java17

sealed classes

Sealed Classes (JEP 409)

- **Concept:** Restricts which classes or interfaces can extend or implement them.
- **Goal:** Provides a controlled and predictable class hierarchy.
- **Key Keywords:**
  - `sealed`: Defines the class as restricted.
  - `permits`: Explicitly lists allowed subclasses.
- **Placement:** The `permits` clause must follow `extends` or `implements` declarations.

Rules for Sealed Classes

- **Location:** All permitted subclasses must be in the **same module** (or same package if in an unnamed module).
- **Explicit Extension:** Every permitted subclass must explicitly extend the sealed superclass.
- **Subclass Modifiers:** Every permitted subclass **must** use one of these three modifiers:
  1. **Final:** Cannot be extended further.
  2. **Sealed:** Can be extended, but only by its own permitted subclasses.
  3. **Non-sealed:** Opens the hierarchy back up for any class to extend

```
package com.itbulls.learnit.javacore.javaupdates;

import java.io.Serializable;

public class Java17 {

	public static void main(String[] args) {
		System.out.println("Demo of Sealed Classes");
	}

}


sealed interface Area permits Circle {}
//sealed class Shape extends Java17 implements Serializable permits Circle, Square, Triangle,
CustomShape {}
sealed class Shape permits Circle, Square, Triangle, CustomShape { }
```

final class Circle extends Shape implements Area {};
non-sealed class Square extends Shape {};
sealed class Triangle extends Shape permits IsoscelesTriangle {};
final class IsoscelesTriangle extends Triangle {}
final class CustomShape extends Shape {};

// can't extend Shape because for type CustomShape2 it is not permitted
//final class CustomShape2 extends Shape {};

code

This code demonstrates **Sealed Classes**, which is the headline feature of Java 17. It allows you to control the inheritance hierarchy by explicitly stating which classes are allowed to extend a base class.

Here is the breakdown of the code components:

---

1. The Sealed Base (The "Gatekeeper")

```java
sealed class Shape permits Circle, Square, Triangle, CustomShape { }
```

- **`sealed`**: This keyword tells the compiler that this class cannot be extended by just any class.
- **`permits`**: This is the "whitelist." Only the classes listed here (`Circle`, `Square`, `Triangle`, `CustomShape`) are allowed to extend `Shape`.

---

2. The Three Requirements for Subclasses

When a class extends a `sealed` class, it **must** declare how it handles its own inheritance using one of three modifiers:

A. `final` (The End of the Line)

```java
final class Circle extends Shape implements Area {};
final class CustomShape extends Shape {};
```

- These classes are permitted to extend `Shape`, but **no one** can extend them. The hierarchy stops here.

B. `non-sealed` (The Open Door)

```java
non-sealed class Square extends Shape {};
```

- This "breaks" the seal. Any class in the future can extend `Square`. It reverts to the behavior of a standard Java class.

C. `sealed` (The Continued Chain)

```java
sealed class Triangle extends Shape permits IsoscelesTriangle {};
```

- `Triangle` is permitted by `Shape`, but it chooses to remain `sealed` itself. It only allows `IsoscelesTriangle` to extend it.

---

3. Sealed Interfaces

```java
sealed interface Area permits Circle {}
```

- Just like classes, interfaces can be sealed. Here, only the `Circle` class is allowed to implement the `Area` interface.

---

4. Compilation Rules & Constraints

- **Same Package/Module:** All permitted subclasses must reside in the same package or module as the sealed class.
- **The Error Case:**

```java
// final class CustomShape2 extends Shape {};
```

- If you uncommented this, the code would **fail to compile**. Why? Because `CustomShape2` was not included in the `permits` list of the `Shape` class.

Summary Table for your PPT

Modifier	Meaning
`sealed`	Limits which classes can extend it.

permits	Defines the authorized list of subclasses.
final	Subclass cannot be extended further.
non-sealed	Subclass is open for extension by any class.

advantages and disadvntages

In Java, **Sealed Classes** (introduced in Java 17) give you control over which classes can extend or implement them. This breaks the traditional "all or nothing" approach where a class was either `final` (no subclasses) or `open` (any subclass).

Based on your hierarchy (**Shape** → **Circle** → **SemiCircle**), here are the specific rules and how they apply:

---

1. The Permit Rule

The top-level sealed class must explicitly list its allowed subclasses using the `permits` keyword.

- **Code:** `public sealed class Shape permits Circle { ... }`
- **The Rule:** Only the classes listed in the `permits` clause can directly extend `Shape`. If you try to create a `Square` class that extends `Shape` without adding it to the list, the code will not compile.

2. The Relationship Rule

The sealed class and its permitted subclasses must be in the **same package** (or the same module if you are using Java Modules). You cannot have a sealed `Shape` in one package and a permitted `Circle` in another.

