

## UD2. Sincronización de tareas

Concurrencia y Sistemas Distribuidos



# Objetivos de la Unidad Didáctica

---

- ▶ Revisión del concepto de programación concurrente
  - ▶ Identificar los problemas que conlleva la programación concurrente
    - ▶ Interferencias y errores de inconsistencia de memoria.
  - ▶ Conocer los mecanismos de cooperación necesarios para implantar una aplicación concurrente
    - ▶ Mecanismos de comunicación y de sincronización
  - ▶ Identificar las partes del código que pueden ocasionar problemas (secciones críticas) y cómo protegerlas
- ▶ Conocer un lenguaje de programación que da soporte a la programación concurrente: Concurrencia en JAVA
  - ▶ Introducir los mecanismos básicos del lenguaje Java para soportar la programación concurrente



## Contenido

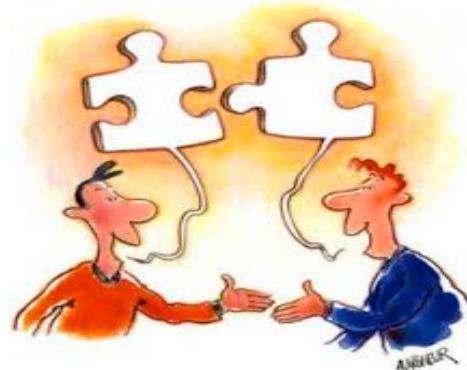
---

- ▶ **Sincronización de tareas**
  - ▶ Mecanismos de comunicación
    - ▶ Problemas de la comunicación con memoria compartida
  - ▶ Modelo de ejecución. Determinismo
  - ▶ Tipos de sincronización
    - ▶ Exclusión mutua
      - Sección Crítica
      - Locks
    - ▶ Sincronización condicional

## Concurrencia entre hilos: requisitos

---

- ▶ La concurrencia implica parallelismo y cooperación de los hilos
- ▶ La **cooperación** requiere:
  - ▶ a) comunicación (intercambio de información)
  - ▶ b) sincronización (establecer determinado orden en casos concretos)





# Mecanismos de comunicación

- ▶ **Mecanismos de comunicación**
  - a) **Memoria compartida:** los hilos comparten espacio de memoria (variables u objetos compartidos)
  - b) **Intercambio de mensajes:** emisor/receptor potencialmente en espacios de memoria disjuntos
- ▶ Nos centramos aquí en *memoria compartida* (opción a)
  - ▶ Requiere mecanismos de sincronización para coordinar las tareas
  - ▶ Mucha mayor difusión
    - ▶ Utilizado en lenguajes como Java y C#, y bibliotecas como pthreads



# Problemas de la comunicación con memoria compartida

## ► **Interferencias entre hilos**

- ▶ Pueden aparecer errores cuando múltiples hilos acceden a datos compartidos.

## ► **Errores de inconsistencia de memoria**

- ▶ Pueden aparecer errores de vistas inconsistentes de la memoria compartida.

Con la **Sincronización** se previenen estos problemas



## Interferencias de hilos → Veamos un ejemplo

- ▶ EJEMPLO: tenemos un objeto c de tipo Counter y dos hilos A y B
  - ▶ A ejecuta 20 veces c.increment()
  - ▶ B ejecuta concurrentemente 20 veces c.decrement()
- ▶ El valor final de c debería ser 0
- ▶ Pero la ejecución **no es determinista**
  - ▶ El valor final depende del **intercalado exacto** (planificación) al ejecutar A y B
    - ▶ Se producen **interferencias** cuando dos operaciones, ejecutándose en diferentes hilos, pero actuando sobre los mismos datos, se **intercalan**.

```
public class Counter {  
    private int c = 0;  
    public void increment() {  
        c++;  
    }  
    public void decrement() {  
        c--;  
    }  
    public int value() {  
        return c;  
    }  
...  
Counter c= new Counter(); //inic. a 0
```



## Interferencias de hilos → Veamos un ejemplo

- ▶ Ojo! Incluso las sentencias simples pueden traducirse en múltiples pasos en la máquina virtual
- ▶ Ejemplo: **c++** se puede descomponer en 3 pasos
  1. Obtener el valor actual de **c**
  2. Incrementar el valor obtenido en 1
  3. Almacenar el valor resultante en **c**

```
public class Counter {  
    private int c = 0;  
    public void increment() {  
        c++;  
    }  
    public void decrement() {  
        c--;  
    }  
    public int value() {  
        return c;  
    }  
...  
Counter c= new Counter(); //inic. a 0
```

(lo mismo para **c--**, pero decrementando)



# Interferencias de hilos → Veamos un ejemplo

## ▶ Ejemplo de intercalado:

- ▶ El valor inicial de **c** es 0
- ▶ **Hilo A** invoca *c.increment()* a la vez que **Hilo B** invoca *c.decrement()*

1. Thread A: Obtiene valor de c (0)
2. Threab B: Obtiene valor de c (0)
3. Thread A: Incrementa el valor obtenido; el resultado es 1
4. Thread B: Decrementa el valor obtenido; el resultado es -1.
5. Thread A: Almacena el resultado en c; c ahora vale 1
6. Thread B: Almacena el resultado en c; c ahora vale -1.

```
public class Counter {  
    private int c = 0;  
    public void increment() {  
        c++;  
    }  
    public void decrement() {  
        c--;  
    }  
    public int value() {  
        return c;  
    }  
}  
...  
Counter c= new Counter(); //inic. a 0
```

Código **NO  
DETERMINISTA**

Se ha producido una **condición  
de carrera**



## Errores de consistencia de memoria

---

- ▶ Los errores de consistencia de memoria ocurren cuando diferentes hilos tienen vistas inconsistentes de lo que debería ser el mismo dato.
  - ▶ Causas:
    - ▶ El compilador puede reordenar las instrucciones para optimizar la ejecución
    - ▶ Los procesadores pueden mover datos entre registros, cachés de memoria y memoria principal en distinto orden que el especificado por el programa.
  - ▶ Solución: relación ocurre-antes
    - ▶ Garantizar que la escritura en memoria de una sentencia es visible por otra determinada sentencia.
-

