The Java and Spring Boot INTERVIEW COMPENDIUM

by YOHAN J. RODRÍGUEZ

Preface

"First, solve the problem. Then, write the code."
John Johnson

Java interviews rarely stop at "what does this keyword do?" A single conversation can run from why comparing two `Integer` values with `==` works for 127 and fails for 128, to how `ConcurrentHashMap` avoids a global lock, to why a `@Transactional` method does nothing when it is called from inside its own class, to how you would move an aging Java 8 and Spring Boot 2 service to Java 21 and Spring Boot 3. The distance between knowing the syntax and understanding the platform is where those conversations are won or lost. This book exists to close it. It is a question-and-answer reference: more than five hundred questions across fifteen chapters, each answer going past the definition to the mechanism behind it, the trade-off it carries, and the way a strong candidate would say it out loud.

It is written for developers who already write Java at work and now need to explain what they know. That includes mid-level engineers preparing for senior interview loops, backend developers who use Spring every day but have not revisited the platform underneath it, Java developers moving into Spring Boot, and engineers who want a broad refresher before a technical screen. It is not a first tutorial. There is no installation walkthrough and no application built step by step; Chapter 2 opens with primitives and wrapper classes, and concepts arrive at the pace of an interview answer. A reader new to Java will get more from an introductory text first, and a reader with years of hands-on experience will find some sections a quick review and others, such as the memory model, garbage collectors, and transaction propagation, worth slowing down for.

What you gain is concrete. You should be able to explain how the language and runtime actually behave: equality and immutability, generics and type erasure, the string pool, class loading, the Java Memory Model, and what a garbage collector does and when you would change it. You should be able to choose between collections or concurrency tools and state what each choice costs, read and write modern Java built on records, sealed types, pattern matching, streams, and virtual threads, and use thread dumps, heap dumps, and GC logs to narrow down a production problem. On the Spring side, you should be able to describe what the container, its proxies, and

auto-configuration do on your behalf, and therefore why some things silently fail to happen. You should also be able to design REST APIs and persistence layers with their usual pitfalls in view (N+1 queries, locking, pagination), test at the right level, and reason about securing, structuring, and deploying a service.

The book moves in two stages, and the proportions are deliberate: roughly two-thirds Java and one-third Spring. The first ten chapters cover Java itself. They begin with how the language evolved from Java 8 to 21, then work through fundamentals and object-oriented programming, collections and data-structure implementations, generics, lambdas, and streams, exception handling and annotations, concurrency, and the JVM with its memory management and performance tuning. Two chapters on design patterns and object-oriented design (SOLID, clean code, immutability, and domain modeling) close that stage. The last five chapters turn to Spring: the IoC container, dependency injection, and proxies; Spring Boot and REST APIs; persistence with JPA and Hibernate; testing with JUnit 5, Mockito, and Spring Boot test slices; and finally Spring Security, architecture, and deployment, from JWT and OAuth 2.0 to hexagonal architecture, resilience patterns, and containers. The stages are not sealed off from each other. Spring appears earlier wherever a Java topic and the framework meet, in annotations and AOP, in sizing a Spring Boot service, in patterns as Spring implements them, and the Spring chapters keep reaching back to the Java underneath.

Every chapter has the same shape. Questions are grouped into sections that move from core concepts to deeper mechanics and scenario questions, and every chapter after the first ends with a section of coding challenges. Each answer combines a written explanation, often a short bulleted summary of the points to remember, and a Java listing. In Chapters 1 through 10 the listings are, with few exceptions, complete programs with their own `main` method, meant to be run, changed, and broken on purpose. In Chapters 11 through 15 they are focused excerpts written against the real Spring, JPA, JUnit, and Spring Security APIs, and they assume a Spring Boot project around them. Where a topic matters in more than one setting, such as retries, circuit breakers, or multi-tenancy, it returns in each chapter from that chapter's angle.

You can read straight through or dip in. A sequential first pass builds the vocabulary that later chapters lean on, including generics, proxies, and the memory model, but the chapters are self-contained enough to be read by need, and the section headings make the book easy to scan the night before an interview. Whatever the order, use it actively: read a question, answer it aloud before reading further, then compare. If you can already explain why a section's questions matter, skim it; spend your time where your answer would have been vague.

A few assumptions are worth stating. You need working knowledge of Java syntax and basic programming, but no prior Spring experience for Chapter 11, which starts from inversion of

control. Chapters 12 through 15 read more easily if you have built a small Spring Boot service, and comfort with HTTP and basic SQL helps in the REST and persistence chapters. The book is written around the long-term-support releases (Java 8, 11, 17, and 21) and Spring Boot 3, and Chapter 1 records where each feature arrived so you can answer for the version your team actually runs. Features still in preview are labeled as such.

By the end, you should be able to explain, compare, and defend Java and Spring decisions in an interview or a design review: what a construct does, why it was designed that way, what it costs, and when you would choose something else. No book can promise an offer, and no book can replace building things. What this one can do is make the reasoning behind each answer your own, so that the next question, including the one you have not seen before, has somewhere to start.

Yohan J. Rodríguez
www.imabit.com

Contents

Preface i

1 Java Fundamentals **1**

1.1 Java Basics 1
1.2 Type System and Conversions 10
1.3 Strings and Immutability 12
1.4 Enums and Constants 16
1.5 Object Class Methods 20
1.6 Advanced Fundamentals 24
1.7 Memory and Performance Fundamentals 37
1.8 Interview Scenario Questions 50
1.9 Debugging and Scenario Questions 65
1.10 Coding Challenge Questions 84

1 | Java Fundamentals

A solid understanding of Java fundamentals is the cornerstone of every successful technical interview. Regardless of whether you are applying for a junior or senior role, interviewers expect candidates to demonstrate mastery of the language's core building blocks — from primitive data types and their wrapper counterparts, to string handling, access control, and the nuances of equality comparisons.

This chapter covers the foundational topics that arise in virtually every Java interview. Each question-and-answer pair is designed to help you articulate these concepts clearly and precisely, backed by practical code examples that illustrate the key behaviors and edge cases you should be prepared to discuss.

1.1 Java Basics

What is the difference between primitive types and wrapper classes in Java?

Java provides eight primitive types: `int`, `byte`, `short`, `long`, `float`, `double`, `char`, and `boolean`. Each has a corresponding wrapper class in the `java.lang` package (`Integer`, `Byte`, `Short`, `Long`, `Float`, `Double`, `Character`, `Boolean`).

Primitives are stored on the stack (when local) and hold raw values directly, making them more memory-efficient. Wrapper classes are full objects stored on the heap, allowing `null` values and participation in generic collections. Java supports **autoboxing** (automatic conversion from primitive to wrapper) and **unboxing** (wrapper to primitive). A common pitfall is unboxing a `null` wrapper, which throws a `NullPointerException`.

Listing 1.1: Primitive types vs wrapper classes

```java
public class PrimitiveTypesVsWrappers {

    public static void main(String[] args) {
        // --- Primitive types ---
        int primitiveInt = 42;
        double primitiveDouble = 3.14;
        boolean primitiveBoolean = true;
        char primitiveChar = 'A';
```

```java

        // --- Wrapper classes ---
        Integer wrappedInt = 42;            // autoboxing: int -> Integer
        Double wrappedDouble = 3.14;        // autoboxing: double -> Double
        Boolean wrappedBoolean = true;      // autoboxing: boolean -> Boolean
        Character wrappedChar = 'A';        // autoboxing: char -> Character

        // --- Autoboxing and Unboxing ---
        Integer boxed = primitiveInt;       // autoboxing
        int unboxed = boxed;                 // unboxing

        System.out.println("Boxed value: " + boxed);     // 42
        System.out.println("Unboxed value: " + unboxed); // 42

        // --- Null handling: wrappers can be null, primitives cannot ---
        Integer nullableInt = null;  // valid: wrapper can hold null
        // int cannotBeNull = null;  // COMPILE ERROR: primitives cannot be null

        // --- Dangerous unboxing of null ---
        try {
            int danger = nullableInt;  // unboxing null throws NPE
        } catch (NullPointerException e) {
            System.out.println("NullPointerException when unboxing null!");
        }

        // --- Integer cache: values between -128 and 127 are cached ---
        Integer a = 127;
        Integer b = 127;
        System.out.println("127 == 127: " + (a == b));  // true (cached)

        Integer c = 128;
        Integer d = 128;
        System.out.println("128 == 128: " + (c == d));  // false (not cached)
        System.out.println("128 equals 128: " + c.equals(d));  // true

        // --- Memory comparison ---
        // Primitive int: 4 bytes on the stack
        // Integer wrapper: ~16 bytes on the heap + reference on the stack
        System.out.println("Primitives are more memory-efficient than wrappers");
    }
}
```

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

The `==` operator compares **references** for objects (i.e., whether two variables point to the same memory location) and **values** for primitives. The `.equals()` method compares **logical equality** — the actual content of the objects.

For `String`, the string pool can cause `==` to return `true` for literals that share the same interned reference, but this is unreliable for dynamically created strings. The `hashCode()` contract states that if two objects are equal according to `equals()`, they must return the same hash code. Violating this contract breaks hash-based collections.

Listing 1.2: Reference equality vs logical equality

```java
import java.util.Objects;

public class EqualsVsDoubleEquals {

    // Custom class demonstrating proper equals/hashCode
    static class Employee {
        private final int id;
        private final String name;

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

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

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

    public static void main(String[] args) {
        // --- Primitives: == compares values ---
        int x = 10;
        int y = 10;
        System.out.println("Primitive ==: " + (x == y));  // true

        // --- String pool behavior ---
        String s1 = "Hello";      // stored in the string pool
        String s2 = "Hello";      // reuses the same pool reference
        String s3 = new String("Hello");  // new object on the heap

        System.out.println("s1 == s2: " + (s1 == s2));          // true (same pool ref)
        System.out.println("s1 == s3: " + (s1 == s3));          // false (different objects)
        System.out.println("s1.equals(s3): " + s1.equals(s3));  // true (same content)

        // --- Custom class equality ---
        Employee emp1 = new Employee(1, "Alice");
        Employee emp2 = new Employee(1, "Alice");
        Employee emp3 = emp1;

        System.out.println("emp1 == emp2: " + (emp1 == emp2));          // false
        System.out.println("emp1 == emp3: " + (emp1 == emp3));          // true
        System.out.println("emp1.equals(emp2): " + emp1.equals(emp2));  // true

        // --- hashCode contract ---
        System.out.println("emp1 hashCode: " + emp1.hashCode());
        System.out.println("emp2 hashCode: " + emp2.hashCode());
        // Equal objects MUST have the same hash code
        System.out.println("Same hashCode: "
            + (emp1.hashCode() == emp2.hashCode()));  // true
```

```
    }
}
```

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

`String` objects in Java are **immutable**: every modification creates a new `String` instance. `StringBuilder` provides a **mutable** character sequence and is **not thread-safe**, making it the preferred choice for single-threaded string manipulation. `StringBuffer` is functionally equivalent to `StringBuilder` but is **thread-safe** due to synchronized methods, at the cost of additional overhead.

For most use cases, `StringBuilder` is recommended. Use `StringBuffer` only when thread safety is explicitly required.

Listing 1.3: String, StringBuilder, and StringBuffer comparison

```java
public class StringStringBuilderStringBuffer {

    public static void main(String[] args) {
        // --- String is immutable ---
        String original = "Hello";
        String modified = original.concat(" World");
        System.out.println("Original: " + original);   // Hello (unchanged)
        System.out.println("Modified: " + modified);    // Hello World (new object)

        // Each concatenation creates a new String object
        String result = "";
        for (int i = 0; i < 5; i++) {
            result += i + " ";  // creates a new String each iteration
        }
        System.out.println("Loop result: " + result);

        // --- StringBuilder: mutable, NOT thread-safe ---
        StringBuilder sb = new StringBuilder("Hello");
        sb.append(" World");
        sb.append("!");
        sb.insert(5, ",");
        sb.replace(0, 5, "Hi");
        System.out.println("StringBuilder: " + sb.toString());  // Hi, World!

        // Efficient string building in a loop
        StringBuilder efficient = new StringBuilder();
        for (int i = 0; i < 5; i++) {
            efficient.append(i).append(" ");
        }
        System.out.println("Efficient: " + efficient.toString());

        // --- StringBuffer: mutable, thread-safe (synchronized) ---
        StringBuffer buffer = new StringBuffer("Thread");
        buffer.append("-Safe");
        System.out.println("StringBuffer: " + buffer.toString());

        // --- Performance comparison ---
        int iterations = 100_000;

```

```java
        // String concatenation (slow due to immutability)
        long start = System.nanoTime();
        String s = "";
        for (int i = 0; i < iterations; i++) {
            s += "a";
        }
        long stringTime = System.nanoTime() - start;

        // StringBuilder (fast, single-threaded)
        start = System.nanoTime();
        StringBuilder sb2 = new StringBuilder();
        for (int i = 0; i < iterations; i++) {
            sb2.append("a");
        }
        long builderTime = System.nanoTime() - start;

        // StringBuffer (slightly slower due to synchronization)
        start = System.nanoTime();
        StringBuffer buf = new StringBuffer();
        for (int i = 0; i < iterations; i++) {
            buf.append("a");
        }
        long bufferTime = System.nanoTime() - start;

        System.out.println("String:        " + stringTime / 1_000_000 + " ms");
        System.out.println("StringBuilder: " + builderTime / 1_000_000 + " ms");
        System.out.println("StringBuffer:  " + bufferTime / 1_000_000 + " ms");
    }
}
```

How does the `final` keyword work in Java?

The `final` keyword has three distinct uses in Java:

- **Final variables** — once assigned, the value cannot be changed (constants). For object references, the reference cannot be reassigned, but the object's state can still be modified.
- **Final methods** — cannot be overridden by subclasses.
- **Final classes** — cannot be extended (e.g., `String`, `Integer`).

A variable that is not declared `final` but is never reassigned after initialization is called **effectively final** and can be used in lambda expressions.

Listing 1.4: The final keyword in Java

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

public class FinalKeyword {

    // --- Final variable (constant) ---
    static final double PI = 3.14159265358979;

```

```java
    // --- Final class: cannot be extended ---
    static final class ImmutablePoint {
        private final int x;
        private final int y;

        ImmutablePoint(int x, int y) {
            this.x = x;
            this.y = y;
        }

        int getX() { return x; }
        int getY() { return y; }
    }

    // static class ExtendedPoint extends ImmutablePoint {} // COMPILE ERROR

    // --- Final method: cannot be overridden ---
    static class BaseClass {
        final void cannotOverride() {
            System.out.println("This method cannot be overridden");
        }

        void canOverride() {
            System.out.println("Base implementation");
        }
    }

    static class SubClass extends BaseClass {
        // void cannotOverride() {} // COMPILE ERROR: cannot override final method

        @Override
        void canOverride() {
            System.out.println("Overridden implementation");
        }
    }

    public static void main(String[] args) {
        // Final variable: cannot reassign
        final int maxRetries = 3;
        // maxRetries = 5;  // COMPILE ERROR

        // Final reference: reference is fixed, but object state can change
        final List<String> names = new ArrayList<>();
        names.add("Alice");      // OK: modifying the list's content
        names.add("Bob");        // OK
        // names = new ArrayList<>();  // COMPILE ERROR: cannot reassign reference
        System.out.println("Final list contents: " + names);

        // --- Effectively final in lambdas ---
        String greeting = "Hello";  // effectively final (never reassigned)
        Runnable r = () -> System.out.println(greeting + ", World!");
        r.run();

        // If we uncomment the next line, 'greeting' is no longer
        // effectively final and the lambda above would not compile:
        // greeting = "Hi";

        // Final method demonstration
        SubClass sub = new SubClass();
```

```
        sub.cannotOverride();  // calls BaseClass method (cannot be overridden)
        sub.canOverride();     // calls SubClass overridden method
    }
}
```

What are the access modifiers in Java and what is their scope?

Java provides four levels of access control:

- `public` — accessible from any class in any package.
- `protected` — accessible within the same package and by subclasses in other packages.
- **Default (package-private)** — accessible only within the same package. No keyword is used.
- `private` — accessible only within the declaring class.

Choosing the most restrictive access level that satisfies your requirements is a best practice that supports encapsulation and reduces coupling.

Listing 1.5: Access modifiers and their scope

```java
// File: AccessModifiers.java
// Demonstrates all four access levels in a single compilation unit.

public class AccessModifiers {

    // --- public: accessible from anywhere ---
    public String publicField = "I am public";

    // --- protected: accessible in same package + subclasses in other packages ---
    protected String protectedField = "I am protected";

    // --- default (package-private): accessible only within the same package ---
    String defaultField = "I am package-private";

    // --- private: accessible only within this class ---
    private String privateField = "I am private";

    public void demonstrateAccess() {
        // All fields are accessible within the declaring class
        System.out.println(publicField);
        System.out.println(protectedField);
        System.out.println(defaultField);
        System.out.println(privateField);
    }

    // Nested class demonstrating access
    static class SamePackageClass {
        void accessOuter() {
            AccessModifiers obj = new AccessModifiers();
            System.out.println(obj.publicField);     // OK
```

```java
            System.out.println(obj.protectedField);   // OK (same file/package)
            System.out.println(obj.defaultField);     // OK (same package)
            // System.out.println(obj.privateField);  // COMPILE ERROR
        }
    }

    // Subclass demonstrating protected access
    static class SubClass extends AccessModifiers {
        void accessInherited() {
            System.out.println(publicField);          // OK
            System.out.println(protectedField);        // OK (inherited)
            System.out.println(defaultField);          // OK (same package)
            // System.out.println(privateField);       // COMPILE ERROR
        }
    }

    /*
     * Access Modifier Visibility Summary:
     *
     * Modifier      | Class | Package | Subclass | World
     * -------------|-------|---------|----------|------
     * public        |  Yes  |   Yes   |   Yes    |  Yes
     * protected     |  Yes  |   Yes   |   Yes    |  No
     * (default)     |  Yes  |   Yes   |   No*    |  No
     * private       |  Yes  |   No    |   No     |  No
     *
     * *Subclass in the same package can access default members.
     *  Subclass in a different package cannot.
     */

    public static void main(String[] args) {
        AccessModifiers obj = new AccessModifiers();
        obj.demonstrateAccess();

        new SamePackageClass().accessOuter();
        new SubClass().accessInherited();
    }
}
```

What is the difference between static and instance members in Java?

Static members belong to the class itself and are shared across all instances. They are accessed via the class name and exist independently of any object. **Instance members** belong to individual objects and require an instance to be accessed.

Static methods cannot access instance variables or methods directly, because there is no implicit `this` reference. Static initialization blocks run once when the class is loaded.

Listing 1.6: Static vs instance members

```java
public class StaticVsInstance {

    // --- Static field: shared across all instances ---
    static int instanceCount = 0;
    static final String COMPANY = "TechCorp";

```

```java
    // --- Static initialization block: runs once when class is loaded ---
    static {
        System.out.println("Static block executed: class loaded");
    }

    // --- Instance fields: unique to each object ---
    private String name;
    private int id;

    // --- Instance initialization block: runs for each new object ---
    {
        instanceCount++;
        this.id = instanceCount;
        System.out.println("Instance block executed: creating object #" + id);
    }

    StaticVsInstance(String name) {
        this.name = name;
    }

    // --- Static method: belongs to the class ---
    static int getInstanceCount() {
        // Cannot access instance members here:
        // System.out.println(name);  // COMPILE ERROR
        return instanceCount;
    }

    // --- Instance method: belongs to the object ---
    void printInfo() {
        // Can access both static and instance members
        System.out.println("Employee #" + id + ": " + name
            + " at " + COMPANY);
    }

    public static void main(String[] args) {
        // Access static members via class name
        System.out.println("Company: " + StaticVsInstance.COMPANY);
        System.out.println("Count before: " + StaticVsInstance.getInstanceCount());

        // Create instances
        StaticVsInstance emp1 = new StaticVsInstance("Alice");
        StaticVsInstance emp2 = new StaticVsInstance("Bob");
        StaticVsInstance emp3 = new StaticVsInstance("Charlie");

        // Instance methods require an object
        emp1.printInfo();
        emp2.printInfo();
        emp3.printInfo();

        // Static field is shared: reflects total count
        System.out.println("Total employees: "
            + StaticVsInstance.getInstanceCount());  // 3
    }
}
```

1.2 Type System and Conversions

What is type casting in Java and what are the different types?

Type casting converts a value from one type to another. For primitives, **widening** (e.g., `int` to `long`) is implicit and safe, while **narrowing** (e.g., `double` to `int`) requires an explicit cast and may lose precision.

For objects, **upcasting** (subclass to superclass) is implicit and always safe. **Downcasting** (superclass to subclass) requires an explicit cast and may throw a `ClassCastException` at runtime if the actual type does not match. Always use `instanceof` to check before downcasting.

Listing 1.7: Type casting: widening, narrowing, upcasting, and downcasting

```java
public class TypeCasting {

    static class Animal {
        void speak() {
            System.out.println("Animal speaks");
        }
    }

    static class Dog extends Animal {
        @Override
        void speak() {
            System.out.println("Dog barks");
        }

        void fetch() {
            System.out.println("Dog fetches the ball");
        }
    }

    static class Cat extends Animal {
        @Override
        void speak() {
            System.out.println("Cat meows");
        }
    }

    public static void main(String[] args) {
        // ===== PRIMITIVE TYPE CASTING =====

        // --- Widening (implicit): smaller type to larger type ---
        int intValue = 100;
        long longValue = intValue;       // int -> long (automatic)
        double doubleValue = longValue;  // long -> double (automatic)
        System.out.println("Widened: " + intValue + " -> "
            + longValue + " -> " + doubleValue);

        // --- Narrowing (explicit): larger type to smaller type ---
        double pi = 3.14159;
        int truncated = (int) pi;        // double -> int (loses decimal)
        System.out.println("Narrowed: " + pi + " -> " + truncated);  // 3

        long bigNumber = 130;
```

```java
        byte smallByte = (byte) bigNumber;  // may overflow
        System.out.println("Narrowed long to byte: " + bigNumber
            + " -> " + smallByte);  // -126 (overflow!)

        // ===== OBJECT TYPE CASTING =====

        // --- Upcasting (implicit): subclass to superclass ---
        Dog dog = new Dog();
        Animal animal = dog;  // upcasting is always safe
        animal.speak();       // "Dog barks" (polymorphism still works)
        // animal.fetch();    // COMPILE ERROR: Animal has no fetch()

        // --- Downcasting (explicit): superclass to subclass ---
        // Must verify the actual type first using instanceof
        Animal myAnimal = new Dog();  // actual type is Dog

        if (myAnimal instanceof Dog myDog) {
            // Pattern matching for instanceof (Java 16+)
            myDog.fetch();  // safe: we know it is a Dog
        }

        // --- Unsafe downcast throws ClassCastException ---
        Animal cat = new Cat();
        try {
            Dog notADog = (Dog) cat;  // ClassCastException at runtime!
        } catch (ClassCastException e) {
            System.out.println("ClassCastException: Cannot cast Cat to Dog");
        }
    }
}
```

What are varargs in Java and how do they work?

Variable-length arguments (**varargs**) allow a method to accept zero or more arguments of a specified type using the `...` syntax. Under the hood, the compiler converts varargs into an array. Key rules:

- A method can have at most one varargs parameter.
- The varargs parameter must be the last parameter in the method signature.
- Varargs can be called with an explicit array or with comma-separated values.

Listing 1.8: Varargs usage and rules

```java
import java.util.Arrays;

public class Varargs {

    // Varargs parameter must be the last parameter
    static int sum(String label, int... numbers) {
        System.out.print(label + ": ");
        int total = 0;
```

```java
        for (int n : numbers) {
            total += n;
        }
        return total;
    }

    // Varargs with different types
    static void printAll(Object... items) {
        System.out.println("Received " + items.length + " items:");
        for (Object item : items) {
            System.out.println("  " + item + " (" + item.getClass().getSimpleName() + ")");
        }
    }

    // Overloaded method: specific version is preferred over varargs
    static void greet(String name) {
        System.out.println("Hello, " + name + "! (specific method)");
    }

    static void greet(String... names) {
        System.out.println("Hello to: " + Arrays.toString(names)
            + " (varargs method)");
    }

    public static void main(String[] args) {
        // Calling with multiple arguments
        System.out.println(sum("Three numbers", 1, 2, 3));  // 6

        // Calling with a single argument
        System.out.println(sum("One number", 42));  // 42

        // Calling with no arguments (empty array)
        System.out.println(sum("No numbers"));  // 0

        // Calling with an explicit array
        int[] values = {10, 20, 30, 40};
        System.out.println(sum("From array", values));  // 100

        // Mixed types via Object varargs
        printAll("Java", 21, 3.14, true);

        // Overload resolution: specific method wins over varargs
        greet("Alice");             // calls greet(String)
        greet("Alice", "Bob");      // calls greet(String...)
        greet();                    // calls greet(String...)
    }
}
```

1.3 Strings and Immutability

How does the `String` Pool work and why are Strings immutable in Java?

The `String` **Pool** (also called the intern pool) is a special memory region where the JVM stores string literals. When a string literal is created, the JVM checks the pool first and reuses an existing

instance if one with the same content already exists. This is why two string literals with identical content compared with `==` return `true` — they reference the same pooled object. However, strings created with `new String("...")` allocate a distinct object on the heap, so `==` returns `false` even when the content matches. The `intern()` method explicitly adds a string to the pool or returns the existing pooled reference.

String immutability in Java serves four critical purposes. First, **thread safety**: immutable strings can be shared across threads without synchronization. Second, **security**: strings are used for network connections, file paths, and class names, and immutability prevents malicious modification after validation. Third, **caching**: the `hashCode()` of a `String` is computed once and cached internally, making strings highly efficient as `HashMap` keys. Fourth, **String Pool safety**: immutability is a prerequisite for safe interning, because if one reference could mutate a pooled string, every other reference sharing that instance would see the change.

A subtle but important detail involves **compile-time constant folding**. The expression `"hel" + "lo"` is resolved by the compiler to `"hello"` and matched to the pool, whereas concatenation involving a variable (e.g., `"hel" + part`) produces a new runtime object. The practical takeaway is to always use `.equals()` for string comparison and to avoid `new String("literal")` since it creates an unnecessary heap object.

Listing 1.9: String Pool, interning, immutability, and constant folding

```java
import java.lang.ref.WeakReference;

public class StringPoolAndImmutability {
    public static void main(String[] args) {
        // === String Pool (interning) ===
        // String literals are stored in a special memory area called the String Pool.
        // When you create a literal, the JVM checks the pool first and reuses
        // an existing instance if one with the same content already exists.

