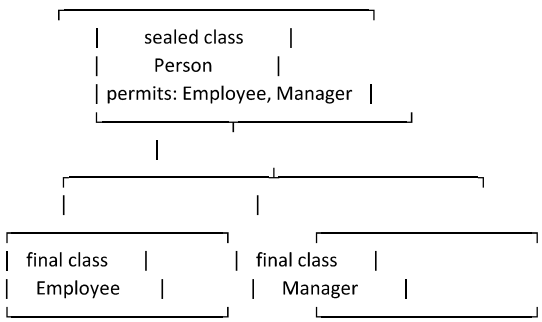
🛑 JEP 409 – Sealed Classes (Finalized in Java 17)

https://www.linkedin.com/in/durgashankarpatra-java-developer/



Theory

Definition:
Sealed classes restrict which other classes or interfaces can extend or implement them.

Why:
Provides better control over class hierarchy and enhances encapsulation.

Real-World Use Case

public sealed class Account permits SavingsAccount, CurrentAccount {}

public final class SavingsAccount extends Account {}
public final class CurrentAccount extends Account {}

```
public final class Employee extends Person {
public Employee(String name) {
super(name);
}

@Override
public void displayRole() {
System.out.println(name + " is an Employee.");
}
}

public final class Manager extends Person {
public Manager(String name) {
super(name);
}

@Override
public void displayRole() {
System.out.println(name + " is a Manager.");
}
}


public sealed class Person permits Employee,
Manager {
String name;

public Person(String name) {
this.name = name;
}

public void displayRole() {
System.out.println("I am a person.");
}
}



Person p1 = new Employe("Alice");
Person p2 = new Manager("Bob");

p1.displayRole(); // Output: Alice is an Employee.
p2.displayRole(); // Output: Bob is an Manager.
```

Question	Sample Answer
Why use sealed classes?	To restrict inheritance, improve design, and allo safe exhaustive handling.
What are the three options for a permitted subclass?	final, sealed, or non-sealed.
Can interfaces be sealed?	Yes, sealed interfaces are allowed from Java 15 onwards (finalized in 17).
Can sealed classes work with packages?	Yes. Subclasses must be in the same module or package.

Switch Expressions in Java (Standard from Java 14)

https://www.linkedin.com/in/durgashankarpatra-java-developer/

A **switch expression** is an enhancement to the traditional switch statement that:

- Allows switch to **return a value**
- Provides **concise syntax**
- Supports both **arrow (->)** and **block** forms
- Eliminates the need for break to prevent fall-through

Feature	Benefit
✓ Arrow syntax	No need for break or fall-through
✓ Expression form	switch can return values directly
✓ Pattern matching ready	Works well with pattern matching (future Java versions)
✓ yield keyword	Return value from multi-line blocks in switch cases

Interview Q&A

Q: What is the difference between switch statement and switch expression?
A: A switch expression can return a value, uses concise syntax (->), and avoids fall-through without needing break.
Q: What is yield used for?
A: yield is used to return a value from a block-style case inside a switch expression.

```
public class EmployeeApplication {

static List<Employee> employees = Arrays.asList(new Employee(1,
"Alice", "IT", 75000, 1),
new Employee(2, "Bob", "HR", 55000, 1), new Employee(3, "Charlie",
"IT", 50000, 0),
new Employee(4, "David", "Finance", 60000, 1));

public static void main(String[] args) {
for (Employee emp : employees) {
String level = switch (emp.getDepartment()) {
case "IT", "Engineering" -> "Technical";
case "HR", "Admin" -> "Support";
case "Finance" -> "Management";
default -> "General";
};

System.out.println(emp.getName() + " works in " +
emp.getDepartment() + " - Department Level: " + level);
}
```

