

★ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



A Comprehensive Journey from Java 8 to Java 21 with Code Examples of Essential API Enhancements"

Hello Friends, In this article I would like to talk about various versions of Java. I wanted to cover each and every important feature and API that has been introduced in each version of Java from Java 8 to Java 21 with coding examples to gain more insight.



Ajay Rathod · Follow

22 min read · 5 days ago



Listen



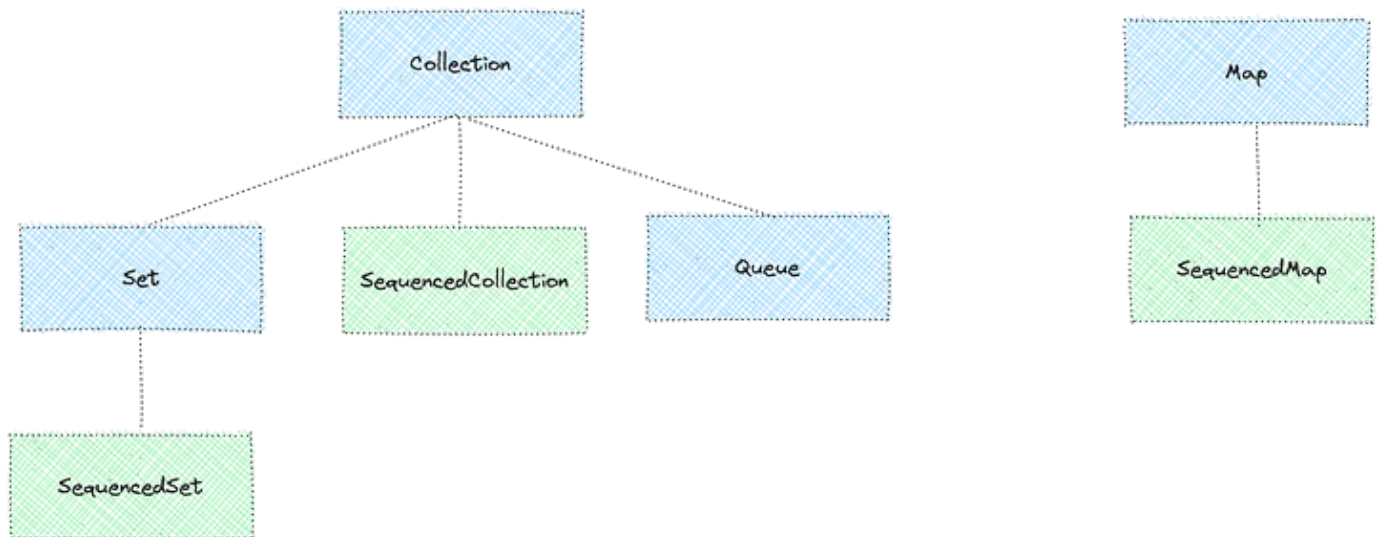
Share



More

This will be helpful to beginners as well as Java developers who are working on old version of java like Java 8 and Java 11 and want to update themselves on whats happening in Java world.

Lets dive in,



Sequenced collection from Java 21

Are you preparing for a job interview as a Java developer?

[Open in app](#)



Search



Download the sample copy here: [Guide To Clear Java Developer Interview](#)[Free Sample Copy]

[Guide To Clear Spring-Boot Microservice Interview](#)[Free Sample Copy]

Java-8

Features:

Lambda Expressions:

- Enables functional programming by allowing the use of anonymous functions.
- Concise syntax for writing functional interfaces.

Functional Interfaces:

- An interface with a single abstract method, facilitating the use of lambda expressions.
- `@FunctionalInterface` annotation to mark such interfaces.

Stream API:

- Introduces a new abstraction called Stream for processing sequences of elements.
- Supports functional-style operations on streams like `filter`, `map`, `reduce`, etc.

Method References:

- Provides a shorthand notation for lambda expressions.
- Allows referring to methods or constructors using the `::` operator.

Optional class:

- A container object that may or may not contain a non-null value.
- Helps to handle null checks more effectively and avoids `NullPointerExceptions`.

New Date and Time API:

- `java.time` package introduced for a more comprehensive and flexible API to handle dates and times.
- Solves various issues with the old `java.util.Date` and `java.util.Calendar` classes.

Default Methods:

- Allows interfaces to have method implementations.
- Helps in evolving interfaces without breaking existing implementations.

Nashorn JavaScript Engine:

- Replaces the old Rhino JavaScript engine.
- Provides better performance and improved compatibility with modern JavaScript standards.

Parallel Streams:

- Allows parallel processing of streams using the `parallel()` method.
- Enhances performance on multi-core systems for certain types of operations.

Collectors:

- Introduces a set of utility methods in the `Collectors` class for common reduction operations, such as `toList()`, `toSet()`, `joining()`, etc.

Functional Interfaces in `java.util.function` package:

- New functional interfaces like `Predicate`, `Function`, `Consumer`, and `Supplier` to support lambda expressions.

Java 9

Improved Process API:

- Java 9 introduced enhancements to the Process API, providing better control over native processes. The new `ProcessHandle` class allows developers to interact with processes and obtain information about them.

```
// Using the ProcessHandle API to get information about the current process
public class ProcessHandleExample {
    public static void main(String[] args) {
        ProcessHandle currentProcess = ProcessHandle.current();
```

```

        System.out.println("Process ID: " + currentProcess.pid());
        System.out.println("Is alive? " + currentProcess.isAlive());
    }
}

```

Collection Factory Methods:

- Java 9 added new static factory methods to the collection interfaces (List, Set, Map, etc.), making it more convenient to create immutable instances of these collections.

```

import java.util.List;

public class CollectionFactoryMethodsExample {
    public static void main(String[] args) {
        // Creating an immutable list using List.of() factory method
        List<String> colors = List.of("Red", "Green", "Blue");
        System.out.println(colors);
    }
}

```

Improved Stream API:

- The Stream API was enhanced with several new methods, such as `takeWhile`, `dropWhile`, and `ofNullable`, which improve the flexibility and functionality of working with streams.

```

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamAPIImprovementsExample {
    public static void main(String[] args) {
        // Example 1: takeWhile
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        List<Integer> lessThanFive = numbers.stream()
            .takeWhile(n -> n < 5)
            .collect(Collectors.toList());

        System.out.println("Numbers less than 5: " + lessThanFive);

        // Example 2: dropWhile
        List<Integer> greaterThanThree = numbers.stream()
            .dropWhile(n -> n <= 3)
            .collect(Collectors.toList());

        System.out.println("Numbers greater than 3: " + greaterThanThree);

        // Example 3: ofNullable

        // Example 3: ofNullable
        String value1 = "Hello";
        String value2 = null;

        // Example with a non-null value
        Stream.ofNullable(value1)
            .ifPresentOrElse(v -> System.out.println("ofNullable Example - Non-null value: " + v),
                () -> System.out.println("ofNullable Example - Null value"));

        // Example with a null value
        Stream.ofNullable(value2)
            .ifPresentOrElse(v -> System.out.println("ofNullable Example - Non-null value: " + v),
                () -> System.out.println("ofNullable Example - Null value"));

        //Example of null safe stream
        List<String> names = Arrays.asList("Alice", "Bob", null, "Charlie", null, "David");
        List<String> nonNullNames = names.stream()
            .flatMap(name -> StreamAPIImprovementsExample.nullSafeStream(name))
            .collect(Collectors.toList());

        System.out.println("Non-null names: " + nonNullNames);
    }

    // Helper method to create a stream from a potentially null value

```

```
private static <T> java.util.stream.Stream<T> nullSafeStream(T value) {
    return value == null ? java.util.stream.Stream.empty() : java.util.stream.Stream.of(value);
}
}
```

In this example:

- `takeWhile` is used to take elements from the stream until a certain condition is met (in this case, numbers less than 5).
- `dropWhile` is used to drop elements from the stream while a certain condition is met (in this case, numbers less than or equal to 3).
- `ofNullable` is used to create a stream from a potentially null value, filtering out null values (in this case, filtering out null names from the list).

