Programación Concurrente en Java

Programación Concurrente – Tema 5

Miguel Ángel Rodríguez García

Carlos Grima

Lucía Serrano Luján

Universidad Rey Juan Carlos

Curso 2017 - 2018

Concurrencia y orientación a objetos

Programación Concurrente en Java - Tema 5.4

Datos compartidos entre hilos

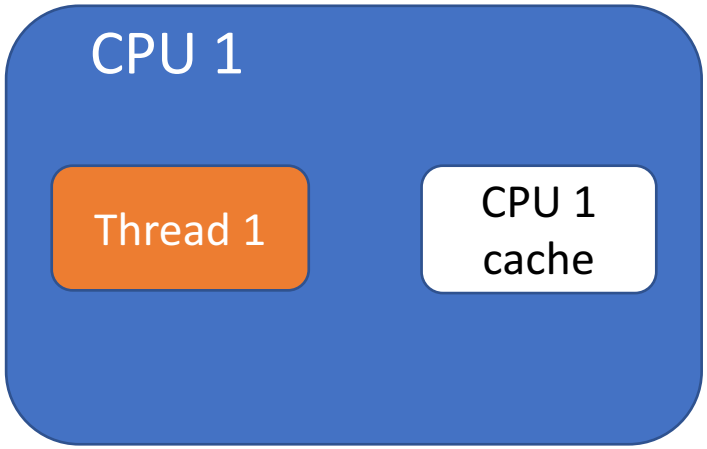
Objetos compartidos entre hilos

Datos compartidos entre hilos

Objetos compartidos entre hilos

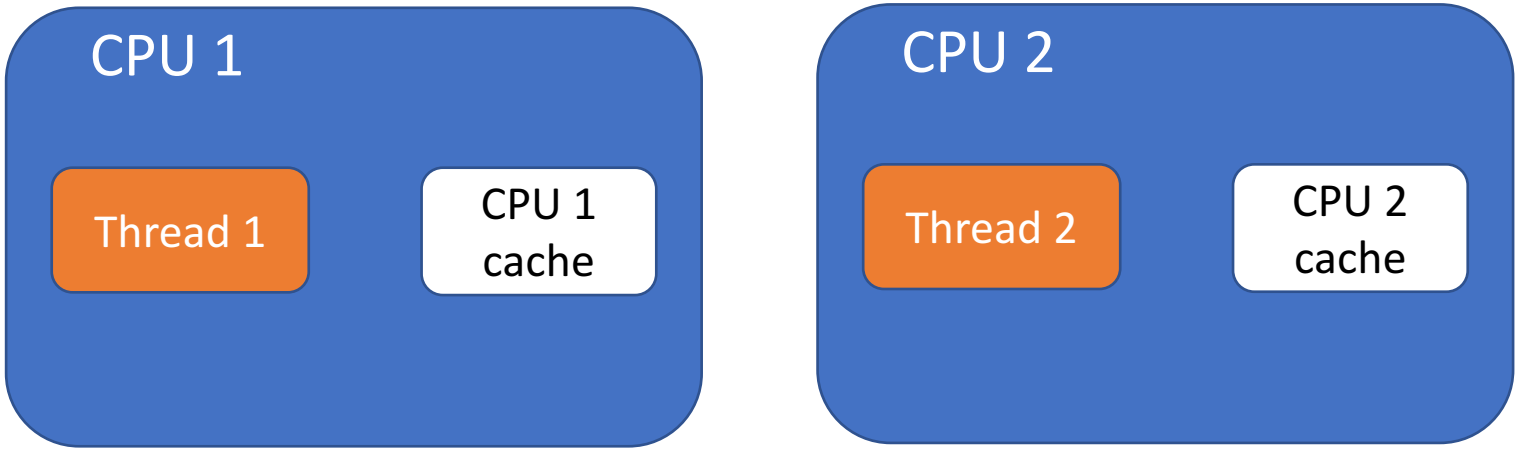
Datos compartidos entre hilos

CUANDO CREAMOS UN HILO….