Using Yield keyword

```
for (Employee emp : employees) {
        String level = switch (emp.getDepartment()) {
          case "IT", "Engineering" -> {
            System.out.println(emp.getName() + " is in a technical role.");
            yield "Technical";
          }
          case "HR", "Admin" -> {
            System.out.println(emp.getName() + " is in a support role.");
            yield "Support";
          }
          case "Finance" -> {
            System.out.println(emp.getName() + " handles money matters.");
            yield "Management";
          }
          default -> {
            System.out.println(emp.getName() + " has an unknown department.");
            yield "General";
          }
        };
```

A **switch expression** in Java is an enhancement over the traditional switch statement. It allows switching over values with a concise and safer syntax, supports returning results directly, and enables multi-line logic via yield. Introduced in Java 14 (preview), it became a **standard feature in Java 17**.

Old Switch Statement (Pre-Java 14)

```
String level;
switch (employee.getDepartment()) {
   case "IT":
   case "Engineering":
      level = "Technical";
      break;
   case "HR":
   case "Admin":
      level = "Support";
      break;
   case "Finance":
      level = "Management";
      break;
   default:
      level = "General";
      break;
}
```

Requires break to avoid fall-through
More verbose and error-prone
Cannot return values directly

```
String level = switch (employee.getDepartment()) {
   case "IT", "Engineering" -> "Technical";
   case "HR", "Admin"      -> "Support";
   case "Finance"          -> "Management";
   default                 -> "General";
};
```

Returns value directly
No need for break
Safer, more readable, concise syntax

Feature	Old Switch Statement	New Switch Expression (Java 14+)
Returns a value?	✗ No	✓ Yes
Arrow syntax (->) support	✗ No	✓ Yes
Requires break?	✓ Yes	✗ No
Fall-through by default?	✓ Yes	✗ No (arrow syntax avoids it)
Use in expressions (assignments)?	✗ No	✓ Yes (can assign switch result)
Supports yield keyword	✗ No	✓ Yes (in block cases)

```
Switch (Expression or Statement)
|
├── Traditional Switch (Statement) [Old]
|   ├── Uses `case` and `break`
|   ├── Does NOT return a value
|   └── Supports fall-through
|
└── Switch Expression (Java 14+, Std in Java 17)
    ├── Returns a value (can be assigned)
    ├── Uses `->` arrow syntax
    ├── Supports `yield` for blocks
    ├── Prevents fall-through
    └── Can be used with:
        ├── Strings
        ├── Integers
        ├── Enums
        └── Pattern Matching (Java 21+)
```

Arrow Label (Single-

```
String result =
switch (value) {
   case 1 -> "One";
   case 2 -> "Two";
   default ->
"Other";
};
```

Pattern Matching for instanceof(Java 16 → Standard in Java 17)

https://www.linkedin.com/in/durgashankarpatra-java-developer/

Definition:
Pattern Matching for instanceof simplifies type checks and casting in Java. It allows you to test if an object is an instance of a specific type and, at the same time, **bind a variable** to that object without needing an explicit cast.

Feature	Traditional instanceof	Pattern Matching instanceof
Type check	✅ Yes	✅ Yes
Manual cast required	✅ Yes	❌ No
Readability	❌ Verbose	✅ Cleaner
Java Version	All versions	Java 16+ (standard in Jav 17)
Variable Scope	Must declare outside if	Declared inline, scoped t

Concept	Details
🔷 Feature	Pattern Matching for instanceof
🏁	Java 14 (preview), **Standard in Java 16/17**
❓ Purpose	Combines instanceof check and cast in a single expression
✅ Benefits	Less boilerplate, type-safe, easier to read
❓ Scope	Pattern variable (var) only exists inside if block
💼 Real Use	Type-based logic in polymorphic class hierarchies