Private Methods in Interfaces:

- Interfaces in Java 9 can have private methods, allowing developers to encapsulate common functionality within an interface without exposing it to external classes.

```
// Interface with private method
public interface PrivateMethodInterface {
    default void publicMethod() {
        // Public method can call private method
        privateMethod();
    }

    private void privateMethod() {
        System.out.println("Private method in interface");
    }
}
```

HTTP/2 Client:

- Java 9 introduced a new lightweight HTTP client that supports HTTP/2 and WebSocket. This client is designed to be more efficient and flexible than the old `URLConnection` API.

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class HttpClientExample {
    public static void main(String[] args) throws Exception {
        HttpClient httpClient = HttpClient.newHttpClient();
        HttpRequest httpRequest = HttpRequest.newBuilder()
            .uri(new URI("https://www.example.com"))
            .GET()
            .build();

        HttpResponse<String> response = httpClient.send(httpRequest, HttpResponse.BodyHandlers.ofString());
        System.out.println("Response Code: " + response.statusCode());
        System.out.println("Response Body: " + response.body());
    }
}
```

Java 10

Local-Variable Type Inference (var):

- Java 10 introduced the ability to use the `var` keyword for local variable type inference. This allows developers to declare local variables without explicitly specifying the type, letting the compiler infer it based on the assigned value.

```

public class LocalVarInference {

    /**
     * Allowed: only as a local variable
     * Not allowed: anywhere else (class field, method param, etc.)
     * User var responsibly!
     *
     * Use:
     * - when it's clear what the type is (string, int)
     * - to shorten very long ugly types
     *
     * Don't use:
     * - returned value is unclear (var data = service.getData();)
     */

    public static void main(String[] args) {

        // allowed, but brings little benefit
        var b = "b";
        var c = 5; // int
        var d = 5.0; // double
        var httpClient = HttpClient.newHttpClient();

        // one hell of an inference :)
        var list = List.of(1, 2.0, "3");

        // the benefit becomes more evident with types with long names
        var reader = new BufferedReader(null);
        // vs.
        BufferedReader reader2 = new BufferedReader(null);
    }
}

```

Optional API — new methods introduced

```

public class OptionalApi {

    /**
     * new .orElseThrow()
     */
    public static void main(String[] args) {

        Optional<Flight> earliestFlight = FlightSchedule.getFlights()
            .stream()
            .filter(f -> "Boston".equals(f.from()))
            .filter(f -> "San Francisco".equals(f.to()))
            .min(comparing(Flight::date));

        earliestFlight.orElseThrow(FlightNotFoundException::new);
    }
}

```

Java 11

HTTP client

```

public class HttpClientBasicExample {

    /**
     * Create a client, send a GET Request, print Response info
     */
    public static void main(String... args) throws Exception {
        HttpClient client = HttpClient.newHttpClient();

        HttpRequest request =
            HttpRequest.newBuilder(URI.create("https://github.com/"))
                .GET() // default, may be omitted
                .build();
    }
}

```

```

        HttpResponse<String> response =
            client.send(request, HttpResponse.BodyHandlers.ofString());

        print("Status code was: " + response.statusCode());

        print(response.headers().map());
    }
}

```

New File Methods:

Java 11 introduced several new methods in the `java.nio.file` package, providing additional functionality for working with files and directories. Some of the notable methods include:

1. `Files.readString(Path path)` and `Files.writeString(Path path, CharSequence content, OpenOption... options)`:
 - These methods simplify reading and writing the contents of a file as a string. The `readString` method reads the entire content of a file into a string, and the `writeString` method writes a string to a file.
2. `Files.readAllLines(Path path)` and `Files.write(Path path, Iterable<? extends CharSequence> lines, OpenOption... options)`:
 - These methods simplify reading and writing the contents of a file as a list of strings. The `readAllLines` method reads all lines from a file into a list, and the `write` method writes a collection of strings to a file.
3. `Files.newBufferedReader(Path path)` and `Files.newBufferedWriter(Path path, OpenOption... options)`:
 - These methods create buffered readers and writers for efficient reading and writing of files. They simplify the process of working with character streams.
4. `Files.mismatch(Path path1, Path path2)`:
 - This method compares the content of two files and returns the position of the first mismatched byte. If the files are identical, it returns -1.

```

public class NewFilesMethods {

    static String filePath = System.getProperty("user.dir") + "/src/main/resources/";
    static String file_1 = filePath + "file_1.txt";

    /**
     * Files.readString() and .writeString()
     */
    public static void main(String[] args) throws IOException {

        // reading files is much easier now
        // not to be used with huge files
        Path path = Paths.get(file_1);
        String content = Files.readString(path);
        print(content);

        Path newFile = Paths.get(filePath + "newFile.txt");
        if(!Files.exists(newFile)) {
            Files.writeString(newFile, "some str", StandardOpenOption.CREATE);
        } else {
            Files.writeString(newFile, "some str", StandardOpenOption.TRUNCATE_EXISTING);
        }
    }
}

```

Java 12

Compact Number Formatting:

Java 12 introduced a new feature called “Compact Number Formatting” as part of JEP 357. This enhancement provides a more concise way to format large numbers in a locale-specific manner.

The `NumberFormat` class in the `java.text` package was enhanced to support the new `Style` enum, including the `Style.SHORT` and `Style.LONG` constants. These styles can be used to format large numbers in a compact form based on the specified locale.

```
import java.text.NumberFormat;
import java.util.Locale;

public class CompactNumberFormattingExample {
    public static void main(String[] args) {
        // Creating a number formatter with compact style
        NumberFormat compactFormatter = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.SHORT);

        // Formatting large numbers
        System.out.println("Short Format: " + compactFormatter.format(1000)); // Output: 1K
        System.out.println("Short Format: " + compactFormatter.format(1000000)); // Output: 1M

        // Creating a number formatter with compact style (long)
        NumberFormat compactLongFormatter = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.LONG);

        // Formatting large numbers in long style
        System.out.println("Long Format: " + compactLongFormatter.format(10000000)); // Output: 10 million
        System.out.println("Long Format: " + compactLongFormatter.format(1000000000)); // Output: 1 billion
    }
}
```

String::indent (JEP 326):

- The `String` class in Java 12 introduced a new method called `indent(int n)`. This method is used to adjust the indentation of each line in a string by a specified number of spaces.

```
String indentedString = "Hello\nWorld".indent(3);
// indentedString is now "   Hello\n   World"
```

New Methods in `java.util.Arrays` (JEP 326):

- Java 12 added several new methods to the `java.util.Arrays` class, including `copyOfRange` and `equals` variants that take a `Comparator`.

Improvements in `java.util.stream.Collectors` (JEP 325):

- The `Collectors` utility class in Java 12 introduced new collectors like `teeing`, which allows combining two collectors into a single collector.

New File Methods:

```
public class NewFilesMethod {

    static String filePath = System.getProperty("user.dir") + "/src/main/resources/";
    static String file_1 = filePath + "file_1.txt";
    static String file_2 = filePath + "file_2.txt";

    public static void main(String[] args) throws IOException {

        // Finds and returns the position of the first mismatched byte in the content of two files,
        // or -1L if there is no mismatch
        long result = Files.mismatch(Paths.get(file_1), Paths.get(file_2));

        print(result); // -1
    }
}
```

Nothing much interesting happend: — API update to ByteBuffer — Update to localization (support for new chars and emojis) — GC updates

Java-14

“Switch Expressions” (SE) instead of “Switch Statements” (SS):

Enhanced Switch Expressions:

- Switch expressions, introduced as a preview feature in Java 12 and finalized in Java 13, allow developers to use switch statements as expressions, providing a more concise and expressive syntax.