        String a = "hello";           // placed in the String Pool
        String b = "hello";           // reuses the same pooled instance
        String c = new String("hello"); // creates a NEW object on the heap

        System.out.println(a == b);        // true  - same pool reference
        System.out.println(a == c);        // false - c is a distinct heap object
        System.out.println(a.equals(c));  // true  - same logical content

        // intern() explicitly adds a string to the pool (or returns the pooled copy)
        String d = c.intern();
        System.out.println(a == d);        // true  - d now points to the pooled "hello"

        // === Why are Strings immutable? ===
        // 1. Thread safety   - Strings can be shared across threads without synchronization.
        // 2. Security         - Network connections, file paths, and class names use Strings;
        //                       immutability prevents malicious modification after validation.
        // 3. Caching          - hashCode() is computed once and cached, making Strings
        //                       efficient HashMap keys.
        // 4. String Pool      - Immutability is a prerequisite for safe interning; if one
        //                       reference could mutate a pooled String, every other reference
        //                       sharing that instance would see the change.

```

```java
        // === hashCode caching demonstration ===
        String key = "cacheDemo";
        int h1 = key.hashCode();   // computed and cached internally
        int h2 = key.hashCode();   // returned from cache (no recomputation)
        System.out.println("hashCode equal: " + (h1 == h2)); // always true

        // === Common mistake: unnecessary object creation ===
        // BAD  - creates an extra heap object for no reason
        String bad = new String("literal");

        // GOOD - use the literal directly; it is already pooled
        String good = "literal";

        // === Concatenation and the pool ===
        String x = "hel" + "lo";   // compile-time constant folding -> "hello"
        System.out.println(a == x); // true  - resolved at compile time

        String part = "lo";
        String y = "hel" + part;    // runtime concatenation -> new object
        System.out.println(a == y); // false - not the same pooled instance

        // === Practical tip: use equals(), never == for Strings ===
        System.out.println("Always compare Strings with .equals(), not ==");
    }
}
```

What are text blocks in Java and how do they handle whitespace?

Text blocks, previewed in Java 13 and standardized in Java 15, provide a cleaner way to write multi-line string literals. They are delimited by triple double-quote markers (`"""`) and eliminate the need for most escape sequences and string concatenation. The opening `"""` must be followed by a newline — the content begins on the next line.

The most important aspect of text blocks is **incidental whitespace removal**. The compiler determines the common leading whitespace based on the position of the closing `"""` delimiter and strips it from every line. This means the indentation of the closing delimiter controls how much leading whitespace is preserved in the resulting string. If the closing delimiter is aligned with the content, no leading whitespace is retained; if it is shifted further left, the difference is preserved as leading spaces.

Text blocks support two new escape sequences: the **line continuation** character (`\` at the end of a line) suppresses the line terminator, allowing a logical single line to be split across multiple source lines; and `\s`, which is an explicit space that prevents trailing-space stripping. Text blocks work seamlessly with `String.formatted()` for template-style interpolation. Importantly, text blocks are regular `String` objects at runtime — they are immutable and interned just like normal string literals, with all processing happening entirely at compile time.

Listing 1.10: Text blocks: multi-line strings, whitespace control, and escape sequences

```java
/**
 * Text Blocks (Java 13 preview, standard since Java 15)
```

```
 *
 * Text blocks provide a cleaner way to write multi-line string literals,
 * eliminating the need for most escape sequences and concatenation.
 */
public class TextBlocks {
    public static void main(String[] args) {
        // === Traditional multi-line string (pre-Java 15) ===
        String oldJson = "{\n" +
                "  \"name\": \"Alice\",\n" +
                "  \"age\": 30\n" +
                "}";
        System.out.println("Traditional:\n" + oldJson);

        // === Text block equivalent ===
        // Delimited by """ ... """ -- the opening """ must be followed by a newline.
        String newJson = """
                {
                  "name": "Alice",
                  "age": 30
                }""";
        System.out.println("\nText block:\n" + newJson);

        // Content is identical
        System.out.println("Equal: " + oldJson.equals(newJson)); // true

        // === Incidental whitespace removal ===
        // The compiler strips common leading whitespace (incidental indentation)
        // based on the position of the closing """.
        String indented = """
                Line A
                Line B
                """;
        // Both lines keep zero leading spaces because the closing """ aligns
        // with the leftmost content column.

        String shifted = """
                    Line A
                    Line B
            """;
        // Here the closing """ is further left, so each line retains 4 extra spaces.
        System.out.println("Shifted:\n" + shifted);

        // === Escape sequences still work ===
        String withEscapes = """
                First line\tTabbed
                Second line \\ backslash
                "No need to escape double quotes"
                Unless you need three: \"""
                """;
        System.out.println(withEscapes);

        // === New escape: \ suppresses the line terminator ===
        String singleLine = """
                This is actually \
                a single line.""";
        System.out.println(singleLine); // "This is actually a single line."

        // === New escape: \s is an explicit space (prevents trailing-space stripping) ===
        String trailing = """
```

```
                Value:  \s
                """;
        // Without \s the two trailing spaces would be stripped.
        System.out.println("Trailing preserved length: " + trailing.lines().findFirst().get().length());

        // === Using text blocks with String.formatted() ===
        String template = """
                Dear %s,
                Your order #%d has shipped.
                """.formatted("Bob", 42);
        System.out.println(template);

        // === Common interview pitfall ===
        // Text blocks are still regular String objects -- they are immutable and
        // interned just like normal string literals. There is no runtime cost
        // difference; the processing happens entirely at compile time.
    }
}
```

1.4 Enums and Constants

What are enums in Java and why are they preferred over integer constants?

Enums (enumerated types) are a special class type in Java that represents a fixed set of constants. Unlike integer constants, enums provide **type safety** — a method accepting an enum parameter can only receive one of the declared constants, not an arbitrary integer. Enums also provide a **namespace**, avoiding the naming collisions common with `static final int` constants.

Every enum implicitly extends `java.lang.Enum`, which provides several built-in methods: `name()` returns the constant's declared name as a string, `ordinal()` returns its position (zero-based), `valueOf(String)` performs a reverse lookup, and `values()` returns an array of all constants in declaration order. Enums are **singletons** — each constant is instantiated exactly once, so the `==` operator is safe and preferred over `.equals()` for enum comparison.

Enums can carry data by defining fields and a private constructor. A common interview example is a `Planet` enum with `mass` and `radius` fields that computes surface gravity and surface weight. Key pitfalls include: never relying on `ordinal()` for persistent storage (adding a constant shifts all subsequent ordinals), `valueOf()` throwing `IllegalArgumentException` for unknown names, and enum constructors being implicitly private.

Listing 1.11: Enums: type safety, built-in methods, and enums with data fields

```
/**
 * Enums in Java -- type-safe constants with built-in functionality.
 */
public class Enums {

    // === Basic enum ===
    enum Direction { NORTH, SOUTH, EAST, WEST }

    // === Enum with fields, constructor, and methods ===
```

```java
    enum Planet {
        MERCURY(3.303e+23, 2.4397e6),
        VENUS(4.869e+24, 6.0518e6),
        EARTH(5.976e+24, 6.37814e6);

        private final double mass;    // in kilograms
        private final double radius;  // in meters

        // Enum constructors are implicitly private
        Planet(double mass, double radius) {
            this.mass = mass;
            this.radius = radius;
        }

        // Gravitational constant
        static final double G = 6.67300E-11;

        double surfaceGravity() {
            return G * mass / (radius * radius);
        }

        double surfaceWeight(double otherMass) {
            return otherMass * surfaceGravity();
        }
    }

    public static void main(String[] args) {
        // === Why enums over constants ===
        // BAD: int constants -- no type safety, no namespace
        final int DIRECTION_NORTH = 0;
        final int DIRECTION_SOUTH = 1;
        // A method accepting int can receive ANY int, not just valid directions.

        // GOOD: enum -- compiler enforces valid values
        Direction dir = Direction.NORTH;

        // === Built-in methods ===
        System.out.println("name():     " + dir.name());         // "NORTH"
        System.out.println("ordinal(): " + dir.ordinal());      // 0
        System.out.println("valueOf(): " + Direction.valueOf("EAST")); // EAST

        // values() returns all constants in declaration order
        for (Direction d : Direction.values()) {
            System.out.println(d + " ordinal=" + d.ordinal());
        }

        // === Enum identity ===
        // Enums are singletons; == is safe and preferred over equals()
        System.out.println(dir == Direction.NORTH);  // true
        System.out.println(dir.equals(Direction.NORTH)); // also true, but == is idiomatic

        // === Enums in switch ===
        switch (dir) {
            case NORTH -> System.out.println("Heading north");
            case SOUTH -> System.out.println("Heading south");
            default    -> System.out.println("Heading " + dir);
        }

        // === Enum with data ===
```

```java
        double earthWeight = 75.0;
        double mass = earthWeight / Planet.EARTH.surfaceGravity();
        for (Planet p : Planet.values()) {
            System.out.printf("Your weight on %s is %6.2f N%n", p, p.surfaceWeight(mass));
        }

        // === Common pitfalls ===
        // 1. Never rely on ordinal() for persistent storage -- adding a new constant
        //    shifts all subsequent ordinals.
        // 2. valueOf() throws IllegalArgumentException for unknown names.
        // 3. Enum constructors cannot be public or protected.
    }
}
```

How can enums implement interfaces and provide per-constant behavior?

One of the most powerful and often underappreciated features of Java enums is their ability to implement interfaces and provide **per-constant method bodies**. This makes enums an excellent vehicle for the **strategy pattern** and the **command pattern**, eliminating fragile `if/else` or `switch` chains.

When an enum declares an **abstract method**, every constant must provide its own implementation in an anonymous class body. For example, an `Operation` enum with constants `ADD`, `SUBTRACT`, `MULTIPLY`, and `DIVIDE` can require each constant to implement `apply(double, double)` with its own logic. An alternative approach stores a `java.util.function.UnaryOperator` or similar functional interface as a field, passing the behavior via a lambda in the constructor. Both approaches produce polymorphic enums where calling the same method on different constants yields different behavior.

Enums can also implement any number of interfaces (but cannot extend a class, since they implicitly extend `java.lang.Enum`). This allows enums to be used polymorphically through the interface type. Per-constant method bodies create anonymous subclasses of the enum at the bytecode level, and all enum constants are inherently thread-safe singletons.

Listing 1.12: Enums with behavior: interface implementation and strategy pattern

```java
/**
 * Enums implementing interfaces and providing per-constant behavior.
 */
public class EnumWithBehavior {

    // === Interface that an enum can implement ===
    interface Describable {
        String describe();
    }

    // === Enum implementing an interface with a shared method ===
    enum Severity implements Describable {
        LOW, MEDIUM, HIGH, CRITICAL;

        @Override
        public String describe() {
```

```java
            return name().charAt(0) + name().substring(1).toLowerCase() + " severity";
        }
    }

    // === Strategy pattern using enums ===
    // Each constant overrides the abstract method to provide its own behavior.
    enum Operation {
        ADD {
            @Override public double apply(double a, double b) { return a + b; }
        },
        SUBTRACT {
            @Override public double apply(double a, double b) { return a - b; }
        },
        MULTIPLY {
            @Override public double apply(double a, double b) { return a * b; }
        },
        DIVIDE {
            @Override public double apply(double a, double b) {
                if (b == 0) throw new ArithmeticException("Division by zero");
                return a / b;
            }
        };

        // Abstract method forces every constant to provide an implementation
        public abstract double apply(double a, double b);
    }

    // === Enum with functional interface field (alternative to abstract methods) ===
    enum Formatter {
        UPPER(String::toUpperCase),
        LOWER(String::toLowerCase),
        TRIM(String::trim);

        private final java.util.function.UnaryOperator<String> fn;

        Formatter(java.util.function.UnaryOperator<String> fn) {
            this.fn = fn;
        }

        public String format(String input) {
            return fn.apply(input);
        }
    }

    public static void main(String[] args) {
        // === Interface implementation ===
        for (Severity s : Severity.values()) {
            System.out.println(s.describe());
        }

        // === Strategy pattern in action ===
        double a = 10, b = 3;
        for (Operation op : Operation.values()) {
            System.out.printf("%s.apply(%.0f, %.0f) = %.2f%n", op, a, b, op.apply(a, b));
        }

        // === Functional approach ===
        String input = "  Hello World  ";
        for (Formatter f : Formatter.values()) {
```

```
            System.out.printf("%-5s -> \"%s\"%n", f, f.format(input));
        }

        // === Polymorphism: enums as interface types ===
        Describable d = Severity.CRITICAL;
        System.out.println("Polymorphic call: " + d.describe());

        // === Key interview points ===
        // 1. Enums can implement any number of interfaces but cannot extend a class
        //    (they implicitly extend java.lang.Enum).
        // 2. Per-constant method bodies create anonymous subclasses of the enum.
        // 3. The strategy/command pattern with enums eliminates fragile if/else chains.
        // 4. Enum constants are instantiated once and are inherently thread-safe singletons.
    }
}
```

1.5 Object Class Methods

What is the purpose of `toString()` and what are the best practices for overriding it?

The `toString()` method, defined in `java.lang.Object`, returns a string representation of an object. The default implementation produces the class name followed by the `@` symbol and the hexadecimal hash code (e.g., `Person@1a2b3c4d`), which is almost never useful for debugging or logging. Overriding `toString()` is considered a best practice for every domain class.

A well-crafted `toString()` should follow the pattern `ClassName{field=value, field=value}` and include all important fields. The `StringJoiner` class provides a clean way to build this format. It is critical to **mask sensitive data** such as passwords, tokens, and API keys — since `toString()` is called implicitly in many contexts (logging, string concatenation, `printf` formatting), an exposed password in `toString()` can easily leak to log files.

Key interview points include: `toString()` is called implicitly by `println()`, string concatenation (`+`), and `String.format()`; records (Java 16+) auto-generate a useful `toString()` from their components; and expensive computations should be avoided in `toString()` since it may be called frequently during debugging and logging.

Listing 1.13: toString(): default behavior, best practices, and sensitive data masking

```
import java.util.StringJoiner;

/**
 * The toString() method -- purpose, default behavior, and best practices.
 */
public class ToStringMethod {

    // === Without overriding toString() ===
    static class BadExample {
        String name;
        int age;
```

```java
        BadExample(String name, int age) { this.name = name; this.age = age; }
        // Inherits Object.toString() -> "BadExample@1a2b3c4d" (class@hexHashCode)
    }

    // === With a well-crafted toString() ===
    static class Person {
        private final String name;
        private final int age;

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

        @Override
        public String toString() {
            // Follow the pattern: ClassName{field=value, field=value}
            return new StringJoiner(", ", Person.class.getSimpleName() + "{", "}")
                    .add("name='" + name + "'")
                    .add("age=" + age)
                    .toString();
        }
    }

    // === Sensitive data: never expose passwords or tokens ===
    static class UserCredentials {
        private final String username;
        private final String password;

        UserCredentials(String username, String password) {
            this.username = username;
            this.password = password;
        }

        @Override
        public String toString() {
            // GOOD: mask sensitive fields
            return "UserCredentials{username='" + username + "', password=***}";
        }
    }

    public static void main(String[] args) {
        // Default toString() is not useful for debugging
        BadExample bad = new BadExample("Alice", 30);
        System.out.println("Default toString: " + bad); // BadExample@<hash>

        // Overridden toString() gives clear, readable output
        Person person = new Person("Alice", 30);
        System.out.println("Custom toString:  " + person); // Person{name='Alice', age=30}

        // toString() is called implicitly in many contexts
        System.out.println(person);                         // println calls toString()
        String msg = "User: " + person;                      // concatenation calls toString()
        System.out.printf("Formatted: %s%n", person);       // %s calls toString()

        // Security consideration
        UserCredentials cred = new UserCredentials("admin", "s3cret");
        System.out.println(cred); // password is masked

        // === Interview key points ===
        // 1. Always override toString() for domain classes -- it vastly improves
        //    debuggability and log readability.
```

```
        // 2. Include all important fields but mask sensitive data.
        // 3. The format should be self-describing and parseable if needed.
        // 4. Records (Java 16+) auto-generate a useful toString() from components.
        // 5. Avoid expensive computations in toString() -- it may be called frequently
        //    in logging and debugging.
    }
}
```

What are the pitfalls of `clone()` in Java and what alternatives exist?

The `clone()` method in Java is one of the most problematic APIs in the language. `Cloneable` is a **marker interface** with no methods — it only signals to `Object.clone()` that it should not throw `CloneNotSupportedException`. The default `Object.clone()` performs a **shallow copy**: primitive fields are copied by value, but object references are shared between the original and the clone.

This shallow copy semantics leads to a common pitfall: if a cloned object contains mutable fields (such as a `List`), both the original and the clone share the same list instance. Modifying the list through one reference affects the other. To produce a correct clone, every mutable field must be **deep-copied** explicitly in the overridden `clone()` method — for example, `copy.hobbies = new ArrayList<>(this.hobbies)`.

The recommended alternative, as advocated by Joshua Bloch in *Effective Java* (Item 13), is to use a **copy constructor** or a **static factory method**. Copy constructors are explicit, type-safe, and do not rely on the fragile `Cloneable` contract. They also properly invoke constructors, ensuring that all class invariants are checked. Another important consideration is that `clone()` bypasses constructors entirely, which can leave objects in an inconsistent state if the constructor enforces invariants.

Listing 1.14: clone() pitfalls: shallow vs deep copy and copy constructor alternative

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

/**
 * The clone() method -- shallow vs deep copy, Cloneable contract, and safer alternatives.
 */
public class ClonePitfalls {

    // === Shallow clone: mutable fields are shared ===
    static class ShallowPerson implements Cloneable {
        String name;
        List<String> hobbies;

        ShallowPerson(String name, List<String> hobbies) {
            this.name = name;
            this.hobbies = hobbies;
        }

        @Override
        public ShallowPerson clone() {
            try {
                return (ShallowPerson) super.clone(); // shallow copy
```

```java
            } catch (CloneNotSupportedException e) {
                throw new AssertionError(); // cannot happen if Cloneable is implemented
            }
        }
    }

    // === Deep clone: mutable fields are independently copied ===
    static class DeepPerson implements Cloneable {
        String name;                    // immutable -- safe to share
        List<String> hobbies;           // mutable -- must be deep-copied

        DeepPerson(String name, List<String> hobbies) {
            this.name = name;
            this.hobbies = hobbies;
        }

        @Override
        public DeepPerson clone() {
            try {
                DeepPerson copy = (DeepPerson) super.clone();
                copy.hobbies = new ArrayList<>(this.hobbies); // deep copy the list
                return copy;
            } catch (CloneNotSupportedException e) {
                throw new AssertionError();
            }
        }
    }

    // === Preferred alternative: copy constructor ===
    static class BetterPerson {
        final String name;
        final List<String> hobbies;

        BetterPerson(String name, List<String> hobbies) {
            this.name = name;
            this.hobbies = new ArrayList<>(hobbies);
        }

        // Copy constructor -- explicit, type-safe, no Cloneable contract
        BetterPerson(BetterPerson other) {
            this.name = other.name;
            this.hobbies = new ArrayList<>(other.hobbies);
        }
    }

    public static void main(String[] args) {
        // --- Shallow clone pitfall ---
        List<String> hobbies = new ArrayList<>(List.of("Reading", "Cycling"));
        ShallowPerson original = new ShallowPerson("Alice", hobbies);
        ShallowPerson shallowCopy = original.clone();

        shallowCopy.hobbies.add("Gaming");
        System.out.println("Original hobbies:    " + original.hobbies);    // [Reading, Cycling, Gaming]
            -- OOPS
        System.out.println("ShallowCopy hobbies: " + shallowCopy.hobbies); // [Reading, Cycling, Gaming]

        // --- Deep clone correctness ---
        List<String> hobbies2 = new ArrayList<>(List.of("Reading", "Cycling"));
        DeepPerson orig2 = new DeepPerson("Bob", hobbies2);
```

```java
        DeepPerson deepCopy = orig2.clone();

        deepCopy.hobbies.add("Gaming");
        System.out.println("\nOriginal hobbies:  " + orig2.hobbies);    // [Reading, Cycling] -- safe
        System.out.println("DeepCopy hobbies:  " + deepCopy.hobbies);   // [Reading, Cycling, Gaming]

        // --- Copy constructor (recommended) ---
        BetterPerson bp = new BetterPerson("Carol", List.of("Painting"));
        BetterPerson bpCopy = new BetterPerson(bp);
        System.out.println("\nCopy-constructor copy: " + bpCopy.hobbies);

        // === Interview key points ===
        // 1. Cloneable is a marker interface with no methods -- it only signals
        //    that Object.clone() should not throw CloneNotSupportedException.
        // 2. Object.clone() performs a SHALLOW copy -- primitives are copied,
        //    but object references are shared.
        // 3. Always deep-copy mutable fields in your clone() override.
        // 4. Prefer copy constructors or static factory methods over clone()
        //    (Joshua Bloch, Effective Java, Item 13).
        // 5. clone() bypasses constructors, which can leave invariants unchecked.
    }
}
```

1.6 Advanced Fundamentals

Is Java pass-by-value or pass-by-reference?

Java is **always pass-by-value**. This is one of the most frequently asked and most commonly misunderstood topics in Java interviews. For primitive types, a copy of the value is passed, so modifications inside the method have no effect on the caller's variable. For object references, a copy of the **reference** (pointer) is passed — the reference itself is passed by value.

The practical consequence is twofold. You **can** modify the internal state of an object received as a parameter (e.g., adding elements to a list), because both the caller and the method hold references to the same heap object. However, you **cannot** make the caller's variable point to a different object by reassigning the parameter inside the method — the reassignment only affects the local copy of the reference. This is conclusively demonstrated by the **swap test**: a method that swaps two `StringBuilder` parameters has no effect on the caller's variables, which would be impossible if Java used pass-by-reference.

Strings are a common interview trap in this context. Since strings are immutable, reassigning the parameter inside a method (e.g., `s = "changed"`) creates a new string object and rebinds the local reference, leaving the caller's original string unaffected. This behavior is consistent with pass-by-value semantics, not pass-by-reference.

