teams can work on services independently.[61]
    - **Easier Maintenance:** Smaller codebases are easier to understand and manage.
    - **Deployment Flexibility:** Independent deployments mean faster release cycles.
- **Disadvantages:**
    - **Increased Complexity:** Distributed systems are inherently more complex (network latency, data consistency, distributed transactions, monitoring, debugging).[62]
    - **Operational Overhead:** More services mean more deployments, more monitoring, more infrastructure.
    - **Data Consistency:** Maintaining data consistency across multiple independent databases can be challenging.
    - **Inter-service Communication:** Needs robust communication mechanisms.[63]
- **Common Microservices Patterns:**
    - **API Gateway:**
        - **Concept:** A single entry point for all client requests. It acts as a reverse proxy that routes requests to the appropriate microservice.[64]
        - **Benefits:** Request routing, authentication/authorization, rate limiting, caching, load balancing, logging, request aggregation.
        - **Example:** Spring Cloud Gateway, Netflix Zuul.
    - **Service Discovery:**
        - **Concept:** A mechanism for services to register themselves and for clients to find instances of services.
        - **Why needed:** In a microservices architecture, service instances are dynamically created/destroyed, and their network locations change frequently.[65]
        - **Types:**
            - **Client-Side Discovery:** Client queries a service registry (e.g., Eureka, Consul) to find available service instances.[66]
            - **Server-Side Discovery:** Load balancer queries the registry and routes requests to instances.

        - **Example:** Netflix Eureka, Apache ZooKeeper, HashiCorp Consul.
    - **Circuit Breaker:**
        - **Concept:** Prevents cascading failures in a distributed system. If a service is repeatedly failing, the circuit breaker "trips" (opens) and stops sending requests to that service for a period, allowing it to recover.[67]
        - **Example:** Resilience4j, Netflix Hystrix (deprecated).
    - **Centralized Configuration:**
        - **Concept:** Externalizes application configuration (e.g., database URLs, API keys) into a central repository.[68]
        - **Example:** Spring Cloud Config Server, HashiCorp Vault.
    - **Load Balancing:** Distributing incoming network traffic across multiple servers to ensure no single server is overloaded.[69] (Client-side with Ribbon/Spring Cloud LoadBalancer, or server-side with Nginx/ELB).
- **Basic Understanding of Communication Between Microservices:**
    - **Synchronous Communication (REST/HTTP):**
        - **Mechanism:** One service makes a direct HTTP request (GET, POST, etc.) to another service and waits for a response.
        - **Pros:** Simple to implement, easy to debug, ubiquitous.
        - **Cons:** Tightly coupled (services need to know each other's endpoints), blocking (calling service waits), cascading failures.
    - **Asynchronous Communication (Messaging Queues):**
        - **Mechanism:** Services communicate by sending messages to and receiving messages from a message broker (e.g., Kafka, RabbitMQ, ActiveMQ).
        - **Pros:** Loose coupling, improved resilience (messages can be retried), scalability, supports pub/sub patterns.
        - **Cons:** Increased complexity, message ordering can be tricky, requires a message broker infrastructure.

Spring Boot and Microservices are vast topics, but for a 3-year experienced role, you're expected to have hands-on experience building REST APIs with Spring Boot, understanding how Spring manages dependencies, interacting with databases using Spring Data JPA, and a good conceptual grasp of microservices architecture and common patterns. Be prepared to discuss your project experiences with these technologies!

Alright, let's break down **"6. Database Technologies & ORM"** in detail. This topic is fundamental for any backend developer, and for someone with 3 years of experience, you're expected to have a solid

understanding of both relational databases and how Java applications interact with them, specifically through JPA/Hibernate, which we briefly touched upon in the previous topic.

---

6. Database Technologies & ORM: Storing and Managing Data

This section covers the core concepts of databases, particularly relational databases, SQL, and the crucial role of Object-Relational Mapping (ORM) tools like JPA and Hibernate in Java applications.

6.1. RDBMS Concepts & SQL

- **RDBMS (Relational Database Management System):**
    - **Concept:** A type of database that stores and provides access to data points that are related to one another. Data is organized into tables (relations), which consist of rows (records) and columns (attributes).
    - **Key Characteristics:**
        - **Structured Data:** Data is organized in a predefined schema.
        - **Tables:** Data stored in tables with rows and columns.
        - **Relationships:** Relationships between tables are established using primary keys (PK) and foreign keys (FK).
        - **Schema-on-Write:** Schema must be defined before data can be inserted.
        - **ACID Properties:** Guarantees data integrity and reliability for transactions.
    - **Examples:** MySQL, PostgreSQL, Oracle, SQL Server, H2 (in-memory for testing).
- **ACID Properties (for Transactions):**
    - **Transaction:** A single logical unit of work that either completes entirely or fails entirely.
    - **Atomicity:**
        - **Concept:** A transaction is treated as a single, indivisible unit. Either all operations within the transaction succeed, or none of them do. If any part of the transaction fails, the entire transaction is rolled back to its initial state.
        - **Analogy:** A money transfer from account A to B. It's either debit A and credit B, or neither.
    - **Consistency:**
        - **Concept:** A transaction brings the database from one valid state to another valid state. It ensures that all data integrity rules (e.g., foreign key constraints, unique constraints, check constraints) are maintained before and after the transaction.
        - **Analogy:** After a money transfer, the total sum of money across accounts remains the same.
    - **Isolation:**

        - **Concept:** Concurrent transactions do not interfere with each other. Each transaction executes as if it were the only transaction running in the system. The intermediate state of a transaction is not visible to other concurrent transactions.
        - **Analogy:** Multiple people can transfer money simultaneously, but each transfer appears to happen in isolation, without seeing partial results of others.
        - **Transaction Isolation Levels (Important for interview):** Define the degree to which one transaction's uncommitted changes are visible to other concurrent transactions.
            - **Read Uncommitted (Dirty Read):** A transaction can read data that has been modified by another transaction but not yet committed. Highly problematic due to dirty reads.
            - **Read Committed (Default in most databases like PostgreSQL, SQL Server):** A transaction can only read data that has been committed by other transactions. Prevents dirty reads.
            - **Repeatable Read (Default in MySQL):** A transaction sees only the data that was committed before the transaction started. Any rows it reads will appear the same if read again within the same transaction. Prevents dirty reads and non-repeatable reads. However, it can suffer from "phantom reads" (new rows inserted by other transactions might appear).
            - **Serializable (Highest isolation, lowest concurrency):** Full isolation. Transactions are executed in a serial fashion, as if they were running one after another. Prevents dirty reads, non-repeatable reads, and phantom reads. Comes with significant performance overhead.
    - **Durability:**
        - **Concept:** Once a transaction has been committed, its changes are permanently recorded in the database and survive system failures (e.g., power loss, crashes). Data is written to non-volatile storage.
        - **Analogy:** Once a transfer is confirmed, even if the system crashes, the change is permanent.
- **SQL (Structured Query Language):**
    - The standard language for managing and manipulating relational databases.
    - **DML (Data Manipulation Language):** For manipulating data within tables.
        - SELECT: Retrieve data.
        - INSERT: Add new rows.
        - UPDATE: Modify existing rows.
        - DELETE: Remove rows.

