Java 17 & 21 features

By

Anand Kulkarni

anand.kulkarni1@zensar.com

Table of Content

Introduction to Java 17 & 21 features

Java offers below Long Term Support(LTS) versions

- 1.8
- 11
- 17
- 21

Introduction to Java 17 & 21 features

Java adds several features in every release. However, these features are divided into below categories:

- **Incubator feature:** Incomplete feature
- **Preview feature:** Complete feature but Java asking for developer's feedback & will be available in future release.
- **Finalized feature:** Fully operational feature & ready for use.

Sealed classes

Sealed classes basics

- Sealed classes in Java, introduced in Java 15 as a preview feature and finalized in Java 17.
- Sealed classes provide a way to control inheritance more precisely.
- Sealed classes allow developers to specify which classes or interfaces are permitted to extend or implement a given class or interface.
- Sealed classes feature is particularly useful for designing more predictable and secure class hierarchies.

Key Features of Sealed Classes

- **Inheritance Control:** Only explicitly permitted classes can extend or implement a sealed class or interface.
- **Improved Readability:** Makes the class hierarchy more understandable by clearly defining the allowed subtypes.
- **Compile-Time Safety:** Prevents unintended extensions, reducing potential bugs.

Sealed classes example

sealed *class Employee* ***permits*** *Manager { }*

sealed *class Manager extends Employee* ***permits*** *SalesManager {} //non-sealed, sealed, final*

non-sealed *class SalesManager extends Manager { } //non-sealed, sealed, final*

class AsianSalesManager extends SalesManager {} //Ordinary class

Permitted Subclasses

The subclasses of a sealed class must be one of the following:

final: *Prevent further subclassing.*

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

sealed: *Further restrict inheritance.*

```
public sealed class Rectangle extends Shape permits FilledRectangle {
}
```

non-sealed: *Allow unrestricted subclassing.*

```
public non-sealed class Square extends Shape {
}
```

Record classes

Record classes

- Record classes are a special kind of class introduced in Java 14 (as a preview feature) and made stable in Java 16.
- They are designed to simplify the creation of immutable data carrier classes, reducing boilerplate code.

Features of record classes

- **Immutable Data:** Fields in a record are final by default, ensuring immutability.
- **Compact Syntax:** Automatically generates boilerplate code like constructors, equals(), hashCode(), and toString() methods.
- **Purpose:** Ideal for modeling plain data aggregates, such as DTOs (Data Transfer Objects) or POJOs (Plain Old Java Objects).
- **Syntax:**

  public record RecordName(Type field1, Type field2, ...) {}

Example of record classes

```
public record Person(String name, int age) {
    // Additional methods or static fields can be added if needed
}
public class Main {
    public static void main(String[] args) {
        Person person = new Person("Alice", 30);
    // Access fields
    System.out.println(person.name()); // Output: Alice
    System.out.println(person.age());  // Output: 30
```

Example of record classes

```
    // Auto-generated toString()
    System.out.println(person); // Output: Person[name=Alice, age=30]


    // Auto-generated equals() and hashCode()
    Person anotherPerson = new Person("Alice", 30);
    System.out.println(person.equals(anotherPerson)); // Output: true
}
}
```

Benefits of record classes

- Less Boilerplate: No need to manually write getters, toString(), equals(), or hashCode().
- Readability: Code is concise and easier to understand.
- Immutability: Encourages immutable design by default.

Limitations of record classes

- **Immutable Fields:** Fields are implicitly final, so they cannot be modified after object creation.
- **No Inheritance:** Records cannot extend other classes (but can implement interfaces).
- **Not for Complex Logic:** Records are best suited for simple data carriers, not for classes with complex behavior.

Pattern matching using switch cases

Pattern Matching for switch

- Java 17 introduced Pattern Matching for switch, which enhances the flexibility and readability of switch statements by allowing patterns to be matched directly.
- No need of “break” statement in switch cases. Use arrow instead.
- Switch can now return a value directly, making it more functional.
- Type Matching: You can match the type of an object directly in a case.
- Guarded Patterns: Add conditions to patterns using when.
- Enhanced Readability: Reduces boilerplate code compared to traditional instanceof checks.

