

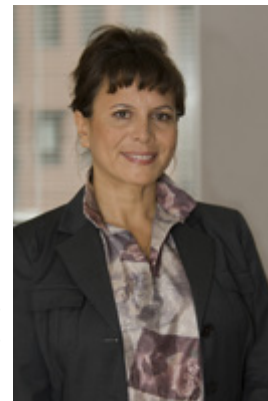
Programación multihilo.

Caso práctico

Esta mañana **Ada** está reunida con **Juan** y **Ana**, y están estudiando los detalles de un nuevo proyecto que ha llegado a la empresa, y que consiste en la programación de un sistema de control climático de invernaderos.

La aplicación que desarrollen debe consultar periódicamente el estado de una serie de sensores repartidos por el complejo.

Ada y **Juan** están valorando las diferentes posibilidades de enfocar y resolver el problema: mediante bucles, mediante la ejecución simultánea de múltiples procesos o mediante hilos. Después de debatir sobre los inconvenientes que puede presentar la resolución de este problema mediante bucles o mediante múltiples procesos concurrentes, han llegado a la conclusión de que la opción más óptima y eficiente es utilizar multihilo.



Ana, apasionada de la programación concurrente desde que la estudió en el ciclo DAM, recuerda que éste fue precisamente uno de los ejemplos que vieron en clase como introducción a la programación de hilos y ahora va a tener la oportunidad de participar en el desarrollo de una aplicación real.

Aunque **Ana** está muy ilusionada, está también algo nerviosa pues debe recordar algunos aspectos de la programación multihilo que tiene un poco olvidados. **Juan** le dice que no se preocupe, que entre los dos harán el repaso, pues él también necesita afianzar algunos conceptos, además en cualquier momento podrán consultar a **Ada** que es toda una experta en el tema.



Materiales formativos de FP Online propiedad del Ministerio de Educación, Cultura y Deporte.

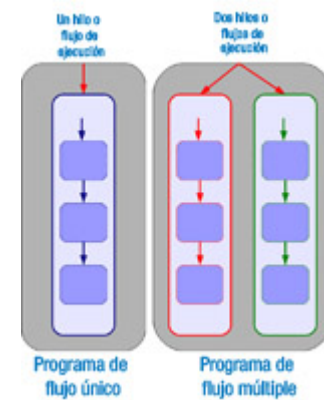
[Aviso Legal](#)

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 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 simultánea o concurrente.

La **programación multihilo** o multithreading 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.

En el siguiente enlace puedes ver un ejemplo muy intuitivo de lo que se puede conseguir mediante programas de flujo múltiple o multihilo, en concreto mediante **threads** en Java.

[Ejemplo de programa con varios hilos o flujos de ejecución.](#)

Para saber más

Consulta el siguiente enlace para conocer las motivaciones e historia de la programación multihilo, así como algunas de sus aplicaciones.

[Historia y aplicaciones de la programación multihilo.](#)



Autoevaluación

Señala si la afirmación siguiente es verdadera o falsa:

La programación multihilo es una manera de hacer programación concurrente.

Verdadero. ☐ Falso. ☐

2.- Conceptos sobre hilos.

Caso práctico

Juan, que ya colaboró hace unos años junto a **Ada** en el desarrollo de una aplicación de gestión de servidores utilizando programación multihilo, sabe que esta técnica de programación es muy poderosa, pero también peligrosa, por lo que es muy importante tener claros todos los conceptos relacionados con ella.

Aunque **Juan** sabe que **Ana** ya ha estudiado en el ciclo DAM este tema le pregunta —¿Qué te parece si comenzamos repasando los conceptos básicos sobre hilos, las ventajas que aportan sobre los procesos y cuando interesa utilizarlos?

Ana responde: —¡Me parece genial!, un repaso me vendrá muy bien, además... ¡me muero de ganas por empezar.



Pero ¿qué es realmente un hilo? Un **hilo**, denominado también **subproceso**, 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.

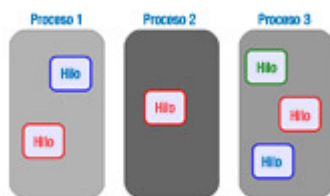
Cuando se ejecuta un programa, el Sistema Operativo crea un proceso y también crea su primer hilo, **hilo primario**, el cual puede a su vez crear hilos adicionales.

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.

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

- ✓ Un hilo no puede existir independientemente de un proceso.
- ✓ 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.



Para saber más

No dejes de visitar el siguiente enlace con la definición de Hilo de ejecución en la wikipedia.

[Hilo de ejecución.](#)



Autoevaluación

Señala si la afirmación siguiente es verdadera o falsa:

Los hilos siempre están asociados a un proceso en particular .

Verdadero. ☐ Falso. ☐

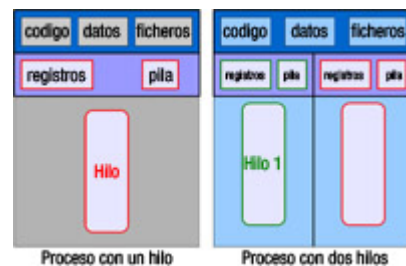
2.1.- Recursos compartidos por los hilos.

Un hilo lleva asociados los siguientes elementos:

- ✔ Un identificador único.
- ✔ Un contador de programa propio.
- ✔ Un conjunto de registros.
- ✔ Una pila (variables locales).

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

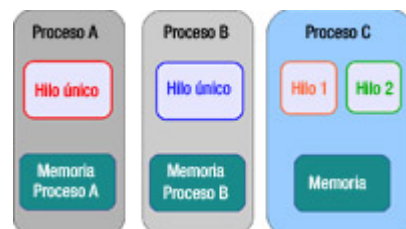
- ✔ Código.
- ✔ Datos (como variables globales).
- ✔ Otros recursos del sistema operativo, como los ficheros abiertos y las señales.



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.

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, o que se bloquean con facilidad y 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? Eso es lo que vamos a ver precisamente en el siguiente apartado.



Autoevaluación

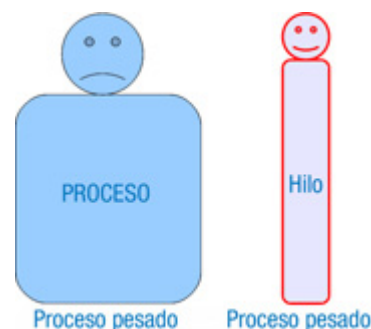
Señala la opción correcta. Un hilo puede compartir con otros hilos:

- ☐ Código y pila.
- ☐ Código, datos y registros.
- ☐ Registros y pila.
- ☐ Código, datos y ficheros.

2.2.- Ventajas y uso de hilos.

Como consecuencia de compartir el espacio de memoria, los hilos aportan las siguientes **ventajas sobre los procesos**:

- ✓ Se consumen menos recursos en el lanzamiento, y la ejecución de un hilo que en el lanzamiento y ejecución de un proceso.
- ✓ Se tarda menos tiempo en crear y terminar un hilo que un proceso.
- ✓ La conmutación entre hilos del mismo proceso 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, como veremos al final de la unidad lo resolveremos mejor con un pool de hilos.

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

Los hilos son más frecuentes de lo que parece. De hecho, todos los programas con interfaz gráfico son multihilo porque los eventos y las rutinas de dibujo de las ventanas corren en un hilo distinto al principal. Por ejemplo en Java, AWT o la biblioteca gráfica Swing usan hilos.



Autoevaluación

Señala si la afirmación siguiente es verdadera o falsa:

La conmutación entre procesos es mas rápida que la conmutación entre hilos.

Verdadero. ☐ Falso. ☐

3.- Multihilo en Java. Librerías y clases.

Caso práctico

Esta mañana **Ana** le pregunta a **Juan** —¿En qué lenguaje vamos a desarrollar la aplicación? —a lo que **Juan** responde— La vamos a desarrollar en Java, porque este lenguaje da soporte a los hilos desde el propio lenguaje y en las últimas versiones se han incluido una serie de clases que facilitan bastante la programación multihilo.

—¡Fenomenal!, —responde **Ana**, y añade— precisamente Java es el lenguaje que he estudiado en el Ciclo Formativo.



Tal y como ha comentado Juan, **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 [JavaSE 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.

Para saber más

En este enlace puedes consultar el tutorial oficial (en inglés) de Oracle sobre Hilos en Java.

