HILOS JAVA

(THREADS)

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

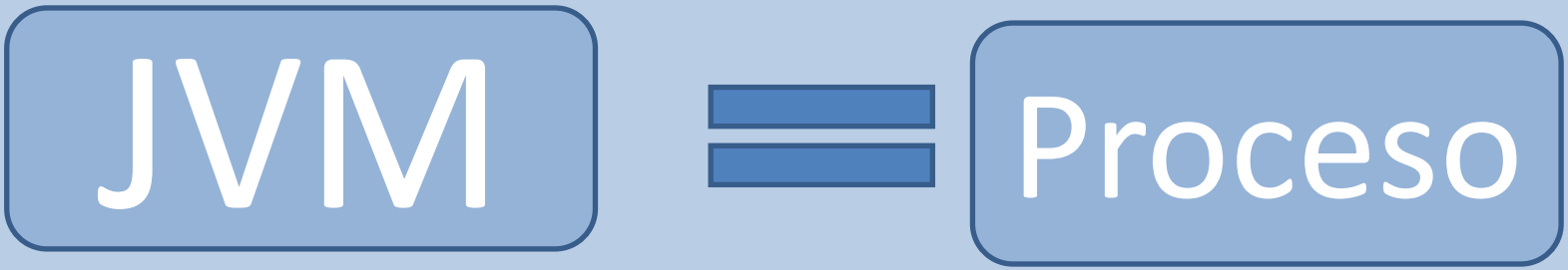
HILOS JAVA

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 interface gráfico (**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 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 inicializacion

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 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 interface 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 si, tiene como utilidad:

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

HILOS JAVA

Ejemplo:

```
/*Se implementa la interface 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());
```

