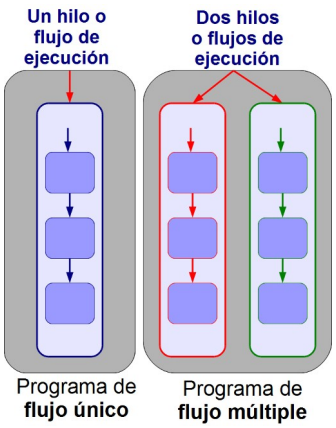
Programación de Servicios y Procesos

U2: Programación Multihilo

1. Introducción

Seguro que en más de una ocasión mientras te descargabas una imagen desde tu navegador web, seguías navegando por Internet e incluso iniciabas la descarga de un nuevo archivo, y todo esto ejecutándose el navegador como un único proceso, es decir, teniendo un único ejemplar del programa en ejecución.



Pues bien, ¿Cómo es capaz de hacer el navegador web varias tareas a la vez? Seguro que estarás pensando en la **programación concurrente**, y así es, pero con un nuevo enfoque de la concurrencia, denominado "**programación multihilo**". Justo lo que vamos a estudiar en esta unidad.

Los programas realizan actividades o tareas, y para ello pueden seguir uno o más flujos de ejecución. Dependiendo del número de flujos de ejecución, podemos hablar de dos tipos de programas:

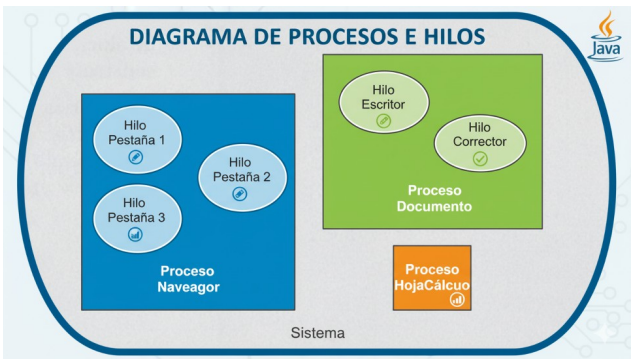
- **Programa de flujo único.** Es aquel que realiza las actividades o tareas que lleva a cabo una a continuación de la otra, de manera **secuencial**, lo que significa que cada una de ellas debe concluir por completo, antes de que pueda iniciarse la siguiente.

- **Programa de flujo múltiple.** Es aquel que coloca las actividades a realizar en diferentes flujos de ejecución, de manera que cada uno de ellos se inicia y termina por separado, pudiéndose ejecutar éstos de manera concurrente.

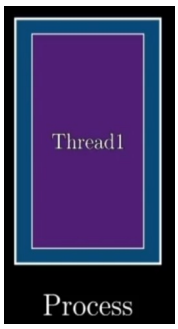
La **programación multihilo** o **multithrearing** consiste en desarrollar programas o aplicaciones de flujo múltiple. Cada uno de esos flujos de ejecución es un **thread** o **hilo**.

En el ejemplo anterior sobre el navegador web, un hilo se encargaría de la descarga de la imagen, otro de continuar navegando y otro de iniciar una nueva descarga. La utilidad de la programación multihilo resulta evidente en este tipo de aplicaciones. El navegador puede realizar "a la vez" estas tareas, por lo que no habrá que esperar a que finalice una descarga para comenzar otra o seguir navegando.

Cuando decimos "a la vez" recuerda que nos referimos a que las tareas se realizan concurrentemente, pues el que las tareas se ejecuten realmente en paralelo dependerá del Sistema Operativo y del número de procesadores del sistema donde se ejecute la aplicación. En realidad, esto es transparente para el programador y usuario, lo importante es la sensación real de que el programa realiza de forma simultánea diferentes tareas.



2. Conceptos sobre hilos

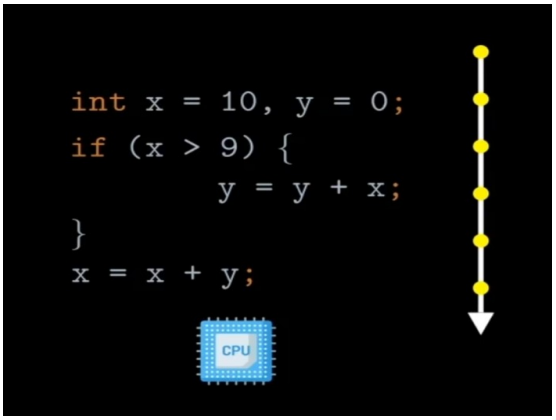


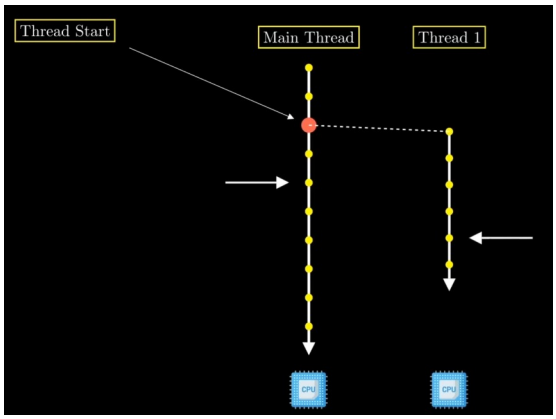
Pero ¿qué es realmente un hilo? Un **hilo** es un flujo de control secuencial independiente dentro de un proceso y está asociado con una secuencia de instrucciones, un conjunto de registros y una pila.

Cuando se ejecuta un programa, el Sistema Operativo crea un proceso y también crea su primer hilo, **hilo primario**, el cual puede a su vez crear hilos adicionales.

Desde este punto de vista, un proceso no se ejecuta, sino que solo es el espacio de direcciones donde reside el código que es ejecutado mediante uno o más hilos.

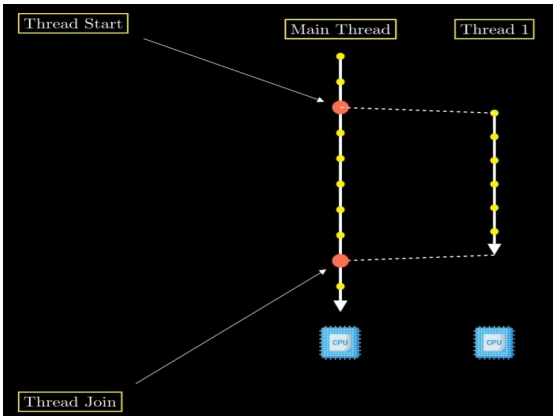
Flujo único	Flujo múltiple: Dos hilos independientes

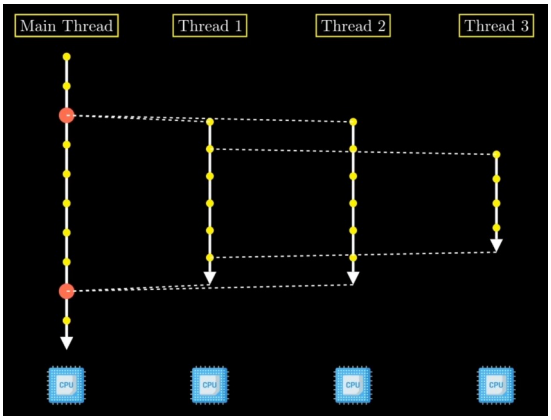




Cuando un programa Java se lanza (se convierte en un proceso) empieza a ejecutarse por su método ***main()*** que lo ejecuta el thread principal (**Main Thread**), un hilo especial creado por la JVM para ejecutar la aplicación.

Flujo múltiple: Dos hilos dependientes	Flujo múltiple: Cuatro hilos dependientes

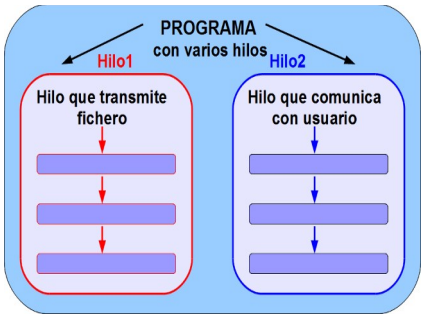




Desde un proceso se pueden crear e iniciar tantos threads como sean necesarios. Estos hilos ejecutarán partes del código de la aplicación en paralelo con el thread principal.

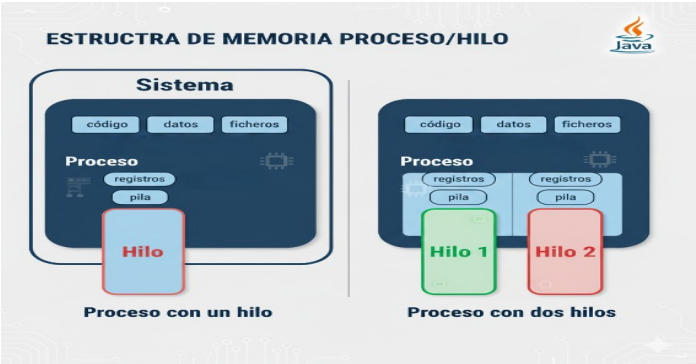
Por lo tanto podemos hacer las siguientes **observaciones**:

- Un hilo no puede existir independientemente de un proceso. Por tanto, si el proceso finaliza por alguna circunstancia, todos sus hilos también lo hacen.
- Un hilo no puede ejecutarse por si 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. Es la habilidad de ejecutar varios hilos dentro de un proceso lo que ofrece algo nuevo y útil ya que cada uno de estos hilos puede ejecutar actividades diferentes al mismo tiempo. Así en un programa un hilo puede encargarse de la comunicación con el usuario, mientras que otro hilo transmite un fichero, otro puede acceder a recursos del sistema (cargar sonidos, leer ficheros, …), etc.

2.1. Recursos compartidos por los hilos

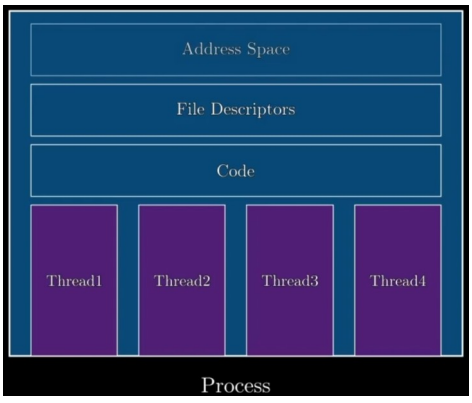


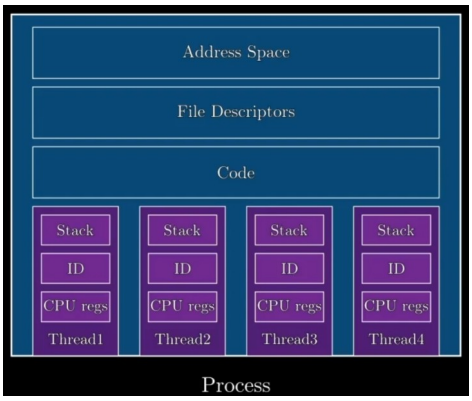
Un hilo lleva asociados los siguientes elementos:

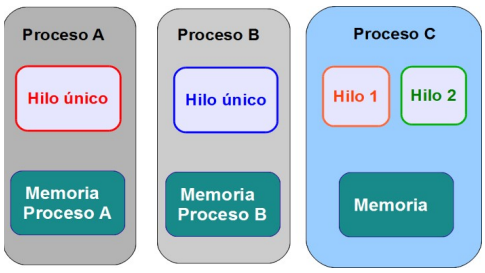
- Un identificador único que lo identifica dentro del proceso.
- Un contador de programa propio.
- Un conjunto de registros de la CPU.
- Una pila (usada para almacenar variables locales y parámetros de métodos).

Por otra parte, **un hilo puede compartir con otros hilos del mismo proceso los siguientes recursos:**

- Código.
- Espacio de direcciones usado por el proceso.
- Datos (como variables globales).
- Otros recursos del sistema operativo, como los ficheros abiertos.

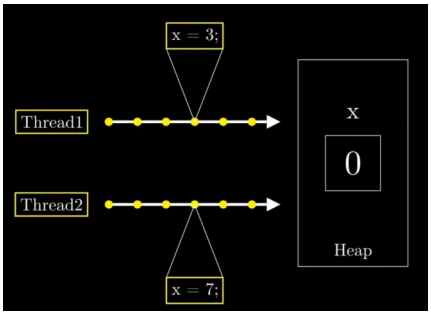






Seguro que te estarás preguntando "si los hilos de un proceso comparten el mismo espacio de memoria, ¿qué pasa si uno de ellos la corrompe?" La respuesta es que los otros hilos también sufrirán las consecuencias. Recuerda que en el caso de procesos, el sistema operativo normalmente protege a un proceso de otro y si un proceso corrompe su espacio de memoria los demás no se verán afectados

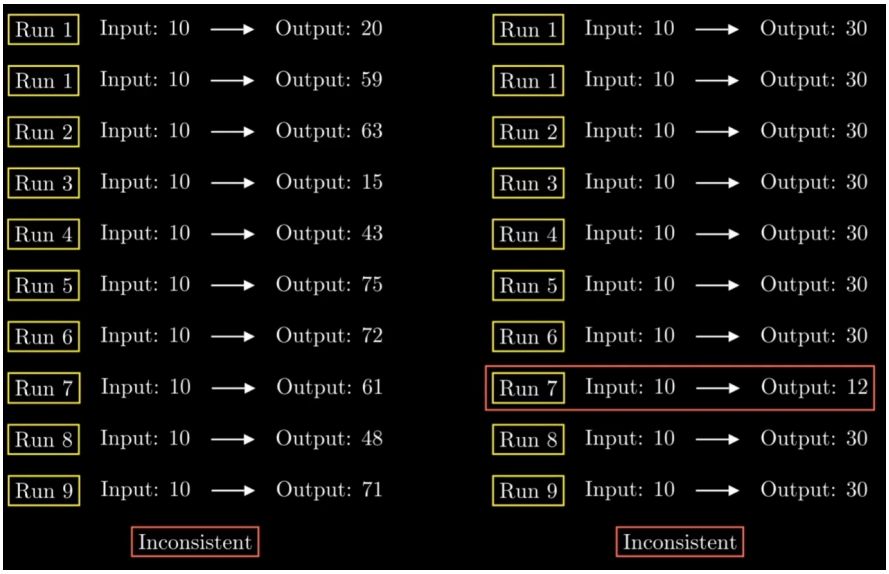
El hecho de que los hilos compartan recursos (por ejemplo, pudiendo acceder a las mismas variables) implica que sea necesario **utilizar esquemas de bloqueo y sincronización,** lo que puede hacer más difícil el desarrollo de los programas y así como su depuración.



Por ejemplo, si dos hilos quieren, en el mismo instante de tiempo, actualizar con diferentes valores una variable x compartida, es necesario garantizar el orden de la operación para tener un **comportamiento consistente**. De lo contrario se tendría un resultado impredecible, algo que debe evitarse cuando se implementan aplicaciones multihilo ya que provocará que se tengan diferentes salidas en diferentes ejecuciones.

