

🔑 JAVA 21

# Gestión de Código Concurrente

Para desarrolladores junior



Hilos



Runnable



Executors



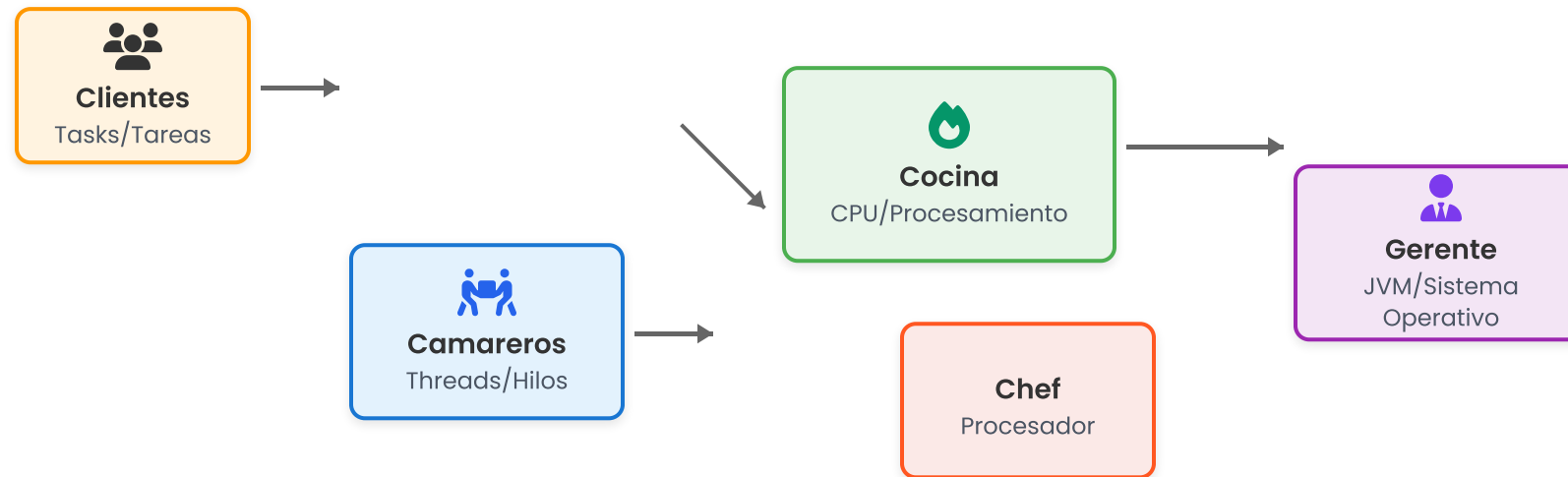
Streams

# Conceptos Básicos de Concurrency



## La Analogía del Restaurante

Comprender la concurrencia a través de un ejemplo cotidiano



### Concurrencia

Como varios camareros atendiendo diferentes mesas. Las tareas pueden iniciarse, ejecutarse y completarse en periodos superpuestos.

### = Paralelismo

Como múltiples cocineros trabajando simultáneamente en diferentes platos. Ejecución real al mismo tiempo en múltiples núcleos.



### Sincronización

Como coordinar el acceso a recursos compartidos. Solo un camarero puede usar la caja registradora a la vez para evitar errores.

### ¿Por qué usar concurrencia?

- 🌀 Mayor rendimiento
- 🔌 Aprovecha múltiples núcleos
- ⚙️ Evita bloqueos en operaciones de E/S

## ¿Qué es un hilo (thread)?

Una secuencia de instrucciones independientes que puede ser programada y ejecutada por el sistema operativo.