3. The "State" Requirement (The 3 Options)

Every class that extends a sealed class **must** declare how it handles its own inheritance. You have exactly three choices for `Circle`:

Modifier	Meaning
`final`	The hierarchy stops here. No one can extend `Circle`.
`sealed`	`Circle` continues the restriction. It must permit specific subclasses (like `SemiCircle`).
`non-sealed`	The hierarchy "opens up." Anyone can now extend `Circle`.

Applying the Rules to Your Example

For your specific case where `Circle` is extended by `SemiCircle`, the code would look like this:

Java

```java
// 1. Base class restricts direct inheritance to Circle
public sealed class Shape permits Circle {
}

// 2. Circle must be 'sealed' because it wants to permit SemiCircle
public sealed class Circle extends Shape permits SemiCircle {
}

// 3. SemiCircle can be 'final' to stop the chain
public final class SemiCircle extends Circle {
}
```

Why this is useful (Pattern Matching)

The biggest advantage is **Exhaustiveness**. When you use a `switch` statement on a sealed type, the compiler knows all possible subclasses. You don't need a `default` case!

Java

```java
public String getAreaDescription(Shape shape) {
    return switch (shape) {
        case Circle c -> "It's a full circle";
        case SemiCircle sc -> "It's exactly half";
        // No 'default' needed! The compiler knows these are the only two.
    };
}
```

Summary of Rules for your Demo:

1. **Direct subclasses** must be permitted by the parent.
2. **Permitted subclasses** must be `final`, `sealed`, or `non-sealed`.
3. **Circle** must be `sealed` if it wants to allow `SemiCircle`.
4. **SemiCircle** must be one of the three modifiers (usually `final` if it's the last one).

Would you like me to show you what happens if you try to make `Circle` **non-sealed** instead?

Sealed classes are a powerful addition to the Java type system, primarily because they allow a developer to define a **closed hierarchy**. Instead of just restricting *how* a class is used (like `final`), they restrict *who* can use it.

Advantages of Sealed Classes

1. Exhaustiveness in Pattern Matching

This is the "killer feature" of sealed classes. When you use a `switch` statement or expression on a sealed type, the compiler knows every possible subclass.

- **No `default` case needed:** If you cover all permitted subclasses, the compiler is satisfied.
- **Compile-time safety:** If you add a new subclass to the `permits` list later, the compiler will instantly flag every `switch` statement in your codebase that needs to be updated.

2. Better Domain Modeling

In many business domains, you want to represent a fixed set of options. For example, a `PaymentMethod` might only ever be `CreditCard`, `PayPal`, or `BankTransfer`. Sealed classes allow you to model this more strictly than standard inheritance, but with more flexibility than an `Enum` (since each subclass can have its own unique fields and state).

3. Restricted Extension (Security)

Traditional inheritance is open by default. Anyone can extend your class and potentially change its behavior in ways you didn't intend. Sealed classes prevent "rogue" extensions from external libraries or other packages, ensuring the integrity of your API.

4. Improved Code Readability and Intent

By looking at a sealed class, a developer immediately sees the entire hierarchy. You don't have to search the whole project to find out what extends `Shape`; the `permits` clause tells you exactly what the options are.

Disadvantages of Sealed Classes

1. Rigidity and Reduced Extensibility

The "closed" nature is a double-edged sword. If you are building a library, sealed classes prevent your users from extending your types.

- **The Problem:** If a user needs a `Triangle` but you only permitted `Circle` and `Square`, they are stuck unless you update your library code.

2. Maintenance Overhead

Every time you add a new subclass, you must update the `permits` clause in the parent class. In a large project with many subclasses, this can feel like unnecessary "bookkeeping."

3. Same-Package Constraint

Sealed classes and their permitted subclasses must reside in the same package (or module). This can be a disadvantage if your project structure requires you to spread related classes across different packages for organizational reasons.

4. Increased Complexity for Beginners

The requirement to choose between `final`, `sealed`, and `non-sealed` for every subclass adds a layer of conceptual overhead. It requires developers to think several steps ahead about the future of the class hierarchy.

Summary Table

Advantage	Disadvantage
Exhaustive switches (no `default` needed)	**Inflexible** for library users
Strict Domain Modeling	**Package-private** limitations
Security (no unauthorized subclasses)	**Manual updates** to `permits` list
Enhanced Readability	**Learning curve** for the 3-modifier rule

Next Step

Would you like to see how **Sealed Classes** compare directly to **Enums** for modeling fixed data sets?

java21

vertual threads

Lesson Notes: Virtual Threads in Java 21 (JEP 444)

Java 21 introduces a revolutionary concurrency model through **Virtual Threads**, a core feature of **Project Loom**. This shift moves Java from a heavyweight, OS-dependent threading model to a lightweight, JVM-managed system designed for massive scalability.

