

Estados de los hilos – ciclo de vida completo

Estado	Descripción	¿Cómo se alcanza?
1. Nuevo (NEW)	El hilo ha sido creado (<code>new Thread()</code>), pero aún no ha sido iniciado. Está esperando ser programado.	Se llama al constructor de la clase <code>Thread</code> .
2. Ejecutable (RUNNABLE)	El hilo se está ejecutando en la JVM o está listo para ejecutarse y esperando su turno en el planificador del sistema operativo (Scheduler).	Se llama al método <code>start()</code> del hilo.
3. Bloqueado (BLOCKED)	El hilo está esperando adquirir un bloqueo de monitor para entrar a un método o bloque <code>synchronized</code> .	Intenta entrar a un método/bloque <code>synchronized</code> cuyo bloqueo está siendo retenido por otro hilo.
4. Esperando (WAITING)	El hilo está esperando indefinidamente a que otro hilo realice una acción particular.	Se llama a <code>Object.wait()</code> (sin tiempo límite), <code>Thread.join()</code> (sin tiempo límite), o <code>LockSupport.park()</code> .
5. Esperando con Tiempo (TIMED_WAITING)	El hilo está esperando por un tiempo específico a que otro hilo realice una acción, o a que expire el tiempo de espera.	Se llama a <code>Thread.sleep(ms)</code> , <code>Object.wait(ms)</code> , <code>Thread.join(ms)</code> , o <code>LockSupport.parkNanos(ns)</code> .
6. Terminado (TERMINATED)	La ejecución del hilo ha finalizado. Puede ser porque su método <code>run()</code> ha terminado normalmente o porque se produjo una excepción no capturada.	El método <code>run()</code> finaliza su ejecución.

Productor – consumidor 1-N

Queremos diseñar un programa en Java que simule el **problema clásico de Productor–Consumidor**, donde:

- Existe **un productor** que va generando números enteros.
- Existen **varios consumidores** que irán recogiendo esos números para procesarlos.
- Todos ellos **comparten una misma cola (buffer)** donde se almacenan temporalmente los datos.

La cola se implementará con una **lista (ArrayList)** y tendrá **un tamaño máximo**. Por lo tanto:

- Si la cola está **llena**, el **productor debe esperar** antes de producir más.
- Si la cola está **vacía**, los **consumidores deben esperar** hasta que haya datos disponibles.

Se debe garantizar que:

1. **No se pierden datos.**
2. **No se consumen datos repetidos.**
3. **No se accede a la cola de forma simultánea** (el acceso debe ser seguro).

Para ello se utilizarán:

- **synchronized** para asegurar la exclusión mutua.
- **wait()** para que un hilo **espere** cuando no puede continuar.
- **notifyAll()** para **despertar a los hilos** que puedan continuar su trabajo cuando cambia el estado de la cola.

Usaremos las siguientes clases:

Clase	Función
Cola	Recurso compartido → almacenar y extraer valores de forma segura.
Productor	Genera números y los coloca en la cola.
Consumidor	Extrae números de la cola y los procesa.

Ejemplo ejecución:

[1, 2, 3] tamaño=3, límite=5

Productor produce → encolar → notifyAll()

Consumidor-1 consume → desencolar → notifyAll()

Consumidor-2 consume → desencolar → notifyAll()

Consumidor-3 encuentra cola vacía → wait()

Aclaración:

- Si la cola **no está llena**, el productor **puede seguir produciendo sin esperar**.
- El consumidor **no tiene que consumir inmediatamente**, puede llegar más tarde.
- Solo cuando la cola **se llena**, el productor **espera** (wait()).
- Solo cuando la cola **se vacía**, los consumidores **esperan** (wait()).

Cola (capacidad = 5):

[] → vacía

Productor produce 1 →

[1]

Productor produce 2 →

[1, 2]

Productor produce 3 →

[1, 2, 3] → (productor sigue, aún hay espacio)

Consumidor consume →

[2, 3] → consumo lento

- **Se puede producir varias veces seguidas.**
- **Un solo consumidor puede consumir cuando llegue su turno.**
- **No hay alternancia estricta 1–1.**

Explicación productor-consumidor

Alternancia exticta

El escenario es el siguiente:

- El productor solo puede producir después de que el consumidor consume.
 - El consumidor solo puede consumir después de que el productor produzca.
 - Solo hay un único valor compartido.

Es decir:

Un turno cada uno → PRODUCIR → CONSUMIR → PRODUCIR → CONSUMIR...

¿Qué tengo que implementar?

Usamos **una variable booleana disponible** para indicar si hay algo para consumir:

disponible	Significa	Quién puede actuar
false	No hay dato	Productor puede producir
true	Hay dato pendiente	Consumidor debe consumir

Y wait() / notify() para que se turnen.

Visualmente sería este el circuito:

Productor produce → Consumidor consume



La salida siempre será sin solapas así:

Productor produce: 1

Consumidor consume: 1

Productor produce: 2

Consumidor consume: 2

Productor produce: 3

Consumidor consume: 3

...

Es decir, internamente podrá estar pasando esto:

Estado inicial:

disponible = false

Consumidor → ve disponible = false → WAITING

Productor → ve disponible = false → PRODUCE → notify()

Consumidor → se despierta → CONSUME → notify()

Según está programado el código, si disponible **empieza en false**, significa que **no hay nada que consumir**, por lo tanto, el **primer turno es del productor**.

Explicación productor-consumidor

💡 Situación inicial: la cola está vacía

Cola (capacidad 5):

[][][][][]

↑

Frontal y Final (vacía)

Productor y consumidor empiezan:

Productor → mete datos

Consumidor → saca datos

🚀 El productor va MÁS RÁPIDO

Va metiendo valores:

Productor produce: 1

[1][][][][]

Productor produce: 2

[1][2][][][]

Productor produce: 3

[1][2][3][][]

Productor produce: 4

[1][2][3][4][]

Productor produce: 5

[1][2][3][4][5]

😲 Hasta aquí todo bien.

Pero el consumidor es más lento

Mientras el productor ya va por 6...

Consumer consume → todavía está con el 1

Productor intenta meter **6** pero:

[1][2][3][4][5]

↑ COLA LLENA

Resultado:

Productor: cola llena, se pierde el dato 6

No se guarda → **se descarta**.

Línea temporal

Tiempo →

Prod: 1 2 3 4 5 6 7 8 ...

Con: 1 2 3

Se pierden: 6, 7, 8 ...

El consumidor **no puede seguir el ritmo** → el productor **pierde datos**.

Conclusión

Si el **productor va más rápido**, la **cola se llena** y los datos que llegan después **se pierden**, porque no hay espacio para guardarlos.

Y si fuera al revés...

Si el **consumidor fuera más rápido**:

Cola vacía → Consumidor: "No hay nada que consumir"



2ºDAM - Programación de servicios y procesos

Tema 2 – Programación multiproceso

CONTENIDOS

- 1. Concepto de hilo**
- 2. Hilo vs. Proceso**
- 3. Estados de un hilo y ciclo de vida**
- 4. Hilos de usuario e hilos de sistema**
- 5. Usos comunes**
- 6. Creación y gestión de hilos**

I. Concepto de hilo

Un hilo, denominado también subprocesso, es un flujo de control secuencial independiente dentro de un proceso y está asociado con una secuencia de instrucciones, un conjunto de registros y una pila. Observaciones:

- Un hilo no puede existir independientemente de un proceso.
- Un hilo no puede ejecutarse por sí solo.
- Dentro de cada proceso puede haber varios hilos ejecutándose.
- Un único hilo es similar a un programa secuencial; por si mismo no nos ofrece nada nuevo.

Lo nuevo es que cada uno de estos hilos puede ejecutar actividades diferentes al mismo tiempo.

I.I. Recursos compartidos por hilos

- Un hilo lleva asociados los siguientes elementos:
 - Un identificador único.
 - Un contador de programa propio.
 - Un conjunto de registros.
 - Una pila (variables locales).

I.I. Recursos compartidos por hilos

Un hilo puede compartir con otros hilos del mismo proceso los siguientes recursos:

- Código.
- Datos (como variables globales).
- Otros recursos del sistema operativo, como los ficheros abiertos y las señales.

El hecho de que los hilos comparten recursos (ej. pudiendo acceder a las mismas variables) implica que sea necesario utilizar esquemas de bloqueo y sincronización, lo que puede hacer más difícil el desarrollo de los programas y así como su depuración.

I.2. Ventajas y uso de hilos

- Como consecuencia de **compartir el espacio de memoria**, los hilos aportan las siguientes ventajas sobre los procesos:
 - Se consumen menos recursos en el lanzamiento y la ejecución de un hilo que en el lanzamiento y ejecución de un proceso.
 - Se tarda menos tiempo en crear y terminar un hilo que un proceso.
 - La conmutación entre hilos del mismo proceso es bastante más rápida que entre procesos.