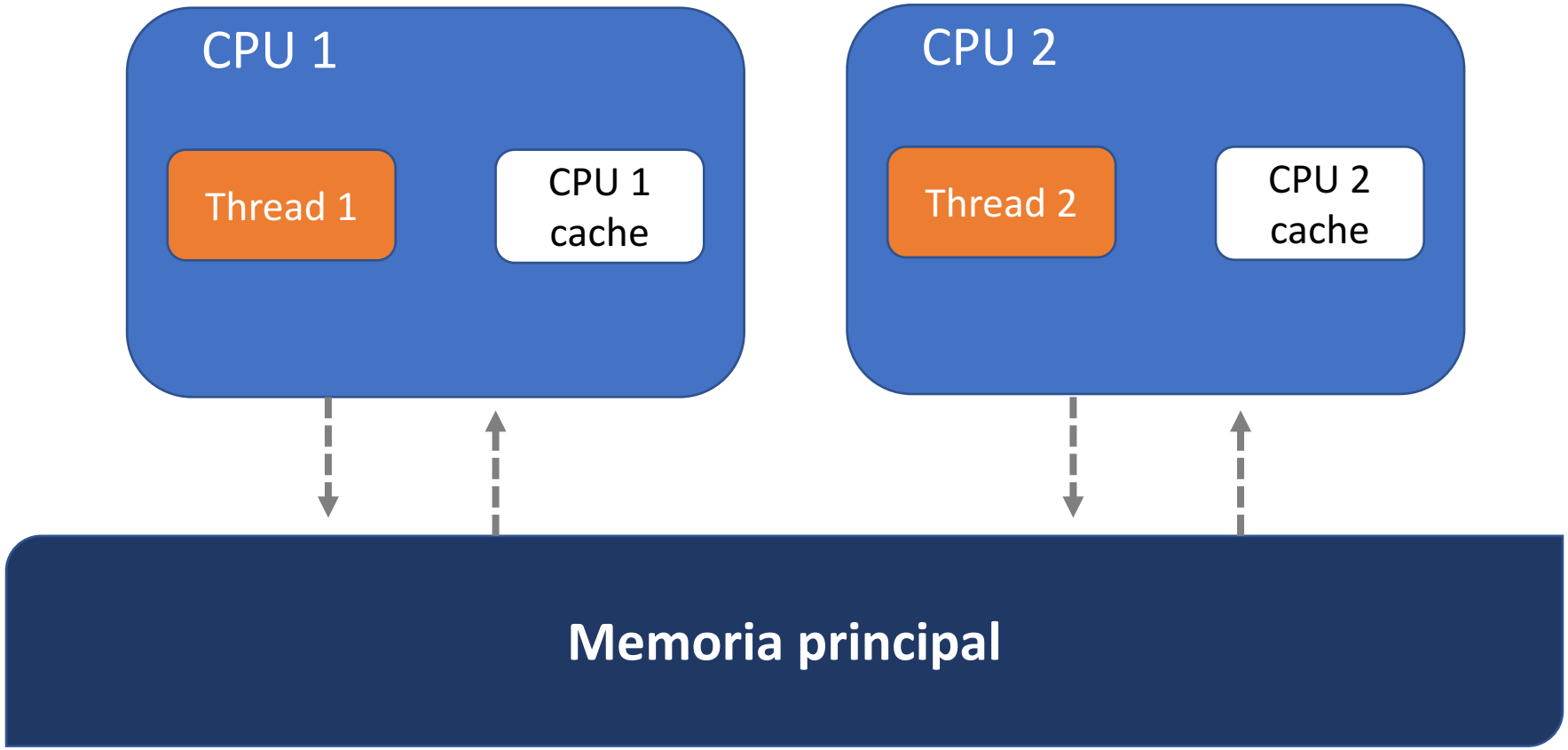
Datos compartidos entre hilos

CUANDO CREAMOS MÁS HILOS….



