UNIT 2: THREADS PROGRAMMING

Programación de Servicios y Procesos

INTRODUCCIÓN

En muchas ocasiones las **sentencias** que forman un **programa** deben ejecutarse de manera **secuencial**, puesto que forman una lista ordenada de pasos a seguir para resolver un problema.

Otras veces, dichos pasos **no tienen** por qué ser **secuenciales**, pudiéndose realizar varios a la vez.

Hay casos en los que la simultaneidad en la ejecución no es opcional, sino **necesario**, como ocurre en las peticiones hacia las **aplicaciones web**.

```
public static void main(String[] args) {
    Scanner sc = new Scanner( in:System.in);
    int edad;
    System.out.println( x:"Introduzca un número");
    edad = sc.nextInt();
    System.out.println("Tienes: " + edad + " años");

}
```

En estos casos la solución se encuentra en la **programación multihilo**, para conseguir procesamiento simultáneo.

Aunque nos ofrece multitud de posibilidades, esta técnica no está exenta de **condiciones y restricciones.**

FUNDAMENTOS DE LA PROGRAMACIÓN MULTIHILO

Desde el punto de vista del **sistema operativo**, un *programa informático en ejecución* es un **proceso** que **compite** con los demás procesos por acceder a los **recursos** del sistema.

Desde **dentro**, un programa es una **sucesión de sentencias** que se ejecutan **una detrás de otra**.

El **sistema operativo** se encarga de **hacer convivir** los diferentes **procesos** y de **repartir los recursos** entre sí → Cuando se programa no hay que tener en cuenta aspectos relacionados en cómo se van a gestionar los procesos o cómo estos van a tener acceso a los recursos. Es el **sistema operativo** el que se encarga de la **concurrencia** a nivel de proceso.

Cuando programamos sí debemos → Optimizar el uso de memoria y conseguir que los algoritmos sean eficientes.

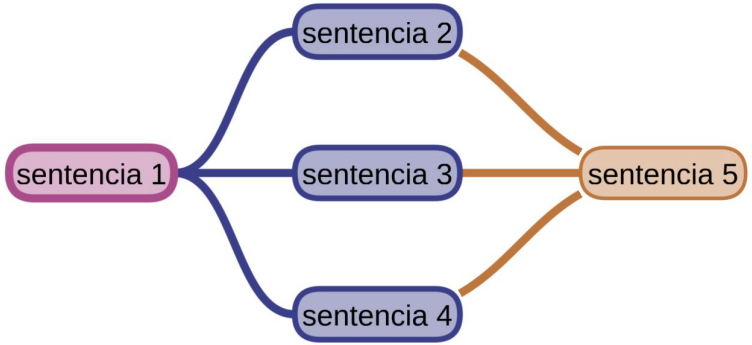
Es dentro del **interior de los procesos** donde la programación tiene algo que decir con respecto a la **concurrencia**, mediante la **programación multihilo**.

FUNDAMENTOS DE LA PROGRAMACIÓN MULTIHILO (II)

En el caso de un programa absolutamente **secuencial**, el hilo de ejecución es único, por lo que **cada sentencia** tiene que **esperar** a que termine la **anterior**.

Esto no significa que no puedan existir estructuras de control, como if o while, sino que las sentencias se ejecutan una detrás de otra.

En el caso de un **programa multihilo**, algunas **sentencias** se ejecutan **simultáneamente**, ya que los hilos creados y activos en un momento dado acceden a los recursos de procesamiento sin necesidad de esperar a que otras partes del programa terminen.



CARACTERÍSTICAS DE LOS HILOS

- **Dependencia del proceso:** No se pueden ejecutar independientemente, siempre se ejecutan **dentro** del **contexto** de un **proceso**.
- **Ligereza**: Al ejecutarse dentro del contexto de un proceso, **no** requiere **generar procesos nuevos**. Se pueden generar gran cantidad de hilos **sin** que provoquen **pérdidas de memoria**.
- **Compartición de recursos**: Dentro del mismo proceso, los **hilos comparten** espacio de **memoria**. Se pueden producir **colisiones** en los accesos a las variables provocando errores de concurrencia.
- **Paralelismo**: Aprovechan los **núcleos del procesador** generando un paralelismo real, siempre dentro de las capacidades del procesador.
- **Concurrencia**: Permiten **atender** de manera concurrente **múltiples peticiones**. Esto es especialmente importante en servidores web y de bases de datos.

FUNCIONAMIENTO DE LOS HILOS

La *ejecución de un proceso comienza* con **un único hilo**, pero se pueden **crear más** sobre la marcha.