Listing 1.15: Pass-by-value: primitives, object mutation, reference reassignment, and the swap test

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

```

```java
/**
 * Java is ALWAYS pass-by-value.
 *
 * For primitives, a copy of the value is passed.
 * For objects, a copy of the REFERENCE (pointer) is passed -- the reference
 * itself is passed by value, so you cannot make the caller's variable point
 * to a different object, but you CAN modify the object's internal state.
 */
public class PassByValue {

    // === Primitive: caller's variable is unaffected ===
    static void tryToChange(int x) {
        x = 99;
        System.out.println("  Inside method: x = " + x);
    }

    // === Object reference: the object's state CAN be modified ===
    static void addElement(List<String> list) {
        list.add("new");
        System.out.println("  Inside method: list = " + list);
    }

    // === Object reference: reassigning the reference does NOT affect the caller ===
    static void tryToReassign(List<String> list) {
        list = new ArrayList<>();      // local copy of reference now points elsewhere
        list.add("only inside method");
        System.out.println("  Inside method: list = " + list);
    }

    // === Common interview trick: String reassignment ===
    static void tryToChangeString(String s) {
        s = "changed";                 // s is a local copy -- caller is unaffected
        System.out.println("  Inside method: s = " + s);
    }

    // === Swap demonstration: proves pass-by-reference does NOT exist ===
    static void swap(StringBuilder a, StringBuilder b) {
        StringBuilder temp = a;
        a = b;
        b = temp;
        System.out.println("  Inside swap: a=" + a + ", b=" + b);
    }

    public static void main(String[] args) {
        // 1. Primitive
        int num = 42;
        System.out.println("Before tryToChange: num = " + num);
        tryToChange(num);
        System.out.println("After  tryToChange: num = " + num); // still 42

        // 2. Object mutation (state changes visible to caller)
        List<String> myList = new ArrayList<>(List.of("a", "b"));
        System.out.println("\nBefore addElement: " + myList);
        addElement(myList);
        System.out.println("After  addElement: " + myList); // [a, b, new]

        // 3. Reference reassignment (NOT visible to caller)
        List<String> myList2 = new ArrayList<>(List.of("x", "y"));
        System.out.println("\nBefore tryToReassign: " + myList2);
```

```java
        tryToReassign(myList2);
        System.out.println("After  tryToReassign: " + myList2); // still [x, y]

        // 4. Strings -- immutable, so reassignment inside method has no effect
        String text = "original";
        System.out.println("\nBefore tryToChangeString: " + text);
        tryToChangeString(text);
        System.out.println("After  tryToChangeString: " + text); // still "original"

        // 5. Swap fails -- proving Java is not pass-by-reference
        StringBuilder sa = new StringBuilder("FIRST");
        StringBuilder sb = new StringBuilder("SECOND");
        System.out.println("\nBefore swap: sa=" + sa + ", sb=" + sb);
        swap(sa, sb);
        System.out.println("After  swap: sa=" + sa + ", sb=" + sb); // unchanged
    }
}
```

What are static and instance initialization blocks and in what order do they execute?

Java provides two types of initialization blocks that run code during class loading and object creation. **Static initialization blocks** (enclosed in `static { ... }`) run once when the class is first loaded by the JVM. **Instance initialization blocks** (enclosed in `{ ... }`) run every time a new object is created, before the constructor body.

The complete execution order is: (1) static fields and static blocks in declaration order (once per class load), (2) instance fields and instance blocks in declaration order (each time an object is created), and (3) the constructor body. The compiler copies each instance initialization block into every constructor at the bytecode level, which is why they execute before every constructor regardless of which constructor is invoked.

Static blocks are commonly used for **one-time expensive initialization**, such as loading configuration files, populating lookup tables, or initializing static final collections. Instance blocks are useful when **multiple constructors share common setup logic**, although delegating constructors with `this()` are often considered a clearer alternative. A key interview point is that static blocks do not execute again when subsequent instances are created — they run exactly once during class loading.

Listing 1.16: Initialization blocks: execution order, static vs instance, and practical use cases

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

/**
 * Static and instance initialization blocks -- execution order and use cases.
 *
 * Execution order:
 * 1. Static fields and static blocks (in declaration order) -- once per class load.
 * 2. Instance fields and instance blocks (in declaration order) -- each time an object is created.
 * 3. Constructor body.
```

```java
 */
public class InitializationBlocks {

    // Static field
    static List<String> log = new ArrayList<>();

    // Static initialization block -- runs once when the class is loaded
    static {
        log.add("1. Static block executed");
        System.out.println(log.get(log.size() - 1));
    }

    // Instance field with inline initializer
    String instanceField = logAndReturn("2. Instance field initialized");

    // Instance initialization block -- runs before every constructor
    {
        log.add("3. Instance block executed");
        System.out.println(log.get(log.size() - 1));
    }

    // Constructor
    InitializationBlocks(String label) {
        log.add("4. Constructor executed (" + label + ")");
        System.out.println(log.get(log.size() - 1));
    }

    // Second static block -- runs after the first, still during class loading
    static {
        log.add("1b. Second static block executed");
        System.out.println(log.get(log.size() - 1));
    }

    private static String logAndReturn(String msg) {
        log.add(msg);
        System.out.println(msg);
        return msg;
    }

    // === Practical use case: static block for expensive one-time setup ===
    static final List<String> VALID_CODES;
    static {
        // Simulate loading configuration once
        VALID_CODES = List.of("A01", "B02", "C03");
        log.add("1c. VALID_CODES loaded: " + VALID_CODES);
        System.out.println(log.get(log.size() - 1));
    }

    // === Practical use case: instance block for shared setup across constructors ===
    static class MultiConstructor {
        final long createdAt;

        // Common setup shared by ALL constructors
        {
            createdAt = System.nanoTime();
            System.out.println("   Instance block: createdAt = " + createdAt);
        }

        MultiConstructor()           { System.out.println("   No-arg constructor"); }
```

```java
        MultiConstructor(String s)  { System.out.println("   String constructor: " + s); }
    }

    public static void main(String[] args) {
        System.out.println("\n--- Creating first instance ---");
        new InitializationBlocks("first");

        System.out.println("\n--- Creating second instance ---");
        new InitializationBlocks("second");
        // Note: static blocks do NOT run again.

        System.out.println("\n--- MultiConstructor demo ---");
        new MultiConstructor();
        new MultiConstructor("hello");

        // === Interview key points ===
        // 1. Static blocks run in textual order, once, when the class is loaded.
        // 2. Instance blocks run in textual order before EVERY constructor.
        // 3. Instance blocks are copied by the compiler into every constructor.
        // 4. Use static blocks for one-time resource initialization.
        // 5. Use instance blocks when multiple constructors share common logic
        //    (though delegating constructors with this() is often clearer).
    }
}
```

How do the this and super keywords work in Java?

The `this` keyword refers to the current object instance and serves three primary purposes: disambiguating instance fields from method parameters (e.g., `this.name = name`), enabling **constructor chaining** via `this(args)` which delegates to another constructor in the same class, and enabling `fluent APIs` by returning `this` from methods for method chaining.

The `super` keyword refers to the parent class and is used to call the parent constructor (`super(args)`), invoke an overridden parent method (`super.method()`), and access hidden parent fields (`super.field`). A critical rule is that `super()` or `this()` must be the **first statement** in a constructor, and you cannot use both in the same constructor. If neither is explicitly written, the compiler inserts a no-argument `super()` call, which fails at compile time if the parent class has no no-argument constructor.

An important distinction is that `super.method()` bypasses polymorphism and directly calls the parent's implementation, whereas a regular method call uses dynamic dispatch. Fields, unlike methods, are **not polymorphic**: `super.field` accesses the parent's field directly, and if a subclass declares a field with the same name, it **hides** (not overrides) the parent's field.

Listing 1.17: this and super: constructor chaining, method dispatch, and field hiding

```java
/**
 * The this and super keywords -- references, constructor chaining, and method dispatch.
 */
public class ThisAndSuper {

    // === Base class ===
```

```java
    static class Animal {
        String name;

        Animal(String name) {
            this.name = name; // 'this' disambiguates field from parameter
            System.out.println("Animal constructor: " + this.name);
        }

        void speak() {
            System.out.println(name + " makes a sound");
        }

        void info() {
            System.out.println("Animal.info(): " + name);
        }
    }

    // === Subclass ===
    static class Dog extends Animal {
        String breed;

        // Constructor chaining with this() and super()
        Dog(String name, String breed) {
            super(name);       // MUST be the first statement -- calls Animal(String)
            this.breed = breed;
            System.out.println("Dog constructor: " + this.breed);
        }

        // Delegating constructor with this()
        Dog(String name) {
            this(name, "Unknown"); // Delegates to Dog(String, String)
        }

        @Override
        void speak() {
            System.out.println(name + " barks");
        }

        @Override
        void info() {
            super.info();      // Explicitly calls Animal.info()
            System.out.println("Dog.info(): breed=" + breed);
        }

        // Returning this for fluent APIs
        Dog printBreed() {
            System.out.println("Breed: " + breed);
            return this;       // enables method chaining
        }

        Dog printName() {
            System.out.println("Name: " + name);
            return this;
        }
    }

    // === Demonstrating super with field hiding (not overriding) ===
    static class Parent {
        String label = "Parent-label";
```

```java
        static String staticField = "Parent-static";
    }

    static class Child extends Parent {
        String label = "Child-label";          // hides Parent.label
        static String staticField = "Child-static"; // hides Parent.staticField

        void showLabels() {
            System.out.println("this.label:  " + this.label);   // Child-label
            System.out.println("super.label: " + super.label);  // Parent-label
        }
    }

    public static void main(String[] args) {
        // === Constructor chaining ===
        System.out.println("--- Two-arg constructor ---");
        Dog dog1 = new Dog("Rex", "Labrador");

        System.out.println("\n--- One-arg constructor (delegates via this()) ---");
        Dog dog2 = new Dog("Buddy");

        // === Calling overridden methods with super ===
        System.out.println("\n--- info() with super call ---");
        dog1.info();

        // === Fluent API with this ===
        System.out.println("\n--- Fluent chaining with this ---");
        dog1.printName().printBreed();

        // === Field hiding vs. overriding ===
        System.out.println("\n--- Field hiding ---");
        new Child().showLabels();

        // === Key interview rules ===
        // 1. super() or this() must be the FIRST statement in a constructor.
        // 2. You cannot use both super() and this() in the same constructor.
        // 3. If no explicit super() or this() is written, the compiler inserts
        //    a no-arg super() call -- which fails if the parent has no no-arg constructor.
        // 4. super.method() bypasses polymorphism and calls the parent's version.
        // 5. Fields are not polymorphic -- super.field accesses the parent's field directly.
    }
}
```

What are switch expressions and how do they differ from traditional switch statements?

Switch expressions, finalized in Java 14, transform `switch` from a statement into an **expression that returns a value**. They use the arrow syntax (`->`) which eliminates fall-through behavior entirely, removing one of the most common sources of bugs in traditional switch statements. Multiple case labels can be combined with commas (e.g., `case SPRING, SUMMER ->`).

When a switch expression branch requires multiple statements, the block form is used with the `yield` keyword to return a value. Switch expressions must be **exhaustive** — they must cover all possible values of the selector type. For enums, the compiler verifies that every constant is

handled, and adding a new enum constant without updating the switch produces a compilation error. For other types, a `default` branch is required.

Java 21 extended switch expressions with **pattern matching** and **null handling**. A `case null` branch can explicitly handle null selectors (which previously threw `NullPointerException`), and type patterns with optional `when` guards enable expressive, type-safe dispatching. Combined with sealed types, pattern matching switch achieves compiler-verified exhaustive type handling, and the switch expression can be assigned to a variable, returned from a method, or passed as an argument.

Listing 1.18: Switch expressions: arrow syntax, yield, exhaustiveness, and pattern matching

```java
/**
 * Switch expressions (Java 14+) -- arrow syntax, yielding values,
 * exhaustiveness, and pattern matching preview.
 */
public class SwitchExpressions {

    enum Season { SPRING, SUMMER, AUTUMN, WINTER }

    public static void main(String[] args) {
        // === Traditional switch statement (pre-Java 14) ===
        Season season = Season.SUMMER;
        String oldResult;
        switch (season) {
            case SPRING:
            case SUMMER:
                oldResult = "Warm";
                break;                // easy to forget -- leads to fall-through bugs
            case AUTUMN:
            case WINTER:
                oldResult = "Cold";
                break;
            default:
                oldResult = "Unknown";
        }
        System.out.println("Old switch: " + oldResult);

        // === Switch EXPRESSION with arrow syntax (Java 14+) ===
        // - No fall-through, no break needed.
        // - Returns a value directly.
        String newResult = switch (season) {
            case SPRING, SUMMER -> "Warm";
            case AUTUMN, WINTER -> "Cold";
            // No default needed -- compiler verifies exhaustiveness for enums
        };
        System.out.println("Arrow switch: " + newResult);

        // === Multi-line blocks with yield ===
        int monthNumber = 8;
        String quarter = switch (monthNumber) {
            case 1, 2, 3 -> "Q1";
            case 4, 5, 6 -> "Q2";
            case 7, 8, 9 -> {
                System.out.println("  (computing Q3)");
                yield "Q3";           // 'yield' returns from a block in a switch expression
            }
```

```java
            case 10, 11, 12 -> "Q4";
            default -> throw new IllegalArgumentException("Invalid month: " + monthNumber);
        };
        System.out.println("Quarter: " + quarter);

        // === Exhaustiveness requirement ===
        // When a switch expression covers all enum constants, the default case is optional.
        // If you add a new constant to the enum later and forget to handle it, the
        // compiler produces an error -- catching bugs at compile time instead of runtime.

        // === Null handling (Java 21+) ===
        // Traditional switch throws NullPointerException if the selector is null.
        // Java 21 allows explicit null case:
        // String desc = switch (nullableString) {
        //     case null  -> "was null";
        //     case "foo" -> "found foo";
        //     default    -> "something else";
        // };

        // === Pattern matching in switch (Java 21+) ===
        Object obj = 42;
        String description = switch (obj) {
            case Integer i when i > 0 -> "Positive int: " + i;
            case Integer i            -> "Non-positive int: " + i;
            case String s             -> "String of length " + s.length();
            case null                  -> "null value";
            default                    -> "Other: " + obj.getClass().getSimpleName();
        };
        System.out.println("Pattern match: " + description);

        // === Key interview points ===
        // 1. Switch expressions MUST be exhaustive (cover all cases or include default).
        // 2. Arrow labels (->) prevent fall-through entirely.
        // 3. Use 'yield' to return a value from a multi-statement block.
        // 4. Pattern matching + sealed types = compiler-verified exhaustive handling.
        // 5. Switch expressions can be assigned to variables, returned, or passed as arguments.
    }
}
```

How do sealed types enable exhaustive pattern matching in Java?

Sealed types, introduced in Java 17, restrict which classes or interfaces may extend or implement a given type using the `permits` clause. Each permitted subtype must be `final`, `sealed` (continuing the restriction), or `non-sealed` (reopening the hierarchy for further extension). Records are implicitly `final`, making them natural partners for sealed interfaces.

The primary benefit of sealed types becomes apparent when combined with pattern matching in switch expressions (Java 21). Because the compiler knows the complete set of permitted subtypes, it can verify that a switch expression handles every possible case **at compile time** — no `default` branch is needed. If a new subtype is later added to the `permits` list, every switch expression that does not handle it will produce a compilation error, catching the omission immediately rather than at runtime.

Guarded patterns using the `when` clause add further expressiveness. For example,

`case Circle c when c.radius() > 10 -> "Large circle"` combines type checking, binding, and a conditional guard in a single case label. This approach represents Java's answer to **algebraic data types** and **tagged unions** found in languages like Rust and Scala, and it enables domain modeling where the type system enforces completeness of handling.

Listing 1.19: Sealed types and exhaustive pattern matching with guarded patterns

```java
/**
 * Sealed interfaces (Java 17) and exhaustive pattern matching (Java 21).
 *
 * Sealed types restrict which classes can implement/extend them, enabling
 * the compiler to verify that pattern matches handle every permitted subtype.
 */
public class SealedPatternMatching {

    // === Sealed interface: only the listed classes may implement it ===
    sealed interface Shape permits Circle, Rectangle, Triangle {
        double area();
    }

    // 'record' classes are implicitly final -- satisfying the sealed requirement
    record Circle(double radius) implements Shape {
        public double area() { return Math.PI * radius * radius; }
    }

    record Rectangle(double width, double height) implements Shape {
        public double area() { return width * height; }
    }

    // A 'final' class also satisfies the sealed contract
    static final class Triangle implements Shape {
        private final double base, height;
        Triangle(double base, double height) { this.base = base; this.height = height; }
        public double area() { return 0.5 * base * height; }
    }

    // === Exhaustive pattern matching (Java 21 switch) ===
    // Because Shape is sealed and all permitted subtypes are listed,
    // the compiler knows this switch is exhaustive -- no default needed.
    static String describe(Shape shape) {
        return switch (shape) {
            case Circle c    -> "Circle with radius %.2f, area=%.2f".formatted(c.radius(), c.area());
            case Rectangle r -> "Rectangle %sx%s, area=%.2f".formatted(r.width(), r.height(), r.area());
            case Triangle t  -> "Triangle with area=%.2f".formatted(t.area());
            // No default needed! The compiler enforces completeness.
        };
    }

    // === Guarded patterns (when clause) ===
    static String classify(Shape shape) {
        return switch (shape) {
            case Circle c when c.radius() > 10   -> "Large circle";
            case Circle c                         -> "Small circle";
            case Rectangle r when r.width() == r.height() -> "Square";
            case Rectangle r                      -> "Rectangle";
            case Triangle t                       -> "Triangle";
        };
    }
```

```java

    // === Sealed with non-sealed subtype for extensibility ===
    sealed interface Result permits Success, Failure, Pending {}
    record Success(String data) implements Result {}
    record Failure(String error) implements Result {}
    // 'non-sealed' reopens the hierarchy for further extension
    non-sealed class Pending implements Result {
        // External code can extend Pending freely
    }

    public static void main(String[] args) {
        Shape[] shapes = {
            new Circle(5),
            new Rectangle(3, 4),
            new Triangle(6, 8),
            new Rectangle(7, 7)
        };

        for (Shape s : shapes) {
            System.out.println(describe(s));
            System.out.println("  -> " + classify(s));
        }

        // === Exhaustiveness safety net ===
        // If a new permitted subtype (e.g., Pentagon) is added to Shape,
        // EVERY switch expression over Shape will fail to compile until
        // the new case is handled -- eliminating an entire class of bugs.

        // === Key interview points ===
        // 1. sealed + permits restricts the subtype hierarchy.
        // 2. Permitted subtypes must be final, sealed, or non-sealed.
        // 3. Records are implicitly final and work naturally with sealed types.
        // 4. Pattern matching switch + sealed types = compile-time exhaustiveness.
        // 5. This is Java's answer to algebraic data types / tagged unions.
    }
}
```

What are the dangers of unchecked casts with generics and how can they be avoided?

Due to **type erasure**, the JVM has no knowledge of generic type parameters at runtime — `List<String>` and `List<Integer>` are both simply `List`. This means that casting an `Object` to `List<String>` is an **unchecked cast** that the JVM cannot verify. The cast succeeds even if the object is actually a `List<Integer>`, and the resulting `ClassCastException` only surfaces later when an element is actually used.

This phenomenon is called **heap pollution**: a variable of a parameterized type refers to an object that does not match that type, but no error is thrown until the corrupted data is accessed. The `@SuppressWarnings("unchecked")` annotation silences the compiler warning but does not make the cast safe — it merely acknowledges the risk.

Three safe patterns exist to avoid these issues. First, **element-wise checking**: iterate over the collection and use `instanceof` to filter elements of the desired type into a new list. Second,

Class<T> type tokens: pass a `Class<T>` object and use `type.cast(item)` to perform a checked cast that throws immediately if the type does not match. Third, `Collections.checkedList()` : wraps a list with runtime type enforcement that rejects insertions of the wrong type at the point of insertion, providing fail-fast behavior. The key interview takeaway is that reifiable types (non-generic types like `String` and `Integer`) can be safely cast at runtime, but parameterized types cannot.

Listing 1.20: Generic casts: type erasure, heap pollution, and safe casting patterns

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

/**
 * Checked vs. unchecked casts with generics -- type erasure, heap pollution,
 * and safe patterns.
 */
public class GenericCasts {

    // === Type erasure recap ===
    // At runtime, List<String> and List<Integer> are both just List.
    // The generic type parameter is erased by the compiler, so the JVM
    // cannot distinguish between them.

    // === Unchecked cast -- compiles with a warning, fails silently ===
    @SuppressWarnings("unchecked")
    static List<String> unsafeCast(Object obj) {
        // This cast CANNOT be verified at runtime due to erasure.
        // It succeeds even if obj is actually a List<Integer>.
        return (List<String>) obj; // WARNING: unchecked cast
    }

    // === Heap pollution: the silent corruption ===
    static void heapPollutionDemo() {
        List<Integer> ints = new ArrayList<>();
        ints.add(42);

        // Deliberately break type safety via raw type
        @SuppressWarnings("unchecked")
        List<String> strings = (List<String>) (List<?>) ints; // unchecked

        // No exception here -- the list doesn't know its element type at runtime
        System.out.println("Size: " + strings.size()); // 1

        // ClassCastException occurs when you actually USE an element
        try {
            String s = strings.get(0); // boom -- Integer cannot be cast to String
        } catch (ClassCastException e) {
            System.out.println("Heap pollution caught: " + e.getMessage());
        }
    }

    // === Safe patterns ===

    // Pattern 1: Check elements individually
    static List<String> safeFilter(List<?> raw) {
        List<String> result = new ArrayList<>();
        for (Object item : raw) {
```

```java
            if (item instanceof String s) {
                result.add(s); // checked cast via pattern matching
            }
        }
        return result;
    }

    // Pattern 2: Use Class<T> token for a type-safe cast
    static <T> List<T> checkedCast(List<?> raw, Class<T> type) {
        List<T> result = new ArrayList<>();
        for (Object item : raw) {
            result.add(type.cast(item)); // throws ClassCastException immediately if wrong
        }
        return result;
    }

    // Pattern 3: Collections.checkedList() -- runtime type enforcement
    static void checkedListDemo() {
        List<String> safe = java.util.Collections.checkedList(
                new ArrayList<>(), String.class
        );
        safe.add("hello");
        try {
            @SuppressWarnings("unchecked")
            List rawRef = safe;
            rawRef.add(123); // ClassCastException at insertion -- fail-fast
        } catch (ClassCastException e) {
            System.out.println("checkedList blocked bad insert: " + e.getMessage());
        }
    }

    public static void main(String[] args) {
        // Unchecked cast -- no immediate error
        List<Integer> original = List.of(1, 2, 3);
        List<String> broken = unsafeCast(original);
        System.out.println("Unchecked cast succeeded (no error yet)");

        // Heap pollution
        heapPollutionDemo();

        // Safe filtering
        List<?> mixed = List.of("a", 1, "b", 2);
        System.out.println("Safe filter: " + safeFilter(mixed)); // [a, b]

        // Type-token cast
        try {
            List<String> checked = checkedCast(List.of("x", "y"), String.class);
            System.out.println("Checked cast: " + checked);
        } catch (ClassCastException e) {
            System.out.println("Checked cast failed: " + e);
        }

        // Collections.checkedList
        checkedListDemo();

        // === Interview key points ===
        // 1. Generic casts are unchecked because type parameters are erased at runtime.
        // 2. Heap pollution occurs when a variable of parameterized type refers to an
        //    object that is not of that type -- errors surface later, far from the cause.
```

```
        // 3. @SuppressWarnings("unchecked") silences warnings but does NOT make the cast safe.
        // 4. Use Class<T> tokens, instanceof checks, or Collections.checkedList() for safety.
        // 5. Reifiable types (non-generic, e.g., String, Integer) can be safely cast at runtime.
    }
}
```

1.7 Memory and Performance Fundamentals

How does autoboxing affect performance and what are the pitfalls?

Autoboxing, introduced in Java 5, automatically converts between primitive types and their wrapper classes. While this feature simplifies code, it introduces several subtle performance and correctness pitfalls that are frequently explored in interviews. Understanding these pitfalls demonstrates that a candidate thinks beyond surface-level convenience.

The most significant performance impact occurs in **tight loops**. When a wrapper type like `Long` is used as a loop accumulator instead of the primitive `long`, every arithmetic operation triggers autoboxing and unboxing, creating millions of short-lived objects on the heap. Benchmarks show this can be 5–10x slower than using primitives. The fix is simple: always use primitive types for loop variables, accumulators, and performance-critical computations.

The most dangerous correctness pitfall involves **identity comparison with** `==`. For `Integer` values in the range `-128` to `127`, the JVM caches wrapper objects (as required by the Java Language Specification), so `==` comparison happens to work. For values outside this range, each autoboxing creates a new object, and `==` compares references, not values. This means `Integer a = 127; Integer b = 127; a == b` returns `true`, but `Integer a = 128; Integer b = 128; a == b` returns `false`. This inconsistency is one of the most common interview trap questions.

A third pitfall is **unboxing null**, which throws a `NullPointerException`. This occurs silently in expressions like `int x = nullableInteger` or when comparing a `null` wrapper with a primitive using `==`. In collections, methods like `Map.get()` return `null` for missing keys, and assigning the result directly to a primitive triggers the NPE.

- **Cache range**: `Integer` caches `-128` to `127`; `Boolean`, `Byte`, `Short`, and `Character` (0–127) are also cached.
- **Performance**: Use primitives in loops and accumulators; wrapper types create garbage.
- **Null safety**: Always null-check before unboxing wrapper types from collections or method returns.
- **Comparison**: Always use `.equals()` for wrapper type comparison, never `==`.

Listing 1.21: Autoboxing pitfalls: identity comparison, cache range, loop performance, and null unboxing

```
/**
 * Demonstrates autoboxing pitfalls: identity comparison, cache range,
 * loop performance, and null unboxing.
```

```java
 */
public class AutoboxingPitfalls {

    // === Pitfall 1: Identity comparison with == ===
    static void identityComparisonTrap() {
        System.out.println("=== Identity Comparison Trap ===");

        // Within cache range (-128 to 127): == works (by accident)
        Integer a = 127;
        Integer b = 127;
        System.out.println("127 == 127 (Integer): " + (a == b));    // true (cached)

        // Outside cache range: == fails
        Integer c = 128;
        Integer d = 128;
        System.out.println("128 == 128 (Integer): " + (c == d));    // false!