No need of ‘break’

```
String day = "Sunday";

switch(day) {

        case "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" ->
System.out.println("Working day " + day);

        case "Saturday" -> System.out.println("Half day Saturday");

        case "Sunday" -> { System.out.println("Holiday"); System.out.println("Fun day");}

        default -> System.out.println("Invalid day");

}
```

Switch can return a value

```
String day = "Sunday";

String message = switch(day) { //switch varible can be number, string or enum

        case "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" -> "Working day";

        case "Saturday" -> "Half day Saturday";

        case "Sunday" -> "Holiday and Fun day";

        default -> { yield "Invalid day"; } //yield is equivalent to the return statement

};

System.out.println("returned message = " + message);
```

Switch Type Matching

```
Object obj = new String(“Hello”);

switch(obj) {

        case Integer a -> System.out.println("Integer variable " + a);

        case String str -> System.out.println("String variable " + str);

        default -> System.out.println("Invalid type");

}
```

Guarded Patterns: Add conditions to patterns

```
switch(obj) {
        case Integer a when a.intValue()>100 -> System.out.println("Integer variable>1000 " + a);
        case Integer a when a.intValue()<=100 -> System.out.println("Integer variable<=1000 " + a);
        case String str -> System.out.println("String variable " + str);
        default -> System.out.println("Invalid type");
}
```

Basic Record Pattern Matching

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

record Circle(double radius) {}

record Rectangle(double length, double width) {}

record Triangle(double base, double height) {}

Point point = new Point(3, 2);

if(point instanceof Point(int x, int y)) {

        System.out.println("x = " + x + " & y = " + y);

}
```

Record Patterns in switch Statements

```
switch(obj) {
        case Circle(double radius) -> System.out.println("Circle radius " + radius);
        case Rectangle(double length, double width) -> System.out.println("Rectangle " +
length + " - " + width);
        case Triangle(double base, double height) -> System.out.println("Triangle " + base + " -
" + height);
        default -> System.out.println("Not found");
}
```

Nested Record Patterns for complex de-structuring

```
record Line(Point start, Point end) {}

Line line = new Line(new Point(1, 2), new Point(3, 4));

if(line instanceof Line(Point(int x, int y), Point(int p, int q))) {

        System.out.println("x = " + x + "\ty = " + y + "\t p = " + p + "\tq = " + q);

}
```

Record patterns with when clauses

```
Point point = new Point(5, 3);

switch(point) {

        case Point(int x, int y) when x>y -> System.out.println("x > y");

        case Point(int x, int y) when x<y -> System.out.println("x < y");

        default -> System.out.println("x==y");

}
```

String templates

String Template Basics

- Java 21 introduced String Templates, a preview feature that simplifies string interpolation, making it easier to embed expressions directly into strings.
- This feature enhances readability and reduces the need for concatenation or String.format().
- Text Blocks: It allows us to write multi line strings.
- Template Expressions: Use ${} to embed expressions directly into strings.
- Compile-Time Safety: Expressions are validated at compile time.
- Enhanced Readability: Eliminates verbose concatenation or formatting.

Text Blocks

Text Block eliminates the need of using \n & concatenation efforts in case of multi line string.

Old style:

String textBlock = "I " + "\n" + "like " + "\n" + "Java 21";

Java 21 style:

String textBlock = """

I

like

Java 21""";

String Templates

```
import static java.lang.StringTemplate.STR;

int age = 25;

String name = "Alice";

String message = STR."My name is \{name} and my age is \{age}";

System.out.println("message = " + message);

// compile with "javac --enable-preview --release 21 xxx.java“

// run with "java --enable-preview --release 21 xxx"
```

New methods in String/StringBuilder/StringBuffer

Newly added methods in String

//Removing incidental leading whitespace from multiline strings.

```
String text = """
              Line 1
                Line 2
              Line 3
              """;

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

//Output

```
Line 1
  Line 2
Line 3
```

Newly added methods in String

//Converts escape sequences (e.g., \n, \t) in a string into their actual characters.

String str = "Hello\n\tWorld";

*System.out.println(str.**translateEscapes()**);*

Output:

Hello

	World

Newly added methods in String

//String formatting

String str = "Hello %s %s";

*System.out.println(str.**formatted**("Anand", "Bipin"));*

//Output

Hello Anand Bipin

Newly added methods in String

```
//Adding Indent

String str = "Line 1\nLine 2";

System.out.println(str.indent(4));
```

//Output

```
    Line 1

    Line 2
```

Newly added methods in StringBuilder

//Appending subsequence of string to StringBuilder

StringBuilder sb = new StringBuilder("Hello");

sb.append("World", 0, 3);

System.out.println(sb);

//Output

HelloWor

Newly added methods in StringBuilder

//Inserting a subsequence of a CharSequence at a specified position.

