

Here's a comprehensive roadmap to prepare for your Java backend role interview, tailored to your experience and skill set.

Road Map

1. Core Java Fundamentals

- **Key Topics:**
 - OOP Principles: Encapsulation, Polymorphism, Inheritance, and Abstraction.
 - Collections Framework: Lists, Sets, Maps, and their implementations.
 - Multithreading and Concurrency: Thread, Runnable, ExecutorService, synchronization, and locks.
 - Exception Handling: Checked vs Unchecked exceptions, custom exceptions.
 - Java 8+ Features: Streams, Lambda Expressions, Functional Interfaces, Optional, Date and Time API.
- **Preparation Resources:**
 - Books: *Effective Java* by Joshua Bloch, *Java: The Complete Reference* by Herbert Schildt.
 - Practice: Solve problems on platforms like HackerRank or LeetCode.

2. Spring Ecosystem

- **Spring Boot:**
 - Understand application structure and annotations (@RestController, @Service, @Repository, @Configuration).
 - Learn to create REST APIs with Spring Boot.
 - Actuator, DevTools, and logging.
- **Spring Batch:**
 - Batch processing basics.
 - Readers, Writers, Processors, and Chunk-oriented processing.
 - Scheduling jobs and handling retries.
- **Spring Security:**
 - Basics of authentication and authorization.
 - JWT and OAuth2.
- **Spring Data:**
 - Integration with SQL databases using JPA/Hibernate.
 - Repository interfaces and custom queries.
 - MongoDB integration using Spring Data MongoDB.

3. Databases

- **SQL:**
 - Master advanced SQL queries: Joins, Subqueries, CTEs, and Window Functions.
 - Indexing, Transactions, ACID properties, and performance tuning.
 - Write and optimize stored procedures and triggers.

- **MongoDB:**
 - Understand collections, documents, and schema design.
 - Aggregation framework and indexes.
 - Integration with Java applications.

4. System Design

- **Key Concepts:**
 - Design RESTful APIs: Resource modeling, versioning, and documentation (Swagger/OpenAPI).
 - Database design: Normalization, indexing, and sharding.
 - Scalability: Load balancing, caching (Redis), and asynchronous processing (Kafka/RabbitMQ).
- **Preparation Resources:**
 - Books: *Designing Data-Intensive Applications* by Martin Kleppmann.
 - Practice: Mock system design interviews for services like URL shortener, e-commerce platforms, etc.

5. Linux

- **Key Topics:**
 - Basic commands: File management, process management, networking (`ssh`, `scp`, etc.).
 - Shell scripting: Automating tasks and scheduling jobs with `cron`.
 - Debugging: Logs, `strace`, `top`, `ps`.
- **Practice:**
 - Manage services and deployments in a Linux environment.

6. Microservices

- **Topics to Cover:**
 - Principles of microservices architecture.
 - Communication: REST, gRPC, event-driven (Kafka/RabbitMQ).
 - Service discovery and configuration management (Eureka, Consul, Spring Cloud Config).
 - Monitoring and logging: ELK stack, Prometheus, and Grafana.

7. Tools & Frameworks

- **Build Tools:** Maven, Gradle.
- **Version Control:** Git (rebasing, merging, branching strategies).
- **CI/CD:** Jenkins, GitLab CI/CD, or GitHub Actions.
- **Containerization:** Docker basics, creating Dockerfiles, and managing containers.
- **Orchestration:** Kubernetes basics (optional but valuable).

8. Problem Solving

- Regularly practice coding problems on:
 - **Data Structures:** Arrays, LinkedLists, Stacks, Queues, HashMaps, Trees, and Graphs.
 - **Algorithms:** Sorting, searching, recursion, dynamic programming.

9. Perl Scripting

- Review and practice:
 - File handling, regular expressions, and text processing.
 - Writing scripts for automation and integration.

10. Mock Interviews

- Conduct mock interviews with peers or on platforms like Pramp or Interviewing.io.
- Focus on:
 - Behavioral questions.
 - Coding rounds (problem-solving and debugging).
 - System design and architecture.

11. Resources and Platforms

- **Courses:**
- **Books:**
- **Online Practice:**
 - LeetCode, Codeforces, HackerRank, GeeksforGeeks.

Daily Preparation Plan

Stay consistent, and adjust the roadmap based on your progress. Let me know if you'd like help diving into specific areas!

Java Breakdown:

Here's a detailed breakdown of **Core Java** topics to focus on for a solid understanding, starting from the basics and moving to advanced concepts:

1. Basics of Java

- Java Features: Platform independence, JVM, JRE, JDK.
- Data Types, Variables, and Constants.

- Operators: Arithmetic, Relational, Logical, Bitwise, Assignment, Ternary.
- Input and Output: `Scanner`, `System.in`, `System.out`, `System.err`.

2. Control Flow Statements

- **Conditional Statements:**
 - `if`, `if-else`, `if-else if`, `switch`.
- **Loops:**
 - `for`, `while`, `do-while`.
 - Enhanced for-loop (`for-each`).
- **Break and Continue:**
 - Handling control flow within loops.

3. Object-Oriented Programming (OOP)

- **Core Principles:**
 - Encapsulation.
 - Inheritance (`extends`, `super`, `final`).
 - Polymorphism (Method Overloading and Method Overriding).
 - Abstraction (`abstract` classes and interfaces).
- **Important Concepts:**
 - Constructors: Default, parameterized, copy constructors.
 - Access Modifiers: `public`, `private`, `protected`, `default`.
 - Static Members and Methods.
 - `this` and `super` keywords.

4. Strings and String Manipulation

- String Class and Methods: `length()`, `charAt()`, `substring()`, `indexOf()`, `replace()`, etc.
- `StringBuilder` and `StringBuffer`: Mutable strings.
- String Pool and Immutability.

5. Arrays

- Single and Multidimensional Arrays.
- Array Manipulation: Sorting, Searching, and Copying.
- Working with `java.util.Arrays`.

6. Collections Framework

- **Core Interfaces:**
 - **List:** `ArrayList`, `LinkedList`, `Vector`.
 - **Set:** `HashSet`, `TreeSet`, `LinkedHashSet`.
 - **Map:** `HashMap`, `TreeMap`, `LinkedHashMap`.
 - **Queue:** `PriorityQueue`, `Deque`.
 - **Sorting and Searching:**
 - `Comparable` and `Comparator`.
 - **Advanced:**
 - Fail-Fast vs Fail-Safe Iterators.
 - Hashing and `HashMap` Internals.
-

7. Exception Handling

- Types: Checked, Unchecked, and Errors.
 - Try-Catch-Finally Blocks.
 - Multiple Catch Blocks.
 - Throw and Throws.
 - Custom Exceptions.
-

8. Multithreading and Concurrency

- **Basics:**
 - Threads and the `Runnable` interface.
 - Thread Lifecycle.
- **Concurrency Utilities:**
 - Executors, Callable, Future.
 - Synchronization and Locks.
 - `volatile` and `ThreadLocal`.
- **Advanced:**
 - Deadlocks, Livelocks, Starvation.
 - `ForkJoinPool` and Parallel Streams.

9. Java 8 and Beyond

- **Functional Programming:**
 - Lambda Expressions.
 - Functional Interfaces (`Predicate`, `Function`, `Consumer`).
- **Streams API:**
 - Intermediate Operations: `filter()`, `map()`, `sorted()`.
 - Terminal Operations: `collect()`, `forEach()`, `reduce()`.
- **Optional Class:**

- Avoiding `NullPointerException`.
- New Date and Time API (`java.time`).

10. File I/O (Input/Output)

- Working with Files: `File` class, `FileWriter`, `FileReader`.
- Streams:
 - Byte Streams: `InputStream`, `OutputStream`.
 - Character Streams: `Reader`, `Writer`.
- Serialization and Deserialization.

11. Generics

- Type Parameters and Bounded Types.
- Wildcards: `?`, `? extends`, `? super`.
- Generic Classes and Methods.

12. Annotations

- Built-in Annotations: `@Override`, `@Deprecated`, `@SuppressWarnings`.
- Custom Annotations.

13. Design Patterns in Core Java

- Singleton.
- Factory.
- Builder.
- Prototype.
- Observer.

14. Memory Management

- Garbage Collection Basics.
- Types of References: Strong, Weak, Soft, Phantom.
- JVM Memory Model: Heap, Stack, Method Area.

15. Reflection and Dynamic Class Loading

- Reflection API: `Class`, `Method`, `Field`.
- Accessing private fields and methods.

16. Networking

- Sockets: TCP and UDP communication.
- URL and HTTP communication.
- Working with `java.net` package.

17. Java Modules

- Modular Programming with `module-info.java`.
- JAR and WAR files.

18. Security

- Basics of Java Security APIs.
- Encryption and Decryption using `java.security`.

19. Java Profiling and Debugging

- Using Debugging Tools: JConsole, VisualVM.
- Profiling Applications.

20. Advanced Topics

- **JVM Internals:**
 - Class Loading, Bytecode.
 - Just-In-Time (JIT) Compilation.
- **Performance Tuning:**
 - Optimize code with tools like JProfiler or YourKit.
- **Custom Classloaders:**
 - Extending `ClassLoader`.

This list is structured to go from beginner-friendly to more advanced topics. Let me know where you'd like to dive in first, and I can guide you with detailed explanations or code examples!

Topic: Basics of Java

Explanation

Java is a popular programming language known for its platform independence and object-oriented nature. Here's an explanation of its fundamental concepts:

1. Features of Java

- **Platform Independent:** Write Once, Run Anywhere (WORA) due to the Java Virtual Machine (JVM).
- **Object-Oriented:** Based on OOP principles (encapsulation, inheritance, etc.).
- **Robust:** Provides strong memory management and exception handling.
- **Secure:** Built-in security features like bytecode verification and access control.
- **Multithreaded:** Support for concurrent execution of multiple threads.
- **Portable:** Independent of hardware and operating system.

2. Java Components

- **JVM (Java Virtual Machine):**
 - Executes the compiled bytecode.
 - Provides platform independence.
- **JRE (Java Runtime Environment):**
 - Includes JVM and libraries to run Java programs.
- **JDK (Java Development Kit):**
 - Includes JRE and development tools (e.g., `javac`, `javadoc`).

3. Data Types

- **Primitive Data Types:**
 - `byte`, `short`, `int`, `long`: Numeric types.
 - `float`, `double`: Decimal types.
 - `char`: Single character.
 - `boolean`: True/False.
- **Non-Primitive Data Types:**
 - Strings, arrays, objects, etc.

4. Variables

- Types of variables:
 - **Instance Variables:** Belong to an object (non-static).
 - **Static Variables:** Shared across all objects of a class.
 - **Local Variables:** Declared inside a method or block.

5. Operators

- Arithmetic (+, -, *, /, %).
- Relational (==, !=, >, <, >=, <=).
- Logical (&&, ||, !).
- Assignment (=, +=, -=, etc.).
- Bitwise (&, |, ^, ~, >>, <<).

6. Input and Output

- **Input:**
 - Use `Scanner` for console input:
 - `Scanner sc = new Scanner(System.in);`
 - `int age = sc.nextInt();`
- **Output:**
 - Use `System.out.println()` for console output:
 - `System.out.println("Hello, World!");`

Q&A

1. What is the difference between JDK, JRE, and JVM?

Answer:

- **JVM (Java Virtual Machine):**
 - Responsible for executing Java bytecode.
 - Provides platform independence.
- **JRE (Java Runtime Environment):**
 - Includes JVM and libraries required to run Java applications.
 - Does not include development tools.
- **JDK (Java Development Kit):**
 - Includes JRE and tools for Java development (e.g., compiler, debugger).

2. Why is Java platform-independent?

Answer: Java achieves platform independence through the use of the JVM. Java programs are compiled into bytecode (via `javac`), which is platform-agnostic. The JVM on each platform interprets the bytecode and executes it, making the program platform-independent.

3. What are the main features of Java?

Answer:

- Platform-independent
- Object-oriented
- Secure
- Robust
- Multithreaded
- High performance (due to JIT compiler)
- Portable

4. What are the types of variables in Java?

Answer:

- **Instance Variables:** Declared within a class but outside methods. Each object gets its own copy.
- **Static Variables:** Declared with the `static` keyword. Shared among all objects of the class.
- **Local Variables:** Declared inside methods or blocks. Only accessible within that scope.
- **Final Variables:** Declared with the `final` keyword, making them constants.

5. What are the 8 primitive data types in Java?

Answer:

1. `byte`: 1 byte (8 bits), range: -128 to 127.
2. `short`: 2 bytes (16 bits), range: -32,768 to 32,767.
3. `int`: 4 bytes (32 bits), range: -2^{31} to $2^{31}-1$.
4. `long`: 8 bytes (64 bits), range: -2^{63} to $2^{63}-1$.
5. `float`: 4 bytes (32 bits), for single-precision floating-point numbers.
6. `double`: 8 bytes (64 bits), for double-precision floating-point numbers.
7. `char`: 2 bytes (16 bits), for Unicode characters.
8. `boolean`: 1 bit, for `true` or `false`.

6. Explain the difference between `==` and `.equals()` in Java.

Answer:

- `==`: Compares memory addresses (reference comparison) for objects and values for primitives.
- `.equals()`: Compares the actual content of the objects (defined in the `Object` class, and can be overridden).

7. What is the default value of a variable in Java?

Answer:

- **Instance variables:**
 - Numeric types: 0
 - boolean: false
 - char: '\u0000'. (null)
 - Object references: null
- **Local variables:** No default value (must be initialized before use).

8. How is memory managed in Java?

Answer: Java uses **Garbage Collection** to manage memory. The JVM automatically deallocates memory for objects that are no longer reachable. Developers don't need to manually manage memory as in languages like C++.

9. What is the significance of the `main` method in Java?

Answer: The `main` method is the entry point of any Java program. It is defined as:

```
public static void main(String[] args)
```

- `public`: Accessible everywhere.
- `static`: Can be called without creating an object.
- `void`: No return value.
- `String[] args`: Command-line arguments.

10. How does Java differ from C++?

Answer:

Feature	Java	C++
Platform	Platform-independent (via JVM)	Platform-dependent
Memory	Automatic garbage collection	Manual memory management
Pointers	No direct pointer access	Supports pointers
OOP	Fully object-oriented	Hybrid (supports procedural)
Multiple Inheritance	Achieved using interfaces	Supported with classes

11. What is the purpose of the `final` keyword in Java?

Answer:

Final Variable: Makes the variable constant (cannot be changed after initialization).

- **Final Method:** Prevents method overriding.
- **Final Class:** Prevents inheritance.

Example:

```
final int MAX = 100;
```

12. What are the different types of operators in Java?

Answer:

1. **Arithmetic:** +, -, *, /, %.
2. **Relational:** ==, !=, >, <, >=, <=.
3. **Logical:** &&, ||, !.
4. **Bitwise:** &, |, ^, ~, <<, >>.
5. **Assignment:** =, +=, -=, etc.
6. **Ternary:** condition ? value1 : value2.

13. Explain the difference between String, StringBuilder, and StringBuffer.

Answer:

- **String:** Immutable; cannot be modified after creation.
- **StringBuilder:** Mutable; not thread-safe but faster.
- **StringBuffer:** Mutable; thread-safe but slower.

Example:

```
StringBuilder sb = new StringBuilder("Hello");  
sb.append(" World"); // Modifies the existing object.
```

14. What is type casting in Java?

Answer: Converting one data type to another:

- **Implicit Casting (Widening):**
 - Automatic conversion, e.g., int to long.
 - int intValue = 100; // An integer value
 - long longValue = intValue; // Implicitly cast to long
- **Explicit Casting (Narrowing):**
 - Requires manual casting, e.g., double to int:
 - int num = (int) 5.67; // Results in 5

15. What are wrapper classes in Java?

Answer: Wrapper classes provide an object representation for primitive data types.

Examples:

- `int → Integer`
- `float → Float`
- `boolean → Boolean`

They also support features like autoboxing and unboxing:

```
Integer num = 10;    // Autoboxing
int val = num;       // Unboxing
```

Control Flow Statements.

Topic: Control Flow Statements

Control flow statements are used to dictate the order in which statements in a program are executed. They include decision-making, looping, and branching mechanisms.

1. Decision-Making Statements

- **if, if-else, if-else if:**
 - Used for conditional execution of code.
 - **Syntax:**
 - `if (condition) {`
 - `// Executes if condition is true`
 - `} else if (anotherCondition) {`
 - `// Executes if anotherCondition is true`
 - `} else {`
 - `// Executes if none of the above conditions are true`
 - `}`
- **switch Statement:**
 - Used for multi-way branching based on the value of a variable or expression.
 - **Syntax:**
 - `switch (expression) {`
 - `case value1:`
 - `// Code block`
 - `break;`
 - `case value2:`
 - `// Code block`
 - `break;`
 - `default:`
 - `// Default block`
 - `}`

2. Looping Statements

- **for Loop:**
 - Used when the number of iterations is known.
 - Syntax:

```
for (initialization; condition; update) {  
    // Loop body  
}
```
- **while Loop:**
 - Used when the condition needs to be checked before each iteration.
 - Syntax:

```
while (condition) {  
    // Loop body  
}
```
- **do-while Loop:**
 - Executes the loop body at least once before checking the condition.
 - Syntax:

```
do {  
    // Loop body  
} while (condition);
```
- **Enhanced for Loop:**
 - Used for iterating through collections or arrays.
 - Syntax:

```
for (dataType element : collection) {  
    // Loop body  
}
```

3. Branching Statements

- **break:**
 - Terminates the nearest enclosing loop or switch.
 - Example:

```
for (int i = 0; i < 5; i++) {  
    if (i == 3) break;  
    System.out.println(i);  
}
```
- **continue:**
 - Skips the current iteration of the nearest enclosing loop.
 - Example:

```
for (int i = 0; i < 5; i++) {  
    if (i == 3) continue;  
    System.out.println(i);  
}
```
- **return:**
 - Exits from the current method and optionally returns a value.

Q&A

1. What is the difference between `if` and `switch` statements?

Answer:

- **if:**
 - Used for complex conditional logic.
 - Can evaluate relational and logical expressions.
 - Example:

```
if (x > 10) {  
    System.out.println("Greater than 10");  
}
```
- **switch:**
 - Best suited for multi-way branching based on a single value.
 - Can only evaluate expressions resulting in byte, short, int, char, String, or enum.
 - Example:

```
switch (x) {  
    case 1:  
        System.out.println("Case 1");  
        break;  
    default:  
        System.out.println("Default case");  
}
```

2. What is the difference between while and do-while loops?

Answer:

- **while:**
 - Checks the condition **before** executing the loop body.
 - May not execute if the condition is false.
 - Example:

```
int i = 0;  
while (i < 5) {  
    System.out.println(i);  
    i++;  
}
```
- **do-while:**
 - Executes the loop body **at least once**, then checks the condition.
 - Example:

```
int i = 0;  
do {  
    System.out.println(i);  
    i++;  
} while (i < 5);
```

3. Can a switch statement work with string values in Java?

Answer: Yes, starting from Java 7, switch supports String. Example:

```
String day = "Monday";  
switch (day) {  
    case "Monday":  
        System.out.println("Start of the week");  
}
```

```

        break;
    case "Friday":
        System.out.println("End of the week");
        break;
    default:
        System.out.println("Midweek");
}

```

4. How does the `break` statement work in loops and `switch`?

Answer:

- **In Loops:**
 - Terminates the nearest enclosing loop.
 - **Example:**
 - `for (int i = 0; i < 5; i++) {`
 - `if (i == 3) break; // Exits the loop when i == 3`
 - `System.out.println(i);`
 - `}`
- **In `switch`:**
 - Prevents fall-through to the next case.
 - **Example:**
 - `switch (x) {`
 - `case 1:`
 - `System.out.println("Case 1");`
 - `break; // Exits the switch`
 - `case 2:`
 - `System.out.println("Case 2");`
 - `}`

5. What is the use of the `continue` statement?

Answer: The `continue` statement skips the current iteration of the loop and proceeds to the next iteration.

Example:

```

for (int i = 0; i < 5; i++) {
    if (i == 3) continue; // Skips when i == 3
    System.out.println(i);
}

```

Output:

```

0
1
2
4

```

6. What are the valid data types for a `switch` expression?

Answer:

- byte
- short
- int
- char
- String (Java 7+)
- enum

7. How is the enhanced `for` loop different from a standard `for` loop?

Answer:

- The enhanced `for` loop is used to iterate over collections or arrays without explicitly managing the index.
- Example:

```
int[] numbers = {1, 2, 3, 4};
for (int num : numbers) {
    System.out.println(num);
}
```

8. What happens if you omit the `break` statement in a `switch`?

Answer: If the `break` statement is omitted, execution will fall through to the next case(s) until a `break` is encountered or the `switch` block ends.

Example:

```
int x = 2;
switch (x) {
    case 1:
        System.out.println("Case 1");
    case 2:
        System.out.println("Case 2"); // Executes
    case 3:
        System.out.println("Case 3"); // Falls through
}
```

Output:

```
Case 2
Case 3
```

9. Can a `switch` statement be used with floating-point numbers?

Answer: No, `switch` does not support `float` or `double` values due to precision issues.

10. How do you implement an infinite loop in Java?

Answer: You can create an infinite loop using `while` or `for`:

```
while (true) {  
    // Infinite loop  
}
```

or

```
for (;;) {  
    // Infinite loop  
}
```

11. Can we use `return` inside a loop?

Answer: Yes, the `return` statement can be used to exit a method from within a loop.

Example:

```
for (int i = 0; i < 5; i++) {  
    if (i == 3) return; // Exits the method when i == 3  
}
```

12. Can you nest loops in Java?

Answer: Yes, loops can be nested. Example:

```
for (int i = 1; i <= 3; i++) {  
    for (int j = 1; j <= 2; j++) {  
        System.out.println("i = " + i + ", j = " + j);  
    }  
}
```

Output:

```
i = 1, j = 1  
i = 1, j = 2  
i = 2, j = 1  
i = 2, j = 2  
i = 3, j = 1  
i = 3, j = 2
```

13. How do you exit from a nested loop?

Answer: Use a labeled `break`:

```
outer:  
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 3; j++) {  
        if (i == 1 && j == 1) break outer; // Exits both loops  
    }  
}
```

```
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

Object-Oriented Programming (OOP).

Topic: Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects rather than actions. Java is inherently object-oriented, making OOP principles fundamental to its development.

1. Core Principles of OOP

Encapsulation

- Binding data (fields) and methods (functions) together in a single unit (class).
- Access to data is controlled using access modifiers (`private`, `protected`, `public`).
- Example:
- ```
class Person {
```
- ```
    private String name;
```
- ```
 public String getName() {
```
- ```
        return name;
```
- ```
 }
```
- ```
    public void setName(String name) {
```
- ```
 this.name = name;
```
- ```
    }
```
- ```
}
```

### *Inheritance*

- Enables one class to inherit the fields and methods of another.
- Promotes code reusability.
- Syntax:
- ```
class Parent {
```
- ```
 void display() {
```
- ```
        System.out.println("Parent class");
```
- ```
 }
```
- ```
}
```
- ```
class Child extends Parent {
```
- ```
    void show() {
```
- ```
 System.out.println("Child class");
```
- ```
    }
```
- ```
}
```

## Polymorphism

- Allows methods to perform different tasks based on the object.
- **Types:**
  - **Method Overloading** (Compile-Time): Same method name, different parameters.
  - **Method Overriding** (Run-Time): Child class redefines a method in the parent class.
- **Example of Overloading:**
- ```
class Math {  
    int add(int a, int b) {  
        return a + b;  
    }  
  
    double add(double a, double b) {  
        return a + b;  
    }  
}
```
- **Example of Overriding:**
- ```
class Parent {
 void display() {
 System.out.println("Parent class");
 }
}

class Child extends Parent {
 @Override
 void display() {
 System.out.println("Child class");
 }
}
```

## Abstraction

- Hiding implementation details while exposing functionality.
- Achieved using:
  - **Abstract Classes:** Contains both abstract (without body) and concrete methods.
  - ```
abstract class Shape {  
    abstract void draw();  
}
```
 - ```
class Circle extends Shape {
 void draw() {
 System.out.println("Drawing Circle");
 }
}
```
  - **Interfaces:** Contains only abstract methods (Java 8+ can include default and static methods).
  - ```
interface Drawable {  
    void draw();  
}  
  
class Square implements Drawable {  
    public void draw() {
```

```

        ○         System.out.println("Drawing Square");
        ○     }
        ○ }

```

2. Other Important OOP Concepts

Classes and Objects

- **Class:** Blueprint for objects.
- **Object:** Instance of a class.

Example:

```

class Car {
    String brand;
    int speed;

    void drive() {
        System.out.println(brand + " is driving at " + speed + " km/h");
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.brand = "Toyota";
        car.speed = 120;
        car.drive();
    }
}

```

Access Modifiers

- Control visibility of class members.
- Types:
 - private: Accessible only within the class.
 - protected: Accessible within the package and subclasses.
 - public: Accessible everywhere.
 - Default (no modifier): Accessible within the package.

Static Members

- Belong to the class rather than instances.
- Example:


```

class Math {
    static int square(int num) {
        return num * num;
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(Math.square(5));
    }
}

```

- }

Final Keyword

- **Final Variable:** Constant value.
- **Final Method:** Cannot be overridden.
- **Final Class:** Cannot be inherited.

Constructors

- Special methods used to initialize objects.
- Types:
 - Default constructor.
 - Parameterized constructor.

Example:

```
class Student {
    String name;

    Student(String name) { // Constructor
        this.name = name;
    }
}
```

Detailed Explanation of Abstraction

Abstraction in Java is the process of hiding implementation details and showing only the essential features of an object. It focuses on **what** an object does instead of **how** it does it.

Abstraction is achieved through:

1. **Abstract Classes:** Classes that cannot be instantiated and may have abstract methods (methods without implementation).
2. **Interfaces:** Purely abstract types that define contracts (method signatures) without implementation.

1. Abstract Classes

- **Definition:**
 - Declared with the `abstract` keyword.
 - May contain both abstract (unimplemented) and concrete (implemented) methods.
 - Useful when classes share some functionality but require specific implementations in subclasses.

Example of Abstract Class

```
// Abstract class
```

```

abstract class Animal {
    // Abstract method (no body)
    abstract void makeSound();

    // Concrete method
    void sleep() {
        System.out.println("Sleeping...");
    }
}

// Subclass providing implementation for the abstract method
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Bark!");
    }
}

// Subclass providing its own implementation
class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Meow!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.makeSound(); // Output: Bark!
        dog.sleep();      // Output: Sleeping...