1. The Evolution of Java Concurrency

- **Traditional Platform Threads:** Historically, `java.lang.Thread` served as a thin wrapper around **Operating System (OS) threads**.
    - **Limitations:** High memory overhead (~1 MB per stack) and expensive context switching. Scaling to millions of concurrent tasks was impossible.
- **The Async Alternative:** While frameworks like Project Reactor or RxJava solved scalability, they introduced "callback hell" and high code complexity.
- **The Virtual Thread Solution:** Virtual threads allow developers to write **simple, blocking code** that scales like asynchronous systems. They decouple Java concurrency from OS limitations.

2. How Virtual Threads Work

Virtual threads are managed by the JVM rather than the OS. They operate using a **Mounter/Carrier** relationship:

- **Carrier Threads:** These are traditional platform threads (OS threads) used as a pool to execute virtual threads.
- **Mounting/Unmounting:** A virtual thread is "mounted" onto a carrier thread to execute code. When it hits a blocking operation (e.g., I/O, `Thread.sleep()`), the JVM **unmounts** (parks) it, freeing the carrier thread for other work.
- **Continuations:** This is the underlying mechanism that saves the execution state (stack) of a virtual thread on the **heap** when it is suspended.

3. Implementation and APIs

There are two primary ways to create virtual threads:

- **Low-Level API:** Using `Thread.startVirtualThread(Runnable)` or `Thread.ofVirtual().start(Runnable)`.
- **Preferred API (ExecutorService):** Using `Executors.newVirtualThreadPerTaskExecutor()`.
    - **Try-with-resources:** It is idiomatic to use virtual thread executors within a `try-with-resources` block, as `ExecutorService` now implements `AutoCloseable`.

4. Comparison: Virtual vs. Platform Threads

Feature	Platform Threads	Virtual Threads
Creation Cost	High (OS call)	Very Low (JVM object)
Memory	~1 MB per stack	Dynamic (bytes/KBs on heap)
Context Switch	Expensive (Kernel level)	Cheap (JVM level)
Scalability	Limited (Thousands)	Massive (Millions)

5. Potential Pitfalls and Diagnostics

- **Thread Pinning:** This occurs when a virtual thread cannot be unmounted from its carrier thread. This typically happens during **Native Blocking Calls** (JNI) or inside `synchronized` blocks (though this is being improved).
- **Detection Tools: * JVM Flag:** `-XX:+EnableNativeAccess=ALL-UNNAMED`
    - **JCMD Command:** `jcmd <PID> Thread.dump_to_file` or specific thread pin diagnostic commands to identify stack traces causing pinning.

6. Scoped Values vs. ThreadLocal

In the virtual thread world, `ThreadLocal` is often a "bad fit" because virtual threads are short-lived and never reused.

- **The Issue:** `ThreadLocal` can lead to memory leaks and unpredictable behavior when virtual threads move between carriers.
- **The Alternative (JEP 429): Scoped Values** provide a way to share immutable data within a bounded execution scope. They are safer, more performant, and follow the execution flow rather than the thread identity.

Summary Paragraph

The introduction of Virtual Threads in Java 21 via JEP 444 marks the most significant shift in Java's concurrency model since its inception. By moving thread management from the

Operating System to the JVM, Project Loom enables developers to maintain a "thread-per-task" model that is both highly scalable and easy to debug. Unlike traditional platform threads that are resource-heavy, virtual threads are heap-allocated and lightweight, utilizing carrier threads and continuations to perform efficient "parking" during blocking I/O. While they simplify modern web and microservice development, developers must remain mindful of thread pinning during native calls and transition from legacy `ThreadLocal` patterns to the more modern Scoped Values (JEP 429) to ensure robust, leak-free applications.

Would you like me to generate a code snippet demonstrating the performance difference between a fixed thread pool and the new virtual thread executor?