COMPORTAMIENTO INCONSISTENTE

COMPORTAMIENTO INDETERMINISTA

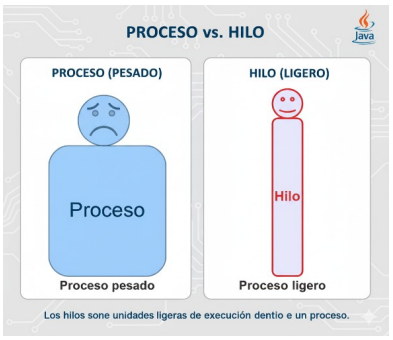


Realmente, es en la sincronización de hilos donde reside el arte de programar con hilos ya que de no hacerlo bien, podemos crear una aplicación totalmente ineficiente o inútil, como por ejemplo, programas que tardan horas en procesar servicios, que se bloquean con facilidad o que intercambian datos de manera equivocada.

Profundizaremos más adelante en la sincronización, comunicación y compartición de recursos entre hilos dentro del contexto de Java.

Pero ¿qué ventajas aportan los hilos y cuando deben utilizarse?

2.2. Ventajas y usos 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 o **cambio de contexto** es bastante más rápida que entre procesos.

Es por esas razones, por lo que a los hilos se les denomina también **procesos ligeros**.

Y ¿**cuándo se aconseja utilizar hilos**? 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.

Por ejemplo, imagina la siguiente situación:

Debes crear una aplicación que se ejecutará en un servidor para atender peticiones de clientes. Esta aplicación podría ser un servidor de bases de datos, o un servidor web.

- Cuando se ejecuta el programa éste abre su puerto y queda a la escucha, esperando recibir peticiones.
- Si cuando recibe una petición de un cliente se pone a procesarla para obtener una respuesta y devolverla, cualquier petición que reciba mientras tanto no podrá atenderla, puesto que está ocupado.

La solución será construir la aplicación con múltiples hilos de ejecución.

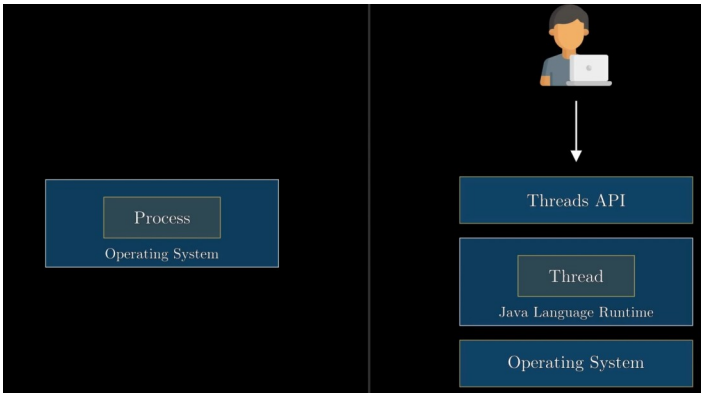
- En este caso, al ejecutar la aplicación se pone en marcha el hilo principal, que queda a la escucha.
- Cuando el hilo principal recibe una petición, creará un nuevo hilo que se encarga de procesarla y generar la consulta, mientras tanto el hilo principal sigue a la escucha recibiendo peticiones y creando hilos.
- De esta manera un gestor de bases de datos puede atender consultas de varios clientes, o un servidor web puede atender a miles de clientes.
- Si el número de peticiones simultáneas es elevado, la creación de un hilo para cada una de ellas puede comprometer los recursos del sistema. En este caso, lo resolveremos mejor con un pool de hilos (contenedor de hilos).

Resumiendo, los hilos son idóneos para programar aplicaciones de entornos interactivos y en red, así como simuladores y animaciones.

2.3. Threads Vs Procesos

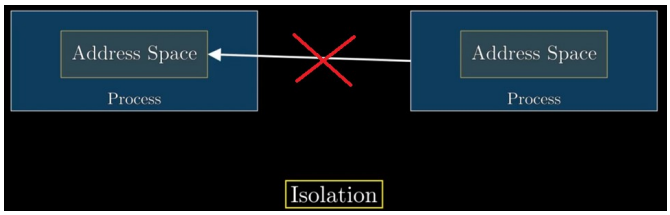
Primera diferencia: Quién los gestiona

Los procesos son objetos con los que se trabaja a nivel de sistema operativo. En cambio, las hebras son objetos con los que se trabaja a nivel de ***Java Runtime*** (se ejecuta un nivel por encima del Sistema Operativo). Por tanto, será el ***Java Runtime*** el encargado de gestionar los hilos y todas sus complejidades.



Segunda diferencia: Aislamiento.

Cada proceso está asociado a una aplicación en ejecución, teniendo cada uno de ellos su propio espacio de direcciones al cual no va a poder acceder ningún otro proceso (**aislamiento**).

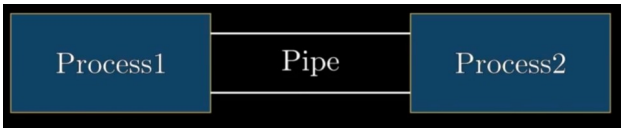


Tercera diferencia: La comunicación.

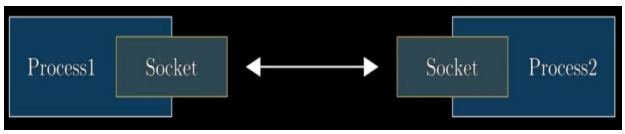
¿Cómo pueden comunicarse los procesos? Como se ha visto en la unidad anterior, si ambos procesos están ejecutándose en la misma máquina, una de las formas más fáciles de lograrlo es mediante el uso de ficheros. Por ejemplo, el proceso 1 produce unos datos que necesita pasar el proceso 2. Para ello los escribe en un fichero y el proceso 2, los lee desde fichero.

En esta solución, ambos procesos necesitan saber el nombre del fichero, tener los permisos correspondiente y acceso exclusivo para que uno no intente leer antes de que el otro haya escrito.

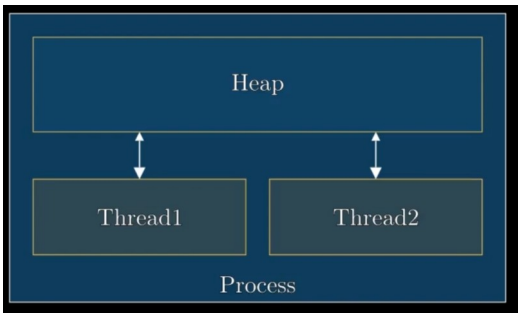
También se puede usar el concepto de Pipe visto en la unidad anterior.



En cambio, si los dos procesos están en máquinas diferentes se puede recurrir al uso de sockets que serán vistos en la unidad siguiente.



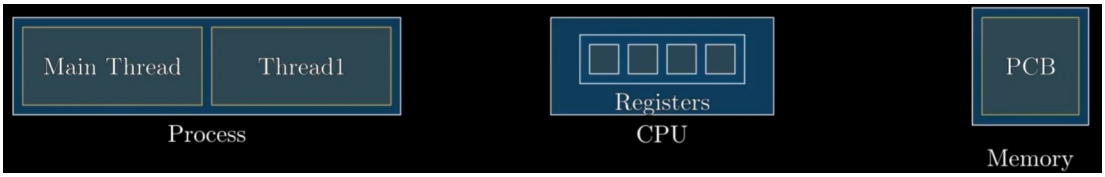
A la hora de llevar a cabo una **comunicación entre hilos**, esta se puede llevar a cabo de un modo muy simple debido a que los hilos pueden compartir el espacio de direcciones del proceso que los crea:



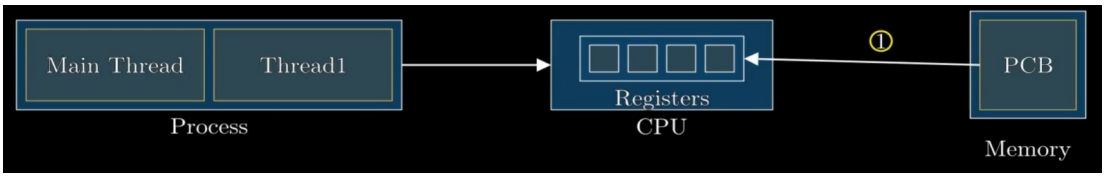
Por tanto, no son necesarios métodos sofisticados ni complejos para efectuar el intercambio de datos ya que los distintos hilos de un mismo proceso comparten la memoria asignada al proceso. Sin embargo, los **hilos deben coordinarse para el acceso a los contenidos de la memoria y a los ficheros, lo cual hace que esa coordinación y sincronización sea la parte complicada de uso**.

Cuarta diferencia: El cambio de contexto entre procesos es más costoso que el cambio de contexto entre hilos.

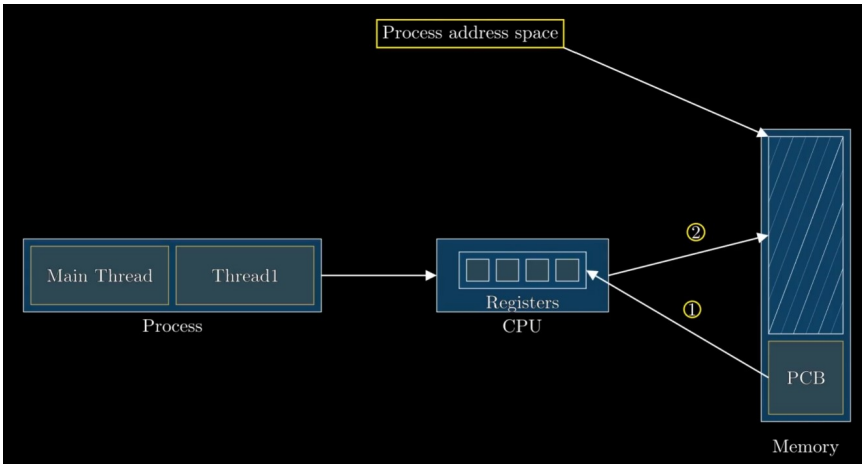
Partimos de una situación en la que se tiene un proceso con su hilo principal y que crea un nuevo hilo.



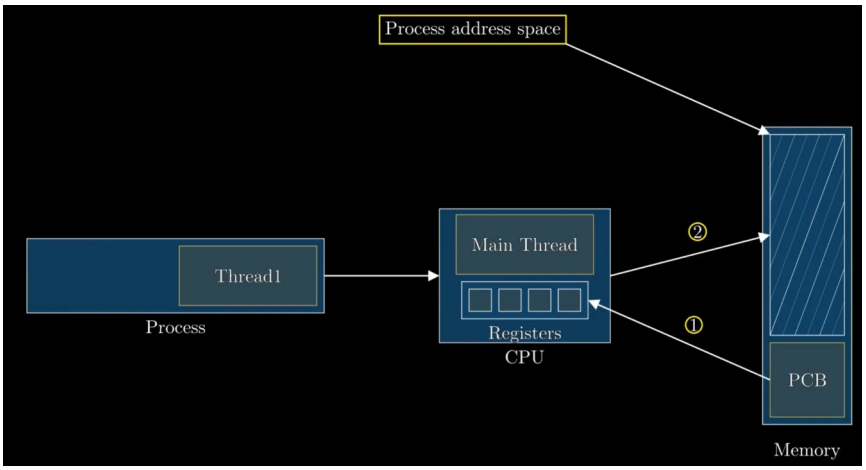
Este proceso está activo y por tanto, cuando el scheduler del sistema operativo le asigna la CPU, se cargan en los registros de la CPU toda la información necesaria leída del PCB.



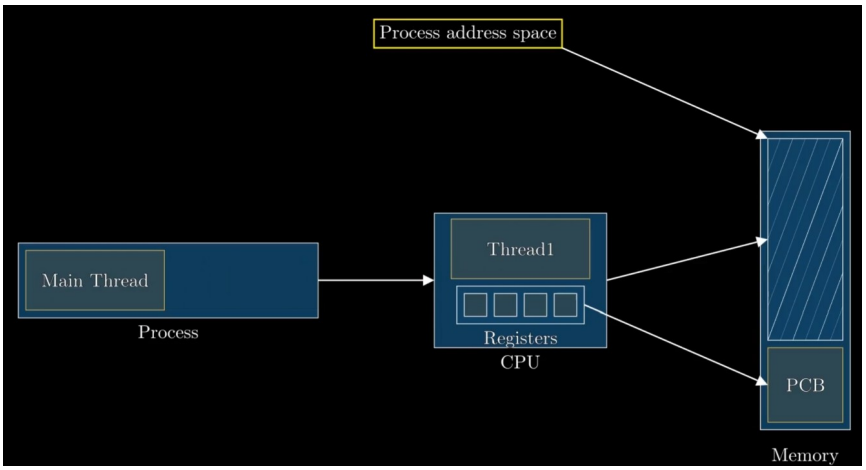
Además, se reserva espacio en memoria para cargar todo el espacio de direcciones del proceso.



A continuación el hilo principal comienza su ejecución.



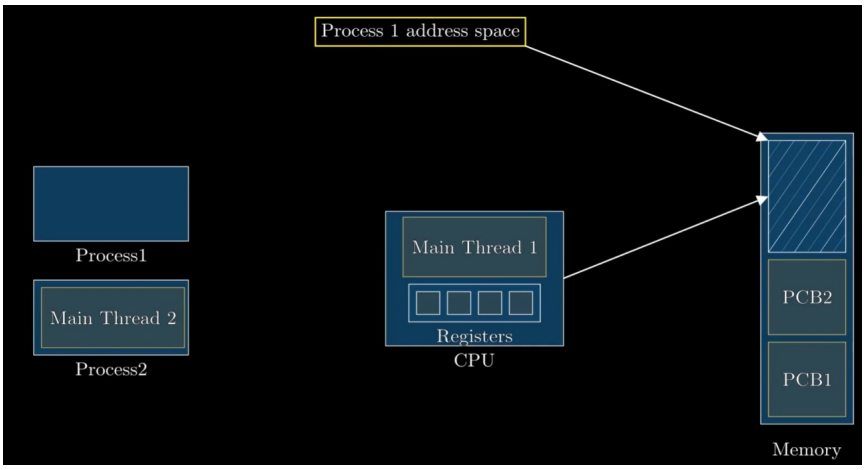
Cuando la CPU vaya a ejecutar el hilo Thread1, el sistema operativo debe actualizar el PCB del proceso, sacar del procesador al hilo principal y asignar el procesador al hilo Thread1.



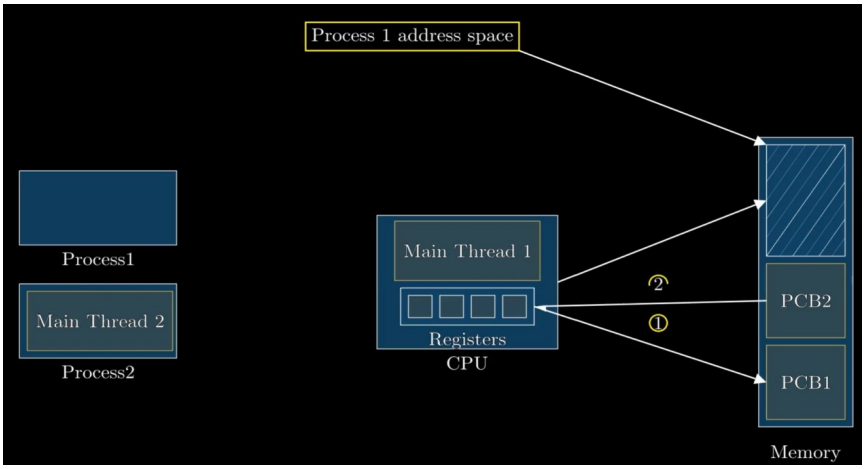
Esto que se acaba de realizar, es lo que se conoce como **cambio de contexto del hilo**.

