🔍 What is `var` in Java?

`var` is a **reserved type name**, introduced via **JEP 286** (Java 10), allowing **local variable type inference**.
It **does not mean dynamic typing** — the type is still resolved at **compile time**.

✅ Where Can You Use `var`?

- Local variables (within methods, constructors, blocks)
- Index variables in `for` loops
- Lambda parameters (from Java 11)

❌ Where You CANNOT Use `var`

- Fields (class-level variables)
- Method parameters
- Method return types
- Standalone declarations without initialization

🚀 Syntax

```
var variableName = expression;
```

Java infers the type based on the expression.

🧠 Examples with Descriptive Names

1. **`VarSimpleStringInference.java`**

```
public class VarSimpleStringInference {
    public static void main(String[] args) {
        var message = "Hello, Java!";
        System.out.println("Message: " + message); // Type inferred as String
    }
}
```

---

2. **`VarLoopCounterExample.java`**

```
public class VarLoopCounterExample {
    public static void main(String[] args) {
        for (var i = 0; i < 5; i++) {
            System.out.println("Counter: " + i); // Type inferred as int
        }
    }
}
```

---

3. **`VarListInitialization.java`**

```
import java.util.*;

public class VarListInitialization {
    public static void main(String[] args) {
        var fruits = List.of("Apple", "Banana", "Cherry"); // Type: List<String>
        fruits.forEach(System.out::println);
    }
}
```

---

4. **`VarWithMapEntryIteration.java`**

```
import java.util.Map;

public class VarWithMapEntryIteration {
    public static void main(String[] args) {
        var capitals = Map.of("France", "Paris", "Japan", "Tokyo");
        for (var entry : capitals.entrySet()) {
            System.out.println(entry.getKey() + " → " + entry.getValue());
            // entry inferred as Map.Entry<String, String>
        }
    }
}
```

---

5. **`VarLambdaParameters.java`** ***(Java 11+)***

```
import java.util.function.Function;

public class VarLambdaParameters {
    public static void main(String[] args) {
        Function<String, String> greeter = (@SuppressWarnings("unused") var name) ->
"Hello, " + name + "!";
        System.out.println(greeter.apply("Alice"));
    }
}
```

---

6. **`VarArrayExample.java`**

```
public class VarArrayExample {
    public static void main(String[] args) {
        var numbers = new int[] {1, 2, 3, 4, 5}; // int[]
        for (var n : numbers) {
            System.out.println("Number: " + n);
        }
    }
}
```

---

7. **`VarObjectInference.java`**

```
public class VarObjectInference {
    public static void main(String[] args) {
        var object = new Object() {
            String name = "Anonymous";
            int id = 101;
        };

        System.out.println("Object Name: " + object.name);
        System.out.println("Object ID: " + object.id);
    }
}
```

Note: In this example, the inferred type is an ***anonymous class*** *— useful for temporary/local usage.*

⚠ Best Practices

✅ Use `var` when:

- The type is obvious from the right-hand side (e.g., `var user = new User();`)
- It improves readability (especially in long generic types)
- In for-loops and lambdas

❌ Avoid `var` when:

- It makes code ambiguous or unclear
- You're trying to "hide" the type (bad for maintainability)

📌 Important Notes

- `var` is **not a keyword**, so it **can still be used as a method name or variable name** (though not recommended).
- `var` cannot be assigned `null` without an explicit cast.
- `// var something = null;  // ✗ Compile-time error`
- `var something = (String) null; // ✓ Valid`

Here's a detailed breakdown of the **new String methods introduced in Java 11**, with clear explanations and **individual example programs** using **unique and meaningful names** for each case.

✓ New String Methods in Java 11

Java 11 added several **convenience methods** to the `String` class to simplify common text processing tasks.

1. `isBlank()`

Returns `true` if the string is empty or contains only whitespace characters.

Example: `CheckUserInputBlank`

```
public class CheckUserInputBlank {
    public static void main(String[] args) {
        String userInput1 = "   ";
        String userInput2 = " Hello ";

        System.out.println("Is userInput1 blank? " + userInput1.isBlank()); // true
        System.out.println("Is userInput2 blank? " + userInput2.isBlank()); // false
    }
}
```

---

2. `strip()`, `stripLeading()`, `stripTrailing()`

- `strip()` removes **leading and trailing** whitespaces (Unicode aware)
- `stripLeading()` removes leading whitespace
- `stripTrailing()` removes trailing whitespace

Example: `FormatUserAddress`

```
public class FormatUserAddress {
    public static void main(String[] args) {
        String address = "  \t123 Main Street  \n";

        System.out.println("Original: [" + address + "]");
        System.out.println("strip(): [" + address.strip() + "]");
        System.out.println("stripLeading(): [" + address.stripLeading() + "]");
        System.out.println("stripTrailing(): [" + address.stripTrailing() + "]");
    }
}
```

---

3. `lines()`

Returns a `Stream<String>` of lines in the string, split by line terminators (`\n`, `\r\n`, etc.).

Example: `ProcessMultiLineBio`

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

public class ProcessMultiLineBio {
    public static void main(String[] args) {
        String bio = "John Doe\nSoftware Engineer\nJava Enthusiast\n";

        bio.lines()
           .map(String::toUpperCase)
           .forEach(System.out::println);
    }
```

```
}
```

---

4. `repeat(int count)`

Repeats the string `count` times and returns the new concatenated result.

Example: `PrintAsciiBanner`

```
public class PrintAsciiBanner {
    public static void main(String[] args) {
        String star = "*";

        System.out.println(star.repeat(30));
        System.out.println("WELCOME TO JAVA 11".toUpperCase());
        System.out.println(star.repeat(30));
    }
}
```

Summary Table

Method	Purpose	Unicode-Aware	Return Type
`isBlank()`	Checks if string is empty or whitespace	✅	`boolean`
`strip()`	Trims whitespace from both ends	✅	`String`
`stripLeading()`	Trims leading whitespace only	✅	`String`
`stripTrailing()`	Trims trailing whitespace only	✅	`String`
`lines()`	Splits string into lines	✅	`Stream<String>`
`repeat(int)`	Repeats string N times	✅	`String`

Sure! Let's dive into **Sealed Classes** in Java 17 (JEP 409) with detailed explanation and multiple uniquely named, relevant code examples.

What Are Sealed Classes? (JEP 409 – Java 17)

Sealed classes restrict which other classes or interfaces may extend or implement them. This allows better control over inheritance and helps model more secure, predictable class hierarchies.

✅ Why Use Sealed Classes?

- Improve **domain modeling** (e.g., finite set of states or types).
- Enable **exhaustive switch statements** with `sealed` + `record`.
- Enforce **stronger compile-time checks**.

Syntax Overview

```
public sealed class Shape permits Circle, Square, Rectangle {
    // common members
}
```

- `sealed`: Marks the class as sealed.
- `permits`: Lists the allowed subclasses.
- Subclasses must be `final`, `sealed`, or `non-sealed`.

Subclass Options

Modifier	Description
`final`	No one else can extend it further.
`sealed`	Can extend further, but must explicitly permit.

Modifier	Description
`non-sealed`	Removes restrictions; open to further extension.

Code Examples

---

1. ShapeHierarchyDemo.java

Classic example: modeling geometric shapes.

```
public sealed class Shape permits Circle, Square, Triangle {}

final class Circle extends Shape {
    double radius;
}

final class Square extends Shape {
    double side;
}

final class Triangle extends Shape {
    double base, height;
}
```

✅ **Use case:** Model a finite set of known shapes. Prevent any new unexpected shape types.

---

2. ResponseTypeDemo.java

Use sealed classes for API response modeling.

```
public sealed class ApiResponse permits SuccessResponse, ErrorResponse {}

final class SuccessResponse extends ApiResponse {
    String message;
}

final class ErrorResponse extends ApiResponse {
    int code;
    String error;
}
```

✅ **Use case:** Cleanly model response types in REST APIs.

---

3. FileSystemNodeDemo.java

Model file system components with further nesting.

```
public sealed class FileSystemNode permits File, Directory {}

non-sealed class Directory extends FileSystemNode {
    List<FileSystemNode> children;
}

final class File extends FileSystemNode {
    String name;
    long size;
}
```

✅ **Use case:** `Directory` is open to further extension; `File` is terminal.

---

4. PaymentMethodDemo.java

Model different payment methods in an e-commerce system.

```
public sealed interface PaymentMethod permits CreditCard, Paypal, Crypto {}

final class CreditCard implements PaymentMethod {
    String cardNumber;
```

```
}

final class Paypal implements PaymentMethod {
    String email;
}

final class Crypto implements PaymentMethod {
    String walletAddress;
}
```

✅ **Use case:** Enforce a closed set of supported payment methods.

---

5. CommandProcessingDemo.java

Model command types in a CLI application.

```
public sealed abstract class Command permits HelpCommand, ExitCommand, RunCommand {}

final class HelpCommand extends Command {}

final class ExitCommand extends Command {}

final class RunCommand extends Command {
    String programName;
}
```

✅ **Use case:** Exhaustively handle all command types.

✅ Benefits of Sealed Classes

- **Stronger encapsulation**: Limit class hierarchy at compile-time.
- **Cleaner maintenance**: No surprise subclasses in unrelated packages.
- **Switch expression support**: Combine with `record` and `switch` for pattern matching.

🛠️ Example: Exhaustive `switch` with `sealed` + `record` (Java 21+)

```
sealed interface Result permits Success, Failure {}

record Success(String message) implements Result {}
record Failure(String error) implements Result {}

class ResultProcessor {
    public void handle(Result result) {
        switch (result) {
            case Success s -> System.out.println("Success: " + s.message());
            case Failure f -> System.out.println("Error: " + f.error());
        }
    }
}
```

📌 Limitations

- All permitted subclasses must be in the **same package** or **same module**.
- All permitted subclasses must explicitly declare their relationship.

🧾 Summary

Feature	Available in Java 17
`sealed`, `permits`	✅ Yes
Must list subclasses	✅ Yes
Interface support	✅ Yes

Feature	Available in Java 17
Combine with `record`	✅ Yes

Would you like a downloadable version (PDF/Markdown/Code files) or an example using **pattern matching with sealed classes** in Java 21?
Here's a **detailed explanation of Pattern Matching for `instanceof`** introduced and finalized in **Java 17 (JEP 394)**, along with **well-named example programs** for different use cases.

✅ What is Pattern Matching for `instanceof`?

In earlier Java versions, using `instanceof` required **manual casting** after the check:

```
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.length());
}
```

With **Pattern Matching for `instanceof`**, Java 17 allows **type test and cast in one step**:

```
if (obj instanceof String s) {
    System.out.println(s.length()); // 's' is in scope here
}
```

🎯 Benefits:

- Reduces boilerplate
- Safer and cleaner syntax
- Improves readability, especially in complex conditional logic

---

🔍 Feature Rules & Scope

1. Variable (`s` in `String s`) is only in scope **within the true branch** of the conditional.
2. Pattern variables can **shadow outer variables** (though not recommended).
3. Can be combined with other logical operations (`&&`, `||`, etc.).
4. Smart casting works in nested and inverted logic too.

✅ Example Programs

---

1. `SimpleTypeCheck.java`

Basic use case of pattern matching

```
public class SimpleTypeCheck {
    public static void main(String[] args) {
        Object obj = "Hello Java 17";

        if (obj instanceof String s) {
            System.out.println("Length: " + s.length());
        } else {
            System.out.println("Not a string");
        }
    }
}
```

---

2. `SmartCastingWithIfElse.java`

Demonstrates automatic cast inside `if-else`

```
public class SmartCastingWithIfElse {
    public static void printLength(Object input) {
        if (input instanceof String s) {
            System.out.println("String length: " + s.length());
        } else {
            System.out.println("Not a string: " + input);
```

```
        }
    }

    public static void main(String[] args) {
        printLength("Pattern Matching");
        printLength(42);
    }
}
```

---

3. `PatternMatchingInComplexCondition.java`

Combines with logical operations

```
public class PatternMatchingInComplexCondition {
    public static void main(String[] args) {
        Object obj = "Java";

        if (obj instanceof String s && s.length() > 3) {
            System.out.println("A long enough string: " + s.toUpperCase());
        }
    }
}
```

---

4. `MultipleInstanceofChecks.java`

Handles multiple types with pattern matching

```
public class MultipleInstanceofChecks {
    public static void describe(Object o) {
        if (o instanceof String s) {
            System.out.println("It's a String of length: " + s.length());
        } else if (o instanceof Integer i) {
            System.out.println("It's an Integer: " + (i * 2));
        } else if (o instanceof Double d) {
            System.out.println("It's a Double: " + Math.round(d));
        } else {
            System.out.println("Unknown type");
        }
    }

    public static void main(String[] args) {
        describe("ChatGPT");
        describe(10);
        describe(3.1415);
    }
}
```

---

5. `NestedPatternMatchingExample.java`

Uses pattern matching inside nested blocks

```
public class NestedPatternMatchingExample {
    public static void main(String[] args) {
        Object obj = "Nested Test";

        if (obj != null) {
            if (obj instanceof String s) {
                System.out.println("Nested pattern matched: " + s.toLowerCase());
            }
        }
    }
}
```

---

6. `ShadowingVariableExample.java`

Demonstrates shadowing (not recommended, but allowed)

```
public class ShadowingVariableExample {
```

```
    public static void main(String[] args) {
        String s = "Outer";
        Object obj = "Inner";

        if (obj instanceof String s1) {
            System.out.println("Inner pattern variable: " + s1);
        }

        System.out.println("Outer variable: " + s);
    }
}
```

⚠ Things to Note

- This pattern **does not allow reusing the same variable name** in the same scope (`obj instanceof String obj` → ✗).
- `instanceof` pattern **is not supported for primitive types**, as they are not instances of any class.
- It is different from full **Pattern Matching for `switch`**, which was finalized in Java 21.

✓ Final Thoughts

Java 17's pattern matching for `instanceof` is a powerful but simple enhancement that:

- Makes code more readable
- Reduces error-prone casting
- Plays well with new features like `sealed`, `records`, and `switch` patterns

Sure! Here's a detailed explanation of **Record Patterns (JEP 440)** in Java 21, along with multiple example programs with unique names to showcase different uses.

Record Patterns (JEP 440) in Java 21 — Detailed Explanation

What are Record Patterns?

Record patterns allow you to deconstruct record values directly in pattern matching expressions, like `instanceof` checks and `switch` statements. Since Java 16 introduced *records* as a compact data carrier class, JEP 440 extends pattern matching to destructure these records in a natural, concise way.

Why Record Patterns?

When you want to extract the components of a record inside an `if` or `switch`, before JEP 440, you'd manually access each component after type checking:

```
if (obj instanceof Point p) {
    int x = p.x();
    int y = p.y();
    // use x, y
}
```

Record patterns simplify this by allowing you to extract components **directly** in the pattern match:

```
if (obj instanceof Point(int x, int y)) {
    // x and y are extracted automatically
}
```

Syntax

```
obj instanceof RecordType(var1, var2, ...)
```

- The components `var1, var2` correspond to the record's components.
- You can use variable names or nested patterns here.

Benefits

- Reduces boilerplate
- Improves readability

- Aligns with Java's move toward pattern matching and data-oriented programming

---

Example Record Definition to Use

```
public record Point(int x, int y) {}
public record Person(String name, int age) {}
public record Rectangle(Point topLeft, Point bottomRight) {}
```

Example Programs with Unique Names

---

1. PointAnalyzer.java

```
public class PointAnalyzer {
    public static String analyze(Object obj) {
        if (obj instanceof Point(int x, int y)) {
            if (x == y) {
                return "Point lies on the line x = y";
            } else {
                return "Point at (" + x + ", " + y + ")";
            }
        }
        return "Not a Point";
    }

    public static void main(String[] args) {
        System.out.println(analyze(new Point(5, 5)));  // Point lies on the line x = y
        System.out.println(analyze(new Point(3, 7)));  // Point at (3, 7)
        System.out.println(analyze("Hello"));          // Not a Point
    }
}
```

---

2. PersonAgeClassifier.java

```
public class PersonAgeClassifier {
    public static String classify(Object obj) {
        if (obj instanceof Person(String name, int age)) {
            if (age < 18) return name + " is a minor.";
            else return name + " is an adult.";
        }
        return "Not a Person";
    }

    public static void main(String[] args) {
        System.out.println(classify(new Person("Alice", 17)));  // Alice is a minor.
        System.out.println(classify(new Person("Bob", 25)));    // Bob is an adult.
    }
}
```

---

3. RectangleInspector.java

```
public class RectangleInspector {
    public static String describe(Object obj) {
        if (obj instanceof Rectangle(Point(int x1, int y1), Point(int x2, int y2))) {
            int width = Math.abs(x2 - x1);
            int height = Math.abs(y2 - y1);
            return "Rectangle width: " + width + ", height: " + height;
        }
        return "Not a Rectangle";
    }

    public static void main(String[] args) {
        Point p1 = new Point(1, 5);
```

```
        Point p2 = new Point(6, 2);
        System.out.println(describe(new Rectangle(p1, p2)));  // Rectangle width: 5,
height: 3
    }
}
```

---

4. ShapeSwitch.java

This example uses record patterns inside a `switch` expression:

```
public class ShapeSwitch {
    public static String identifyShape(Object shape) {
        return switch (shape) {
            case Point(int x, int y) -> "Point at (" + x + ", " + y + ")";
            case Rectangle(Point(int x1, int y1), Point(int x2, int y2)) ->
                "Rectangle with corners (" + x1 + "," + y1 + ") and (" + x2 + "," + y2 +
")";
            default -> "Unknown shape";
        };
    }

    public static void main(String[] args) {
        System.out.println(identifyShape(new Point(0, 0)));  // Point at (0, 0)
        System.out.println(identifyShape(new Rectangle(new Point(1, 2), new Point(3,
4))));
        // Rectangle with corners (1,2) and (3,4)
    }
}
```

---

5. NestedRecordMatcher.java

You can nest record patterns deeply:

```
public class NestedRecordMatcher {
    public static String checkNested(Object obj) {
        if (obj instanceof Rectangle(Point(int x1, int y1), Point(int x2, int y2))) {
            if (x1 == y1 && x2 == y2) {
                return "Both corners lie on the line x = y";
            }
            return "Corners are at (" + x1 + "," + y1 + ") and (" + x2 + "," + y2 + ")";
        }
        return "Not a Rectangle";
    }

    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(new Point(2, 2), new Point(5, 5));
        Rectangle r2 = new Rectangle(new Point(1, 2), new Point(3, 4));
        System.out.println(checkNested(r1));  // Both corners lie on the line x = y
        System.out.println(checkNested(r2));  // Corners are at (1,2) and (3,4)
    }
}
```

Summary

- **Record Patterns** allow concise destructuring of records in `instanceof` and `switch`.
- They reduce boilerplate by extracting components directly in the pattern.
- Support nested destructuring for records inside other records.
- Widely useful for domain modeling, clean code, and data-driven logic.

Sure! Let's deep dive into the evolution and usage of the **`switch` statement** in Java **up to Java 21**, with detailed explanations and multiple unique example programs.

Java `switch` Statement: Evolution up to Java 21

The `switch` statement is a control flow construct that allows branching execution based on the value of an expression. Over the years, it has evolved significantly.

---

1. Classic `switch` (Java 1.0 to Java 12)

- Works only with `int`, `char`, `byte`, `short`, their wrappers, and `enum` types.
- Uses `case` labels with constant values.
- Requires explicit `break` to prevent fall-through.
- No return value.

Example: ClassicSwitchDemo

```
public class ClassicSwitchDemo {
    public static void main(String[] args) {
        int day = 3;
        String dayName;
        switch (day) {
            case 1: dayName = "Monday"; break;
            case 2: dayName = "Tuesday"; break;
            case 3: dayName = "Wednesday"; break;
            case 4: dayName = "Thursday"; break;
            case 5: dayName = "Friday"; break;
            default: dayName = "Weekend"; break;
        }
        System.out.println("Day: " + dayName);
    }
}
```

---

2. Enhanced `switch` (Preview in Java 12-13, Finalized in Java 14)

Introduced **expression-style switch** and **arrow labels** `->`, making switch more concise and powerful.

- Switch can return a value.
- No need for explicit `break` with `->` syntax.
- Multiple labels per case supported.
- Block of code allowed with `{}`.

Example: EnhancedSwitchExpressionDemo

```
public class EnhancedSwitchExpressionDemo {
    public static void main(String[] args) {
        int day = 2;
        String dayType = switch (day) {
            case 1, 2, 3, 4, 5 -> "Weekday";
            case 6, 7 -> "Weekend";
            default -> "Invalid day";
        };
        System.out.println("Day type: " + dayType);
    }
}
```

---

3. Pattern Matching for `switch` (Preview in Java 17 and finalized in Java 21)

Allows using **type patterns** inside switch cases for more expressive and safer type handling.

- `case` can match on the **type** of an object, not just values.
- Combined with **record patterns** (Java 21) for destructuring.

Example: PatternMatchingSwitchDemo (Java 21)

```
public class PatternMatchingSwitchDemo {
```

```
    public static void main(Object obj) {
        String result = switch (obj) {
            case Integer i -> "Integer: " + i;
            case String s -> "String: " + s;
            case null -> "Null value";
            default -> "Unknown type";
        };
        System.out.println(result);
    }

    public static void main(String[] args) {
        main(123);
        main("Hello");
        main(3.14);
        main(null);
    }
}
```

---

4. Record Patterns in `switch` (Java 21)

Records are concise immutable data carriers. Pattern matching on records inside a switch allows unpacking fields directly.

Example: RecordPatternSwitchDemo

```
record Point(int x, int y) {}

public class RecordPatternSwitchDemo {
    public static String describe(Object obj) {
        return switch (obj) {
            case Point(int x, int y) when x == y -> "Point on diagonal: " + x + "," + y;
            case Point p -> "Point at: " + p.x() + "," + p.y();
            default -> "Not a point";
        };
    }

    public static void main(String[] args) {
        System.out.println(describe(new Point(5, 5)));
        System.out.println(describe(new Point(3, 7)));
        System.out.println(describe("Hello"));
    }
}
```

---

5. Summary of `switch` features by Java version:

Java Version	Switch Feature	Description
<= Java 11	Classic switch	Value-based, no expression support
Java 12-14	Enhanced switch expressions	Expression style, arrow labels, multiple labels
Java 17	Pattern matching preview	Type-based case labels
Java 21	Pattern matching finalized + record patterns	Destructuring records in switch cases

---

Bonus: Fall-through with classic switch

Classic switch requires `break` to avoid fall-through:

