

 Java 21+

# Concurrencia Moderna en Java 21

## Virtual Threads y Más



Una guía para desarrolladores junior



 2025



## ¿Qué es un Thread?

- **Unidad básica de ejecución** en Java
- Permite ejecutar código de forma **concurrente**
- Comparten memoria y recursos dentro del mismo proceso

💡 **Analogía:** Como trabajadores independientes que ejecutan tareas en paralelo

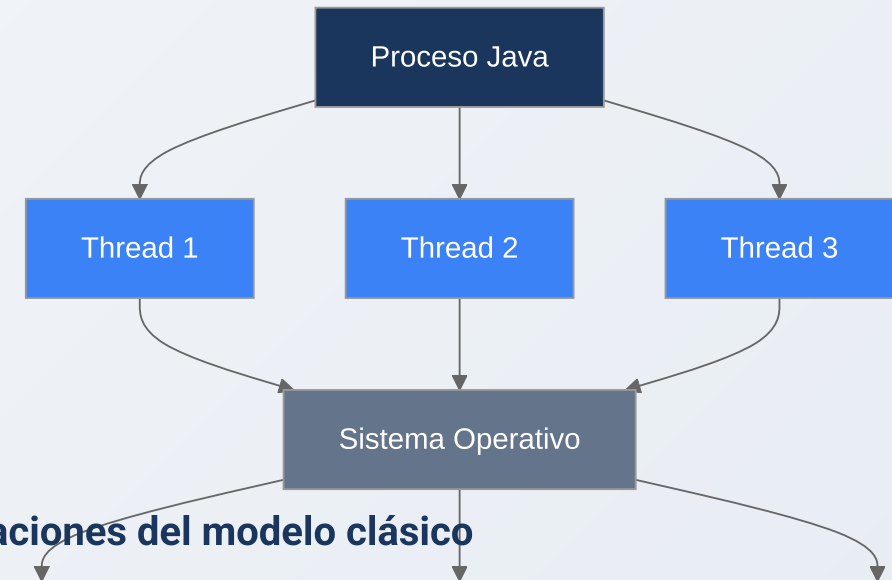
## Creación tradicional

Dos formas principales:

```
// 1. Extendiendo Thread
class MiThread extends Thread {
    public void run() {
        System.out.println("Ejecutando ...");
    }
}
new MiThread().start();
```

```
// 2. Implementando Runnable
Runnable tarea = () → {
    System.out.println("Ejecutando ...");
};
new Thread(tarea).start();
```

```
// Con ExecutorService
ExecutorService executor = Executors.newFixedThreadPool(10);
executor.submit(() → {
    System.out.println("Tarea en el pool");
});
```



## Limitaciones del modelo clásico



### Peso pesado

Cada thread de Java se mapea 1:1 con un thread del sistema operativo, consumiendo recursos significativos



### Alto costo de creación

Crear y destruir threads es costoso en términos de rendimiento



### Bloqueo de recursos

Cuando un thread espera (ej: I/O, network), bloquea un thread del OS completo



### Problemas de escalabilidad

Difícil escalar a miles de tareas concurrentes (limitado por OS)