Supongamos ahora que, en lugar de un proceso con dos hilos, tenemos dos procesos con sus respectivos hilos principales. Ambos procesos están activos, y por consiguiente, a cualquiera de los dos se le podrá asignar la CPU.

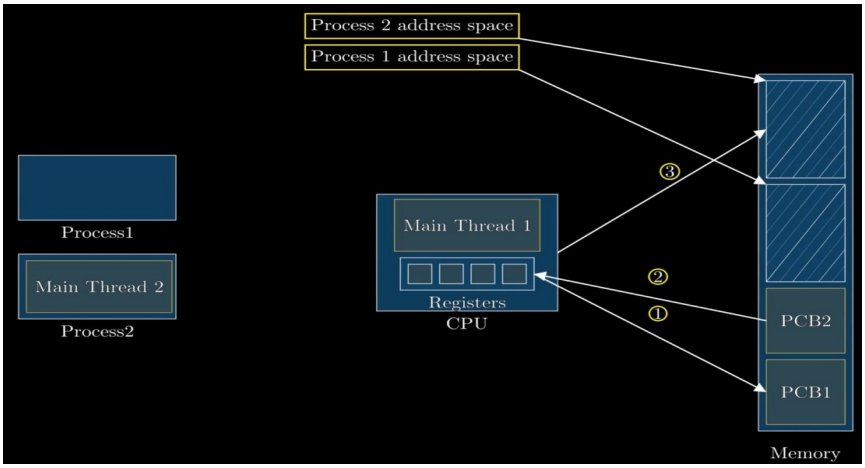
El scheduler del sistema operativo decide asignar la CPU al primero de los procesos:



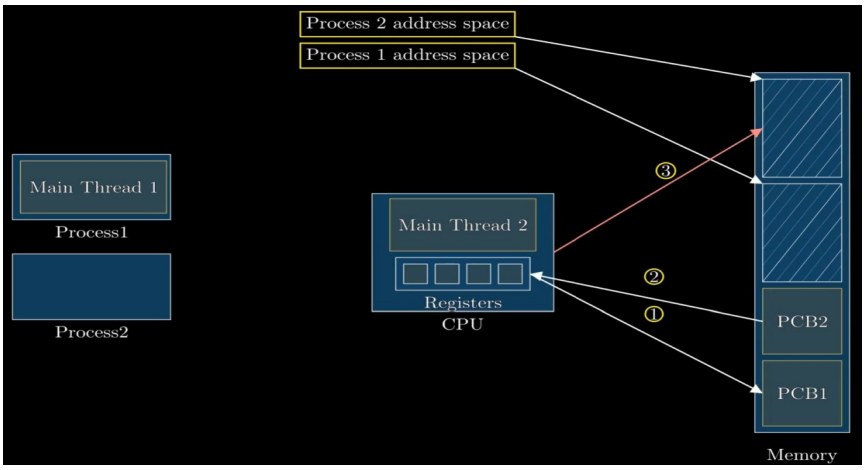
Si el scheduler le arrebata la CPU al proceso 1 para asignarla al proceso 2, debe haber un cambio de contexto a nivel de proceso. Por tal motivo, debe guardar todo el contexto del proceso 1 en su PCB (PCB1). Además, también deberá cargar en los registros de la CPU toda la información necesaria del proceso 2, leyéndola del PCB2.



Además, necesita cargar en memoria el espacio de direcciones del proceso 2:



Finalmente, el hilo principal del proceso 2 comenzará/continuará con su ejecución:



Se puede observar que el cambio de contexto entre procesos, supone una sobrecarga en comparación con el cambio de contexto entre hilos.

3. Multihilos en Java. Librería y clases

Java da soporte al concepto de hilo desde el propio lenguaje, con algunas **clases e interfaces** definidas en el paquete java.lang y con **métodos** específicos para la manipulación de hilos en la clase Object. A partir de la versión Java 5.0, se incluye el paquete java.util.concurrent con nuevas utilidades para desarrollar aplicaciones multihilo e incluso aplicaciones con un alto nivel de concurrencia.

3.1. Utilidades de concurrencia del paquete java.lang

Dentro del **paquete** java.lang disponemos de una interfaz y las siguientes clases para trabajar con hilos:

- **Clase Thread**. Es la clase responsable de producir hilos funcionales para otras clases y proporciona gran parte de los métodos utilizados para su gestión.

  Tiene algunos métodos estáticos importantes, entre los que se pueden destacar:

  - ***currentThread***: Nos devuelve un objeto de tipo ***Thread*** a través del cual se puede acceder a toda la información sobre la hebra que está actualmente ejecutando el código.
  - ***sleep***: Provoca que el hilo se duerma (se suspenda) una determinada cantidad de tiempo en milisegundos. Necesita tratar la excepción ***InterruptedException***.
  - ***getName***: Nos devuelve el nombre del hilo. Por defecto, el nombre del hilo principal se llama ***main***. En caso de no asignarle ningún nombre, a los hilos que se vayan creando se les asignará un nombre por defecto con el formato Thread-xx, siendo xx un número que empieza en 0.
  - ***setName***: Permite cambiar el nombre del hilo.
  - ***start***: Método que inicia o arranca el hilo.

  Ver: MetodosEstaticosThreads.

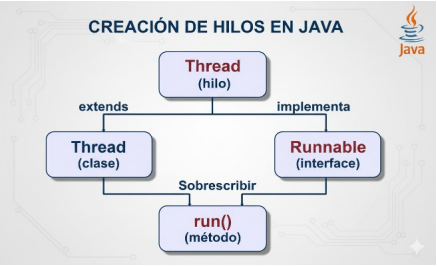
- **Interfaz Runnable**. Proporciona la capacidad de añadir la funcionalidad de hilo a una clase simplemente implementando la interfaz, en lugar de derivándola de la clase thread.
- **Clase ThreadDeath.** Es una clase de error, deriva de la clase Error, y proporciona medios para manejar y notificar errores.
- **Clase ThreadGroup.** Esta clase se utiliza para manejar un grupo de hilos de modo conjunto, de manera que se pueda controlar su ejecución de forma eficiente.
- **Clase Object.** Esta clase no es estrictamente de apoyo a los hilos, pero proporciona unos cuantos métodos cruciales dentro de la arquitectura multihilo de Java. Estos métodos son wait(), notify() y notifyAll().

4. Creación de hilos

En Java, un **hilo** se representa mediante una instancia de la clase java.lang.thread. Este **objeto thread** se emplea para iniciar, detener o cancelar la ejecución del hilo de ejecución.

Los hilos o threads se pueden **implementar o definir de dos formas**:

- Extendiendo la clase Thread.
- Implementando la interfaz Runnable.



En **ambos casos, se debe proporcionar una definición del método** *run(),* ya que este método es el que contiene el código que ejecutará el hilo, es decir, su comportamiento.

El procedimiento de construcción de un hilo es independiente de su uso, pues una vez creado se emplea de la misma forma.

Constructores	
Thread()	Crea una hebra que hereda de Thread y su nombre es asignado por Java.
Thread(Runnable target)	Crea una hebra que implementa la interfaz Runnable y su nombre es asignado por Java.
Thread(String name)	Crea una hebra que hereda de Thread y cuyo nombre es name.
Thread(Runnable target, String name)	Crea una hebra que implementa la interfaz Runnable y cuyo nombre es name

Entonces, ¿**cuando utilizar uno u otro procedimiento?** No hay nada que indique que una forma es mejor que otra. Ambos métodos son similares y el resultado es el mismo.

- Extender la clase Thread es el procedimiento más sencillo, pero no siempre es posible. Si la clase ya hereda de alguna otra clase padre, no será posible heredar también de la clase Thread (recuerda que Java no permite la herencia múltiple), por lo que habrá que recurrir al otro procedimiento.
- El método preferido debería ser implementar Runnable, y pasarle la instancia al constructor de Thread. Implementar Runnable siempre es posible, es el procedimiento más general y también el más flexible.

RECUERDA: Cuando la Máquina Virtual Java (JVM) arranca la ejecución de un programa, ya hay un hilo ejecutándose, denominado **hilo principal del programa**, controlado por el método main(), que se ejecuta cuando comienza el programa y es el último hilo que termina su ejecución, ya que cuando este hilo finaliza, el programa termina.

4.1. Creación de hilos extendiendo de la clase Thread

Para **definir** y **crear un hilo extendiendo la clase Thread**, haremos lo siguiente:

- Implementar una nueva clase que herede de la clase Thread.
- Redefinir en la nueva clase el método run() con el código asociado al hilo. Son las sentencias que ejecutará el hilo.
- Declarar y crear un objeto de la nueva clase Thread. Éste será realmente el hilo.

Una vez creado el hilo, para **ponerlo en marcha o iniciarlo**:

- Invocar al método start() del objeto thread (el hilo que hemos creado).

Ver: CreacionHilosHeredando

4.2. Creación de hilos implementado la interfaz Runnable

Para **definir y crear hilos implementando la interfaz Runnable** seguiremos los siguientes pasos:

- Implementar una nueva clase que implemente Runnable.
- Redefinir en la nueva clase el método run() con el código asociado al hilo. Son las sentencias que ejecutará el hijo
- Declarar un objeto de la clase Thread y se llama al constructor de la clase Thread pasando al constructor un objeto cuya clase implementa Runnable. Este será realmente el hilo.

Una vez creado el hilo, para ponerlo en marcha o iniciarlo:

- Invocar al método start() del objeto thread (el hilo que hemos creado).

Ver: CreacionHilosRunnable; EjemploRunnable;

4.3. Creación de hilos mediante clases anónimas que implementan la interfaz Runnable

Para ello seguiremos los siguientes pasos:

- Declarar y crear un objeto de tipo Runnable.
- Redefinir el método run() con el código asociado al hilo.
- Declarar un objeto de la clase Thread y se instancia llamando al constructor de la clase Thread pasando al constructor, el objeto runnable previamente declarado y creado.

Una vez creado el hilo, para ponerlo en marcha o iniciarlo:

- Invocar al método start() del objeto thread (el hilo que hemos creado).

Ver: CreacionHilosAnonimoRunnable

4.4. Creación de hilos mediante expresiones lambda

Permite hacer lo mismo que en el caso anterior pero de una forma más elegante. Para ello seguiremos los siguientes pasos:

- Declarar un objeto de tipo Runnable y se le asocia la expresión lambda que incluye el código asociado al hilo.
- Declarar un objeto de la clase Thread y se instancia llamando al constructor de la clase Thread pasando al constructor, el objeto runnable previamente declarado y creado.

Una vez creado el hilo, para ponerlo en marcha o iniciarlo:

- Invocar al método start() del objeto thread (el hilo que hemos creado).

Esto tiene sentido usarlo en el caso de que la tarea la vamos a usar en un solo punto de la aplicación y no se va a volver a instanciar.

Ver: CreacionHilosExpresionLambda;

4.5. Iniciar un hilo

Cuando se crea un nuevo hilo mediante el método new(), esto no implica que el hilo ya se pueda ejecutar.

Para que el hilo se pueda ejecutar, debe estar en el estado "Ejecutable", y para conseguir ese estado es necesario **iniciar o arrancar el hilo** mediante el método start() de la clase Thread.

En realidad el método start() **realiza las siguientes tareas:**

- Crea los recursos del sistema necesarios para ejecutar el hilo.
- Se encarga de **llamar a su método** run().

Es por esto último que cuando se invoca a start() se suele decir que el hilo está **Runnable**, pero esto no significa que el hilo esté ejecutándose en todo momento, ya que **un hilo "Ejecutable" puede estar "Runnable" o "Running"** según tenga o no asignado tiempo de procesamiento.

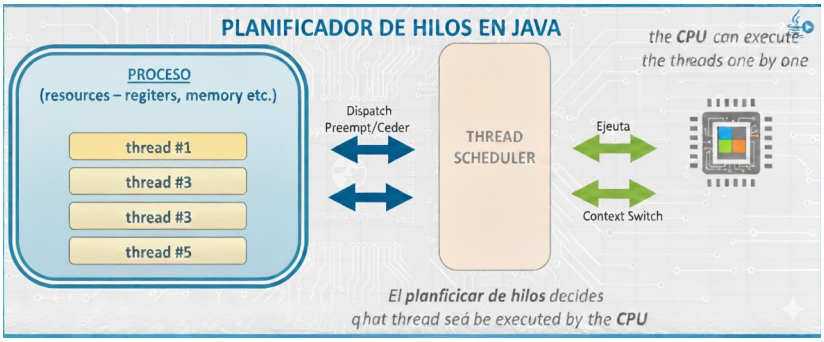
Algunas **consideraciones importantes** que debes tener en cuenta son las siguientes:

- Puedes invocar directamente al método run(), por ejemplo poner hilo1.run(); y se ejecutará el código asociado a run() dentro del hilo actual (como cualquier otro método), pero no comenzará un nuevo hilo.
- Una vez que se ha llamado al método start() de un hilo, no puedes volver a realizar otra llamada al mismo método. Si lo haces, obtendrás una excepción IllegalThreadStateException.
- El orden en el que inicies los hilos mediante start() no influye en el orden de ejecución de los mismos, lo que pone de manifiesto que **el orden de ejecución de los hilos es no-determinístico** (no se conoce la secuencia en la que serán ejecutadas la instrucciones del programa).

Ver: CreacionVariosHilos

5. Prioridades de hilos

En Java, **cada hilo tiene una prioridad** representada por un valor de tipo entero **entre 1 y 10**. Cuanto mayor es el valor, mayor es la prioridad del hilo.



Por defecto, el **hilo principal** de cualquier programa, o sea, el que ejecuta su método main() siempre es creado con prioridad 5.

El resto de **hilos secundarios** (creados desde el hilo principal, o desde cualquier otro hilo en funcionamiento), **heredan la prioridad que tenga en ese momento su hilo padre**.

En la clase Thread se definen 3 constantes para manejar estas prioridades:

- MAX_PRIORITY (= 10). Es el valor que simboliza la máxima prioridad.
- MIN_PRIORITY (=1). Es el valor que simboliza la mínima prioridad.
- NORM_PRIORITY (= 5). Es el valor que simboliza la prioridad normal, la que tiene por defecto el hilo donde corre el método main().

Además en cualquier momento se puede **obtener** y **modificar la prioridad de un hilo**, mediante los siguientes métodos de la clase Thread:

- getPriority(). Obtiene la prioridad de un hilo. Este método devuelve la prioridad del hilo.
- setPriority(int prioridad). Modifica la prioridad de un hilo. Este método toma como argumento un entero entre 1 y 10, que indica la nueva prioridad del hilo.