I.2. Ventajas y uso de hilos

- Es por esas razones, por lo que a los hilos se les denomina también **procesos ligeros**. Se aconseja utilizar hilos en una aplicación cuando:
 - La aplicación maneja entradas de varios dispositivos de comunicación.
 - La aplicación debe poder realizar diferentes tareas a la vez.
 - Interesa diferenciar tareas con una prioridad variada. Por ejemplo, una prioridad alta para manejar tareas de tiempo crítico y una prioridad baja para otras tareas.
 - La aplicación se va a ejecutar en un entorno multiprocesador.

2. Hilo vs. Proceso

Hay dos tipos distintos de multitareas: **basado en procesos** (como hemos visto en el tema anterior) y **basado en hilos**. Es importante entender la diferencia entre los dos.

- Un **proceso** es, en esencia, un **programa que se está ejecutando**. Por lo tanto, la multitarea basada en procesos es la característica que le permite a su computadora ejecutar dos o más programas al mismo tiempo. Por ejemplo, es una **multitarea basada en procesos** que le permite ejecutar el compilador de Java al mismo tiempo que utiliza un editor de texto o navega por Internet. En la multitarea basada en procesos, **un programa es la unidad de código más pequeña que puede enviar el planificador del sistema operativo**.

2. Hilo vs. Proceso

- En un entorno **multitarea basado en hilos**, el **hilo es la unidad más pequeña de código distribuible**. Esto significa que un solo programa puede realizar dos o más tareas a la vez. Por ejemplo, un editor de texto puede formatear texto al mismo tiempo que está imprimiendo, siempre que estas dos acciones se realicen mediante dos hilos separados. Aunque los programas Java utilizan entornos multitarea basados en procesos como hemos visto en el anterior tema, **la multitarea basada en procesos no está bajo el control de Java**. La **multitarea multihilo** sí lo está bajo el control en este caso de Java.

3. Estados de un hilo y ciclo de vida

- En un entorno **multitarea basado en hilos**, el **hilo es la unidad más pequeña de código distribuible**. Esto significa que un solo programa puede realizar dos o más tareas a la vez. Por ejemplo, un editor de texto puede formatear texto al mismo tiempo que está imprimiendo, siempre que estas dos acciones se realicen mediante dos hilos separados. Aunque los programas Java utilizan entornos multitarea basados en procesos como hemos visto en el anterior tema, **la multitarea basada en procesos no está bajo el control de Java**. La **multitarea multihilo** sí lo está bajo el control en este caso de Java.

3. Estados de un hilo y ciclo de vida

Un hilo puede estar en uno de estos estados:

- **New (Nuevo).** Es el estado cuando se crea un objeto hilo con el operador new. En este estado el hilo aún no se ejecuta; es decir, el programa no ha comenzado la ejecución del código del método *run()* del hilo.
- **Runnable (Ejecutable).** Cuando se invoca al método *start()*, el hilo pasa a este estado. El sistema operativo tiene que asignar tiempo de CPU al hilo para que se ejecute; por tanto, en este estado el hilo puede estar o no en ejecución.

3. Estados de un hilo y ciclo de vida

- **Blocked (Bloqueado).** En este estado podría ejecutarse el hilo, pero hay algo que lo evita. Un hilo entra en estado bloqueado cuando ocurre una de las siguientes acciones:
 - a) Alguien llama al método ***sleep()*** del hilo, es decir, se ha puesto a dormir.
 - b) El hilo está esperando a que se complete una operación de entrada/salida.
 - c) El hilo llama al método ***wait()***. El hilo no se volverá ejecutable hasta que reciba los mensajes ***notify()*** o ***notifyAll()***.
 - d) El hilo intenta bloquear un objeto que está actualmente bloqueado por otro hilo.
 - e) Alguien llama al método ***suspend()*** del hilo. No se volverá a ejecutar de nuevo hasta que reciba el mensaje ***resume()***.

3. Estados de un hilo y ciclo de vida

- **Dead (Muerto).** Un hilo muere por varias razones:
 - De muerte natural, porque el método **run()** finaliza con normalidad.
 - Repentinamente debido a alguna excepción no capturada en el método **run()**.
 - Invocando al método **stop()** que lanza una excepción **ThreadDeath** que mata al hilo, pero este método está en desuso y no se debe llamar ya que cuando un hilo se detiene, inmediatamente no libera los bloqueos de los objetos que ha bloqueado. Para detener un hilo de manera segura, se puede usar una variable.

3. Estados de un hilo y ciclo de vida

Del estado bloqueado se pasa a ejecutable cuando: expira el número de milisegundos de *sleep()*, se completa la operación de E/S, recibe los mensajes *notify()* o *notifyAll()*, el bloqueo del objeto finaliza, o se llama al método *resume()*.

Los métodos *resume()*, *suspend()* y *stop()* están en desuso.

Veamos el [ejemplo HiloEjemploDead.java](#)

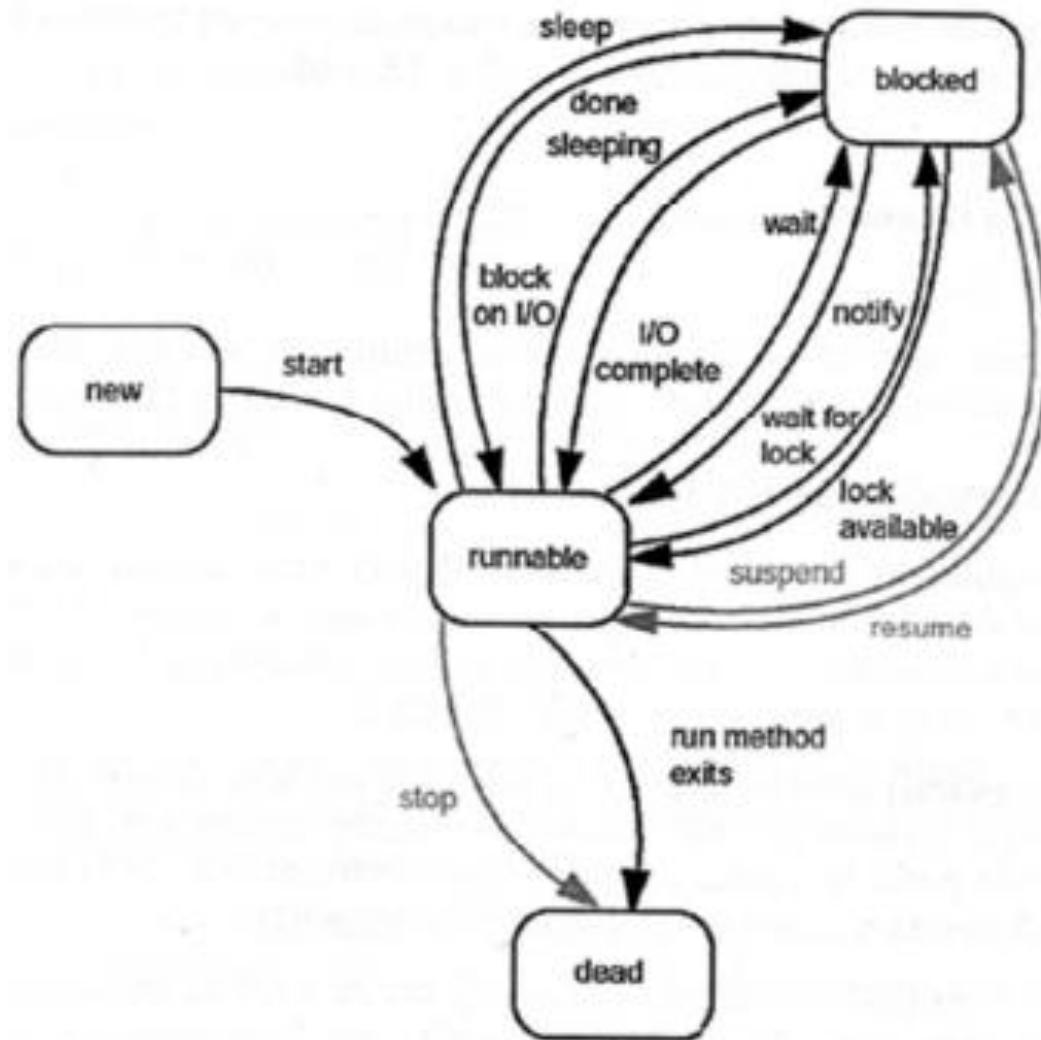
3. Estados de un hilo y ciclo de vida

Cuando un hilo está bloqueado (o cuando muere), otro hilo está previsto para funcionar. Cuando un hilo bloqueado se reactiva (por ejemplo porque finaliza el número de milisegundos que permanece dormido o porque la E/S que se esperaba se ha completado), el planificador comprueba si tiene una prioridad más alta que el hilo que se está ejecutando actualmente. Si es así, se antepone al actual hilo y selecciona el nuevo hilo para ejecutarlo. En una máquina con varios procesadores, cada procesador puede ejecutar un hilo, se pueden tener varios hilos en paralelo. En tal máquina, un hilo con máxima prioridad se adelantará a otro si no hay disponible procesador para ejecutarlo.