Los distintos hilos de un mismo proceso **comparten**:

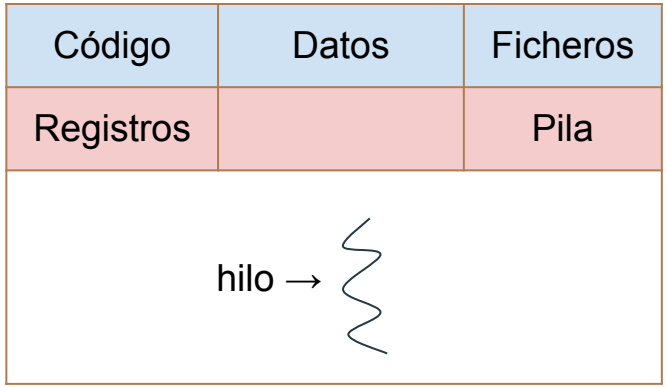
- El *espacio de memoria* asignado al proceso.
- La información de acceso a *ficheros*. No sólo se utilizan para almacenar datos, sino también para controlar dispositivos de entrada/salida (E/S).

Cada hilo tiene sus **propios valores** para:

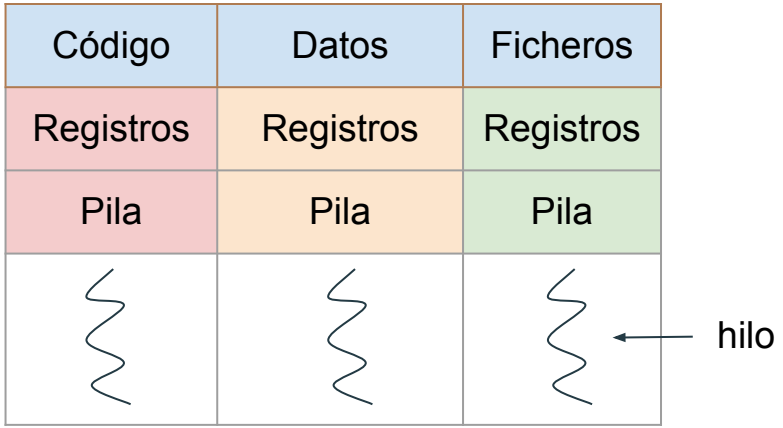
- Los **registros** del procesador.
- El estado de su **pila** (*stack*). En la pila se guarda información acerca de las llamadas en curso de ejecución a métodos de diversos objetos.

FUNCIONAMIENTO DE LOS HILOS (II)

A continuación se muestran esquemas de cómo sería un proceso con un sólo hilo y un proceso con varios hilos:



Proceso con un solo hilo



Proceso multihilo

ANALOGÍA

El **camarero** de una cafetería recibe a un **cliente** que le pide un **café**, una **tostada** y una **tortilla francesa**. Si el camarero trabaja como un proceso de **un único hilo** pondrá la **cafetera** a preparar el café, esperará **a que termine** para poner el **pan** a tostar y no pedirá la **tortilla** hasta que el **pan no esté tostado**. Probablemente, cuando todo esté dispuesto para **servir**, el **café** y la **tostada** estarán **fríos** y el **cliente aburrido** de esperar.

Los **camareros** (la gran mayoría de ellos) suelen trabajar como **procesos multihilo**: ponen la **cafetera** a preparar el café, el **pan** a tostar y piden a la cocina los **platos sin esperar** a que cada una de las **demás** tareas esté **terminada**. Dado que cada una de ellas consume **diferentes recursos**, no es necesario hacerlo. Transcurrido el tiempo que tarda en estar lista la tarea más lenta, el cliente estará atendido.

Al igual que pasa en los ordenadores, los **recursos** de una cafetería son **limitados**, por lo que hay una **restricción física** que impide que se hagan **más tareas simultáneamente** de las que permiten los recursos. Si solo hay una tostadora con capacidad para dos rebanadas de pan no se podrán tostar más de dicho número en un mismo instante de tiempo.

CREACIÓN DE HILOS EN JAVA

En Java existen **dos formas de crear hilos**:

- **Implementando** la interfaz ***java.lang.Runnable***.

  La implementación de la interfaz Runnable **obliga** a programar el método sin argumentos ***run()***.

```java
package apuntes;

public class HiloViaInterface implements Runnable{

    @Override
    public void run() {
        // TODO Auto-generated method stub

    }

}
```

- **Heredando** de la clase ***java.lang.Thread***.