Datos compartidos entre hilos

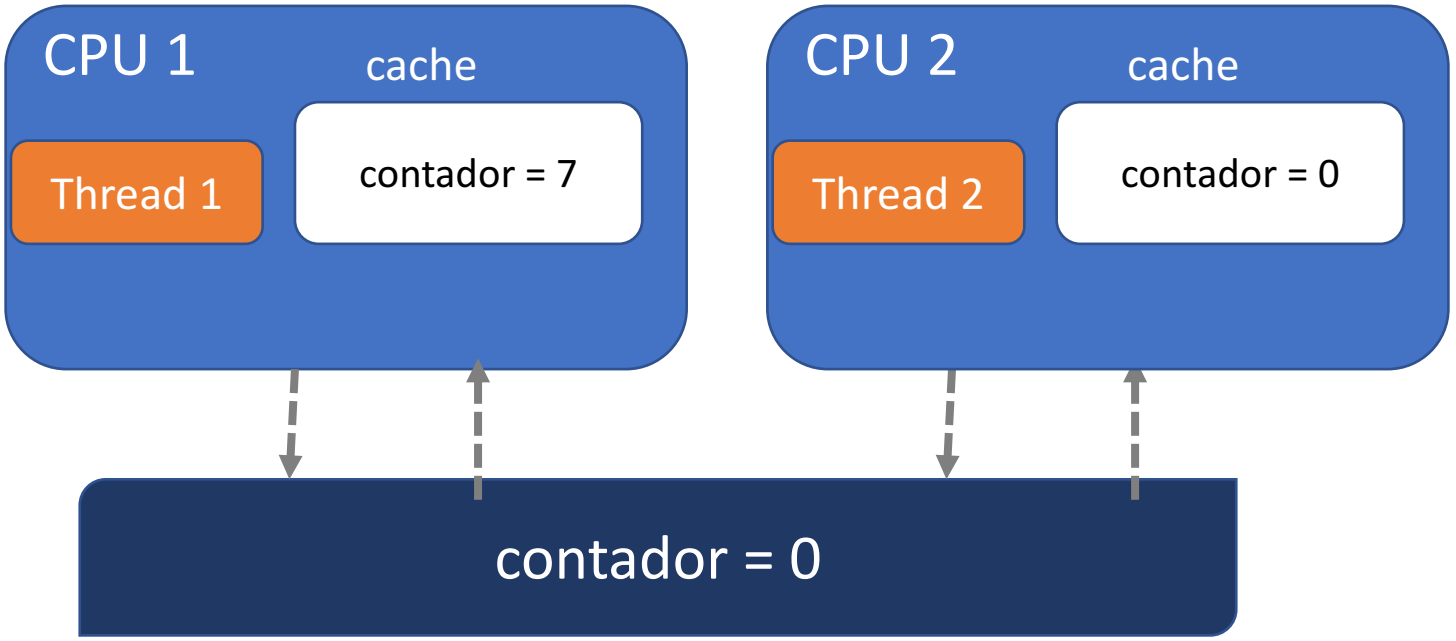
CUANDO CREAMOS MÁS HILOS….



Datos compartidos entre hilos

- Dos o más hilos tienen acceso a una variable:

```
public class objetoCompartido{
    public int contador = 0;
}
```



Datos compartidos entre hilos

- Los procesadores utilizan memoria **cache** y **registros** para almacenar los valores de las **variables**
- En sistemas de varios procesadores (o núcleos), es habitual que exista una **cache por cada procesador**
- ¿Qué ocurre con las **variables compartidas** si dos procesos se ejecutan cada uno en un procesador?
- ¿Cuándo se **sincronizan** las cachés?
- ¿Cuándo son **visibles** los cambios de las variables por todos los hilos?

Datos compartidos entre hilos

- Para un compilador **es imposible** determinar qué atributos se comparten entre hilos y cuales no
- **No es eficiente** sincronizar los valores de todos los atributos cada vez que se hace una escritura en ellos por si otro procesador los usa
- Existe un problema de **visibilidad**
- ¿Cómo se resuelve el problema?

