HILOS JAVA

(THREADS)

Ing. Laura Sandoval Montaño, Ing. Manuel Enrique Castañeda Castañeda
PAPIME 104911

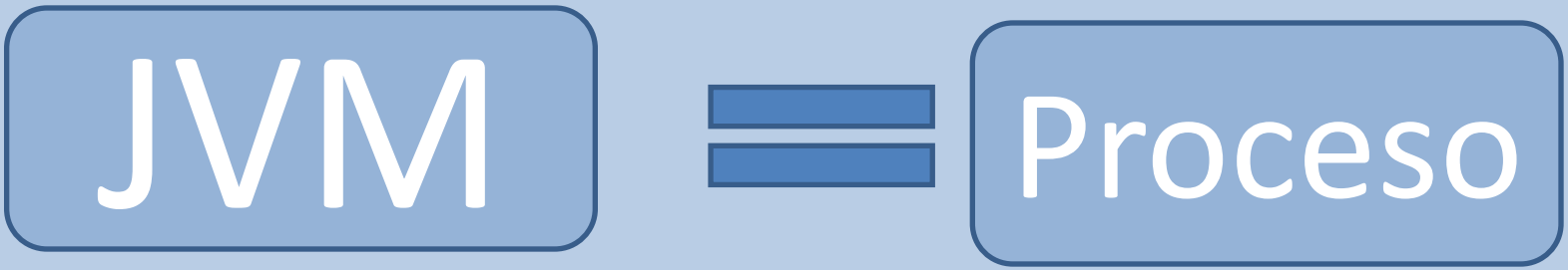
HILOS JAVA

La Máquina Virtual Java (JVM) es un sistema multihilo.

La JVM gestiona todos los detalles, asignación de tiempos de ejecución, prioridades, etc., de forma similar a como gestiona un Sistema Operativo múltiples procesos.

HILOS JAVA

La diferencia básica entre un proceso de Sistema Operativo y un **Thread** Java es que los hilos corren dentro de la JVM



HILOS JAVA

Desde el punto de vista de las aplicaciones los hilos son útiles porque permiten que el flujo del programa sea divido en dos o más partes, cada una ocupándose de alguna tarea de forma independiente

Todos los programas con interfaz gráfica (**AWT o Swing)** **son multihilo** porque los eventos y las rutinas de dibujado de las ventanas corren en un hilo distinto al principal.

HILOS JAVA

En Java un hilo o Thread es una instancia de la clase **java.lang.Thread**

http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.html

Una instancia se refiere a la declaración de un objeto, por ejemplo:

Object objTest = new Object(); // Instancia e inicialización

HILOS JAVA

Si analizamos la estructura de dicha clase podremos encontrar bastantes métodos que nos ayudan a controlar el comportamiento de los hilos

http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.html

HILOS JAVA

Métodos que siempre tenemos que tener presentes con respecto a los hilos son:

Método	Descripción
start()	Comienza la ejecución de un hilo, JVM llama al método run().
run()	Método que contiene el código a ejecutar por el hilo
yield()	Establecer la prioridad de un hilo
sleep()	Pausa un poco la ejecución de un proceso en particular forzándolo a dormir durante un tiempo determinado.

HILOS JAVA

Para definir e instanciar un nuevo Thread (hilo, proceso) existen 2 formas:

1. Extendiendo (o heredando) a la clase ***java.lang.Thread***

Herencia: La herencia es uno de los mecanismos de la programación orientada a objetos, por medio del cual una clase se deriva de otra; significa que las subclases disponen de todos los métodos y propiedades de su superclase

HILOS JAVA

Extendiendo a la clase ***java.lang.Thread***

La forma más directa para hacer un programa multihilo es extender la clase **Thread, y redefinir** el método **run().**

Este método es invocado cuando se inicia el hilo (mediante una llamada al método **start() de la clase Thread**).

HILOS JAVA

Extendiendo a la clase ***java.lang.Thread***

El hilo se inicia con la llamada al método **run() y** termina cuando termina éste.