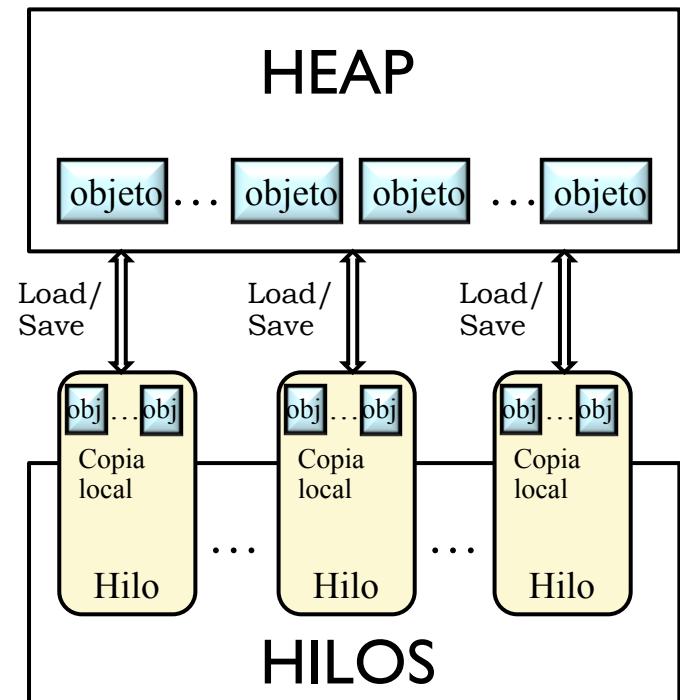


## Errores de consistencia de memoria

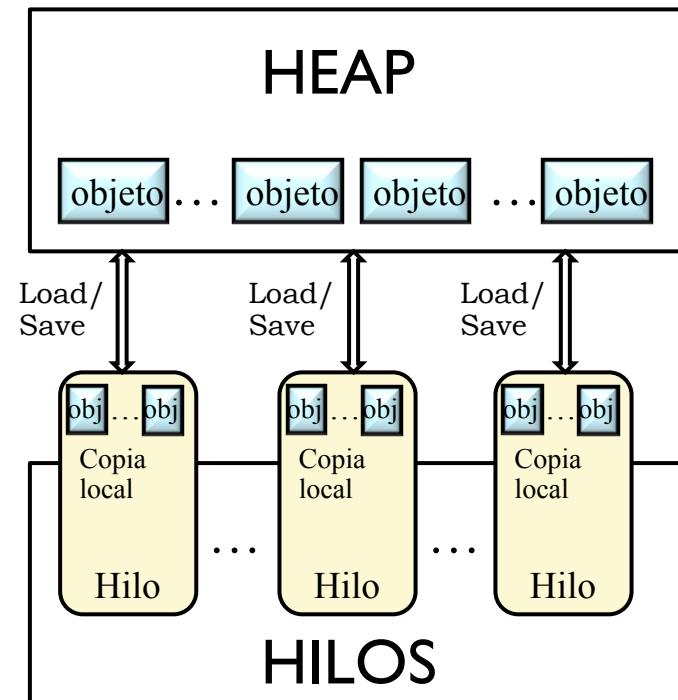
- ▶ Ejemplo de errores de consistencia de memoria:
  - ▶ Dos hilos A, B comparten el campo *int counter*, inicialmente inicializado a 0.
  - ▶ El hilo A incrementa su valor: `counter++;`
  - ▶ Justo después, el hilo B imprime el valor: `System.out.println(counter)`
  - ▶ No hay garantía que el cambio del hilo A sobre *counter* sea ya visible por el hilo B cuando éste imprime su valor.
  - ▶ A menos que el programador haya establecido una relación **ocurre-antes** sobre estas dos sentencias

Se requiere **Sincronización**

- ▶ Java utiliza un modelo de memoria compartida (objetos compartidos)
  - ▶ Denominamos **objeto compartido** a aquel objeto que utilizarán de forma activa dos o más hilos
  - ▶ Todos los objetos compartidos están definidos en el **heap**, y por tanto son accesibles desde cualquier hilo del proceso
  - ▶ Cada hilo posee una copia local de los objetos que manipula
    - ▶ Los cambios en la copia local no son visibles para otros hilos



- ▶ Copiar de *heap* a copia local o viceversa puede no ser una **operación atómica**
  - ▶ Ej. con long, double
- ▶ Por eficiencia
  - ▶ El compilador puede reordenar instrucciones
  - ▶ Puede que los valores se modifiquen sólo en la copia local temporalmente
- ▶ La JVM no garantiza que los cambios en las copias locales actualicen inmediatamente el *heap*





## Relaciones ocurre-antes en Java

---

- ▶ Acciones que crean una relación **ocurre-antes** en Java:
  - ▶ **Thread.start()**
    - ▶ Los efectos del código que dieron lugar a la creación del nuevo hilo son visibles para ese nuevo hilo.
  - ▶ **Thread.join()**
    - ▶ Los efectos del código en el hilo que ha terminado son ahora visibles para el hilo que ha realizado el *join*.
    - ▶ Se garantiza así que la copia local del hilo que ha terminado se ha actualizado en el heap.
  - ▶ **Mecanismos de Sincronización**



## Contenido

---

- ▶ **Sincronización de tareas**
  - ▶ Mecanismos de comunicación
    - ▶ Problemas de la comunicación con memoria compartida
  - ▶ **Modelo de ejecución. Determinismo**
  - ▶ Tipos de sincronización
    - ▶ Exclusión mutua
      - Sección Crítica
      - Locks
    - ▶ Sincronización condicional



## Modelo de ejecución

- ▶ Un hilo transforma su estado mediante la ejecución de sentencias

**Sentencia:** secuencia de **acciones atómicas** que realizan transformaciones indivisibles

**Acción Atómica:** transforma el estado y no puede dividirse en acciones menores

- ▶ Ejemplos de acciones atómicas:
  - ▶ Las instrucciones máquina no interrumpibles
  - ▶ Varias acciones agrupadas como una **acción atómica** [...]
    - Si  $[y:=x; z:=y]$  es una acción atómica, no podemos observar ningún estado con  $x \neq z$



## ¿Cómo garantizar la coherencia?

- ▶ Agrupando los pasos de transformación del estado en una *acción atómica*.
  - ▶ Se garantiza que los demás hilos sólo pueden acceder a **estados consistentes** del objeto
  - ▶ Los estados intermedios (incoherentes) no son visibles para los otros hilos

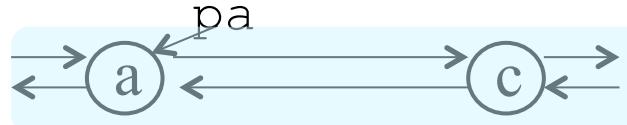
**Estado consistente**: estado que cumple todos los invariantes del objeto

# Ejemplo.- Inserta **b** entre **a** y **c** en lista doblemente enlazada

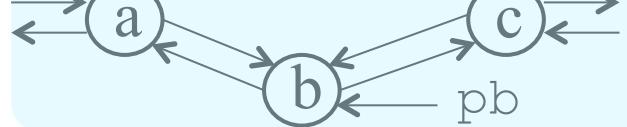
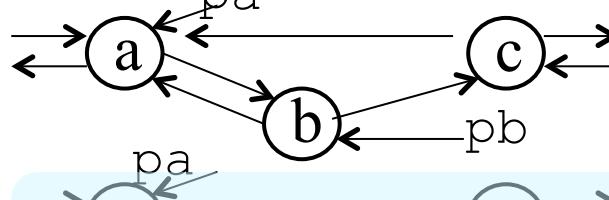
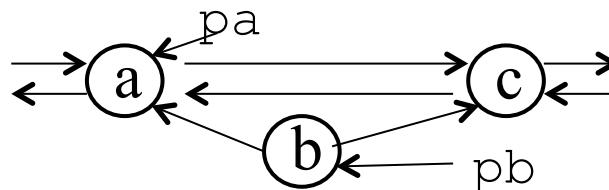
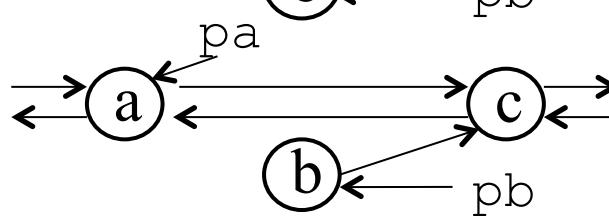
Como acción atómica

```

Nodo pa=.. //inserta tras pa
Nodo pb = new Nodo(b);
pb.sig = pa.sig;
pb.prev = pa;
pa.sig = pb;
(pb.sig).prev = pb;
    
```