```
public class FallThroughDemo {
    public static void main(String[] args) {
        int number = 2;
        switch (number) {
            case 1:
            case 2:
                System.out.println("One or Two");
```

```
                break;
            case 3:
                System.out.println("Three");
                break;
            default:
                System.out.println("Other");
        }
    }
}
```

Sure! Let's dive into **String Templates in Java 21** with detailed explanation and example programs with unique names.

String Templates in Java 21 (JEP 430) — Detailed Explanation

What are String Templates?

String Templates provide a **concise, readable, and safe way to embed expressions directly inside string literals**. Unlike traditional string concatenation or `String.format`, string templates allow you to embed expressions without breaking the string syntax, making your code cleaner and less error-prone.

Why String Templates?

- Avoids boilerplate code and concatenations like `"Hello " + name + ", you have " + count + " messages."`
- Safer than manual concatenation, less risk of mistakes.
- More readable and maintainable.
- Supports expressions inside the template, not just variables.

Basic Syntax

String templates use **`${}`** to embed expressions inside a string literal. The string literal uses backticks `` ` `` instead of double quotes `"`.

Example:

```
var name = "Alice";
var count = 5;
var message = STR.`Hello ${name}, you have ${count} new messages.`;
System.out.println(message);
```

- `STR` is the string template processor (can vary based on context or API).
- Expressions inside `${}` are evaluated and their result is embedded.

Key Points

- Expressions inside `${}` can be any valid Java expression.
- Supports multi-line strings.
- Allows formatting, escaping, and customization.

Example Programs

---

1. WelcomeMessageGenerator

```
public class WelcomeMessageGenerator {
    public static void main(String[] args) {
        var user = "John";
        var unread = 7;
```

```
        var message = STR.`Welcome ${user}! You have ${unread} unread notifications.`;
        System.out.println(message);
    }
}
```

Output:

```
Welcome John! You have 7 unread notifications.
```

---

2. InvoicePrinter

```
public class InvoicePrinter {
    public static void main(String[] args) {
        var item = "Laptop";
        var price = 1200.50;
        var quantity = 2;
        var total = price * quantity;

        var invoice = STR.`
            Invoice:
            Item: ${item}
            Price: $${price}
            Quantity: ${quantity}
            Total: $${total}
        `;

        System.out.println(invoice);
    }
}
```

Output:

```
Invoice:
Item: Laptop
Price: $1200.5
Quantity: 2
Total: $2401.0
```

---

3. ExpressionEvaluationDemo

```
public class ExpressionEvaluationDemo {
    public static void main(String[] args) {
        var a = 10;
        var b = 20;
        var result = STR.`The sum of ${a} and ${b} is ${a + b}, and their product is ${a
* b}.`;
        System.out.println(result);
    }
}
```

Output:

```
The sum of 10 and 20 is 30, and their product is 200.
```

---

4. MultiLineTemplateExample

```
public class MultiLineTemplateExample {
    public static void main(String[] args) {
        var title = "Java 21 Features";
        var year = 2025;
        var text = STR.`
            Report:
            Title: ${title}
            Year: ${year}

            Highlights:
            - String Templates
            - Virtual Threads
            - Pattern Matching
        `;
```

```
        System.out.println(text);
    }
}
```

Output:

```
Report:
Title: Java 21 Features
Year: 2025

Highlights:
- String Templates
- Virtual Threads
- Pattern Matching
```

---

5. EscapeSequencesDemo

```
public class EscapeSequencesDemo {
    public static void main(String[] args) {
        var path = "C:\\Users\\Admin";
        var message = STR.`The file path is: ${path}`;
        System.out.println(message);
    }
}
```

Output:

```
The file path is: C:\Users\Admin
```

Summary

- Java 21 String Templates let you embed Java expressions inside strings using backtick literals and `${}`.
- Improves readability and reduces errors in string construction.
- Supports multi-line templates, expressions, and escapes.
- A modern replacement for concatenation and `String.format`.

In Java, both `trim()` and `strip()` are methods used to remove whitespace from strings, but they differ in how they define and handle whitespace characters.

`trim()`

- **Removes:** Leading and trailing characters with Unicode value ≤ U+0020 (i.e., basic ASCII whitespace like space `' '`, tab `'\t'`, newline `'\n'`, carriage return `'\r'`).
- **Returns:** A new string with those characters removed.
- **Java Version:** Available since early Java versions (Java 1.0).
- **Limitation:** Does **not** handle all Unicode whitespace characters.

Example:

```
String s = "\u2001 hello \t";
System.out.println("[" + s.trim() + "]");  // Output: [hello]
```

`\u2001` is a Unicode whitespace character (EM QUAD) — **not removed** by `trim()`.

`strip()`

- **Removes:** All leading and trailing **Unicode whitespace** characters, including those not covered by `trim()`.
- **Returns:** A new string with **all Unicode-defined whitespace** removed from both ends.
- **Java Version:** Introduced in **Java 11**.
- **More comprehensive** than `trim()`.

Example:

```
String s = "\u2001 hello \t";
System.out.println("[" + s.strip() + "]");  // Output: [hello]
```

`\u2001` **is removed** by `strip()`.

Summary Table

Feature	`trim()`	`strip()`
Removes	ASCII whitespace (`<= U+0020`)	All Unicode whitespace
Unicode aware	✗	✓
Java version	1.0	11

Recommendation

- Use `trim()` for simple, legacy-safe cleanup.
- Use `strip()` if you're working in Java 11+ and want full Unicode whitespace support.

In Java, these two declarations:

```
List list = new ArrayList();
```

vs

```
ArrayList list1 = new ArrayList();
```

are both valid, but they have **different implications** in terms of **flexibility, abstraction, and usage**. Here's a breakdown:

✓ `List list = new ArrayList();` — Preferred in most cases

Key Points:

- **List** is an **interface**, and `ArrayList` is one of its many **implementations**.
- You're programming to an **interface**, which is a common best practice in Java (design principle: *"Program to an interface, not an implementation."*)
- This makes your code more **flexible**. For example, you can later change the implementation to `LinkedList`, `Vector`, etc., without changing the rest of your code.

Example:

```
List<String> names = new ArrayList<>();
names = new LinkedList<>();  // easily changeable
```

🔍 `ArrayList list1 = new ArrayList();` — More specific

Key Points:

- You're tying your code to a **specific implementation** (`ArrayList`).
- Slightly less flexible, as you can't easily switch to another implementation without refactoring.
- However, you get access to `ArrayList`-specific methods (though rare and generally not recommended to rely on).

💡 Summary Table

Aspect	`List list = new ArrayList();`	`ArrayList list1 = new ArrayList();`
Type	Interface (`List`)	Concrete class (`ArrayList`)
Flexibility	✓ High (can switch implementation)	✗ Low
Best Practice	✓ Recommended	✗ Not recommended for general use
Access to `ArrayList`-specific methods	✗ No	✓ Yes

✓ Best Practice (with Generics):

Always use generics to avoid type-safety issues:

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

In Java (and object-oriented programming in general), **generalization** and **specialization** refer to two **opposite** concepts in class hierarchy design—how classes are related and how behavior and attributes are abstracted or made more specific.

◆ Generalization

Definition:
Generalization is the process of extracting shared characteristics (fields or methods) from two or more classes and combining them into a generalized superclass.

Key Traits:

- **"Is-a" relationship** (e.g., `Dog is a Animal`)
- Promotes **code reuse** and **abstraction**
- Moves from **specific → general**

Example:

```
class Animal {
    void makeSound() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Bark");
    }
}
```

Here, `Animal` is a **generalized** class. It holds behavior common to all animals. `Dog` inherits this behavior and may specialize it.

◈ Specialization

Definition:
Specialization is the process of creating a subclass from a more general class by adding more specific behavior or attributes.

Key Traits:

- Subclass **extends** a superclass.
- Adds or overrides functionality.
- Moves from **general → specific**

Example (continued):

```
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof");
    }
}
```

Here, `Dog` is a **specialized** version of `Animal`, with its own behavior (`makeSound` is overridden to be more specific).

VS Comparison Table

Feature	Generalization	Specialization
Direction	Specific → General	General → Specific
Purpose	Code reuse, abstraction	Extend functionality, customization
Result	Superclass creation	Subclass creation
Example	Create `Animal` from `Dog` and `Cat`	Create `Dog` from `Animal`

Feature	Generalization	Specialization
Usage	`Dog extends Animal` (focus on Animal)	`Dog extends Animal` (focus on Dog)

Visual Representation

```
Generalization:
Dog, Cat → Animal

Specialization:
Animal → Dog, Cat
```

Real-world Analogy

- **Generalization:** "Car" is a generalization of "Sedan" and "SUV".
- **Specialization:** "Sedan" is a specialization of "Car".

Yes, **`Thread.suspend()`** and **`Thread.resume()`** are **deprecated** in Java.

⊘ Deprecated Methods

Method	Deprecated Since
`Thread.suspend()`	Java 1.2
`Thread.resume()`	Java 1.2

⚠ Why Were They Deprecated?

They were deprecated because they are **unsafe** and can lead to **serious concurrency issues**, particularly **deadlocks**.

⚠ Problems:

- **`suspend()`** can freeze a thread **while it's holding a lock**.
- If another thread needs that lock to proceed, you get a **deadlock**.
- **`resume()`** might never be called if the thread that was supposed to resume it is waiting on the same lock.

✓ Modern Alternatives

To safely **pause and resume threads**, use higher-level concurrency controls like:

1. `wait()` / `notify()`

Good for coordinated thread communication, requires synchronization:

```
class SharedObject {
    private boolean paused = false;

    synchronized void pause() throws InterruptedException {
        paused = true;
    }

    synchronized void resume() {
        paused = false;
        notify();
    }

    synchronized void doWork() throws InterruptedException {
        while (paused) {
            wait();
        }
        // work logic here
    }
}
```

2. `Lock` and `Condition` (from `java.util.concurrent.locks`)

More flexible and safer than `wait/notify`.

```
Lock lock = new ReentrantLock();
Condition condition = lock.newCondition();
boolean paused = false;

void pause() {
    lock.lock();
    try {
        paused = true;
    } finally {
        lock.unlock();
    }
}

void resume() {
    lock.lock();
    try {
        paused = false;
        condition.signalAll();
    } finally {
        lock.unlock();
    }
}
```

3. Using `volatile` flags or semaphores

Example with a `volatile` flag:

```
class MyRunnable implements Runnable {
    private volatile boolean paused = false;

    public void run() {
        while (true) {
            if (!paused) {
                // perform task
            }
        }
    }

    public void pause() {
        paused = true;
    }

    public void resume() {
        paused = false;
    }
}
```

✅ Summary

Method	Status	Use Instead
`Thread.suspend()`	⊘ Deprecated	`wait()`, `Lock`, flags
`Thread.resume()`	⊘ Deprecated	`notify()`, `Condition`

✅ `notify()` in Java – Explained

In Java, `notify()` is a method used in **multithreading** to **wake up one thread** that is waiting on the same object's **monitor** (lock). It is part of the **`Object` class**, meaning every Java object inherits it.

🔁 Context: `wait()` / `notify()` / `notifyAll()`

- `wait()`: Causes the current thread to **wait** (pause execution) until another thread calls `notify()` or `notifyAll()` on the same object.
- `notify()`: **Wakes up one** thread that's waiting on that object.
- `notifyAll()`: Wakes up **all** waiting threads.

⚠ Important Conditions:

1. Must be used in a **synchronized** block or method.
2. The thread must **own the object's monitor lock**.
3. Failing to follow this will result in a `IllegalMonitorStateException`.

🛠 Example:

```
class SharedResource {
    synchronized void waitForSignal() {
        try {
            System.out.println(Thread.currentThread().getName() + " waiting...");
            wait();  // releases the lock and waits
            System.out.println(Thread.currentThread().getName() + " resumed!");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    synchronized void sendSignal() {
        System.out.println(Thread.currentThread().getName() + " sending signal...");
        notify();  // wakes up one waiting thread
    }
}

public class NotifyExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        Thread t1 = new Thread(() -> resource.waitForSignal(), "Thread-1");
        Thread t2 = new Thread(() -> resource.waitForSignal(), "Thread-2");

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

        try {
            Thread.sleep(1000); // let threads start and wait
        } catch (InterruptedException e) {}

        Thread t3 = new Thread(() -> resource.sendSignal(), "Notifier");
        t3.start();
    }
}
```

□ What Happens:

1. `Thread-1` and `Thread-2` both call `wait()` and **suspend** inside `waitForSignal()`.
2. `Notifier` thread calls `notify()` and **wakes up one** of the waiting threads (not both).
3. The awakened thread continues after `wait()`.
   Note: You can't control **which** thread gets notified — it's chosen arbitrarily by the JVM.

🛠 Use Case: Producer-Consumer

`wait()` and `notify()` are often used in **producer-consumer** scenarios where one thread produces data and another consumes it.

⚠ Common Mistakes to Avoid:

- Calling `notify()` **outside** a synchronized block → ✗ Throws `IllegalMonitorStateException`
- Not using `wait()` in a **loop** → Can lead to **spurious wakeups** (use `while` instead of `if`)
- `while (!condition) {`
- `    wait();`
- `}`

✅ Summary

Method	Purpose	Requires lock?	Wakes up
`wait()`	Causes thread to wait	✅ Yes	—
`notify()`	Wakes up **one** waiting thread	✅ Yes	1 thread
`notifyAll()`	Wakes up **all** waiting threads	✅ Yes	All threads

In Java, **exception handling with `try`, `catch`, `throw`, and `finally`** has specific behavior, especially when it comes to:

- Throwing exceptions
- Execution of `finally` blocks
- Behavior of `System.out.println()` after an exception

🔥 Key Concepts Recap:

```
try {
   // code that may throw an exception
} catch (ExceptionType e) {
   // handle exception
} finally {
   // cleanup code (always executes, even if exception occurs)
}
```

✅ ALL POSSIBLE CASES with `throw`, `catch`, `finally`, and `System.out.println`

◆ Case 1: Exception occurs, caught, and `finally` executes

```
try {
    System.out.println("Before exception");
    throw new RuntimeException("Error!");
} catch (RuntimeException e) {
    System.out.println("Caught: " + e.getMessage());
} finally {
    System.out.println("Finally block executed");
}
System.out.println("After try-catch-finally");
```

✅ Output:

```
Before exception
Caught: Error!
Finally block executed
After try-catch-finally
```

✅ **Explanation:** Exception is thrown → caught → finally runs → program continues

---

◆ Case 2: Exception occurs, not caught, but `finally` still executes

```
try {
    System.out.println("Before exception");
    throw new Exception("Uncaught Error");
} catch (RuntimeException e) {
```

```
    System.out.println("Caught: " + e.getMessage());
} finally {
    System.out.println("Finally block executed");
}
System.out.println("This will NOT execute"); // unreachable
```

✗ Output:

```
Before exception
Finally block executed
Exception in thread "main" java.lang.Exception: Uncaught Error
    at ...
```

✗ Explanation: `Exception` is **not** caught (no `catch(Exception)`), so it propagates → `finally` still runs → program exits → remaining code not executed

---

◆ Case 3: No exception occurs → `finally` still executes

```
try {
    System.out.println("No error here");
} catch (Exception e) {
    System.out.println("Won't be printed");
} finally {
    System.out.println("Finally always runs");
}
System.out.println("Continues normally");
```

✓ Output:

```
No error here
Finally always runs
Continues normally
```

◆ Case 4: Exception thrown inside `catch` → `finally` still executes

```
try {
    throw new RuntimeException("Initial error");
} catch (RuntimeException e) {
    System.out.println("Caught: " + e.getMessage());
    throw new IllegalArgumentException("New error from catch");
} finally {
    System.out.println("Finally after catch");
}
```

✗ Output:

```
Caught: Initial error
Finally after catch
Exception in thread "main" java.lang.IllegalArgumentException: New error from catch
    at ...
```

✗ Explanation: Even though new exception is thrown in `catch`, `finally` still executes → exception propagates

---

◆ Case 5: Exception thrown inside `finally` → overrides previous exception

```
try {
    throw new RuntimeException("Try error");
} catch (Exception e) {
    System.out.println("Caught: " + e.getMessage());
    throw new IllegalArgumentException("Catch error");
} finally {
    System.out.println("In finally");
    throw new NullPointerException("Finally error");
}
```

✗ Output:

```
Caught: Try error
In finally
Exception in thread "main" java.lang.NullPointerException: Finally error
```

```
    at ...
```

⚠ **Explanation:**

- The **exception in `finally`** (`NullPointerException`) **suppresses** the `catch`'s `IllegalArgumentException`.
- Java gives priority to the `finally`-thrown exception.

◆ Case 6: `System.exit()` inside `try` — `finally` does not execute

```
try {
    System.out.println("About to exit");
    System.exit(0);
} finally {
    System.out.println("This won't run");
}
```

✗ **Output:**

```
About to exit
```

✗ **Explanation:** JVM terminates immediately. `finally` is **skipped**.

Summary Table

Scenario	Finally executes?	Program continues?	Notes
Exception caught	✓ Yes	✓ Yes	Normal flow
Exception not caught	✓ Yes	✗ No	Finally runs, then crash
No exception	✓ Yes	✓ Yes	Normal
Exception thrown in catch	✓ Yes	✗ No (if unhandled)	Finally runs
Exception thrown in finally	✓ Yes	✗ No	Suppresses other exceptions
`System.exit(0)` called in try	✗ No	✗ No	JVM shuts down

Great question! Let's break it down clearly.

✓ Why Do We Need a Constructor in an Abstract Class in Java?

Even though **abstract classes cannot be instantiated directly**, their **constructors are still important**. Here's why:

◆ 1. For Initialization of Common State (Fields)

Abstract classes can have fields (variables) that are shared across subclasses. The constructor in the abstract class initializes those fields.

Example:

```
abstract class Animal {
    String name;

    Animal(String name) {
        this.name = name;
        System.out.println("Animal constructor called");
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name); // calls Animal constructor
        System.out.println("Dog constructor called");
    }
}
```

Output:

```
Animal constructor called
Dog constructor called
```

✓ **Explanation:** Even though we instantiate `Dog`, the `Animal` constructor runs first, ensuring `name` is initialized.

◆ 2. To Enforce Subclass Initialization Requirements

An abstract class can define a constructor that **requires arguments**, forcing subclasses to supply them via `super()`.

Example:

```
abstract class Vehicle {
    int wheels;

    Vehicle(int wheels) {
        this.wheels = wheels;
    }
}

class Bike extends Vehicle {
    Bike() {
        super(2);  // must call parent constructor
    }
}
```

✓ This ensures all `Vehicle`s have their `wheels` initialized.

◆ 3. For Code Reuse and Maintenance

If multiple subclasses share common setup code, placing that in the abstract class constructor avoids duplication.

◆ 4. Constructors Get Called in the Inheritance Chain

Even though **you can't directly instantiate an abstract class**, its **constructor is always called** when a subclass is instantiated.
Java always starts constructor execution **from the top of the inheritance chain down**.

! What Happens Without a Constructor?

If you don't define a constructor in the abstract class, Java provides a **default no-argument constructor**. But if the abstract class has a **parameterized constructor**, you **must call `super(args)`** from the subclass.

□ Summary

Why Constructor in Abstract Class?	Purpose
Initialize common fields	Shared state for all subclasses
Enforce required arguments via `super()`	Forces subclass to provide data
Enable constructor chaining	Ensures proper object construction
Avoid duplication	Clean, reusable code

Great question — let's compare and explain:

✓ `Object.hashCode()` vs `String.hashCode()` vs Custom Class `hashCode()` in Java

These are all implementations of the same method:

```
public int hashCode()
```

But they behave **differently** based on where they are defined or overridden.

---

1. 🔷 `Object.hashCode()`

📌 Default Implementation

- Provided by the root `Object` class.
- Returns a hash code based on the **memory address** of the object (typically).
- Not useful for value comparison.

🔧 Example:

```
Object obj1 = new Object();
Object obj2 = new Object();
System.out.println(obj1.hashCode()); // e.g., 123456
System.out.println(obj2.hashCode()); // different, even if both empty
```

✓ Good for object identity, ✗ bad for value-based comparison.

---

2. 🔷 `String.hashCode()`

📌 Overridden by `String` class

- Uses the **contents of the string** to compute hash.
- Implements this formula:

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

Where `s[i]` is the i-th character of the string.

🔧 Example:

```
String s1 = "hello";
String s2 = "hello";
System.out.println(s1.hashCode()); // Always the same (99162322)
System.out.println(s2.hashCode()); // Same, even for different objects
```

✓ Content-based, consistent, ideal for using `String` as a key in `HashMap`.

---

3. 🔷 Custom Class `hashCode()`

📌 You override `hashCode()` in your class to define how two logically equal objects should produce the same hash code.

This is **crucial** when:

- Using custom objects in hash-based collections (`HashMap`, `HashSet`)
- You're also overriding `equals()`

🔧 Example:

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

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

    @Override
    public int hashCode() {
        return Objects.hash(name, age); // Java 7+
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Person other = (Person) obj;
        return age == other.age && name.equals(other.name);
```

```
    }
}
```

✅ Good Practice:

- Always override `hashCode()` when you override `equals()`.
- Make sure that **equal objects** return the **same hash code**.

🔄 Comparison Table

Feature	`Object.hashCode()`	`String.hashCode()`	Custom `hashCode()`
Based on	Object identity (memory addr)	Characters of the string	Your field logic
Deterministic?	No (may vary between runs)	Yes (same for same string always)	Yes (if implemented properly)
Equal objects same hash?	❌ No	✅ Yes	✅ Yes (you define it)
Used in `HashMap`	Only for identity	Ideal key	Works if `equals()` and `hashCode()` match

---

🚫 Common Mistakes

- Overriding `equals()` but **not** `hashCode()` → breaks `HashMap`, `HashSet`.
- Including **mutable fields** in `hashCode()` → hash code may change after adding to a map → breaks lookup.
- Using `==` instead of `equals()` inside `equals()` method.

🧠 Final Tip

Always follow the contract:
If `a.equals(b)` is true, then `a.hashCode() == b.hashCode()` must be true.
But:
The reverse is **not** necessarily true. Two unequal objects may still have the same hash code (hash collision).

Excellent question! Let's explore the **differences between `Object.equals()`, `String.equals()`, and custom `equals()`** method implementations in Java.

✅ `Object.equals()` vs `String.equals()` vs Custom `equals()` in Java

All three are about **comparing objects**, but they work differently depending on how they're implemented.

