Concurrencia Estructurada en Java 25

De vuelta a la simplicidad sin sacrificar performance

Andrés Alcarraz

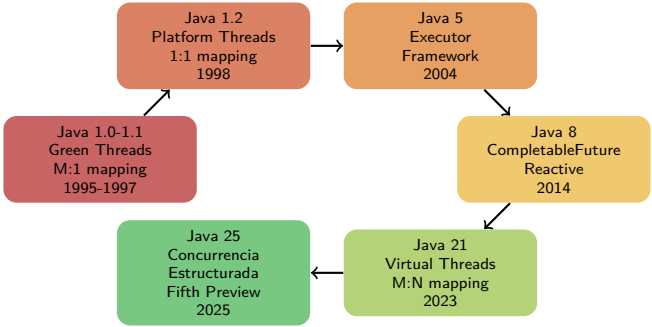
Peru JUG - JConf Perú 2025

3 de diciembre de 2025

¿Sobre mí?

- Uruguayo
- Ingeniero Eléctrico e Ingeniero de Sistemas por la Universidad de la República del Uruguay
- Staff Software Engineer en Cabal Uruguay S.A.
- Más de 25 años desarrollando en Java
- Redes sociales:
  - @ alcarraz@gmail.com
  - linkedin.com/in/andresalcarraz
  - X/andresalcarraz
  - github.com/alcarraz
  - stackoverflow/3444205

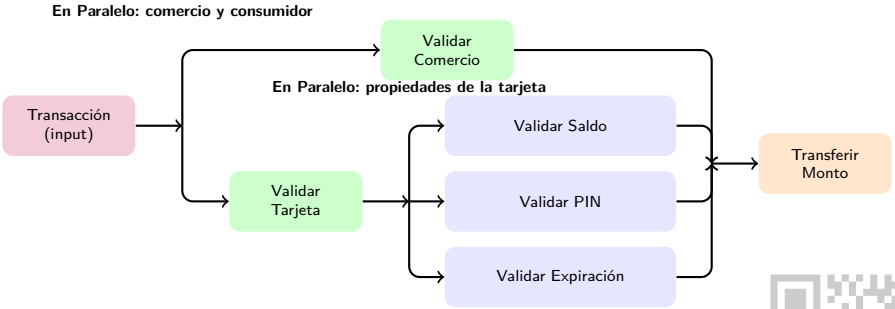
Agenda



Estado en Java 25

- Virtual Threads: **Estable** (desde Java 21)
- Scoped Values: **Estable** desde Java 25 [**JEP506**]
- Concurrencia Estructurada: **Quinta Preview** en Java 25 [**JEP505**]

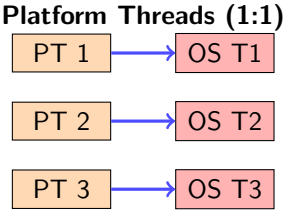
Flujo ficticio

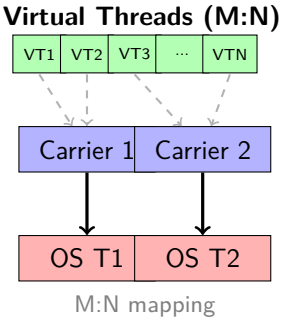


Flujo Optimizado

1. En paralelo: validar comercio y consumidor (tarjeta)
2. Validaciones paralelas de tarjeta (saldo, PIN, expiración)
3. Transferir sólo si todas las validaciones pasan

Agenda





Ventajas Clave

- Millones de Virtual Threads con pocos OS Threads
- Muy bajo costo de creación (1KB vs 2MB)
- Scheduling inteligente (park/unpark automático)

- **Escalabilidad masiva:** Bajo costo (creación + memoria) permite millones de hilos, habilita el patrón fork/join
- **Bloqueo simplificado:** Park/unpark automático, código bloqueante es natural
- **Scheduling automático:** Los VTs se administran solos, sin configurar pools
- **Modelo mental directo:** 1 tarea = 1 thread, mapeo natural

Comparación de Costos

Recurso	Platform Thread	Virtual Thread
Memoria (stack)	2 MB	1 KB
Tiempo creación	1 ms	1 µs
Context switch	10 µs	1 µs
Máximo práctico	miles	millones

Resultado

Hilos virtuales + Concurrencia estructurada = código coordinado con alta paralelización simple.