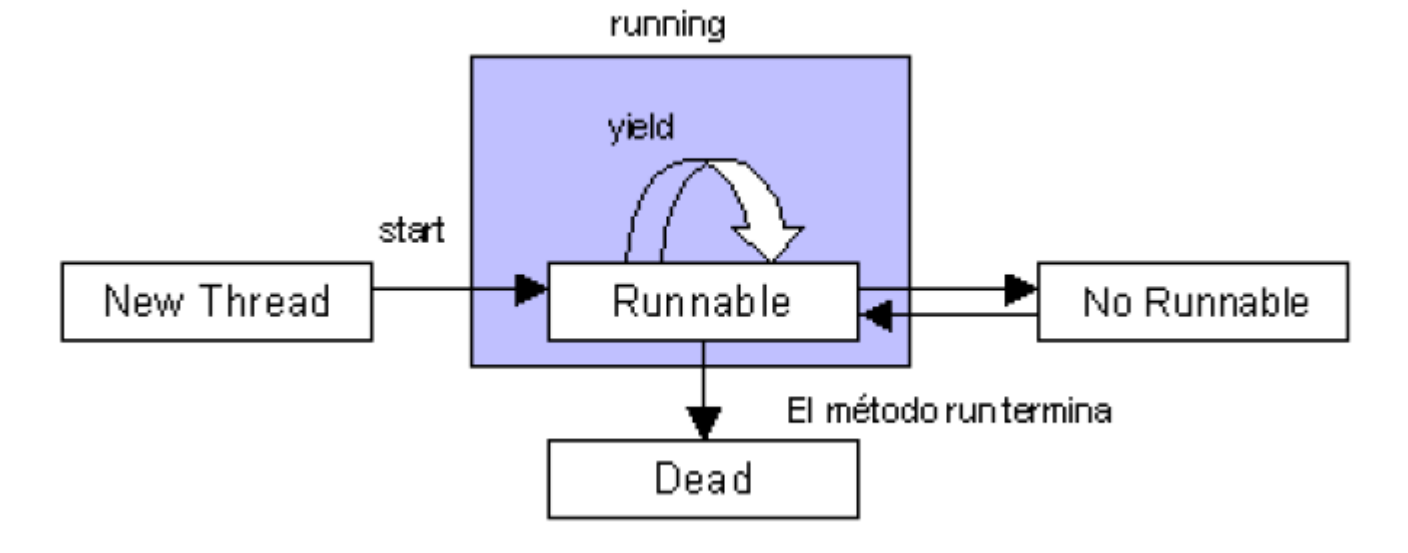
HILOS JAVA

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

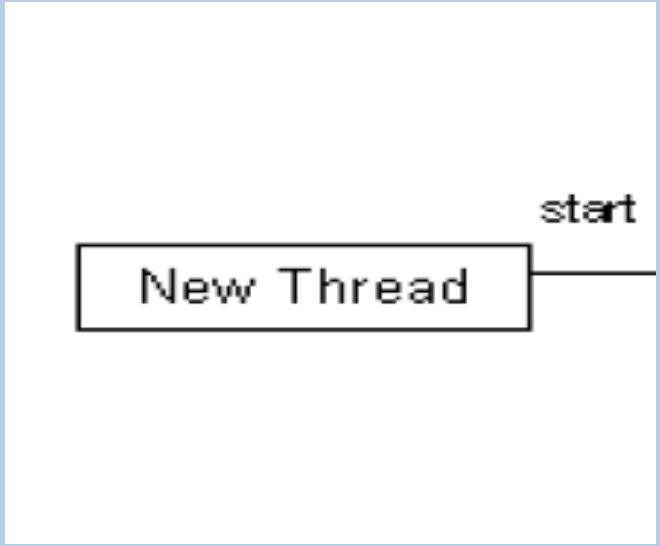
Ciclo de vida de un hilo en java



HILOS JAVA

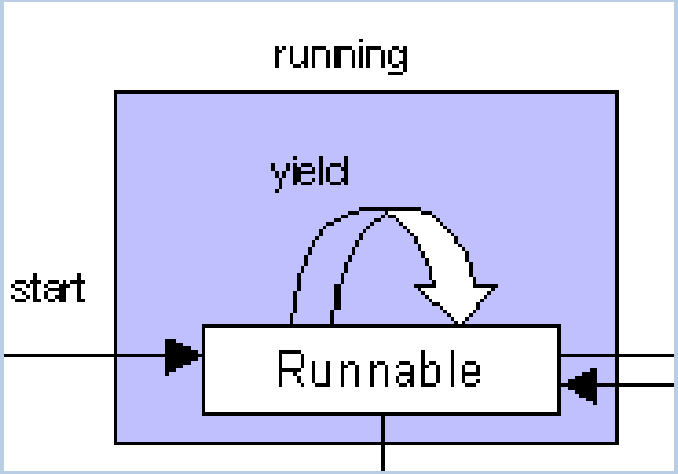
Ciclo de vida de un hilo en java

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

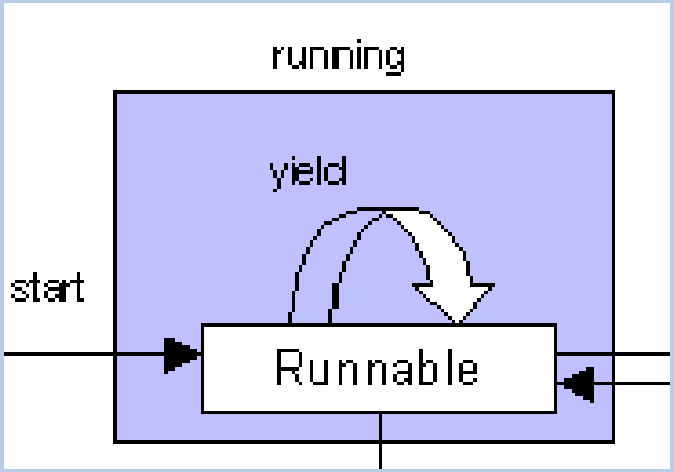


En ejecución Runnable: Este es el estado en que un hilo puede ser elegido para ser ejecutado por el programador de hilos pero aún no está corriendo en el procesador. Se obtiene este estado inmediatamente después de hacer la llamada al método ***start()*** de una instancia de la clase Thread.

HILOS JAVA

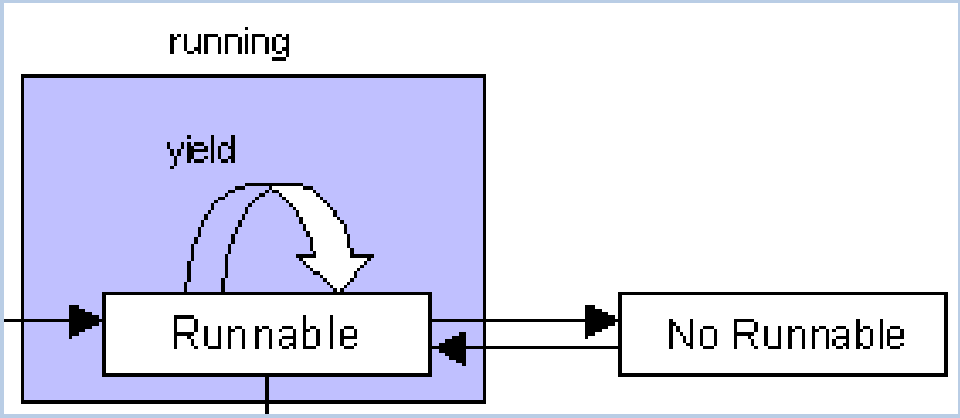
Ciclo de vida de un hilo en java

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

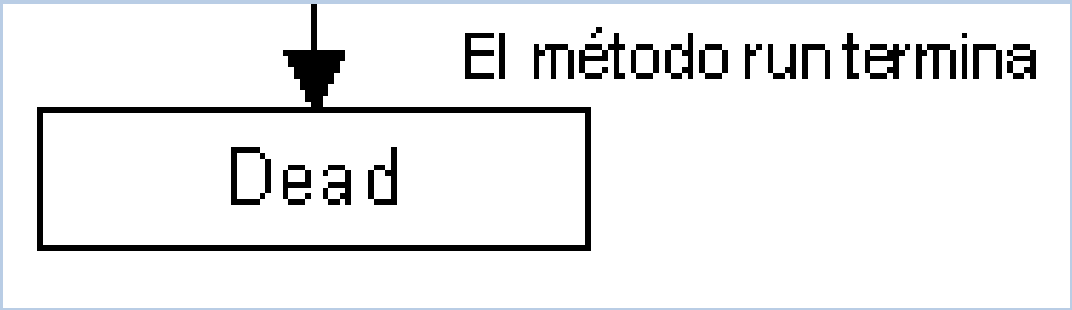
Ciclo de vida de un hilo en java



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

Programador de hilos

En la JVM el programador 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 programador de hilos a tomar una decisión sobre qué hilo ejecutar y en qué momento. Los métodos son los siguientes:

HILOS JAVA

Programador 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>.metodoEstatico

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 en base a 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 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 de 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. En código se utiliza así:

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.

