Concurrencia Estructurada en Java 25

De vuelta a la simplicidad sin sacrificar performance

Andrés Alcarraz

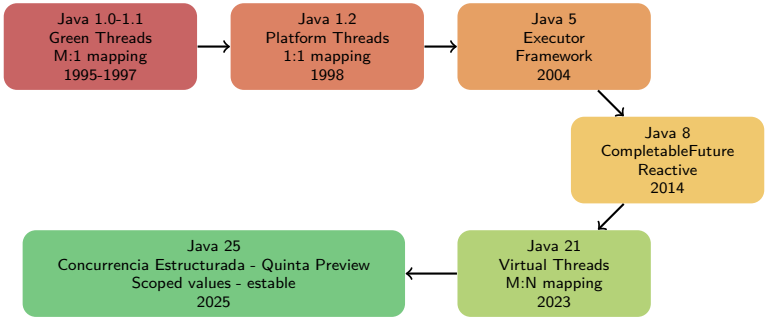
Peru JUG - JConf Perú 2025

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 [Ope25]
- Concurrencia Estructurada: **Quinta Preview** en Java 25 [Ope24]

¡La promesa!

- Código que **se lee como se ejecuta**

¡La promesa!

- Código que **se lee como se ejecuta**
- Rendimiento equiparable a programación reactiva, con beneficios.

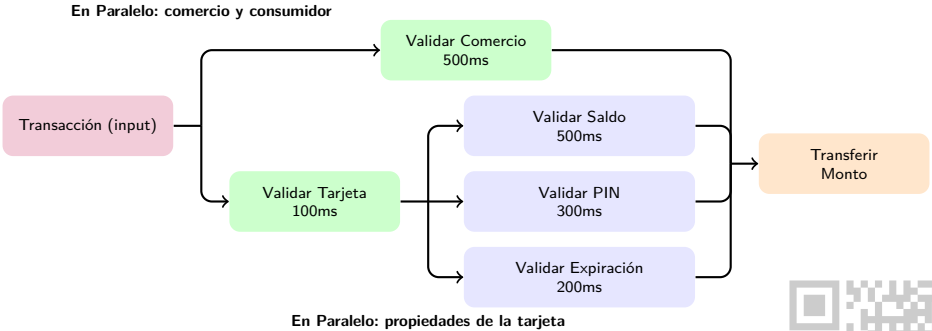
¡La promesa!

- Código que **se lee como se ejecuta**
- Rendimiento equiparable a programación reactiva, con beneficios.
- Sin callbacks.

¡La promesa!

- Código que **se lee como se ejecuta**
- Rendimiento equiparable a programación reactiva, con beneficios.
- Sin callbacks.
- **De vuelta a la simplicidad**

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

Problemas del Enfoque Reactivo

- **Legibilidad:** Callback Hell $\rightarrow$ Lógica anidada difícil de seguir

Problemas del Enfoque Reactivo

- **Legibilidad:** Callback Hell $\rightarrow$ Lógica anidada difícil de seguir
- **Manejo de recursos:** No es trivial garantizar su liberación.

Problemas del Enfoque Reactivo

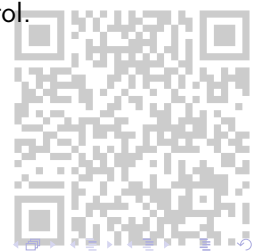
- **Legibilidad:** Callback Hell $\rightarrow$ Lógica anidada difícil de seguir
- **Manejo de recursos:** No es trivial garantizar su liberación.
- **Excepciones:** Trazas del stack confusas, pueden saltar en el framework en lugar del código que ls genera

Problemas del Enfoque Reactivo

- **Legibilidad:** Callback Hell $\rightarrow$ Lógica anidada difícil de seguir
- **Manejo de recursos:** No es trivial garantizar su liberación.
- **Excepciones:** Trazas del stack confusas, pueden saltar en el framework en lugar del código que ls genera
- **Carga cognitiva:** Alto costo mental para entender el código

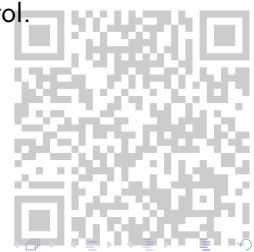
Problemas del Enfoque Reactivo

- **Legibilidad:** Callback Hell $\rightarrow$ Lógica anidada difícil de seguir
- **Manejo de recursos:** No es trivial garantizar su liberación.
- **Excepciones:** Trazas del stack confusas, pueden saltar en el framework en lugar del código que ls genera
- **Carga cognitiva:** Alto costo mental para entender el código
- **Mantenibilidad:** Lógica de negocio mezclada con la lógica de control.



