

# Hilos (Threads) en Java

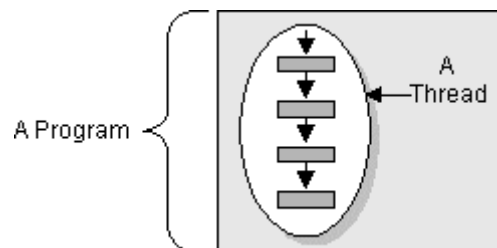
En la primera parte de este curso vimos procesos en sistemas operativos que surgen de Unix. En esos ejemplos cada proceso se caracterizaba por tener un único control de flujo, es decir la ejecución del programa seguía una secuencia única. Cuando creamos un nuevo proceso con fork, creamos una nueva secuencia que se ejecuta concurrentemente con el proceso padre, pero no comparten las zonas de datos y la comunicación entre ellos es muy limitada. Por ello aparecen las pipes y los otros mecanismos de comunicación entre procesos.

Los hilos son otra forma de crear la posibilidad de concurrencia de actividades; sin embargo, la gran diferencia es que los hilos comparten el código y el acceso a algunos datos en forma similar a como un objeto tiene acceso a otros objetos. En Java un hilo es un objeto con capacidad de correr en forma concurrente el método run(). En cierta manera es como tener dos "program counters" para un mismo código. Una diferencia con los procesos es que carece de sentido y no es posible en este enfoque hacer mutar un proceso con algo similar a exec().

## Definición Thread

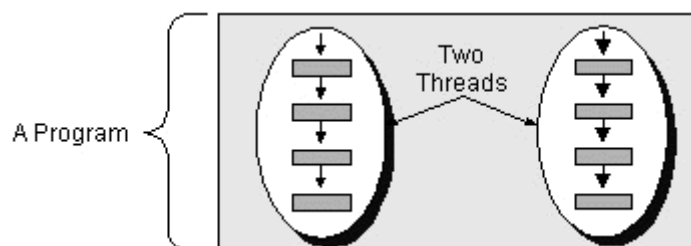
Una thread es un único flujo de control dentro de un programa. Algunas veces es llamado contexto de ejecución porque cada thread debe tener sus propios recursos, como el program counter y el stack de ejecución, como el contexto de ejecución. Sin embargo, toda thread en un programa aún comparte muchos recursos, tales como espacio de memoria y archivos abiertos. Threads también son llamadas procesos livianos (lightweight process).

NOTA: Es mucho más simple crear y destruir una thread que un proceso.



## Thread Paralelas y Concurrentes

Cuando dos threads corren en paralelo, ambas están siendo corridas al mismo tiempo en diferentes CPUs. Dos thread concurrentes están en progreso, o intentando de obtener tiempo de ejecución de la CPU al mismo tiempo, pero no necesariamente están corriendo en forma simultánea en dos CPUs diferentes.



### **Ejemplo: Programa de thread única**

En el programa [WithoutThread.java](#), el método run() en la clase NoThreadPseudoIO es creado para simular una operación de IO de 10 segundos. El programa primero simulará una operación de IO.

### **Métodos del Programa de Thread única**

El método showElapsedTime() imprime el tiempo en segundos desde que el programa partió, junto con el mensaje del usuario. El método currentTimeMillis() de la clase System del paquete java.lang retorna un entero que corresponde a la diferencia de tiempo en milisegundos desde la hora 00:00:00 GMT on January 1, 1970. a la actual.

### **Ejemplo de programa Multithreaded (multihilo)**

El programa multithreaded [WithThread.java](#) declara la clase para simular IO como una subclase de la clase Thread. Después que la thread es creada, el programa multihilos (multithreaded) usa el método start() de la clase Thread para hacer partir la operación de IO. El método start() a su vez llama a run() de la subclase.

## **Creación y ejecución de Threads**

**Hay dos formas de hacer una tarea correr concurrentemente con otra: crear una nueva clase como subclase de la clase Thread o declarar una clase e implementar la interfaz Runnable.**

### **Uso de Subclase**

Cuando se crea una subclase de Thread, la subclase debería definir su propio método run() para sobre montar el método run() de la clase Thread. La **tarea concurrente es desarrollada en este método run()**.

### **Ejecución del método run()**

Una instancia de la subclase es creada con new, luego llamamos al método start() de la thread para hacer que la máquina virtual Java ejecute el método run(). Ojo para iniciar la concurrencia invocamos a start(), así invocamos a run() en forma indirecta. Si invocamos a run() directamente, se comportará como el llamado a cualquier método llamado dentro de un mismo hilo (sin crear uno independiente).

### **Implementación de la Interfaz Runnable**

La interfaz Runnable requiere que sólo un método sea implementado, el método run(). Primero creamos una instancia de esta clase con new, luego creamos una instancia de Thread con otra sentencia new y usamos el objeto recién creado en el constructor. Finalmente, llamamos el método start() de la instancia de Thread para iniciar la tarea definida en el método run().

### **[RunnableThread.java](#)**

Una instancia de una clase que defina el método run() - ya sea como subclase de Thread o implementando la interfaz Runnable - debe ser pasada como argumento

en la creación de una instancia de Thread. Cuando el método start() de esta instancia es llamado, Java run time sabe qué método run() ejecutar.

## Control de la Ejecución de una Thread

Varios métodos de la clase java.lang.Thread controlan la ejecución de una thread.

### Métodos de uso común:

void start(): usado para iniciar el cuerpo de la thread definido por el método run().

void sleep(): pone a dormir una thread por un tiempo mínimo especificado.

void join(): usado para esperar por el término de la thread sobre la cual el método es invocado, por ejemplo por término de método run().

void yield(): Mueve a la thread desde el estado de corriendo al final de la cola de procesos en espera por la CPU.

### [MethodTest.java](#)

Java 2 dejó obsoleto (deprecated) varios de estos métodos definidos en versiones previas (Java 1.0 y Java 1.1.) para prevenir inconsistencia de datos y deadlock. Se recomienda evitar el uso de estos métodos. Ellos son:

void stop() el cual detiene la ejecución de la thread no importando consideración alguna.

void suspend() el cual para temporalmente la ejecución de una thread.

void resume() reactiva una thread suspendida.

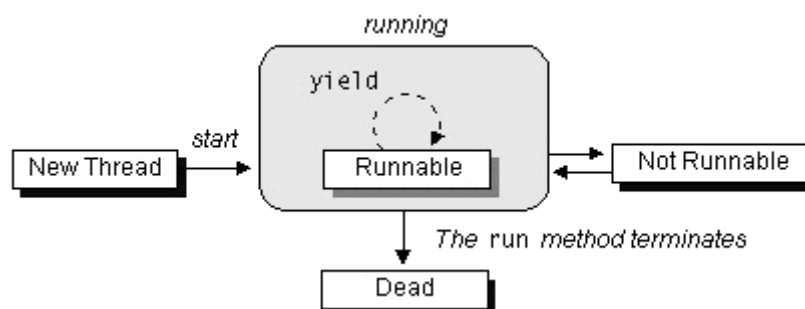
Otros ejemplos:

Múltiples hilos cada uno con [contador down](#).

Múltiples hilos controlando [bloques de colores](#) que cambian

## Ciclo de Vida de una Thread

Cada hilo, después de su creación y antes de su destrucción, estará en uno de cuatro estados: recién creada, "corrible", bloqueada, o muerta.



**Recién creada** (New thread): entra aquí inmediatamente después de su creación. Es decir luego del llamado a new. En este estado los datos locales son ubicados e iniciados. Luego de la invocación a start(), el hilo pasa al estado "corrible".

**Corrible** (Runnable): Aquí el contexto de ejecución existe y el hilo puede ocupar la

CPU en cualquier momento. Este estado puede subdividirse en dos: Corriendo y encolado. La transición entre estos dos estados es manejado por el itinerador de la máquina virtual.

Nota: Un hilo que invoca al método `yield()` voluntariamente se mueve a sí misma al estado encolado desde el estado corriendo.

**Bloqueada** (not Runnable): Se ingresa cuando: se invoca `suspend()`, el hilo invoca el método `wait()` de algún objeto, el hilo invoca `sleep()`, el hilo espera por alguna operación de I/O, o el hilo invoca `join()` de otro hilo para esperar por su término. El hilo vuelve al estado Corriendo cuando el evento por que espera ocurre.

**Muerta** (Dead): Se llega a este estado cuando el hilo termina su ejecución (concluye el método `run()`) o es detenida por otro hilo llamando al su método `stop()`. Esta última acción no es recomendada.