3. Estados de un hilo y ciclo de vida

El **ciclo de vida de un hilo** comprende los diferentes estados en los que puede estar un hilo desde que se crea o nace hasta que finaliza o muere.

En la siguiente imagen se puede observar los distintos estados por los que puede transitar un hilo:



4. Hilos de usuario e hilos de sistema

- Hay dos grandes categorías en la implementación de hilos:
 - Hilos a nivel de usuario
 - Hilos a nivel de Kernel

También conocidos como **ULT** (User Level Thread) y **KLT** (Kernel Level Thread).

4. I. Hilos a nivel de usuario

En una aplicación ULT pura, todo el trabajo de gestión de hilos lo realiza la aplicación y el núcleo o kernel no es consciente de la existencia de hilos. Es posible programar una aplicación como multihilo mediante una biblioteca de hilos. La misma contiene el código para crear y destruir hilos, intercambiar mensajes y datos entre hilos, para planificar la ejecución de hilos y para salvar y restaurar el contexto de los hilos. Todas las operaciones descritas se llevan a cabo en el espacio de usuario de un mismo proceso. El kernel continúa planificando el proceso como una unidad y asignándole un único estado (Listo, bloqueado, etc.).

4. I. Hilos a nivel de usuario

Las **ventajas** que ofrecen los **ULT** son:

- El intercambio de los hilos no necesita los privilegios del modo kernel, porque todas las estructuras de datos están en el espacio de direcciones de usuario de un mismo proceso. Por lo tanto, el proceso no debe cambiar a modo kernel para gestionar hilos.
- Se evita la sobrecarga de cambio de modo y con esto el sobrecoste.
- Se puede realizar una planificación específica. Dependiendo de qué aplicación sea, se puede decidir por una u otra planificación según sus ventajas.

Los **ULT** pueden ejecutar en cualquier sistema operativo.

4. I. Hilos a nivel de usuario

Las **desventajas** más relevantes son:

- En la mayoría de los sistemas operativos las llamadas al sistema (System calls) son bloqueantes. Cuando un hilo realiza una llamada al sistema, se bloquea el mismo y también el resto de los hilos del proceso.
- En una estrategia ULT pura, una aplicación multihilo no puede aprovechar las ventajas de los multiprocesadores. El núcleo asigna un solo proceso a un solo procesador, ya que como el núcleo no interviene y ve al conjunto de hilos como un solo proceso.

Una solución al bloqueo mediante a llamadas al sistema es usando la técnica de *jacketing*, que es convertir una llamada bloqueante en no bloqueante.

4.2. Hilos a nivel de sistema

- En una aplicación **KLT** pura, todo el trabajo de gestión de hilos lo realiza el kernel. En el área de la aplicación no hay código de gestión de hilos, únicamente un API (interfaz de programas de aplicación) para la gestión de hilos en el núcleo. Windows 2000, Linux y OS/2 utilizan (utilizaban) este método.

4.2. Hilos a nivel de sistema

Las principales **ventajas** de los KLT:

- El kernel puede planificar simultáneamente múltiples hilos del mismo proceso en múltiples procesadores.
- Si se bloquea un hilo, puede planificar otro del mismo proceso.
- Las propias funciones del kernel pueden ser multihilo.

Las **desventajas** que presentan son:

- El paso de control de un hilo a otro precisa de un cambio de modo.

5. Usos comunes de hilos

Algunos de los casos más claros en los cuales el uso de hilos se convierte en una tarea fundamental para el desarrollo de aplicaciones son:

- **Trabajo interactivo y en segundo plano:** por ejemplo, en un programa de hoja de cálculo un hilo puede estar visualizando los menús y leer la entrada del usuario mientras que otro hilo ejecuta las órdenes y actualiza la hoja de cálculo. Esta medida suele aumentar la velocidad que se percibe en la aplicación, permitiendo que el programa pida la orden siguiente antes de terminar la anterior.

5. Usos comunes de hilos

- **Procesamiento asíncrono:** los elementos asíncronos de un programa se pueden implementar como hilos. Un ejemplo es como el software de procesamiento de texto guarda archivos temporales cuando se está trabajando en dicho programa. Se crea un hilo que tiene como función guardar una copia de respaldo mientras se continúa con la operación de escritura por el usuario sin interferir en la misma.

5. Usos comunes de hilos

- **Aceleración de la ejecución:** se pueden ejecutar, por ejemplo, un lote mientras otro hilo lee el lote siguiente de un dispositivo.
- **Estructuración modular de los programas:** puede ser un mecanismo eficiente para un programa que ejecuta una gran variedad de actividades, teniendo las mismas bien separadas mediante a hilos que realizan cada una de ellas.

6. Creación y gestión de hilos

- En Java existen dos formas para crear hilos:
 - Extendiendo la clase **Thread**
 - Implementando la interfaz **Runnable**.

Ambas son parte del paquete `java.lang`. El hilo encapsula un hilo de ejecución. Para crear un nuevo hilo, su programa extenderá **Thread** o implementará la interfaz **Runnable**.

6. I. Clase Thread

La forma más sencilla de añadir funcionalidad de hilo a una clase es extender la clase Thread. Esta subclase debe sobrescribir el método ***run()*** con las acciones que el hilo debe desarrollar. La clase Thread también define otros métodos como ***start()*** y ***stop()*** (este último en desuso) para iniciar y parar la ejecución del hilo.

API clase [Thread](#)

6. I. Clase Thread

 **Ejemplo:** declara la clase **PrimerHilo.java** que extiende la clase **Thread**, desde el constructor se inicializa una variable numérica que se usará para pintar un número de veces un mensaje; en el método **run()** se escribe la funcionalidad del hilo.

6. I. Clase Thread

Ejemplo: se crea una clase **HiloEjemplo1.java** que extiende Thread. Dentro de la clase se definen el constructor, el método run() con la funcionalidad que realizará el hilo y el método main() donde se crearán 3 hilos. La misión del hilo, descrita en el método **run()**, será visualizar un mensaje donde se muestre el nombre del hilo que se está ejecutando y el contenido de un contador. Se utiliza una variable para mostrar el nombre del hilo que se ejecuta, esta variable se pasa al constructor y éste se lo pasa al constructor de la clase base Thread mediante la palabra reservada super, para acceder a este nombre se usa el método **getName()**. Desde el método **main()** se crean los hilos y para iniciar cada hilo usamos el método **start()**. En este ejemplo, se ha incluido el método **main()** dentro de la clase hilo. Podemos definir por un lado la clase hilo y por otro la clase que usa el hilo.

6. I. Clase Thread

 **Ejemplo:** partiendo del ejemplo anterior vemos cómo se puede utilizar una clase con el hilo y otra con el *main()*. Clases ***HiloEjemploI_V2.java*** y ***UsaHiloEjemploI_V2.java*** respectivamente.

6.1. Clase Thread

En las siguientes tablas se muestran algunos métodos útiles sobre los hilos:

MÉTODOS	MISIÓN
start()	Hace que el hilo comience la ejecución; la máquina virtual de Java llama al método run() de este hilo.
boolean isAlive()	Comprueba si el hilo está vivo
sleep(long millis)	Hace que el hilo actualmente en ejecución pase a dormir temporalmente durante el número de milisegundos especificado. Puede lanzar la excepción <i>InterruptedException</i> .
run()	Constituye el cuerpo del hilo. Es llamado por el método start() después de que el hilo apropiado del sistema se haya inicializado. Si el método run() devuelve el control, el hilo se detiene. Es el único método de la interfaz Runnable .
String toString()	Devuelve una representación en formato cadena de este hilo, incluyendo el nombre del hilo, la prioridad, y el grupo de hilos. Ejemplo: Thread[HILO1,2,main]
long getId()	Devuelve el identificador del hilo.
void yield()	Hace que el hilo actual de ejecución pare temporalmente y permita que otros hilos se ejecuten.
String getName()	Devuelve el nombre del hilo.
setName(String name)	Cambia el nombre de este hilo, asignándole el especificado como argumento.