StringBuilder sb = new StringBuilder("Hello");

sb.insert(**2,** "World"**, 1, 4**);

System.out.println(sb);

//Output

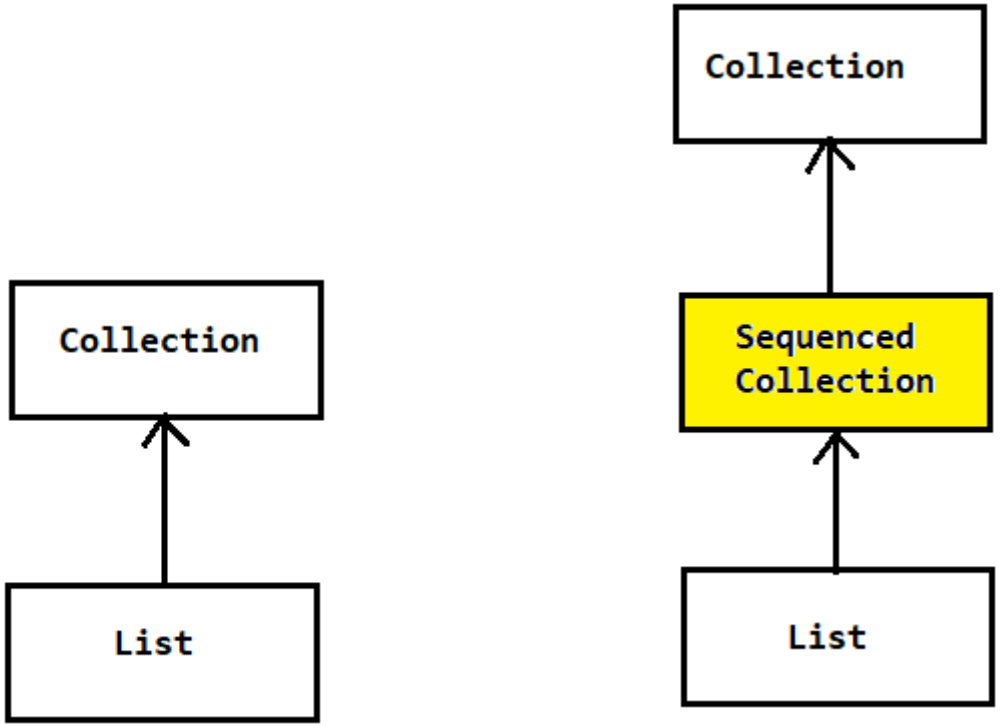
Heorlllo

Sequenced collections

Sequenced Collection

- In Java 21, a new feature called Sequenced Collections was introduced, enhancing the Java Collections Framework.
- This feature provides a more intuitive way to work with collections that maintain a specific order of elements.
- New default methods allow direct access to the first and last elements of a collection.
- You can obtain a reversed view of the collection using the reversed() method, making it easier to iterate in reverse order.
- The SequencedCollection interface is implemented by several existing collection types, such as List, Deque, and Set, ensuring consistency across these data structures.

Sequenced Collection



SequencedCollection

```
SequencedSet<String> sequencedSet = new LinkedHashSet<>(); sequencedSet.add("Apple");
sequencedSet.add("Banana"); sequencedSet.add("Cherry");

// Access first and last elements

System.out.println("First: " + sequencedSet.getFirst()); // Output: Apple

System.out.println("Last: " + sequencedSet.getLast());   // Output: Cherry

// Reversed view

SequencedSet<String> reversedSet = sequencedSet.reversed();

System.out.println("Reversed: " + reversedSet); // Output: [Cherry, Banana, Apple]
```