source code

```
import java.util.List;
import java.util.concurrent.*;
import java.util.stream.IntStream;

public class VirtualThreadDemo {

    public static void main(String[] args) throws InterruptedException {
        System.out.println("=== DEMO: Virtual Threads in Java 21 ===");

        // --- 1. Basic Hello World using Thread.startVirtualThread() ---
        Thread vThread = Thread.startVirtualThread(() -> {
            System.out.println("Hello from a virtual thread (Thread.startVirtualThread)");
        });

        // Wait for the virtual thread to finish
        vThread.join();

        // --- 2. Preferred API: Executors.newVirtualThreadPerTaskExecutor() ---
        try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
            // Submit a simple task
            Future<String> result = executor.submit(() -> {
                System.out.println("Hello from a virtual thread in ExecutorService!");
                return "Task completed";
            });

            // Wait and print result
            System.out.println("Result: " + result.get());
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        // --- 3. Parallel task processing example ---
        int numberOfTasks = 100;

        System.out.println("\nRunning " + numberOfTasks + " tasks in parallel using virtual
threads...");
        long startTime = System.nanoTime();

        try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
            List<Future<?>> futures = IntStream.range(0, numberOfTasks)
                    .<Future<?>>mapToObj(i -> executor.submit(() -> {
                        System.out.println("Task #" + i + " is running on " + Thread.currentThread());
```

```java
                // Simulate some work
                try {
                    Thread.sleep(100);          // Simulate work
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }))
            .toList();

    for (Future<?> future : futures) {
        try {
            future.get(); // block until task completes
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

long endTime = System.nanoTime();
double virtualThreadsTime = (endTime - startTime) / 1_000_000.0;
                System.out.printf("Finished %d tasks in %.2f ms%n",
        numberOfTasks, virtualThreadsTime);

startTime = System.nanoTime();
int cpus = Runtime.getRuntime().availableProcessors();
try (ExecutorService executor = Executors.newFixedThreadPool(cpus)) {
    List<Future<?>> futures = IntStream.range(0, numberOfTasks)
            .<Future<?>>mapToObj(i -> executor.submit(() -> {
                System.out.println("Task #" + i + " is running on " + Thread.currentThread());
                // Simulate some work
                try {
                    Thread.sleep(100);          // Simulate work
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }))
            .toList();

    for (Future<?> future : futures) {
        try {
            future.get(); // block until task completes
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
        }
    }

    endTime = System.nanoTime();
    double platformThreadsTime = (endTime - startTime) / 1_000_000.0;
            System.out.printf("Finished %d tasks in %.2f ms%n",
        numberOfTasks, platformThreadsTime);
            System.out.println("=== Result ===");
            System.out.println("Virtual Threads Time: " + virtualThreadsTime);
            System.out.println("Platform Threads Time: " + platformThreadsTime);

    // --- 4. Memory usage (rough check) ---

  }
}
```

advantages and disadvantages

Virtual threads are not a "magic button" that makes every application faster. They are specifically designed for **high-concurrency, I/O-bound workloads**.

Here is a breakdown of where they shine and where they might actually hinder your performance.

Where to Use Virtual Threads

The best use case for virtual threads is any application that handles a high volume of concurrent requests that spend most of their time **waiting**.

- **Web Servers & Microservices:** Handling thousands of simultaneous HTTP requests (e.g., Spring Boot, Micronaut, Quarkus).
- **Database-Heavy Applications:** Services that perform many CRUD operations where the thread usually sits idle waiting for the database to respond.
- **API Gateways/Proxies:** Systems that call multiple downstream REST or gRPC services and aggregate the results.
- **Message Processing:** Consuming messages from Kafka or RabbitMQ where processing involves external network calls.

Advantages

1. **Massive Throughput:** You can scale to millions of threads, allowing you to use the simple "thread-per-request" model instead of complex asynchronous/reactive code.
2. **Low Memory Footprint:** Unlike platform threads that take ~1MB of memory, virtual threads start at just a few hundred bytes on the heap.
3. **Familiar Debugging:** You can use standard debugger tools, stack traces, and `try-catch` blocks. Unlike reactive programming, the stack trace actually shows you where the error happened in your logic.
4. **Cheap Context Switching:** Moving between virtual threads is handled by the JVM in user space, which is significantly faster than the OS switching between kernel threads.

Disadvantages

1. **CPU-Bound Tasks:** If your code is doing heavy calculations (e.g., video encoding, complex math, cryptography) and doesn't "wait," virtual threads offer **zero** benefit. In fact, they might be slightly slower due to management overhead.
2. **Thread Pinning:** As discussed earlier, if your code uses `synchronized` blocks or native methods (JNI), the virtual thread "pins" the carrier thread. This prevents other virtual threads from running and can lead to a performance bottleneck.

3. **No Rate Limiting:** Because threads are now "free," it's easy to accidentally overwhelm your database or external APIs. You can no longer rely on a small thread pool to limit your concurrency; you must use **Semaphores** or other rate-limiters.
4. **Compatibility with Legacy Code:** Old libraries that rely heavily on `ThreadLocal` or assume that a `Thread` is a precious resource might behave unexpectedly or leak memory.

