1. Sealed Classes 🔒

Sealed classes let you control which classes can extend or implement a given class. This enhances security and improves readability by limiting the class hierarchy, making it easier to maintain.

Example:

```
public abstract sealed class Shape permits Circle, Square {}
final class Circle extends Shape {
    double radius;
}
final class Square extends Shape {
    double side;
}
// Now only Circle and Square can extend Shape, and no other class can!
```

Why it matters:

It gives you more control over class hierarchies and prevents unintended extensions of your classes.

---

2. Pattern Matching for `switch` 🔄

This feature simplifies the `switch` statement by enabling pattern matching. You can now test an expression against several patterns, improving code clarity.

Example:

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

```
        switch (obj) {
            case Integer i -> System.out.println("Integer: " + i);
            case String s -> System.out.println("String: " + s);
            default -> System.out.println("Unknown type");
        }
    }
}
```

Why it matters:

Pattern matching makes the code more concise, eliminating the need for verbose `if-else` chains or casting when dealing with different types.

---

3. Text Blocks 📜

Text blocks make it easier to handle multi-line strings, like JSON, XML, or SQL queries, without the hassle of concatenating strings manually.

Example:

```
public class Main {
    public static void main(String[] args) {
        String json = """
                {
                    "name": "Java",
                    "version": 17,
                    "features": [
                        "Sealed Classes",
                        "Pattern Matching",
                        "Text Blocks"
                    ]
                }
```

```
                """;
        System.out.println(json);
    }
}
```

Why it matters:

This feature cleans up string handling, especially for multi-line text. It reduces errors and makes the code much more readable.

---

4. Foreign Function & Memory API (Incubator) 🧠

This API allows Java programs to call native libraries and work directly with memory outside the JVM. It's in the incubator stage, but it's a game-changer for performance-sensitive applications.

Example:

```
import jdk.incubator.foreign.*;

public class Main {
    public static void main(String[] args) {
        try (MemorySegment segment =
MemorySegment.allocateNative(4)) {
            MemoryAccess.setIntAtOffset(segment, 0, 42);
            int value = MemoryAccess.getIntAtOffset(segment, 0);
            System.out.println("Value: " + value); // Output: 42
        }
    }
}
```

Why it matters:

It makes it easier to call native functions and work with memory directly, leading to

performance improvements for certain applications.

---

5. Enhanced `Stream` API 💧

Java 17 builds upon the already powerful `Stream` API, making it more versatile and easy to use. One such feature is the `toList()` method, which simplifies list creation.

Example:

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

public class Main {
    public static void main(String[] args) {
        List<String> names = Stream.of("Alice", "Bob",
"Charlie").toList();
        System.out.println(names); // Output: [Alice, Bob, Charlie]
    }
}
```

Why it matters:

This makes working with streams more convenient, removing the need to convert streams manually into lists.

---

6. Deprecation of `SecurityManager` 🚨

Java 17 has officially deprecated the `SecurityManager`, a long-standing feature that allowed applications to implement a security policy. This change is a sign that the platform is moving toward newer, more modern security practices.

Why it matters:

This marks a shift in how security will be handled in future versions, signaling that

developers should start looking into alternatives.

---

7. Performance Improvements ⚡

Java 17 comes with various JVM performance improvements, from better memory management to faster startup times. These optimizations lead to better runtime performance and reduce the overall footprint of Java applications.

---

8. Records 📄

Introduced in Java 16 and continued in Java 17, **Records** provide a concise way to define immutable data carriers, where you only care about the data they hold, not the behavior.

Example:

```
public record Person(String name, int age) {}

public class Main {
    public static void main(String[] args) {
        Person person = new Person("John", 30);
        System.out.println(person.name()); // Output: John
        System.out.println(person.age());  // Output: 30
    }
}
```

Why it matters:

Records reduce boilerplate code by automatically generating `equals()`, `hashCode()`, `toString()`, and getters for the fields, making data classes much cleaner.

---

9. NullPointerException Improvements 🐞

Java 17 comes with better handling of `NullPointerExceptions`, giving more detailed and helpful messages when a `NullPointerException` occurs.

Example:

```
public class Main {
    public static void main(String[] args) {
        String name = null;
        System.out.println(name.length()); // Throws
NullPointerException with detailed message
    }
}
```

Output:

```
csharp
Copy code
Exception in thread "main" java.lang.NullPointerException: Cannot
invoke "String.length()" because "name" is null
```

Why it matters:

The improved `NullPointerException` messages make debugging faster by pointing out exactly which variable is null.

---

10. `instanceof` Pattern Matching 🔍

Java 17 enhances the `instanceof` operator by introducing pattern matching, which eliminates the need for manual type casting.

Example:

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

```
        if (obj instanceof String s) {
            System.out.println(s.toUpperCase()); // Output: HELLO,
JAVA 17!
        }
    }
}
```

Why it matters:

It simplifies the code by removing the need for separate casting, making `instanceof` checks more readable and concise.

---

11. JEP 356: Enhanced Pseudo-Random Number Generators

Java 17 improves the `Random` API by providing more flexible, stream-based pseudorandom number generators.

Example:

```
import java.util.random.RandomGenerator;
import java.util.random.RandomGeneratorFactory;

public class Main {
    public static void main(String[] args) {
        RandomGenerator generator =
RandomGeneratorFactory.of("Xoshiro256PlusPlus").create();
        generator.doubles(5).forEach(System.out::println);
    }
}
```

Why it matters:

It gives you more control over the generation of random numbers and offers better algorithms, which is useful for simulations, games, and more.

12. Vector API (Incubator) 🧮

The **Vector API** allows you to perform vector computations more efficiently. It's still in an incubator stage, but it opens up new possibilities for high-performance computing.

Example:

```
import jdk.incubator.vector.*;

public class Main {
    public static void main(String[] args) {
        var vectorA = IntVector.fromArray(IntVector.SPECIES_256, new
int[] {1, 2, 3, 4}, 0);
        var vectorB = IntVector.fromArray(IntVector.SPECIES_256, new
int[] {5, 6, 7, 8}, 0);

        var result = vectorA.add(vectorB); // Perform vector addition
        result.intoArray(new int[4], 0);

        System.out.println(result); // Output: [6, 8, 10, 12]
    }
}
```

Why it matters:

The Vector API allows for more efficient computation by leveraging SIMD (Single Instruction, Multiple Data) operations, which can significantly boost performance for numerical applications.

---

13. JEP 382: New macOS Rendering Pipeline 🍏

Java 17 introduced a new rendering pipeline for macOS, called the **Metal API**, replacing the old OpenGL-based pipeline. This results in better graphics performance on macOS systems.

Why it matters:
For developers building desktop applications on macOS, the Metal API offers improved performance and better resource management for graphics rendering.

---

14. Deprecation of `Applets` 🚫

Java 17 deprecates **Applets**, signaling the end of an era. Applets were once used for web-based Java applications, but with the modern web moving to JavaScript frameworks and other technologies, this feature is no longer needed.

Why it matters:
This encourages developers to move towards more modern, secure alternatives for web applications.

---

15. JEP 306: Always-Strict Floating-Point Semantics ➗

Java 17 makes **strict floating-point semantics** the default, which was previously optional. This ensures more predictable and consistent behavior for floating-point calculations across different platforms.

Example:

```
public class Main {
    public static void main(String[] args) {
        double result = 0.1 * 0.2;
        System.out.println(result); // Output: 0.020000000000000004
    }
}
```

Why it matters:

It provides better consistency for numerical operations, especially in scientific and financial applications where precision is critical.