        Animal cat = new Cat();
        cat.makeSound(); // Output: Meow!
        cat.sleep();      // Output: Sleeping...
    }
}

```

Key Points for Abstract Classes:

1. Abstract classes can have instance variables and constructors.
2. If a class has at least one abstract method, it must be declared abstract.
3. Subclasses must provide implementations for all abstract methods unless they are also abstract.

2. Interfaces

- **Definition:**
 - A blueprint for a class, specifying method signatures that must be implemented.
 - Interfaces cannot have instance variables or constructors.
 - From Java 8 onwards:
 - Interfaces can have **default methods** (methods with a body).
 - Interfaces can have **static methods**.

Example of an Interface

```
// Interface definition
interface Vehicle {
    // Abstract method
    void start();

    // Default method
    default void stop() {
        System.out.println("Vehicle stopped.");
    }
}

// Class implementing the interface
class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car started.");
    }
}

// Another class implementing the same interface
class Bike implements Vehicle {
    @Override
    public void start() {
        System.out.println("Bike started.");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.start(); // Output: Car started.
        car.stop();  // Output: Vehicle stopped.

        Vehicle bike = new Bike();
        bike.start(); // Output: Bike started.
        bike.stop();  // Output: Vehicle stopped.
    }
}
```

Key Points for Interfaces:

1. Methods in an interface are **implicitly public and abstract** (until Java 8).
2. Fields in an interface are **implicitly public, static, and final**.
3. A class can implement multiple interfaces (supporting multiple inheritance).

Abstract Class vs Interface

Feature	Abstract Class	Interface
Inheritance	Supports single inheritance.	Supports multiple inheritance.
Implementation	Can have both abstract and concrete methods.	All methods were abstract until Java 8 (can now have default and static methods).
Constructors	Can have constructors.	Cannot have constructors.
Access	Abstract methods can be	All methods are implicitly <code>public</code> .

Modifiers	protected, public, or package-private.	
Fields	Can have instance variables.	Can only have public static final variables.

3. Real-World Example

Imagine a **Payment System**:

- Payment could be made via **CreditCard** or **PayPal**.
- The process of payment differs depending on the payment method, but the essential steps are the same.

Using **Abstraction**:

```
// Abstract class
abstract class Payment {
    abstract void makePayment(); // Abstract method

    // Concrete method
    void paymentDetails() {
        System.out.println("Payment is being processed...");
    }
}

// Subclass implementing the abstract method
class CreditCardPayment extends Payment {
    @Override
    void makePayment() {
        System.out.println("Processing credit card payment...");
    }
}

// Subclass implementing the abstract method
class PayPalPayment extends Payment {
    @Override
    void makePayment() {
        System.out.println("Processing PayPal payment...");
    }
}

public class Main {
    public static void main(String[] args) {
        Payment payment1 = new CreditCardPayment();
        payment1.paymentDetails(); // Output: Payment is being
        processed...
        payment1.makePayment();    // Output: Processing credit card
        payment...

        Payment payment2 = new PayPalPayment();
        payment2.paymentDetails(); // Output: Payment is being
        processed...
        payment2.makePayment();    // Output: Processing PayPal
        payment...
    }
}
```

Use Cases of Abstraction

1. **Code Reusability:** Shared behaviors in abstract classes or interfaces reduce code duplication.
2. **Flexibility:** Multiple classes can implement the same interface differently, supporting polymorphism.
3. **Maintainability:** Changes in abstraction don't affect dependent classes as long as the contract remains the same.

Q&A

1. What is the difference between abstraction and encapsulation?

Answer:

Feature	Abstraction	Encapsulation
Purpose	Hides implementation details; shows functionality.	Hides data by restricting direct access.
Focus	Focuses on <u>what</u> an object does.	Focuses on <u>how</u> data is protected.
Achieved By	Achieved using abstract classes and interfaces.	Achieved using access modifiers (private, public, etc.).
Example	An abstract class or interface defines methods without implementation.	Fields in a class are private, and getters/setters are used.

2. What is polymorphism? Explain with examples.

Answer: Polymorphism allows methods to perform different tasks based on the object. It is of two types:

1. Compile-Time Polymorphism (Method Overloading):

- o Same method name, different parameter lists.

```
2. class Math {  
3.     int add(int a, int b) {  
4.         return a + b;  
5.     }  
6.  
7.     double add(double a, double b) {  
8.         return a + b;  
9.     }  
10. }
```

11. Run-Time Polymorphism (Method Overriding):

- o A subclass provides a specific implementation of a method from the superclass.

```
12. class Parent {  
13.     void show() {
```

```

14.         System.out.println("Parent class");
15.     }
16. }
17.
18. class Child extends Parent {
19.     @Override
20.     void show() {
21.         System.out.println("Child class");
22.     }
23. }

```

3. What are access modifiers in Java?

Answer: Access modifiers define the visibility of a class, method, or variable:

1. **Private:** Accessible only within the class.
2. **Default:** Accessible within the same package.
3. **Protected:** Accessible within the same package and by subclasses.
4. **Public:** Accessible everywhere.

4. What is the difference between abstract class and interface?

Answer:

Feature	Abstract Class	Interface
Inheritance	Single inheritance.	Multiple inheritance.
Methods	Can have abstract and concrete methods.	Abstract methods (until Java 8), default, and static methods (Java 8+).
Constructors	Can have constructors.	Cannot have constructors.
Fields	Can have instance variables.	Only public static final variables.

5. Can we create an object of an abstract class?

Answer: No, you cannot instantiate an abstract class. However, you can create a reference to an abstract class and instantiate its subclass:

```

abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle");
    }
}

Shape shape = new Circle(); // Allowed

```

6. What is the difference between method overloading and method overriding?

Answer:

Feature	Method Overloading	Method Overriding
Definition	Same method name, different parameter list.	Subclass redefines a method from the superclass.
Compile/Run Time	Resolved at compile-time.	Resolved at runtime.
Inheritance	Not required.	Requires inheritance.
Return Type	Can vary.	Must be the same or covariant.

7. What is the purpose of the `final` keyword in Java?

Answer:

- **Final Variable:** Prevents the variable from being reassigned.
`final int MAX = 100;`
- **Final Method:** Prevents method overriding.
`final void display() {
 System.out.println("Cannot override me!");
}`
- **Final Class:** Prevents inheritance.
`final class Animal {}`

8. Can an abstract class have a constructor?

Answer: Yes, abstract classes can have constructors. These constructors are called during the instantiation of subclasses.

Example:

```
abstract class Shape {  
    Shape() {  
        System.out.println("Shape constructor called");  
    }  
}  
  
class Circle extends Shape {  
    Circle() {  
        System.out.println("Circle constructor called");  
    }  
}  
  
Circle c = new Circle();  
// Output:  
// Shape constructor called
```

```
// Circle constructor called
```

9. What are **this** and **super** keywords in Java?

Answer:

- **this:**
 - Refers to the current object.
 - Used to resolve ambiguity between instance variables and parameters.
 - **Example:**
 - ```
class Demo {
 int x;
 Demo(int x) {
 this.x = x;
 }
}
```
- **super:**
  - Refers to the parent class's methods or variables.
  - Used to call the parent class's constructor.
  - **Example:**
  - ```
class Parent {  
    void display() {  
        System.out.println("Parent class");  
    }  
}  
  
class Child extends Parent {  
    void show() {  
        super.display();  
    }  
}
```

10. What is an interface in Java?

Answer: An interface is a blueprint for a class that defines method signatures but not implementations. Classes implementing the interface must provide implementations for its methods.

Example:

```
interface Animal {  
    void sound();  
}  
  
class Dog implements Animal {  
    @Override  
    public void sound() {  
        System.out.println("Bark");  
    }  
}
```

11. Can we implement multiple interfaces in a single class?

Answer: Yes, a class can implement multiple interfaces:

```
interface A {
    void methodA();
}

interface B {
    void methodB();
}

class C implements A, B {
    @Override
    public void methodA() {
        System.out.println("Method A");
    }

    @Override
    public void methodB() {
        System.out.println("Method B");
    }
}
```

12. What is the purpose of the `static` keyword?

Answer:

- **Static Variable:** Shared across all instances of a class.
- **Static Method:** Can be called without creating an object of the class.
- **Static Block:** Used for static initialization.

Example:

```
class Demo {
    static int count = 0;
    static void displayCount() {
        System.out.println("Count: " + count);
    }

    static {
        count = 10; // Static initialization
    }
}
```

13. What is the difference between `is-a` and `has-a` relationships in OOP?

Answer:

- **Is-A Relationship:** Represents inheritance.
 - Example: A Dog **is-a** Animal.
- **Has-A Relationship:** Represents composition or aggregation.
 - Example: A Car **has-a** Engine.

In essence:

- **Is-a** focuses on the type or kind of an object.
- **Has-a** focuses on the object's composition or the objects it contains.

Key Points:

- **Inheritance** is about specialization and creating a hierarchy of classes.
- **Composition and Aggregation** are about creating objects from other objects, allowing for code reusability and modularity.
- `// Is-a relationship`
- `class Animal {`
- `void eat() {`
- `System.out.println("Eating");`
- `}`
- `}`
- `class Dog extends Animal {`
- `void bark() {`
- `System.out.println("Woof!");`
- `}`
- `}`
- `// Has-a relationship (Composition)`
- `class Car {`
- `private Engine engine;`
- `Car(Engine engine) {`
- `this.engine = engine;`
- `}`
- `}`
- `class Engine {`
- `void start() {`
- `System.out.println("Engine started");`
- `}`
- `}`
- in this example, `Dog` inherits from `Animal`, demonstrating the "is-a" relationship. The `Car` class has an `Engine` as a member, representing the "has-a" relationship.

14. Can a constructor be overridden?

Answer: No, constructors cannot be overridden because they are not inherited by subclasses.

15. What is a covariant return type?

Answer: When a method is overridden, the return type in the subclass can be a subtype of the return type in the superclass.

Example:

```
class Parent {
    Parent get() {
        return this;
    }
}

class Child extends Parent {
    @Override
    Child get() {
        return this;
    }
}
```

Strings and String Manipulation.

Topic: Strings and String Manipulation

Strings in Java are objects that represent sequences of characters. Java provides the `String` class in the `java.lang` package for working with strings.

1. Key Features of Strings in Java

- 1. **Immutable:** Once a `String` object is created, it cannot be changed. Operations like concatenation or substring create new objects.
- 2. **String Pool:** Strings are stored in a special memory area called the string pool, enabling reusability and saving memory.
- 3. **Thread-Safe:** Since strings are immutable, they are inherently thread-safe.

2. Commonly Used String Methods

Method	Description	Example
<code>length()</code>	Returns the length of the string.	<code>"hello".length()</code> → 5
<code>charAt(int index)</code>	Returns the character at the specified index.	<code>"hello".charAt(1)</code> → 'e'

<code>substring(int beginIndex)</code>	Returns a substring starting from the specified index.	<code>"hello".substring(2) → "llo"</code>
<code>substring(int beginIndex, int endIndex)</code>	Returns a substring within the range.	<code>"hello".substring(1, 4) → "ell"</code>
<code>equals(Object obj)</code>	Compares the content of two strings.	<code>"hello".equals("world") → false</code>
<code>equalsIgnoreCase(String)</code>	Compares two strings, ignoring case differences.	<code>"HELLO".equalsIgnoreCase("hello") → true</code>
<code>toUpperCase()</code>	Converts the string to uppercase.	<code>"hello".toUpperCase() → "HELLO"</code>
<code>toLowerCase()</code>	Converts the string to lowercase.	<code>"HELLO".toLowerCase() → "hello"</code>
<code>replace(char oldChar, char newChar)</code>	Replaces all occurrences of a character.	<code>"hello".replace('l', 'p') → "heppo"</code>
<code>trim()</code>	Removes leading and trailing spaces.	<code>" hello ".trim() → "hello"</code>
<code>split(String regex)</code>	Splits the string into an array based on the delimiter.	<code>"a,b,c".split(",") → ["a", "b", "c"]</code>
<code>indexOf(String str)</code>	Returns the index of the first occurrence of the substring.	<code>"hello".indexOf("l") → 2</code>
<code>lastIndexOf(String str)</code>	Returns the index of the last occurrence of the substring.	<code>"hello".lastIndexOf("l") → 3</code>
<code>contains(CharSequence s)</code>	Checks if the string contains the specified sequence.	<code>"hello".contains("ll") → true</code>
<code>isEmpty()</code>	Checks if the	<code>"".isEmpty() → true</code>

	string is empty.	
--	------------------	--

3. StringBuilder and StringBuffer

Since `String` is immutable, Java provides `StringBuilder` and `StringBuffer` for mutable strings.

StringBuilder

- Faster and non-thread-safe.
- Useful for single-threaded applications.

StringBuffer

- Thread-safe (synchronized).
- Useful in multi-threaded applications.

Example of StringBuilder

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb); // Output: Hello World
```

Example of StringBuffer

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb); // Output: Hello World
```

4. String vs StringBuilder vs StringBuffer

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread Safety	Thread-safe due to immutability	Not thread-safe	Thread-safe
Performance	Slower for frequent changes	Faster than String	Slower than StringBuilder

5. String Pool

When a string literal is created, it is stored in the string pool. If another string with the same value is created, it refers to the same object in the pool.

Example

```
String s1 = "hello";
String s2 = "hello";
System.out.println(s1 == s2); // true (same reference in the pool)
```

6. Important Concepts

1. Immutable Nature:

- Any modification to a `String` creates a new object.
- Example:
 - `String str = "Hello";`
 - `str.concat(" World");`
 - `System.out.println(str);` // Output: Hello (unchanged)

2. String Comparison:

- Use `equals()` for content comparison.
- Use `==` for reference comparison.

Q&A

1. What is the difference between `String`, `StringBuilder`, and `StringBuffer`?

Answer:

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread Safety	Thread-safe due to immutability	Not thread-safe	Thread-safe (synchronized)
Performance	Slower for frequent changes	Faster than String	Slower than StringBuilder

2. Why are strings immutable in Java?

Answer:

- **Security:** Immutable strings prevent data tampering, especially when used for sensitive information (e.g., file paths, URLs).
- **Thread Safety:** Immutable strings can be safely shared across threads without synchronization.
- **Performance:** String literals are stored in a string pool, enabling reusability and saving memory.

3. What is the difference between `equals()` and `==` in strings?

Answer:

- `equals()`: Compares the **content** of the strings.
- `==`: Compares the **memory reference** of the strings.

Example:

```
String s1 = new String("hello");
String s2 = new String("hello");

System.out.println(s1.equals(s2)); // true (same content)
System.out.println(s1 == s2);      // false (different memory
references)
```

4. What does the `intern()` method do?

Answer: The `intern()` method ensures that a string is added to the string pool. If the string already exists in the pool, it returns the reference to the pooled string.

Example:

```
String s1 = new String("hello");
String s2 = s1.intern();

String s3 = "hello";
System.out.println(s2 == s3); // true (both refer to the string pool)
```

5. How can you reverse a string in Java?

Answer: You can reverse a string using `StringBuilder`:

```
String str = "hello";
String reversed = new StringBuilder(str).reverse().toString();
System.out.println(reversed); // Output: olleh
```

6. How do you split a string into an array?

Answer: Use the `split()` method:

```
String str = "a,b,c";
String[] arr = str.split(",");
System.out.println(Arrays.toString(arr)); // Output: [a, b, c]
```

7. How do you check if a string is a palindrome?

Answer: A palindrome is a string that reads the same backward as forward:

```
String str = "madam";
String reversed = new StringBuilder(str).reverse().toString();
if (str.equals(reversed)) {
    System.out.println("Palindrome");
} else {
    System.out.println("Not a Palindrome");
}
```

8. How can you count the occurrences of a character in a string?

Answer: You can count occurrences using a loop or `replace`:

```
String str = "hello";
char target = 'l';
long count = str.chars().filter(ch -> ch == target).count();
System.out.println(count); // Output: 2
```

9. What is the output of the following code?

```
String s1 = "abc";
String s2 = "abc";
System.out.println(s1 == s2);          // ?
System.out.println(s1.equals(s2));     // ?
```

Answer:

- `s1 == s2: true` (Both refer to the same object in the string pool).
- `s1.equals(s2): true` (Contents are the same).

10. How do you convert a `String` to a `char[]`?

Answer: Use the `toCharArray()` method:

```
String str = "hello";
char[] charArray = str.toCharArray();
System.out.println(Arrays.toString(charArray)); // Output: [h, e, l, l, o]
```

11. What is the output of the following code?

```
String s1 = new String("hello");
String s2 = "hello";
System.out.println(s1 == s2);          // ?
System.out.println(s1.equals(s2));     // ?
```

Answer:

- `s1 == s2: false` (Different memory references).
- `s1.equals(s2): true` (Contents are the same).

12. How do you compare two strings ignoring case?

Answer: Use `equalsIgnoreCase()`:

```
String s1 = "HELLO";
String s2 = "hello";
System.out.println(s1.equalsIgnoreCase(s2)); // true
```

13. How do you check if a string contains a specific substring?

Answer: Use the `contains()` method:

```
String str = "hello world";
System.out.println(str.contains("world")); // true
```

14. How do you remove whitespace from the start and end of a string?

Answer: Use the `trim()` method:

```
String str = " hello ";
System.out.println(str.trim()); // Output: hello
```

15. What is the difference between `String` and `CharSequence`?

Answer:

- `String` is a concrete class implementing the `CharSequence` interface.
 - `CharSequence` is an interface that defines a readable sequence of characters. Other classes like `StringBuilder` and `StringBuffer` also implement it.
-

Arrays.

Topic: Arrays

An **array** is a data structure that stores a fixed-size sequential collection of elements of the same type. In Java, arrays are objects, and they can be single-dimensional or multi-dimensional.

1. Key Characteristics of Arrays

1. **Fixed Size:** Once declared, the size of an array cannot be changed.
2. **Index-Based:** Array elements are accessed using zero-based indexing.
3. **Homogeneous:** All elements in an array must be of the same data type.
4. **Default Values:** When an array is declared, its elements are initialized to default values:
 - Numeric types: 0
 - boolean: false

- Objects: null

2. Declaration and Initialization

- **Declaration:**
- `int[] arr;` // Preferred syntax
- `int arr[];` // Also valid
- **Initialization:**
- `int[] arr = new int[5];` // Creates an array of size 5
- `int[] arr = {1, 2, 3, 4, 5};` // Inline initialization

3. Accessing Elements

- Using an index:
- `int[] arr = {10, 20, 30};`
- `System.out.println(arr[1]);` // Output: 20

4. Iterating Over Arrays

- **For Loop:**
- `for (int i = 0; i < arr.length; i++) {`
- `System.out.println(arr[i]);`
- `}`
- **Enhanced For Loop:**
- `for (int num : arr) {`
- `System.out.println(num);`
- `}`

5. Multi-Dimensional Arrays

- **Declaration:**
- `int[][] matrix = new int[3][3];`
- **Initialization:**
- `int[][] matrix = {`
- `{1, 2, 3},`
- `{4, 5, 6},`
- `{7, 8, 9}`
- `};`
- **Accessing Elements:**
- `System.out.println(matrix[1][2]);` // Output: 6

6. Common Operations on Arrays

- **Finding Length:**

- `int[] arr = {1, 2, 3};`
- `System.out.println(arr.length);` // Output: 3
- **Copying Arrays:**
 - Using `System.arraycopy()`:
 - `int[] src = {1, 2, 3};`
 - `int[] dest = new int[3];`
 - `System.arraycopy(src, 0, dest, 0, src.length);`
 - Using `Arrays.copyOf()`:
 - `int[] copy = Arrays.copyOf(arr, arr.length);`

In Java, `Arrays.sort(arr)` internally uses a combination of algorithms.

The specific algorithm used depends on the type of the array elements:

- **For primitive data types (like `int`, `double`, `char`):** It uses a **dual-pivot quicksort** algorithm. This is a variant of quicksort that generally provides better performance than the traditional single-pivot quicksort.
- **For objects that implement the `Comparable` interface:** It also uses a **dual-pivot quicksort** algorithm. The `Comparable` interface defines the `compareTo()` method, which allows the elements to be compared with each other.
- **For objects that don't implement `Comparable`:** You need to provide a `Comparator` object to the `Arrays.sort()` method. The `Comparator` interface defines the `compare()` method, which specifies how to compare two objects. In this case, the algorithm used is again a **dual-pivot quicksort**.

Key Points:

- The `Arrays.sort()` method is highly optimized for performance.
- The choice of the dual-pivot quicksort algorithm makes it efficient for sorting large arrays.
- The use of the `Comparable` interface or a `Comparator` object provides flexibility in sorting objects based on different criteria.

If you have a very specific sorting requirement or need to optimize for a particular use case, you might consider implementing your own sorting algorithm. However, for most general-purpose sorting tasks, `Arrays.sort()` is a highly efficient and recommended approach.

- `Arrays.sort(arr);`
- **Searching:**
- `int index = Arrays.binarySearch(arr, key);`

7. Array Class in `java.util`

The `java.util.Arrays` class provides utility methods for working with arrays:

- **`Arrays.toString()`**: Converts an array to a string.
`System.out.println(Arrays.toString(arr));` // Output: [1, 2, 3]
- **`Arrays.equals()`**: Compares two arrays for equality.
`System.out.println(Arrays.equals(arr1, arr2));`

8. Jagged Arrays

A jagged array is an array of arrays with different sizes.

