# THREADS

Hilo, tarea: es la clase base de **Java** para definir hilos de ejecución concurrentes dentro de un mismo programa.
En **Java**, como lenguaje O.O., el concepto de concurrencia está asociado a los objetos: Son los objetos los que actúan concurrentemente con otros.

Los S.O permiten la multitarea, es decir, la realización simultánea de dos o más actividades. En realidad, una sola CPU no puede realizar dos actividades a la vez. Sin embargo los S.O. son capaces de ejecutar varios programas simultáneamente aunque sólo se disponga de una CPU, repartiendo el tiempo entre dos (o más) procesos, o bien utilizan los "tiempos muertos" de una actividad (por ejemplo, las operaciones de lectura de datos desde el teclado) para trabajar en la otra tarea.
En computadoras con dos o más procesadores, o en "clusters" de computadoras, la multitarea es real, ya que cada procesador puede ejecutar un proceso diferente.

**¿Qué es un proceso?**
Un proceso es un programa ejecutándose de forma independiente y con un espacio propio de memoria. Un sistema operativo multitarea es capaz de ejecutar más de un proceso simultáneamente.

**¿Qué es un thread?**
Un Thread (hilo) es un flujo secuencial simple dentro de un proceso. Un único proceso puede tener varios hilos ejecutándose.

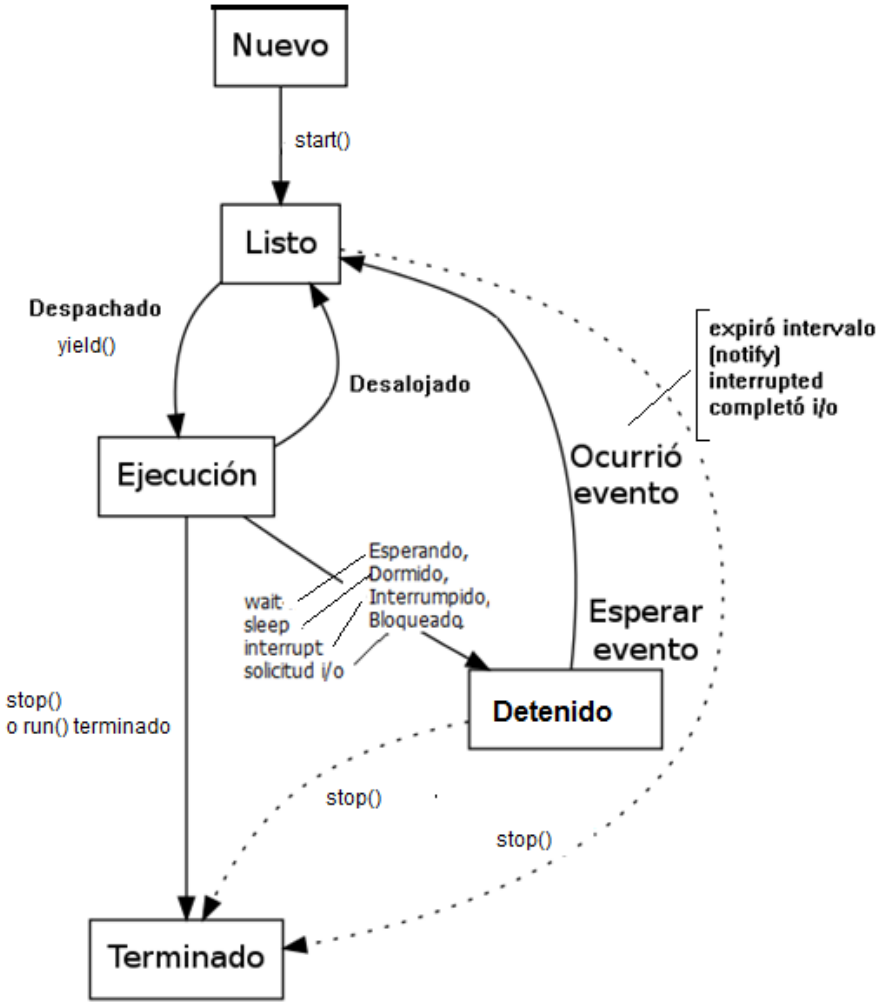
## Las características básicas de los tipos de multitarea son:

**En base a procesos:**
1. Permite a la computadora ejecutar más de un proceso concurrentemente.
2. La unidad de código más pequeña es el proceso.
3. Los procesos son tareas pesadas que necesitan su propio espacio de direccionamiento.
4. La comunicación entre procesos es costosa y limitada.