Datos compartidos entre hilos

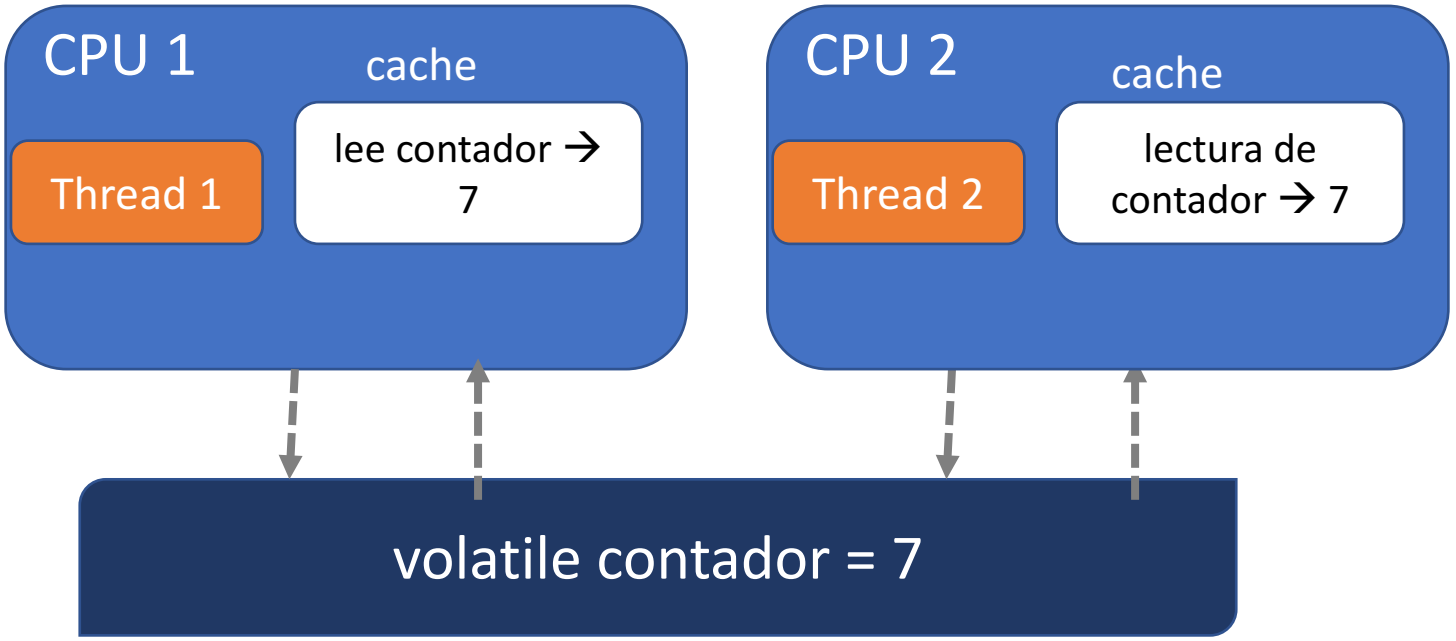
- El desarrollador tiene que **marcar explícitamente** aquellos atributos **compartidos** entre hilos que pueden **cambiar de valor**
- Existen dos formas de hacerlo:
  - Anadir el modificador ***volatile***
  - Sincronizar los hilos usando alguna **herramienta de sincronización** (**semáforos**, etc...) desde que se escribe el valor hasta que se lee el valor (**exclusión mutua**, **sincronización condicional**...)

Datos compartidos entre hilos

- Dos o más hilos tienen acceso a una variable:

```
public class objetoCompartido{
    public volatile int contador = 0;
}
```