Estado estable



Estados intermedios

Estado estable



## ¿Qué es el “Determinismo”?

**Determinismo:** ante una misma combinación de datos de entrada siempre (bajo mismas condiciones) genera una misma salida

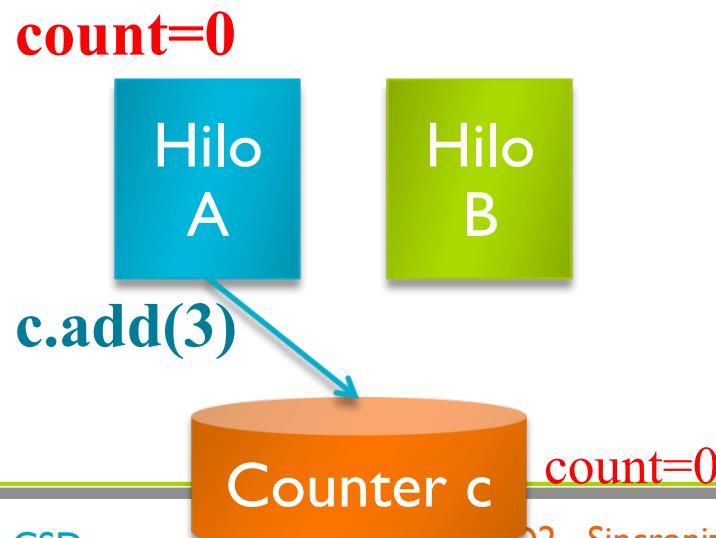
- ▶ **Programa secuencial** → generalmente determinista
- ▶ **Programa concurrente** → puede ser **no determinista**
  
- ▶ Ejecutar un conjunto de acciones atómicas de forma concurrente equivale a ejecutarlas en un orden arbitrario
  - ▶ Se pueden producir interferencias entre hilos
  - ▶ Se pueden producir errores de consistencia de memoria

## Determinismo → Veamos un ejemplo

### ► Ejemplo de intercalado:

*Hilo A ejecuta c.add(3)*

**A cargo c.count (copia local=0)**



```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
    public long getCount() {  
        return count;  
    }  
...  
Counter c= new Counter(); // inic a 0
```

## Determinismo → Veamos un ejemplo

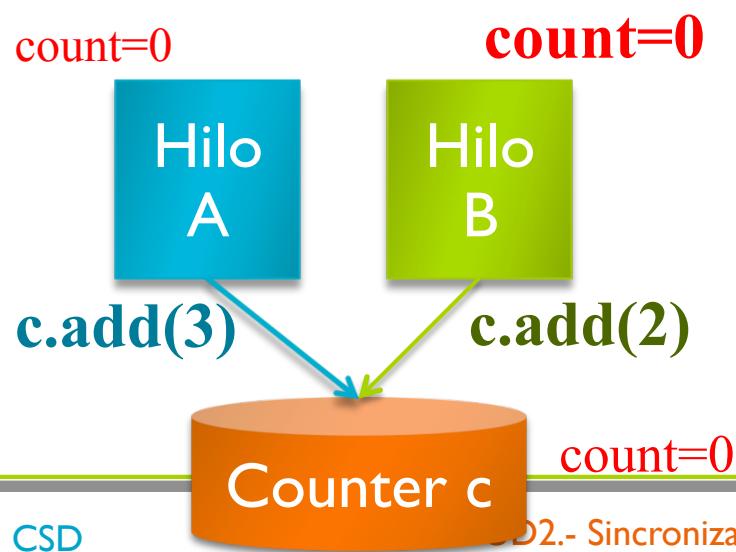
### ► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

A carga `c.count` (copia local=0)

Hilo B ejecuta `c.add(2)`

B carga `c.count` (copia local=0)



```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
  
    public long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```

## Determinismo → Veamos un ejemplo

### ► Ejemplo de intercalado:

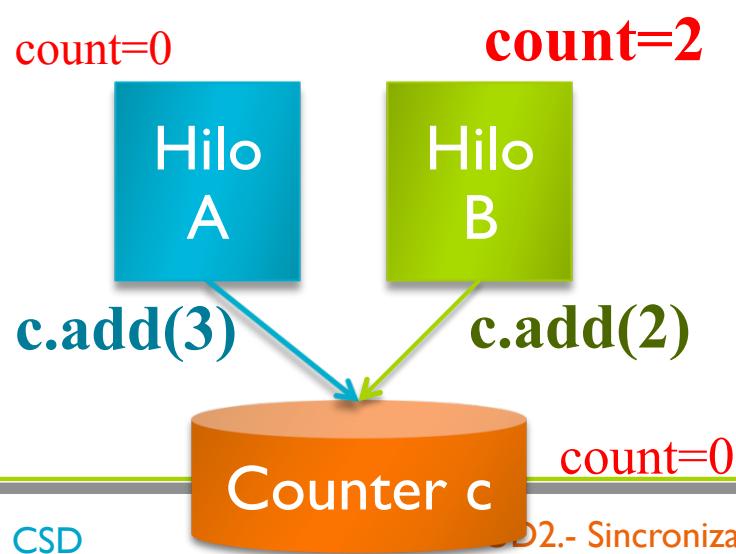
Hilo A ejecuta `c.add(3)`

A carga `c.count` (copia local=0)

Hilo B ejecuta `c.add(2)`

B carga `c.count` (copia local=0)

**B suma 2 (copia local=2)**



```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
  
    public long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```

## Determinismo → Veamos un ejemplo

### ► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

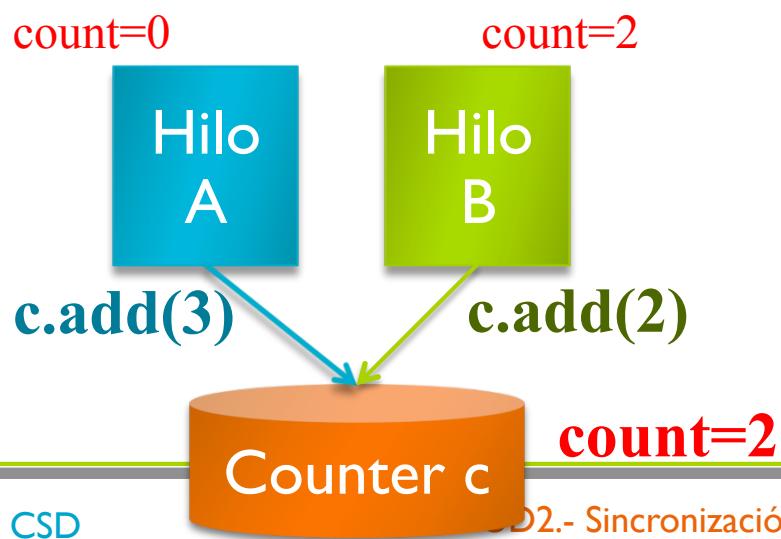
A carga `c.count` (copia local=0)