Comparison Summary

Feature	Best for Virtual Threads	Best for Platform Threads
Workload Type	I/O-bound (Waiting for network/disk)	CPU-bound (Calculation/Processing)
Concurrency	Millions of tasks	Limited by CPU cores (e.g., 8–32)
Programming Style	Simple, synchronous, blocking	Parallel, compute-intensive
I/O Handling	Parks the thread (efficient)	Blocks the OS thread (inefficient)

structuaral concurrency

Understanding Structured Concurrency in Java

Structured Concurrency is an approach to multi-threaded programming that treats groups of related tasks running in different threads as a single unit of work. Introduced as a preview feature in recent JDKs (associated with Project Loom), it aims to eliminate the "fire and forget" risks of traditional concurrent programming.

In the past, when you release a thread, it became a "free agent." If the main task failed or timed out, the child thread might keep running in the background (a "leak"), consuming resources and making debugging a nightmare. Structured concurrency ensures that **child threads live and die within the same scope as their parent.**

How it Works with Virtual Threads

Virtual threads provide the **capability** to run millions of tasks, but Structured Concurrency provides the **discipline** to manage them. They work together through the `StructuredTaskScope` API.

1. The Scope Hierarchy

When using `StructuredTaskScope`, you create a syntactic block (usually a `try-with-resources` statement). Any virtual thread started within this scope is strictly bound to it.

- If a subtask fails, the scope can automatically trigger the cancellation of all other subtasks.
- The parent thread waits at the end of the block until all children have finished or the scope is shut down.

2. The "Fork and Join" Pattern

Instead of manually managing `Future.get()` calls, which can block indefinitely and lead to complex error handling, structured concurrency uses a "fork" mechanism:

- **Fork:** You "fork" several subtasks into the scope.
- **Join:** You call `scope.join()`. This is a single synchronization point.
- **Handle:** After joining, you can handle the results or errors collectively.

Key Policies: Shutdown on Success vs. Failure

Structured concurrency typically uses two main patterns to manage virtual threads:

Policy	Behavior	Best Use Case

ShutdownOnFailure	If one subtask fails, all other subtasks are cancelled.	When you need **all** parts of a task to succeed (e.g., booking a flight AND a hotel).
ShutdownOnSuccess	As soon as the first subtask succeeds, it cancels the rest.	When you only need **one** result (e.g., querying three different servers for the same data).

Why the Combination is Revolutionary

1. **Observability:** Because threads are structured in a tree-like hierarchy, the JVM can produce **Thread Dumps** that show relationships. You can see exactly which "Parent Task" spawned which "Virtual Thread," making it easy to find bottlenecks.
2. **Error Propagation:** If a subtask throws an exception, it is automatically propagated to the parent. You no longer have to worry about silent failures in background threads.
3. **Efficiency:** Virtual threads are so cheap that you can use a new scope for even the smallest sub-operations. The overhead is minimal, but the safety gain is massive.
4. **Automatic Cancellation:** If a user cancels a web request, structured concurrency ensures that all virtual threads working on that request (database queries, API calls) are terminated instantly, preventing resource waste.

> **Analogy:** If traditional concurrency is like a group of hikers wandering off in different directions without a map, Structured Concurrency is like a leader-led expedition where everyone is roped together. If one person falls or decides to turn back, the whole group stays coordinated.

To provide a solid demo of structured concurrency, you need to show how it handles the **lifecycle** of subtasks. The core API used is `StructuredTaskScope`.

Since these features are currently in **preview**, ensure you compile and run with the `--enable-preview` flag.

Demo 1: Shutdown On Failure (The "All or Nothing" Pattern)

This is the most common use case. If you are fetching data from two different services (e.g., a User Service and an Order Service), and one fails, there is no point in waiting for the other.

```java
import java.util.concurrent.StructuredTaskScope;
import java.util.function.Supplier;

public class ConcurrencyDemo {

    public void handleOrder(String orderId) {
        // Create a scope that shuts down if any subtask fails
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

            // Forking virtual threads for subtasks
            Supplier<String> user = scope.fork(() -> fetchUser());
            Supplier<String> order = scope.fork(() -> fetchOrderDetails(orderId));

            // Wait for both to finish or one to fail
            scope.join();
            scope.throwIfFailed(); // Throws exception if any subtask failed

            // If we reach here, both succeeded
            System.out.println("Results: " + user.get() + " and " + order.get());

        } catch (Exception e) {
            System.err.println("Task failed: " + e.getMessage());
        }
    }
```

```
    private String fetchUser() throws InterruptedException {
        Thread.sleep(100); // Simulate network
        return "John Doe";
    }

    private String fetchOrderDetails(String id) throws InterruptedException {
        Thread.sleep(150);
        return "Laptop Pro";
    }
}
```

Demo 2: Shutdown On Success (The "Racing" Pattern)

Use this when you want the fastest result from multiple sources (e.g., querying three different DNS servers or weather APIs). As soon as **one** succeeds, the others are automatically cancelled.