Problemas del Enfoque Reactivo

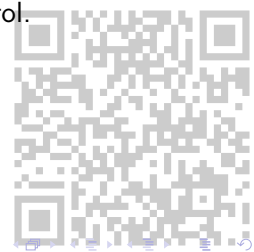
- **Legibilidad:** Callback Hell $\rightarrow$ Lógica anidada difícil de seguir
- **Manejo de recursos:** No es trivial garantizar su liberación.
- **Excepciones:** Trazas del stack confusas, pueden saltar en el framework en lugar del código que ls genera
- **Carga cognitiva:** Alto costo mental para entender el código
- **Mantenibilidad:** Lógica de negocio mezclada con la lógica de control.
- **Depuración:** Difícil de seguir la secuencia de instrucciones.



Problemas del Enfoque Reactivo

- **Legibilidad:** Callback Hell → Lógica anidada difícil de seguir
- **Manejo de recursos:** No es trivial garantizar su liberación.
- **Excepciones:** Trazas del stack confusas, pueden saltar en el framework en lugar del código que ls genera
- **Carga cognitiva:** Alto costo mental para entender el código
- **Mantenibilidad:** Lógica de negocio mezclada con la lógica de control.
- **Depuración:** Difícil de seguir la secuencia de instrucciones.

▶ Veamos un código de ejemplo



Agenda

- **Código Legible:** Se lee como se ejecuta

- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático

- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático
- **Depuración:** Las trazas del stack muestran dónde se produce el error.

- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático
- **Depuración:** Las trazas del stack muestran dónde se produce el error.
- **Rendimiento:** Ejecución paralela similar, no se pierde.

- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático
- **Depuración:** Las trazas del stack muestran dónde se produce el error.
- **Rendimiento:** Ejecución paralela similar, no se pierde.
- **Ciclo de vida estructurado:** Tareas anidadas mueren con el alcance

- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático
- **Depuración:** Las trazas del stack muestran dónde se produce el error.
- **Rendimiento:** Ejecución paralela similar, no se pierde.
- **Ciclo de vida estructurado:** Tareas anidadas mueren con el alcance
- **Fallar tempranamente por defecto:** `open()` cancela automáticamente si una tarea tira una excepción.

- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático
- **Depuración:** Las trazas del stack muestran dónde se produce el error.
- **Rendimiento:** Ejecución paralela similar, no se pierde.
- **Ciclo de vida estructurado:** Tareas anidadas mueren con el alcance
- **Fallar tempranamente por defecto:** `open()` cancela automáticamente si una tarea tira una excepción.
- **Observabilidad:** Mejor monitoreo y depuración, volcado estructurado de hilos

- **Código Legible:** Se lee como se ejecuta
- **Manejo de recursos:** Try-with-resources automático
- **Depuración:** Las trazas del stack muestran dónde se produce el error.
- **Rendimiento:** Ejecución paralela similar, no se pierde.
- **Ciclo de vida estructurado:** Tareas anidadas mueren con el alcance
- **Fallar tempranamente por defecto:** `open()` cancela automáticamente si una tarea tira una excepción.
- **Observabilidad:** Mejor monitoreo y depuración, volcado estructurado de hilos

▶ Show me the code!!

Agenda

¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

Reactive Basic: Sin fail-fast

- `allOf()` espera que TODAS las tareas terminen

¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

Reactive Basic: Sin fail-fast

- `allOf()` espera que TODAS las tareas terminen
- Sin cancelación → desperdicio de recursos

¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

Reactive Basic: Sin fail-fast

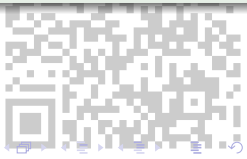
- `allOf()` espera que TODAS las tareas terminen
- Sin cancelación → desperdicio de recursos
- 570ms incluso si falla en 200ms

¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

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



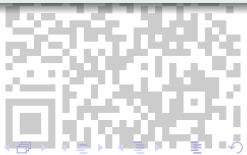
¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

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



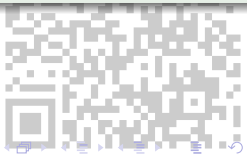
¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

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



¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

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

Structured: Fail-fast automático

¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

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

Structured: Fail-fast automático

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

¿Qué pasa cuando una validación falla antes de que las otras terminen de ejecutar?