Garantiza la visibilidad



Datos compartidos entre hilos

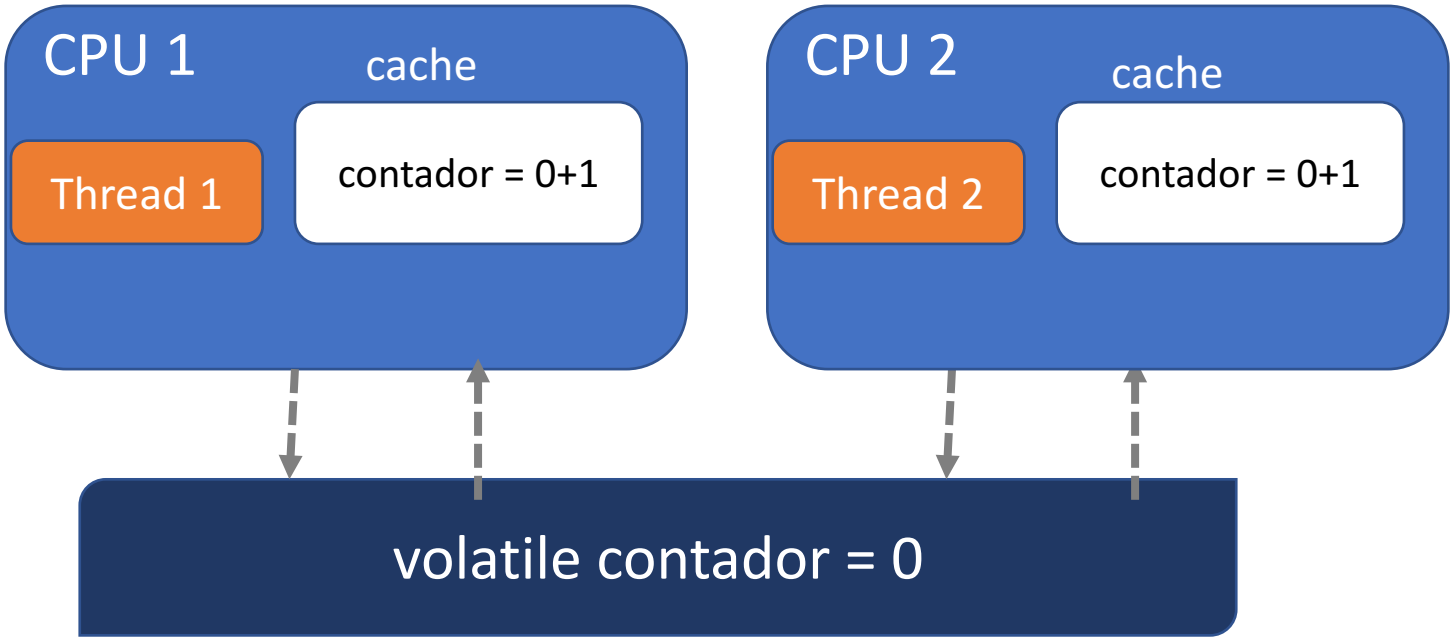
- El sistema se asegura de que un hilo **lea el último valor** de un atributo compartido si...
  - ... el atributo está marcado como **volatile** o...
  - ... el hilo ha usado alguna herramienta de sincronización (**semáforo**...) antes de leer el atributo

Datos compartidos entre hilos

- Dos o más hilos tienen acceso a una variable:

```
public class objetoCompartido{
    public volatile int contador = 0;
}
```

¡¡¡Excepción!!! cuando dos hilos leen y escribe sobre la variable volatile



Datos compartidos entre hilos

- ¿Qué ocurre si un atributo es compartido entre varios hilos y no se ha marcado como **volatile** y no se usa ninguna **herramienta de sincronización** entre los procesos?
  - Es posible que un hilo lea un **valor antiguo** (el que estaba en la caché de ese proceso) y no el ultimo valor del atributo
  - Es posible que el compilador o la JVM **elimine la variable** si puede optimizar el código considerando que sólo un hilo accede a ella

Datos compartidos entre hilos

- **¿Siempre hay que usar *volatile* para variables compartidas entre hilos?**
- **No, no es necesario** utilizar ***volatile*** si el hilo que escribe la variable se **sincroniza** con el hilo que lee la variable (usando herramientas de sincronización)
- Por ejemplo:
  - Si los atributos **compartidos** están bajo **exclusión mutua** con un semáforo, se garantiza que se leerán los valores correctos (sin usar ***volatile***)
  - Si un proceso escribe un atributo y desbloquea a otro proceso, el otro proceso leerá el valor escrito (sin usar ***volatile***)

Datos compartidos entre hilos

- Esta forma de hacer que dos hilos puedan acceder a la información de forma coherente está definida en el **Java Memory Model**

  http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html

- Siempre hay que usar ***volatile*** o **sincronizar** los hilos para que no se produzcan problemas de **datos obsoletos** o **eliminado de variables**

Datos compartidos entre hilos

Objetos compartidos entre hilos

Objetos compartidos entre hilos

- Varios hilos pueden **compartir** valores de **tipos primitivos** y **arrays**
- Los hilos también pueden **compartir** objetos de **clases** de la librería estándar o desarrolladas por nosotros
- **Por ejemplo:** Los objetos **SimpleSemaphore** se comparten entre hilos

Objetos compartidos entre hilos

- Un objeto **se puede compartir** entre varios hilos sin problemas en alguno de los siguientes casos:
  - Se accede a los métodos del objeto bajo **exclusión mutua** (para evitar que varios hilos ejecuten métodos de forma **concurrente** y produzcan **intercalaciones** no deseadas)
  - La clase de ese objeto está **preparada** para la ejecución **concurrente** de sus métodos.
  - Los objetos **no se modifican** durante el tiempo que se comparten entre los hilos
  - La clase es **inmutable** (sus objetos no cambian **nunca** de estado)

Objetos compartidos entre hilos

Clases *thread-safe*

- Una clase es ***thread-safe*** si los objetos de la clase se pueden compartir entre hilos sin necesidad de ponerlos bajos exclusión mutua
- Todas las **clases inmutables** (cuyos objetos no cambian de estado) son ***thread-safe*** (p.e: String)
- En la librería existen muchas clases ***thread-safe*** (hay que revisar la documentación)

Objetos compartidos entre hilos

Entonces... ¿Todas las clases que programamos tienen que ser *thread-safe*?