```
int[][] jagged = {
    {1, 2},
    {3, 4, 5},
    {6}
};
System.out.println(jagged[1][2]); // Output: 5
```

9. Limitations of Arrays

- Fixed size: Once created, you cannot change the size of an array.
- No direct support for advanced operations like adding or removing elements (use collections like `ArrayList` for such operations).

Q&A

1. What are arrays in Java?

Answer: An array is a collection of elements of the same data type, stored in a contiguous memory location, and accessed using an index.

2. How do you declare and initialize an array in Java?

Answer:

- **Declaration:**
`int[] arr;`
- **Initialization:**
`arr = new int[5];` // Array of size 5
`int[] arr = {1, 2, 3, 4, 5};` // Inline initialization

3. What is the default value of elements in an array?

Answer:

Data Type	Default Value
Numeric types	0
boolean	false
Objects	null

4. How do you find the length of an array in Java?

Answer: Use the `.length` property:

```
int[] arr = {1, 2, 3};  
System.out.println(arr.length); // Output: 3
```

5. How do you iterate over an array in Java?

Answer:

- **Using a for loop:**
• `for (int i = 0; i < arr.length; i++) {`
• `System.out.println(arr[i]);`
• `}`
- **Using an enhanced for loop:**
• `for (int num : arr) {`
• `System.out.println(num);`
• `}`

6. What are multi-dimensional arrays? Provide an example.

Answer: Multi-dimensional arrays are arrays of arrays. A common example is a 2D array:

```
int[][] matrix = {  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9}  
};  
System.out.println(matrix[1][2]); // Output: 6
```

7. What is a jagged array?

Answer: A jagged array is an array of arrays with different lengths. Example:

```
int[][] jagged = {
    {1, 2},
    {3, 4, 5},
    {6}
};
System.out.println(jagged[1][2]); // Output: 5
```

8. How do you copy an array in Java?

Answer:

- **Using `System.arraycopy()`:**
- `int[] src = {1, 2, 3};`
- `int[] dest = new int[3];`
- `System.arraycopy(src, 0, dest, 0, src.length);`
- **Using `Arrays.copyOf()`:**
- `int[] copy = Arrays.copyOf(arr, arr.length);`

9. How do you sort an array in Java?

Answer: Use `Arrays.sort()`:

```
int[] arr = {3, 1, 2};
Arrays.sort(arr);
System.out.println(Arrays.toString(arr)); // Output: [1, 2, 3]
```

10. How do you search for an element in a sorted array?

Answer: Use `Arrays.binarySearch()`:

```
int[] arr = {1, 2, 3, 4, 5};
int index = Arrays.binarySearch(arr, 3);
System.out.println(index); // Output: 2
```

11. What happens if you access an array index out of bounds?

Answer: An `ArrayIndexOutOfBoundsException` is thrown. Example:

```
int[] arr = {1, 2, 3};
System.out.println(arr[3]); // Throws ArrayIndexOutOfBoundsException
```

12. Can you store objects in an array?

Answer: Yes, arrays can store objects:

```
String[] arr = {"Hello", "World"};
System.out.println(arr[0]); // Output: Hello
```

13. How do you convert an array to a string?

Answer: Use `Arrays.toString()`:

```
int[] arr = {1, 2, 3};
System.out.println(Arrays.toString(arr)); // Output: [1, 2, 3]
```

14. How do you reverse an array in Java?

Answer: Manually swap elements:

```
int[] arr = {1, 2, 3, 4};
for (int i = 0; i < arr.length / 2; i++) {
    int temp = arr[i];
    arr[i] = arr[arr.length - 1 - i];
    arr[arr.length - 1 - i] = temp;
}
System.out.println(Arrays.toString(arr)); // Output: [4, 3, 2, 1]
```

15. What are the differences between an array and an ArrayList in Java?

Answer:

Feature	Array	ArrayList
Size	Fixed size	Dynamic size
Type	Can store primitive types	Stores only objects (autoboxing used for primitives)
Performance	Faster	Slower due to overhead
Methods	No utility methods	Provides utility methods like <code>add()</code> , <code>remove()</code>

16. Can arrays be resized in Java?

Answer: No, arrays cannot be resized after creation. To achieve dynamic resizing, use `ArrayList`.

17. What is the difference between a shallow copy and a deep copy of an array?

Answer:

- **Shallow Copy:** Copies references, not the actual objects. Changes in the copied array affect the original.

- **Deep Copy:** Copies the actual objects. Changes in the copied array do not affect the original.

Example:

```
int[][] shallow = {{1, 2}, {3, 4}};
int[][] deep = new int[shallow.length][];
for (int i = 0; i < shallow.length; i++) {
    deep[i] = Arrays.copyOf(shallow[i], shallow[i].length);
}
```

18. What is the difference between single-dimensional and multi-dimensional arrays?

Answer:

- **Single-Dimensional:** Stores elements in a linear format.
`int[] arr = {1, 2, 3};`
 - **Multi-Dimensional:** Stores elements in a grid-like format.
`int[][] matrix = {`
 - `{1, 2},`
 - `{3, 4}`
 - `};`
-

Collections Framework.

Topic: Collections Framework

The **Collections Framework** in Java is a set of classes and interfaces that provide data structures (like lists, sets, and maps) and algorithms for managing and manipulating collections of objects efficiently.

1. Core Interfaces in the Collections Framework

Interface	Description	Common Implementations
List	Ordered collection (allows duplicates).	<code>ArrayList</code> , <code>LinkedList</code> , <code>Vector</code> , <code>Stack</code>
Set	Unordered collection (no duplicates).	<code>HashSet</code> , <code>LinkedHashSet</code> , <code>TreeSet</code>
Map	Key-value pairs (keys must be unique).	<code>HashMap</code> , <code>TreeMap</code> , <code>LinkedHashMap</code> , <code>Hashtable</code>
Queue	FIFO (First-In-First-Out)	<code>PriorityQueue</code> , <code>Deque</code> , <code>LinkedList</code>

	collection.	
--	-------------	--

2. Commonly Used Classes

1. List Implementations

- **ArrayList:**
 - Dynamic array.
 - Faster access (indexed).
 - Slower insertion/removal (shifting elements).
 - Example:


```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
System.out.println(list); // Output: [1, 2]
```
- **LinkedList:**
 - Doubly linked list.
 - Faster insertion/removal (no shifting).
 - Slower access (traversal required).
 - Example:


```
List<Integer> list = new LinkedList<>();
list.add(1);
list.add(2);
System.out.println(list); // Output: [1, 2]
```

2. Set Implementations

- **HashSet:**
 - No duplicates.
 - Unordered.
 - Example:


```
Set<Integer> set = new HashSet<>();
set.add(1);
set.add(2);
set.add(1); // Duplicate, ignored
System.out.println(set); // Output: [1, 2]
```
- **TreeSet:**
 - Sorted set.
 - Example:


```
Set<Integer> set = new TreeSet<>();
set.add(2);
set.add(1);
System.out.println(set); // Output: [1, 2]
```

3. Map Implementations

- **HashMap:**
 - Key-value pairs.
 - Allows null keys and values.
 - Unordered.
 - Example:


```
Map<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "Two");
```

- `System.out.println(map); // Output: {1=One, 2=Two}`
- **TreeMap:**
 - Key-value pairs sorted by keys.
 - Example:
 - `Map<Integer, String> map = new TreeMap<>();`
 - `map.put(2, "Two");`
 - `map.put(1, "One");`
 - `System.out.println(map); // Output: {1=One, 2=Two}`

4. Queue Implementations

The output `[1, 3, 2]` for a **PriorityQueue** is because the `PriorityQueue` class does not guarantee the order of elements when printing or iterating through the queue. It only guarantees that the **head of the queue** (the element retrieved by `poll()` or `peek()`) will always be the smallest element (based on natural ordering or a custom comparator).

Key Points About PriorityQueue Behavior

1. **Natural Ordering:**
 - Internally, the `PriorityQueue` uses a **binary heap** data structure.
 - The heap is partially sorted, ensuring that the smallest element is always at the root of the heap (the "head" of the queue).
 - However, the other elements may not appear in a fully sorted order when printing the queue.
2. **Iteration:**
 - When you print the `PriorityQueue` or iterate over it, the elements are displayed in the order they are stored internally in the heap, not in sorted order.

Example to Demonstrate PriorityQueue Behavior

```
import java.util.PriorityQueue;

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        pq.add(3);
        pq.add(1);
        pq.add(2);

        // Printing the PriorityQueue
        System.out.println(pq); // Output: [1, 3, 2] (internal heap
                                // structure)

        // Polling elements (removing in priority order)
        while (!pq.isEmpty()) {
            System.out.println(pq.poll()); // Output: 1, 2, 3 (sorted
            // order when removed)
        }
    }
}
```

- Elements are ordered based on natural order or a custom comparator.
- **Example:**
- `Queue<Integer> queue = new PriorityQueue<>();`
- `queue.add(3);`
- `queue.add(1);`
- `queue.add(2);`
- `System.out.println(queue);` // Output: [1, 3, 2] (natural ordering)

3. Common Operations on Collections

- **Adding Elements:**
- `list.add(1);` // Adds an element
- **Removing Elements:**
- `list.remove(0);` // Removes element at index 0
- **Iterating Through a Collection:**
 - Using a for-each loop:
 - `for (Integer num : list) {`
 - `System.out.println(num);`
 - `}`
 - Using an iterator:
 - `Iterator<Integer> it = list.iterator();`
 - `while (it.hasNext()) {`
 - `System.out.println(it.next());`
 - `}`

4. Key Concepts

Comparable vs Comparator

- **Comparable:**
 - Natural ordering of objects.
 - Implemented using `compareTo()` in the class.
 - **Example:**
 - `class Student implements Comparable<Student> {`
 - `int id;`
 - `String name;`
 - `public int compareTo(Student s) {`
 - `return this.id - s.id; // Ascending order`
 - `}`
 - `}`
- **Comparator:**
 - Custom ordering.
 - Implemented using `compare()` in a separate class.
 - **Example:**
 - `class NameComparator implements Comparator<Student> {`
 - `public int compare(Student s1, Student s2) {`
 - `return s1.name.compareTo(s2.name);`
 - `}`

- }

Fail-Fast vs Fail-Safe Iterators

- **Fail-Fast:**
 - Throws `ConcurrentModificationException` if the collection is modified during iteration (e.g., `ArrayList`, `HashMap`).
- **Fail-Safe:**
 - Works on a copy of the collection and does not throw exceptions (e.g., `ConcurrentHashMap`).

5. Thread-Safe Collections

- Use classes like `Vector`, `Stack`, `Hashtable`.
- Alternatively, wrap collections with `Collections.synchronizedList()`:
- ```
List<Integer> synchronizedList = Collections.synchronizedList(new ArrayList<>());
```

## Q&A

### 1. What is the difference between `ArrayList` and `LinkedList`?

Answer:

| Feature            | <code>ArrayList</code>              | <code>LinkedList</code>                 |
|--------------------|-------------------------------------|-----------------------------------------|
| Structure          | Resizable array                     | Doubly linked list                      |
| Access             | Faster for random access (indexed)  | Slower for random access                |
| Insertion/Deletion | Slower (requires shifting elements) | Faster (requires pointer adjustments)   |
| Memory Usage       | Uses less memory                    | Uses more memory (due to node pointers) |

### 2. What is the difference between `HashSet` and `TreeSet`?

Answer:

| Feature     | <code>HashSet</code>             | <code>TreeSet</code>       |
|-------------|----------------------------------|----------------------------|
| Ordering    | No order                         | Sorted order               |
| Performance | Faster for add, remove, contains | Slower due to sorting      |
| Null Values | Allows one null value            | Does not allow null values |

### 3. What is the difference between `HashMap` and `TreeMap`?

Answer:

| Feature     | HashMap                          | TreeMap                               |
|-------------|----------------------------------|---------------------------------------|
| Ordering    | No order                         | Sorted by keys                        |
| Performance | Faster for operations ( $O(1)$ ) | Slower ( $O(\log n)$ ) due to sorting |
| Null Keys   | Allows one null key              | Does not allow null keys              |

#### 4. How do you iterate through a `HashMap`?

Answer:

- Using `entrySet()`:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "Two");

for (Map.Entry<Integer, String> entry : map.entrySet()) {
 System.out.println(entry.getKey() + " = " + entry.getValue());
}
```
- Using a key set:

```
for (Integer key : map.keySet()) {
 System.out.println(key + " = " + map.get(key));
}
```

#### 5. What is the difference between `Comparable` and `Comparator`?

Answer:

| Feature        | Comparable                      | Comparator                       |
|----------------|---------------------------------|----------------------------------|
| Purpose        | Natural ordering                | Custom ordering                  |
| Implementation | Implemented in the class itself | Implemented in a separate class  |
| Method         | <code>compareTo()</code>        | <code>compare()</code>           |
| Usage          | Single sorting logic            | Multiple sorting logics possible |

#### 6. What is a `fail-fast` iterator?

Answer:

- A `fail-fast` iterator throws a `ConcurrentModificationException` if the collection is structurally modified while iterating.
- Example:

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);

for (Integer num : list) {
 list.add(3); // Throws ConcurrentModificationException
}
```

- }

## 7. How is a **fail-safe** iterator different from a **fail-fast** iterator?

**Answer:**

- A **fail-safe** iterator works on a copy of the collection, so it does not throw exceptions if the collection is modified during iteration.
- **Example:**
- ```
ConcurrentHashMap<Integer, String> map = new  
ConcurrentHashMap<>();
```
- ```
map.put(1, "One");
```
- 
- ```
for (Map.Entry<Integer, String> entry : map.entrySet()) {
```
- ```
 map.put(2, "Two"); // No exception
```
- ```
}
```

8. How do you synchronize a collection?

Answer:

- Using `Collections.synchronizedList()`:
- ```
List<Integer> synchronizedList = Collections.synchronizedList(new
ArrayList<>());
```
- Using thread-safe classes like `Vector` or `ConcurrentHashMap`.

## 9. What is the difference between **HashMap** and **Hashtable**?

**Answer:**

| Feature                 | HashMap                                      | Hashtable                          |
|-------------------------|----------------------------------------------|------------------------------------|
| <b>Thread Safety</b>    | Not thread-safe                              | Thread-safe (synchronized)         |
| <b>Null Keys/Values</b> | Allows one null key and multiple null values | Does not allow null keys or values |
| <b>Performance</b>      | Faster                                       | Slower                             |

## 10. What is a **PriorityQueue**?

**Answer:** A `PriorityQueue` is a queue where elements are ordered based on their natural ordering or a custom comparator.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(3);
pq.add(1);
pq.add(2);
```

```
System.out.println(pq); // Output: [1, 3, 2] (natural order)
```

## 11. What is the difference between `ArrayList` and `Vector`?

Answer:

| Feature       | <code>ArrayList</code> | <code>Vector</code>           |
|---------------|------------------------|-------------------------------|
| Thread Safety | Not thread-safe        | Thread-safe                   |
| Performance   | Faster                 | Slower due to synchronization |
| Growth Policy | Increases size by 50%  | Doubles its size              |

## 12. What is the difference between `Collection` and `Collections`?

Answer:

- **Collection:** An interface that defines a group of objects (e.g., `List`, `Set`).
- **Collections:** A utility class that provides static methods to operate on collections (e.g., `sort()`, `shuffle()`).

## 13. How do you sort a collection?

Answer:

- Natural order:

```
List<Integer> list = new ArrayList<>(Arrays.asList(3, 1, 2));
Collections.sort(list);
System.out.println(list); // Output: [1, 2, 3]
```
- Custom order using a comparator:

```
Collections.sort(list, (a, b) -> b - a); // Descending order
System.out.println(list); // Output: [3, 2, 1]
```

## 14. How do you convert a collection to an array?

Answer: Use the `toArray()` method:

```
List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
String[] arr = list.toArray(new String[0]);
System.out.println(Arrays.toString(arr)); // Output: [A, B, C]
```

## 15. What are the key differences between `Queue` and `Stack`?

Answer:

| Feature | <code>Queue</code> | <code>Stack</code> |
|---------|--------------------|--------------------|
|---------|--------------------|--------------------|

|                       |                           |                          |
|-----------------------|---------------------------|--------------------------|
| <b>Ordering</b>       | FIFO (First-In-First-Out) | LIFO (Last-In-First-Out) |
| <b>Use Case</b>       | Task scheduling           | Undo operations          |
| <b>Common Classes</b> | PriorityQueue, Deque      | Stack (legacy class)     |

## Exception Handling.

### Topic: Exception Handling

**Exception Handling** in Java is a mechanism to handle runtime errors, ensuring the normal flow of a program even when unexpected issues occur.

## 1. Key Concepts

1. **Exception:**
  - An event that disrupts the normal flow of a program.
  - Examples: `ArithmeticException`, `NullPointerException`.
2. **Error:**
  - Serious issues that the application cannot recover from.
  - Examples: `OutOfMemoryError`, `StackOverflowError`.
3. **Hierarchy:**
  - **Throwable:** The root class.
    - **Exception:** Recoverable issues.
      - Checked Exceptions (e.g., `IOException`, `SQLException`).
      - Unchecked Exceptions (e.g., `NullPointerException`, `ArithmeticException`).
    - **Error:** Irrecoverable issues (e.g., JVM errors).

## 2. Types of Exceptions

1. **Checked Exceptions:**
  - Checked at compile time.
  - Must be declared using `throws` or handled with `try-catch`.
  - Examples: `IOException`, `SQLException`.
2. **Unchecked Exceptions:**
  - Not checked at compile time.
  - Occur due to programming errors.
  - Examples: `NullPointerException`, `ArithmeticException`.

## 3. Exception Handling Keywords

1. **try:**
  - o Contains code that might throw an exception.
  - o Example:

```
try {
 int result = 10 / 0;
}
```
2. **catch:**
  - o Handles the exception thrown by the `try` block.
  - o Example:

```
try {
 int result = 10 / 0;
} catch (ArithmeticException e) {
 System.out.println("Cannot divide by zero");
}
```
3. **finally:**
  - o Always executed, regardless of whether an exception occurred or not.
  - o Example:

```
try {
 int result = 10 / 2;
} catch (Exception e) {
 System.out.println("Error");
} finally {
 System.out.println("This block always executes");
}
```
4. **throw:**
  - o Used to explicitly throw an exception.
  - o Example:

```
throw new IllegalArgumentException("Invalid argument");
```
5. **throws:**
  - o Declares the exceptions that a method might throw.
  - o Example:

```
void readFile() throws IOException {
 // Code that might throw IOException
}
```

## 4. Custom Exceptions

You can define your own exception classes by extending the `Exception` class.

```
class MyException extends Exception {
 public MyException(String message) {
 super(message);
 }
}

public class Main {
 public static void main(String[] args) {
 try {
 throw new MyException("Custom exception occurred");
 } catch (MyException e) {
 System.out.println(e.getMessage());
 }
 }
}
```

## 5. Commonly Used Exception Classes

| Exception                      | Description                                                                |
|--------------------------------|----------------------------------------------------------------------------|
| ArithmeticException            | Thrown when an illegal arithmetic operation occurs (e.g., divide by zero). |
| NullPointerException           | Thrown when an application tries to use a null reference.                  |
| ArrayIndexOutOfBoundsException | Thrown when an array is accessed with an illegal index.                    |
| ClassNotFoundException         | Thrown when a class is not found.                                          |
| IOException                    | Thrown for input/output operations.                                        |
| FileNotFoundException          | Subclass of <code>IOException</code> for missing files.                    |
| SQLException                   | Thrown when there is a database access error.                              |
| IllegalArgumentException       | Thrown when an illegal argument is passed to a method.                     |

## 6. Best Practices

### 1. Catch Specific Exceptions:

- Avoid using generic `Exception` unless necessary.
- Example:
- ```
try {
```
- ```
 // Code
```
- ```
} catch (IOException e) {
```
- ```
 // Handle IO issues
```
- ```
} catch (SQLException e) {
```
- ```
 // Handle SQL issues
```
- ```
}
```

2. Use Finally for Cleanup:

- Always release resources like files or database connections in the `finally` block.

3. Log Exceptions:

- Use a logging framework to record exceptions.

4. Avoid Swallowing Exceptions:

- Don't catch exceptions and do nothing:
- ```
try {
```
- ```
    // Code
```
- ```
} catch (Exception e) {
```
- ```
    // Bad practice: no action taken
```
- ```
}
```

### 5. Throw Meaningful Custom Exceptions:

- Provide descriptive messages when throwing custom exceptions.

## Q&A

### 1. What is the difference between `throw` and `throws`?

**Answer:**

| Feature | throw                                            | throws                                                 |
|---------|--------------------------------------------------|--------------------------------------------------------|
| Purpose | Used to explicitly throw an exception.           | Declares exceptions that a method can throw.           |
| Usage   | Inside a method or block.                        | In the method signature.                               |
| Example | <code>throw new<br/>IOException("Error");</code> | <code>void readFile() throws<br/>IOException {}</code> |

## 2. What is the difference between checked and unchecked exceptions?

Answer:

| Feature      | Checked Exceptions                                                                  | Unchecked Exceptions                                                      |
|--------------|-------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| Compile-Time | Checked at compile time.                                                            | Not checked at compile time.                                              |
| Handling     | Must be handled with <code>try-catch</code> or declared using <code>throws</code> . | Handling is optional.                                                     |
| Examples     | <code>IOException</code> , <code>SQLException</code> .                              | <code>NullPointerException</code> ,<br><code>ArithmeticException</code> . |

## 3. Can a `finally` block be skipped?

Answer: The `finally` block is always executed except in these scenarios:

1. When the JVM exits during the `try` or `catch` block using `System.exit()`.
2. When a thread is interrupted or killed during execution.

Example:

```
try {
 System.exit(0); // Skips the finally block
} finally {
 System.out.println("This won't be printed");
}
```

## 4. What happens if an exception is not handled?

Answer: If an exception is not handled:

1. It propagates up the call stack.
2. If no method handles it, the JVM terminates the program and prints a stack trace.

## 5. Can you have multiple `catch` blocks for a single `try` block?

Answer: Yes, you can have multiple `catch` blocks to handle different exceptions.



Example:

```
try {
 int result = 10 / 0;
} catch (ArithmeticException e) {
 System.out.println("Arithmetic exception");
} catch (Exception e) {
 System.out.println("Generic exception");
}
```

## 6. Can you catch multiple exceptions in a single `catch` block?

**Answer:** Yes, using Java 7+ you can catch multiple exceptions in one `catch` block using a pipe (`|`).

Example:

```
try {
 int[] arr = new int[2];
 arr[3] = 10; // ArrayIndexOutOfBoundsException
} catch (ArrayIndexOutOfBoundsException | ArithmeticException e) {
 System.out.println("Exception caught: " + e);
}
```

## 7. What is the difference between `final`, `finally`, and `finalize()`?

**Answer:**

| Feature        | <b>final</b>                                                          | <b>finally</b>                               | <b>finalize()</b>                                          |
|----------------|-----------------------------------------------------------------------|----------------------------------------------|------------------------------------------------------------|
| <b>Purpose</b> | Used to define constants, prevent inheritance, or prevent overriding. | Used for cleanup code in exception handling. | Used for garbage collection before an object is destroyed. |
| <b>Usage</b>   | With classes, methods, or variables.                                  | With <code>try-catch</code> blocks.          | As a method in the <code>Object</code> class.              |

## 8. What happens if an exception is thrown in the `catch` block?

**Answer:** If an exception is thrown in the `catch` block, it propagates up the call stack unless handled in a nested `try-catch`.

Example:

```
try {
 int result = 10 / 0;
} catch (ArithmeticException e) {
 throw new RuntimeException("Exception in catch block"); //
 Propagates further
}
```

## 9. Can a `try` block exist without a `catch` block?

**Answer:** Yes, a `try` block can exist without a `catch` block, provided there is a `finally` block.

Example:

```
try {
 System.out.println("Try block");
} finally {
 System.out.println("Finally block");
}
```

## 10. Can you rethrow an exception in Java?

**Answer:** Yes, you can rethrow an exception using the `throw` keyword.

Example:

```
try {
 throw new IOException("Initial exception");
} catch (IOException e) {
 System.out.println("Caught exception");
 throw e; // Rethrows the exception
}
```

## 11. How do you create a custom exception in Java?

**Answer:** Define a class that extends `Exception` or `RuntimeException`.

Example:

```
class MyException extends Exception {
 public MyException(String message) {
 super(message);
 }
}

public class Main {
 public static void main(String[] args) {
 try {
 throw new MyException("Custom exception");
 } catch (MyException e) {
 System.out.println(e.getMessage());
 }
 }
}
```

## 12. Can you have a `try` block inside a `catch` block?

**Answer:** Yes, you can nest `try-catch` blocks.

Example:

```
try {
 int result = 10 / 0;
} catch (ArithmeticException e) {
 try {
 int[] arr = new int[2];
 arr[3] = 10; // Nested try
 } catch (ArrayIndexOutOfBoundsException ex) {
 System.out.println("Nested exception");
 }
}
```

### 13. What is a `stack trace` in Java?

**Answer:** A stack trace is a report of the active stack frames at a specific point in time, typically displayed when an exception occurs. It shows the sequence of method calls leading to the exception.

### 14. Can you have both `catch` and `finally` blocks for the same `try` block?

**Answer:** Yes, both can coexist for the same `try` block.

Example:

```
try {
 int result = 10 / 0;
} catch (ArithmeticException e) {
 System.out.println("Catch block");
} finally {
 System.out.println("Finally block");
}
```

### 15. Why is exception handling important?

**Answer:**

1. Ensures smooth program execution by handling errors.
2. Prevents program termination on encountering exceptions.
3. Allows graceful recovery from runtime issues.

## Multithreading and Concurrency.

## Topic: Multithreading and Concurrency

**Multithreading** in Java allows multiple threads to run concurrently, enabling parallel execution and efficient use of CPU resources. **Concurrency** involves managing access to shared resources and ensuring thread-safe operations.

### 1. Key Concepts

1. **Thread:**
  - A lightweight process that runs independently.
  - Created using the `Thread` class or the `Runnable` interface.
2. **Multithreading:**
  - A process of executing multiple threads simultaneously.
  - Benefits:
    - Efficient CPU usage.
    - Reduced response time for complex applications.
3. **Concurrency:**
  - Managing the execution of threads that share resources to avoid conflicts.