  No te **obliga** a implementar el método ***run()*** pero tendremos que **definirlo** si queremos un comportamiento **multihilo**:

```java
package apuntes;

public class HiloViaHerencia extends Thread {

}
```

El método **run()** se ejecuta de forma **asíncrona**, por lo que el hilo principal no se detiene.

ESTADOS DE LOS HILOS

Durante el ciclo de vida de los hilos, estos pasan por diversos estados.

Estos estados en Java se guardan dentro de la *enumeración* ***State*** contenida dentro de la clase ***java.lang.Thread***.

Para obtener el estado de un hilo usamos el método ***getState()*** de la clase ***Thread***, que devolverá alguno de los valores posibles recogidos en la enumeración indicada anteriormente. Los posibles valores se muestran en la siguiente tabla:

Estado	Valor en Thread.State	Descripción
Nuevo	**NEW**	El hilo está creado, pero aún no se ha arrancado.
Ejecutable	**RUNNABLE**	El hilo está arrancado y podría estar en ejecución o pendiente de ejecución
Bloqueado	BLOCKED	Bloqueado por un monitor. No puede acceder a un recurso porque está en uso.
Esperando	WAITING	El hilo está esperando a que otro hilo realice una acción determinada.
Esperando un tiempo	TIMED_WAITING	El hilo está esperando a que otro hilo realice una acción determinada en un período de tiempo concreto.
Finalizado	**TERMINATED**	El hilo ha terminado su ejecución.

PROBLEMAS DE CONCURRENCIA

En programación concurrente las **dificultades** aparecen cuando los **distintos hilos** acceden a un **recurso compartido** y **limitado**.

Si en el ejemplo de los ratones, estos comiesen a través de un plato al que solo pudiesen acceder de uno en uno, la concurrencia será imposible.

El acceso a los recursos compartidos limitados, por lo tanto, se debe **gestionar correctamente**.

Existen **técnicas, clases y librerías** que implementan soluciones para la gestión de recursos compartidos.

Un aspecto importante referente a los **problemas de concurrencia** es que **no siempre provocan un error** en tiempo de ejecución. Esto significa que en sucesivas ejecuciones del programa multihilo el **error** se producirá **en algunas de ellas y en otras no.**

Este tipo de problemas se resuelven, en ciertos casos, convirtiendo en secuenciales determinadas partes del código.

LA CLASE THREAD (I)

Método	Descripción
`void run()`	Se ejecuta cuando se lanza el hilo. Es el punto de entrada del hilo, como el método main() es el punto de entrada del proceso.
`void start()`	Lanza el hilo. La JVM crea el hilo y ejecuta su método **`run()`**.
`static void sleep(long ms)` `static void sleep(long ms, long ns)`	Detiene la ejecución del hilo actualmente en ejecución durante un tiempo, que se puede indicar en microsegundos o en una combinación de microsegundos y nanosegundos.

`void join()`
`void join(long ms)`
`void join(long ms, long ns)`
	Espera a que termine el hilo. Se puede indicar un tiempo máximo de espera, bien en milisegundos, bien en una combinación de milisegundos y nanosegundos.

`void interrupt()`
`boolean isInterrupted()`
`static boolean interrupted()`
	1º método: Interrumpe la ejecución de un hilo.
	2º método: Verifica si se ha interrumpido un hilo.
	3º método: Verifica si se ha interrumpido la ejecución del hilo actual, y borra el estado interrupción, de manera que una llamada posterior devolvería **`false`**.

`boolean isAlive()`	Comprueba si el hilo está vivo. Un hilo está vivo cuando se ha iniciado y no ha terminado su ejecución.
`static void yield()`	Indica al planificador que está dispuesto a ceder su uso actual de procesador. El planificador decide si atiende o no esta sugerencia.

LA CLASE THREAD (II)

Método	Descripción
`int getPriority()` `void setPriority(int nuevaPrioridad)`	Obtiene la prioridad de un hilo. Asigna prioridad a un hilo.
`static Thread currentThread()`	Devuelve un objeto de la clase `Thread` correspondiente al hilo en ejecución actualmente.
`long getId()`	Devuelve el identificador del hilo.
`String getName()` `void setName(String nombre)`	Obtiene el nombre del hilo. Asigna un nombre al hilo.
`Thread.State getState()`	Devuelve el estado del hilo.
`boolean isDaemon()` `void setDaemon(boolean on)`	Comprueba si el hilo es de tipo daemon. Establecemos si el hilo es de tipo daemon. La distinción es importante, porque la JVM termina su ejecución cuando no queda ningún hilo activo o cuando solo quedan hilos de tipo daemon.

