1. Lambda Expressions

Theory:

Lambda expressions introduce a way to treat functionality as a method argument or pass a block of code (function) as data. Essentially, they bring functional programming capabilities into Java, allowing for a more concise, readable, and expressive coding style.

Syntax:
java
Copy code
```
(parameters) -> expression
// or
(parameters) -> { statements }
```

Example:
java
Copy code
```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(name -> System.out.println(name));
```

Reason:

- **Conciseness**: Before Java 8, passing functionality involved using anonymous inner classes, which were verbose and cumbersome.
- **Functional Programming**: Java developers needed the ability to use functional programming concepts, where behavior can be passed around as an argument (first-class citizens).
- **Parallelism & Collections**: Lambda expressions enabled better support for parallel processing through the new Stream API (discussed next).

---

2. Stream API

Theory:

The Stream API is a new abstraction introduced in Java 8 that allows developers to process sequences of data in a declarative manner. Streams represent a pipeline of operations, which can be performed on collections or arrays, such as filtering, mapping, and reducing data.

Example:
java
Copy code

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<String> filteredNames = names.stream()
    .filter(name -> name.startsWith("A"))
    .collect(Collectors.toList());
```

Reason:

- **Declarative Programming**: The Stream API allows developers to work with collections in a declarative way rather than the imperative way (using loops).
- **Parallel Processing**: Streams make parallel processing easier and more efficient with methods like `.parallelStream()`, allowing for more readable and scalable parallel code.
- **Functional Composition**: Operations such as `filter()`, `map()`, `reduce()`, etc., support functional programming principles and allow for clean and modular code.

---

3. Functional Interfaces

Theory:

Functional interfaces are interfaces that have exactly one abstract method, making them suitable for lambda expressions. Java 8 introduces a number of built-in functional interfaces such as `Predicate`, `Function`, `Supplier`, `Consumer`, etc., and also allows users to define their own functional interfaces.

Example:
java
Copy code
```java
@FunctionalInterface
interface MyFunctionalInterface {
    void display(String message);
}
```

Reason:

- **Lambda Integration**: Lambda expressions work seamlessly with functional interfaces. Without functional interfaces, lambdas wouldn't have a target type to refer to.
- **Reduced Boilerplate**: Functional interfaces provide a way to define behavior concisely and reduce boilerplate code.

---

4. Default and Static Methods in Interfaces

Theory:

Java 8 allows interfaces to have methods with a default implementation using the `default` keyword. This allows interfaces to evolve over time by adding new methods without breaking existing implementations.

Example:
java
Copy code
```
interface Vehicle {
    default void start() {
        System.out.println("Vehicle is starting");
    }
}
```

Reason:

- **Backward Compatibility**: Before Java 8, any change in an interface would require all implementing classes to update. Default methods allow new methods to be added to interfaces without breaking existing implementations.
- **Multiple Inheritance**: While Java does not support multiple inheritance for classes, default methods allow a form of multiple inheritance where classes can inherit behavior from multiple interfaces.

---

5. Optional Class

Theory:

`Optional` is a container object introduced in Java 8 that may or may not contain a non-null value. It provides a way to avoid `NullPointerException` and makes the presence or absence of a value explicit in the API.

Example:
java
Copy code
```
Optional<String> name = Optional.ofNullable("John");
name.ifPresent(System.out::println);
```

Reason:

- **NullPointerException Prevention**: One of the most common runtime exceptions in Java is `NullPointerException`. `Optional` provides a more expressive way to deal with optional values, encouraging safer handling of potentially null objects.

- **Encapsulation of Null**: Instead of checking for `null` explicitly, `Optional` offers a structured way to check and act upon the presence or absence of values.

---

6. Date and Time API (java.time)

Theory:

Java 8 introduced a new `java.time` package that replaced the outdated `java.util.Date` and `java.util.Calendar` classes. The new API is much more modern, immutable, and thread-safe.

Example:
java
Copy code
```java
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1990, Month.JANUARY, 1);
```

Reason:

- **Old API Problems**: The old `Date` and `Calendar` classes were mutable, inconsistent, and not thread-safe, making them difficult to work with in modern applications.
- **Immutability**: The new `java.time` classes are immutable, meaning their values cannot be changed after creation, which makes them thread-safe and easier to reason about.
- **Better Design**: The new API provides more meaningful methods and constants, offering a much cleaner and intuitive interface for developers.

---

7. Method References

Theory:

Method references provide a shorthand for referring to a method without executing it. They make lambda expressions even more concise and readable when you're simply passing an existing method as a lambda expression.

Example:
java
Copy code
```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(System.out::println);
```

Reason:

- **Conciseness**: In many cases, a lambda expression is only used to refer to an existing method. Method references offer a simpler way to do this.
- **Readability**: They make code more readable by removing unnecessary boilerplate that comes with lambda expressions.

---

8. Nashorn JavaScript Engine

Theory:

Nashorn is a JavaScript engine introduced in Java 8 to replace the old Rhino engine. It allows embedding JavaScript code within Java applications and provides better performance and compliance with the ECMAScript specification.

Example:
java
Copy code
```
ScriptEngine engine = new
ScriptEngineManager().getEngineByName("nashorn");
engine.eval("print('Hello from JavaScript');");
```

Reason:

- **JavaScript Interoperability**: As JavaScript became more important for web development, Nashorn provided better performance and interoperability with Java.
- **Polyglot Applications**: It allows developers to combine Java and JavaScript, providing more flexibility when dealing with dynamic languages.

---

9. Parallel Array Sorting

Theory:

Java 8 introduced a more efficient way to sort arrays in parallel. This is particularly useful for large arrays where parallelization can significantly speed up the sorting process.

Example:
java
Copy code
```
int[] numbers = {3, 1, 4, 1, 5, 9};
Arrays.parallelSort(numbers);
```

Reason:

- **Performance**: Sorting large datasets in parallel can take advantage of multi-core processors, improving performance for computationally intensive operations.
- **Scalability**: Parallel sorting can be scaled across multiple cores for faster execution.

---

10. Collectors API in Stream

Theory:

Collectors are part of the Stream API and are used to accumulate elements of a stream into various collection types or other results like strings, sums, or averages.

Example:
java
Copy code
```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
String result = names.stream().collect(Collectors.joining(", "));
Collectors.ToList
.toSet
```

Reason:

- **Functional Composition**: Collectors provide a flexible and functional way to accumulate data from streams, which fits well with the functional programming paradigm introduced in Java 8.
- **Ease of Use**: Instead of manually processing elements, Collectors handle common operations like joining strings, summing numbers, or converting streams to lists or sets

1. TOList

- **Description**: Collects the elements of the stream into a `List`.

Example:
java
Copy code
```java
List<String> names = Stream.of("Alice", "Bob", "Charlie")