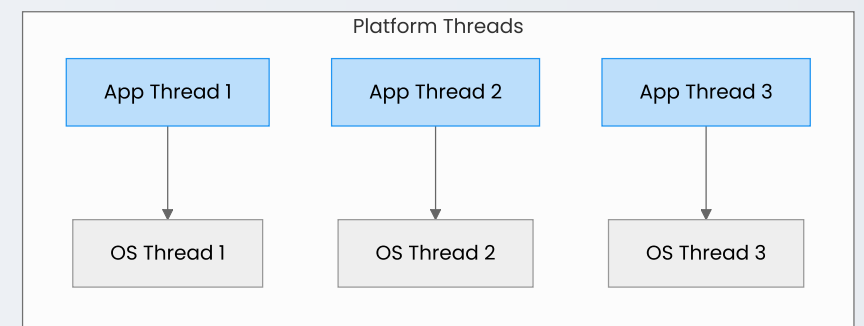
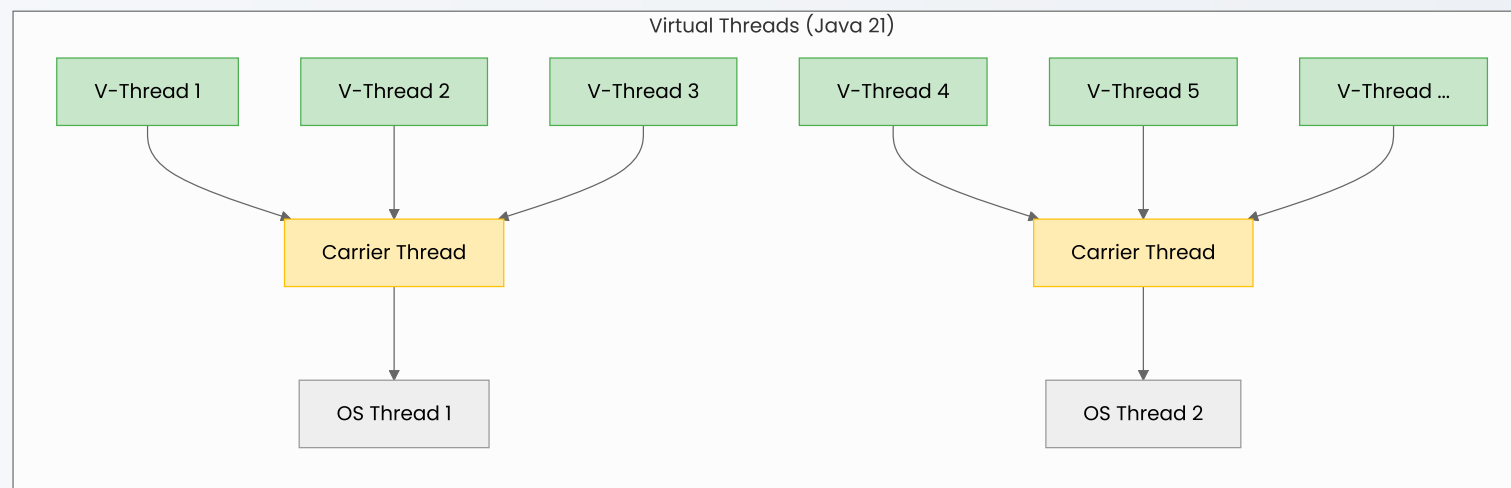
### Platform Threads

- ✓ Mapeo 1:1 con hilos del SO
- ✓ Recurso caro (1-2MB por hilo)
- ✓ Cambios de contexto costosos
- ✓ Limitados a miles de hilos



### Virtual Threads

- ✓ Gestionados por la JVM
- ✓ Ligeros (~1KB por hilo)
- ✓ Cambios de contexto eficientes
- ✓ Escalables a millones de hilos



#### Platform Thread

```
// Creación tradicional de hilo
Thread thread = new Thread(() -> {
    System.out.println("Ejecutando en platform thread");
});
thread.start();
```

#### Virtual Thread (Java 21)

```
// Creación de hilo virtual
Thread vThread = Thread.startVirtualThread(() -> {
    System.out.println("Ejecutando en virtual thread");
});
```

**Importante:** Los hilos virtuales son siempre daemon threads y están diseñados para tareas con I/O bloqueante, no para computación intensiva.



Runnable

@FunctionalInterface

Java 1.0

```
</> Runnable.java

public interface Runnable {
    void run();
}
```

- ✓ No devuelve resultado
- ✓ No puede lanzar excepciones checked
- ✓ Uso tradicional con Thread