ACTIVIDADES

1. Escribe un programa que obtenga el objeto **`Thread`** correspondiente al hilo actual de ejecución y muestre información acerca de él. Este programa no tiene que crear ningún hilo, solo mostrar información sobre el hilo actualmente en ejecución.
2. Averigua la **mínima y máxima prioridad** que puede tener un hilo. Consulta para ello la documentación de la clase **`Thread`**, y escribe un programa que muestre esta información.

INTERRUPCIONES

En computación, una **interrupción** es una **suspensión** temporal de la **ejecución** de un **proceso** o de un **hilo** de ejecución.

Las **interrupciones** no suelen pertenecer a los programas, sino al **sistema operativo**, viniendo generadas por peticiones realizadas por los dispositivos periféricos.

En **Java**, una interrupción es una indicación a un **hilo** de que debe **detener su ejecución** para hacer otra cosa. Es responsabilidad del programador decidir qué quiere hacer ante una interrupción, siendo lo más habitual detener la ejecución de un hilo.

Como se vio en la tabla anterior, se puede **interrumpir la ejecución de un hilo** con el método `interrupt()`.

Si el **hilo** se encuentra **bloqueado** en una llamada a `sleep()` (clase Thread) o a `wait()` (clase Object), se produce una **excepción** de tipo `InterruptedException`, que se debe capturar.

Un hilo puede detectar que ha sido interrumpido con el método `isInterrupted()`.

COMPARTICIÓN DE INFORMACIÓN

Varios hilos pueden crearse como instancias de la clase Thread. Los **atributos** de dichos hilos, si **no** son **estáticos**, serán **específicos** de cada uno de ellos, por lo que no se podrán utilizar para compartir información. (Ejemplo de la clase Raton)

Si se desea que **varios hilos compartan información**, existen varias alternativas:

- Utilizar **atributos estáticos**. Los atributos estáticos son comunes a todas las instancias, por lo que independientemente de la manera de construir los hilos, la información es compartida.
- Utilizando **referencias a objetos comunes** accesibles desde todos los hilos. (Código *ComparticionInstanciaUnica*)
- Utilizando **atributos no estáticos** de la instancia de una clase que implemente Runnable y construyendo los hilos a partir de dicha instancia. (Código *ComparticionRunnable*)

Existen **otras formas de compartir información**, ya sea a través de *ficheros*, *bases de datos* o *servicios de red e internet*. En todos los casos hay que tener en cuenta que la información compartida por varios hilos para lectura y escritura es una potencial fuente de errores de concurrencia.

SINCRONIZACIÓN

De manera intrínseca, la **programación concurrente** tiene **dos características** que pueden ser **fuentes de errores**:

- Los **recursos compartidos**:
  - Cómo se modifica el valor de una variable.
  - Cómo se accede a los datos de una estructura de datos.
  - Cómo se ejecuta un bloque de código.
  - Cómo se accede a un recurso limitado.
- El **orden de ejecución:**
  - Dependencias que se pueden producir entre los bloques de código en función del orden de ejecución.
  - **Ejemplo**: Un bloque de código necesita como entrada, datos generados como salida por otro bloque, esto provocará una dependencia, dado que en un sistema multihilo no se tiene control sobre el orden de las ejecuciones salvo que se establezcan mecanismos de sincronización.

EJEMPLO DE SINCRONIZACIÓN

Si ejecutamos el ejemplo **VariableCompartida** varias veces, se puede observar que nunca obtenemos el valor esperado: 1000000.

Esto se debe a que la operación de **incremento** con el operador unario **++ no** realiza una **operación atómica** (que se ejecuta sin interrupciones), sino que **consta de varios pasos** y la ejecución se puede interrumpir en cualquiera de ellos, provocando un problema de inconsistencia de memoria.

Suponiendo que el proceso de incremento en 1 de una variable está compuesto de los siguientes pasos:

1. Lectura del valor de la variable.
2. Incremento en 1 del valor de la variable.
3. Escritura del valor de la variable.

Si un hilo lee el valor de la variable siendo este 2, calcula el nuevo valor y antes de escribir el 3 resultante otro hilo toma su lugar en el procesador, este leerá de nuevo 2, calculará el nuevo valor que será 3 y lo escribirá, cuando el primer hilo vuelva al procesador continuará ejecutando el tercer paso y escribiendo el valor 3. Dos hilos han incrementado en 1 el valor de una misma variable, pero al finalizar solo se ha reflejado uno de los incrementos. Esto explica las pérdidas de incrementos que se muestran en el ejemplo anterior.