Java

```java
import java.util.concurrent.StructuredTaskScope;

public class RacingDemo {

    public String getFastestWeather() {
        try (var scope = new StructuredTaskScope.ShutdownOnSuccess<String>()) {

            // Forking multiple sources
```

```java
        scope.fork(() -> callServiceA());
        scope.fork(() -> callServiceB());
        scope.fork(() -> callServiceC());

        // Wait for the first success
        scope.join();

        // Returns the value of the first task that finished successfully
        return scope.result();

    } catch (Exception e) {
        return "Default Weather: Sunny";
    }
}

private String callServiceA() throws InterruptedException {
    Thread.sleep(500);
    return "Service A: 25°C";
}

private String callServiceB() throws InterruptedException {
    Thread.sleep(100); // This one will win!
    return "Service B: 24°C";
}
```

```java
    private String callServiceC() throws InterruptedException {

        Thread.sleep(300);

        return "Service C: 26°C";

    }

}
```

Key Takeaways for your Presentation:

- **The `try-with-resources` block:** Emphasize that the scope ends at the closing brace. If a subtask is still running, `join()` ensures the parent waits, or the scope cancels them if the policy is met.
- **Implicit Cancellation:** In the "Racing" demo, point out that as soon as `Service B` finishes, the JVM automatically sends an interruption signal to `A` and `C`. You don't have to write manual cleanup code.
- **Observability:** If you run a thread dump while `scope.join()` is active, the dump will show a clear **tree structure** (e.g., *Parent Thread -> TaskScope -> Subtask 1, Subtask 2*).

Why this is better than `CompletableFuture`?

In `CompletableFuture`, if you forget to handle an exception or a timeout, a thread might stay alive forever (a leak). With `StructuredTaskScope`, the structure of the code **guarantees** that no child thread escapes the block.

Would you like me to explain how to handle **timeouts** specifically within these structured scopes?

thread local

What is ThreadLocal?

ThreadLocal is a class in Java that allows you to create variables that can only be read and written by the **same thread**. Essentially, it provides "thread-local" variables—a way to store data that is globally accessible within the thread's execution but isolated from all other threads.

How it Works

1. **Storage:** Each Thread object internally holds a map (the ThreadLocalMap).
2. **Key-Value Pair:** When you call threadLocal.set(value), the ThreadLocal object acts as a key, and your data is the value stored inside the *current* thread's map.
3. **Isolation:** When Thread A calls get(), it looks into its own map. Even if Thread B uses the same ThreadLocal variable name, it looks into its own separate map, so they never see each other's data.

Common Use Case: Storing a database connection, a user session ID, or a transaction context so you don't have to pass it as a parameter through every single method in your application.

Why ThreadLocal is Problematic for Virtual Threads

While ThreadLocal worked well for the old "Platform Thread" model, it creates significant issues when you switch to Java 21's **Virtual Threads**.

1. Unbounded Memory Usage (The "Millions" Problem)

- **Platform Threads:** You usually have a small pool (e.g., 200 threads). Having 200 ThreadLocalMaps is manageable.
- **Virtual Threads:** You might have **1,000,000** virtual threads. If each one creates a ThreadLocal map with even a small amount of data, you could quickly consume gigabytes of heap memory, leading to an OutOfMemoryError.

2. Excessive Inheritance Overhead

Many frameworks use InheritableThreadLocal, which copies the map from a parent thread to a child thread. In a virtual thread environment where tasks constantly spawn sub-tasks (Structured Concurrency), the overhead of constantly deep-copying these maps becomes a major performance bottleneck.

3. Mutation and Lifetime Issues

Virtual threads are designed to be **disposable**. They are created for a single task and then thrown away. ThreadLocal was designed for long-lived threads that are reused (like in a thread pool). Using ThreadLocal with virtual threads often leads to "leaky" code where data persists longer than intended or is difficult to clean up because the thread identity itself is so fleeting.

The Modern Alternative: Scoped Values (JEP 429)

To solve these issues, Java introduced **Scoped Values**.

- **Immutable:** Unlike `ThreadLocal`, Scoped Values are usually immutable. You set them for a specific "scope," and they cannot be changed during that execution.
- **Memory Efficient:** They are designed to be shared efficiently across millions of virtual threads without copying the entire map.
- **Automatic Cleanup:** Once the code block (the scope) finishes, the value is automatically discarded. There is no risk of the data "hanging around" in a thread pool.