6.1. Clase Thread

En las siguientes tablas se muestran algunos métodos útiles sobre los hilos:

MÉTODOS	MISIÓN
<code>int getPriority()</code>	Devuelve la prioridad del hilo.
<code>setPriority(int p)</code>	Cambia la prioridad del hilo al valor entero p.
<code>void interrupt()</code>	Interrumpe la ejecución del hilo
<code>boolean interrupted()</code>	Comprueba si el hilo actual ha sido interrumpido.
<code>Thread currentThread()</code>	Devuelve una referencia al objeto hilo que se está ejecutando actualmente.
<code>boolean isDaemon()</code>	Comprueba si el hilo es un hilo Daemon. Los hilos daemon o demonio son hilos con prioridad baja que normalmente se ejecutan en segundo plano. Un ejemplo de hilo demonio que está ejecutándose continuamente es el recolector de basura (<i>garbage collector</i>).
<code>setDaemon(boolean on)</code>	Establece este hilo como hilo Daemon, asignando el valor <i>true</i> , o como hilo de usuario, pasando el valor <i>false</i> .
<code>void stop()</code>	Detiene el hilo. Este método está en desuso.
<code>Thread currentThread()</code>	Devuelve una referencia al objeto hilo actualmente en ejecución.
<code>int activeCount()</code>	Este método devuelve el número de hilos activos en el grupo de hilos del hilo actual.
<code>Thread.State getState()</code>	Devuelve el estado del hilo: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED

6. I. Clase Thread

El método **getState()** devuelve una constante que indica el estado del hilo. Un hilo puede estar en uno de los siguientes estados:

- NEW: Un hilo que aún no ha comenzado está en este estado.
- RUNNABLE: Un subprocesso que se ejecuta en la máquina virtual Java se encuentra en este estado.
- BLOCKED: Un subprocesso que está bloqueado esperando un bloqueo de monitor se encuentra en este estado.

6. I. Clase Thread

- WAITING: Un hilo que está esperando indefinidamente a que otro hilo realice una acción en particular se encuentra en este estado.
- TIMED_WAITING: Un subprocesso que está esperando a que otro subprocesso realice una acción durante un tiempo de espera especificado se encuentra en este estado.
- TERMINATED: Un hilo que ha salido está en este estado.

Un hilo puede estar en un solo estado en un momento dado. Estos estados **son estados de máquinas virtuales** que no reflejan ningún estado de subprocessos del sistema operativo.

6. I. Clase Thread

Ejemplo: en la clase *HiloEjemplo2.java* se muestra el uso de algunos de los métodos anteriores. El método ***toString()*** devuelve un string que representa al hilo: Thread[nombre del hilo, la prioridad, grupo de hilos], el método ***currentThread()*** devuelve una referencia al objeto hilo actualmente en ejecución y ***activeCount()*** devuelve el número de hilos activos actualmente dentro del grupo.

6. I. Clase Thread

Todo hilo de ejecución en Java debe formar parte de un grupo. Por defecto, si no se especifica ningún grupo en el constructor, los hilos serán miembros del grupo *main*, que es creado por el sistema cuando arranca la aplicación Java.

La clase **ThreadGrupo** se utiliza para manejar grupos de hilos en las aplicaciones Java. La clase Thread proporciona constructores en los que se puede especificar el grupo del hilo que se está creando en el mismo momento de instanciarlo.

6. I. Clase Thread

 **Ejemplo:** la clase **HiloEjemplo2Grupos.java** crea un grupo de hilos de nombre “Grupo de hilos” y después crea tres hilos usando el constructor de la clase Thread:

Thread (grupo ThreadGroup, destino Runnable, nombre String)

Donde se especifica: grupo de hilos, el objeto hilo y el nombre del hilo.

6.2. Interfaz Runnable

Para añadir la funcionalidad de hilo a una clase que deriva de otra clase (por ejemplo, un applet), siendo esta distinta de Thread, se utiliza la **interfaz Runnable**. Esta interfaz añade la funcionalidad de hilo a una clase con solo implementarla.

Por ejemplo, para añadir la funcionalidad de hilo a un applet definimos la clase como:

public class Reloj extends Applet implements Runnable()

6.2. Interfaz Runnable

La interfaz Runnable proporciona un único método, el método **run()**. Este es ejecutado por el objeto hilo asociado. La forma general de declarar un hilo implementando la interfaz **Runnable** es la siguiente:

```
class NombreHilo implements Runnable {  
    // propiedades, constructores y métodos de la clase  
  
    public void run() {  
        // acciones que lleva a cabo el hilo  
    }  
}
```

6.2. Interfaz Runnable

Para crear un objeto hilo con el comportamiento de NombreHilo se escribe lo siguiente:

```
NombreHilo h=new NombreHilo();
```

Y para iniciar su ejecución utilizamos el método start():

```
new Thread(h).start();
```

O bien en dos pasos:

```
Thread h1= new Thread(h);  
h1.start();
```

6.2. Interfaz Runnable

Ejemplo: declaramos la clase ***PrimerHiloR.java*** que implementa la interfaz **Runnable**, en el método **run()** se indica la funcionalidad del hilo, en este caso es pintar un mensaje y visualizar el identificador del hilo actualmente en ejecución. A continuación, creamos la clase ***UsaPrimerHiloR.java*** donde se lanzan varios hilos del tipo anterior de distintas formas.

6.3. Suspensión de un hilo

Usamos el método `sleep()` para detener un hilo un número de milisegundos. Realmente el hilo no se detiene, sino que se queda “dormido” el número de milisegundos que indiquemos. Lo podemos utilizar por ejemplo en ejercicios de contadores para ver cómo se va incrementando el contador.

6.3. Suspensión de un hilo

El método **suspend()** permite detener la actividad del hilo durante un intervalo de tiempo indeterminado. Este método es útil cuando se realizan applets con animaciones y en algún momento se decide para la animación para luego continuar cuando lo decida el usuario. Para volver a activar el hilo se usa el método **resume()**. El método **suspend()** y **resume()** están obsoletos.

6.3. Suspensión de un hilo

Para suspender de forma segura el hilo se debe introducir en el hilo una variable, por ejemplo suspender y comprobar su valor dentro del método **run()**, es lo que se hace en la llamada al método **suspender.esperandoParaReanudar()** del ejemplo siguiente. El método **Reanuda()** da el valor false para que detenga la suspensión y continúe ejecutándose el hilo:

```
public class MyHilo extends Thread {  
    private SolicitaSuspender suspender= new  
    SolicitaSuspender();  
  
    //Petidicion de suspender el hilo  
  
    public void Suspender() {  
        suspender.set(true);  
    }  
}
```

6.3. Suspensión de un hilo

```
//Petición de continuar

public void Reanuda() {
    suspender.set(false);
}

public void run() {
    try {
        //mientras hay trabajo por hacer (espera con un while
        suspender.esperandoParaReanudar(); //comprobar
    }catch(InterruptedException exception) {
        //muestro información de excepción
    }
}
}//Fin clase MiHilo
```

6.3. Suspensión de un hilo

```
public class SolicitaSuspender {  
    //atributos  
    private boolean suspender;  
  
    public synchronized void set (boolean b) {  
        suspender = b; //cambio de estado sobre el objeto  
        notifyAll();  
    }  
  
    public synchronized void esperandoParaReanudar() throws  
    InterruptedException{  
        while (suspender)  
            wait(); //suspender hilo hasta recibir notify() o notifyAll()  
    }  
}//Fin clase SolicitaSuspender
```

6.3. Suspensión de un hilo

Como se puede observar la variable en la clase **SolicitaSuspender** se envuelve. En esta clase con el método **set()** se le da valor true o false y llama al método **notifyAll()** que notifica a todos los hilos que esperan (han ejecutado un **wait()**) un cambio de estado sobre el objeto. En el método **esperandoParaReanudar()** se hace un **wait()** cuando el valor de la variable es true, el método **wait()** hace que el hilo espere hasta que le llegue un **notify()** o un **notifyAll()**.

El método **wait()** sólo puede ser llamado desde dentro de un método sincronizado (synchronized). Estos tres métodos: **wait()**, **notify()** y **notifyAll()** se usan en sincronización de hilos. Forman parte de la clase Object, y no parte de Thread. Se tratarán más adelante en el curso.

6.4. Parada de un hilo

- El método **stop()** detiene la ejecución de un hilo de forma permanente pero está en desuso. En lugar de usar este método se suele usar una variable como se hizo en el ejemplo **HiloEjemploDead.java**.
- El método **isAlive()** devuelve true si el hilo está vivo, es decir ha llamado a su método **run()** y aún no ha terminado su ejecución o no ha sido detenido con **stop()**. En caso contrario devuelve false.

6.4. Parada de un hilo

- El método ***interrupt()*** envía una petición de interrupción a un hilo. Si el hilo se encuentra bloqueado por una llamada a ***sleep()*** o ***wait()*** se lanza una excepción ***InterruptedException***. El método ***isInterrupted()*** devuelve true si el hilo ha sido interrumpido, en caso contrario devuelve false.

6.4. Parada de un hilo

Ejemplo: en el ejemplo **HiloEjemploInterrup.java** usa interrupciones para detener el hilo. En el método **run()** se comprueba en el bucle while si el hilo está interrumpido. Si no lo está se ejecuta el código. El método **interrumpir()** ejecuta el método **interrupt()** que lanza una interrupción que es recogida por el manejador (catch). Si en el ejemplo le quitamos el **sleep()** también hay que quitar la parte de **try-catch**, pues la interrupción será recogida por **isInterrupted()**, que será true con lo que la ejecución del hilo terminará ya que finaliza el método **run()**.