```
int dayOfWeek = 2;
String dayType = switch (dayOfWeek) {
    case 1, 2, 3, 4, 5 -> "Weekday";
    case 6, 7 -> "Weekend";
    default -> throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeek);
};
```

“Yield” Statement:

- The “yield” statement was introduced in Java 14 to complement switch expressions. It allows you to specify a value to be returned from a switch arm, providing more flexibility in combining both imperative and functional styles.

```
String dayType = switch (dayOfWeek) {
    case 1, 2, 3, 4, 5 -> {
        System.out.println("Working day");
        yield "Weekday";
    }
    case 6, 7 -> {
        System.out.println("Weekend");
        yield "Weekend";
    }
    default -> throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeek);
};
```

One More example,

```
/**
 * "Switch Expressions" (SE) instead of "Switch Statements" (SS)
 * (Both can be used, but SE is better than SS)
 */
public class SwitchExpressions {

    public static void main(String[] args) {

        oldStyleWithBreak(FruitType.APPLE);

        withSwitchExpression(FruitType.PEAR);

        switchExpressionWithReturn(FruitType.KIWI);

        switchWithYield(FruitType.PINEAPPLE);
    }

    // Old style is more verbose and error-prone (forgotten "break;" causes the switch to fall through)
    private static void oldStyleWithBreak(FruitType fruit) {
        print("=== Old style with break ===");
        switch (fruit) {
            case APPLE, PEAR:
                print("Common fruit");
                break;
            case PINEAPPLE, KIWI:
                print("Exotic fruit");
                break;
            default:
                print("Unknown fruit");
        }
    }
}
```



```

        print("Undefined fruit");
    }
}

private static void withSwitchExpression(FruitType fruit) {
    print("==== With switch expression ====");
    switch (fruit) {
        case APPLE, PEAR -> print("Common fruit");
        case PINEAPPLE -> print("Exotic fruit");
        default -> print("Undefined fruit");
    }
}

private static void switchExpressionWithReturn(FruitType fruit) {
    print("==== With return value ====");

    // or just "return switch" right away
    String text = switch (fruit) {
        case APPLE, PEAR -> "Common fruit";
        case PINEAPPLE -> "Exotic fruit";
        default -> "Undefined fruit";
    };
    print(text);
}

/**
 * "Yield" is like "return" but with an important difference:
 * "yield" returns a value and exits the switch statement. Execution stays within the enclosing method
 * "return" exits the switch and the enclosing method
 */
// https://stackoverflow.com/questions/58049131/what-does-the-new-keyword-yield-mean-in-java-13
private static void switchWithYield(FruitType fruit) {
    print("==== With yield ====");
    String text = switch (fruit) {
        case APPLE, PEAR -> {
            print("the given fruit was: " + fruit);
            yield "Common fruit";
        }
        case PINEAPPLE -> "Exotic fruit";
        default -> "Undefined fruit";
    };
    print(text);
}

public enum FruitType {APPLE, PEAR, PINEAPPLE, KIWI}
}

```

Java 15

Text-block

Text blocks are a new kind of string literals that span multiple lines. They aim to simplify the task of writing and maintaining strings that span several lines of source code while avoiding escape sequences.

Example without text blocks:

```

String html = "<html>\n" +
    "    <body>\n" +
    "        <p>Hello, world</p>\n" +
    "    </body>\n" +
    "</html>";

```

Example with text blocks:

```

String html = """
    <html>
    <body>
        <p>Hello, world</p>
    </body>
    """

```

```
</html>
"";
```

Key features of text blocks include:

1. Multiline Strings: Text blocks allow you to represent multiline strings more naturally, improving code readability.
2. Whitespace Control: Leading and trailing whitespaces on each line are removed, providing better control over the indentation.
3. Escape Sequences: Escape sequences are still valid within text blocks, allowing the inclusion of special characters.

Text blocks were designed to make it easier to express strings that include multiple lines of content, such as HTML, XML, JSON, or SQL queries. If there have been any updates or new features related to text blocks in Java 15 or subsequent releases, it's advisable to check the official documentation or release notes for the specific version.

```
/**
 * Use cases for TextBlocks (What's New in Java 15 > Text Blocks in Practice)
 * - Blocks of text using markdown
 * - Testing, defining hard-coded JSON strings
 * - Simple templating
 */
public class TextBlocks {

    public static void main(String[] args) {
        oldStyle();
        emptyBlock();
        jsonBlock();
        jsonMovedEndQuoteBlock();
        jsonMovedBracketsBlock();
    }

    private static void oldStyle() {
        print("***** Old style *****");

        String text = "{\n" +
            "  \"name\": \"John Doe\",\n" +
            "  \"age\": 45,\n" +
            "  \"address\": \"Doe Street, 23, Java Town\"\n" +
            "}";
        print(text);
    }

    private static void emptyBlock() {
        print("***** Empty Block *****");
        String text = ""
            "";
        print("|" + text + "|");
    }

    private static void jsonBlock() {
        print("***** JSON Block *****");

        String text = ""
            {
                "name": "John Doe",
                "age": 45,
                "address": "Doe Street, 23, Java Town"
            }
            ""; // <-- no indentation if char is aligned with first "
        print(text);
    }

    private static void jsonMovedEndQuoteBlock() {
        print("***** Json Moved End Quote Block *****");

        String text = ""
            {
                "name": "John Doe",
                "age": 45,
                "address": "Doe Street, 23, Java Town"
            }
            ""
        }
    }
}
```

```
        """;
    print(text);
}

private static void jsonMovedBracketsBlock() {
    print("***** Json Moved Brackets Block *****");

    String text = ""
        {
            "name": "John Doe",
            "age": 45,
            "address": "Doe Street, 23, Java Town"
        }
        """; // <-- indented by 2 spaces as it is aligned with third "
    print(text);
}
}
```

Java 16

Pattern matching for instanceof

Java 16's pattern matching for `instanceof` is a nifty feature that improves type checking and extraction. Here's a rundown of its key aspects:

What it does:

- Introduces type patterns instead of just checking against a single type.
- Allows declaring a variable within the `instanceof` check to hold the extracted object.
- Combines type checking and casting into a single, more concise and readable expression.

Benefits:

- Reduced boilerplate: Eliminates the need for separate `instanceof` checks, casts, and variable declarations.
- Improved readability: Makes code clearer and easier to understand, especially for complex type hierarchies.
- Reduced errors: Less chance of casting exceptions due to mistaken types.

Syntax

```
if (obj instanceof String s) {
    // Use "s" directly as a String here
} else if (obj instanceof List<Integer> list) {
    // Use "list" directly as a List<Integer> here
} else {
    // Handle other cases
}
```

Example

```
public class PatternMatchingForInstanceof {

    public static void main(String[] args) {

        Object o = new Book("Harry Potter", Set.of("Jon Doe"));

        // old way
        if (o instanceof Book) {
            Book book = (Book) o;
            print("The book's author(s) are " + book.getAuthors());
        }

        // new way
        if (o instanceof Book book) {
```

```
        print("The book's author(s) are " + book.getAuthors());
    }
}
}
```

Record:

Records in Java are a special type of class specifically designed for holding immutable data. They help reduce boilerplate code and improve readability and maintainability when dealing with simple data structures.

- Here's a breakdown of their key characteristics:

1. Conciseness:

- Unlike traditional classes, records require minimal code to define. You just specify the data fields (components) in the record declaration, and the compiler automatically generates essential methods like:
- Constructor with parameters for each component.
- Getters for each component.
- `equals` and `hashCode` methods based on component values.
- `toString` method representing the record's state.

2. Immutability:

- Record fields are declared as `final`, making the data stored within them unmodifiable after the record is created. This ensures data consistency and simplifies thread safety concerns.

3. Readability:

- The auto-generated methods and predictable behavior of records enhance code clarity and make it easier to understand what the record represents and how it interacts with other parts of your program.

4. Reduced Errors:

- By minimizing boilerplate, records reduce the risk of common mistakes like forgetting getters or implementing `equals` incorrectly. This leads to more robust and reliable code.

Overall, records are a valuable tool for Java developers to create concise, immutable, and readable data structures, leading to cleaner, more maintainable code.