🔹 1. `Object.equals()`

📌 Default implementation in `java.lang.Object`

- Compares **reference equality** (`this == obj`)
- Returns `true` **only if both references point to the same object in memory**

🛠️ Example:

```
Object o1 = new Object();
Object o2 = new Object();
System.out.println(o1.equals(o2)); // false
System.out.println(o1.equals(o1)); // true
```

✅ Use case: identity comparison

❌ Not suitable for comparing values inside objects

🔹 2. `String.equals()`

📌 Overridden in `java.lang.String`

- Compares **content (character-by-character)** equality
- Case-sensitive
- Returns `true` if both strings have **identical sequences of characters**

🔧 Example:

```
String s1 = new String("hello");
String s2 = new String("hello");
System.out.println(s1 == s2);         // false (different objects)
System.out.println(s1.equals(s2));    // true (same content)
```

✅ Ideal for content comparison

✅ Consistent, predictable behavior

🔹 3. Custom Class `equals()`

📌 You must override `equals()` in custom classes to compare object contents instead of reference

🔧 Example:

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

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

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;

        Person other = (Person) obj;
        return age == other.age && name.equals(other.name);
    }
}
```

🔎 Usage:

```
Person p1 = new Person("Alice", 30);
Person p2 = new Person("Alice", 30);
System.out.println(p1.equals(p2)); // true (value-based comparison)
```

❗ Without overriding `equals()`, you'd get `false` (reference comparison from `Object`)

⚠️ Rules for Overriding `equals()` (Java Contract)

When overriding `equals()`, you **must ensure**:

1. **Reflexive**: `x.equals(x)` must be true.
2. **Symmetric**: `x.equals(y)` ⇔ `y.equals(x)`
3. **Transitive**: if `x.equals(y)` and `y.equals(z)` ⇒ `x.equals(z)`
4. **Consistent**: repeated calls give same result if no data changes.
5. **Null-safe**: `x.equals(null)` must return `false`.

🔄 Comparison Table

Feature	`Object.equals()`	`String.equals()`	Custom `equals()`
Compares	References only	Character content	Custom logic (your fields)

Feature	`Object.equals()`	`String.equals()`	Custom `equals()`
Overridden?	✗ No	✓ Yes	✓ You must override
Result for equal content	✗ false	✓ true	✓ true (if correctly written)
Best for	Identity comparison	Text equality	Value-based comparison

---

⊘ Common Mistakes

- Forgetting to override `equals()` when comparing objects by value.
- Using `==` instead of `equals()` for strings and objects.
- Failing to follow the contract (especially null checks).
- Not overriding `hashCode()` along with `equals()` (important for hash collections).

✓ Summary

You want to...	Use...
Check if two references are the same	`==` or `Object.equals()`
Check if two strings have same content	`String.equals()`
Check if two custom objects are equal	Override `equals()`

✓ Yes — If two objects are equal, then their `hashCode()` values must be the same.

This is **required by the Java contract** for `equals()` and `hashCode()`.

◆ Java's Contract (from `Object` class documentation):

If `a.equals(b)` is `true`, then `a.hashCode() == b.hashCode()` must also be true.

But:

If `a.hashCode() == b.hashCode()`, that **does NOT** mean `a.equals(b)` is `true`.

This is because **different objects can have the same hash code** — it's called a **hash collision**.

Example: Valid Implementation

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

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

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

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

🔎 Usage:

```
Person p1 = new Person("Alice", 30);
Person p2 = new Person("Alice", 30);

System.out.println(p1.equals(p2));      // true
System.out.println(p1.hashCode() == p2.hashCode()); // true ✅
```

🔥 What if the contract is broken?

If `equals()` says two objects are equal but `hashCode()` gives different values:

- You'll break collections like `HashMap`, `HashSet`, `Hashtable`
- Lookups or inserts will fail or behave unpredictably

❓ Summary

Condition	Must be true?
`a.equals(b)` ⇒ `a.hashCode() == b.hashCode()`	✅ Yes
`a.hashCode() == b.hashCode()` ⇒ `a.equals(b)`	❌ No (not guaranteed)

Absolutely! Below is a **curated list of advanced and commonly asked HashMap-related interview questions**, with detailed explanations to help you prepare for **Java interviews** — from mid-level to senior roles.

✅ Advanced & Important HashMap Interview Questions

---

1. ◆ How does HashMap work internally in Java?

Answer:

- HashMap uses an **array of buckets**.
- Each bucket is a **linked list (or tree since Java 8)** of key-value pairs.
- When you put a key, it:

1. Calculates `hashCode()` of the key.
2. Applies hash function to get an index.
3. Inserts the key-value pair into that index.

- On collision, it:
  - Adds to the list (chaining).
  - From Java 8 onward, converts the list to a **balanced tree** (TreeNode) if size > 8 and bucket array length > 64.

---

2. ◆ What happens if two keys have the same hashCode?

Answer:

- Java uses **`equals()`** to distinguish them after hash collision.
- The key-value pairs are **stored in a list or tree** at the same bucket index.
- HashMap uses **chaining** to handle such collisions.

---

3. ◆ What is the load factor? Why is it important?

Answer:

- Load factor = `number of entries / capacity`.
- Default is **0.75**.
- When load factor is exceeded, the HashMap **resizes** (doubles in capacity).
- Important for **performance trade-off** between space and time.

---

4. ◆ What is rehashing in HashMap? When does it occur?

Answer:

- Rehashing happens when the HashMap **resizes**.
- All existing entries are **re-calculated** and placed into new buckets.
- Happens when number of elements exceeds `capacity * load factor`.

---

5. ◆ Can a HashMap have `null` key or values?

Answer:

- ✅ HashMap **allows one `null` key** and multiple `null` values.
- Because `null.hashCode()` is undefined, it's stored in bucket 0.

---

6. ◆ What is the difference between `HashMap` and `Hashtable`?

Feature	HashMap	Hashtable
Thread-safe	❌ No	✅ Yes (synchronized)
Performance	✅ Faster	❌ Slower
Null keys/values	✅ Allowed	❌ Not allowed
Introduced in	Java 1.2	Java 1.0

---

7. ◆ What happens if you override `equals()` but not `hashCode()` in a key object?

Answer:

- Breaks the contract:
  - Even if keys are **logically equal**, they end up in **different buckets**.
- HashMap behaves incorrectly — fails to retrieve the value.

✅ Always override both `equals()` and `hashCode()` together!

---

8. ◆ How is HashMap different in Java 8?

Answer:

- Java 8 introduced:
  - **Tree-based buckets**: when a bucket's size exceeds 8 and the overall capacity is >64, it switches from a **LinkedList to a balanced Tree (Red-Black Tree)**.
  - Improves worst-case performance from **O(n)** to **O(log n)** in case of many collisions.

---

9. ◆ Can the keys in a HashMap be mutable? What are the consequences?

Answer:

- ❌ Not recommended.
- If you **change a key after insertion**, the `hashCode()` changes, and the key is now in the **wrong bucket**.
- Retrieval, update, or removal **fails** because it looks in the wrong place.

---

10. ◆ How would you implement a custom object as a key in a HashMap?

Answer:

- Override both `equals()` and `hashCode()` in your custom class.
- Ensure:
  - `equals()` uses business logic to compare fields.
  - `hashCode()` uses the same fields.
- Use `Objects.hash(...)` or manual hash combining.

---

11. ◆ How do you iterate over a HashMap efficiently?

Answer:

- Use `entrySet()` for best performance:

```
for (Map.Entry<K, V> entry : map.entrySet()) {
    System.out.println(entry.getKey() + " = " + entry.getValue());
}
```

- Avoid calling `get()` repeatedly inside a keySet loop.

---

12. ◆ Can two unequal objects have the same hashCode()?

Answer:

- ✅ Yes. It's called a **hash collision**.
- That's why Java always uses `equals()` after matching hash codes.

---

13. ◆ What are alternatives to HashMap when thread safety is required?

Answer:

- `ConcurrentHashMap` – Thread-safe and faster than `Hashtable`
- `Collections.synchronizedMap(...)` – Wraps a normal map with synchronization
- Avoid `Hashtable` unless legacy compatibility is needed.

---

14. ◆ What if a `hashCode()` is always the same (e.g., returns constant)?

Answer:

- All keys land in the **same bucket** → degrades to a **LinkedList (O(n))**
- Horrible performance in `HashMap`

---

15. ◆ Can HashMap cause memory leaks? How?

Answer:

- Yes, especially if keys are **mutable** and used in a long-living map.
- Weak references in `WeakHashMap` are often used to avoid this.

🧠 Bonus Real-Life Scenario:

❓ How would you design a cache system using HashMap?

Answer:

- Use `LinkedHashMap` for **LRU (Least Recently Used)** eviction
- Or use `ConcurrentHashMap` with eviction logic for thread-safe cache

✅ HashMap Collision in Java — Explained Clearly

A **HashMap collision** happens when **two different keys** generate the **same hash bucket index** during insertion into a `HashMap`.

🔷 How HashMap Stores Entries

Internally, `HashMap` uses:

- An **array of buckets**
- Each bucket is a **linked list** (or **tree** since Java 8)

When inserting:

1. It calculates `hashCode()` of the key.
2. Applies internal hashing (spread function).
3. Maps it to a bucket index: `index = hash & (array.length - 1)`
4. Inserts the entry into that bucket.

What is a Collision?

Two keys `k1` and `k2` such that:

```
!k1.equals(k2) && k1.hashCode() == k2.hashCode()
```

→ both keys **map to the same bucket index**

This is called a **collision**.

Example:

```
class Key {
    int id;
    Key(int id) { this.id = id; }

    @Override
    public int hashCode() {
        return 1;  // Force collision
    }

    @Override
    public boolean equals(Object obj) {
        return obj instanceof Key && ((Key) obj).id == this.id;
    }
}
Map<Key, String> map = new HashMap<>();
map.put(new Key(1), "A");
map.put(new Key(2), "B");
```

Both keys go into the **same bucket**, causing a **collision**, but `equals()` keeps them separate.

What Happens Internally?

- Java resolves collisions using **chaining**:
  - If two entries go to the same bucket, they are stored in a **linked list** (or tree from Java 8).
  - When you retrieve, it:
    - Goes to the bucket
    - Traverses the list or tree using `equals()`

Java 8+: Tree-Based Buckets

When:

- More than 8 entries in a single bucket **AND**
- Capacity ≥ 64

Java switches from **LinkedList** to **Red-Black Tree** in that bucket.

✓ Improves lookup time from **O(n)** to **O(log n)**

⚠ Consequences of Too Many Collisions

- Poor performance (`O(n)` instead of `O(1)`)
- Security risk (DoS attacks via hash collisions)
- That's why Java uses **better hash spreading functions**

Key Notes

Concept	Explanation
What is a collision?	Two different keys hash to the same bucket
How is it handled?	Chaining (LinkedList or Tree)
From Java 8 onward	Tree used if bucket size > 8 and capacity > 64

Concept	Explanation
What ensures uniqueness?	`equals()` method
Worst-case performance	`O(n)` without treeification, `O(log n)` with tree

---

Great question! Understanding the difference between **Collection** and **Stream** in Java is important for writing **clean, efficient, and modern Java code**, especially since the introduction of **Streams in Java 8**.

✅ Collection vs Stream in Java — Key Differences

Feature	Collection	Stream
Belongs to	`java.util` package	`java.util.stream` package
Purpose	To **store and manage data**	To **process data**
Type	**Data structure**	**Pipeline for computation** (not storage)
Storage	Stores elements in memory	Doesn't store — processes data on-the-fly
Traversal	External iteration (via loops or iterators)	Internal iteration (via functional ops)
Mutable?	Yes (you can add/remove elements)	No (usually immutable and stateless)
Can be reused?	Yes	❌ No — streams can be used only **once**
Operations	CRUD operations (add, remove, etc.)	Map, filter, reduce, collect, etc.
Parallel Processing	Manual (using threads)	Built-in with `.parallelStream()`

🔹 What is a Collection?

A **Collection** is an **in-memory container** that holds multiple elements.

Examples:

- `List`, `Set`, `Queue`, `Map` (via `entrySet()`)

Example usage:

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

🔹 What is a Stream?

A **Stream** is a **pipeline** of data processing steps. It doesn't hold data — it **reads from a data source**, applies operations, and outputs the result.

Example:

```
List<String> names = Arrays.asList("Alice", "Bob", "Alex");

names.stream()
     .filter(n -> n.startsWith("A"))
     .map(String::toUpperCase)
     .forEach(System.out::println);
```

✅ Streams are **declarative**, making code more readable and parallelizable.

⚠️ Stream is not a data structure

```
Stream<String> stream = names.stream();
```

- It reads data **lazily** from the collection
- Once consumed, it **cannot be reused**

🔄 When to Use Which?

Use Case	Use Collection	Use Stream
Storing data	✅	❌
Iterating with simple logic	✅	✅
Chaining filters and transformations	❌	✅
Large data + parallel processing	❌	✅
Modifying elements (add/remove)	✅	❌

🧠 Summary Analogy:

Think of **Collection** as a **warehouse** (stores items), and **Stream** as a **conveyor belt** (processes items).

Great question! Understanding the **difference between intermediate and terminal operations** is essential to effectively using the **Stream API** in Java (introduced in Java 8).

✅ Intermediate vs Terminal Operators in Java Streams

Feature	Intermediate Operation	Terminal Operation
Purpose	Build a **stream pipeline**	**Trigger** the pipeline and produce result
Returns	A new **Stream**	A **non-stream** result or **side-effect**
Executed Immediately?	❌ **Lazy** (evaluated only when terminal op is called)	✅ **Eager** — executes the pipeline
Can chain?	✅ Yes, multiple intermediates allowed	❌ No, it's the end of the pipeline
Examples	`filter()`, `map()`, `sorted()`, `limit()`	`collect()`, `forEach()`, `count()`, `reduce()`

🔹 Intermediate Operations (Lazy)

They **don't process data immediately**, just define the steps to process.

Examples:

- `stream.filter(...)`
- `stream.map(...)`
- `stream.sorted()`
- `stream.distinct()`
- `stream.limit(n)`

💡 These are **composed into a pipeline**, but **nothing happens until a terminal op is hit**.

```
Stream<String> names = Stream.of("Alice", "Bob", "Alex");
Stream<String> filtered = names.filter(s -> s.startsWith("A")); // No processing yet!
```

🔸 Terminal Operations (Eager)

They **execute** the pipeline and produce the **final result** or **side-effect**.

Examples:

- `forEach(...)`
- `collect(...)`
- `count()`
- `reduce(...)`
- `anyMatch(...)`, `allMatch(...)`, `noneMatch(...)`

```
Stream.of("Alice", "Bob", "Alex")
      .filter(s -> s.startsWith("A"))    // Intermediate
```

```
    .map(String::toUpperCase)          // Intermediate
    .forEach(System.out::println);     // Terminal (executes pipeline)
```

🔁 Analogy

Think of a stream pipeline like **building a train**:

- **Intermediate ops**: Add or modify train cars (filter, map, sort).
- **Terminal op**: Start the train and make it go (collect, forEach, count).

❓ Important Notes:

- Once a **terminal operation** is invoked, the **stream is consumed** and **cannot be reused**.
- You can **chain multiple intermediate operations**, but **only one terminal operation**.

Sure! Here's a comprehensive list of all the **Terminal Operations** in the **Java Stream API**, introduced in **Java 8** — these operations **trigger stream processing** and produce a final result or side-effect.

✅ Terminal Operators in Java Streams

◆ 1. forEach()

- Performs an action for each element.

```
stream.forEach(System.out::println);
```

◆ 2. forEachOrdered()

- Maintains encounter order in **parallel streams**.

```
stream.parallel().forEachOrdered(System.out::println);
```

◆ 3. toArray()

- Converts stream to array.

```
String[] arr = stream.toArray(String[]::new);
```

◆ 4. reduce()

- Combines elements into one using an accumulator.

```
int sum = stream.reduce(0, Integer::sum);
```

◆ 5. collect()

- Converts stream to a collection or result via `Collector`.

```
List<String> list = stream.collect(Collectors.toList());
```

◆ 6. min() / max()

- Returns the minimum/maximum element based on comparator.

```
Optional<String> min = stream.min(String::compareTo);
```

◆ 7. count()

- Returns the number of elements.

```
long total = stream.count();
```

◆ 8. anyMatch()

- Returns true if **any** element matches the predicate.

```
boolean found = stream.anyMatch(s -> s.startsWith("A"));
```

◆ 9. allMatch()

- Returns true if **all** elements match the predicate.

```
boolean allStartWithA = stream.allMatch(s -> s.startsWith("A"));
```

◆ 10. noneMatch()

- Returns true if **no** elements match the predicate.

```
boolean noneMatchZ = stream.noneMatch(s -> s.startsWith("Z"));
```

◆ 11. findFirst()

- Returns the **first** element in the stream (Optional).

```
Optional<String> first = stream.findFirst();
```

◆ 12. findAny()

- Returns **any** element (useful for parallel streams).

```
Optional<String> any = stream.findAny();
```

⍰ Key Notes:

Operation Type	Examples
Side-effect	`forEach()`, `forEachOrdered()`
Reduction	`reduce()`, `collect()`
Quantifier	`anyMatch()`, `allMatch()`, `noneMatch()`
Element selection	`findFirst()`, `findAny()`
Aggregate	`count()`, `min()`, `max()`

Absolutely! Here's a complete and categorized list of all commonly used **Intermediate Operators** in the **Java Stream API**.

✓ Intermediate Operations in Java Streams

These operations are **lazy** — they **don't execute** until a **terminal operation** is called. They **return a new Stream**, allowing for **method chaining** to build a pipeline.

◆ 1. filter(Predicate)

Filters elements based on a condition.

```
stream.filter(s -> s.startsWith("A"))
```

◆ 2. map(Function)

Transforms each element.

```
stream.map(String::toUpperCase)
```

◆ 3. flatMap(Function)

Flattens a stream of streams into a single stream.

```
stream.flatMap(list -> list.stream())
```

◆ 4. distinct()

Removes duplicate elements (based on `equals()`).

```
stream.distinct()
```

◆ 5. sorted()

Sorts elements in natural order.
`stream.sorted()`

◆ 6. sorted(Comparator)
Sorts elements using a custom comparator.
`stream.sorted(Comparator.reverseOrder())`

◆ 7. peek(Consumer)
Performs a side-effect (like logging) **without modifying the stream**.
`stream.peek(System.out::println)`

◆ 8. limit(long maxSize)
Limits the stream to a specified number of elements.
`stream.limit(5)`

◆ 9. skip(long n)
Skips the first `n` elements.
`stream.skip(2)`

◆ 10. takeWhile(Predicate) *(Java 9+)*
Takes elements **while** the condition is true.
`stream.takeWhile(s -> s.length() < 5)`

◆ 11. dropWhile(Predicate) *(Java 9+)*
Skips elements **while** the condition is true, then includes the rest.
`stream.dropWhile(s -> s.length() < 5)`

Summary Table

Operation	Description
`filter()`	Filters elements based on condition
`map()`	Transforms each element
`flatMap()`	Flattens nested streams
`distinct()`	Removes duplicates
`sorted()`	Sorts elements
`peek()`	For debugging/side-effects
`limit()`	Limits the stream to `n` elements
`skip()`	Skips the first `n` elements
`takeWhile()`	Takes elements while condition is true *(Java 9+)*
`dropWhile()`	Drops elements while condition is true *(Java 9+)*

Great! Here's the difference between **`clear()`** and **`removeAll()`** methods in Java Collections:

---

1. `clear()`

- **Purpose:** Removes **all elements** from the collection.
- **Effect:** The collection becomes **empty**.
- **Usage:** No arguments needed.

- **Example:**
- `List<String> list = new ArrayList<>(Arrays.asList("a", "b", "c"));`
- `list.clear();  // list is now empty`

---

2. `removeAll(Collection<?> c)`

- **Purpose:** Removes **all elements** from the collection that are also contained in the specified collection `c`.
- **Effect:** Only elements **present in both collections** are removed.
- **Usage:** Pass another collection as an argument.
- **Example:**
- `List<String> list1 = new ArrayList<>(Arrays.asList("a", "b", "c"));`
- `List<String> list2 = Arrays.asList("b", "c");`
- `list1.removeAll(list2);  // list1 now contains ["a"]`

Summary Table

Method	Removes	Parameter	Result
`clear()`	All elements	None	Collection becomes empty
`removeAll()`	Only elements present in the given collection	Another collection	Removes matching elements, others remain

---

Quick analogy:

- `clear()` = **wipe the board clean** (remove everything).
- `removeAll()` = **erase specific words** from the board (only those matching another list).

Sure! To **print an exception** in Java and see its details (like message and stack trace), you typically do one of the following inside a `catch` block:

---

1. Print exception message only

```
try {
    // code that may throw exception
} catch (Exception e) {
    System.out.println("Exception occurred: " + e.getMessage());
}
```

- Prints just the exception message (a string describing the error).

---

2. Print full stack trace (most common and useful)

```
try {
    // code that may throw exception
} catch (Exception e) {
    e.printStackTrace();
}
```

- Prints the full stack trace to `System.err` — shows exception type, message, and where it occurred in code.

---

3. Print using `toString()`

```
try {
    // code that may throw exception
} catch (Exception e) {
    System.out.println(e);
}
```

- Prints exception class name and message (like `java.lang.NullPointerException: ...`).

Example:

```
public class ExceptionDemo {
```

```
    public static void main(String[] args) {
        try {
            int x = 5 / 0;  // ArithmeticException
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
    at ExceptionDemo.main(ExceptionDemo.java:4)
```

Notes:

1. You cannot use `this` and `super` inside the static context.
2. Array is an object and has a **length** (not size).
3. `Collections.shuffle()` is used to shuffle the data.

Sure! Here's a detailed explanation of **unreachable code** and **dead code** specifically in the context of **exception handling** in Java, covering all common cases:

Unreachable Code vs Dead Code

- **Unreachable Code:** Code that **cannot be executed ever** because the control flow never reaches it.
- **Dead Code:** Code that **can be reached**, but its execution **has no effect** on the program (e.g., variables never used).

Focus: Unreachable code in Java Exception Handling

Java compiler throws an error if it detects **unreachable code**, which often happens with `try-catch-finally` blocks due to `return`, `throw`, or infinite loops.

---

1. Unreachable code after `throw` or `return` inside try/catch/finally

```
try {
    throw new Exception();
    // System.out.println("Unreachable");  // Compile error: unreachable code
} catch (Exception e) {
    return;
    // int a = 5;  // Compile error: unreachable code
} finally {
    System.out.println("Finally always runs");
}
```

- Any statement **after `throw` or `return` inside the same block** is unreachable.

---

2. Unreachable code after `System.exit()`

```
try {
    System.exit(0);
    // code here is unreachable
} catch (Exception e) {
    // ...
}
```

---

3. Unreachable code after infinite loop or unconditional throw

```
try {
    while(true) {
        // infinite loop
    }
    // unreachable code here
} catch (Exception e) {}
```

Or:

```
try {
    throw new RuntimeException();
    // unreachable
} catch (Exception e) {}
```

---

4. Finally block after `return` or `throw`

- **`finally` always executes**, even if try/catch has return/throw.
- But **code after a `return` or `throw` in finally is unreachable**:

```
try {
    return;
} finally {
    return;          // OK (overrides try's return)
    // System.out.println("Unreachable"); // Error: unreachable code
}
```

Dead code in Exception Handling

- Dead code is **not a compile-time error**, but bad practice.
- For example, catch block that catches exceptions never thrown by try:

```
try {
    int a = 5 / 1;
} catch (ArithmeticException e) {
    System.out.println("Won't be called"); // This block is effectively dead code here
}
```

Summary Table

Scenario	Result	Reason
Code after `throw` or `return`	Unreachable	Control flow exits method/block
Code after `System.exit()`	Unreachable	JVM terminates
Code after infinite loop	Unreachable	Loop never exits
Dead catch block for unused exception	Dead code	No exception thrown to catch
Code after return in finally	Unreachable	Finally return overrides

---

Here's a clear comparison of **`blank final`** vs **`static final`** in Java, with definitions, examples, and key differences.

🔹 1. `blank final` in Java

✅ Definition:

A **blank final** variable is a `final` variable that is **declared but not initialized** at the point of declaration.
It **must be initialized exactly once**, typically in:

- An instance initializer block
- A constructor

🔸 Example:

```
class Test {
    final int x; // blank final

    Test(int x) {
        this.x = x; // initialized in constructor
    }
}
```

🔍 Key Points:

- Cannot be changed once assigned.

- Must be initialized before object construction is complete.
- Used for **immutable instance-level fields**.

🔷 2. `static final` in Java

✅ Definition:

A `static final` variable is a **constant** — a value shared across all instances and cannot be changed once assigned.

🔹 Example:

```
class Test {
    static final int CONSTANT = 10;
}
```

OR using a static block:

```
class Test {
    static final int CONSTANT;

    static {
        CONSTANT = 42;
    }
}
```

🔍 Key Points:

- Must be initialized at declaration or inside a **static block**.
- Belongs to the class, **not to objects**.
- Used for **constants** (`public static final` is common).

✅ Summary Table

Feature	`blank final`	`static final`
Initialization	In constructor or initializer block	At declaration or static block
Scope	Per instance	Per class
Access	Only via object	Via class name or object
Reassignment	❌ Not allowed	❌ Not allowed
Use case	Immutable object-specific value	Constants like `PI`, config values

---

No, **interfaces do not extend the `Object` class** in Java — **but** every **object** created from a class that implements an interface **still inherits from `Object`**.

✅ Key Points:

1. Interfaces do not extend any class, including `Object`.