CONCEPTOS DE SINCRONIZACIÓN

- **Recursos compartidos:**
  - Un recurso compartido es un elemento del sistema que es utilizado por varios hilos simultáneamente en ejecución. Puede ser:
    - Atributo estático de una clase.
    - Atributo no estático de un objeto compartido por todos los hilos.
    - Un método estático de una clase.
    - Un método no estático de un objeto compartido por todos los hilos.
    - Una conexión a una base de datos.
    - Un socket.
    - …

CONCEPTOS DE SINCRONIZACIÓN

- **Dependencias:**
  - **No todas las tareas** se pueden **ejecutar** en un entorno **multihilo**. Para hacerlo hay que tener la certeza de que los segmentos de código que se van a ejecutar en paralelo son independientes y el orden en el que se ejecutan es irrelevante.
  - Una tarea no se podrá ejecutar en un entorno multihilo si tiene dependencias.
  - Existen **3 tipos** de dependencias principales:
    - **Dependencias de datos**: Varios segmentos de código utilizan el mismo dato.
    - **Dependencias de flujos**: Debido a que el **orden de ejecución** del programa **no se puede determinar,** por existir **instrucciones de control de flujo,** existen dependencias potenciales que no se pueden determinar en un análisis estático del código.
    - **Dependencias de recursos:** Varios segmentos de código acceden simultáneamente a recursos del procesador.

La **existencia de dependencias** de cualquier tipo **impide la programación concurrente** salvo que se puedan establecer **mecanismos de sincronización**.

CONCEPTOS DE SINCRONIZACIÓN

- **Condiciones de Bernstein:**
  - El cumplimiento de las condiciones de Bernstein **determina** si **dos segmentos de código** pueden ser **ejecutados** en **paralelo**.
  - Dados dos segmentos de código **S_1** y **S_2**, se determina que **son independientes** y pueden ser **ejecutados en paralelo** si:
    - Las **entradas de S_2 son distintas de las salidas de S_1**. De no ser así se produce lo que se conoce como **dependencia de flujo.**
    - Las **entradas de S_1 son distintas de las salidas de S_2**. De no ser así se produce lo que se conoce como **antidependencia**.
    - Las **salidas de S_1 son distintas de las salidas de S_2**. De no ser así se produce lo que se conoce como **dependencia de salida**.
  - Si dos segmentos de código cumplen con todas las condiciones de Bernstein su ejecución se puede paralelizar.

EJEMPLO CONDICIONES DE BERNSTEIN

A partir del siguiente conjunto de instrucciones, hay que indicar las que se pueden ejecutar concurrentemente:

- Instrucción 1 (I1) ⟶ a = x + z
- Instrucción 2 (I2) ⟶ x = b - 10
- Instrucción 3 (I3) ⟶ y = b + c

En primer lugar es necesario formar **2 conjuntos de instrucciones**:

- **Conjunto de lectura** ⟶ Formado por instrucciones que cuentan con variables a las que se accede en modo lectura durante su ejecución:
  - L(I1) = {x,z}
  - L(I2) = {b}
  - L(I3) = {b,c}
- **Conjunto de escritura** ⟶Formado por instrucciones que cuentan con variables a las que se accede en modo escritura durante su ejecución.
  - E(I1) = {a}
  - E(I2) = {x}
  - E(I3) = {y}

EJEMPLO CONDICIONES DE BERNSTEIN

Instrucciones que cumplen la primera regla:

- I1 e I2
- I2 e I3
- I1 e I3

Instrucciones que cumplen la segunda regla:

- I2 e I3
- I1 e I3

Instrucciones que cumplen la tercera regla:

- I1 e I2
- I2 e I3
- I1 e I3

Las instrucciones que **cumplen las 3 reglas** son **I2 e I3**, e **I1 e I3**, por lo tanto estas son las instrucciones que se podrían **ejecutar en paralelo**, pero habría que realizar algún método de sincronización entre las instrucciones I1 e I2.