Agenda

Código de Ejemplo

```
public CompletableFuture<TransactionResult> processTransaction(TransactionRequest request) {
    return CompletableFuture
        .supplyAsync(() -> cardValidationService.validate(request.cardNumber()))
        .thenCompose(cardResult -> {
            if (!cardResult.success())
                return CompletableFuture.completedFuture(TransactionResult.failure(xxx));

            var balanceValidation = CompletableFuture.supplyAsync(() ->
                balanceService.validate(request.cardNumber(), request.amount()));
            // ... más validaciones paralelas

            return CompletableFuture.allOf(balanceValidation, pinValidation, xxx)
                .thenCompose(_ -> CompletableFuture.supplyAsync(() ->
                        balanceService.debit(request.cardNumber(), request.amount())));
        });
}
```

Problemas del Enfoque Reactivo

Complejidad Innecesaria

- **Callback Hell:** Lógica anidada difícil de seguir
- **Manejo de Errores:** Excepciones complejas con `CompletionException`
- **Manejo de recursos:** Difícil garantizar cleanup
- **Depuración:** Stack traces confusos
- **Carga cognitiva:** Alto costo mental para leer/mantener

Resultado

- Rendimiento: Ejecución paralela
- Legibilidad: Código complejo
- Mantenibilidad: Lógica de negocio mezclada con la lógica de control.
- Testing: Complicado de probar

Agenda

```java
// STRUCTURED: Código que se lee como se ejecuta
public TransactionResult processTransaction(TransactionRequest request) throws InterruptedExce
    ValidationResult cardResult = cardValidationService.validate(request.cardNumber());
    if (!cardResult.success()) return TransactionResult.failure(cardResult.message());

    try (var scope = StructuredTaskScope.open(StructuredTaskScope.Joiner.awaitAll())) {
        var balanceTask = scope.fork(() ->
            balanceService.validate(request.cardNumber(), request.amount()));
        // ... más validaciones paralelas

        scope.join();

        List<ValidationResult> results = List.of(balanceTask.get(), pinTask.get(), xxx);
        Optional<ValidationResult> failure = results.stream().filter(r -> !r.success()).findFi
        if (failure.isPresent())
            return TransactionResult.failure(failure.get().message());
    }
    ValidationResult debitResult = balanceService.debit(request.cardNumber(), request.amount()
    return debitResult.success() ? TransactionResult.success(xxx) : TransactionResult.failure(
}
```

Simplicidad y Poder

- **Código Legible:** Se lee como se ejecuta.
- **Manejo de errores simple:** Excepciones tradicionales.
- **Manejo de recursos:** Try-with-resources automático.
- **Depuración:** Stack traces normales.
- **Rendimiento:** Misma ejecución paralela.

Principios Clave

- **Ciclo de vida estructurado:** Tareas anidadas mueren con el alcance.
- **Fallar tempranamente por defecto:** `open()` cancela automáticamente si una tarea falla.
- **Opcionalmente, esperar por todos:** `open(awaitAll())` para recopilar todos los resultados.
- **Observabilidad:** Mejor monitoreo y depuración, volcado estructurado de hilos.

Demo Time!