**En base a hilos:**
1. Un programa simple puede realizar más de una tarea simultáneamente.
2. La unidad de código más pequeña es el hilo.
3. Los hilos consumen menos recursos del S.O. que los procesos.
4. Comparten el mismo espacio de direcciones y el mismo proceso.
5. La comunicación entre hilos es ágil y el cambio de contexto de un hilo al siguiente es menos costoso.

Java hace uso de entornos multitarea basados en procesos, pero implementa esquemas de multitarea basada en hilos, bajo el control de Java. La VM de Java utiliza un esquema de prioridades para determinar cómo debe tratar cada hilo con respecto a los demás. La prioridad de un hilo se utiliza para decidir cuándo se va a ejecutar otro hilo (cambio de contexto).

## Estados de un thread




## Las causas que hacen que un hilo ceda el control son:

1. Un hilo puede ceder voluntariamente el control por abandonó explícito, al dormirse o bloquearse en espera de una operación de E/S pendiente. Entonces se selecciona entre los hilos restantes, el que tiene mayor prioridad, para darle el control.
2. Un hilo que no libera la CPU puede ser desalojado por otro de mayor prioridad. En el caso de hilos de igual prioridad que compiten por la CPU, el S.O. decide a quien asignar CPU en base a un algoritmo de planificación de tareas.

Sincronización

Java implementa la sincronización entre procesos, por medio de un monitor. Así, cada objeto tiene su propio monitor, en el que se entra cuando se llama a uno de los métodos sincronizados del objeto. Una vez que un hilo está dentro de un método sincronizado, ningún otro hilo puede llamar a otro método sincronizado del mismo objeto.

Intercambio de mensajes

Java tiene métodos predefinidos que permiten que un hilo entre en un método sincronizado de un objeto, y espere ahí hasta que otro hilo le notifique explícitamente que debe salir de él.

## Prioridades

Para crear un nuevo hilo en Java, se debe crear una clase que herede de la clase Thread o que implemente la interfaz Runnable. La clase Thread provee un conjunto de métodos para manipular los hilos. Algunos de ellos son:
public final String getName():Obtiene el nombre de un hilo.

public final int getPriority():Obtiene la prioridad de un hilo.

Un tema fundamental dentro de la programación multihilo es la planificación de los hilos. Este concepto se refiere a la política a seguir de que hilo toma el control del procesador y cuando. Obviamente en el caso de que un hilo este bloqueado esperando una operación de I/O este hilo debería dejar el control del procesador y que este control lo tomara otro hilo que si pudiera hacer uso del tiempo de CPU. ¿Pero qué pasa si hay más de un hilo esperando? ¿A cuál de ellos le otorgamos el control del procesador?

Para determinar que hilo debe ejecutarse primero, cada hilo posee su propia prioridad: un hilo de prioridad más alta que se encuentre en el estado LISTO entrara antes en el estado EN EJECUCION que otro de menor prioridad.

Para establecer la prioridad de un thread se utiliza el método setPriority() de la siguiente manera:

```
h1.setPriority(10);
h1.setPriority(1);
```

También existen constantes definidas para la asignación de prioridades estas son:

```
MIN_PRIORITY = 1
NORM_PRIORITY = 5
MAX_PRIORITY = 10
```

Las cuales se pueden utilizar de la siguiente manera:

```
h1.setPriority(Thread. MAX_PRIORITY); //Le concede la mayor prioridad
h1.setPriority(Thread. MIN_PRIORITY); //Le concede la menor prioridad
```

public final native boolean isAlive():Comprueba si un hilo se está ejecutando.

El interfaz de programación de la clase Thread incluye el método isAlive(), que devuelve true si el hilo ha sido arrancado (con start()) y no ha sido detenido (con stop()). Por ello, si el método isAlive()devuelve false, sabemos que estamos ante un Nuevo Thread o ante un thread Muerto. Si devuelve true, se sabe que el hilo se encuentra en estado Ejecutable o Parado.

public final void join():Espera hasta que finalice el hilo sobre el que se llama.

Por que se llama así? Crear un proceso es como una bifurcación, se abren 2 caminos, que uno espere a otro es lo contrario, una unificación.
El metodo join() que llamamos al final hace que el programa principal espere hasta que este Thread este “muerto”(finalize su ejecucion).

## Métodos

public final void resume():Reanuda la ejecución de un hilo suspendido.
public void run():Método principal para ejecutar un hilo.
public void setPriority(int priority): establece la prioridad de ejecución.
public static native void sleep(long millis): Suspende un hilo durante un período de tiempo
public native synchronized void start():Comienza la ejecución un hilo llamando a run(). Este método indica al intérprete de Java que cree un contexto del hilo del sistema y comience a ejecutarlo. A continuación, el método run() de este hilo será invocado en el nuevo contexto del hilo. Hay que tener precaución de no llamar al método start() más de una vez sobre un hilo determinado.
public final void stop():Detiene la ejecución de un hilo.
public final void suspend(): Suspende la ejecución un hilo.
public static native void yield(): tiene la función de hacer que un hilo que se está ejecutando, regrese al estado “listo para ejecutar” (cese temporalmente la ejecución), para permitir que otros hilos de la misma prioridad puedan ejecutarse.