SequencedCollection

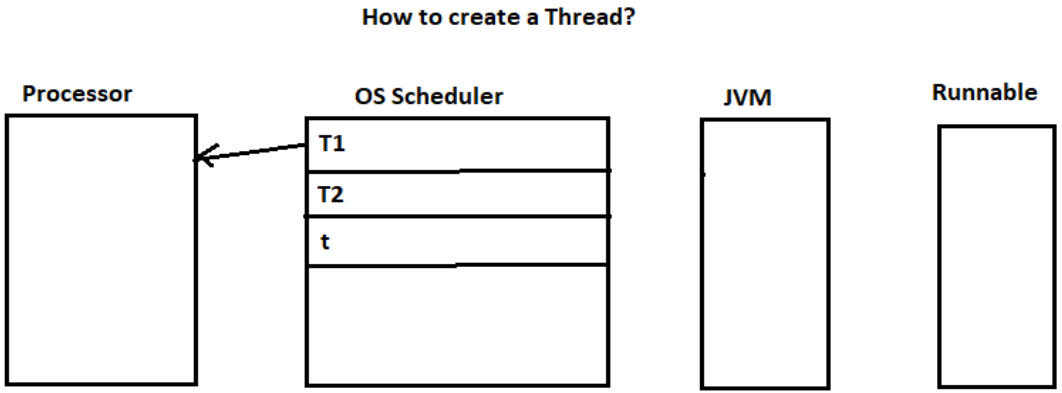
```
List<Integer> list = new ArrayList<Integer>(Arrays.asList(1,2,3,4,5));

list.addFirst(0);

list.addLast(6);

list.removeFirst();

list.removeLast();

System.out.println("First element: " + list.getFirst()); //1

System.out.println("Last element: " + list.getLast()); //5
```

Benefits of Sequenced Collection

- Simplifies operations on ordered collections.
- Reduces boilerplate code for accessing boundary elements or reversing collections.
- Enhances readability and maintainability of code.