Rezarle a los dioses de las demos 🙏

Demo Block 1: Concurrencia Estructurada

- `./gradlew demoReactive` - CompletableFuture básico
- `./gradlew demoReactiveExceptions` - Reactive con excepciones (¡no fail-fast!)
- `./gradlew demoStructured` - Concurrencia Estructurada limpio
- `./gradlew demoCompareFailure` - **Comparación fail-fast** (60 % más rápido)
- `./gradlew demoCompare` - Comparación caso éxito (paridad)

Escenarios de Prueba

- ✓ Transacción válida: Customer 12345, Card 4532-***-9012
- × Tarjeta expirada: Customer 12345, Card 5555-***-2222 (¡fail-fast!)

Agenda

Joiners Incluidos en Java 25

- `allSuccessfulOrThrow()` - Fail-fast (por defecto con `open()`)
- `awaitAll()` - Espera todas las tareas
- `anySuccessfulResultOrThrow()` - Primera tarea exitosa
- `allUntil(Instant)` - Todas las tareas hasta deadline

```
// Custom Joiner: Recoger solo N primeros resultados
try (var scope = StructuredTaskScope.open(
        Joiner.first(3))) {  // Espera primeros 3

    for (int i = 0; i < 10; i++) {
        scope.fork(() -> fetchData(i));
    }

    scope.join();  // Retorna cuando hay 3 exitosos
    List<Result> first3 = scope.results();
}
```

```
try (var outerScope = StructuredTaskScope.open()) {
    var userData = outerScope.fork(() -> {
        // Inner scope para validaciones paralelas del usuario
        try (var userScope = StructuredTaskScope.open()) {
            var profile = userScope.fork(() -> fetchProfile(userId));
            var preferences = userScope.fork(() -> fetchPreferences(userId));

            userScope.join();
            return new UserData(profile.get(), preferences.get());
        }
    });
    var orderData = outerScope.fork(() -> fetchOrders(userId));
    outerScope.join();
    return combine(userData.get(), orderData.get());
}
```


Ventaja

Cancela automáticamente TODOS los scopes anidados si falla el padre

```java
// Timeout con Instant deadline
Instant deadline = Instant.now().plus(Duration.ofSeconds(5));

try (var scope = StructuredTaskScope.open(
        Joiner.allUntil(deadline))) {

    var result1 = scope.fork(() -> slowOperation1());
    var result2 = scope.fork(() -> slowOperation2());
    var result3 = scope.fork(() -> slowOperation3());

    scope.join();  // Cancela todos si pasa el deadline

    return combineResults(
        result1.get(), result2.get(), result3.get());

} catch (TimeoutException e) {
    return TransactionResult.failure(
        "Timeout: operations took longer than 5 seconds");
}
```

× Anti-Pattern 1: Leaking Tasks

- **NO hagas:** Fork tasks sin esperar `join()`
- **Problema:** Tasks huérfanas, resource leaks
- **Solución:** Siempre usa try-with-resources

× Anti-Pattern 2: Thread Pools con Virtual Threads

- **NO hagas:** `Executors.newFixedThreadPool(...)` para VTs
- **Problema:** Overhead innecesario, los VTs no necesitan pooling
- **Solución:** Crea VTs directamente con `Thread.ofVirtual()`

× Anti-Pattern 3: Ignorar InterruptedException

- **NO hagas:** Catch vacío o sin lógica de cancelación
- **Problema:** VTs dependen de interrupciones para cancelación
- **Solución:** Maneja interrupciones en tareas forked

✓ 1. Try-With-Resources Siempre

- Usa `try (var scope = ...)` para garantizar cleanup
- El scope cancela tasks pendientes automáticamente
- Evita leaks y garantiza resource management correcto

✓ 2. Manejo de Excepciones en Tareas

- Captura y maneja excepciones dentro de cada tarea forked
- Usa joiner apropiado: fail-fast para críticos, awaitAll para recolección
- Propaga contexto de error de forma clara

✓ 3. Scoped Values para Contexto

- Usa `ScopedValue` en lugar de ThreadLocal
- Inmutable, thread-safe, heredado por tareas forked
- Más eficiente con millones de virtual threads