//Pattern matching (Java 16+)

```
public class EmployeeApplication {

   static List<Employee> employees = Arrays.asList(
         new Employee(1, "Alice", "IT", 75000, 1),
         new Manager(2, "Bob", "HR", 85000, 1, "Senior"),
         new Employee(3, "Charlie", "IT", 50000, 0),
         new Manager(4, "David", "Finance", 95000, 1, "Mid")
   );

   public static void main(String[] args) {

      for (Employee emp : employees) {
         if (emp instanceof Manager mgr) {
            System.out.println(mgr.getName() + " is a Manager at level: " + mgr.getLevel());
         } else {
            System.out.println(emp.getName() + " is a regular employee.");
         }
      }

      SpringApplication.run(SpringBootEmployeeApplication.class, args);
   }
}
```

```
Alice is a regular employee.
Bob is a Manager at level: Senior
Charlie is a regular employee.
David is a Manager at level: Mid
```

Advantages:
- No casting required
- Cleaner, more readable
- Less error-prone

Traditional Way (Java 8 and earlier)

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

      if (obj instanceof String) {
         String str = (String) obj;  // Manual casting required
         System.out.println(str.toUpperCase());
      }
   }
}
```

//This is the fully traditional way — it works in Java 8 and does not use any newer features like pattern matching or type inference inside instanceof.

```
import java.util.Arrays;
import java.util.List;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class SpringBootEmployeeApplication {

   static List<Employee> employees = Arrays.asList(
      new Employee(1, "Alice", "IT", 75000, 1),
      new Manager(2, "Bob", "HR", 85000, 1, "Senior"),
      new Employee(3, "Charlie", "IT", 50000, 0),
      new Manager(4, "David", "Finance", 95000, 1, "Mid")
   );

   public static void main(String[] args) {
      for (Employee emp : employees) {
         if (emp instanceof Manager) {
            Manager mgr = (Manager) emp;
            System.out.println(mgr.getName() + " is a Manager at level: " + mgr.getLevel());
         } else {
            System.out.println(emp.getName() + " is a regular employee.");
         }
      }

      SpringApplication.run(SpringBootEmployeeApplication.class, args);
   }
}
```

Pattern Matching (Java 16+, finalized in Ja 17)

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

      if (obj instanceof String str) {  // Pattern matching: no need to cast
         System.out.println(str.toUpperCase());
      }
   }
}
```

✅ **Works**, but:
- Repeats the type name
- Requires **explicit cast**, increasing verbosity and risk

◆ Text Blocks in Java

https://www.linkedin.com/in/durgashankarpatra-java-developer/

Definition:
Text blocks, introduced in **Java 13** as a preview feature and **standardized in Java 15**, are a new way to represent multi-line string literals in Java. They are designed to improve readability and maintainability of string literals that span multiple lines, such as JSON, HTML, SQL queries, or XML content.
Text blocks eliminate the need for concatenation or escape sequences, making code more concise and easier to work with, especially for large strings.
Syntax:
Text blocks are enclosed by three double-quote marks ("""), and they automatically handle newline characters and indentation for you.

Feature	Traditional Approach	Latest Approach (Text Blocks)
Newlines	Requires \n for newlines	Automatically preserved
String Concatenation	Needs manual concatenation using +	No concatenation requir
Readability	Harder to read with multiple lines	Clean, easy to read and maintain
Formatting	Difficult to preserve indentation	Indentation is automatic handled
Java Version	Java 8 and earlier	Java 15 and above

Key Benefits:

1. **Multiline Support**: You can represent strings spanning multiple lines without needing to use escape characters for newlines (\n).
2. **Preserved Formatting**: Text blocks automatically maintain the formatting of the content, including leading and trailing whitespace.
3. **Improved Readability**: It's easier to read and maintain multiline strings, especially when dealing with JSON, HTML, or SQL queries.

Traditional Approach (Java 8 and earlier)

In the traditional approach, you need to use escape sequences like \n for newlines.

```
public class Example {
    public static void main(String[] args) {
        String address = "John Doe\n" +
                         "1234 Elm St.\n" +
                         "Springfield, IL 62701\n" +
                         "USA";

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

```
John Doe
1234 Elm St.
Springfield, IL 62701
USA
```

Explanation:

- You need to concatenate the string parts using + and manually include \n for line breaks.
- This approach can be cumbersome and hard to read with large strings.

Latest Approach (Text Blocks - Java 15+)

With the introduction of text blocks, you can write multi-line strings directly, and Java will automatically handle the line breaks and indentation for you.