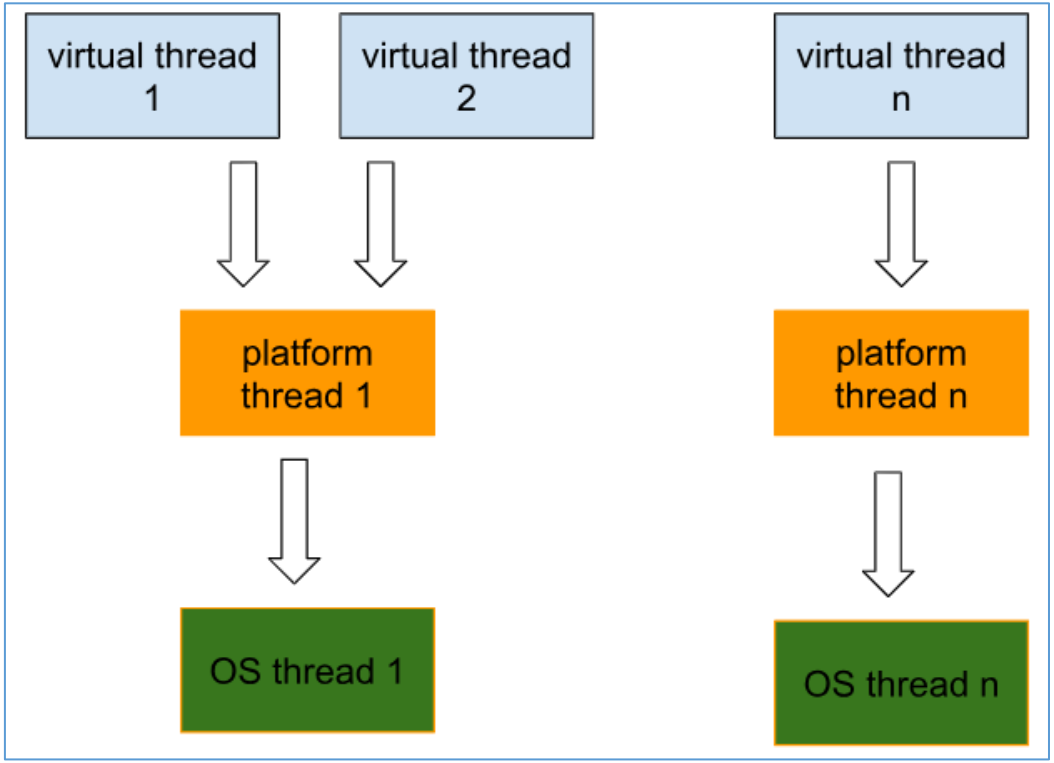
Virtual Threads

Traditional Threads

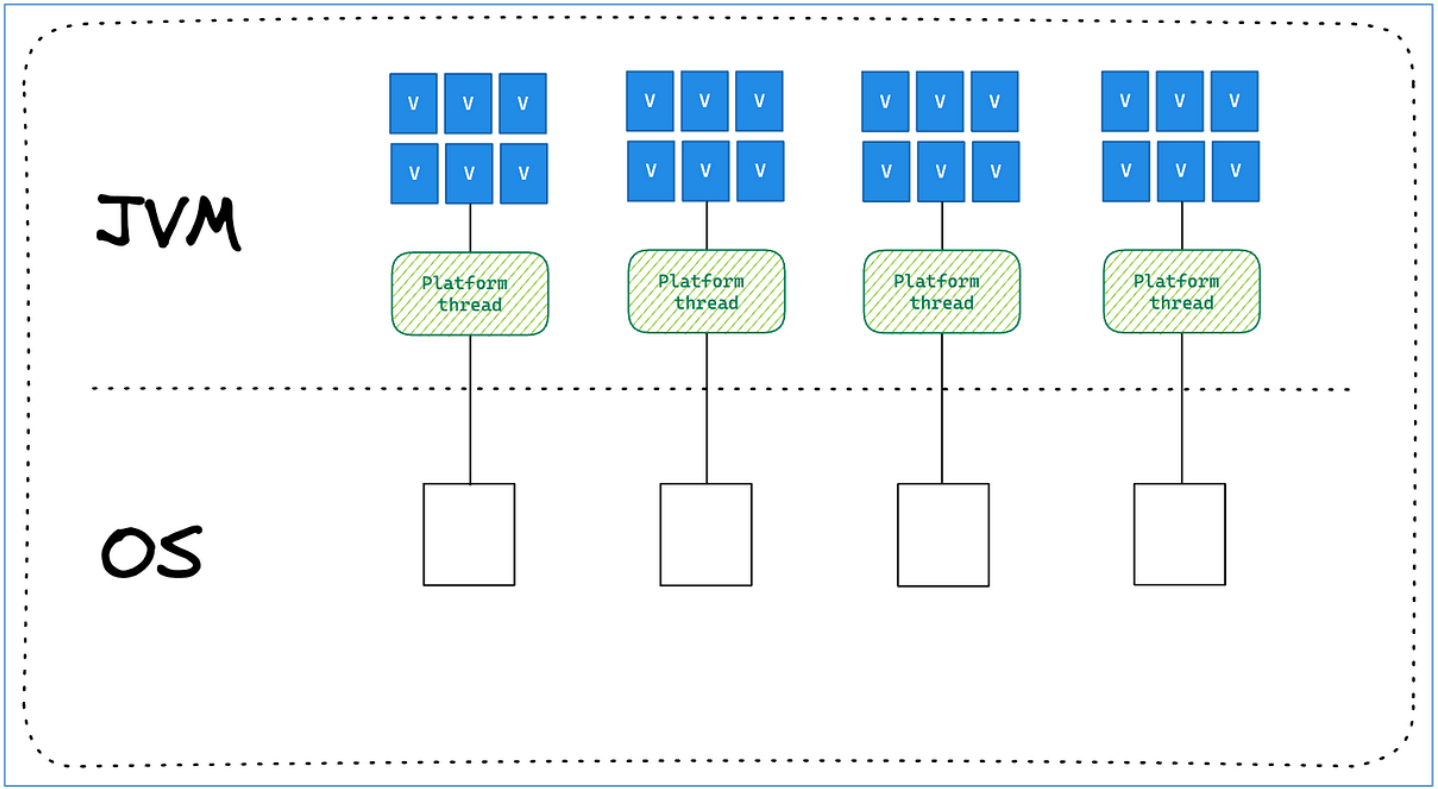



```
class Game implements Runnable {
        Thread t;
        public Game() {
                t = new Thread(this);
                t.start();
        }
        run() {
                //logic to move ball
upwards
        }
}
```

Virtual Threads



Virtual Threads



Virtual Threads

- Virtual threads introduced in Java 21 are lightweight threads managed by the JVM rather than the operating system.
- Unlike traditional threads, which are mapped 1:1 to OS threads, virtual threads are decoupled from OS threads, making them highly scalable and efficient.
- Virtual threads are instances of java.lang.Thread but are not tied to OS threads.
- When a virtual thread performs a blocking operation (e.g., I/O), the JVM suspends it without blocking the underlying OS thread. This allows the OS thread to be reused for other tasks, enabling the creation of millions of virtual threads with minimal resource overhead.

Benefits of Virtual Threads

- **Scalability:** Virtual threads allow applications to handle a massive number of concurrent tasks without exhausting system resources.
- **Simplified Concurrency:** Developers can write synchronous, readable code without complex asynchronous programming.
- **Reduced Memory Usage:** Virtual threads consume significantly less memory compared to traditional threads.
- **Compatibility:** They work seamlessly with existing Java APIs and libraries.