[Tutorial de Oracle 'Hilos en Java'.](#)

En este otro enlace te proporcionamos un tutorial básico sobre hilos en Java, en el que puedes consultar diferentes ejemplos resueltos.

[Tutorial con ejemplos de 'Hilos en Java'.](#)

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.
- ✓ **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()`.

En el siguiente enlace tienes una **tabla resumida de la clase thread**, en español, que incluye los métodos más comunes para gestionar y controlar hilos.

[Tabla resumida de la clase thread.](#)

Para saber más

En el siguiente enlace dispones de todos los métodos incluidos en la clase `Thread` que es conveniente que conozcas.

[Documentación oficial de la clase thread.](#)



Autoevaluación

Señala si la afirmación siguiente es verdadera o falsa:

La clase `Thread` permite proporcionar hilos en una aplicación Java e incluye gran parte de los métodos más comunes para gestionarlos.

Verdadero. ☐ Falso. ☐

3.2.- Utilidades de concurrencia del paquete `java.util.concurrent`.

El paquete `java.util.concurrent` incluye una serie de clases que facilitan enormemente el desarrollo de aplicaciones multihilo y aplicaciones complejas, ya que están concebidas para utilizarse como bloques de diseño.

java.util.concurrent
.atomic | **.locks**

Concretamente estas utilidades están dentro de los siguientes **paquetes**:

- ✔ `java.util.concurrent`. En este paquete están definidos los siguientes elementos:
 - ➡ **Clases de sincronización**. Semaphore, CountdownLatch, CyclicBarrier y Exchanger.
 - ➡ **Interfaces para separar la lógica de la ejecución**, como por ejemplo Executor, ExecutorService, Callable y Future.
 - ➡ **Interfaces para gestionar colas de hilos**. BlockingQueue, LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, PriorityBlockingQueue y DelayQueue.
- ✔ `java.util.concurrent.atomic`. Incluye un conjunto de clases para ser usadas como variables atómicas en aplicaciones multihilo y con diferentes tipos de dato, por ejemplo AtomicInteger y AtomicLong.
- ✔ `java.util.concurrent.locks`. Define una serie de clases como uso alternativo a la cláusula `synchronized`. En este paquete se encuentran algunas interfaces como por ejemplo Lock, ReadWriteLock.

A lo largo de esta unidad estudiaremos las clases e interfaces más importantes de este paquete.

Citas para pensar

"Vale más saber alguna cosa de todo, que saberlo todo de una sola cosa".

Blaise Pascal

Para saber más

En este enlace puedes consultar detalladamente las utilidades proporcionadas por el paquete `java.util.concurrent`.

[Documentación oficial del paquete `java.util.concurrent`.](#)

4.- Creación de hilos.

Caso práctico

Ana recuerda que existen diferentes maneras de crear hilos en Java, y que en algunas ocasiones sólo era posible utilizar una de ellas.

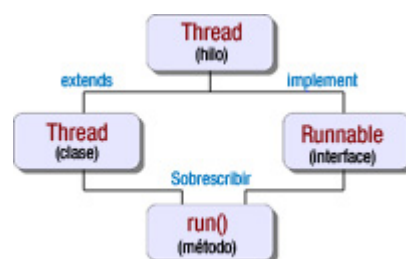
Ana le pregunta a **Juan**: —¿Te importa si vemos algunos ejemplos sobre las distintas formas de crear hilos en Java?, he traído mis apuntes de DAM y creo que nos pueden servir. **Juan** le responde —Me parece perfecto, ¡vamos a ello!



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`.
- ✓ Mediante 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. Entonces, **¿cuando utilizar uno u otro procedimiento?**

- ✓ 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.
- ✓ Implementar `Runnable` siempre es posible, es el procedimiento más general y también el más flexible.

Por ejemplo, piensa en la programación de **applets**, cualquiera de ellos tiene que heredar de la clase **java.applet.Applet**; y en consecuencia ya no puede heredar de **thread** si se quiere utilizar hilos. En este caso, no queda más remedio que crear los hilos implementando **Runnable**.

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.

El siguiente ejemplo muestra lo que te acabamos de comentar, siempre hay un hilo que ejecuta el método **main()**, y por defecto, este hilo se llama "main". Observa que para saber qué hilo se está ejecutando en un momento dado, el hilo en curso, utilizamos el método **currentThread()** y que obtenemos su nombre invocando al método **getName()**, ambos de la clase **thread**.

[Código para mostrar el hilo principal de un programa.](#) (0.01 MB)



Autoevaluación

Señala si la afirmación siguiente es verdadera o falsa:

Los hilos en Java siempre se implementan mediante una clase que hereda de la clase `thread` .

Verdadero. ☐ Falso. ☐

4.1.- Creación de hilos extendiendo la clase Thread.

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

- ✓ Crear una nueva clase que herede de la clase thread.
- ✓ Redefinir en la nueva clase el método `run()` con el código asociado al hilo. Las sentencias que ejecutará el hilo.
- ✓ 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).

En el siguiente ejemplo puedes ver los pasos indicados anteriormente para la creación de un hilo extendiendo la clase **thread**. El hilo que se crea (objeto **thread** hilo1) imprime un mensaje de saludo. Para simplificar el ejemplo se ha incluido el método **main()** que inicia el programa en la propia clase **Saludo**.

```
public class Saludo extends Thread {
    //clase que extiende a Thread
    public void run() {
        // se redefine el método run() con el código asociado al hilo
        System.out.println("Saludo desde un hilo extendiendo Thread");
    }
    public static void main(String args[]) {
        Saludo hilo=new Saludo();
        //se crea un objeto Thread, el hilo hilo1
        hilo.start();
        //llamada a start() y pone en marcha el hilo hilo1
    }
}
```

Código para crear hilo extendiendo thread. (0.01 MB)

Creación de un hilo extendiendo thread. (0.01 MB)



Autoevaluación

Señala si la afirmación siguiente es verdadera o falsa:

Al crear un hilo mediante la clase thread, no es necesario sobrescribir en la nueva clase el método `run()`.

Verdadero. ☐ Falso. ☐



4.2.- Creación de hilos mediante la interfaz Runnable.

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

- ✓ Declarar una nueva clase que implemente a Runnable.
- ✓ Redefinir (o sombrear) en la nueva clase el método `run()` con el código asociado al hilo. Lo que queremos que haga el hilo.
- ✓ Crear un objeto de la nueva clase.
- ✓ Crear un objeto de la clase `thread` pasando como argumento al constructor, el objeto cuya clase tiene el método `run()`. 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).

El siguiente ejemplo muestra cómo crear un hilo implementado **Runnable**. El hilo que se crea (objeto **thread hilo1**) imprime un mensaje de saludo, como en el caso anterior.

```
public class Saludo implements Runnable {
    // Clase que implementa a Runnable
    public void run() {
        // Se define el método run() con el código asociado al hilo
        System.out.println("Saludo desde un hilo creado con Runnable");
    }
    public static void main(String args[]) {
        Saludo hilo = new Saludo();
        // Se crea un objeto Thread (el hilo hilo1) pasando como argumento
        // el constructor de la clase Saludo
        Thread hilo1 = new Thread(hilo);
        hilo1.start();
        // Se llama al método start() del hilo hilo1
    }
}
```

[Código para crear un hilo implementando Runnable \(0.01 MB\)](#)

[Creación de un hilo implementando la interfaz Runnable \(0.01 MB\)](#)



Autoevaluación

Señala la opción correcta. Un hilo creado mediante Runnable es:

- ☐ No necesita ser un objeto `thread`.
- ☐ Es un objeto `thread` a cuyo constructor se le pasa como argumento un objeto de la clase que implementa `Runnable` y define un método `run()`.
- ☐ No necesita método `run()`.
- ☐ Es un objeto de la clase que implementa `Runnable` y en la que se define el método `run()`.

5.- Estados de un hilo.

Caso práctico

Hasta el momento, **Juan** ha podido comprobar que **Ana** se desenvuelve muy bien con todos los aspectos repasados sobre hilos en Java, pero lo más complicado comienza ahora. Juan le pregunta a **Ana**, —¿Tienes claro los diferentes estados que puede tener un hilo a lo largo de su vida y cómo se pasa de uno a otro? A lo que **Ana** le responde, —Mas o menos..., pero me vendría bien dar un repaso, porque además quiero recordar que había unos cuantos métodos para pasar de un estado a otro cuyo uso es peligroso y además ya están considerados obsoletos debido a los problemas que ocasionan.