CONCEPTOS DE SINCRONIZACIÓN

- **Acción y acceso atómicos:**
  - Una **acción atómica** es aquella que se ejecuta **sin interrupciones**, de **una única vez**. Cualquier efecto de la acción sólo es visible al finalizar la misma.
  - En Java, algunas acciones sencillas son atómicas:
    - **Leer y escribir variables** de los **tipos primitivos**, **excepto** los tipos **long** y **double**.
    - Leer y escribir todas las **variables** declaradas ***volatile***, **incluidos** los tipos **long** y **double**.
  - Java proporciona además un **paquete** donde encontramos **tipos atómicos**, que ya implementan una versión atómica de las operaciones más habituales como el incremento o el decremento:

    https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

- **Sección crítica:**
  - La sección crítica de un programa multihilo es el **bloque de código** que accede **a recursos compartidos**, por lo que **solo** debe ser **accedido** por **un único hilo** de ejecución.
  - Determinar correctamente la sección crítica permite sincronizar correctamente el programa para evitar errores de concurrencia, así como hacerlo eficiente para poder aprovechar al máximo el paralelismo.
  - El garantizar que ***a la sección crítica sólo acceda un hilo de ejecución*** es lo que se conoce como **exclusión mutua**.

CONCEPTOS DE SINCRONIZACIÓN

- **Exclusión mutua:**
  - Se denomina así (una *sección crítica*) para garantizar la integridad del sistema.a la ***técnica de programación*** consistente en hacer que, en un entorno concurrente, ***un proceso excluya a todos los demás*** del uso de un **recurso compartido**
  - Una manera de **conseguir la exclusión mutua** con Java es con la palabra clave `synchronized`. Esta palabra se añade en la definición de un método que no queramos que se ejecute por más de un hilo a la vez o a un bloque de código, como se verá más adelante.
- **Seguridad en hilos, seguro frente a hilos o *Thread safety*:**
  - Estos términos hacen referencia a la *propiedad* que tiene *un elemento de software* (una clase o una estructura de datos, por ejemplo) para ser **ejecutado en un entorno de múltiples hilos de forma segura**.
  - En **Java**, la **documentación** de las estructuras de datos suele especificar si son ***Thread safety*** o en cambio no están sincronizados (no son seguras frente a hilos).

PROBLEMAS DE LA PROGRAMACIÓN CONCURRENTE

En esta sección se presentan algunos de los **problemas** que surgen como consecuencia de la **programación multihilo**, así como las **soluciones** que se pueden adoptar, tanto genéricas como específicas del lenguaje Java:

- **Interbloqueo o *deadlock***:
  - Se produce cuando ***dos o más hilos están bloqueados entre sí.***
  - Por ejemplo, si el hilo A está esperando a que termine el hilo B para continuar su proceso y este, a su vez, está esperando a que termine el hilo A para continuar su proceso se produce un interbloqueo.
- **Muerte por inanición:**
  - Este problema se produce cuando se establece una política de prioridades que provoca que algunos hilos nunca tengan acceso a la CPU.
- **Condiciones de carrera:**
  - Se produce cuando dos bloques de código concurrente tienen **dependencias entre los datos de entrada y de salida** y **no se ejecutan en el orden correcto** (*dependencia de flujo* o *antidependencia*).
- **Inconsistencia de memoria:**
  - Ocurre cuando **dos o más hilos** tienen **simultáneamente valores diferentes** para la **misma variable**.
- **Condiciones deslizadas:**
  - Se produce cuando en un proceso **se evalúa una condición** para determinar si se tiene que ejecutar una sección de código y, **después de la evaluación y antes de la ejecución**, la **condición cambia de valor**. Se estaría ejecutando un bloque de código cuya condición no se está cumpliendo.

SINCRONIZACIÓN BÁSICA: VARIABLES *VOLATILE*

En un entorno de computación de **múltiples núcleos**, los procesadores disponen de técnicas de optimización y algunas de ellas se basan en el uso de **memoria caché**.

Estas técnicas habitualmente son **ventajosas**, pero en la **programación concurrente** pueden ser una **fuente de errores**.

Cuando **varios hilos comparten la misma variable,** si esta **se almacena** en las **cachés** de los núcleos, puede ser que los hilos vean **copias distintas de la misma variable**, lo que puede provocar **inconsistencia de memoria**.

Para **evitar** que una **variable** se **almacene** en la **caché** de procesador y que todos los hilos accedan a la misma copia, en Java se utiliza la palabra clave `volatile`. Declarando una variable como `volatile` solo existirá una copia en el procesador.

El siguiente código de ejemplo muestra la declaración de una variable `volatile`:

```
private volatile static long contador;
```

Esta solución **no resuelve** por sí sola **todos** los **problemas** de **inconsistencia** de memoria. Si **varios hilos modifican concurrentemente** la misma variable, aunque sea declarada como `volatile`, se podría seguir produciendo el error. Para solucionarlo habría que incluir **mecanismos de sincronización**.

Las variables `volatile` serían **apropiadas** para sistemas en los que **un único hilo modifica** el valor de la variable y el **resto** solamente lo **consultan**.