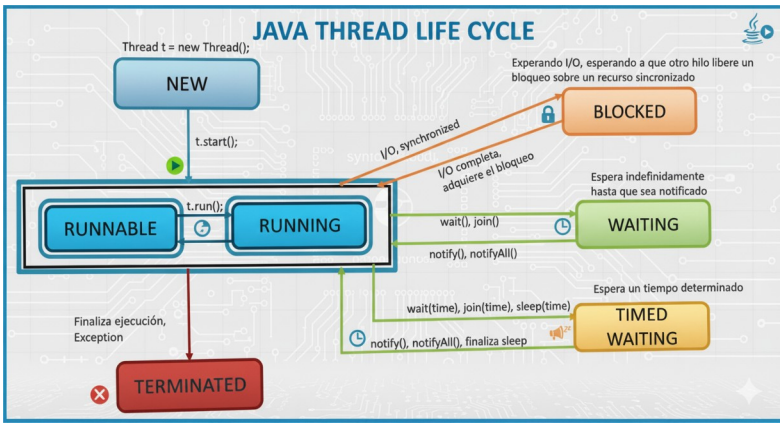
Por tanto, Java tiene 10 niveles de prioridad que no tienen por qué coincidir con los del sistema operativo sobre el que está corriendo. Por ello, lo mejor es que utilices en tu código sólo las constantes MAX_PRIORITY, NORM_PRIORITY y MIN_PRIORITY.

Podemos conseguir **aumentar el rendimiento de una aplicación multihilo** gestionando adecuadamente las prioridades de los diferentes hilos, por ejemplo utilizando una prioridad alta para tareas de tiempo crítico y una prioridad baja para otras tareas menos importantes.

Ver: PrioridadesHilos.

Ejemplo en el que se declara un hilo cuya tarea es llenar un vector con 20000 caracteres. Se inician 15 hilos con prioridades diferentes, 5 con prioridad máxima, 5 con prioridad normal y 5 con prioridad mínima. Al ejecutar el programa comprobarás que los hilos con prioridad más alta tienden a finalizar antes.

6. Estados de un hilo



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

- **NEW**

  Estado de un hilo cuando se crea con el operador *new*. En este estado el hilo aún no se ha iniciado, es decir, no se ha comenzado a ejecutar el código del método *run()* del hilo.

- **RUNNABLE**

  Estado de un hilo cuando está listo para ejecutarse, es decir, está esperando a que se le asigne la CPU. Pasará al estado de RUNNING cuando el scheduler de hilos lo selecciona y lo asigna a uno de los núcleos de la CPU, pasando a ejecutar las sentencias de su método *run()*.

- **BLOCKED**

  En este estado el hilo está bloqueado esperando a que otro hilo libere un recurso por el que compiten. También se encuentra en este estado aquellos hilos que están esperando a que se complete una operación de E/S.

- **WAITING**

  En este estado el hilo espera indefinidamente hasta que otro hilo realice una acción concreta. Pasará al estado de RUNNABLE cuando otro hilo realice una notificación (*notify*).

- **TIMED_WAITING**

  En este estado se encuentran aquellos hilos que están esperando un tiempo específico a que otro hilo realice una acción. Pasará al estado RUNNABLE cuando otro hilo realice una notificación (*notify*) o se haya completado el tiempo de espera.

- **TERMINATED**

  Estado que alcanza un hilo cuando finaliza. Un hilo finaliza por cualquiera de las siguientes razones:

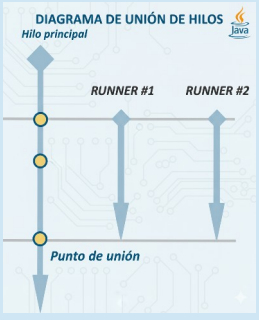
  - ➜ Cuando el código del hilo se ha ejecutado completamente, es decir, cuando se ha finalizado la ejecución de todas las sentencias de su método *run()*.
  - ➜ Cuando se produce algún error inusual que provoca una excepción no controlada.

Se usará el método *getState()* para, en cualquier momento, consultar el estado en el que se encuentra un hilo.

Ver: EstadosHilo: Programa cuyo hilo principal lanza un hilo secundario que realiza una cuenta atrás desde 10 hasta 1. Desde el hilo principal se verificará la muerte del hilo secundario mediante la función isAlive(). Además mediante el método getState() de la clase Thread vamos obteniendo el estado del hilo secundario. Se usa también el método join() que espera hasta que el hilo termina.

7. Métodos de la clase java.lang.Thread

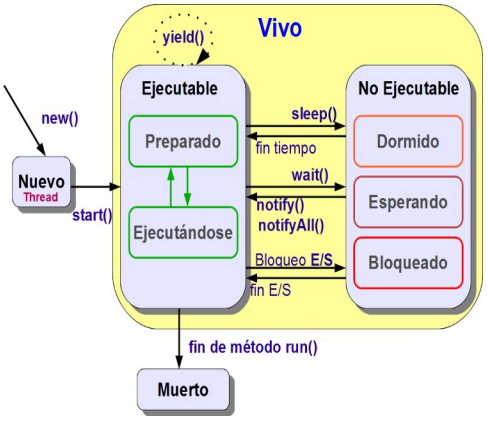
Veamos algunos de los métodos de la clase Thread más utilizados:

Método	Descripción
boolean isAlive()	Comprueba si un thread está vivo o no. Un hilo se considera que está vivo desde que se llama a su método start() hasta que finaliza su ejecución. Por tanto, este método devolverá: • **False**: Estamos ante un nuevo hilo recién "creado" o ante un hilo "muerto". • **True**: sabemos que el hilo se encuentra en un estado diferente a ***NEW*** y a ***TERMINATED***.
sleep(long ms) sleep (long milisegundos, int nanosegundos)	Cambia el estado del thread a ***TIMED_WAITING*** durante los ms indicados.
String toString()	Devuelve una representación legible de un thread: nombre, priority, nombre_del_grupo.
~~long getId()~~	Deprecated. Devuelve el identificador del thread.
void yield()	Hace que el hilo pare su ejecución instantáneamente volviendo a la cola y permitiendo que otros hilos y/o procesos se ejecuten. Sin embargo, el funcionamiento de Thread.yield() no está garantizado. Puede que después de que un hilo invoque a Thread.yield() y pase a "RUNNABLE", éste vuelva a ser elegido para ejecutarse. No garantiza que el hilo actual deje de ejecutarse. El scheduler puede ignorarlo.
void join()	Se llama desde otro hilo y hace que el hilo que lo invoca se bloquee hasta que el thread termine. Es parecido a p.waitFor() para los procesos. 

7.1. Detener temporalmente un hilo

¿Qué significa que un hilo se ha detenido temporalmente? Significa que **el hilo ha pasado a un estado "No Ejecutable"**, concretamente al estado ***TIMED_WAITING*** o ***WAITING***.

Un hilo pasará al estado ***TIMED_WAITING (hilo dormido)*** por alguna de estas circunstancias:



- Se ha invocado al método sleep() de la clase Thread, indicando el tiempo que el hilo permanecerá deteniendo. Transcurrido ese tiempo, el hilo se vuelve "Ejecutable", en concreto pasa al estado ***RUNNABLE***.

  Hay dos formas de llamar al método sleep():

  - La primera le pasa como argumento un entero (positivo) que representa el tiempo en milisegundos que el hilo permanecerá dormido.
  - La segunda le agrega un segundo argumento entero (esta vez, entre 1 y 999999), que representa un tiempo extra en nanosegundos (Un nanosegundo es la milmillonésima parte de un segundo, (10-9 s)) que se sumará al primer argumento:

- Cualquier llamada a sleep() puede provocar una excepción, que el compilador de Java nos obliga a controlar ineludiblemente mediante un bloque try-catch.

Un hilo pasará al estado ***WAITING (hilo esperando)*** cuando:

- El hilo ha detenido su ejecución mediante la llamada al método wait(), y no se reanudará, no pasará a ***RUNNABLE***, hasta que otro hilo produzca una llamada al método notify() o notifyAll(). Estudiaremos detalladamente estos métodos de la clase Object cuando veamos la sincronización y comunicación de hilos.

Un hilo pasará al estado BLOCKED (hilo bloqueado) cuando:

- El hilo está pendiente de que finalice una operación de E/S en algún dispositivo, o a la espera de algún otro tipo de recurso(ha sido bloqueado por el sistema operativo). Cuando finaliza el bloqueo, vuelve al estado ***RUNNABLE***.

Ver: JoinSleep; Viaje

7.2. Hilos demonio (Daemon Thread)

Todos los hilos que hemos visto hasta ahora son llamados hilos de usuario (**User Threads**). Sin embargo, también existen los conocidos como hilos de sistema (**System Threads**) que la JVM crea automáticamente para manejar tareas internas como gestionar recursos, gestionar señales del sistema operativo, finalización de objetos, etc.

Ver: HilosJVM

Además, en una aplicación se pueden tener hilos demonio (**Daemon Thread**) que sí los puede crear el desarrollador de la aplicación.

Un **Daemon Thread** es un hilo background cuyo propósito es asistir a otros hilos (por ejemplo, liberación de recursos, tareas de limpieza, timers, logging en segundo plano).

La característica esencial de estos hilos es que la JVM los puede finalizar automáticamente si no queda ningún **User Thread** vivo.

Los **Daemon Thread** se crean a partir de hilos de usuario, siendo convertidos en hilos daemon usando el método setDaemon(true). Esto debe hacerse antes de llamar al método start() del hilo. Si se intenta después de la llamada al método start() se lanzará un IllegalThreadStateException.

Por otro lado, para saber si un hilo es un hilo de usuario o un hilo demonio se puede utilizar el método isDaemon().

Ver: UserThread; DaemonThread

8. Sincronización y comunicación de hilos

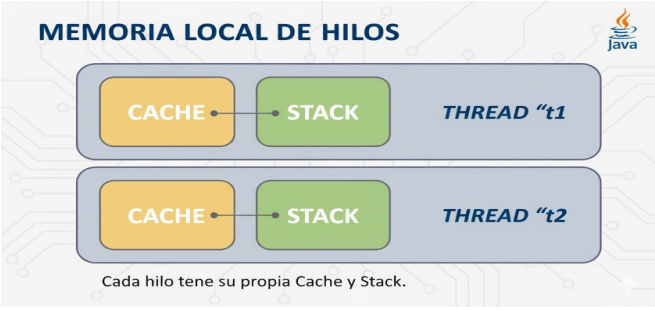
Los ejemplos realizados hasta ahora utilizan hilos independientes, es decir, una vez iniciados los hilos, éstos no se relacionan con los demás y no acceden a los mismos datos u objetos, por lo que no hay conflictos entre ellos.

Sin embargo, hay ocasiones en las que distintos hilos de un programa necesitan establecer alguna relación entre sí y **compartir recursos o información**. Se pueden presentar las siguientes situaciones:

- Dos o más hilos **compiten por obtener un mismo recurso**, por ejemplo dos hilos que quieren escribir en un mismo fichero o acceder a la misma variable para modificarla.
- Dos o más hilos **colaboran para obtener un fin común** y para ello, necesitan comunicarse a través de algún recurso. Por ejemplo un hilo produce información que utilizará otro hilo.

En cualquiera de estas situaciones, es necesario que los hilos se ejecuten de manera controlada y coordinada, para evitar posibles interferencias que pueden desembocar en programas que se bloquean con facilidad y que intercambian datos de manera equivocada.

Ver: ProblemaSincronizacion



¿Cómo conseguimos que los hilos se ejecuten de manera coordinada? Utilizando sincronización y comunicación de hilos:

- **Sincronización**. Es la capacidad de informar de la situación de un hilo a otro. El objetivo es establecer la secuencialidad correcta del programa.
- **Comunicación**. Es la capacidad de transmitir información desde un hilo a otro. El objetivo es el intercambio de información entre hilos para operar de forma coordinada.

En Java la sincronización y comunicación de hilos se consigue mediante:

- **Monitores**. Se crean al marcar bloques de código con la palabra synchronized.
- **Semáforos**. Podemos implementar nuestros propios semáforos, o bien utilizar la clase Semaphore incluida en el paquete java.util.concurrent.
- **Notificaciones**. Permiten comunicar hilos mediante los métodos wait(), notify() y notifyAll() de la clase java.lang.Object.
- Otras Clases del paquete java.util.concurrent: Además, de Semaphore, Java proporciona unas **clases de sincronización** que permiten la sincronización y comunicación entre diferentes hilos de una aplicación multithreadring: CountDownLatch, CyclicBarrier y Exchanger.

Ver: ProblemaSincronizacionResuelto

8.1. Información compartida entre hilos

Las **secciones críticas** son aquellas secciones de código que no pueden ejecutarse concurrentemente, debido a que en ellas se encuentran recursos o información que comparten diferentes hilos.

Un ejemplo sencillo que ilustra lo que puede ocurrir cuando varios hilos actualizan una misma variable es el clásico "ejemplo de los jardines". En él, se pone de manifiesto el problema conocido como la "**condición de carrera**", que se produce cuando varios hilos acceden a la vez a un mismo recurso, por ejemplo a una variable, cambiando su valor y obteniendo de esta forma un valor no esperado de la misma.

En el ejemplo del "problema de los jardines", el recurso que comparten diferentes hilos es la variable contador **cuenta**. Las secciones de código donde se opera sobre esa variable son dos secciones críticas, los métodos incrementaCuenta() y decrementaCuenta().

La forma de proteger las secciones críticas es mediante sincronización. La **sincronización** se consigue mediante:

- **Exclusión mutua**. Asegurar que un hilo tiene acceso a la sección crítica de forma exclusiva y por un tiempo finito.
- **Por condición**. Asegurar que un hilo no progrese hasta que se cumpla una determinada condición.

En Java, la sincronización para el acceso a recursos compartidos se basa en el concepto de monitor.

8.2. Monitores. Métodos synchronized

En Java, cada objeto tiene asociado un bloqueo intrínseco llamado **monitor**. Para **crearlo**, hay que **marcar un bloque de código con la palabra synchronized**, pudiendo ser ese bloque:

- Un **método completo**.
- Cualquier **segmento de código**.

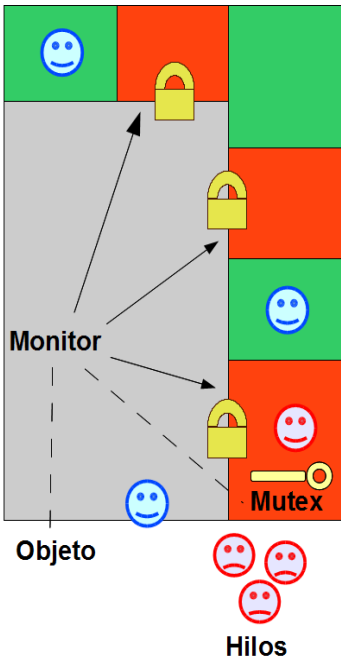
Añadir synchronized a un método significará que:

- Hemos creado un monitor asociado al objeto.
- Sólo un hilo puede ejecutar el método synchronized de ese objeto a la vez.
- Los hilos que necesitan acceder a ese método synchronized permanecerán bloqueados en una cola de hilos que está asociada al objeto (**BLOCKED**).
- Cuando el hilo finaliza la ejecución del método synchronized, los hilos en espera de poder ejecutarlo se desbloquearán. La JVM seleccionará a uno de ellos.

sintaxis para declarar **un método** synchronized	sintaxis para declarar **un segmento de código** synchronized

public **synchronized** tipodato metodo()
{
//sentencias ;
}
	public **synchronized** tipodato metodo()
	{
	//sentencias;
	synchronized (objeto)
	{
	//sentencias;
	}
	//sentencias;
	}

Y **funciona de la siguiente forma**:



- Un monitor está asociado con un objeto específico y **solo se asocia un monitor por objeto**, aunque éste tenga más de un bloque synchronized.
- Sólo un hilo puede tener el mutex o candado de un objeto en un momento dado.
- El resto de hilos que también necesitan hacerse con el mutex de ese objeto para acceder a los bloques synchronized, permanecerán bloqueados a la espera de que se libere el mutex del objeto.
- Cuando el hilo finaliza la ejecución de un bloque synchronized libera el mutex.
- Al liberarse el mutex, todos los hilos que estaban en espera para obtenerlo, se reactivarán y la JVM cederá el mutex a uno de ellos.
- El monitor sólo permitirá que un hilo pueda ejecutar un bloque synchronized a la vez. Por tanto, si existen varios bloques synchronized dentro de un objeto, sólo uno de ellos podrá ejecutarse al mismo tiempo.