                           .collect(Collectors.toList());
```

-
- **Result**: A `List` containing ["Alice", "Bob", "Charlie"].

---

2. toSet()

- **Description**: Collects the elements of the stream into a `Set`, eliminating duplicates.

Example:
java
Copy code
```
Set<String> names = Stream.of("Alice", "Bob", "Charlie", "Alice")
                              .collect(Collectors.toSet());
```

-
- **Result**: A `Set` containing ["Alice", "Bob", "Charlie"] (duplicates are removed).

---

3. toMap()

- **Description**: Collects the elements into a `Map` using a key-mapper and value-mapper function.

Example:
java
Copy code
```
Map<String, Integer> nameLengthMap = Stream.of("Alice", "Bob", "Charlie")


.collect(Collectors.toMap(name -> name, name -> name.length()));
```

-
- **Result**: A `Map` where keys are names and values are their lengths: `{"Alice" -> 5, "Bob" -> 3, "Charlie" -> 7}`.

---

4. joining()

- **Description**: Concatenates the elements of a stream into a single `String`.
- **Variants**:
  - `joining()`: Joins elements without any delimiter.
  - `joining(CharSequence delimiter)`: Joins elements with a delimiter.
  - `joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`: Joins elements with a delimiter, prefix, and suffix.

Example:
java

Copy code
```
String result = Stream.of("Alice", "Bob", "Charlie")
                      .collect(Collectors.joining(", "));
```

-
- **Result**: A `String`: `"Alice, Bob, Charlie"`.

---

5. counting()

- **Description**: Counts the number of elements in the stream.

Example:
java
Copy code
```
long count = Stream.of("Alice", "Bob", "Charlie")
                   .collect(Collectors.counting());
```

-
- **Result**: `3`.

---

6. summarizingInt(), summarizingDouble(), summarizingLong()

- **Description**: Collects statistics, such as count, sum, min, average, and max of the elements.

Example:
java
Copy code
```
IntSummaryStatistics stats = Stream.of(1, 2, 3, 4, 5)

.collect(Collectors.summarizingInt(Integer::intValue));
```

-
- **Result**: `IntSummaryStatistics` containing count, sum, min, max, and average.

---

7. averagingInt(), averagingDouble(), averagingLong()

- **Description**: Calculates the average of the elements.

Example:
java
Copy code
```
double average = Stream.of(1, 2, 3, 4, 5)


.collect(Collectors.averagingInt(Integer::intValue));
```

-
- **Result**: `3.0`.

---

8. summingInt(), summingDouble(), summingLong()

- **Description**: Calculates the sum of the elements.

Example:
java
Copy code
```
int sum = Stream.of(1, 2, 3, 4, 5)
                .collect(Collectors.summingInt(Integer::intValue));
```

-
- **Result**: `15`.

---

9. maxBy() and minBy()

- **Description**: Finds the maximum or minimum element according to a provided comparator.

Example:
java
Copy code
```
Optional<String> maxName = Stream.of("Alice", "Bob", "Charlie")


.collect(Collectors.maxBy(Comparator.comparing(String::length)));
```

-
- **Result**: `Optional[Charlie]`.

---

10. groupingBy()

- **Description**: Groups elements by a classifier function and returns a `Map` where the keys are the group identifiers and the values are lists of the elements in those groups.
- **Variants**:
  - `groupingBy(Function classifier)`: Basic grouping.
  - `groupingBy(Function classifier, Collector downstream)`: Grouping with downstream collection.
  - `groupingBy(Function classifier, Supplier mapFactory, Collector downstream)`: Custom `Map` implementation.

Example:
java
Copy code
```
Map<Integer, List<String>> groupedByLength = Stream.of("Alice", "Bob", "Charlie")


.collect(Collectors.groupingBy(String::length));
```

-
- **Result**: `{5=[Alice], 3=[Bob], 7=[Charlie]}`.

---

11. partitioningBy()

- **Description**: Partitions the elements into two groups based on a predicate. Returns a `Map<Boolean, List<T>>` where the key is `true` or `false` depending on whether the element satisfies the predicate.

Example:
java
Copy code
```
Map<Boolean, List<String>> partitioned = Stream.of("Alice", "Bob", "Charlie")


.collect(Collectors.partitioningBy(name -> name.length() > 3));
```

-
- **Result**: `{false=[Bob], true=[Alice, Charlie]}`.

---

12. reducing()

- **Description**: Performs a reduction on the elements using an associative accumulation function, such as summing, concatenation, or finding the maximum/minimum.
- **Variants**:
  - `reducing(BinaryOperator<T> accumulator)`: No identity element.
  - `reducing(T identity, BinaryOperator<T> accumulator)`: Identity and accumulator.
  - `reducing(U identity, Function<T,U> mapper, BinaryOperator<U> accumulator)`: Mapper, identity, and accumulator.

Example:
java
Copy code

```
Optional<Integer> sum = Stream.of(1, 2, 3, 4, 5)


.collect(Collectors.reducing(Integer::sum));
```

-
- **Result**: `Optional[15]`.

---

13. collectingAndThen()

- **Description**: Applies a finishing function to the result of another collector.

Example:
java
Copy code

```
List<String> unmodifiableList = Stream.of("Alice", "Bob", "Charlie")