        // Always use .equals() for wrapper comparison
        System.out.println("128.equals(128): " + c.equals(d));      // true

        // Boolean is always cached (only two values)
        Boolean t1 = true;
        Boolean t2 = true;
        System.out.println("true == true (Boolean): " + (t1 == t2)); // true (cached)

        // Byte is fully cached (-128 to 127 = entire range)
        Byte b1 = 100;
        Byte b2 = 100;
        System.out.println("100 == 100 (Byte): " + (b1 == b2));     // true (cached)
    }

    // === Pitfall 2: Performance in loops ===
    static void loopPerformancePitfall() {
        System.out.println("\n=== Loop Performance Pitfall ===");

        // BAD: Using Long wrapper as accumulator (creates millions of objects)
        long startBad = System.nanoTime();
        Long sumBad = 0L;  // wrapper type!
        for (long i = 0; i < 1_000_000; i++) {
            sumBad += i;  // unbox, add, re-box on every iteration
        }
        long badTime = System.nanoTime() - startBad;

        // GOOD: Using long primitive as accumulator
        long startGood = System.nanoTime();
        long sumGood = 0L;  // primitive type
        for (long i = 0; i < 1_000_000; i++) {
            sumGood += i;  // no boxing at all
        }
        long goodTime = System.nanoTime() - startGood;

        System.out.println("Wrapper (Long) time: " + badTime / 1_000_000 + " ms");
        System.out.println("Primitive (long) time: " + goodTime / 1_000_000 + " ms");
        System.out.println("Both results equal: " + (sumBad.equals(sumGood)));
        System.out.println("Rule: Always use primitives in loops and accumulators");
    }

    // === Pitfall 3: Null unboxing ===
    static void nullUnboxingPitfall() {
```

```java
        System.out.println("\n=== Null Unboxing Pitfall ===");

        // Unboxing a null wrapper throws NullPointerException
        Integer nullableValue = null;
        try {
            int primitive = nullableValue;  // NPE! unboxing null
        } catch (NullPointerException e) {
            System.out.println("NPE on null unboxing: " + e.getMessage());
        }

        // Common scenario: Map.get() returns null for missing keys
        java.util.Map<String, Integer> scores = java.util.Map.of("Alice", 95);
        try {
            int bobScore = scores.get("Bob");  // Map.get returns null, unbox throws NPE
        } catch (NullPointerException e) {
            System.out.println("NPE from Map.get(): missing key 'Bob'");
        }

        // Safe approach: use getOrDefault or check for null
        int safeScore = scores.getOrDefault("Bob", 0);
        System.out.println("Safe score with getOrDefault: " + safeScore);

        // Ternary operator pitfall
        boolean condition = true;
        Integer boxedNull = null;
        try {
            // Compiler unboxes boxedNull to match the int 42
            int result = condition ? 42 : boxedNull;  // NPE if condition were false!
            System.out.println("Ternary result: " + result);
        } catch (NullPointerException e) {
            System.out.println("Ternary NPE: " + e.getMessage());
        }
    }

    // === Pitfall 4: Misleading comparisons ===
    static void misleadingComparisons() {
        System.out.println("\n=== Misleading Comparisons ===");

        // Comparing different wrapper types with ==
        Integer intVal = 0;
        Long longVal = 0L;
        // intVal == longVal;  // Compile error: incomparable types
        System.out.println("Integer(0).equals(Long(0)): " + intVal.equals(longVal));  // false!
        // Reason: Integer.equals checks (obj instanceof Integer)

        // Correct cross-type comparison
        System.out.println("intVal.intValue() == longVal.intValue(): "
                + (intVal.intValue() == longVal.intValue()));  // true

        // Integer.valueOf() vs new Integer() (deprecated)
        Integer cached1 = Integer.valueOf(100);  // may return cached instance
        Integer cached2 = Integer.valueOf(100);  // same cached instance
        System.out.println("valueOf(100) == valueOf(100): " + (cached1 == cached2));  // true
    }

    public static void main(String[] args) {
        identityComparisonTrap();
        loopPerformancePitfall();
        nullUnboxingPitfall();
```

```
        misleadingComparisons();

        System.out.println("\n=== Autoboxing Rules of Thumb ===");
        System.out.println("1. Use primitives for loops, accumulators, and arithmetic");
        System.out.println("2. Use .equals() not == for wrapper comparisons");
        System.out.println("3. Null-check before unboxing (Map.get, method returns)");
        System.out.println("4. Cache range: Integer -128..127, Byte full range, Boolean both");
        System.out.println("5. Prefer Integer.valueOf() over new Integer() (deprecated)");
    }
}
```

What is the difference between shallow equality, deep equality, and identity?

Understanding the three forms of equality in Java is essential for writing correct code and is a frequent interview topic. The distinctions are subtle but have profound implications for how objects behave in collections, assertions, and business logic.

Identity (also called reference equality) checks whether two variables refer to the *exact same object in memory*. In Java, this is tested with the `==` operator for objects. Identity is the default behavior of `Object.equals()`, which simply returns `this == other`. Two objects can have identical state but different identity if they are separate instances.

Shallow equality compares the immediate fields of two objects but does not recursively compare the objects those fields reference. For example, two `Person` objects with the same `name` and `age` fields are shallowly equal, even if they reference different `Address` objects (which might themselves have different content). Records in Java 16+ implement shallow equality by default: the auto-generated `equals()` compares each component using its own `equals()`, but this is only one level deep if the components are mutable objects.

Deep equality recursively compares all referenced objects until it reaches primitives or immutable leaf types. Two object graphs are deeply equal if every node has the same value. Deep equality must be implemented manually and requires care to avoid infinite recursion with circular references. The `Arrays.deepEquals()` method provides deep equality for nested arrays.

- **Identity**: `==` for objects; tests same reference; cheapest check.
- **Shallow equality**: Compares immediate fields; default for records and most `equals()` implementations.
- **Deep equality**: Recursively compares entire object graph; expensive but thorough.
- **Best practice**: Override `equals()` using the type of equality appropriate for your domain.

Listing 1.22: Identity, shallow equality, and deep equality demonstrated with nested objects

```
import java.util.Arrays;
import java.util.List;
import java.util.Objects;

/**
 * Demonstrates the three forms of equality in Java:
 * identity (==), shallow equality, and deep equality.
 */
```

```java
public class EqualityTypes {

    // === A mutable Address class (to illustrate shallow vs deep) ===
    static class Address {
        String city;
        String zip;

        Address(String city, String zip) {
            this.city = city;
            this.zip = zip;
        }

        // Shallow equality: compares immediate fields
        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (!(o instanceof Address a)) return false;
            return Objects.equals(city, a.city) && Objects.equals(zip, a.zip);
        }

        @Override
        public int hashCode() { return Objects.hash(city, zip); }

        @Override
        public String toString() { return city + " " + zip; }
    }

    // === Person with a mutable Address reference ===
    static class Person {
        String name;
        Address address;

        Person(String name, Address address) {
            this.name = name;
            this.address = address;
        }

        // Shallow equality: delegates to field equals() but does not
        // deeply verify that Address content matches independently
        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (!(o instanceof Person p)) return false;
            return Objects.equals(name, p.name) && Objects.equals(address, p.address);
        }

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

    // === 1. Identity (Reference Equality) ===
    static void identityExample() {
        System.out.println("=== 1. Identity (==) ===");

        Address addr = new Address("London", "EC1");
        Address sameRef = addr;
        Address sameContent = new Address("London", "EC1");

        System.out.println("addr == sameRef:       " + (addr == sameRef));       // true
```

```java
        System.out.println("addr == sameContent:   " + (addr == sameContent));   // false
        System.out.println("addr.equals(sameContent): " + addr.equals(sameContent)); // true

        // String identity vs equality
        String s1 = "hello";
        String s2 = "hello";
        String s3 = new String("hello");
        System.out.println("s1 == s2 (pool):       " + (s1 == s2));    // true (interned)
        System.out.println("s1 == s3 (new):        " + (s1 == s3));    // false
        System.out.println("s1.equals(s3):         " + s1.equals(s3)); // true
    }

    // === 2. Shallow Equality ===
    static void shallowEqualityExample() {
        System.out.println("\n=== 2. Shallow Equality ===");

        Address addr1 = new Address("London", "EC1");
        Address addr2 = new Address("London", "EC1");

        Person p1 = new Person("Alice", addr1);
        Person p2 = new Person("Alice", addr2);

        // Shallow equals: both persons are equal because their fields are equal
        System.out.println("p1.equals(p2): " + p1.equals(p2));  // true

        // But they share no references
        System.out.println("p1 == p2: " + (p1 == p2));                    // false
        System.out.println("p1.address == p2.address: " + (p1.address == p2.address)); // false

        // Mutation through shared reference exposes shallow equality limits
        Person p3 = new Person("Alice", addr1);  // shares addr1 with p1
        System.out.println("p1.equals(p3): " + p1.equals(p3));  // true
        addr1.city = "Paris";  // mutate the shared address!
        System.out.println("After mutation:");
        System.out.println("  p1.address: " + p1.address);  // Paris
        System.out.println("  p3.address: " + p3.address);  // Paris (same object!)
        System.out.println("  p2.address: " + p2.address);  // London (independent)
    }

    // === 3. Deep Equality ===
    static void deepEqualityExample() {
        System.out.println("\n=== 3. Deep Equality ===");

        // Arrays.equals: shallow (compares elements with ==)
        // Arrays.deepEquals: recursive deep comparison for nested arrays
        int[][] matrix1 = {{1, 2}, {3, 4}};
        int[][] matrix2 = {{1, 2}, {3, 4}};

        System.out.println("Arrays.equals (shallow):   " + Arrays.equals(matrix1, matrix2));     // false!
        System.out.println("Arrays.deepEquals (deep):  " + Arrays.deepEquals(matrix1, matrix2)); // true

        // List equality is element-wise (shallow deep: one level)
        List<List<Integer>> list1 = List.of(List.of(1, 2), List.of(3, 4));
        List<List<Integer>> list2 = List.of(List.of(1, 2), List.of(3, 4));
        System.out.println("Nested list equality: " + list1.equals(list2));  // true
        // This works because List.equals recursively calls element.equals()
    }

    // === 4. Records: auto-generated shallow equality ===
```

```java
    record Coordinate(double x, double y) {}
    record Route(String name, List<Coordinate> waypoints) {}

    static void recordEqualityExample() {
        System.out.println("\n=== 4. Record Equality (auto shallow) ===");

        var c1 = new Coordinate(51.5, -0.1);
        var c2 = new Coordinate(51.5, -0.1);
        System.out.println("Record equality: " + c1.equals(c2));  // true

        // Records with mutable collections
        var mutableList = new java.util.ArrayList<>(List.of(c1));
        var r1 = new Route("A", mutableList);
        var r2 = new Route("A", List.of(c1));
        System.out.println("Route equality: " + r1.equals(r2));  // true (List.equals)

        // But mutable list can be changed after construction!
        mutableList.add(c2);
        System.out.println("After mutation: " + r1.equals(r2));  // false!
        System.out.println("Record does NOT defensively copy mutable components");
    }

    public static void main(String[] args) {
        identityExample();
        shallowEqualityExample();
        deepEqualityExample();
        recordEqualityExample();

        System.out.println("\n=== Equality Summary ===");
        System.out.println("Identity (==):  Same object in memory; cheapest check");
        System.out.println("Shallow equals: Compares immediate fields; default for records");
        System.out.println("Deep equals:    Recursively compares object graph; manual impl");
        System.out.println("Best practice:  Choose equality type appropriate for your domain");
    }
}
```

How do you implement immutable classes in Java?

Immutability is a cornerstone of robust Java programming. An immutable object cannot be modified after construction, which eliminates an entire class of concurrency bugs, simplifies reasoning about program state, and makes objects safe to use as `HashMap` keys and `Set` elements. Interviewers frequently ask candidates to design an immutable class from scratch and identify common mistakes that break immutability.

The recipe for implementing an immutable class in pre-Java 16 code has five components. First, declare the class `final` to prevent subclasses from adding mutable state or overriding methods. Second, declare all fields `private` and `final` to prevent reassignment and direct access. Third, provide no setter methods — only getter methods that return values. Fourth, perform **defensive copies** of any mutable objects received in the constructor (such as `Date`, `List`, or arrays) to prevent external code from modifying the internal state through the original reference. Fifth, return **defensive copies** from getter methods that expose mutable objects, so callers cannot modify the internal state through the returned reference.

Java 16+ **records** provide a concise alternative. Records are implicitly final, their components are private and final, and they have no setters. However, records do *not* automatically perform defensive copies: if a record component is a mutable `List`, the record stores the original reference. To make a record truly immutable, use a compact constructor that wraps mutable components in unmodifiable collections (e.g., `List.copyOf(items)`).

- **Final class**: Prevents mutable subclasses from violating the contract.
- **Final fields**: Prevents reassignment; also provides safe publication in concurrent code.
- **Defensive copies**: Both inbound (constructor) and outbound (getters) for mutable fields.
- **Records**: Almost immutable by default, but require manual defensive copies for mutable components.
- **Thread safety**: Immutable objects are inherently thread-safe with no synchronization needed.

Listing 1.23: Immutable class design: defensive copies, final fields, and records

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

/**
 * Demonstrates how to implement immutable classes in Java:
 * - Manual implementation (pre-Java 16)
 * - Record-based implementation (Java 16+)
 * - Common mistakes that break immutability
 */
public class ImmutableClasses {

    // === Manual Immutable Class (Classic Approach) ===
    static final class ImmutablePerson {
        // 1. All fields are private and final
        private final String name;
        private final int age;
        private final List<String> hobbies;

        // 2. Constructor performs defensive copies of mutable arguments
        public ImmutablePerson(String name, int age, List<String> hobbies) {
            this.name = Objects.requireNonNull(name, "name must not be null");
            this.age = age;
            // Defensive copy: do NOT store the caller's list reference
            this.hobbies = List.copyOf(hobbies);  // unmodifiable copy
        }

        // 3. Only getters, no setters
        public String getName() { return name; }
        public int getAge() { return age; }

        // 4. Return defensive copy for mutable types
        public List<String> getHobbies() {
            return hobbies;  // already unmodifiable from List.copyOf
        }

        @Override
        public String toString() {
```

```java
            return "ImmutablePerson{name='%s', age=%d, hobbies=%s}".formatted(name, age, hobbies);
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (!(o instanceof ImmutablePerson p)) return false;
            return age == p.age && name.equals(p.name) && hobbies.equals(p.hobbies);
        }

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

    // === BROKEN Immutable Class (Common Mistakes) ===
    static class BrokenImmutable {
        private final String name;
        private final List<String> items;  // mutable list stored directly!

        BrokenImmutable(String name, List<String> items) {
            this.name = name;
            this.items = items;  // BUG: stores the caller's mutable reference
        }

        public List<String> getItems() {
            return items;  // BUG: exposes internal mutable state
        }
    }

    // === Record-Based Immutable Class (Java 16+) ===
    record PersonRecord(String name, int age, List<String> hobbies) {
        // Compact constructor for validation and defensive copy
        PersonRecord {
            Objects.requireNonNull(name, "name must not be null");
            Objects.requireNonNull(hobbies, "hobbies must not be null");
            hobbies = List.copyOf(hobbies);  // defensive copy!
        }
    }

    // === Immutable class with mutable java.util.Date (legacy) ===
    static final class Event {
        private final String title;
        private final java.util.Date date;  // Date is mutable!

        public Event(String title, java.util.Date date) {
            this.title = Objects.requireNonNull(title);
            this.date = new java.util.Date(date.getTime());  // defensive copy in
        }

        public String getTitle() { return title; }

        public java.util.Date getDate() {
            return new java.util.Date(date.getTime());  // defensive copy out
        }
    }

    public static void main(String[] args) {
        System.out.println("=== Correct Immutable Class ===");
        List<String> hobbies = new ArrayList<>(List.of("reading", "hiking"));
```

```java
        ImmutablePerson person = new ImmutablePerson("Alice", 30, hobbies);

        // External modification has no effect on the immutable object
        hobbies.add("gaming");
        System.out.println("Original list modified: " + hobbies);
        System.out.println("Person's hobbies unchanged: " + person.getHobbies());

        // Getter returns unmodifiable list
        try {
            person.getHobbies().add("swimming");
        } catch (UnsupportedOperationException e) {
            System.out.println("Cannot modify hobbies via getter: " + e.getClass().getSimpleName());
        }

        System.out.println("\n=== BROKEN Immutable Class ===");
        List<String> items = new ArrayList<>(List.of("item1"));
        BrokenImmutable broken = new BrokenImmutable("Bob", items);

        // External modification leaks through!
        items.add("item2");
        System.out.println("Broken - items leaked: " + broken.getItems());

        // Getter exposes mutable state!
        broken.getItems().add("item3");
        System.out.println("Broken - modified via getter: " + broken.getItems());

        System.out.println("\n=== Record-Based Immutable ===");
        List<String> recordHobbies = new ArrayList<>(List.of("chess", "cooking"));
        PersonRecord record = new PersonRecord("Charlie", 25, recordHobbies);
        recordHobbies.add("yoga");
        System.out.println("Record hobbies safe: " + record.hobbies());
        System.out.println("Record toString: " + record);

        System.out.println("\n=== Thread Safety of Immutable Objects ===");
        // Immutable objects can be shared freely across threads
        Thread t1 = new Thread(() -> System.out.println("Thread 1: " + person.getName()));
        Thread t2 = new Thread(() -> System.out.println("Thread 2: " + person.getHobbies()));
        t1.start();
        t2.start();

        System.out.println("\n=== Immutability Checklist ===");
        System.out.println("1. Class is final (prevents mutable subclasses)");
        System.out.println("2. All fields are private and final");
        System.out.println("3. No setter methods");
        System.out.println("4. Defensive copy in constructor for mutable parameters");
        System.out.println("5. Defensive copy in getters for mutable fields");
        System.out.println("6. Records: use compact constructor with List.copyOf()");
    }
}
```

What are the common pitfalls with floating-point arithmetic?

Floating-point arithmetic in Java (and virtually every other language) is governed by the IEEE 754 standard, which represents real numbers in a binary format with limited precision. This leads to well-known but frequently misunderstood pitfalls that interviewers use to test a candidate's

depth of understanding.

The most famous pitfall is that **decimal fractions cannot be represented exactly** in binary floating-point. The expression `0.1 + 0.2` evaluates to `0.30000000000000004`, not `0.3`. This means that direct equality comparison with `==` is unreliable for floating-point values. Instead, use an **epsilon comparison** (check whether the absolute difference is below a small threshold) or, for financial and decimal-exact calculations, use `java.math.BigDecimal`.

`BigDecimal` provides arbitrary-precision decimal arithmetic and is the correct choice for **monetary calculations**. A critical subtlety is that `new BigDecimal(0.1)` captures the imprecise binary representation of `0.1`, while `new BigDecimal("0.1")` creates an exact decimal value. Always construct `BigDecimal` from strings or use `BigDecimal.valueOf(0.1)`. When dividing, always specify a `MathContext` or scale with a rounding mode; otherwise, a non-terminating decimal result throws `ArithmeticException`.

Additional pitfalls include **special values**: `Double.NaN` is not equal to itself (`Double.NaN == Double.NaN` is `false`), `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY` behave according to mathematical rules, and `-0.0 == 0.0` is `true` but `Double.compare(-0.0, 0.0)` returns `-1`. These distinctions matter for numerical algorithms and for correct `equals()` and `compareTo()` implementations on classes with floating-point fields.

- **Never use `==` for float/double**: Use epsilon comparison or `BigDecimal`.
- **Money**: Always use `BigDecimal` with string constructors and explicit rounding.
- `NaN`: `NaN != NaN`; use `Double.isNaN()` to check.
- **Precision loss**: `float` has 7 decimal digits; `double` has 15 decimal digits.