Ejemplo:

HILOS JAVA

1. Extendiendo a la clase ***java.lang.Thread***

```
/*La clase Thread está en el paquete java.lang. Por tanto, no es
necesario el import.*/

public class ThreadEjemplo extends Thread {

/*El constructor public Thread(String str) recibe un parámetro que es la
identificación del Thread.*/

   public ThreadEjemplo(String str) {
   super(str);  /*super, sólo representa la parte heredada de la
                                clase base*/
}
```

HILOS JAVA

```
/*El método run() contiene el bloque de ejecución del Thread.
    Dentro de él, el método getName() devuelve el nombre del
    Thread (el que se ha pasado como argumento al
    constructor).*/

public void run() {

for (int i = 0; i < 10 ; i++)
   System.out.println(i + " " + getName());

System.out.println("Termina thread " + getName());
}
```

HILOS JAVA

/*El método main crea dos objetos de clase ThreadEjemplo y los inicia con la llamada al método start()
El hilo cambia de estado nuevo o new a estado de ejecución o runnable.
Cuando el hilo tenga su turno de ejecutarse, el método run() del objeto al que refiere se ejecuta*/

```
public static void main (String [] args) {

    new ThreadEjemplo("Pepe").start();
    new ThreadEjemplo("Juan").start();
    System.out.println("Termina thread main");
}
}
```

HILOS JAVA

```
public class ThreadEjemplo extends Thread {
public ThreadEjemplo(String str) {
super(str);
}
public void run() {
for (int i = 0; i < 10 ; i++)
System.out.println(i + " " + getName());
System.out.println("Termina thread " + getName());
}
public static void main (String [] args) {
new ThreadEjemplo("Pepe").start();
new ThreadEjemplo("Juan").start();
System.out.println("Termina thread main");
}
}
```

HILOS JAVA

Termina thread main
0 Pepe
1 Pepe
2 Pepe
3 Pepe
0 Juan
4 Pepe
1 Juan
5 Pepe
2 Juan
6 Pepe
3 Juan
7 Pepe
4 Juan
8 Pepe
5 Juan
9 Pepe
6 Juan
Termina thread Pepe
7 Juan
8 Juan
9 Juan
Termina thread Juan

HILOS JAVA

En la salida el primer mensaje de finalización es el del thread main. La ejecución de los hilos es asíncrona. Realizada la llamada al método start(), éste le devuelve control y continua su ejecución, independiente de los otros hilos.

En la salida los mensajes de un hilo y otro se van mezclando. La máquina virtual asigna tiempos a cada hilo.

HILOS JAVA

Para definir e instanciar un nuevo Thread existen 2 formas:

2. Implementando la interfaz ***Runnable***

La interfaz Runnable proporciona un método alternativo a la utilización de la clase Thread

HILOS JAVA

Interfaz: Sistema que permite que entidades inconexas o no relacionadas interactúen entre sí. Tiene como utilidad:

- Captar similitudes entre clases no relacionadas sin forzar entre ellas una relación
- Declarar métodos que una o más clases deben implementar en determinadas situaciones

HILOS JAVA

Ejemplo:

```
/*Se implementa la interfaz Runnable en lugar de extender la clase Thread.
El constructor que había antes no es necesario.*/

public class ThreadEjemplo1 implements Runnable {
public void run() {
for (int i = 0; i < 5 ; i++)
   System.out.println(i + " " +    Thread.currentThread().getName());
System.out.println("Termina thread " +
                          Thread.currentThread().getName());
}
```

HILOS JAVA

/*Primero se crea la instancia de nuestra clase.
Después se crea una instancia de la clase Thread, pasando como parámetros la referencia de nuestro objeto y el nombre del nuevo thread.
Por último se llama al método start de la clase thread. Este método iniciará el nuevo thread y llamará al método run() de nuestra clase.*/

```
public static void main (String [] args) {
    ThreadEjemplo1 ejemplo = new ThreadEjemplo1();
    Thread thread = new Thread (ejemplo, "Pepe");
    thread.start();
    ThreadEjemplo1 ejemplo1 = new ThreadEjemplo1();
    Thread thread1 = new Thread (ejemplo1, "Juan");
    thread1.start();
    System.out.println("Termina thread main");
}
}
```

HILOS JAVA

Por último, obsérvese que la llamada al método getName() desde run(). getName es un método de la clase Thread, por lo que nuestra clase debe obtener una referencia al thread propio. Es lo que hace el método estático currentThread() de la clase Thread.

```
System.out.println(i + " " + Thread.currentThread().getName());
```