SINCRONIZACIÓN BÁSICA: *WAIT, NOTIFY Y NOTIFYALL*

Los métodos **`wait`**, **`notify`** y **`notifyAll`** son propios de la clase **`Object`**, por lo que las clases en Java disponen de ellos.

Todos estos métodos se tienen que **invocar desde** segmentos de código de un hilo que disponga de un monitor, como, por ejemplo, **un bloque o un segmento sincronizados**, y **obliga a capturar** una excepción del tipo **`InterruptedException`**.

El método **`wait`** **detiene la ejecución del hilo** y los métodos notify y notifyAll producen la reactivación de los hilos detenidos.

El método **`notify`** hace continuar a un único segmento al azar de los que están detenidos con **`wait`**.

El método **`notifyAll`** hace continuar a todos los segmentos detenidos con **`wait`**.

Ejemplo: **WaitNotifySimple**

SINCRONIZACIÓN BÁSICA: *WAIT, NOTIFY Y NOTIFYALL*

Los métodos **`wait`**, **`notify`** y **`notifyAll`** se utilizan cuando **un hilo puede realizar o no** una **operación** dependiente del estado del sistema. La **verificación** de ese **estado** debe hacerse dentro de un **método o bloque `synchronized`**.

- **No es posible** realizar la operación → llamamos a **`wait()`** sobre el **objeto de bloqueo.**
- **Sí se puede** realizar la operación → Se cambia el estado del objeto y se llama a **`notify()`** o **`notifyAll()`** sobre el **objeto de bloqueo**.

La estructura de un programa que utiliza estos métodos debe tener la siguiente estructura.

```
synchronized(objetoBloqueo) { //Acceso a recurso con exclusión mutua
    while(!condicionParaOperacion) { // Depende de estado de objetoBloqueo
        try {
            objetoBloqueo.wait(); //Espera a que alguien cambie de estado y notifique
        } catch(InterruptedException ex) {

        }
    }

    // Ya se puede realizar la operación
    // No puede interferir otro hilo desde comprobación de condición
    // en while(), porque todo está dentro de un bucle synchronized,
    // en el que solo se cede el control con wait()
    realizar_operacion();

    // Si como consecuencia de realizar la operación pudiera continuar otro hilo,
    // habría que notificar que el recurso se ha liberado
    objetoBloqueo.notifyAll(); // También se podría utilizar objetoBloqueo.notify();

}
```

SINCRONIZACIÓN BÁSICA: *JOIN*

El método `join` permite indicar a un hilo que debe suspender su ejecución hasta que termina otro hilo de referencia.

Este método debe ejecutarse dentro del **bloque asíncrono** del código, ya que de lo contrario no tendrá ningún efecto.

Ejemplo: **JoinBasico**

SINCRONIZACIÓN BÁSICA: *ESTRUCTURAS DE DATOS*

Las **estructuras de datos** convencionales proporcionadas por Java satisfacen cualquier necesidad relacionada con el almacenamiento de datos en memoria.

Desde el punto de vista de la programación concurrente hay que tener en cuenta que las clases *ArrayList*, *Vector*, *HashMap* o *HashSet*, todas ellas del paquete *java.util* **no están sincronizadas**, lo que implica que no están programadas para ser consistentes frente al acceso desde múltiples hilos.

Para poder utilizar estas estructuras hay que:

- Usar alguna de las distintas **técnicas de sincronización** ya vistas.
- Convertirlas a **estructuras sincronizadas** con los métodos estáticos proporcionados por la clase *java.util.Collections* (por ejemplo, *synchronizedList* para objetos que implementen la interfaz List).
- Utilizar las estructuras de datos proporcionadas por el paquete *java.util.concurrent*.

SINCRONIZACIÓN AVANZADA:
EXCLUSIÓN MUTUA, SYNCHRONIZED Y MONITORES

Uno de los mecanismos proporcionados por Java para **sincronizar segmentos de código** consiste en utilizar la palabra clave ***synchronized***.

Mediante el uso de ***synchronized*** se puede **limitar el acceso a un segmento del código** a un único hilo concurrentemente, lográndose así la **exclusión mutua** o ***mutex***.

Permite **sincronizar tanto métodos como segmentos de código** (declaraciones sincronizadas), permitiendo esta última alternativa ***delimitar la sección crítica*** con más precisión.