```
/**
 * Record are data-only immutable classes (thus have specific use cases)
 * They are a restricted (specialized) form of a class (like enums)
 * Not suitable for objects that are meant to change state, etc.
 * <p>
 * Records are NOT:
 * - Boilerplate reduction mechanism
 * <p>
 * Records generate constructors, getters, fields; equals, hashCode, toString
 * <p>
 * Use cases:
 * - to model immutable data
 * - to hold read-only data in memory
 * - DTOs - Data Transfer Objects
 */
public class RecordsDemo {

    public static void main(String[] args) {
        Product p1 = new Product("milk", 50);
        Product p2 = new Product("milk", 50);

        print(p1.price()); // without "get" prefix
        print(p1);          // auto-generated toString() - Product[name=milk, price=50]
    }
}
```

```

        print(p1 == p2);        // false    - different objects
        print(p1.equals(p2));   // true     - values of fields (milk, 50) are compared by the auto-generated equals()/hashCo
    }

}

/**
 * params are called "Components"
 * want more fields - must add into the signature
 * Extending not allowed, implementing interfaces IS allowed
 */
public record Product(String name, int price) {

    // static fields allowed, but not non-static
    static String country = "US";

    // constructor with all fields is generated

    // can add validation
    public Product {
        if(price < 0) {
            throw new IllegalArgumentException();
        }
    }

    // possible to override auto-generated methods like toString()
}

```

Date Time Formatter API:

- General usage and features of the `DateTimeFormatter` API in Java 16: This includes understanding format patterns, creating custom formats, parsing dates and times, and available formatting options.
- New features introduced in Java 16 for date formatting: Specifically, the day period support using the "B" symbol and its various styles.
- Comparison of `DateTimeFormatter` with older formatters like `SimpleDateFormat` : Exploring the advantages and disadvantages of each approach.
- Examples of using `DateTimeFormatter` for specific formatting tasks: Like formatting dates in different locales, handling time zones, or generating human-readable representations.

```

public class DateTimeFormatterApi {

    static Map<TextStyle, Locale> map = Map.of(
        TextStyle.FULL, Locale.US,
        TextStyle.SHORT, Locale.FRENCH,
        TextStyle.NARROW, Locale.GERMAN
    );

    public static void main(String[] args) {

        for (var entry : map.entrySet()) {

            LocalDateTime now = LocalDateTime.now();
            DateTimeFormatter formatter = new DateTimeFormatterBuilder()
                .appendPattern("yyyy-MM-dd hh:mm ")
                .appendDayPeriodText(entry.getKey())    // at night, du soir, abends, etc.
                .toFormatter(entry.getValue());

            String formattedDateTime = now.format(formatter);
            print(formattedDateTime);

        }
    }
}

```

Changes in Stream API:

Java 16 brought some exciting changes to the Stream API, making it even more powerful and convenient to use. Here are the key highlights:

1. `Stream.toList()` method: This new method provides a concise way to collect the elements of a stream into a `List`. Previously, you had to use `collect(Collectors.toList())`, which is now slightly redundant.
2. `Stream.mapMulti()` method: This method allows you to map each element of a stream to zero or more elements, creating a new stream of the resulting elements. It's handy for splitting or flattening complex data structures.
3. Enhanced line terminator handling: Java 16 clarifies the definition of line terminators in the `java.io.LineNumberReader` class. This eliminates inconsistencies and ensures consistent behavior when reading line-based data.
4. Other minor changes:
 - String streams now support the `limit` and `skip` methods directly, removing the need for intermediate operations.
 - The `peek` method can now be used with parallel streams, allowing side effects without impacting parallelism.

```
public class StreamApi {

    public static void main(String[] args) {

        List<Integer> ints = Stream.of(1, 2, 3)
            .filter(n -> n < 3)
            .toList(); // new, instead of the verbose .collect(Collectors.toList())

        ints.forEach(System.out::println);

    }
}
```

Java 17

Sealed classes(Subclassing):

Sealed classes are a brand new feature introduced in Java 17 (JEP 409) that gives you more control over inheritance hierarchies. They essentially let you restrict which classes can extend or implement your class or interface. This can be incredibly useful for a variety of reasons, including:

1. **Enhanced Type Safety:** By specifying allowed subclasses, you prevent unexpected or unwanted extensions that could break your code or introduce security vulnerabilities.
2. **Improved Library Design:** You can create closed ecosystems within your libraries, ensuring users only work with approved extensions and don't create incompatible implementations.
3. **Easier Code Maintenance:** Knowing the exact set of possible subclasses simplifies reasoning about your code and makes it easier to understand and maintain.

How do sealed classes work?

You declare a class or interface as `sealed` using the `sealed` keyword. Then, you use the `permits` clause to specify a list of classes that are allowed to extend or implement it. Only these permitted classes can directly inherit, while all other classes are prohibited.

```
sealed class Shape {
    permits Circle, Square, Triangle;
    // ... implementation details
}

class Circle extends Shape {
    // ...
}

// This will cause a compile-time error because Rectangle isn't permitted
class Rectangle extends Shape {
```

```
// ...
}
```

A sealed class or interface can be extended or implemented only by those classes and interfaces permitted to do so.

Benefits:

- 1) help enforce a well-defined and limited set of possible implementations — communicates INTENTION
- 2) Better security — help prevent unexpected or unauthorized subclassing and behavior from third-party code

Rules:

- 1. A sealed class uses “permits” to allow other classes to subclass it.
- 2. A child class MUST either be final, sealed or non-sealed. (or code won't compile)
- 3. A permitted child class MUST extend the parent sealed class. Permitting without using the permit is now allowed.
- 4. The classes specified by permits must be located near the superclass:
 - either in the same module (if the superclass is in a named module) (see Java 9 modularity)
 - or in the same package (if the superclass is in the unnamed module).

More on point 4:

- The motivation is that a sealed class and its (direct) subclasses are tightly coupled since they must be compiled and maintained together.
- In a modular world, this means “same module”; in a non-modular world, the best approximation for this is “same package”.
- If you use modules, you get some additional flexibility, because of the safety boundaries modules give you.

Java 18

UTF-8 by Default:

Java 18 makes UTF-8 the default character encoding for the platform, aligning with modern standards and simplifying character handling.

```
public class Utf8ByDefault {

    // https://openjdk.org/jeps/400 - Platform Default Encoding

    public static void main(String[] args) throws IOException {

        // Problem:
        // 1) On Windows, use the below FileWriter to write characters outside the ASCII table, e.g. some exotic Unicode ch
        // 2) Copy or transfer the file to a UNIX-based OS, like a Mac, and read the file using the default char encoding o
        // 3) Likely result - garbled output
        // Hence the problem - unpredictable behavior.
        FileWriter writer = new FileWriter("out.txt");

        // Solution before Java 18: always specify the charset, (and good luck not forgetting it!)
        FileWriter writer2 = new FileWriter("out.txt", StandardCharsets.UTF_8);

        // Solution since Java 18: UTF-8 is now default, so no need to specify the Char set

    }

}
```

Simple Web Server:

This new API provides a basic web server for serving static files, ideal for quick prototyping and embedded applications.

- Ensure you have Java 18 or later installed on your system.
- Have your static files (HTML, CSS, JavaScript, images, etc.) ready in a specific directory.

