# Guía de Estudio: Threads en Java para Desarrolladores Junior

## Introducción: ¿Qué son los Threads?

Imagina que estás cocinando. Puedes hacer una sola cosa a la vez: cortar verduras, luego poner la olla, luego añadir ingredientes. Esto es similar a cómo funciona un programa tradicional de una sola tarea (single-threaded). Ahora, imagina que tienes un ayudante. Mientras tú cortas las verduras, tu ayudante puede ir poniendo la olla en el fuego. Ambos trabajan en paralelo en la misma cocina (el programa) para lograr el objetivo final (la comida lista) más rápido. En Java, estos "ayudantes" son los **Threads** (hilos).

Un **Thread** es la unidad más pequeña de ejecución dentro de un proceso. Un proceso puede tener múltiples hilos ejecutándose concurrentemente, compartiendo los mismos recursos (como la memoria), pero cada uno con su propio camino de ejecución (su propia pila de llamadas o *stack*).

**¿Por qué usar Threads?**

1. **Mejorar el rendimiento:** Permiten realizar múltiples tareas simultáneamente, aprovechando los procesadores multi-núcleo modernos. Tareas que pueden bloquearse (como esperar una respuesta de red o leer un archivo grande) pueden ejecutarse en un hilo separado, permitiendo que otros hilos continúen trabajando y manteniendo la aplicación responsiva.
2. **Interfaz de Usuario Responsiva:** En aplicaciones con interfaz gráfica (GUI), las tareas largas ejecutadas en el hilo principal pueden congelar la interfaz. Usar hilos separados para estas tareas mantiene la GUI fluida y receptiva a las acciones del usuario.
3. **Simplicidad en ciertos modelos:** A veces, modelar diferentes partes de una tarea compleja como hilos separados puede simplificar el diseño del programa.

Java proporciona un soporte robusto para la programación multihilo desde sus inicios, integrado directamente en el lenguaje y la plataforma (Java Virtual Machine - JVM). En esta guía, exploraremos los conceptos fundamentales que necesitas entender como desarrollador junior para empezar a trabajar con hilos en Java.

## Creación de Threads en Java

Java ofrece principalmente dos maneras de crear un hilo. Ambas son fundamentales y es importante entender cuándo usar cada una.

**1. Extendiendo la clase Thread:**

La forma más directa es crear una nueva clase que herede de la clase java.lang.Thread. Dentro de esta nueva clase, debes sobrescribir el método run(). Este método contiene el código que se ejecutará en el nuevo hilo. Una vez que tienes una instancia de tu clase personalizada, llamas a su método start() para iniciar la ejecución del hilo. Es crucial llamar a start() y no directamente a run(). Si llamas a run() directamente, el código se ejecutará en el hilo actual, no en uno nuevo.

class MiPrimerHilo extends Thread {

private String nombre;

public MiPrimerHilo(String nombre) {

this.nombre = nombre;

}

@Override

public void run() {

// El código que se ejecutará en este hilo

System.out.println("Ejecutando el hilo: " + nombre);

try {

// Simula una tarea que toma tiempo

for (int i = 0; i < 5; i++) {

System.out.println("Hilo " + nombre + ", contador: " + i);

Thread.sleep(500); // Pausa el hilo por 500 milisegundos

}

} catch (InterruptedException e) {

System.out.println("Hilo " + nombre + " interrumpido.");

}

System.out.println("Hilo " + nombre + " ha terminado.");

}

}

// Para usarlo:

public class EjemploCreacionThread {

public static void main(String[] args) {

MiPrimerHilo hilo1 = new MiPrimerHilo("Alfa");

MiPrimerHilo hilo2 = new MiPrimerHilo("Beta");

System.out.println("Iniciando hilos...");

hilo1.start(); // Inicia la ejecución del hilo1

hilo2.start(); // Inicia la ejecución del hilo2

System.out.println("Hilos iniciados.");

}

}

*Ventaja:* Simple y directo si tu clase representa conceptualmente un "hilo de ejecución". *Desventaja:* Java no permite herencia múltiple. Si tu clase ya necesita heredar de otra clase, no podrás usar este método.

