Script Detallado - Structured Concurrency Java 25 (15 minutos)

Preparación Técnica

- **Terminal preparado** con: `cd demo-structured-concurrency`
- **Gradle listo**: `./gradlew build` ejecutado previamente
- **Pantalla dividida**: Código visible + terminal para demos
- **Comandos preparados**: Todos los `./gradlew demo*` listos

[0:00-2:00] Introducción y Contexto (2 minutos)

[0:00-0:30] Presentación Personal

"¡Hola Peru JUG! Soy Andrés Alcarraz, uruguayo, 25 años desarrollando en Java. Hoy vamos a ver como Java 25 nos devuelve a la simplicidad sin sacrificar performance."

[0:30-1:30] Evolución de la Concurrencia en Java

[PANTALLA: Evolución de la Concurrencia]

"Primero, un repaso histórico. La concurrencia en Java ha evolucionado constantemente:

- **Green Threads** (Java 1.0-1.1, 1995-1997): Mapeo M:1, limitado por el OS
- **Platform Threads** (Java 1.2+, 1998): Mapeo 1:1 con threads del OS
- **Executor Framework** (Java 5, 2004): Abstracción sobre thread pools
- **CompletableFuture** (Java 8, 2014): Programación reactiva
- **Virtual Threads** (Java 21, 2023): Mapeo M:N, threads ligeros
- **Structured Concurrency** (Java 25, 2025): Quinta preview, coordinación de tareas"

[1:30-2:00] Estado Actual en Java 25

"¿Dónde estamos hoy? - **Virtual Threads**: ✅ Estables desde Java 21 - **Scoped Values**: ✅ **ESTABLES** en Java 25 - ¡adopten hoy! - **Structured Concurrency**: Quinta preview - muy maduro, experimenten ahora"

[2:00-3:00] El Problema de Negocio (1 minuto)

[2:00-3:00] Presentación del Caso de Uso

[PANTALLA: Flujo Realista de Validación y Débito]

“Ahora veamos un problema real: **procesar una transacción financiera** con un flujo optimizado:

Paso 1: Validar tarjeta primero (300ms) **Paso 2**: Si la tarjeta es válida, validaciones en paralelo: - ✅ Verificar saldo disponible (500ms) - ✅ Validar PIN (400ms) - ✅ Verificar fecha de expiración (200ms)

Paso 3: Debitar monto solo si todas las validaciones pasan

¿Por qué este orden? No tiene sentido validar el saldo de una tarjeta inválida. Este es un **flujo secuencial-paralelo realista**.”

[3:00-6:00] El Problema: Reactive Programming (3 minutos)

[3:00-4:00] Mostrar Código CompletableFuture

[PANTALLA: ReactivePaymentProcessor.java]

“Tradicionalmente usaríamos CompletableFuture. Veamos cómo implementamos el flujo secuencial-paralelo…”

[Mostrar el flujo con thenCompose]

```
// Paso 1: Validar tarjeta primero
return CompletableFuture
  .supplyAsync(() -> cardValidationService.validate(request.cardNumber()), executor)
  .thenCompose(cardResult -> {
    if (!cardResult.success()) {
      return CompletableFuture.completedFuture(TransactionResult.failure(...));
    }

    // Paso 2: Validaciones paralelas si tarjeta OK
    CompletableFuture<ValidationResult> balanceValidation = ...;
    // Paso 3: Debit si todo OK
    return CompletableFuture.allOf(...).thenCompose(...);
  });
```

“Observen la complejidad: thenCompose, callbacks anidados, manejo manual del flujo condicional.”

[4:00-5:00] Demo Reactive - Caso Exitoso

[TERMINAL]

```
./gradlew demoReactive
```

“Observen: ~520ms para caso exitoso. Funciona, pero el código es complejo.”

[5:00-6:00] Problemas del Approach Reactivo

"Los problemas del código reactive: - **Callback Hell**: Lógica anidada con .thenCompose() - **Manejo de Errores**: CompletionException wrapping - **Flujo Condicional Complejo**: if-else dentro de callbacks - **Complejidad Mental**: Código que no lee como se ejecuta"

[6:00-10:00] La Solución: Structured Concurrency (4 minutos)

[6:00-7:30] Mostrar Código Structured

[PANTALLA: StructuredPaymentProcessor.java]

"Ahora veamos Structured Concurrency - mismo flujo secuencial-paralelo, código elegante..."