```
import com.sun.net.httpserver.HttpServer;
import com.sun.net.httpserver.SimpleFileServer;

import java.net.InetSocketAddress;

public class SimpleWebServer {
    public static void main(String[] args) throws Exception {
        String documentRoot = "/path/to/your/static/files"; // Replace with your actual directory
        int port = 8080; // You can change the port if needed

        HttpServer server = HttpServer.create(new InetSocketAddress(port), 0);
        SimpleFileServer fileServer = new SimpleFileServer(documentRoot);
        server.setExecutor(null); // Use single-threaded executor
        server.createContext("/", fileServer);

        server.start();
        System.out.println("Server started on port " + port);
    }
}
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Title of the document</title>
</head>

<body>
    This page can be served with Java's Simple Web Server using the "jwebserver" command
</body>

</html>
```

. Compile and run:

- Compile the Java file: `javac SimpleWebServer.java`
- Run the compiled class: `java SimpleWebServer`

4. Access the page:

- Open your web browser and navigate to `http://localhost:8080` (or the specified port).
- You should see the default file (usually `index.html`) from your static files directory being served.

5. Stopping the server:

- To stop the server, press `Ctrl+C` in the terminal where it's running.

The simplest way to get started:

- 1) open the terminal in this package (java18)
- 2) run "java -version" and make sure it is at least java 18
- 3) run the "jwebserver" command

It should output:

Binding to loopback by default. For all interfaces use "-b 0.0.0.0" or "-b ::".

Serving path/to/your/subdirectory and subdirectories on 127.0.0.1 port 8000

The HTML page is now served at: <http://127.0.0.1:8000/java18/doc.html>

IP address, port and other parameters may be changed

HEAD() convenience method added:

```
public class HttpHeadersDemo {  
  
    /**  
     * HEAD() convenience method added  
     */  
    public static void main(String[] args) throws IOException, InterruptedException {  
        HttpRequest head = HttpRequest.newBuilder(URI.create("https://api.github.com/"))  
            .HEAD()  
            .build();  
  
        var response = HttpClient.newHttpClient().send(head, HttpResponse.BodyHandlers.ofString());  
  
        print(response);  
    }  
}
```

Reimplement Core Reflection with Method Handles: This reimplementation aims to improve performance and stability of reflection functionalities.

Deprecate Finalization for Removal: Finalization, intended for resource cleanup, has inherent drawbacks. Its deprecation paves the way for safer and more reliable alternatives.

Java 19

Either preview or incubator features:

<p>Oracle Releases Java 19</p> <p>New release delivers seven JDK Enhancement Proposals to increase developer productivity, improve the Java language, and...</p> <p>www.oracle.com</p>	
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--

Java 20

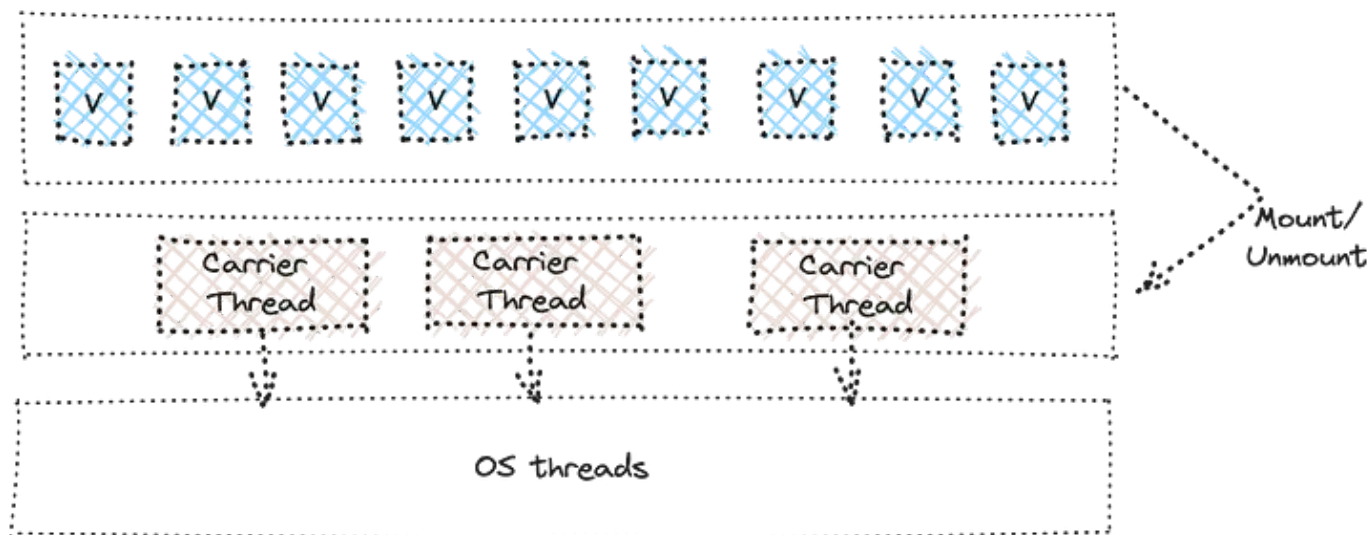
Either preview or incubator features:

<p>Oracle Releases Java 20</p> <p>New release delivers seven JDK Enhancement Proposals to increase developer productivity, improve the Java language, and...</p> <p>www.oracle.com</p>	
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--

Java 21

Virtual Threads:

this feature introduces lightweight threads that run on top of the operating system threads, aiming to simplify concurrent programming and improve performance for certain workloads.



Here's a simple demo showcasing virtual threads in Java 21:

1. Create Virtual Threads:

```
Thread vThread1 = Thread.ofVirtual().start(() -> {
    for (int i = 0; i < 10; i++) {
        System.out.println("Virtual Thread 1: " + i);
    }
});

Thread vThread2 = Thread.ofVirtual().start(() -> {
    for (int i = 0; i < 10; i++) {
        System.out.println("Virtual Thread 2: " + i);
    }
});
```

2. Wait for Completion:

```
vThread1.join();
vThread2.join();
```

Output:

This will interleave the outputs from both virtual threads, demonstrating concurrent execution without the overhead of full OS threads. You might see something like:

```
Virtual Thread 1: 0
Virtual Thread 2: 0
Virtual Thread 1: 1
Virtual Thread 2: 1
...
Virtual Thread 1: 9
Virtual Thread 2: 9
```

Virtual Thread Explanation:

Virtual threads are lightweight units of execution that run on top of a smaller pool of underlying OS threads. They offer several advantages:

- **Lighter weight:** Compared to OS threads, virtual threads have significantly lower creation and context switching costs.

- Improved concurrency: More virtual threads can be efficiently managed within a limited number of OS threads, allowing better utilization of resources for certain workloads.
- Simpler concurrency programming: Virtual threads eliminate the need for complex thread management and synchronization, making concurrent programming easier for developers.

The following is an example of virtual threads and a good contrast to OS/platform threads. The program uses the `ExecutorService` to create 10,000 tasks and waits for all of them to be completed. Behind the scenes, the JDK will run this on a limited number of carrier and OS threads, providing you with the durability to write concurrent code with ease.

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 10_000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(1));
            return i;
        });
    });
} // executor.close() is called implicitly, and waits
```

Record Patterns (Project Amber):

Records were introduced as a preview in Java 14, which also gave us Java enums. `record` is another special type in Java, and its purpose is to ease the process of developing classes that act as data carriers only.