6.4. Parada de un hilo

Ejemplo: El método **join()** provoca que el hilo que hace la llamada espere la finalización de otros hilos.

Si en el hilo actual **h1.join()**, el hilo se queda en espera hasta que muera el hilo sobre el que se realiza el **join()**, en este caso **h1**. En este ejemplo vemos que el método **run()** de la clase **HiloJoin.java** visualiza en un bucle for un contador que empieza en **1** hasta un valor **n** que recibe el constructor del hilo.

En el método **main()** de la clase **EjemploJoin.java** se crean 3 hilos, cada uno de un valor diferente a la **n**, el primero el valor más pequeño y el tercero el valor más grande, parece lógico que por los valores del contador el primer hilo debe terminar el primero y el tercer hilo el último.

6.4. Parada de un hilo

Continuación **Ejemplo:** Llamando a ***join()*** podemos hacer que ***main()*** espere a la finalización de los hilos y cada hilo finalice en el orden marcado según la llamada a ***join()***, cuando salgan del bloque ***try-catch*** los tres hilos habrán finalizado y el texto FINAL DE PROGRAMA se visualizará al final. Si en el ejemplo quitamos los ***join()*** veremos que el texto FINAL DE PROGRAMA no se mostrará al final. El método ***join()*** puede lanzar la excepción ***InterruptedException***, por ello se incluye en un bloque ***try-catch***.

TEMA 4 – SINCORNIZACIÓN ENTRE HILOS

PARTE I

1. Comunicación y sincronización entre hilos

La sincronización nace de la necesidad de evitar que dos o más threads traten de acceder a los mismos recursos al mismo tiempo. Así, por ejemplo, si un thread tratara de escribir en un fichero, y otro thread estuviera al mismo tiempo tratando de borrar dicho fichero, se produciría una situación no deseada. Otra situación en la que hay que sincronizar threads se produce cuando un thread debe esperar a que estén preparados los datos que le debe suministrar el otro thread. Para solucionar estos tipos de problemas es importante poder sincronizar los distintos threads.

2. Variables locales al hilo

Cada thread tiene sus variables locales:

- Parámetros o argumentos de métodos.
- Variables creadas dentro de los métodos.
- En principio son privadas en Java.

Pero puede haber compartición si se pasan referencias:

- Objetos compartidos.
- Arrays compartidos.

3. Condiciones de carrera

“Una **condición de carrera** es un comportamiento del software en el cual la salida depende de un orden de ejecución de eventos que no se encuentran bajo control y que puede provocar resultados incorrectos”

- Una condición de carrera se puede producir cuando varios threads NO acceden en exclusión mutua a un recurso compartido.
- Se convierte en un fallo siempre que el orden de ejecución no sea el esperado.
- El nombre viene de la idea de dos procesos compiten en una carrera para acceder a un recurso compartido.

El comportamiento de un programa con una condición de carrera es imprevisible. El comportamiento depende de la ejecución simultánea de múltiples threads accediendo a recursos compartidos.

- Difíciles de detectar
- Difíciles de reproducir
- Difíciles de depurar

La solución es prevenirlas mediante una sincronización adecuada.

Las condiciones de carrera se producen cuando dos o más threads acceden a memoria compartida. En Java esto puede ocurrir de dos formas:

Con **variables estáticas**:

- Si el código de dos o más threads accede (lectura y escritura) a la misma variable estática.
- Si hay dos instancias de la misma clase thread que accede a una variable estática.

Con **referencias al mismo objeto** compartidas entre threads:

- En Java cualquier objeto o array se pasa por referencia.
- Si se comparte una referencia varios threads pueden estar accediendo al mismo objeto.

4. Sección crítica -> Bloques sincronizados y métodos sincronizados

Cuando un proceso accede a un recurso compartido, decimos que está accediendo a una **sección crítica**. Un mismo proceso puede acceder a más de una sección crítica.

La ejecución de la sección crítica debe ser en **exclusión mutua**:

- Únicamente un proceso/hilo puede estar en el código de la sección crítica (SC).
- Es necesario un protocolo para controlar el acceso a la SC
 - Es necesario algún mecanismo para pedir permiso para entrar en la sección crítica.
 - También es necesario avisar de que hemos salido para que otros puedan entrar.

De este modo unas instrucciones se aseguran que son tratadas como una operación indivisible, una **operación atómica**. Así se permite el acceso seguro a una sección crítica.

La sincronización es un proceso que lleva bastante tiempo a la CPU, por tanto, se debe minimizar su uso, ya que el programa será más lento cuanta más sincronización incorpore. Podemos sincronizar bloques de código o directamente sobre métodos.

4.2. Bloques sincronizados

Ejemplo 01: veamos un ejemplo donde se comparte un objeto de la clase *Contador.java* con dos hilos. Un hilo incrementa el contador en una unidad y el otro hilo lo decrementa en una unidad.

Los valores varían de una ejecución a otra ya que no controlamos que la ejecución sea atómica. Es decir, hay que asegurarse que mientras hacemos la suma nadie hace la resta, y viceversa.

Para resolver este problema utilizamos “**synchronized**” en la parte del código que queremos que se ejecute de forma atómica. Es decir, Java marcará esa parte de código como una **región crítica**.

El formato sería el siguiente:

```
synchronized (Object){  
    //sentencias críticas  
}
```

Es decir, estamos bloqueando el objeto “contador” de forma que el hilo que intenta acceder a un objeto bloqueado queda suspendido y queda a la espera a que se quede libre. El objeto quedará libre cuando se termine la ejecución, se ejecute un *return* o bien se dispare una excepción.

Ejercicio 01: modificar el anterior programa para que se bloquee el objeto contador y comprobar los resultados.

4.3. Métodos sincronizados

Se debe evitar la sincronización de bloques de código y sustituirlas siempre que sea posible por la sincronización de métodos, es decir, realizar **exclusión mutua** de los procesos respecto a la variable compartida.

Ejemplo 02: dos personas comparten una cuenta y pueden sacar dinero de ella en cualquier momento. Antes de retirar dinero se comprueba siempre si existe saldo. La cuenta tiene 50€, una de las personas quiere retirar 40 y la otra 30. La primera llega al cajero, revisa el saldo, comprueba que hay dinero y se prepara para retirar el dinero, pero antes de retirarlo llega la otra persona a otro cajero, comprueba el saldo que todavía muestra los 50€ y también se dispone a retirar el dinero. Las dos personas retiran el dinero, pero entonces el saldo actual sería ahora de -20€. Para sincronizar un método, añadimos la palabra *synchronized* a su declaración.

El uso de métodos sincronizados implica que no es posible invocar dos métodos sincronizados del mismo objeto a la vez. Cuando un hilo está ejecutando un método sincronizado de un objeto, los demás hilos que invoquen a métodos sincronizados para el mismo objeto se bloquean hasta que el primer hilo termine con la ejecución del método.

Veamos los métodos que podemos usar para el **bloqueo de hilos**:

- **synchronized** en métodos ya que la variable crítica debe estar protegida.
- Para mantener la sincronización además el hilo que está activo debe avisar al resto para que no lo utilice y se esperen y viceversa. Veamos los siguientes métodos de la clase Object:
 - **wait()**: un hilo que llama a wait() de un objeto queda suspendido hasta que otro hilo llame al método notify() o notifyAll() del mismo objeto.
 - **notify()**: despierta sólo a uno de los hilos que realizó una llamada wait() sobre el mismo objeto notificando que hubo un cambio de estado sobre el objeto. Si son varios los hilos que esperan sólo uno será el elegido para despertarse (elección arbitraria).
 - **notifyAll()**: despierta a todos los hilos que están esperando el objeto.

Los métodos wait(), notify() y notifyAll() deben ejecutarse en exclusión mutua (en métodos o bloques sincronizados). En caso contrario lanzan IllegalMonitorStateException.

Con este tipo de estructuras seguras se puede simular lo que se conoce como monitores:

[https://es.wikipedia.org/wiki/Monitor_\(concurrency\)](https://es.wikipedia.org/wiki/Monitor_(concurrency))

5. Escenarios clásicos de sincronización

5.1. Productores y consumidores - synchronized

Qué pasa cuando un hilo productor produce datos más rápido que otro hilo consumidor. El consumidor en este caso se tendría que saltar algún dato para poder llevar el mismo ritmo que el productor. Si fuera al revés también podría ocurrir que el consumidor recogiera el mismo dato o se detuviera.

El ejemplo típico es un hilo que escribe datos en un archivo y otro proceso hilo lee los datos de ese archivo. El fichero sería el recurso compartido y deberán sincronizarse para hacer bien su tarea.

Ejemplo 03: productor-consumidor.