```
public class TextBlock {
    public static void main(String[] args) {
        String address = """
            John Doe
            1234 Elm St.
            Springfield, IL 62701
            USA
            """;

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

Explanation:

- The """ syntax allows you to write the string exactly as it appears, including line breaks.
- There's no need for \n or string concatenation.
- The formatting and indentation are automatically preserved, making it more readable.

306: Restore Always-Strict Floating-Point Semantics

https://www.linkedin.com/in/durgashankarpatra-java-developer/

What is JEP 306?

JEP 306 was introduced in **Java 17**.

Before Java 17

```
double a = 1.0 / 10.0;
double b = a * 10;
System.out.println(b == 1.0); // Might be false on some comput
```

After Java 17

```
double a = 1.0 / 10.0;
double b = a * 10;
System.out.println(b == 1.0); // Always true!
```

➤ Purpose:

It **restores strict floating-point semantics by default**, making the behavior of floating-point operations consistent across platforms and compilers by always applying strictfp.

Interview Question

Q: Explanation of JEP 306 (Java 17) and What Java 17 (JEP 306) does ?

JEP 306 – Simple Explanation

In Java 17, floating-point math (float and double) always gives the **s**
result on every computer.
Before Java 17, the result could be **slightly different** on different computers.

Answer Summary

In older versions of Java, floating-point math (like with float or double) could give slightly different results on different computers.
This is because some computers used extra precision (more decimal places), which could cause small differences in the answers.

It makes sure that floating-point math always works **the same way on all computers**, by following strict rules — just like writing strictfp, but now it's **automatic**.

JEP 356 – Simple Random Number Upgrade

https://www.linkedin.com/in/durgashankarpatra-java-developer/

Before Java 17: You only had a couple of random generators like Random and SecureRandom.
In Java 17: You get **more powerful and customizable** random generators. You can even choose different algorithms based on your needs (faster, more secure, or repeatable).

Before Java 17 – Old Way

```
import java.util.Random;

public class OldRandomExample {
   public static void main(String[] args) {
      Random random = new Random();
      int number = random.nextInt(100);
      System.out.println("Old random: " + number);
   }
}
```

Explanation:

- This uses a basic random number generator.
- You can't choose or change the algorithm easily.
- Only limited options.