Hilo B ejecuta `c.add(2)`

B carga `c.count` (copia local=0)

B suma 2 (copia local=2)

**B vuelve al heap (`c.count=2`)**



```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
  
    public long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```

## Determinismo → Veamos un ejemplo

### ► Ejemplo de intercalado:

*Hilo A ejecuta c.add(3)*

A carga c.count (copia local=0)

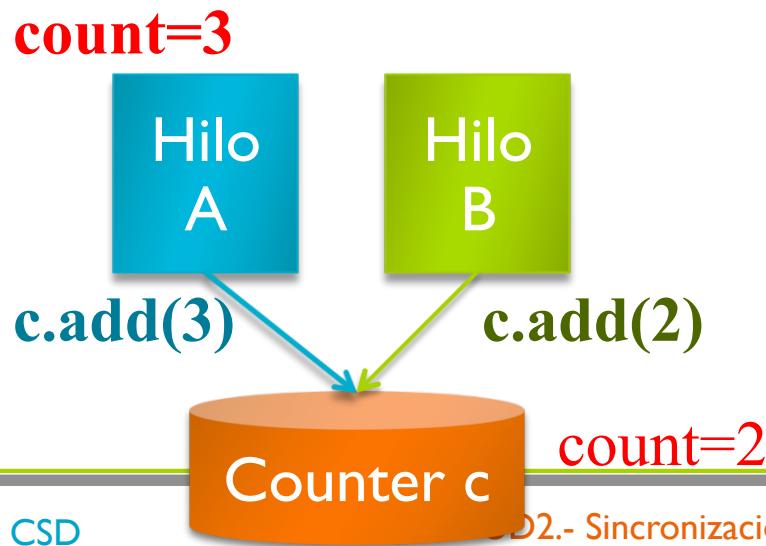
*Hilo B ejecuta c.add(2)*

B carga c.count (copia local=0)

B suma 2 (copia local=2)

B vuelca al heap (c.count=2)

**A suma 3 (copia local=3)**



```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
    public long getCount() {  
        return count;  
    }  
...  
Counter c= new Counter(); // inic a 0
```

## Determinismo → Veamos un ejemplo

### ► Ejemplo de intercalado:

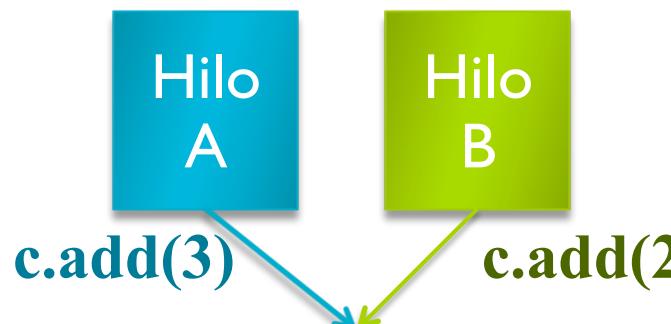
*Hilo A ejecuta c.add(3)*

A carga c.count (copia local=0)

A suma 3 (copia local=3)

**A vuelca al heap (c.count=3)**

count=3



*Hilo B ejecuta c.add(2)*

B carga c.count (copia local=0)

B suma 2 (copia local=2)

B vuelca al heap (c.count=2)

```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
  
    public long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```

## Determinismo → Veamos un ejemplo

### ► Ejemplo de intercalado:

*Hilo A ejecuta c.add(3)*

A carga c.count (copia local=0)

A suma 3 (copia local=3)

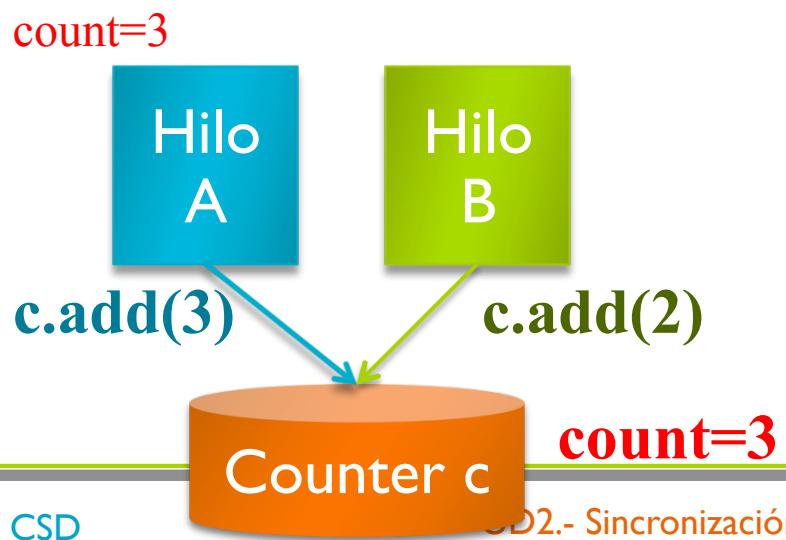
A vuelca al heap (c.count=3)

*Hilo B ejecuta c.add(2)*

B carga c.count (copia local=0)

B suma 2 (copia local=2)

B vuelca al heap (c.count=2)



Valor final: 3  
¡Esperábamos 5 !!!

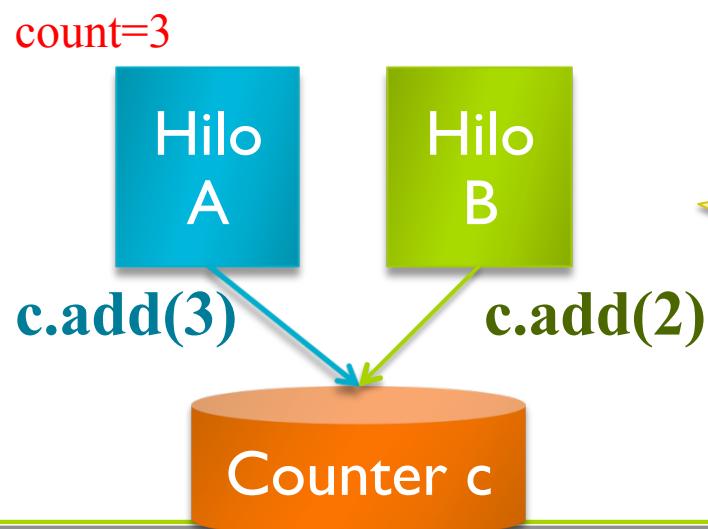
CÓDIGO NO  
DETERMINISTA



## Determinismo → ¿Cómo conseguirlo?

- ▶ El programa debería ser **CORRECTO** bajo cualquier intercalado posible
  - ▶ Que funcione sin errores algunas veces, no implica que sea correcto
- ▶ El número de intercalados posibles es enorme
  - ▶ Para  $(b; c; d; e; f; g) \parallel (h; i; j; k; l; m)$ ,  $12!/(6! * 6!) = 924$  intercalados posibles
  - ▶ No se pueden probar todos

**Condiciones de carrera:** si en varios intercalados en el tiempo el resultado es INCORRECTO



Ejemplo: se produce una **condición de carrera** en el acceso concurrente al objeto compartido Counter c