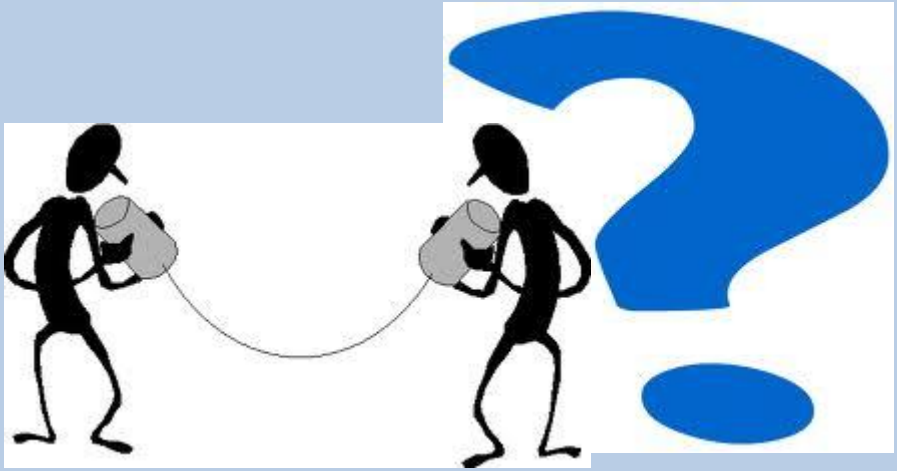
No todos los métodos de una clase deben ser sincronizados, una misma clase puede tener métodos sincronizados y no sincronizados.

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 programador 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

COMO OPTIMIZAR EL USO DE MULTI – NÚCLEOS O DE MULTIPROCESADORES EN JAVA

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

MULTIPROCESOS EN JAVA.

Existen dos maneras para implementar multiprocesos:

UNO: Instanciar un objeto Runnable y ejecutarlo con la llamada al método Thread.star(), sincronizando el fragmento de código que se desea paralelizar.

DOS: Implementando la interfaz Runnable o la interfaz Callable. Las tareas serán pasadas a través del ExecutorService del paquete java.util.concurrent.

MULTIPROCESOS EN JAVA.

Conseguir un buen rendimiento en un sistema multi – núcleo es una tarea difícil y llena de complicaciones.

1.-Los procesos de escritura y lectura se alentan, solo un hilo debe hacer uso a la vez de este recurso.

2.-La sincronización de los objetos proporciona seguridad a las operaciones multi-hilo, pero alenta el trabajo.

MULTIPROCESOS EN JAVA.

3.-Si las tareas son demasiado triviales (pequeños fragmentos de código de ejecución rápida) los gastos generales de su gestión son mayores de lo que se gana al usar múltiples núcleos.

4.-La programación en paralelo es no intuitiva, compleja y las abstracciones son débiles.

MULTIPROCESOS EN JAVA

Sincronizar threads

Todos los objetos tienen un bloqueo asociado, lock o cerrojo, que puede ser adquirido y liberado mediante el uso de métodos y sentencias synchronized.

La sincronización fuerza a que la ejecución de los dos hilos sea mutuamente exclusiva en el tiempo.

Mecanismos de bloqueo:

- Métodos synchronized (exclusión mutua).
- Bloques synchronized (regiones críticas).

Mecanismos de comunicación de los threads (variables de condición): Wait(), notify(),notifyAll()...

MULTIPROCESOS EN JAVA.

Métodos synchronized: Monitores

Los métodos de una clase Java se pueden declarar como synchronized. Esto significa que se garantiza que se ejecutará con régimen de exclusión mutua respecto de otro método del mismo objeto que también sea synchronized.

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

Bloques synchronized.

Es el mecanismo mediante el cual se implementan en Java las regiones críticas.

Un bloque de código puede ser definido como synchronized respecto de un objeto. En ese caso solo se ejecuta si se obtiene el lock asociado al objeto

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

CUANDO SINCRONIZAR.

En cualquier momento en el que dos o más threads acceden al mismo objeto o bloque de código.

CUANDO NO SINCRONIZAR

"

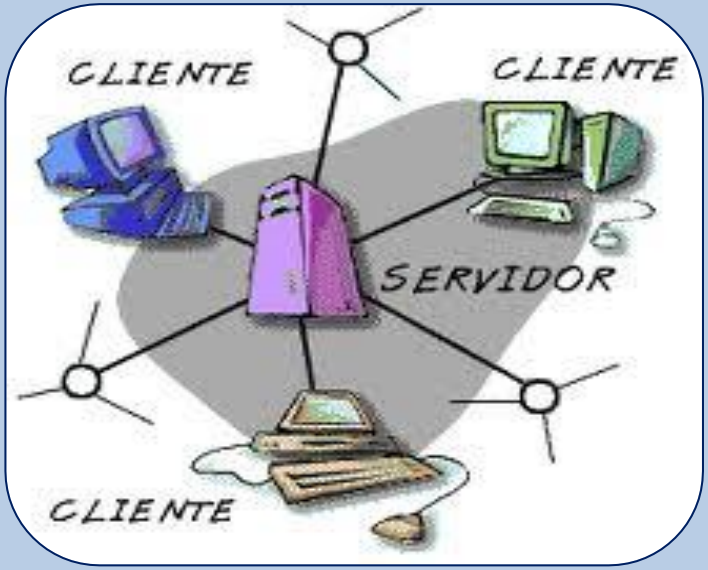
Sobresincronizar causa retrasos innecesarios, cuando dos o más threads no tratan de ejecutar el mismo bloque de código.

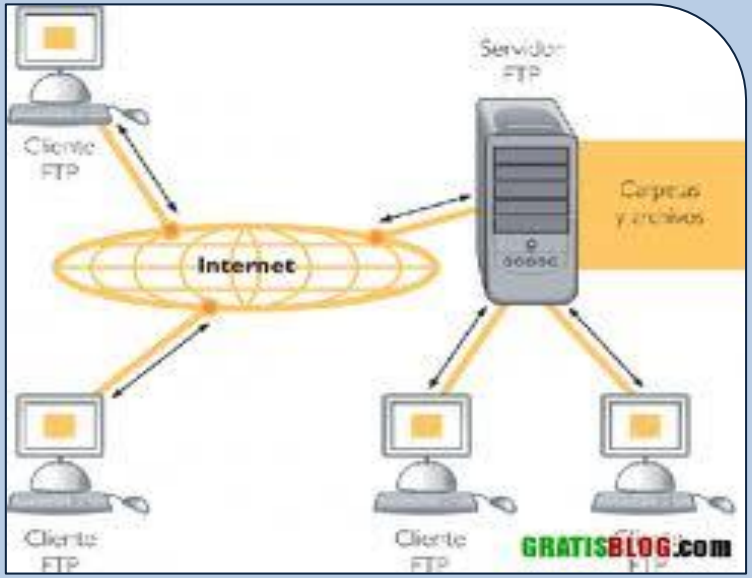
No sincronizar un método que usa variables locales. Las variables locales son almacenadas en el stack, y cada thread tiene su propio stack, así que, no habrá problemas de concurrencia, Ej. public int square(int n) {int s = n * n;return s;}

THREADS POOL

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

EN LAS APLICACIONES CON ESTRATEGIA CLIENTE/SERVIDOR, LAS TAREAS RESULTAN:





Para implementar los clientes

En los servidores para atender los requerimientos de los clientes.

TAREAS CONCURRENTES

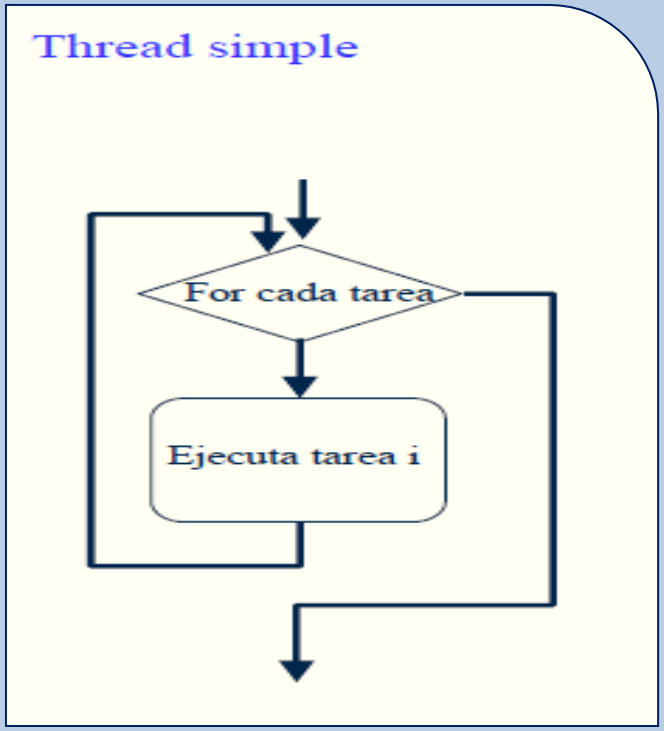
Organizar las actividades de los servicios como tareas concurrentes tiene como ventajas:

- Incrementa la capacidad de prestar servicios (*throughput).*
- Mejora los tiempos de respuesta (*responsiveness) con carga de trabajo normal.*
- Presenta una degradación gradual de prestaciones cuando la carga se incrementa.

ESTRATEGIAS DE EJECUCIÓN DE TAREAS

Hay diferentes estrategias para ejecutar las tareas que constituyen un programa:

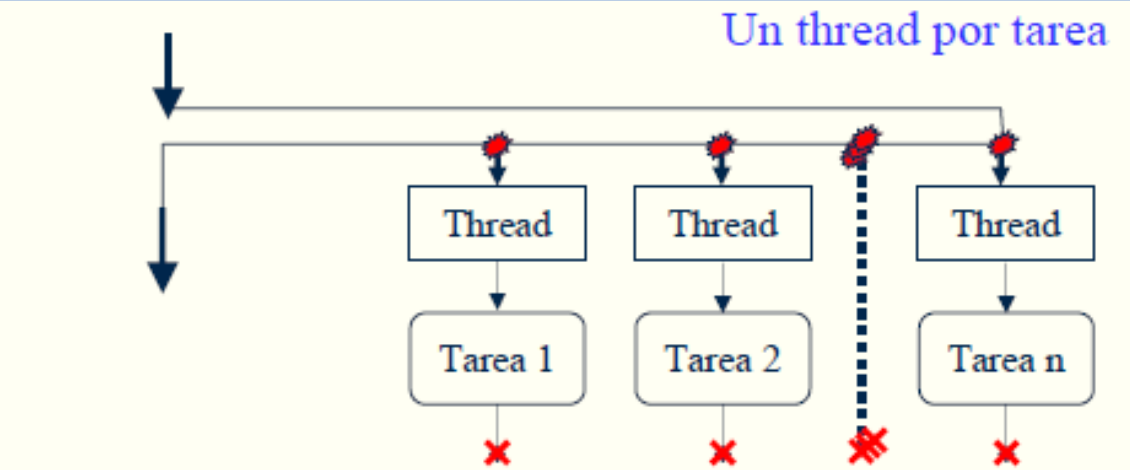
Ejecución secuencial de las tareas.



Ofrecen bajo throughput y tiempo de respuesta.

Creación explícita de un thread por cada tarea.

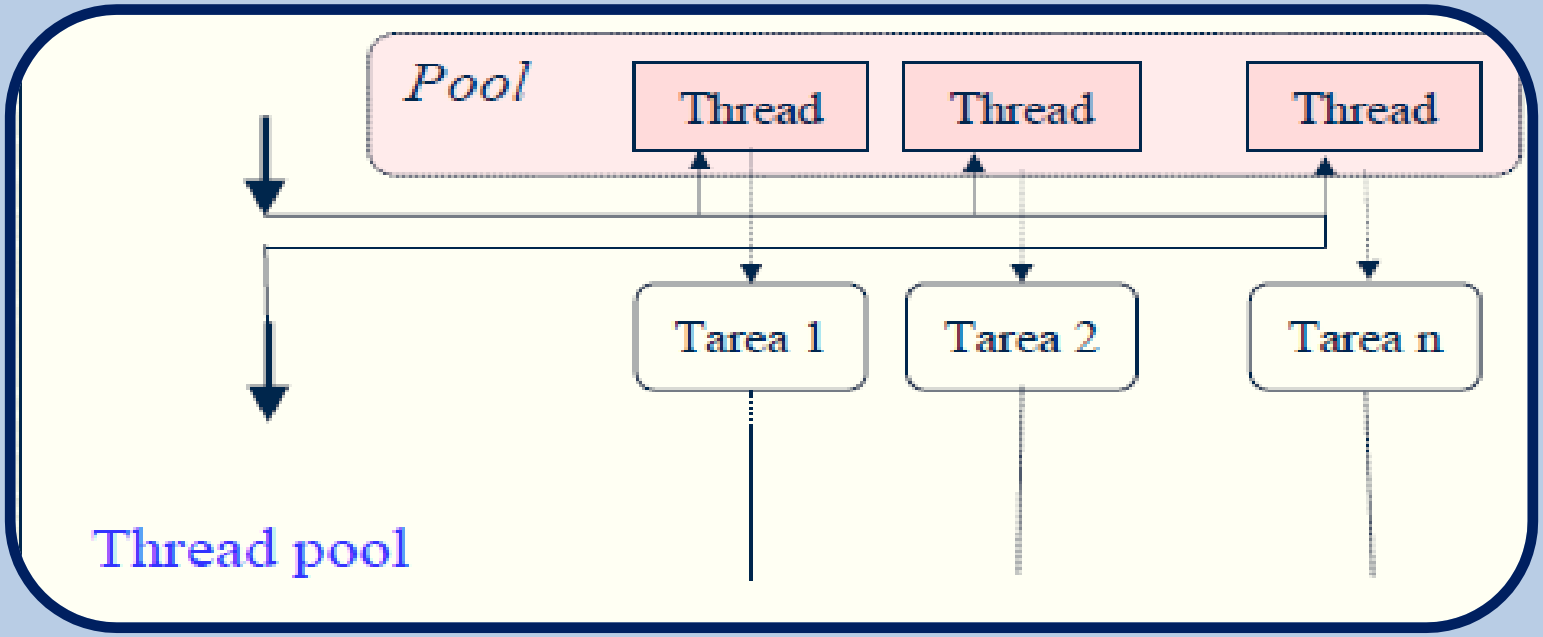
La ejecución concurrente de las tareas mejora el uso de los recursos.



Cuando las tareas son muy breves, la creación/destrucción de thread supone una sobre carga relevante.

Con baja carga tiene buena respuesta. Con alta carga tiene peligro de fallo por agotamiento de los recursos.

Creación de un grupo de thread (pool) para posteriormente aplicarlos



Se elimina la sobrecarga de creación y destrucción de thread.

Se limita la cantidad de recursos que se acaparan.

Se consigue la estabilidad con alta carga.

Executor framework

Executor es la base de un framework flexible y potente de ejecución asíncrona de tareas que independiza la definición de las tareas como unidades lógicas de trabajo y los threads como los mecanismos con los que las tareas se ejecutan concurrentemente y asíncronamente.

Framework es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de *software* concretos, con base a la cual, otro proyecto de *software* puede ser más fácilmente organizado y desarrollado.

Típicamente, puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto

Tiene como objeto diferenciar la interfaz java.util.concurrent.Executor:

http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Executor.html

```
public interface Executor{
void execute(Runnable command);
}
```

Flexibiliza el mantenimiento de programas concurrentes a lo largo de su ciclo de vida, posibilitando el cambio de las políticas de ejecución sin cambiar el código de la propia ejecución.

Políticas de ejecución

La utilidad de usar el framework Executor es poder modificar la política de ejecución de un conjunto de tareas con sólo cambiar a un tipo de ejecutor diferente.

Una política de ejecución incluye:

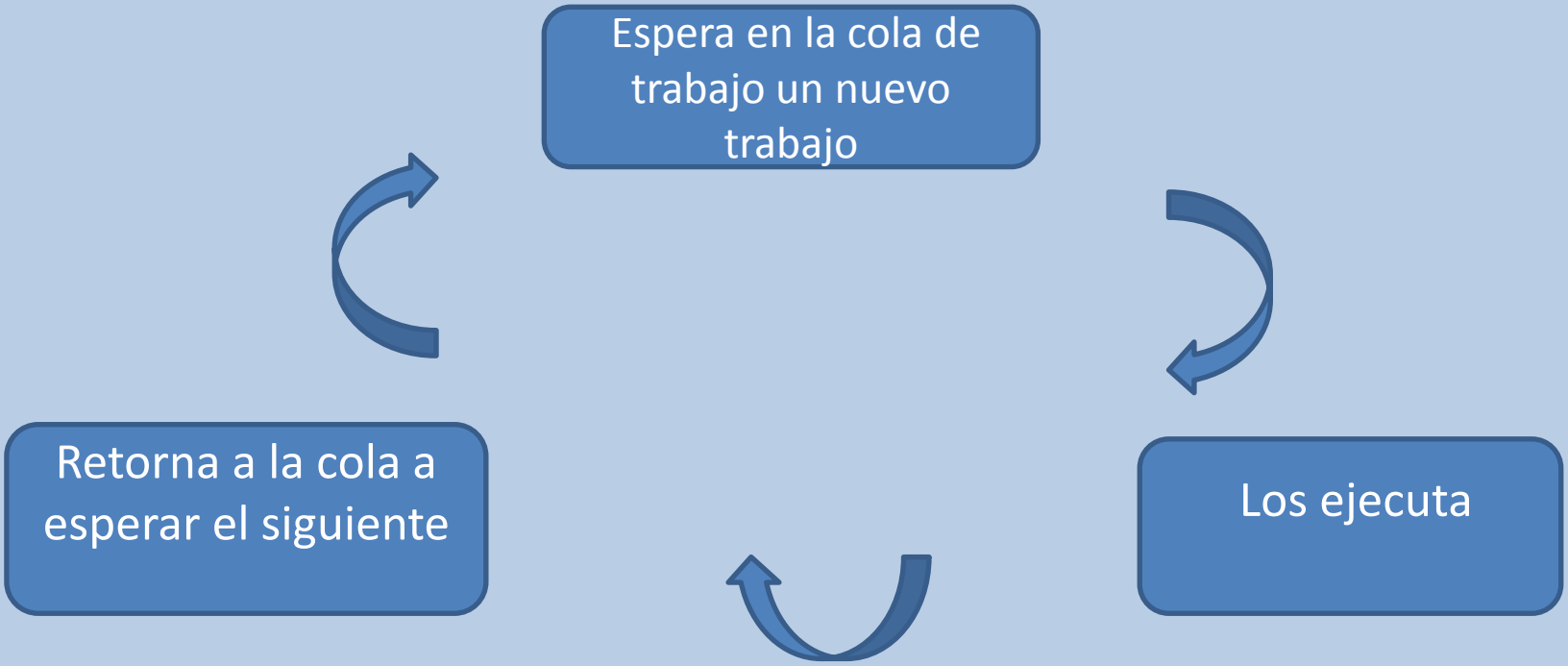
- Definir en que thread se ejecutan las tareas.
- Definir el orden en el que se ejecutan las tareas (FIFO,LIFO, Por Prioridad, etc.
- El número de tareas que se permiten ejecutar concurrentemente.
- El número de tareas que pueden encolarse en espera de ejecución.
- Selección de la tarea que debe ser rechazada si hay sobrecarga.
- Definir las acciones que deben ejecutarse antes y después de ejecutarse una tarea.

Depende de los recursos disponibles y de la calidad de servicio (*throughput) que se requiere.*

Thread pools

Un thread pool es una estructura que gestiona un conjunto homogéneo de threads dispuestos a ejecutar la tarea que se les encargue a través de una cola de trabajos

El ciclo de vida de un thread de un thread pool es muy simple:



Ejecutar tareas a través de un pool de threads tiene las siguientes ventajas:

Reutilizar Threads ya creados, implica ahorrase las actividades de creación y destrucción de nuevas tareas.

Disminuye el tiempo de respuesta, ya que las tareas están creadas cuando llega el requerimiento de ejecución del trabajo.

Regulando la dimensión del pool se puede equilibrar la carga con los recursos de CPU y memoria que se requieren.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Main {
    public static void main(String args[]) {

/*La parte más importante es Executors.newFixedThreadPool(5). Con esto estamos
definiendo un contenedor de un tamaño máximo de 5 espacios. En estos espacios se
insertaran hilos y se ejecutarán. Cuando uno de ellos termine su ejecución, se sacará el
hilo de ese espacio y se insertara otro de los que este en la cola.*/

        ExecutorService exec = Executors.newFixedThreadPool(5);

        for (int i = 0; i < 100; i++) {
            exec.execute(new Runnable() {
                public void run() {
                    System.out.println("Ejecutandose en: " + Thread.currentThread());
                }
            });
        }
```

/*Después de crear 100 hilos y ordenar que se ejecuten, invocamos al método shutdown() de la Thread Pool. Con esta llamada indicamos que, una vez se hayan ejecutado todos los hilos, se habrá terminado la función run.*/

```
        exec.shutdown();
```

/*El método awaitTermination detiene la ejecución del programa hasta que haya acabado todo el trabajo o hasta que se sobrepase el número de segundos de espera que se pasan como argumento*/

```
        try {
            boolean b = exec.awaitTermination(50, TimeUnit.SECONDS);

            System.out.println("Terminarón todos los hilos: " + b);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Ejemplo Ejecutor.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Main {
    public static void main(String args[]) {
        ExecutorService exec = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 100; i++) {
            exec.execute(new Runnable() {
                public void run() {
                    System.out.println("Ejecutandose en            : " + Thread.currentThread());
                }
            });
        }
        exec.shutdown();
        try {
            boolean b = exec.awaitTermination(50, TimeUnit.SECONDS);

            System.out.println("Terminaron todos los hilos: " + b);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Ejecutores definidos en las librerías Java

En *java.util.concurrent.executors hay cuatro métodos que* generan instancias concretas que implementan Executor y ofrecen políticas de ejecución diferentes:

newFixedTrhreadPool(int nThreads) => Genera un pool de threads con un número de elementos constantes. Si alguno de ellos muere es sustituido por otro.

newCachedThreadPool() => Genera un pool con un número base de threads. Este número se incrementa si la carga de trabajo crece.

newSingleThreadExecutor() => Crea un único thread que ejecuta secuencialmente las tareas que se le encomiendan. Si el thread muere es sustituido por otro.

newScheduledThreadPool(int nThread) => Crea un pool de threads con un número de elemento prefijado. Los threads ofrecen la interfaz *ScheduledExecutorService con la que se puede planificar la ejecución* periódica de los threads.

