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");
	}

}
```

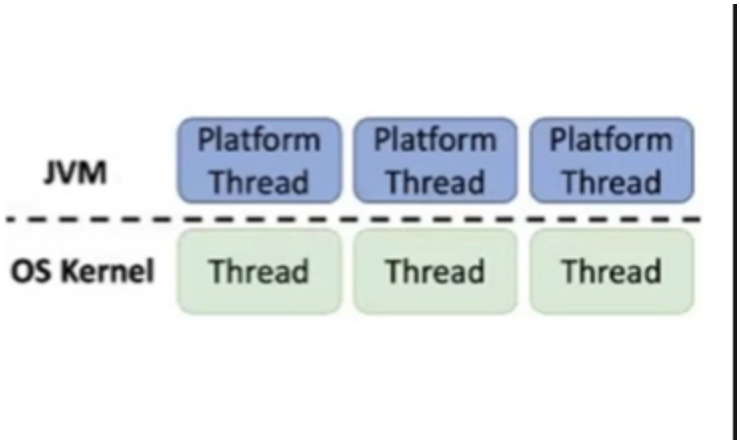
```
sealed interface Area permits Circle {}
//sealed class Shape extends Java17 implements Serializable permits Circle, Square, Triangle, CustomShape {}
sealed class Shape permits Circle, Square, Triangle, CustomShape { }

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:

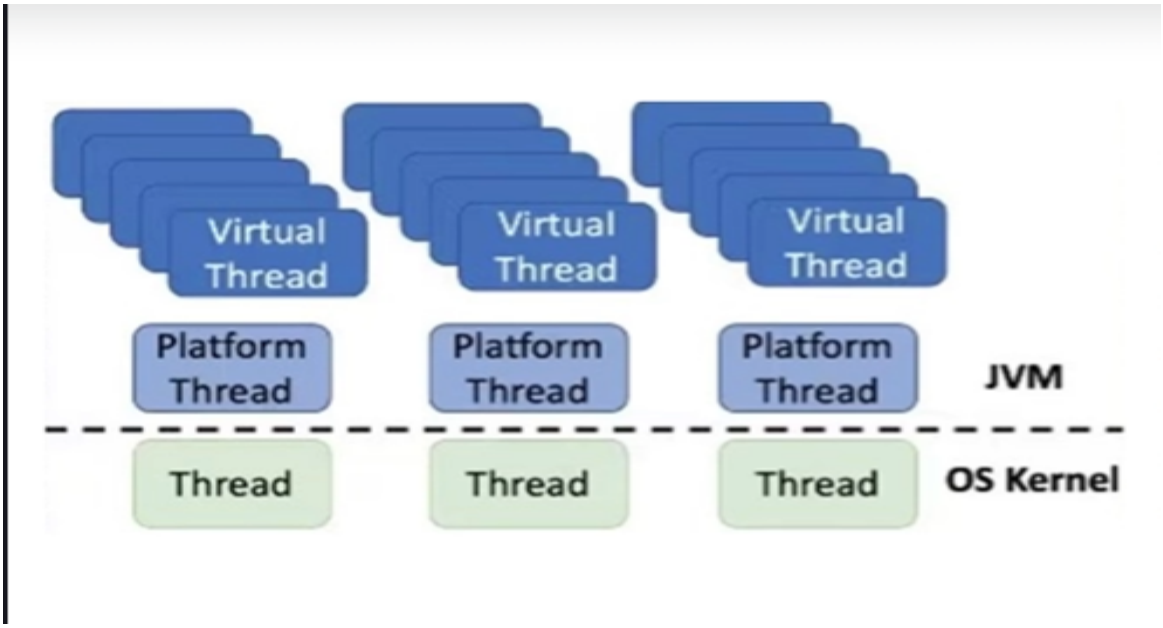
—----------



- **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 :

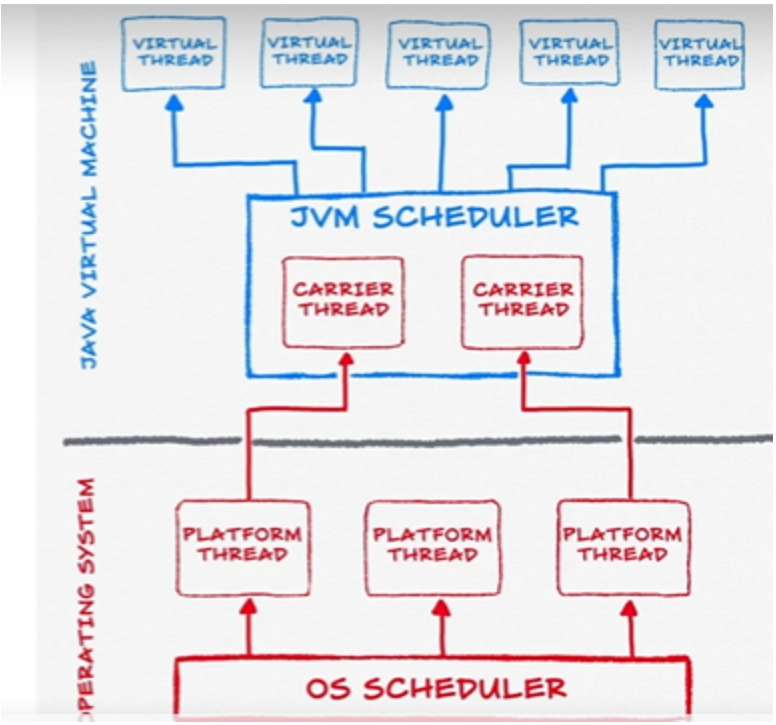
—----------



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