```
interface MyInterface {
    void doSomething();
}
```

- `MyInterface` does **not** implicitly or explicitly extend `Object`.

---

2. Every class in Java extends `java.lang.Object`, either directly or indirectly.

So, if a class implements an interface, the resulting object **inherits all `Object` methods** like `toString()`, `hashCode()`, `equals()`, etc., from `Object`.

```
class MyClass implements MyInterface {
    public void doSomething() {
        System.out.println("Doing something");
    }
}
```

```
MyInterface obj = new MyClass();
System.out.println(obj.toString()); // Valid — comes from Object class
```

---

3. Why it works:

Even though the **interface doesn't extend `Object`**, the **implementing class does**, so the instance can still use `Object` methods.

✓ Summary:

Statement	True / False	Explanation
Interface extends `Object`	✗ False	Interfaces don't extend any class, not even `Object`
Interface has access to `Object` methods	⚠ No	Only through implementing class instances
Class implementing interface extends `Object`	✓ True	All classes in Java implicitly extend `Object`
Object of an interface type can call `toString()`	✓ True	Because actual object is an instance of a class that extends `Object`

Great question! Let's break down **throwing** and **rethrowing** exceptions in Java — concepts that often show up in interviews and production-level code.

✓ 1. Throwing an Exception

You "throw" an exception using the `throw` keyword. This is how you signal that something went wrong.

◆ Syntax:

```
throw new ExceptionType("error message");
```

◆ Example:

```
public void validate(int age) {
    if (age < 18) {
        throw new IllegalArgumentException("Age must be 18+");
    }
}
```

- The method **throws** an exception when an invalid condition occurs.

✓ 2. Rethrowing an Exception

Rethrowing means catching an exception and **throwing it again**, either the same or a different one.

◆ Case 1: Rethrow the same exception

```
public void process() throws Exception {
    try {
        // some code that throws Exception
    } catch (Exception e) {
        System.out.println("Caught exception: " + e.getMessage());
        throw e; // rethrowing the same exception
    }
}
```

- You must declare `throws Exception` in method signature.
- Useful for **logging** or **wrapping** before passing up the call stack.

◆ Case 2: Rethrow a different exception (wrap it)

```
try {
```

```
    someRiskyMethod();
} catch (IOException e) {
    throw new RuntimeException("Wrapping IO Exception", e);
}
```

- You're **converting a checked exception into an unchecked one**.
- Helps simplify exception handling or attach extra info.

◆ Case 3: Java 7+ rethrow feature (precise rethrow)

If a caught exception is not modified, Java 7+ allows rethrowing without changing the method signature for multiple exception types.

```
void read() throws IOException, SQLException {
    try {
        // may throw IOException or SQLException
    } catch (Exception e) {
        throw e; // Java infers which types are possible
    }
}
```

Important Notes:

Concept	Details
`throw`	Used to explicitly throw an exception
`throws`	Declares what exceptions a method can throw
Rethrow same exception	Just `throw e;` after catching it
Rethrow different type	`throw new ExceptionType(e);`
Java 7 precise rethrow	Infers which checked exceptions are rethrown based on try block analysis

---

Chained Exception in Java

Chained Exception is a mechanism in Java that allows you to **associate one exception with another** — useful when one exception causes another.

Why Use Chained Exceptions?

- To **preserve original cause** while throwing a new, higher-level exception.
- Useful in **exception wrapping** when layering application logic.

Syntax

```
Throwable initCause(Throwable cause)
Throwable getCause()
```

Or directly via constructor:

```
new Exception(String message, Throwable cause)
```

Example: Chained Exception Using Constructor

```
public class ChainedExample {
    public static void main(String[] args) {
        try {
            validate();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static void validate() throws Exception {
        try {
```

```
            int a = 5 / 0; // Causes ArithmeticException
        } catch (ArithmeticException e) {
            throw new Exception("Wrapped exception", e); // Chaining
        }
    }
}
```

🔍 Output (stack trace shows cause):

```
java.lang.Exception: Wrapped exception
    at ChainedExample.validate(ChainedExample.java:12)
    at ChainedExample.main(ChainedExample.java:5)
Caused by: java.lang.ArithmeticException: / by zero
    at ChainedExample.validate(ChainedExample.java:10)
```

Alternative: Using `initCause()`

```
Exception ex1 = new Exception("Root cause");
Exception ex2 = new Exception("Top-level exception");
ex2.initCause(ex1);
throw ex2;
```

Best Practices

- Use chained exceptions when:
  - You catch a low-level exception (e.g., `SQLException`) and want to throw a higher-level one (`CustomAppException`).
  - You want to log or expose the full error trail.

Summary

Method	Purpose
`getCause()`	Retrieves the original exception
`initCause()`	Sets the cause (if not using constructor)
Constructor with cause	Preferred for chaining

Here's a **visual representation** of the `byte` range in Java:

□ Java `byte` Basics

- Type: `byte`
- Size: **8 bits**
- Range: **-128 to 127**
- Formula: -2^7 to $2^7 - 1$

Visual Representation

```
Binary        | Decimal Value
--------------|---------------
1000 0000     | -128   ← MIN_VALUE
1000 0001     | -127
1000 0010     | -126
...           | ...
1111 1110     | -2
1111 1111     | -1
0000 0000     |  0
0000 0001     |  1
0000 0010     |  2
...           | ...
0111 1110     | 126
```

```
0111 1111     | 127   ← MAX_VALUE
```

📌 Notes

- **Signed 8-bit integer**.
- Uses **two's complement** representation for negative values.
- **Default value**: `0` (when uninitialized in class-level variables).

□ Code to Print Range:

```
public class ByteRange {
    public static void main(String[] args) {
        System.out.println("Byte range: " + Byte.MIN_VALUE + " to " + Byte.MAX_VALUE);
    }
}
```

Output:

```
Byte range: -128 to 127
```

Here are the **rules and behaviors for `static` methods in Java**, especially in the context of **inheritance, overriding, overloading, access, and more**:

✓ 1. Static Method Declaration

- Belongs to the **class**, not an instance.
- Syntax:
- `public static void methodName() { }`

🔁 2. Overloading Static Methods → ✓ Allowed

- You can have multiple static methods with the **same name but different parameters** in the same class.

```
class Test {
    static void display() { }
    static void display(int x) { }
}
```

✓ This is valid **method overloading**.

✗ 3. Overriding Static Methods → ✗ Not allowed (Called method hiding)

- Static methods are **not overridden**, they are **hidden** if redefined in a subclass.

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

class Child extends Parent {
    static void greet() {
        System.out.println("Child");
    }
}
Parent obj = new Child();
obj.greet(); // Output: Parent
```

- The method call depends on **reference type**, not object type.

🔒 4. Accessing Static Methods

- Call using class name (recommended):
- `ClassName.methodName();`
- Can also be called via object reference (but not preferred).

❓ 5. No `this` or `super` in static methods

- Because `this` and `super` refer to object context, which `static` methods don't have.

🏗️ 6. Static Methods in Interface (Java 8+)

- Allowed, but they **cannot be inherited**.

```
interface MyInterface {
    static void show() {
        System.out.println("Static in interface");
    }
}

MyInterface.show(); // Must be called using interface name
```

❓ 7. Static Method Cannot Be Abstract or Final

- `abstract`: Needs to be overridden, but static methods can't be.
- `final`: Static methods are already non-overridable (so `final` is redundant but allowed).

🛠️ 8. Main Method is Static

- Entry point in Java: `public static void main(String[] args)`
- Static so it can be called **without creating an object**.

✅ Summary Table

Feature	Static Method Behavior
Overloading	✅ Allowed
Overriding	❌ Not allowed (method hiding instead)
Access via object	✅ Allowed but discouraged
Access via class	✅ Recommended
Uses `this` or `super`	❌ Not allowed
Can be abstract	❌ No
Can be final	✅ Yes, but unnecessary
In interface	✅ Since Java 8, but not inherited

Here's a complete guide to **`private` methods in Java**, focusing on how they behave in terms of **overriding, overloading, inheritance, access**, and related rules.

🔐 Private Method Rules in Java

✅ 1. Access Level

- `private` methods are accessible **only within the class** in which they are declared.
- **Not accessible** to:
  - Subclasses
  - Other classes in the same package
  - Instances of the same class

🔁 2. Overloading → ✅ Allowed

You can **overload** a `private` method in the same class by changing the method signature.

```
class Example {
    private void show() {
        System.out.println("No args");
    }

    private void show(String msg) {
        System.out.println("With msg: " + msg);
    }
}
```

✓ Overloading is allowed because it's resolved at compile time and based on method signature.

✗ 3. Overriding → ✗ Not Allowed

- `private` methods are **not visible to subclasses**, so **they cannot be overridden**.
- If a subclass defines a method with the same signature, it's a **new, independent method** — not an override.

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

class Child extends Parent {
    private void display() { // This is not overriding
        System.out.println("Child");
    }
}
```

□ 4. Method Resolution

```
Parent obj = new Child();
obj.display(); // ✗ Compile-time error if display() is private in Parent
```

- Private methods are **not polymorphic**.
- **Static binding** occurs — decision made at compile time.

□ 5. Reflection Access

Although private methods are normally inaccessible, they **can be accessed using reflection**:

```
Method m = obj.getClass().getDeclaredMethod("privateMethod");
m.setAccessible(true);
m.invoke(obj);
```

⚠ Use with caution — breaks encapsulation and can lead to security risks.

🏭 6. Private Static Methods

Allowed — useful for helper methods within a class.

```
private static void helper() {
    // internal logic
}
```

□ 7. Private Methods in Interface (Java 9+)

- From Java 9 onward, **interfaces can have private methods**.
- Used to share common code between `default` and `static` methods inside the interface.

```
interface MyInterface {
    private void log(String msg) {
        System.out.println("Log: " + msg);
    }
}
```

✅ Summary Table

Feature	Private Method Behavior
Accessible in same class	✅ Yes
Accessible in subclass	❌ No
Overloading	✅ Allowed
Overriding	❌ Not allowed
Inheritance	❌ Not inherited
Reflection access	✅ Yes (with `setAccessible(true)`)
In interface	✅ Since Java 9

Great question! Here's a clear explanation comparing **synchronized**, **fail-fast**, and **performance aspects** of collection classes in Java — all important in real-world, multi-threaded and concurrent programming.

✅ 1. Synchronized Collections

🔹 What it means:

A **synchronized** collection ensures that only one thread can access the collection at a time — making it **thread-safe**.

🔹 Examples:

- `Collections.synchronizedList(new ArrayList<>())`
- `Vector`, `Hashtable` (legacy synchronized collections)

✅ Pros:

- Thread-safe in multi-threaded environments
- Easy to use for basic use cases

❌ Cons:

- **Lower performance** due to locking overhead
- Not suitable for high-concurrency scenarios

✅ 2. Fail-Fast Collections

🔹 What it means:

Fail-fast collections throw a **`ConcurrentModificationException`** if they detect structural modification (like `add()`/`remove()`) **while iterating**, **outside of the iterator itself**.

🔹 Examples:

- `ArrayList`, `HashMap`, `HashSet`, etc.

⚠️ Example:

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

for (String s : list) {
    list.remove(s); // 🔥 Throws ConcurrentModificationException
}
```

✅ Pros:

- Detects potential bugs early during development
- Fast because it doesn't use locks

❌ Cons:

- Not thread-safe
- Must use **Iterator's own `remove()`** method to safely modify

✅ 3. Performance Comparison

Feature	Synchronized Collections	Fail-Fast Collections	Concurrent Collections
Thread-safety	✅ Yes (via lock)	❌ No	✅ Yes (lock-free or fine-grained locks)
Performance	❌ Slower due to locking	✅ Faster	✅✅ Fastest in high concurrency
Modification during iteration	✅ Allowed (with care)	❌ ConcurrentModificationException	✅ Allowed (safe iteration)
Examples	`Vector`, `Hashtable`	`ArrayList`, `HashMap`	`ConcurrentHashMap`, `CopyOnWriteArrayList`

✅ Best Practice Recommendations

Use Case	Recommended Collection
Single-threaded, fast access	`ArrayList`, `HashMap`
Multi-threaded (simple)	`Collections.synchronizedX()`
Multi-threaded (high performance)	`ConcurrentHashMap`, `CopyOnWriteArrayList`
Safe iteration under modification	`Concurrent` collections only

📝 Summary

- **Synchronized**: Thread-safe, but slower due to locks.
- **Fail-fast**: Fast but not safe for concurrent modification.
- **Performance**: Best with **concurrent collections** like `ConcurrentHashMap`.

🔐 Synchronized and Multiple Thread Access in Java – Explained

In Java, `synchronized` is a **keyword** used to control **access to shared resources** when **multiple threads** are involved. It ensures that **only one thread** can access a **critical section** of code at a time, thus preventing **data inconsistency, race conditions, or thread interference**.

🤔💥 Why Synchronization Is Needed in Multi-threaded Context

When multiple threads try to **read/write shared data simultaneously**, unpredictable results can occur.

❌ Without Synchronization (Race Condition):

```
class Counter {
    int count = 0;

    void increment() {
        count++;
    }
}
```

If 1000 threads call `increment()`, expected `count = 1000`, but you might get 800–999 due to race conditions.

✅ With Synchronization

```
class Counter {
    int count = 0;
```

```
    synchronized void increment() {
        count++;
    }
}
```

Now only **one thread at a time** can execute `increment()`, ensuring `count` is updated safely.

Types of Synchronization in Java

Type	How It Works
Method-level	Whole method is locked using `synchronized` keyword
Block-level	Synchronize only a part of the method (`synchronized(obj)`)
Static synchronization	Locks at the class level (useful for static methods)

Method Synchronization Example

```
public synchronized void printData() {
    // only one thread can access this at a time
}
```

Block Synchronization Example

```
public void printData() {
    synchronized(this) {
        // critical section
    }
}
```

How Synchronization Works Internally

- Java uses **monitor locks** (aka intrinsic locks).
- Each object has a **monitor**.
- When a thread enters a synchronized block/method, it acquires the **monitor lock**.
- Other threads **block** until the lock is released.

⚠ Pitfalls of Using `synchronized`

Problem	Description
Deadlock	Two threads wait on each other's locks forever
Thread contention	Performance drops when many threads are blocked
Reduced concurrency	Locking reduces parallelism

Real Use Case

For example, in a **banking app**, if two threads update the **balance** of the same account without synchronization, users might see incorrect balances.

✓ Summary

Concept	Meaning
`synchronized`	Ensures mutual exclusion (one thread at a time)
Multi-threaded access	Multiple threads accessing shared data simultaneously
Risk without sync	Race conditions, data corruption
Solution	Use synchronized methods or blocks

Great question! Let's clarify the relationship between **synchronized classes** and **fail-fast behavior** in Java:

🔍 Do Synchronized Classes Support Fail-Fast?

➤ ❌ No — synchronized collections do not reliably support fail-fast behavior.

Even though they are thread-safe for individual operations, **modifying them while iterating may still throw `ConcurrentModificationException`**, or worse, cause **unpredictable behavior**.

🔒 Synchronized Collections Example

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

for (String item : list) {      // Potentially unsafe!
    list.remove(item);          // May throw ConcurrentModificationException
}
```

Even though the list is synchronized, this loop can throw a **fail-fast exception**, because the **iterator is not automatically synchronized**.

✅ To Make Iteration Thread-Safe on Synchronized Collections:

You must **manually synchronize** the entire iteration block:

```
synchronized(list) {
    Iterator<String> it = list.iterator();
    while (it.hasNext()) {
        String item = it.next();
        it.remove();  // Safe
    }
}
```

⍰ Key Concepts

Term	Description
Synchronized Collection	Thread-safe for individual operations, but not safe for iteration by default
Fail-Fast	Iterator detects concurrent modification and throws `ConcurrentModificationException`
Fail-Safe	Iterator doesn't throw error (e.g. `CopyOnWriteArrayList`)

🔄 Summary

Question	Answer
Are synchronized collections fail-fast?	❌ Not guaranteed
Can they throw `ConcurrentModificationException`?	✅ Yes, during iteration
How to safely iterate?	Use `synchronized(list)` block manually

Great question! Let's clarify:

✅ Does Thread-Safe Mean Synchronized?

Not necessarily.

Thread-safe means a class or method can be **used safely by multiple threads** at the same time **without corrupting data or producing unpredictable results**.

🔑 Synchronization is one way to achieve thread safety, but not the only way.

🔄 Difference Explained

Concept	Meaning
Thread-safe	The code behaves correctly when accessed by multiple threads concurrently
Synchronized	A **mechanism** (keyword) to make code **thread-safe** by locking shared resources

---

🧪 Examples

1. ✅ Thread-safe via `synchronized` (Lock-based)

```
public synchronized void increment() {
    count++;
}
```

- Ensures only one thread at a time updates `count`.

---

2. ✅ Thread-safe via lock-free or concurrent design

- **`ConcurrentHashMap`** is thread-safe but doesn't use full synchronization.
- **`AtomicInteger`** uses **CAS (Compare-And-Swap)**, not `synchronized`.

```
AtomicInteger counter = new AtomicInteger();
counter.incrementAndGet(); // Thread-safe without synchronized
```

---

3. ✅ Thread-safe via immutable objects

```
String s = "Java"; // Immutable and thread-safe by design
```

- **No synchronization needed**, since data never changes.

⚠️ Important

- `synchronized` provides **mutual exclusion** (locking), but it can lead to:
  - **Performance bottlenecks**
  - **Deadlocks**, if misused

✅ Summary

Statement	True?
Thread-safe always means synchronized	❌ No
Synchronized is one way to achieve thread safety	✅ Yes
Thread safety can be achieved without locks	✅ Yes (via immutability, atomic classes, concurrent collections)

Here's a **detailed and professional explanation of `IntStream`** in Java with various **unique class name examples** to help understand and apply it in different scenarios.

🔷 What is `IntStream`?

`IntStream` is a **specialized stream** in Java 8+ for working **specifically with `int` primitives** — avoiding boxing overhead that comes with `Stream<Integer>`.
It's part of the **`java.util.stream`** package.

🔷 Why Use `IntStream`?

- Avoids unnecessary boxing/unboxing of `int` values.
- Offers **efficient operations** like sum, average, range, etc.

- Works with **functional-style** operations on primitive `int` values.

Common Methods of `IntStream`

Method	Description
`range(int, int)`	Returns a stream from start (inclusive) to end (exclusive)
`rangeClosed(int, int)`	Inclusive range
`of(int...)`	Creates a stream from given values
`iterate(...)`	Infinite stream (with limit) using a seed + function
`map`, `filter`, `sum`, `average`, `distinct`, `sorted`	Standard operations

✓ 1. Basic Range Stream

□ Class: `NumberPrinter`

```
import java.util.stream.IntStream;

public class NumberPrinter {
    public static void main(String[] args) {
        IntStream.range(1, 6).forEach(System.out::println);
    }
}
```

Output: `1 2 3 4 5`

✓ 2. Filter Even Numbers

□ Class: `EvenNumberFilter`

```
public class EvenNumberFilter {
    public static void main(String[] args) {
        IntStream.rangeClosed(1, 10)
                 .filter(n -> n % 2 == 0)
                 .forEach(System.out::println);
    }
}
```

Output: `2 4 6 8 10`

✓ 3. Sum of Squares

□ Class: `SquareSummer`

```
public class SquareSummer {
    public static void main(String[] args) {
        int sum = IntStream.range(1, 6)
                           .map(n -> n * n)
                           .sum();
        System.out.println("Sum of squares: " + sum);
    }
}
```

Output: `Sum of squares: 55`

✓ 4. Generate Multiples of 3

□ Class: `MultiplesGenerator`

```
public class MultiplesGenerator {
    public static void main(String[] args) {
        IntStream.iterate(3, n -> n + 3)
                 .limit(5)
                 .forEach(System.out::println);
    }
```

```
}
```
Output: `3 6 9 12 15`

✅ 5. Convert to List

📄 Class: `StreamToList`

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class StreamToList {
    public static void main(String[] args) {
        List<Integer> list = IntStream.range(1, 5)
                                      .boxed()
                                      .collect(Collectors.toList());

        System.out.println(list);
    }
}
```
Output: `[1, 2, 3, 4]`

✅ 6. Statistical Operations

📄 Class: `StatsCalculator`

```
import java.util.OptionalDouble;
import java.util.stream.IntStream;

public class StatsCalculator {
    public static void main(String[] args) {
        OptionalDouble avg = IntStream.of(4, 8, 15, 16, 23, 42).average();
        avg.ifPresent(a -> System.out.println("Average: " + a));
    }
}
```
Output: `Average: 18.0`

✅ 7. Remove Duplicates and Sort

📄 Class: `UniqueSorter`

```
public class UniqueSorter {
    public static void main(String[] args) {
        IntStream.of(5, 2, 2, 8, 3, 5)
                 .distinct()
                 .sorted()
                 .forEach(System.out::println);
    }
}
```
Output: `2 3 5 8`

📌 Key Points Summary

- **Efficient for primitive `int`** operations.
- Avoids boxing — better performance than `Stream<Integer>`.
- Works seamlessly with **map, filter, reduce**, etc.
- Can be converted to boxed streams or collected into collections.