# Introducción a Virtual Threads (Proyecto Loom)

## ¿Qué son los Virtual Threads?

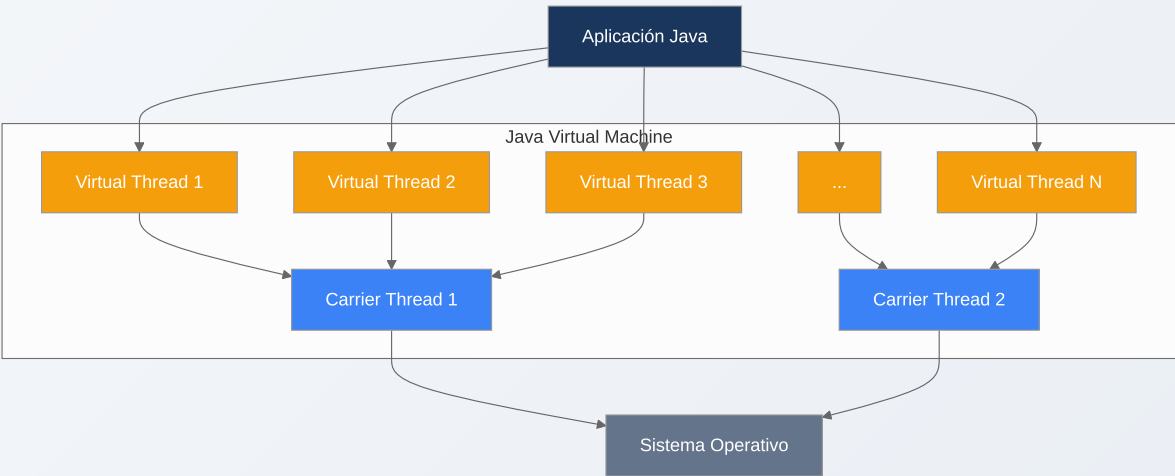
Hilos ligeros implementados por la JVM, no directamente por el sistema operativo

💡 **Analogía:** Si los Platform Threads son camiones pesados, los Virtual Threads son bicicletas ágiles y ligeras

## Problema que resuelven

La paradoja de la concurrencia moderna:

- Aplicaciones modernas manejan **miles de conexiones simultáneas**
- Cada conexión necesita **esperar respuestas** (I/O, red, BD)
- Los Platform Threads **bloquean recursos** mientras esperan
- Crear miles de Platform Threads es **inviable** por su costo



## Diferencias con Platform Threads

Platform Threads	Virtual Threads
Mapeado 1:1 con hilo del OS	Administrados por la JVM
Peso pesado (~2MB por thread)	Extremadamente ligero (~1KB)
Número limitado (~miles)	Escala a millones
Costo alto de creación	Creación muy económica
Bloqueo = Recurso inutilizado	Bloqueo = Libera carrier thread

## Beneficios Principales



### Mayor rendimiento

Ideal para aplicaciones I/O-bound (servicios web, bases de datos)



### Alta escalabilidad

Maneja millones de tareas concurrentes con recursos limitados



### Código más simple

Estilo síncrono sin complejos callbacks o reactive streams



### Compatibilidad

API Thread existente sin cambios en código heredado

# Cómo Usar Virtual Threads (Ejemplos Prácticos)

▶ Creación Directa

☰ Con ExecutorService

🔗 Casos Prácticos

## Creación de un Virtual Thread

```
// Método 1: Factory method estático
Thread vThread = Thread.startVirtualThread(() -> {
    System.out.println(";Ejecutando en Virtual Thread!");
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {}
});

// Método 2: Builder pattern
Thread vt = Thread.ofVirtual()
    .name("mi-virtual-thread")
    .start(() -> {
        System.out.println("Tarea en " +
Thread.currentThread().getName());
    });

// Método 3: Builder sin iniciar
Thread vt2 = Thread.ofVirtual()
    .name("otro-virtual")
    .unstarted(() -> { ... });
vt2.start(); // Iniciamos cuando queramos
```

🌿 Ligero

⚡ Rápido de crear

✅ API familiar

## Comparativa con Platform Threads

Platform Thread (tradicional)

```
// Forma tradicional
Thread thread = new Thread(() -> {
    System.out.println("Hilo tradicional");
});
thread.start();
```

VS

Virtual Thread (Java 21+)

```
// Forma moderna
Thread vThread = Thread.startVirtualThread(() -> {
    System.out.println("Virtual Thread");
});
```

💡 Los Virtual Threads son también instancias de `Thread`, por lo que la API es familiar

# Platform Threads vs. Virtual Threads: ¿Cuándo usar cuál?

Una guía simple para elegir el tipo de thread adecuado para cada situación

 **Platform Threads**

**Usar cuando:**

-  Tareas **intensivas en CPU** (cálculos, procesamiento)
-  Operaciones de **tiempo real** con latencia crítica
-  Tareas que necesitan **alta prioridad** del sistema
-  Interacción **directa con hilos del OS** o recursos nativos