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

De manera general, los **diferentes estados** en los que se puede encontrar un hilo son los siguientes:



- ✔ **Nuevo** (new): se ha creado un nuevo hilo, pero aún no está disponible para su ejecución.
- ✔ **Ejecutable** (runnable): el hilo está preparado para ejecutarse. Puede estar **Ejecutándose**, siempre y cuando se le haya asignado tiempo de procesamiento, o bien que no esté ejecutándose en un instante determinado en beneficio de otro hilo, en cuyo caso estará **Preparado**.
- ✔ **No Ejecutable o Detenido** (no runnable): el hilo podría estar ejecutándose, pero hay alguna actividad interna al propio hilo que se lo impide, como por ejemplo una espera producida por una operación de **Entrada/Salida** (E/S). Si un hilo está en estado "No Ejecutable", no tiene oportunidad de que se le asigne tiempo de procesamiento.
- ✔ **Muerto o Finalizado** (terminated): el hilo ha finalizado. La forma natural de que muera un hilo es finalizando su método `run()`.

El método **getState()** de la clase **Thread**, permite obtener en cualquier momento el estado en el que se encuentra un hilo. Devuelve por tanto: **NEW**, **RUNNABLE**, **NO RUNNABLE** o **TERMINATED**.

En la imagen anterior, puedes ver algunos de los métodos que permiten obtener cada uno de esos estados. Los veremos con más detalle en los siguientes apartados.



Autoevaluación

Señala si la afirmación siguiente es verdadera o falsa:

A un hilo en estado 'No Ejecutable' nunca se le asignará tiempo de procesamiento.

Verdadero. ☐ Falso. ☐

5.1.- Iniciar un hilo.

Cuando se crea un nuevo hilo o **thread** mediante el método **new()**, 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 los ejemplos anteriores, recuerda que teníamos el código **hilo1.start()**; que precisamente se encargaba de iniciar el hilo representado por el objeto **thread hilo1**.

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()** y lo ejecuta como un subproceso nuevo e independiente.

Es por esto último que cuando se invoca a **start()** se suele decir que el hilo está "corriendo" ("running"), pero recuerda que esto no significa que el hilo esté ejecutándose en todo momento, ya que un hilo "Ejecutable" puede estar "Preparado" o "Ejecutándose" 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 como subproceso independiente.
- ✔ 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 las instrucciones del programa).

En el siguiente recurso didáctico puedes ver un programa que define dos hilos, contruidos cada uno de ellos por los procedimientos vistos anteriormente. Cada hilo imprime una palabra 5 veces. Observa que si ejecutas varias veces el programa, el orden de ejecución de los hilos no es siempre el mismo y que no influye en absoluto el orden en el que se inician con **start()** (el orden de ejecución de los hilos es no-determinístico).

Nota. puede que tengas que aumentar el número de iteraciones (número de palabras que imprime cada hilo) para apreciar las observaciones indicadas anteriormente.

[Resumen textual alternativo](#)

[Código del proyecto que crea y ejecuta varios hilos.](#)(0.01 MB)



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa

El método **start()** se encarga de llamar al método **run()** del hilo para que se ejecute como un subproceso independiente.

Verdadero. ☐ Falso. ☐

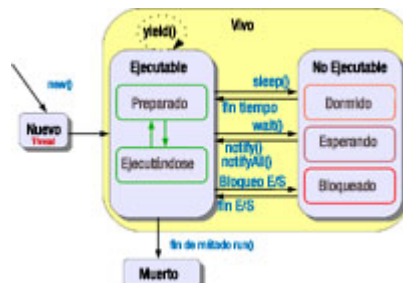
5.2.- Detener temporalmente un hilo.

¿Qué significa que un hilo se ha detenido temporalmente? Significa que **el hilo ha pasado al estado "No Ejecutable"**.

Y ¿cómo puede **pasar un hilo al estado "No Ejecutable"**? Un hilo pasará al estado "No Ejecutable" o "Detenido" por alguna de estas circunstancias:

- ✓ **El hilo se ha dormido.** 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 a "Preparado".
- ✓ **El hilo está esperando.** El hilo ha detenido su ejecución mediante la llamada al método `wait()`, y no se reanuda, pasará a "Ejecutable" (en concreto "Preparado") hasta que se produzca una llamada al método `notify()` o `notifyAll()` por otro hilo. Estudiaremos detalladamente estos métodos de la clase `Object` cuando veamos la sincronización y comunicación de hilos.
- ✓ **El hilo se ha bloqueado.** 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 "Ejecutable", en concreto "Preparado".

En la siguiente imagen puedes ver un esquema con los diferentes métodos que hacen que un hilo pase al estado "No Ejecutable", así como los que permiten salir de ese estado y volver al estado "Ejecutable".



El método **`suspend()`** (actualmente en desuso o deprecated) también permite detener temporalmente un hilo, y en ese caso se reanuda mediante el método **`resume()`** (también en desuso). No debes utilizar estos métodos, de la clase **`Thread`** ya que no son seguros y provocan muchos problemas. Te lo indicamos simplemente porque puede que encuentres programas que aún utilizan estos métodos.

Para saber más

Consulta este enlace para conocer algunos métodos de la clase **`Thread`** declarados obsoletos o no recomendados, como **`suspend()`** y **`resume()`** y el porqué.

[Métodos obsoletos de manipulación de hilos.](#)



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

Un hilo que se ha dormido con `sleep()` estará en el estado "No Ejecutable" hasta que se invoque al método `notify`.

Verdadero. ☐ Falso. ☐

5.3.- Finalizar un hilo.

La forma natural de que muera o finalice un hilo es cuando **termina de ejecutarse su método `run()`**, pasando al **estado 'Muerto'**.

Una vez que el hilo ha muerto, no lo puedes iniciar otra vez con **`start()`**. Si en tu programa deseas **realizar otra vez el trabajo desempeñado por el hilo**, tendrás que:

- ✔ Crear un nuevo hilo con `new()`.
- ✔ Iniciar el hilo con `start()`.



Y ¿hay alguna forma de **comprobar si un hilo no ha muerto**?

No exactamente, pero puedes utilizar el método **`isAlive()`** de la clase **`thread`** para comprobar si un hilo está vivo o no. Un hilo se considera que está **vivo (`alive`)** desde la llamada a su método **`start()`** hasta su muerte. **`isAlive()`** devuelve verdadero (`true`) o falso (`false`), según que el hilo esté vivo o no.

Cuando el método **`isAlive()`** devuelve:

- ✔ `False`: sabemos que estamos ante un nuevo hilo recién "creado" o ante un hilo "muerto".
- ✔ `True`: sabemos que el hilo se encuentra en estado "ejecutable" o "no ejecutable".

El método **`stop()`** de la clase **`thread`** (actualmente en desuso) también finaliza un hilo, pero es poco seguro. No debes utilizarlo. Te lo indicamos aquí simplemente porque puede que encuentres programas utilizando este método.

En el siguiente ejemplo, te proporcionamos un 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 **`thread.join()`** que espera hasta que el hilo muere .

[Ejemplo uso de `isAlive\(\)` y `getState\(\)`](#). (0.01 MB)



Autoevaluación

Señala las opciones correctas. Un hilo se considera vivo :

- ☐ Desde el momento en que se crea con `new()`
- ☐ Cuando está en el estado 'Ejecutable'.
- ☐ Cuando está en el estado 'No Ejecutable'.
- ☐ Cuando está en cualquier estado diferente a 'Muerto'

[Mostrar Información](#)

Para saber más

[Las razones por las que `stop\(\)` es inseguro y se desaconseja.](#)

5.4.- Ejemplo. Dormir un hilo con sleep.

¿Por qué puede interesar dormir un hilo? Pueden ser diferentes las razones que nos lleven a dormir un hilo durante unos instantes. En este apartado veremos un ejemplo en el que si no durmiéramos unos instantes al hilo que realiza un cálculo, no le daría tiempo al hilo que dibuja el resultado a presentarlo en pantalla.

¿Cómo funciona el método `sleep()`?

El método `sleep()` de la clase `Thread` recibe como argumento el tiempo que deseamos dormir el hilo que lo invoca. Cuando transcurre el tiempo especificado, el hilo vuelve a estar "Ejecutable" ("Preparado") para continuar ejecutándose.



Hay dos formas de llamar a este método.