**2. Implementando la interfaz Runnable:**

Este es el enfoque preferido y más flexible. Creas una clase que implementa la interfaz java.lang.Runnable. Esta interfaz solo define un método: run(). Al igual que antes, este método contiene el código que ejecutará el hilo. Para ejecutar este código en un nuevo hilo, creas una instancia de tu clase Runnable y luego la pasas al constructor de la clase Thread. Finalmente, llamas al método start() de la instancia de Thread.

class MiTarea implements Runnable {

private String nombre;

public MiTarea(String nombre) {

this.nombre = nombre;

}

@Override

public void run() {

// El código que se ejecutará en este hilo

System.out.println("Ejecutando la tarea: " + nombre);

try {

for (int i = 0; i < 5; i++) {

System.out.println("Tarea " + nombre + ", contador: " + i);

Thread.sleep(500);

}

} catch (InterruptedException e) {

System.out.println("Tarea " + nombre + " interrumpida.");

}

System.out.println("Tarea " + nombre + " ha terminado.");

}

}

// Para usarlo:

public class EjemploCreacionRunnable {

public static void main(String[] args) {

MiTarea tarea1 = new MiTarea("Gamma");

MiTarea tarea2 = new MiTarea("Delta");

Thread hilo1 = new Thread(tarea1);

Thread hilo2 = new Thread(tarea2);

System.out.println("Iniciando hilos con Runnable...");

hilo1.start();

hilo2.start();

System.out.println("Hilos iniciados.");

}

}

*Ventaja:* Permite que tu clase herede de otra clase si es necesario, ya que solo implementa una interfaz. Promueve una mejor separación de conceptos: la tarea (Runnable) está separada del mecanismo de ejecución (Thread). Una misma instancia de Runnable puede ser ejecutada por múltiples hilos si es necesario (aunque debes tener cuidado con el estado compartido). *Desventaja:* Requiere un paso extra (crear la instancia de Thread).

**¿Cuál elegir?** Generalmente, se prefiere implementar Runnable. Es más flexible y se alinea mejor con los principios de diseño orientado a objetos, separando la tarea a realizar de quién la realiza.

**Expresiones Lambda (Java 8+):**

Con Java 8 y versiones posteriores, puedes crear hilos de forma más concisa usando expresiones lambda, especialmente cuando la lógica del run() es simple:

public class EjemploLambda {

public static void main(String[] args) {

// Usando lambda para implementar Runnable

Runnable tareaLambda = () -> {

System.out.println("Ejecutando tarea desde Lambda");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println("Hilo Lambda interrumpido.");

}

System.out.println("Tarea Lambda terminada.");

};

Thread hiloLambda = new Thread(tareaLambda);

System.out.println("Iniciando hilo Lambda...");

hiloLambda.start();

System.out.println("Hilo Lambda iniciado.");

// Forma aún más corta

new Thread(() -> System.out.println("Otro hilo Lambda corriendo...")).start();

}

}

Esta sintaxis reduce la verbosidad para tareas sencillas, aprovechando que Runnable es una interfaz funcional.

## Ciclo de Vida de un Thread en Java

Un hilo en Java, desde su creación hasta su finalización, pasa por diferentes estados. Comprender este ciclo de vida es crucial para manejar hilos correctamente y depurar problemas de concurrencia. El diagrama que proporcionaste ilustra muy bien estos estados y transiciones. Vamos a detallarlos:

graph TD

New -- start() --> Runnable;

Runnable -- Obtiene CPU --> Running;

Running -- run() termina --> Terminated;

Running -- Cede CPU / Preempted --> Runnable;

Running -- Intenta adquirir lock (fallido) / Espera I/O --> Blocked;

Blocked -- Lock adquirido / I/O completada --> Runnable;

Running -- wait() / join() / park() --> Waiting;

Waiting -- notify() / notifyAll() / unpark() --> Runnable;

Running -- sleep(t) / wait(t) / join(t) / parkNanos(t) / parkUntil(t) --> TimedWaiting;

TimedWaiting -- Timeout / notify() / notifyAll() / unpark() --> Runnable;

subgraph "Estados Principales"

New

Runnable

Running

Blocked

Waiting

TimedWaiting

Terminated

end

style New fill:#f9f,stroke:#333,stroke-width:2px

style Runnable fill:#ccf,stroke:#333,stroke-width:2px

style Running fill:#cfc,stroke:#333,stroke-width:2px

style Blocked fill:#fcc,stroke:#333,stroke-width:2px

style Waiting fill:#ffc,stroke:#333,stroke-width:2px

style TimedWaiting fill:#fec,stroke:#333,stroke-width:2px

style Terminated fill:#ccc,stroke:#333,stroke-width:2px

*(Nota: Este diagrama Mermaid es una representación simplificada basada en la imagen y la explicación. El diagrama interactivo final será más detallado)*

**1. New (Nuevo):**

* **Descripción:** Cuando creas una instancia de la clase Thread (o una clase que la hereda), pero antes de llamar al método start(), el hilo se encuentra en este estado. Esencialmente, el objeto Thread existe, pero el hilo de ejecución real aún no ha comenzado en el sistema operativo.
* **Transición:** new Thread(...)
* **Siguiente Estado:** Runnable (al llamar a start()).