Listing 1.24: Floating-point pitfalls: precision, BigDecimal, NaN, and money calculations

```java
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;

/**
 * Demonstrates common floating-point pitfalls in Java:
 * - Precision errors with double/float
 * - Proper use of BigDecimal for exact arithmetic
 * - Special values: NaN, infinity, negative zero
 * - Money calculation best practices
 */
public class FloatingPointPitfalls {

    // === Pitfall 1: Binary representation imprecision ===
    static void precisionPitfall() {
        System.out.println("=== Precision Pitfall ===");

        // 0.1 + 0.2 is NOT exactly 0.3 in IEEE 754
        double result = 0.1 + 0.2;
        System.out.println("0.1 + 0.2 = " + result);                    // 0.30000000000000004
        System.out.println("0.1 + 0.2 == 0.3? " + (result == 0.3));     // false!

        // Accumulated error in loops
        double sum = 0.0;
```

```java
        for (int i = 0; i < 10; i++) {
            sum += 0.1;
        }
        System.out.println("0.1 * 10 = " + sum);                        // 0.9999999999999999
        System.out.println("== 1.0? " + (sum == 1.0));                  // false!

        // Epsilon comparison (the correct approach for doubles)
        double epsilon = 1e-10;
        System.out.println("Within epsilon? " + (Math.abs(sum - 1.0) < epsilon)); // true
    }

    // === Pitfall 2: BigDecimal constructor from double ===
    static void bigDecimalConstructorPitfall() {
        System.out.println("\n=== BigDecimal Constructor Pitfall ===");

        // BAD: captures the imprecise binary representation of 0.1
        BigDecimal bad = new BigDecimal(0.1);
        System.out.println("new BigDecimal(0.1):     " + bad);
        // 0.1000000000000000055511151231257827021181583404541015625

        // GOOD: creates an exact decimal representation
        BigDecimal good = new BigDecimal("0.1");
        System.out.println("new BigDecimal(\"0.1\"): " + good);  // 0.1

        // GOOD: valueOf uses Double.toString() first
        BigDecimal alsoGood = BigDecimal.valueOf(0.1);
        System.out.println("BigDecimal.valueOf(0.1): " + alsoGood);  // 0.1

        // Correct addition
        BigDecimal sum = new BigDecimal("0.1").add(new BigDecimal("0.2"));
        System.out.println("0.1 + 0.2 (BigDecimal): " + sum);  // 0.3
    }

    // === Pitfall 3: Division without scale ===
    static void divisionPitfall() {
        System.out.println("\n=== Division Pitfall ===");

        BigDecimal a = new BigDecimal("10");
        BigDecimal b = new BigDecimal("3");

        // BAD: non-terminating decimal throws ArithmeticException
        try {
            BigDecimal result = a.divide(b);
        } catch (ArithmeticException e) {
            System.out.println("Division error: " + e.getMessage());
        }

        // GOOD: specify scale and rounding mode
        BigDecimal result = a.divide(b, 10, RoundingMode.HALF_UP);
        System.out.println("10 / 3 (scale 10): " + result);

        // GOOD: use MathContext
        BigDecimal result2 = a.divide(b, MathContext.DECIMAL64);
        System.out.println("10 / 3 (DECIMAL64): " + result2);
    }

    // === Pitfall 4: Special values (NaN, Infinity, -0.0) ===
    static void specialValues() {
        System.out.println("\n=== Special Values ===");
```

```java

        // NaN (Not a Number)
        double nan = Double.NaN;
        System.out.println("NaN == NaN:      " + (nan == nan));          // false!
        System.out.println("Double.isNaN():  " + Double.isNaN(nan));     // true
        System.out.println("NaN < 0:         " + (nan < 0));             // false
        System.out.println("NaN > 0:         " + (nan > 0));             // false
        System.out.println("NaN != NaN:      " + (nan != nan));          // true!

        // Infinity
        double posInf = Double.POSITIVE_INFINITY;
        double negInf = Double.NEGATIVE_INFINITY;
        System.out.println("1.0 / 0.0:       " + (1.0 / 0.0));           // Infinity
        System.out.println("Inf + Inf:       " + (posInf + posInf));     // Infinity
        System.out.println("Inf - Inf:       " + (posInf - posInf));     // NaN
        System.out.println("Inf * 0:         " + (posInf * 0));          // NaN

        // Negative zero
        double negZero = -0.0;
        double posZero = 0.0;
        System.out.println("-0.0 == 0.0:     " + (negZero == posZero));  // true!
        System.out.println("compare(-0, 0):  " + Double.compare(negZero, posZero)); // -1
    }

    // === Best Practice: Money calculations with BigDecimal ===
    static void moneyCalculation() {
        System.out.println("\n=== Money Calculation Best Practice ===");

        // Represent money as BigDecimal with 2 decimal places
        BigDecimal price = new BigDecimal("19.99");
        int quantity = 3;
        BigDecimal taxRate = new BigDecimal("0.08");

        BigDecimal subtotal = price.multiply(BigDecimal.valueOf(quantity));
        BigDecimal tax = subtotal.multiply(taxRate)
                .setScale(2, RoundingMode.HALF_UP);
        BigDecimal total = subtotal.add(tax);

        System.out.println("Price:    $" + price);
        System.out.println("Quantity: " + quantity);
        System.out.println("Subtotal: $" + subtotal);
        System.out.println("Tax (8%): $" + tax);
        System.out.println("Total:    $" + total);

        // compareTo for BigDecimal (equals considers scale!)
        BigDecimal a = new BigDecimal("1.0");
        BigDecimal b = new BigDecimal("1.00");
        System.out.println("\n1.0.equals(1.00):     " + a.equals(b));        // false!
        System.out.println("1.0.compareTo(1.00):  " + a.compareTo(b));      // 0
        System.out.println("Rule: Use compareTo() for BigDecimal comparison");
    }

    public static void main(String[] args) {
        precisionPitfall();
        bigDecimalConstructorPitfall();
        divisionPitfall();
        specialValues();
        moneyCalculation();

```

```
        System.out.println("\n=== Floating-Point Rules ===");
        System.out.println("1. Never use == for float/double comparison");
        System.out.println("2. Use BigDecimal(String) for exact decimal arithmetic");
        System.out.println("3. Always specify RoundingMode for BigDecimal division");
        System.out.println("4. Use compareTo() not equals() for BigDecimal comparison");
        System.out.println("5. Use Double.isNaN() to check for NaN (NaN != NaN)");
        System.out.println("6. Money: always BigDecimal, never double");
    }
}
```

1.8 Interview Scenario Questions

What happens when you override `equals()` but not `hashCode()`?

This is one of the most classic Java interview scenarios because it exposes a fundamental contract violation with visible, reproducible consequences. The `Object.hashCode()` contract states that if two objects are equal according to `equals()`, they **must** produce the same hash code. Violating this contract causes silent, hard-to-debug failures in every hash-based collection.

When you override `equals()` to compare, say, a `name` field, but rely on the default `hashCode()` (which returns a value derived from the object's memory address), two logically equal objects almost certainly produce different hash codes. When such an object is inserted as a key in a `HashMap`, the map uses `hashCode()` to determine the bucket. A subsequent lookup with an equal key computes a different hash code, searches a different bucket, and fails to find the entry — even though `equals()` would return `true` if the objects were compared directly.

The same bug manifests in `HashSet`: adding two equal objects results in duplicates because they land in different buckets. The failure is silent — no exception is thrown, and the incorrect behavior only surfaces through missing lookups or unexpected set sizes. The fix is straightforward: always override `hashCode()` whenever you override `equals()`, using the same fields in both methods. Modern IDEs, Lombok's `@EqualsAndHashCode`, and Java records all generate both methods together to prevent this mistake.

- `HashMap` **failure**: `put(key, value)` succeeds, but `get(equalKey)` returns `null`.
- `HashSet` **failure**: Duplicate equal objects are stored because they hash to different buckets.
- **Contract**: Equal objects **must** have the same hash code; unequal objects **may** have the same hash code.
- **Prevention**: Always generate `equals()` and `hashCode()` together using IDE tools, Lombok, or records.

Listing 1.25: Broken hashCode: demonstrating HashMap and HashSet failure

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Objects;

/**
```

```java
 * Demonstrates what happens when equals() is overridden but hashCode() is not.
 * Shows HashMap lookup failure and HashSet duplicate insertion.
 */
public class BrokenHashCode {

    // === BROKEN: equals() without hashCode() ===
    static class BrokenKey {
        private final String name;
        private final int id;

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

        // equals() overridden to compare by name and id
        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (!(o instanceof BrokenKey key)) return false;
            return id == key.id && Objects.equals(name, key.name);
        }

        // hashCode() NOT overridden! Uses Object.hashCode() (memory-based)
        // This VIOLATES the contract: equal objects MUST have equal hash codes

        @Override
        public String toString() {
            return "BrokenKey{name='%s', id=%d}".formatted(name, id);
        }
    }

    // === CORRECT: equals() AND hashCode() overridden together ===
    static class CorrectKey {
        private final String name;
        private final int id;

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

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (!(o instanceof CorrectKey key)) return false;
            return id == key.id && Objects.equals(name, key.name);
        }

        @Override
        public int hashCode() {
            return Objects.hash(name, id);  // uses same fields as equals()
        }

        @Override
        public String toString() {
            return "CorrectKey{name='%s', id=%d}".formatted(name, id);
        }
    }
```

```java

    static void demonstrateHashMapFailure() {
        System.out.println("=== HashMap Failure (Broken hashCode) ===");

        HashMap<BrokenKey, String> map = new HashMap<>();
        BrokenKey key1 = new BrokenKey("Alice", 1);
        map.put(key1, "Engineer");

        // Create an equal key (different object, same content)
        BrokenKey key2 = new BrokenKey("Alice", 1);

        System.out.println("key1.equals(key2): " + key1.equals(key2));           // true
        System.out.println("key1.hashCode():   " + key1.hashCode());
        System.out.println("key2.hashCode():   " + key2.hashCode());             // different!
        System.out.println("Same hash?         " + (key1.hashCode() == key2.hashCode())); // false!

        // Lookup with equal key FAILS because hashCode leads to a different bucket
        String result = map.get(key2);
        System.out.println("map.get(key2):     " + result);  // null! Expected "Engineer"
        System.out.println("containsKey(key2): " + map.containsKey(key2));  // false!

        // The entry IS in the map - just can't find it
        System.out.println("map.size():        " + map.size());  // 1
    }

    static void demonstrateHashSetDuplicate() {
        System.out.println("\n=== HashSet Duplicate (Broken hashCode) ===");

        HashSet<BrokenKey> set = new HashSet<>();
        BrokenKey key1 = new BrokenKey("Bob", 2);
        BrokenKey key2 = new BrokenKey("Bob", 2);

        set.add(key1);
        set.add(key2);  // Should be rejected as duplicate, but isn't!

        System.out.println("key1.equals(key2): " + key1.equals(key2));  // true
        System.out.println("set.size():        " + set.size());          // 2! Should be 1
        System.out.println("Contains key1:     " + set.contains(key1));  // true
        System.out.println("Contains key2:     " + set.contains(key2));  // true
        System.out.println("Duplicates allowed due to different hash codes!");
    }

    static void demonstrateCorrectBehavior() {
        System.out.println("\n=== Correct Behavior (Proper hashCode) ===");

        HashMap<CorrectKey, String> map = new HashMap<>();
        CorrectKey key1 = new CorrectKey("Alice", 1);
        map.put(key1, "Engineer");

        CorrectKey key2 = new CorrectKey("Alice", 1);
        System.out.println("key1.equals(key2): " + key1.equals(key2));           // true
        System.out.println("Same hash?         " + (key1.hashCode() == key2.hashCode())); // true
        System.out.println("map.get(key2):     " + map.get(key2));                // "Engineer"

        // HashSet correctly rejects duplicates
        HashSet<CorrectKey> set = new HashSet<>();
        set.add(new CorrectKey("Bob", 2));
        set.add(new CorrectKey("Bob", 2));
        System.out.println("set.size():        " + set.size());  // 1 (correct!)
```

```java
    }

    // === Record: auto-generates both equals() and hashCode() ===
    record RecordKey(String name, int id) {}

    static void recordExample() {
        System.out.println("\n=== Records: Always Correct ===");

        HashMap<RecordKey, String> map = new HashMap<>();
        map.put(new RecordKey("Charlie", 3), "Manager");

        String result = map.get(new RecordKey("Charlie", 3));
        System.out.println("Record map lookup: " + result);  // "Manager"
        System.out.println("Records auto-generate consistent equals/hashCode");
    }

    public static void main(String[] args) {
        demonstrateHashMapFailure();
        demonstrateHashSetDuplicate();
        demonstrateCorrectBehavior();
        recordExample();

        System.out.println("\n=== The hashCode Contract ===");
        System.out.println("1. If a.equals(b) then a.hashCode() == b.hashCode() (REQUIRED)");
        System.out.println("2. If a.hashCode() != b.hashCode() then !a.equals(b) (contrapositive)");
        System.out.println("3. Unequal objects MAY have equal hash codes (collisions are OK)");
        System.out.println("4. ALWAYS override hashCode() when you override equals()");
        System.out.println("5. Use Objects.hash() or IDE generation for consistent implementation");
    }
}
```

Why is `String` immutable in Java and what are the benefits?

The immutability of `String` in Java is a deliberate design decision with far-reaching consequences for security, performance, and thread safety. This is a favorite interview question because the answer touches multiple foundational concepts simultaneously.

Security is the primary motivation. Strings are used pervasively as parameters for network connections, file paths, database URLs, and class loading. If strings were mutable, a malicious or buggy caller could pass a string, have it validated, and then modify it before it is used. For example, a file path validated as `"/safe/dir/file.txt"` could be changed to `"/etc/shadow"` between validation and use. Immutability ensures that once a string passes validation, its content cannot change.

String Pool and caching: The JVM maintains a string pool where string literals are interned. Multiple references to the same literal point to the same object, saving memory. This is only safe because strings are immutable — if one reference could modify the shared string, every other reference would see the change. Additionally, `String` caches its `hashCode()` after the first computation (lazy initialization), making strings extremely efficient as `HashMap` and `HashSet` keys.

Thread safety: Immutable objects are inherently thread-safe because they cannot be modified

after construction. Strings can be shared freely across threads without synchronization, and they can be safely used as keys in concurrent collections. This also makes strings safe for use in class loading, where the class name (a string) must not change between the security check and the actual loading.

- **Security**: Prevents modification after validation for network/file/class operations.
- `String` **Pool**: Safe interning requires immutability; saves memory for duplicate literals.
- **hashCode caching**: Computed once, reused forever; makes Strings ideal as map keys.
- **Thread safety**: No synchronization needed; safe sharing across threads.
- **Class loading**: Class names (strings) must not change between security check and loading.

Listing 1.26: String immutability benefits: security, caching, thread safety, and the string pool

```java
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;

/**
 * Demonstrates why String is immutable in Java and the benefits:
 * - Security (validation cannot be bypassed)
 * - String Pool (safe interning)
 * - hashCode caching (efficient map keys)
 * - Thread safety (no synchronization needed)
 * - Class loading safety
 */
public class StringImmutabilityBenefits {

    // === 1. Security: Strings cannot be modified after validation ===
    static void securityBenefit() {
        System.out.println("=== 1. Security ===");

        // Simulating a file access check
        String path = "/safe/directory/file.txt";
        if (isPathAllowed(path)) {
            // If String were mutable, an attacker could change path here
            // between validation and use. Immutability prevents this.
            System.out.println("Access granted to: " + path);
            // path is guaranteed to still be "/safe/directory/file.txt"
        }

        // Database connection strings, network hosts, etc.
        // are all protected by String immutability
        String dbUrl = "jdbc:mysql://trusted-host:3306/mydb";
        System.out.println("DB URL cannot be modified after validation: " + dbUrl);
    }

    static boolean isPathAllowed(String path) {
        return path.startsWith("/safe/");
    }

    // === 2. String Pool: Safe memory sharing ===
    static void stringPoolBenefit() {
        System.out.println("\n=== 2. String Pool (Safe Interning) ===");

        // String literals are automatically interned in the pool
        String s1 = "hello";
```

```java
        String s2 = "hello";
        System.out.println("s1 == s2 (pooled): " + (s1 == s2));  // true (same object)

        // Because String is immutable, sharing is safe
        // If s1 could be modified, s2 would see the change!
        System.out.println("Pool entries are shared safely because Strings are immutable");

        // Manual interning
        String s3 = new String("hello").intern();
        System.out.println("s1 == s3.intern(): " + (s1 == s3));  // true

        // Compile-time constant folding also uses the pool
        String s4 = "hel" + "lo";  // resolved at compile time
        System.out.println("\"hel\" + \"lo\" == s1: " + (s4 == s1));  // true
    }

    // === 3. hashCode Caching: Efficient map keys ===
    static void hashCodeCachingBenefit() {
        System.out.println("\n=== 3. hashCode Caching ===");

        String key = "important_key";
        // String caches its hashCode after first computation
        int hash1 = key.hashCode();  // computed
        int hash2 = key.hashCode();  // returned from cache (no recomputation)
        System.out.println("hash1 == hash2: " + (hash1 == hash2));

        // This makes Strings extremely efficient as HashMap/HashSet keys
        HashMap<String, Integer> map = new HashMap<>();
        for (int i = 0; i < 1000; i++) {
            map.put("key_" + i, i);
        }
        // Each lookup uses cached hashCode - O(1) without recomputation
        System.out.println("Map lookup: " + map.get("key_500"));

        // If String were mutable, hashCode would need to be recomputed
        // on every call, or cached values would become stale
        System.out.println("String hashCode is cached on first call (lazy init)");
    }

    // === 4. Thread Safety: No synchronization needed ===
    static void threadSafetyBenefit() throws InterruptedException {
        System.out.println("\n=== 4. Thread Safety ===");

        // Strings can be shared across threads without synchronization
        final String sharedString = "I am shared safely across threads";
        Set<String> observations = java.util.Collections.synchronizedSet(new HashSet<>());

        Thread[] threads = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                // No synchronization needed to read the string
                observations.add(sharedString.substring(0, 5));
                observations.add(sharedString.toUpperCase().substring(0, 5));
            });
            threads[i].start();
        }
        for (Thread t : threads) {
            t.join();
        }
```

```java
        System.out.println("All threads saw consistent String state");
        System.out.println("Observations: " + observations);
    }

    // === 5. Class Loading Safety ===
    static void classLoadingSafety() {
        System.out.println("\n=== 5. Class Loading Safety ===");

        // Class names are strings - they must not change between
        // security check and actual loading
        String className = "java.util.ArrayList";
        System.out.println("Class: " + className);
        // If className could be mutated to "evil.MaliciousClass" after
        // the security check, the class loader would load the wrong class
        System.out.println("Immutability ensures class name integrity during loading");
    }

    // === Contrast: What if String were mutable? ===
    static void whatIfMutable() {
        System.out.println("\n=== What If String Were Mutable? ===");

        System.out.println("If String were mutable:");
        System.out.println("  - String Pool would be unsafe (shared modifications)");
        System.out.println("  - hashCode cache would go stale after mutation");
        System.out.println("  - HashMap keys would become corrupted");
        System.out.println("  - Thread safety would require synchronization everywhere");
        System.out.println("  - Security validation could be bypassed (TOCTOU attacks)");
        System.out.println("  - Class loading could be exploited");

        // StringBuilder IS mutable and demonstrates these risks:
        StringBuilder mutable = new StringBuilder("key");
        HashMap<String, String> map = new HashMap<>();
        map.put(mutable.toString(), "value");

        // If we could use mutable strings as keys:
        // mutable.replace(0, 3, "xxx");  // would corrupt the map!
        System.out.println("StringBuilder is mutable - never use as a map key");
    }

    public static void main(String[] args) throws InterruptedException {
        securityBenefit();
        stringPoolBenefit();
        hashCodeCachingBenefit();
        threadSafetyBenefit();
        classLoadingSafety();
        whatIfMutable();

        System.out.println("\n=== String Immutability Summary ===");
        System.out.println("Security:     Prevents modification after validation");
        System.out.println("String Pool:  Enables safe memory sharing of literals");
        System.out.println("hashCode:     Cached on first call, never stale");
        System.out.println("Thread safe:  Shared across threads without locks");
        System.out.println("Class loading: Ensures integrity of class names");
    }
}
```

What is the difference between Comparable and Comparator?

`Comparable` and `Comparator` are the two standard mechanisms for defining ordering relationships in Java. Understanding when to use each and how they interact with the collections framework is a foundational interview topic.

`Comparable<T>` defines the **natural ordering** of a class. A class implements `Comparable<T>` and provides a `compareTo(T)` method that returns a negative integer, zero, or a positive integer to indicate whether `this` is less than, equal to, or greater than the other object. The natural ordering is used by default in `Collections.sort()`, `Arrays.sort()`, `TreeSet`, and `TreeMap`. A critical best practice is to ensure that `compareTo()` is **consistent with equals** — that is, `a.compareTo(b) == 0` if and only if `a.equals(b)`. Inconsistency causes subtle bugs in sorted collections.

`Comparator<T>` defines an **external ordering** that is independent of the class itself. It is useful when you need multiple different orderings (e.g., sort employees by name, by salary, or by department) or when you cannot modify the class to implement `Comparable`. Since Java 8, `Comparator` provides powerful factory methods: `Comparator.comparing()`, `thenComparing()`, `reversed()`, `nullsFirst()`, and `nullsLast()`. These methods enable concise, readable, and composable comparator chains.

- **Comparable**: Internal, single natural ordering; implement in the class itself.
- **Comparator**: External, multiple orderings; passed to sort methods and sorted collections.
- **Consistency**: `compareTo` should be consistent with `equals` for correct `TreeSet`/`TreeMap` behavior.
- **Java 8+**: Use `Comparator.comparing()` chains instead of manual comparison logic.
- **Null safety**: Use `Comparator.nullsFirst()` or `nullsLast()` to handle null elements.

Listing 1.27: Comparable vs Comparator: natural ordering, factory methods, and null-safe comparisons

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

/**
 * Demonstrates Comparable (natural ordering) vs Comparator (external ordering),
 * including Java 8+ factory methods and null-safe comparisons.
 */
public class ComparableVsComparator {

    // === Comparable: Natural ordering (internal to the class) ===
    static class Employee implements Comparable<Employee> {
        private final String name;
        private final String department;
        private final double salary;

        Employee(String name, String department, double salary) {
            this.name = name;
            this.department = department;
            this.salary = salary;
        }

        // Natural ordering: by name (alphabetical)
        @Override
```

```java
        public int compareTo(Employee other) {
            return this.name.compareTo(other.name);
        }

        // Consistent with equals (important for TreeSet/TreeMap)
        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (!(o instanceof Employee e)) return false;
            return Double.compare(salary, e.salary) == 0
                    && name.equals(e.name) && department.equals(e.department);
        }

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

        @Override
        public String toString() {
            return "%s (%s, $%.0f)".formatted(name, department, salary);
        }

        public String getName() { return name; }
        public String getDepartment() { return department; }
        public double getSalary() { return salary; }
    }

    // === Natural ordering with Comparable ===
    static void naturalOrderingExample() {
        System.out.println("=== Natural Ordering (Comparable) ===");

        List<Employee> employees = new ArrayList<>(List.of(
                new Employee("Charlie", "Engineering", 95000),
                new Employee("Alice", "Marketing", 85000),
                new Employee("Bob", "Engineering", 105000)
        ));

        // Uses compareTo() - sorts by name (natural ordering)
        Collections.sort(employees);
        System.out.println("Sorted by name (natural): " + employees);

        // TreeSet uses natural ordering
        TreeSet<Employee> tree = new TreeSet<>(employees);
        System.out.println("TreeSet (natural order): " + tree);
    }

    // === External ordering with Comparator ===
    static void comparatorExamples() {
        System.out.println("\n=== External Ordering (Comparator) ===");

        List<Employee> employees = new ArrayList<>(List.of(
                new Employee("Charlie", "Engineering", 95000),
                new Employee("Alice", "Marketing", 85000),
                new Employee("Bob", "Engineering", 105000),
                new Employee("Diana", "Marketing", 92000)
        ));

        // Sort by salary (ascending) using lambda
        employees.sort((e1, e2) -> Double.compare(e1.getSalary(), e2.getSalary()));
        System.out.println("By salary (asc): " + employees);
```

```java

        // Sort by salary (descending) using Comparator.comparing + reversed
        employees.sort(Comparator.comparingDouble(Employee::getSalary).reversed());
        System.out.println("By salary (desc): " + employees);

        // Multi-field sort: by department, then by salary descending
        employees.sort(Comparator
                .comparing(Employee::getDepartment)
                .thenComparing(Comparator.comparingDouble(Employee::getSalary).reversed())
        );
        System.out.println("By dept, then salary desc: " + employees);
    }

    // === Java 8+ Comparator Factory Methods ===
    static void factoryMethods() {
        System.out.println("\n=== Comparator Factory Methods (Java 8+) ===");

        List<String> names = new ArrayList<>(List.of("Charlie", "alice", "Bob", "diana"));

        // Case-insensitive sort
        names.sort(String.CASE_INSENSITIVE_ORDER);
        System.out.println("Case-insensitive: " + names);

        // Comparator.comparing with key extractor
        names.sort(Comparator.comparing(String::length));
        System.out.println("By length: " + names);

        // Chained comparison: by length, then alphabetically
        names.sort(Comparator.comparing(String::length)
                .thenComparing(Comparator.naturalOrder()));
        System.out.println("By length, then alpha: " + names);

        // Reverse natural order
        names.sort(Comparator.reverseOrder());
        System.out.println("Reverse natural: " + names);
    }

    // === Null-Safe Comparisons ===
    static void nullSafeComparisons() {
        System.out.println("\n=== Null-Safe Comparisons ===");

        List<String> withNulls = new ArrayList<>(Arrays.asList("Charlie", null, "Alice", null, "Bob"));

        // nullsFirst: null elements sort before non-null elements
        withNulls.sort(Comparator.nullsFirst(Comparator.naturalOrder()));
        System.out.println("nullsFirst: " + withNulls);

        // nullsLast: null elements sort after non-null elements
        withNulls.sort(Comparator.nullsLast(Comparator.naturalOrder()));
        System.out.println("nullsLast:  " + withNulls);

        // Null-safe field extraction
        record Product(String name, String category) {}
        List<Product> products = List.of(
                new Product("Widget", "Tools"),
                new Product("Gadget", null),
                new Product("Doohickey", "Electronics")
        );
        List<Product> sorted = products.stream()
```

```
                .sorted(Comparator.comparing(Product::category,
                        Comparator.nullsLast(Comparator.naturalOrder())))
                .toList();
        System.out.println("Products (null category last): " + sorted);
    }

    // === Consistency with equals ===
    static void consistencyExample() {
        System.out.println("\n=== Consistency with equals ===");

        // BigDecimal violates consistency: compareTo treats 1.0 == 1.00, but equals doesn't
        var bd1 = new java.math.BigDecimal("1.0");
        var bd2 = new java.math.BigDecimal("1.00");
        System.out.println("BD compareTo: " + bd1.compareTo(bd2));  // 0
        System.out.println("BD equals:    " + bd1.equals(bd2));     // false!

        // This causes issues with TreeSet (uses compareTo) vs HashSet (uses equals)
        TreeSet<java.math.BigDecimal> treeSet = new TreeSet<>();
        treeSet.add(bd1);
        treeSet.add(bd2);
        System.out.println("TreeSet size: " + treeSet.size());  // 1 (compareTo says equal)

        HashSet<java.math.BigDecimal> hashSet = new HashSet<>();
        hashSet.add(bd1);
        hashSet.add(bd2);
        System.out.println("HashSet size: " + hashSet.size());  // 2 (equals says different)
    }

    public static void main(String[] args) {
        naturalOrderingExample();
        comparatorExamples();
        factoryMethods();
        nullSafeComparisons();
        consistencyExample();

        System.out.println("\n=== Comparable vs Comparator Summary ===");
        System.out.println("Comparable: Single natural ordering, implements in the class");
        System.out.println("Comparator: Multiple external orderings, passed to sort methods");
        System.out.println("Use Comparator.comparing() chains (Java 8+) for clean code");
        System.out.println("Use nullsFirst/nullsLast for null-safe sorting");
        System.out.println("Ensure compareTo is consistent with equals for sorted collections");
    }
}
```

How do you handle null safely in modern Java?