Java 17 – New Way

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

public class NewRandomExample {
   public static void main(String[] args) {
      RandomGenerator rng =
RandomGeneratorFactory.of("Xoroshiro128PlusPlus").create();
      int number = rng.nextInt(100);
      System.out.println("New random: " + number);
   }
}
```

Explanation:

- RandomGeneratorFactory.of("Xoroshiro128PlusPlus") lets you
  a **specific algorithm**.
- You get more **control**, **better performance**, or **more randomn**
  depending on what you choose.
- Works the same way as Random, but more flexible.

Feature	Old (Before Java 17)	New (Java 17 – JEP 356)
Class used	Random, SecureRandom	RandomGenerator, RandomGeneratorFactory
Custom algorithms	✗ Not possible	✓ Yes, choose from many algorithms
Flexibility	✗ Limited	✓ Highly customizable
Performance tuning	✗ No	✓ Yes, via algorithm choice

🍎 Java 17 – JEP 382: New macOS Rendering Pipeline

https://www.linkedin.com/in/durgashankarpatra-java-develo

In **Java 17**, Java on macOS uses **Metal** instead of **OpenGL** to draw graphics.

Why?

- **OpenGL** is old and Apple doesn't support it anymore.
- **Metal** is fast, modern, and works better on new Macs (like M1, M2).

Interview Question (Simple)

Q: What does JEP 382 do in Java 17?
A: It makes Java use Apple's **Metal** graphics system instead of old **OpenGL** on macOS, so graphics run faster and more reliably — especially on newer Macs.

What Changed for Developers?

- **Nothing to change in your code!**
- Java apps (like Swing or JavaFX) will just run **faster and smoother** on macOS.

Feature	Old Pipeline (Before Java 17)	New Pipeline (Java 17 - JEP 382)
Graphics Engine Used	OpenGL	Metal
Supported on Apple Silicon	Partially	✅ Full support
Future compatibility	❌ Risk (OpenGL is deprecated)	✅ Future-proof (Metal is supported)
Developer changes needed	❌ Yes (if OpenGL fails)	✅ None — automatic switch
Performance	May be slower on M1	✅ Improved on newer Macs

Before Java 17:

Java used **OpenGL**
Apple deprecated OpenGL → not future-safe

After Java 17:

Java uses **Metal**
🚀 Faster and works better on new Macs

Swing vs JavaFX (UI Frameworks)

✅ What is Swing?

- **Swing** is an older Java framework for building **graphical user interfaces (GUIs)**.
- It lets you create buttons, text fields, and other UI elements.
- It's **cross-platform** (works on Windows, macOS, Linux) but feels a bit **dated** compared to modern UI frameworks.

What is JavaFX?

- **JavaFX** is a **newer** Java framework for building **modern UIs**.
- It supports **rich graphics**, animations, and modern UI features.
- JavaFX has better support for things like **3D graphics** and **media playback**.

What is Metal?

- **Metal** is Apple's **modern graphics system**.
- It's **faster** and **more efficient** than older systems.
- Metal is used for drawing graphics, rendering animations, and handling video in **macOS** and **iOS** apps.

What is OpenGL?

- **OpenGL** is an older **cross-platform graphics system**.
- It works on many different operating systems (macOS, Windows, Linux).
- **Apple deprecated OpenGL** because it's old and less efficient, and they now prefer **Metal** on macOS.

JEP 391 – macOS/AArch64 Port

https://www.linkedin.com/in/durgashankarpatra-java-developer/

JEP 391 makes **Java run faster** on **new Macs** that use **Apple's M1, M2**, and other **Apple Silicon chips**.

Why This is Important?

- Older Macs used **Intel chips** (x86 architecture).
- New Macs use **Apple Silicon** (AArch64 architecture), which is different.
- **JEP 391** ensures **Java works great** on these new Macs.

Feature	Old Macs (Intel)	New Macs (Apple Silicon - AArch64)
Chip Architecture	Intel x86-64	Apple Silicon (AArch64)
Java Support	Supported but no optimizations	Fully optimized with JEP 39
Performance	Slower on Apple Silicon	Faster, better performance

What Changed?

- **Java now runs natively** on Apple Silicon, making it **faster and more efficient**.

JEP 398 – Deprecate the Applet API for Removal

https://www.linkedin.com/in/durgashankarpatra-java-developer/

What is JEP 398?

JEP 398 marks the **deprecation of the Applet API** in Java. This means that the **Applet** technology, which was once used to run small applications inside web browsers, is now officially outdated and will be removed in the future.

Why?

- **Applets** were used to run small apps inside web browsers, but they are **old** and **no longer supported** by most browsers.
- **JEP 398** says that Applets will be **removed** from Java soon.

What Does This Mean?

- **If you're using Applets**, you need to stop and **move to newer technologies** (like **JavaFX** or web tools like **HTML5**).
- **Applets** are officially **deprecated** (they will be removed in the future).

In Short:

- **Applets** are old and don't work well anymore.
- **JEP 398** is making sure Applets will be **removed** from J
- Time to switch to **newer, better tools** for web apps.

🔒 JEP 403 – Strongly Encapsulated JDK Internals

https://www.linkedin.com/in/durgashankarpatra-java-developer/

What is JEP 403?

JEP 403 makes **internal JDK classes** (classes used by Java itself) **more secure** by hiding them from public access. These classes were previously accessible through reflection (a way to access private parts of Java), but now they are **strictly hidden**.

Before JEP 403 (Accessing JDK Internals Using Reflection)

In earlier versions of Java, you could use **reflection** to access private internal classes or methods in the JDK, even if they were not meant for public use.

Before JEP 403