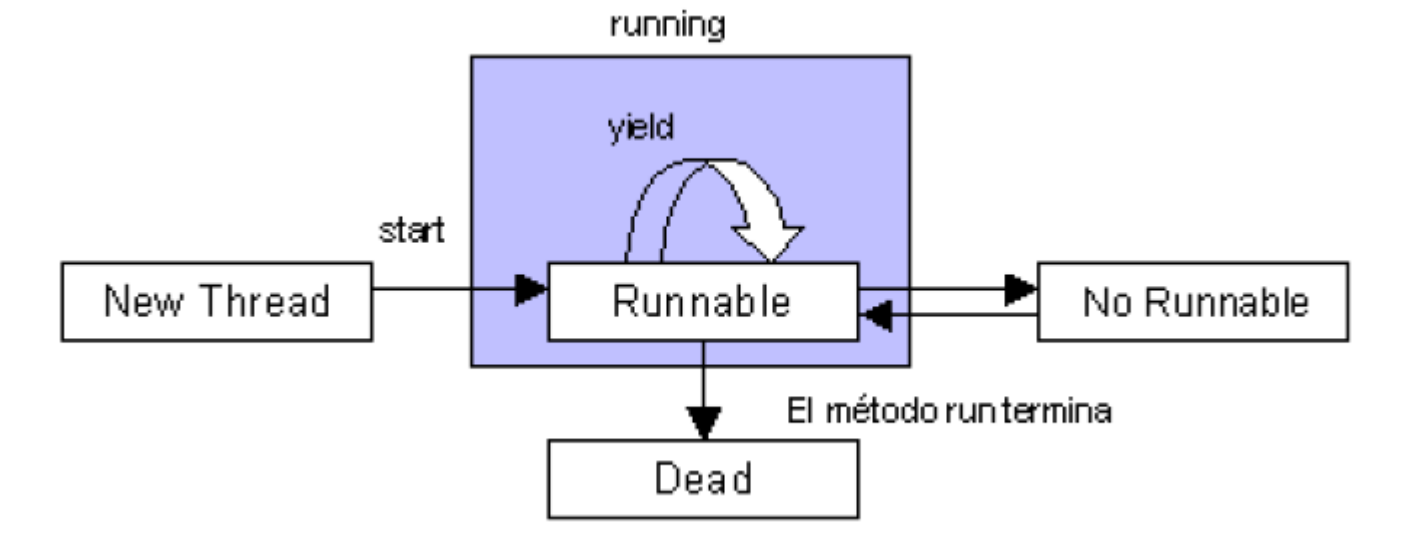
HILOS JAVA

Código completo:

```
public class ThreadEjemplo1 implements Runnable {
public void run() {
for (int i = 0; i < 5 ; i++)
    System.out.println(i + " " + Thread.currentThread().getName());
System.out.println("Termina thread " + Thread.currentThread().getName());
}
public static void main (String [] args) {
   ThreadEjemplo1 ejemplo = new ThreadEjemplo1();
   Thread thread = new Thread (ejemplo, "Pepe");
    thread.start();
   ThreadEjemplo1 ejemplo1 = new ThreadEjemplo1();
   Thread thread1 = new Thread (ejemplo1, "Juan");
   thread1.start();
   System.out.println("Termina thread main");
}
}
```

HILOS JAVA

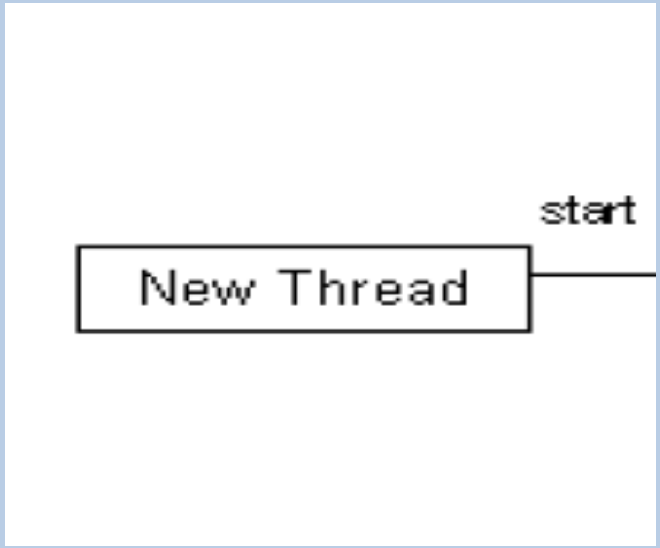
Ciclo de vida de un hilo en java



HILOS JAVA

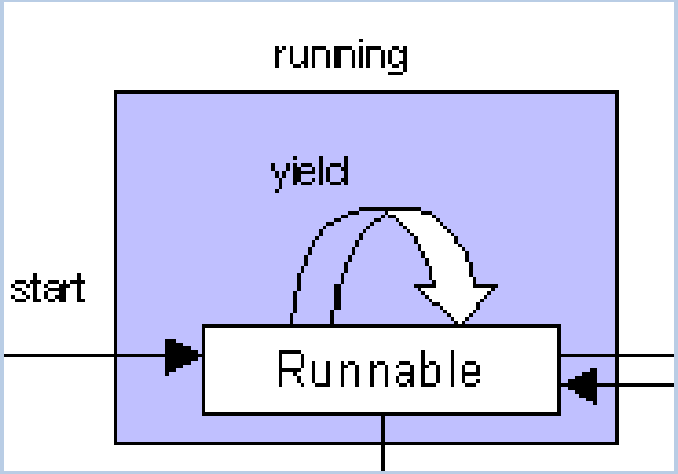
Ciclo de vida de un hilo en java

Nuevo (new): Este es el estado en que un hilo se encuentra después de que un objeto de la clase Thread ha sido instanciado pero antes de que el método ***start()*** sea llamado.



HILOS JAVA

Ciclo de vida de un hilo en java

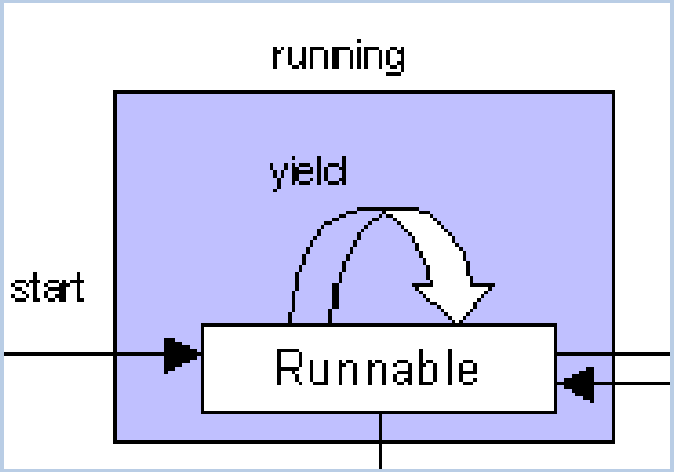


En ejecución Runnable: Este es el estado en que un hilo puede ser elegido para ser ejecutado por el programador de hilos pero aún no está corriendo en el procesador. Se obtiene este estado inmediatamente después de hacer la llamada al método ***start()*** de una instancia de la clase Thread.

HILOS JAVA

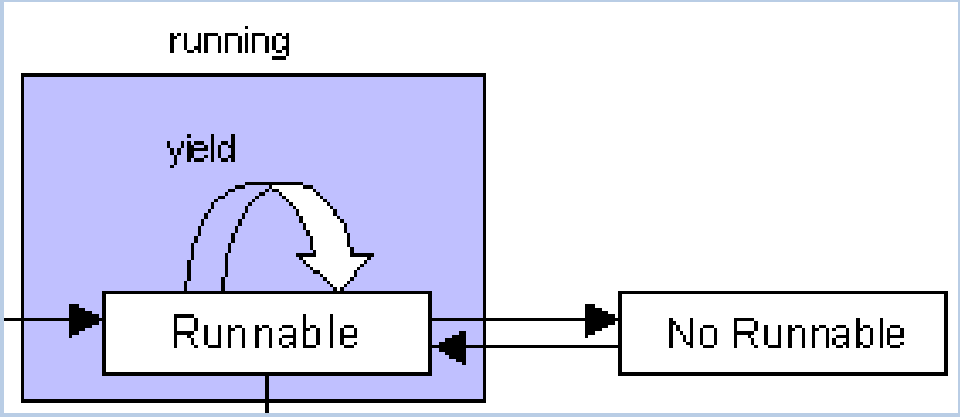
Ciclo de vida de un hilo en java

Ejecutándose (running): Este es el estado en el que el hilo está realizando lo que debe de hacer, es decir, está realizando el trabajo para el cual fue diseñado.



HILOS JAVA

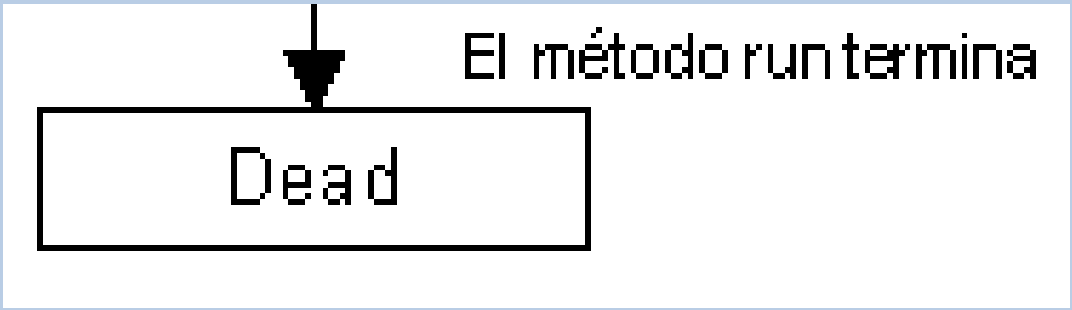
Ciclo de vida de un hilo en java



Esperando/bloqueado/dormido (waiting/blocked/sleeping) : Es el estado en el cual el hilo está vivo aún pero no es elegible para ser ejecutado, es decir, no está en ejecución pero puede estarlo nuevamente si algún evento en particular sucede.

HILOS JAVA

Ciclo de vida de un hilo en java



Un hilo está muerto cuando se han completado todos los procesos y operaciones contenidos en el método ***run()***. Una vez que un hilo ha muerto **NO** puede volver nunca a estar vivo, no es posible llamar al método s***tart()*** más de una vez para un solo hilo.

HILOS JAVA

Planificador de hilos

En la JVM el planificador de hilos es el encargado de decidir qué hilo es el que se va a ejecutar por el procesador en un momento determinado y cuándo es que debe de parar o pausar su ejecución.

Algunos métodos de la clase ***java.lang.Thread*** nos pueden ayudar a influenciar al planificador de hilos a tomar una decisión sobre qué hilo ejecutar y en qué momento. Los métodos son los siguientes:

HILOS JAVA

Planificador de hilos

Método	Descripción
yield()	Establecer la prioridad de un hilo, permitiendo cambiar el estado de un hilo de runnable a running
sleep()	Pausa un poco la ejecución de un proceso en particular forzándolo a dormir durante un tiempo determinado.
join()	Permite al hilo "formarse en la cola de espera" de otro hilo
setPriority	Establece la prioridad de un hilo

HILOS JAVA

Hilo Durmiendo (sleeping)

El método *sleep() es un método estático de la clase Thread. Generalmente lo usamos en el* código para pausar un poco la ejecución de un proceso en particular forzándolo a dormir durante un tiempo determinado.

Para invocar a un método estático **no se necesita crear un objeto de la** clase en la que se define:
Si se invoca desde la clase en la que se encuentra definido, basta con escribir su nombre.
Si se le invoca desde una clase distinta, debe anteponerse a su nombre, el de la clase en la que se encuentra seguido del operador punto (.) <NombreClase>.métodoEstático

HILOS JAVA

Hilo Durmiendo (sleeping)

Para forzar un hilo a dormir podemos usar un código parecido a lo siguiente:

```
try{
    Thread.sleep(5*60*1000); //Duerme durante 5 minutos
}catch(InterruptedException ex){}
```

Normalmente cuando llamamos al método ***sleep() encerramos el código en un bloque try/catch debido*** a que dicho método arroja una excepción.

Excepción: es la indicación de un problema que ocurre durante la ejecución de una aplicación

HILOS JAVA

Hilo Durmiendo (sleeping)

Consideraciones:

Si un hilo deja de dormir, no significa que volverá a estar ejecutándose al momento de despertar; el tiempo especificado dentro del método sleep() es el mínimo de tiempo que un hilo debe de dormir, aunque puede ser mayor.

Un hilo no puede poner a dormir a otro, el método sleep() siempre afecta al hilo que se encuentra ejecutando al momento de hacer la llamada.

HILOS JAVA

Prioridades de un hilo

Recordando:

Aunque un programa utilice varios hilos y aparentemente estos se ejecuten simultáneamente, el sistema ejecuta una única instrucción cada vez, aunque las instrucciones se ejecutan concurrentemente (entremezclándose).

HILOS JAVA

Prioridades de un hilo

El mecanismo por el cual un sistema controla la ejecución concurrente de procesos se llama planificación (*scheduling).*

Java soporta un mecanismo simple denominado planificación por prioridad fija (*fixed priority scheduling).*

Esto significa que la planificación de los hilos se realiza con base en la prioridad relativa de un hilo frente a las prioridades de otros.

HILOS JAVA

Prioridades de un hilo

La prioridad de un hilo es un valor entero, del uno al diez, que puede asignarse con el método **setPriority.**

La clase Thread tiene 3 constantes (variables estáticas y finales) que definen un rango de prioridades de hilo:

- Thread.MIN_PRIORITY (1)
- Thread.NORM_PRIORITY (5)
- Thread.MAX_PRIORITY (10)

Por defecto la prioridad de un hilo es igual a la del hilo que lo creó.

HILOS JAVA

Prioridades de un hilo

Cuando hay varios hilos en condiciones de ser ejecutados (estado runnable), la JVM elige el hilo que tiene una prioridad más alta, que se ejecutará hasta que:

- Un hilo con una prioridad más alta esté en condiciones de ser ejecutado (runnable), o
- El hilo termina (termina su método **run**), o
- Se detiene voluntariamente, o
- Alguna condición hace que el hilo no sea ejecutable (runnable), como una operación de entrada/salida o, si el sistema operativo tiene planificación por división de tiempos (*time slicing), cuando expira el tiempo asignado.*

HILOS JAVA

Prioridades de un hilo

Si dos o más hilos están listos para ejecutarse y tienen la misma prioridad, la máquina virtual va cediendo control de forma cíclica (*round-robin).*

El hecho de que un hilo con una prioridad más alta interrumpa a otro se denomina 'planificación apropiativa' (*preemptive scheduling).*

Cuando un hilo entra en ejecución y no cede voluntariamente el control para que puedan ejecutarse otros hilos, se dice que es un hilo egoísta (*selfish thread).*

HILOS JAVA

Prioridades de un hilo

En estas condiciones el Sistema Operativo asigna tiempos a cada hilo y va cediendo el control consecutivamente a todos los que compiten por el control de la CPU, impidiendo que uno de ellos se apropie del sistema durante un intervalo de tiempo prolongado.

Este mecanismo lo proporciona el sistema operativo, no Java.

```
public class ThreadEjemplo3 implements Runnable {
   public void run() {
      for (int i = 0; i < 5 ; i++)
         System.out.println(i + " " + Thread.currentThread().getName());
      System.out.println("Termina thread " + Thread.currentThread().getName());
   }
   public static void main (String [] args) {
      ThreadEjemplo3 ejemplo = new ThreadEjemplo3();
      Thread thread = new Thread (ejemplo, "Pepe");

      thread.setPriority(Thread.MIN_PRIORITY);

      ThreadEjemplo3 ejemplo1 = new ThreadEjemplo3();
      Thread thread1 = new Thread (ejemplo1, "Juan");

      thread1.setPriority(Thread.MAX_PRIORITY);

      thread.start();
      thread1.start();
      System.out.println("Termina thread main");
   }
}
```

Ejemplo: ThreadEjemplo3.java

HILOS JAVA

Prioridades de un hilo

yield()

Tiene la función de hacer que un hilo que se está ejecutando dé regreso al estado en ***ejecución(runnable)*** para permitir que otros hilos de la misma prioridad puedan ejecutarse.

El método ***yield()*** nunca causará que un hilo pase a estado de espera/bloqueado/dormido, simplemente pasa de **ejecutándose(running)** a en **ejecución(runnable)**.

```
public class ThreadEjemplo5 implements Runnable {
   public void run() {
      for (int i = 0; i < 5 ; i++)
          System.out.println(i + " " + Thread.currentThread().getName());
      System.out.println("Termina thread " + Thread.currentThread().getName());
   }
   public static void main (String [] args) {
       ThreadEjemplo5 ejemplo = new ThreadEjemplo5();
       Thread thread = new Thread (ejemplo, "Pepe");

        thread.setPriority(Thread.MIN_PRIORITY);

        ThreadEjemplo5 ejemplo1 = new ThreadEjemplo5();
        Thread thread1 = new Thread (ejemplo1, "Juan");

        thread1.setPriority(Thread.MAX_PRIORITY);

        thread.start();
        thread1.start();
        thread1.yield();

        System.out.println("Termina thread main");
    }
}
```

Ejemplo: ThreadEjemplo5.java

HILOS JAVA

Prioridades de un hilo

join()

El método no estático ***join()*** *permite al hilo "formarse en la cola de espera" de otro hilo.*

Si se tiene un hilo B que no puede comenzar a ejecutarse hasta que se complete el proceso del hilo A, entonces se necesita que B se forme en la cola de espera de A. Esto significa que B nunca podrá ejecutarse si A no completa su proceso.