⚠️ Common Pitfall

Don't confuse `Stream<Integer>` and `IntStream` — they are **not interchangeable** without boxing/unboxing.

```
IntStream is = IntStream.of(1, 2, 3);
Stream<Integer> boxed = is.boxed(); // Converts to Stream<Integer>
```

Yes! The line:
`Map<String, String> sampleMap = Map.of("X", "Xylophone", "Y", "Yogurt");`
is a **Java 9+ concise way** to create an **immutable map**. But there are several **alternative ways** to create maps in Java — depending on your needs (mutable vs immutable, size, performance, etc).

✅ Different Ways to Create a Map in Java

---

1. Using `Map.of(...)` – Java 9+

```
Map<String, String> map = Map.of("X", "Xylophone", "Y", "Yogurt");
```

- ✅ Simple and clean
- ❌ Immutable – you cannot add/remove after creation
- ❌ Maximum 10 entries (use `Map.ofEntries()` for more)

---

2. Using `Map.ofEntries(...)` – Java 9+

```
Map<String, String> map = Map.ofEntries(
    Map.entry("X", "Xylophone"),
    Map.entry("Y", "Yogurt")
);
```

- ✅ Immutable
- ✅ Supports more than 10 entries

---

3. Using `HashMap` – Mutable

```
Map<String, String> map = new HashMap<>();
map.put("X", "Xylophone");
map.put("Y", "Yogurt");
```

- ✅ Fully mutable (can add, remove, update)
- ✅ Common in real-world applications
- ❌ Not thread-safe (unless wrapped)

---

4. Using `Collections.unmodifiableMap()` – Immutable wrapper

```
Map<String, String> temp = new HashMap<>();
temp.put("X", "Xylophone");
temp.put("Y", "Yogurt");

Map<String, String> map = Collections.unmodifiableMap(temp);
```

- ✅ Makes a mutable map immutable
- Useful for returning read-only maps from APIs

---

5. Using Java 8 Streams (e.g., from a list)

```
List<String> keys = Arrays.asList("X", "Y");
List<String> values = Arrays.asList("Xylophone", "Yogurt");

Map<String, String> map = IntStream.range(0, keys.size())
    .boxed()
    .collect(Collectors.toMap(keys::get, values::get));
```

- ✅ Dynamic and flexible
- ✅ Good for data transformation

---

6. Using Double Brace Initialization (not recommended)

```
Map<String, String> map = new HashMap<>() {{
    put("X", "Xylophone");
    put("Y", "Yogurt");
```

```
}};
```

- ✅ Quick for demos/tests
- ❌ Creates an anonymous inner class
- ❌ Can cause memory leaks – **avoid in production**

📝 Summary Table

Method	Mutable	Java Version	Notes
`Map.of()`	❌ No	9+	Max 10 entries
`Map.ofEntries()`	❌ No	9+	Unlimited entries
`new HashMap<>()`	✅ Yes	All	Most flexible
`Collections.unmodifiableMap()`	❌ No	All	Wraps mutable map
`Streams.toMap()`	✅ Yes	8+	Good for transformation
Double-brace init	✅ Yes	All	Avoid in production

Absolutely! Let's walk through the execution of your Java class step by step, focusing on **memory management** — specifically, how the **stack** and **heap** are used during runtime.

✅ Java Memory Areas Overview

- **Stack**:
  - Stores **method call frames** (local variables, parameters).
  - Each thread has its **own stack**.
  - Memory is automatically freed when a method ends.
- **Heap**:
  - Stores **objects and instance variables**.
  - Shared across all threads.
  - Garbage collected when objects are no longer referenced.

📄 Your Code

```
class StackAndHeapMemory
{
    static void methodOne() {
        System.out.println("From Method One");
        methodTwo();
    }

    static void methodTwo() {
        System.out.println("From Method Two");
    }

    public static void main(String[] args) {
        System.out.println("Main Method Started");
        methodOne();
        System.out.println("Main Method Ended");
    }
}
```

🧠 Step-by-Step Execution with Memory Visualization

🔹 1. Program Starts – `main()` is called

Call Stack:

```
| main()                | ← top of stack (active frame)
```

- Stack frame for `main()` is created.
- `args` is stored in this frame (passed to `main()`).
- `"Main Method Started"` is printed.

---

2. Call to `methodOne()` from `main()`

Call Stack:

```
| methodOne()           |  ← active
| main()                |
```

- New frame for `methodOne()` is pushed onto the stack.
- `"From Method One"` is printed.

---

3. Call to `methodTwo()` from `methodOne()`

Call Stack:

```
| methodTwo()           |  ← active
| methodOne()           |
| main()                |
```

- New frame for `methodTwo()` is added.
- `"From Method Two"` is printed.

---

4. `methodTwo()` finishes

- Its frame is **popped** from the stack.

Call Stack:

```
| methodOne()           |  ← active
| main()                |
```

---

5. `methodOne()` finishes

- Its frame is popped as well.

Call Stack:

```
| main()                |  ← active
```

- `"Main Method Ended"` is printed.

---

6. `main()` finishes

- Entire stack is cleared.
- Program terminates.

📌 Heap Memory in This Program

- Since there are **no object instances** (`new` keyword not used), the **heap is unused** in this example.
- If you had done something like:
- `StackAndHeapMemory obj = new StackAndHeapMemory();`
  then `obj` would be stored in the **heap**, and the reference `obj` would be in the **stack**.

🔁 Summary Table

Operation	Stack Change	Heap Change
`main()` starts	Push `main()` frame	Nothing
Call `methodOne()`	Push `methodOne()` frame	Nothing
Call `methodTwo()`	Push `methodTwo()` frame	Nothing
`methodTwo()` ends	Pop `methodTwo()`	Nothing
`methodOne()` ends	Pop `methodOne()`	Nothing
`main()` ends	Pop `main()`	Nothing

□ Bonus Tip

- Stack memory is **fast**, but **limited in size**.
- Heap memory is **larger** but **slower**, and managed by the garbage collector.

Great example! This code demonstrates **class loading**, **static initializations**, and how **Java manages memory** between the **stack**, **heap**, and **method area (part of metaspace in modern JVMs)**.
Let's walk through it **step by step** and explain memory behavior, especially how the **stack**, **heap**, and **method area** are used.

□ Java Memory Overview for This Example

1. Method Area (Metaspace)

- Stores **class metadata**: class names, static variables, static blocks, method definitions.
- Each class is loaded once per classloader.

2. Heap

- Stores **objects and instance variables** (not used here).

3. Stack

- Stores **method call frames** (local variables, return address).
- Each thread has its own stack.

✓ Code Breakdown

StaticComponents.java

```
class StaticComponents {
    static int staticVariable;

    static {
        System.out.println("StaticComponents SIB");
        staticVariable = 10;
    }

    static void staticMethod() {
        System.out.println("From StaticMethod");
        System.out.println(staticVariable);
    }
}
```

MainClass.java

```
public class MainClass {
    static {
        System.out.println("MainClass SIB");
    }

    public static void main(String[] args) {
        StaticComponents.staticVariable = 20;
        StaticComponents.staticMethod();
    }
}
```

🔍 Step-by-Step Execution with Memory Management

Step 1: Program starts → `MainClass` is loaded

- Class `MainClass` is loaded into **method area**
- Executes **static block** of `MainClass` (SIB = Static Initialization Block)

📤 Output:

```
MainClass SIB
```

Step 2: `main()` method starts

- Stack frame for `main(String[] args)` is created in the **stack**
- `args` is stored in that frame.

Stack:

```
| main()                | ← active frame
```

Step 3: Accessing `StaticComponents.staticVariable = 20`

- Before accessing static members of `StaticComponents`, the JVM checks if `StaticComponents` is already loaded.
- It is **not loaded yet**, so:
  - `StaticComponents` is loaded into the **method area**
  - Its **static block** is executed once

  📤 Output:

```
StaticComponents SIB
```

- `staticVariable` is initialized to `10` by static block
- Immediately after, it is updated to `20` by `main()`

Step 4: Call `StaticComponents.staticMethod()`

- `staticMethod()` is resolved from **method area**
- A new **stack frame** is added for `staticMethod()`

📤 Output:

```
From StaticMethod
20
```

Step 5: `staticMethod()` returns

- Stack frame for `staticMethod()` is popped

Step 6: `main()` completes

- Stack frame for `main()` is popped
- Program terminates

🧠 Memory Area Summary

Component	What It Stores
Method Area	Class info for `MainClass` and `StaticComponents`Static variables and methods
Heap	❌ Not used (no `new` object creation)
Stack	Stack frames for `main()` and `staticMethod()`

🖨️ Final Output

```
MainClass SIB
StaticComponents SIB
From StaticMethod
20
```

This is a **great example** to understand **Java memory management** — especially how **static** vs **non-static (instance)** members are handled between **heap**, **stack**, and **method area (metaspace)**.

🔍 Java Memory Areas Involved

Area	Purpose
Method Area (Metaspace)	Stores class metadata and **static** variables/methods
Heap	Stores **objects** (instances of class `A`)
Stack	Stores **method call frames** (local variables, references like `a1`)

✅ Code Breakdown and Memory Step-by-Step

Code Summary

```
class A {
    int nonStaticVariable;
    static int staticVariable;

    static void staticMethod() {
        System.out.println(staticVariable);
        // System.out.println(nonStaticVariable); ← ERROR
    }

    void nonStaticMethod() {
        System.out.println(staticVariable);
        System.out.println(nonStaticVariable);
    }
}
class MainClass {
    public static void main(String[] args) {
        A.staticVariable = 10;
        A.staticMethod(); // prints 10

        A a1 = new A();   // Object created in heap
        A a2 = new A();   // Another object

        System.out.println(a1.nonStaticVariable); // 0 (default)
        System.out.println(a1.staticVariable);    // 10

        a1.nonStaticMethod(); // prints 10, 0
        a1.staticMethod();    // prints 10

        System.out.println(a2.staticVariable); // 10
        a1.staticVariable = 20;                // static → shared
        System.out.println(a2.staticVariable); // 20
    }
}
```

❔ Step-by-Step Memory Behavior

✅ 1. Class Loading (`A` and `MainClass`)

- JVM loads `A` into **method area**
- `staticVariable` and `staticMethod()` live in method area.
- `nonStaticVariable` and `nonStaticMethod()` are blueprint only (used per object).

✅ 2. `main()` starts

Stack:

```
| main()                    | ← Active frame
```

- `args` stored in main's stack frame.

✅ 3. Set Static Variable

```
A.staticVariable = 10;
```

- `staticVariable` is updated in **method area**, shared by all instances.

✅ 4. Call Static Method

```
A.staticMethod(); // prints 10
```

- Adds new frame to stack → prints staticVariable = 10.

Stack:

```
| staticMethod()          |
| main()                  |
```

Then popped after execution.

✅ 5. Create Object a1

```
A a1 = new A();
```

- `a1` is a reference stored in **main()'s stack frame**.
- Object created in **heap**:
  - `nonStaticVariable = 0` (default int)

Heap:

```
a1 → A { nonStaticVariable: 0 }
```

✅ 6. Create Object a2

```
A a2 = new A();
```

- `a2` is another reference in stack
- Another object in heap (independent of `a1`)

Heap:

```
a1 → A { nonStaticVariable: 0 }
a2 → A { nonStaticVariable: 0 }
```

✅ 7. Access Instance and Static Variables

```
System.out.println(a1.nonStaticVariable); // 0
System.out.println(a1.staticVariable);    // 10
```

- Instance variable → comes from heap (`a1`)
- Static variable → comes from method area (shared)

✅ 8. Call Instance Method

```
a1.nonStaticMethod();
```

- Frame added to stack
- Inside:
  - Access `staticVariable = 10` → from method area
  - Access `nonStaticVariable = 0` → from heap object pointed by `a1`

✅ 9. Call Static Method Again via Object

```
a1.staticMethod(); // prints 10
```

- Even though called from object, it's still static
- Uses shared value from method area

✅ 10. Test Shared Static Behavior

```
System.out.println(a2.staticVariable); // 10
a1.staticVariable = 20;
System.out.println(a2.staticVariable); // 20
```

- `staticVariable` changed using `a1`, but affects `a2` (same shared copy in method area)

Memory Table Summary

Item	Memory Area	Notes
`staticVariable`	Method Area	Shared by all instances
`nonStaticVariable`	Heap (per object)	Unique for each object
`a1`, `a2` references	Stack (main)	Point to objects in heap
`staticMethod()`	Method Area	Called without object
`nonStaticMethod()`	Method Area (code), Stack (frame)	Uses heap and static data

---

Final Output

```
10
0
10
10
0
10
20
```

Great question! Arrays in Java are **objects**, and they are always stored in the **heap memory**, regardless of whether they are arrays of primitives or references.

Key Points

1. Arrays Are Objects

- Arrays are dynamically created using `new`, so they live in the **heap**.
- Even primitive arrays like `int[]`, `char[]` are objects.
- Arrays have:
  - a fixed size (known at creation)
  - an implicit `length` property (not a method)

2. Reference in Stack, Data in Heap

When you write:

```
int[] arr = new int[5];
```

- `arr` is a **reference variable** → stored in the **stack**.
- The actual array object and its elements are stored in the **heap**.

Memory Visualization

```
int[] arr = new int[3]; // arr[0] = 0, arr[1] = 0, arr[2] = 0
```

Stack:

```
| arr | --> reference to array in heap
```

Heap:

```
Array Object:
  [0][0][0]
```

If you assign values:

```
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
```

Heap changes:

```
Array Object:
  [10][20][30]
```

For Object Arrays

```
String[] names = new String[2];
```

```
names[0] = "Alice";
names[1] = "Bob";
```

- `names` reference → stack
- array object `[ref1, ref2]` → heap
- `"Alice"` and `"Bob"` → **String objects** → heap

! Bonus Facts

- **Multidimensional arrays** are arrays of arrays:
  `int[][] matrix = new int[2][3];`
- This creates:
  - `matrix` (reference in stack)
  - An array of 2 references → heap
  - Each reference points to an array of 3 `int`s → heap

✓ Summary Table

Component	Memory Area	Notes
Array reference	Stack	Variable pointing to array
Array object	Heap	Where array data is stored
Primitive elements	Heap	Stored directly in array
Object elements	Heap	References in array, real objects elsewhere in heap

---

? Garbage Collection (GC) and `finalize()` in Java

Garbage Collection in Java is the **automatic process** of reclaiming memory used by **objects that are no longer reachable** from the program. It helps avoid memory leaks and makes memory management easier for developers.

✓ 1. What is Garbage Collection?

- In Java, when objects are **no longer referenced**, they become eligible for garbage collection.
- Java has a **Garbage Collector (GC)** running in the background that:
  - Finds unreachable objects
  - Reclaims their memory

✓ Example:

```
public class GCExample {
    public static void main(String[] args) {
        GCExample obj = new GCExample();
        obj = null; // Eligible for GC
        System.gc(); // Request GC (not guaranteed)
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("finalize() called");
    }
}
```

? 2. What is `finalize()` Method?

- `finalize()` is a method defined in `java.lang.Object`.
- It is **called by the GC before the object is removed** from memory.
- Used to perform **cleanup** operations like:
  - Closing files

- Releasing resources
- Logging object destruction

⚠ Important:

- `finalize()` **is not guaranteed** to be called.
- JVM may **not run it** if program exits before GC.
- It can only be called **once** per object.
- As of Java 9+, it's **deprecated** and **removed in Java 18** due to **unpredictability and performance issues**.

🛑 Deprecated Warning

```
@Deprecated(since="9")
protected void finalize() throws Throwable
```

Use alternatives like:

- `try-with-resources` (for `Closeable`/`AutoCloseable` resources)
- Custom cleanup methods

✅ When Is an Object Eligible for GC?

An object becomes **unreachable** when:

1. No references to it exist.
2. Reference is set to `null`.
3. It goes out of scope.
4. The object is part of a **reference cycle** that is unreachable from any GC root.

🔄 GC Roots

GC roots include:

- Local variables
- Static fields
- Active thread stacks
- JNI references

🔥 Example with finalize:

```
class Resource {
    @Override
    protected void finalize() {
        System.out.println("Resource finalized");
    }
}

public class Test {
    public static void main(String[] args) {
        Resource r = new Resource();
        r = null;
        System.gc(); // May trigger finalize()
    }
}
```

Output (may or may not appear):

```
Resource finalized
```

✅ Summary

Feature	Description
`Garbage Collection`	Automatic memory management (cleans unreachable objects)
`finalize()`	Called by GC before object is destroyed (deprecated)

Feature	Description
`System.gc()`	Requests GC (not guaranteed to run immediately)
Best Practice	Avoid `finalize()`, use try-with-resources / explicit cleanup

---

In Java, the **java.lang.ref** package provides different types of references beyond the default **strong reference**. These specialized references help control **how objects are managed by the Garbage Collector (GC)**, especially in memory-sensitive applications like caching or resource management.

✓ 1. Strong Reference (Default)

This is the **normal reference** type used in Java.

◆ Example:

```
String s = new String("Hello");
```

- As long as `s` is reachable, the object is **not eligible for GC**.
- GC **never reclaims** strongly reachable objects.

✓ 2. Soft Reference (`java.lang.ref.SoftReference`)

- GC **may reclaim** if memory is **low**, but **not aggressively**.
- Ideal for **caching**: objects remain as long as memory is available.

◆ Example:

```
import java.lang.ref.SoftReference;

public class SoftRefExample {
    public static void main(String[] args) {
        SoftReference<String> softRef = new SoftReference<>(new String("Soft Data"));
        System.out.println(softRef.get()); // Output: Soft Data
    }
}
```

If JVM is running low on memory, `softRef.get()` may return `null`.

✓ 3. Weak Reference (`java.lang.ref.WeakReference`)

- GC **collects the object immediately** when no strong references exist.
- Used in **WeakHashMap** keys, metadata, and plugin systems.