**2. Runnable (Ejecutable):**

* **Descripción:** Una vez que invocas el método start() sobre el objeto Thread, el hilo pasa al estado Runnable. En este estado, el hilo está listo para ejecutarse y está esperando que el planificador de hilos del sistema operativo (OS scheduler) le asigne tiempo de CPU. Es importante notar que Java agrupa los estados "Listo" (esperando CPU) y "Ejecutando" (con CPU asignada) bajo el término "Runnable". El diagrama que proporcionaste separa conceptualmente "Runnable" (listo) de "Running" (ejecutando), lo cual es útil para entender el flujo.
* **Transiciones de Entrada:**
  + Desde New: Llamando a start().
  + Desde Running: El planificador le quita la CPU (preemption) o el hilo cede voluntariamente la CPU (ej. yield()).
  + Desde Blocked/Waiting: Se obtiene el lock que estaba esperando.
  + Desde Waiting: Recibe una notificación (notify()/notifyAll()).
  + Desde Timed Waiting: El tiempo de espera expira o recibe una notificación.
* **Siguiente Estado:** Running (cuando el planificador le asigna CPU).

**3. Running (En Ejecución):**

* **Descripción:** Este es el estado en el que el hilo está activamente ejecutando el código dentro de su método run(). El planificador de hilos le ha asignado un intervalo de tiempo de CPU.
* **Transiciones de Entrada:**
  + Desde Runnable: El planificador le asigna CPU.
* **Siguientes Estados:**
  + Runnable: Si el planificador le quita la CPU o llama a yield().
  + Blocked/Waiting: Si intenta adquirir un lock que está en posesión de otro hilo.
  + Waiting: Si llama a Object.wait() (sin timeout), Thread.join() (sin timeout).
  + Timed Waiting: Si llama a Thread.sleep(), Object.wait(timeout), Thread.join(timeout).
  + Terminated: Si el método run() finaliza su ejecución normalmente o por una excepción no capturada.

**4. Blocked / Waiting (Bloqueado / Esperando por Lock):**

* **Descripción:** Un hilo entra en este estado cuando intenta ejecutar un bloque de código sincronizado (synchronized) o un método sincronizado, pero el *monitor lock* asociado a ese objeto ya está en posesión de otro hilo. El hilo permanecerá bloqueado hasta que el lock sea liberado y el planificador le permita adquirirlo. En la API de Java (Thread.State), este estado se llama BLOCKED.
* **Transiciones de Entrada:**
  + Desde Running: Intenta entrar en un bloque/método synchronized cuyo lock está ocupado.
* **Siguiente Estado:** Runnable (cuando adquiere el lock).

**5. Waiting (Esperando Indefinidamente):**

* **Descripción:** Un hilo entra en este estado cuando espera explícitamente por alguna condición o la finalización de otro hilo, sin un límite de tiempo. Esto ocurre típicamente al llamar a:
  + Object.wait(): Espera a que otro hilo llame a notify() o notifyAll() sobre el mismo objeto.
  + Thread.join(): Espera a que el hilo sobre el cual se llamó join() termine.
  + LockSupport.park(): Mecanismo de bloqueo más avanzado. Un hilo en estado WAITING no consume CPU y solo puede volver al estado Runnable si es explícitamente "despertado" por otro hilo.
* **Transiciones de Entrada:**
  + Desde Running: Llama a Object.wait(), Thread.join(), LockSupport.park().
* **Siguiente Estado:** Runnable (cuando recibe notify()/notifyAll(), el hilo unido termina, o recibe unpark()).