- ✓ La primera le pasa como argumento un entero (positivo) que representa milisegundos:

```
sleep(long milisegundos)
```

- ✓ La segunda le agrega un segundo argumento entero (esta vez, entre 1 y 999999), que representa un tiempo extra en nanosegundos que se sumará al primer argumento:

```
sleep(long milisegundos, int nanosegundos)
```

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

El siguiente recurso didáctico ilustra la necesidad de dormir un hilo en una aplicación que muestra como avanza un marcador gráfico desde 0 hasta 20. Podrás comprobar que si no utilizamos un hilo auxiliar y lo dormimos, no podremos apreciar como se va incrementando el marcador.

[Resumen textual alternativo](#)

[Código del proyecto que duerme hilo en marcador gráfico.](#) (0.01 MB)



Autoevaluación

Relaciona cada invocación del método `sleep()` con el tiempo correspondiente.

Ejercicio de relacionar

Invocación	Relación	Tiempo dormido
<code>sleep(2000)</code>	<input type="checkbox"/>	1. Medio segundo.
<code>sleep(1,100000)</code>	<input type="checkbox"/>	2. Dos segundos.
<code>sleep(500)</code>	<input type="checkbox"/>	3. Un segundo y una diezmilésima.

Enviar

6.- Gestión y planificación de hilos.

Caso práctico

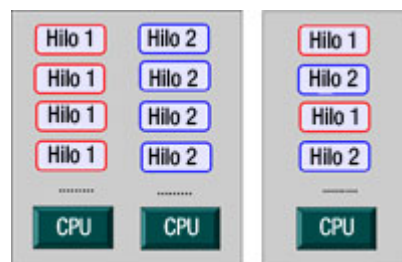
Ana está algo preocupada y pensativa, porque sabe que programar con hilos es un tema muy peliagudo y hay que tener claro cómo los gestiona Java para conseguir que el programa realmente sea eficiente y se ejecute de igual forma en cualquier SO. Después de un descanso en el parque, ha decidido repasar este tema y así adelantar trabajo para mañana. Realmente está muy ilusionada con el proyecto, pues nota como **Juan** y **Ada** apoyan las sugerencias que va haciendo respecto a la aplicación.



La ejecución de hilos se puede realizar mediante:

- ✓ **Paralelismo**. En un sistema con múltiples CPU, cada CPU puede ejecutar un hilo diferente.
- ✓ **Pseudoparalelismo**. Si no es posible el paralelismo, una CPU es responsable de ejecutar múltiples hilos.

La ejecución de múltiples hilos en una sola CPU requiere la planificación de una secuencia de ejecución (sheduling).



El **planificador de hilos de Java** (Sheduler) utiliza un algoritmo de secuenciación de hilos denominado fixed priority scheduling que está basado en un sistema de prioridades relativas, de manera que el algoritmo secuencia la ejecución de hilos en base a la prioridad de cada uno de ellos.

El **funcionamiento** del algoritmo es el siguiente:

- ✓ El hilo elegido para ejecutarse, siempre es el hilo "Ejecutable" de prioridad más alta.
- ✓ Si hay más de un hilo con la misma prioridad, el orden de ejecución se maneja mediante un algoritmo por turnos (round-rubin) basado en una cola circular FIFO (Primero en entrar, primero en salir).
- ✓ Cuando el hilo que está "ejecutándose" pasa al estado de "No Ejecutable" o "Muerto", se selecciona otro hilo para su ejecución.
- ✓ La ejecución de un hilo se interrumpe, si otro hilo con prioridad más alta se vuelve "Ejecutable". El hecho de que un hilo con una prioridad más alta interrumpa a otro se denomina "**planificación apropiativa**" ('preemptive sheudling').

Pero la responsabilidad de ejecución de los hilos es del Sistemas Operativos sobre el que corre la JVM, y **Sistemas Operativos distintos manejan los hilos de manera diferente**:

- ✓ En un **Sistema Operativo que implementa** time-slicing (subdivisión de tiempo), el hilo que entra en ejecución, se mantiene en ella sólo un micro-intervalo de tiempo fijo o cuanto (quantum) de procesamiento, de manera que el hilo que está "ejecutándose" no solo es interrumpido si otro hilo con prioridad más alta se vuelve "Ejecutable", sino también cuando su "cuanto" de ejecución se acaba. Es el patrón seguido por Linux, y por todos los Windows a partir de Windows 95 y NT.
- ✓ En un **Sistema Operativo que no implementa** time-slicing el hilo que entra en ejecución, es ejecutado hasta su muerte; salvo que regrese a "No ejecutable", u otro hilo de prioridad más alta alcance el estado de "Ejecutable" (en cuyo caso, el primero regresa a "preparado" para que se ejecute el segundo). Es el

patrón seguido en el Sistema Operativo Solaris.



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa

La ejecución de un hilo sólo puede interrumpirse por otro hilo de mayor prioridad o, porque el hilo que está 'Ejecutándose' pase a estado 'No ejecutable'.

Verdadero. ☐ Falso. ☐

Para saber más

En el siguiente enlace encontrarás información sobre HyperThreading, marca registrada de la empresa Intel que permite a los programas multihilo procesarlos en paralelo dentro de un único procesador.

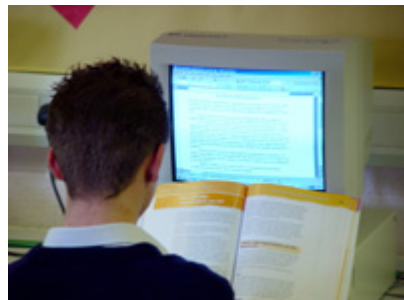
[Información sobre HyperThreading.](#)

6.1.- Prioridad 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()**. Modifica la prioridad de un hilo. Este método toma como argumento un entero entre 1 y 10, que indica la nueva prioridad del hilo.

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.

En el siguiente enlace tienes un 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. Observa que se usa también el método **yield()** del que hablaremos en el siguiente apartado.

[Código del proyecto que gestiona prioridades de hilos.](#) (0.01 MB)

Debes conocer

Los hilos demonio (daemon) son hilos que se ejecutan en **segundo plano**. No dejes de visitar este enlace para conocer más sobre hilos demonio.

[Información sobre los hilos demonio.](#)



Autoevaluación

Señala si la afirmación siguiente es verdadera o falsa:

Cuando se crea un nuevo hilo, éste tendrá prioridad 5.

Verdadero. ☐ Falso. ☐

Para saber más

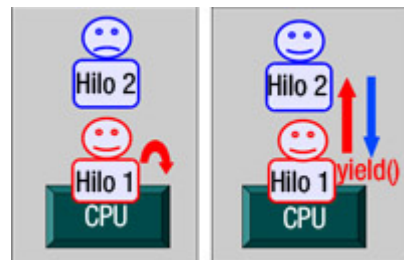
En el siguiente enlace puedes obtener más información y descargar varios ejemplos sobre la planificación y prioridad de los **Threads**.

[Ejemplos sobre planificación y prioridad de hilos.](#)

6.2.- Hilos egoístas y programación expulsora.

¿De verdad existen los hilos egoístas? En un Sistema Operativo que no implemente time-slicing puede ocurrir que un hilo que entra en "ejecución" no salga de ella hasta su muerte, de manera que no dará ninguna posibilidad a que otros hilos "preparados" entren en "ejecución" hasta que él muera. Este hilo se habrá convertido en un **hilo egoísta**.

Por ejemplo, supongamos la siguiente situación en un Sistema Operativo sin time-slicing:



- ✓ La tarea asociada al método `run()` de un hilo consiste en imprimir 100 veces la palabra que se le pasa al constructor más el número de orden.
- ✓ Se inician dos hilos en `main()`, uno imprimiendo "Azul" y otro "Rojo".
- ✓ El hilo que sea seleccionado en primer lugar por el planificador se ejecutará íntegramente, por ejemplo el que imprime "Rojo" 100 veces. Después se ejecutará el otro hilo, tal y como muestra la imagen parcial de la derecha.
- ✓ Este hilo tiene un **comportamiento egoísta**.

```
Salida - psp02_Hilo_Egoista (run)
run:
Rojo1
Rojo2
Rojo3
Rojo4
Rojo5
Rojo6
```

En un Sistema Operativo que si implemente time-slicing la ejecución de esos hilos se entremezcla, tal y como muestra la imagen parcial de la izquierda, lo cual indica que no hay comportamiento egoísta de ningún hilo, esto es, el Sistema operativo combate los hilos egoístas.

```
Salida - psp02_Hilo_Egoista (run)
run:
Rojo1
Azul1
Rojo2
Azul2
Rojo3
Rojo4
```

El proyecto completo lo puedes descargar desde el siguiente enlace.