◆ Example:

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

public class WeakRefExample {
    public static void main(String[] args) {
        WeakReference<String> weakRef = new WeakReference<>(new String("Weak Data"));
        System.gc();
        System.out.println(weakRef.get()); // Might return null
    }
}
```

Very sensitive to GC. **Not reliable** for holding important data.

✓ 4. Phantom Reference (`java.lang.ref.PhantomReference`)

- Used for **post-mortem cleanup after** object is finalized but **before** memory is reclaimed.
- `get()` always returns `null`.
- Must be used with a `ReferenceQueue`.

◆ Example:

```
import java.lang.ref.*;
```

```
public class PhantomRefExample {
    public static void main(String[] args) {
        ReferenceQueue<String> refQueue = new ReferenceQueue<>();
        PhantomReference<String> phantomRef = new PhantomReference<>(new
String("Phantom"), refQueue);

        System.out.println(phantomRef.get()); // Always null
    }
}
```

Used in **low-level resource cleanup**, such as direct memory buffers or file handles.

🔁 Reference Type Comparison

Reference Type	Cleared by GC When?	`.get()` works?	Use Case
Strong	Never (until unreachable)	✅ Yes	Regular programming
Soft	Memory low	✅ Yes	Caching
Weak	On next GC pass	✅ Maybe (often null)	WeakHashMap, observers
Phantom	After finalization, before GC	❌ Always null	Advanced cleanup, memory tracking

---

🔥 Real-world Use Cases

Reference Type	Example Use Case
Strong	Regular object usage
Soft	Image cache in an image-loading library
Weak	`WeakHashMap` for storing plugin metadata
Phantom	Cleanup of native resources, large object pools

Great question! In Java, **`throw`**, **`throws`**, and **`Throwable`** are related to exception handling but serve very different purposes. Here's a clear breakdown:

---

1. `throw` (keyword)

- Used **to actually throw an exception** from a method or block of code.
- Followed by an instance of `Throwable` (usually an Exception or Error object).
- Syntax:

```
throw new IOException("File not found");
```

- Example:

```
public void readFile(String file) {
    if (file == null) {
        throw new NullPointerException("File is null");
    }
    // code to read file
}
```

---

2. `throws` (keyword)

- Used in a method **declaration** to **declare that this method may throw certain exceptions**.
- Specifies the exceptions that the method might throw so the caller is aware and can handle or propagate them.
- Syntax:

```
public void readFile(String file) throws IOException, SQLException {
    // method implementation
}
```

- Example:
- `public void readFile(String file) throws IOException {`
- `    // code that might throw IOException`
- `}`

---

3. `Throwable` (class)

- **Superclass of all errors and exceptions in Java**.
- Both `Exception` and `Error` classes inherit from `Throwable`.
- Objects of `Throwable` (or subclasses) are what you actually **throw** or **catch**.
- Structure:
- `java.lang.Object`
- `  ↳ java.lang.Throwable`
- `      ↳ java.lang.Error`
- `      ↳ java.lang.Exception`
- You usually deal with subclasses like `Exception`, `RuntimeException`, or `Error`, but technically you can throw or catch `Throwable`.

Summary Table

Keyword/Class	What it is	Purpose	Used where
`throw`	Keyword	To actually throw an exception instance	Inside method or block
`throws`	Keyword	To declare exceptions a method can throw	In method signature/declaration
`Throwable`	Class	Superclass of all errors and exceptions	Exception hierarchy, can be thrown or caught

---

Absolutely! **`map()`** and **`flatMap()`** are two very common methods in Java 8 streams, and while they seem similar, they have distinct behaviors and use cases. Here's a clear comparison:

---

1. `map()`

- Purpose: **Transforms each element** in the stream by applying a function to it.
- The function passed to `map()` takes one element and returns exactly **one** transformed element.
- The result is a stream of the same shape, but with transformed elements.
- If the transformation function returns a stream or collection, you end up with a **stream of streams** or a nested structure.

Example:

```
List<String> words = Arrays.asList("hello", "world");

// Convert each word to its length
List<Integer> lengths = words.stream()
                             .map(String::length)
                             .collect(Collectors.toList());

// Output: [5, 5]
```

---

2. `flatMap()`

- Purpose: Similar to `map()`, but **flattens** the result.
- The function passed to `flatMap()` takes one element and returns a **stream** of elements.
- `flatMap()` then **flattens** these streams into a **single continuous stream**.
- This is useful when each input element maps to multiple output elements.

Example:

```
List<String> sentences = Arrays.asList("hello world", "java streams");

// Split each sentence into words and flatten all words into a single stream
List<String> words = sentences.stream()
```

```
                                .flatMap(sentence -> Arrays.stream(sentence.split(" ")))
                                .collect(Collectors.toList());

// Output: ["hello", "world", "java", "streams"]
```

Summary Table

Aspect	`map()`	`flatMap()`
Input	Single element	Single element
Function returns	Single object (e.g., Integer)	Stream of objects
Result	Stream of transformed elements	Flattened stream (no nested streams)
Use case	One-to-one transformation	One-to-many transformation
Example output	Stream → Stream	Stream → Stream (flattened)

Key Takeaway

- Use **`map()`** when you want to transform each element into exactly one element.
- Use **`flatMap()`** when each element can be transformed into multiple elements (e.g., splitting strings into words) and you want to avoid nested streams.

Sure! Let's dive into **Java 8 Spliterator** — it's a somewhat advanced but powerful feature introduced to enhance iteration and parallel processing.

What is a Spliterator in Java 8?

- **Spliterator** stands for **Split + Iterator**.
- It's an **interface** introduced in Java 8 as part of the `java.util` package.
- Purpose: To **traverse and partition elements** of a source (like a Collection or Stream) efficiently.
- Mainly designed to support **parallel processing** by splitting a data source into multiple parts that can be processed in parallel.
- Combines iteration with the ability to **split** data into subparts.

Where is Spliterator used?

- Internally by **Streams API** to split data for parallel operations.
- You can also use it directly to iterate collections or arrays with more control.

Core Features

Feature	Description
tryAdvance(Consumer)	Performs an action on the next element if present.
forEachRemaining(Consumer)	Performs an action on all remaining elements.
trySplit()	Attempts to split this Spliterator into two parts for parallelism. Returns a new Spliterator for one part, keeping the original for the other.
estimateSize()	Gives an estimate of the number of elements remaining.
characteristics()	Returns properties (like ORDERED, DISTINCT, SORTED, SIZED, etc.) describing the behavior of the Spliterator.

Why use Spliterator?

- Supports **efficient parallel iteration** by splitting workload.
- Gives more control than traditional `Iterator`.
- Enables streams to be efficiently parallelized.

Simple example: Using Spliterator to iterate a list

```
import java.util.*;
import java.util.function.Consumer;

public class SpliteratorExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("Java", "Python", "C++", "JavaScript");

        Spliterator<String> spliterator = list.spliterator();

        // tryAdvance: process one element at a time
        System.out.println("Using tryAdvance:");
        while (spliterator.tryAdvance(System.out::println)) {
            // prints each element
        }

        // Reset spliterator
        spliterator = list.spliterator();

        // forEachRemaining: process all remaining elements at once
        System.out.println("\nUsing forEachRemaining:");
        spliterator.forEachRemaining(System.out::println);
    }
}
```

Example: Using trySplit to split workload

```
Spliterator<String> spliterator1 = list.spliterator();
Spliterator<String> spliterator2 = spliterator1.trySplit();

System.out.println("Spliterator1:");
spliterator1.forEachRemaining(System.out::println);

System.out.println("Spliterator2:");
spliterator2.forEachRemaining(System.out::println);
```

Summary

Concept	Explanation
Spliterator	Advanced iterator that supports splitting and parallelism
`tryAdvance`	Process next element if available
`forEachRemaining`	Process all remaining elements
`trySplit`	Split Spliterator into two parts to enable parallelism
Use case	Parallel processing, streams

In Java, `Iterator`, `ListIterator`, and `Enumeration` are interfaces used for traversing collections, but they differ in functionality, usage, and compatibility with Java collections. Here's a comparison:

◆ 1. Enumeration

- **Legacy interface** (from Java 1.0).
- Used with legacy classes like `Vector`, `Hashtable`.
- **Read-only**: can only traverse (read) elements.
- **No support for removal or bidirectional traversal.**

Key methods:

```
boolean hasMoreElements()
```

```
E nextElement()
```

Example:

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

◆ 2. Iterator

- Introduced in **Java 1.2** with the **Collection Framework**.
- Can be used with all collections (`ArrayList`, `HashSet`, etc.).
- Allows **removal of elements** during iteration.

Key methods:

```
boolean hasNext()
E next()
void remove()  // optional
```

Example:

```
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    String str = it.next();
    if (str.equals("removeMe")) {
        it.remove();
    }
}
```

◆ 3. ListIterator

- Subinterface of `Iterator`, added in Java 1.2.
- Works **only with List implementations** (`ArrayList`, `LinkedList`, etc.).
- Allows **bidirectional traversal** and **modification** (add, remove, set) of elements.

Key methods:

```
boolean hasNext()
E next()
boolean hasPrevious()
E previous()
int nextIndex()
int previousIndex()
void remove()
void set(E e)
void add(E e)
```

Example:

```
ListIterator<String> lit = list.listIterator();
while (lit.hasNext()) {
    String str = lit.next();
    if (str.equals("changeMe")) {
        lit.set("changed");
    }
}
```

🔄 Comparison Table

Feature	Enumeration	Iterator	ListIterator
Legacy / Modern	Legacy (pre-1.2)	Modern (since 1.2)	Modern (since 1.2)
Collection support	Legacy (`Vector`, `Hashtable`)	All `Collection` types	Only `List` types
Direction	Forward only	Forward only	Bidirectional
Element removal	❌ No	✅ Yes	✅ Yes
Element update/addition	❌ No	❌ No	✅ Yes (`set`, `add`)

Feature	Enumeration	Iterator	ListIterator
Fail-fast behavior	✗ No	✓ Yes	✓ Yes
Interface type	Interface	Interface	Interface (extends Iterator)

✓ When to Use What

- **Enumeration**: Only when working with legacy code (`Vector`, `Hashtable`).
- **Iterator**: For general-purpose, forward-only traversal of modern collections.
- **ListIterator**: When you need to:
  - Traverse in both directions.
  - Modify the list during iteration.
  - Get element indexes while iterating.

In Java, `parallelStream()` is a method used to **process stream elements in parallel** using multiple threads, leveraging **ForkJoinPool.commonPool** for performance gains in large data sets or CPU-intensive operations.

◆ What is `parallelStream()`?

- Available from **Java 8**.
- A parallel version of `stream()`.
- Automatically partitions the data and processes in **multiple threads**.
- Used with **collections** (like `List`, `Set`) and **Stream API**.
- Not always faster — use with **CPU-bound tasks** and **large datasets**.

✓ Key Characteristics

Feature	`stream()`	`parallelStream()`
Processing	Sequential	Parallel (multi-threaded)
Threading	Single-threaded	ForkJoinPool (multi)
Order preservation	Yes (mostly)	Not guaranteed
Performance	Predictable	May improve (not always)

📃 Examples with Unique Class Names

□ 1. Simple Parallel Stream Example

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

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

        names.parallelStream().forEach(name -> {
            System.out.println(Thread.currentThread().getName() + " - " + name);
        });
    }
}
```

📝 **Output may vary** (different threads print different elements).

□ 2. Parallel Sum with `reduce()`

```
import java.util.stream.IntStream;
```

```
public class ParallelExampleTwo {
    public static void main(String[] args) {
        int sum = IntStream.rangeClosed(1, 100)
                           .parallel()
                           .reduce(0, Integer::sum);

        System.out.println("Sum from 1 to 100 = " + sum);
    }
}
```

✅ Good use case — **CPU-bound numeric task**.

□ 3. Performance Comparison

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class ParallelExampleThree {
    public static void main(String[] args) {
        List<Integer> list = IntStream.rangeClosed(1, 1000000)
                                      .boxed()
                                      .collect(Collectors.toList());

        long startSequential = System.currentTimeMillis();
        list.stream().map(i -> i * 2).collect(Collectors.toList());
        long endSequential = System.currentTimeMillis();

        long startParallel = System.currentTimeMillis();
        list.parallelStream().map(i -> i * 2).collect(Collectors.toList());
        long endParallel = System.currentTimeMillis();

        System.out.println("Sequential time: " + (endSequential - startSequential) +
"ms");
        System.out.println("Parallel time: " + (endParallel - startParallel) + "ms");
    }
}
```

📊 **May show parallel stream is faster**, depending on CPU and load.

□ 4. Using `filter()` and `collect()`

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class ParallelExampleFour {
    public static void main(String[] args) {
        List<Integer> evenNumbers = IntStream.rangeClosed(1, 100)
                                             .parallel()
                                             .filter(i -> i % 2 == 0)
                                             .boxed()
                                             .collect(Collectors.toList());

        System.out.println("Even numbers: " + evenNumbers);
    }
}
```

✅ Demonstrates combining multiple stream operations in parallel.

□ 5. Avoiding Race Conditions

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

```
public class ParallelExampleFive {
    public static void main(String[] args) {
        List<Integer> unsafeList = new ArrayList<>();

        // This causes race condition
        IntStream.rangeClosed(1, 1000).parallel().forEach(i -> unsafeList.add(i));

        System.out.println("Size (should be 1000): " + unsafeList.size());
    }
}
```

✗ `ArrayList` is **not thread-safe**. You'll get wrong results.

✓ Fix with thread-safe collection:

```
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
import java.util.stream.IntStream;

public class ParallelExampleSix {
    public static void main(String[] args) {
        List<Integer> safeList = Collections.synchronizedList(new ArrayList<>());

        IntStream.rangeClosed(1, 1000).parallel().forEach(i -> safeList.add(i));

        System.out.println("Size (safe): " + safeList.size());
    }
}
```

⚠ When Not to Use `parallelStream()`

- Small datasets — overhead > benefit.
- When thread safety is hard to manage.
- When result **order matters** (use `.forEachOrdered()` if needed).
- When working with I/O operations (can increase contention).

Sure! Let's dive into `**forEachOrdered()**` in Java Streams, with **detailed explanation**, **comparison**, and **examples using unique class names**, just like we did for `parallelStream()`.

🔹 What is `forEachOrdered()`?

`forEachOrdered()` is a terminal operation in Java Stream API that **maintains the encounter order** of elements **even when the stream is parallel**.

✓ Key Points

Feature	`forEach()`	`forEachOrdered()`
Stream type	Sequential or Parallel	Sequential or Parallel
Order guarantee	✗ No (especially in parallel)	✓ Yes (preserves order)
Performance	Faster in parallel	Slightly slower (due to ordering)
Use case	When order doesn't matter	When order **must** be preserved

---

📃 Examples with Unique Class Names

□ 1. Simple Parallel with `forEach()` (unordered)

```
import java.util.Arrays;
```

```
import java.util.List;

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

        names.parallelStream().forEach(name -> {
            System.out.println(Thread.currentThread().getName() + " - " + name);
        });
    }
}
```

🔁 Output **may be unordered**, e.g.:

```
ForkJoinPool.commonPool-worker-3 - Charlie
main - Alice
ForkJoinPool.commonPool-worker-5 - Bob
...
```

□ 2. Using `forEachOrdered()` to Preserve Order

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

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

        names.parallelStream().forEachOrdered(name -> {
            System.out.println(Thread.currentThread().getName() + " - " + name);
        });
    }
}
```

📌 Output **is always ordered**:

```
main - Alice
main - Bob
main - Charlie
main - David
main - Eve
```

□ 3. Compare `forEach` vs `forEachOrdered`

```
import java.util.stream.IntStream;

public class ForEachExampleThree {
    public static void main(String[] args) {
        System.out.println("Using forEach:");
        IntStream.range(1, 10).parallel().forEach(i -> System.out.print(i + " "));

        System.out.println("\nUsing forEachOrdered:");
        IntStream.range(1, 10).parallel().forEachOrdered(i -> System.out.print(i + " "));
    }
}
```

□ Output (approximate):

```
Using forEach:
3 1 2 5 4 6 7 8 9
Using forEachOrdered:
1 2 3 4 5 6 7 8 9
```

□ 4. Realistic Case: Printing Sorted Data

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

public class ForEachExampleFour {
```

```
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(9, 5, 3, 7, 1);

        numbers.stream()
               .sorted()
               .parallel()
               .forEachOrdered(num -> System.out.print(num + " "));
    }
}
```

✅ Output will **always be sorted**:

```
1 3 5 7 9
```

🔹 5. Using forEachOrdered with Custom Objects

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

class Product {
    int id;
    String name;
    Product(int id, String name) {
        this.id = id;
        this.name = name;
    }
    public String toString() {
        return id + "-" + name;
    }
}

public class ForEachExampleFive {
    public static void main(String[] args) {
        List<Product> products = Arrays.asList(
            new Product(3, "Pen"),
            new Product(1, "Notebook"),
            new Product(2, "Pencil")
        );

        products.stream()
                .sorted((p1, p2) -> Integer.compare(p1.id, p2.id))
                .parallel()
                .forEachOrdered(p -> System.out.println(p));
    }
}
```

✅ Output maintains **product order by ID**:

```
1-Notebook
2-Pencil
3-Pen
```

⚠️ When to Use `forEachOrdered()`

Use Case	Recommendation
Need **guaranteed order** in output (e.g., reports, sorted data)	✅ Yes
Order **does not matter** and you want **max performance**	❌ Use `forEach()` instead
Writing to a file/DB where order is crucial	✅ Use it
Printing for logs or debugging with sequence	✅ Use it

✅ Summary

- `forEachOrdered()` = ordered and deterministic.

- Slight performance cost in **parallel streams**.
- Very useful when **order matters**, like sorted output or stable iteration.

In Java, the `**Comparable**` interface is used to define the **natural ordering** of objects. It allows objects of a class to be **compared to each other**, typically for sorting.

🔷 Comparable Interface Overview

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

✔ Purpose:

- Defines **default sort order** for a class.
- Commonly used with `Collections.sort()` and `Arrays.sort()`.

🔹 Method: `compareTo(T o)`

- Returns:
  - `0` if this object is **equal** to `o`.
  - A **positive number** if this object is **greater** than `o`.
  - A **negative number** if this object is **less** than `o`.

❓ Example: Custom Class with Comparable

```
import java.util.*;

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

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

    // Natural order by id
    @Override
    public int compareTo(Student other) {
        return Integer.compare(this.id, other.id);
    }

    @Override
    public String toString() {
        return id + " - " + name;
    }
}

public class ComparableExampleOne {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(3, "Alice"));
        students.add(new Student(1, "Bob"));
        students.add(new Student(2, "Charlie"));

        Collections.sort(students);  // uses compareTo()

        students.forEach(System.out::println);
    }
}
```

✔ Output:

```
1 - Bob
2 - Charlie
3 - Alice
```

❓ Why Use Comparable?

- To define **natural/default sorting**.
- Works well with Java's sorting algorithms.
- Ensures that classes can be consistently ordered.

🔄 Comparable vs Comparator

Feature	Comparable	Comparator
Location	In the **class itself**	In a **separate class or lambda**
Method	`compareTo(T o)`	`compare(T o1, T o2)`
Use case	Natural/default order	Custom or multiple sort orders
Flexibility	Less flexible	Very flexible (multiple sort logics)
Example use	`Collections.sort(list)`	`list.sort(comparator)`

❓ Bonus: Sort by Name with Comparator (not Comparable)

```
import java.util.*;

class Student {
    int id;
    String name;

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

    public String toString() {
        return id + " - " + name;
    }
}

public class ComparatorExample {
    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
            new Student(3, "Charlie"),
            new Student(1, "Alice"),
            new Student(2, "Bob")
        );

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

        list.forEach(System.out::println);
    }
}
```

✅ Summary

- Use `Comparable` when a class has a **default sort order** (like sorting Students by ID).
- Implement the `compareTo()` method carefully — avoid inconsistent or null-unsafe comparisons.
- For **custom or multiple sort criteria**, prefer `Comparator`.

Great! Let's explore **edge cases** in a `Comparable` implementation — especially handling:

1. `null` values

2. Case sensitivity in `String` fields
3. Mixed-type or inconsistent comparison logic
4. Ensuring `compareTo` is **consistent with `equals()`**

✅ Full Example with Edge Case Handling

We'll write a `Student` class with `id` (nullable), `name` (nullable), and implement `Comparable<Student>` carefully.

```
import java.util.*;

class Student implements Comparable<Student> {
    Integer id;
    String name;

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

    @Override
    public int compareTo(Student other) {
        // Edge Case 1: Null checks for 'id'
        if (this.id == null && other.id == null) return 0;
        if (this.id == null) return -1;
        if (other.id == null) return 1;

        // Edge Case 2: Fallback to name if IDs are equal
        int idCompare = this.id.compareTo(other.id);
        if (idCompare != 0) return idCompare;

        // Edge Case 3: Null checks for 'name'
        if (this.name == null && other.name == null) return 0;
        if (this.name == null) return -1;
        if (other.name == null) return 1;

        // Edge Case 4: Case-insensitive name comparison
        return this.name.compareToIgnoreCase(other.name);
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Student)) return false;
        Student other = (Student) obj;
        return Objects.equals(id, other.id) && Objects.equals(name, other.name);
    }

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

    @Override
    public String toString() {
        return (id != null ? id : "null") + " - " + (name != null ? name : "null");
    }
}
```

🧪 Usage:

```
public class ComparableEdgeCaseExample {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
```

```
            new Student(3, "Charlie"),
            new Student(null, "Eve"),
            new Student(1, null),
            new Student(2, "alice"),
            new Student(2, "Alice"),   // Same ID, test name case-insensitivity
            new Student(null, null)
        );

        Collections.sort(students);

        students.forEach(System.out::println);
    }
}
```

✅ Output (Expected):

```
null - null
null - Eve
1 - null
2 - alice
2 - Alice
3 - Charlie
```

🔍 Highlights:

- `null id` comes first.
- If `id` is equal, fallback to name comparison.
- Handles `null name` safely.
- Uses `compareToIgnoreCase()` for consistent sorting.

✅ Best Practices When Implementing Comparable

1. **Always handle `null`** if your fields can be null.
2. Avoid mixing fields of unrelated types in `compareTo`.
3. Keep `compareTo` consistent with `equals()`.
4. Be cautious of side effects or expensive operations in `compareTo`.

Absolutely! Let's now explore the `**Comparator**` approach in Java and how to handle **edge cases** like `null`, case-insensitive sorting, and multi-level comparison — all cleanly and flexibly using `Comparator`.

🔽 What is `Comparator`?

The `Comparator<T>` interface allows you to define **custom sort orders outside** the class you're sorting. Ideal when:

- You want **multiple sorting strategies**.
- You **can't or don't want** to modify the class (unlike `Comparable`).

✅ Comparator Syntax (Java 8+)

```
Comparator<Student> comparator = Comparator.comparing(Student::getId);
```

Can chain multiple comparators with `thenComparing()`, handle nulls with `nullsFirst()` or `nullsLast()`.

🧠 Example with Edge Case Handling

Let's build a **`Student` class** with `id` and `name`, where:

- `id` may be `null`
- `name` may be `null`
- Sorting is **by `id` ascending**, then **name (case-insensitive)**

✅ Step 1: Define the Class

```
class Student {
    private Integer id;
    private String name;

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

    public Integer getId() { return id; }
    public String getName() { return name; }

    @Override
    public String toString() {
        return (id != null ? id : "null") + " - " + (name != null ? name : "null");
    }
}
```

✅ Step 2: Define the Comparator with Edge Case Handling

```
import java.util.*;

public class ComparatorEdgeCaseExample {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
            new Student(3, "Charlie"),
            new Student(null, "Eve"),
            new Student(1, null),
            new Student(2, "alice"),
            new Student(2, "Alice"),
            new Student(null, null)
        );

        Comparator<Student> studentComparator = Comparator
            .comparing(Student::getId, Comparator.nullsFirst(Integer::compareTo)) //
nulls first
            .thenComparing(Student::getName,
Comparator.nullsLast(String.CASE_INSENSITIVE_ORDER)); // nulls last

        students.sort(studentComparator);

        students.forEach(System.out::println);
    }
}
```

✅ Output (Expected):

```
null - Eve
null - null
1 - null
2 - alice
2 - Alice
3 - Charlie
```

🔍 Explanation of Key Parts

Code	Purpose
`Comparator.nullsFirst(...)`	Puts `null` IDs first
`Integer::compareTo`	Natural order of Integer

Code	Purpose
`Comparator.nullsLast(...)`	Moves null names to end if IDs are equal
`String.CASE_INSENSITIVE_ORDER`	Ensures `"Alice"` and `"alice"` are treated equally

🔁 Alternate Sort: Nulls Last

```
Comparator<Student> altComparator = Comparator
    .comparing(Student::getId, Comparator.nullsLast(Integer::compareTo))
    .thenComparing(Student::getName,
Comparator.nullsFirst(String.CASE_INSENSITIVE_ORDER));
```

This would push null IDs to the end.

✅ Summary

Feature	How it's handled
`null` IDs	`Comparator.nullsFirst()` or `nullsLast()`
`null` names	`nullsLast()` after `thenComparing()`
Case-insensitive	`String.CASE_INSENSITIVE_ORDER`
Multi-level sort	Chain using `thenComparing()`

Absolutely! Let's dive into **method references** in Java with full detail and many **unique class examples**. Method references make code **shorter, cleaner, and more readable**, especially when used with **Streams, Lambdas, and Functional Interfaces**.

🔹 What is a Method Reference?

A **method reference** is a shorthand for a **lambda expression** that calls a method. It uses the `::` operator.

✅ Syntax:

```
ClassName::methodName
```

🔸 Types of Method References

Type	Syntax	Example
Static method	`ClassName::staticMethod`	`Math::abs`
Instance method of particular object	`instance::instanceMethod`	`System.out::println`
Instance method of arbitrary object	`ClassName::instanceMethod`	`String::toLowerCase`
Constructor reference	`ClassName::new`	`ArrayList::new`

🔷 Full Examples with Unique Class Names

🧮 1. Static Method Reference

```
import java.util.function.Function;

class TemperatureConverter {
    public static double celsiusToFahrenheit(double c) {
        return (c * 9/5) + 32;
    }
}

public class StaticMethodRefExample {
```

```
    public static void main(String[] args) {
        Function<Double, Double> converter = TemperatureConverter::celsiusToFahrenheit;
        System.out.println(converter.apply(25.0)); // 77.0
    }
}
```

□ 2. Instance Method of a Particular Object

```
import java.util.function.Consumer;

class MessagePrinter {
    public void print(String msg) {
        System.out.println("Message: " + msg);
    }
}

public class InstanceMethodRefExample {
    public static void main(String[] args) {
        MessagePrinter printer = new MessagePrinter();
        Consumer<String> consumer = printer::print;
        consumer.accept("Hello via method reference!");
    }
}
```

□ 3. Instance Method of Arbitrary Object (of a particular type)

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

public class ArbitraryInstanceMethodRefExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        names.forEach(String::toUpperCase); // this won't print anything unless returned
        names.forEach(System.out::println); // now print the names
    }
}
```

□ 4. Constructor Reference

```
import java.util.function.Supplier;
import java.util.function.Function;

class Product {
    String name;
    Product() {
        this.name = "Unnamed Product";
    }
    Product(String name) {
        this.name = name;
    }
    public String toString() {
        return name;
    }
}

public class ConstructorRefExample {
    public static void main(String[] args) {
        Supplier<Product> defaultProduct = Product::new; // no-arg constructor
        Product p1 = defaultProduct.get();
        System.out.println(p1); // Unnamed Product

        Function<String, Product> namedProduct = Product::new; // one-arg constructor
        Product p2 = namedProduct.apply("Laptop");
        System.out.println(p2); // Laptop
```

```
    }
}
```

□ 5. Comparator with Method Reference

```
import java.util.*;

class Book {
    String title;
    Book(String title) {
        this.title = title;
    }
    public String getTitle() {
        return title;
    }
    public String toString() {
        return title;
    }
}

public class ComparatorMethodRefExample {
    public static void main(String[] args) {
        List<Book> books = Arrays.asList(
            new Book("Zebra"),
            new Book("Apple"),
            new Book("Monkey")
        );

        // Using method reference instead of lambda
        books.sort(Comparator.comparing(Book::getTitle));
        books.forEach(System.out::println);
    }
}
```

□ 6. BiFunction with Static Method

```
import java.util.function.BiFunction;

class MathUtils {
    public static int add(int a, int b) {
        return a + b;
    }
}

public class BiFunctionMethodRefExample {
    public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> adder = MathUtils::add;
        System.out.println(adder.apply(10, 20)); // 30
    }
}
```

✓ When to Use Method References

Use method references when:

- The lambda is **just a method call**.
- You want **cleaner and shorter** syntax.
- You're using **functional interfaces** (e.g. `Function`, `Consumer`, `Comparator`).