**6. Timed Waiting (Esperando con Tiempo Límite):**

* **Descripción:** Similar a WAITING, pero el hilo esperará solo durante un período de tiempo especificado. Entra en este estado al llamar a métodos como:
  + Thread.sleep(long millis): Pausa la ejecución por un tiempo determinado.
  + Object.wait(long timeout): Espera una notificación o hasta que pase el tiempo.
  + Thread.join(long millis): Espera la finalización de otro hilo o hasta que pase el tiempo.
  + LockSupport.parkNanos(long nanos) / LockSupport.parkUntil(long deadline). El hilo volverá a Runnable si es notificado antes, si es interrumpido, o si el tiempo de espera expira.
* **Transiciones de Entrada:**
  + Desde Running: Llama a sleep(), wait(timeout), join(timeout), parkNanos(), parkUntil().
* **Siguiente Estado:** Runnable (cuando expira el timeout, recibe notificación, es interrumpido, o recibe unpark()).

**7. Terminated / Dead (Terminado / Muerto):**

* **Descripción:** Un hilo alcanza este estado cuando ha completado la ejecución de su método run(). Una vez que un hilo está en estado TERMINATED, no puede volver a ningún otro estado ni ser iniciado de nuevo. El objeto Thread puede seguir existiendo, pero el hilo de ejecución asociado ha finalizado.
* **Transiciones de Entrada:**
  + Desde Running: El método run() finaliza (normalmente o por excepción).
* **Siguiente Estado:** Ninguno (estado final).

Comprender estas transiciones es fundamental para diseñar aplicaciones multihilo robustas y evitar problemas como *deadlocks* (bloqueos mutuos) o *race conditions* (condiciones de carrera).

## Sincronización de Threads en Java

Cuando múltiples hilos acceden y modifican recursos compartidos (como variables de instancia, objetos estáticos, archivos, etc.), pueden surgir problemas inesperados. Imagina que dos hilos intentan incrementar un contador al mismo tiempo. Ambos leen el valor actual (digamos 5), ambos calculan el nuevo valor (6), y ambos escriben 6 de vuelta. El resultado debería ser 7, pero es 6. Este es un ejemplo clásico de una **condición de carrera (race condition)**.

Para prevenir estos problemas, Java proporciona mecanismos de **sincronización**, que controlan el acceso de los hilos a los recursos compartidos, asegurando que solo un hilo pueda modificar el recurso a la vez.

**1. La Palabra Clave synchronized:**

Es el mecanismo de sincronización más básico y ampliamente utilizado en Java. Se puede aplicar de dos maneras:

* **Métodos Sincronizados:** Al declarar un método con la palabra clave synchronized, Java asegura que solo un hilo pueda ejecutar ese método *en una instancia particular del objeto* a la vez. Si un hilo está ejecutando un método sincronizado, cualquier otro hilo que intente ejecutar *cualquier* método sincronizado *en la misma instancia* será bloqueado hasta que el primer hilo salga del método.

class ContadorCompartido {

private int contador = 0;

// Solo un hilo a la vez puede ejecutar este método en esta instancia

public synchronized void incrementar() {

contador++;

}

public synchronized int getValor() {

return contador;

}

}

Si el método es static synchronized, el bloqueo se aplica a nivel de clase, no de instancia. Solo un hilo podrá ejecutar *cualquier* método estático sincronizado de esa clase a la vez.

* **Bloques Sincronizados:** Permiten sincronizar solo una parte del código dentro de un método, en lugar del método completo. Esto puede mejorar el rendimiento al reducir el tiempo que se mantiene el bloqueo. Requieren especificar un objeto que actuará como *monitor lock*. Solo un hilo puede poseer el lock de un objeto monitor en un momento dado.

public class EjemploBloqueSincronizado {

private final Object lockObjeto = new Object(); // Objeto dedicado para el lock

private int contador = 0;

public void hacerAlgo() {

// Código no sincronizado...

System.out.println("Hilo " + Thread.currentThread().getName() + " entrando a bloque sincronizado.");

synchronized (lockObjeto) {

// --- Inicio de la sección crítica ---

// Solo un hilo a la vez puede ejecutar este bloque

contador++;

System.out.println("Hilo " + Thread.currentThread().getName() + " incrementó contador a: " + contador);

// --- Fin de la sección crítica ---

}

// Más código no sincronizado...

}

}

Se puede usar this para sincronizar sobre la instancia actual, o NombreClase.class para sincronizar a nivel de clase (similar a métodos estáticos sincronizados).

