

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?

- Uruguay 🇺🇾
- 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](https://www.linkedin.com/in/andresalcarraz)
 - [X/andresalcarraz](https://twitter.com/andresalcarraz)
 - github.com/alcarraz
 - [stackoverflow/3444205](https://stackoverflow.com/users/3444205)



Agenda

1 Introducción

- El Gancho
- Evolución de la Concurrencia en Java
- Problema de ejemplo: Procesamiento de Transacción Financiera

2 Implementación reactiva

3 Concurrencia Estructurada

4 La Funcionalidad Estrella: Fail-Fast

5 Características avanzadas

6 Comparación de Performance

7 Scoped Values

8 Conclusiones

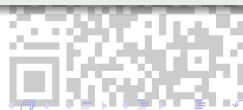
9 Preguntas?

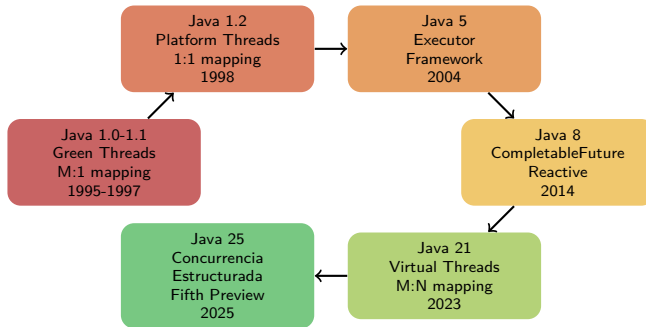


¿Y si pudiéramos escribir código bloqueante simple que supera al código reactivo complejo?

La Promesa de Java 25

- Código que **lee como se ejecuta**
- Performance **igual o mejor** que reactive
- Sin callbacks.
- **De vuelta a la simplicidad**

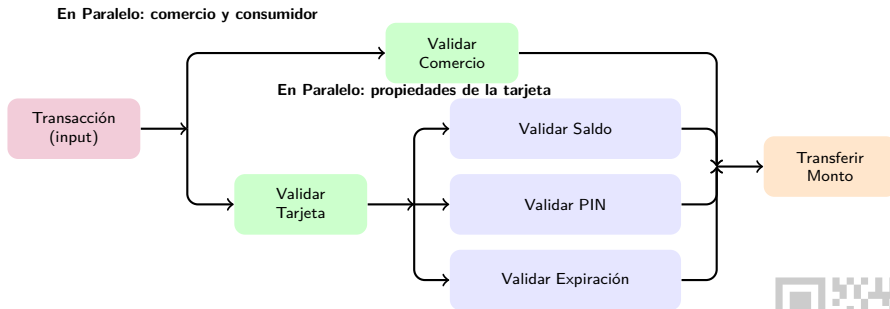




Estado en Java 25

- 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

- 1 Introducción
- 2 Implementación reactiva**
 - Demo: Reactive Basic con REST API
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Características avanzadas
- 6 Comparación de Performance
- 7 Scoped Values
- 8 Conclusiones
- 9 Preguntas?
- 10 Referencias



Código de Ejemplo

```
public CompletableFuture<TransactionResult> processTransaction(TransactionRequest request) {  
    CompletableFuture<ValidationResult> merchantValidation =  
        CompletableFuture.supplyAsync(() -> merchantValidationService.validate(request));  
    CompletableFuture<ValidationResult> consumerValidation =  
        CompletableFuture.supplyAsync(() -> {  
            ValidationResult cardResult = cardValidationService.validate(request);  
            if (!cardResult.success()) return cardResult;  
  
            var validations = cardValidations.stream()  
                .map(s -> CompletableFuture.supplyAsync(() -> s.validate(request)))  
                .toList();  
            CompletableFuture.allOf(validations.toArray(new CompletableFuture[0])).join();  
        });  
    return CompletableFuture.allOf(merchantValidation, consumerValidation)  
        .thenCompose(_ -> {  
            return CompletableFuture.runAsync(() -> balanceService.transfer(request))  
                .thenApply(_ -> TransactionResult.success(...));  
        });  
}
```



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

Probemos el approach reactivo

```
curl -X POST http://localhost:8080/api/reactive/basic \  
-H "Content-Type: application/json" \  
-d '{  
  "cardNumber": "1234-5678-9012-3456",  
  "expirationDate": "2512",  
  "pin": "1234",  
  "amount": 100.00,  
  "merchant": "Demo Store"  
}
```

Respuesta esperada

```
{  
  "success": true,  
  "transactionId": "uuid-here",  
  "amount": 100.00,  
  "message": "Transaction processed successfully",  
  "processingTimeMs": 520  
}
```

Observa

Funciona... pero el código es complejo. ¿Hay una mejor forma?

Agenda

- 1 Introducción
- 2 Implementación reactiva
- 3 Concurrencia Estructurada**
 - Ejemplo
 - Características
 - Demo: Structured Concurrency con REST API
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Características avanzadas
- 6 Comparación de Performance
- 7 Scoped Values
- 8 Conclusiones
- 9 Preguntas?



// STRUCTURED: Código que se lee como se ejecuta

```
public TransactionResult processTransaction(TransactionRequest request) throws InterruptedException {
    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()).findFirst();
        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(xxx);
}
```

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.

Problemas Structured Concurrency

```
curl -X POST http://localhost:8080/api/structured/normal \
-H "Content-Type: application/json" \
-d '{
  "cardNumber": "1234-5678-9012-3456",
  "expirationDate": "2512",
  "pin": "1234",
  "amount": 100.00,
  "merchant": "Demo Store"
}'
```

Respuesta esperada

```
{
  "success": true,
  "transactionId": "uuid-here",
  "amount": 100.00,
  "message": "Transaction processed successfully",
  "processingTimeMs": 520
}
```

¡Mismo performance, código más simple!

Agenda

- 1 Introducción
- 2 Implementación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast**
 - El Problema del Fail-Fast
 - Comparación de Código
 - Demo: La Comparación Dramática
 - Por Qué Esto Importa
- 5 Características avanzadas
- 6 Comparación de Performance
- 7 Scoped Values
- 8 Conclusiones



¿Qué pasa cuando una validación falla temprano?

Reactive Basic: Sin fail-fast

- `allOf()` espera que TODAS las tareas terminen
- Sin cancelación → desperdicio de recursos
- 570ms incluso si falla en 200ms

Reactive "Fixed": Fail-fast manual

- Se puede lograr... pero requiere lógica compleja
- `failFast.completeExceptionally()` + coordinación manual
- 150+ líneas, propenso a race conditions

Structured: Fail-fast automático

- `StructuredTaskScope.open()` = fail-fast por defecto

Reactive Basic (sin fail-fast)

```
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()));  
            var pinValidation = CompletableFuture.supplyAsync(() ->  
                pinValidationService.validate(request.cardNumber(), request.pin()));  
            var expirationValidation = CompletableFuture.supplyAsync(() ->  
                expirationService.validate(request.expirationDate()));  
  
            // allOf() ESPERA TODAS - sin cancelación  
            return CompletableFuture.allOf(balanceValidation, pinValidation, expirationValidation)  
                .thenCompose(_ -> {  
                    List<ValidationResult> results = List.of(  
                        balanceValidation.join(), pinValidation.join(), expirationValidation.join());  
  
                    Optional<ValidationResult> failure =  
                        results.stream().filter(r -> !r.success()).findFirst();  
                    if (failure.isPresent()) {  
                        balanceService.releaseAmount(request); // Rollback manual  
                        return CompletableFuture.completedFuture(  
                            TransactionResult.failure(failure.get().message()));  
                    }  
                })  
                .supplyAsync(() ->  
                    balanceService.transfer(request));  
        }  
    }  
}
```



Structured Fail-Fast (automático)

```
// FAIL-FAST: Cancela automáticamente al primer fallo
public TransactionResult processTransaction(TransactionRequest request)
    throws InterruptedException {
    ValidationResult cardResult = cardValidationService.validate(request.cardNumber());
    if (!cardResult.success())
        return TransactionResult.failure(cardResult.message());

    try (var scope = StructuredTaskScope.open()) { // open() sin joiner = fail-fast!
        var balanceTask = scope.fork(() -> {
            ValidationResult result = balanceService.validate(
                request.cardNumber(), request.amount());
            if (!result.success())
                throw new RuntimeException(result.message());
            return result;
        });
        var pinTask = scope.fork(() -> { /* similar */ });
        var expirationTask = scope.fork(() -> { /* similar */ });
    }
}
```



Problemos fail-fast con REST API

```
# Escenario: Tarjeta expirada (falla en 200ms)
curl -X POST http://localhost:8080/api/compare \
  -H "Content-Type: application/json" \
  -d '{
    "cardNumber": "1111-2222-3333-4444",
    "expirationDate": "2020",
    "pin": "1234",
    "amount": 50.00,
    "merchant": "Test Store"
  }'
```

Resultado Esperado

- **Reactive Basic:** 570ms (esperó todas las validaciones)
- **Structured Fail-Fast:** 230ms (canceló inmediatamente)
- **Mejora:** **60 % más rápido!**

Balance Locking

Impacto en la Experiencia de Usuario

- **Falla rápido:** Reporta errores inmediato, mejor UX
- **Menor latencia:** 60 % más rápido en casos de error
- **Feedback claro:** Usuario no espera innecesariamente

Impacto en Recursos y Costos

- **Eficiencia:** Cancela trabajo innecesario automáticamente
- **Ahorro de CPU:** No procesa validaciones que no se usarán
- **Resource safety:** Balance locks liberados automáticamente
- **Menor carga:** Sistema responde más rápido en picos de errores

Esto es Arquitectural, No Solo Performance

Fail-fast cambia cómo diseñamos sistemas: de .optimista y costoso.^a "pragmático y eficiente"

Agenda

- 1 Introducción
- 2 Implementación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 **Características avanzadas**
 - Custom Joiners: Estrategias de Coordinación
 - Scopes anidados: Composición Jerárquica
 - Timeout Patterns: Deadlines con Concurrencia Estructurada
 - Anti-Patterns: Lo Que NO Debes Hacer
 - Best Practices: Patrones Recomendados
- 6 Comparación de Performance
- 7 Scoped Values
- 8 Conclusiones



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

```
// 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

- 1 Introducción
- 2 Implementación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Características avanzadas
- 6 Comparación de Performance**
 - Performance: Caso Exitoso - Paridad
 - Performance: Caso Fallido - Ventaja Dramática
- 7 Scoped Values
- 8 Conclusiones
- 9 Preguntas?
- 10 Referencias



Cuando todas las validaciones pasan...

Comando: `./gradlew demoCompare`

- **Reactive (CompletableFuture):** 520ms
- **Concurrencia Estructurada:** 520ms
- **Resultado:** Misma performance en casos exitosos

¿Por qué iguales?

Ambos ejecutan las 4 validaciones en paralelo y esperan que todas terminen:

- Balance: 500ms, Card: 300ms, Expiration: 200ms, PIN: 400ms
- **Total:** $\max(500, 300, 200, 400) = 500\text{ms} + \text{overhead} \approx 520\text{ms}$



Cuando una validación falla...

Comando: `demoCompareFailure`

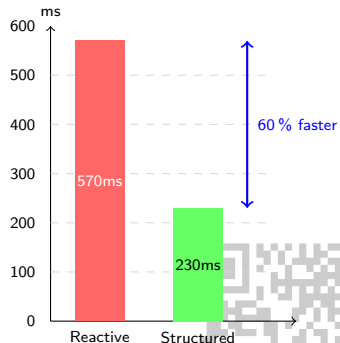
- **Reactive:** 570ms
- **Structured:** 230ms
- **Mejora:** **60 % más rápido**

¿Por qué?

- **Reactive:** `allOf()` espera TODAS
- **Structured:** Cancela automático

Resultado

Mejor UX y menor uso de recursos en errores



Agenda

- 1 Introducción
- 2 Implementación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Características avanzadas
- 6 Comparación de Performance
- 7 Scoped Values**
- 8 Conclusiones
- 9 Preguntas?
- 10 Referencias



```

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(xxx);
            }
        });
    }
}

```



```

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

- 1 Introducción
- 2 Implementación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Características avanzadas
- 6 Comparación de Performance
- 7 Scoped Values
- 8 Conclusiones**
 - ¿Por qué adoptar concurrencia estructurada?
 - Mensajes Clave
- 9 Preguntas?
- 10 Referencias



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

- 1 Introducción
- 2 Implementación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Características avanzadas
- 6 Comparación de Performance
- 7 Scoped Values
- 8 Conclusiones
- 9 Preguntas?**
- 10 Referencias



¿Preguntas?

¡Gracias!

Andrés Alcarraz
@ alcarraz@gmail.com



Agenda

- 1 Introducción
- 2 Implementación reactiva
- 3 Concurrencia Estructurada
- 4 La Funcionalidad Estrella: Fail-Fast
- 5 Características avanzadas
- 6 Comparación de Performance
- 7 Scoped Values
- 8 Conclusiones
- 9 Preguntas?
- 10 Referencias**