□ Tip: Method Reference ≈ Shortcut for a lambda

```
// Lambda
list.forEach(s -> System.out.println(s));
```

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

✅ Summary

Type	Example
Static method	`Math::max`
Instance method (particular obj)	`System.out::println`
Instance method (any object)	`String::toLowerCase`
Constructor	`ArrayList::new`

Absolutely! Let's dive deep into **Dynamic Method Dispatch** in Java, explained clearly with **multiple example programs** and **unique class names**.

🔷 What is Dynamic Method Dispatch?

Dynamic Method Dispatch is Java's mechanism for resolving **overridden methods** at **runtime**, not compile time.

✅ In other words:

- You call a method on a **reference of the parent class**, but the actual method that runs is from the **child class**.
- It's the core of Java's **runtime polymorphism**.

🔶 Key Concepts:

Term	Meaning
Superclass	The base class (e.g. `Animal`)
Subclass	A class that overrides methods (e.g. `Dog`, `Cat`)
Reference type	Declared type of reference (`Animal obj = ...`)
Object type	Actual object created (`new Dog()`)

🧪 Example 1: Basic Dynamic Dispatch

```
class Vehicle {
    void start() {
        System.out.println("Starting generic vehicle...");
    }
}

class Car extends Vehicle {
    void start() {
        System.out.println("Starting car with key...");
    }
}

class Bike extends Vehicle {
    void start() {
        System.out.println("Starting bike with kick...");
    }
}

public class DynamicDispatchExampleOne {
    public static void main(String[] args) {
        Vehicle v;

        v = new Car();   // Upcasting
```

```
        v.start();       // Calls Car's start()

        v = new Bike();  // Upcasting again
        v.start();       // Calls Bike's start()
    }
}
```

✅ Output:

```
Starting car with key...
Starting bike with kick...
```

🔹 Example 2: With Additional Method in Subclass

```
class Media {
    void play() {
        System.out.println("Playing media...");
    }
}

class Music extends Media {
    void play() {
        System.out.println("Playing music track...");
    }

    void download() {
        System.out.println("Downloading music...");
    }
}

public class DynamicDispatchExampleTwo {
    public static void main(String[] args) {
        Media m = new Music(); // Upcasting
        m.play();              // Music's play() is called

        // m.download(); ❌ Not allowed: reference is type Media
    }
}
```

🔸 The `download()` method can't be called even though the object is `Music`, because the **reference is type `Media`.**

🔹 Example 3: With Parameters and Overloading

```
class PaymentProcessor {
    void process(int amount) {
        System.out.println("Processing generic payment: $" + amount);
    }
}

class CreditCardPayment extends PaymentProcessor {
    void process(int amount) {
        System.out.println("Processing credit card payment: $" + amount);
    }
}

public class DynamicDispatchExampleThree {
    public static void main(String[] args) {
        PaymentProcessor p = new CreditCardPayment();
        p.process(500); // CreditCardPayment's process() is called
    }
}
```

🔹 Example 4: Hierarchy Chain

```
class Employee {
    void work() {
        System.out.println("Working generally...");
    }
}

class Developer extends Employee {
    void work() {
        System.out.println("Writing code...");
    }
}

class Manager extends Developer {
    void work() {
        System.out.println("Managing developers...");
    }
}

public class DynamicDispatchExampleFour {
    public static void main(String[] args) {
        Employee e;

        e = new Developer(); // Runtime picks Developer's work()
        e.work();

        e = new Manager();   // Runtime picks Manager's work()
        e.work();
    }
}
```

🔹 Summary: When Does Dynamic Dispatch Apply?

Feature	Supported	Notes
Overridden methods	✅ Yes	Chosen at **runtime**
Overloaded methods	❌ No	Chosen at **compile time**
Variables	❌ No	Accessed based on reference type
Constructors	❌ No	Not inherited or overridden

🔹 Bonus: Dispatch with Abstract Class

```
abstract class Shape {
    abstract void draw();
}

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

class Square extends Shape {
    void draw() {
        System.out.println("Drawing a Square.");
    }
}

public class DynamicDispatchExampleFive {
    public static void main(String[] args) {
```

```
        Shape shape;

        shape = new Circle();
        shape.draw(); // Circle's draw()

        shape = new Square();
        shape.draw(); // Square's draw()
    }
}
```

✅ Key Takeaways

- Dynamic dispatch is Java's **runtime polymorphism**.
- The object's actual type determines the method executed — **not** the reference type.
- Only **overridden methods** are resolved at runtime — **not variables, static, or overloaded methods**.

Great! Let's comprehensively cover **method overriding** and **method overloading** in Java — specifically focusing on **non-access modifiers** (like `static`, `final`, `abstract`, `synchronized`, `native`, `strictfp`, etc.) — and how each one behaves in **Java SE 8 to 21** (no breaking changes in these versions regarding these rules).

◆ 1. Method Overriding — Non-Access Modifiers Rules

Method overriding = same **signature**, subclass method replaces superclass method.

✅ What's allowed & what's not?

Modifier	Can be overridden?	Notes
`static`	❌ No	It's **hidden**, not overridden (called method hiding)
`final`	❌ No	Cannot override a `final` method
`abstract`	✅ Must override	Subclass **must** implement it unless abstract
`synchronized`	✅ Yes	Can override, may or may not retain `synchronized`
`native`	✅ Yes	Can override with Java body (and vice versa)
`strictfp`	✅ Yes	Optional in override
`default` (interface)	✅ Yes	Can override default method in implementing class
`static final`	❌ No	Cannot be overridden or hidden; `final` blocks it
`private`	❌ No	Not visible in subclass; not inherited, hence not overridden

---

✅ Example Table for Overriding

```
class SuperClass {
    final void finalMethod() { }          // ❌ Cannot override
    static void staticMethod() { }        // ❌ Hidden only
    void normalMethod() { }               // ✅ Can override
    synchronized void syncMethod() { }    // ✅ Can override, sync optional
    strictfp void strictMethod() { }      // ✅ Optional to override with/without strictfp
    native void nativeMethod();           // ✅ Can override
}
class SubClass extends SuperClass {
    // void finalMethod() {}           ❌ Compilation error
    static void staticMethod() { }       // ✅ Method hiding
```

```
    @Override void normalMethod() { }    // ✅ Valid
    @Override void syncMethod() { }      // ✅ Can omit sync
    @Override strictfp void strictMethod() { } // ✅ Valid
    @Override void nativeMethod() { }    // ✅ Can override native method with Java code
}
```

⚠ Important:

- You **cannot** override `private` methods — they are **not inherited**.
- You **can** override `protected` and `public` methods.
- **Static methods** are resolved at **compile time** (method hiding), not overridden.

◆ 2. Method Overloading — Non-Access Modifiers Rules

Method overloading = same method **name**, different **parameters** (number, types, or order).

✅ Overloading Rules (Modifiers)

Modifier	Affects overloading?	Notes
`static`	✅ Can overload	Each overload is distinct regardless of `static`
`final`	✅ Can overload	Final method can be overloaded (final only blocks override)
`synchronized`	✅ Can overload	Treated as separate methods
`native`	✅ Can overload	OK if signatures differ
`strictfp`	✅ Can overload	Treated independently
`private`	✅ Can overload	Private methods are overloadable within same class
`abstract`	❌ Not directly	Cannot overload within same class unless multiple methods declared abstract

---

✅ Overloading Examples:

```
class Payment {
    final void process() {
        System.out.println("No-arg process");
    }

    static void process(int amount) {
        System.out.println("Static process with amount: " + amount);
    }

    synchronized void process(String type) {
        System.out.println("Synchronized process with type: " + type);
    }

    private void process(double amount) {
        System.out.println("Private overloaded process");
    }
}
```

All of these are **valid overloads** — they differ by **parameter list**, not modifier.

◈ Summary Chart: Non-Access Modifier Behavior

Modifier	Overriding	Overloading	Notes
`static`	✗ hidden	✓ Yes	Resolved at compile time
`final`	✗ No	✓ Yes	Final = no override, but can overload
`abstract`	✓ Must	✗ Not directly	Only one method per signature allowed
`synchronized`	✓ Optional	✓ Yes	Has no impact on overloading
`native`	✓ Yes	✓ Yes	Native can be overridden or overloaded
`strictfp`	✓ Optional	✓ Yes	Doesn't affect resolution
`private`	✗ No	✓ Yes	Can be overloaded in same class

◆ Java 9–21 Changes?

There have been **no changes to non-access modifier rules for overriding or overloading** in JDK versions 9 through 21. These rules remain consistent.

⍰ Final Notes

- You can **overload anything** — but **override only instance methods** that are:
  - Not `private`, `static`, or `final`
  - Accessible in the subclass
- Overriding happens at **runtime**; overloading at **compile time**
- Use `@Override` to ensure correctness

Sure! Here's a clear, concise **table overview** of Java **access modifiers** with their **visibility** in different contexts (within the same class, same package, subclass, and everywhere else):

Access Modifier	Class Itself	Same Package	Subclass (different package)	Anywhere (world)
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
default (no modifier)	Yes	Yes	No	No
`private`	Yes	No	No	No

Explanation:

- **public**: Visible everywhere.
- **protected**: Visible inside the same package and in subclasses (even if subclass is in a different package).
- **default** (package-private): Visible only inside the same package.
- **private**: Visible only within the same class.

Quick summary:

Context	Accessible modifiers
Inside the same class	public, protected, default, private
Inside same package	public, protected, default
In subclass (diff package)	public, protected
Outside package (non-subclass)	public

In an **abstract class** in Java, you can use **any access modifier** (`public`, `protected`, `private`, or default) for:

◆ 1. The Class Itself

Modifier	Allowed?	Notes
`public`	✓ Yes	The abstract class is visible everywhere
default	✓ Yes	Package-private: visible only within the same package
`protected`	✗ No	Not allowed on top-level classes
`private`	✗ No	Not allowed on top-level classes

✓ So: `abstract class` can be **public** or **default** (no modifier).

✗ You **cannot declare** an abstract top-level class as `private` or `protected`.

◆ 2. Abstract Class Members (Fields & Methods)

✓Allowed Modifiers for Methods in Abstract Class:

Modifier	Allowed?	Abstract Method?	Concrete Method?	Notes
`public`	✓ Yes	✓ Yes	✓ Yes	Most common for API methods
`protected`	✓ Yes	✓ Yes	✓ Yes	For subclass-only access
default	✓ Yes	✓ Yes	✓ Yes	Visible within package
`private`	✓ Yes	✗ No	✓ Yes	Cannot be abstract, but valid for concrete methods
`final`	✓ Yes	✗ No	✓ Yes	Final methods cannot be overridden
`static`	✓ Yes	✗ No	✓ Yes	Static methods cannot be abstract
`abstract`	✓ Yes	✓ Yes	✗ No	Used only with abstract methods

✓Allowed Modifiers for Variables (Fields):

Modifier	Allowed?	Notes
`public`	✓ Yes	Visible everywhere
`protected`	✓ Yes	Visible to subclasses
default	✓ Yes	Package-private
`private`	✓ Yes	Encapsulation within the class
`final`	✓ Yes	Makes variable a constant
`static`	✓ Yes	Belongs to the class, not instances

❗ **Abstract fields are not allowed** — fields cannot be marked `abstract` in Java.

✓ Examples

```
public abstract class Animal {
    public int age;                  // allowed
    protected String name;           // allowed
    private final double weight = 5.0; // allowed

    public abstract void makeSound();   // abstract + public
    protected abstract void sleep();    // abstract + protected
    abstract void eat();                // abstract + default (package-private)

    private void internalLogic() {      // concrete + private
        System.out.println("Internal logic");
```

```
    }

    static void generalInfo() {          // static method
        System.out.println("Animals are multicellular");
    }
}
```

❌ Invalid Declarations

```
abstract private void walk();   // ❌ abstract methods can't be private
abstract static void run();     // ❌ abstract methods can't be static
```

✅ Summary

Member Type	Valid Modifiers
Abstract Class	`public`, default (no modifier)
Fields	`public`, `protected`, `private`, default, `static`, `final`
Methods	`public`, `protected`, default (for abstract); all valid for concrete except `abstract` `private/static`
Abstract Methods	`public`, `protected`, default (but NOT `private`, `static`, `final`)

Absolutely! Here's a comprehensive breakdown of **access modifiers in Java interfaces**, including what's allowed on **interfaces, their methods, and fields**, updated to **Java 21**.

🔷 1. Interface Declaration

Modifier	Allowed?	Notes
`public`	✅ Yes	Most common; accessible everywhere
default	✅ Yes	Package-private if no modifier is used
`protected`	❌ No	Not allowed on top-level interfaces
`private`	❌ No	Not allowed on top-level interfaces

✅ So: An **interface** can be `public` or default (no modifier).

❌ **Top-level interfaces cannot** be `private` or `protected`.

🔷 2. Interface Fields (Variables)

All fields in an interface are implicitly:

`public static final`

Modifier	Allowed?	Notes
`public`	✅ Yes	Explicit or implicit
`static`	✅ Yes	Always static
`final`	✅ Yes	Always final (constants)
`private`	❌ No	Not allowed (JDK 8+)
`protected`	❌ No	Not allowed
default	❌ No	Treated as public implicitly

⊘ You **cannot have instance variables** in interfaces.

⊘ You **cannot make variables mutable** (i.e., not `final`).

```
interface Config {
    int MAX = 100; // implicit: public static final
}
```

🔷 3. Interface Methods

Java 8+ allows different types of methods in interfaces:

Method Type	Allowed Modifiers	Overridable?	Notes
Abstract method	`public` (implicit)	✅ Yes	Default until Java 7
`default` method	`default`, `public` (implicit)	✅ Yes	Java 8+
`static` method	`static`, `public` (implicit)	❌ No	Java 8+
`private` method	`private`, `static` or instance	❌ No	Java 9+ internal helper methods only

---

✅ Valid Examples

```
interface Vehicle {

    // Implicit: public abstract
    void start();

    // Default method (Java 8+)
    default void stop() {
        System.out.println("Vehicle stopped");
    }

    // Static method (Java 8+)
    static void info() {
        System.out.println("Vehicle interface");
    }

    // Private helper method (Java 9+)
    private void internalCheck() {
        System.out.println("Internal check...");
    }
}
```

❌ Invalid Examples

```
interface WrongInterface {
    protected void test(); // ❌ Cannot be protected

    private int x = 5;     // ❌ Fields cannot be private

    final void doIt();     // ❌ final not allowed on abstract/interface methods

    abstract default void tryIt(); // ❌ Cannot combine abstract and default
}
```

✅ Summary Table

Interface Member	Implicit Modifiers	Allowed Modifiers	Not Allowed
Interface (top-level)	—	`public`, default (package-private)	`private`, `protected`
Fields	`public static final`	Only those (must be constants)	`private`, mutable, instance vars

Interface Member	Implicit Modifiers	Allowed Modifiers	Not Allowed
Abstract Methods	`public abstract`	`public`	`private`, `protected`, `final`
Default Methods	—	`default`, `public`	`abstract`, `final`, `private`
Static Methods	—	`static`, `public`	`protected`, `private` (as API)
Private Methods	—	`private`, optionally `static`	Only usable within interface

---

Final Notes (Java 8–21):

Java Version	Feature
Java 8	Default & static methods in interfaces
Java 9	Private methods in interfaces
Java 10–21	No changes to interface modifiers

Here's a **short and clear explanation** of each Java tool:

◆ `javac` — Java Compiler

- **What it does:** Compiles `.java` source files into `.class` bytecode.
- **Usage:**
- `javac MyClass.java`

◆ `java` — Java Application Launcher

- **What it does:** Runs the compiled `.class` files using the Java Virtual Machine (JVM).
- **Usage:**
- `java MyClass`

◆ `javadoc` — Documentation Generator

- **What it does:** Generates HTML documentation from Java source code comments (`/** ... */`).
- **Usage:**
- `javadoc MyClass.java`

◆ `jar` — Java Archive Tool

- **What it does:** Packages multiple `.class` files and resources into a `.jar` (Java Archive) file.
- **Usage:**
- `jar cf myapp.jar MyClass.class`

◆ `jdb` — Java Debugger

- **What it does:** Debugs Java programs at runtime (step-by-step execution, breakpoints, etc.).
- **Usage:**
- `jdb MyClass`

◆ `jps` — Java Process Status

- **What it does:** Lists all Java processes currently running on the machine (like `ps` for Java).
- **Usage:**
- `jps`

◆ `JIT` — Just-In-Time Compiler (part of JVM)

- **What it does:** Improves performance by compiling bytecode to native machine code **at runtime**.
- **Usage:** Automatically used by JVM; no manual command.

The **SOLID principles** are five key design rules in **Object-Oriented Programming** that help make software **more maintainable, flexible, and understandable**. Let's explain each one **in simple words**, with **real-world-like Java examples** using **unique class names**.

S – Single Responsibility Principle (SRP)

A class should have **only one reason to change** — it should **do one thing only**.

◆ Simple Idea:

Don't mix responsibilities (e.g., printing + saving in one class = bad).

✅ Java Example:

```
// Does ONLY user data storage
class UserDataManager {
    public void saveUser(String name) {
        System.out.println("User saved: " + name);
    }
}

// Does ONLY user display
class UserPrinter {
    public void printUser(String name) {
        System.out.println("User: " + name);
    }
}
```

O – Open/Closed Principle (OCP)

Software should be **open for extension**, but **closed for modification**.

◆ Simple Idea:

You should be able to add new features **without changing existing code**.

✅ Java Example:

```
abstract class Invoice {
    public abstract double calculateTotal();
}

class RetailInvoice extends Invoice {
    public double calculateTotal() {
        return 100;
    }
}

class WholesaleInvoice extends Invoice {
    public double calculateTotal() {
        return 80;
    }
}

class InvoicePrinter {
    public void printInvoice(Invoice invoice) {
        System.out.println("Total: " + invoice.calculateTotal());
    }
}
```

✅ You can add a `DigitalInvoice` later **without modifying** `InvoicePrinter`.

🔷 L – Liskov Substitution Principle (LSP)

Subclasses should be able to **replace their parent** classes **without breaking the app**.

◆ Simple Idea:

If `Bird` can fly, then `Penguin` shouldn't extend `Bird` (because it can't fly).

✅ Java Example:

```
class Engine {
    public void start() {
        System.out.println("Engine starts");
    }
}

class Car extends Engine {
    public void drive() {
        start(); // Safe
        System.out.println("Car is driving");
    }
}
```

⊘ But if a subclass **changes the behavior** so much it **breaks** expectations — LSP is violated.

🔷 I – Interface Segregation Principle (ISP)

Don't force a class to implement **methods it doesn't use**.

◆ Simple Idea:

Better to split big interfaces into smaller ones.

✅ Java Example:

```
interface Printer {
    void print();
}

interface Scanner {
    void scan();
}

class OfficePrinter implements Printer, Scanner {
    public void print() {
        System.out.println("Printing document");
    }
    public void scan() {
        System.out.println("Scanning document");
    }
}

class BasicPrinter implements Printer {
    public void print() {
        System.out.println("Printing only");
    }
}
```

✅ Each class only implements what it **actually needs**.

🔷 D – Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules — both should depend on **abstractions**.

◆ Simple Idea:

Don't hardcode dependencies — **depend on interfaces**.