Null handling is a perennial source of bugs and interview discussion. Modern Java provides several complementary strategies for dealing with null values, and interviewers expect candidates to articulate when each approach is appropriate.

The first line of defense is **fail-fast validation** using `Objects.requireNonNull()`. This method throws `NullPointerException` with a descriptive message at the point where the null is detected, rather than allowing it to propagate and fail at an unrelated location. It should be used at the entry point of every public method for parameters that must not be null. Records with

compact constructors and `Objects.requireNonNull()` provide an elegant way to enforce non-null invariants on construction.

`Optional<T>` is the standard mechanism for return types that may legitimately be absent. It forces callers to explicitly handle the absent case, preventing accidental null dereference. However, `Optional` should **never** be used as a field type (it is not `Serializable` and adds overhead), as a method parameter (use method overloading instead), or as a collection element. The `map()`, `flatMap()`, and `or()` methods enable clean, functional-style null-safe navigation chains.

Java 21's **pattern matching switch with** `case null` provides the most recent advancement. It allows null to be handled as an explicit case alongside type patterns, eliminating the need for null checks before switch statements. Combined with sealed types, this produces comprehensive, exhaustive handling of all possible values including null.

- `Objects.requireNonNull()`: Fail-fast at method entry; descriptive NPE message.
- `Optional<T>`: For return types; forces caller to handle absence.
- **Pattern matching**: `case null` in switch (Java 21+) for explicit null handling.
- **Annotations**: `@NonNull` / `@Nullable` (JSR 305, JetBrains, Eclipse) for static analysis.
- **Anti-patterns**: `Optional` as field/parameter/collection element; catching NPE instead of checking.

Listing 1.28: Null safety: requireNonNull, Optional, pattern matching, and annotations

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

/**
 * Demonstrates modern null-safety strategies in Java:
 * - Objects.requireNonNull (fail-fast)
 * - Optional<T> (nullable return types)
 * - Pattern matching switch with case null (Java 21)
 * - Best practices and anti-patterns
 */
public class NullSafety {

    // === 1. Objects.requireNonNull: Fail-Fast Validation ===
    record User(String name, String email) {
        User {
            Objects.requireNonNull(name, "name must not be null");
            Objects.requireNonNull(email, "email must not be null");
            if (name.isBlank()) throw new IllegalArgumentException("name must not be blank");
        }
    }

    static void failFastValidation() {
        System.out.println("=== 1. Fail-Fast with requireNonNull ===");

        // Good: fails immediately with descriptive message
        try {
            new User(null, "test@example.com");
        } catch (NullPointerException e) {
            System.out.println("Caught at creation: " + e.getMessage());
        }

```

```java
        // Method parameter validation
        try {
            processUser(null);
        } catch (NullPointerException e) {
            System.out.println("Caught at method entry: " + e.getMessage());
        }

        // Default value for nullable (Java 9)
        String name = null;
        String safe = Objects.requireNonNullElse(name, "Anonymous");
        System.out.println("requireNonNullElse: " + safe);
    }

    static void processUser(User user) {
        Objects.requireNonNull(user, "user must not be null");
        System.out.println("Processing: " + user.name());
    }

    // === 2. Optional<T>: Nullable Return Types ===
    record Product(String name, Double price) {}

    static final Map<String, Product> CATALOG = Map.of(
            "WIDGET", new Product("Widget", 9.99),
            "GADGET", new Product("Gadget", 24.99)
    );

    // Return Optional to signal that the result may be absent
    static Optional<Product> findProduct(String code) {
        return Optional.ofNullable(CATALOG.get(code));
    }

    static void optionalExamples() {
        System.out.println("\n=== 2. Optional<T> for Nullable Returns ===");

        // map chain for safe navigation
        String desc = findProduct("WIDGET")
                .map(p -> p.name() + " ($" + p.price() + ")")
                .orElse("Product not found");
        System.out.println("Found: " + desc);

        String missing = findProduct("UNKNOWN")
                .map(p -> p.name())
                .orElse("Product not found");
        System.out.println("Missing: " + missing);

        // orElseThrow for mandatory values
        try {
            Product product = findProduct("UNKNOWN")
                    .orElseThrow(() -> new NoSuchElementException("Product not in catalog"));
        } catch (NoSuchElementException e) {
            System.out.println("Thrown: " + e.getMessage());
        }

        // ifPresentOrElse (Java 9)
        findProduct("GADGET").ifPresentOrElse(
                p -> System.out.println("Action: ship " + p.name()),
                () -> System.out.println("Action: notify out of stock")
        );

```

```java
        // or() for fallback Optional chain (Java 9)
        Optional<Product> fallback = findProduct("UNKNOWN")
                .or(() -> findProduct("WIDGET"));
        System.out.println("Fallback: " + fallback.map(Product::name).orElse("none"));

        // Stream integration with ofNullable
        List<String> codes = List.of("WIDGET", "MISSING", "GADGET", "UNKNOWN");
        List<String> found = codes.stream()
                .map(CATALOG::get)
                .flatMap(p -> Stream.ofNullable(p))
                .map(Product::name)
                .toList();
        System.out.println("Found products: " + found);
    }

    // === 3. Pattern Matching Switch with case null (Java 21) ===
    sealed interface Response permits Success, Error, Pending {}
    record Success(String data) implements Response {}
    record Error(String message) implements Response {}
    record Pending(int retryAfter) implements Response {}

    static String handleResponse(Response response) {
        return switch (response) {
            case null               -> "No response received";
            case Success(var data) -> "OK: " + data;
            case Error(var msg)    -> "ERROR: " + msg;
            case Pending(var secs) -> "Retry after " + secs + "s";
        };
    }

    static void patternMatchingNull() {
        System.out.println("\n=== 3. Pattern Matching with case null (Java 21) ===");

        System.out.println(handleResponse(new Success("payload")));
        System.out.println(handleResponse(new Error("timeout")));
        System.out.println(handleResponse(new Pending(30)));
        System.out.println(handleResponse(null));
    }

    // === 4. Anti-Patterns to Avoid ===
    static void antiPatterns() {
        System.out.println("\n=== 4. Anti-Patterns ===");

        // ANTI-PATTERN 1: Optional as a field
        // class BadEntity { Optional<String> name; }  // Don't do this!
        // Optional is not Serializable and adds 16 bytes overhead per field
        System.out.println("Anti-pattern: Optional as field (not Serializable, overhead)");

        // ANTI-PATTERN 2: Optional as method parameter
        // void process(Optional<String> name) { }  // Don't do this!
        // Use method overloading or @Nullable annotation instead
        System.out.println("Anti-pattern: Optional as parameter (use overloading instead)");

        // ANTI-PATTERN 3: Optional.get() without isPresent()
        // Optional.of("x").get();  // Works but defeats the purpose
        // Use orElse, orElseGet, orElseThrow instead
        System.out.println("Anti-pattern: Optional.get() without checking (use orElse*)");

        // ANTI-PATTERN 4: Catching NPE instead of null-checking
```

```java
        // try { obj.method(); } catch (NullPointerException e) { }  // Terrible!
        System.out.println("Anti-pattern: catching NPE instead of null-checking");

        // ANTI-PATTERN 5: Returning null from Optional-returning method
        // static Optional<X> find() { return null; }  // Defeats the entire purpose!
        System.out.println("Anti-pattern: returning null from Optional method");
    }

    // === 5. Comprehensive Null-Safe Service Pattern ===
    static class UserService {
        private final Map<Integer, User> users = Map.of(
                1, new User("Alice", "alice@example.com"),
                2, new User("Bob", "bob@example.com")
        );

        // Return Optional for lookups
        Optional<User> findById(int id) {
            return Optional.ofNullable(users.get(id));
        }

        // Require non-null for mutations
        void update(int id, User user) {
            Objects.requireNonNull(user, "user must not be null");
            // ... update logic
        }
    }

    static void servicePattern() {
        System.out.println("\n=== 5. Null-Safe Service Pattern ===");

        var service = new UserService();

        // Safe lookup with Optional
        String email = service.findById(1)
                .map(User::email)
                .orElse("not found");
        System.out.println("User 1 email: " + email);

        String missing = service.findById(99)
                .map(User::email)
                .orElse("not found");
        System.out.println("User 99 email: " + missing);
    }

    public static void main(String[] args) {
        failFastValidation();
        optionalExamples();
        patternMatchingNull();
        antiPatterns();
        servicePattern();

        System.out.println("\n=== Null Safety Strategy ===");
        System.out.println("1. requireNonNull: fail-fast at method entry and constructors");
        System.out.println("2. Optional<T>: for return types that may be absent");
        System.out.println("3. case null: explicit null handling in switch (Java 21+)");
        System.out.println("4. @NonNull/@Nullable: annotations for static analysis");
        System.out.println("5. Never: Optional as field/param, catch NPE, return null Optional");
    }
}
```

1.9 Debugging and Scenario Questions

What happens when you concatenate strings in a loop and how do you fix it?

String concatenation using the `+` operator inside a loop is one of the most common performance pitfalls in Java. While a single concatenation expression like `"Hello " + name + "!"` is efficiently optimized by the compiler (using `invokedynamic` and `StringConcatFactory` since Java 9), concatenation inside a loop tells a fundamentally different story. Each iteration of `result += word` allocates a new `String` object, copies the entire accumulated content plus the new word into a fresh backing array, and discards the previous string as garbage. For a loop of n iterations with an average string length of k, this produces $O(n^2 \cdot k)$ total character copies.

The fix is straightforward: use `StringBuilder` with an estimated initial capacity. `StringBuilder` maintains a resizable internal buffer and appends without creating intermediate `String` objects. The amortized cost of `n` appends is $O(n \cdot k)$. For delimiter-based joining, `String.join()` and `Collectors.joining()` provide clean, optimized alternatives. A common interview follow-up asks about the difference between `StringBuilder` (not synchronized, preferred) and `StringBuffer` (synchronized, legacy).

Pre-sizing the `StringBuilder` avoids internal array resizing. The default capacity is 16 characters; when exceeded, the buffer doubles plus two. If you can estimate the total output length (e.g., number of words times average word length), passing that estimate to the constructor eliminates resize overhead entirely.

- **Single expression**: `+` is compiler-optimized; no issue.
- **Loop concatenation**: $O(n^2)$ copies; always use `StringBuilder`.
- `String.join()`: Ideal for delimiter-separated concatenation.
- **Pre-size**: Estimate total length to avoid buffer resizing.
- `StringBuffer`: Synchronized legacy version; prefer `StringBuilder`.

Listing 1.29: String concatenation performance: loop pitfall, StringBuilder, and alternatives

```java
import java.util.*;

/**
 * Demonstrates string concatenation performance issues:
 * - Naive loop concatenation vs StringBuilder
 * - Bytecode behavior of the + operator
 * - Benchmarking the difference
 * - String.join() and Collectors.joining() alternatives
 */
public class StringConcatPerformance {

    // === 1. The Problem: String Concatenation in a Loop ===
    static String naiveConcatenation(List<String> words) {
        String result = "";
        for (String word : words) {
            // Each += creates a new String object:
            // 1. Allocates a new char[]/byte[] array
            // 2. Copies existing content + new word
```

```
            // 3. Old String becomes garbage
            result += word + " ";
        }
        return result.trim();
    }

    // === 2. The Fix: StringBuilder ===
    static String builderConcatenation(List<String> words) {
        StringBuilder sb = new StringBuilder(words.size() * 8); // estimated capacity
        for (String word : words) {
            if (!sb.isEmpty()) sb.append(' ');
            sb.append(word);
        }
        return sb.toString();
    }

    // === 3. Modern Alternatives ===
    static String joinConcatenation(List<String> words) {
        return String.join(" ", words);  // Java 8+
    }

    static String streamConcatenation(List<String> words) {
        return words.stream()
                .collect(java.util.stream.Collectors.joining(" "));
    }

    // === 4. Benchmark: Measure the Difference ===
    static void benchmark() {
        System.out.println("=== Benchmark: Concatenation Approaches ===");

        List<String> words = new ArrayList<>();
        for (int i = 0; i < 10_000; i++) {
            words.add("word" + i);
        }

        // Naive concatenation (O(n^2) due to repeated copying)
        long start = System.nanoTime();
        String r1 = naiveConcatenation(words);
        long naiveTime = System.nanoTime() - start;

        // StringBuilder (O(n) amortized)
        start = System.nanoTime();
        String r2 = builderConcatenation(words);
        long builderTime = System.nanoTime() - start;

        // String.join
        start = System.nanoTime();
        String r3 = joinConcatenation(words);
        long joinTime = System.nanoTime() - start;

        System.out.printf("Naive +=:       %,d ms%n", naiveTime / 1_000_000);
        System.out.printf("StringBuilder:  %,d ms%n", builderTime / 1_000_000);
        System.out.printf("String.join:    %,d ms%n", joinTime / 1_000_000);
        System.out.printf("Speedup (SB vs naive): ~%dx%n",
                naiveTime / Math.max(builderTime, 1));
    }

    // === 5. Single Expression: Compiler Optimizes Automatically ===
    static void singleExpressionDemo() {
```

```java
        System.out.println("\n=== Single Expression vs Loop ===");

        // The compiler optimizes a single concatenation expression:
        // Since Java 9+, javac uses invokedynamic + StringConcatFactory
        String name = "Alice";
        int age = 30;
        String single = "Name: " + name + ", Age: " + age;
        // This is efficient! No need for StringBuilder here.
        System.out.println(single);

        // The problem is ONLY in loops, where each iteration
        // creates a new StringBuilder (pre-Java 9) or new concat call
        System.out.println("Single expression: compiler-optimized, no issue");
        System.out.println("Loop concatenation: O(n^2) copies, use StringBuilder");
    }

    // === 6. Pre-sizing StringBuilder ===
    static void preSizingDemo() {
        System.out.println("\n=== Pre-sizing StringBuilder ===");

        // Default capacity is 16 characters
        StringBuilder defaultSb = new StringBuilder();
        System.out.println("Default capacity: " + defaultSb.capacity());

        // When capacity is exceeded, internal array doubles + 2
        // Pre-sizing avoids repeated resizing
        int estimatedSize = 1000;
        StringBuilder preSized = new StringBuilder(estimatedSize);
        System.out.println("Pre-sized capacity: " + preSized.capacity());
        System.out.println("Tip: Estimate total length to avoid resizing");
    }

    public static void main(String[] args) {
        benchmark();
        singleExpressionDemo();
        preSizingDemo();

        System.out.println("\n=== String Concatenation Rules ===");
        System.out.println("1. Single expression: + operator is fine (compiler optimizes)");
        System.out.println("2. Loop: always use StringBuilder or String.join()");
        System.out.println("3. Pre-size StringBuilder when total length is estimable");
        System.out.println("4. String.join() / Collectors.joining() for delimiter-based concat");
        System.out.println("5. Naive loop concat is O(n^2); StringBuilder is O(n)");
    }
}
```

How does Java handle integer overflow and how do you detect it?

Java's integer arithmetic uses fixed-width two's complement representation and silently wraps around on overflow. When you add 1 to `Integer.MAX_VALUE` (2,147,483,647), the result is `Integer.MIN_VALUE` (-2,147,483,648) — no exception is thrown, no warning is issued. This silent wraparound is a frequent source of subtle bugs in areas such as array index calculations, financial computations, and timestamp conversions.

Since Java 8, the `Math` class provides **exact arithmetic methods** that throw

`ArithmeticException` on overflow: `Math.addExact()`, `Math.subtractExact()`, `Math.multiplyExact()`, `Math.incrementExact()`, `Math.decrementExact()`, and `Math.negateExact()`. These methods perform the same operation as the standard operators but include an overflow check. They are the recommended approach whenever overflow would represent a logic error rather than an expected wraparound.

For preventive coding, the most effective strategy is to **widen before the operation**. Cast one operand to `long` before multiplication or addition: `long result = (long) a * b`. This ensures the intermediate result has 64 bits of range. For counters and accumulators that may exceed `int` range over time, declare them as `long` from the start. For truly arbitrary precision, `BigInteger` is available but carries a significant performance cost.

- **Default behavior**: Silent two's complement wraparound.
- **Detection**: `Math.addExact()`, `multiplyExact()`, etc. throw on overflow.
- **Prevention**: Cast to `long` *before* the operation, not after.
- **Counters**: Use `long` for values that accumulate over time.
- **Common traps**: Array size calculation, millis-to-nanos conversion, pixel indexing.

Listing 1.30: Integer overflow: silent wraparound, Math.addExact, and defensive strategies

```java
import java.util.*;

/**
 * Demonstrates Java integer overflow behavior:
 * - Silent wraparound (default behavior)
 * - Detection with Math.addExact / Math.multiplyExact
 * - Overflow in common scenarios (array size, counters)
 * - Defensive coding strategies
 */
public class IntegerOverflow {

    // === 1. Silent Wraparound: The Default Behavior ===
    static void silentWraparound() {
        System.out.println("=== 1. Silent Wraparound ===");

        int max = Integer.MAX_VALUE;  // 2,147,483,647
        int overflowed = max + 1;     // Wraps to -2,147,483,648
        System.out.println("MAX_VALUE:     " + max);
        System.out.println("MAX_VALUE + 1: " + overflowed);
        System.out.println("Wrapped to MIN_VALUE: " + (overflowed == Integer.MIN_VALUE));

        // Multiplication overflow
        int billion = 1_000_000_000;
        int product = billion * 3;    // Wraps silently
        System.out.println("1B * 3 (int):  " + product + " (WRONG, expected 3B)");
        long correct = (long) billion * 3;
        System.out.println("1B * 3 (long): " + correct + " (CORRECT)");
    }

    // === 2. Detection with Math.xxxExact() (Java 8+) ===
    static void exactArithmetic() {
        System.out.println("\n=== 2. Math.addExact / multiplyExact ===");

        // addExact throws ArithmeticException on overflow
```

```java
        try {
            int result = Math.addExact(Integer.MAX_VALUE, 1);
            System.out.println("Result: " + result); // never reached
        } catch (ArithmeticException e) {
            System.out.println("addExact caught overflow: " + e.getMessage());
        }

        // multiplyExact
        try {
            int result = Math.multiplyExact(1_000_000_000, 3);
        } catch (ArithmeticException e) {
            System.out.println("multiplyExact caught overflow: " + e.getMessage());
        }

        // subtractExact
        try {
            int result = Math.subtractExact(Integer.MIN_VALUE, 1);
        } catch (ArithmeticException e) {
            System.out.println("subtractExact caught overflow: " + e.getMessage());
        }

        // Safe increment / decrement
        try {
            int result = Math.incrementExact(Integer.MAX_VALUE);
        } catch (ArithmeticException e) {
            System.out.println("incrementExact caught overflow: " + e.getMessage());
        }
    }

    // === 3. Real-World Overflow Scenarios ===
    static void realWorldScenarios() {
        System.out.println("\n=== 3. Real-World Overflow Scenarios ===");

        // Scenario A: Array index calculation
        int width = 50_000;
        int height = 50_000;
        int pixelIndex = width * height;  // 2.5 billion > MAX_VALUE
        System.out.println("50000 x 50000 pixels (int): " + pixelIndex + " (OVERFLOW)");
        long safeIndex = (long) width * height;
        System.out.println("50000 x 50000 pixels (long): " + safeIndex + " (correct)");

        // Scenario B: Milliseconds to nanos conversion
        long millis = 1_000_000_000L;  // ~11.5 days
        // int nanos = (int)(millis * 1_000_000); // would overflow int
        long nanos = millis * 1_000_000L;
        System.out.println("Millis-to-nanos (long): " + nanos);

        // Scenario C: Sum of large collection
        int sum = 0;
        for (int i = 0; i < 100_000; i++) {
            sum += Integer.MAX_VALUE / 50_000; // accumulates past MAX_VALUE
        }
        System.out.println("Accumulated sum (int): " + sum + " (likely overflowed)");
    }

    // === 4. Defensive Strategies ===
    static long safeMultiply(int a, int b) {
        return (long) a * b;  // Widen BEFORE multiplication
    }
```

```java

    static int checkedAdd(int a, int b) {
        return Math.addExact(a, b);  // Throws on overflow
    }

    static void defensiveStrategies() {
        System.out.println("\n=== 4. Defensive Strategies ===");

        // Strategy 1: Widen to long before the operation
        System.out.println("Safe multiply: " + safeMultiply(100_000, 100_000));

        // Strategy 2: Use Math.xxxExact for critical calculations
        System.out.println("Checked add: " + checkedAdd(100, 200));

        // Strategy 3: Use long from the start for counters and accumulators
        long counter = 0L;  // not int
        for (int i = 0; i < 100_000; i++) counter += 100_000;
        System.out.println("Long counter: " + counter);

        // Strategy 4: BigInteger for arbitrary precision (rare)
        var big = java.math.BigInteger.valueOf(Long.MAX_VALUE)
                .multiply(java.math.BigInteger.valueOf(2));
        System.out.println("BigInteger: " + big);
    }

    public static void main(String[] args) {
        silentWraparound();
        exactArithmetic();
        realWorldScenarios();
        defensiveStrategies();

        System.out.println("\n=== Integer Overflow Rules ===");
        System.out.println("1. Java integer arithmetic wraps silently on overflow");
        System.out.println("2. Use Math.addExact/multiplyExact for overflow detection");
        System.out.println("3. Cast to long BEFORE the operation, not after");
        System.out.println("4. Use long for counters, accumulators, and size calculations");
        System.out.println("5. Be especially careful with array indexing and pixel math");
    }
}
```

What is the difference between `String.format()`, MessageFormat, and string concatenation?

Java offers several string formatting mechanisms, each suited to different scenarios. The `+` concatenation operator is the simplest and fastest for basic expressions. Since Java 9, the compiler translates single-expression concatenations into `invokedynamic` calls to `StringConcatFactory`, which generates highly optimized bytecode. For simple cases like `"Name: " + name + ", Age: " + age`, the `+` operator is the right choice.

`String.format()` (and its instance counterpart `formatted()` introduced in Java 15) provides C-style format specifiers: `%s` for strings, `%d` for integers, `%.2f` for fixed-precision decimals, `%x` for hexadecimal, and width/alignment modifiers. It is the standard choice when you need precise numeric formatting, padding, or alignment. However, it is significantly slower than

concatenation because it parses the format string at each invocation.

`MessageFormat` from `java.text` is designed for internationalization. It uses positional placeholders (`{0}`, `{1}`) that can be reused multiple times in the pattern, supports locale-aware number and date formatting, and provides choice formats for pluralization (`{0,choice,...}`). It is the slowest option due to its rich formatting capabilities but is essential for localized user-facing messages.

- **Concatenation (+)**: Fastest; compiler-optimized; use for simple expressions.
- `String.format()`: Precise formatting (padding, precision, hex); moderate cost.
- `MessageFormat`: Localization, pluralization, reusable positional args; slowest.
- `StringBuilder`: Loop concatenation; pre-sized buffer for known output size.
- **Text blocks +** `formatted()`: Multi-line templates (Java 15+).

Listing 1.31: String formatting: concatenation, String.format, MessageFormat, and text blocks