A **nivel de método** basta con añadir la palabra synchronized en la declaración del método:

```
public synchronized void calcular()
```

A **nivel de segmento de código**, la sincronización se efectúa delimitando un bloque de sentencias haciendo una declaración sincronizada.

```
synchronized (objetoBloqueo){
     // Bloque de sentencias sincronizadas
}
```

SINCRONIZACIÓN AVANZADA:
EXCLUSIÓN MUTUA, SYNCHRONIZED Y MONITORES

Este sistema utiliza un concepto conocido como ***bloque intrínseco***, ***bloqueo de monitor*** o **monitor**.

El **monitor** es un elemento **asociado** a una **instancia** que hace la **función de candado**. Cuando se ejecuta un *método o bloque sincronizado* utilizando un **determinado monitor**, este se **bloquea** y **no puede utilizarse** hasta que no **queda liberado**, impidiendo ejecutar un código que utilice el mismo bloque.

Cuando se utilizan **métodos sincronizados** se usa como **monitor** el ***objeto en el que están***. Esto supone que cuando un método sincronizado de un objeto se está ejecutando ningún otro método sincronizado de ese objeto se puede ejecutar.

La exclusión es, por tanto, muy genérica y puede que sea poco eficiente en muchos casos.

Ejemplo: **sincronizacion.SincronizacionMetodos**

SINCRONIZACIÓN AVANZADA:
EXCLUSIÓN MUTUA, SYNCHRONIZED Y MONITORES

La sincronización a **nivel de segmento** necesita también un monitor, pero al no depender del objeto en el que se está ejecutando es más flexible.

Utilizando bloques sincronizados **no es necesario bloquear todos los segmentos de un objeto** como ocurre con los métodos de objeto, sino que se pueden ***agrupar en monitores distintos***.

Ejemplo: **sincronizacion.SincronizacionSegmento**

SINCRONIZACIÓN AVANZADA: SEMÁFOROS

Cuando una sección crítica permite ser ejecutada por más de un hilo, pero el número de estos está limitado se utiliza lo que se conoce como **semáforos**.

Los semáforos se implementan en Java con la **clase *Semaphore***.

Los semáforos se suelen utilizar cuando **un recurso tiene una capacidad limitada** y se desea controlar el número de consumidores de dicho recurso.

Al construir el semáforo, se indica en el **constructor una capacidad**, que hace referencia al **número de hilos** que puede estar ejecutando concurrentemente.

Esta **capacidad** se convierte en **número de permisos** de **acceso** que se conceden a los **bloques** de código que quieran hacer **uso del recurso limitado**. Si **todos** los **permisos** están **concedidos**, los **hilos** quedan **en espera** a que los **propietarios** de los permisos los **liberen**.

SINCRONIZACIÓN AVANZADA: SEMÁFOROS

Método	Descripción
`void acquire()` `void acquire(int permits)`	Adquiere uno o más permisos del semáforo si hay alguno disponible. En caso contrario, el hilo se queda en espera.
`void release()` `void release(int permits)`	Libera uno o más permisos concedidos previamente.

`boolean tryAcquire()`
`boolean tryAcquire(int permits)`
`boolean tryAcquire(int permits, long timeout, TimeUnit unit)`
`boolean tryAcquire(long timeout, TimeUnit unit)`
	Intenta obtener uno o más permisos, pudiendo quedar en espera durante un tiempo limitado.

Para más información de la clase Semaphore, consultar la documentación de java:
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html

Ejemplo: **sincronizacion.SemaforoBasico**

EJERCICIO: PRODUCTORES Y CONSUMIDORES

A continuación, se presenta una variante del típico problema de **PRODUCTORES y CONSUMIDORES**. En este tipo de problemas, uno o más procesos PRODUCTORES ponen datos a disposición de uno o más procesos CONSUMIDORES.

Si un proceso CONSUMIDOR intenta obtener un dato y no hay ninguno disponible, debe esperar a que algún proceso productor introduzca alguno.

Los **datos** se suelen poner en una **cola**, de la que se extraen los elementos en el mismo orden en que se introducen, según un planteamiento **FIFO** o *first in, first out*.

Se puede limitar el número de elementos que se admiten en la cola, de manera que, cuando contiene el máximo número de datos, antes de introducir uno nuevo, los PRODUCTORES deben esperar a que algún proceso CONSUMIDOR retire uno.

En el proyecto de ejemplo, hay una posible solución a este problema en el que sólo hay un PRODUCTOR y un CONSUMIDOR, y en el que el CONTENEDOR sólo puede almacenar un dato. Analiza este ejemplo y piensa en cómo se debe modificar para que haya varios elementos en el CONTENEDOR en vez de sólo uno.