**Ejemplos de aplicación:**

- Procesamiento de imágenes/video
- Simulaciones científicas
- Operaciones matemáticas complejas
- Procesamiento en tiempo real (juegos, trading)

 **Importante:** Limitar su número (normalmente, cercano al número de cores)

 **Virtual Threads**

**Usar cuando:**

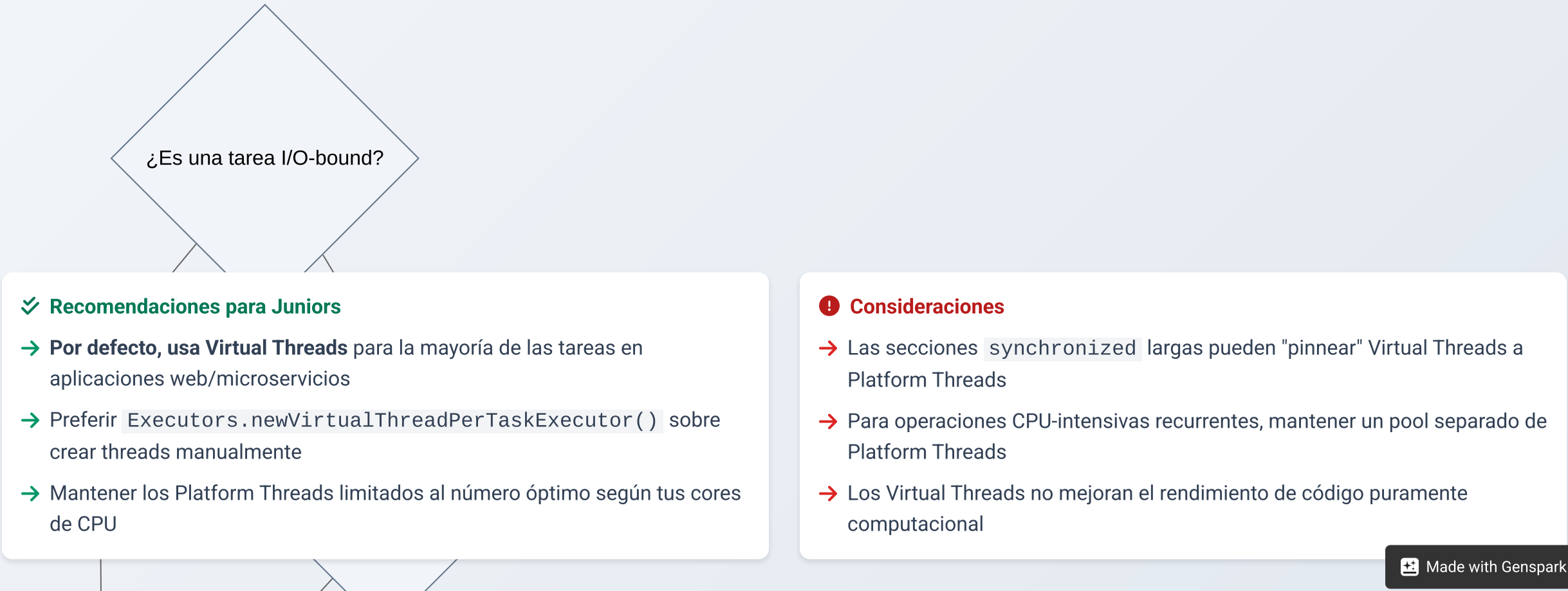
-  Tareas con mucho tiempo de **espera I/O** (red, archivos)
-  Operaciones **concurrentes** con bases de datos
-  Aplicaciones con **muchas conexiones** simultáneas
-  Cuando necesitas **gran cantidad** de tareas concurrentes

**Ejemplos de aplicación:**

- Servidores HTTP/API REST
- Microservicios
- Aplicaciones de base de datos
- Procesamiento asíncrono de eventos