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

Structured: Fail-fast automático

- `StructuredTaskScope.open()` = fail-fast por defecto
- Cancelación automática en cascada

Impacto en la Experiencia de Usuario

- **Falla rápido**: Reporta errores inmediatamente, mejor UX

Impacto en la Experiencia de Usuario

- **Falla rápido**: Reporta errores inmediatamente, mejor UX
- **Menor latencia** en casos de error

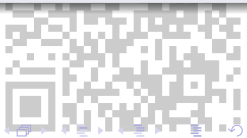
Impacto en la Experiencia de Usuario

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

Impacto en la Experiencia de Usuario

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

Impacto en Recursos y Costos

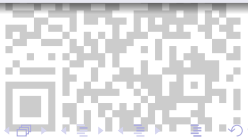


Impacto en la Experiencia de Usuario

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

Impacto en Recursos y Costos

- **Eficiencia**: Cancela trabajo innecesario automáticamente

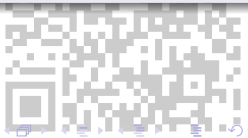


Impacto en la Experiencia de Usuario

- **Falla rápido**: Reporta errores inmediatamente, mejor UX
- **Menor latencia** 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.

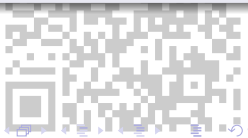


Impacto en la Experiencia de Usuario

- **Falla rápido**: Reporta errores inmediatamente, mejor UX
- **Menor latencia** 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.
- **Menor carga**: Sistema responde más rápido en picos de errores



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

Resumen de Performance: Éxito vs Fallo

Caso Exitoso: Paridad

- **Reactive**: 520ms
- **Structured**: 520ms
- **Resultado**: Misma performance

Caso Fallido: 60 % Más Rápido

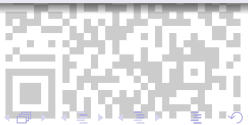
- **Reactive**: 570ms
- **Structured**: 230ms
- **Mejora**: **60 %**

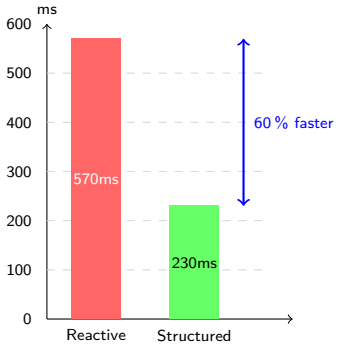
¿Por qué iguales?

Ambos ejecutan todas las validaciones en paralelo y esperan que terminen

¿Por qué?

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





Conclusión

Mejor UX y menor uso de recursos en errores. Mismo performance en éxitos.

Agenda

Scoped Values = Contexto Sin Parameter Drilling

El Problema

¿Cómo pasar contexto (audit info, request ID, user context) a través de validaciones paralelas?

- **Parameter drilling**: Pasar en cada firma de método (verbose, invasivo)
- **ThreadLocal**: Problemático con hilos virtuales masivos

La Solución: Scoped Values

- **Propagación automática**: `ScopedValue.where(KEY, value).run(...)`
- **Acceso directo**: `KEY.get()` desde cualquier método
- **Thread-safe**: Funciona perfectamente con structured concurrency
- **Casos de uso**: Audit trails, request IDs, user context, tracing

▶ ¡ESTABLE en Java 25!

JEP 506: Scoped Values ESTABLES

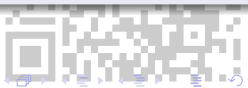
Agenda

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

[Ope24] OpenJDK. *JEP 505: Structured Concurrency (Fifth Preview)*. Accessed: 2025-09-16. 2024. URL: https://openjdk.org/jeps/505.

[Ope25] OpenJDK. *JEP 506: Scoped Values*. Accessed: 2025-09-16. 2025. URL: https://openjdk.org/jeps/506.