### 2. Creating Threads

1. **Extending the `Thread` Class:**
  - Override the `run()` method.
  - Example:

```
class MyThread extends Thread {
 public void run() {
 System.out.println("Thread is running");
 }
}
```
  - ```
public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // Starts the thread
    }
}
```
2. **Implementing the `Runnable` Interface:**
 - Pass the `Runnable` object to a `Thread` object.
 - Example:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running");
    }
}
```
 - ```
public class Main {
 public static void main(String[] args) {
 Thread thread = new Thread(new MyRunnable());
 thread.start(); // Starts the thread
 }
}
```

### 3. Thread States

| State                  | Description                                                        |
|------------------------|--------------------------------------------------------------------|
| <b>New</b>             | A thread is created but not started ( <code>new Thread()</code> ). |
| <b>Runnable</b>        | A thread is ready to run but waiting for CPU time.                 |
| <b>Running</b>         | A thread is currently executing.                                   |
| <b>Blocked/Waiting</b> | A thread is waiting for a resource or another thread.              |
| <b>Terminated</b>      | A thread has completed execution.                                  |

### 4. Thread Methods

| Method                          | Description                                                |
|---------------------------------|------------------------------------------------------------|
| <code>start()</code>            | Starts the thread and calls the <code>run()</code> method. |
| <code>run()</code>              | Defines the thread's task.                                 |
| <code>sleep(long millis)</code> | Pauses the thread for a specified duration.                |
| <code>join()</code>             | Waits for the thread to finish execution.                  |
| <code>isAlive()</code>          | Checks if the thread is still active.                      |
| <code>setPriority(int)</code>   | Sets the thread's priority.                                |

### 5. Synchronization

- Ensures that only one thread accesses a critical section at a time.
- **Synchronized Block:**
- ```
synchronized (this) {
```
- ```
 // Critical section
```
- ```
}
```
- **Synchronized Method:**
- ```
synchronized void display() {
```
- ```
    // Critical section
```
- ```
}
```

### 6. Inter-Thread Communication

- Use methods like `wait()`, `notify()`, and `notifyAll()` within synchronized blocks.
- **Example:**
- ```
class SharedResource {
```
- ```
 synchronized void produce() throws InterruptedException {
```
- ```
        System.out.println("Producing...");
```
- ```
 wait();
```
- ```
        System.out.println("Resumed production");
```
- ```
 }
```
- ```
}
```
- ```
synchronized void consume() {
```

```

• System.out.println("Consuming...");
• notify();
• }
• }
•
• public class Main {
• public static void main(String[] args) {
• SharedResource resource = new SharedResource();
•
• Thread producer = new Thread(() -> {
• try {
• resource.produce();
• } catch (InterruptedException e) {
• e.printStackTrace();
• }
• });
•
• Thread consumer = new Thread(resource::consume);
•
• producer.start();
• consumer.start();
• }
• }

```

## 7. Thread Safety

- Use synchronization or concurrent classes to avoid data inconsistency.
- Examples of thread-safe classes:
  - ConcurrentHashMap
  - CopyOnWriteArrayList

## 8. Executors Framework

Introduced in Java 5 to simplify thread management.

- **Creating a Thread Pool:**
- `ExecutorService executor = Executors.newFixedThreadPool(3);`
- `executor.submit(() -> System.out.println("Task executed"));`
- `executor.shutdown();`

## 9. Deadlocks

A deadlock occurs when two or more threads are waiting for each other's resources indefinitely.

**Example:**

```

class Deadlock {
 public static void main(String[] args) {
 String resource1 = "Resource1";
 String resource2 = "Resource2";

 Thread t1 = new Thread(() -> {
 synchronized (resource1) {
 System.out.println("Thread 1: Locked resource1");
 synchronized (resource2) {
 System.out.println("Thread 1: Locked resource2");
 }
 }
 });

 Thread t2 = new Thread(() -> {
 synchronized (resource2) {
 System.out.println("Thread 2: Locked resource2");
 synchronized (resource1) {
 System.out.println("Thread 2: Locked resource1");
 }
 }
 });

 t1.start();
 t2.start();
 }
}

```

## 10. Volatile and Atomic Variables

### 1. Volatile:

- Ensures visibility of changes to variables across threads.
- Example:
- `private volatile boolean flag = true;`

### 2. Atomic Variables:

- Provides atomic operations for variables.
- Example:
- `AtomicInteger counter = new AtomicInteger(0);`
- `counter.incrementAndGet();`

## Q&A

### 1. What is the difference between a process and a thread?

**Answer:**

| Feature              | Process                               | Thread                                     |
|----------------------|---------------------------------------|--------------------------------------------|
| <b>Definition</b>    | A program in execution.               | A smaller unit of a process.               |
| <b>Memory</b>        | Has its own memory space.             | Shares memory with other threads.          |
| <b>Communication</b> | Inter-process communication required. | Shared memory allows faster communication. |

| Feature  | Process          | Thread          |
|----------|------------------|-----------------|
| Overhead | Higher overhead. | Lower overhead. |

## 2. What are the two ways to create a thread in Java?

Answer:

### 1. Extending the Thread class:

- Override the `run()` method.
- Example:
- ```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running");
    }
}
```
- ```
MyThread thread = new MyThread();
thread.start();
```

### 2. Implementing the Runnable interface:

- Pass the Runnable object to a Thread object.
- Example:
- ```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread running");
    }
}
```
- ```
Thread thread = new Thread(new MyRunnable());
thread.start();
```

## 3. What is the difference between start() and run()?

Answer:

| Feature | start()                                            | run()                                              |
|---------|----------------------------------------------------|----------------------------------------------------|
| Purpose | Starts a new thread and calls <code>run()</code> . | Executes <code>run()</code> in the current thread. |
| Thread  | Creates a new thread.                              | Does not create a new thread.                      |

Example:

```
Thread t = new Thread(() -> System.out.println("Running"));
t.start(); // Executes in a new thread
t.run(); // Executes in the current thread
```

## 4. What are the states of a thread in Java?

Answer:

1. **New:** Thread is created but not started (`new Thread()`).
2. **Runnable:** Thread is ready to run but waiting for CPU.



3. **Running:** Thread is executing.
4. **Blocked/Waiting:** Thread is waiting for a resource or signal.
5. **Terminated:** Thread has completed execution.

## 5. What is a daemon thread?

**Answer:**

- A daemon thread is a background thread that provides services to other threads.
- It terminates automatically when all non-daemon threads finish execution.
- **Example:**
- ```
Thread t = new Thread() -> System.out.println("Daemon thread");
```
- ```
t.setDaemon(true); // Sets the thread as a daemon
```
- ```
t.start();
```

6. How do you achieve thread synchronization in Java?

Answer:

- Use the `synchronized` keyword to ensure that only one thread can execute a critical section at a time.
- **Example:**
- ```
class Counter {
```
- ```
    private int count = 0;
```
-
- ```
 synchronized void increment() {
```
- ```
        count++;
```
- ```
 }
```
- ```
}
```

7. What is the difference between `wait()` and `sleep()`?

Answer:

Feature	<code>wait()</code>	<code>sleep()</code>
Belongs To	Object class.	Thread class.
Releases Lock	Yes, releases the lock on the object.	No, does not release the lock.
Usage	Used in inter-thread communication.	Pauses thread execution for a duration.

8. What is the difference between `notify()` and `notifyAll()`?

Answer:

Feature	<code>notify()</code>	<code>notifyAll()</code>
Purpose	Wakes up one thread waiting on the object's monitor.	Wakes up all threads waiting on the object's monitor.
Usage	Use when only one thread needs to be resumed.	Use when multiple threads need to be resumed.

9. What is the `Executor` framework in Java?

Answer: The `Executor` framework provides a mechanism to manage and control thread execution efficiently. It allows thread pooling to reduce the overhead of thread creation and destruction.

Example:

```
ExecutorService executor = Executors.newFixedThreadPool(3);
executor.submit(() -> System.out.println("Task executed"));
executor.shutdown();
```

10. What is a deadlock, and how can you prevent it?

Answer:

- **Deadlock:** Occurs when two or more threads are waiting for each other to release resources, and neither can proceed.
- **Prevention:**
 1. Avoid nested locks.
 2. Use a consistent locking order.
 3. Use a timeout with `tryLock()` (from `ReentrantLock`).

Example of Deadlock:

```
class Deadlock {
    public static void main(String[] args) {
        String resource1 = "Resource1";
        String resource2 = "Resource2";

        Thread t1 = new Thread(() -> {
            synchronized (resource1) {
                synchronized (resource2) {
                    System.out.println("Thread 1 completed");
                }
            }
        });

        Thread t2 = new Thread(() -> {
            synchronized (resource2) {
                synchronized (resource1) {
                    System.out.println("Thread 2 completed");
                }
            }
        });
    }
}
```

```

        t1.start();
        t2.start();
    }

```

11. What is the purpose of the `volatile` keyword?

Answer:

- Ensures that changes to a variable are visible to all threads.
- Prevents threads from caching the variable locally.
- Example:
- `private volatile boolean flag = true`

12. What are atomic variables in Java?

Answer: Atomic variables provide thread-safe operations without synchronization. They are part of the `java.util.concurrent.atomic` package. Example:

```

AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet(); // Atomically increments the value

```

13. What is thread starvation?

Answer:

- Occurs when a thread is perpetually denied access to resources because other threads with higher priority are always favored.
- **Solution:**
 - Avoid using thread priorities excessively.
 - Use fair locks (e.g., `ReentrantLock` with fairness).

14. How does `join()` work in Java?

Answer: The `join()` method makes the current thread wait until the specified thread finishes execution.

Example:

```

Thread t = new Thread(() -> {
    System.out.println("Thread running");
});
t.start();
t.join(); // Waits for thread t to finish
System.out.println("Main thread continues");

```

15. How do you stop a thread in Java?

Answer:

- **Deprecated Way:** Using `stop()` (not recommended due to unsafe termination).
- **Proper Way:** Use a flag to signal the thread to stop.
- ```
class MyThread extends Thread {
 private volatile boolean running = true;
 •
 •
 public void run() {
 while (running) {
 System.out.println("Running");
 }
 }
 •
 public void stopThread() {
 running = false;
 }
 •
}
```

## Java 8

### Detailed Explanation of Java 8 Features with Examples

Java 8 introduced several major enhancements, making it one of the most significant updates in the history of Java. Below are the **key features** with detailed explanations and examples.

## 1. Lambda Expressions

Lambda expressions provide a concise way to write anonymous functions. They enable functional programming and reduce boilerplate code.

### Syntax

```
(parameters) -> expression
(parameters) -> { statements; }
```

### Example: Using a Lambda Expression

```
import java.util.*;

public class LambdaExample {
 public static void main(String[] args) {
 List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

 // Using a lambda expression to iterate
 names.forEach(name -> System.out.println(name));
 }
}
```

## 2. Functional Interfaces

A **functional interface** has exactly one abstract method but can have multiple default or static methods. The `@FunctionalInterface` annotation is used for better readability.

### Examples

- Built-in Functional Interfaces:
  - **Predicate:** Takes a value and returns `true` or `false`.
  - **Consumer:** Takes a value and returns nothing.
  - **Function:** Takes one value and returns another.

#### Example: Custom Functional Interface

```
@FunctionalInterface
interface Calculator {
 int calculate(int a, int b);
}

public class FunctionalInterfaceExample {
 public static void main(String[] args) {
 Calculator add = (a, b) -> a + b;
 System.out.println(add.calculate(10, 5)); // Output: 15
 }
}
```

## 3. Streams API

The **Streams API** is used for processing sequences of elements, such as collections, in a functional style. It supports operations like **filter**, **map**, and **reduce**.

#### Example: Using Streams

```
import java.util.*;
import java.util.stream.*;

public class StreamExample {
 public static void main(String[] args) {
 List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

 // Filtering and mapping
 List<Integer> evenNumbers = numbers.stream()
 .filter(num -> num % 2 == 0)
 .map(num -> num * num)

 .collect(Collectors.toList());

 System.out.println(evenNumbers); // Output: [4, 16]
 }
}
```

## 4. Default and Static Methods in Interfaces

Default methods allow interfaces to have method implementations. This avoids the need to modify all implementing classes when a new method is added.

### *Example: Default and Static Methods*

```
interface Greeting {
 default void sayHello() {
 System.out.println("Hello!");
 }

 static void sayGoodbye() {
 System.out.println("Goodbye!");
 }
}

public class DefaultMethodExample implements Greeting {
 public static void main(String[] args) {
 new DefaultMethodExample().sayHello(); // Output: Hello!
 Greeting.sayGoodbye(); // Output: Goodbye!
 }
}
```

## **5. Optional Class**

The `Optional` class is used to represent optional values, avoiding `NullPointerException`. It provides methods to check, get, or act on a value if present.

### *Example: Using Optional*

```
import java.util.Optional;

public class OptionalExample {
 public static void main(String[] args) {
 Optional<String> name = Optional.ofNullable(null);

 // Check if a value is present
 System.out.println(name.isPresent()); // Output: false

 // Provide a default value
 System.out.println(name.orElse("Default Name")); // Output:
Default Name
 }
}
```

## **6. New Date and Time API**

The new `java.time` package provides better date and time handling with immutability and thread-safety.

### *Key Classes*

- `LocalDate`: Represents a date without time.
- `LocalTime`: Represents a time without a date.
- `LocalDateTime`: Represents both date and time.
- `Duration` and `Period`: Measure time durations.

### *Example: Using Date and Time API*

```
import java.time.*;
```

```

public class DateTimeExample {
 public static void main(String[] args) {
 LocalDate today = LocalDate.now();
 System.out.println("Today's Date: " + today); // Output: e.g.,
2025-01-25

 LocalTime now = LocalTime.now();
 System.out.println("Current Time: " + now);

 LocalDateTime dateTime = LocalDateTime.of(2025, Month.JANUARY,
25, 10, 30);
 System.out.println("Custom Date and Time: " + dateTime);

 // Adding 10 days
 System.out.println("10 Days Later: " + today.plusDays(10));
 }
}

```

## 7. Method References

Method references are a shorthand for lambda expressions that invoke methods directly.

### *Example: Method Reference*

```

import java.util.*;

public class MethodReferenceExample {
 public static void main(String[] args) {
 List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

 // Using method reference
 names.forEach(System.out::println);
 }
}

```

## 8. Collectors in Streams

Collectors is a utility class used to collect the results of a stream into various forms like lists, sets, or maps.

### *Example: Collectors Usage*

```

import java.util.*;
import java.util.stream.*;

public class CollectorsExample {
 public static void main(String[] args) {
 List<String> names = Arrays.asList("Alice", "Bob", "Charlie",
"Alice");

 // Collecting as a list
 List<String> distinctNames = names.stream()
 .distinct()
 .collect(Collectors.toList());
 System.out.println(distinctNames); // Output: [Alice, Bob,
Charlie]
 }
}

```

```

 // Counting elements
 long count = names.stream().filter(name ->
name.startsWith("A")).count();
 System.out.println(count); // Output: 2
 }
}

```

## 9. Parallel Streams

Parallel streams divide the processing of a stream into multiple threads for better performance.

### *Example: Using Parallel Streams*

```

import java.util.*;
import java.util.stream.*;

public class ParallelStreamExample {
 public static void main(String[] args) {
 List<Integer> numbers = IntStream.range(1,
10).boxed().collect(Collectors.toList());

 // Using parallel stream
 numbers.parallelStream()
 .forEach(num ->
System.out.println(Thread.currentThread().getName() + " - " + num));
 }
}

```

## Q&A

Google it as there are my possibilities

## Topic: File I/O and Serialization

### Topic

Java provides APIs for handling file input and output operations, as well as serializing objects for storage or transmission.

### 1. File I/O in Java

Java uses classes from the `java.io` and `java.nio` packages for file operations.

#### *Common Classes for File I/O*

| Class | Description                              |
|-------|------------------------------------------|
| File  | Represents file and directory pathnames. |



|                  |                                              |
|------------------|----------------------------------------------|
| FileReader       | Reads character files.                       |
| FileWriter       | Writes character files.                      |
| BufferedReader   | Reads text from an input stream efficiently. |
| BufferedWriter   | Writes text to an output stream efficiently. |
| FileInputStream  | Reads binary data from a file.               |
| FileOutputStream | Writes binary data to a file.                |

### *Example: File Reading and Writing*

```
import java.io.*;

public class FileExample {
 public static void main(String[] args) throws IOException {
 // Writing to a file
 FileWriter writer = new FileWriter("example.txt");
 writer.write("Hello, File I/O!");
 writer.close();

 // Reading from a file
 FileReader reader = new FileReader("example.txt");
 BufferedReader bufferedReader = new BufferedReader(reader);
 String line;
 while ((line = bufferedReader.readLine()) != null) {
 System.out.println(line);
 }
 bufferedReader.close();
 }
}
```

## 2. Serialization

Serialization is the process of converting an object into a byte stream, so it can be saved to a file or sent over a network. Deserialization reverses this process.

### *Steps for Serialization*

1. A class must implement the `Serializable` interface.
2. Use `ObjectOutputStream` to write the object.
3. Use `ObjectInputStream` to read the object.

### *Example: Serialization and Deserialization*

```
import java.io.*;

// Serializable class
class Person implements Serializable {
 private static final long serialVersionUID = 1L;
 String name;
 int age;

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

 @Override
```

```

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

 public class SerializationExample {
 public static void main(String[] args) throws IOException,
 ClassNotFoundException {
 // Serialization
 Person person = new Person("Alice", 25);
 ObjectOutputStream out = new ObjectOutputStream(new
 FileOutputStream("person.ser"));
 out.writeObject(person);
 out.close();

 // Deserialization
 ObjectInputStream in = new ObjectInputStream(new
 FileInputStream("person.ser"));
 Person deserializedPerson = (Person) in.readObject();
 in.close();

 System.out.println(deserializedPerson); // Output:
 Person{name='Alice', age=25}
 }
 }

```

### 3. File Class

The `File` class represents file and directory pathnames.

#### *Common Methods in File Class*

| Method                       | Description                                  |
|------------------------------|----------------------------------------------|
| <code>createNewFile()</code> | Creates a new file.                          |
| <code>delete()</code>        | Deletes a file.                              |
| <code>exists()</code>        | Checks if a file exists.                     |
| <code>getName()</code>       | Returns the file name.                       |
| <code>length()</code>        | Returns the file size in bytes.              |
| <code>list()</code>          | Returns a list of file names in a directory. |

#### *Example: File Operations*

```

import java.io.*;

public class FileClassExample {
 public static void main(String[] args) throws IOException {
 File file = new File("example.txt");

 // Create a new file
 if (file.createNewFile()) {
 System.out.println("File created: " + file.getName());
 } else {
 System.out.println("File already exists.");
 }

 // Check if file exists
 if (file.exists()) {
 System.out.println("File size: " + file.length() + "
 bytes");
 }
 }
}

```

```

 }

 // Delete the file
 if (file.delete()) {
 System.out.println("File deleted: " + file.getName());
 }
}
}

```

## 4. Buffered Streams

Buffered streams improve performance by reducing the number of I/O operations.

### *Example: Using Buffered Streams*

```

import java.io.*;

public class BufferedStreamExample {
 public static void main(String[] args) throws IOException {
 String content = "Buffered streams are fast!";

 // Writing using BufferedWriter
 BufferedWriter writer = new BufferedWriter(new
 FileWriter("buffered.txt"));
 writer.write(content);
 writer.close();

 // Reading using BufferedReader
 BufferedReader reader = new BufferedReader(new
 FileReader("buffered.txt"));
 String line;
 while ((line = reader.readLine()) != null) {
 System.out.println(line); // Output: Buffered streams are
fast!
 }
 reader.close();
 }
}

```

## 5. New I/O (NIO)

The `java.nio` package provides advanced features like non-blocking I/O, file channels, and memory mapping.

### *Example: File NIO*

```

import java.nio.file.*;
import java.io.IOException;

public class NIOExample {
 public static void main(String[] args) throws IOException {
 Path path = Paths.get("nio_example.txt");

 // Write to a file
 Files.write(path, "Hello, NIO!".getBytes());

 // Read from a file
 }
}

```

```

 String content = Files.readString(path);
 System.out.println(content); // Output: Hello, NIO!

 // Delete the file
 Files.delete(path);
 }
}

```

## 6. Best Practices for File I/O

1. Always close streams in a **finally** block or use a **try-with-resources** statement.
2. `try (FileReader reader = new FileReader("example.txt")) {`
3.     `// Reading logic`
4. `}`
5. Use **Buffered Streams** for better performance.
6. Handle exceptions using `try-catch` blocks.

## Q&A

### 1. What is the difference between `FileReader` and `FileInputStream`?

Answer:

| Feature            | <code>FileReader</code>                    | <code>FileInputStream</code>                                |
|--------------------|--------------------------------------------|-------------------------------------------------------------|
| <b>Purpose</b>     | Reads character files.                     | Reads binary files.                                         |
| <b>Input Type</b>  | Character data (e.g., <code>.txt</code> ). | Binary data (e.g., <code>.jpg</code> , <code>.mp3</code> ). |
| <b>Performance</b> | Suitable for text files.                   | Suitable for binary files.                                  |

### 2. What is the difference between `BufferedReader` and `BufferedWriter`?

Answer:

| Feature         | <code>BufferedReader</code> | <code>BufferedWriter</code> |
|-----------------|-----------------------------|-----------------------------|
| <b>Purpose</b>  | Reads text efficiently.     | Writes text efficiently.    |
| <b>Use Case</b> | Reading large text files.   | Writing large text files.   |

### 3. What is Serialization in Java?

**Answer:** Serialization is the process of converting an object into a byte stream so it can be stored or transmitted. Deserialization is the reverse process of reconstructing the object.

### 4. What is the `serialVersionUID`?

**Answer:** The `serialVersionUID` is a unique identifier for a `Serializable` class. It is used during deserialization to verify that the sender and receiver of a serialized object are compatible.

Example:

```
private static final long serialVersionUID = 1L;
```

## 5. How do you create a file in Java?

**Answer:** Use the `File` class and call the `createNewFile()` method.

```
File file = new File("example.txt");
if (file.createNewFile()) {
 System.out.println("File created: " + file.getName());
}
```

## 6. How do you delete a file in Java?

**Answer:** Use the `delete()` method of the `File` class.

```
File file = new File("example.txt");
if (file.delete()) {
 System.out.println("File deleted: " + file.getName());
}
```

## 7. What is the difference between `FileOutputStream` and `FileWriter`?

**Answer:**

| Feature  | <code>FileOutputStream</code>    | <code>FileWriter</code>          |
|----------|----------------------------------|----------------------------------|
| Purpose  | Writes binary data to a file.    | Writes character data to a file. |
| Use Case | Binary files (e.g., .png, .mp3). | Text files (e.g., .txt).         |

## 8. How do you read a file line by line in Java?

**Answer:** Use the `BufferedReader` class.

```
BufferedReader reader = new BufferedReader(new
FileReader("example.txt"));
String line;
while ((line = reader.readLine()) != null) {
 System.out.println(line);
}
reader.close();
```

## 9. What is the purpose of the `try-with-resources` statement?

**Answer:** The `try-with-resources` statement automatically closes resources like streams after usage.

Example:

```
try (BufferedReader reader = new BufferedReader(new
FileReader("example.txt"))) {
 String line;
 while ((line = reader.readLine()) != null) {
 System.out.println(line);
 }
}
```

## 10. Can a `static` field be serialized?

**Answer:** No, static fields are not serialized because they belong to the class, not to any instance.

## 11. How do you write binary data to a file in Java?

**Answer:** Use the `FileOutputStream` class.

```
FileOutputStream fos = new FileOutputStream("binaryfile.dat");
fos.write(new byte[]{1, 2, 3, 4});
fos.close();
```

## 12. What is the difference between `java.io` and `java.nio`?

**Answer:**

| Feature                      | <code>java.io</code>           | <code>java.nio</code>                |
|------------------------------|--------------------------------|--------------------------------------|
| <b>Blocking/Non-Blocking</b> | Blocking I/O                   | Non-blocking I/O                     |
| <b>Performance</b>           | Slower due to thread blocking. | Faster with non-blocking operations. |
| <b>Use Case</b>              | Simple file operations.        | High-performance file handling.      |

---

## 13. How do you check if a file exists in Java?

**Answer:** Use the `exists()` method of the `File` class.

```
File file = new File("example.txt");
if (file.exists()) {
 System.out.println("File exists");
}
```

## 14. How do you list all files in a directory?

**Answer:** Use the `list()` method of the `File` class.

```
File directory = new File("myFolder");
String[] files = directory.list();
for (String file : files) {
 System.out.println(file);
}
```

## 15. What is the advantage of using `BufferedReader` over `FileReader`?

**Answer:** `BufferedReader` reads data in chunks, reducing the number of I/O operations and improving performance compared to `FileReader`.

## 16. How do you copy the contents of one file to another in Java?

**Answer:** Using `BufferedReader` and `BufferedWriter`:

```
try (BufferedReader reader = new BufferedReader(new
FileReader("source.txt"));
 BufferedWriter writer = new BufferedWriter(new
FileWriter("destination.txt"))) {
 String line;
 while ((line = reader.readLine()) != null) {
 writer.write(line);
 writer.newLine();
 }
}
```

## 17. How do you delete a directory in Java?

**Answer:** Use the `delete()` method, but the directory must be empty.

```
File directory = new File("myFolder");
if (directory.delete()) {
 System.out.println("Directory deleted");
} else {
 System.out.println("Failed to delete directory");
}
```

## 18. What is the difference between `writeObject()` and `readObject()`?

**Answer:**

- `writeObject()`: Serializes an object to an output stream.
- `readObject()`: Deserializes an object from an input stream.

### Example:

```
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("object.ser"));
out.writeObject(obj);

ObjectInputStream in = new ObjectInputStream(new
FileInputStream("object.ser"));
Object obj = in.readObject();
```

## Generics in Java

### Topic: Generics in Java

Generics in Java allow you to write **type-safe and reusable code** by enabling classes, interfaces, and methods to operate on a specific type while maintaining type safety at compile time.

## 1. Type Parameters and Bounded Types

### *Type Parameters*

- Represent generic types in classes, interfaces, and methods.
- Common type parameter names:
  - **T**: Type.
  - **E**: Element (used in collections).
  - **K**: Key.
  - **V**: Value.

### *Example: Type Parameters*

```
class Box<T> {
 private T value;

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

 public T get() {
 return value;
 }
}

public class Main {
 public static void main(String[] args) {
 Box<String> box = new Box<>();
 box.set("Hello");
 System.out.println(box.get()); // Output: Hello
 }
}
```

### *Bounded Types*

- Restrict the type parameter to a specific type or its subclasses.



### Syntax

<T extends SuperClass>

### Example: Bounded Types

```
class NumericBox<T extends Number> {
 private T value;

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

 public T get() {
 return value;
 }
}

public class Main {
 public static void main(String[] args) {
 NumericBox<Integer> intBox = new NumericBox<>();
 intBox.set(10);
 System.out.println(intBox.get()); // Output: 10
 }
}
```

## 2. Wildcards: ?, ? extends, ? super

Wildcards are used when you don't know the exact type of the generic parameter.

### ? (Unbounded Wildcard)

- Accepts any type.
- Example:
- ```
public static void printList(List<?> list) {
```
- ```
 for (Object element : list) {
```
- ```
        System.out.println(element);
```
- ```
 }
```
- ```
}
```

? extends Type (Upper Bounded Wildcard)

- Accepts Type or any subclass of Type.
- Example:
- ```
public static void printNumbers(List<? extends Number> list) {
```
- ```
    for (Number num : list) {
```
- ```
 System.out.println(num);
```
- ```
    }
```
- ```
}
```

### ? super Type (Lower Bounded Wildcard)

- Accepts Type or any superclass of Type.
- Example:
- ```
public static void addNumbers(List<? super Integer> list) {
```
- ```
 list.add(10);
```
- ```
    list.add(20);
```

- }

Example Combining Wildcards

```
import java.util.*;

public class WildcardExample {
    public static void main(String[] args) {
        List<Integer> integers = Arrays.asList(1, 2, 3);
        List<Double> doubles = Arrays.asList(1.1, 2.2);

        printNumbers(integers); // Accepts List<? extends Number>
        printNumbers(doubles);  // Accepts List<? extends Number>
    }

    public static void printNumbers(List<? extends Number> list) {
        for (Number num : list) {
            System.out.println(num);
        }
    }
}
```

3. Generic Classes and Methods

Generic Classes

A generic class is a class with type parameters.