    - **DDL (Data Definition Language):** For defining and managing database schema.
        - CREATE: Create tables, databases, indexes, etc.
        - ALTER: Modify table structure.
        - DROP: Delete tables, databases.
        - TRUNCATE: Remove all rows from a table (DDL, faster than DELETE, cannot be rolled back).
    - **DCL (Data Control Language):** For managing permissions and access.
        - GRANT: Give users privileges.
        - REVOKE: Remove users' privileges.
    - **TCL (Transaction Control Language):** For managing transactions.
        - COMMIT: Make changes permanent.
        - ROLLBACK: Undo changes.
        - SAVEPOINT: Set a point within a transaction to which you can roll back.
- **Common SQL Constructs:**
    - JOIN (INNER, LEFT, RIGHT, FULL OUTER)
    - WHERE clause
    - GROUP BY clause with aggregate functions (COUNT, SUM, AVG, MIN, MAX)
    - HAVING clause (filters GROUP BY results)
    - ORDER BY clause
    - LIMIT/OFFSET (for pagination)
    - Subqueries, UNION, INTERSECT, EXCEPT.
- **Indexes:**
    - **Concept:** A special lookup table that the database search engine can use to speed up data retrieval operations. Like an index in a book.
    - **Types:**
        - **Clustered Index:** Determines the physical order of data storage in the table. A table can have only one clustered index (often the Primary Key).
        - **Non-Clustered Index:** A logical ordering of data. Stores a copy of the indexed columns and pointers to the actual data rows. A table can have multiple non-clustered indexes.
    - **Benefits:** Significantly speeds up SELECT queries, especially with WHERE clauses, JOINs, and ORDER BY clauses.
    - **Drawbacks:**

        - Increases storage space.
        - Slows down INSERT, UPDATE, and DELETE operations because the index itself must also be updated.
    - **When to Use:** On columns frequently used in WHERE clauses, JOIN conditions, ORDER BY clauses, or for enforcing uniqueness (UNIQUE constraint often implies an index).

6.2. JDBC (Java Database Connectivity)

- **Concept:** A Java API that provides a standard way for Java applications to connect to and interact with various relational databases.
- **Architecture:**
    - **Application:** Your Java code.
    - **JDBC API:** The interfaces and classes defined in java.sql and javax.sql.
    - **JDBC Driver Manager:** Manages the various JDBC drivers.
    - **JDBC Driver:** A vendor-specific implementation (e.g., MySQL Connector/J) that translates JDBC API calls into the database's native protocol.
    - **Database:** The actual relational database.
- **Typical Steps to use JDBC:**
    1. Load the JDBC driver (Class.forName("com.mysql.cj.jdbc.Driver"); - usually implicit now).
    2. Establish a Connection (DriverManager.getConnection(url, username, password)).
    3. Create a Statement or PreparedStatement.
        - **Statement:** For simple, static SQL queries. Prone to SQL injection.
        - **PreparedStatement:** (Highly Recommended) For parameterized queries. Prevents SQL injection, improves performance for repeated queries.
    4. Execute the query (executeQuery for SELECT, executeUpdate for INSERT/UPDATE/DELETE).
    5. Process the ResultSet (if it's a SELECT query).
    6. Close resources (in finally blocks): ResultSet, Statement, Connection (in that order).
- **Problem with JDBC:** Verbose, requires lots of boilerplate code for common operations, error-prone (forgetting to close resources). This led to the development of ORMs.

6.3. JPA (Java Persistence API)

- **Concept:** A **specification** for object-relational mapping (ORM) in Java. It defines how to manage relational data in Java applications using an object-oriented approach. It's a standard API for persistence.
- **Key Features:**

- **Annotations:** Uses annotations (e.g., @Entity, @Table, @Id) to map Java classes to database tables and fields to columns.
- **EntityManager:** The primary interface for interacting with the persistence context (a set of managed entities). Used for performing CRUD operations, finding entities, and querying.
- **JPQL (Java Persistence Query Language):** An object-oriented query language defined by JPA. It queries entities and their relationships, not directly database tables.

- **Why JPA?** It hides the complexity of JDBC, allowing developers to focus on business logic rather than SQL.

6.4. Hibernate

- **Concept:** A powerful, open-source **ORM framework** that is a popular **implementation** of the JPA specification. While it can be used standalone, it's most commonly used as the underlying ORM provider for JPA in Spring applications.
- **Features:**
  - **Native SQL, HQL (Hibernate Query Language), Criteria API:** Provides various ways to query data. HQL is similar to JPQL but has some Hibernate-specific extensions.
  - **Caching (First-Level and Second-Level):**
    - **First-Level Cache (Session Cache):**
      - Default, enabled by EntityManager (or Session in native Hibernate).
      - Per-session cache. All entities loaded or persisted within a single EntityManager instance are stored here.
      - When you query an entity, Hibernate first checks the first-level cache. If found, it returns the cached instance; otherwise, it fetches from the database.
      - **Scope:** Short-lived, associated with the current EntityManager (transaction).
    - **Second-Level Cache (Shared Cache):**
      - Optional, needs to be explicitly enabled and configured (e.g., Ehcache, Redis).
      - Shared across multiple EntityManager (or Session) instances.
      - When an entity is loaded from the database, it's stored in the second-level cache. Subsequent requests for the same entity from different sessions can retrieve it from this cache without hitting the database.
      - **Scope:** Long-lived, shared across the entire application's EntityManagerFactory.
      - **Types:** Read-only, Read-write, Non-strict read-write, Transactional.