Ejemplo:newFixedThreadPool(int numThreads)

newFixedTrhreadPool(int nThreads) => Genera un pool de threads con un número de elementos constantes. Si alguno de ellos muere es sustituido por otro.

Ejemplo:

•Sistema inactivo
• llegan 10 tareas simultáneamente
• pasan 10 segundos
• llegan 5 tareas Simultáneamente
• pasan 5 minutos
• Llegan 20 tareas simultáneamente.
• Cada tarea toma entre 5 y 15 segundos en procesarse.

• Las primeras 10 tareas que llegan se ponen en la cola y los threads desocupados toman cada uno una tarea y la procesan.

• Cuando llegan las siguientes 5 tareas, se ponen en la cola y los primeros threads que se desocupen irán tomando las tareas de la cola para procesarlas.

•Cuando lleguen 20 tareas 5 minutos después, se ponen en la cola y los threads van a tomar las primeras 10 (dado que están todos desocupados).

•Cada thread cuando termine de procesar su tarea, tomará otra tarea de la cola.

Ejemplo:newCachedThreadPool()

newCachedThreadPool() => Genera un pool con un número base de threads. Este número se incrementa si la carga de trabajo crece.

Ejemplo:

- Sistema inactivo
- llegan 10 tareas Simultáneamente
- pasan 10 segundos
- llegan 5 tareas Simultáneamente
- pasan 5 minutos
- Llegan 20 tareas simultáneamente.
- Cada tarea toma entre 5 y 15 segundos en procesarse.

Cada una de las 10 tareas que llegan primero, se ejecutan cada una en un thread nuevo.
Cuando llegan las siguientes 5 tareas, se buscan primero los threads que ya hayan terminado de procesar su tarea; si hay threads libres, se ponen las tareas a procesarse dentro de dichos threads, y las tareas faltantes se procesan en threads nuevos:

Por ejemplo, si ya hay 3 threads libres, se procesan ahi 3 de las tareas nuevas, y se crean 2 threads nuevos para procesar las 2 tareas faltantes

Ejemplo:newCachedThreadPool()

newCachedThreadPool() => Genera un pool con un número base de threads. Este número se incrementa si la carga de trabajo crece.

Ejemplo:

- Sistema inactivo
- llegan 10 tareas Simultáneamente
- pasan 10 segundos
- llegan 5 tareas Simultáneamente
- pasan 5 minutos
- Llegan 20 tareas simultáneamente.
- Cada tarea toma entre 5 y 15 segundos en procesarse.

En este momento se tienen 12 threads funcionando, que al final habrán procesado 15 tareas.

Los threads que llevan mucho tiempo inactivos son terminados automáticamente por el pool, de manera que el número de threads para procesar tareas va disminuyendo cuando el sistema está inactivo e incluso se puede quedar vacío, como al principio.

Cuando lleguen nuevas tareas, se crearán los threads necesarios para procesarlas.
Es decir, cuando lleguen 20 tareas 5 minutos después, seguramente van a correr en 20 threads nuevos.

Ejemplo: newSingleThreadExecutor()

newSingleThreadExecutor() => Crea un único thread que ejecuta secuencialmente las tareas que se le encomiendan. Si el thread muere es sustituido por otro.

Ejemplo:

- Sistema inactivo
- llegan 10 tareas

Simultáneamente

- pasan 10 segundos
- llegan 5 tareas

Simultáneamente

- pasan 5 minutos
- Llegan 20 tareas

simultáneamente.

- Cada tarea toma entre 5 y 15 segundos en procesarse.

Crea un pool de un solo thread, con una cola en donde se ponen las tareas a procesar.

El thread toma una tarea de la cola, la procesa y toma la siguiente, en un ciclo.

Llegan 10 tareas, y se van a procesar de manera secuencial, las siguientes 5 se van a encolar y se procesarán cuando se terminen las primeras 10, y las ultimas 20 se procesarán una tras otra.

Ejemplo: newScheduledExecutor()

newScheduledThreadPool(int nThread) => Crea un pool de threads con un número de elemento prefijado. Los threads ofrecen la interfaz *ScheduledExecutorService con la que se puede planificar la ejecución* periódica de los threads.

Ejemplo:

- Sistema inactivo
- llegan 10 tareas Simultáneamente
- pasan 10 segundos
- llegan 5 tareas Simultáneamente
- pasan 5 minutos
- Llegan 20 tareas simultáneamente.
- Cada tarea toma entre 5 y 15 segundos en procesarse.

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 timer, pero con la diferencia de que puede tener varios threads que irán realizando las tareas programadas conforme se desocupen.

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

Tablet Epad HD 10.2 Android 32 Gb doble núcleo.

Procesador de doble núcleo
memoria RAM de 512 MB
cámara de 5 mpx
salida Hdmi
WiFi y 3G
Gps Incluye antena
Bocinas Estéreo
Batería de ion litio de larga duración
Incluye gastos d envío a toda la republica por estafeta entrega de 1 a 2 días hábiles