## Contenido

- ▶ **Sincronización de tareas**
  - ▶ Mecanismos de comunicación
    - ▶ Problemas de la comunicación con memoria compartida
  - ▶ Modelo de ejecución. Determinismo
  - ▶ Tipos de sincronización
    - ▶ Exclusión mutua
      - Sección Crítica
      - Locks
    - ▶ Sincronización condicional



## Objetivos de la sincronización

### **Objetivos de los Mecanismos de sincronización**

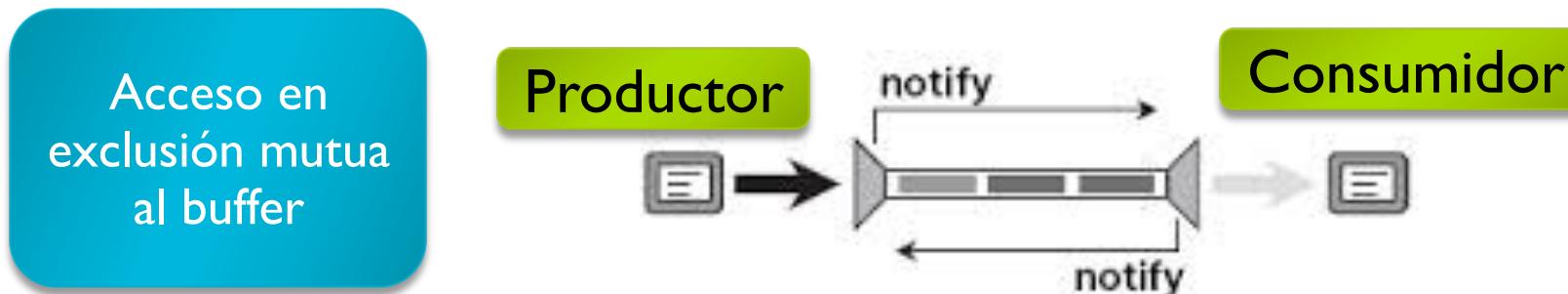
- garantizar orden de ejecución de sentencias
- respetar restricciones en la ejecución de código

### ► **Tipos de sincronización:**

- ▶ **Exclusión mutua**
- ▶ **Sincronización condicional**

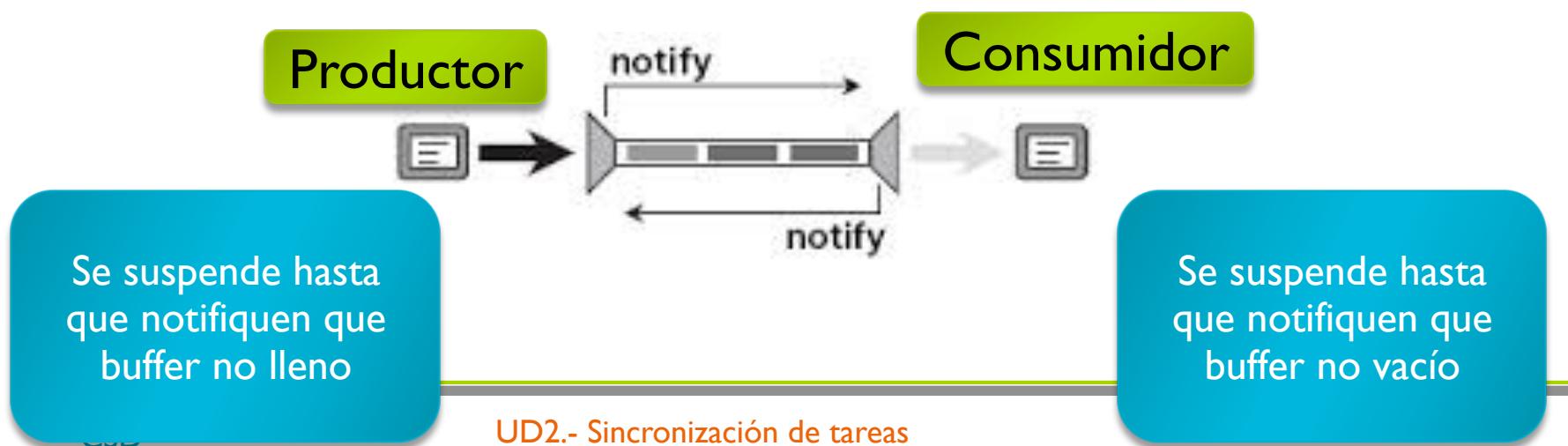
## ► Exclusión mutua

- SOLAMENTE UN HILO puede ejecutar la sección en cada momento
- Necesario para evitar interferencias entre hilos
- Aplicarlo para variables/objetos compartidos



## ► Sincronización condicional

- ▶ Un hilo debe suspenderse hasta que se cumpla determinada condición
- ▶ La condición depende del valor de alguna variable compartida
- ▶ Otros hilos, al modificar esas variables, conseguirán que se cumpla la condición, reactivando a los hilos suspendidos





## Sección Crítica → ¿Qué es?

**Sección crítica (SC):** fragmento de código que puede provocar condiciones de carrera

- ▶ Acceso a *variables locales* → no es crítico
- ▶ Acceso a *objetos compartidos inmutables (constantes)* → no crítico
- ▶ Acceso a **variables u objetos compartidos** → **crítico**



**Sección crítica (SC):** fragmento que accede a variables u objetos compartidos por más de un hilo



# Sección Crítica → ¿Cómo protegerla?

Resto de código  
....

*protocolo de entrada*

**Sección Crítica**

*protocolo de salida*

Resto de código  
(Sección no crítica)  
....

## Protocolos de entrada/salida

que garantizan (en común):

- **Exclusión mutua**
- **Progreso:** todo servicio solicitado se completa en algún momento
- **Espera limitada**

**Código thread-safe:** puede ser ejecutado concurrentemente por distintos hilos de forma segura



## Sección Crítica → ¿Cómo protegerla?

- ▶ Estrategias de solución según el nivel:

### Hardware

- Inhabilitación de interrupciones
- Sólo para codificar el núcleo del SO

### Sistema Operativo

- Llamadas al sistema para utilizar semáforos, mutex, eventos, condiciones

### Lenguaje de Programación

- Locks, condiciones, monitores, regiones críticas, objetos protegidos
- Ejemplos: Pascal Concurrente, Modula-2, Modula-3, Ada, Java, C#



## Locks → ¿Qué es? ¿Cómo utilizarlos?

- ▶ Un **lock** es un objeto con dos estados (*abierto/cerrado*) y dos operaciones (*abrir/cerrar*).
- ▶ Al crear el lock, se encontrará inicialmente abierto.
- ▶ Uso del lock:

cerrar lock

SC

abrir lock

Sección no crítica



# Locks → ¿Cómo utilizarlos?

## ► Cerrar lock:

- ▶ Si abierto → lo cierra
- ▶ Si cerrado por otro hilo → se suspende
- ▶ Si cerrado por mismo hilo → no tiene efecto

## ► Abrir lock:

- ▶ Si abierto → no tiene efecto
- ▶ Si cerrado por otro hilo → no tiene efecto
- ▶ Si cerrado por mismo hilo → se abre el lock
  - ▶ Si alguien espera, quedará cerrado por uno de los que esperan
    - La elección depende de la implementación, aunque normalmente cumplirá equidad