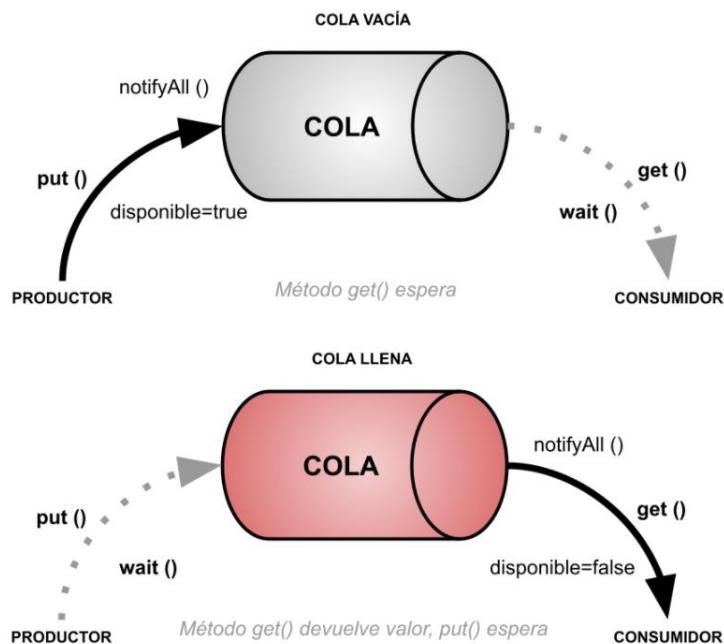
Definimos 3 clases: **Cola** (objeto compartido) **Productor** y **Consumidor**.

El productor produce simplemente números y los coloca en la cola (recurso compartido) para que sean consumidos por el consumidor.

En primer lugar necesitamos una cola que permita encolar y desencolar elementos de una manera segura.

Sabemos que antes de encolar hay que comprobar si la cola está llena. También sabemos que antes de desencolar hay que comprobar si la cola está vacía.

Así, necesitaremos hacer operaciones encolar y desencolar que funcionen de manera atómica y nos avisen de si consiguen hacer la operación o no.



5.2. Lectores y escritores- synchronized

El problema de los lectores y escritores presenta el siguiente enunciado:

- Los lectores quieren leer el libro.
- Los escritores quieren escribir en el libro.
- Puede haber varios lectores leyendo el libro.
- Habrá un escritor sólo a la vez escribiendo el libro.

Veamos cómo se puede resolver este problema sincronizando nuestro recurso compartido, el “libro”. Tenemos las siguientes clases:

- Clase *Lector* que extiende de hilo. Sigue la solicitud de leer.
- Clase *Escritor* que extiende de hilo. Sigue la solicitud de escribir.
- Clase *Recurso* que es donde haremos las sincronizaciones
- Clase *LyESimple* que nos lanzará un pequeño programa con un escritor y dos lectores.

1. SEMÁFOROS

Los semáforos son una herramienta para resolver problemas tanto de exclusión mutua como de sincronización.

- Exclusión mutua: sólo uno de los procesos/hilos debe estar en la sección crítica en un instante dado.
- Sincronización: un proceso/hilo debe esperar la ocurrencia de un evento para poder seguir ejecutándose.

Un semáforo es un objeto que lleva la cuenta de un cierto número de permisos. Hay dos operaciones básicas:

acquire(n): bloquea a la thread llamante hasta que hay tantos permisos disponibles como se solicitan. En ese momento, se descuentan los permisos solicitados y la thread se desbloquea.

release(n): devuelve los permisos indicados. Como consecuencia de esta devolución, puede que algún que otro thread bloqueado en un acquire() pueda proseguir.

Los semáforos más sencillos son binarios. Para entrar en una zona crítica un *thread* debe adquirir el derecho de acceso, y al salir lo libera.

```
Semaphore semaphore = new Semaphore(1);

semaphore.acquire();

// zona crítica

semaphore.release();
```

Por solidez, nunca debemos olvidar liberar un semáforo al salir de la zona crítica. El código previo lo hace si salimos normalmente; pero si se sale de forma abrupta (*return* o excepción) entonces el semáforo no se liberaría. Es por ello, que normalmente se sigue este patrón:

```
try {

    semaphore.acquire();

    // zona crítica

} finally {

    semaphore.release();

}
```

Más general, el semáforo puede llevar cuenta de N permisos. Los *threads* solicitan algunos permisos; si los hay, los retiran y siguen; si no los hay, quedan esperando a que los haya. Cuando ha terminado, el *thread* devuelve los permisos.

Ejemplo 01: tenemos un contador compartido por varios hilos. Para sincronizarlos utilizaremos un semáforo antes y después de recuperar la información del contador e incrementarla. (Exclusión mutua).

Ejemplo02: Un banco intenta hacer ingresos y retiradas de una misma cuenta. Utiliza semáforos para sincronizarlos.

2. BARRERAS

Para la sincronización entre hilos, imagina que queremos que 4 personas que están situadas en diferentes ciudades realicen un viaje, que vayan a un punto B tras haberse encontrado los 4 en un punto A primeramente, lo más probable es que alguno llegue antes que otro, los que ya hayan llegado tendrán que esperar en un punto a que lleguen los restantes, usaremos un tipo de dato llamado barrera que se encargará de controlar la espera en el punto A, definimos el comportamiento de la barrera se obtiene a partir del número de personas a esperar(bloqueos) y los siguientes métodos:

- **Esperar/await()** : Si el número de bloqueos disponibles es superior a 0 entonces la persona(hilo) consumirá un bloqueo y decrementará el número disponible además de esperar a que el número de bloqueos llegue a 0.
- **Reset()** : Si alguien usa este método la barrera volverá a tener el número de permisos original, este método no se suele utilizar ya que la implementación de Java lo empleará de forma implícita cuando el número de permisos llegue a 0 y haya dejado pasar a los a hilos bloqueados.

Para el uso de barreras utilizaremos la clase **CyclicBarrier** que trae implementada Java, veamos la situación anterior en el [ejemplo3](#).

3. Problemas en la sincronización: inanición e interbloqueos

Los mecanismos de sincronización deben utilizarse de manera conveniente para evitar **interbloqueos**. Los interbloqueos se clasifican en:

- **Deadlock**: un hilo A está a la espera de que un hilo B lo desbloquee, al tiempo que este hilo B está también bloqueado a la espera de que A lo desbloquee.
- **Livelock**: similar al Deadlock, pero en esta situación A responde a una acción de B, y a causa de esta respuesta B responde con una acción a A, y así sucesivamente. Se ocupa toda la CPU produciendo un bloqueo de acciones continuas.
- **Stravation (Inanición)**: Un hilo necesita consumir recursos, o bien ocupar CPU, pero se lo impide la existencia de otros hilos "hambrientos" que operan más de la cuenta sobre dichos recursos. Se impide el funcionamiento fluido del hilo o incluso da la impresión de bloqueo.

PROGRAMACION CONCURRENTE Y DISTRIBUIDA

III.3 Conurrencia con Java: Sincronización



J.M. Drake
L.Barros

Notas:

Recursos de Java para sincronizar threads

- # Todos los objetos tienen un bloqueo asociado, lock o cerrojo, que puede ser adquirido y liberado mediante el uso de métodos y sentencias synchronized.
- # La sincronización fuerza a que la ejecución de los dos hilos sea mutuamente exclusiva en el tiempo.
- # Mecanismos de bloqueo:
 - Métodos synchronized ([exclusión mutua](#)).
 - Bloques synchronized ([regiones críticas](#)).
- # Mecanismos de comunicación de los threads ([variables de condición](#)):
 - Wait(), notify(),notifyAll()...
- # Cualquier otro mecanismo:
 - Dado que con monitores se pueden implementar los restantes mecanismos de sincronización (Semáforos, Comunicación síncrona, Invocación de procedimientos remotos, etc.) se pueden encapsular estos elementos en clases y basar la concurrencia en ellos.

Lock asociado a los componentes Java.

- Cada **objeto** derivado de la clase object (esto es, prácticamente todos) tienen asociado un elemento de sincronización o lock intrínseco, que afecta a la ejecución de los métodos definidos como synchronized en el objeto:
 - Cuando un objeto ejecuta un método synchronized, toma el lock, y cuando termina de ejecutarlo lo libera.
 - Cuando un thread tiene tomado el lock, ningún otro thread puede ejecutar ningún otro método synchronized del mismo objeto.
 - El thread que ejecuta un método synchronized de un objeto cuyo lock se encuentra tomado, se suspende hasta que el objeto es liberado y se le concede el acceso.
- Cada **clase Java** derivada de Object, tiene también un mecanismo lock asociado a ella (que es independiente del asociado a los objetos de esa clase) y que afecta a los procedimientos estáticos declarados synchronized.

Bloques synchronized.

- Es el mecanismo mediante el que se implementan en Java las regiones críticas.
- Un bloque de código puede ser definido como synchronized respecto de un objeto. En ese caso solo se ejecuta si se obtiene el lock asociado al objeto

```
synchronized (object){  
    Bloque de estamentos  
}
```

- Se suele utilizar cuando se necesita utilizar en un entorno concurrente, un objeto diseñado para un entorno secuencial.