.collect(Collectors.collectingAndThen(Collectors.toList(),
Collections::unmodifiableList));
```

-
- **Result**: An unmodifiable `List` of ["Alice", "Bob", "Charlie"].

---

14. mapping()

- **Description**: Applies a function to transform elements before passing them to another collector. It's useful for transforming values while grouping or partitioning.

Example:
java
Copy code
```
Map<Integer, List<Character>> result = Stream.of("Alice", "Bob",
"Charlie")

.collect(Collectors.groupingBy(String::length,

Collectors.mapping(name -> name.charAt(0), Collectors.toList())));
```

-
- **Result**: A `Map` with the lengths of the names as keys and the first characters of the names as values: `{3=[B], 5=[A], 7=[C]}`.

---

15. flatMapping()

- **Description**: Similar to `mapping()`, but it transforms each element into a stream of values and then "flattens" these streams into a single stream.

Example:
java
Copy code
```
List<String> result = Stream.of("Alice", "Bob", "Charlie")

                            .collect(Collectors.flatMapping(s ->
Arrays.stream(s.split("")), Collectors.toList()));
```

-
- **Result**: A `List` of individual characters `["A", "l", "i", "c", "e", "B", "o", "b", "C", "h", "a", "r", "l", "i", "e"]`.

---

16. teeing() (Java 12+)

- **Description**: Combines the results of two collectors into a final result using a merger function.

Example:
java
Copy code
```
int result = Stream.of(1, 2, 3, 4, 5)
```

```java
                    .collect(Collectors.teeing(
                        Collectors.summingInt(i -> i),
                        Collectors.counting(),
                        (sum, count) -> sum / count
                    ));
```

-
- **Result**: `3` (average of 1, 2, 3, 4, 5).

Summary of Key Collector Methods:

- **toList(), toSet(), toMap()**: Convert streams into collections or maps.
- **joining()**: Concatenate stream elements into a string.
- **counting()**: Count the number of elements.
- **summingInt(), averagingInt()**: Summarize numerical data.
- **groupingBy(), partitioningBy()**: Group or partition elements

---

11. CompletableFuture and Asynchronous Programming

Theory:

`CompletableFuture` is a new class in Java 8 that provides a more flexible way to handle asynchronous programming, replacing `Future`. It allows non-blocking operations and chaining of multiple asynchronous operations.

Example:
java
Copy code
```java
CompletableFuture.supplyAsync(() ->
"Hello").thenAccept(System.out::println);
```

Reason:

- **Concurrency Simplification**: `CompletableFuture` simplifies working with asynchronous tasks and makes chaining asynchronous operations easier compared to traditional `Future` and thread handling.

- **Non-blocking**: It allows non-blocking, event-driven programming, making it suitable for modern, highly concurrent applications.

Conclusion

Java 8 brought a significant shift towards **functional programming**, improving the language's capabilities for **parallel processing** and **stream-based data handling**. It also focused on improving **developer productivity** by reducing boilerplate code with **lambda expressions**, **method references**, and **new APIs** for **date/time** and **asynchronous programming**. The motivation behind these changes was to modernize Java, making it competitive with other languages that had already embraced functional programming, parallelism, and immutability.

4o

You said:

in function interface - say there is one default and one static method , will the implimenting class inherit these 2?

ChatGPT said:

ChatGPT

Yes, if an interface contains both a **default method** and a **static method**, the implementing class will inherit the **default method** but **not** the **static method**.

Here's the breakdown:

1. **Default Methods**:
    - **Definition**: A default method in an interface is an instance method with a default implementation. An implementing class inherits this method unless it explicitly overrides it.
    - **Inheritance**: The implementing class **inherits** the default method automatically, and it can use it like any other instance method. The class can also choose to override it if it needs a different behavior.
2. **Static Methods**:
    - **Definition**: A static method in an interface is a method that belongs to the interface itself, not to instances of the implementing classes.
    - **Inheritance**: **Static methods are not inherited** by the implementing class. To access a static method, it must be called on the interface itself, not through an instance of the implementing class.
    - **Usage**: You can access a static method using the **interface name**, like `InterfaceName.staticMethod()`, rather than through an instance of the implementing class.

Example:

java

Copy code