        - **When to Use:** For data that is frequently read and rarely updated.
    - **Dirty Checking:**
        - **Concept:** Hibernate automatically detects changes made to managed entities (entities loaded into the persistence context) without needing explicit update() calls.
        - **How it works:** When an entity is loaded, Hibernate stores a snapshot of its state. Before flushing (synchronizing changes to the database), it compares the current state of the entity with its snapshot. If changes are detected, an UPDATE statement is automatically generated and executed.
        - **Benefits:** Simplifies update operations, makes code cleaner.

6.5. N+1 Problem

- **Concept:** A common performance anti-pattern in ORM frameworks (like Hibernate/JPA). It occurs when, to retrieve a collection of "parent" entities, an initial query is executed (1 query), and then for each parent entity, a separate query is executed to fetch its "child" (associated) entities (N queries). This results in N+1 queries instead of a single, optimized query.
- **Scenario:**
    - Imagine Book and Author entities, where a Book has one Author (@ManyToOne) and an Author can have multiple Books (@OneToMany).
    - If you load N Author entities and then iterate through them to access their books collection (which is lazy-loaded), Hibernate will execute N separate queries to fetch the books for each author.
- **Example (Conceptual):**

Java

```
// N+1 problem likely if books collection is LAZY
List<Author> authors = authorRepository.findAll(); // 1 query to get N authors
for (Author author : authors) {
   System.out.println(author.getName());
   for (Book book : author.getBooks()) { // N queries, one for each author's books
      System.out.println("  - " + book.getTitle());
   }
}
```

- **Solutions to N+1 Problem:**
    1. **FetchType.EAGER (Use with Caution!):**
        - Can set the fetch type on the @OneToMany, @ManyToMany, @ManyToOne, @OneToOne annotations to FetchType.EAGER.

- **Problem:** Causes immediate fetching of all associated data, leading to potentially large result sets and performance issues if not all eager-fetched data is needed. Not a general solution.
- Example: @OneToMany(mappedBy = "author", fetch = FetchType.EAGER)

2. **JOIN FETCH in JPQL/HQL:** (Most common and recommended for specific use cases)
    - Explicitly tells the ORM to fetch the associated collection/entity in the same query using a JOIN clause.
    - Java
    - // In your Spring Data JPA repository:
    - @Query("SELECT a FROM Author a JOIN FETCH a.books")
    - List<Author> findAllAuthorsWithBooks();
    - **Benefits:** Fetches all necessary data in a single query, avoiding N+1.
    - **Drawbacks:** Can lead to a Cartesian product if multiple collections are eagerly fetched, which can bloat the result set.
3. **@EntityGraph (Spring Data JPA specific):**
    - Provides a way to define a "fetch plan" for entities and their associated relationships.
    - You can specify which attributes of an entity graph (e.g., entity itself and its associations) to fetch eagerly.
    - Java
    - // In your Spring Data JPA repository:
    - @EntityGraph(attributePaths = {"books"})
    - List<Author> findAll();
    - **Benefits:** Cleaner than JOIN FETCH for simple graph fetching, often preferred.
    - **Type:** Can be FETCH (forces joins) or LOAD (allows lazy loading for unspecifed attributes).
4. **Batch Fetching (e.g., batch_size in Hibernate):**
    - A less common but effective strategy. Hibernate can fetch lazy collections/entities in batches rather than one by one.
    - Configure hibernate.default_batch_fetch_size or @BatchSize annotation on relationships.
    - Example: @BatchSize(size = 10) on a @OneToMany relationship. When accessing the collection, instead of 1 query per parent, it might do 1 query for every 10 parents. Reduces N to N/batch_size.

5. **Separate Queries (if JOIN FETCH is not ideal):**
    - Sometimes, it's better to fetch entities and their related data in separate queries if the relations are complex or the data is not always needed together.
    - Can use Maps to store and associate entities.
6. **DTOs (Data Transfer Objects):**
    - Instead of fetching entire entities, fetch only the necessary data into a DTO. This avoids loading associated lazy collections at all if you only need a subset of data.
    - Often combined with constructor-based projections in Spring Data JPA (@Query("SELECT new com.example.dto.AuthorDto(a.id, a.name) FROM Author a")).

6.6. Database Normalization & Denormalization

- **Database Normalization:**
    - **Concept:** A systematic process of organizing the columns and tables of a relational database to minimize data redundancy and improve data integrity. It involves dividing large tables into smaller, more manageable tables and defining relationships between them.
    - **Normal Forms (1NF, 2NF, 3NF, BCNF):** A set of guidelines for database design.
        - **1NF (First Normal Form):** Each column must contain atomic (indivisible) values. No repeating groups.
        - **2NF (Second Normal Form):** Must be in 1NF, and all non-key attributes must be fully dependent on the primary key. (Eliminates partial dependencies).
        - **3NF (Third Normal Form):** Must be in 2NF, and all non-key attributes must not depend on other non-key attributes (eliminate transitive dependencies).
        - **BCNF (Boyce-Codd Normal Form):** A stronger version of 3NF, addresses certain types of anomalies not covered by 3NF.
    - **Benefits:**
        - Reduces data redundancy (less storage, easier to maintain).
        - Improves data integrity (avoids update, insert, delete anomalies).
        - Makes the database schema more flexible and easier to modify.
    - **Drawbacks:**
        - Can lead to more tables and more complex JOIN operations, potentially impacting read performance.
- **Database Denormalization:**

- **Concept:** The process of intentionally introducing redundancy into a database schema, often by combining data from multiple tables into a single table. It's typically done after a database has been normalized.
- **Purpose:** To improve read performance (especially for complex reports or frequently accessed queries) by reducing the number of JOIN operations required.
- **Trade-offs:**
  - **Pros:** Faster read queries, simpler queries.
  - **Cons:** Increases data redundancy, potential for update/insert/delete anomalies (data inconsistency), higher storage requirements.
- **When to Use:** When read performance is critical and outweighs the risks of data redundancy and potential inconsistency, especially in data warehousing, reporting, or heavily read-optimized applications. Careful management (e.g., using triggers or batch processes to maintain consistency) is often required.