HILOS JAVA

Prioridades de un hilo

join()

En código se utiliza así:

```
Thread t = new Thread();
t.start();
t.join();
```

```
public class ThreadEjemplo4 implements Runnable {
public void run() {
for (int i = 0; i < 5 ; i++)
System.out.println(i + " " + Thread.currentThread().getName());
System.out.println("Termina thread " + Thread.currentThread().getName());
}
public static void main (String [] args) {

try{
ThreadEjemplo4 ejemplo = new ThreadEjemplo4();
Thread thread = new Thread (ejemplo, "Pepe");

ThreadEjemplo4 ejemplo1 = new ThreadEjemplo4();
Thread thread1 = new Thread (ejemplo1, "Juan");

thread.start();
thread.join();

thread1.start();
thread1.join();

System.out.println("Termina thread main");
  }
 catch(Exception e){
            System.out.println(e.toString());
                        }
  }
```

Ejemplo: ThreadEjemplo4.java

HILOS JAVA

Prioridades de un hilo

2 hilos que están accediendo:

A la misma instancia de una clase, ejecutando el mismo método y accediendo incluso al mismo objeto y mismas variables, cada uno cambiando el estado primario de dicho objeto prácticamente de manera simultánea, lo mismo sucede con las variables.

HILOS JAVA

Si los datos que se están modificando indican al programa cómo funcionar, el pensar en esta situación originaría un resultado desastroso.

```
class CuentaBanco {
    private int balance = 50;

    public int getBalance(){
        return balance;
    }

    public void retiroBancario(int retiro){
        balance = balance - retiro;
    }
}
```

Ejemplo: PeligroCuenta.java

```
    public class PeligroCuenta implements Runnable{

        private CuentaBanco cb = new CuentaBanco();

        public void run(){
        for(int x = 0; x <10; x++)
            hacerRetiro(10);
        if (cb.getBalance()<0)
            System.out.println("La cuenta está sobregirada.");
        }
```

```
    private void hacerRetiro(int cantidad){
    if(cb.getBalance()>=cantidad)
    {
        System.out.println(Thread.currentThread().getName()+" va a hacer un retiro.");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        cb.retiroBancario(cantidad);
        System.out.println(Thread.currentThread().getName() + " realizó el retiro con éxito.");
    }
    else {
        System.out.println("No ha suficiente dinero en la cuenta para realizar el retiro Sr." +
                                Thread.currentThread().getName());
        System.out.println("su saldo actual es de "+cb.getBalance());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

HILOS JAVA

```
    public static void main (String[] args)
   {
        PeligroCuenta pl = new PeligroCuenta();
        Thread uno = new Thread(pl);
        Thread dos = new Thread(pl);
        uno.setName("Luis");
        dos.setName("Manuel");

        uno.start();
        dos.start();

   }
}
```

Luis va a hacer un retiro.
Manuel va a hacer un retiro.
Manuel realizó el retiro con éxito.
Luis realizó el retiro con éxito.
Luis va a hacer un retiro.
Manuel va a hacer un retiro.
Manuel realizó el retiro con éxito.
Manuel va a hacer un retiro.
Luis realizó el retiro con éxito.
Luis va a hacer un retiro.
Luis realizó el retiro con éxito.
No ha suficiente dinero en la cuenta para realizar el retiro Sr.Luis
su saldo actual es de -10
Manuel realizó el retiro con éxito.
No ha suficiente dinero en la cuenta para realizar el retiro Sr.Manuel
su saldo actual es de -10
No ha suficiente dinero en la cuenta para realizar el retiro Sr.Manuel
su saldo actual es de -10
No ha suficiente dinero en la cuenta para realizar el retiro Sr.Luis
su saldo actual es de -10
La cuenta está sobregirada.
La cuenta está sobregirada.

HILOS JAVA

Mientras Luis estaba checando el estado de cuenta y vio que era posible el retiro, Manuel estaba retirando y viceversa, finalmente, Manuel verificó que había 10 pesos en el saldo y decidió retirarlos, pero oh sorpresa! Luis los acababa de retirar, sin embargo el retiro de Manuel también se completó dejando la cuenta sobregirada.

HILOS JAVA

A dicho escenario se le llama **"condición de carrera"**, cuando 2 o más procesos pueden acceder a las mismas variables y objetos al mismo tiempo y los datos pueden corromperse si un proceso **"corre"** lo suficientemente rápido como para vencer al otro.

HILOS JAVA

¿Qué hacemos para proteger los datos? Dos cosas:

- **Marcar las variables como privadas.**

  Para marcar las variables como privadas utilizamos los identificadores de control de acceso, en este caso la palabra private.

- **Sincronizar el código que modifica las variables.**

  Para sincronizar el código utilizamos la palabra synchronized