## Locks → ¿Qué es? ¿Cómo utilizarlos?

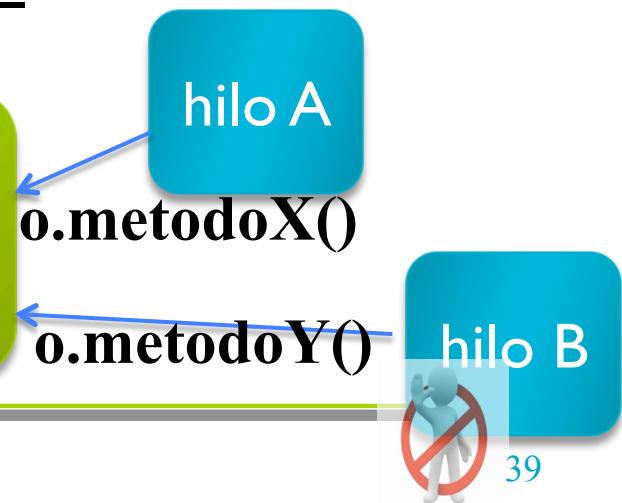
- ▶ Usando *locks* convertimos una sección crítica en una **acción atómica**
  - ▶ **Atomicidad:** únicamente un hilo puede ejecutar el código protegido en un momento dado
  - ▶ Garantiza actualización fiable de variables u objetos compartidos
    - Libre de condiciones de carrera o corrupción de estado
    - Sólo son visibles los estados consistentes
    - Asegura que todo hilo accede siempre al valor más reciente de cada variable u objeto compartido.



# Locks en Java → ¿Cómo implementarlos?

- ▶ Todo objeto posee un lock asociado **implícito**
  - ▶ Etiquetar con **synchronized** los métodos que forman parte de la sección crítica
  - ▶ Al entrar en un método etiquetado como '**synchronized**' se cierra el lock, y al salir se abre.
    - ▶ Al salir, se establece una relación “ocurre-antes” con otras invocaciones posteriores de métodos sincronizados del mismo objeto.
  - ▶ Para un mismo objeto, todos sus métodos **synchronized** se ejecutan **en exclusión mutua entre sí**

```
public synchronized void metodoX(){...}  
public synchronized void metodoY(){...}
```





## Locks en Java → ¿Cómo implementarlos?

---

- ▶ **Implementación** → añadir etiqueta ***synchronized*** a todo método que:
  - ▶ Modifique un atributo que luego debe leer otro hilo
  - ▶ Lea un atributo actualizado por otro hilo



# Locks en Java → ¿Cómo implementarlos?

```
public class Counter {  
    private long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
  
    public synchronized long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```

Ejemplo: Versión determinista del contador



## Locks en Java → Veamos un ejemplo

- ▶ Ejemplo de intercalado:

Inicialmente: lock implícito de  
**Counter** está abierto



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```

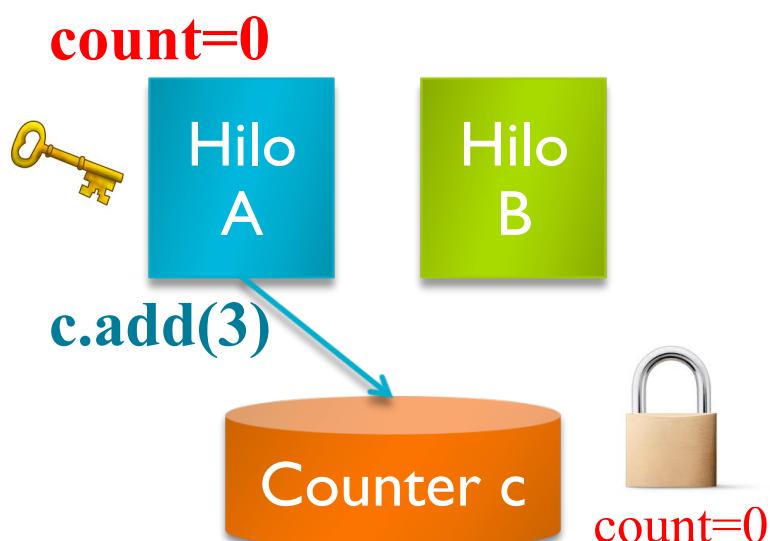
## Locks en Java → Veamos un ejemplo

### ► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

A cargo `c.count` (copia local=0)

Y se cierra  
**lock** implícito



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
    ...  
    Counter c = new Counter(); // inic a 0
```

# Locks en Java → Veamos un ejemplo

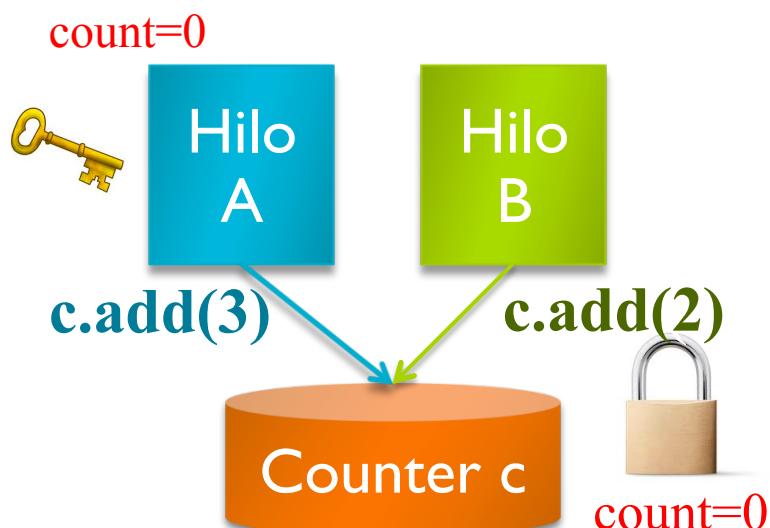
## ► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

A carga `c.count` (copia local=0)

Hilo B ejecuta `c.add(2)`

B queda bloqueado (lock está cerrado, por otro hilo)



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
  
    public synchronized long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```



## Locks en Java → Veamos un ejemplo

### ► Ejemplo de intercalado:

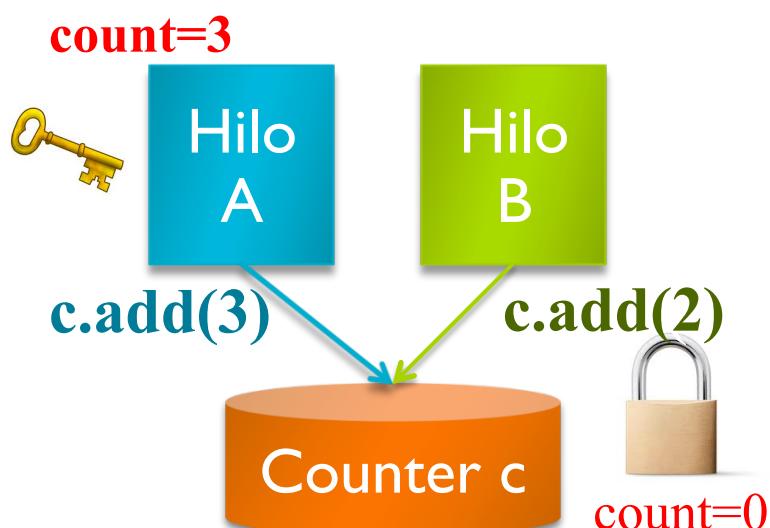
Hilo A ejecuta `c.add(3)`

A carga `c.count` (copia local=0)