```
interface MyInterface {
    // Default method
    default void defaultMethod() {
        System.out.println("Default method in interface");
    }

    // Static method
    static void staticMethod() {
        System.out.println("Static method in interface");
    }
}

class MyClass implements MyInterface {
    // Inherits defaultMethod() from MyInterface
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();

        // Calling the inherited default method
        obj.defaultMethod(); // Output: Default method in interface

        // Cannot call staticMethod() from the object of MyClass
        // obj.staticMethod(); // Error: Cannot call static method
from instance

        // Correct way to call the static method
        MyInterface.staticMethod(); // Output: Static method in
interface
    }
}
```

Key Points:

- The **default method** is inherited by the implementing class, and you can call it using an instance of that class.
- The **static method** belongs to the interface itself and cannot be called through an instance of the implementing class. It can only be accessed using the interface name.

ava 9 (2017)

1. **Module System (Project Jigsaw)**: Enables modular programming, allowing better encapsulation and dependency management.
2. **JShell (Read-Eval-Print Loop)**: An interactive tool for running and testing small Java code snippets.
3. **Collection Factory Methods**: `List.of()`, `Set.of()`, and `Map.of()` methods to create immutable collections.
4. **Private Interface Methods**: Allowed methods to be implemented within interfaces for code reusability.

Java 10 (2018)

1. **Local-Variable Type Inference**: The `var` keyword was introduced, allowing local variables to be declared without explicit type definitions, making code more concise.
2. **Garbage Collector Interface**: A clean way to allow future changes to garbage collection algorithms.

Java 11 (2018)

1. **New String Methods**: Added methods like `isBlank()`, `strip()`, `lines()`, and `repeat()`.
2. **Lambda Parameter `var` Syntax**: Allows `var` to be used in lambda expressions.
3. **HTTP Client API**: Introduced a new modern HTTP client API to handle HTTP/2 and WebSocket communication.
4. **Single-file Source Code Programs**: You can now run Java source files directly without compilation using `java filename.java`.

Java 12 (2019)

1. **Switch Expressions (Preview)**: Introduced an improved version of `switch`, allowing it to be used as an expression.
2. **Pattern Matching for `instanceof` (Preview)**: Simplifies casting and checking types in `instanceof` checks.

Java 13 (2019)

1. **Text Blocks (Preview)**: A multi-line string literal for easier writing of long text, like HTML or JSON, without needing escape sequences.
2. **Switch Expressions (Second Preview)**: Refined from Java 12, adding support for the `yield` keyword.

Java 14 (2020)

1. **Records (Preview)**: A new way to declare classes meant for immutability with minimal boilerplate.
2. **Pattern Matching for `instanceof` (Preview)**: Continued development of this feature.
3. **NullPointerException Enhancements**: Improved NPE messages with detailed information about what went wrong.
4. **Switch Expressions (Standardized)**: Finalized the improved `switch` structure introduced in earlier versions.

Java 15 (2020)

1. **Text Blocks (Standard)**: Finalized as a standard feature.
2. **Hidden Classes**: Allows the creation of classes that are not discoverable and can be used by frameworks internally.
3. **Records (Second Preview)**: Continued development of the `record` feature.
4. **Pattern Matching for `instanceof` (Second Preview)**: Further refinements.

Java 16 (2021)

1. **Records (Standardized)**: Finalized the `record` class for creating immutable data objects.
2. **Pattern Matching for `instanceof` (Standard)**: Became a full feature, simplifying casting in `instanceof`.
3. **Sealed Classes (Preview)**: A new feature that allows developers to control which other classes or interfaces can extend or implement them.

Java 17 (2021 - LTS)

1. **Sealed Classes (Standard)**: Allows a class to restrict which other classes can extend or implement it.
2. **Pattern Matching for Switch (Preview)**: Extends pattern matching to `switch` expressions and statements.
3. **Foreign Function & Memory API (Incubator)**: Allows Java programs to interoperate with native code.