Callable<V>

@FunctionalInterface

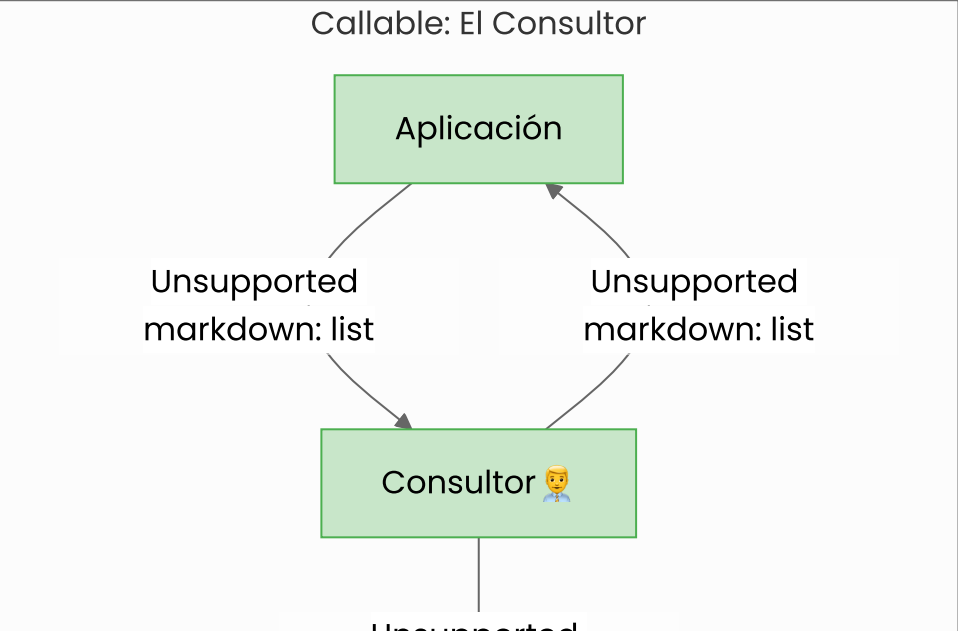
Java 1.5

```
</> Callable.java

public interface Callable<V> {
    V call() throws Exception;
}
```

- ✓ Devuelve un resultado genérico <V>
- ✓ Puede lanzar excepciones checked
- ✓ Uso con ExecutorService y Future

💡 Analogía: El Mensajero vs El Consultor



🏃 Ejemplo Runnable

```
// Creación y ejecución de Runnable
Runnable task = () -> {
    System.out.println("Ejecutando tarea sin retorno");
    // No retorna valor
};

// Opción 1: Con Thread
Thread thread = new Thread(task);
thread.start();

// Opción 2: Con ExecutorService
executor.execute(task);
```

📞 Ejemplo Callable

```
// Creación y ejecución de Callable
Callable<Integer> task = () -> {
    System.out.println("Calculando resultado...");
    if (error) throw new Exception("Error");
    return 42; // Retorna un valor
};

// Con ExecutorService y Future
Future<Integer> future =
    executor.submit(task);

// Obtener resultado (bloqueante)
Integer result = future.get();
```

❓ ¿Cuándo usar cada uno?

🏃 Usa **Runnable** para tareas que no necesitan devolver resultados: notificaciones, actualizaciones de UI, registro de eventos.

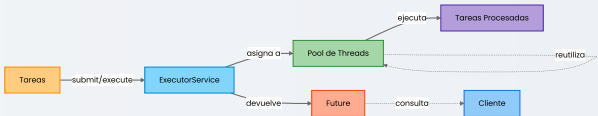
Mensajero 🏃

📞 Usa **Callable** para tareas que deben retornar valores, verificar excepciones o controlar su estado: cálculos, consultas a bases de datos, llamadas a APIs.



La agencia de contratación

Un **ExecutorService** es como una agencia de trabajo que gestiona trabajadores (hilos) y les asigna tareas. Tú le entregas las tareas a la agencia y ella se encarga de asignarlas a trabajadores disponibles, sin que tengas que gestionar directamente a cada trabajador.



SingleThreadExecutor

UN SOLO TRABAJADOR  
*Tareas en secuencia*

```
Executors.newSingleThreadExecutor()
```



FixedThreadPool

EQUIPO FIJO  
*N trabajadores permanentes*

```
Executors.newFixedThreadPool(n)
```



CachedThreadPool

EQUIPO FLEXIBLE  
*Crea hilos según demanda*

```
Executors.newCachedThreadPool()
```