 **Ventaja:** Puedes crear miles o millones sin preocuparte

## Guía de Decisión Rápida



# Structured Concurrency (Concurrencia Estructurada)

## ¿Qué es?

Un modelo que organiza tareas concurrentes en una estructura jerárquica de **padres e hijos**, donde:

**Una tarea principal no termina hasta que finalizan todas sus subtareas**  
Garantiza que todas las tareas iniciadas en un contexto se completen antes de que el contexto termine

**Analogía:** Como un equipo donde un gerente (tarea principal) espera a que todos sus empleados (subtareas) terminen su trabajo antes de dar el proyecto por finalizado

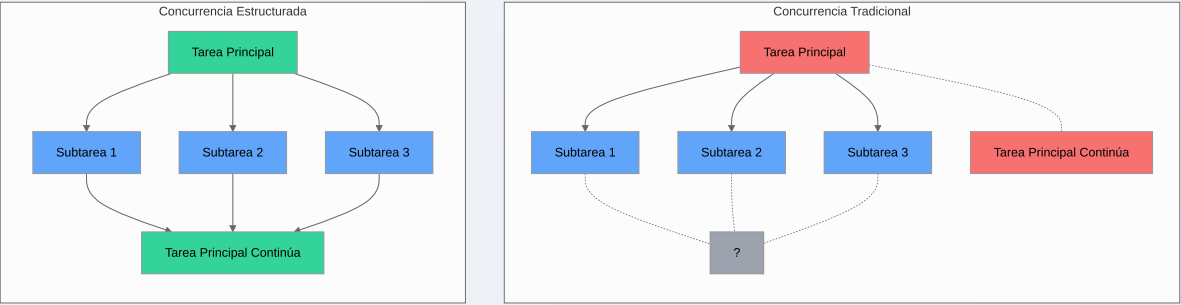
## Problema que resuelve

- ❌ **Tareas huérfanas:** En la concurrencia tradicional, es fácil perder el control de tareas iniciadas
- ❌ **Manejo de errores complejo:** Difícil propagar excepciones entre tareas independientes
- ❌ **Cancelación complicada:** Difícil cancelar grupos de tareas relacionadas

### Beneficios Principales

- ✓ Código más **legible** y fácil de razonar
- ✓ Mejor **manejo de errores y cancelación** de tareas
- ✓ Evita **fugas de recursos** y tareas abandonadas
- ✓ Código más **robusto** y con menos bugs

## Visualizando la Structured Concurrency



## Ejemplo Simplificado

```
// API de Concurrencia Estructurada (preview)
try (StructuredTaskScope scope = new StructuredTaskScope<>()) {

    // Lanzar tareas hijas
    Future datos = scope.fork(() -> consultarDatos());
    Future config = scope.fork(() -> cargarConfig());

    // Esperar que TODAS las subtareas terminen
    scope.join(); // Bloquea hasta que todas las subtareas finalicen

    // Procesar resultados después
    String resultado = datos.resultNow();
    Integer configuracion = config.resultNow();

} // Al salir del bloque, se garantiza que todas las subtareas han terminado
```

### Relación con Virtual Threads

- Virtual Threads y Structured Concurrency son complementarios:**
- Virtual Threads hacen **viable** tener miles de tareas concurrentes
  - Structured Concurrency las hace más **manejables y robustas**
  - Juntos, permiten código concurrente que es a la vez **escalable y fácil de mantener**

## ¿Qué es Structured Concurrency?

Un modelo que organiza tareas concurrentes como una **jerarquía**, donde las subtareas pertenecen a una tarea principal que controla su ciclo de vida.