4. **Deprecation of the `Applets` API**: Applets were finally deprecated.

Java 18 (2022)

1. **Simple Web Server**: A minimal HTTP server for serving static content, useful for prototyping.
2. **Code Snippets in Java API Documentation**: Enhancements for code snippets in Javadoc.
3. **Vector API (Third Incubator)**: Continued progress on the vector API for SIMD operations.

Java 19 (2022)

1. **Virtual Threads (Preview)**: Introduced lightweight threads to simplify concurrency handling.
2. **Structured Concurrency (Incubator)**: A simplified way to manage multiple tasks running in parallel.
3. **Foreign Function & Memory API (Second Incubator)**: Continued enhancements.
4. **Pattern Matching for `switch` (Second Preview)**: Further refinements for the `switch` pattern matching.
5. **Vector API (Fourth Incubator)**: Ongoing progress on the vector operations.

Java 20 (2023)

1. **Scoped Values (Preview)**: Adds the ability to define variables with a defined scope in a multi-threaded environment.
2. **Record Patterns (Second Preview)**: Expands the matching logic for `records`.
3. **Virtual Threads (Second Preview)**: Further work on lightweight virtual threads for high-concurrency scenarios.
4. **Structured Concurrency (Second Incubator)**: Refinements for simplifying task management.

Java 21 (2023 - LTS)

1. **Virtual Threads (Standardized)**: Finalized and available as a standard feature.
2. **Sequenced Collections**: Adds a new interface `SequencedCollection` for ordered lists and maps.
3. **Pattern Matching for `switch` (Standard)**: Finalized the feature, making `switch` more powerful with pattern matching.
4. **String Templates (Preview)**: A feature allowing for embedded expressions within strings, similar to templates in other languages.
5. **Unnamed Classes and Instance Main Methods (Preview)**: A preview feature simplifying the entry point of Java applications.

Future/Upcoming Features

1. **Foreign Function & Memory API**: Likely to become a standard feature, allowing Java programs to interact more easily with native code.
2. **Value Objects**: A project underway (Project Valhalla) aiming to introduce new ways of handling data immutability and efficient memory usage.

These features illustrate the ongoing effort in the Java ecosystem to improve performance, ease of use, and modern development practices.

4o

You said:

Java 8 to 21 features with example

ChatGPT said:

ChatGPT

Here's a breakdown of the major features introduced in **Java 8 to Java 21**, along with code examples for each:

Java 8 (2014)

Lambda Expressions: Enables functional programming by introducing lambda syntax.
java
Copy code
```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(name -> System.out.println(name));
```

1.

Stream API: Provides functional-style operations on streams of elements.
java
Copy code
```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<String> filteredNames = names.stream()
    .filter(name -> name.startsWith("A"))
    .collect(Collectors.toList());
```

2.

Default Methods in Interfaces: Allows methods in interfaces with a default implementation.
java
Copy code
```java
interface Vehicle {
    default void start() {
        System.out.println("Vehicle is starting");
    }
}
```

3.

Optional Class: Helps avoid `NullPointerException` by representing optional values.
java
Copy code
```java
Optional<String> name = Optional.ofNullable("John");
name.ifPresent(System.out::println);
```

4.

Java 9 (2017)

Module System (Project Jigsaw): Introduces modularity into Java applications.
java
Copy code
```
module com.example {
    requires java.base;
    exports com.example.myapp;
}
```

1.

JShell (Read-Eval-Print Loop - REPL): A tool to test Java code snippets interactively.
Example use in JShell:
shell
Copy code
```
jshell> System.out.println("Hello, JShell");
Hello, JShell
```

2.

Factory Methods for Collections:
java
Copy code
```
List<String> names = List.of("Alice", "Bob", "Charlie");
```

3.

Java 10 (2018)

Local-Variable Type Inference (`var`): Allows for simpler variable declarations.
java
Copy code
```
var message = "Hello, Java 10";
System.out.println(message);
```

1.

Java 11 (2018)

New String Methods:
java
Copy code
```
String text = " Hello ";
System.out.println(text.isBlank());  // false
System.out.println(text.strip());    // "Hello"
```