In JDK 21, record patterns and type patterns can be nested to enable a declarative and composable form of data navigation and processing.

```
// To create a record:

public record Todo(String title, boolean completed){}

// To create an Object:

Todo t = new Todo("Learn Java 21", false);
```

Before JDK 21, the entire record would need to be deconstructed to retrieve accessors.. However, now it is much more simplified to get the values. For example:

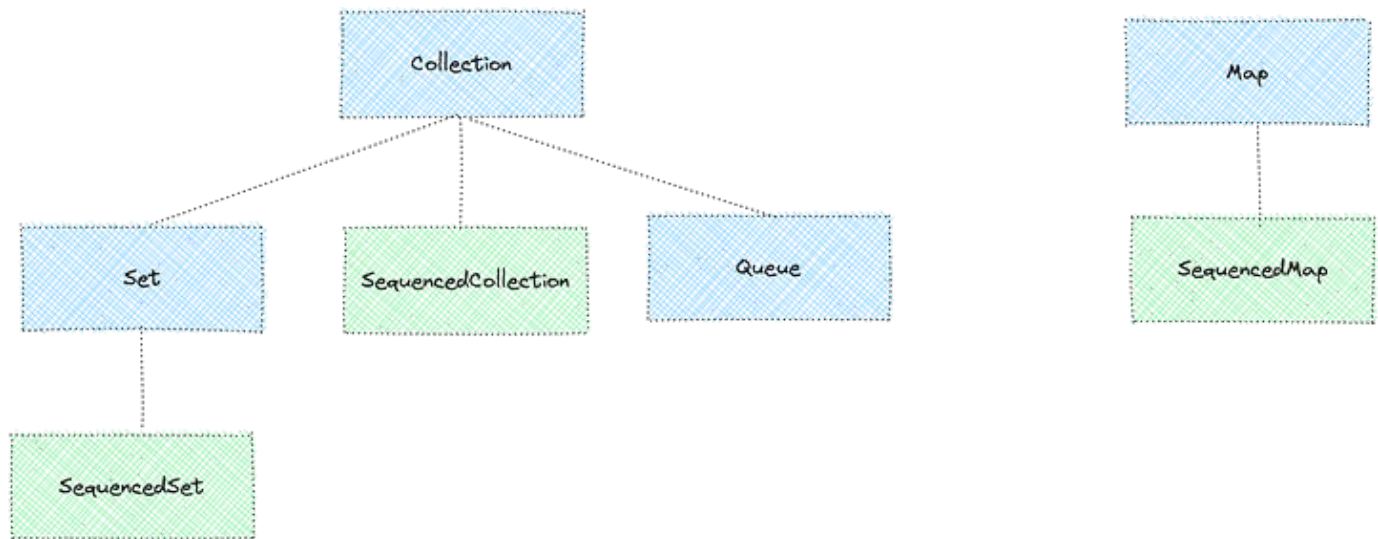
```
static void printTodo(Object obj) {
    if (obj instanceof Todo(String title, boolean completed)) {
        System.out.print(title);
        System.out.print(completed);
    }
}
```

The other advantage of record patterns is also nested records and accessing them. An example from the JEP definition itself shows the ability to get to the `Point` values, which are part of `ColoredPoint`, which is nested in a `Rectangle`. This makes it way more useful than before, when all the records needed to be deconstructed every time.

```
// As of Java 21
static void printColorOfUpperLeftPoint(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint(Point p, Color c),
                               ColoredPoint lr)) {
        System.out.println(c);
    }
}
```

Sequenced collections:

In [JDK 21](#), a new set of collection interfaces are introduced to enhance the experience of using collections. For example, if one needs to get a reverse order of elements from a collection, depending on which collection is in use, it can be tedious. There can be inconsistencies retrieving the encounter order depending on which collection is being used; for example, `SortedSet` implements one, but `HashSet` doesn't, making it cumbersome to achieve this on different data sets.



To fix this, the [SequencedCollection](#) interface aids the encounter order by adding a `reverse` method as well as the ability to get the first and the last elements. Furthermore, there are also `SequencedMap` and `SequencedSet` interfaces.

```

interface SequencedCollection<E> extends Collection<E> {
    // new method
    SequencedCollection<E> reversed();
    // methods promoted from Deque
    void addFirst(E);
    void addLast(E);
    E getFirst();
    E getLast();
    E removeFirst();
    E removeLast();
}
  
```

String templates:

[String templates](#) are a preview feature in JDK 21. However, it attempts to bring more reliability and better experience to `String` manipulation to avoid common pitfalls that can sometimes lead to undesirable results, such as injections. Now you can write template expressions and render them out in a `String`.

```

// As of Java 21
String name = "Ajay"
String greeting = "Hello \{name}";
System.out.println(greeting);
  
```

In this case, the second line is the expression, and upon invoking, it should render `Hello Ajay`. Furthermore, in cases where there is a chance of illegal `Strings`—for example, SQL statements or HTML that can cause security issues—the template rules only allow escaped quotes and no illegal entities in HTML documents.

Thanks for reading

- 👏 Please clap for the story and follow me 🙌
- 📖 Read more content on my [Medium](#) (50 stories on Java Developer interview)

Find my books here:

- *Guide To Clear Java Developer Interview* here [Amazon](#) (Kindle Book) and [Gumroad](#) (PDF Format).
- *Guide To Clear Spring-Boot Microservice Interview* here [Gumroad](#) (PDF Format) and [Amazon](#) (Kindle eBook).
- 🔔 Follow me: [LinkedIn](#) | [Twitter](#) | [Youtube](#)

Java

Coding

Programming

Spring

Microservices



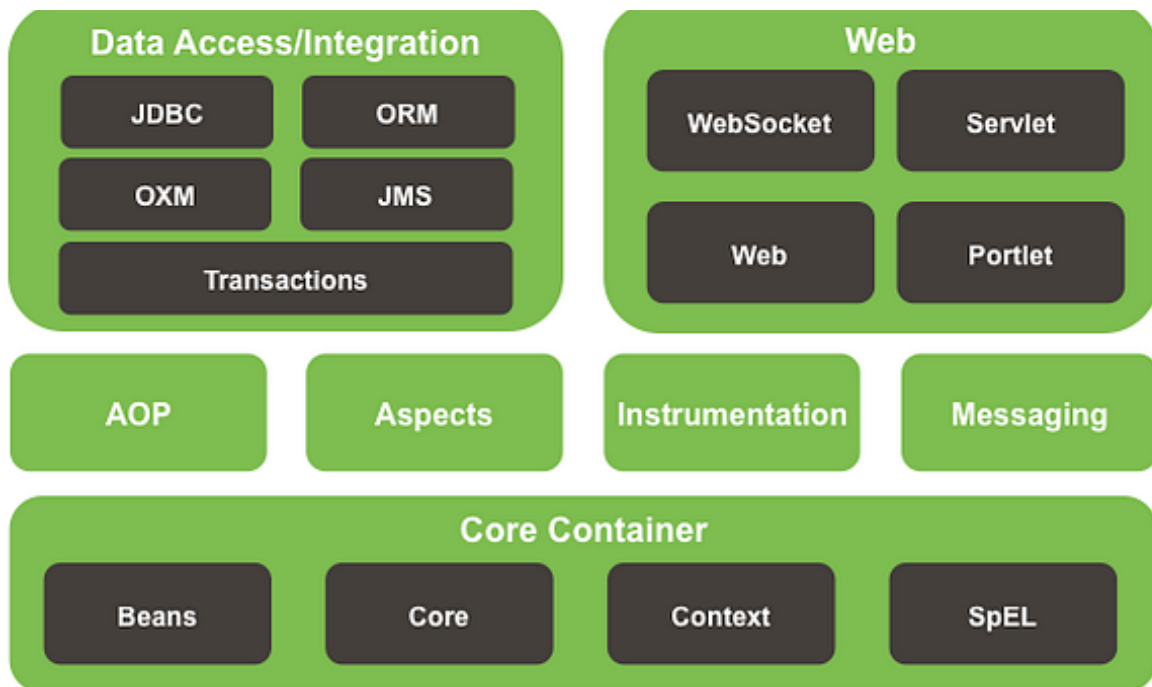
Follow

Written by Ajay Rathod

6K Followers

Software Engineer @Cisco

More from Ajay Rathod



Ajay Rathod

Top 60 Spring-Framework Interview Questions for Java Developers 2024(Contain All the Questions from...

Hey there! If you're a Java Developer, mastering the Spring framework is crucial for backend development. It serves as the foundation for...