- Implementar una clase ***thread-safe*** es más **difícil** que una clase que no lo es
- Si el objeto finalmente **no se comparte** entre varios hilos, los mecanismos que hemos implementado para hacerla *thread-safe* posiblemente la hagan **menos eficiente**
- Lo ideal es hacer que la clase sea **inmutable**, porque eso la hace *thread-safe* y no tiene ninguna penalización en el rendimiento, pero muchas veces no es posible

Objetos compartidos entre hilos

¿Como sabemos si una clase de la librería es *thread-safe* o no?

- Si es **inmutable**, es ***thread-safe***
- En otro caso... hay que revisar la **documentación (JavaDoc)** o consultar tutoriales o **manuales** oficiales
- En algunos casos existen **dos versiones** de la misma clase, una para **compartir** entre hilos y la otra para usarse en un **único hilo (más eficiente)**

Objetos compartidos entre hilos

Ejemplo de **dos versiones** de la misma clase:

- **StringBuilder:** A **mutable** sequence of characters. This class provides an API compatible with **StringBuffer**, but with no guarantee of **synchronization**. This class is designed for use as a drop-in **replacement** for StringBuffer in places where the string buffer was being used by **a single thread** (as is generally the case). Where possible, it is recommended that this class be used in preference to StringBuffer as it will be **faster** under most implementations.
- **StringBuffer:** A **thread-safe**, mutable sequence of characters. A string buffer is like a String, but can be modified.

Objetos compartidos entre hilos

¿Mejor usar una clase *thread-safe* o poner bajo exclusión mutua objetos de clase no *thread safe*?

- Siempre que exista una clase ***thread-safe*** **es mejor** usarla
- Poner un objeto bajo **exclusión mutua** puede **reducir** el **paralelismo** innecesariamente
- Los implementadores de una clase ***thread-safe*** intentan que sea lo más **paralela** posible, es decir, reducen al mínimo las zonas que no pueden ejecutarse de forma concurrente

Objetos compartidos entre hilos

¿Cómo se implementa una clase *thread-safe*?

- Basta con asegurarse de que sus **métodos** se pueden **ejecutar** por varios hilos de forma **concurrente** sin condiciones de carrera ni intercalaciones no deseadas
- Cómo
  - Haciéndola **inmutable** o…
  - **Sincronizando los hilos** de alguna forma... (exclusión mutua, sincronización...) o...
  - Con **atributos *thread-safe...***

Objetos compartidos entre hilos
EJERCICIO

Crear una clase *thread-safe* llamada **SincCond** que implemente una sincronización condicional entre dos hilos

- Métodos:
  - **await:** El hilo que ejecute ese método no podrá continuar hasta que otro hilo ejecute *signal*
  - **signal:** Cuando se ejecute este método, el hilo que ejecuta *await* podrá continuar
- La clase tendrá los atributos necesarios para implementar estos métodos con **espera activa**

Objetos compartidos entre hilos
EJERCICIO 2

- Refactorizar el programa ProdCons para usar la clase SincCond

```
public class ProdCons {

   static volatile boolean producido = false;
   static volatile double producto;

   public static void productor() {
      producto = Math.random();
      producido = true;
   }

   public static void consumidor() {
      while (!producido);
      print("Producto: "+producto);
   }

   public void exec() {
      new Thread(()-> productor()).start();
      new Thread(()-> consumidor()).start();
   }

   public static void main(String[] args){
       new ProdCons().exec();
   }
}
```

Objetos compartidos entre hilos
EJERCICIO 2

- Mejorar la clase **SincCond** para que permita la creación de productos indefinidos
- Actualizar la clase **ProdCons** para que se produzcan y consuman 10 productos