```java
import java.text.MessageFormat;
import java.util.*;

/**
 * Demonstrates string formatting approaches in Java:
 * - String concatenation (+)
 * - String.format() / formatted()
 * - MessageFormat
 * - StringBuilder
 * - Text blocks with formatting
 * - Performance and use-case comparison
 */
public class StringFormatting {

    // === 1. String.format() / String.formatted() ===
    static void stringFormatExamples() {
        System.out.println("=== 1. String.format() and formatted() ===");

        String name = "Alice";
        double price = 49.99;
        int quantity = 3;

        // Classic String.format (static method)
        String s1 = String.format("Customer: %s, Total: $%.2f", name, price * quantity);
        System.out.println(s1);

        // Instance method formatted() (Java 15+)
        String s2 = "Customer: %s, Items: %d".formatted(name, quantity);
        System.out.println(s2);

        // Format specifiers
        System.out.printf("Padded: [%20s]%n", name);           // right-aligned
        System.out.printf("Padded: [%-20s]%n", name);          // left-aligned
        System.out.printf("Hex: %#x, Octal: %#o%n", 255, 255);
        System.out.printf("Scientific: %e%n", 123456.789);
        System.out.printf("Locale number: %,d%n", 1_000_000);
    }

    // === 2. MessageFormat (Localization-Friendly) ===
    static void messageFormatExamples() {
```

```java
        System.out.println("\n=== 2. MessageFormat ===");

        // Positional arguments with {0}, {1}, etc.
        String pattern = "At {0}, {1} purchased {2} items for {3,number,currency}.";
        String result = MessageFormat.format(pattern,
                new Date(), "Bob", 5, 149.95);
        System.out.println(result);

        // Reusing the same argument
        String repeated = MessageFormat.format(
                "{0} said: 'My name is {0} and I am {1} years old.'",
                "Charlie", 30);
        System.out.println(repeated);

        // Choice format for pluralization
        String choicePattern = "{0} {0,choice,0#items|1#item|1<items} in cart.";
        System.out.println(MessageFormat.format(choicePattern, 0));
        System.out.println(MessageFormat.format(choicePattern, 1));
        System.out.println(MessageFormat.format(choicePattern, 5));
    }

    // === 3. Concatenation Operator (+) ===
    static void concatenationExamples() {
        System.out.println("\n=== 3. Concatenation (+) ===");

        String name = "Diana";
        int age = 28;

        // Simple concatenation: compiler-optimized since Java 9
        // Uses invokedynamic + StringConcatFactory (efficient)
        String s = "Name: " + name + ", Age: " + age;
        System.out.println(s);

        // Type coercion rules
        System.out.println("Result: " + 1 + 2);     // "Result: 12" (string context)
        System.out.println("Result: " + (1 + 2));    // "Result: 3" (parens force int add)
        System.out.println(1 + 2 + " result");       // "3 result" (int add first)
    }

    // === 4. Text Blocks with Formatting (Java 15+) ===
    static void textBlockFormatting() {
        System.out.println("\n=== 4. Text Blocks with Formatting ===");

        String name = "Eve";
        double balance = 1234.56;

        String json = """
                {
                    "name": "%s",
                    "balance": %.2f
                }
                """.formatted(name, balance);
        System.out.println(json);
    }

    // === 5. Performance Comparison ===
    static void performanceComparison() {
        System.out.println("=== 5. Performance Comparison ===");

```

```java
        int iterations = 100_000;
        String name = "Test";
        int value = 42;

        // Concatenation
        long start = System.nanoTime();
        for (int i = 0; i < iterations; i++) {
            String s = "Name: " + name + ", Value: " + value;
        }
        long concatTime = System.nanoTime() - start;

        // String.format
        start = System.nanoTime();
        for (int i = 0; i < iterations; i++) {
            String s = String.format("Name: %s, Value: %d", name, value);
        }
        long formatTime = System.nanoTime() - start;

        // MessageFormat
        start = System.nanoTime();
        for (int i = 0; i < iterations; i++) {
            String s = MessageFormat.format("Name: {0}, Value: {1}", name, value);
        }
        long msgTime = System.nanoTime() - start;

        // StringBuilder
        start = System.nanoTime();
        for (int i = 0; i < iterations; i++) {
            String s = new StringBuilder("Name: ").append(name)
                    .append(", Value: ").append(value).toString();
        }
        long sbTime = System.nanoTime() - start;

        System.out.printf("Concatenation:  %,d ms%n", concatTime / 1_000_000);
        System.out.printf("StringBuilder:  %,d ms%n", sbTime / 1_000_000);
        System.out.printf("String.format:  %,d ms%n", formatTime / 1_000_000);
        System.out.printf("MessageFormat:  %,d ms%n", msgTime / 1_000_000);
    }

    public static void main(String[] args) {
        stringFormatExamples();
        messageFormatExamples();
        concatenationExamples();
        textBlockFormatting();
        performanceComparison();

        System.out.println("\n=== When to Use What ===");
        System.out.println("+ operator:     Simple expressions; fastest; compiler-optimized");
        System.out.println("String.format:  Complex formatting (padding, precision, hex)");
        System.out.println("MessageFormat:  Localization, reusable patterns, pluralization");
        System.out.println("StringBuilder:  Loop concatenation; pre-sized buffer");
        System.out.println("Text block:     Multi-line templates with .formatted()");
    }
}
```

How do you implement a proper value-based class in Java?

A **value-based class** is one whose instances are distinguished solely by their content, not by their identity. Two instances with the same field values are interchangeable, and the class makes no guarantees about identity (i.e., whether `==` returns true for equal instances). Since Java 16, the JDK marks several classes with `@jdk.internal.ValueBased`, including `Integer`, `Long`, `Double`, `Optional`, `LocalDate`, `Instant`, and all `java.time` types.

To implement a value-based class: make it **final** (or use a record), make all fields **final**, provide no public constructors (use **static factory methods** like `of()` or `from()`), implement `equals()` and `hashCode()` based purely on field content, and ensure all modification operations return **new instances** rather than mutating state. Records in modern Java are naturally value-based: they are implicitly final, have final fields, auto-generate content-based `equals()`/`hashCode()`, and their canonical constructor enforces immutability.

Critical gotchas: never **synchronize** on value-based instances (the JVM may share instances across threads via caching), and never rely on `==` for comparison. The integer cache (`-128` to `127`) makes `==` appear to work for small `Integer` values but fail for larger ones. These pitfalls become more severe as Project Valhalla introduces inline types, where value-based classes may lose their identity entirely.

- **Immutable**: All fields final; operations return new instances.
- **No identity**: `equals()`/`hashCode()` based on content; never use `==`.
- **Factory methods**: Static `of()`/`from()` instead of public constructors.
- **No synchronization**: Never use value-based instances as lock targets.
- **Records**: Ideal modern implementation; auto-generated value semantics.

Listing 1.32: Value-based classes: factory methods, immutability, records, and gotchas

```java
import java.util.*;

/**
 * Demonstrates value-based classes in Java:
 * - What makes a class value-based (no identity)
 * - Factory methods instead of constructors
 * - Records as modern value-based classes
 * - @ValueBased annotation (Java 16+)
 * - Gotchas: synchronization, identity comparison
 */
public class ValueBasedClass {

    // === 1. Classic Value-Based Class (Manual Implementation) ===
    // A value-based class has NO identity; only its content matters.
    static final class Money {
        private final String currency;
        private final long amountInCents;

        // Private constructor: force use of factory methods
        private Money(String currency, long amountInCents) {
            this.currency = Objects.requireNonNull(currency);
            this.amountInCents = amountInCents;
        }
```

```java

        // Factory method (enables caching, validation, future Valhalla migration)
        static Money of(String currency, long amountInCents) {
            return new Money(currency, amountInCents);
        }

        static Money dollars(double amount) {
            return new Money("USD", Math.round(amount * 100));
        }

        // Derived operations return new instances (immutable)
        Money add(Money other) {
            if (!this.currency.equals(other.currency)) {
                throw new IllegalArgumentException("Currency mismatch");
            }
            return new Money(currency, this.amountInCents + other.amountInCents);
        }

        Money multiply(int factor) {
            return new Money(currency, amountInCents * factor);
        }

        // Identity-free equals: based on content only
        @Override
        public boolean equals(Object o) {
            return o instanceof Money m
                    && currency.equals(m.currency)
                    && amountInCents == m.amountInCents;
        }

        @Override
        public int hashCode() { return Objects.hash(currency, amountInCents); }

        @Override
        public String toString() {
            return "%s %.2f".formatted(currency, amountInCents / 100.0);
        }
    }

    // === 2. Record as Value-Based Class (Modern Approach) ===
    record Point(double x, double y) {
        // Records are naturally value-based:
        // - Immutable (final fields)
        // - Auto-generated equals/hashCode based on content
        // - No identity guarantees

        Point translate(double dx, double dy) {
            return new Point(x + dx, y + dy);  // returns new instance
        }

        double distanceTo(Point other) {
            return Math.sqrt(Math.pow(x - other.x, 2) + Math.pow(y - other.y, 2));
        }
    }

    // === 3. Gotchas with Value-Based Classes ===
    static void identityGotchas() {
        System.out.println("=== 3. Value-Based Identity Gotchas ===");

```

```java
        // GOTCHA 1: Do NOT synchronize on value-based instances
        Integer boxed = 42;
        // synchronized (boxed) { } // WARNING: Integer is value-based since Java 16

        // JDK classes marked @ValueBased (Java 16+):
        // Integer, Long, Double, Optional, LocalDate, Instant, etc.
        System.out.println("WARNING: Do not synchronize on value-based instances");
        System.out.println("Affected JDK types: Integer, Long, Optional, LocalDate, etc.");

        // GOTCHA 2: Do NOT rely on == for value-based instances
        Integer a = 200;
        Integer b = 200;
        System.out.println("a == b (Integer 200): " + (a == b)); // false (outside cache)
        System.out.println("a.equals(b):          " + a.equals(b)); // true

        Integer c = 42;
        Integer d = 42;
        System.out.println("c == d (Integer 42):  " + (c == d));  // true (cached -128..127)
        System.out.println("Lesson: always use equals() for value-based types");

        // GOTCHA 3: Optional is value-based
        Optional<String> opt1 = Optional.of("hello");
        Optional<String> opt2 = Optional.of("hello");
        System.out.println("Optional == :     " + (opt1 == opt2)); // false
        System.out.println("Optional equals:  " + opt1.equals(opt2)); // true
    }

    // === 4. Characteristics of Value-Based Classes ===
    static void characteristics() {
        System.out.println("\n=== 4. Value-Based Class Characteristics ===");

        Money m1 = Money.dollars(19.99);
        Money m2 = Money.dollars(19.99);

        // Content equality, not identity
        System.out.println("m1.equals(m2): " + m1.equals(m2));  // true
        System.out.println("m1 == m2:      " + (m1 == m2));      // false (but irrelevant)

        // Immutable: operations return new instances
        Money m3 = m1.add(m2);
        System.out.println("m1: " + m1);
        System.out.println("m3 (m1 + m2): " + m3);
        System.out.println("m1 unchanged: " + m1);

        // Records as value-based
        Point p1 = new Point(1, 2);
        Point p2 = p1.translate(3, 4);
        System.out.println("p1: " + p1 + " (unchanged)");
        System.out.println("p2: " + p2 + " (new instance)");
    }

    public static void main(String[] args) {
        identityGotchas();
        characteristics();

        System.out.println("\n=== Value-Based Class Rules ===");
        System.out.println("1. Immutable: all fields final, operations return new instances");
        System.out.println("2. No identity: equals/hashCode based on content, not reference");
        System.out.println("3. Factory methods: use static of() instead of public constructors");
```

```
        System.out.println("4. Never synchronize on value-based instances");
        System.out.println("5. Never use == for comparison; always use equals()");
        System.out.println("6. Records are ideal for value-based classes in modern Java");
    }
}
```

What are the gotchas with Java's type promotion rules?

Java's type promotion rules are a frequent source of interview questions and real-world bugs. The fundamental rule is that `byte`, `short`, and `char` are **promoted to** `int` for any arithmetic operation. This means `byte + byte` produces an `int`, not a `byte`. Assigning the result back to a `byte` requires an explicit cast: `byte c = (byte)(a + b)`. This applies to all binary operators including `+`, `-`, `*`, `/`, `%`, and bitwise operators.

The **ternary operator** (`? :`) unifies the types of its two branches to the wider type. If one branch is `int` and the other is `double`, the result is `double`. A subtle trap involves `Integer` boxing: if one branch is `Integer` (boxed) and the other is `int` (primitive), the `Integer` is unboxed, which throws `NullPointerException` if it is `null`. Constant expressions that fit in the target type are special-cased: `byte b = true ? x : 2` compiles because `2` is a constant that fits in a `byte`.

Compound assignment operators (`+=`, `-=`, `*=`) contain an **implicit cast** that can silently truncate values. `byte b = 127; b += 1;` compiles without error but wraps to `-128` because it is equivalent to `b = (byte)(b + 1)`. This implicit narrowing cast is one of Java's most dangerous hidden behaviors.

- **Arithmetic**: `byte`/`short`/`char` always promoted to `int`.
- **Ternary**: Unifies to wider type; beware `Integer` unboxing NPE.
- **Compound assignment**: Implicit narrowing cast (`+=` hides truncation).
- **Char arithmetic**: `'A' + 1` is `int`; requires cast for `char` result.
- **Overloading**: Resolved to smallest matching type in the promotion chain.

Listing 1.33: Type promotion gotchas: arithmetic promotion, ternary widening, and compound assignment

```
import java.util.*;

/**
 * Demonstrates Java's type promotion rules and gotchas:
 * - byte + byte = int (arithmetic promotion)
 * - Ternary operator type widening
 * - Compound assignment implicit casting
 * - Char arithmetic
 * - Promotion in method overloading
 */
public class TypePromotion {

    // === 1. Arithmetic Promotion: byte + byte = int ===
    static void arithmeticPromotion() {
        System.out.println("=== 1. Arithmetic Promotion ===");

        byte a = 10;
        byte b = 20;
```

```java
        // byte c = a + b;  // COMPILE ERROR: a + b is promoted to int
        int c = a + b;      // OK: result is int
        byte d = (byte)(a + b);  // OK: explicit cast (may lose data)

        System.out.println("byte + byte = int: " + c + " (type: int)");

        short s1 = 100;
        short s2 = 200;
        // short s3 = s1 + s2;  // COMPILE ERROR: promoted to int
        int s3 = s1 + s2;
        System.out.println("short + short = int: " + s3);

        // Rule: byte, short, char are promoted to int for ANY arithmetic
        char ch = 'A';
        // char ch2 = ch + 1;  // COMPILE ERROR: promoted to int
        char ch2 = (char)(ch + 1);
        System.out.println("'A' + 1 = '" + ch2 + "' (requires cast to char)");
    }

    // === 2. Ternary Operator Type Widening ===
    static void ternaryWidening() {
        System.out.println("\n=== 2. Ternary Operator Widening ===");

        // The ternary operator unifies types to the wider type
        boolean condition = true;

        // char and int: result type is int
        char ch = 'X';
        int num = 88;
        // char result = condition ? ch : num;  // COMPILE ERROR if num is variable
        System.out.println("char vs int in ternary: result is int");

        // int and double: result type is double
        int i = 42;
        double d = 3.14;
        double result = condition ? i : d;  // i is promoted to double
        System.out.println("int vs double ternary: " + result + " (type: double)");

        // Subtle: Integer unboxing with null risk
        Integer boxed = null;
        // int unboxed = (boxed != null) ? boxed : 0;  // safe here
        // int danger = condition ? boxed : 0;  // NPE if boxed is null and condition true
        System.out.println("Ternary with Integer: beware of unboxing NPE");

        // Byte literal constant in ternary
        byte b = 1;
        // byte r = true ? b : 2;  // 2 is a constant that fits in byte, so OK
        byte r = true ? b : 2;
        System.out.println("Constant literal ternary: byte result = " + r);
    }

    // === 3. Compound Assignment: Implicit Cast ===
    static void compoundAssignment() {
        System.out.println("\n=== 3. Compound Assignment Implicit Cast ===");

        byte b = 100;
        // b = b + 1;   // COMPILE ERROR: b + 1 is int
        b += 1;         // OK: compound assignment implicitly casts (byte)(b + 1)
        System.out.println("b += 1: " + b);
```

```java

        // This can silently truncate!
        byte overflow = 127;
        overflow += 1;  // No error! Silently wraps to -128
        System.out.println("127 += 1: " + overflow + " (silent overflow!)");

        short s = 30000;
        s += 30000;  // Silently wraps
        System.out.println("30000 += 30000 (short): " + s + " (overflow)");

        // *= also has implicit cast
        byte x = 10;
        x *= 20;  // (byte)(10 * 20) = (byte)200 = -56
        System.out.println("10 *= 20 (byte): " + x + " (truncated)");
    }

    // === 4. Char Arithmetic ===
    static void charArithmetic() {
        System.out.println("\n=== 4. Char Arithmetic ===");

        char digit = '7';
        int numericValue = digit - '0';  // char - char = int
        System.out.println("'7' - '0' = " + numericValue);

        // Iterating over letters
        for (char c = 'A'; c <= 'F'; c++) {
            System.out.print(c + " ");  // prints A B C D E F
        }
        System.out.println();

        // char is unsigned 16-bit (0 to 65535)
        char maxChar = '\uffff';
        System.out.println("Max char value: " + (int) maxChar);

        // Gotcha: char + char = int
        char a = 'A';  // 65
        char b = 'B';  // 66
        System.out.println("'A' + 'B' = " + (a + b) + " (int, not char)");
    }

    // === 5. Method Overloading and Promotion ===
    static void print(int x)    { System.out.println("  int: " + x); }
    static void print(long x)   { System.out.println("  long: " + x); }
    static void print(double x) { System.out.println("  double: " + x); }

    static void overloadingPromotion() {
        System.out.println("\n=== 5. Overloading Resolution with Promotion ===");

        byte b = 42;
        short s = 42;
        print(b);    // promotes byte -> int (smallest matching overload)
        print(s);    // promotes short -> int
        print(42);   // int matches directly
        print(42L);  // long matches directly
        print(42.0); // double matches directly

        // Promotion order: byte -> short -> int -> long -> float -> double
        System.out.println("Promotion chain: byte->short->int->long->float->double");
    }
```

```java

    public static void main(String[] args) {
        arithmeticPromotion();
        ternaryWidening();
        compoundAssignment();
        charArithmetic();
        overloadingPromotion();

        System.out.println("\n=== Type Promotion Rules ===");
        System.out.println("1. byte/short/char are promoted to int for arithmetic");
        System.out.println("2. Ternary unifies to the wider type of both branches");
        System.out.println("3. Compound assignment (+=, *=) has an implicit cast");
        System.out.println("4. char is unsigned 16-bit; char arithmetic produces int");
        System.out.println("5. Overloading resolves to the smallest matching type");
    }
}
```

How does the class loading mechanism affect the static keyword?

Understanding the interaction between class loading and the `static` keyword is essential for debugging initialization-order bugs, implementing lazy singletons, and diagnosing `NoClassDefFoundError`. Java loads and initializes classes **lazily** — a class is not initialized until it is actively used for the first time. Active use includes: creating an instance, accessing a non-constant static field, invoking a static method, or using reflection. Notably, accessing a **compile-time constant** (`static final` with a literal value) does *not* trigger initialization, because the constant is inlined by the compiler.

The initialization order follows strict rules. When a class is first used, its **parent class** is initialized first (recursively up the hierarchy). Within a class, static fields and static initializer blocks execute in **textual order** — the order they appear in the source file. This means a static field declared before a static block is initialized before that block runs. The JVM guarantees that static initialization happens exactly once and is thread-safe (only one thread performs the initialization; others block until it completes).

If a static initializer throws an exception, the JVM wraps it in `ExceptionInInitializerError` on the first access. On subsequent access attempts, the class is permanently broken and throws `NoClassDefFoundError`. This is distinct from `ClassNotFoundException`, which occurs when `Class.forName()` or `ClassLoader.loadClass()` cannot find the class at all. The **initialization-on-demand holder idiom** exploits lazy class loading for thread-safe singletons: an inner static class is not loaded until the `getInstance()` method references it, providing lazy initialization without synchronization.

- **Lazy loading**: Classes initialized on first active use; parent before child.
- **Textual order**: Static fields and blocks execute in source order.
- **Compile-time constants**: Inlined at use site; do not trigger class initialization.
- **Failed initialization**: Permanently breaks the class (`NoClassDefFoundError`).
- **Holder idiom**: Inner static class for thread-safe lazy singleton without locks.

Listing 1.34: Static keyword and class loading: initialization order, failure, and the holder idiom

```java
import java.util.*;

/**
 * Demonstrates the interaction between static keyword and class loading:
 * - Class loading triggers and static initialization order
 * - Static initializer blocks and field initialization order
 * - ClassNotFoundException vs NoClassDefFoundError
 * - Lazy loading and initialization-on-demand holder idiom
 * - Circular static dependencies
 */
public class StaticClassLoading {

    // === 1. Static Initialization Order ===
    static class Parent {
        static final String PARENT_FIELD;
        static {
            System.out.println("[Parent] static initializer block");
            PARENT_FIELD = "parent-value";
        }

        Parent() {
            System.out.println("[Parent] constructor");
        }
    }

    static class Child extends Parent {
        static final String CHILD_FIELD;
        static {
            System.out.println("[Child] static initializer block");
            CHILD_FIELD = "child-value";
        }

        Child() {
            System.out.println("[Child] constructor");
        }
    }

    static void initializationOrder() {
        System.out.println("=== 1. Static Initialization Order ===");
        System.out.println("Before creating Child...");
        new Child();
        System.out.println("After creating Child.");
        // Output order: Parent static -> Child static -> Parent constructor -> Child constructor
        System.out.println();
        System.out.println("Creating second Child (statics NOT re-run)...");
        new Child();
    }

    // === 2. What Triggers Class Loading ===
    static class LazyClass {
        static {
            System.out.println("[LazyClass] loaded and initialized!");
        }
        static final int CONSTANT = 42;              // compile-time constant
        static final String RUNTIME_VALUE = UUID.randomUUID().toString(); // NOT compile-time
        static void doWork() { System.out.println("Working..."); }
    }

```

```java
    static void loadingTriggers() {
        System.out.println("\n=== 2. What Triggers Class Loading ===");

        // Accessing a compile-time constant does NOT trigger initialization
        System.out.println("Compile-time constant: " + LazyClass.CONSTANT);
        System.out.println("(LazyClass NOT initialized yet for compile-time constants)");

        // Accessing a runtime-computed static field DOES trigger initialization
        System.out.println("Runtime field: " + LazyClass.RUNTIME_VALUE);
    }

    // === 3. Initialization-on-Demand Holder (Lazy Singleton) ===
    static class ExpensiveService {
        private ExpensiveService() {
            System.out.println("[ExpensiveService] heavy initialization...");
        }

        // Inner class is NOT loaded until getInstance() is called
        private static class Holder {
            static final ExpensiveService INSTANCE = new ExpensiveService();
        }

        static ExpensiveService getInstance() {
            return Holder.INSTANCE;
        }
    }

    static void holderIdiom() {
        System.out.println("\n=== 3. Initialization-on-Demand Holder ===");
        System.out.println("Before getInstance...");
        ExpensiveService svc = ExpensiveService.getInstance();
        System.out.println("After getInstance: " + svc);
        System.out.println("Second call (same instance): "
                + (svc == ExpensiveService.getInstance()));
    }

    // === 4. ClassNotFoundException vs NoClassDefFoundError ===
    static void classLoadingErrors() {
        System.out.println("\n=== 4. Class Loading Errors ===");

        // ClassNotFoundException: explicit loading of a missing class
        try {
            Class.forName("com.example.NonExistent");
        } catch (ClassNotFoundException e) {
            System.out.println("ClassNotFoundException: " + e.getMessage());
            System.out.println("  Cause: Class.forName() or ClassLoader.loadClass() fails");
        }

        // NoClassDefFoundError: class was available at compile time but missing at runtime
        // Cannot easily demo without removing a .class file, so we explain:
        System.out.println();
        System.out.println("NoClassDefFoundError:");
        System.out.println("  Cause: class existed at compile time but is missing at runtime");
        System.out.println("  Also: static initializer threw an exception (ExceptionInInitializerError)");
        System.out.println("  Subsequent access throws NoClassDefFoundError");
    }