✅ Java Example:

```
interface NotificationService {
    void send(String message);
```

```
}

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

class OrderProcessor {
    private final NotificationService service;

    public OrderProcessor(NotificationService service) {
        this.service = service;
    }

    public void processOrder() {
        System.out.println("Order processed");
        service.send("Order confirmation");
    }
}

// Usage
class AppLauncher {
    public static void main(String[] args) {
        NotificationService email = new EmailNotification();
        OrderProcessor order = new OrderProcessor(email);
        order.processOrder();
    }
}
```

✅ The `OrderProcessor` **depends on abstraction** (`NotificationService`) not concrete class.

✅ Summary Table

Principle	Focus	Good Practice
SRP	One job per class	One reason to change
OCP	Open for extension	Extend via subclassing, don't modify directly
LSP	Respect inheritance rules	Subclasses shouldn't break parent behavior
ISP	Small, focused interfaces	No "fat" interfaces
DIP	Use abstraction	High-level code depends on interfaces, not classes

---

Rules for Abstract Class in Java

1. **Definition:**
   - An abstract class is a class declared with the `abstract` keyword.
   - It **cannot be instantiated** directly (you cannot do `new AbstractClass()`).
2. **Abstract Methods:**
   - Can contain **abstract methods** (methods without a body).
   - Abstract methods **must be implemented** by subclasses (unless the subclass is also abstract).
   - Example:
   - `abstract void doSomething();`
3. **Concrete Methods:**
   - Can have **regular (concrete) methods** with implementations.
   - Subclasses may override these methods.
4. **Constructors:**
   - Can have constructors.
   - Constructors are called when a subclass instance is created.

5. **Fields and Variables:**
   - Can have instance variables and static variables.
   - Can have all kinds of variables (private, protected, public).
6. **Modifiers allowed:**
   - Abstract classes can have **any access modifier** (public, protected, default, private) on methods and fields **except abstract methods cannot be private**.
   - Abstract methods **cannot be `private` or `static` or `final`**.
   - Class itself can be **public or default** (no modifier).
7. **Inheritance:**
   - Abstract classes are intended to be **extended**.
   - Subclass must implement all abstract methods, unless it is also abstract.
8. **Multiple Inheritance:**
   - Java does **not allow multiple inheritance** of classes, abstract or not.
   - Abstract classes can implement interfaces.

Quick example:

```
public abstract class Vehicle {
    protected String brand;

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

    // Abstract method (no body)
    public abstract void startEngine();

    // Concrete method
    public void printBrand() {
        System.out.println("Brand: " + brand);
    }
}
```

Key takeaway:

- **You cannot create instances of abstract classes.**
- **You must subclass them and implement their abstract methods (unless the subclass is abstract too).**

Great questions! Let's break them down:

What does `synchronized` mean in Java?

- The `synchronized` keyword in Java is used to **control access to a block of code or method so that only one thread can execute it at a time**.
- It **prevents multiple threads from interfering with each other** when accessing shared resources, thus avoiding **race conditions**.
- When a method or block is marked `synchronized`, the thread must acquire the **lock (monitor)** on the specified object before executing the code.
- Example:

```
public synchronized void increment() {
    count++;
}
```

Or for a block:

```
synchronized(this) {
    // critical section
}
```

What does thread safe mean in Java?

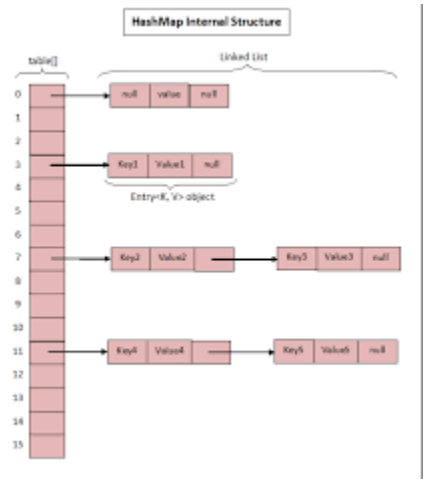
- A class or method is **thread safe** if it **functions correctly when accessed by multiple threads simultaneously**, without causing data corruption or unexpected behavior.
- Thread safety means **the internal state is protected from race conditions**, no matter how many threads use it concurrently.
- Thread-safe code may use various techniques to ensure safety, such as:
  - `synchronized` keyword
  - Locks (e.g., `ReentrantLock`)
  - Atomic variables (`AtomicInteger`)
  - Immutable objects
  - Concurrent collections (`ConcurrentHashMap`)

Comparison: `synchronized` vs. Thread Safe

Aspect	`synchronized`	Thread Safe
Definition	A keyword used to lock a block or method to one thread at a time.	A property of code that guarantees correctness under concurrent access.
Scope	Mechanism for controlling access to critical sections.	Overall characteristic of code or objects.
Implementation	Uses intrinsic locks (monitors).	Can use `synchronized`, locks, atomic classes, or immutable design.
Usage	Applied to methods or code blocks.	Achieved by design and techniques ensuring safe concurrency.
Guarantees	Ensures mutual exclusion in specific code parts.	Ensures no data corruption or inconsistent state in multi-threaded usage.
Overhead	Can cause blocking and performance hits if overused.	Depends on implementation; thread-safe doesn't always mean heavy locking.

Summary

- **`synchronized` is a tool** to help make code thread-safe by preventing concurrent access.
- **Thread safe is a goal/state** meaning code behaves correctly with concurrent threads.
- You can make code thread-safe with `synchronized`, but also with other means (locks, atomics, immutability).
- `synchronized` alone does not guarantee thread safety if used improperly.



Certainly! Let's delve into the internal workings of `HashMap`, `Hashtable`, and `ConcurrentHashMap` in Java, highlighting their structures, behaviors, and differences.

---

1. HashMap

Internal Structure

- **Buckets**: `HashMap` uses an array of buckets (or bins), where each bucket is a linked list (or a balanced tree in Java 8 and later) to handle collisions.(FreeCodeCamp)
- **Hashing**: The `hashCode()` of the key determines the bucket index. If multiple keys hash to the same index, they are stored in a linked list or tree structure within that bucket.(GeeksforGeeks)
- **Resizing**: When the number of entries exceeds the product of the load factor and the current capacity, the `HashMap` resizes by doubling the number of buckets and rehashing the entries.

Thread Safety

- `HashMap` is **not thread-safe**. Concurrent access by multiple threads can lead to data inconsistency and corruption.

---

2. Hashtable

Internal Structure

- **Buckets**: Similar to `HashMap`, `Hashtable` uses an array of buckets, each containing a linked list of entries.(FreeCodeCamp)
- **Hashing**: The `hashCode()` of the key is used to determine the bucket index. Collisions are handled by chaining entries in the same bucket.(DigitalOcean)
- **Resizing**: `Hashtable` resizes when the number of entries exceeds the product of the load factor and the current capacity, similar to `HashMap`.

Thread Safety

- `Hashtable` is **synchronized**, meaning it is thread-safe. However, this synchronization can lead to performance bottlenecks in multi-threaded environments.

---

3. ConcurrentHashMap

Internal Structure

- **Segments**: In Java 7 and earlier, `ConcurrentHashMap` divides the map into segments, each acting as a separate `HashMap`. Each segment can be locked independently, allowing concurrent access.(JavaInUse)
- **Buckets**: Each segment contains an array of buckets, and each bucket handles collisions similarly to `HashMap`.(Medium)
- **Java 8 and Later**: Introduced a more fine-grained locking mechanism using `synchronized` blocks and `CAS` (Compare-And-Swap) operations, allowing better concurrency without the need for segment-level locks.

Thread Safety

- `ConcurrentHashMap` allows **concurrent reads** without locking and **concurrent writes** by locking only the affected segments or buckets, providing high concurrency without compromising thread safety.(lankydan.dev)

Comparison Summary

Feature	HashMap	Hashtable	ConcurrentHashMap
Thread Safety	No	Yes	Yes
Null Keys/Values	Yes	No	No
Synchronization	Not synchronized	Synchronized	Fine-grained locking
Performance	High (non-concurrent)	Lower due to synchronization	High concurrency support
Ideal Use Case	Single-threaded applications	Legacy applications needing thread safety	Multi-threaded applications requiring high concurrency

---

For a more visual and detailed explanation, you might find the following video helpful:

How does a HashMap Work Internally? | Core Java Interview Question
Feel free to ask if you need further clarification or examples!
Certainly! Here's a comprehensive comparison of `HashMap`, `Hashtable`, and `ConcurrentHashMap` in Java, covering various aspects:

🔍 Comparison Table: `HashMap` VS `Hashtable` VS `ConcurrentHashMap`

Feature	`HashMap`	`Hashtable`	`ConcurrentHashMap`
Thread Safety	Not synchronized	Synchronized	Thread-safe with fine-grained locking
Null Keys/Values	Allows 1 null key, multiple null values	Does not allow null keys or values	Does not allow null keys or values
Performance	High in single-threaded environments	Lower due to synchronization overhead	Optimized for high concurrency
Locking Mechanism	No internal locking	Locks entire map	Locks at bucket/segment level
Iteration	Fail-fast iterator	Fail-fast enumerator	Weakly consistent iterator
Legacy	Introduced in JDK 1.2	Introduced in JDK 1.0	Introduced in JDK 1.5
Fail-Fast Support	Yes	Yes	No (fail-safe)
Insertion Order	Not guaranteed	Not guaranteed	Not guaranteed
Deletion Order	Not guaranteed	Not guaranteed	Not guaranteed
Retrieval Order	Not guaranteed	Not guaranteed	Not guaranteed
Duplicate Keys	Not allowed (overwrites existing value)	Not allowed (overwrites existing value)	Not allowed (overwrites existing value)
Duplicate Values	Allowed	Allowed	Allowed
Default Size	Initial capacity: 16, Load factor: 0.75	Initial capacity: 11, Load factor: 0.75	Initial capacity: 16, Load factor: 0.75
Use Case	Single-threaded applications, caching	Legacy systems requiring thread safety	High-concurrency applications

---

Key Takeaways:

- **Thread Safety:** `HashMap` is not thread-safe and requires external synchronization. `Hashtable` is thread-safe but has performance drawbacks due to synchronized methods. `ConcurrentHashMap` offers thread safety with better performance by allowing concurrent reads and fine-grained locking for writes.
- **Null Handling:** `HashMap` allows one null key and multiple null values. Both `Hashtable` and `ConcurrentHashMap` do not permit null keys or values to avoid ambiguity in concurrent environments.(Medium)
- **Performance:** `HashMap` provides high performance in single-threaded scenarios. `Hashtable` has lower performance due to synchronization overhead. `ConcurrentHashMap` is optimized for high-concurrency scenarios, offering better performance than `Hashtable`.
- **Iteration Behavior:** `HashMap` and `Hashtable` use fail-fast iterators, throwing `ConcurrentModificationException` if the map is structurally modified during iteration. `ConcurrentHashMap` uses weakly consistent iterators, allowing safe iteration during concurrent modifications.(Baeldung)
- **Use Cases:** `HashMap` is suitable for single-threaded applications or when external synchronization is applied. `Hashtable` is considered legacy and is generally not recommended for new development. `ConcurrentHashMap` is ideal for high-concurrency applications where thread safety is required without compromising performance.

If you need further details or examples on any of these classes, feel free to ask!
Certainly! Here's a comprehensive comparison of **`ArrayList`**, **`LinkedList`**, and **`Vector`** in Java, covering various aspects:

🔍 Comparison Table: `ArrayList` VS `LinkedList` VS `Vector`

Feature	`ArrayList`	`LinkedList`	`Vector`
Thread Safety	Not synchronized	Not synchronized	Synchronized (thread-safe)
Null Keys/Values	Allows null keys and values	Allows null keys and values	Allows null keys and values
Performance (Synchronized)	Not synchronized (faster)	Not synchronized (faster)	Synchronized (slower due to overhead)
Locking	No internal locking	No internal locking	Synchronized methods (locks entire object)
Iteration	Fail-fast iterator	Fail-fast iterator	Fail-fast iterator
Legacy	Introduced in JDK 1.2	Introduced in JDK 1.2	Introduced in JDK 1.0 (legacy)
Fail-Fast Support	Yes	Yes	Yes
Insertion Order	Maintained	Maintained	Maintained
Deletion Order	Maintained	Maintained	Maintained
Retrieval Order	Maintained	Maintained	Maintained
Duplicate Keys	Not applicable (does not store key-value pairs)	Not applicable (does not store key-value pairs)	Not applicable (does not store key-value pairs)
Duplicate Values	Allowed	Allowed	Allowed
Default Size	Initial capacity: 10, Load factor: 0.75	N/A (doubly linked list)	Initial capacity: 10, Load factor: 1.0
Use Case	Frequent retrievals, dynamic resizing	Frequent insertions/deletions, queue/stack usage	Thread-safe operations in multi-threaded environments

---

Key Takeaways:

- **Thread Safety:** `Vector` is synchronized, making it thread-safe but slower due to synchronization overhead. Both `ArrayList` and `LinkedList` are not synchronized and are faster in single-threaded scenarios.
- **Performance:** `ArrayList` and `LinkedList` offer better performance in single-threaded environments due to the lack of synchronization. `Vector`'s performance is hindered by its synchronized methods.
- **Use Cases:** Use `ArrayList` for scenarios requiring frequent retrievals and dynamic resizing. Opt for `LinkedList` when frequent insertions and deletions are needed, such as in implementing queues or stacks. Choose `Vector` when thread safety is a priority in multi-threaded environments, though consider modern alternatives like `CopyOnWriteArrayList` for better performance.

If you need further details or examples on any of these classes, feel free to ask!
Certainly! Here's a comprehensive comparison of **`HashSet`**, **`LinkedHashSet`**, and **`TreeSet`** in Java, covering various aspects:

🔍 Comparison Table: `HashSet` VS `LinkedHashSet` VS `TreeSet`

Feature	`HashSet`	`LinkedHashSet`	`TreeSet`
Thread Safety	Not synchronized	Not synchronized	Not synchronized
Null Keys/Values	Allows at most one null element	Allows at most one null element	Does not allow null elements

Feature	`HashSet`	`LinkedHashSet`	`TreeSet`
Performance (Ops)	O(1) for add, remove, contains	O(1) for add, remove, contains	O(log n) for add, remove, contains
Locking Mechanism	No internal locking	No internal locking	No internal locking
Iteration	Fail-fast iterator	Fail-fast iterator	Fail-fast iterator
Legacy	Introduced in JDK 1.2	Introduced in JDK 1.2	Introduced in JDK 1.2
Fail-Fast Support	Yes	Yes	Yes
Insertion Order	Not guaranteed	Maintains insertion order	Maintains natural or custom order
Deletion Order	Not guaranteed	Maintains insertion order	Maintains natural or custom order
Retrieval Order	Not guaranteed	Maintains insertion order	Maintains natural or custom order
Duplicate Keys	Not allowed	Not allowed	Not allowed
Duplicate Values	Allowed	Allowed	Allowed
Default Size	Initial capacity: 16, Load factor: 0.75	Initial capacity: 16, Load factor: 0.75	Initial capacity: 16, Load factor: 0.75
Use Case	General-purpose set operations	Set operations with predictable iteration order	Sorted set operations

---

Key Takeaways:

- **Thread Safety:** None of these implementations are synchronized. For thread-safe operations, consider using `CopyOnWriteArraySet` or `ConcurrentSkipListSet`.(2much2learn.com)
- **Null Handling:** `HashSet` and `LinkedHashSet` allow at most one null element, whereas `TreeSet` does not allow null elements due to the need for comparison.(Medium)
- **Performance:** `HashSet` and `LinkedHashSet` offer constant-time performance (O(1)) for basic operations, while `TreeSet` provides logarithmic-time performance (O(log n)) due to its underlying Red-Black tree structure.
- **Ordering:** `LinkedHashSet` maintains the insertion order of elements, while `TreeSet` maintains elements in their natural order or according to a specified comparator. `HashSet` does not guarantee any specific order.(GeeksforGeeks)
- **Use Cases:** Use `HashSet` for general-purpose set operations where order doesn't matter. Opt for `LinkedHashSet` when the order of insertion needs to be preserved. Choose `TreeSet` when a sorted order is required.

If you need further details or examples on any of these classes, feel free to ask!

Certainly! Here's a comprehensive comparison of **`HashMap`**, **`LinkedHashMap`**, and **`TreeMap`** in Java, covering various aspects:

🔍 Comparison Table: `HashMap` VS `LinkedHashMap` VS `TreeMap`

Feature	`HashMap`	`LinkedHashMap`	`TreeMap`
Thread Safety	Not synchronized	Not synchronized	Not synchronized
Null Keys/Values	Allows one null key and multiple null values	Allows one null key and multiple null values	Does not allow null keys; allows multiple null values

Feature	`HashMap`	`LinkedHashMap`	`TreeMap`
Performance (Ops)	O(1) for get, put, containsKey, and remove	O(1) for get, put, containsKey, and remove	O(log n) for get, put, containsKey, and remove
Locking Mechanism	No internal locking	No internal locking	No internal locking
Iteration	Fail-fast iterator	Fail-fast iterator	Fail-fast iterator
Legacy	Introduced in JDK 1.2	Introduced in JDK 1.2	Introduced in JDK 1.2
Fail-Fast Support	Yes	Yes	Yes
Insertion Order	Not guaranteed	Maintains insertion order	Not guaranteed
Deletion Order	Not guaranteed	Maintains insertion order	Not guaranteed
Retrieval Order	Not guaranteed	Maintains insertion order	Sorted according to natural order or comparator
Duplicate Keys	Not allowed	Not allowed	Not allowed
Duplicate Values	Allowed	Allowed	Allowed
Default Size	Initial capacity: 16, Load factor: 0.75	Initial capacity: 16, Load factor: 0.75	Initial capacity: 16, Load factor: 0.75
Use Case	General-purpose map operations	Map operations with predictable iteration order	Sorted map operations

---

Key Takeaways:

- **Thread Safety:** None of these implementations are synchronized. For thread-safe operations, consider using `ConcurrentHashMap`.
- **Null Handling:** `HashMap` and `LinkedHashMap` allow one null key and multiple null values. `TreeMap` does not allow null keys but permits multiple null values.
- **Performance:** `HashMap` and `LinkedHashMap` offer constant-time performance (O(1)) for basic operations, while `TreeMap` provides logarithmic-time performance (O(log n)) due to its underlying Red-Black tree structure.
- **Ordering:** `LinkedHashMap` maintains the insertion order of elements. `TreeMap` maintains elements in their natural order or according to a specified comparator. `HashMap` does not guarantee any specific order.
- **Use Cases:** Use `HashMap` for general-purpose map operations where order doesn't matter. Opt for `LinkedHashMap` when the order of insertion needs to be preserved. Choose `TreeMap` when a sorted order is required.

If you need further details or examples on any of these classes, feel free to ask!

Certainly! Here's a detailed comparison of **`CopyOnWriteArrayList`**, **`List.of()`**, **`Collections.synchronizedList()`**, and **`Collections.unmodifiableList()`** in Java, focusing on their characteristics and suitable use cases:

🔍 Comparison Table

Feature	`CopyOnWriteArrayList`	`List.of()`	`Collections.synchronizedList()`	`Collections.unmodifiableList()`
Thread Safety	✅ Yes – thread-safe via copy-on-write mechanism	❌ No – not thread-safe	✅ Yes – synchronized wrapper	❌ No – not thread-safe

Feature	`CopyOnWriteArrayList`	`List.of()`	`Collections.synchronizedList()`	`Collections.unmodifiableList()`
Mutability	✅ Mutable – supports add, remove, etc.	❌ Immutable – no modifications allowed	✅ Mutable – allows modifications	❌ Immutable – no modifications allowed
Null Elements	✅ Allowed	❌ Not allowed – throws `NullPointerException`	✅ Allowed	✅ Allowed if original list allows
Performance (Reads)	⚡ Fast – lock-free reads	⚡ Fast – optimized for read-only	🐌 Slower – synchronized reads	⚡ Fast – read-only
Performance (Writes)	🐌 Slower – entire array copied on each write	❌ Not applicable – immutable	🐌 Slower – synchronized writes	❌ Not applicable – immutable
Iterator Behavior	✅ Fail-safe – safe during concurrent modifications	✅ Fail-safe – no modifications allowed	❌ Fail-fast – requires external synchronization during iteration	✅ Fail-safe – no modifications allowed
Use Case	Ideal for concurrent scenarios with frequent reads and infrequent writes	Suitable for creating small, fixed-size, immutable lists	Useful when a thread-safe, mutable list is needed, and external synchronization is acceptable	Best for exposing read-only views of existing lists
Introduced In	Java 5	Java 9	Java 1.2	Java 1.2
Modification Support	✅ Yes – supports all standard list modifications	❌ No – throws `UnsupportedOperationException` on modification attempts	✅ Yes – supports all standard list modifications	❌ No – throws `UnsupportedOperationException` on modification attempts
Underlying Implementation	Internally creates a new copy of the array on each write operation	Returns an unmodifiable list backed by an array	Wraps an existing list with synchronized methods	Wraps an existing list to prevent modifications

---

📝 Summary

- **`CopyOnWriteArrayList`**: Designed for concurrent applications where read operations vastly outnumber write operations. It provides thread safety by creating a new copy of the underlying array on each write, allowing for lock-free reads. However, write operations can be costly due to the array copying overhead.
- **`List.of()`**: Introduced in Java 9, this method provides a convenient way to create immutable lists. These lists are fixed-size, do not allow null elements, and any attempt to modify them results in an `UnsupportedOperationException`. They are ideal for creating small, read-only collections.
- **`Collections.synchronizedList()`**: Wraps an existing list to make it thread-safe by synchronizing all its methods. While it ensures thread safety, it can lead to performance bottlenecks due to the overhead of synchronization, especially in scenarios with high contention. External synchronization is also required during iteration to prevent `ConcurrentModificationException`.
- **`Collections.unmodifiableList()`**: Provides a read-only view of an existing list. Any attempt to modify the list results in an `UnsupportedOperationException`. It's useful when you want to expose a list to clients

without allowing them to modify it. Note that if the underlying list is modified, those changes will be reflected in the unmodifiable view.

Choosing the Right List Implementation:

- Use `CopyOnWriteArrayList` when you have a multi-threaded application with frequent read operations and infrequent writes.
- Opt for `List.of()` when you need to create a small, fixed-size, immutable list quickly.
- Choose `Collections.synchronizedList()` when you need a thread-safe, mutable list and are willing to manage synchronization overhead.
- Use `Collections.unmodifiableList()` when you want to provide a read-only view of an existing list to prevent external modifications.

If you have specific scenarios or further questions about these list implementations, feel free to ask!