```
import sun.misc.Unsafe;

public class UnsafeExample {
    public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException {
        // Accessing the internal "Unsafe" class using reflection (not recommended!)
        Unsafe unsafe = Unsafe.getUnsafe();
        System.out.println("Unsafe: " + unsafe);
    }
}
```

- Here, the Unsafe class is an internal JDK class that could be accessed with reflection.
- This was **dangerous** because the Unsafe class provides low-level memory operations that could cause issues if misused.

After JEP 403 (Encapsulated JDK Internals)

With **JEP 403**, accessing these internal classes (like Unsafe) directly is **blocked**. The JDK **encapsulates** them, so developers can no longer access them directly.

After JEP 403

```
public class UnsafeExample {
    public static void main(String[] args) {
        // This code will no longer work because internal classes are now strongly
encapsulated
        // Unsafe unsafe = Unsafe.getUnsafe();
        System.out.println("Access to Unsafe class is blocked.");
    }
}
```

Why Did This Change?

- **Security**: Stopping access to these secret classes **protects** your program from potentially dangerous code.
- **Stability**: It prevents developers from using parts of Java that could break or crash their programs.

In Short:

- **JEP 403** locks away Java's hidden parts.
- It makes Java **safer** and **more stable** by preventing dangerous access.

JEP 407 – Removal of RMI Activation

https://www.linkedin.com/in/durgashankarpatra-java-developer/

What is JEP 407?

JEP 407 removes **RMI Activation** from Java. **RMI (Remote Method Invocation)** is a technology that allows Java programs to call methods on objects that are on **different machines** (remotely). **RMI Activation** allowed RMI to start objects automatically when needed.

Why is this happening?

- **RMI Activation** was used to start objects remotely, but it's **old** and not used much anymore.
- **Better technologies** (like **REST APIs** and **gRPC**) have replaced it.

What Does It Mean for You?

- If you're still using **RMI Activation**, you'll need to switch to **modern ways** of handling remote calls, like **REST** or **gRPC**.

In Short:

- **JEP 407** removes an outdated feature.
- Time to use **newer technologies** for remote communication.

Feature	RMI Activation	REST API (New Way)
Technology	RMI (Remote Method Invocation)	REST (Representational State Transfer)
How Objects are Started	Automatically started through RMI Activation	Client sends HTTP requests to the server
Usage	Old and rarely used now	Modern and widely used for web services

Before JEP 407 - Using RMI Activation

```
import java.rmi.*;
import java.rmi.activation.*;

public class HelloService implements Remote {
   public String sayHello() {
      return "Hello, RMI!";
   }

   public static void main(String[] args) throws Exception {
      // Register the object with the RMI registry
      HelloService service = new HelloService();
      ActivationGroup group = new ActivationGroup();
      ActivationDesc desc = new ActivationDesc("HelloService", grc
      Activator.activate(desc); // RMI Activation starts the object
   }
}
```

After JEP 407

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.web.bind.annotation.*;

@SpringBootApplication
public class HelloService {

   public static void main(String[] args) {
      SpringApplication.run(HelloService.class, args);
   }

   @RestController
   class HelloController {
      @GetMapping("/hello")
      public String sayHello() {
         return "Hello, REST!";
      }
   }
}
```

🛑 JEP 410 – Removal of the Experimental AOT and JIT Compiler

https://www.linkedin.com/in/durgashankarpatra-java-developer/

What is JEP 410?

JEP 410 removes old, **experimental compilers** (AOT and JIT) from Java.

Why Is This Happening?

- These compilers were **experimental**, meaning they were not fully developed or widely used.
- **Java has moved on** to better ways of optimizing performance.

What Does It Mean for You?