```
class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) {
        Pair<String, Integer> pair = new Pair<>("Age", 25);
        System.out.println(pair.getKey() + ": " + pair.getValue()); //
Output: Age: 25
    }
}
```

Generic Methods

A generic method is a method that defines its own type parameters.

Syntax

`<T> ReturnType methodName(T param)`

Example: Generic Method

```
class Utility {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.println(element);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        String[] names = {"Alice", "Bob"};
        Integer[] numbers = {1, 2, 3};

        Utility.printArray(names); // Output: Alice, Bob
        Utility.printArray(numbers); // Output: 1, 2, 3
    }
}
```

4. Generics and Inheritance

- Generic classes do not support polymorphism in the same way as regular classes.
- Example:

```
List<Object> objList = new ArrayList<Object>();
List<String> strList = new ArrayList<String>();
// objList = strList; // Compilation Error
```

5. Advantages of Generics

1. **Type Safety:** Compile-time checking reduces runtime errors.
2. **Reusability:** Code works with different types without duplication.
3. **Elimination of Type Casting:** No explicit type casting required.
4.

```
List<String> list = new ArrayList<>();
```
5.

```
list.add("Hello");
```
6.

```
String s = list.get(0); // No casting required
```

Q&A

1. What are Generics in Java?

Answer: Generics are a feature in Java that allow you to write **type-safe and reusable code**. They enable classes, interfaces, and methods to operate on specific types while maintaining type safety at compile time.

2. What is the syntax for declaring a generic class?

Answer:

```
class ClassName<T> {  
    private T value;  
  
    public void set(T value) {  
        this.value = value;  
    }  
  
    public T get() {  
        return value;  
    }  
}
```

Example:

```
class Box<T> {  
    private T value;  
  
    public void set(T value) {  
        this.value = value;  
    }  
  
    public T get() {  
        return value;  
    }  
}
```

3. What is the difference between `?`, `? extends`, and `? super`?

Answer:

Wildcard	Description	Example
<code>?</code>	Represents an unknown type.	<code>List<?> list</code>
<code>? extends T</code>	Represents <code>T</code> or any of its subclasses.	<code>List<? extends Number></code>
<code>? super T</code>	Represents <code>T</code> or any of its superclasses.	<code>List<? super Integer></code>

4. Can you create an array of a generic type?

Answer: No, you cannot create an array of a generic type directly because of **type erasure**.

Example:

```
// Not allowed  
T[] array = new T[10]; // Compilation error
```

However, you can use a workaround:

```
@SuppressWarnings("unchecked")  
T[] array = (T[]) new Object[10];
```

5. What is a bounded type in Generics?

Answer: A bounded type restricts the types that can be used as arguments for a generic parameter.

Example:

```
class NumericBox<T extends Number> {  
    private T value;  
  
    public void set(T value) {  
        this.value = value;  
    }  
  
    public T get() {  
        return value;  
    }  
}
```

Here, T can only be a subclass of Number (e.g., Integer, Double).

6. What are the benefits of using Generics in Java?

Answer:

1. **Type Safety:** Ensures compile-time type checking.
2. **Eliminates Type Casting:** No need for explicit casting.
3. **Reusability:** Code can work with different types.
4. **Readability:** Makes the code more readable and maintainable.

7. What is the purpose of the `@FunctionalInterface` annotation in Generics?

Answer:

- It marks an interface as a **functional interface**, ensuring that the interface has exactly one abstract method.
- Useful when working with lambda expressions and streams.

Example:

```
@FunctionalInterface  
interface Calculator<T> {  
    T calculate(T a, T b);  
}
```

8. What happens to Generics at runtime?

Answer:

- Generics are **removed at runtime** due to **type erasure**.
- The type parameter is replaced with its bound (`Object` if unbounded) at runtime.

Example:

```
List<String> list = new ArrayList<>();  
System.out.println(list instanceof List); // true
```

9. Can you overload a generic method?

Answer: Yes, you can overload a generic method, but the type parameters must differ.

Example:

```
class Example {  
    public <T> void print(T value) {  
        System.out.println(value);  
    }  
  
    public <T, U> void print(T value, U other) {  
        System.out.println(value + " " + other);  
    }  
}
```

10. Can you use Generics with static methods?

Answer: Yes, static methods can use generics, but they need their own type parameters.

Example:

```
class Utility {  
    public static <T> void print(T value) {  
        System.out.println(value);  
    }  
}  
  
Utility.print("Hello, Generics!"); // Output: Hello, Generics!
```

11. What is a generic method?

Answer: A method that defines its own type parameter(s).

Example:

```
public <T> void printArray(T[] array) {  
    for (T element : array) {  
        System.out.println(element);  
    }  
}
```

12. What is type erasure in Generics?

Answer: Type erasure is the process by which generic types are replaced with their bounds or `Object` during runtime. This ensures backward compatibility with earlier versions of Java.

Example:

```
List<String> list = new ArrayList<>();  
// At runtime, it becomes  
List list = new ArrayList();
```

13. What is the difference between `List<Object>` and `List<?>`?

Answer:

Feature	<code>List<Object></code>	<code>List<?></code>
Type Restriction	Accepts only <code>Object</code> type or its subtypes explicitly.	Accepts any type.
Use Case	When working with specific <code>Object</code> elements.	When working with unknown types.

14. Can Generics be used with primitives?

Answer: No, generics work only with objects, not primitives. Use wrapper classes like `Integer`, `Double`, etc.

Example:

```
List<Integer> list = new ArrayList<>();
```

15. How do you create a generic class with multiple type parameters?

Answer: You can use multiple type parameters separated by commas.

Example:

```
class Pair<K, V> {  
    private K key;  
    private V value;  
  
    public Pair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    public K getKey() {
```

```

        return key;
    }

    public V getValue() {
        return value;
    }
}

```

16. What is the difference between `? extends T` and `? super T`?

Answer:

Feature	<code>? extends T</code>	<code>? super T</code>
Type Restriction	Accepts <code>T</code> and its subtypes.	Accepts <code>T</code> and its supertypes.
Usage	Used for reading elements.	Used for adding elements.

17. Can you define a bounded type parameter with multiple bounds?

Answer: Yes, you can define multiple bounds using `&`.

Example:

```

<T extends Number & Comparable<T>> T findMax(T a, T b) {
    return a.compareTo(b) > 0 ? a : b;
}

```

18. What is a wildcard in Generics?

Answer: A wildcard (`?`) represents an unknown type in generics. It allows flexibility when working with different generic types.

19. Why are Generics not reified in Java?

Answer: Generics are not reified (i.e., retained at runtime) to maintain backward compatibility with older versions of Java.

20. What are some common use cases of Generics in Java?

Answer:

1. Collections Framework (`List<T>`, `Map<K, V>`).
2. Custom reusable classes like `Pair<K, V>`.
3. Generic utility methods like `Collections.sort()`.
4. Type-safe data structures.

Annotations in Java.

Topic: Annotations in Java

Annotations are metadata that provide information about the code but do not directly affect program execution. They are used for:

- 1. Compiler instructions.
- 2. Runtime processing.
- 3. Code documentation.

1. Built-In Annotations

1.1. Annotations for Compiler

- **@Override:** Ensures that the method is overriding a method from its superclass.
- ```
class Parent {
 void display() {
 System.out.println("Parent");
 }
}
```
- ```
class Child extends Parent {  
    @Override  
    void display() { // Ensures overriding  
        System.out.println("Child");  
    }  
}
```
- **@SuppressWarnings:** Suppresses specific compiler warnings.
- ```
@SuppressWarnings("unchecked")
List list = new ArrayList();
```
- **@Deprecated:** Marks a method or class as deprecated.
- ```
@Deprecated  
void oldMethod() {  
    System.out.println("This method is deprecated");  
}
```

1.2. Meta-Annotations

Meta-annotations are used to define annotations. Key meta-annotations:

Meta-Annotation	Description
@Retention	Specifies how long annotations are retained (e.g., runtime).
@Target	Specifies where the annotation can be applied (e.g., methods).

@Documented	Marks an annotation for inclusion in Javadoc.
@Inherited	Allows annotations to be inherited by subclasses.

Example:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME) // Annotation available at runtime
@Target(ElementType.METHOD)        // Can only be applied to methods
@interface CustomAnnotation {
    String value();
}
```

2. Custom Annotations

You can define your own annotations using the `@interface` keyword.

Example: Custom Annotation

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME) // Retained at runtime
@Target(ElementType.METHOD)        // Can be applied to methods
@interface Info {
    String author();
    String version();
}

public class CustomAnnotationExample {
    @Info(author = "John Doe", version = "1.0")
    public void myMethod() {
        System.out.println("My custom annotated method");
    }
}
```

3. Retention Policies

Specifies how long annotations are retained.

Retention Policy	Description
SOURCE	Retained only in the source code.
CLASS	Retained in the <code>.class</code> file but not available at runtime.
RUNTIME	Available at runtime via reflection.

4. Target Types

Specifies where an annotation can be applied.

Target Type	Description
<code>ElementType.METHOD</code>	Applied to methods.
<code>ElementType.FIELD</code>	Applied to fields or attributes.

<code>ElementType.TYPE</code>	Applied to classes, interfaces, or enums.
<code>ElementType.PARAMETER</code>	Applied to method parameters.

5. Processing Annotations

You can process annotations at runtime using **reflection**.

Example: Processing Custom Annotation

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Info {
    String author();
    String version();
}

public class AnnotationProcessor {
    @Info(author = "Alice", version = "2.0")
    public void annotatedMethod() {
        System.out.println("Method with custom annotation");
    }

    public static void main(String[] args) throws Exception {
        Method method =
            AnnotationProcessor.class.getMethod("annotatedMethod");
        Info info = method.getAnnotation(Info.class);

        if (info != null) {
            System.out.println("Author: " + info.author());
            System.out.println("Version: " + info.version());
        }
    }
}
```

Output:

```
Author: Alice
Version: 2.0
```

6. Commonly Used Annotations in Java

6.1. JUnit Testing Annotations

- **@Test:** Marks a test method.
- **@BeforeEach:** Runs before each test method.
- **@AfterEach:** Runs after each test method.

Example:

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

```

public class TestExample {
    @BeforeEach
    void setup() {
        System.out.println("Setting up...");
    }

    @Test
    void testMethod() {
        System.out.println("Running test...");
    }

    @AfterEach
    void teardown() {
        System.out.println("Tearing down...");
    }
}

```

6.2. Spring Framework Annotations

- **@Controller:** Marks a class as a controller in MVC.
- **@Service:** Marks a class as a service.
- **@Autowired:** Injects a dependency automatically.

Example:

```

@Service
public class MyService {
    public void serve() {
        System.out.println("Service method");
    }
}

@Controller
public class MyController {
    @Autowired
    private MyService service;

    public void control() {
        service.serve();
    }
}

```

7. Best Practices

1. **Use @Override when overriding methods:**
 - Ensures correctness and readability.
 - Example:

```

@Override
public String toString() {
    return "Custom toString";
}

```
2. **Document Annotations:**
 - Use @Documented to include annotations in the Javadoc.
3. **Choose Appropriate Retention Policy:**
 - Use `RUNTIME` for runtime processing.

- Use `SOURCE` for compile-time checks.
4. **Avoid Overusing Annotations:**
- Too many annotations can reduce code readability.

Q&A

1. What are annotations in Java?

Answer: Annotations in Java are metadata that provide information about the code but do not directly affect the program execution. They are used for:

1. Compiler instructions (`@Override`, `@SuppressWarnings`).
2. Runtime processing (custom annotations).
3. Code documentation (`@Deprecated`).

2. What are the built-in annotations in Java?

Answer:

Annotation	Description
<code>@Override</code>	Ensures the method is overriding a superclass method.
<code>@SuppressWarnings</code>	Suppresses specific compiler warnings.
<code>@Deprecated</code>	Marks a method or class as deprecated.

3. What are meta-annotations?

Answer: Meta-annotations are annotations that define other annotations. Common meta-annotations:

Meta-Annotation	Description
<code>@Retention</code>	Specifies how long annotations are retained (e.g., runtime).
<code>@Target</code>	Specifies where an annotation can be applied.
<code>@Documented</code>	Marks an annotation for inclusion in Javadoc.
<code>@Inherited</code>	Allows annotations to be inherited by subclasses.

4. What are the retention policies in Java?

Answer:

Retention Policy	Description
------------------	-------------

SOURCE	Retained only in the source code (not in <code>.class</code>).
CLASS	Retained in the <code>.class</code> file but not available at runtime.
RUNTIME	Retained and available at runtime via reflection.

5. How do you create a custom annotation?

Answer: Use the `@interface` keyword.

Example:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Info {
    String author();
    String version();
}
```

6. How do you process annotations at runtime?

Answer: You can use reflection to process annotations.

Example:

```
Method method = MyClass.class.getMethod("myMethod");
Info info = method.getAnnotation(Info.class);
System.out.println(info.author());
System.out.println(info.version());
```

7. What is the purpose of `@Override`?

Answer:

- Ensures that the annotated method is overriding a method from its superclass.
- Helps catch errors at compile time.

Example:

```
class Parent {
    void display() {
        System.out.println("Parent");
    }
}

class Child extends Parent {
    @Override
    void display() { // Ensures correct overriding
        System.out.println("Child");
    }
}
```

8. What is the purpose of `@Deprecated`?

Answer:

- Marks a method, class, or field as outdated or unsafe.
- The compiler generates a warning when it is used.

Example:

```
@Deprecated
void oldMethod() {
    System.out.println("This method is deprecated");
}
```

9. What is the difference between `@Documented` and `@Retention`?

Answer:

Feature	@Documented	@Retention
Purpose	Marks an annotation to be included in Javadoc.	Specifies how long annotations are retained.
Example Use Case	Used for documentation annotations.	Used to define runtime or compile-time retention.

10. What is the purpose of the `@Target` annotation?

Answer: Specifies where an annotation can be applied.

Example:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@interface MethodOnly {}
```

The `@Target(ElementType.METHOD)` restricts this annotation to methods only.

11. What are the possible `ElementType` values in `@Target`?

Answer:

ElementType	Description
TYPE	Classes, interfaces, or enums.
FIELD	Fields or attributes.
METHOD	Methods.

PARAMETER	Method parameters.
CONSTRUCTOR	Constructors.
ANNOTATION_TYPE	Other annotations.

12. What is @Inherited in Java?

Answer: @Inherited allows an annotation to be inherited by subclasses.

Example:

```
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface Info {}

@Info
class Parent {}

class Child extends Parent {}
```

Here, Child inherits the @Info annotation from Parent.

13. Can annotations have default values?

Answer: Yes, annotations can have default values.

Example:

```
@interface Info {
    String author() default "Unknown";
    String version() default "1.0";
}
```

14. What is the difference between @Retention(RetentionPolicy.RUNTIME) and @Retention(RetentionPolicy.CLASS)?

Answer:

Retention Policy	Description
RUNTIME	Annotation is retained at runtime and available via reflection.
CLASS	Annotation is retained in the .class file but not available at runtime.

15. What is the purpose of @FunctionalInterface?

Answer:

- Ensures that the interface has exactly one abstract method.
- Used with lambda expressions.

Example:

```
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}
```

16. What is the difference between annotations and comments?

Answer:

Feature	Annotations	Comments
Purpose	Provide metadata for code.	Provide explanations for developers.
Impact	Processed by compiler and JVM.	Ignored by compiler and JVM.

17. What is the `@SuppressWarnings` annotation?

Answer: Suppresses specific compiler warnings.

Example:

```
@SuppressWarnings("unchecked")
List list = new ArrayList();
```

18. Can you apply multiple annotations to the same element?

Answer: Yes, using **repeating annotations** or arrays.

Example:

```
@Info(author = "Alice", version = "1.0")
@Info(author = "Bob", version = "2.0")
```

19. Can you use annotations in runtime frameworks like Spring?

Answer: Yes, annotations like `@Controller`, `@Service`, and `@Autowired` are extensively used in Spring for dependency injection and MVC.

Example:

```
@Service
class MyService {}

@Controller
```

```
class MyController {
    @Autowired
    private MyService service;
}
```

20. What are some real-world use cases of annotations?

Answer:

1. **JPA:** @Entity, @Id, @Table for ORM.
2. **Spring:** @Autowired, @Component, @RestController.
3. **JUnit:** @Test, @BeforeEach, @AfterEach.
4. **Serialization:** @JsonProperty, @JsonIgnore (Jackson library).

Design Patterns in Core Java

Topic Design patterns

Design patterns are **proven solutions** to common software design problems. They provide a **structured approach** to writing scalable and maintainable code.

1. Singleton Pattern

The **Singleton Pattern** ensures that only **one instance** of a class is created and provides a global access point.

Implementation Approaches

1.1. Eager Initialization (Thread-Safe)

- The instance is created at class loading.
- **Drawback:** May create an instance even if it's never used.

```
class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {} // Private constructor

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

1.2. Lazy Initialization (Not Thread-Safe)

- Creates an instance only when requested.
- **Drawback:** Not thread-safe in multi-threaded environments.

```

class Singleton {
    private static Singleton instance;

    private Singleton() {}

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

```

1.3. Thread-Safe Singleton (Double-Checked Locking)

- Uses **synchronized** and **volatile** for thread safety.

```

class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}

```

2. Factory Pattern

The **Factory Pattern** provides an interface for creating objects but **delegates instantiation** to subclasses.

Example

```

// Step 1: Create an interface
interface Vehicle {
    void drive();
}

// Step 2: Implement concrete classes
class Car implements Vehicle {
    public void drive() {
        System.out.println("Driving a Car.");
    }
}

class Bike implements Vehicle {
    public void drive() {
        System.out.println("Riding a Bike.");
    }
}

```

```
// Step 3: Create a Factory Class
class VehicleFactory {
    public static Vehicle getVehicle(String type) {
        if ("car".equalsIgnoreCase(type)) {
            return new Car();
        } else if ("bike".equalsIgnoreCase(type)) {
            return new Bike();
        }
        return null;
    }
}

// Step 4: Client Code
public class FactoryExample {
    public static void main(String[] args) {
        Vehicle vehicle = VehicleFactory.getVehicle("car");
        vehicle.drive(); // Output: Driving a Car.
    }
}
```

3. Builder Pattern

The **Builder Pattern** is used to **construct complex objects** step by step.

Example

```
class Car {
    private String engine;
    private int wheels;
    private boolean sunroof;

    // Private constructor
    private Car(CarBuilder builder) {
        this.engine = builder.engine;
        this.wheels = builder.wheels;
        this.sunroof = builder.sunroof;
    }

    // Static Builder Class
    static class CarBuilder {
        private String engine;
        private int wheels;
        private boolean sunroof;

        public CarBuilder(String engine, int wheels) {
            this.engine = engine;
            this.wheels = wheels;
        }

        public CarBuilder setSunroof(boolean sunroof) {
            this.sunroof = sunroof;
            return this;
        }

        public Car build() {
            return new Car(this);
        }
    }
}
```

```

        @Override
        public String toString() {
            return "Car [engine=" + engine + ", wheels=" + wheels + ",
sunroof=" + sunroof + "]";
        }
    }

    public class BuilderExample {
        public static void main(String[] args) {
            Car car = new Car.CarBuilder("V8", 4).setSunroof(true).build();
            System.out.println(car);
        }
    }

```

Output:

```
Car [engine=V8, wheels=4, sunroof=true]
```

4. Prototype Pattern

The **Prototype Pattern** is used to **create object clones** instead of creating new instances.

Example

```

import java.util.HashMap;
import java.util.Map;

// Step 1: Create the Cloneable Interface
interface Animal extends Cloneable {
    Animal clone();
}

// Step 2: Implement Concrete Classes
class Dog implements Animal {
    public Dog() {
        System.out.println("Dog Created.");
    }

    public Animal clone() {
        return new Dog();
    }
}

class Cat implements Animal {
    public Cat() {
        System.out.println("Cat Created.");
    }

    public Animal clone() {
        return new Cat();
    }
}

// Step 3: Create a Prototype Registry
class AnimalRegistry {
    private static Map<String, Animal> animalMap = new HashMap<>();

    static {

```

```

        animalMap.put("dog", new Dog());
        animalMap.put("cat", new Cat());
    }

    public static Animal getAnimal(String type) {
        return animalMap.get(type).clone();
    }
}

// Step 4: Client Code
public class PrototypeExample {
    public static void main(String[] args) {
        Animal clonedDog = AnimalRegistry.getAnimal("dog");
        Animal clonedCat = AnimalRegistry.getAnimal("cat");
    }
}

```

Output:

```

Dog Created.
Cat Created.
Dog Created.
Cat Created.

```

5. Observer Pattern

The **Observer Pattern** is used when an object (**Subject**) has **multiple observers** that need to be updated when the state changes.

Example

```

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

// Step 1: Create an Observer Interface
interface Observer {
    void update(String message);
}

// Step 2: Implement Concrete Observers
class User implements Observer {
    private String name;

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

    public void update(String message) {
        System.out.println(name + " received message: " + message);
    }
}

// Step 3: Create the Subject Interface
interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(String message);
}

```

```
// Step 4: Implement the Subject
class Channel implements Subject {
    private List<Observer> observers = new ArrayList<>();

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

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

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

// Step 5: Client Code
public class ObserverExample {
    public static void main(String[] args) {
        Channel channel = new Channel();

        Observer user1 = new User("Alice");
        Observer user2 = new User("Bob");

        channel.addObserver(user1);
        channel.addObserver(user2);

        channel.notifyObservers("New Video Uploaded!");
    }
}
```

Output:

```
Alice received message: New Video Uploaded!
Bob received message: New Video Uploaded!
```

Summary of Design Patterns

Pattern	Purpose
Singleton	Ensures only one instance of a class exists.
Factory	Encapsulates object creation logic.
Builder	Constructs complex objects step by step .
Prototype	Clones objects instead of creating new instances.
Observer	Used in event-driven systems (e.g., event listeners).

Q&A

1. What is the Singleton Pattern, and when should you use it?

Answer:

- **Definition:** The **Singleton Pattern** ensures that a class has **only one instance** and provides a **global access point** to it.
- **Use Cases:**
 1. Database connections.
 2. Logger classes.
 3. Configuration managers.
 4. Caching mechanisms.

Example (Thread-Safe Singleton):

```
class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

2. What are different ways to implement the Singleton Pattern?

Answer:

1. **Eager Initialization** (Creates instance at class loading).

```
2. class Singleton {
3.     private static final Singleton INSTANCE = new Singleton();
4.     private Singleton() {}
5.     public static Singleton getInstance() { return INSTANCE; }
6. }
```

7. **Lazy Initialization** (Creates instance only when needed, **not thread-safe**).

```
8. class Singleton {
9.     private static Singleton instance;
10.    private Singleton() {}
11.    public static Singleton getInstance() {
12.        if (instance == null) {
13.            instance = new Singleton();
14.        }
15.        return instance;
16.    }
17. }
```

18. **Thread-Safe Singleton using Double-Checked Locking** (Best practice).

```
19. class Singleton {
20.     private static volatile Singleton instance;
21.     private Singleton() {}
22.     public static Singleton getInstance() {
23.         if (instance == null) {
24.             synchronized (Singleton.class) {
25.                 if (instance == null) {
```



```

26.             instance = new Singleton();
27.         }
28.     }
29. }
30.     return instance;
31. }
32. }

```

33. Bill Pugh Singleton (Best performance, avoids synchronization overhead).

```

34. class Singleton {
35.     private Singleton() {}
36.     private static class Holder {
37.         private static final Singleton INSTANCE = new
Singleton();
38.     }
39.     public static Singleton getInstance() {
40.         return Holder.INSTANCE;
41.     }
42. }

```

3. What is the Factory Pattern, and why is it useful?

Answer:

- **Definition:** The **Factory Pattern** provides an interface for **creating objects** but allows subclasses to decide which class to instantiate.
- **Use Cases:**
 1. When object creation is **complex**.
 2. When the exact type of object is **unknown at compile time**.

Example:

```

interface Vehicle {
    void drive();
}

class Car implements Vehicle {
    public void drive() {
        System.out.println("Driving a Car.");
    }
}

class VehicleFactory {
    public static Vehicle getVehicle(String type) {
        if ("car".equalsIgnoreCase(type)) {
            return new Car();
        }
        return null;
    }
}

public class FactoryExample {
    public static void main(String[] args) {
        Vehicle car = VehicleFactory.getVehicle("car");
        car.drive(); // Output: Driving a Car.
    }
}

```

4. What is the difference between Factory and Builder patterns?

Feature	Factory Pattern	Builder Pattern
Purpose	Encapsulates object creation logic.	Builds complex objects step by step.
Complexity	Good for simple objects.	Used when many attributes are optional.
Flexibility	Uses a single method (<code>getInstance()</code>).	Uses a step-by-step approach.

5. What is the Builder Pattern, and when should you use it?

Answer:

- **Definition:** The **Builder Pattern** is used for **constructing complex objects step by step**.
- **Use Cases:**
 1. When a class has **many optional parameters**.
 2. When you want **immutable objects**.