[Ejemplo de posible hilo egoísta.](#) (0.01 MB)

Según lo anterior, un mismo programa Java se puede ejecutar de diferente manera según el Sistema Operativo donde corra. Entonces, ¿que pasa con la [portabilidad](#) de Java? Java da solución a este problema mediante lo que se conoce como **programación expulsora** a través del método `yield()` de la clase `java.lang.thread`:

- ✓ `yield()` hace que un hilo que está "ejecutándose" pase a "preparado" **para permitir que otros hilos de la misma prioridad puedan ejecutarse**.

Sobre el **método `yield()`** y el egoísmo de los threads debes tener en cuenta que:

- ✓ El funcionamiento de `yield()` no está garantizado, puede que después de que un hilo invoque a `yield()` y pase a "preparado", éste vuelva a ser elegido para ejecutarse.
- ✓ No debes asumir que la ejecución de una aplicación se realizará en un Sistema Operativo que implementa time-slicing.
- ✓ En la aplicación debes incluir adecuadamente llamadas al método `yield()`, incluso a `sleep()` o `wait()`, si el hilo no se bloquea por una Entrada/Salida.

El siguiente código muestra la forma de invocar a **yield()** dentro del método **run()** de un hilo. Ten en cuenta que si la invocación se hace desde un hilo Runnable tendrás que poner **thread.yield()**;

```
public void run() {  
    //se imprime 100 veces el valor del color = i  
    for(int i=1;i<=100;i++)  
        System.out.println(color + i);  
    yield(); //llamada a yield()  
}
```

yield() o Thread.yield()

Código de invocación al método yield(). (0.01 MB)



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

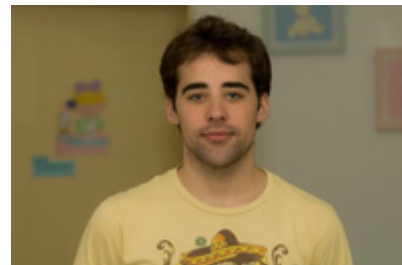
La invocación de `yield()` permite combatir los hilos egoístas en un SO sin time-slicing.

Verdadero. ☐ Falso. ☐

7.- Sincronización y comunicación de hilos.

Caso práctico

Antonio, compañero de estudios de **Ana**, ha quedado esta tarde con ella para preguntarle algunas dudas que le han surgido en un programa Java que está desarrollando en unas prácticas. El programa que desarrolla **Antonio** utiliza hilos que necesitan acceder y manipular las mismas variables, y no consigue que funcione correctamente. **Ana** le dice a **Antonio**: —¿Te has acordado de que debes sincronizar los hilos de tu programa cuando éstos acceden a recursos compartidos y qué puede que tus hilos deban comunicarse para realizar su trabajo de forma coordinada?



¡No te preocupes!, ayer estuve repasando todo esto para el proyecto que estoy realizando en la empresa BK Programación y verás que no es tan complicado. Además con las nuevas utilidades de sincronización incluidas en Java, ¡todo es más fácil!

Los ejemplos realizados hasta ahora utilizan hilos independientes; 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.

¿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`.

Por otra parte, Java proporciona en el paquete **`java.util.concurrent`** varias **clases de sincronización** que permiten la sincronización y comunicación entre diferentes hilos de una aplicación multithreading, como son: **`Semaphore`, `CountDownLatch`, `CyclicBarrier` y `Exchanger`**.

En los siguientes apartados veremos ejemplos de todo esto.

Citas para pensar

"Daría todo cuanto sé por la mitad de lo que ignoro".

René Descartes

7.1.- Información compartida entre hilos.

Las **secciones críticas** son aquellas secciones de código que no pueden ejecutarse concurrentemente, pues en ellas se encuentran los recursos o información que comparten diferentes hilos, y que por tanto pueden ser problemáticas.



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 siguiente enlace te facilitamos este ejemplo detallado.

[Resumen textual alternativo](#)

[Código del proyecto sobre el problema de los jardines.](#) (0.01 MB)

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.

Para saber más

En el siguiente enlace encontrarás un ejemplo clásico, 'El problema del barbero durmiente' que ilustra el problema de la condición de carrera entre hilos.

[Ejemplo del problema 'condición de carrera' entre hilos.](#)

7.2.- Monitores. Métodos synchronized.

En Java, un monitor es una porción de código protegida por un **mutex** o lock. Para **crear un monitor** en Java, 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 y en espera.
- ✓ Cuando el hilo finaliza la ejecución del método `synchronized`, los hilos en espera de poder ejecutarlo se desbloquearán. El planificador Java seleccionará a uno de ellos.



En el siguiente documento encontrarás una explicación detallada sobre el funcionamiento de un monitor Java.

[Explicación del funcionamiento de un monitor Java.](#) (0.08 MB)

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 `jardín`, `jardín.incrementaCuenta()`, ningún otro hilo podrá ejecutarlo.

```
public synchronized void incrementaCuenta() {
    //Método que incrementa en 1 la variable cuenta
    System.out.println("Hilo " + Thread.currentThread().getName()
        + " entra en jardín");
    //Incrementa el hilo que entra en el método
    cuenta++;
    System.out.println("Hilo " + Thread.currentThread().getName()
        + " sale del jardín");
    //Incrementa cada vez que un hilo entra y sale el número de visitantes
}
```

En el siguiente enlace dispones del proyecto completo con los dos métodos sincronizados.

[Ejemplo 'problema de los jardines' sincronizado.](#) (0.01 MB)

En el siguiente recurso didáctico dispones de otro sencillo ejemplo que simula el acceso simultáneo de 4 terminales a un servidor utilizando monitores Java.

[Resumen textual alternativo](#)

[Código del proyecto que sincroniza método.](#) (0.01 MB)

Debes conocer

En el siguiente enlace encontrarás el significado de la siguiente afirmación: "Los monitores java son re-entrantes".

[¿Qué significa que los monitores java son re-entrantes?.](#)



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

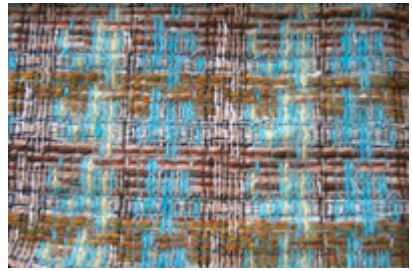
Para crear un monitor Java hay que marcar un bloque de código con la palabra `Synchronized`.

Verdadero. ☐ Falso. ☐

7.3.- Monitores. Segmentos de código synchronized.

Hay casos en los que no se puede, o no interesa 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. La forma de resolver esta situación es **poner las llamadas a los métodos que se quieren sincronizar dentro de segmentos sincronizados** de la siguiente forma:

```
synchronized (objeto){ // sentencias segmento; }
```



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

- ✓ El objeto que se pasa al segmento, es el objeto donde está el método que se quiere sincronizar.
- ✓ Dentro del segmento se hará la llamada al método que se quiere sincronizar.
- ✓ 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.

En el ejemplo del problema de los jardines, aplicando este procedimiento, habría que sincronizar el objeto que denominaremos **jardín** 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 acceso
    for (int i = 0; i < 10; i++) { //Se ejecuta 10 accesos
        //Segmento que se sincroniza
        synchronized (jardín) {
            jardín.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 **jardín**, aún cuando su clase no contiene ningún segmento ni método **synchronized**.

En el siguiente enlace dispones del ejemplo completo con los cambios que acabamos de indicar.

[Código del proyecto que sincroniza objeto en 'Problema de los Jardines'.](#) (0.01 MB)

Debes **tener en cuenta** que:

- ✓ Declarar un método o segmento de código como sincronizado ralentizará la ejecución del programa, ya que la **adquisición y liberación de monitores genera una sobrecarga**.
- ✓ Siempre que sea posible, por legibilidad del código, es **mejor sincronizar métodos completos**.
- ✓ Al declarar bloques **synchronized** puede aparecer un **nuevo problema, denominado interbloqueo** (lo veremos más adelante).