VirtualThreadPerTask

HILOS VIRTUALES  
*Millones de trabajadores ligeros*

```
Executors.newVirtualThreadPerTaskExecutor()
```

```
</> Ejemplo de uso de ExecutorService

// Crear un pool con 4 hilos
ExecutorService executor = Executors.newFixedThreadPool(4);

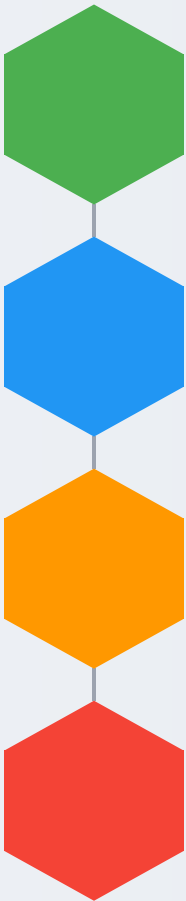
// Enviar tareas para ejecución
Future<String> future1 = executor.submit(() -> {
    return "Tarea 1 completada";
});

executor.execute(() -> {
    System.out.println("Tarea sin retorno");
});

// Obtener resultado (bloqueante)
String result = future1.get();

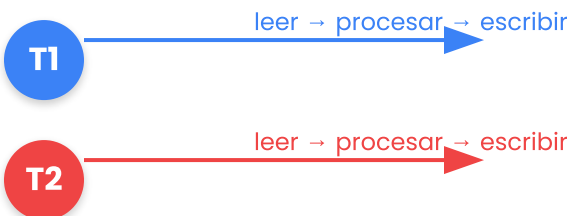
// Cerrar el executor correctamente
executor.shutdown();
boolean terminated = executor.awaitTermination(
    10, TimeUnit.SECONDS);
```

🔄 Ciclo de Vida de un ExecutorService



⚠️ **¡Importante!** Siempre debes cerrar correctamente un ExecutorService con shutdown() o shutdownNow(). De lo contrario, la aplicación podría no terminar porque los hilos del pool siguen activos.

El Problema: Condiciones de Carrera



cuenta:  
\$100

Cuando múltiples hilos acceden y modifican un mismo recurso sin coordinación, pueden producirse resultados inesperados y errores de consistencia.

synchronized

- ✓ **Intrínseco** - Incorporado en el lenguaje
- ✓ **Automático** - Libera bloqueo al salir del bloque
- ✓ **Simple** - Sintaxis concisa
- ✗ **Limitado** - Sin timeouts, tryLock o condiciones

ReentrantLock

- ✓ **Explícito** - Más control, desbloqueo manual
- ✓ **Avanzado** - Fairness, tryLock con timeout
- ✓ **Flexible** - Condiciones múltiples (await/signal)
- ✗ **Complejo** - Requiere try/finally para unlock

Ejemplo con synchronized

```

Bloque synchronized

public void depositarDinero(double monto) {
    // Usamos this como monitor
    synchronized (this) {
        double nuevoBalance = balance + monto;
        // Simulamos procesamiento
        sleep(50);
        balance = nuevoBalance;
    } // Liberación automática
}
```

```

Método synchronized

// Método completo synchronized
public synchronized void retirarDinero(double monto) {
    if (balance >= monto) {
        double nuevoBalance = balance - monto;
        sleep(50);
        balance = nuevoBalance;
    } else {
        throw new RuntimeException("Fondos insuficientes");
    }
}
```

Ejemplo con ReentrantLock

```

Usando ReentrantLock

// Campo de clase
private final Lock lock = new ReentrantLock(true); // fair

public void transferir(Cuenta destino, double monto) {
    // Intenta adquirir el lock con timeout
    try {
        if (lock.tryLock(1000, TimeUnit.MILLISECONDS)) {
            try {
                if (balance >= monto) {
                    balance -= monto;
                    destino.depositar(monto);
                } else {
                    throw new RuntimeException("Fondos insuficientes");
                }
            } finally {
                // ¡Importante! Siempre liberar el lock
                lock.unlock();
            }
        } else {
            throw new TimeoutException();
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
```

Mantén bloqueos pequeños  
Minimiza el código dentro de secciones críticas.

Evita bloqueos anidados  
Pueden causar interbloqueos (deadlocks).