```
System.out.println("Hello\nWorld".lines().count());  // 2
```

1.

HTTP Client API:
java
Copy code

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request =
HttpRequest.newBuilder(URI.create("https://example.com"))
    .GET().build();
HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

2.

Java 12 (2019)

Switch Expressions (Preview):
java
Copy code

```
String day = "MONDAY";
String typeOfDay = switch (day) {
    case "MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY" ->
"Weekday";
    case "SATURDAY", "SUNDAY" -> "Weekend";
    default -> throw new IllegalStateException("Invalid day: " +
day);
};
```

1.

Java 13 (2019)

Text Blocks (Preview): Multiline strings without needing escapes.
java
Copy code

```
String json = """
    {
        "name": "Alice",
        "age": 30
    }
```

```
    """;
```

1.

Java 14 (2020)

Records (Preview): Compact syntax for immutable data classes.
java
Copy code
```
public record Person(String name, int age) {}

Person person = new Person("Alice", 30);
System.out.println(person.name());
```

1.

NullPointerException Enhancements:
java
Copy code
```
String s = null;
s.length();  // Throws NPE with detailed message about the variable
that was null.
```

2.

Java 15 (2020)

Text Blocks (Standard):
java
Copy code
```
String html = """
    <html>
        <body>
            <p>Hello, World!</p>
        </body>
    </html>
    """;
```

1.
2. **Hidden Classes**: Classes that are not discoverable via reflection, useful for frameworks.

Java 16 (2021)

Pattern Matching for `instanceof` (Standard):
java
Copy code
```java
Object obj = "Hello, Java";
if (obj instanceof String s) {
    System.out.println(s.toUpperCase());
}
```

1.

Records (Standard):
java
Copy code
```java
public record Product(String name, double price) {}
Product product = new Product("Laptop", 1200.0);
System.out.println(product.name());
```

2.

Java 17 (2021)

Sealed Classes:
java
Copy code
```java
public abstract sealed class Shape permits Circle, Rectangle {}

public final class Circle extends Shape {}
public final class Rectangle extends Shape {}
```

1.

Pattern Matching for `switch` (Preview):
java
Copy code
```java
static String formatterPattern(Object o) {
    return switch (o) {
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        default        -> o.toString();
    };
}
```

2.

Java 18 (2022)

Simple Web Server:
bash
Copy code
```
java -m jdk.httpserver
```

1.
2. **Code Snippets in JavaDoc**: Supports embedding runnable code in documentation.

Java 19 (2022)

Virtual Threads (Preview):
java
Copy code
```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 10).forEach(i -> executor.submit(() -> {
        System.out.println("Task " + i);
    }));
}
```

1.

Structured Concurrency (Incubator):
java
Copy code
```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Future<String> user = scope.fork(() -> fetchUser());
    Future<Integer> order = scope.fork(() -> fetchOrder());
    scope.join();  // Wait for both
}
```

2.

Java 20 (2023)

1. **Scoped Values (Preview)**: Safe to share between threads.
2. **Record Patterns (Preview)**: Allows matching against `record` components.

Java 21 (2023)

Virtual Threads (Standard):
java
Copy code
```
Thread.startVirtualThread(() -> System.out.println("Hello from 
Virtual Thread!"));
```

1.

Pattern Matching for `switch` (Standard):
java
Copy code
```
static String formatShape(Shape shape) {
    return switch (shape) {
        case Circle c    -> "Circle with radius " + c.radius();
        case Rectangle r -> "Rectangle with area " + (r.length() * 
r.width());
        default -> "Unknown shape";
    };
}
```

2.

Sequenced Collections:
java
Copy code
```
SequencedMap<Integer, String> map = new LinkedHashMap<>();
map.put(1, "One");
map.put(2, "Two");
```

3. `System.out.println(map.firstEntry());  // Output: 1=One`