Muchos métodos de las clases predefinidas de Java ya están sincronizados. Por ejemplo, el método de la clase **Component** de Java **AWT** que agrega un objeto **MouseListener** a un **Component** (para que **MouseEvents** se registren en el **MouseListener**) está sincronizado. Si compruebas el código fuente de **AWT** y **Swing**, encontrarás que el prototipo de este método es: `public synchronized void addMouseListener(MouseListener l)`



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

Solo se puede sincronizar un bloque de código, si éste está dentro de en un método sincronizado.

Verdadero. ☐ Falso. ☐

7.4.- Comunicación entre hilos 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 (pasa a "no ejecutable"), el cual no se reanuda hasta que otro hilo notifique que ha ocurrido lo esperado.
- ✓ `wait(long tiempo)`. Como el caso anterior, solo que ahora el hilo también puede reanudarse (pasar a "ejecutable") si ha concluido el tiempo pasado como parámetro.
- ✓ `notify()`. Notifica a uno de los hilos puestos en espera para el mismo objeto, que ya puede continuar.
- ✓ `notifyAll()`. Notifica a todos los hilos puestos en espera para el mismo objeto que ya pueden continuar.

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

¿Cómo funcionan realmente estos métodos? En el siguiente documento tienes la explicación de cómo funcionan estos métodos y un ejemplo de su uso.

[Explicación del funcionamiento de los métodos de comunicación entre hilos.](#) (0.15 MB)

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

- ✓ El problema del **Productor-Consumidor**. Del que has visto un ejemplo anteriormente y 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.

En el siguiente recurso tienes el ejemplo del problema de los lectores-escritores, resuelto mediante los métodos `wait()` y `notify()` vistos anteriormente.

[Resumen textual alternativo](#)

[Código del ejemplo de los Lectores_Escritores.](#) (0.01 MB)

En el siguiente enlace puedes ver otro ejemplo de Productor_Consumidor.

[Ejemplo del Productor_Consumidor.](#)

Debes conocer

En el siguiente enlace encontrarás información y un ejemplo de lo que se conoce como el **búfer circular**.

[Ejemplo del búfer circular.](#)



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

Los métodos `wait()` y `notify()` se pueden invocar desde cualquier parte del código de la aplicación.

Verdadero. ☐ Falso. ☐

7.5.- El problema del 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 hilo entra en un bloque **synchronized**, y a su vez llama a otro bloque **synchronized**, o bien al utilizar clases de **java.util.concurrent** que llevan implícita la exclusión mutua.

En el siguiente enlace puedes consultar un artículo y ejemplo detallado sobre el interbloqueo.

[El problema de los interbloqueos.](#)

En el siguiente enlace encontrarás un ejemplo que produce interbloqueo. Observa que no finaliza la ejecución del programa y que en la barra de estado del IDE NetBeans (a la derecha) aparece la indicación de programa bloqueado. Habrá que finalizar manualmente el programa.

[Código del ejemplo de Interbloqueo.](#) (0.01 MB)

Otro problema, menos frecuente, es la **inanición** (starvation), que consiste en que un hilo es desestimado para su ejecución. Se produce cuando un hilo no puede tener acceso regular a los recursos compartidos y no puede avanzar, quedando bloqueado. Esto puede ocurrir porque el hilo nunca es seleccionado para su procesamiento o bien porque otros hilos que compiten por el mismo recurso se lo impiden.

Está claro que los programas que desarrollemos deben estar exentos de estos problemas, por lo que habrá que ser cuidadosos en su diseño.

Debes conocer

La interface **Lock** del paquete **java.util.concurrent** permite resolver algunos problemas de concurrencia como el interbloqueo. En el siguiente enlace encontrarás una explicación y ejemplo de uso.

[La interface **Lock** para resolver problemas de concurrencia.](#)

Para saber más

En este enlace, puedes consultar los principios del interbloqueo, ejemplos, y estrategias de prevención.

[Principios y ejemplos de interbloqueo.](#)



Autoevaluación

Señala la opción correcta. El interbloqueo:

- ☐ Es una propiedad de los hilos.
- ☐ Puede aparecer cuando se sincronizan bloques de código.
- ☐ En Java nunca puede aparecer.
- ☐ Se produce cuando un hilo entra en un bloque synchronized.

7.6.- La clase Semaphore.

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()` 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.



¿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.

En el siguiente enlace tienes un ejemplo del uso de **Semaphore** para proteger secciones críticas o recursos compartidos. Es el ejemplo que vimos del acceso simultáneo de 4 terminales a un Servidor, pero resuelto ahora con la clase **Semaphore** en vez de con **synchronized**.

[Código del proyecto 'Accesos a un servidor' con Semaphore.](#) (0.01 MB)

En el siguiente enlace tienes un ejemplo del uso de **Semaphore** para comunicar hilos. Es el ejemplo de los Lectores-Escritores. Se inician 5 hilos lectores y 2 escritores, como en el ejemplo que resolvimos con `wait()` y `notify()`.

[Código del proyecto 'Lectores-escritores' con Semaphore.](#) (0.01 MB)

En el siguiente enlace, puedes ver otro ejemplo con la clase **Semaphore**. En este ejemplo se usa **Semaphore** para controlar que en primer lugar se ejecuten siempre dos hilos concretos y en en segundo lugar otros.

[Ejemplo que sincroniza hilos mediante la clase Semaphore.](#)



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

El método `acquire()` de la clase `Semaphore` permite indicar que se quiere consumir un recurso.

Verdadero. ☐ Falso. ☐

Para saber más

En el siguiente enlace puedes consultar otros métodos de la clase **Semaphore**.

Clase **Semaphore** en la web oficial.

7.7.- La 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 x).`
- ✓ `exchange(V x, 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 búfer 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.

En el siguiente enlace encontrarás un ejemplo donde un hilo productor se encarga de rellenar una cadena de diez caracteres (o sea, una lista de 10 caracteres) mientras que un hilo consumidor la imprime.

[Código del proyecto ejemplo de uso de Exchanger.](#) (0.01 MB)



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

La clase `Exchanger` siempre intercambia datos de tipo cadena.

Verdadero. ☐ Falso. ☐

Para saber más

En este otro enlace encontrarás otro ejemplo ilustrativo de cómo utilizar la clase **Exchanger**.

[Ejemplo de la clase Exchanger.](#)

7.8.- Las clase `CountDownLatch`.

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**.

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 reanuda el trabajo del hilo o hilos interrumpidos y puestos en espera.
- ✓ No será posible volver a utilizar la "cuenta atrás", es decir, **no se puede reiniciar**. Si fuera necesario reiniciar la "cuenta atrás" habrá que pensar en utilizar la clase `CyclicBarrier`.



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 mediante la invocación del método `countDown()`, y cuando ésta llega al valor cero se libera el hilo o hilos que estaban en espera, continuando su ejecución.
- ✓ No se puede reiniciar o volver a utilizar la "cuenta atrás" una vez que ésta toma el valor cero. Si esto fuera necesario, entonces debemos pensar en utilizar la clase `CyclicBarrier`.
- ✓ El método `getCount()` obtiene el valor actual de la "cuenta atrás" y generalmente se utiliza durante las pruebas y depuración del programa.

En el siguiente enlace puedes ver un 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.

[Código del proyecto ejemplo de uso de `CountDownLatch` Ejemplo.](#) (0.01 MB)



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

El método `await()` de la clase `CountDownLatch` permite al hilo que lo invoca esperar en la "puerta de cierre" hasta que la "cuenta atrás" tome el valor cero.

Verdadero. ☐ Falso. ☐

7.9.- La clase **CyclicBarrier**.

La clase **CyclicBarrier** del paquete **java.util.concurrent** es una utilidad de sincronización que permite que uno o más **threads** se esperen hasta que todos ellos finalicen su trabajo.

El **funcionamiento esquemático** de **CyclicBarrier** o "barrera cíclica" 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.
- ✔ 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, y también **se puede reiniciar**.

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`. Este número sirve para disparar la barrera.
- ✔ 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`.
- ✔ 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

En el siguiente enlace puedes ver un ejemplo parecido al anterior, pero ahora resuelto con **CyclicBarrier**. Cada fila de la matriz ahora representa los valores recaudados por un cobrador. 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 5 hilos y que se utilizará por tanto de forma cíclica dos veces.

[Código del proyecto ejemplo de uso de CyclicBarrier](#). (0.01 MB)



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

El método `await()` de la clase `CyclicBarrier` permite indicar que el hilo en curso ha concluido su trabajo y queda a la espera de que lo hagan los demás.