Creating a virtual thread

*Thread virtualThread = Thread.**startVirtualThread**(()->System.out.println("Virtual Task completed"));*

virtualThread.join();

//OR

*Thread virtualThread = Thread.**ofVirtual().start(**()->System.out.println("Virtual Task completed"));*

virtualThread.join();

Virtual thread pool

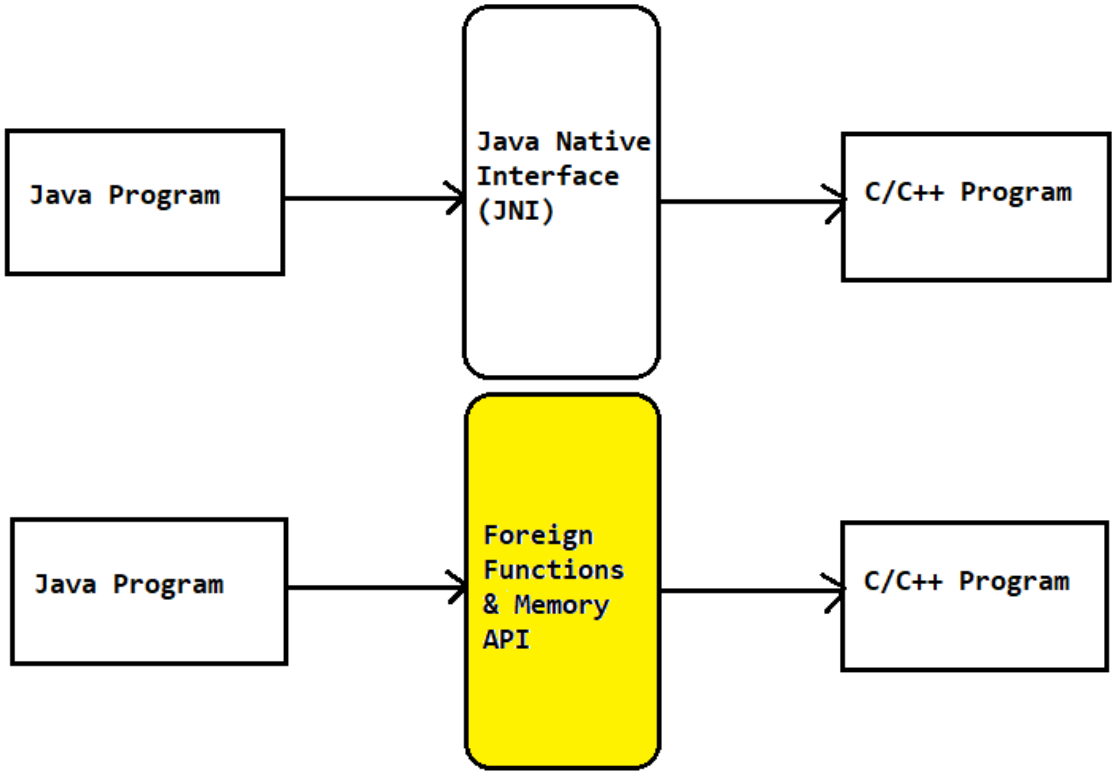
```
ExecutorService executorService = Executors.newVirtualThreadPerTaskExecutor();

for (int i = 0; i < 10; i++) {

        executorService.submit(() -> {

      System.out.println("Running task in a virtual thread: "

         + Thread.currentThread().threadId());

    });

}

executorService.close();
```

Traditional Threads Vs Virtual Threads

- **Resource Usage:** Virtual threads are lightweight and managed by the JVM, while traditional threads rely on OS resources.
- **Concurrency:** Virtual threads enable millions of concurrent tasks, whereas traditional threads are limited by OS constraints.
- Virtual threads are a game-changer for building high-throughput, concurrent applications in Java.

Foreign Function & Memory API

Foreign Functions & Memory API



Foreign Function & Memory API

- The Foreign Function and Memory API in Java 21, introduced as part of Project Panama, allows Java programs to interact with native code and memory in a safer and more efficient way compared to traditional approaches like JNI (Java Native Interface).
- Enables Java programs to call native functions written in languages like C or C++.
- Simplifies interaction with native libraries without the complexity of JNI.

Foreign Function & Memory API

- **Foreign Memory Access:** Provides mechanisms to access and manipulate memory outside the JVM (foreign memory). Ensures safety by avoiding common pitfalls like memory leaks or unsafe access.
- **Improved Safety:** The API is designed to minimize risks associated with native code, such as segmentation faults or memory corruption.
- **Performance:** Offers better performance for tasks requiring native code interaction, as it avoids the overhead of JNI.

Thank you!!