> **Summary:** `ThreadLocal` is like a personal locker in a gym. It works if there are 50 members. But if a million people show up for a one-time event, building a million lockers is impossible—you're better off giving everyone a temporary wristband (a Scoped Value) that they throw away when they leave.

presentation document

Agenda for sealed classes , interface and virtual threads
Sealed classes:

- What is sealed classes and interface
- What are the keywords we will use
- Rules for Sealed Classes
- Code demo

Sealed classes and interfaces :

—---

- **Concept:** Restricts which classes or interfaces can extend or implement them.
- **Goal:** Provides a controlled and predictable class hierarchy.
- **Key Keywords:**
  - `sealed`: Defines the class as restricted.
  - `permits`: Explicitly lists allowed subclasses.
- **Placement:** The `permits` clause must follow `extends` or `implements` declarations.

Rules for Sealed Classes

- **Location:** All permitted subclasses must be in the **same module** (or same package if in an unnamed module).
- **Explicit Extension:** Every permitted subclass must explicitly extend the sealed superclass.
- **Subclass Modifiers:** Every permitted subclass **must** use one of these three modifiers:
  1. **Final:** Cannot be extended further.
  2. **Sealed:** Can be extended, but only by its own permitted subclasses.
  3. **Non-sealed:** Opens the hierarchy back up for any class to extend

Advantages :

- **Exhaustive switches**
- **Strict Domain Modeling**
- **Security**
- **Enhanced Readability**

Disadvantage :

- **Inflexible** for library users
- **Package-private** limitations
- **Manual updates** to permits list

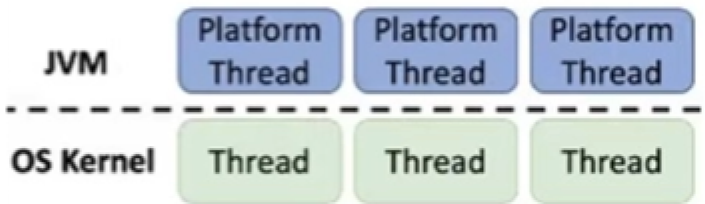
Code demo :

```
public class Java17 {

	public static void main(String[] args) {
		System.out.println("Demo of Sealed Classes");
	}

}


sealed interface Area permits Circle {}
//sealed class Shape extends Java17 implements Serializable permits Circle, Square, Triangle, CustomShape {}
sealed class Shape permits Circle, Square, Triangle, CustomShape { }

final class Circle extends Shape implements Area {};
non-sealed class Square extends Shape {};
sealed class Triangle extends Shape permits IsoscelesTriangle {};
final class IsoscelesTriangle extends Triangle {}
final class CustomShape extends Shape {};

// can't extend Shape because for type CustomShape2 it is not permitted
//final class CustomShape2 extends Shape {};
```

Virtual Threads :

Agenda for virtual classes :
What is traditional threads
Why we are using virtual threads
What is virtual threads
How it works internally
How to create virtual threads
Demo for virtual threads
Structural concurrency with virtual threads
Where can we use virtual threads

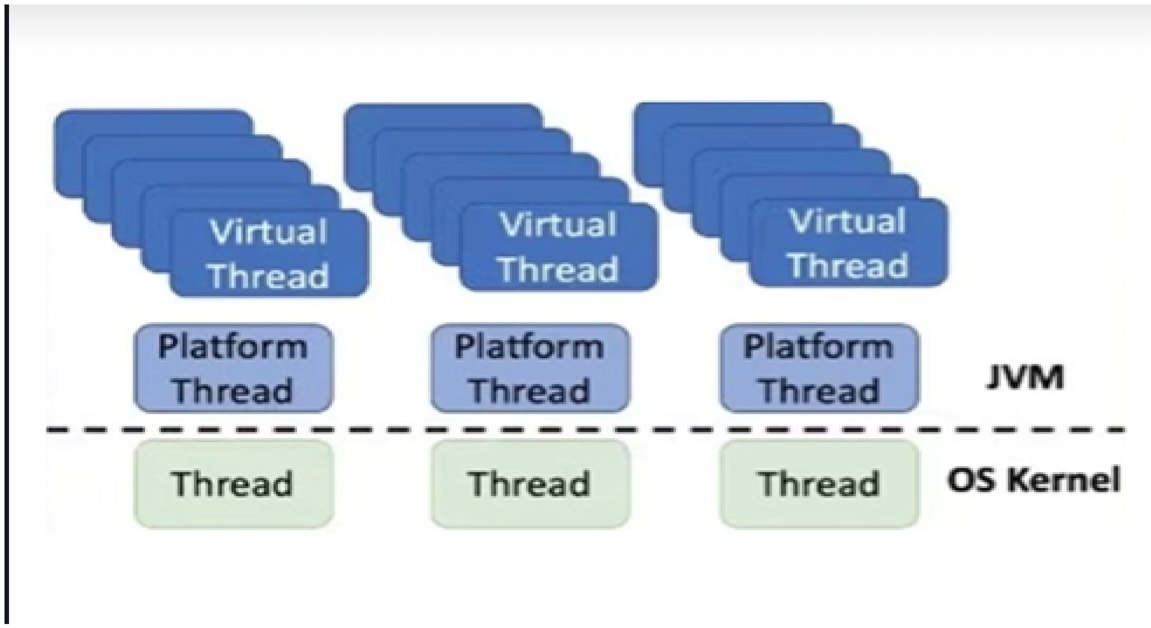
Traditional Threads:
—-------------------------------