Por tanto, **el hilo que quiere entrar en un método o bloque synchronized se puede encontrar** con las siguientes situaciones:

- Mutex libre. El hilo tomará el mutex, ejecutará el método y lo liberará cuando haya finalizado la ejecución de dicho método.
- Mutex en posesión de otro hilo. El hilo se bloqueará en espera de que el otro hilo lo libere.

Y ¿**qué bloques interesa marcar** como synchronized? Precisamente los que se correspondan con secciones críticas y contengan el código o datos que comparten los hilos.

En el ejemplo anterior, "Problema de los jardines" se debería sincronizar tanto el método *incrementaCuenta()*, como el *decrementaCuenta()* tal y como ves en el siguiente código, ya que estos métodos contienen la variable *cuenta*, la cual es modificada por diferentes hilos. Así mientras un hilo ejecuta el método *incrementaCuenta()* del objeto *jardin,* ningún otro hilo podrá ejecutarlo.

```
public synchronized void incrementaCuenta() {
    //método que increamenta en 1 la varibale cuenta
    System.out.println("hilo " + Thread.currentThread().getName()
            + "----- Entra en Jardín");
    //muestra el hilo que entra en el método
    cuenta++;
    System.out.println(cuenta + " en jardín");
    //cuenta cada acceso al jardín y muestra el número de accesos
}
```

Ver: JardinesSincronizado ; SincronizacionFrases; SincronizaMetodo

Ver presentación: new_threads.pptx

El problema de usar synchronized es que si tenemos varios métodos independientes que tienen que ser sincronizados y varios hilos, cada uno usando uno de esos métodos independientes, al existir un solo monitor por objeto, si uno de los hilos accede al primer método sincronizado, el otro hilo no podrá acceder al segundo método sincronizado aunque sea independiente. Esto provocará que la aplicación sea más lenta y no se esté maximizando el rendimiento del equipo.

Por lo tanto, cuando dentro de una clase se declaran múltiples métodos synchronized, todos los hilos competirán por el mismo monitor.

Solución: Declarar synchronized un segmento de código en lugar de un método completo.

En sistemas con muchos hilos o alto volumen de peticiones, el rendimiento se degrada significativamente.

Ver: ProblemaSincronizacionMultiple; ProblemaSincronizacionMultipleConEstado

8.3. Monitores. Segmentos de código synchronized

En este caso en lugar de poner el modificador synchronized en la cabecera del método, se pone en la parte del código del método donde existe sección crítica.

```
public synchronized tipodato metodo()
{
   //sentencias;
   synchronized (objeto)
   {
      //sentencias;
   }
   //sentencias;
}
```

En este caso el **funcionamiento** es el siguiente:

- El hilo que entra en el segmento declarado synchronized se hará con el monitor del objeto, si está libre, o se bloqueará en espera de que quede libre. El monitor se libera al salir el hilo del segmento de código synchronized.

- Sólo un hilo puede ejecutar el segmento synchronized a la vez.

Ver: ProblemaSincronizacionMultipleResuelto ; ProblemaSincronizacionMultipleConEstadosResuelto

Hay otros casos en los que no se puede sincronizar un método. Por ejemplo, no podremos sincronizar un método que no hemos creado nosotros y que por tanto no podemos acceder a su código fuente para añadir synchronized en su definición. En esta situación debemos hacer lo siguiente:

- El objeto que se pasa al segmento, y que por tanto, se debe marcar como synchronized, es el objeto donde está el método que se quiere sincronizar.

- Dentro del segmento se hará la llamada al método que se quiere sincronizar.

Ver: SincroSegmentoJardines

En el ejemplo del problema de los jardines, aplicando este procedimiento, habría que sincronizar el objeto que denominaremos *jardin* y que será desde donde se invoca al método *incrementaCuenta(),* método que manipula la variable *cuenta* que modifican diferentes hilos:

```
public void run() {
    //método que incrementa la cuenta de accesos
    for (int i = 1; i <= 10; i++) //se simulan 10 accesos
    {
        //segmento que se sincroniza
        synchronized (jardin){
            jardin.incrementaCuenta();
        }
    }
}
```

Observa que:

- Ahora el método incrementaCuenta() no será synchronized (se haría igual para decrementaCuenta()).

- Se está consiguiendo un acceso con exclusión mutua sobre el objeto *jardin*, aún cuando su clase no contiene ningún segmento ni método synchronized.

Algunas consideraciones sobre el uso de synchronized debido a algunas limitaciones que presenta:

- Cuellos de botella, ya que otros hilos deben esperar a que el bloqueo se libere, incluso si no acceden a datos compartidos.

- En sistemas con muchos hilos o alto volumen de peticiones, el rendimiento se degrada significativamente ya que la **adquisición y liberación de monitores genera una sobrecarga**.

- Posibilidad de inanición(**starvation**), problema que surge cuando hay hilos que nunca pueden ejecutar el código syncronized debido a la aleatoriedad de la selección de los hilos que salen de bloqueados, es decir, cuando hay hilos que monopolizan el uso del recurso.

- Al declarar bloques synchronized puede aparecer un **nuevo problema, denominado interbloqueo**.

Para solventar estas limitaciones, Java ofrece algunas alternativas basadas en el uso de Locks.

8.4. Uso de Lock para sincronización

En Java, un Lock es una alternativa más flexible al uso del bloque synchronized. Se encuentra en el paquete java.util.concurrent. Al igual que synchronized, sirve para controlar el acceso a recursos compartidos entre varios hilos y así evitar condiciones de carrera (race conditions).

Para ello, todo el código correspondiente a la sección crítica, se situará entre la llamada a los métodos lock y unlock de un objeto de tipo Lock, garantizando que todo el código existente entre ambos será ejecutado como mucho por un solo hilo. El resto de hilos que intenten entrar en la sección crítica pasarán al estado de **BLOCKED**, pasando a **RUNNABLE** cuando se ejecute el método unlock.

```
Lock objetoLock=new ReentrantLock(); //Objeto de tipo Lock
objetoLock.lock();  //Bloquea el recurso
try
{
        valor++;
}
finally
{
        objetoLock.unlock(); //Siempre se libera, incluso si hay excepción
}
```

Ver: EjemploSinLock; EjemploConLock; SumaParalelaVector;
Convertir el ejemplo ProblemaSincronizacionResuelto usando ReentrantLock.

Sin embargo, presenta algunas diferencias:

- Permiten más control sobre el bloqueo y desbloqueo, pudiendo abarcar varios métodos. En cambio, con synchronized, solo se puede abarcar como máximo el código de un método.

- Pueden intentar adquirir el lock sin bloquearse indefinidamente, usando el método tryLock. Con este método intenta adquirir el bloqueo, pero no espera si está ocupado. Devuelve **true** o **false**. El método está sobrecargado permitiendo pasar un parámetro que indica que, en caso de bloquearse, cuanto tiempo va a estarlo.

- Se pueden interrumpir mientras esperan usando el método lockInterruptibly().

- Muy importante no olvidar poner el unlock porque de no hacerlo, se pueden provocar un bloqueo considerable. En cambio, con synchronized se tiene asegurado que el bloque de la sección crítica finaliza donde termina el método establecido como synchronized o el bloque de código marcado como synchronized.

Ver: TryLockBanco; TryLockCuenta

Existen varios tipos de Locks:

- ReentrantLock, es una implementación concreta de la interface Lock que funciona similar a synchronized. Se llama así, porque se puede usar el mismo bloqueo múltiples veces. En este caso, es importante llamar al método unlock tantas veces como haya sido llamado el método lock.

  Al constructor se le puede pasar un parámetro booleano que cuando es **true**, permite que cuando varios hilos están bloqueados (esperando a ejecutar en la sección crítica) por el mismo objeto, la **JVM** los libera cuando se desbloquea el objeto y de los bloqueados selecciona para ejecutar la sección crítica a un hilo siguiendo un criterio FIFO.

  Por tanto, los hilos adquieren el lock en el orden en el que ellos lo solicitaron, **evitando la inanición de hilos**. Sin embargo, esto último puede bajar el rendimiento.

- ReentrantReadWriteLock (para lectura/escritura concurrente)

- StampedLock (más avanzado, desde Java 8)

8.4.1 ReentrantReadWriteLock

Es un tipo de Lock usado para lectura/escritura concurrente. Es una implementación pensada para escenarios donde hay muchas operaciones de lectura y pocas de escritura.

Permite que varios hilos lean simultáneamente (concurrencia en lectura) pero garantiza exclusividad para escrituras. Esto puede mejorar el rendimiento frente a un synchronized/ReentrantLock exclusivo cuando la carga es mayoritariamente de lectura.

Consta de varios métodos a destacar:

- readLock **(bloqueo de lectura)**

  Permite crear múltiples lectores simultáneos mientras no haya un escritor activo, logrando que todos se puedan ejecutar en paralelo.

- writeLock **(bloqueo de escritura)**

  Cuando un hilo tiene el write lock, ningún otro hilo puede leer ni escribir, pasando a estar bloqueados.

En definitiva, los lectores (readLock) pueden acceder al mismo tiempo al recurso compartido, sin embargo, el escritor (writeLock) cuando intenta acceder al recurso compartido, si hay alguien leyendo espera a que todos los lectores terminen y bloquea nuevas lecturas mientras escribe.

Ver: ReadWriteLocks; ReadWriteLocksBiblioteca;

8.4.2 Variables atómicas

En Java, las atomic variables son clases que permiten realizar operaciones atómicas sobre variables sin necesidad de usar synchronized. Esto significa que las operaciones sobre estas variables son seguras en entornos multihilo (threads) y **no requieren bloqueos explícitos**.

Se encuentran en el paquete java.util.concurrent.atomic. En dicho paquete se pueden encontrar diferentes clases que soportan operaciones atómicas sobre variables de tipo int, long, boolean, (AtomicInteger, AtomicLong, AtomicBoolean,…).

Las clases atómicas resuelven el problema de la sección crítica utilizando eficientemente operaciones a nivel de hardware y un algoritmo llamado **Compare-And-Swap (CAS)**.

CAS (Comparar y Reemplazar): Es una instrucción de hardware que toma tres operandos:
- V: La ubicación de la memoria donde está la variable.
- A: El valor esperado actual (el valor antiguo).
- B: El nuevo valor.

La operación CAS **actualiza atómicamente el valor de V a B solo si el valor actual en V es igual a A**. Devuelve un booleano que indica si la sustitución se realizó con éxito. Si falla, el thread puede reintentarlo (a menudo en un loop). Esto elimina la necesidad de bloqueos explícitos, lo que reduce la contención y mejora el rendimiento.

Algunos métodos a destacar de este tipo de variables son:

Método	Descripción
incrementAndGet()	Incrementa y retorna el nuevo valor de manera atómica.
getAndIncrement()	Retorna el valor antes de incrementar:
get()	Obtiene el valor actual.
set(newValue)	Establece el valor en newValue.
decrementAndGet()	Decrementa y retorna el nuevo valor de manera atómica.
getAndDecrement()	Retorna el valor antes de decrementar:
compareAndSet(expectedValue,newValue)	Realiza el cambio solo si el valor actual coincide con expectedValue.

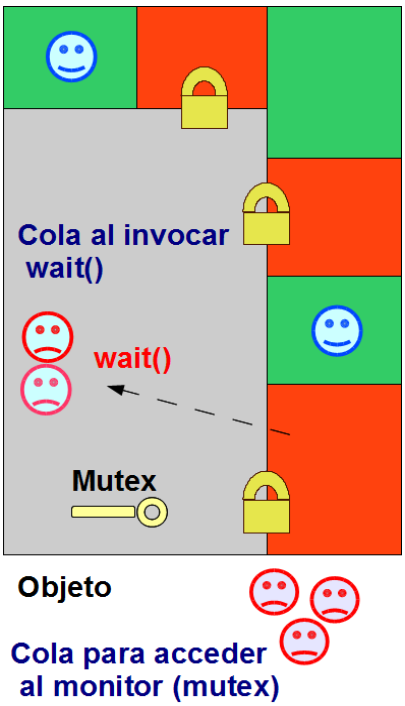
Convertir el ejemplo ProblemaSincronizacionResuelto usando variables atómicas.

8.5. Sincronización y comunicación con métodos de java.lang.Object.

La comunicación entre hilos la podemos ver como un mecanismo de auto-sincronización, que consiste en lograr que un hilo actúe solo cuando otro ha concluido cierta actividad (y viceversa).

Java soporta **comunicación entre hilos** mediante los siguientes métodos de la clase java.lang.Object.

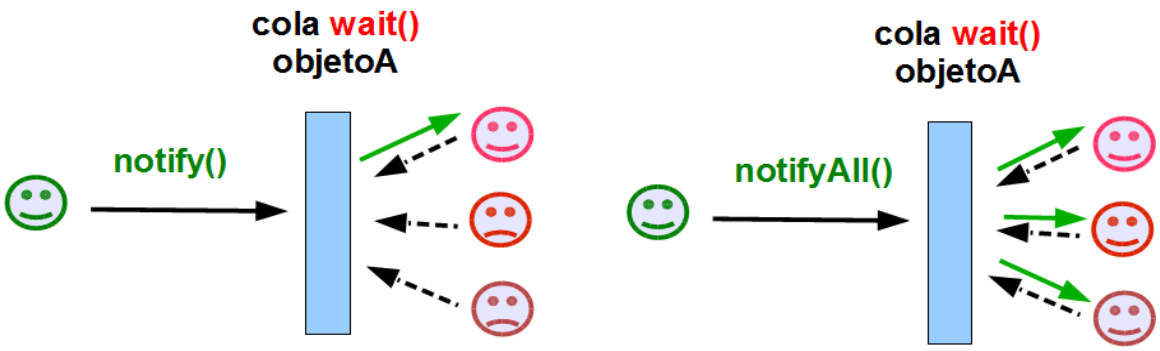
- wait(). Detiene el hilo que pasa al estado ***WAITING***. El hilo no se reanudará hasta que otro hilo notifique que ha ocurrido lo esperado, es decir, haga un notify o un notifyAll.
- wait (long tiempo). Detiene el hilo que pasa al estado ***TIMED_WAITING***. El hilo podrá reanudarse no solo cuando otro hilo haga un notify o un notifyAll, también cuando haya transcurrido el tiempo pasado como parámetro.
- notify(). Notifica para que uno de los hilos puestos en espera (***WAITING*** o ***TIMED_WAITING***) sobre el mismo objeto, pueda continuar.
- notifyAll(). Notifica para que todos los hilos puestos en espera (***WAITING*** o ***TIMED_WAITING***) sobre el mismo objeto, puedan continuar.