### Problema que resuelve:

- ❌ Tareas que **se pierden** o quedan huérfanas
- ❌ Manejo de **errores inconsistente** entre subtareas
- ❌ Fugas de recursos por **tareas no finalizadas**
- ❌ Dificultad para **rastrear y cancelar** grupos de tareas relacionadas

### 💡 Analogía:

Como un **equipo de trabajo organizado**:

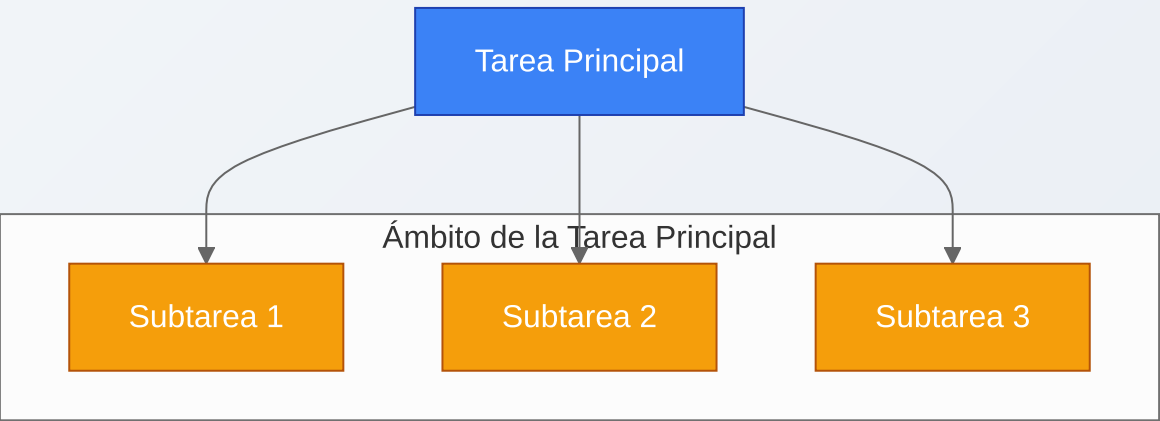
- Un **gerente** (tarea principal) asigna tareas a **empleados** (subtareas)
- El gerente **espera** a que todos terminen
- Si un empleado tiene **problemas**, el gerente es informado
- El gerente puede **cancelar** todo el proyecto si es necesario



Preview

Característica incubadora en versiones recientes de Java

## Visualización del Concepto



### Concurrencia Tradicional

- Tareas se crean y operan **independientemente**
- Control manual de **inicios y finalizaciones**
- **Difícil** rastrear errores entre tareas relacionadas

### Concurrencia Estructurada

- Tareas operan dentro de un **ámbito definido**
- **Gestión automática** del ciclo de vida
- Propagación **controlada de errores**

## Ejemplo Básico (Versión Futura)

```
try (StructuredTaskScope<String> scope = new
StructuredTaskScope.ShutdownOnFailure()) {

    // Lanzar múltiples subtareas
    Future<String> user = scope.fork(() → fetchUserData(userId));
    Future<String> account = scope.fork(() →
fetchAccountData(accountId));

    // Esperar a que todas finalicen (o fallen)
    scope.join();

    // Si llegamos aquí, todo fue exitoso
    return processData(user.resultNow(), account.resultNow());

} // Cierre automático del ámbito
```

### 🔗 Relación con Virtual Threads

- **Combinación perfecta:** Virtual Threads ligeros + organización estructurada
- **Escalabilidad confiable:** Manejo de miles de tareas concurrentes sin perder control
- **Código comprensible:** Mantiene la simplicidad incluso con mucha concurrencia