**A suma 3 (copia local=3)**

Hilo B ejecuta `c.add(2)`

**B queda bloqueado (lock está cerrado, por otro hilo)**



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
    ...  
    Counter c = new Counter(); // inic a 0
```



## Locks en Java → Veamos un ejemplo

### ► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

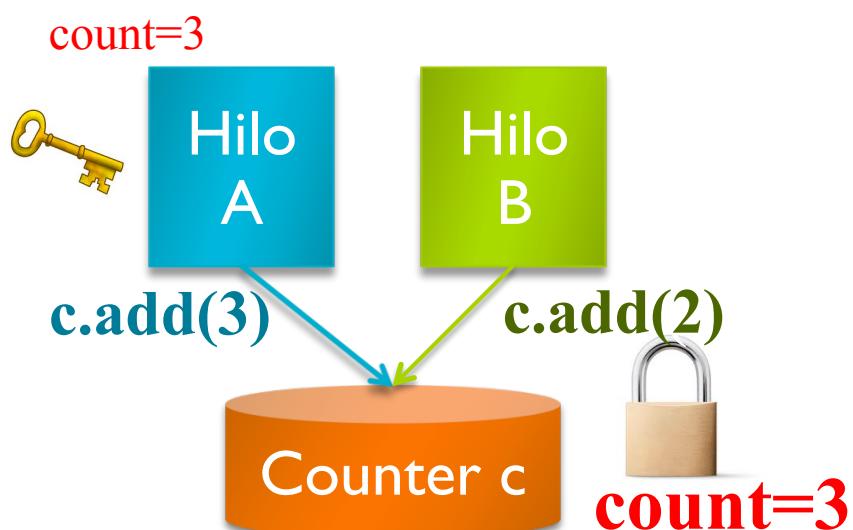
A carga `c.count` (copia local=0)

A suma 3 (copia local=3)

**A vuela al heap (`c.count=3`)**

Hilo B ejecuta `c.add(2)`

**B queda bloqueado (lock está cerrado, por otro hilo)**



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
  
    public synchronized long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```

# Locks en Java → Veamos un ejemplo

## ► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

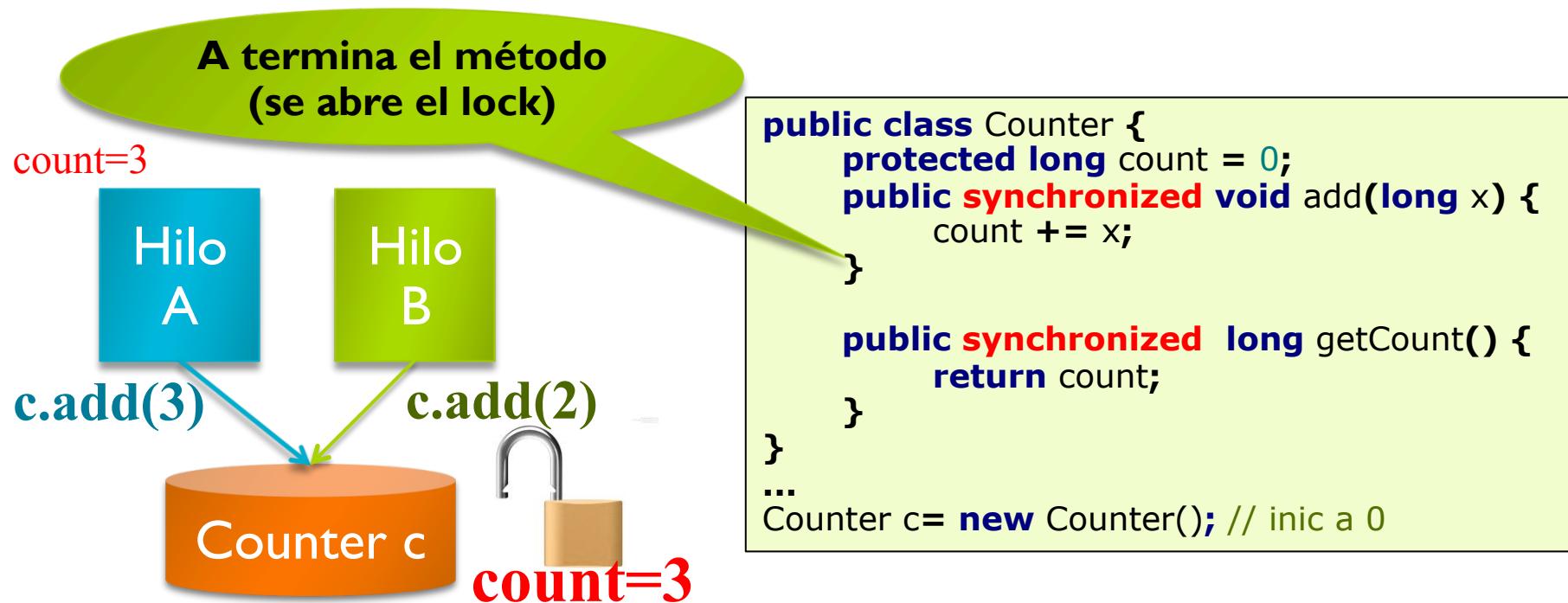
A carga `c.count` (copia local=0)

A suma 3 (copia local=3)

A vuelca al heap (`c.count=3`)

Hilo B ejecuta `c.add(2)`

B queda bloqueado (lock está cerrado, por otro hilo)





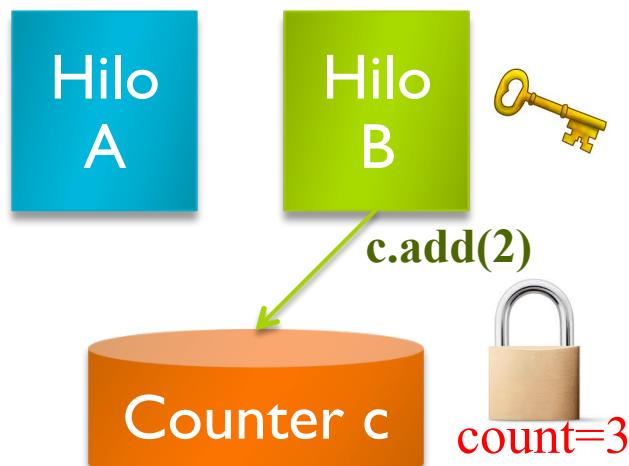
# Locks en Java → Veamos un ejemplo

- ▶ Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

Hilo B ejecuta `c.add(2)`

**B se reactiva y cierra el lock**



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
  
    public synchronized long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```



# Locks en Java → Veamos un ejemplo

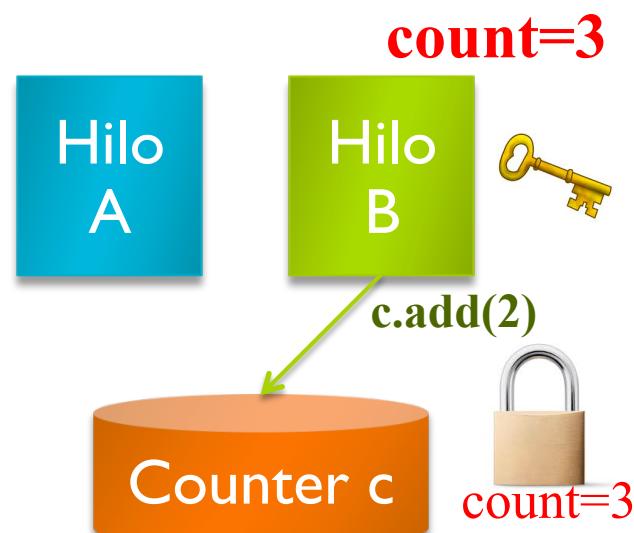
- ▶ Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

Hilo B ejecuta `c.add(2)`

B se reactiva y cierra el lock

**B carga c.count (copia local=3)**



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
  
    public synchronized long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```

# Locks en Java → Veamos un ejemplo

- ▶ Ejemplo de intercalado:

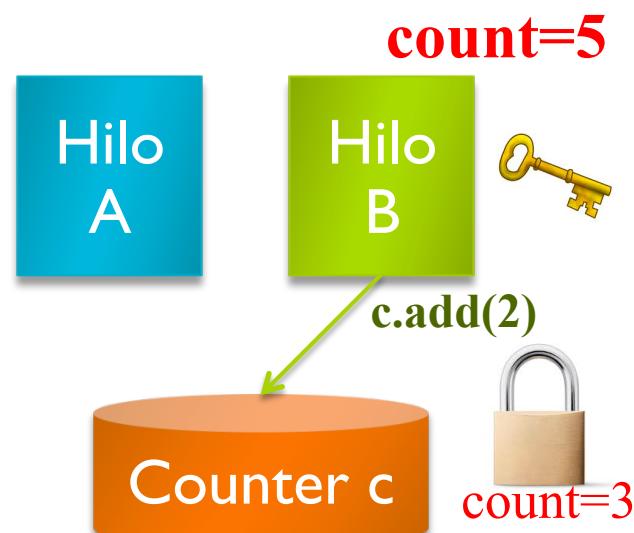
Hilo A ejecuta `c.add(3)`

Hilo B ejecuta `c.add(2)`

B se reactiva y cierra el lock

B carga `c.count` (copia local=3)

**B suma 2 (copia local=5)**



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
  
    public synchronized long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```

# Locks en Java → Veamos un ejemplo

- ▶ Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

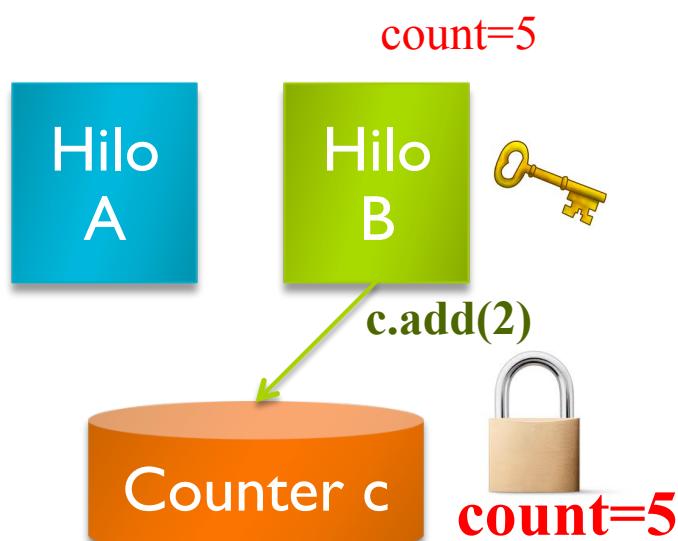
Hilo B ejecuta `c.add(2)`

B se reactiva y cierra el lock

B carga `c.count` (copia local=3)

B suma 2 (copia local=5)

**B vuelca al heap (`c.count=5`)**



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
  
    public synchronized long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```

# Locks en Java → Veamos un ejemplo

- ▶ Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`



Hilo B ejecuta `c.add(2)`

B se reactiva y cierra el lock  
B carga `c.count` (copia local=3)  
B suma 2 (copia local=5)  
B vuelca al heap (`c.count=5`)