DESTACADO. La llamada a estos métodos se realiza dentro de bloques synchronized.

¿Cómo funcionan realmente los métodos wait(), notify() y notifyAll()

- wait(): **Detiene al hilo que lo invoca** hasta que le sea notificada la posibilidad de continuar.
  - El método wait() se debe invocar sobre un objeto compartido por los hilos a sincronizar.
  - Para poder invocar a wait() **el hilo debe tener el mutex** del objeto compartido.
  - La invocación de wait() detiene al hilo, pasa a "**WAITING** o **TIMED_WAITING**", **lo pone en una cola de espera asociada al objeto (cola wait del objeto)**, y **libera el mutex del objeto**.
  - El método wait() puede provocar una InterruptedException.
- notify(): **Notifica** a un hilo que invocó wait() sobre el mismo objeto y que está en la cola de espera del objeto (cola wait), **que ya puede continuar**.
  - Saca un hilo de la cola de espera del objeto (**cola wait**) pasándolo al estado "***RUNNABLE***", y se bloquea (***BLOCKED***) hasta conseguir el mutex del objeto para continuar su ejecución.
  - Si hay más de un hilo en la cola de espera (cola wait), notify() reactivará solo a uno de ellos. **El orden en que se desbloquean los hilos en un objeto de bloqueo vuelve a ser indeterminista**.
  - Una vez re-obtenido el mutex del objeto, el hilo que salió de la lista de espera (cola wait) continuará la ejecución del método en la instrucción siguiente a la llamada a wait().
- notifyAll(): Notifica a todos los hilos puestos en espera para el mismo objeto (cola wait) que ya pueden continuar.
- **Este modelo de notificación es indirecto**



El hilo que invoca notify() no tiene ninguna referencia del hilo que está en espera por haber invocado wait() sobre el mismo objeto. Cuando un hilo invoca a notify(), otro hilo (no se sabe cual) de los que están en espera es reactivado (pasa a "***RUNNABLE***").

Con notifyAll() se reactivan, volverán "***RUNNABLE***" todos los hilos que estaban bloqueados en la cola de espera del objeto. Sin embargo el mutex, solo podrán tomarlo de uno en uno.

Dos **problemas clásicos** que permiten ilustrar la necesidad de sincronizar y comunicar hilos son:

- El problema del **Productor-Consumidor**. Que permite modelar situaciones en las que se divide el trabajo entre los hilos. Modela el acceso simultáneo de varios hilos a una estructura de datos u otro recurso, de manera que unos hilos producen y almacenan los datos en el recurso y otros hilos (consumidores) se encargan de eliminar y procesar esos datos.
- El problema de los **Lectores-Escritores**. Permite modelar el acceso simultáneo de varios hilos a una base de datos, fichero u otro recurso, unos queriendo leer y otros escribir o modificar los datos.

```
synchronized(objBloqueo)
{
	while(!condiciónParaPoderSeguir)
	{
		try
		{
			// Espera que la condición cambie y otro hilo avise
			objBloqueo.wait()
		}
		catch (InterruptedException e)
		{
			...
		}
	}

	// Si el hilo ha llegado hasta aquí, significa que o bien al principio
	// o bien tras haber realizado una o más esperas y haber sido notificado
	// de cambios por parte de otros hilos, la condición se ha cumplido

	// Además ha conseguido el bloqueo del monitor para poder continuar
	// dentro del bloque synchronized
	realizarOperacion();


	if(condiciónParaQueOtrosSigan)
	{
		//objetoBloqueo.notify(); o objetoBloqueo.notifyAll()
	}
}
```

La realidad es esta: Debes usar notifyAll() cuando no sabes exactamente quién se va a despertar y si el que se despierta es capaz de avanzar.

Si usas notify() (que despierta a **UN** solo hilo elegido al azar por el sistema operativo) corres el riesgo de sufrir el problema de la **Notificación Perdida**, lo que lleva a un **Deadlock** (bloqueo mortal), donde todos los hilos se quedan durmiendo para siempre.

Veamos un ejemplo de esto último. Tenemos una fábrica con un almacén pequeño (capacidad 1). Tenemos Múltiples productores y un consumidor.

El Problema con notify(). Ejemplo FabricaPeligrosa:

1. Imagina que el almacén está **LLENO**.
2. Un **Productor A** intenta entrar, ve que está lleno, y hace wait().
3. Un **Productor B** intenta entrar, ve que está lleno, y hace wait().
    - *(Ahora tenemos dos productores durmiendo).*
4. Llega un **Consumidor**, se lleva el producto y deja el almacén **VACÍO**.
5. El Consumidor llama a notify() (singular).
    - Su intención es despertar a un Productor para que rellene el hueco.
    - **Pero...** en Java no puedes elegir a quién despiertas.
    - Imagina que por mala suerte, despierta al **Productor A**.
6. El **Productor A** rellena el hueco y llama a notify().
    - **Situación desastrosa….**
    - La intención del **Productor A** es despertar a un Consumidor para que se lleve el producto.
    - Pero el sistema operativo, caprichoso, decide despertar al **Productor B** (que estaba durmiendo desde el paso 3).
7. El **Productor B** se despierta, comprueba el while ("¿Está lleno?"), ve que SÍ (lo acaba de llenar A), y **se vuelve a dormir (**wait**)**.

Resultado: El **Productor A** ya trabajó y se fue a dormir. El **Productor B** se despertó y se volvió a dormir. El **Consumidor** nunca recibió el aviso y sigue durmiendo. **El sistema se congela aunque hay trabajo por hacer.**

En el 99% de los casos reales (como colas de tareas, pools de conexiones, o sistemas complejos), usa siempre notifyAll(). Es preferible desperdiciar unos microciclos de CPU despertando a hilos que se volverán a dormir, a tener un sistema bloqueado en producción a las horas inadecuadas de la madrugada.

8.6 Modelo productor-consumidor

Es un patrón que desacopla la generación de datos de su procesamiento, permitiendo que ocurran de forma concurrente y eficiente.

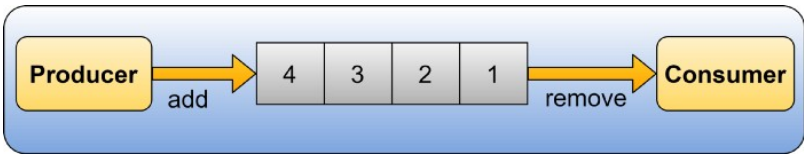
La forma de proceder de este modelo es el siguiente:

- El productor genera datos o tareas y los almacena (cola o búfer compartido).
- El consumidor extrae los datos y los procesa.

Se utilizan mecanismos de sincronización para evitar que:

- El productor sea más rápido que el consumidor y genere más información de la que el sistema pueda almacenar.
- El consumidor puede obtener los elementos producidos más de una vez, excediendo la producción del mismo (poder dejar la cuenta en números rojos en el ejemplo del banco, o que un lector pueda leer un libro antes de estar terminado).
- El consumidor intente extraer datos de un buffer vacío.

Todas estas circunstancias son las que conocemos como condiciones de carrera o **race conditions**. El esquema de este modelo Productor-Consumidor sería:

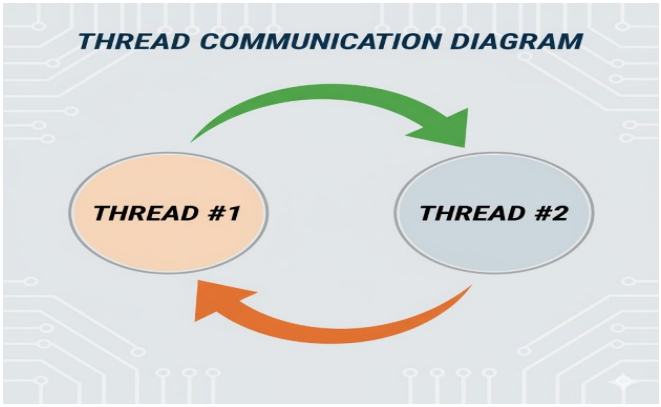


Ver PintorVendedor. Simulamos un hilo que pinta 30 cuadros, y otro hilo debe venderlos de uno en uno. Cuando se pinta un cuadro éste se deposita en un almacén, y hasta que no es retirado del almacén no se podrá depositar otro. Por tanto en el almacén de cuadros sólo puede haber un cuadro almacenado. El pintor pinta cuadros, pero no deposita ningún cuadro en el almacén hasta que éste está vacío. Observa que el recurso compartido por los hilos es el almacén, el pintor lo usa para poner cuadros y el vendedor para sacarlos.

Ver Buffer; ColaCadenas; Panaderia; Libro; LectorEscritor

9. Problema de interbloqueo (deadlock)

El **interbloqueo o** bloqueo mutuo (deadlock) consiste en que uno a más hilos, **se bloquean o esperan indefinidamente**.



¿Cómo se llega a una situación de interbloqueo? A dicha situación se llega:

- porque **cada hilo espera a que le llegue un aviso de otro hilo que nunca le llega**.
- porque todos **los hilos, de forma circular, esperan para acceder a un recurso**.

El problema del bloqueo mutuo, en las aplicaciones concurrentes, se podrá dar fundamentalmente cuando un hilo1 tiene un el lock del recurso1 y espera obtener el lock del recurso2, y cuando un hilo2 tiene el el lock del recurso2 y espera obtener el lock del recurso1.

Por tanto, ninguno puede proceder porque cada uno está esperando un recurso que tiene el otro.

Ver: InterbloqueoEjemplo encontrarás un ejemplo que produce interbloqueo. Observa que no finaliza la ejecución del programa. Habrá que finalizar manualmente el programa.

Solución:

1. Orden consistente de bloqueo

   Hay que asegurar que todos los hilos adquieran los bloqueos en el mismo orden. Ejemplo: siempre bloquear primero lock1 y luego lock2. Si siempre bloqueas **lock1** antes que **lock2**, el ciclo que provoca el deadlock no puede formarse.

   Ver: SolucionDeadLockOrdenConsistente

2. Bloqueo con timeout

   En el caso de que la solución anterior no sea posible, intentar obtener un bloqueo con **timeout**. De esta forma se asegura que los hilos no se bloqueen indefinidamente si no logran el **lock**. Usar tryLock() de ReentrantLock. Con el uso de ReentrantLock.tryLock(timeout, unit) se intenta obtener un lock y, si no se consigue en cierto tiempo, se libera lo que se tiene y se reintenta o se realiza otra acción (evita esperar indefinidamente). Así se rompe la espera circular indefinida.

   Ver:SolucionRobustaDeadLock

10. Mecanismos alternativos de Sincronización.

El uso de los métodos wait() y notify() (o notifyAll()) en la sincronización de hilos en Java puede ser poderoso, pero también presenta varios inconvenientes e incluso peligros si no se usan correctamente. Aquí tienes los principales problemas e inconvenientes:

1. Fácil de usar incorrectamente

wait() y notify() solo pueden usarse dentro de un bloque sincronizado, usando el mismo objeto como monitor. Si se olvidan las palabras clave synchronized, se lanza una excepción (IllegalMonitorStateException).

También es común equivocarse con el objeto sobre el que se hace wait() y el que hace notify().

```
synchronized(obj1) {
    obj2.wait(); // Error: el hilo no posee el monitor de obj2
}
```

2. Posibilidad de condiciones de carrera y pérdida de notificaciones

Si notify() se llama antes de que otro hilo entre en wait(), la notificación se pierde. Esto puede dejar al hilo esperando para siempre, lo que causa bloqueos difíciles de depurar.

3. Comportamiento no determinista y difícil de depurar

El orden de ejecución de los hilos y el momento en que se hacen las señales son no deterministas. Esto hace que los errores sean intermitentes: a veces el programa funciona, otras se bloquea.

4. Posibilidad de spurious wakeups

La Regla del while: Nunca se debe usar un if para comprobar la condición antes de hacer wait(). Usad siempre un while.

¿Por qué? Por los "Spurious Wakeups" (Despertares Espurios). A veces, el sistema operativo despierta a un hilo sin razón aparente. Si usas if, el hilo continuará y causará errores. Si usas while, al despertar volverá a comprobar si la condición es real.

Por todas estas razones en sistemas complejos, manejar múltiples monitores y condiciones con wait() y notify() se vuelve complejo y caótico. Por eso, se recomienda usar las clases más modernas del paquete java.util.concurrent: Semaphore, CountDownLatch, CyclicBarrier, Phaser, Exchanger, BlockingQueue...

10.1. Clase Semaphore

Los semáforos, introducidos por Dijkstra, son esencialmente variables simples, pero con un comportamiento especial que les permite controlar de forma segura el acceso a recursos compartidos en sistemas concurrentes.

Realiza un seguimiento de cuántas unidades de un recurso en particular están disponibles. Por tanto, si no hay unidades disponibles, se debe esperar. Los semáforos solo llevan la cuenta del número de recursos disponibles; no hacen un seguimiento de qué recursos específicos están disponibles.

La clase Semaphore del paquete java.util.concurrent, **permite definir un semáforo** para controlar el acceso a un recurso compartido.

Para **crear y usar un objeto Semaphore** haremos lo siguiente:

- Indicar al constructor Semaphore (int permisos) el total de permisos que se pueden dar para acceder al mismo tiempo al recurso compartido. Este valor coincide con el número de hilos que pueden acceder a la vez al recurso.

- Indicar al semáforo mediante el método acquire(), que queremos acceder al recurso, o bien mediante acquire (int permisosAdquirir) cuántos permisos se quieren consumir al mismo tiempo.

- Indicar al semáforo mediante el método release(), que libere el permiso, o bien mediante release(int permisosLiberar), cuantos permisos se quieren liberar al mismo tiempo.

- Hay otro constructor Semaphore (int permisos, boolean justo) que mediante el parámetro ***justo*** permite garantizar que el primer hilo en invocar acquire() y que está en espera, será el primero en adquirir un permiso cuando sea liberado. Esto es, garantiza el orden de adquisición de permisos, según el orden en que se solicitan.

También se puede usar otros métodos interesantes:

Método	Descripción
availablePermits()	Para saber en un momento dado cuántos permisos hay disponibles
drainPermits()	Quita y devuelve todos los permisos disponibles (útil para "vaciar" el semáforo).
tryAcquire()	Intenta adquirir 1 permiso y retorna true/false inmediatamente (no espera).
tryAcquire(long timeout, TimeUnit unit)	Intenta adquirir 1 permiso y espera hasta timeout.

Algunos usos típicos son: limitar conexiones a BD, número de descargas simultáneas, el número de accesos a un parking, coordinar productores y consumidores en buffers acotados…

Algunos aspectos importantes a destacar:

- No garantiza exclusión mutua por sí sólo salvo si lo configuras con permisos = 1 (binary semaphore).
- No es reentrante: si un hilo hace acquire() y luego intenta acquire() otra vez sin release(), se bloqueará.
- Poner el release() dentro de un finally.

Semáforos binarios

Tipo particular de semáforo en el que solo se permite un acceso simultáneo al recurso. Por lo tanto, es la forma de usar semáforos para garantizar la exclusión mutua. Solo un hilo podrá acceder a la sección crítica.