Made with Genspark

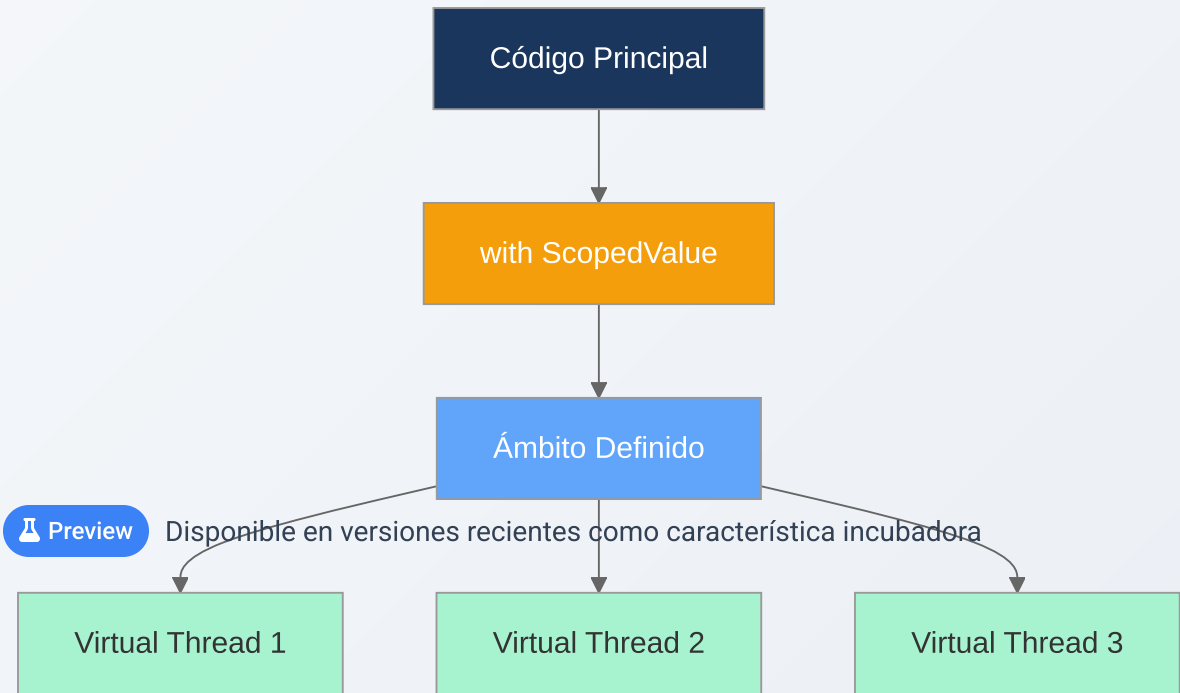


¿Qué son los Scoped Values?

Una forma de compartir datos **inmutables** y **de sólo lectura** dentro de un contexto específico de ejecución, como un Virtual Thread.

💡 ¿Qué problema resuelven?

- ✔ **Alternativa segura** a `ThreadLocal`, especialmente con Virtual Threads
- ✔ Evita **fugas de memoria** cuando se usan millones de Virtual Threads
- ✔ Proporciona **alcance explícito** en lugar de asociación permanente a un thread



ThreadLocal vs. Scoped Values

Característica	ThreadLocal	ScopedValue
Mutabilidad	Mutable (set/get)	Inmutable (sólo lectura)
Alcance	Toda la vida del thread	Sólo el bloque definido
Uso de memoria	Puede causar fugas	Control automático
Con Virtual Threads	Problemático (alto uso de memoria)	Eficiente y seguro
Visibilidad de uso	Implícito	Explícito

Ejemplo básico

```
// 1. Define un ScopedValue
final static ScopedValue<User> CURRENT_USER =
    ScopedValue.newInstance();

// 2. Establece el valor dentro de un ámbito
User user = new User("alice");

ScopedValue.where(CURRENT_USER, user)
    .run(() -> {
        // 3. Crea subtareass que pueden acceder al valor
        Thread.startVirtualThread(() -> {
            // Obtiene el valor actual (sólo lectura)
            User currentUser = CURRENT_USER.get();
            System.out.println(currentUser.name());
        });
    });
```

Beneficios clave

- ✔ Perfecto para **información de contexto** (usuario actual, transacción, etc.)
- ✔ Valor **inmutable** (previene errores)
- ✔ **Eficiencia** con millones de Virtual Threads

Casos de uso

- ➔ Información del **usuario actual** en un servidor web
- ➔ Contexto de **seguridad**
- ➔ ID de **transacción** en microservicios

📌 **Consejo para juniors:** Cuando necesites compartir datos entre muchos Virtual Threads que trabajan juntos, considera ScopedValue en lugar de ThreadLocal para mayor seguridad y eficiencia.