**¿Cómo funciona synchronized?** Utiliza un concepto llamado *monitor* o *lock intrínseco* asociado a cada objeto Java. Cuando un hilo entra en un método o bloque sincronizado, intenta adquirir el lock del objeto monitor correspondiente. Si lo consigue, ejecuta el código. Al salir, libera el lock. Si no puede adquirir el lock (porque otro hilo ya lo tiene), entra en el estado BLOCKED hasta que el lock esté disponible.

**2. La Palabra Clave volatile:**

volatile es diferente de synchronized. No proporciona exclusión mutua (no bloquea). En cambio, garantiza dos cosas importantes para una variable:

* **Visibilidad:** Asegura que cualquier cambio hecho a la variable volatile por un hilo sea inmediatamente visible para otros hilos. Sin volatile, un hilo podría trabajar con una copia en caché de la variable, no viendo las actualizaciones de otros hilos.
* **Ordenamiento:** Previene ciertas reordenaciones de instrucciones por parte del compilador o la JVM que podrían afectar la lógica en entornos multihilo (garantía de *happens-before*).

Se usa típicamente para variables de estado simples (como un flag booleano para indicar la detención de un hilo) que son escritas por un hilo y leídas por otros.

class TrabajadorVolatile extends Thread {

private volatile boolean detener = false; // Variable de estado volátil

public void detenerTrabajo() {

detener = true;

}

@Override

public void run() {

while (!detener) {

// Hacer trabajo...

System.out.println("Trabajando...");

try {

Thread.sleep(100);

} catch (InterruptedException e) { /\* Ignorar \*/ }

}

System.out.println("Trabajador detenido.");

}

}

**3. wait(), notify() y notifyAll():**

Estos métodos de la clase Object se usan para la *coordinación* entre hilos que comparten un mismo lock (deben ser llamados desde un bloque o método sincronizado).

* wait(): Hace que el hilo actual libere el lock y entre en estado WAITING (o TIMED\_WAITING si se usa wait(timeout)), esperando a ser notificado.
* notify(): Despierta a *un* hilo que esté esperando (WAITING o TIMED\_WAITING) en el monitor de ese objeto. El hilo despertado no se ejecuta inmediatamente, sino que pasa a BLOCKED o RUNNABLE, compitiendo por el lock cuando este sea liberado.
* notifyAll(): Despierta a *todos* los hilos que estén esperando en el monitor de ese objeto.

Son la base para implementar patrones como Productor-Consumidor.

**4. El Paquete java.util.concurrent:**

Java 5 introdujo un paquete muy potente (java.util.concurrent) con utilidades de concurrencia más avanzadas y flexibles que los mecanismos básicos:

* **Locks Explícitos (Lock, ReentrantLock):** Ofrecen más control que synchronized (ej. intentar adquirir un lock sin bloquear, adquirirlo de forma interrumpible, fairness).
* **Variables Atómicas (AtomicInteger, AtomicBoolean, etc.):** Proporcionan operaciones atómicas (indivisibles) sobre tipos primitivos y referencias, a menudo sin necesidad de locks (usando operaciones hardware como Compare-And-Swap), lo cual puede ser más eficiente.
* **Colecciones Concurrentes (ConcurrentHashMap, CopyOnWriteArrayList, etc.):** Colecciones diseñadas para ser seguras en entornos multihilo con mejor rendimiento que envolver colecciones estándar con Collections.synchronizedMap/List.
* **Sincronizadores (Semaphore, CountDownLatch, CyclicBarrier, Phaser):** Clases para coordinar el flujo de ejecución entre múltiples hilos.
* **Executor Framework (ExecutorService, ThreadPoolExecutor, ScheduledExecutorService):** Un marco robusto para gestionar pools de hilos y la ejecución de tareas asíncronas.

Como desarrollador junior, es fundamental dominar synchronized y entender volatile. A medida que avances, explorar el paquete java.util.concurrent te permitirá escribir código concurrente más sofisticado y eficiente.

## Problemas Comunes en la Programación Multihilo

Si bien los hilos ofrecen grandes beneficios, también introducen una nueva clase de errores potenciales que pueden ser difíciles de diagnosticar y corregir. Aquí describimos algunos de los más comunes:

**1. Condiciones de Carrera (Race Conditions):**

* **Descripción:** Ocurren cuando el resultado de una computación depende del orden o el *timing* impredecible en que múltiples hilos acceden y modifican un recurso compartido. El ejemplo del contador incrementado incorrectamente que vimos antes es un caso clásico.
* **Causa:** Acceso concurrente no sincronizado a datos mutables compartidos.
* **Solución:** Usar mecanismos de sincronización (synchronized, locks explícitos, variables atómicas) para asegurar que el acceso al recurso compartido sea atómico o mutuamente exclusivo en las secciones críticas.