```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
  
    public synchronized long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```



# Locks en Java → Veamos un ejemplo

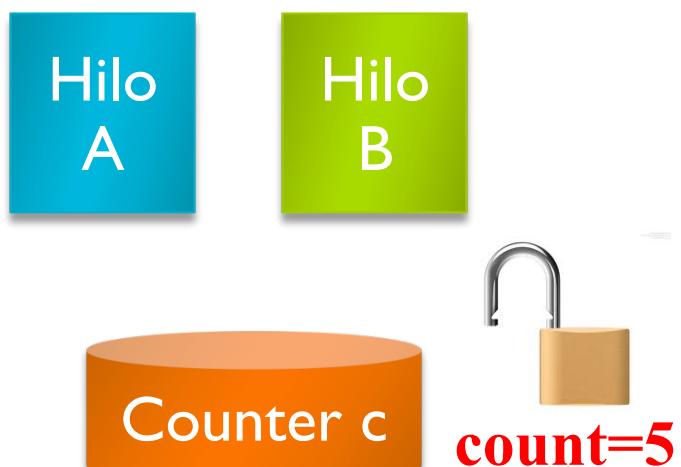
- ▶ Independientemente del intercalado:

Hilo A ejecuta `c.add(3)`

Hilo B ejecuta `c.add(2)`

Valor final: SIEMPRE 5!!

CÓDIGO  
DETERMINISTA



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```



## Sincronización en Java

- ▶ Java proporciona dos expresiones básicas de sincronización:
  - ▶ **Métodos sincronizados:** utilizando la etiqueta **synchronized** en la declaración del método
  - ▶ **Sentencias sincronizadas (*Synchronized Statements*):** indicando el objeto que proporciona el lock implícito
    - ▶ Permite utilizar más de un lock dentro de un mismo método.

- Los métodos inc1 e inc2 pueden intercalarse
- Pero la actualización de c1 es en exclusión mutua
- Idem para c2

```
public class MsLunch{  
    private long c1 = 0;  
    private long c2 = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void inc1() {  
        synchronized(lock1){ c1++; }  
    }  
  
    public void inc2() {  
        synchronized(lock2){ c2++; }  
    }  
}
```



## Sincronización Condicional

---

- ▶ El concepto de sección crítica evita interferencias en el acceso a las variables compartidas
- ▶ Pero también debemos resolver el problema de la *sincronización condicional*
  - ▶ Hacer esperar a un hilo hasta que se cumpla una determinada condición
- ▶ En el tema siguiente estudiamos el concepto de *monitor*
  - ▶ Resuelve ambos problemas (exclusión mutua y sincronización condicional) mediante una única construcción



## Resultados de aprendizaje de la Unidad Didáctica

---

- ▶ Al finalizar esta unidad, el alumno deberá ser capaz de:
  - ▶ Identificar las secciones de una aplicación que deban o puedan ser ejecutadas concurrentemente por diferentes actividades.
  - ▶ Identificar los problemas de la comunicación con memoria compartida. Describir el problema de determinismo.
  - ▶ Identificar los tipos de sincronización.
  - ▶ Distinguir las secciones críticas de una aplicación y protegerlas mediante los mecanismos de sincronización adecuados.