Example:

```
class Car {
    private String engine;
    private int wheels;
    private boolean sunroof;

    private Car(CarBuilder builder) {
        this.engine = builder.engine;
        this.wheels = builder.wheels;
        this.sunroof = builder.sunroof;
    }

    static class CarBuilder {
        private String engine;
        private int wheels;
        private boolean sunroof;

        public CarBuilder(String engine, int wheels) {
            this.engine = engine;
            this.wheels = wheels;
        }

        public CarBuilder setSunroof(boolean sunroof) {
            this.sunroof = sunroof;
            return this;
        }

        public Car build() {
```

```

        return new Car(this);
    }
}

```

Usage:

```

Car car = new Car.CarBuilder("V8", 4).setSunroof(true).build();
System.out.println(car);

```

6. What is the Prototype Pattern?

Answer:

- **Definition:** The **Prototype Pattern** is used to create **object clones** instead of instantiating new objects.
- **Use Cases:**
 1. When object creation is **expensive**.
 2. When multiple instances of similar objects are needed.

Example:

```

interface Animal extends Cloneable {
    Animal clone();
}

class Dog implements Animal {
    public Dog() {
        System.out.println("Dog Created.");
    }

    public Animal clone() {
        return new Dog();
    }
}

```

Usage:

```

Animal clonedDog = new Dog().clone();

```

7. What is the Observer Pattern?

Answer:

- **Definition:** The **Observer Pattern** is used when one object (**Subject**) notifies multiple **Observers** about changes.
- **Use Cases:**
 1. Event-driven systems (e.g., notifications, UI event listeners).
 2. Publish-subscribe systems.

Example:

```
import java.util.*;

interface Observer {
    void update(String message);
}

class User implements Observer {
    private String name;
    public User(String name) {
        this.name = name;
    }

    public void update(String message) {
        System.out.println(name + " received message: " + message);
    }
}

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

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

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

public class ObserverExample {
    public static void main(String[] args) {
        Channel channel = new Channel();
        Observer user1 = new User("Alice");
        Observer user2 = new User("Bob");

        channel.addObserver(user1);
        channel.addObserver(user2);

        channel.notifyObservers("New Video Uploaded!");
    }
}
```

Output:

Alice received message: New Video Uploaded!
Bob received message: New Video Uploaded!

8. What is the difference between Singleton and Prototype Patterns?

Feature	Singleton Pattern	Prototype Pattern
Purpose	Ensures only one instance exists.	Creates multiple clones of objects.
Object	Uses lazy initialization or eager	Uses cloning (<code>clone()</code>).

Creation	loading.	
----------	----------	--

9. What are some real-world examples of these patterns?

Answer:

Pattern	Real-World Example
Singleton	Logger, Database Connection.
Factory	<code>java.sql.DriverManager.getConnection()</code> .
Builder	<code>StringBuilder</code> in Java.
Prototype	Clone feature in apps.
Observer	UI Event Listeners, Notifications.

10. Which design pattern is used in the Java Collections Framework?

Answer:

- **Factory Pattern:** `Collections.synchronizedList(List<T>)`
- **Singleton Pattern:** `Runtime.getRuntime()`
- **Builder Pattern:** `StringBuilder`
- **Observer Pattern:** `ActionListener` in Swing

Summary

These patterns help in **structuring code efficiently**, improving **scalability**, and **reducing redundancy**.

Memory Management in Java

Topic: Memory Management in Java

Memory management in Java is handled **automatically** by the **JVM (Java Virtual Machine)** using **Garbage Collection (GC)**. This ensures that **unused objects are removed**, preventing memory leaks.

1. Garbage Collection Basics

Garbage Collection (GC) automatically **reclaims memory** occupied by objects **no longer in use**.

How Garbage Collection Works

1. **An object becomes unreachable** when there are **no references** pointing to it.
2. **GC identifies unreachable objects** and removes them.
3. **GC runs automatically**, but can be **suggested** using `System.gc()`. (*Not guaranteed to execute immediately*).

Example: Object Becomes Eligible for GC

```
class GarbageCollectionExample {
    public static void main(String[] args) {
        GarbageCollectionExample obj1 = new GarbageCollectionExample();
        GarbageCollectionExample obj2 = new GarbageCollectionExample();

        obj1 = null; // Eligible for garbage collection
        obj2 = null; // Eligible for garbage collection

        System.gc(); // Requesting GC (not guaranteed)
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("Garbage Collector is running...");
    }
}
```

Output (Not deterministic):

```
Garbage Collector is running...
Garbage Collector is running...
```

2. Types of References in Java

Java provides **four types of references** in `java.lang.ref` package:

2.1. Strong References (Default)

- Objects **are not eligible** for garbage collection **unless** explicitly set to `null`.
- **Example:**
- `String strongRef = "Hello";`
- `strongRef = null; // Now eligible for GC`

2.2. Soft References (For Caching)

- Objects **are eligible** for GC **only if memory is low**.
- Useful for **caching** data (e.g., image caching).
- **Example:**
- `import java.lang.ref.*;`
-
- `public class SoftReferenceExample {`
- `public static void main(String[] args) {`
- `SoftReference<String> softRef = new SoftReference<>(new`
- `String("Soft Reference"));`

- `System.out.println(softRef.get()); // Output: Soft Reference`
- `System.gc();`
- `System.out.println(softRef.get()); // Output: Might be null if memory is low`
- `}`
- `}`

2.3. Weak References (GC Collects Anytime)

- Objects **are eligible** for garbage collection **immediately** when no strong references exist.
- Used in **WeakHashMap** (removes unused keys).
- **Example:**
- `import java.lang.ref.*;`
-
- `public class WeakReferenceExample {`
- `public static void main(String[] args) {`
- `WeakReference<String> weakRef = new WeakReference<>(new String("Weak Reference"));`
- `System.out.println(weakRef.get()); // Output: Weak Reference`
- `System.gc();`
- `System.out.println(weakRef.get()); // Output: null (most likely)`
- `}`
- `}`

2.4. Phantom References (Used for Cleanup Tasks)

- Objects **are collected but still tracked** before finalization.
- Used for **monitoring object finalization** (e.g., **cleaning resources**).
- **Example:**
- `import java.lang.ref.*;`
-
- `public class PhantomReferenceExample {`
- `public static void main(String[] args) {`
- `ReferenceQueue<String> queue = new ReferenceQueue<>();`
- `PhantomReference<String> phantomRef = new PhantomReference<>(new String("Phantom"), queue);`
-
- `System.out.println(phantomRef.get()); // Output: null (Phantom references return null)`
- `}`
- `}`

3. JVM Memory Model

The JVM divides memory into **several areas**:

3.1. Heap Memory

- Stores **objects** and **class instances**.
- Divided into:
 - **Young Generation** (short-lived objects)
 - **Old Generation (Tenured Space)** (long-lived objects)
 - **Permanent Generation (Metaspace in Java 8+)** (stores metadata, class definitions)

3.2. Stack Memory

- Stores **method calls, local variables, and references**.
- Each thread has its **own stack**.

3.3. Method Area (MetaSpace)

- Stores **class metadata, static variables, and method bytecode**.

JVM Memory Layout

Memory Area	Stores
Heap	Objects, Class Instances
Stack	Method Calls, Local Variables
Method Area	Class Metadata, Static Variables
PC Register	Current thread execution line
Native Method Stack	Native (JNI) method execution

4. Garbage Collection in Depth

Garbage collectors automatically clean up unused objects.

Types of Garbage Collectors

Garbage Collector	Description
Serial GC	Single-threaded, for small applications.
Parallel GC	Uses multiple threads for GC (default in Java 8).
CMS (Concurrent Mark-Sweep)	Runs alongside application threads, reduces pause times.
G1 (Garbage First)	Default in Java 9+, replaces CMS, improves latency.

How to Select a Garbage Collector

Use JVM arguments:

```
# Serial GC (For small applications)
-XX:+UseSerialGC

# Parallel GC (For multi-threaded environments)
-XX:+UseParallelGC
```



```
# CMS GC (For low-latency applications)
-XX:+UseConcMarkSweepGC

# G1 GC (For large heaps, Java 9+)
-XX:+UseG1GC
```

5. Best Practices for Memory Management

1. **Use Weak References** where appropriate (e.g., `WeakHashMap` for caching).
2. **Avoid Memory Leaks** by explicitly **removing unused objects**.
3. **Prefer Try-With-Resources for Cleanup:**
4.

```
try (BufferedReader br = new BufferedReader(new
    FileReader("file.txt"))) {
```
5.

```
    System.out.println(br.readLine());
```
6.

```
}
```
7. **Monitor Memory Usage:**
8.

```
Runtime runtime = Runtime.getRuntime();
```
9.

```
System.out.println("Free Memory: " + runtime.freeMemory());
```
10. **Use Profiling Tools:**
 - o **VisualVM** (`jvisualvm`)
 - o **Eclipse Memory Analyzer (MAT)**
 - o **JConsole**

Q&A

1. What is Garbage Collection (GC) in Java?

Answer: Garbage Collection (GC) is a process by which the JVM **automatically reclaims memory** occupied by objects **no longer in use**, preventing memory leaks.

2. How does Java identify objects for garbage collection?

Answer: Java identifies objects for garbage collection when **they have no strong references**. If an object:

1. **Has no active references**, it becomes eligible for GC.
2. **Goes out of scope**, it becomes unreachable.
3. **Is explicitly set to `null`**, it becomes unreachable.

Example:

```
String str = new String("Hello");
str = null; // Now eligible for garbage collection
System.gc();
```

3. What are the different types of references in Java?

Answer:

Reference Type	Description
Strong Reference	Default, objects are not garbage collected unless explicitly set to null.
Soft Reference	Garbage collected only when JVM is running low on memory (Used for caching).
Weak Reference	Garbage collected immediately when no strong references exist.
Phantom Reference	Used for finalization tasks , GC removes objects but keeps a reference in a queue.

Example (Weak Reference):

```
import java.lang.ref.WeakReference;

public class WeakReferenceExample {
    public static void main(String[] args) {
        WeakReference<String> weakRef = new WeakReference<>(new
String("Weak Reference"));
        System.out.println(weakRef.get()); // Output: Weak Reference
        System.gc();
        System.out.println(weakRef.get()); // Output: null (most likely)
    }
}
```

4. What is the difference between Stack and Heap memory in Java?

Answer:

Feature	Stack Memory	Heap Memory
Purpose	Stores method calls and local variables.	Stores objects and class instances.
Size	Smaller and limited.	Larger than stack.
Lifetime	Short-lived, removed when method exits.	Objects live until GC removes them.
Access Speed	Faster	Slower

Example:

```
void method() {
    int a = 10; // Stored in stack
    String str = new String("Heap Object"); // Stored in heap
}
```

5. What is the difference between Young Generation and Old Generation in Heap?

Answer:

Memory Area	Description
Young Generation	Stores new objects (Most objects die here quickly).
Old Generation (Tenured Space)	Stores long-lived objects .
Metaspace (Java 8+)	Stores class metadata, static variables .

6. How can you request garbage collection in Java?

Answer: You can **suggest** garbage collection, but JVM may ignore it:

```
System.gc(); // Suggest GC (not guaranteed)
Runtime.getRuntime().gc(); // Alternative way
```

However, these methods are not recommended for production use.

7. What are the different types of Garbage Collectors in Java?

Answer:

Garbage Collector	Description
Serial GC	Single-threaded, for small applications.
Parallel GC	Uses multiple threads for GC (default in Java 8).
CMS (Concurrent Mark-Sweep)	Runs alongside application threads, reduces pause times.
G1 (Garbage First)	Default in Java 9+, improves low-latency performance .

8. How do you specify a garbage collector in Java?

Answer: Use JVM arguments:

```
# Serial GC (For small applications)
-XX:+UseSerialGC

# Parallel GC (For multi-threaded environments)
-XX:+UseParallelGC

# CMS GC (For low-latency applications)
-XX:+UseConcMarkSweepGC

# G1 GC (For large heaps, Java 9+)
-XX:+UseG1GC
```

9. What is `finalize()` method, and why is it discouraged?

Answer:

- The `finalize()` method is called before an object is garbage collected.
- **It is discouraged because:**
 1. **Unreliable** – JVM may never call it.
 2. **Performance overhead** – Slows down GC.
 3. **Deprecated in Java 9+.**

Example (Not recommended):

```
@Override
protected void finalize() throws Throwable {
    System.out.println("Finalizing...");
}
```

Instead, use:

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt")))
{
    System.out.println(br.readLine());
} // Auto-closes
```

10. What is a memory leak in Java?

Answer: A **memory leak** occurs when objects **are not garbage collected** due to lingering references.

Example:

```
import java.util.*;

public class MemoryLeakExample {
    private static List<int[]> list = new ArrayList<>();

    public static void main(String[] args) {
        while (true) {
            list.add(new int[1000000]); // Memory leak (keeps adding
objects)
        }
    }
}
```

Solution:

- Use **Weak References** for caches.
- Explicitly **remove objects** from collections.

11. How does `WeakHashMap` help in memory management?

Answer:

- WeakHashMap removes keys **automatically** when there are no strong references.
- Helps prevent **memory leaks** in caching scenarios.

Example:

```
import java.util.*;

public class WeakHashMapExample {
    public static void main(String[] args) {
        Map<Object, String> weakMap = new WeakHashMap<>();
        Object key = new Object();
        weakMap.put(key, "Value");

        key = null; // Remove strong reference
        System.gc(); // Now key-value pair is eligible for GC
        System.out.println(weakMap.size()); // Output: 0 (Most likely)
    }
}
```

12. How do you monitor memory usage in Java?

Answer: Use Runtime class:

```
Runtime runtime = Runtime.getRuntime();
System.out.println("Free Memory: " + runtime.freeMemory());
System.out.println("Total Memory: " + runtime.totalMemory());
```

Use **profiling tools**:

- **VisualVM** (`jvisualvm`)
- **Eclipse Memory Analyzer (MAT)**
- **JConsole** (`jconsole`)

13. What is OutOfMemoryError (OOM) in Java?

Answer: OOM occurs when JVM runs out of heap memory.

Example:

```
public class OutOfMemoryExample {
    public static void main(String[] args) {
        List<int[]> list = new ArrayList<>();
        while (true) {
            list.add(new int[1000000]); // Causes OOM error
        }
    }
}
```

Solution:

1. Increase heap size: `-Xmx512m`

2. Use memory profiling tools.

14. What is the difference between `System.gc()` and `Runtime.getRuntime().gc()`?

Answer: Both **suggest** garbage collection, but neither guarantees execution.

Method	Description
<code>System.gc()</code>	Static method, calls <code>Runtime.getRuntime().gc()</code> .
<code>Runtime.getRuntime().gc()</code>	Directly requests GC via <code>Runtime</code> .

15. How can you avoid memory leaks in Java?

Answer:

1. **Close resources** (`try-with-resources`).
2. **Use Weak References** (`WeakHashMap` for caching).
3. **Avoid static references** (can prevent GC).
4. **Monitor memory** (profiling tools).
5. **Explicitly remove objects** from collections (`list.clear()`).

Summary

Concept	Key Points
Garbage Collection	Automatically removes unused objects.
References	Strong, Soft, Weak, Phantom.
JVM Memory Areas	Heap (Young, Old), Stack, Metaspace.
GC Types	Serial, Parallel, CMS, G1.
Memory Leaks	Caused by lingering references, solved using Weak References.

Reflection and Dynamic Class Loading in Java

Topic: Reflection and Dynamic Class Loading in Java

Reflection in Java allows **introspection** and **modification** of classes, methods, and fields at runtime. This is useful for **frameworks, libraries, and dependency injection**.

1. Reflection API Overview

The **Reflection API** is part of the `java.lang.reflect` package and allows:

1. **Inspecting classes, methods, fields, and constructors at runtime.**
2. **Dynamically invoking methods and modifying fields.**
3. **Accessing private fields and methods.**

Key Reflection Classes

Class	Description
<code>Class<?></code>	Represents a class or interface at runtime.
<code>Method</code>	Represents methods in a class.
<code>Field</code>	Represents fields (variables) in a class.
<code>Constructor<?></code>	Represents constructors in a class.

2. Getting Class Metadata Using Reflection

We can obtain metadata (class name, methods, fields) using reflection.

Example: Getting Class Name and Methods

```
import java.lang.reflect.*;

class Example {
    private int number;
    public void display() {
        System.out.println("Hello, Reflection!");
    }
}

public class ReflectionExample {
    public static void main(String[] args) {
        Class<?> clazz = Example.class; // Get class reference

        System.out.println("Class Name: " + clazz.getName());

        Method[] methods = clazz.getDeclaredMethods(); // Get all
methods
        for (Method method : methods) {
            System.out.println("Method: " + method.getName());
        }
    }
}
```

Output:

```
Class Name: Example
Method: display
```

3. Accessing and Modifying Private Fields and Methods

Java allows access to **private fields and methods** using **reflection**.

Example: Accessing Private Field

```
import java.lang.reflect.*;

class Person {
    private String name = "John Doe";
}

public class PrivateFieldExample {
    public static void main(String[] args) throws Exception {
        Person person = new Person();
        Class<?> clazz = person.getClass();

        Field field = clazz.getDeclaredField("name"); // Access private
field        field.setAccessible(true); // Bypass access restriction

        System.out.println("Before: " + field.get(person));

        field.set(person, "Alice"); // Modify private field

        System.out.println("After: " + field.get(person));
    }
}
```

Output:

Before: John Doe
After: Alice

Example: Invoking a Private Method

```
import java.lang.reflect.*;

class Secret {
    private void secretMessage() {
        System.out.println("This is a private method!");
    }
}

public class PrivateMethodExample {
    public static void main(String[] args) throws Exception {
        Secret obj = new Secret();
        Method method =
obj.getClass().getDeclaredMethod("secretMessage");
        method.setAccessible(true); // Bypass private access
        method.invoke(obj); // Invoke method
    }
}
```

Output:

This is a private method!

4. Creating Objects Dynamically Using Reflection

We can **dynamically instantiate objects** at runtime.

Example: Using `newInstance()`

```
import java.lang.reflect.*;

class DynamicClass {
    public void showMessage() {
        System.out.println("Dynamic Object Created!");
    }
}

public class DynamicObjectExample {
    public static void main(String[] args) throws Exception {
        Class<?> clazz = Class.forName("DynamicClass");
        Object obj = clazz.getDeclaredConstructor().newInstance(); //
        // Create object dynamically
        Method method = clazz.getMethod("showMessage");
        method.invoke(obj);
    }
}
```

Output:

Dynamic Object Created!

5. Dynamic Class Loading

Java allows **loading classes at runtime** using `Class.forName()` and `ClassLoader`.

Example: Dynamic Loading Using `Class.forName()`

```
public class DynamicLoadingExample {
    public static void main(String[] args) throws Exception {
        String className = "java.util.ArrayList"; // Dynamically load
        // ArrayList class
        Class<?> clazz = Class.forName(className);
        System.out.println("Loaded Class: " + clazz.getName());
    }
}
```

Output:

Loaded Class: java.util.ArrayList

Example: Custom Class Loader

```
class CustomClassLoader extends ClassLoader {
    public Class<?> loadClass(String name) throws ClassNotFoundException
    {
        return super.loadClass(name);
    }
}
```

```

    }

    public class CustomClassLoaderExample {
        public static void main(String[] args) throws Exception {
            CustomClassLoader loader = new CustomClassLoader();
            Class<?> clazz = loader.loadClass("java.lang.String");
            System.out.println("Loaded: " + clazz.getName());
        }
    }
}

```

Output:

Loaded: java.lang.String

6. Reflection Performance Considerations

- **Reflection is slower** than direct method calls because:
 1. It **bypasses optimizations** done by the compiler.
 2. It **requires security checks**.
 3. It uses **dynamic lookups**, which are **slower**.

When to Use Reflection

- ✓ **Frameworks** (Spring, Hibernate, JUnit).
- ✓ **Dynamic dependency injection**.
- ✓ **Serialization and deserialization**.
- ✗ **Avoid using reflection in performance-critical applications.**

7. Best Practices for Using Reflection

1. **Minimize Use** – Use reflection only when necessary.
2. **Use `setAccessible(true)` Carefully** – It breaks encapsulation and security.
3. **Cache Reflection Results** – Avoid repeated lookups.
4. **Use `MethodHandle` for Performance** – Instead of reflection, use `java.lang.invoke.MethodHandle` (Java 7+).
5. **Prefer Interfaces Over Reflection** – Use interfaces when possible.

Q&A

1. What is Reflection in Java?

Answer: Reflection is a feature in Java that allows **inspecting and modifying** classes, methods, and fields **at runtime**. It is part of the `java.lang.reflect` package.

Example:

```
Class<?> clazz = Class.forName("java.util.ArrayList");
System.out.println("Class Name: " + clazz.getName());
```

Use Cases:

1. Frameworks (Spring, Hibernate).
 2. Dependency Injection.
 3. Serialization and Deserialization.
-

2. What are the key classes in the Reflection API?

Answer:

Class	Description
Class<?>	Represents a class at runtime.
Method	Represents a method in a class.
Field	Represents a field (variable) in a class.
Constructor<?>	Represents constructors of a class.

3. How do you get a class object in Java?

Answer: Three ways:

1. **Using `forName()`:**
 2. `Class<?> clazz = Class.forName("java.util.ArrayList");`
 3. **Using `.class` property:**
 4. `Class<?> clazz = String.class;`
 5. **Using `getClass()` method:**
 6. `Object obj = new String("Hello");`
 7. `Class<?> clazz = obj.getClass();`
-

4. How do you get the methods of a class using Reflection?

Answer:

```
import java.lang.reflect.*;

class Example {
    public void display() {}
}

public class ReflectionExample {
    public static void main(String[] args) {
        Method[] methods = Example.class.getDeclaredMethods();
        for (Method method : methods) {
            System.out.println("Method: " + method.getName());
        }
    }
}
```

```
    }  
    }  
}
```

Output:

Method: display

5. How do you invoke a method using Reflection?

Answer:

```
import java.lang.reflect.*;  
  
class Example {  
    public void sayHello() {  
        System.out.println("Hello, Reflection!");  
    }  
}  
  
public class InvokeMethodExample {  
    public static void main(String[] args) throws Exception {  
        Example obj = new Example();  
        Method method = Example.class.getMethod("sayHello");  
        method.invoke(obj);  
    }  
}
```

Output:

Hello, Reflection!

6. How do you access private fields using Reflection?

Answer:

```
import java.lang.reflect.*;  
  
class Person {  
    private String name = "John Doe";  
}  
  
public class PrivateFieldExample {  
    public static void main(String[] args) throws Exception {  
        Person person = new Person();  
        Field field = person.getClass().getDeclaredField("name");  
        field.setAccessible(true);  
        System.out.println("Before: " + field.get(person));  
  
        field.set(person, "Alice");  
        System.out.println("After: " + field.get(person));  
    }  
}
```

Output:

Before: John Doe
After: Alice

7. How do you invoke a private method using Reflection?

Answer:

```
import java.lang.reflect.*;

class Secret {
    private void hiddenMessage() {
        System.out.println("This is a private method!");
    }
}

public class PrivateMethodExample {
    public static void main(String[] args) throws Exception {
        Secret obj = new Secret();
        Method method =
obj.getClass().getDeclaredMethod("hiddenMessage");
        method.setAccessible(true);
        method.invoke(obj);
    }
}
```

Output:

This is a private method!

8. How do you dynamically create an object using Reflection?

Answer:

```
import java.lang.reflect.*;

class Example {
    public Example() {
        System.out.println("Object Created!");
    }
}

public class DynamicObjectExample {
    public static void main(String[] args) throws Exception {
        Class<?> clazz = Class.forName("Example");
        Object obj = clazz.getDeclaredConstructor().newInstance();
    }
}
```

Output:

Object Created!

9. What is Dynamic Class Loading in Java?

Answer: Dynamic class loading allows **loading classes at runtime** using:

- 1. **Class.forName("ClassName")** - Loads a class dynamically.
- 2. **Custom Class Loaders** - Loads classes explicitly.

Example:

```
Class<?> clazz = Class.forName("java.util.HashMap");
System.out.println("Loaded Class: " + clazz.getName());
```

10. What is a custom ClassLoader?

Answer: A custom **ClassLoader** extends `ClassLoader` to **load classes dynamically**.

Example:

```
class CustomClassLoader extends ClassLoader {
    public Class<?> loadClass(String name) throws ClassNotFoundException
    {
        return super.loadClass(name);
    }
}

public class CustomClassLoaderExample {
    public static void main(String[] args) throws Exception {
        CustomClassLoader loader = new CustomClassLoader();
        Class<?> clazz = loader.loadClass("java.lang.String");
        System.out.println("Loaded: " + clazz.getName());
    }
}
```

Output:

```
Loaded: java.lang.String
```

11. What are the advantages and disadvantages of Reflection?

Answer:

Advantages	Disadvantages
Flexibility - Can inspect and modify code at runtime.	Slow Performance - Reflection is slower than direct calls.
Used in Frameworks - Spring, Hibernate, JUnit use Reflection.	Security Issues - Can break encapsulation and modify private fields.
No Compile-Time Dependency - Can work with unknown classes.	Complex Debugging - Errors are runtime-based.

12. What are the security risks of using Reflection?

Answer:

1. **Bypasses Encapsulation** - Can modify private fields/methods.
 2. **Exposes Internal Details** - Can reveal sensitive information.
 3. **Security Manager Restriction** - Some environments restrict Reflection.
-

13. What is the difference between `Class.forName()` and `ClassLoader.loadClass()`?

Answer:

Feature	<code>Class.forName()</code>	<code>ClassLoader.loadClass()</code>
Initialization	Initializes the class immediately.	Loads the class without initializing it.
Use Case	When you need to execute static blocks.	When you just want to load the class.

Example:

```
Class<?> clazz1 = Class.forName("java.util.ArrayList"); // Initializes
Class<?> clazz2 =
ClassLoader.getSystemClassLoader().loadClass("java.util.ArrayList"); //
Only loads
```

14. What are some real-world use cases of Reflection?

Answer: ☒ **Frameworks** - Spring, Hibernate, JUnit.
☒ **Dependency Injection** - Loading beans dynamically.
☒ **Serialization & Deserialization** - JSON parsing (Jackson, Gson).
☒ **Class Scanners** - Scanning for annotations.

15. Why is Reflection slow compared to normal method calls?

Answer:

- **Bypasses optimizations** done by the JVM.
- **Requires security checks** before execution.
- **Uses dynamic lookup** instead of direct method calls.

Alternative for Performance: `MethodHandle` (Java 7+)

```
import java.lang.invoke.*;
```

```

class Example {
    public void hello() {
        System.out.println("Hello, MethodHandle!");
    }
}

public class MethodHandleExample {
    public static void main(String[] args) throws Throwable {
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        MethodHandle methodHandle = lookup.findVirtual(Example.class,
"hello", MethodType.methodType(void.class));

        Example obj = new Example();
        methodHandle.invoke(obj);
    }
}

```

Output:

Hello, MethodHandle!

Summary

Concept	Key Takeaways
Reflection	Allows accessing private methods/fields dynamically.
Dynamic Class Loading	Loads classes at runtime (<code>Class.forName()</code>).
Custom ClassLoader	Loads classes explicitly.
Security Risks	Reflection breaks encapsulation.
Performance	Reflection is slow , prefer MethodHandle for better speed.

Topic: Networking in Java

Java provides built-in support for **network programming** through the `java.net` package, allowing applications to communicate over networks using **TCP**, **UDP**, **HTTP**, and other protocols.

1. Sockets: TCP and UDP Communication

A **Socket** is an **endpoint for communication** between two machines.

1.1. TCP (Transmission Control Protocol)

- **Reliable, connection-oriented protocol.**
- Ensures **data delivery and order**.
- Used in **HTTP, FTP, SSH, etc..**

Example: TCP Server and Client

TCP Server

```
import java.io.*;
import java.net.*;

public class TCPServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(5000); // Server
        listening on port 5000
        System.out.println("Server is waiting for client...");

        Socket socket = serverSocket.accept(); // Accept client
        connection
        System.out.println("Client connected.");

        BufferedReader input = new BufferedReader(new
        InputStreamReader(socket.getInputStream()));
        PrintWriter output = new PrintWriter(socket.getOutputStream(),
        true);

        String received = input.readLine();
        System.out.println("Received from client: " + received);

        output.println("Hello from server!");
        socket.close();
        serverSocket.close();
    }
}
```

TCP Client

```
import java.io.*;
import java.net.*;

public class TCPClient {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("localhost", 5000); // Connect to
        server on port 5000

        BufferedReader input = new BufferedReader(new
        InputStreamReader(socket.getInputStream()));
        PrintWriter output = new PrintWriter(socket.getOutputStream(),
        true);

        output.println("Hello Server!");
        String response = input.readLine();
        System.out.println("Server response: " + response);

        socket.close();
    }
}
```

✅ **Run the server first, then start the client.**

✅ **Output:**

Server: Received from client: Hello Server!
Client: Server response: Hello from server!

1.2. UDP (User Datagram Protocol)

- **Connectionless, faster but unreliable.**
- **No guarantee of message delivery or order.**
- **Used in video streaming, VoIP, DNS, etc..**

Example: UDP Server and Client

UDP Server

```
import java.net.*;

public class UDPServer {
    public static void main(String[] args) throws Exception {
        DatagramSocket socket = new DatagramSocket(9876);
        byte[] buffer = new byte[1024];

        DatagramPacket packet = new DatagramPacket(buffer,
buffer.length);
        System.out.println("Server waiting for client...");
        socket.receive(packet); // Receive data

        String received = new String(packet.getData(), 0,
packet.getLength());
        System.out.println("Received from client: " + received);

        String response = "Hello from UDP Server!";
        DatagramPacket responsePacket = new
DatagramPacket(response.getBytes(), response.length(),
        packet.getAddress(), packet.getPort());
        socket.send(responsePacket); // Send response

        socket.close();
    }
}
```

UDP Client

```
import java.net.*;

public class UDPClient {
    public static void main(String[] args) throws Exception {
        DatagramSocket socket = new DatagramSocket();
        InetAddress address = InetAddress.getByName("localhost");

        String message = "Hello UDP Server!";
        DatagramPacket packet = new DatagramPacket(message.getBytes(),
message.length(), address, 9876);
        socket.send(packet); // Send message

        byte[] buffer = new byte[1024];
        DatagramPacket responsePacket = new DatagramPacket(buffer,
buffer.length);
        socket.receive(responsePacket); // Receive response

        String response = new String(responsePacket.getData(), 0,
responsePacket.getLength());
        System.out.println("Server response: " + response);

        socket.close();
    }
}
```

✓ **Run the server first, then start the client.**

✓ **Output:**

```
Server: Received from client: Hello UDP Server!  
Client: Server response: Hello from UDP Server!
```

2. URL and HTTP Communication

Java provides classes like `URL`, `URLConnection`, and `HttpURLConnection` to communicate with web servers.

2.1. Reading from a URL

```
import java.io.*;  
import java.net.*;  
  
public class URLReader {  
    public static void main(String[] args) throws Exception {  
        URL url = new URL("https://www.example.com");  
        BufferedReader reader = new BufferedReader(new  
InputStreamReader(url.openStream()));  
  
        String line;  
        while ((line = reader.readLine()) != null) {  
            System.out.println(line);  
        }  
        reader.close();  
    }  
}
```

2.2. Sending HTTP Requests using `HttpURLConnection`

```
import java.io.*;  
import java.net.*;  
  
public class HTTPClient {  
    public static void main(String[] args) throws Exception {  
        URL url = new  
URL("https://jsonplaceholder.typicode.com/posts/1");  
        HttpURLConnection connection = (HttpURLConnection)  
url.openConnection();  
  
        connection.setRequestMethod("GET"); // Set request type  
        int responseCode = connection.getResponseCode();  
        System.out.println("Response Code: " + responseCode);  
  
        BufferedReader reader = new BufferedReader(new  
InputStreamReader(connection.getInputStream()));  
        String line;  
        while ((line = reader.readLine()) != null) {  
            System.out.println(line);  
        }  
        reader.close();  
    }  
}
```

✓ **Output:** JSON response from API.

2.3. Sending POST Requests

```
import java.io.*;
import java.net.*;

public class HTTPPostRequest {
    public static void main(String[] args) throws Exception {
        URL url = new URL("https://jsonplaceholder.typicode.com/posts");
        HttpURLConnection connection = (HttpURLConnection)
url.openConnection();

        connection.setRequestMethod("POST");
        connection.setRequestProperty("Content-Type",
"application/json");
        connection.setDoOutput(true);

        String jsonData = "{\"title\":\"Java HTTP\",\"body\":\"Hello,
World!\",\"userId\":1}";
        try (OutputStream os = connection.getOutputStream()) {
            os.write(jsonData.getBytes());
        }

        BufferedReader reader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
        reader.close();
    }
}
```

3. Working with `java.net` Package

The `java.net` package provides classes for networking in Java.

Common Classes in `java.net`

Class	Description
Socket	Represents a TCP socket.
ServerSocket	Listens for TCP connections.
DatagramSocket	Represents a UDP socket.
InetAddress	Represents an IP address.
URL	Represents a web address.
URLConnection	Represents a connection to a web resource.
HttpURLConnection	Handles HTTP requests and responses.

Best Practices for Networking in Java

- ✔ **Use Buffered Streams** for better performance (`BufferedReader`, `BufferedWriter`).
- ✔ **Close Sockets and Streams** properly (`socket.close()`).
- ✔ **Handle Timeouts** to avoid blocking:

```
socket.setSoTimeout(5000); // 5 seconds timeout
```

- ✔ **Use Multi-threading** for handling multiple clients (`ExecutorService` for thread pool).

Q&A

1. What is the difference between TCP and UDP?

Answer:

Feature	TCP (Transmission Control Protocol)	UDP (User Datagram Protocol)
Connection	Connection-oriented	Connectionless
Reliability	Reliable (ensures data delivery)	Unreliable (no guarantee of delivery)
Speed	Slower due to acknowledgment	Faster (no acknowledgment overhead)
Use Cases	HTTP, FTP, Email (where order matters)	Video streaming, VoIP, DNS (where speed is critical)

2. What is a socket in Java?

Answer: A **socket** is an **endpoint for communication** between two machines over a network.

- **Server Socket:** Listens for incoming connections.
- **Client Socket:** Connects to a server.

3. How do you create a TCP server and client in Java?

Answer:

- **TCP Server:**
 - `ServerSocket serverSocket = new ServerSocket(5000);`
 - `Socket socket = serverSocket.accept(); // Accept client connection`
- **TCP Client:**
 - `Socket socket = new Socket("localhost", 5000);`

4. How do you send and receive data over TCP in Java?

Answer:

- **Sending Data (Server/Client):**
 - `PrintWriter output = new PrintWriter(socket.getOutputStream(), true);`
 - `output.println("Hello Client!");`
 - **Receiving Data (Server/Client):**
 - `BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));`
 - `String message = input.readLine();`
-

5. How do you implement UDP communication in Java?

Answer:

- **UDP Server:**
 - `DatagramSocket socket = new DatagramSocket(9876);`
 - `byte[] buffer = new byte[1024];`
 - `DatagramPacket packet = new DatagramPacket(buffer, buffer.length);`
 - `socket.receive(packet); // Receive data`
 - **UDP Client:**
 - `InetAddress address = InetAddress.getByName("localhost");`
 - `DatagramPacket packet = new DatagramPacket(message.getBytes(), message.length(), address, 9876);`
 - `socket.send(packet); // Send message`
-

6. What is the `java.net` package used for?

Answer: The `java.net` package provides classes for **network communication**.

Class	Description
Socket	Represents a TCP connection.
ServerSocket	Listens for TCP connections.
DatagramSocket	Represents a UDP connection.
InetAddress	Handles IP addresses.
URL	Represents a web URL.
URLConnection	Handles HTTP requests.

7. How do you make an HTTP GET request in Java?

Answer: Using `URLConnection`:

```
URL url = new URL("https://jsonplaceholder.typicode.com/posts/1");
URLConnection conn = (URLConnection) url.openConnection();
conn.setRequestMethod("GET");
```

```
BufferedReader reader = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
String line;
```

```
while ((line = reader.readLine()) != null) {
    System.out.println(line);
}
reader.close();
```

8. How do you make an HTTP POST request in Java?

Answer:

```
URL url = new URL("https://jsonplaceholder.typicode.com/posts");
URLConnection conn = (URLConnection) url.openConnection();
conn.setRequestMethod("POST");
conn.setRequestProperty("Content-Type", "application/json");
conn.setDoOutput(true);

String jsonData = "{\"title\":\"Java HTTP\",\"body\":\"Hello, World!\",\"userId\":1}";
try (OutputStream os = conn.getOutputStream()) {
    os.write(jsonData.getBytes());
}

BufferedReader reader = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
String line;
while ((line = reader.readLine()) != null) {
    System.out.println(line);
}
reader.close();
```

9. What is `InetAddress` used for?

Answer: `InetAddress` is used to retrieve IP addresses of hosts.

Example:

```
InetAddress address = InetAddress.getByName("google.com");
System.out.println("IP Address: " + address.getHostAddress());
```

10. What is the difference between `URLConnection` and `HttpURLConnection`?

Answer:

Feature	<code>URLConnection</code>	<code>HttpURLConnection</code>
Protocol Support	Can handle multiple protocols (HTTP, FTP)	Specific to HTTP
Use Cases	Used for general URL connections	Used for sending HTTP requests

11. How do you handle socket timeouts in Java?

Answer: Use `setSoTimeout()` to prevent blocking indefinitely.

```
socket.setSoTimeout(5000); // 5 seconds timeout
```

12. What is the difference between blocking and non-blocking sockets?

Answer:

Feature	Blocking Sockets	Non-Blocking Sockets
Behavior	Waits for data	Does not wait
Performance	Slower due to waiting	Faster (CPU-efficient)
Use Case	Simple applications	High-performance applications

13. How do you implement a multi-threaded server in Java?

Answer: Use **threads** to handle multiple clients.

Example:

```
class ClientHandler extends Thread {
    private Socket clientSocket;

    public ClientHandler(Socket socket) {
        this.clientSocket = socket;
    }

    public void run() {
        try {
            BufferedReader input = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            PrintWriter output = new
PrintWriter(clientSocket.getOutputStream(), true);

            String message = input.readLine();
            System.out.println("Received: " + message);
            output.println("Hello from server!");

            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In main server class:

```
while (true) {
    Socket clientSocket = serverSocket.accept();
    new ClientHandler(clientSocket).start(); // Start a new thread for
each client
}
```

14. How do you check if a port is available in Java?

Answer:

```
try (ServerSocket serverSocket = new ServerSocket(8080)) {
    System.out.println("Port is available.");
} catch (IOException e) {
    System.out.println("Port is in use.");
}
```

15. What are some best practices for networking in Java?

Answer: ☒ Use Buffered Streams (BufferedReader, BufferedWriter).

☒ Close Sockets and Streams properly (socket.close()).

☒ Set Timeouts (socket.setSoTimeout(5000)).

☒ Use Multi-threading for handling multiple clients.

☒ Handle Exceptions to prevent crashes.

16. What is the difference between GET and POST in HTTP?

Answer:

Feature	GET	POST
Use Case	Retrieves data	Sends data to server
Data in URL?	Yes (in query string)	No (in body)
Security	Less secure	More secure
Idempotent?	Yes	No

17. How do you send a file over a network in Java?

Answer:

1. Read file into a byte array.
2. Send the byte array over a socket.

Example:

```
byte[] buffer = Files.readAllBytes(Paths.get("file.txt"));
OutputStream out = socket.getOutputStream();
out.write(buffer);
out.flush();
```

18. What is WebSockets in Java?

Answer: WebSockets provide **real-time bidirectional communication** over a single connection.

Example:

```
@ServerEndpoint("/chat")
public class ChatServer {
    @OnMessage
    public void onMessage(String message, Session session) {
        session.getAsyncRemote().sendText("Echo: " + message);
    }
}
```

19. What is the difference between REST and WebSockets?

Answer:

Feature	REST API	WebSockets
Type	Request-Response	Real-time communication
Connection	Stateless (New connection per request)	Persistent connection
Use Case	APIs, CRUD operations	Chat applications, real-time data

20. How can you monitor network performance in Java?

Answer:

- Use **Netty** for high-performance networking.
- Use **JConsole** or **Wireshark** for packet inspection.
- Log **response times and error rates**.

Topic: Security in Java

Java provides a robust set of **Security APIs** under the `java.security` package, allowing developers to implement **authentication, encryption, decryption, hashing, and access control**.

1. Basics of Java Security APIs

Java Security APIs provide mechanisms for:

- ✔ **Cryptographic operations** (encryption, decryption, hashing).
- ✔ **Secure random number generation.**
- ✔ **Digital signatures and certificate handling.**
- ✔ **Access control** using Java Policy files.

Key Java Security APIs

Class/Package	Description
java.security	Provides cryptography, key management, digital signatures.
javax.crypto	Supports encryption and decryption using AES, RSA, etc.
java.security.KeyPair	Generates public-private key pairs.
java.security.MessageDigest	Implements hashing algorithms (MD5, SHA-256).
javax.crypto.Cipher	Handles encryption and decryption.
java.security.SecureRandom	Generates cryptographically secure random numbers.
java.security.Signature	Supports digital signatures.

2. Hashing (One-Way Encryption)

Hashing converts **data into a fixed-length hash** and is **irreversible**. It is commonly used for **password storage**.

Example: Hashing a String using SHA-256

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Base64;

public class HashingExample {
    public static void main(String[] args) throws
    NoSuchAlgorithmException {
        String password = "SecurePassword123";

        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] hash = digest.digest(password.getBytes());

        String hashedPassword =
        Base64.getEncoder().encodeToString(hash);
        System.out.println("SHA-256 Hash: " + hashedPassword);
    }
}
```

✔ **Output (Example):**

SHA-256 Hash: Vze8g65.... (Base64 Encoded)

- ✔ **Common Hashing Algorithms:** MD5, SHA-1, SHA-256, SHA-512.
 - ✗ **Avoid MD5 and SHA-1** (They are weak and insecure).
-

3. Encryption and Decryption using Java Security APIs

Encryption converts **plain text into cipher text**, and decryption converts it back.

Types of Encryption

Encryption Type	Example Algorithms	Use Case
Symmetric	AES, DES	Faster, single key for encryption & decryption
Asymmetric	RSA, ECC	Secure, uses public & private key pair

3.1. Symmetric Encryption (AES)

AES (Advanced Encryption Standard) is widely used for **fast encryption**.

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.util.Base64;

public class AESEncryption {
    public static void main(String[] args) throws Exception {
        String text = "Hello, Secure World!";

        // Generate AES Key
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(256); // AES-256 for strong encryption
        SecretKey secretKey = keyGen.generateKey();

        // Encrypt
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        byte[] encrypted = cipher.doFinal(text.getBytes());
        String encryptedText =
Base64.getEncoder().encodeToString(encrypted);
        System.out.println("Encrypted: " + encryptedText);

        // Decrypt
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        byte[] decrypted =
cipher.doFinal(Base64.getDecoder().decode(encryptedText));
        System.out.println("Decrypted: " + new String(decrypted));
    }
}
```

✓ Output:

Encrypted: u3P8... (Base64 Encoded)
Decrypted: Hello, Secure World!

✓ Use AES-256 for strong encryption.

✓ Keep the secret key safe.

3.2. Asymmetric Encryption (RSA)

RSA (Rivest-Shamir-Adleman) uses a **public key for encryption** and a **private key for decryption**.

```
import java.security.*;
import javax.crypto.Cipher;
import java.util.Base64;

public class RSAEncryption {
    public static void main(String[] args) throws Exception {
        String text = "Hello, Secure World!";

        // Generate RSA Key Pair
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
        keyGen.initialize(2048); // RSA-2048 for security
        KeyPair keyPair = keyGen.generateKeyPair();
        PublicKey publicKey = keyPair.getPublic();
        PrivateKey privateKey = keyPair.getPrivate();

        // Encrypt with Public Key
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);
        byte[] encrypted = cipher.doFinal(text.getBytes());
        String encryptedText =
Base64.getEncoder().encodeToString(encrypted);
        System.out.println("Encrypted: " + encryptedText);

        // Decrypt with Private Key
        cipher.init(Cipher.DECRYPT_MODE, privateKey);
        byte[] decrypted =
cipher.doFinal(Base64.getDecoder().decode(encryptedText));
        System.out.println("Decrypted: " + new String(decrypted));
    }
}
```

✓ Output:

```
Encrypted: W+89... (Base64 Encoded)
Decrypted: Hello, Secure World!
```

✓ **Use RSA-2048 or higher** for secure encryption.

✓ **Private keys must be kept secret.**

4. Secure Random Number Generation

Java provides `SecureRandom` to **generate cryptographically secure random numbers**.

```
import java.security.SecureRandom;
import java.util.Base64;

public class SecureRandomExample {
    public static void main(String[] args) {
        SecureRandom random = new SecureRandom();
        byte[] bytes = new byte[16]; // Generate 16 bytes
    }
}
```

```

        random.nextBytes(bytes);

        String randomToken = Base64.getEncoder().encodeToString(bytes);
        System.out.println("Secure Random Token: " + randomToken);
    }
}

```

✅ Output:

Secure Random Token: aB34fzK87...

✅ Use `SecureRandom` instead of `Math.random()` for security.

5. Digital Signatures

Digital signatures **verify the integrity and authenticity** of a message.

```

import java.security.*;

public class DigitalSignatureExample {
    public static void main(String[] args) throws Exception {
        String message = "Secure Message";

        // Generate Key Pair
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
        keyGen.initialize(2048);
        KeyPair keyPair = keyGen.generateKeyPair();
        PrivateKey privateKey = keyPair.getPrivate();
        PublicKey publicKey = keyPair.getPublic();

        // Sign Message
        Signature signature = Signature.getInstance("SHA256withRSA");
        signature.initSign(privateKey);
        signature.update(message.getBytes());
        byte[] digitalSignature = signature.sign();
        System.out.println("Digital Signature: " +
Base64.getEncoder().encodeToString(digitalSignature));

        // Verify Signature
        signature.initVerify(publicKey);
        signature.update(message.getBytes());
        boolean isValid = signature.verify(digitalSignature);
        System.out.println("Is Signature Valid? " + isValid);
    }
}

```

✅ Output:

Digital Signature: aBcdE... (Base64 Encoded)
Is Signature Valid? true

✅ Use RSA with SHA-256 for strong digital signatures.

Best Practices for Java Security

- ✔ **Use Strong Encryption** (AES-256, RSA-2048).
- ✔ **Never Hardcode Keys or Passwords.**
- ✔ **Use `SecureRandom` for Random Number Generation.**
- ✔ **Validate Input to Prevent Injection Attacks.**
- ✔ **Hash Passwords Before Storing** (Use `bcrypt`, `PBKDF2`, `Argon2`).

Q&A

1. What are Java Security APIs used for?

Answer: Java Security APIs provide cryptographic functions such as:

- Encryption & Decryption** (`javax.crypto.Cipher` for AES, RSA).
- Hashing** (`java.security.MessageDigest` for SHA-256).
- Random Number Generation** (`java.security.SecureRandom`).
- Digital Signatures** (`java.security.Signature`).
- Access Control** (`java.security.Policy`).

2. What is the difference between Hashing and Encryption?

Answer:

Feature	Hashing	Encryption
Purpose	Converts data into a fixed-length hash	Converts data into ciphertext
Reversibility	Irreversible (one-way function)	Reversible (can be decrypted)
Common Algorithms	SHA-256, SHA-512, MD5	AES, RSA
Use Cases	Password storage, Data integrity	Secure communication, File protection

3. What is the best hashing algorithm for passwords in Java?

Answer: Use `bcrypt`, `PBKDF2`, or `Argon2` for password hashing.

Example using **SHA-256** (for non-password use cases):

```
MessageDigest digest = MessageDigest.getInstance("SHA-256");
```

```
byte[] hash = digest.digest("password".getBytes());
System.out.println(Base64.getEncoder().encodeToString(hash));
```

✗ Avoid MD5 and SHA-1 (they are weak and insecure).