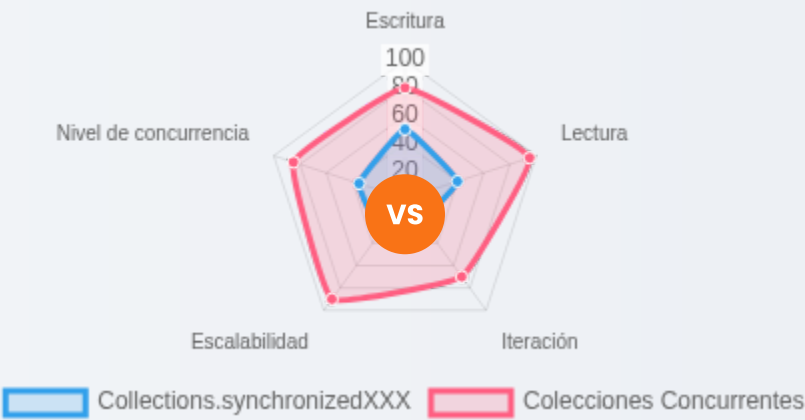
Considera la granularidad  
Bloqueos finos (por objeto) vs gruesos (globales).



## ¿Por qué necesitamos colecciones concurrentes?

Las colecciones estándar de Java (ArrayList, HashMap, etc.) **no son thread-safe** y las versiones sincronizadas tradicionales tienen problemas de rendimiento al escalar. Las colecciones concurrentes están diseñadas para altos niveles de concurrencia sin comprometer la seguridad.

### ↔ Sincronizado vs Concurrente



#### # ConcurrentHashMap

Reemplazo thread-safe para HashMap con mejor rendimiento que Collections.synchronizedMap()

- ✓ Segmentación interna (striping)
- ✓ Lecturas no bloqueantes
- ✓ Escrituras con bloqueo fino
- ✓ Operaciones atómicas compuestas

#### ☰ CopyOnWriteArrayList

Variante thread-safe de ArrayList optimizada para escenarios de muchas lecturas y pocas escrituras

- ✓ Lecturas sin sincronización
- ✓ Escrituras mediante copia
- ✓ Iteradores sin fallos
- ⚠ Costosa para modificaciones frecuentes

#### ☰ Otras Colecciones

- ConcurrentSkipListMap**  
Implementación concurrente de SortedMap basada en skip lists
- ConcurrentSkipListSet**  
Set ordenado concurrente
- CopyOnWriteArraySet**  
Set basado en CopyOnWriteArrayList
- ConcurrentLinkedQueue/Deque**  
Implementaciones de colas concurrentes

#### # Ejemplo ConcurrentHashMap

```
// Creación
Map<String, Integer> map = new ConcurrentHashMap<>();

// Operaciones atómicas compuestas
map.putIfAbsent("clave", 1);

Integer valorAnterior = map.computeIfPresent(
    "clave", (k, v) -> v + 1);

// Operación con ForEach paralelo
map.forEach(4, (k, v) ->
    System.out.println(k + ": " + v));

// Reducción paralela
Integer sum = map.reduceValues(
    10,
    v -> v,
    Integer::sum);
```

#### ☰ Ejemplo CopyOnWriteArrayList

```
// Creación
List<String> list = new CopyOnWriteArrayList<>();

// Agregar elementos
list.add("item1");
list.add("item2");

// Iteración thread-safe sin bloqueo
for (String item : list) {
    // Incluso si otro hilo modifica la lista
    // mientras iteramos, no veremos cambios
    // ni excepciones ConcurrentModificationException
    System.out.println(item);
}

// Comparación vs Collections.synchronizedList
List<String> syncList =
    Collections.synchronizedList(new ArrayList<>());
```

#### 💡 Cuándo usar cada colección

- # **ConcurrentHashMap** para mapas con alto nivel de concurrencia y múltiples lecturas/escrituras.
- ☰ **CopyOnWriteArrayList** para listas con muchas lecturas y escrituras ocasionales (como listeners).
- ☰ **ConcurrentSkipListMap** cuando necesitas ordenación y alto nivel de concurrencia.





¿Qué son los Streams Paralelos?

Los streams paralelos permiten procesar colecciones utilizando múltiples hilos, aprovechando los múltiples núcleos de la CPU para mejorar el rendimiento. Son especialmente útiles para conjuntos de datos grandes y operaciones independientes.