Semáforos de conteo

Son los semáforos en los que se permite más de un acceso simultáneo al recurso.

¿Desde dónde se deben invocar estos métodos? Esto dependerá del **uso de** Semaphore.

- Si se usa **para proteger secciones críticas,** la llamada a los métodos acquire() y release() se hará desde el recurso compartido o sección crítica, y el número de permisos pasado al constructor será 1.
- Si se usa **para comunicar hilos**, en este caso un hilo invocará al método acquire() y otro hilo invocará al método release() para así trabajar de manera coordinada. El número de permisos pasado al constructor coincidirá con el número máximo de hilos bloqueados en la cola o lista de espera para adquirir un permiso.

Ver BancoSimulacion; TerminalWeb; Pizzeria.

10.2. Clase CountDownLatch

Imaginad que estáis en el centro de control de la NASA preparándoos para lanzar un cohete. El director de vuelo no puede pulsar el botón de "Despegue" hasta que varios sistemas confirmen que están listos:

Combustible: OK Sistemas de navegación: OK. Sistemas de comunicaciones: OK.

La clase CountDownLatch del paquete java.util.concurrent es una utilidad de sincronización que permite que uno o más threads esperen hasta que otros threads finalicen su trabajo.

Por tanto, hace que uno o más threads esperen hasta que un conjunto de operaciones que se están ejecutando en otros hilos se completen.

El **funcionamiento esquemático** de CountDownLatch o "cuenta atrás de cierre" es el siguiente:

- Implementa un punto de espera que denominaremos "puerta de cierre", donde uno o más hilos esperan a que otros finalicen su trabajo.
- Los hilos que deben finalizar su trabajo se controlan mediante un contador que llamaremos "cuenta atrás".
- Cuando la "cuenta atrás" llega a cero se reanudará el trabajo del hilo o hilos puestos en espera.
- No será posible volver a utilizar la "cuenta atrás", es decir, **no se puede reiniciar.**

Los **aspectos más importantes** al usar la clase CountDownLatch son los siguientes:

- Al constructor *countDownLatch (int cuenta)* se le indica, mediante el parámetro "cuenta", el total de hilos que deben completar su trabajo, que será el valor de la "cuenta atrás".
- El hilo en curso desde el que se invoca al método *await()* esperará en la "puerta de cierre" hasta que la "cuenta atrás" tome el valor cero.

  También se puede utilizar el método *await (long tiempoespera, TimeUnit unit)*, para indicar que la espera será hasta que la cuenta atrás llegue a cero o bien se sobrepase el tiempo de espera especificado mediante el parámetro *tiempoespera.*
- La "cuenta atrás" se irá decrementando conforme los hilos que están realizando su tarea vayan finalizando. Para ello deberán llamar al método *countDown().*

  Es muy importante poner la llamada a este método dentro de un finally ya que si uno de los hilos a los que hay que esperar porque está realizando una tarea lanza una excepción, ese hilo no decrementará el contador (no hará CountDown), y el hilo que esté esperando se quedará esperando para siempre (**Deadlock**).

  Cuando la cuenta llega a cero, se libera el hilo o hilos que estaban en espera (habían llamado al método await), continuando su ejecución.

- No se puede reinicar o volver a utilizar la "cuenta atrás" una vez que ésta toma el valor cero.

- El método *getCount()* obtiene el valor actual de la "cuenta atrás" y generalmente se utiliza durante las pruebas y depuración del programa.

Ver ServidorBoot: Ejemplo de 1 hilo espera a varios hilos.

Ver CarreraCaballos: Ejemplo varios hilos esperan a 1 hilo.

Ver SumaMatriz: Ejemplo de 1 hilo espera a muchos. Ejemplo de cómo utilizar CountDownLatch para sumar todos los elementos de una matriz. Cada fila de la matriz es sumada por un hilo. Cuando todos los hilos han finalizado su trabajo, se ejecuta el procedimiento que realiza la suma global.

¿Por qué no usar join y sí await? Con join se espera a que el hilo muera (termine por completo), en cambio, con await se espera a que el contador llegue a 0 (el hilo puede seguir vivo).

Ver BusquedaSincronizada. Esto permite que un hilo establezca la posición encontrada solo si aún no estaba establecida. Todos los hilos consultan periódicamente si ya se encontró el número para terminar antes.

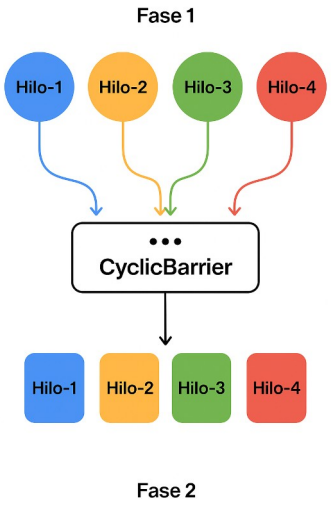
10.3. Clase CyclicBarrier

Imagina que organizas una excursión con 3 amigos. Acordáis encontraros en la gasolinera antes de salir a la carretera.

1.- Tú llegas y esperas.

2.- Tu amigo llega y espera.

3.- El último amigo llega... ¡y solo entonces todos arrancáis los coches a la vez!



En Java, CyclicBarrier (del paquete paquete java.util.concurrent) es ese punto de encuentro (la gasolinera). Es una herramienta de sincronización que permite que un conjunto de hilos espere hasta que todos lleguen a un punto común de ejecución antes de continuar, es decir, permite que uno o más threads se esperen hasta que todos ellos finalicen su trabajo.

Por lo tanto, es una herramienta muy útil cuando se realizan tareas agrupadas por fases, en las que antes de comenzar una nueva fase, deben haberse finalizado todas las tareas de la fase anterior.

El **funcionamiento esquemático** de CyclicBarrier es el siguiente:

- Implementa un punto de espera que llamaremos "**barrera**", donde cierto número de hilos esperan a que todos ellos finalicen su trabajo.
- Finalizado el trabajo de estos hilos, se dispara la ejecución de una determinada acción o bien el hilo interrumpido continúa su trabajo, permitiendo ejecutar un hilo de acción final cuando todos los hilos llegan a la barrera.
- La barrera se llama cíclica, porque se puede **volver a utilizar** después de que los hilos en espera han sido liberados tras finalizar todos su trabajo.

Los **aspectos más importantes** al usar la clase CyclicBarrier son los siguientes:

- Indicar al constructor *CyclicBarrier (int hilosAcceden)* el total de hilos que van a usar la barrera mediante el parámetro *hilosAcceden*. Por tanto, hasta que los hilos indicados como parámetro no lleguen a la barrera, no se podrá continuar.

- La barrera se dispara cuando llega el último hilo.

- Cuando se dispara la barrera, dependiendo del constructor, *CyclicBarrier(int hilosAcceden)* o *CyclicBarrier(int hilosAcceden, Runnable acciónBarrera)* se lanzará o no una acción, y entonces se liberan los hilos de la barrera. Esa acción puede ser realizada mediante cualquier objeto que implemente Runnable.

- El método principal de esta clase es *await()* que se utiliza para indicar que el hilo en curso ha concluido su trabajo y queda a la espera de que lo hagan los demás.

Otros métodos de esta clase que puedes utilizar son:

- El método *await (long tiempoespera, TimeUnit unit)* funciona como el anterior, pero en este caso el hilo espera en la barrera hasta que los demás finalicen su trabajo o se supere el *tiempoespera.* En caso de superar el tiempo de espera se lanza una excepción.

- El método *reset()* permite reiniciar la barrera a su estado inicial.

- El método *getNumberWaiting()* devuelve el número de hilos que están en espera en la barrera.

- El método *getParties()* devuelve el número de hilos requeridos para esa barrera.

Ver GameLobby: Imaginad un videojuego online. No puede empezar la partida hasta que los 4 jugadores hayan cargado el mapa. Si uno tiene un PC lento, los otros 3 deben esperar.

Ver SumaMatriz: Ejemplo realizado con CountDownLatch pero ahora resuelto con CyclicBarrier. Cada fila es sumada por un hilo. Cuando 5 de estos hilos finalizan su trabajo, se dispara un objeto que implementa Runnable para obtener la suma recaudada hasta el momento. Como la matriz del ejemplo tiene 10 filas, la suma de sus elementos se hará mediante una barrera de 10 hilos.

Ver MultiplesFases: Ejemplo de un procesamiento de datos que hay primero que descargar desde 3 fuentes diferentes, y una vez descargados, se analizan en paralelo.

Algunas precauciones:

- **Hilo Olvidado**

  Si inicializas la barrera con 4 *parties*, pero solo lanzas 3 hilos... esos 3 hilos esperarán eternamente (**Deadlock**). ¡Contad bien vuestros hilos!

- **Barrera rota (BrokenBarrierException)**

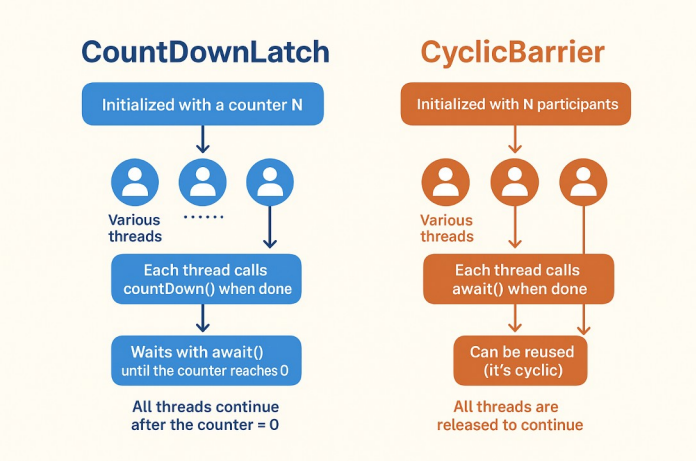
  La excepción BrokenBarrierException es la forma que tiene Java de decir: **"El pacto de espera se ha roto. Ya no tiene sentido seguir esperando porque uno de los hilos del grupo ha fallado o se ha ido".**

  Para que se lance esta excepción, suele ocurrir una de estas dos cosas mientras los hilos están esperando en .await():

  - **Interrupción (InterruptedException):** A uno de los hilos que estaba esperando se le ha llamado con algún tipo de interrupción no comprobada o con un thread.interrupt(). Como ese hilo deja de esperar, la barrera no puede completarse nunca. Para evitar que el resto de hilos se queden bloqueados infinitamente ("Deadlock"), la barrera se rompe y lanza BrokenBarrierException a todos los demás.
  - **Tiempo de espera agotado (TimeoutException):** Si usas **barrier.await(tiempo, unidad)** y el tiempo se agota antes de que lleguen todos, ese hilo salta por *timeout* y la barrera se rompe para el resto.

Esto permite **abortar la operación en grupo**. Si uno falla, todos deben detenerse. Por tanto, hay que asegurarse de capturar esta excepción para limpiar recursos si algo sale mal.

Ver TransaccionBancaria: Imagina un sistema bancario donde para aprobar una operación compleja, necesitamos la validación de 3 subsistemas: **Fraude, Saldo y Base de Datos**. Si uno falla (por ejemplo, el sistema de Fraude detecta una anomalía y lanza excepción), los otros dos deben cancelar su parte de la operación inmediatamente (hacer un rollback). No queremos descontar saldo si hay fraude.



Característica	CountDownLatch	CyclicBarrier
Uso	Un solo uso. Una vez llega a cero, no sirve más.	Reutilizable. Se resetea tras abrirse.
Acción	Un hilo espera a que otros eventos terminen (N espera a M).	Un grupo de hilos se espera *mutuamente* (N esperan a N).
Focus	"Espero a que termines tú".	"Esperamos a estar todos juntos".
Constructor	No acepta una acción final (Runnable).	Acepta una **barrierAction** al completarse.

10.3. Clase Phaser

Phaser es la primitiva de sincronización más flexible y moderna de Java. Funciona de modo similar al CountDownLatch y CyclicBarrier, pero con una diferencia fundamental: el número de hilos participantes puede cambiar dinámicamente a lo largo del tiempo.

Usos principales:

- Sincronización de tareas multihilo en fases.
- Procesamiento paralelo con dependencias entre etapas.
- Sistemas donde los hilos pueden llegar o retirarse dinámicamente.

```
Phaser phaser = new Phaser(); // sin participantes iniciales

Phaser phaser = new Phaser(3); // con 3 participantes iniciales, 3 hilos registrados desde el inicio
```

En el primer caso, se crea un Phaser vacío pero se pueden registrar hilos más tarde con register().

Método	Comportamiento	¿Bloquea el hilo?
register()	Registra el hilo y lo añade dinámicamente	No
arriveAndAwaitAdvance()	Indica que el hilo ha llegado y espera a los demás. "He terminado y espero aquí hasta que todos los demás terminen." (Comportamiento de Barrera).	Sí
arriveAndDeregister()	Indica que el hilo ha llegado y lo elimina dinámicamente. "He terminado y me retiro del grupo. No contéis conmigo para la siguiente fase."	No
awaitAdvance(int phase)	Espera a que el Phaser pase de la fase indicada a la siguiente.	Sí
onAdvance()	Método que se llama automáticamente al terminar todos los hilos una fase. Similar al Runnable de un CyclicBarrier. Puede sobreescribirse para implementar alguna lógica entre fases o para decidir cuándo el Phaser debe morir (terminar).	No

Método	Comportamiento
getPhase()	Devuelve el número de la fase actual.
getRegisteredParties()	Cuántos hilos están apuntados actualmente.
getArrivedParties()	Cuántos hilos ya han llegado a la barrera en esta fase.
IsTerminated()	Devuelve true si el Phaser ha finalizado y ya no admite sincronización.

Terminar un Phaser NO termina los hilos. Solo desactiva la sincronización. Si los hilos tienen más código debajo, lo ejecutarán sin esperarse unos a otros

Ver: Procesamiento de pedidos. Ejemplo de un sistema de procesamiento de pedidos en un e-commerce. Cada pedido pasa por tres fases:

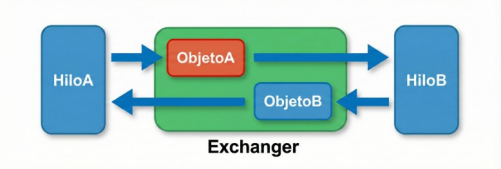
- Validación – Verificar datos del pedido.
- Empaquetado – Preparar el pedido para envío.
- Envío – Simular que se envía el pedido.

Cada fase debe completarse para todos los pedidos antes de pasar a la siguiente.

Ver: GameLoader. Simula la carga de un videojuego moderno. Tenemos 3 tareas que deben ocurrir en Fase 1 (Cargar assets, sonidos, mapa) y luego, una vez todas terminen, pasamos a la Fase 2 (Conectar al servidor, verificar perfil).

Ver: CarreraEliminación. Realiza una carrera de 3 rondas con 3 corredores. Si quedan participantes, se sigue. Si se llega a la fase 3 (o se van todos), terminamos.

10.4. Clase Exchanger



La clase Exchanger, del paquete java.util.concurrent, establece un **punto de sincronización donde se intercambian objetos entre dos hilos**. La clase Exchanger<V> es genérica, lo que significa que tendrás que especificar en <V> el tipo de objeto a compartir entre los hilos.