4. What is the difference between Symmetric and Asymmetric Encryption?

Answer:

Feature	Symmetric Encryption	Asymmetric Encryption
Keys Used	One key (shared secret)	Public key (encryption) & Private key (decryption)
Speed	Faster	Slower
Examples	AES, DES	RSA, ECC
Use Cases	Data encryption (files, database)	Secure communication (SSL/TLS, digital signatures)

5. How do you perform AES Encryption in Java?

Answer: AES uses a **secret key** for both encryption and decryption.

```
import javax.crypto.*;
import java.util.Base64;

KeyGenerator keyGen = KeyGenerator.getInstance("AES");
keyGen.init(256);
SecretKey secretKey = keyGen.generateKey();

Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
byte[] encrypted = cipher.doFinal("Hello AES".getBytes());
System.out.println(Base64.getEncoder().encodeToString(encrypted));
```

6. How does RSA encryption work in Java?

Answer: RSA uses a **public key to encrypt** and a **private key to decrypt**.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(2048);
KeyPair keyPair = keyGen.generateKeyPair();
PublicKey publicKey = keyPair.getPublic();
PrivateKey privateKey = keyPair.getPrivate();

// Encrypt with public key
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.ENCRYPT_MODE, publicKey);
byte[] encrypted = cipher.doFinal("Hello RSA".getBytes());

// Decrypt with private key
cipher.init(Cipher.DECRYPT_MODE, privateKey);
```



```
byte[] decrypted = cipher.doFinal(encrypted);
System.out.println(new String(decrypted));
```

7. What is `SecureRandom` and why is it used?

Answer: `SecureRandom` is a cryptographically secure random number generator.

Example:

```
SecureRandom random = new SecureRandom();
byte[] bytes = new byte[16];
random.nextBytes(bytes);
System.out.println(Base64.getEncoder().encodeToString(bytes));
```

✅ Use `SecureRandom` instead of `Math.random()` for security-critical applications.

8. What is a Digital Signature and how does it work in Java?

Answer: A Digital Signature verifies the authenticity and integrity of data.

Example:

```
Signature signature = Signature.getInstance("SHA256withRSA");
signature.initSign(privateKey);
signature.update("Message".getBytes());
byte[] digitalSignature = signature.sign();
System.out.println(Base64.getEncoder().encodeToString(digitalSignature));
;
```

✅ Ensures data integrity and authenticity.

9. What is the difference between `MessageDigest` and `Cipher`?

Answer:

Feature	<code>MessageDigest</code>	<code>Cipher</code>
Purpose	Hashing	Encryption/Decryption
Reversibility	Irreversible	Reversible
Example Algorithm	SHA-256	AES, RSA
Use Case	Password storage	Secure data transfer

10. What are some common security vulnerabilities in Java?

Answer:

1. **SQL Injection** (Unvalidated input in SQL queries).
 2. **XSS (Cross-Site Scripting)** (Injecting malicious scripts into web apps).
 3. **Insecure Cryptography** (Using weak algorithms like MD5, DES).
 4. **Hardcoded Secrets** (Storing passwords/API keys in code).
 5. **Improper Input Validation** (Allowing unexpected user input).
-

11. How do you securely store passwords in Java?

Answer:  Use a secure hashing algorithm like PBKDF2, bcrypt, or Argon2.

 **Never store plaintext passwords.**

 **Use `SecureRandom` for salt generation.**

Example using PBKDF2:

```
import java.security.spec.*;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import java.util.Base64;

char[] password = "securepassword".toCharArray();
byte[] salt = new byte[16];
new SecureRandom().nextBytes(salt);

PBEKeySpec spec = new PBEKeySpec(password, salt, 65536, 256);
SecretKeyFactory factory =
    SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
byte[] hash = factory.generateSecret(spec).getEncoded();
System.out.println(Base64.getEncoder().encodeToString(hash));
```

 **PBKDF2 applies multiple iterations to slow down brute-force attacks.**

12. What is an SSL/TLS Certificate in Java?

Answer: SSL/TLS certificates:

- Encrypt communication between a client and server.
- Are stored in a **Java KeyStore (JKS)** file.
- Are used in HTTPS connections.

Example: Load an SSL certificate in Java:

```
System.setProperty("javax.net.ssl.trustStore", "mykeystore.jks");
System.setProperty("javax.net.ssl.trustStorePassword", "password");
```

13. What is Java Keystore (JKS)?

Answer: A **Java KeyStore (JKS)** is a repository of **security certificates, private keys, and public keys**.

✔ Use the **keytool** command to manage keystores:

```
keytool -genkey -alias mykey -keyalg RSA -keystore mykeystore.jks -storepass mypassword
```

14. What is the role of `java.policy` file in Java Security?

Answer: The `java.policy` file **defines permissions for Java applications**.

Example policy file:

```
grant {  
    permission java.io.FilePermission "C:/data/*", "read, write";  
    permission java.net.SocketPermission "localhost:8080", "connect";  
};
```

✔ **Restricts file access, network access, and system resources.**

15. What is Java Authentication and Authorization Service (JAAS)?

Answer: JAAS provides **authentication & access control** in Java.

Example authentication code:

```
LoginContext loginContext = new LoginContext("MyApp");  
loginContext.login();
```

✔ **Used in enterprise applications for role-based security.**

16. How can you prevent security vulnerabilities in Java applications?

Answer: ✔ **Validate all user inputs** (prevent SQL Injection & XSS).

✔ **Use Strong Cryptography** (AES-256, RSA-2048).

✔ **Store Passwords Securely** (bcrypt, PBKDF2).

✔ **Use HTTPS for Secure Communication.**

✔ **Avoid Hardcoding Secrets** (use environment variables or vaults).

Topic: Java Profiling and Debugging

Java **profiling and debugging** help developers identify performance bottlenecks, memory leaks, and runtime errors. Java provides tools such as **JConsole**, **VisualVM**, and **profilers** to monitor and optimize applications.

1. Using Debugging Tools: JConsole & VisualVM

1.1. What is JConsole?

JConsole (**Java Monitoring and Management Console**) is a GUI tool for **monitoring Java applications in real-time**.

✅ Features of JConsole:

- **Monitors CPU usage, memory, and threads.**
- **Detects memory leaks.**
- **Analyzes Garbage Collection (GC) activity.**

How to Use JConsole?

1. **Run your Java application with JMX enabled:**
 2. `java -Dcom.sun.management.jmxremote -jar myapp.jar`
 3. **Open JConsole:**
 4. `jconsole`
 5. **Select the running Java process and start monitoring.**
-

1.2. What is VisualVM?

VisualVM is a **powerful profiling tool** that provides:

- ✅ **Heap Dump Analysis** (Identifies memory leaks).
- ✅ **Thread Monitoring** (Detects deadlocks).
- ✅ **CPU & Memory Profiling** (Optimizes performance).

How to Use VisualVM?

1. **Run VisualVM:**
 2. `jvisualvm`
 3. **Select your running Java process.**
 4. **Monitor heap usage, CPU performance, and threads.**
-

2. Profiling Applications

2.1. What is Java Profiling?

Java **profiling** is the process of analyzing:

- ✅ **Method execution time.**
- ✅ **Memory allocation.**
- ✅ **Thread activity and synchronization issues.**

✅ **Popular Profiling Tools:**

Tool	Description
VisualVM	Built-in Java profiler for monitoring CPU & memory.
JProfiler	Advanced profiling tool for performance tuning.
YourKit	Provides CPU & memory profiling with rich UI.
Eclipse Memory Analyzer (MAT)	Finds memory leaks using heap dumps.

2.2. Profiling a Java Application

To profile a Java application using VisualVM:

1. **Run VisualVM:**
2. `jvisualvm`
3. **Attach your Java application.**
4. **Analyze the following metrics:**
 - **CPU usage** (to find slow methods).
 - **Heap memory usage** (to detect memory leaks).
 - **Garbage Collection (GC) activity.**

2.3. Analyzing Heap Dumps

Heap dumps help in **finding memory leaks** by analyzing **live objects in memory**.

✅ **Generate a heap dump manually:**

```
jmap -dump:format=b,file=heapdump.hprof <pid>
```

✅ **Analyze heap dump using Eclipse MAT:**

```
java -jar mat.jar heapdump.hprof
```

2.4. Finding Memory Leaks

A **memory leak** occurs when objects are **not garbage collected** due to lingering references.

Common Causes of Memory Leaks:

1. **Unclosed Streams or Connections.**
2. **Static Collections Holding References.**
3. **Thread Locals** (Long-lived references).
4. **Listeners that are not removed.**

Detect leaks using jmap:

```
jmap -histo:live <pid>
```

Fix leaks by closing unused references:

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt")))
{
    System.out.println(br.readLine());
} // Auto-closes
```

2.5. Monitoring Threads

To **detect thread deadlocks**, use:

```
jstack <pid>
```

Example Deadlock Detection:

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

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (lock1) {
                System.out.println("Thread 1: Holding Lock1...");
                try { Thread.sleep(100); } catch (InterruptedException
e) {}
                synchronized (lock2) { System.out.println("Thread 1:
Holding Lock2..."); }
            }
        }).start();

        new Thread(() -> {
            synchronized (lock2) {
                System.out.println("Thread 2: Holding Lock2...");
                try { Thread.sleep(100); } catch (InterruptedException
e) {}
                synchronized (lock1) { System.out.println("Thread 2:
Holding Lock1..."); }
            }
        }).start();
    }
}
```

```
}
```

✅ **Detect Deadlocks using `jstack`:**

```
jstack <pid>
```

✅ **Fix Deadlocks by Acquiring Locks in the Same Order.**

2.6. Measuring CPU Usage

To monitor CPU usage:

```
top -p <pid>
```

✅ **Use `perf` for deeper profiling:**

```
perf record -p <pid>
```

Best Practices for Java Profiling

- ✅ Use JConsole for real-time monitoring.
- ✅ Use VisualVM for deep memory and CPU profiling.
- ✅ Analyze heap dumps to detect memory leaks.
- ✅ Use `jstack` to detect deadlocks.
- ✅ Optimize performance by identifying slow methods using profilers.

Q&A

1. What is Java profiling, and why is it important?

Answer: Java profiling is the process of **analyzing and optimizing** application performance by monitoring:

1. **CPU usage** (identifying slow methods).
2. **Memory consumption** (detecting memory leaks).
3. **Thread execution** (finding deadlocks and contention issues).
4. **Garbage Collection (GC) activity** (understanding memory management).

Profiling helps **optimize performance, reduce memory usage, and eliminate bottlenecks.**

2. What tools are available for Java profiling and debugging?

Answer:

Tool	Description
JConsole	Monitors memory, CPU, threads, and garbage collection in real-time.
VisualVM	Profiles heap, CPU usage, and detects memory leaks.
JProfiler	Commercial tool for in-depth profiling (memory leaks, CPU).
YourKit	Advanced Java profiler with visual analysis.
Eclipse Memory Analyzer (MAT)	Analyzes heap dumps to detect memory leaks.
JStack	Analyzes thread states, detects deadlocks.
JMap	Captures heap memory dumps.
JCmd	Monitors JVM performance in real-time.

3. How do you start JConsole for monitoring a Java application?

Answer:

1. Run your Java application **with JMX enabled**:
2. `java -Dcom.sun.management.jmxremote -jar myapp.jar`
3. **Open JConsole:**
4. `jconsole`
5. Select the running **Java process** and start monitoring **CPU, memory, and threads**.


4. How do you start VisualVM and connect it to a Java process?

Answer:

1. Start **VisualVM**:
2. `jvisualvm`
3. In the **Applications Window**, locate the running Java process.
4. Click **Monitor** to analyze **CPU, memory, and garbage collection**.
5. Click **Sampler** to profile **method execution and performance**.

5. How do you capture a heap dump for memory analysis?

Answer: Heap dumps **help detect memory leaks**.

 Capture a heap dump manually:

```
jmap -dump:format=b,file=heapdump.hprof <pid>
```

 **Analyze heap dump using Eclipse MAT:**

```
java -jar mat.jar heapdump.hprof
```

6. How do you detect memory leaks in Java?

Answer: Memory leaks occur when **objects are not garbage collected** due to lingering references.

✓ Common causes:

1. **Static collections (e.g., List or Map not cleared).**
2. **Unclosed resources (e.g., streams, database connections).**
3. **ThreadLocals that are not removed.**
4. **Listeners that are not de-registered.**

✓ Detect leaks using JMap:

```
jmap -histo:live <pid>
```

✓ Fix leaks by closing references properly:

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt")))
{
    System.out.println(br.readLine());
} // Auto-closes
```

7. How do you monitor Java threads and detect deadlocks?

Answer: Deadlocks occur when **two or more threads are waiting for each other's locks.**

✓ Detect deadlocks using JStack:

```
jstack <pid>
```

✓ Example Deadlock Code:

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

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (lock1) {
                System.out.println("Thread 1: Holding Lock1...");
                try { Thread.sleep(100); } catch (InterruptedException
e) {}
                synchronized (lock2) { System.out.println("Thread 1:
Holding Lock2..."); }
            }
        }).start();

        new Thread(() -> {
            synchronized (lock2) {
                System.out.println("Thread 2: Holding Lock2...");
                try { Thread.sleep(100); } catch (InterruptedException
e) {}
```

```
        synchronized (lock1) { System.out.println("Thread 2:
Holding Lock1..."); }
    }
    }).start();
}
```

✅ **Fix Deadlocks by Acquiring Locks in the Same Order.**

8. How do you measure CPU usage of a Java application?

Answer: ✅ **Monitor CPU usage using `top` (Linux/Mac):**

```
top -p <pid>
```

✅ **Profile CPU performance using VisualVM:**

1. Open **VisualVM** (`jvisualvm`).
2. Select **your Java process**.
3. Click **Sampler** → **CPU**.

✅ **Use `perf` for deeper profiling:**

```
perf record -p <pid>
```

9. How do you analyze Garbage Collection (GC) activity?

Answer: GC logs help optimize **memory management**.

✅ **Enable GC logging in Java 8:**

```
-XX:+PrintGCDetails -Xloggc:gc.log
```

✅ **Monitor GC activity using JConsole:**

1. Open **JConsole** (`jconsole`).
2. Select **Garbage Collection tab**.

✅ **Use VisualVM for GC analysis:**

- Open **VisualVM** (`jvisualvm`).
 - Click **Profiler** → **GC Monitoring**.
-

10. What is the difference between `jmap` and `jstack`?

Answer:

Tool	Purpose
JMap	Captures heap dumps and memory analysis.
JStack	Captures thread stack traces, detects deadlocks.

11. How do you enable remote monitoring of a Java application?

Answer: Enable **JMX** remote monitoring:

```
java -Dcom.sun.management.jmxremote -  
Dcom.sun.management.jmxremote.port=9010 -  
Dcom.sun.management.jmxremote.ssl=false -  
Dcom.sun.management.jmxremote.authenticate=false -jar myapp.jar
```

✔ **Connect using JConsole or VisualVM** remotely.

12. What are some best practices for Java debugging and profiling?

Answer:

- ✔ Use **JConsole** for real-time monitoring.
 - ✔ Use **VisualVM** for deep memory and CPU profiling.
 - ✔ Analyze heap dumps to detect memory leaks.
 - ✔ Use **jstack** to detect deadlocks.
 - ✔ Optimize performance by identifying slow methods using profilers.
 - ✔ Enable GC logging to analyze garbage collection performance.
-

13. How do you find slow methods in a Java application?

Answer:

1. Use **VisualVM** or **JProfiler** to analyze method execution time.
 2. **Enable Java Flight Recorder (JFR):**
 3. `java -XX:+UnlockCommercialFeatures -XX:+FlightRecorder -jar myapp.jar`
 4. **Analyze logs for slow methods.**
-

14. What is Java Mission Control (JMC)?

Answer: JMC is a **low-overhead profiling tool** built into the JDK.

✅ **Start JMC:**

jmc

✅ **Monitor CPU, memory, and threads** with minimal performance impact.

15. How can you debug a running Java application using a remote debugger?

Answer:

1. Start your Java application **with remote debugging enabled**:
2.

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005 -jar myapp.jar
```
3. In your IDE (IntelliJ/Eclipse), set up **Remote Debugging on port 5005**.

✅ **Use breakpoints to inspect runtime values.**

16. How do you find and fix memory leaks in Java?

Answer:

1. **Enable heap dump analysis:**
 2.

```
jmap -dump:format=b,file=heap.hprof <pid>
```
 3. **Analyze heap dump using Eclipse MAT.**
 4. **Fix memory leaks by closing resources properly:**
 5.

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
```
 6.

```
    System.out.println(br.readLine());
```
 7.

```
}
```
-


Summary

- ✅ Use JConsole for real-time monitoring.
- ✅ Use VisualVM for CPU and memory profiling.
- ✅ Detect deadlocks with JStack.
- ✅ Analyze heap dumps using JMap and MAT.
- ✅ Enable GC logging to optimize garbage collection.
- ✅ Debug Java applications remotely using JDWP.

Topic: Advanced Topics in Java

This section covers **JVM Internals**, **Performance Tuning**, and **Custom Classloaders** for deeper understanding and optimization of Java applications.

1. JVM Internals

The **Java Virtual Machine (JVM)** is responsible for executing Java programs by: 
Loading `.class` files.

 **Converting bytecode into machine code via Just-In-Time (JIT) Compilation.**

 **Managing memory and garbage collection.**

1.1. Class Loading in JVM

Java **ClassLoader** loads Java classes **dynamically** at runtime.

 **Types of ClassLoaders:**

ClassLoader	Description
Bootstrap ClassLoader	Loads core Java classes (e.g., <code>java.lang.String</code>).
Extension ClassLoader	Loads classes from <code>lib/ext</code> directory.
Application ClassLoader	Loads classes from <code>classpath</code> .
Custom ClassLoader	User-defined class loading mechanism.


 **Example: Checking ClassLoader:**


```
public class ClassLoaderExample {
    public static void main(String[] args) {
        System.out.println(String.class.getClassLoader()); // Bootstrap
-> returns null
        System.out.println(ClassLoaderExample.class.getClassLoader());
// Application ClassLoader
    }
}
```

 **Output:**

```
null
sun.misc.Launcher$AppClassLoader
```

1.2. Bytecode in Java

 **Java programs compile to `.class` files containing bytecode.**

 **Bytecode is executed by the JVM or converted to machine code using JIT.**

✔ **Disassemble Bytecode using javap:**

```
javac Example.java
javap -c Example
```

✔ **Example Bytecode Output:**

```
0: getstatic      #2 // Get System.out
3: ldc            #3 // Load "Hello, World!"
6: invokevirtual #4 // Call println method
9: return
```

1.3. Just-In-Time (JIT) Compilation

✔ **JIT compiles frequently used methods to native code.**

✔ **Improves performance by reducing interpretation overhead.**

✔ **Types of JIT Compilers:**

JIT Mode	Description
Client (C1) Compiler	Optimized for fast startup (used in GUI apps).
Server (C2) Compiler	Optimized for long-running applications.
Tiered Compilation	Mixes C1 and C2 for balanced performance.

✔ **Enable JIT Debugging:**

```
java -XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation -jar myapp.jar
```

✔ **Check JIT Compilation Details:**

```
java -XX:+PrintCompilation -jar myapp.jar
```

2. Performance Tuning

Java performance tuning involves: ✔ **Optimizing CPU and memory usage.**

✔ **Reducing garbage collection (GC) overhead.**

✔ **Profiling the application** with tools like **JProfiler**, **YourKit**.

2.1. Using JProfiler for Performance Optimization

✔ **JProfiler detects:**

1. **CPU bottlenecks.**
2. **Memory leaks.**
3. **Thread contention.**

✅ Steps to Profile Java App:

1. Install **JProfiler**.
 2. Attach **JProfiler** to the running JVM.
 3. Analyze **CPU, memory, and threads**.
-

2.2. Optimizing Garbage Collection

✅ Use JVM Flags to Tune GC:

```
-XX:+UseG1GC -Xms512m -Xmx4g
```

✅ Monitor GC logs:

```
java -XX:+PrintGCDetails -Xloggc:gc.log -jar myapp.jar
```

✅ Use G1 GC for Large Applications:

```
-XX:+UseG1GC
```

2.3. Reducing CPU Usage

✅ Use JIT Compiler Flags:

```
-XX:+TieredCompilation
```

✅ Use Java Flight Recorder:

```
java -XX:+UnlockCommercialFeatures -XX:+FlightRecorder -jar myapp.jar
```

✅ Use `jstack` to detect high CPU usage:

```
jstack <pid>
```

3. Custom ClassLoaders

A Custom `ClassLoader` is useful for: ✅ Loading encrypted classes.

✅ Dynamically modifying bytecode.

✅ Loading plugins at runtime.

3.1. Creating a Custom ClassLoader

```
import java.io.*;

class MyClassLoader extends ClassLoader {
    public Class<?> findClass(String name) throws ClassNotFoundException
    {
```

```

        try {
            byte[] bytes = loadClassData(name);
            return defineClass(name, bytes, 0, bytes.length);
        } catch (IOException e) {
            throw new ClassNotFoundException();
        }
    }

    private byte[] loadClassData(String name) throws IOException {
        FileInputStream fis = new FileInputStream(name + ".class");
        byte[] data = new byte[fis.available()];
        fis.read(data);
        fis.close();
        return data;
    }
}

public class CustomClassLoaderExample {
    public static void main(String[] args) throws Exception {
        MyClassLoader loader = new MyClassLoader();
        Class<?> clazz = loader.findClass("HelloWorld");
        System.out.println("Class Loaded: " + clazz.getName());
    }
}

```

- ✅ **Dynamically loads HelloWorld.class.**
- ✅ **Useful for secure class loading and plugin systems.**

Best Practices for Advanced Java

- ✅ **Use VisualVM/JProfiler for Performance Tuning.**
- ✅ **Monitor JIT Compilation for Performance Gains.**
- ✅ **Use G1 GC for better garbage collection.**
- ✅ **Avoid classloading issues by structuring dependencies properly.**
- ✅ **Use Custom ClassLoaders for secure class loading.**

Q&A

1. What is the role of the JVM in Java?

Answer: The **Java Virtual Machine (JVM)** is responsible for:

1. **Loading class files using the ClassLoader.**
2. **Converting bytecode to machine code via the JIT compiler.**
3. **Managing memory using Garbage Collection (GC).**
4. **Executing Java programs in a platform-independent manner.**

✅ **JVM consists of:**

- **ClassLoader**
- **Execution Engine (JIT)**
- **Memory Areas (Heap, Stack, Method Area)**

- **Garbage Collector (GC)**

2. How does the Java ClassLoader work?

Answer: The **ClassLoader** loads `.class` files into the JVM at runtime.

✅ Types of ClassLoaders:

ClassLoader	Description
Bootstrap ClassLoader	Loads Java core classes (<code>java.lang.*</code>).
Extension ClassLoader	Loads classes from <code>lib/ext</code> directory.
Application ClassLoader	Loads classes from the classpath.
Custom ClassLoader	User-defined class loading mechanism.

✅ Check ClassLoader of a Class:

```
System.out.println(String.class.getClassLoader()); // null (Bootstrap)
System.out.println(MyClass.class.getClassLoader()); // Application
ClassLoader
```

✅ Example of a Custom ClassLoader:

```
class MyClassLoader extends ClassLoader {
    public Class<?> findClass(String name) throws ClassNotFoundException
    {
        return super.findClass(name);
    }
}
```

3. What is Bytecode in Java?

Answer: Bytecode is an **intermediate representation** of Java source code, executed by the JVM.


✅ Disassemble Bytecode using `javap`:

```
javac MyClass.java
javap -c MyClass
```

✅ Example Bytecode Output:

```
0: getstatic      #2 // Load System.out
3: ldc            #3 // Load constant "Hello World"
6: invokevirtual #4 // Call println method
9: return
```

4. What is Just-In-Time (JIT) Compilation?

Answer:  JIT compiles frequently used bytecode into native machine code for better performance.

 **Types of JIT Compilation:**

JIT Mode	Description
Client (C1) Compiler	Fast startup, optimized for GUI apps.
Server (C2) Compiler	Optimized for high-performance applications.
Tiered Compilation	Mix of C1 and C2 for balance.

 **Enable JIT Debugging:**

```
java -XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation -jar myapp.jar
```

5. What are the different Garbage Collection (GC) algorithms in Java?

Answer:  GC manages memory by removing unused objects.

GC Algorithm	Description
Serial GC	Single-threaded, best for small applications.
Parallel GC	Multi-threaded GC, faster collection.
CMS GC	Low-pause GC, used in enterprise apps.
G1 GC	Default in Java 9+, balances speed and memory efficiency.


 **Enable G1 GC:**

```
java -XX:+UseG1GC -jar myapp.jar
```


 **Enable GC Logs:**

```
java -XX:+PrintGCDetails -Xloggc:gc.log -jar myapp.jar
```

6. How can you monitor JVM performance?

Answer:  Use `jconsole` for real-time monitoring:

```
jconsole
```

 Use `jvisualvm` for deep profiling:

```
jvisualvm
```

 **Check memory usage with `jmap`:**

```
jmap -histo:live <pid>
```

7. How do you analyze a heap dump in Java?

Answer: Heap dumps help **find memory leaks**.

✅ Generate Heap Dump:

```
jmap -dump:format=b,file=heapdump.hprof <pid>
```

✅ Analyze Heap Dump using Eclipse MAT:

```
java -jar mat.jar heapdump.hprof
```

8. What is Java Flight Recorder (JFR)?

Answer: Java Flight Recorder **records JVM performance metrics** with minimal overhead.

✅ Enable JFR:

```
java -XX:+UnlockCommercialFeatures -XX:+FlightRecorder -jar myapp.jar
```

✅ Start JFR for 60 seconds:

```
jcmd <pid> JFR.start duration=60s filename=myrecord.jfr
```

9. How do you reduce CPU usage in a Java application?

Answer: ✅ **Profile CPU usage with VisualVM:**

```
jvisualvm
```

✅ Use JIT optimizations:

```
-XX:+TieredCompilation
```

✅ Optimize Loops & Data Structures:

```
for (String str : list) {  
    if ("match".equals(str)) break; // Efficient String comparison  
}
```

10. How do you detect and fix memory leaks?

Answer: ✅ **Common Causes:**

1. **Static collections (List, Map) holding references.**
2. **Unclosed resources (File, Socket, Database).**

3. Improper use of ThreadLocal variables.

✅ Detect leaks using jmap:

```
jmap -histo:live <pid>
```

✅ Fix by closing resources properly:

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))  
{  
    System.out.println(br.readLine());  
} // Auto-closes
```

11. What are Custom ClassLoaders, and why are they used?

Answer: Custom ClassLoaders **load classes dynamically** at runtime.

✅ Use Cases:

1. Loading encrypted classes.
2. Dynamically modifying bytecode.
3. Loading plugins at runtime.

✅ Example Custom ClassLoader:

```
class MyClassLoader extends ClassLoader {  
    public Class<?> findClass(String name) throws ClassNotFoundException  
{  
    return super.findClass(name);  
}  
}
```

12. How do you enable remote debugging in Java?

Answer: ✅ Start Java with Remote Debugging Enabled:

```
java -  
agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=:5005 -jar  
myapp.jar
```

✅ Connect using IntelliJ/Eclipse on port 5005.

13. How do you monitor and analyze Java threads?

Answer: ✅ Check thread status using jstack:

```
jstack <pid>
```

✅ **Example of Deadlock Detection:**

```
jstack <pid> | grep -A 10 "Found one Java-level deadlock"
```

14. How do you optimize Java applications for high performance?

Answer: ✅ **Enable G1 GC for faster memory cleanup:**

```
-XX:+UseG1GC
```

✅ **Use Just-In-Time (JIT) optimizations:**

```
-XX:+TieredCompilation
```

✅ **Minimize object creation:**

```
StringBuilder sb = new StringBuilder("Efficient String Manipulation");
```

15. What are some best practices for JVM tuning?

Answer: ✅ **Use the right GC for your application:**

```
-XX:+UseG1GC
```

✅ **Enable GC logging:**

```
-XX:+PrintGCDetails -Xloggc:gc.log
```

✅ **Optimize thread pools:**

```
ExecutorService executor = Executors.newFixedThreadPool(10);
```

✅ **Avoid large heap sizes unless necessary:**

```
-Xms512m -Xmx4g
```

Summary

- ✅ **JVM Internals:** Class Loading, JIT Compilation, Bytecode Execution.
- ✅ **Performance Tuning:** Profiling with VisualVM, JFR, JProfiler.
- ✅ **Custom ClassLoaders:** Dynamic class loading.
- ✅ **Garbage Collection Tuning:** Optimize with G1 GC.
- ✅ **Memory Leak Detection:** Heap dumps, JMap, Eclipse MAT.

THE END

Happy Learning