[Mostrar el flujo estructurado]

```
// Paso 1: Validar tarjeta primero (secuencial)
ValidationResult cardResult = cardValidationService.validate(request.cardNumber());
if (!cardResult.success()) {
    return TransactionResult.failure(cardResult.message(), processingTime);
}

// Paso 2: Validaciones paralelas si tarjeta OK
try (var scope = StructuredTaskScope.open(StructuredTaskScope.Joiner.awaitAll())) {
    var balanceTask = scope.fork(() -> balanceService.validate(...));
    var pinTask = scope.fork(() -> pinValidationService.validate(...));
    var expirationTask = scope.fork(() -> expirationService.validate(...));

    scope.join(); // Espera todas o falla rápido

    // Paso 3: Debitar si todo OK
    ValidationResult debitResult = balanceService.debit(...);
}
```

"¡Observen! El código lee **exactamente como se ejecuta**: primero tarjeta, luego paralelo, luego debit."

[7:30-8:30] Demo Structured

[TERMINAL]

```
./gradlew demoStructured
```

"Mismo resultado, código muchísimo más limpio y legible."

[8:30-10:00] Ventajas Clave

"Las ventajas son enormes:

1. **Código Legible**: Lee exactamente como se ejecuta
2. **Flujo Natural**: Secuencial-paralelo sin callbacks
3. **Resource Management**: try-with-resources automático
4. **Error Handling Simple**: Excepciones Java tradicionales
5. **Architectural Simplicity**: Sin abstracción innecesaria"

[10:00-13:00] Comparación de Performance (3 minutos)

[10:00-11:00] Caso Exitoso - Misma Performance

[TERMINAL]

```
./gradlew demoCompare
```

"Comparación lado a lado - caso exitoso: **ambos ~520ms**. Misma performance cuando todo va bien."

[11:00-12:30] Fail-Fast vs AwaitAll

[PANTALLA: FailFastStructuredPaymentProcessor.java]

"Structured Concurrency también ofrece fail-fast automático con el joiner por defecto..."

[TERMINAL]

```
./gradlew demoCompareFailure
```

"¡Aquí está la diferencia arquitectural! En casos de fallo temprano: - **Fail-Fast Structured**: ~230ms (cancelación automática) - **Reactive Programming**: ~570ms (espera todas las tareas)

¡Más del doble de velocidad en casos de error!"

[12:30-13:00] Arquitectura vs Performance

"La ventaja no es solo performance - es **architectural simplicity**. Structured Concurrency nos da ambas opciones: - **awaitAll()**: Comportamiento similar a reactive - **Por defecto**: Fail-fast automático

Mismo API, diferentes estrategias de coordinación."

[13:00-14:30] Scoped Values: Context Propagation (1.5 minutos)

[13:00-14:00] Demo Scoped Values

[TERMINAL]

```
./gradlew demoScopedValues
```

“Scoped Values: context propagation sin ThreadLocal ni parameter drilling. Observen cómo cada validación accede automáticamente al correlation ID…”

[14:00-14:30] Estables y Listos

[ENFASIS FUERTE]

“**Scoped Values son ESTABLES en Java 25** - ¡adopten hoy en producción!

- No más `ThreadLocal.set()` y `remove()`
- No más parameter drilling
- Context inmutable y thread-safe automáticamente”

[14:30-15:00] Conclusiones y Llamada a la Acción (0.5 minutos)

[14:30-15:00] Mensajes Clave y Roadmap

“Mensajes para llevar:

1. **‘De vuelta a la simplicidad sin sacrificar performance’**
2. **Código que lee como se ejecuta**
3. **Architectural simplicity sobre abstracción innecesaria**

Roadmap de adopción: - **Hoy**: Scoped Values en producción - **Ahora**: Experimenten con Structured Concurrency - **Java 26**: Adopción completa cuando sea estable

GitHub: `structured-concurrency-java25`

¡Gracias Peru JUG! ¿Preguntas?”

Notas para el Presentador

Timing Crítico:

- **Total estricto**: 15 minutos
- **Buffer**: Mantener 30 segundos para Q&A
- **Demos rápidos**: Cada comando `./gradlew` debe ejecutarse sin demora

Pantalla:

- **Split view**: Código izquierda, terminal derecha

- **Font size**: Grande para visibility en stream
- **Preparar**: Todos los archivos abiertos en tabs

Comandos Secuenciales:

1. ./gradlew demoReactive
2. ./gradlew demoStructured
3. ./gradlew demoCompare
4. ./gradlew demoCompareFailure
5. ./gradlew demoScopedValues

Key Moments:

- **230ms vs 570ms**: Momento más impactante
- **"ESTABLES en Java 25"**: Enfatizar adoption path
- **Fail-fast automático**: Diferenciador clave vs reactive