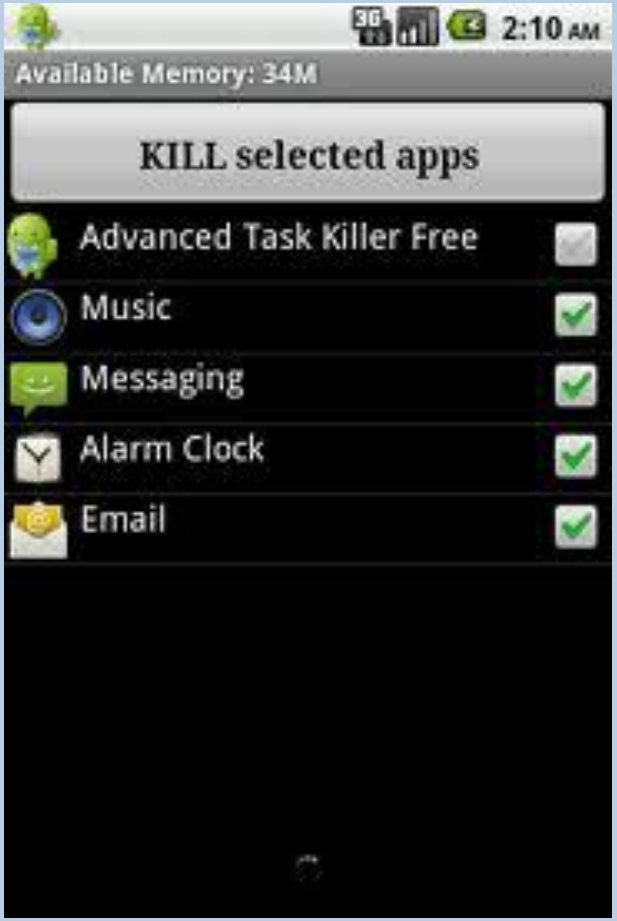
$ 2, 700

Para cada actividad, Android dispone de un hilo principal, de forma que cualquier código de la actividad se ejecuta en ese hilo.

El problema viene cuando la actividad trata de hacer acciones que tardan mucho tiempo en llevarse a cabo, lo cual si no se controla puede llevar a bloqueos de ese hilo principal, y por tanto, de la interfaz de usuario asociada a la actividad en cuestión.

Por tanto, todas las operaciones que sean potencialmente lentas (acceso a red, acceso a base de datos, acceso a archivo, etc.) deberemos ejecutarlas en segundo plano, utilizando la **concurrencia de hilos (threads).**

Sin embargo, en este caso, los hilos presentan el problema de que no pueden actualizar la interfaz de usuario desde segundo plano.



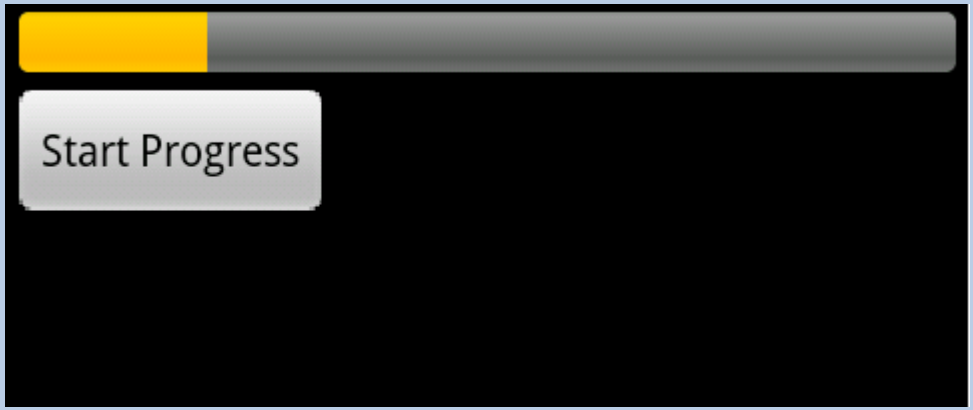
Para solventar esto, Android en sus orígenes proporcionó la clase *Handler*.

http://developer.android.com/reference/android/os/Handler.html

Un *handler* no es más que una clase que proporciona métodos para recibir mensajes, como *handleMessage()*, y objetos ***Runnable***, como *post()*. Supongamos un ejemplo como el siguiente, un botón que arranca una barra de progreso.

Android1.txt

Con esto, en cada ciclo del bucle, el *handler* actualiza la barra de progreso, y se simula su crecimiento de una forma bastante simple.



En las últimas versiones, Android presenta un mecanismo más cómodo: *AsyncTask*, que encapsula hilo y *Handler* en la misma clase.

Para ello, implementa el método *doInBackground()*, el cual define qué acción debe ser hecha en segundo plano, la cual de forma transparente se ejecuta en un hilo separado.

Una vez terminada esa acción, automáticamente se llama al método *onPostExecute()* para la actualización de la interfaz de usuario.

Supongamos ahora que quisiéramos leer una página web utilizando un objeto *AsyncTask*

Android2.txt

En definitiva, se pueden hacer cosas totalmente equivalentes usando estos dos mecanismos de concurrencia de Android.

Herramientas necesarias para preparar el entorno

1.- Instalar Jdk java, se trata de un conjunto de herramientas (programas y librerías) que permiten desarrollar programas en lenguaje Java (compilar, ejecutar, generar documentación, etc.).

http://www.oracle.com/technetwork/java/javase/downloads/index.html

2.- Descargar el SDK recomendado de Android, el SDK es un kit de desarrollo que contiene herramientas para desarrollar en Android, contiene herramientas como por ejemplo la maquina virtual Dalvik.

http://developer.android.com/sdk/index.html

3.- Configurando SDK, abrimos la aplicación que se a instalado en mis "Archivos de Programas->Android->Android-sdk-windows" y ejecutamos SDK Setup.exe.

4.- Una vez instalado el JDK, procedemos a descargar Eclipse. Este es un entorno de desarrollo integrado de código abierto multiplataforma. Para Eclipse 3.5 o más reciente, utilice la versión recomendada "Eclipse Classic ". Es solo descargar, descomprimir y utilizar.

http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/heliossr2

5.- Instalar componentes de Android para eclipse, en Eclipse Help->Install New Software.

6.- Falta solo un paso, que es configurar las preferencias del Android en el Eclipse, nos dirigimos a Window->Preferences.

http://www.htcmania.com/showthread.php?t=218891