Syntax error in text  
mermaid version 11.6.0

Se secuencial vs Paralelo

```
// Stream secuencial
List<Integer> numeros = List.of(1, 2, 3, 4, 5, 6, 7, 8);

int suma = numeros.stream()
    .filter(n -> n % 2 == 0)
    .mapToInt(n -> n * 2)
    .sum();

// Stream paralelo (dos formas)
// 1. Usando parallelStream() directo
int sumaParalela = numeros.parallelStream()
    .filter(n -> n % 2 == 0)
    .mapToInt(n -> n * 2)
    .sum();

// 2. Convirtiendo stream() a parallel()
int sumaParalela2 = numeros.stream()
    .parallel()
    .filter(n -> n % 2 == 0)
    .mapToInt(n -> n * 2)
    .sum();
```

Operaciones avanzadas

```
// Reduce - combinando resultados
Optional<Integer> productoParalelo = numeros
    .parallelStream()
    .reduce((a,b) -> a * b);

// ForEach no determinista
numeros.parallelStream()
    .forEach(n ->
        System.out.println("Procesando: " + n));

// ForEachOrdered mantiene orden
numeros.parallelStream()
    .forEachOrdered(n ->
        System.out.println("En orden: " + n));

// Collect con paralelismo
Map<Boolean, List<Integer>> agrupadosPorParidad =
    numeros.parallelStream()
        .collect(Collectors.groupingBy(
            n -> n % 2 == 0));
```

Operaciones en Streams Paralelos

Intermedias

- filter
- map
- flatMap
- sorted
- distinct
- limit
- skip
- peek

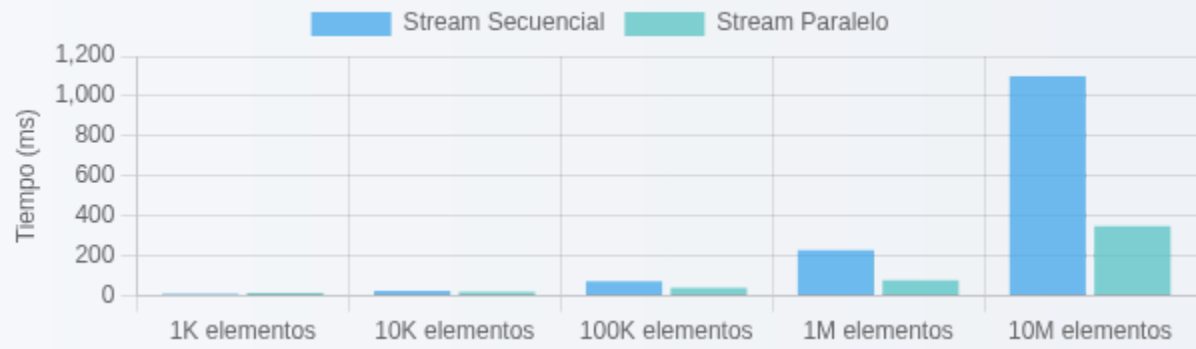
Terminales

- forEach
- forEachOrdered
- toArray
- reduce
- collect
- min/max
- count
- anyMatch
- allMatch
- noneMatch
- findFirst
- findAny

Consideraciones

- Las funciones deben ser **asociativas** y **sin estado**
- findFirst() es más costoso que findAny() en paralelo
- sorted() requiere recolectar todos los elementos

Rendimiento: Secuencial vs Paralelo



¿Cuándo usar Streams Paralelos?

Usar Paralelo cuando:

- Colecciones con muchos elementos (>10K)
- Operaciones CPU-intensivas
- Operaciones independientes
- Hardware con múltiples cores

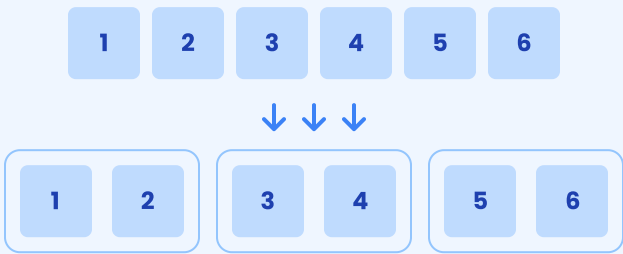
Usar Secuencial cuando:

- Conjuntos de datos pequeños
- Operaciones I/O-bound
- Operaciones con orden crítico
- Lambda con efectos secundarios



## Descomposición

División de un stream en partes independientes para procesamiento paralelo.

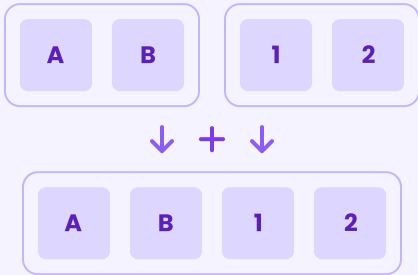


```
// Descomposición automática con parallelStream
List<Integer> numeros = List.of(1, 2, 3, 4, 5, 6);

int suma = numeros.parallelStream()
    .mapToInt(n -> n * 2)
    .sum();
// La división en chunks ocurre internamente
```

## Concatenación

Combinación de múltiples streams en uno solo para procesamiento unificado.