Verdadero. ☐ Falso. ☐

Para saber más

En el siguiente enlace puedes consultar otros métodos de la clase **CyclicBarrier**.

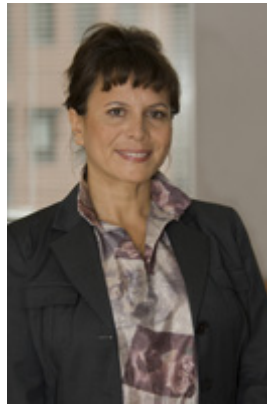
Clase **CyclicBarrier** en la web oficial.

8.- Aplicaciones multihilo.

Caso práctico

Ada está muy satisfecha con la marcha del proyecto sobre invernaderos. Sabe que es un proyecto muy ambicioso y complicado, pero **Juan y Ana** lo están desarrollando realmente bien. Hoy se ha reunido con ellos para recordarles que no deben pasar por alto ningún detalle en la aplicación, que no olviden utilizar siempre que puedan las utilidades de concurrencia que proporciona Java, lo cual repercutirá en la eficiencia y facilidad de mantenimiento de la aplicación, y que por supuesto no olviden que la documentación en este tipo de aplicaciones es crucial.

Juan le dice a **Ada**: —No te preocupes, estamos depurando la aplicación y desde luego no olvidaremos nada de eso.



Una aplicación multihilo debe reunir las siguientes **propiedades**:

- ✔ **Seguridad.** La aplicación no llegará a un estado inconsistente por un mal uso de los recursos compartidos. Esto implicará sincronizar hilos asegurando la exclusión mutua.
- ✔ **Viveza.** La aplicación no se bloqueará o provocará que un hilo no se pueda ejecutar. Esto implicará un comportamiento no egoísta de los hilos y ausencia de interbloqueos e inanición.

La **corrección de la aplicación** se mide en función de las propiedades anteriores, pudiendo tener:

- ✔ **Corrección parcial.** Se cumple la propiedad de seguridad. El programa termina y el resultado es el deseado.
- ✔ **Corrección total.** Se cumplen las propiedades de seguridad y viveza. El programa termina y el resultado es el correcto.

Por tanto, al desarrollar una aplicación multihilo habrá que **tener en cuenta los siguientes aspectos**:

- ✔ La situación de los hilos en la aplicación: hilos independientes o colaborando/compitiendo.
 - Independientes. No será necesario sincronizar y/o comunicar los hilos.
 - Colaborando y/o compitiendo. Será necesario sincronizar y/o comunicar los hilos, evitando interbloqueos y esperas indefinidas.
- ✔ Gestionar las prioridades, de manera que los hilos más importantes se ejecuten antes.
- ✔ No todos los Sistemas Operativos implementan time-slicing.
- ✔ La ejecución de hilos es no-determinística.

Por lo general, las aplicaciones multihilo son más difíciles de desarrollar y complicadas de depurar que una aplicación secuencial o de un solo hilo; pero si utilizamos las librerías que aporta el lenguaje de programación, podemos obtener algunas ventajas:

- ✔ **Facilitar la programación.** Requiere menos esfuerzo usar una clase estándar que desarrollarla para realizar la misma tarea.
- ✔ **Mayor rendimiento.** Los algoritmos utilizados han sido desarrollados por expertos en concurrencia y rendimiento.

- ✓ **Mayor fiabilidad.** Usar librerías o bibliotecas estándar, que han sido diseñadas para evitar interbloqueos (deadlocks), cambios de contexto innecesarios o condiciones de carrera, nos permiten garantizar un mínimo de calidad en nuestro software.
- ✓ **Menor mantenimiento.** El código que generemos será más legible y fácil de actualizar.
- ✓ **Mayor productividad.** El uso de una API estándar permite mejor coordinación entre desarrolladores y reduce el tiempo de aprendizaje.

Teniendo en cuenta esto último, cuando vayas a desarrollar una aplicación multihilo debes hacer uso de las utilidades que ofrece el propio lenguaje. Esto facilitará la puesta a punto del programa y su depuración.



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

La viveza de un aplicación multihilo significa que la aplicación no hará un mal uso de los recursos compartidos.

Verdadero. ☐ Falso. ☐

8.1.- Otras utilidades de concurrencia.

Además de las utilidades de sincronización que hemos visto en apartados anteriores, el paquete **java.util.concurrent** incluye estas otras **utilidades de concurrencia**:

- ✓ La interfaz `Executor`
- ✓ Colecciones.
- ✓ La clase `Locks`
- ✓ Variables atómicas



El **programador de tareas `Executor`** es una interfaz que permite:

- ✓ Realizar la ejecución de tareas en un único hilo en segundo plano (como eventos Swing), en un hilo nuevo, o en un pool de hilos
- ✓ Diseñar políticas propias de ejecución y añadirlas a `Executor`.
- ✓ Ejecutar tareas mediante el método `execute()`. Estas tareas tienen que implementar la interfaz `Runnable`.
- ✓ Hacer uso de diferentes implementaciones de `Executor`, como `ExecutorService`.

Entre las **colecciones** hay que destacar:

- ✓ La interfaz `Queue`. Es una colección diseñada para almacenar elementos antes de procesarlos, ofreciendo diferentes operaciones como inserción, extracción e inspección.
- ✓ La interfaz `BlockingQueue`, diseñada para colas de tipo productor/consumidor, y que son thread-safe (aseguran un funcionamiento correcto de los accesos simultáneos multihilo a recursos compartidos). Son capaces de esperar mientras no haya elementos almacenados en la cola.
- ✓ Implementaciones concurrentes de `Map` y `List`.

La clase de **bloqueos**, **java.util.concurrent.locks**, proporciona diferentes implementaciones y diversos tipos de bloqueos y desbloqueos entre métodos. Su funcionalidad es equivalente a **Synchronized**, pero proporciona métodos que hacen más fácil el uso de bloqueos y condiciones. Entre ellos.

- ✓ El método `newCondition()`, que permite tener un mayor control sobre el bloqueo y genera un objeto del tipo `Condition` asociado al bloqueo. Así el método `await()` indica cuándo deseamos esperar, y el método `signal()` permite indicar si una condición del bloqueo se activa, para finalizar la espera.
- ✓ La implementación `ReentrantLock`, permite realizar exclusión mutua utilizando monitores. El método `lock()` indica que deseamos utilizar el recurso compartido, y el método `unlock()` indica que hemos terminado de utilizarlo.

Las **variables atómicas** incluidas en las utilidades de concurrencia clase **java.util.concurrent.atomic**, permiten definir recursos compartidos, sin la necesidad de proteger dichos recursos de forma explícita, ya que ellas internamente realizan dichas labores de protección.

Si desarrollamos aplicaciones multihilo más complejas, por ejemplo para plataformas multiprocesador y sistemas con multi-núcleo (multi-core) que requieren un alto nivel de concurrencia, será muy conveniente hacer uso de todas estas utilidades.

Citas para pensar

"Lo que sabemos es una gota de agua; lo que ignoramos es el océano". *Isaac Newton*

Para saber más

En el siguiente enlace, en inglés, puedes consultar el tutorial oficial Java de Oracle sobre hilos,

incluyendo ejemplos de uso de las utilidades incluidas en el paquete **java.util.concurrent** para aplicaciones multihilo.

[Tutorial Multihilo en Java de Oracle.](#)

En este otro enlace encontrarás múltiples ejemplos Java utilizando también las clases de **java.util.concurrent**.

[Ejemplos Java utilizando las clases de java.util.concurrent.](#)

8.2.- La interfaz **Executor** y los pools de hilos.

Cuando trabajamos con **aplicaciones tipo servidor**, éstas tienen que atender un número masivo y concurrente de peticiones de usuario, en forma de tareas que deben ser procesadas lo antes posible. Al principio de la unidad ya te indicamos mediante un ejemplo la conveniencia de utilizar hilos en estas aplicaciones, pero si ejecutamos cada tarea en un hilo distinto, se pueden llegar a crear tantos hilos que el incremento de recursos utilizados puede comprometer la estabilidad del sistema. Los pools de hilos ofrecen una solución a este problema.



Y ¿qué es un pool de hilos? Un **pool de hilos (thread pools)** es básicamente un contenedor dentro del cual se crean y se inician un número limitado de hilos, para ejecutar todas las tareas de una lista.

Para **declarar un pool**, lo más habitual es hacerlo **como un objeto del tipo `ExecutorService`** utilizando alguno de los siguientes **métodos de la clase estática `Executors`**:

- ✔ `newFixedThreadPool(int numeroHilos)`: crea un pool con el número de hilos indicado. Dichos hilos son reutilizados cíclicamente hasta terminar con las tareas de la cola o lista.
- ✔ `newCachedThreadPool()`: crea un pool que va creando hilos conforme se van necesitando, pero que puede reutilizar los ya concluidos para no tener que crear demasiados. Los hilos que llevan mucho tiempo inactivos son terminados automáticamente por el pool.
- ✔ `newSingleThreadExecutor()`: crea un pool de un solo hilo. La ventaja que ofrece este esquema es que si ocurre una excepción durante la ejecución de una tarea, no se detiene la ejecución de las siguientes.
- ✔ `newScheduledExecutor()`: crea un pool que va a ejecutar tareas programadas cada cierto tiempo, ya sea una sola vez o de manera repetitiva. Es parecido a un objeto `Timer`, pero con la diferencia de que puede tener varios threads que irán realizando las tareas programadas conforme se desocupen.

Los objetos de tipo **`ExecutorService`** implementan la interfaz **`Executor`**. Esta interfaz define el método **`execute(Runnable)`**, al que hay que llamar una vez por cada tarea que deba ser ejecutada por el pool (la tarea se pasa como argumento del método).

La interface **`ExecutorService`** proporciona una serie de métodos para el control de la ejecución de las tareas, entre ellos el método **`shutdown()`**, para indicarle al pool que los hilos no se van a reutilizar para nuevas tareas y deben morir cuando finalicen su trabajo.

En el siguiente enlace dispones de un ejemplo en el que se define una clase que implementa **`Runnable`** cuya tarea es generar e imprimir 10 números aleatorios. Se creará un pool de 2 hilos capaz de realizar 30 de esas tareas.

[Código del proyecto Java ejemplo de uso de `ExecutorService`.](#) (0.01 MB)

Para saber más

En el siguiente enlace, en inglés, encontrarás otro ejemplo de uso de **`Executor`**, así como de otras clases del paquete **`java.util.concurrent`**

[Utilidades de concurrencia Java.](#)



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

`ExecutorService` es una clase derivada de `Executors`.

Verdadero. ☐ Falso. ☐

8.3.- Gestión de excepciones.

¿Cómo podemos gestionar la excepciones de una aplicación multihilo?

Para gestionar las excepciones de una aplicación multihilo puedes utilizar el método `uncaughtExceptionHandler()` de la clase `Thread`, que permite definir un manejador de excepciones.

Para **crear un manejador de excepciones** haremos lo siguiente:

- ✓ Crear una clase que implemente la interfaz `Thread.UncaughtExceptionHandler`.
- ✓ Implementar el método `uncaughtException()`.



Por ejemplo, podemos crear un manejador de excepciones que utilizarán todos los hilos de una misma aplicación de la siguiente forma:

- ✓ El manejador sólo mostrará qué hilo ha producido la excepción y la pila de llamadas de la excepción.

```
public class ManejadorExcepciones implements Thread.UncaughtExceptionHandler {
    //Manejador de excepciones para toda la aplicación
    //Implementa el método uncaughtException()
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println("Thread que lanzó la excepción: " + t.getName());
        //Muestra en consola el hilo que produce la excepción
        e.printStackTrace();
        //Muestra en consola la pila de llamadas
    }
}
```

En el siguiente enlace te facilitamos el proyecto completo. Se creará el anterior manejador de excepciones y se implementará un hilo que divide el número 100 por un número aleatorio comprendido entre 0 y 4, dando así la posibilidad de dividir por 0. Se crean e inician 5 hilos que harán uso del manejador.

[Código del proyecto con manejador de excepciones.](#) (0.01 MB)



Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa.

En la clase `Thread` se dispone de un método que permite crear un manejador de excepciones para aplicaciones multihilo.

Verdadero. ☐ Falso. ☐

8.4.- Depuración y documentación.

Dos tareas muy importantes cuando desarrollamos software de calidad son la depuración y documentación de nuestras aplicaciones.

- ✓ Mediante la **depuración** trataremos de corregir fallos y errores de funcionamiento del programa.
- ✓ Mediante la **documentación interna** aportaremos legibilidad a nuestros programas.



La **depuración de aplicaciones multihilo** es una tarea difícil debido a que:

- ✓ La ejecución de los hilos tiene un comportamiento no determinístico.
- ✓ Hay que controlar varios flujos de ejecución.
- ✓ Aparecen nuevos errores potenciales debidos a la compartición de recursos entre varios hilos:
- ✓ Errores porque no se cumple la exclusión mutua.
- ✓ Errores porque se produce interbloqueo.

Podemos realizar seguimientos de la pila de Java tanto estáticos como dinámicos, utilizando los siguientes métodos de la clase **thread**:

- ✓ `dumpStack()`. Muestra una traza de la pila del hilo (thread) en curso.
- ✓ `getAllStackTraces()`. Devuelve un Map de todos los hilos vivos en la aplicación. (Map es la interfaz hacia objetos `StackTraceElement`, que contiene el nombre del fichero, el número de línea, y el nombre de la clase y el método de la línea de código que se está ejecutando).
- ✓ `getStackTrace()`. Devuelve el seguimiento de la pila de un hilo en una aplicación.

Tanto **`getAllStackTraces()`** como **`getStackTrace()`** permiten grabar los datos del seguimiento de pila en un log.

En el siguiente enlace encontrarás un ejemplo de cómo depurar aplicaciones multihilo desde el IDE de NetBeans. Te puedes descargar además las aplicaciones de ejemplo para practicar.

[Depurando aplicaciones mutihilo con NetBeans.](#)



A la hora de **documentar una aplicación multihilo** no debemos escatimar en comentarios. Si son importantes en cualquier aplicación, con más motivo en una aplicación multihilo, debido a su mayor complejidad.


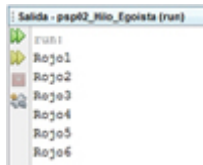
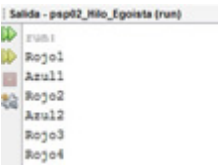








Para documentar nuestra aplicación Java, utilizaremos el generador **JavaDoc**, que de forma automática genera la documentación de la aplicación a partir del código fuente. Como ya debes de saber si has estudiado el módulo de "Programación", este sistema consiste en incluir comentarios en el código, utilizando las etiquetas `/**` y `*/`, que después pueden procesarse y generar un conjunto de páginas navegables [HTML](#).

Te recordamos mediante el vídeo (debes tener instalado JavaDoc en tu equipo, cómo obtener la documentación de tu programa con JavaDoc y Netbeans a partir de tu código fuente.

[Resumen textual alternativo](#)

Anexo.- Licencias de recursos.

Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos del recurso (2)
	Autoría: Jenni Ripley. Licencia: CC-by-nc. Procedencia: http://www.flickr.com/photos/queen_of_subtle/1813611687/		Autoría: Isabel M. Cruz Granados. Licencia: Uso educativo-no comercial. Procedencia: Captura de pantalla de la Salida de NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.
	Autoría: Isabel M. Cruz Granados. Licencia: Uso educativo-no comercial. Procedencia: Captura de pantalla de la Salida de NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.		Autoría: Héctor Pierna Sánchez. Licencia: CC-by-nc-sa. Procedencia: http://www.flickr.com/photos/hexmar/2035303614/
	Autoría: Saulo Alvarado Mateos. Licencia: CC-by-nc-nd. Procedencia: http://www.flickr.com/photos/luipermom/4138090307/		Autoría: David Buedo. Licencia: CC-by-nc-sa. Procedencia: http://www.flickr.com/photos/dbuedo/2425690907/in/photostream
	Autoría: Andrés Nieto Porras. Licencia: CC-by-sa. Procedencia: http://www.flickr.com/photos/anieto2k/4446687060/		Autoría: Rodrigo Vera. Licencia: CC-by-nc-nd. Procedencia: http://www.flickr.com/photos/rodrigovera/2181558093/
	Autoría: Gov/Ba. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/agecombahia/4479505820/in/photostream/		Autoría: Chris & Lara Pawluk. Licencia: CC-by-nc. Procedencia: http://www.flickr.com/photos/larachris/15684312/
	Autoría: Criterion. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/criterion/5621843243/		