Este mecanismo presenta el inconveniente de las Regiones críticas, y es que los diferentes bloques que interaccionan a través de una región crítica resultan dispersos por múltiples módulos de la aplicación, y ello hace que su mantenimiento sea muy complejo y delicado.

Observese que en Java no hay Regiones Críticas Condicionales, por lo que no se pueden resolver con este mecanismo todos los problemas.

Ejemplo de utilización de bloque synchronized

```
// Hace que todos los elementos del array sean no negativos
public static void abs(int[] valores){
    synchronized (valores){
        // Sección crítica
        for (int i=0; i<valores.length; i++){
            if (valores[i]<0) valores[i]= -valores[i];
        }
    }
}
```

Implementación de Monitores: Métodos synchronized

- ▣ Los métodos de una clase Java se pueden declarar como **synchronized**. Esto significa que se garantiza que se ejecutará con régimen de exclusión mutua respecto de otro método del mismo objeto que también sea synchronized.
- ▣ Cuando un thread invoca un método synchronized, trata de tomar el lock del objeto a que pertenezca. Si está libre, lo toma y se ejecuta. Si el lock está tomado por otro thread, se suspende el que invoca hasta que aquel finalice y libere el lock.
- ▣ Si el método synchronized es estático (static), el lock al que hace referencia es de clase y no de objeto, por lo que se hace en exclusión mútua con cualquier otro método estático synchronized de la misma clase.

Ejemplo de método synchronized.

```
class CuentaBancaria{  
    private class Deposito{  
        protected double cantidad;  
        protected String moneda =“Euro”  
    }  
    Deposito elDeposito;  
    public CuentaBancaria(double initialDeposito,String moneda){  
        elDeposito.cantidad= initialDeposito;  
        elDeposito.moneda=moneda; }  
    public synchronized double saldo(){return elDeposito.cantidad;}  
    public synchronized void ingresa(double cantidad){  
        elDeposito.cantidad=elDeposito.cantidad + cantidad;  
    }  
}
```

Exclusión mutua

Thread 2->lock tomado->Espera

```
public synchronized double saldo(){  
    return elDeposito.cantidad;  
}
```

Thread 1->Toma el lock->Ejecuta

```
public synchronized void ingresa(double cantidad){  
    elDeposito.cantidad=elDeposito.cantidad + cantidad;  
}
```

Notas:

Consideraciones sobre métodos synchronized.

- El lock es tomado por el thread, por lo que mientras un thread tiene tomado el lock de un objeto puede acceder a otro método synchronized del mismo objeto.
- El lock es por cada instancia del objeto.
- Los métodos de clase (static) también pueden ser synchronized. Por cada clase hay un lock y es relativo a todos los métodos synchronized de la clase. Este lock no afecta a los accesos a los métodos synchronized de los objetos que son instancia de la clase.
- Cuando una clase se extiende y un método se sobreescribe, este se puede definir como synchronized o no, con independencia de cómo era y como sigue siendo el método de la clase madre.

Métodos de Object para sincronización.

Todos son métodos de la clase object. **Solo se pueden invocar por el thread propietario del lock (p.e. dentro de métodos synchronized)**. En caso contrario lanzan la excepción **IllegalMonitorStateException**

public final void wait() throws InteruptedException

Espera indefinida hasta que reciba una notificación.

public final void wait(long timeout) throws InteruptedException

El thread que ejecuta el método se suspende hasta que, o bien recibe una notificación, o bien transcurre el timeout establecido en el argumento. `wait(0)` representa una espera indefinida hasta que llegue la notificación.

public final wait(long timeout, int nanos)

Wait en el que el tiempo de timeout es $1000000 * \text{timeout} + \text{nanos}$ nanosegundos

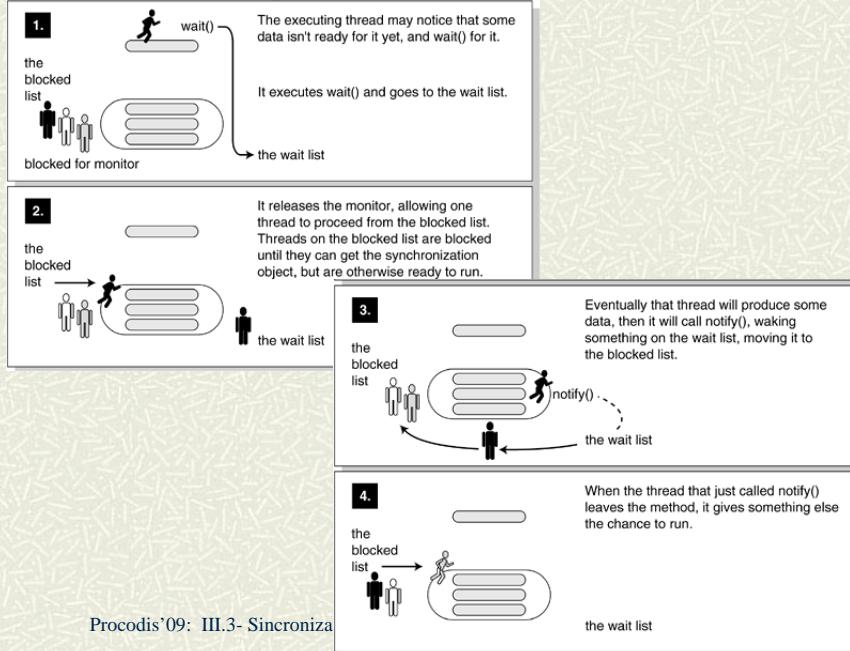
public final void notify()

Notifica al objeto un cambio de estado, esta notificación es transferida a solo uno de los threads que esperan (han ejecutado un `wait`) sobre el objeto. No se puede especificar a cual de los objetos que esperan en el objeto será despertado.

public final void notifyAll()

Notifica a todos los threads que esperan (han ejecutado un `wait`) sobre el objeto.

Protocolo Wait/Notify



11

Notas:

It's not enough just to say, "Don't run while I am running." We need the threads to be able to say, "OK, I have some data ready for you," and to suspend themselves if there isn't data ready.

Uso de los métodos wait().

- Debe utilizarse siempre dentro de un método synchronized y habitualmente dentro de un ciclo indefinido que verifica la condición:

```
synchronized void HazSiCondicion(){  
    while (!Condicion) wait();  
    ..... // Hace lo que haya que hacer si la condición es cierta.  
}
```

- Cuando se suspende el thread en el wait, libera el lock que poseía sobre el objeto. La suspensión del thread y la liberación del lock son atómicos (nada puede ocurrir entre ellos).
- Cuando el thread es despertado como consecuencia de una notificación, la activación del thread y la toma del lock del objeto son también atómicos (Nada puede ocurrir entre ellos).

Uso de los métodos notify().

- # Debe utilizarse siempre dentro de un método synchronized:

```
synchronized void changeCondition(){  
..... // Se cambia algo que puede hacer que la condición se satisfaga.  
notifyAll();  
}
```

- # Muchos thread pueden estar esperando sobre el objeto:

- Si se utiliza notify() solo un thread (no se sabe cual) es despertado.
- Si se utiliza notifyAll() todos los thread son despertados y cada uno decide si la notificación le afecta, o si no, vuelve a ejecutar el wait() (dentro del while).

- # El proceso suspendido debe esperar hasta que el procedimiento que invoca notify() o notifyAll ha liberado el lock del objeto.

Ejemplo de sincronización: Establecimiento de una variable.

```
Thread 1  
public synchronized guardedJoy() {  
    //This guard only loops once for each  
    //special event, which may not  
    //be the event we're waiting for.  
  
    while(!joy) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    System.out.println("Joy and  
        efficiency have been achieved!");  
}  
  
Thread 2  
public synchronized notifyJoy() {  
    joy = true; notifyAll();  
}
```

Ejemplo Monitor: Buffer de capacidad finita

```
public interface Buffer {  
    public void put(Object obj)  
        throws InterruptedException;  
    public Object get()  
        throws InterruptedException;  
}  
  
class FixedBuffer implements Buffer{  
    Object[] buf;  
    int in = 0;  
    int out = 0;  
    int count= 0;  
    int size;  
    public FixedBuffer(int size){  
        this.size= size;  
        buf = new Object[size];  
    }  
  
    public synchronized void put(Object obj)  
        throws InterruptedException{  
        while (count == size) wait();  
        buf[in]=obj;  
        count= count +1;  
        in=(in+1)%size;  
        notifyAll();  
    }  
    public synchronized Object get()  
        throws InterruptedException{  
        while (count == 0) wait();  
        Object obj= buf[out];  
        buf[out]=null;  
        count= count -1;  
        out= (out+1) % size;  
        notifyAll();  
        return (obj);  
    }  
}
```

Semáforo en Java

```
public class Semaphore{  
    int value;  
  
    public Semaphore(int initialValue){  
        value = initialValue;  
    }  
    synchronized public void signal(){  
        value= value+1;  
        notify();  
    }  
    synchronized public void await() throws InterruptedException{  
        while (value == 0) wait();  
        value = value - 1;  
    }  
}
```