```
private synchronized void hacerRetiro(int cantidad){
    if(cb.getBalance()>=cantidad)
    {
        System.out.println(Thread.currentThread().getName()+" va a hacer un retiro.");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }

        cb.retiroBancario(cantidad);
        System.out.println(Thread.currentThread().getName() + " realizó el retiro con éxito.");
    } else {
        System.out.println("No ha suficiente dinero en la cuenta para realizar el retiro Sr." +
        Thread.currentThread().getName());
        System.out.println("su saldo actual es de "+cb.getBalance());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
        ex.printStackTrace();
        }
    }
}
```

El método que realiza las operaciones con las variables es ***hacerRetiro()***, por lo tanto, es el método que necesitamos que sea privado y sincronizado

HILOS JAVA

Sincronización

Con un **seguro**

Cada objeto en Java posee un seguro que previene su acceso, dicho seguro se activa únicamente cuando el objeto se encuentra dentro de un método sincronizado.

Debido a que solo existe un seguro por objeto, una vez que un hilo ha adquirido dicho seguro, ningún otro hilo podrá utilizar el objeto hasta que su seguro sea liberado.

Puntos clave con la sincronización y el seguro de los objetos:

Solo métodos (o bloques) pueden ser sincronizados, nunca una variable o clase.

Cada objeto tiene solamente un seguro.

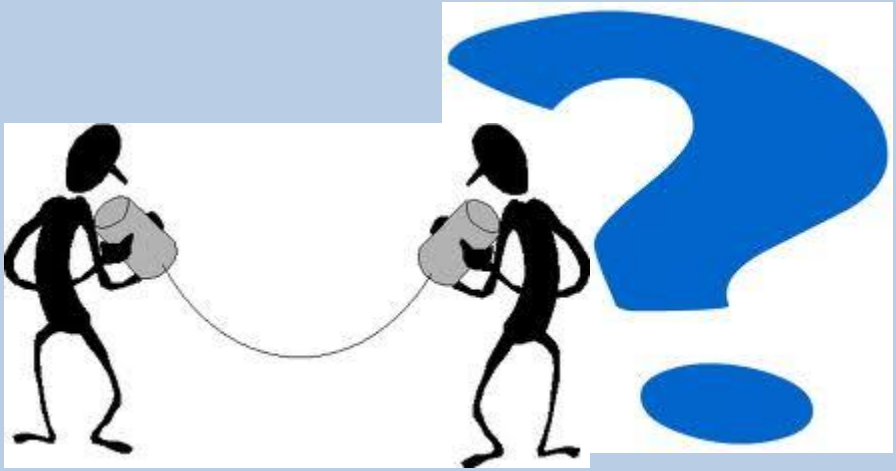
No todos los métodos de una clase deben ser sincronizados, una misma clase puede tener métodos sincronizados y no sincronizados.

Si una clase tiene ambos tipos de métodos, múltiples hilos pueden acceder a sus métodos no sincronizados, el único código protegido es aquel dentro de un método sincronizado.

Si un hilo pasa a estado dormido(sleep) no libera el o los seguros que pudiera llegar a tener, los mantiene hasta que se completa.

Se puede sincronizar un bloque de código en lugar de un método.

HILOS JAVA



Método	Descripción
wait()	Posiciona al hilo en la lista de espera del objeto
notify()	Libera el seguro del objeto para un hilo
notifyAll()	Libera el seguro del objeto para un hilo para todos los hilos que tenga en espera

Consideraciones

Si existen varios hilos en la cola de espera del objeto, solo uno podrá ser escogido (sin un orden garantizado) por el planificador de hilos para acceder a dicho objeto y obtener su seguro.

wait(), notify() y notifyAll() deben ser llamados desde dentro de un contexto sincronizado.

```
public class ThreadA {

    public ThreadA() {
    }

    public static void main(String[] args)
    {
        ThreadB b = new ThreadB();
        b.start();

        synchronized(b){
            try {
                System.out.println("Esperando a que B se
                                    complete...");
                b.wait();
            } catch (InterruptedException ex) { }

            System.out.println("Total:" +b.total);
        }
    }
}
```

```
class ThreadB extends Thread{
 long total;

  public void run(){
    synchronized(this)
    {
    for(long i=0; i<1000000000;i++)
            total+=i;
     notify();
     }

  }
}
```

Un punto a tomar en cuenta es que para poder llamar al método ***wait()***, se tuvo que sincronizar el bloque de código y obtener el seguro de **b**

Ejemplo: ThreadA.java