A strong understanding of these database concepts, particularly SQL, ACID properties, transaction isolation levels, and ORM usage (JPA/Hibernate with Spring Data JPA), including performance considerations like the N+1 problem, is essential for a seasoned Java developer. Be ready to discuss specific examples from your projects.

Alright, let's break down **"7. Testing"** in detail. For a Java developer with 3 years of experience, testing is not just a concept but a critical part of the daily development workflow. You're expected to write effective tests, understand different testing levels, and use modern testing frameworks and practices.

---

7. Testing: Ensuring Code Quality and Reliability

Testing is an integral part of the software development lifecycle, aimed at verifying that an application functions as expected, meets requirements, and is free of defects. In modern Java development, a robust testing strategy is crucial.

7.1. Unit Testing

- **Concept:** The smallest level of testing, where individual units or components of a software are tested in isolation from the rest of the application.
  - **Unit:** Typically a single class or even a single method.
- **Purpose:** To verify that each unit of the source code performs exactly as intended.
- **Characteristics:**
  - **Isolated:** Units are tested independently, often using **mocks** or **stubs** to isolate them from their dependencies (e.g., databases, external services, other complex classes).

    - **Fast:** Unit tests should execute very quickly, allowing developers to run them frequently.
    - **Automated:** Typically automated and run as part of the build process.
    - **High Coverage:** Aims for high code coverage to ensure most of the logic is tested.
- **Frameworks/Tools:**
    - **JUnit 5 (Jupiter, Platform, Vintage):** The de-facto standard testing framework for Java.
        - **@Test:** Marks a method as a test method.
        - **@DisplayName:** Provides a more readable name for the test.
        - **@BeforeEach / @AfterEach:** Methods run before/after each test method in a class.
        - **@BeforeAll / @AfterAll:** Methods run once before/after all test methods in a class (must be static).
        - **@Disabled:** Skips a test.
        - **Assertions class:** Provides static methods for asserting expected outcomes (e.g., assertEquals, assertTrue, assertNotNull, assertThrows).
        - **Parameterized Tests (@ParameterizedTest, @ValueSource, @CsvSource, @MethodSource):** Run the same test method multiple times with different input arguments.
    - **Mockito:** A popular mocking framework for Java.
        - **Purpose:** To create **mock objects** for dependencies that are too complex, slow, or external to be used in a unit test.
        - **@Mock:** Creates a mock object.
        - **@InjectMocks:** Injects the mocks into the object under test.
        - **when().thenReturn():** Defines the behavior of a mocked method call.
        - **verify():** Verifies that a method on a mock object was called a certain number of times or with specific arguments.
        - **any(), eq():** Argument matchers for when() and verify().
- **Example (JUnit + Mockito):**

Java

```
import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;
```

```
import org.mockito.junit.jupiter.MockitoExtension;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.times;

// A simple service we want to test
class UserService {
    private UserRepository userRepository; // Dependency

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

    public String getUserFullName(Long id) {
        User user = userRepository.findById(id);
        if (user != null) {
            return user.getFirstName() + " " + user.getLastName();
        }
        return null;
    }

    public void createUser(String firstName, String lastName) {
        User newUser = new User(firstName, lastName);
        userRepository.save(newUser);
    }
}

// A dependency interface
interface UserRepository {
```

```
    User findById(Long id);
    void save(User user);
}

// A simple POJO
class User {
    private String firstName;
    private String lastName;

    public User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
}

@ExtendWith(MockitoExtension.class) // Integrates Mockito with JUnit 5
class UserServiceTest {

    @Mock // Creates a mock instance of UserRepository
    private UserRepository userRepository;

    @InjectMocks // Injects the mocked userRepository into userService
    private UserService userService;

    @Test
    void testGetUserFullName_ExistingUser() {
        // Define mock behavior: when findById(1L) is called, return a specific User object
        when(userRepository.findById(1L)).thenReturn(new User("John", "Doe"));
```

```
        String fullName = userService.getUserFullName(1L);

        // Assertions
        assertEquals("John Doe", fullName);

        // Verify that findById was called exactly once with argument 1L
        verify(userRepository, times(1)).findById(1L);
    }

    @Test
    void testGetUserFullName_NonExistingUser() {
        when(userRepository.findById(2L)).thenReturn(null);

        String fullName = userService.getUserFullName(2L);

        assertEquals(null, fullName);
        verify(userRepository, times(1)).findById(2L);
    }

    @Test
    void testCreateUser() {
        userService.createUser("Jane", "Smith");

        // Verify that save was called exactly once with any User object
        verify(userRepository, times(1)).save(Mockito.any(User.class));
    }
}
```

7.2. Integration Testing

- **Concept:** Tests the interaction between different components or layers of an application. It verifies that modules or services work together correctly.

- **Purpose:** To find defects in the interfaces and interactions between integrated components.
- **Characteristics:**
    - **Less Isolated:** Involves multiple components, possibly including real databases, external services, or message queues.
    - **Slower:** Can take longer to run due to external dependencies and setup.
    - **Automated:** Still automated, often run less frequently than unit tests.
- **Frameworks/Tools:**
    - **Spring Boot Test:** Provides excellent support for writing integration tests for Spring Boot applications.
        - **@SpringBootTest:** Loads the full Spring application context (or a slice of it). Can be configured to run with a specific WebEnvironment (e.g., RANDOM_PORT).
        - **@Autowired:** Can inject actual Spring beans from the context into the test.
        - **@DataJpaTest:** Specifically for testing JPA repositories. It sets up an in-memory database (e.g., H2) and auto-configures JPA components. Useful for testing database interactions without loading the full web layer.
        - **@WebMvcTest:** Specifically for testing Spring MVC controllers. It auto-configures Spring MVC components but doesn't load the full application context or data layer. Often used with MockMvc.
    - **MockMvc:** Provided by Spring Test. Used for testing Spring MVC controllers without starting a full HTTP server. It simulates HTTP requests and responses.
    - **Testcontainers:** A powerful library that allows you to spin up lightweight, throwaway instances of databases, message brokers, web browsers, or anything else that can run in a Docker container for your tests. Ideal for true integration tests with real dependencies.
    - **WireMock:** A mock server for HTTP-based APIs. Useful for simulating external service dependencies in integration tests.
- **Example (Spring Boot Integration Test with MockMvc and @WebMvcTest):**

Java

```
import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

import org.springframework.boot.test.mock.mockito.MockBean;

import org.springframework.test.web.servlet.MockMvc;


import static org.hamcrest.Matchers.containsString;
```

```
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

// Assume a simple controller
@RestController
class MyController {
    private MyService myService; // Assume MyService is a dependency

    public MyController(MyService myService) {
        this.myService = myService;
    }

    @GetMapping("/hello")
    public String hello() {
        return myService.greet();
    }
}

interface MyService { // Service dependency
    String greet();
}

@WebMvcTest(MyController.class) // Tests only the web layer (MyController)
class MyControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc; // Used to simulate HTTP requests

    @MockBean // Mocks MyService and adds it to the Spring context
```

```
    private MyService myService;


    @Test
    void testHelloEndpoint() throws Exception {
        // Define behavior for the mocked service
        when(myService.greet()).thenReturn("Hello from service!");


        mockMvc.perform(get("/hello")) // Perform a GET request to /hello
                .andExpect(status().isOk()) // Expect HTTP 200 OK
                .andExpect(content().string(containsString("Hello from service!"))); // Expect specific content
    }
}
```

7.3. Test Driven Development (TDD)

- **Concept:** A software development process where tests are written *before* the actual code that implements the functionality. It's an iterative development approach.
- **The "Red-Green-Refactor" Cycle:**
    1. **Red (Write a failing test):** Write a new unit test for a small piece of functionality that doesn't exist yet. The test should fail.
    2. **Green (Make the test pass):** Write *just enough* production code to make the failing test pass. Don't worry about perfect design or optimization yet.
    3. **Refactor (Improve the code):** Once the test passes, refactor the production code (and potentially the test code) to improve its design, readability, and performance, ensuring all tests still pass.
    4. Repeat the cycle.
- **Benefits:**
    - **Forces Good Design:** Encourages writing small, focused, testable units of code.
    - **Improved Code Quality:** Leads to cleaner, more modular code with fewer bugs.
    - **Built-in Regression Suite:** Every new feature adds to your safety net of tests.
    - **Reduces Debugging Time:** Issues are caught early.
    - **Clearer Requirements:** Writing tests first clarifies understanding of requirements.
    - **Confidence in Changes:** Refactoring and adding features is less risky with a comprehensive test suite.

- **Drawbacks:**
    - Initial learning curve.
    - Can feel slower initially (though faster in the long run).
    - Requires discipline.

7.4. Mocking vs. Spying

Both Mockito features, used to control the behavior of dependencies in tests.

- **Mocking (@Mock):**
    - **Concept:** Creates a completely fake object. It's a "dummy" implementation of an interface or class that you programmatically control.
    - **Behavior:** By default, mocks do nothing. You *must* explicitly define the behavior of any method calls you expect on the mock using when().thenReturn() (stubbing).
    - **Use Case:** When you want to isolate the code under test completely from its dependencies, especially when dependencies are complex, external, or have side effects. You're testing *how* the code under test interacts with its dependencies.
    - **Example:** UserRepository in the Unit Testing example above. We don't want to hit a real database; we just want to control what findById returns.
- **Spying (@Spy or Mockito.spy()):**
    - **Concept:** Creates a **partial mock** or a "spy." It wraps a *real* object.
    - **Behavior:** By default, a spy invokes the *real methods* of the actual object. You can selectively override (stub) certain method calls if needed, while other methods will still call the original implementation.
    - **Use Case:** When you want to test a real object but need to mock *only specific methods* of that object, or when you want to verify calls on a real object. This is useful for legacy code or when the object itself has internal state you want to preserve for some methods, while mocking others.
    - **Example:** If you have a complex PaymentProcessor that performs many steps, and you only want to mock the final sendToGateway() call while letting other calculation methods run normally.
- **Key Difference Summary:**
    - **Mock:** A completely fake object. You tell it *what to do*.
    - **Spy:** A real object that you observe/partially override. It does its *real work* unless you explicitly tell it otherwise.
- **Example (Spying):**

Java

```
import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;
```

```
import org.mockito.Spy;
import org.mockito.junit.jupiter.MockitoExtension;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.times;

class Calculator {
    public int add(int a, int b) {
        System.out.println("Real add method called.");
        return a + b;
    }
    public int multiply(int a, int b) {
        System.out.println("Real multiply method called.");
        return a * b;
    }
}

@ExtendWith(MockitoExtension.class)
class CalculatorTest {

    @Spy // Spy on a real Calculator instance
    private Calculator calculator; // Mockito will create a real instance and wrap it

    @Test
    void testAddWithSpy() {
        int result = calculator.add(2, 3); // Calls the real add method
        assertEquals(5, result);
        verify(calculator, times(1)).add(2, 3); // Verify the real method was called
    }
```

```
    @Test
    void testMultiplyWithSpyAndStubbing() {
        // Stub the multiply method to return a specific value
        when(calculator.multiply(2, 3)).thenReturn(100);

        int result = calculator.multiply(2, 3); // Calls the stubbed behavior
        assertEquals(100, result);
        verify(calculator, times(1)).multiply(2, 3); // Verify it was called
    }
}
```

7.5. Code Coverage

- **Concept:** A metric that measures the percentage of your application's source code that is executed by your test suite. It helps identify untested areas of your code.
- **Types of Coverage:**
    - **Line Coverage:** Percentage of executable lines that were executed.
    - **Branch Coverage / Decision Coverage:** Percentage of if/else, switch, loops where all branches (true/false paths) were executed.
    - **Method Coverage:** Percentage of methods that were called.
    - **Class Coverage:** Percentage of classes that were loaded.
- **Tools:**
    - **JaCoCo (Java Code Coverage):** A popular open-source code coverage library for Java. Integrates well with Maven/Gradle. Generates reports (HTML, XML, CSV) showing coverage percentages and highlighting uncovered lines.
    - **SonarQube:** A platform for continuous inspection of code quality and security. Integrates with JaCoCo to display coverage metrics alongside other code quality issues.
- **Importance:**
    - Provides an indicator of how thoroughly your code is being tested.
    - Helps identify "dead code" (unreachable code).
    - Guides future testing efforts (focus on low-coverage areas).
- **Limitations:** High code coverage does **not** guarantee a bug-free application. It only tells you *what* code was run, not *if* the code works correctly for all possible scenarios or edge cases.

It's a quantitative metric, not qualitative. You can have 100% coverage but still have logical bugs if your assertions are weak or you miss edge cases in your tests.

For a 3-year experienced Java developer, you should be comfortable not only writing unit and integration tests but also explaining their purpose, the trade-offs, and the tools involved. You should be familiar with TDD principles and why it's beneficial. Understanding the nuances of mocking vs. spying and the meaning of code coverage are also key indicators of a solid testing background.

Got it! Let's delve into **"8. Version Control (Git)"** in detail. For a Java developer (or any developer for that matter) with 3 years of experience, a deep understanding of Git is non-negotiable. It's the backbone of collaborative development and project management.

---

8. Version Control (Git): Managing Code Changes

Version Control Systems (VCS) are tools that help software teams manage changes to source code over time. They keep track of every modification made to the code, allowing developers to revert to previous versions, compare changes, and collaborate efficiently without overwriting each other's work.

Git is by far the most popular distributed version control system (DVCS).

8.1. Why Version Control?

- **Tracking Changes:** Records who changed what, when, and why.
- **Collaboration:** Enables multiple developers to work on the same codebase simultaneously without conflicts.
- **Reversion:** Easily revert to any previous state of the code.
- **Branching & Merging:** Facilitates parallel development of features and bug fixes without affecting the main codebase.
- **Backup & Disaster Recovery:** The entire history is distributed, providing redundancy.
- **Audit Trail:** Provides a history for compliance and debugging.

8.2. Git Basics

- **Distributed Version Control System (DVCS):**
    - Unlike centralized VCS (like SVN, Perforce) where there's a single central repository, in a DVCS, every developer has a **complete copy of the entire repository** (including its full history) on their local machine.
    - **Benefits:**
        - **Offline Work:** Developers can commit changes locally even without network access.
        - **Faster Operations:** Most operations (commits, diffs, Browse history) are done locally, making them very fast.

        - **Resilience:** No single point of failure; if the central server goes down, development can continue using local copies.
        - **Flexibility:** Easier to experiment with branches.

- **Core Concepts:**
    - **Repository (Repo):** A .git directory at the root of your project that contains all the history, commits, and other metadata for your project.
    - **Commit:** A snapshot of your project's files at a specific point in time. Each commit has a unique SHA-1 hash, a commit message, author, date, and a pointer to its parent commit(s).
    - **Branch:** A lightweight, movable pointer to a commit. Branches allow you to diverge from the main line of development and work on new features or bug fixes in isolation.
    - **Head:** A pointer to the tip of the current branch.
    - **Index (Staging Area/Cache):** An intermediate area between your working directory and your repository. It's where you prepare the next commit. Files in the staging area are the ones that will be included in the next commit.
    - **Working Directory:** The actual files you are currently working on in your file system.
- **Common Commands & Workflow:**
    1. **git init:** Initializes a new Git repository in the current directory.
    2. **git clone <repository_url>:** Copies an existing remote repository to your local machine.
    3. **git status:** Shows the status of your working directory and staging area (which files are modified, staged, or untracked).
    4. **git add <file(s)> / git add .:** Adds changes from the working directory to the staging area.
    5. **git commit -m "Your commit message":** Records the staged changes to the repository. The commit message should be descriptive.
    6. **git log:** Displays the commit history.
        - git log --oneline: Concise log.
        - git log --graph --oneline --all: Visualizes branch history.
    7. **git diff:** Shows changes between different states:
        - git diff: Shows changes in the working directory not yet staged.
        - git diff --staged: Shows changes in the staging area not yet committed.
        - git diff <commit1> <commit2>: Shows difference between two commits.
    8. **git push:** Uploads your local commits to a remote repository.

9. **git pull:** Fetches changes from a remote repository and merges them into your current local branch. (Equivalent to git fetch followed by git merge).
10. **git fetch:** Downloads commits, files, and refs from a remote repository into your local repository, but **does not merge** them into your working branch.

8.3. Branching Strategies

Branching is Git's superpower for parallel development.

- **git branch:**
    - git branch: Lists all local branches.
    - git branch <branch_name>: Creates a new branch.
    - git branch -d <branch_name>: Deletes a local branch (only if merged).
    - git branch -D <branch_name>: Forces deletion of a local branch (even if unmerged).
- **git checkout <branch_name>:** Switches to an existing branch.
- **git checkout -b <new_branch_name>:** Creates a new branch and switches to it. (Shorthand for git branch <new_branch_name> then git checkout <new_branch_name>).
- **git merge <branch_to_merge>:** Integrates changes from a specified branch into the current branch.
    - **Fast-Forward Merge:** If the target branch has no new commits since the source branch branched off, Git simply moves the pointer forward.
    - **Three-Way Merge:** If both branches have diverged, Git creates a new "merge commit" that combines the changes from both.
- **git rebase <base_branch>:**
    - **Concept:** Rewrites commit history. It takes your commits from your current branch and reapplies them one by one *on top* of the specified <base_branch>.
    - **Result:** Creates a linear history, avoiding merge commits.
    - **When to Use:** To incorporate changes from the main branch into your feature branch before merging, making the history cleaner for a later fast-forward merge.
    - **Caution: Never rebase public/shared branches** (like main or develop) because it rewrites history, which can cause significant problems for other collaborators. Only rebase your private feature branches.
- **Common Strategies:**
    - **Git Flow:** A more complex, strict branching model with long-running branches (master, develop) and supporting branches (feature, release, hotfix). Good for larger, regulated projects.
    - **GitHub Flow / GitLab Flow:** Simpler, lightweight strategies often centered around main (or master) branch. Features are developed on short-lived branches, merged

into main, and then main is directly deployed. Preferred for continuous delivery and smaller teams.

8.4. Handling Conflicts

- **What is a Conflict?**
    - Occurs when Git tries to merge two branches, and the same line(s) of code (or the same file) have been modified differently in both branches. Git cannot automatically decide which change to keep.
- **When Conflicts Happen:**
    - During git merge.
    - During git rebase.
    - During git pull (which performs a fetch and merge).
- **Resolution Process:**
    1. Git will mark the conflicting sections in the file using special markers (<<<<<<<, =======, >>>>>>>).
    2. You manually edit the file to resolve the conflict (decide which version to keep, or combine them).
    3. After editing, git add <conflicted_file> to stage the resolved file.
    4. git commit -m "Merge conflict resolved" (if merging) or git rebase --continue (if rebasing).
- **Example of Conflict Markers:**
- <<<<<<< HEAD (or current branch)
- This is line A in my current branch.
- =======
- This is line A from the incoming branch.
- >>>>>>> feature/new-feature (or incoming branch)

You would edit this section to:

This is the combined and resolved line A.