- **Platform threads -wrappers over OS threads (java.long.Thread)**
- **Reliable , but heavyweight model**
- **Each thread requires ~1 MB stack memory**
- **Context switching managed by OS -costly**
- **Limits scalability to large numbers of concurrent tasks**
- **Battleneck for high throughput apps(web services)**

Virtual Threads :

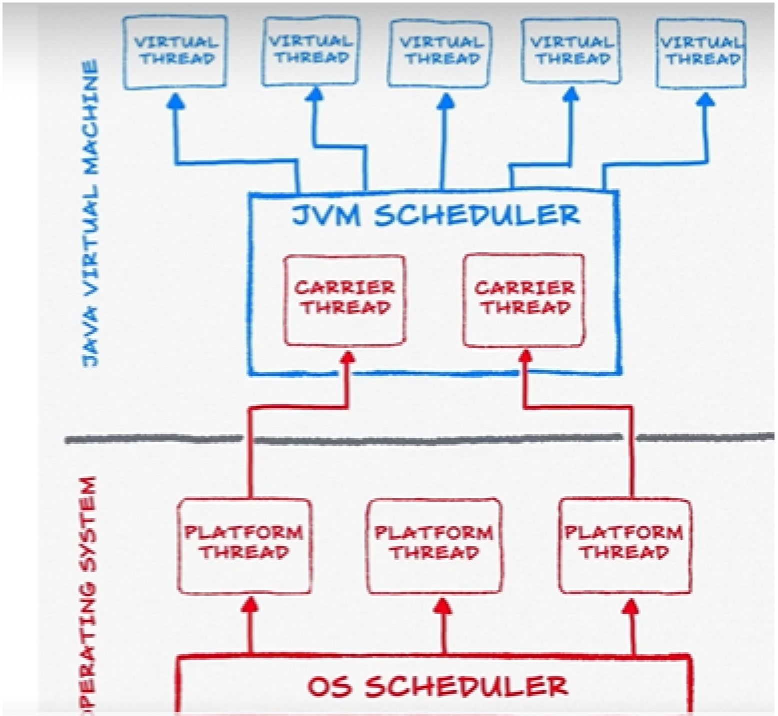
—--------------------



How Virtual threads Internally works :

—--

- **Carrier Threads**
- **Mounting/Unmounting**
- **Continuations**



- **Virtual threads : light weight java.long.Thread**
- **Not bound to 1: 1 to OS threads**
- **Managed by JVM with a pool of carrier threads(OS Threads)**
- **Mounted on a carrier only when running**
- **Parked during Blacking ops(I/O, sleep()), freeing carrier**
- **Unparked later - can resume on any carrier**
- **Preserves java state,maximizes scalability**

How to create virtual threads :

There are two primary ways to create virtual threads:

- **Low-Level API:** Using `Thread.startVirtualThread(Runnable)` or `Thread.ofVirtual().start(Runnable)`.
- **Preferred API (ExecutorService):** Using `Executors.newVirtualThreadPerTaskExecutor()`.
    - **Try-with-resources:** It is idiomatic to use virtual thread executors within a `try-with-resources` block, as `ExecutorService` now implements `AutoCloseable`.

Where to Use Virtual Threads :

- **Micro services and web servers**
- **Database-Heavy Applications**
- **API Gateways/Proxies**
- **Message Processing**

Advantages :

- **Massive Throughput**
- **Low Memory Footprint**
- **Familiar Debugging**
- **Cheap Context Switching**

Disadvantages :

- **CPU-Bound Tasks:**
- **Thread Pinning**
- **No Rate Limiting**
- **Compatibility with Legacy Code**

Understanding Structured Concurrency in Java

Structured Concurrency is an approach to multi-threaded programming that treats groups of related tasks running in different threads as a single unit of work. Introduced as a preview feature in recent JDKs (associated with Project Loom), **it aims to eliminate the "fire and forget" risks of traditional concurrent programming**.

How it Works with Virtual Threads

- **The Scope Hierarchy**
- **The "Fork and Join" Pattern**

Key Policies: Shutdown on Success vs. Failure

- **ShutdownOnSuccess**
- **ShutdownOnFailure**

Why the Combination is Revolutionary

1. **Observability**
2. **Error Propagation**
3. **Efficiency**
4. **Automatic Cancellation**

Code demo