- If you used these old compilers, you'll need to **use other optimizations** provides now.
- **Java's focus** is now on **newer** and **better optimization methods**.

In Short:

- **JEP 410** removes **unused old compilers**.
- Java now uses **better ways** to improve performance.

411: Deprecate the Security manager for Removal

https://www.linkedin.com/in/durgashankarpatra-java-developer/

What is JEP 411?

JEP 411 is about **deprecating** the **Security Manager** in Java for **removal** in future versions.

Why is this Happening?

- The **Security Manager** was used to protect Java applications by enforcing **security policies**, like restricting file access or network connections.
- Over time, it has become **harder to maintain** and **not widely used**.
- Java now has **better security features** (like the **Java Module System** and **new security APIs**).

What Does It Mean for You?

- If your code relies on the **Security Manager**, it will eventually **be removed** from Java.
- You'll need to **migrate** to **newer security methods** that Java offers.

In Short:

- **JEP 411** marks the **Security Manager** for removal in future versions of Java.
- Java now has **better and more modern** security features.

Before JEP 411

```
public class Example {
    public static void main(String[] args) {
        // Setting a security manager
        System.setSecurityManager(new SecurityManager());

        // Trying to access a restricted action
        System.out.println("Attempting restricted action...");
        System.exit(1);  // This might be blocked by the Security Manager
    }
}
```

The SecurityManager would block potentially harmful operations (like file or network ac
depending on the security policy you set.

```
// No SecurityManager in the new Java versions
public class Example {
    public static void main(String[] args) {
        // No Security Manager - Java now relies on modules and other security features
        System.out.println("No security manager, use modules for security.");
    }
}
```

- Instead of relying on the **Security Manager**, Java now focuses on **modules** and **othe
  tools to control access and improve security.

🛠 JEP 412 – Foreign Functions & Memory API (Incubator)

https://www.linkedin.com/in/durgashankarpatra-java-developer/

What is JEP 412?

JEP 412 lets Java programs interact with **native code** (like C or C++) and manage **memory** directly, outside the Java heap. This helps Java work with faster, low-level code that's not part of the Java Virtual Machine (JVM).

Why is it Useful?

- Java normally works with memory managed by the **JVM**, but sometimes you need to use **native libraries** (written in C/C++) or manage memory directly for **performance** reasons.
- JEP 412 gives you a **safe and easier way** to do this without using complicated methods like **JNI**.

What Does it Mean for You?

- If you need to **use C libraries** or directly handle **memory** in Java, this API makes it **easier** and **safer**.
- This is still in the **Incubator** phase, meaning it's a **preview feature** that may change in future Java versions.

In Short:

- **JEP 412** helps Java interact with **native code** (C/C++) and manage **memory** directly.
- It's a **new** and **easier** way to handle low-level operations in Java.

Step 1: C Code

First, you write a simple C function. Let's say it adds two numbers.

```
// C code (add.c)
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}
```

This C code defines a function add() that adds two numbers. You compile it into a native library, like libnative.so (on Linux) or native.dll (on Windows).

Step 2: Java Code Using JEP 412

Now, in Java, we will use the **Foreign Function & Memory API** to call the add() function from C.

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

public class ForeignFunctionExample {
    public static void main(String[] args) {
        // Open a memory session
        try (var session = MemorySession.openConfined()) {
            // Load the native library (compiled C code)
            var library = Library.lookup("libnative");

            // Look up the 'add' function from the C library
            var addFunction = library.lookup("add");

            // Call the C function 'add' with 5 and 3 as arguments
            int result = addFunction.invoke(5, 3);