8.5. git reset vs. git revert

Both commands are used to undo changes, but they do so in fundamentally different ways.

- **git revert <commit_hash>:**
    - **Concept:** Creates a *new commit* that undoes the changes introduced by the specified commit. It does not rewrite history.
    - **Effect:** The original commit still exists in the history, but its effects are cancelled out by the new revert commit.

- **When to Use:** When you need to undo changes on a **public/shared branch** (e.g., main, develop) because it preserves the history and doesn't affect other developers' clones.
- **Analogy:** You made a mistake, so you wrote a new instruction to undo the mistake, but the original instruction is still on the record.

- **git reset <mode> <commit_hash>:**
  - **Concept:** Moves the HEAD pointer (and potentially the branch pointer and index/working directory) to a specified commit. It effectively **rewrites history** by discarding subsequent commits.
  - **Modes:**
    - **--soft:** Moves HEAD and the current branch pointer to the target commit. The changes from the "undone" commits are kept in the **staging area**.
    - **--mixed (Default):** Moves HEAD and the current branch pointer. The changes are kept in the **working directory** (unstaged).
    - **--hard:** Moves HEAD and the current branch pointer. **Discards all changes** (from working directory, staging area, and commits) from the target commit onwards. **DANGEROUS! Use with extreme caution as data can be lost.**
  - **When to Use:** To undo changes on your **local, private feature branches** before you have pushed them to a remote repository. It keeps your history clean and linear.
  - **Analogy:** You erased the mistake from the record as if it never happened.

- **Summary Table:**

Feature	git revert	git reset
History	**Does NOT rewrite history.** Adds a new commit.	**Rewrites history.** Discards subsequent commits.
Safety	Safe for shared/public branches.	Dangerous on shared branches (causes divergence). Safe on private branches.
Action	Undoes changes by adding an inverse commit.	Moves branch pointer (and data depending on mode).
Goal	Undo changes publicly, keep history.	Clean up private history, discard changes.

8.6. git stash

- **Concept:** Temporarily saves changes that you don't want to commit immediately, allowing you to switch branches or perform other operations without committing incomplete work.

- **Use Case:**
    - You're working on a feature, and a high-priority bug needs fixing on main. You don't want to commit incomplete work. Stash your changes, switch to main, fix the bug, then come back to your feature branch and reapply the stash.
- **Commands:**
    - **git stash save "message" / git stash:** Saves your current uncommitted changes (both staged and unstaged) to a temporary stash. Your working directory becomes clean.
    - **git stash list:** Shows a list of stashed changes.
    - **git stash apply:** Applies the latest stash back to your working directory, but *keeps* the stash in the stash list.
    - **git stash pop:** Applies the latest stash and then *removes* it from the stash list. (More common)
    - **git stash drop:** Removes a specific stash from the stash list.
    - **git stash clear:** Removes all stashes.

8.7. .gitignore

- **Concept:** A text file (.gitignore) placed in the root of your repository that tells Git which files or directories to **ignore** and not track.
- **Purpose:** To prevent unnecessary or undesirable files from being committed to the repository.
- **Commonly Ignored Files/Directories:**
    - Compiled class files (.class, .jar, .war)
    - IDE-specific files (e.g., .idea/ for IntelliJ, .vscode/ for VS Code)
    - Build tool outputs (target/ for Maven, build/ for Gradle)
    - Log files (*.log)
    - Dependency directories (node_modules/, vendor/)
    - Temporary files, configuration files with sensitive data (e.g., application-dev.properties if it contains local DB credentials).
- **Syntax:**
    - # for comments
    - filename.txt: Ignores a specific file.
    - *.log: Ignores all files ending with .log.
    - /dir/: Ignores a directory and its contents.
    - dir/: Ignores a directory anywhere in the repo.
    - !file.txt: Excludes a previously ignored file (negation).

For a 3-year experienced developer, you should be able to perform all these Git operations confidently, understand the implications of commands like rebase and reset, explain common branching strategies, and effectively resolve merge conflicts. Git is a tool you'll use every day, so proficiency is paramount.

No problem, let's dive into **"9. Project Management Tools & Methodologies"** in detail. For a 3-year experienced Java developer, understanding these concepts isn't just about coding; it's about how software projects are organized, delivered, and how teams collaborate effectively. You'll likely be part of, or even lead, aspects of these processes.

---

9. Project Management Tools & Methodologies: Organizing Software Development

This topic covers the frameworks and tools used to plan, execute, and monitor software development projects. The goal is to deliver high-quality software efficiently and effectively.

9.1. Agile Methodologies

Agile is an iterative and incremental approach to project management and software development that helps teams deliver value to their customers faster and with fewer headaches. It emphasizes flexibility, collaboration, and rapid response to change.

- **Core Principles (from Agile Manifesto):**
    - **Individuals and interactions** over processes and tools.
    - **Working software** over comprehensive documentation.
    - **Customer collaboration** over contract negotiation.
    - **Responding to change** over following a plan.
- **Key Characteristics:**
    - **Iterative & Incremental:** Projects are broken into small, manageable iterations (sprints/cycles) where a small piece of working software is delivered.
    - **Customer Collaboration:** Continuous feedback from the customer is encouraged.
    - **Self-Organizing Teams:** Teams are empowered to decide how to best achieve their goals.
    - **Adaptive Planning:** Plans are flexible and evolve as requirements become clearer.
    - **Continuous Improvement:** Teams regularly reflect on their work and find ways to improve.
- **Common Agile Frameworks:**
    - **Scrum:**
        - The most popular Agile framework. It provides a lightweight framework for developing and delivering complex products.