Buffer de capacidad finita basado en semáforos.

```
public interface Buffer {  
    public void put(Object obj)  
        throws InterruptedException;  
    public Object get() throws  
        InterruptedException;  
}  
  
class FixedSemaphoreBuffer implements Buffer{  
    protected Object[] buf;  
    protected int in = 0;  
    protected int out = 0;  
    protected int count= 0;  
    protected int size;  
  
    Semaphore full = new Semaphore(0);  
    Semaphore empty;  
    FixedSemaphoreBuffer(int size){  
        this.size= size;  
        buf = new Object[size];  
        empty = new Semaphore(size);  
    }  
  
    public void put(Object obj)  
        throws InterruptedException{  
        empty.Wait();  
        synchronized (this) {  
            buf[in]=obj; count= count +1;  
            in=(in+1)%size;  
        }  
        full.Signal();  
    }  
  
    public Object get() throws InterruptedException{  
        full.Wait();  
        synchronized (this) {  
            Object obj= buf[out]; buf[out]=null;  
            count= count -1;  
            out= (out+1) % size;  
        }  
        empty.Signal();  
        return (obj);  
    }  
}
```

Ejemplo de monitor: Lectores y escritores.

```
public class ReadWriteController {  
    Recurso elRecurso;  
  
    int writerWaiting= 0;  
    int readerInside= 0;  
    int writerInside= 0;  
  
    public Recurso get(){  
        goInReader();//obtiene acceso  
        return (elRecurso);  
        goOutReader();//libera recurso  
    }  
  
    public void write(Recurso newValue){  
        goInWriter();//obtiene acceso  
        elRecurso=newValue;  
        goOutWriter();//libera recurso  
    }  
  
    public synchronized void goInReader(){  
        try{  
            while ((writerWaiting+writerInside) != 0) wait();  
        }catch (InterruptedException e){};  
        readerInside=readerInside+1;  
    }  
  
    public synchronized void goOutReader(){  
        readerInside=readerInside-1;  
        notifyAll();}  
    public synchronized void goInWriter(){  
        writerWaiting=writerWaiting+1;  
        try{  
            while((writerInside+readerInside)!=0) wait();  
        }catch (InterruptedException e){};  
        writerWaiting=writerWaiting-1;  
        writerInside=writerInside+1; }  
    public synchronized void goOutWriter(){  
        writerInside=writerInside-1;  
        notifyAll();}  
    }
```

Procodis'09: III.3- Sincronización de thread Java

José M.Drake

18

Problema de Lectores y Escritores:

Se desea diseñar un monitor que permita el acceso a un recurso compartido en dos modos, como lector y como escritor.

Se puede permitir el acceso concurrente al recurso como lector, pero se ha de evitar que cuando acceda un escritor, pueda concurrir con él, otro escritor a lector.

Se utiliza el protocolo que establece que cuando un escritor espera para acceder, no se permita el acceso de un nuevo cliente hasta que la sala esté vacía, esto es hayan salido todos los lectores (si los hubiera).

Ejercicio I: ¿Qué ocurrirá si lo ejecutamos?

```
public class WaNot{
    int i=0;
    public static void main(String argv[]){
        WaNot w = new WaNot();
        w.amethod();
    }

    public void amethod(){
        while(true){
            try{
                wait();
            }catch (InterruptedException e) {}
            i++;
       }//End of while

    }//End of amethod
}//End of class
```

Procodis'09: III.3- Sincronización de thread Java

José M.Drake

19

•e

Notas:

Exception in thread "main" java.lang.IllegalMonitorStateException

Ejercicio II: ¿Cómo comunicar dos threads?

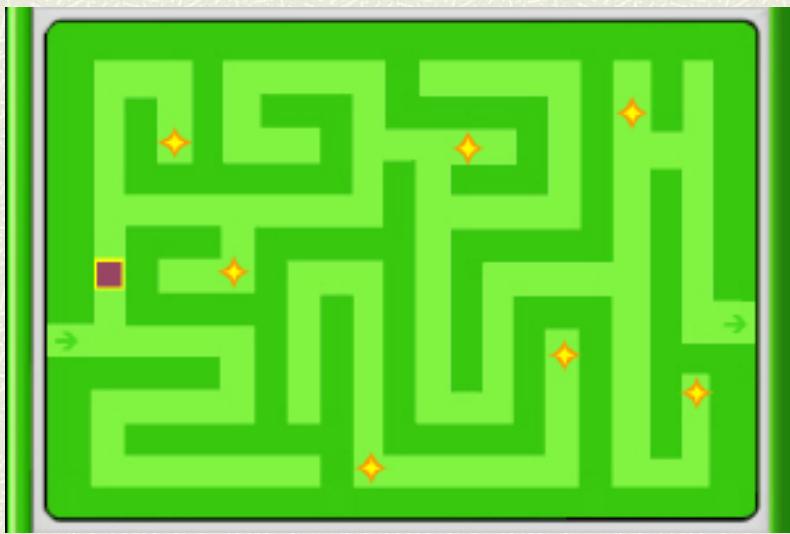
Thread A está esperando que el Thread B le envíe un mensaje:

```
// Thread A  
public void waitForMessage() {  
    while (hasMessage == false) {  
        Thread.sleep(100);  
    }  
}  
  
// Thread B  
public void setMessage(String message) {  
    ...  
    hasMessage = true;  
}
```

```
// Thread A  
public synchronized void waitForMessage() {  
    try {  
        wait();  
    } catch (InterruptedException ex) {}  
}  
  
// Thread B  
public synchronized void setMessage(String message) {  
    ...  
    notify();  
}
```

Notas:

Maze game I



Procodis'09: III.3- Sincronización de thread Java

José M.Drake

21

Notas:

Maze game II. Código sin sincronización

```
public class Maze {  
    private int playerX;  
    private int playerY;  
    public boolean isAtExit() {  
        return (playerX == 0 &&  
               playerY == 0);  
    }  
    public void setPosition(int x, int y)  
    {  
        playerX = x;  
        playerY = y;  
    }  
}
```

Player moves from (1,0) to (0,1):

- # 1. Starting off, the object's variables are playerX = 1 and playerY = 0.
- # 2. Thread A calls setPosition(0,1).
- # 3. The line playerX = x; is executed. Now playerX = 0.
- # 4. Thread A is pre-empted by Thread B.
- # 5. Thread B calls isAtExit().
- # 6. Currently, playerX = 0 and playerY = 0, so isAtExit() returns true!

Notas:

Most of the time, this code works fine. But keep in mind that threads can be pre-empted at any time. Imagine this scenario, in which the player moves from (1,0) to (0,1):

1. Starting off, the object's variables are playerX = 1 and playerY = 0.
2. Thread A calls setPosition(0,1).
3. The line playerX = x; is executed. Now playerX = 0.
4. Thread A is pre-empted by Thread B.
5. Thread B calls isAtExit().
6. Currently, playerX = 0 and playerY = 0, so isAtExit() returns true!

In this scenario, the player is reported as solving the maze when it's not the case. To fix this, you need to make sure the setPosition() and isAtExit() methods can't execute at the same time.

Maze game III. Soluciones de sincronización

```
public class Maze {  
    private int playerX;  
    private int playerY;  
    public synchronized boolean  
    isAtExit() {  
        return (playerX == 0 &&  
               playerY == 0);  
    }  
    public synchronized void  
    setPosition(int x, int y) {  
        playerX = x;  
        playerY = y;  
    }  
}
```

→

```
public void setPosition(int x, int y) {  
    synchronized(this) {  
        playerX = x;  
        playerY = y;  
    }  
}
```

Notas:

The only exception is that the second example has some extra bytecode instructions.

Object synchronization is useful when you need more than one lock, when you need to acquire a lock on something other than this, or when you don't need to synchronize an entire method.

A lock can be any object, even arrays—basically, anything except primitive types. If you need to roll your own lock, just create a plain Object:

```
Object myLock = new Object();  
...  
synchronized (myLock) {  
    ...  
}
```

Cuando sí/no sincronizar

Cuándo sincronizar

En cualquier momento en el que dos o más threads acceden al mismo objeto o campo.

Cuándo no sincronizar

- “Sobresincronizar” causa retrasos innecesarios cuando dos o más threads tratan de ejecutar el mismo bloque de código .Por ejemplo, no sincronizar el método entero, cuando sólo una parte necesita ser sincronizada. Poner sólo un bloque sincronizado en esa parte del código:

```
public void myMethod() {  
    synchronized(this) {  
        // code that needs to be synchronized  
    }  
    // code that is already thread-safe  
}
```

- No sincronizar un método que usa variables locales. Las variables locales son almacenadas en el stack, y cada thread tiene su propio stack, así que, no habrá problemas de concurrencia:

```
public int square(int n) {  
    int s = n * n;  
    return s;  
}
```

Notas: