UNIDAD 2: PROGRAMACIÓN MULTIHILO

Módulo Profesional:
Programación de Servicios y Procesos

Índice

RESUMEN INTRODUCTORIO 3
INTRODUCCIÓN 3
CASO INTRODUCTORIO 4
1. HILOS 5
1.1 Estados de un hilo. Cambios de estado y recursos compartidos 6
1.1.1 Cambios de estado 7
1.1.2 Recursos de un hilo 8
1.2 Hilos de usuario vs. hilos de sistema. Modelos de hilos. Planificación de hilos 9
1.3 Elementos relacionados con la programación de hilos. Librerías y clases 11
1.4 Programación de aplicaciones multihilo 14
2. GESTIÓN DE HILOS 15
2.1 Creación, ejecución y finalización de hilos 16
2.2 Sincronización de hilos. Recursos compartidos 19
2.2.1 Exclusión mutua. Condiciones de sincronización 20
2.2.2 Compartición de información entre hilos y mecanismos de comunicación 21
2.2.3 Mecanismos de comunicación y sincronización de hilos (semáforos, monitores, paso de mensajes) 22
2.2.4 Variables volatile 23
2.3 Problemas. Inanición, interbloqueos 26
2.4 Prioridades 28
2.5 Hilos demonio 29
2.6 Grupos (pool) de hilos 30
2.7 Temporizadores y tareas periódicas 31
RESUMEN FINAL 32

RESUMEN INTRODUCTORIO

En esta unidad profundizaremos en los conceptos centrados en los hilos, comenzando con la definición de un hilo, así como los tipos que nos podemos encontrar.

A partir de esta base ahondaremos en la gestión de los hilos, sus estados y transiciones, ya que es importante conocer cómo se comportan para poder conocer el uso de las librerías dentro de la programación y sus posibilidades.

Uno de los aspectos más importantes dentro de la programación multihilo es la de sincronización de estos y, aunque ya introdujimos conceptos y posibles soluciones, con la programación multihilo cambian las situaciones de espacios críticos.

Por último, trabajaremos múltiples conceptos alrededor de esta programación como son el concepto de prioridad, cómo gestionar grupos de hilos o las tareas periódicas.

INTRODUCCIÓN

La programación multihilo permite que dos o más partes de un programa se ejecuten al mismo tiempo realizando diferentes tareas. Esto permite un uso óptimo de los recursos del sistema.

Las herramientas fundamentales de este tipo de programación son los hilos o threads que permiten dividir la ejecución de un programa en varias instancias independientes que pueden realizar tareas distintas y ejecutarse de forma paralela.

Para poder llevar a cabo este tipo de programación hay que contemplar la necesidad de sincronizar los procesos para ejecutar las secciones de código críticas de una forma segura.

El lenguaje de programación Java es un lenguaje multihilo por lo que en esta unidad se aprovecharán los recursos que este lenguaje nos ofrece para mejorar el rendimiento de nuestras aplicaciones.

CASO INTRODUCTORIO

Trabajamos como desarrolladores dentro del departamento de informática de una pyme, cuyo negocio está orientado a la compra y venta de bebidas para el entorno de la restauración.

Es importante poder controlar perfectamente el stock del almacén principal cuando se hacen pedidos desde la central de compras y ventas de los comerciales.

Para ello, necesitamos realizar un programa intermedio en Java que permita realizar ese control de stock de una forma controlada y sabiendo que múltiples procesos accederán a nuestra clase en Java.

¿Cómo programaremos esas compras y ventas? ¿Cómo gestionaremos el acceso a los arrays de stock? ¿Cómo gestionaremos las prioridades de los accesos?

Al finalizar la unidad tendremos los recursos y habilidades para responder a esas preguntas.

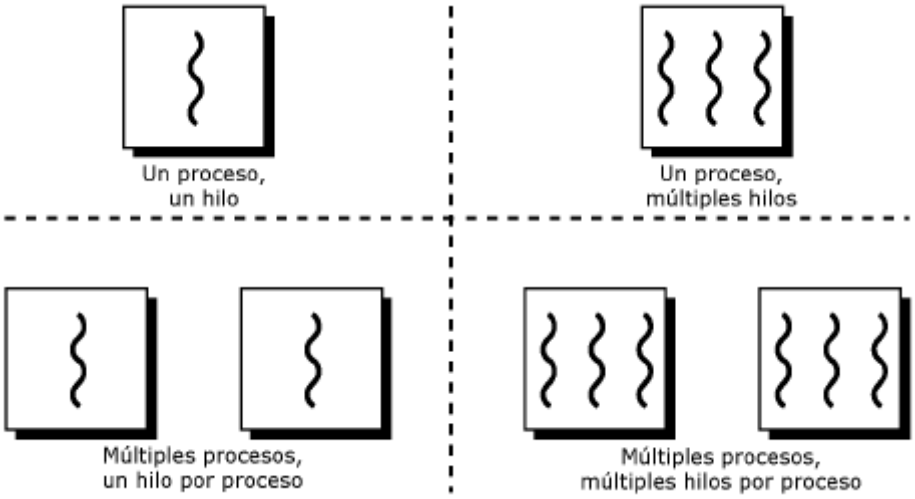
1. HILOS

Igual que conocimos cómo funcionan los procesos, y en concreto cómo funcionan los procesos con Java, nos preguntamos cómo funcionan los hilos.

¿Los estados de los hilos con Java son los mismos que con los procesos? ¿Qué clases están relacionadas con los hilos en Java?

A partir de estas cuestiones estableceremos el entorno de trabajo y las metodologías adecuadas para la programación multihilo de nuestra aplicación.

En la unidad anterior estudiábamos que los procesos son programas en ejecución y que cada proceso tendrá siempre un hilo, en el que corre el propio programa, pero puede tener más hilos.



Hilos y procesos.

Un hilo es un punto de ejecución de un proceso. Consta de un contador de programa, un conjunto de registros y un espacio de pila.

Los hilos representan un método software para mejorar el rendimiento y eficacia de los sistemas operativos, ya que permiten que varias instrucciones se estén ejecutando concurrentemente compartiendo el espacio de direcciones y las estructuras de datos.

ENLACE DE INTERÉS

Para ampliar información sobre los procesos y los hilos se recomienda realizar la siguiente lectura:

http://sistop.gwolf.org/laminas/05-procesos-e-hilos.pdf

1.1 Estados de un hilo. Cambios de estado y recursos compartidos

Antes de abordar las operaciones que se pueden realizar con hilos en Java es necesario conocer los estados de un hilo en este lenguaje, así como los métodos que provocan el paso de un estado a otro.

Básicamente los estados posibles de un hilo son:

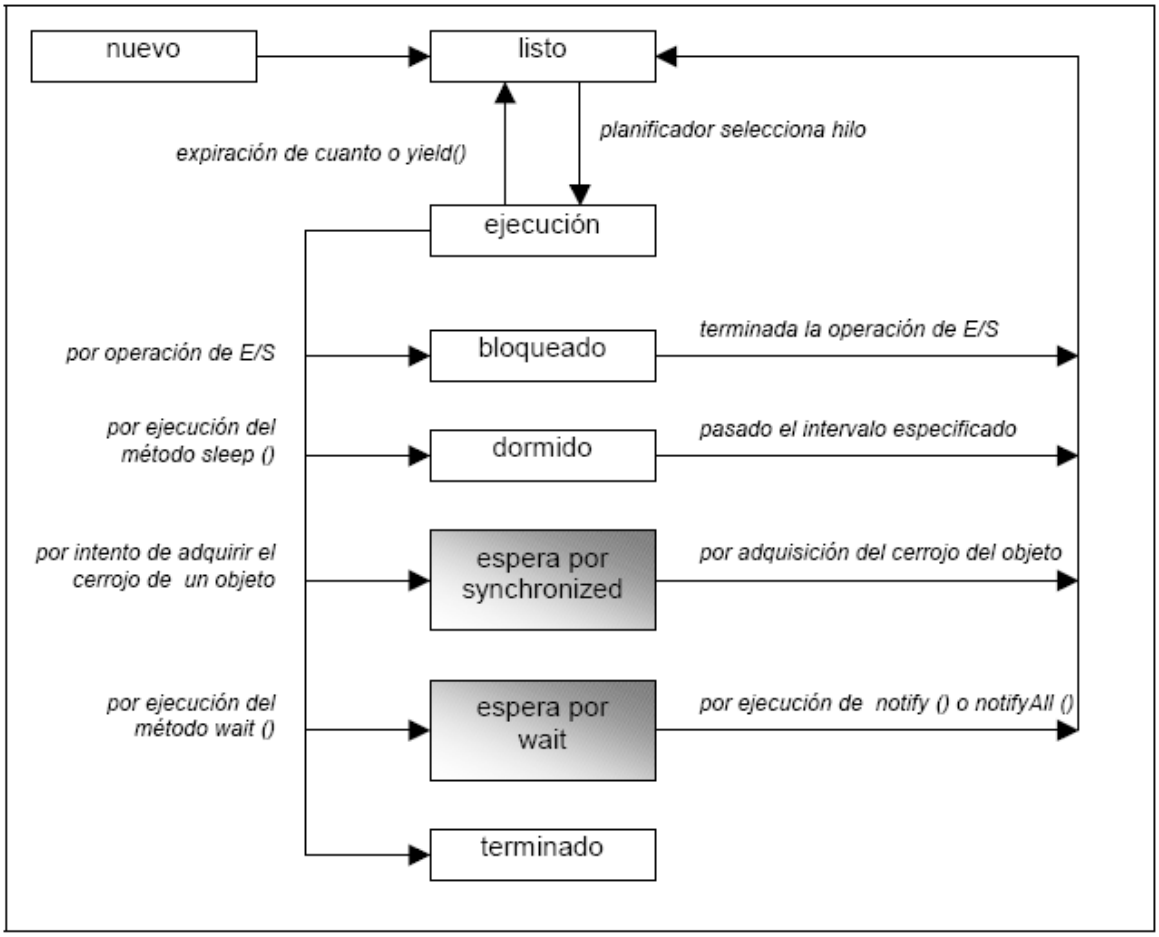
- **Nuevo**: En general, cuando se crea un nuevo proceso se crea también un hilo para ese proceso. Posteriormente, ese hilo puede crear nuevos hilos dándoles un puntero de instrucción y algunos argumentos. Ese hilo se colocará en la cola de preparados.

- **Bloqueado**: Cuando un hilo debe esperar por un suceso, se le bloquea guardando sus registros. De esta forma, el procesador pasará a ejecutar otro hilo que se encuentre en el estado de preparado. Hay que destacar que, si se bloquea un hilo de un proceso automáticamente, se bloquean todos los hilos de ese mismo proceso.

- **Preparado**: Cuando se produce el suceso por el que un hilo se bloqueó pasa a la cola de listos.

- **En ejecución**: El hilo se está ejecutando.

- **Terminado**: Cuando un hilo finaliza, se liberan su contexto y sus pilas.



Estados de un hilo.

Como se muestra en la imagen, los estados de los hilos son semejantes a los de los procesos. El planificador de los hilos de las aplicaciones Java se encuentra implementado en la máquina virtual.

1.1.1 Cambios de estado

Cuando un hilo finalice su ejecución llegando al final del método run, pasará al estado de acabado y diremos que está muerto (*dead*), porque desde este estado no podrá volverse a ejecutar más.

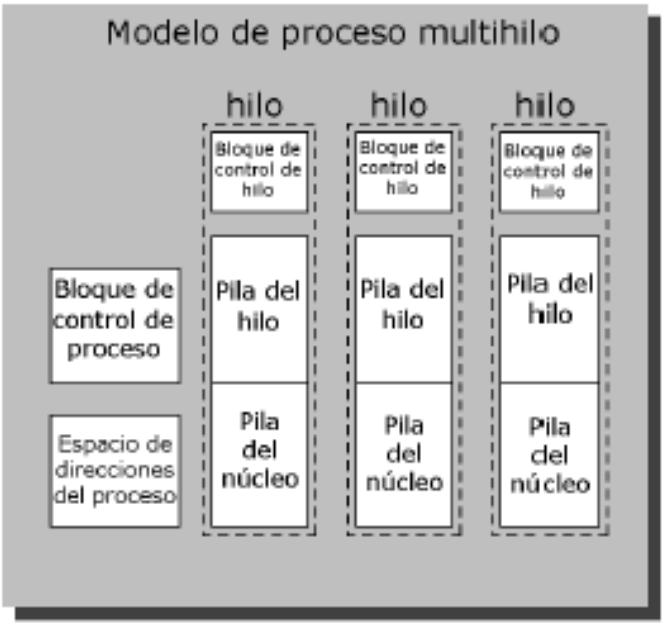
Así pues, la invocación del método start de un hilo acabado, provocará una excepción en tiempo de ejecución.

La finalización de un hilo no implica que el objeto Thread desaparezca y libere la memoria. Las instancias de Thread siguen las mismas reglas que cualquier otro objeto, mientras se encuentren asignadas a alguna variable, se mantendrán en memoria. Podemos forzar la liberación de memoria asignando un valor null a la variable que mantenía el objeto Thread, una vez este haya finalizado.
Mientras el hilo está en ejecución, podrá salir por varios motivos:

- Ha expirado el tiempo de uso del procesador por parte del hilo o se ejecuta el método *yield*().
- Inicia una operación de entrada/salida y pasa a un estado de bloqueado hasta que la operación acabe.
- Llamamiento al método *join*() de otro hilo para esperar la finalización del mismo. El hilo invocador pasa al estado de bloqueado hasta que el hilo invocado pase al estado acabado.
- Intenta ejecutar un código que está sincronizado (*synchronized*) y no puede porque hay otro hilo ejecutándolo, pasa a un estado de bloqueado hasta que el bloque sincronizado quede liberado.
- Se llama al método *wait*() pasando al estado de espera. Cuando un hilo entro en estado de espera, solo podrá salir si se invoca algunos de sus métodos *notify*() o *notifyAll*().
- Se invoca el método *sleep*. El hilo pasa a estado dormido hasta que haya transcurrido el tiempo indicado como parámetro. El estado dormido es muy útil para mantener un hilo parado durante un tiempo determinado minimizando al máximo los recursos utilizados. Suponemos que un programa de tratamiento de textos dispone de un hilo que guarda los documentos automáticamente cada 10 minutos.

1.1.2 Recursos de un hilo

Los hilos de un mismo proceso comparten el mismo espacio de direcciones y tienen acceso a los mismos datos. Sin embargo, cada uno de ellos tiene su propio estado, su propia pila, su propio bloque de control de hilos y su propia copia de los registros de la CPU, tal y como se muestra en la imagen.

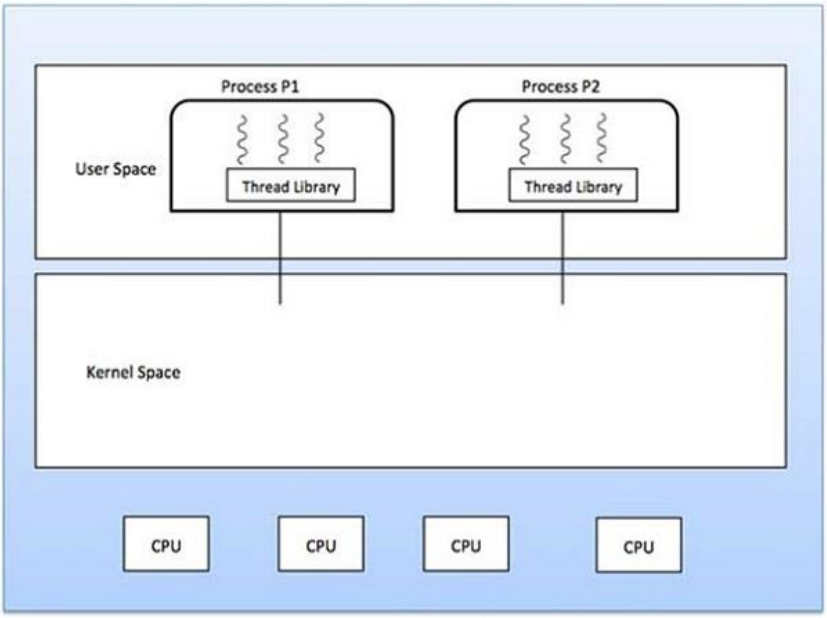


Modelo proceso multihilo.

1.2 Hilos de usuario vs. hilos de sistema. Modelos de hilos. Planificación de hilos

Tal y como indica el título, una de las primeras clasificaciones que podemos realizar de los hilos que nos podemos encontrar en un sistema operativo es:

- User threads (hilos de usuario). Son hilos creados por el usuario, es decir, por programas controlados por el usuario y, por lo tanto, el núcleo del sistema operativo no controla ni los subprocesos ni la creación y destrucción de los mismos. Como podemos observar en la imagen, el kernel del sistema operativo únicamente puede acceder a el proceso principal.

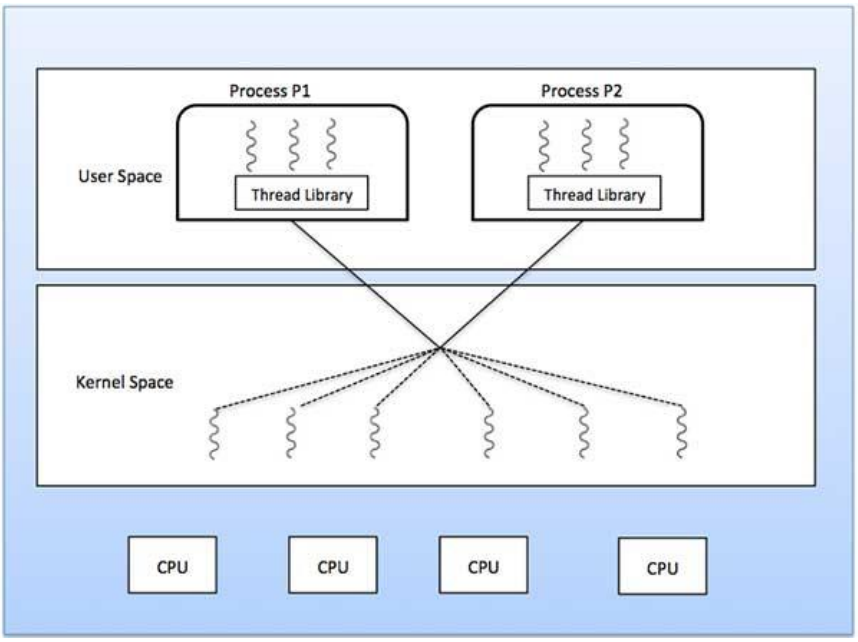


Hilos de usuario.
Fuente: https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

- Kernel threads (hilos de kernel o sistema). Al contrario del anterior modelo, los hilos los controla el kernel no habiendo código de proceso en el área de usuario. El Kernel, por lo tanto, realiza la creación y administración de los hilos, también en el espacio de kernel.

Algunos sistemas operativos, pueden ofrecer combinaciones de hilos de usuario y de hilos de sistema que podemos clasificar, a su vez, en tres modelos:

- Modelos muchos a muchos, en el que se multiplexan cualquier número de hilos de usuario con cualquier hilo de sistema tal y como vemos en la imagen.



Modelo muchos a muchos.

Fuente: https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

- Modelos muchos a uno, en este caso un proceso que contiene múltiples hilos de usuario se relaciona con un hilo de sistema.

- Modelo uno a uno, donde un único hilo de usuario se relaciona con un único hilo de sistema.

COMPRUEBA LO QUE SABES

Acabamos de estudiar diferentes modelos de hilos, ¿serías capaz de poner un ejemplo sobre el modelo muchos a muchos? Razona el ejemplo y coméntalo en el foro.

1.3 Elementos relacionados con la programación de hilos. Librerías y clases

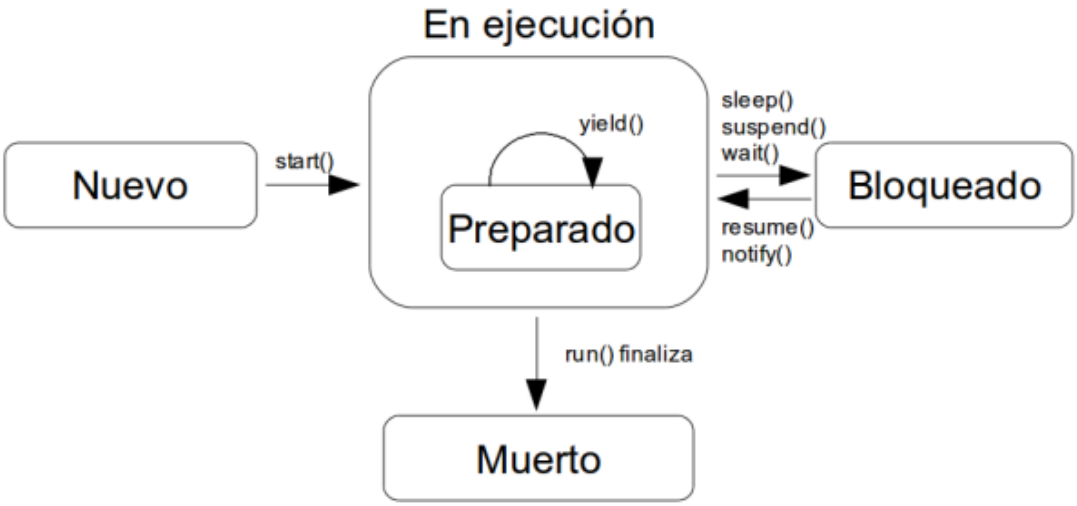
En Java disponemos de dos opciones para la creación de hilos:

- Heredar de la clase Thread.
- Implementar la interfaz Runnable.

En la siguiente imagen se muestran los métodos de la clase Thread y de la interfaz Runnable. No obstante, los explicaremos más adelante un poco más detallados.

Clase Thread.

La primera opción disponible sería **heredar** de la clase Thread que, a su vez, implementa la interfaz Runnable. Esta clase se encuentra en el paquete java.lang.Thread y nos ofrece un conjunto de métodos para trabajar con hilos. En la siguiente imagen se observan los estados anteriormente estudiados de los procesos y los métodos de la clase Thread para pasar de uno a otro:



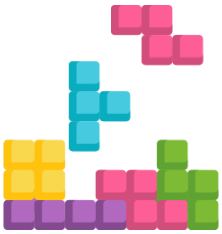
Estados de un hilo.

- **Yield():** Para permutar la ejecución de una tarea y la siguiente disponible.
- **Sleep(long):** Para pausar la tarea en curso durante un número de milisegundos indicados por la variable long.
- **Start():** Para iniciar un proceso o tarea. Llama automáticamente al método run().
- **Run():** Cuerpo de una tarea o hilo.
- **Stop():** Para la ejecución de una tarea y la destruye.
- **Suspend():** Para la ejecución de una tarea sin destruirla.
- **Resume():** Para revivir una tarea que ha sido parada o suspendida.
- **Wait():** Para pausar la ejecución de una tarea hasta recibir una señal.
- **isAlive():** Para conocer el estado de un thread.

ENLACE DE INTERÉS

Como en la anterior unidad, es muy recomendable visitar el API de documentación de Oracle:

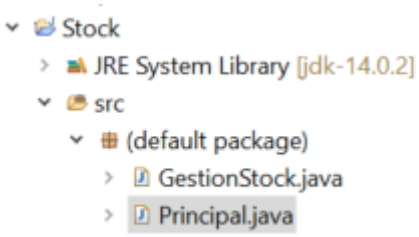
https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html



EJEMPLO PRÁCTICO

Queremos comenzar a plantear el esqueleto de nuestro gestor de stock y, para ello, nuestro jefe nos plantea un reto sencillo: crear una clase que se pueda lanzar multihilo y probarla. ¿Cómo realizar esta tarea inicial?

1. Para ello, creamos un primer proyecto sencillo que se denomine Stock con dos clases.



Proyecto Stock.
Fuente: Elaboración propia.

2. Creamos una clase que extienda de Thread y que implemente el método run:

```
public class GestionStock extends Thread{
	public void run() {
		System.out.println("Iniciada la gestión de stock");
	}
}
```

3. Creamos una primera prueba desde el método main:

```
public class Principal {

	public static void main(String[] args) {
		//Primera prueba de nuestro gestor de Stock
		GestionStock compras= new GestionStock();
		GestionStock ventas= new GestionStock();

		//Creamos los threads
		Thread cThread=new Thread(compras);
		Thread vThread=new Thread(ventas);

		//Arrancamos procesos
		cThread.start();
		vThread.start();

	}

}
```

1.4 Programación de aplicaciones multihilo

El lenguaje de programación Java permite manejar programas que tienen distintos procesos llamados threads. Esto se utiliza para crear aplicaciones que realicen varias tareas simultáneamente. Realmente no se ejecutan los hilos simultáneamente, sino que el sistema operativo se encarga de alternarlos de manera que da esta sensación al usuario.

Para crear una clase multihilo, lo primero que tenemos que hacer es indicar que esta hereda de la clase Thread. Para esto utilizamos la siguiente sintaxis:

public class nombreClase extends Thread

Además, se debe sobrescribir el método run(). Este método pertenece a la interfaz Runnable, pero la clase Thread implementa esta interfaz por lo que heredando de Thread tenemos disponible dicho método.

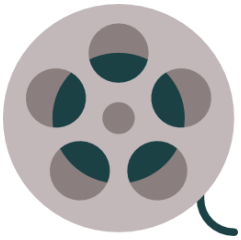
Es importante sobrescribir el método run(), ya que este contendrá la funcionalidad que se ha de ejecutar en ese hilo. Además, puede ser necesario que se sobrescriban más métodos para, por ejemplo, poner en reposo el hilo o pararlo.

Sin embargo, puede haber ocasiones en las que no se puede heredar de la clase Thread porque la clase ya hereda de otras clases. En este caso podríamos recurrir a la opción de implementar directamente la interfaz Runnable. Si implementamos la interfaz Runnable lo único que tenemos que hacer es sobrescribir el método run() de dicha interfaz para que los procesos implementados se ejecuten en un hilo diferente.

Para indicar que la clase implementa la interfaz Runnable utilizamos:

public class nombre implements Runnable

A continuación, como decíamos anteriormente, sobrescribimos el método run() y creamos el método principal en el cual, a través de la clase Thread, creamos tantos hilos como necesitemos y los iniciamos con el método start.



VIDEO DE INTERÉS

En el siguiente vídeo encontrarás un ejemplo de programación multihilo con Java:

https://www.youtube.com/watch?v=qXhc4wbDaqU

2. GESTIÓN DE HILOS

Una vez conocidos los mecanismos generales sobre los estados y el funcionamiento de los hilos, nos toca ponernos manos a la obra en el desarrollo de nuestra aplicación con Java.

¿Cómo creamos un hilo? ¿Cómo gestionamos las sincronizaciones cuando varios hilos necesitan acceder a un mismo recurso? ¿Cómo agrupamos hilos?

Es necesario responder a todas están preguntas para una buena gestión de nuestro programa.

La programación multihilo permite llevar a cabo diferentes hilos de ejecución a la vez, es decir, permite realizar diferentes tareas en una aplicación de forma concurrente. La mayoría de lenguajes de programación permiten trabajar con hilos y realizar programación multihilo. También son multihilo la mayoría de las aplicaciones que usamos en nuestros ordenadores (editores de texto, navegadores, editores gráficos…), hecho que nos da la sensación de más agilidad de ejecución.

En un editor de texto, por ejemplo, un hilo puede estar controlando la ortografía del texto, otro puede estar atento a las pulsaciones sobre los iconos de la interfaz gráfica y el otro guardando el documento.

La programación tradicional está diseñada para trabajar secuencialmente, de forma que cuando acaba un proceso se pone en marcha otro. Los hilos son una forma de ejecutar paralelamente diferentes partes de un mismo programa. En Java los hilos son conocidos como threads, tal y como hemos visto.

2.1 Creación, ejecución y finalización de hilos

Existen dos formas de creación de hilos en Java:

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

A su vez, si utilizamos la creación mediante la clase Thread, tenemos varias opciones:

Thread identificador=new Thread();

Thread identificador=new Thread(referenciaAObjeto);

Thread identificador=new Thread(nombre);

Thread identificador=new Thread(referenciaAObjeto, nombre);

Donde:

- nombre representa un nombre para distinguir un hilo de otro.
- referenciaAObjeto indica la referencia al objeto de la clase que implementa el método run().

Cada hilo pasa por distintas etapas que quedan determinadas a través de los métodos que nos proporciona la clase Thread y que deben ser sobrescritos para que realicen las operaciones oportunas.

Cuando se crea un hilo no se le asigna ningún recurso del sistema, sería un hilo vacío. Para crear los recursos se utiliza el método start() que se encarga de planificar el inicio del hilo y llamar al método run(). Este método contendrá las operaciones que deben realizarse durante la vida del hilo.

La ejecución del método run no garantiza una ejecución en un hilo independiente. Si queremos invocar el procesamiento de un hilo habrá que llamar al método start(). Su invocación iniciará la ejecución de un nuevo hilo justo a la primera instrucción del método run(). Es decir, el llamamiento de start implica la invocación posterior del método run.

Veámoslo a través de un código ejemplo:

```
package heretafil;

public class HeretaFil extends Thread {

   String strImprimir;
   public HeredaHilo(String strP) {
```

```
        strImprimir=strP;
    }

    public void run(){
        for(int x=0;x<5;x++){
            System.out.println(strImprimir+ " " + x);
        }

    }

    public static void main(String[] args) {

        Thread primero = new HeredaHilo ("Hilo 1");
        Thread segundo = new HeredaHilo ("Hilo 2");
        // Hemos creado dos hilos pero no se han ejecutado
        // Para ejecutarlos …
        primero.start();
        segundo.start();

        System.out.println("Final Hilo Principal");

    }
}
```

En el anterior ejemplo se han creado dos hilos desde el hilo principal que han ejecutado el mismo código. Una posible salida del programa podría ser:

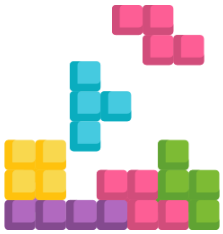
```
run:
Fil 2 0
Fil 1 0
Fil 2 1
Fil 1 1
Final Hilo Principal
Hilo 2 2
Hilo 1 2
Hilo 2 3
Hilo 1 3
Hilo 2 4
Hilo 1 4
BUILD SUCCESSFUL (total time: 0 seconds)
```

Como se puede observar, la orden se va intercalando. Tenemos tres hilos de ejecución: el principal y los dos creados. Esta puede ser una posible salida, puesto que la ejecución de los tres hilos es independiente y puede ser normal que acabe el hilo principal antes de los dos creados por él.

COMPRUEBA LO QUE SABES

Acabamos de estudiar cómo se pueden crear y finalizar hilos. ¿Podrías nombrar las diferencias con los procesos? ¿Qué ventajas y desventajas tenemos entre ambos mecanismos?

Coméntalo en el foro.



EJEMPLO PRÁCTICO

Nuestro gestor de stock va a mantener en memoria un array con el stock de nuestros productos. Sin pensar ahora mismo en la gestión de la sincronización, nuestro jefe nos plantea que implementemos la gestión de stock multithread con un array de productos iniciales. ¿Cómo lo implementaríamos?

1. Sobre la clase de gestión de stock deberemos realizar dos modificaciones:
    a. La primera es la de los parámetros de entrada a la clase para poder realizar las operaciones adecuadas.
    b. La segunda, la de realizar las modificaciones sobre el stock.

2. El código final de la clase de gestión de stock quedaría de la siguiente manera:

```
public class GestionStock extends Thread{
	private int[] stock;
	private int idProducto,cantidad;
	public GestionStock(int[] stock, int idProducto, int cantidadMod) {
		this.stock  = stock;
		this.idProducto=idProducto;
		this.cantidad=cantidadMod;
	}
	public void run() {
		System.out.println("Iniciada la gestión de stock");
		System.out.println("Peración sobre id:"+this.idProducto+",
con una modificacion de cantidad de "+this.cantidad);

	this.stock[this.idProducto]=this.stock[this.idProducto]+this.cantid
ad;
	}
}
```

3. Mientras que la clase principal quedaría de esta forma:

```
public class Principal {

	public static void main(String[] args) {
		int[] stockRefrescos= {10,23,12,-5,4};
		//Primera prueba de nuestro gestor de Stock
		GestionStock compras= new GestionStock(stockRefrescos,3,10);
		GestionStock ventas= new GestionStock(stockRefrescos,4,-5);

		//Creamos los threads
		Thread cThread=new Thread(compras);
		Thread vThread=new Thread(ventas);

		//Arrancamos procesos
		cThread.start();
		vThread.start();

	}

}
```

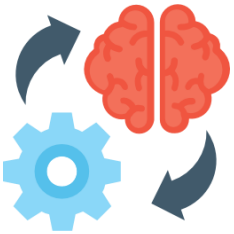
2.2 Sincronización de hilos. Recursos compartidos

En entornos multihilos que se ejecutan concurrentemente o de forma paralela es habitual que entre los diferentes hilos haya recursos compartidos:

- Un ejemplo se nos puede presentar cuando usamos variables compartidas para indicar que un hilo ha acabado cierta función o no y, por lo tanto, otro hilo podrá ejecutar la parte de código que estaba bloqueada por esta variable.
- Otra cuando se comparten datos de un objeto común. Varios hilos tienen que modificar una propiedad de un objeto.
- También cuando se utilizan recursos como ficheros o streams de comunicaciones.

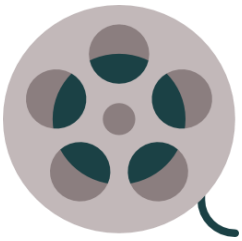
En todos los casos, conviene sincronizar el acceso a los recursos compartidos para evitar inconsistencia de datos o interferencias. Cualquier variable accesible puede ser manipulada por un hilo o el hilo principal. Si este acceso no se controla, puede provocar situaciones no esperadas o erróneas.

Al contrario que ocurría con los procesos, trabajar con hilos esta compartición de variables es muy fácil y nada costosa, puesto que los hilos de un mismo proceso comparten el mismo espacio de memoria. Java proporciona mecanismos para sincronizar los hilos que tratan estos problemas.



RECUERDA

En la unidad 1, en el apartado 2.4, estuvimos introduciendo los conceptos sobre concurrencia y sincronización sobre procesos. Como estamos estudiando en esta unidad, muchos de estos conceptos son similares.



VIDEO DE INTERÉS

En el siguiente vídeo encontrarás un ejemplo de programación multihilo con sincronización:

https://www.youtube.com/watch?v=2spoFK_9JEo&t=193s

2.2.1 Exclusión mutua. Condiciones de sincronización

Se pueden dar casos de datos corruptos o resultados no esperados cuando más de un hilo tiene que acceder al mismo recurso, ya que hay operaciones que se ejecutan concurrentemente utilizando el mismo objeto o recurso. Se denominan secciones críticas a estas zonas de código que acceden a un recurso compartido y que no pueden ser accesibles al mismo tiempo por más de un hilo.

Se denomina **exclusión mutua** a bloquear un objeto, esto significa que no deja que otros hilos cojan este bloqueo hasta que el hilo que lo tiene lo libere. Si esta acción se realiza correctamente, los hilos de ejecución no realizarán operaciones que interfieran entre sí. El objetivo final es intentar que las operaciones sobre las secciones críticas sean atómicas.

Por defecto, en Java un objeto no está protegido. Esto quiere decir que cualquier número de hilos puede ejecutar código dentro del objeto. La exclusión mutua se consigue en Java con la palabra reservada **synchronized**. Esta palabra puede ser aplicada a bloques de código dentro de un método y también a métodos enteros.

El modificador **synchronized** nos garantiza que ningún otro método sincronizado del objeto podrá ejecutarse. Cuando un hilo ejecuta un método sincronizado coge el bloqueo del objeto hasta que acaba su ejecución y se libera del bloqueo. Si otro hilo grita el método sincronizado del mismo objeto, queda bloqueado hasta que el hilo que lo está ejecutando lo libere del bloqueo.

En el siguiente código se puede ver como sincronizamos algunos métodos de una clase y otros no. Aquellos que queremos que se ejecuten en exclusión mutua los sincronicemos con la palabra reservada synchronized:

```
public class sincronizaMetodos {

public void metodoNoSincronitzado_1(){
//.........
}

public synchronized void metodoSincronizado_1(){
//.........
}
public synchronized void metodoSincronizado _2(){
//.........
}
public void metodoNoSincronitzado_2(){
//.........
}
}
```

2.2.2 Compartición de información entre hilos y mecanismos de comunicación

La palabra clave synchronized nos da seguridad e integridad en métodos o partes de código, pero, además de sincronización y protección, los hilos también tienen que poder comunicarse entre ellos. Java nos proporciona métodos que hacen que los hilos esperen y continúen su ejecución. Son los métodos: **wait(), notify() y notifyAll().**

Imaginamos dos hilos que controlan el envío y el procesamiento de correos electrónicos. El primer hilo está buscando correos electrónicos para poner en la cola de enviados y el que procesa los correos los construye para dejarlos a la cola.

¿Por qué el primer hilo tiene que estar malgastando tiempo y recursos cada dos segundos buscando a la cola de correos para enviar si todavía no hay? Es mejor si lo posamos en suspensión y cuando llegue un correo a la cola ya lo avisaremos y verificaremos el correo para enviarlo.

- **wait()**: Saca el hilo en curso y mantiene el bloqueo de la zona de exclusión mutua. Lo envía a una cola de espera o conjunto de espera y lo pasa al estado de espera.

- **notify()**: Un hilo de la cola de espera es seleccionado para pasar al estado preparado.
- **notifyAll()**: Todos los hilos de la cola de espera pasan al estado de preparado.

Tanto **notify()** como **notifyAll()** ponen a preparado los hilos de la cola de espera, si hay. Si no hay ningún hilo esperando, la instrucción es ignorada. Estos métodos son de la clase Object, por lo tanto, son heredados por todas las clases de forma implícita.

2.2.3 Mecanismos de comunicación y sincronización de hilos (semáforos, monitores, paso de mensajes)

Hay muchos mecanismos que se utilizan en la programación de procesos e hilos que nos permiten sincronizar estos.

Uno de los más sencillos pueda ser la sincronización por **semáforos** y dentro de este mecanismo también encontraremos múltiples maneras de implementarlo, pero el fundamento está en que un proceso/hilo utilice una variable o un método para bloquear un determinado proceso (semáforo activado) para después de realizar el trabajo liberar el semáforo. Una posible implementación sería:

1. Llamar a la función que activa el semáforo, por ejemplo, semaphoreStart antes de ejecutar el código de la sección crítica.
2. Llamar a la función semaphoreEnd para liberar el semáforo.

Se puede introducir también una creación y destrucción del semáforo o implementar un mecanismo contrario de desactivación y activación del semáforo. Sea cual sea el mecanismo, el objetivo es tener un sistema de bloqueo sencillo para la sección crítica.

Mientras que los semáforos están pensados como herramientas para que los hilos los utilicen, el siguiente mecanismo, los **monitores**, están pensados como zonas de exclusión para ejecutar su código los threads sin que puedan ser bloqueados. Dentro de un monitor, únicamente un thread puede estar ejecutándose.

Como regla general, un monitor consta de cuatro componentes:
- Inicialización. Tiene el código a ejecutar.
- Datos privados. Código que se puede ejecutar únicamente dentro del monitor y no son visibles desde fuera.
- Métodos del monitor. Métodos y procedimientos que sí son visibles.
- Cola de entrada. Corresponde a la cola donde están los hilos para ser ejecutados.

Dentro del entorno que estamos trabajando, y sin entrar en programación distribuida, podemos crear un espacio de comunicación o **paso de mensajes** entre threads. Un mecanismo puede ser el uso de colas mediante la clase Queue y, en concreto, usando la clase BlockingQueue.

2.2.4 Variables volatile

Las variables primitivas en Java son atómicas en lectura y actualización, pero no en operaciones. Por lo tanto, podemos omitir la sincronización si lo único que queremos hacer es modificar un valor de una variable primitiva. De este modo aumentamos el rendimiento de la aplicación. Tenemos que tener en cuenta, sin embargo, un aspecto importante. La máquina virtual de Java optimiza código cuando los compila si se encuentra un código como el siguiente:

```
class metodoEjemplo {
private bolean valor = false;
public void cambiarValor () {
valor = true;
}
public class claseEjemplo {

   private boolean flag = false;

   public void cambiarFlag(){
      flag=true;
   }

   public synchronized void metodoEjemplo () {
      flag = false;
      // otro hilo podria llamar al metodo cambiarFlag()
      if (flag) {
         // …
      }
    }
 }
```

La máquina virtual de Java puede optimizar el código pensando que el que se encuentra dentro de *if (flag)* no se ejecutará nunca, puesto que *flag* no puede ser nunca true porque la línea anterior la pone a false. Si cree que es código muerto no se ejecutará nunca. Pero otro hilo puede modificar el valor de *flag* accediendo al método cambiarFlag() cuando un primer hilo está a punto de entrar a *if* y, por lo tanto, sí que se podría entrar al bloque de *if.*

La forma de evitarlo es declarar la variable *flag* como *volatile*. De este modo le podemos decir a la máquina virtual de Java que otro hilo puede modificar este valor. La palabra reservada *volatile* aplicada sobre variables indica que se hará la actualización de forma atómica.

De esta forma podemos implementar una clase que se llame Mensaje:

```
static class Mensaje {
  final String msg;

  public Message(String msg) {
    this.msg = msg;
  }

  public String toString() {
    return msg;
  }

}
```

Y dentro de nuestro thread, implementar una cola de mensajes:

```
private BlockingQueue< Mensaje > colaMensajes;
```

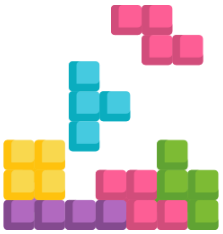
ENLACE DE INTERÉS

En el siguiente enlace encontrarás el API de BlockingQueue:

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html

COMPRUEBA LO QUE SABES

Acabamos de estudiar diferentes técnicas de sincronización mediante semáforos o monitores. ¿Cuándo utilizaríamos uno y otro? Razona tu respuesta con un ejemplo y coméntalo en el foro.



EJEMPLO PRÁCTICO

Nuestro sencillo gestor de stock únicamente necesita de un semáforo para poder implementar que el stock no descuadre cuando múltiples vendedores realicen ventas. Nuestro jefe nos motiva a implementar un semáforo.

¿Cómo lo implementaríamos?

- En el caso de usar la clase por múltiples compradores, podemos implementar un semáforo sencillo que se utilizaría desde fuera de la clase:

```
public class GestionStock extends Thread{

	private int[] stock;

	private int idProducto,cantidad;

	private boolean puedoActuar=true;


	public GestionStock(int[] stock, int idProducto, int cantidadMod) {

		this.stock  = stock;

		this.idProducto=idProducto;

		this.cantidad=cantidadMod;

	}

	public void run() {

		if(puedoActuar) {

			puedoActuar=false;

			System.out.println("Iniciada la gestión de stock");

			System.out.println("Peración sobre
id:"+this.idProducto+", con una modificacion de cantidad de
"+this.cantidad);


	this.stock[this.idProducto]=this.stock[this.idProducto]+this.cantid
ad;

			puedoActuar=true;

		}

	}
```

```
        public boolean getPuedoActuar() {return puedoActuar;}

}
```

Como vemos mediante ese sencillo semáforo, puedoActuar y su método asociado getPuedoActuar, se controlaría la ejecución del método run para que no existan incoherencias.

2.3 Problemas. Inanición, interbloqueos

Las ventajas de sincronizar métodos en Java son evidentes. No podríamos realizar concurrencia sin este tipo de seguridad y bloqueos, pero tenemos que ir con cuidado cuando aplicamos estos métodos:

1. En primer lugar, solo tenemos que sincronizar aquellos métodos o bloques de código necesarios, puesto que abusar de la sincronización penaliza el rendimiento de la aplicación.
2. En segundo lugar, porque si no sincronizamos correctamente se pueden producir errores. Evidentemente podemos realizar bloqueos sin usar las primitivas que nos proporciona Java, pero la fuente de errores es mayor, puesto que tenemos que recordar de coger el bloqueo y liberarlo, cosa que con synchronized, tal como hemos visto, no es necesaria.

El error más grave que se puede producir cuando se trabaja en programación multihilo relativamente compleja es el denominado interbloqueo (en inglés *deadlock*). El interbloqueo se produce cuando dos hilos se están ejecutando concurrentemente y cada uno tiene un bloqueo exclusivo que el otro necesita para continuar. El fenómeno puede pasar con más de dos hilos y por norma general pasa cuando los hilos cogen los mismos bloqueos en diferente orden.

Imaginemos una aplicación que gestiona los recursos de una entidad bancaria (cuentas bancarias, documentos, etc.) que permitan que los datos puedan ser transferidas de un recurso a otro. Cada recurso tiene un bloqueo asociado para él. Si se quiere hacer una transferencia, el hilo tiene que coger el bloqueo de todos los recursos que intervienen en la transferencia. Si dos hilos inician la transferencia que afecta a dos recursos y los hilos cogen el bloqueo de los recursos en diferente orden ya tenemos un interbloqueo. Veámoslo a través del siguiente código ejemplo:

```
public class HacerTransferencia {
 private static class CuentaCorriente {
  //Código de la Clase
}
private CuentaCorriente transferenciaA = new CuentaCorriente ();
private CuentaCorriente transferenciaB = new CuentaCorriente ();

public int metodoTransferenciaArchivos() {
  synchronized(transferenciaA) { // Posible interbloqueo
  synchronized(transferenciaB) {
  //Código del método
  }
  }
}
public void metodoTransferenciaDinero() {
synchronized(transferenciaB) { // Posible interbloqueo
synchronized(transferenciaA) {
// Código del método
}
}
}
}
```

La clase es estática, cosa muy usual cuando queremos proteger con sincronización los objetos de una aplicación. Miramos el código anterior e imaginamos un hilo que empieza a ejecutar el método metodoTransferenciaArchivos y coge el bloqueo sobre el objeto transferenciaA. Un segundo hilo entra y ejecuta el método metodoTransferenciaDinero y coge el bloqueo sobre el método transferenciaB.

En este momento se ha producido un interbloqueo. El primer hilo espera que se libere la transferenciaB para poder coger el bloqueo y continuar su ejecución. En cambio, el segundo hilo está esperando que se libere transferenciaA para poder coger el bloqueo y continuar su ejecución. Evidentemente ninguno de las dos cosas pasará. Habrá una espera infinita, un interbloqueo. Es posible que este mismo código se ejecute siempre bien, sin que se produzca un interbloqueo nunca, pero la probabilidad está.

En este caso, la solución es realmente sencilla. Solo hay que asegurar que la sincronización de ambos objetos (transferenciaA y transferenciaB) se produzca siempre en el mismo orden. En el código ejemplo, simplemente habrá que cambiar las órdenes de sincronización en uno de los métodos, por ejemplo, en metodoTransferenciaDinero, para que la orden de sincronización coincida con la del método metodoTransferenciaArchivos.

ARTÍCULO DE INTERÉS

En el siguiente enlace podrás profundizar sobre el interbloqueo en Java:

https://wrapper.tanukisoftware.com/doc/spanish/qna-deadlock.html

2.4 Prioridades

La manera que tiene el sistema operativo de comportarse con la ejecución de hilos es un tema importante en la programación multihilo. La planificación hace referencia a qué política se tiene que seguir para decidir qué hilo coge el control del procesador y en qué momento. También se tiene que planificar cuando tiene que dejar de ejecutarse.

Java está diseñado para que sea un sistema portable. El sistema de hilos tiene que ser soportado por cualquier plataforma y, puesto que cada plataforma funciona de forma diferente, el tratamiento de los hilos tiene que ser muy general. Como características más importantes de la planificación de hilos tenemos:

- Todos los hilos tienen asignada una prioridad y el encargado de decidir qué hilo se ejecuta tiene que garantizar que los hilos con prioridad más alta tengan preferencia, pero esto no implica que en un momento dado un hilo con prioridad más baja esté ejecutándose.
- Se tiene que garantizar que todos los hilos se ejecuten en algún momento.
- El tiempo asignado a cada hilo para hacer uso del procesador puede aplicarse o no dependiendo del sistema operativo en el cual se ejecuta la máquina virtual.

Si recordamos de la unidad anterior, al crear un nuevo hilo, este se coloca en una cola esperando su turno para ejecutarse. Este hilo irá pasando por diferentes estados y se les irán asignando distintas prioridades de las que dependerá que estos se ejecuten en más o menos tiempo. La prioridad de los hilos se puede cambiar en cualquier momento.

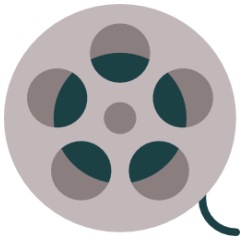
Java dispone de las siguientes variables para controlar la prioridad de los hilos:

- MAX_PRIORITY: Simboliza la máxima prioridad.

- NORM_PRIORITY: Simboliza la prioridad normal que es aquella que se asigna durante el arranque de la máquina virtual.
- MIN_PRIORITY: Simboliza la mínima prioridad.

Así como los siguientes métodos:

- setPriority(): Permite establecer la prioridad del hilo o hebra. Se utilizan las variables anteriores.
- getPriority(): Devuelve la prioridad de la hebra o hilo.



VIDEO DE INTERÉS

En el siguiente enlace encontrarás un interesante vídeo sobre mecanismos de sincronización y prioridades:

https://www.youtube.com/watch?v=o0Lzvm1OFiA

2.5 Hilos demonio

Podemos indicar que el hilo será ejecutado como un demonio (*daemon* en inglés) cuando creamos un hilo y antes de llamar al método start(). Esto permitirá al hilo ejecutarse en un segundo plan.

Los hilos demonios tienen la prioridad más baja. Si un hilo demonio crea otro hilo, este también será un demonio y no se puede cambiar una vez el hilo ha sido inicializado.

Se utiliza el método setDaemon(true) para indicar que un hilo es un demonio. En el caso de setDaemon(false), el hilo es de usuario, la opción por defecto.

Los hilos demonios finalizan cuando finaliza el último hilo de usuario y la aplicación acaba.

Se utiliza el método isDaemon() para ver si un hilo es un demonio, que devolverá un booleano indicando si el hilo es un demonio o no.

ENLACE DE INTERÉS

En el siguiente enlace encontrarás un ejemplo de programación de demonios con Java:

https://javaparajavatos.wordpress.com/2017/05/19/demonios/

2.6 Grupos (pool) de hilos

Un ejemplo que podemos poner para ver la utilidad de los grupos de hilos son los servidores que reciben muchas peticiones a la vez: de programas en ejecución, de clientes, cálculos complejas, transacciones de base de datos, etc.
Si el número de peticiones es grande, el número de hilos también lo es y la administración de los hilos es complicada.
La clase ThreadGroup es la que define e implementa todo aquello relacionado con grupos de hilos y permite simplificar esta de hilos. En esta clase tenemos dos constructores:

```
ThreadGroup(String name)
ThreadGroup(ThreadGroup parent, String name)
```

Todos los hilos tienen un grupo y todos los grupos creados a la aplicación tienen un grupo padre. Los dos constructores crean grupos y les dan un nombre. La diferencia es la elección de a qué subgrupo pertenecen. Como se puede observar, con el segundo constructor podemos definir a que padre pertenece el hilo. Veamos un pequeño ejemplo:

```
public static void main (String [] args) {
ThreadGroup sGrupFils1 = new ThreadGroup ("subGrup_1");
ThreadGroup sGrupFils1_2 = new ThreadGroup (sGrupFils1,
"subGrup_1_2");
}
```

COMPRUEBA LO QUE SABES

Acabamos de estudiar cómo crear grupos de hilos, ¿para qué usaríamos la creación de un grupo de hilos? Razona tu respuesta con un ejemplo y coméntalo en el foro.

2.7 Temporizadores y tareas periódicas

Cuando realizamos programas y aplicaciones seguro que nos aparece la necesidad de programar actividades y acciones programadas y que se ejecuten o bien periódicamente o bien en un cierto intervalo de tiempo.

Para estas tareas, dentro de Java tenemos la librería Timer, de java.swing.Timer, que a su vez depende de otras dos, ActionEvent y ActionListener:

```
int delay = 1000; //milliseconds
  ActionListener taskPerformer = new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
          //...Perform a task...
      }
  };
  new Timer(delay, taskPerformer).start();
```

Como vemos en el anterior ejemplo, el uso del Timer es bastante sencillo, ya que:

- El primer parámetro define la periodicidad de tiempo.
- El segundo parámetro indica la tarea que hay que realizar

Para realizar la parada de una actividad programada usaríamos el método stop().

ARTÍCULO DE INTERÉS

A continuación, encontrarás un artículo que sirve de resumen de todo lo visto en esta unidad con más código:

https://oscarmaestre.github.io/servicios/textos/tema2.html

RESUMEN FINAL

Los hilos o thread son pequeños procesos o piezas independientes de un proceso que, a diferencia de los procesos, comparten memoria. Se denominan entidades ligeras.

Un hilo o thread es la unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo. Es cada una de las secuencias de control que hay dentro de un proceso que ejecuta sus instrucciones de forma independiente. Es una tarea que puede ser ejecutada al mismo tiempo que otra tarea.

Java define la funcionalidad principal de la gestión de los hilos con la clase thread, cuyos estados y ciclo de vida se parecen mucho a los procesos: nuevo, preparado, en ejecución, bloqueado, dormido, en espera o acabado.

Una de las tareas más importantes como programador en entornos multihilos que se ejecutan concurrentemente y comparten recursos es la sincronización del acceso a estos recursos para evitar inconsistencia de datos o interferencias.

Si queremos que los métodos o partes de código de una clase se ejecuten en exclusión mutua, tendremos que posar el modificador synchronized a sus métodos o partes de código que formen parte de la sección crítica.

Como hemos visto, la planificación de hilos se basa en qué política se tiene que seguir para decidir qué hilo coge el control del procesador y en qué momento. Esta es quizá una de las tareas más importantes y complicadas cuando realizamos programación multihilo.