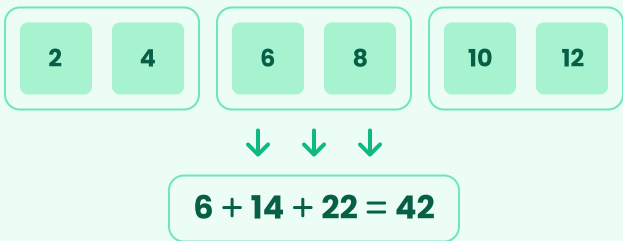
```
// Concatenación de streams
Stream<String> stream1 = Stream.of("A", "B");
Stream<String> stream2 = Stream.of("1", "2");

// Unir streams
Stream<String> streamConcatenado =
    Stream.concat(stream1, stream2);

// Mantener paralelismo
Stream<String> streamParalelo =
    streamConcatenado.parallel();
```

## Reducción

Combina los resultados de operaciones paralelas en un único resultado final.



```
// Reducción con identity, acumulador y combinador
List<Integer> numeros = List.of(1, 2, 3, 4, 5, 6);

int suma = numeros.parallelStream()
    .reduce(
        0, // identidad
        (a, b) -> a + b, // acumulador
        (a, b) -> a + b // combinador
    );

// Otras operaciones de reducción comunes
int max = numeros.parallelStream()
    .reduce(Integer::max)
    .orElse(0);
```


Organización de elementos basada en algún criterio (agrupamiento) o predicado (particionamiento).



```
// Agrupamiento paralelo
Map<String, List<Producto>> porCategoria =
    productos.parallelStream()
        .collect(Collectors.groupingByConcurrent(
            Producto::getCategoria
        ));

// Particionamiento paralelo
Map<Boolean, List<Integer>> pareseImpares =
    numeros.parallelStream()
        .collect(Collectors.partitioningBy(
            n -> n % 2 == 0
        ));
```


### Mejores Prácticas para Operaciones Avanzadas con Streams Paralelos



**Usa funciones asociativas** para la reducción: operaciones como suma, máximo o multiplicación funcionan bien, mientras que la resta no.



**Prefiere Collectors.groupingByConcurrent** sobre groupingBy para mejorar rendimiento en streams paralelos.



**Evita operaciones con estado** como sorted(), distinct() o limit() en streams paralelos cuando sea posible.

# Gestión de Código Concurrente en Java 21

Conclusiones y próximos pasos

## 💡 Conceptos Clave



### Hilos en Java 21

- ✓ Platform Threads (1:1 con SO)
- ✓ Virtual Threads (millones, ligeros)



### Runnable vs Callable

- ✓ Runnable: sin retorno
- ✓ Callable: con retorno y excepciones



### ExecutorService

- ✓ Pool de hilos reutilizables
- ✓ Virtual Thread Per Task Executor



### Bloqueos

- ✓ synchronized (intrínseco)
- ✓ ReentrantLock (explícito)



### Colecciones

- ✓ ConcurrentHashMap
- ✓ CopyOnWriteArrayList



### Streams Paralelos

- ✓ parallelStream(), reduce()
- ✓ Grouping, partitioning

## ★ Beneficios Clave

- **Mayor rendimiento**  
Aprovecha todos los núcleos de la CPU
- **Mejor tiempo de respuesta**  
Procesamiento en paralelo de tareas independientes
- **Eficiencia con I/O bloqueante**  
Virtual threads facilitan programación concurrente
- **Escalabilidad**  
Millones de hilos virtuales con Java 21

## 📖 Recursos Adicionales

📄 **Documentación Java 21**  
[docs.oracle.com](https://docs.oracle.com)

🔗 **Project Loom**  
[openjdk.org/projects/loom](https://openjdk.org/projects/loom)

🎓 **Baeldung: Java Concurrency**  
[baeldung.com/java-concurrency](https://baeldung.com/java-concurrency)

**¡Aplica la concurrencia con sabiduría!**

Recuerda: No siempre es necesaria, pero cuando lo es, puede transformar completamente el rendimiento de tu aplicación.