        - **Key Roles:**
            - **Product Owner:** Represents the voice of the customer. Manages the Product Backlog, ensures value, prioritizes features.
            - **Scrum Master:** A servant-leader who facilitates the Scrum process, removes impediments, and coaches the team on Agile principles. Not a project manager or team lead in the traditional sense.
            - **Development Team:** A self-organizing, cross-functional group responsible for delivering the product increment. Typically 3-9 members.
        - **Key Events (Ceremonies):**
            - **Sprint Planning:** (Beginning of Sprint) Team selects items from the Product Backlog to work on during the sprint and plans how to achieve the Sprint Goal.
            - **Daily Scrum (Stand-up):** (Daily, 15 mins) Team members answer: "What did I do yesterday?", "What will I do today?", "Are there any impediments?".
            - **Sprint Review:** (End of Sprint) Inspects the Increment and adapts the Product Backlog if needed. Stakeholders provide feedback.
            - **Sprint Retrospective:** (End of Sprint) Team inspects how the last sprint went with regard to individuals, interactions, processes, and tools, and identifies improvements for the next sprint.
        - **Key Artifacts:**
            - **Product Backlog:** A prioritized, dynamic list of all known requirements for the product. Managed by the Product Owner.
            - **Sprint Backlog:** A subset of the Product Backlog selected for a sprint, plus the plan for delivering the increment. Managed by the Development Team.
            - **Increment (Potentially Shippable Product):** The sum of all Product Backlog items completed during a sprint and all previous sprints. It must be in a "Done" state.
        - **Sprint:** A time-boxed iteration (typically 1-4 weeks) during which a "Done," usable, and potentially releasable product Increment is created.
    - **Kanban:**
        - Focuses on visualizing workflow, limiting work in progress (WIP), and maximizing flow.
        - **Key Principles:**
            - Visualize the workflow (Kanban board with columns like "To Do," "In Progress," "Done").

        - Limit Work in Progress (WIP) to prevent bottlenecks and ensure focus.
        - Manage flow (optimize the speed and smoothness of items moving through the workflow).
        - Make process policies explicit.
        - Implement feedback loops.
        - Improve collaboratively, evolve experimentally.
    - **Use Case:** Ideal for maintenance teams, support teams, or workflows with unpredictable demand where continuous flow is more important than fixed iterations.

9.2. Scrum vs. Kanban (Key Differences)

Feature	Scrum	Kanban
Cadence	Time-boxed sprints (e.g., 2-4 weeks)	Continuous flow, no fixed iterations
Roles	Defined roles (Product Owner, Scrum Master, Dev Team)	No prescribed roles, team-driven
Artifacts	Product Backlog, Sprint Backlog, Increment	Kanban Board, WIP Limits, Flow metrics
Meetings	Daily Scrum, Sprint Planning, Review, Retro	No prescribed meetings, focus on ad-hoc sync
Change	Changes discouraged within a sprint	Changes can be introduced at any time
Goals	Deliver potentially shippable increment per sprint	Improve flow, reduce lead time, deliver continuously
WIP Limits	Implicit (via Sprint Planning)	Explicitly defined on the board

Export to Sheets

9.3. Project Management Tools

These tools facilitate the implementation of Agile and other methodologies, providing platforms for task management, collaboration, and tracking.

- **Jira:**
    - **Type:** Issue tracking and project management software.
    - **Features:** Highly configurable for Scrum, Kanban, and custom workflows.
        - **Issue Types:** Supports various issue types like Stories, Tasks, Bugs, Epics.
        - **Workflows:** Customizable workflows to define the lifecycle of issues.

        - **Boards:** Scrum Boards (sprints, burndown charts) and Kanban Boards (WIP limits, swimlanes).
        - **Reporting:** Dashboards, various agile reports (velocity, sprint reports).
        - **Integrations:** Integrates with Git (Bitbucket, GitHub), CI/CD tools (Jenkins), Confluence, etc.
    - **Use Case:** Widely used in software development for tracking features, bugs, tasks, and managing Agile projects.
- **Trello:**
    - **Type:** Simple, visual project management tool based on Kanban boards.
    - **Features:**
        - **Boards, Lists, Cards:** Core elements. Boards represent projects, lists represent stages, cards represent tasks.
        - **Checklists, Due Dates, Attachments:** Basic task management features.
        - **Power-Ups:** Integrations with other tools (e.g., Slack, Google Drive).
    - **Use Case:** Excellent for small teams, personal task management, or simple projects where visual organization and ease of use are priorities. Less suited for complex enterprise-level projects requiring detailed reporting and strict workflows.
- **Confluence:**
    - **Type:** Collaboration and knowledge management software, often used with Jira.
    - **Features:**
        - **Wikis:** Create and organize documentation, meeting notes, project requirements, knowledge bases.
        - **Templates:** Predefined templates for various documents (e.g., meeting notes, retrospectives, technical specs).
        - **Integrations:** Deep integration with Jira for linking documentation to issues.
        - **Versioning:** Tracks changes to pages.
    - **Use Case:** Ideal for creating and sharing project documentation, team wikis, decision records, and knowledge management within an organization.
- **Asana / Monday.com / ClickUp (and many others):**
    - General-purpose project management and work management tools that can often be configured to support Agile methodologies. They offer varying levels of features, complexity, and integrations, catering to different team sizes and needs.

9.4. Other Methodologies (Briefly Mentioned)

While Agile is dominant, it's good to be aware of others:

- **Waterfall Model:**

    - **Concept:** A traditional, linear, sequential approach where each phase (Requirements, Design, Implementation, Testing, Deployment, Maintenance) must be completed before the next phase begins.
    - **Characteristics:** Heavy documentation, rigid planning, difficult to adapt to changes.
    - **Use Case:** Rarely used for complex software projects today, sometimes for small, well-defined, low-risk projects.
- **DevOps (Practice, not strictly a PM methodology):**
    - **Concept:** A set of practices that combines software development (Dev) and IT operations (Ops) to shorten the systems development life cycle and provide continuous delivery with high software quality.
    - **Emphasis:** Automation, continuous integration, continuous delivery, continuous monitoring, and collaboration between development and operations teams.
    - **Impact on PM:** Fosters a culture of shared responsibility and faster feedback loops.

For a 3-year experienced developer, you should be able to articulate the advantages of Agile, specifically Scrum, and discuss the roles, events, and artifacts. You should also be familiar with the practical application of tools like Jira, Trello, and Confluence, having likely used them in your previous roles. Understanding the "why" behind these methodologies and tools, and how they contribute to successful software delivery, is as important as knowing their features.