Agenda

```java
public class ScopedPaymentProcessor {
    public static final ScopedValue<TransactionRequest> TRANSACTION_REQUEST = ScopedValue.newInstance();
    public TransactionResult processTransaction(TransactionRequest request) {
        return ScopedValue.where(TRANSACTION_REQUEST, request).call(() -> {
            ValidationResult cardResult = cardValidationService.validate();
            if (!cardResult.success()) {
                auditLog("Transaction failed: " + cardResult.message());
                return TransactionResult.failure(cardResult.message());
            }
            try (var scope = StructuredTaskScope.open(StructuredTaskScope.Joiner.awaitAll())) {
                var balanceTask = scope.fork(balanceService::validate);  // ¡Sin parámetros!
                // ... más validaciones
                scope.join();
                checkResults(xxx);
                ValidationResult debitResult = balanceService.debit();
                return debitResult.success() ? TransactionResult.success(xxx) : TransactionResult.failure
            }
        });
    }
}
```

```
public class ScopedBalanceService {
    public ValidationResult validate() {
        TransactionRequest request = ScopedPaymentProcessor.TRANSACTION_REQUEST.get();  // ¡Sin parámetros!
        String cardNumber = request.cardNumber();
        auditLog("Starting balance validation for card: " + cardNumber.substring(cardNumber.length() - 4));
        DemoUtil.simulateNetworkDelay(500);
        if (request.amount().doubleValue() > 1000) {
            auditLog("Balance validation failed: Insufficient funds");
            return ValidationResult.failure("Balance Check: Insufficient funds");
        }

        auditLog("Balance validation successful");
        return ValidationResult.success("Balance Check: Validation successful");
    }

    public ValidationResult debit() {
        TransactionRequest request = ScopedPaymentProcessor.TRANSACTION_REQUEST.get();
        String cardNumber = request.cardNumber();
        auditLog("Starting debit for card: " + cardNumber.substring(cardNumber.length() - 4));
        DemoUtil.simulateNetworkDelay(300);
        return ValidationResult.success("Balance Debit: Amount successfully debited");
    }
}
```


► ¡Scoped Values son ESTABLES en Java 25!

No más ThreadLocal ni parameter drilling. **JEP 506**: ¡Propagación automática de contexto pronta para usar en producción!

Demo: Scoped Values en Acción

Propagación de contexto automática

¿Qué veremos?

- Definición de `ScopedValue<TransactionRequest>`
- Propagación automática a través de `scope.fork()`
- Acceso directo con `TRANSACTION_REQUEST.get()`
- Audit logs con los datos del request
- Integración perfecta con Concurrencia Estructurada.

Sin Scoped Values

Necesitarías: Pasar datos del `TransactionRequest` como parámetros a cada método, o usar ThreadLocal (problemático con hilos virtuales masivos).

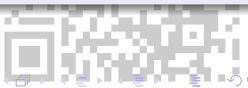
Agenda

Beneficios Inmediatos

- **Legibilidad:** Código que lee como se ejecuta
- **Mantenibilidad:** Más fácil de modificar y extender
- **Depuración:** Stack traces claros y útiles
- **Rendimiento:** Misma velocidad, mejor código
- **Manejo de recursos:** Garantías automáticas de cleanup

⋆ Roadmap de Adopción

- **HOY (Producción):** Scoped Values (**ESTABLES** en Java 25)
- **AHORA (Experimentar):** Concurrencia Estructurada (quinta preview - muy maduro)
- **Java 26:** Concurrencia Estructurada estable 🤞



1. **Scoped Values ya son estables:** Empieza a adoptarlos
2. **De vuelta a la simplicidad sin sacrificar performance**
3. **Concurrencia Estructurada aún está en preview**
4. **Código que lee como se ejecuta**
5. **Mejor experiencia del desarrollador sin trade-offs**

GitHub: `https://github.com/alcarraz/structured-concurrency`

Agenda

¿Preguntas?

¡Gracias!

Andrés Alcarraz
@ alcarraz@gmail.com

Agenda