    // === 5. Static Initializer Failure ===
```

```java
    static class FailingInit {
        static final int VALUE;
        static {
            // If this throws, the class becomes permanently unusable
            if (System.getProperty("force.fail") != null) {
                throw new RuntimeException("Init failed!");
            }
            VALUE = 100;
        }
    }

    static void initializerFailure() {
        System.out.println("\n=== 5. Static Initializer Failure ===");

        System.out.println("When a static initializer throws an exception:");
        System.out.println("  1st access: ExceptionInInitializerError wrapping the cause");
        System.out.println("  2nd access: NoClassDefFoundError (class is permanently broken)");
        System.out.println("  The class can NEVER be used in this JVM session");
        System.out.println("  This is why static initializers should be kept simple");

        // Safe access when init succeeds
        System.out.println("FailingInit.VALUE = " + FailingInit.VALUE);
    }

    // === 6. Field Initialization Order Within a Class ===
    static class OrderDemo {
        static String a = initField("a");
        static String b;
        static {
            b = initField("b-in-block");
        }
        static String c = initField("c");

        static String initField(String name) {
            System.out.println("  Initializing: " + name);
            return name;
        }
    }

    static void fieldOrder() {
        System.out.println("\n=== 6. Static Field Initialization Order ===");
        System.out.println("Fields and static blocks execute in textual order:");
        var unused = OrderDemo.a; // triggers full initialization
    }

    public static void main(String[] args) {
        initializationOrder();
        loadingTriggers();
        holderIdiom();
        classLoadingErrors();
        initializerFailure();
        fieldOrder();

        System.out.println("\n=== Static & Class Loading Rules ===");
        System.out.println("1. Parent static init runs before child static init");
        System.out.println("2. Static init runs once, on first active use of the class");
        System.out.println("3. Compile-time constants do NOT trigger class initialization");
        System.out.println("4. Failed static init makes the class permanently unusable");
        System.out.println("5. Use Holder idiom for thread-safe lazy singletons");
```

```
    }
}
```

1.10 Coding Challenge Questions

How do you reverse a string without using `StringBuilder.reverse()`?

Reversing a string is one of the most frequently asked warm-up coding challenges in Java interviews. The key insight is that Java's `String` is immutable, so every approach must produce a new string from the characters of the original. The most efficient solution uses a **character array with a two-pointer swap**: convert the string to `char[]`, then swap characters from both ends moving inward. This runs in $O(n)$ time with $O(n)$ space for the array.

A **recursive approach** takes the last character and prepends it to the reverse of the remaining substring. While elegant, it creates $O(n)$ stack frames and $O(n^2)$ intermediate strings due to substring concatenation, making it impractical for long strings. The **stream-based approach** uses `IntStream.range()` to iterate indices in reverse and collect characters, which is concise but carries boxing and collector overhead.

A critical follow-up interviewers ask is **Unicode safety**. The `char`-based approaches break on supplementary characters (emojis, CJK extensions) because a single code point occupies two `char` values (a surrogate pair). Reversing at the `char` level splits the pair, producing invalid Unicode. The safe solution is to use `String.codePoints()` to get an `int[]` of code points, reverse that array, and reconstruct the string with `new String(int[], 0, length)`.

- **Char array swap**: $O(n)$ time, $O(n)$ space — best general approach.
- **Recursive**: Elegant but $O(n)$ stack depth and $O(n^2)$ string copies.
- **Stream**: Declarative but boxing overhead; good for readability.
- **Code points**: Required for Unicode correctness (emojis, supplementary chars).
- **Bonus variant**: Reverse words in a sentence (swap word order, not characters).

Listing 1.35: String reversal: char array swap, recursion, streams, and Unicode-safe code points

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

/**
 * Demonstrates multiple approaches to reversing a string without StringBuilder.reverse():
 * - Character array swap (in-place, O(n) time, O(n) space for char[])
 * - Recursive approach (conceptual, stack depth O(n))
 * - Stream-based approach using IntStream and codePoints
 * - Two-pointer technique for char array
 * - Handling Unicode surrogate pairs correctly
 */
public class ReverseString {

    // === 1. Char array with two-pointer swap ===
    static String reverseCharArray(String input) {
        if (input == null || input.length() <= 1) return input;
```

```java
        char[] chars = input.toCharArray();
        int left = 0, right = chars.length - 1;
        while (left < right) {
            char temp = chars[left];
            chars[left] = chars[right];
            chars[right] = temp;
            left++;
            right--;
        }
        return new String(chars);
    }

    // === 2. Recursive approach ===
    static String reverseRecursive(String input) {
        if (input == null || input.length() <= 1) return input;
        // Take last char + reverse of the rest
        return input.charAt(input.length() - 1)
                + reverseRecursive(input.substring(0, input.length() - 1));
    }

    // === 3. Stream-based with IntStream (char indices) ===
    static String reverseStream(String input) {
        if (input == null || input.isEmpty()) return input;
        return IntStream.range(0, input.length())
                .mapToObj(i -> String.valueOf(input.charAt(input.length() - 1 - i)))
                .collect(Collectors.joining());
    }

    // === 4. Unicode-safe reversal using code points ===
    // Handles surrogate pairs (emojis, supplementary characters) correctly
    static String reverseCodePoints(String input) {
        if (input == null || input.isEmpty()) return input;
        int[] codePoints = input.codePoints().toArray();
        int left = 0, right = codePoints.length - 1;
        while (left < right) {
            int temp = codePoints[left];
            codePoints[left] = codePoints[right];
            codePoints[right] = temp;
            left++;
            right--;
        }
        return new String(codePoints, 0, codePoints.length);
    }

    // === 5. Reverse words in a sentence (bonus interview variant) ===
    static String reverseWords(String sentence) {
        String[] words = sentence.trim().split("\\s+");
        int left = 0, right = words.length - 1;
        while (left < right) {
            String temp = words[left];
            words[left] = words[right];
            words[right] = temp;
            left++;
            right--;
        }
        return String.join(" ", words);
    }

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

```java
        String test = "Hello, World!";

        System.out.println("=== 1. Char Array Two-Pointer ===");
        System.out.println("Input:  " + test);
        System.out.println("Output: " + reverseCharArray(test));

        System.out.println("\n=== 2. Recursive ===");
        System.out.println("Output: " + reverseRecursive(test));

        System.out.println("\n=== 3. Stream-Based ===");
        System.out.println("Output: " + reverseStream(test));

        System.out.println("\n=== 4. Unicode-Safe (Code Points) ===");
        String unicode = "Hello \uD83D\uDE00";  // "Hello [emoji]"
        System.out.println("Input:  " + unicode);
        System.out.println("Char reverse (BROKEN):      " + reverseCharArray(unicode));
        System.out.println("CodePoint reverse (CORRECT): " + reverseCodePoints(unicode));

        System.out.println("\n=== 5. Reverse Words ===");
        String sentence = "Java is awesome";
        System.out.println("Input:  " + sentence);
        System.out.println("Output: " + reverseWords(sentence));

        // Edge cases
        System.out.println("\n=== Edge Cases ===");
        System.out.println("null:   " + reverseCharArray(null));
        System.out.println("empty:  '" + reverseCharArray("") + "'");
        System.out.println("single: " + reverseCharArray("A"));
    }
}
```

How do you check if two strings are anagrams?

Two strings are anagrams if they contain exactly the same characters with the same frequencies, regardless of order. This is a classic interview question with several approaches, and interviewers often ask for trade-off analysis between them.

The **sorting approach** converts both strings to character arrays, sorts them, and compares with `Arrays.equals()`. It is simple to implement and handles any character set, but runs in $O(n \log n)$ time due to sorting. The **frequency count approach** uses an `int[26]` array (for lowercase English letters) or a `HashMap<Character, Integer>` (for full Unicode). Increment counts for the first string and decrement for the second; if all counts are zero, the strings are anagrams. This runs in $O(n)$ time with $O(1)$ space for the array variant.

A common follow-up is **grouping anagrams** from a list of words. The standard technique is to use `Collectors.groupingBy()` with a sorted-character key: for each word, sort its characters to produce a canonical form, and group words sharing the same canonical form. This runs in $O(N \cdot k \log k)$ where N is the number of words and k is the average word length.

- **Sorting**: $O(n \log n)$ time; simple, universal.
- **Array count**: $O(n)$ time, $O(1)$ space; fastest for ASCII/lowercase.
- `HashMap` **count**: $O(n)$ time; handles full Unicode.

- **Stream**: Declarative but carries boxing overhead.
- **Group anagrams**: `groupingBy` with sorted-character canonical key.

Listing 1.36: Anagram check: sorting, frequency count, HashMap, and group-anagrams follow-up

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

/**
 * Demonstrates multiple approaches to check if two strings are anagrams:
 * - Sorting approach (simple, O(n log n))
 * - Frequency count with array (O(n), best for ASCII)
 * - Frequency count with HashMap (O(n), handles Unicode)
 * - Stream-based approach
 * - Group anagrams from a list (common follow-up)
 */
public class AnagramCheck {

    // === 1. Sorting approach: O(n log n) time, O(n) space ===
    static boolean isAnagramSort(String a, String b) {
        if (a == null || b == null) return a == b;
        if (a.length() != b.length()) return false;

        char[] aChars = a.toLowerCase().toCharArray();
        char[] bChars = b.toLowerCase().toCharArray();
        Arrays.sort(aChars);
        Arrays.sort(bChars);
        return Arrays.equals(aChars, bChars);
    }

    // === 2. Frequency count with int array: O(n) time, O(1) space (fixed 26) ===
    // Works for lowercase English letters only
    static boolean isAnagramArray(String a, String b) {
        if (a == null || b == null) return a == b;
        if (a.length() != b.length()) return false;

        int[] freq = new int[26];
        String al = a.toLowerCase(), bl = b.toLowerCase();
        for (int i = 0; i < al.length(); i++) {
            freq[al.charAt(i) - 'a']++;
            freq[bl.charAt(i) - 'a']--;
        }
        for (int count : freq) {
            if (count != 0) return false;
        }
        return true;
    }

    // === 3. HashMap frequency count: O(n), handles full Unicode ===
    static boolean isAnagramMap(String a, String b) {
        if (a == null || b == null) return a == b;
        if (a.length() != b.length()) return false;

        Map<Character, Integer> freq = new HashMap<>();
        String al = a.toLowerCase(), bl = b.toLowerCase();

        for (char c : al.toCharArray()) {
            freq.merge(c, 1, Integer::sum);
        }
```

```java
        for (char c : bl.toCharArray()) {
            freq.merge(c, -1, Integer::sum);
            if (freq.get(c) == 0) freq.remove(c);
        }
        return freq.isEmpty();
    }

    // === 4. Stream-based approach ===
    static boolean isAnagramStream(String a, String b) {
        if (a == null || b == null) return a == b;
        if (a.length() != b.length()) return false;

        return a.toLowerCase().chars().sorted().boxed().collect(Collectors.toList())
                .equals(b.toLowerCase().chars().sorted().boxed().collect(Collectors.toList()));
    }

    // === 5. Group anagrams (common follow-up interview question) ===
    static Map<String, List<String>> groupAnagrams(List<String> words) {
        return words.stream().collect(Collectors.groupingBy(word -> {
            char[] chars = word.toLowerCase().toCharArray();
            Arrays.sort(chars);
            return new String(chars);
        }));
    }

    public static void main(String[] args) {
        System.out.println("=== 1. Sorting Approach ===");
        System.out.println("listen/silent: " + isAnagramSort("listen", "silent"));
        System.out.println("hello/world:   " + isAnagramSort("hello", "world"));
        System.out.println("Anagram/nagaram: " + isAnagramSort("Anagram", "nagaram"));

        System.out.println("\n=== 2. Array Frequency Count ===");
        System.out.println("listen/silent: " + isAnagramArray("listen", "silent"));
        System.out.println("rat/car:       " + isAnagramArray("rat", "car"));

        System.out.println("\n=== 3. HashMap Frequency Count ===");
        System.out.println("listen/silent: " + isAnagramMap("listen", "silent"));
        System.out.println("diff lengths:  " + isAnagramMap("abc", "ab"));

        System.out.println("\n=== 4. Stream-Based ===");
        System.out.println("listen/silent: " + isAnagramStream("listen", "silent"));

        System.out.println("\n=== 5. Group Anagrams ===");
        List<String> words = List.of("eat", "tea", "tan", "ate", "nat", "bat");
        Map<String, List<String>> groups = groupAnagrams(words);
        groups.forEach((key, group) ->
                System.out.println("  " + key + " -> " + group));

        System.out.println("\n=== Performance Comparison ===");
        System.out.println("Sorting:   O(n log n) time, O(n) space -- simple, universal");
        System.out.println("Array:     O(n) time, O(1) space -- fastest for ASCII");
        System.out.println("HashMap:   O(n) time, O(n) space -- handles any character set");
        System.out.println("Stream:    O(n log n) time -- declarative but boxing overhead");
    }
}
```

How do you find duplicate characters in a string?

Finding duplicate characters tests a candidate's ability to choose the right data structure for frequency counting. The most common approach uses a `HashMap<Character, Integer>` (or `LinkedHashMap` to preserve insertion order) where you iterate through the string, incrementing the count for each character using `merge(c, 1, Integer::sum)`. After the pass, filter entries where the count exceeds one.

The **stream-based approach** uses `Collectors.groupingBy()` with `Collectors.counting()` as the downstream collector, then filters for counts greater than one. For a **Set-based approach** that only identifies which characters repeat (without counting), maintain a `HashSet` of seen characters and a `LinkedHashSet` of duplicates: if `seen.add(c)` returns `false`, the character is a duplicate.

For **ASCII-only input**, a `boolean[128]` array provides the fastest solution with $O(1)$ space. For **Unicode-safe** handling, use `String.codePoints()` instead of `toCharArray()` to correctly handle supplementary characters. A frequent follow-up question is finding the **first duplicate character**: iterate the string and return the first character where `seen.add(c)` returns `false`.

- `HashMap`: $O(n)$ time, $O(k)$ space; with counts; order-preserving with `LinkedHashMap`.
- **Stream**: Declarative `groupingBy + counting`; good for pipelines.
- **Set-based**: $O(n)$ time; only identifies duplicates, no counts.
- **Boolean array**: $O(n)$ time, $O(1)$ space; fastest for ASCII input.
- **Code points**: Use `codePoints()` for emojis and supplementary characters.

Listing 1.37: Finding duplicate characters: HashMap, streams, Set-based, and ASCII boolean array

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

/**
 * Demonstrates multiple approaches to find duplicate characters in a string:
 * - HashMap frequency count
 * - Stream with Collectors.groupingBy and counting
 * - IntStream / codePoints approach
 * - LinkedHashMap to preserve insertion order
 * - Bonus: find first duplicate character
 */
public class FindDuplicates {

    // === 1. HashMap frequency count ===
    static Map<Character, Integer> findDuplicatesMap(String input) {
        Map<Character, Integer> freq = new LinkedHashMap<>();
        for (char c : input.toCharArray()) {
            freq.merge(c, 1, Integer::sum);
        }
        // Filter to only duplicates
        freq.entrySet().removeIf(e -> e.getValue() < 2);
        return freq;
    }

    // === 2. Stream with groupingBy and counting ===
    static Map<Character, Long> findDuplicatesStream(String input) {
```

```java
        return input.chars()
                .mapToObj(c -> (char) c)
                .collect(Collectors.groupingBy(c -> c, LinkedHashMap::new, Collectors.counting()))
                .entrySet().stream()
                .filter(e -> e.getValue() > 1)
                .collect(Collectors.toMap(
                        Map.Entry::getKey, Map.Entry::getValue,
                        (a, b) -> a, LinkedHashMap::new));
    }

    // === 3. Using a Set to detect duplicates (no count, just which chars repeat) ===
    static Set<Character> findDuplicateCharsSet(String input) {
        Set<Character> seen = new HashSet<>();
        Set<Character> duplicates = new LinkedHashSet<>();  // preserves order
        for (char c : input.toCharArray()) {
            if (!seen.add(c)) {
                duplicates.add(c);
            }
        }
        return duplicates;
    }

    // === 4. IntStream approach for finding duplicates ===
    static Map<Integer, Long> findDuplicateCodePoints(String input) {
        return input.codePoints()
                .boxed()
                .collect(Collectors.groupingBy(cp -> cp, Collectors.counting()))
                .entrySet().stream()
                .filter(e -> e.getValue() > 1)
                .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
    }

    // === 5. Find first duplicate character ===
    static OptionalInt firstDuplicate(String input) {
        Set<Character> seen = new HashSet<>();
        for (int i = 0; i < input.length(); i++) {
            if (!seen.add(input.charAt(i))) {
                return OptionalInt.of(i);
            }
        }
        return OptionalInt.empty();
    }

    // === 6. Boolean array for ASCII-only (fastest) ===
    static Set<Character> findDuplicatesAscii(String input) {
        boolean[] seen = new boolean[128];
        Set<Character> duplicates = new LinkedHashSet<>();
        for (char c : input.toCharArray()) {
            if (c < 128) {
                if (seen[c]) duplicates.add(c);
                else seen[c] = true;
            }
        }
        return duplicates;
    }

    public static void main(String[] args) {
        String test = "programming";

```

```java
        System.out.println("=== Finding Duplicates in: \"" + test + "\" ===");

        System.out.println("\n1. HashMap (char -> count):");
        findDuplicatesMap(test).forEach((c, count) ->
                System.out.printf("   '%c' appears %d times%n", c, count));

        System.out.println("\n2. Stream groupingBy (char -> count):");
        findDuplicatesStream(test).forEach((c, count) ->
                System.out.printf("   '%c' appears %d times%n", c, count));

        System.out.println("\n3. Set-based (which chars, no count):");
        System.out.println("   Duplicates: " + findDuplicateCharsSet(test));

        System.out.println("\n4. CodePoints (Unicode-safe):");
        findDuplicateCodePoints(test).forEach((cp, count) ->
                System.out.printf("   '%c' (U+%04X) appears %d times%n",
                        (char) cp.intValue(), cp, count));

        System.out.println("\n5. First duplicate character:");
        firstDuplicate(test).ifPresentOrElse(
                idx -> System.out.printf("   First duplicate '%c' at index %d%n",
                        test.charAt(idx), idx),
                () -> System.out.println("   No duplicates found"));

        System.out.println("\n6. ASCII boolean array:");
        System.out.println("   Duplicates: " + findDuplicatesAscii(test));

        System.out.println("\n=== Approach Comparison ===");
        System.out.println("HashMap:      O(n) time, O(k) space -- with counts, order-preserving");
        System.out.println("Stream:       O(n) time -- declarative, good for pipelines");
        System.out.println("Set:          O(n) time -- only identifies which chars, no counts");
        System.out.println("CodePoints:   O(n) time -- handles emojis and supplementary chars");
        System.out.println("Boolean[]:    O(n) time, O(1) space -- fastest for ASCII-only input");
    }
}
```

What is the diamond problem with default methods and how does Java resolve it?

The **diamond problem** arises when a class implements two interfaces that both define a default method with the same signature. Without resolution rules, the compiler cannot determine which default implementation to use. Java addresses this with a clear set of priority rules that every developer should know.

Rule 1: Class always wins. If a class provides a concrete implementation of a method, it takes precedence over any interface default. This is why `Object.toString()` always wins over an interface's `default toString()`. **Rule 2: Most specific interface wins.** If interface `B` extends interface `A` and both provide a default for the same method, `B`'s version wins because it is more specific. **Rule 3: Explicit resolution required.** If two unrelated interfaces define the same default method, the implementing class *must* override it. The class can delegate to a specific interface using `InterfaceName.super.method()`.

This resolution mechanism enables powerful composition: a class can implement multiple interfaces with default methods and selectively combine their behavior. The `InterfaceName.super.method()` syntax allows calling both defaults in sequence, calling only one, or providing an entirely new implementation. This is Java's practical answer to multiple inheritance of behavior without the full complexity of C++ virtual inheritance.

- **Rule 1**: Class method always wins over interface default.
- **Rule 2**: More specific sub-interface wins over parent interface.
- **Rule 3**: If ambiguous, the class must override and resolve explicitly.
- **Super call**: `InterfaceName.super.method()` selects a specific default.
- **Practical**: Enables composing behavior from multiple interfaces safely.

Listing 1.38: Diamond problem: conflicting defaults, resolution rules, and InterfaceName.super delegation

```java
import java.util.*;

/**
 * Demonstrates the diamond problem with default methods in Java:
 * - Two interfaces with the same default method signature
 * - Compilation error without explicit resolution
 * - Resolution strategies: override, super-call, delegation
 * - Three-level diamond with interface hierarchy
 * - Practical example with logging/auditing conflict
 */
public class DiamondDefaultMethods {

    // === 1. Two interfaces with conflicting default methods ===
    interface Flyable {
        default String describe() {
            return "I can fly";
        }
        default void prepare() {
            System.out.println("Flyable: spreading wings");
        }
    }

    interface Swimmable {
        default String describe() {
            return "I can swim";
        }
        default void prepare() {
            System.out.println("Swimmable: holding breath");
        }
    }

    // This class MUST override describe() and prepare() -- compiler error otherwise
    static class Duck implements Flyable, Swimmable {
        @Override
        public String describe() {
            // Resolution strategy 1: Provide a completely new implementation
            return "I can fly AND swim";
        }

        @Override
        public void prepare() {
            // Resolution strategy 2: Explicitly delegate to a chosen super
```

```java
            Flyable.super.prepare();
            Swimmable.super.prepare();
        }
    }

    // === 2. Three-level diamond: A -> B, A -> C, D implements B & C ===
    interface Animal {
        default String sound() { return "..."; }
    }

    interface Pet extends Animal {
        @Override
        default String sound() { return "friendly noise"; }
    }

    interface WildAnimal extends Animal {
        @Override
        default String sound() { return "wild roar"; }
    }

    // Must resolve: both Pet and WildAnimal override Animal.sound()
    static class WolfDog implements Pet, WildAnimal {
        @Override
        public String sound() {
            // Pick one or combine
            return Pet.super.sound() + " / " + WildAnimal.super.sound();
        }
    }

    // === 3. No conflict when one interface is more specific ===
    interface Base {
        default String greet() { return "Hello from Base"; }
    }

    interface Extended extends Base {
        @Override
        default String greet() { return "Hello from Extended"; }
    }

    // No conflict: Extended.greet() wins (most specific interface)
    static class Greeter implements Base, Extended {
        // No override needed -- Extended is more specific than Base
    }

    // === 4. Practical example: logging conflict ===
    interface JsonLogger {
        default void log(String msg) {
            System.out.println("{\"log\": \"" + msg + "\"}");
        }
    }

    interface TextLogger {
        default void log(String msg) {
            System.out.println("[LOG] " + msg);
        }
    }

    static class DualLogger implements JsonLogger, TextLogger {
        @Override
```

```java
        public void log(String msg) {
            System.out.print("JSON: "); JsonLogger.super.log(msg);
            System.out.print("TEXT: "); TextLogger.super.log(msg);
        }
    }

    // === 5. Class method always wins over default method ===
    interface Printable {
        default String toString() {
            return "Printable default toString";
        }
    }

    static class MyClass implements Printable {
        // Object.toString() wins over interface default -- class always takes priority
        // Printable.toString() is effectively ignored
    }

    public static void main(String[] args) {
        System.out.println("=== 1. Basic Diamond: Duck ===");
        Duck duck = new Duck();
        System.out.println("describe(): " + duck.describe());
        duck.prepare();

        System.out.println("\n=== 2. Three-Level Diamond: WolfDog ===");
        WolfDog wd = new WolfDog();
        System.out.println("sound(): " + wd.sound());

        System.out.println("\n=== 3. Most-Specific Wins (No Conflict) ===");
        Greeter greeter = new Greeter();
        System.out.println(greeter.greet());

        System.out.println("\n=== 4. Practical: Dual Logger ===");
        new DualLogger().log("Application started");

        System.out.println("\n=== 5. Class Always Beats Interface Default ===");
        MyClass mc = new MyClass();
        System.out.println("toString(): " + mc.toString());
        System.out.println("(Object.toString wins, not Printable.toString)");

        System.out.println("\n=== Java Diamond Resolution Rules ===");
        System.out.println("1. Class method ALWAYS wins over interface default");
        System.out.println("2. More specific interface wins (sub-interface > super)");
        System.out.println("3. If ambiguous, implementing class MUST override");
        System.out.println("4. Use InterfaceName.super.method() to choose explicitly");
    }
}
```