Existen dos **métodos** definidos en esta clase:

- exchange (V objeto)
- exchange (V objeto, long timeout, TimeUnit unit)

Ambos métodos exchange() permiten intercambiar objetos entre dos hilos. El hilo que desea obtener la información, esperará realizando una llamada al método exchange() hasta que el otro hilo sitúe la información utilizando el mismo método, o hasta que pase un periodo de tiempo establecido mediante el parámetro timeout.

El **funcionamiento,** tal y como puedes apreciar en la imagen anterior, sería el siguiente:

- Dos hilos (hiloA e hiloB) intercambiarán objetos del mismo tipo, objetoA y objetoB.
- El hiloA invocará a exchange (objetoA) y el hiloB invocará a exchange (objetoB).
- El hilo que procese su llamada a exchange (objeto) en primer lugar, se bloqueará y quedará a la espera de que lo haga el segundo. Cuando eso ocurra y se libere el bloqueo sobre ambos hilos, la salida del método exchange (objetoA) proporciona el objeto objetoB al hiloA, y la del método exchange (objetoB) el objeto objetoA al hiloB.

Te estarás preguntando ¿y **cuándo puede ser útil** Exchanger? Los intercambiadores se emplean típicamente cuando un hilo productor está rellenando una lista o buffer de datos, y otro hilo consumidor los está consumiendo.

De esta forma cuando el consumidor empieza a tratar la lista de datos entregados por el productor, el productor ya está produciendo una nueva lista. Precisamente, esta es la principal utilidad de los intercambiadores: que la producción y el consumo de datos, puedan tener lugar concurrentemente.

Ver: IntercambioSecretos

Algunos aspectos importantes:

- El Exchanger solo funciona con dos hilos. Si intentáis usarlo con tres, uno se quedará esperando eternamente (**deadlock**) o tendréis un comportamiento impredecible.

11. Concurrencia moderna en Java

Antes de Java 5, la única forma de gestionar la concurrencia era creando hilos manualmente (new Thread()) y coordinándolos con synchronized, wait() y notify().

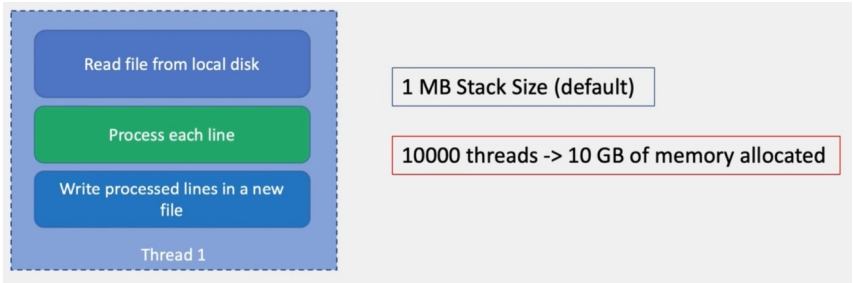
Sin embargo, el uso de Thread Pools (o el marco ExecutorService) es el estándar industrial hoy en día.

¿Por qué se debe preferir los pools de hebras sobre la gestión manual?

1. Rendimiento: El costo de crear vs. Reutilizar
Crear un hilo en Java no es gratis; es una operación costosa para el Sistema Operativo.

- **Gestión Manual (new Thread())**: Cada vez que creas un hilo, el sistema debe asignar memoria para la pila (stack) del hilo y realizar llamadas al sistema operativo. Si tu aplicación recibe miles de peticiones, crear y destruir miles de hilos consume muchísima CPU y memoria solo en "burocracia" administrativa.
- **Thread Pool:** Funciona bajo el principio de **reciclaje**. Las hebras se crean una vez y se mantienen vivas. Cuando una tarea termina, la hebra no muere; vuelve al "pool" a esperar la siguiente tarea. Esto elimina la latencia de creación.

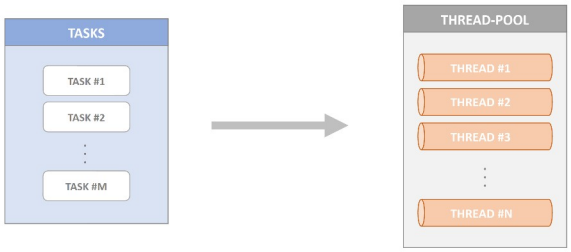
2. Estabilidad: Control de Recursos



Este es quizás el punto más crítico para aplicaciones en producción.

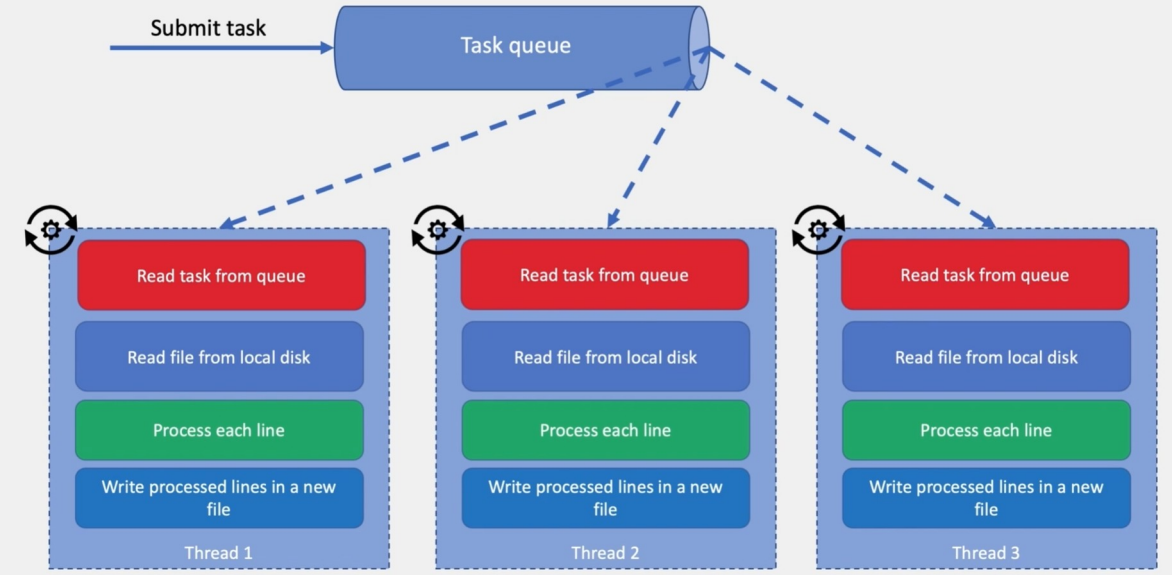
- **Gestión Manual**: Si creas un hilo por cada tarea (modelo *unbounded*), un pico de tráfico puede hacer que tu aplicación cree 10,000 hilos simultáneamente. Esto provocará un **OutOfMemoryError** o saturará la CPU con cambios de contexto (context switching), bloqueando el servidor por completo.
- **Thread Pool**: Te permite poner un **límite**. Puedes configurar un pool con un máximo de 50 hebras. Si llegan 100 tareas, 50 se ejecutan y las otras 50 esperan ordenadamente en una cola (**BlockingQueue**). El servidor no se cae, simplemente se mantiene ocupado a su máxima capacidad segura.

La solución a esto pasa por **reusar Threads** para lograr que cada Thread pueda realizar muchas tareas y no solo una tarea y ser destruida.



Para lograrlo se establece un límite de Threads en la aplicación y algún mecanismo que permita asignar diferentes tareas a un mismo Thread.

Por tanto, básicamente lo que se debe decir es "aquí hay un trabajo", y el mecanismo decide qué hilo lo ejecuta, cuándo y cómo. Esto hace el código mucho más limpio y mantenible.



En la imagen se puede ver el funcionamiento. Los Threads esperan a que se le asignen tareas existentes en la cola. Una vez asignada la tarea, el thread correspondiente la ejecuta.

Para esto Java nos ofrece los Executors y los ThreadPoolExecutors. El primero de ellos simplifica la creación de pools sin necesidad de configurar todos sus parámetros manualmente. En cambio, el segundo es mucho más flexible ya que puedes configurar prácticamente todo, por ejemplo:

- número mínimo y máximo de hilos.
- tamaño de la cola de tareas.
- política de rechazo (RejectedExecutionHandler).
- tiempo de espera de hilos inactivos.
- tipo de cola (LinkedBlockingQueue, SynchronousQueue, etc.).

De hecho, los Executors usan internamente los ThreadPoolExecutors.

En sistemas críticos de alta carga, no se usa Executors. Se instancia ThreadPoolExecutor manualmente para limitar el tamaño de la cola de espera.

Por tanto, ambos proporcionan un marco para crear y administrar hilos facilitando la labor de:

- **Creación de hilos**: Proporciona varios métodos para grupos de hilos (Thread Pool) que la aplicación puede usar para ejecutar tareas al mismo tiempo.
- **Gestión de hilos**: No hay que preocuparse por si los hilos están activos, ocupados o destruidos antes de enviar una tarea para su ejecución. No es necesario hacer llamadas a los métodos start ni join ya que la propia herramienta se encarga de la creación, ejecución y parada de la hebra.
- **Envío y ejecución de tareas**: Proporcionan métodos para enviar tareas para su ejecución y para decidir cuándo se ejecutarán dichas tareas (programar las tareas o ejecución periódica de tareas).
- **Devolución de datos**: Permite que las tareas asignadas a las hebras puedan devolver datos.

11.1. Executors

Java a través de su API java.util.concurrent proporciona varias interfaces relacionadas con Executors que es una factoría que consta de varios métodos estáticos para la creación de diferentes tipos de Pools:

- ExecutorService: Subinterfaz de Executor que a través de los métodos execute(Runnable) o submit(Runnable) ejecuta la tarea pasada como parámetro cuando le sea posible en uno de los hilos del Pool.

  Por tanto, a este método hay que llamarlo una vez por cada tarea que deba ser ejecutada por el pool.

  El ExecutorService sería como un gerente de recursos humanos al que le damos trabajos (la tarea) y él decide qué trabajador (hilo) la ejecuta y cuándo.

- ScheduledExecutorService: Subinterfaz de ExecutorService que proporciona funcionalidades para programar la ejecución de tareas.

Los métodos estáticos de Executors son los siguientes:

Método	Descripción	Uso
newFixedThreadPool (númeroHilos)	Crea un Pool con un número fijo de hilos. Una vez que todos los hilos han sido creados y están ocupados, si llegan nuevas tareas, estas se irán a la cola de tareas a esperar que algún hilo quede disponible. Cuando una tarea finaliza, el hilo queda disponible para que se le asigne una nueva tarea.	Cuando conoces la carga de trabajo y quieres evitar saturar la CPU (ej. procesamiento de imágenes).
newCachedThreadPool(númeroHilos)	Igual que el anterior con la salvedad de que si un hilo está unos 60 segundos sin ser usado, automáticamente será eliminado.	Muchas tareas, cortas y asíncronas.

newSingleThreadExecutor()	El pool consta de un solo hilo. Por lo tanto, solo se puede ejecutar una tarea al mismo tiempo. Si el hilo se destruye por alguna circunstancia, vuelve a ser creado.	Adecuado cuando se tienen tareas secuenciales.
newScheduledThreadPool(número Hilos)	Permite programar la ejecución de las tareas. Tiene el inconveniente de que si el tamaño del Pool es pequeño y las tareas tienen larga duración, las nuevas tareas pueden retrasarse.	Usarlo cuando sean necesario ejecutar tareas en un instante de tiempo o en intervalos regulares, teniendo siempre presente la posibilidad de retraso.

Por defecto, la aplicación no finalizará a pesar de que los hilos no estén ejecutando ninguna tarea porque estas hayan ya terminado. Para evitar esto Java proporciona varios métodos:

- shutdown(): Deja de aceptar nuevas tareas. Espera a que las tareas que ya están corriendo y las que están en la cola terminen. En ese momento ExecutorService se cerrará.
- shutdownNow(): Intenta detener las tareas activas (envía una interrupción Thread.interrupt()) y devuelve una lista de las tareas que estaban en cola y no se llegaron a ejecutar. En este caso no hay garantías de que las tareas en ejecución se detengan o realicen su ejecución hasta el final.
- awaitTermination(long tiempo, TimeUnit unidad): Bloquea el hilo de ejecución principal hasta que ExecutorService se cierre por completo o hasta que se consuma el tiempo de espera indicado. Este método suele ser llamado después de cualquiera de los dos anteriores. Devuelve **true** si el ejecutor terminó o **false** si transcurrió el tiempo sin que haya terminado.

Ver: SingleExecutor, FixedExecutor.

Respecto a la planificación de tareas Java ofrece dos formas de repetición de las tareas:

- schedule(Runnable, delay, unit)

  Uso: "Ejecuta esto UNA sola vez dentro de xx unidades de tiempo". No es periódico. Es un disparo retardado.

- scheduleAtFixedRate(Runnable, initial, period, unit)

  Intenta mantener una frecuencia estricta. El reloj empieza a contar desde el inicio de la ejecución anterior.
  Ejemplo: Si configuras cada 10s, las tareas se lanzarán a las 12:00:00, 12:00:10, 12:00:20…

  Riesgo: Si tu tarea tarda 15s en ejecutarse y el periodo es de 10s, las tareas se empezarán a solapar o amontonar en la cola, consumiendo recursos rápidamente.

- scheduleWithFixedDelay(Runnable, initial, delay, unit)

  Lógica: Garantiza un "descanso". El reloj empieza a contar desde el final de la ejecución anterior.

  Ejemplo: Tarea tarda 5s, descanso (delay) de 10s. Inicio: 00s -> Fin: 05s -> (espera 10s) -> Inicio: 15s.

  Ventaja: Es más seguro para tareas pesadas o de duración variable, ya que garantiza que el sistema respire.

Por tanto, la diferencia entre estos dos últimos estriba en que el FixedRate ignora cuánto tarda la tarea (mientras sea menor al periodo) y dispara rigurosamente a :00, :03, :06. Mientras que el FixedDelay espera a terminar (:03) y suma el delay (4s), lanzando el siguiente a las :07.

Ver: ScheduledExecutor; ScheduledRateDelayExecutor

11.2. Callable y Future

Las formas previamente explicadas sobre la creación de hebras se caracterizan por no devolver un resultado cuando el método run() finaliza. Para este tipo de situaciones se usará la interfaz Callable, es decir, se usará para definir tareas que devuelven un resultado.

Callable al igual que Runnable es una interfaz funcional usada para representar tareas que pueden ser ejecutadas por Threads o por ExecutorService, aunque hay algunas diferencias entre ambas interfaces:

- Runnable: Representa una tarea que no devuelve un resultado y no puede lanzar una excepción comprobada. La tarea asociada puede ser ejecutada usando los métodos execute o submit de un objeto ExecutorService.

- Callable: Representa una tarea que devuelve un resultado de tipo Future y puede lanzar una excepción comprobada. Se debe ejecutar usando el método submit de un objeto ExecutorService.

```
Callable<String> tarea=()->{ return “Hola desde Callable”;};

ExecutorService executor=Executors.newSingleThreadExecutor();
Future<String> future=executor.submit(tarea);

String result=future.get(); //Se bloquea hasta que el resultado esté disponible.
System.out.println(result);
```