El siguiente código puede ser usado para evitar el llamado a `stop()` y así evitar estados de inconsistencia. Invocando a `safeStop()` se consigue terminar el hilo la próxima vez que la variable `done` es chequeada.

```
boolean done = false;
public void run() {
    while(!done) {
        ....
    }
}
public safeStop() {
    done = true;
}
```

El método `isAlive()` invocado sobre un hilo, permite saber si aún puede correr.

## Sincronización de hilos

Todos los hilos de un programa comparten el espacio de memoria, haciendo posible que dos hilos accedan la misma variable o corran el mismo método de un objeto al "mismo tiempo". Se crea así la necesidad de disponer de un mecanismo para bloquear el acceso de un hilo a un dato crítico si el dato está siendo usado por otro hilo.

Por ejemplo, si en un sistema un método permite hacer un depósito y luego dos hilos lo invocan, es posible que al final sólo un depósito quede registrado al ejecutar las instrucciones en forma traslapadas.

[Deposit.java](#)

## Modelo de Monitores

Java utiliza la idea de monitores para sincronizar el acceso a datos. Un monitor es un lugar bajo guardia donde todos los recursos tienen el mismo candado. Sólo una llave abre todos los candados dentro de un monitor, y un hilo debe obtener la llave

para entrar al monitor y acceder a esos recursos protegidos. Cada objeto en Java posee un candado y una llave para manejo de zonas críticas. También existe un candado y llave por cada clase, éste se usa para los métodos estáticos.

Si varios hilos desean entrar al monitor simultáneamente, sólo uno obtiene la llave. Los otros son dejados fuera (bloqueados) hasta que el hilo con la llave termine de usar el recurso exclusivo y devuelva la llave a la Máquina Virtual Java.

Puede ocurrir deadlock si dos hilos están esperando por el recurso cuya llave la tiene el otro.

En un momento un hilo puede tener las llaves de varios monitores.

En Java los recursos protegidos por un monitor son fragmento de programa en forma de métodos o bloques de sentencias encerradas con paréntesis {}.

Palabra reservada **synchronized**: es usada para indicar que el siguiente método o bloque de sentencias es sincronizada por un monitor (aquel del objeto que tiene el método). Cuando queremos sincronizar un bloque, un objeto encerrado por ( ) sigue a la palabra synchronized, así la máquina virtual sabe qué monitor chequear.

Por ejemplo, el método deposit() puede ser sincronizado para correr sólo un hilo a la vez.

La salida del programa alternativo es casi igual, excepto que el primer mensaje del segundo hilo es traslapado con el de la primera, porque el primer println() no está dentro del bloque synchronized.

[Deposit2.java](#)

Llave del monitor

Existe un llave única por cada objeto que contiene un método sincronizado (synchronized) asociado a su instancia (método no estático), o que es referenciado en un bloque synchronized (es usado como argumento de esta sentencia). Por ello cada objeto y cada clase puede tener un monitor si hay cualquier método sincronizado o bloque de sentencias asociado con ellos. Más aún, la llave de un monitor de una clase es diferente de las llaves de los monitores de las instancias (esto porque el método puede ser llamado antes de existir ninguna instancia).

## Multihilos más avanzada

Para lograr buena sincronización entre las tareas en ocasiones debemos hacer uso de otros mecanismos de sincronización. Por ello veremos el ejemplo de un productor de números enteros y un consumidor de ellos.

**Productor consumidor**

[NoWaitPanc.java](#)

Este programa no garantiza que todos los enteros generados serán leído y que cada entero será leído sólo una vez. La sentencia Synchronized garantiza que sólo una operación sea hecha a la vez, pero no garantiza que ambos hilos lo hagan el forma

alternada.

Para lograr alternancia, usamos los llamados wait y notify.

Método **wait()** hará que el hilo que invoca se bloquee hasta que ocurra un timeout u otro hilo llame el método notify() o notifyAll() sobre el mismo objeto (lo primero que ocurra).

Cuando un hilo llama a wait(), la llave que éste tiene es liberada, así otro proceso que esperaba por ingresar al monitor puede hacerlo. **notify()** sólo despierta o desbloquea un hilo, si lo hay esperando. notifyAll() despierta a todos los que estén esperando. Luego que un hilo despierta y como parte del wait() tratará de reingresar al monitor pidiendo la llave nuevamente, podría tener que esperar a que otro hilo la libere.

Los llamados wait(), notify() y notifyAll(), sólo pueden ser llamados dentro de un método o bloque sincronizado.

[PandC.java](#)