39 min read · Jan 6



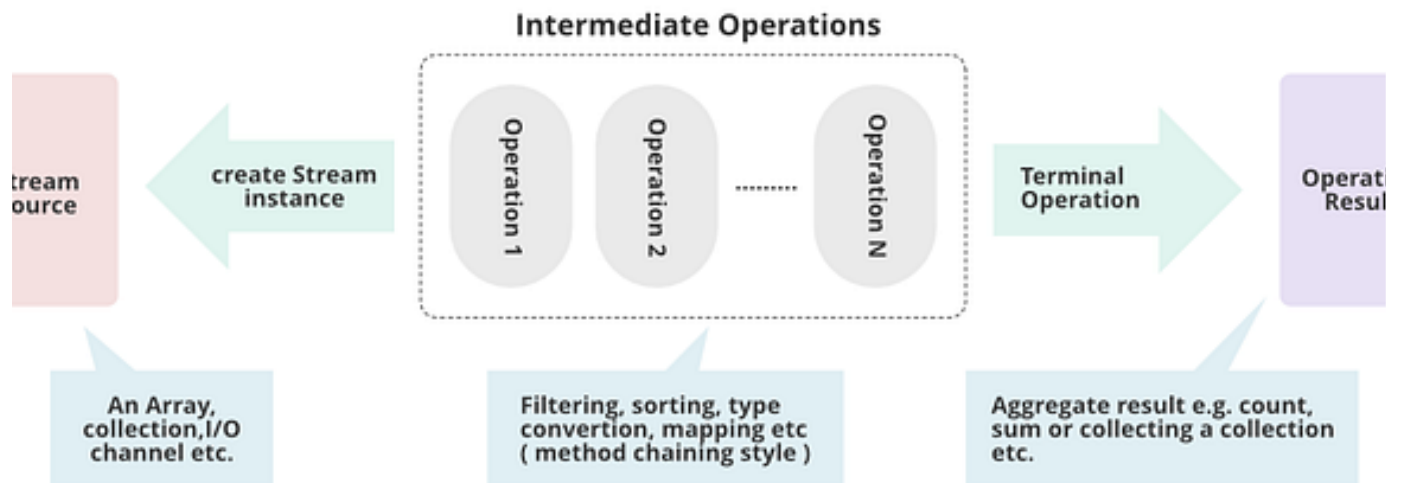
296




2



Java Streams



 Ajay Rathod in Javarevisited

Must Know Java 8 Stream Interview Questions for Java Developers series-16


Here is the list of Java 8 interview questions asked in an interview, posting answers as well for better preparation.

4 min read · May 4, 2023

 210 



 Ajay Rathod in Level Up Coding

Experienced Spring/Spring Boot Interview Questions for Java Developers-2023[5-10 years]

This article explores recently encountered interview questions for experienced Java developers who are familiar with Spring and Spring...

8 min read · Oct 2, 2023

 482  8

Top 30 Java-8 Interview Questions & Answers to Ace Your Java Interview

 Ajay Rathod

Top 30 Java-8 Interview Questions & Answers to Ace Your Java Interview (Includes Coding questions)

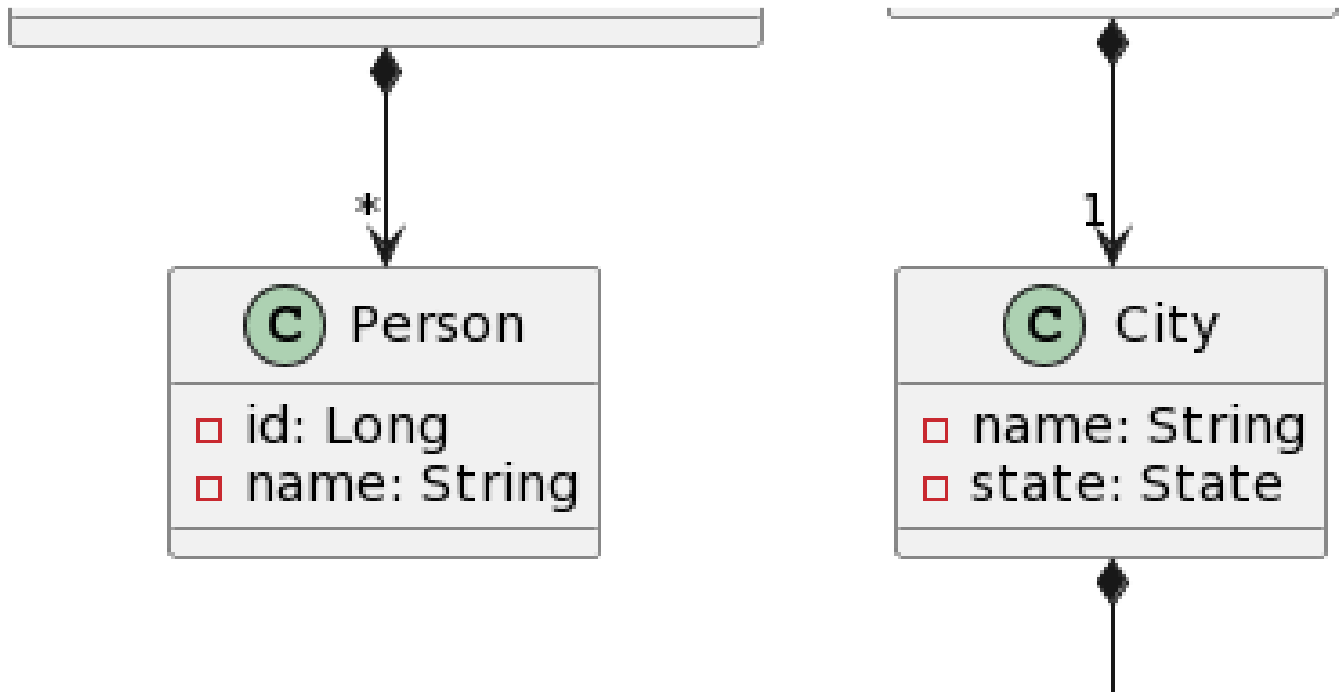
Greetings, readers! In this article, I present a compilation of crucial Java 8 interview questions that can significantly contribute to...

23 min read · Jan 24

 90  3

Recommended from Medium



Renan Schmitt


7 Tricks of Java Streams

Streams were introduced many years ago; however, as Java developers, we still haven't fully managed the power of this versatile tool. In...

★ · 2 min read · Jan 16

151 2



 Vikas Taank

Senior Java Lead Interview Questions.(Advanced)-05

What is A configuration server in Spring Boot Application?