**2. Bloqueo Mutuo (Deadlock):**

* **Descripción:** Es una situación en la que dos o más hilos se bloquean indefinidamente, esperando cada uno a que el otro libere un recurso que necesita. Imagina dos hilos, Hilo A y Hilo B, y dos recursos, R1 y R2. Hilo A tiene R1 y necesita R2. Hilo B tiene R2 y necesita R1. Ninguno puede avanzar.
* **Causa:** Generalmente se produce cuando se cumplen las cuatro condiciones de Coffman:
  1. *Exclusión Mutua:* Los recursos no pueden ser compartidos (solo un hilo los usa a la vez).
  2. *Retención y Espera (Hold and Wait):* Un hilo mantiene al menos un recurso mientras espera adquirir otros recursos retenidos por otros hilos.
  3. *No Apropiación (No Preemption):* Un recurso solo puede ser liberado voluntariamente por el hilo que lo posee.
  4. *Espera Circular (Circular Wait):* Existe una cadena de hilos esperando, donde Hilo1 espera por Hilo2, Hilo2 por Hilo3, ..., y el último hilo espera por Hilo1.
* **Solución:** Romper una de las condiciones de Coffman. La estrategia más común es evitar la espera circular estableciendo un orden global para adquirir los locks. Si todos los hilos adquieren los locks R1 y R2 siempre en el mismo orden (ej. siempre R1 primero, luego R2), el deadlock no puede ocurrir. Otras técnicas incluyen usar timeouts al intentar adquirir locks (Lock.tryLock(timeout)) o algoritmos de detección y recuperación de deadlocks (más complejos).

**3. Inanición (Starvation):**

* **Descripción:** Ocurre cuando un hilo es sistemáticamente impedido de acceder a los recursos que necesita para progresar, a pesar de que estos estén disponibles. Por ejemplo, si el planificador de hilos siempre da prioridad a otros hilos, o si un hilo de alta prioridad monopoliza la CPU, un hilo de baja prioridad podría nunca ejecutarse.
* **Causa:** Políticas de planificación injustas, uso inadecuado de prioridades de hilos, o algoritmos de sincronización que no garantizan la equidad (ej. synchronized no garantiza que el hilo que más tiempo lleva esperando obtenga el lock).
* **Solución:** Usar mecanismos de bloqueo justos (new ReentrantLock(true)), diseñar cuidadosamente las prioridades de los hilos (y usarlas con moderación), y asegurar que los hilos liberen los recursos de manera oportuna.

**4. Livelock:**

* **Descripción:** Es similar al deadlock en que los hilos no pueden progresar, pero a diferencia del deadlock, los hilos no están bloqueados, sino que están activamente intentando responder a las acciones de los otros, sin lograr avanzar. Imagina dos personas intentando pasarse en un pasillo estrecho; ambos se mueven a un lado, luego al otro, simultáneamente, y nunca logran pasar.
* **Causa:** Algoritmos de recuperación de conflictos mal diseñados donde las acciones de recuperación de varios hilos se interfieren mutuamente.
* **Solución:** Introducir aleatoriedad en los reintentos o en los tiempos de espera, o rediseñar el protocolo de interacción para evitar estos ciclos de respuesta improductivos.

**5. Problemas de Visibilidad y Ordenamiento:**

* **Descripción:** Como mencionamos al hablar de volatile, sin la sincronización adecuada, los cambios realizados por un hilo en la memoria compartida pueden no ser visibles para otros hilos, o las operaciones pueden ser reordenadas por el compilador o la CPU de maneras inesperadas.
* **Causa:** Falta de sincronización (ausencia de relaciones *happens-before*).
* **Solución:** Usar synchronized, volatile, locks explícitos o variables atómicas para establecer las garantías de memoria necesarias (visibilidad y ordenamiento) entre las operaciones de diferentes hilos.

Entender estos problemas es el primer paso para evitarlos. Escribir código concurrente correcto requiere cuidado, atención al detalle y un buen conocimiento de los mecanismos de sincronización y el modelo de memoria de Java.