            // Print the result
            System.out.println("Result from C function add: " + result);  // Should pr
        }
    }
}
```

How It Works:

- **Library.lookup("libnative")**: Finds and loads the compiled native library (libnative.so or native.dll).
- **lookup("add")**: Finds the add() function from the C library.
- **invoke(5, 3)**: Calls the add() function with the numbers 5 and 3.

🛠 JEP 414 – Vector API (Second Incubator)

https://www.linkedin.com/in/durgashankarpatra-java-developer/

What is JEP 414?

JEP 414 adds a **Vector API** in Java, which helps Java programs do **fast calculations** on large amounts of data (like arrays of numbers) by using **special instructions** in your computer's processor (CPU).

Why is it Useful?

- Modern processors (CPUs) can process multiple data points at once, making things like **adding two arrays** of numbers much **faster**.
- With the **Vector API**, Java can take advantage of these **fast CPU instructions** to do the work in **less time**.
- This feature is still **experimental** (in an incubator phase) but is usable.

In Short:

- **JEP 414** gives Java the ability to do faster math on large datasets by using **special CPU instructions**.
- It's still **new**, so it might change later.

Example of Array Addition

Without Vector API (Normal Java):

Here's how you would usually add two arrays in Java:

```
public class ArrayAddition {
   public static void main(String[] args) {
      int[] a = {1, 2, 3, 4};
      int[] b = {5, 6, 7, 8};
      int[] result = new int[4];

      for (int i = 0; i < 4; i++) {
         result[i] = a[i] + b[i];
      }

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

This method adds numbers one at a time, which isn't the most **efficient**.

Why is This Better?

- **Faster operations**: The CPU can add multiple numbers **at the same time**.
- **Less code**: You can use the **Vector API** to write faster code without needing to use complex techniques.

With Vector API (Faster Addition):

Using the **Vector API**, the addition can happen **faster** because the CPU can add multiple numbers a same time:

```
java
CopyEdit
import jdk.incubator.vector.*;
public class VectorAddition {
   public static void main(String[] args) {
      int[] a = {1, 2, 3, 4};
      int[] b = {5, 6, 7, 8};
      int[] result = new int[4];

      var vectorA = IntVector.fromArray(VectorSpecies.of(4), a, 0);
      var vectorB = IntVector.fromArray(VectorSpecies.of(4), b, 0);
var vectorResult = vectorA.add(vectorB);

      vectorResult.intoArray(result, 0);

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

- The CPU can now **add all 4 numbers** at once, making it **faster**.

🛠️ JEP 415 – Context-Specific Deserialization Filters

https://www.linkedin.com/in/durgashankarpatra-java-developer/

What is JEP 415?

JEP 415 makes **deserialization** in Java safer by allowing you to define **filters** that control which objects can be deserialized. **Deserialization** means converting data back into Java objects, like reading objects from a file or network.

Why is it Useful?

- **Deserialization** can be risky if you're reading data from an **untrusted source**, as it might create harmful or malicious objects.
- **JEP 415** allows you to specify **rules** to only allow safe objects to be deserialized, helping to prevent security problems.

In Short:

- **JEP 415** lets you control what types of objects can be deserialized, making it **safer** to handle data from untrusted sources.

Example:

Without Filter (Unsafe):

Imagine you deserialize data without any checks. This could let harmful data cause problems.

```
ObjectInputStream in = new ObjectInputStream(new
FileInputStream("data.ser"));
Object object = in.readObject();  // No checks on what is being
deserialized.
```

With Filter (Safe):

With **JEP 415**, you can **filter** what objects are allowed to be deserialized. For example, only allowing a class called SafeClass

```
ObjectInputStream in = new ObjectInputStream(new
FileInputStream("data.ser")) {
  @Override
  protected ObjectStreamClass readClassDescriptor() {
    ObjectStreamClass desc = super.readClassDescriptor();
    // Allow only 'SafeClass' to be deserialized
    if ("com.example.SafeClass".equals(desc.getName())) {
      return desc;
    }
    throw new InvalidClassException("Unauthorized class: " +
desc.getName());
  }
};
```

This ensures that **only safe objects** (like SafeClass) are deserial blocking anything dangerous.

Why is this Better?

- **JEP 415 prevents** deserializing unsafe or malicious object making your program **more secure**.