★ · 3 min read · Jan 28


 13


 2







Lists

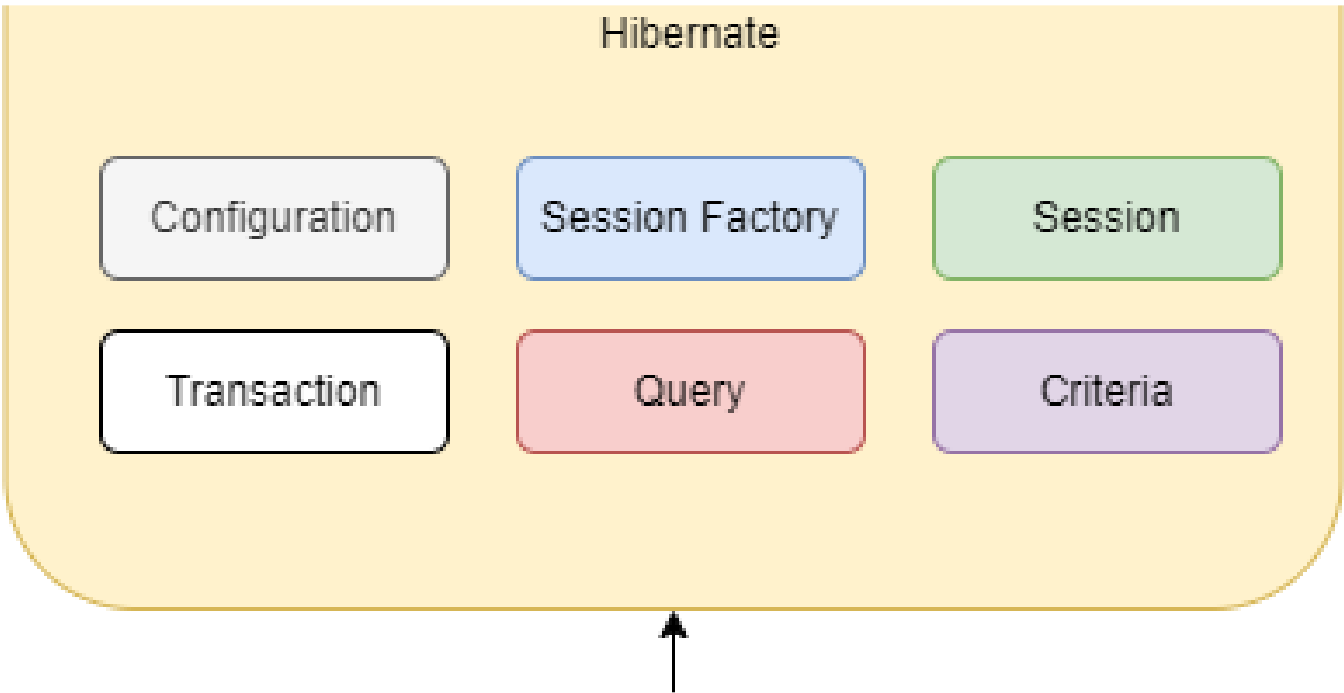
- 

General Coding Knowledge
20 stories · 874 saves
- 

Stories to Help You Grow as a Software Developer
19 stories · 776 saves
- 

Coding & Development
11 stories · 423 saves
- 

ChatGPT
21 stories · 437 saves



 Berat Yesbek


Hibernate, JPA, Spring DATA JPA, Hibernate Proxy and Architectures

Why should we use ORM Tools?

7 min read · Jan 29

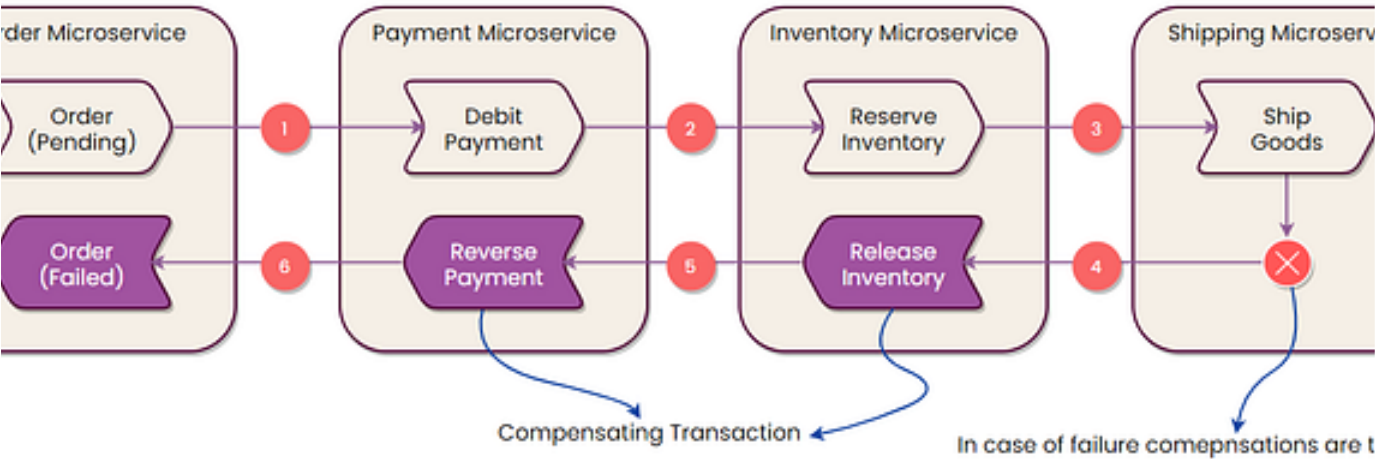
 233

 1





Saga



The Java Trail

Microservice Distributed Transactions 101: Guide to Choose the Best Strategy

Your application relies on multiple services, and maintaining data consistency across distributed transactions is crucial. How would you...

6 min read · Jan 27

244 4

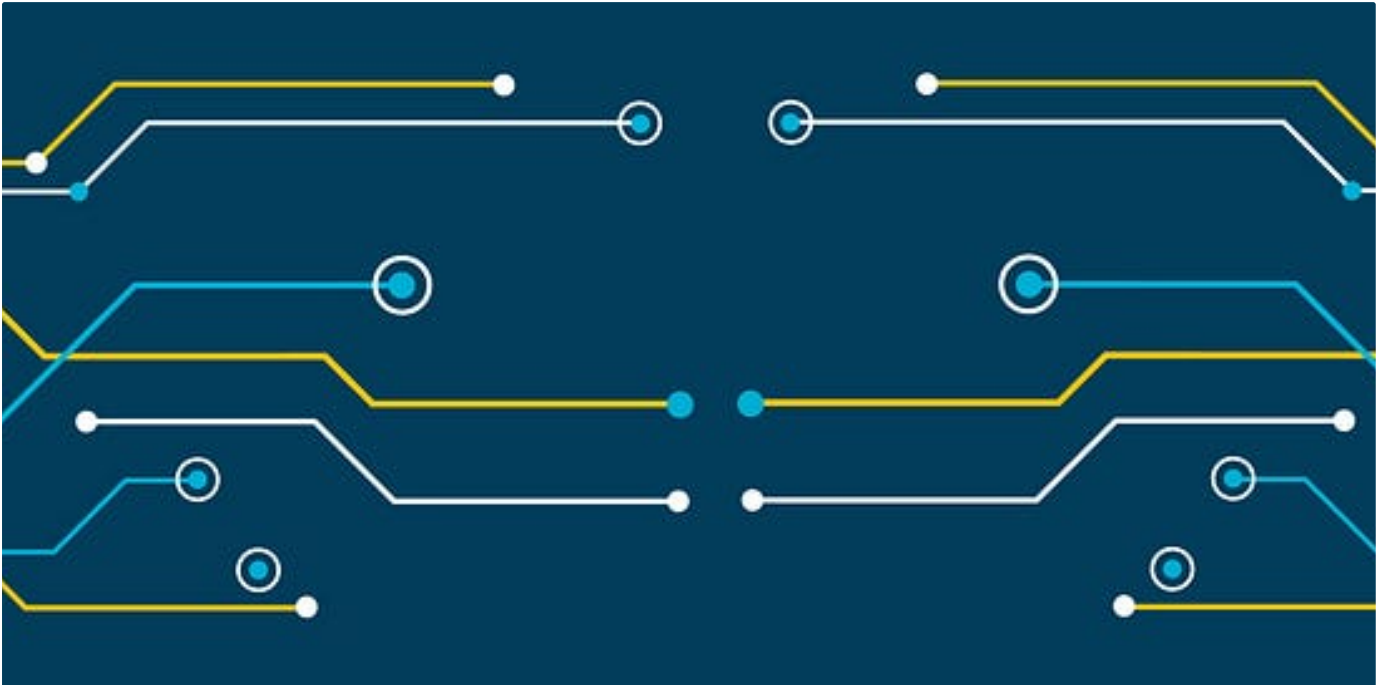
...

 Swastik

Common mistakes to avoid when using @Transactional in Spring

In the world of Spring Framework, managing transactions using @Transactional is a powerful feature. However, with great power comes great...

4 min read · Jan 28

 32   Capital One Tech in Capital One Tech

10 microservices design patterns for better architecture

Consider using these popular design patterns in your next microservices app and make organization more manageable.

11 min read · Jan 9

 763  

See more recommendations