## Creación de un hilo

Cuando se ejecuta un programa Java se crea un hilo, llamado hilo principal, desde donde se crean el resto de hilos del programa.
El método run() es la función inicial “de arranque”, de un nuevo hilo de ejecución concurrente dentro de un programa. Un hilo termina cuando finaliza el método run().
Para que un hilo comience su ejecución se debe activar el método start().

EJEMPLO

```
class MiThread extends Thread {
MiThread() {
super("Padre");
System.out.println("Hijo: " + this);
}
public void run() {
try {
for(int i = 0; i < 5; i++) {
System.out.println("Hijo: " + i);
Thread.sleep(500);
}
}
catch (InterruptedException e) {
System.out.println("Hijo interrumpido");
}
System.out.println("Termina hijo");
}
}
```

```
class PruebaThread {
public static void main(String args[]) {
MiThread obj = new MiThread();
obj.start();
try {
for(int i = 0; i < 5; i++) {
System.out.println("Programa: " + i);
Thread.sleep(1000);
}
}
catch (InterruptedException e) {
System.out.println("Programa interrumpido");
}
System.out.println("Termina programa");
}
}
```

## Sincronización de hilos

Muchas veces los hilos deberán trabajar de forma coordinada, por lo que es necesario un mecanismo de sincronización entre ellos.

Un primer mecanismo de sincronización es la variable cerrojo incluida en todo objeto Object, que permitirá evitar que más de un hilo entre en la sección crítica para un objeto determinado. Los métodos declarados como synchronized utilizan el cerrojo del objeto al que pertenecen evitando que más de un hilo entre en ellos al mismo tiempo.

```
1    public synchronized void metodo_seccion_critica()
2    {
3        // Código sección crítica
4    }
5
```

Todos los métodos synchronized de un mismo objeto (no clase, sino objeto de esa clase), comparten el mismo cerrojo, y es distinto al cerrojo de otros objetos (de la misma clase, o de otras).

También podemos utilizar cualquier otro objeto para la sincronización dentro de nuestro método de la siguiente forma:

```
1    synchronized(objeto_con_cerrojo)
2    {
3      // Código sección crítica
4    }
5
```

De esta forma sincronizaríamos el código que escribiésemos dentro, con el código synchronized del objeto objeto_con_cerrojo.

El bloque synchronized lleva entre paréntesis la referencia a un objeto. Cada vez que un thread intenta acceder a un bloque sincronizado le pregunta a ese objeto si no hay algún otro thread que ya tenga el lock para ese objecto. En otras palabras, le pregunta si no hay otro thread ejecutando algun bloque sincronizado con ese objeto (y recalco que es ese objeto porque en eso radica la clave para entender el funcionamiento)
Si el lock está tomado por otro thread, entonces el thread actual es suspendido y puesto en espera hasta que el lock se libere. Si el lock está libre, entonces el thread actual toma el lock del objeto y entra a ejecutar el bloque. Al tomar el lock, cuando venga el proximo thread a intentar ejecutar un bloque sincronizado con ese objeto, será puesto en espera.
¿Cuándo se libera el lock? Se libera cuando el thread que lo tiene tomado sale del bloque por cualquier razón: termina la ejecución del bloque normalmente, ejecuta un return o lanza una excepción.

Es importante notar una vez más que el lock es sobre un objeto en particular. Si hay dos bloques synchronized que hacen referencia a distintos objetos (por más que ambos utilicen el mismo nombre de variable), la ejecución de estos bloques no será mutuamente excluyente

Además podemos hacer que un hilo quede bloqueado a la espera de que otro hilo lo desbloquee cuando suceda un determinado evento. Para bloquear un hilo usaremos la función wait(), para lo cual el hilo que llama a esta función debe estar en posesión del monitor, cosa que ocurre dentro de un método synchronized, por lo que sólo podremos bloquear a un proceso dentro de estos métodos.

Para desbloquear a los hilos que haya bloqueados se utilizará notifyAll(), o bien notify() para desbloquear sólo uno de ellos aleatoriamente. Para invocar estos métodos ocurrirá lo mismo, el hilo deberá estar en posesión del monitor.

Cuando un hilo queda bloqueado liberará el cerrojo para que otro hilo pueda entrar en la sección crítica del objeto y desbloquearlo.

Por último, puede ser necesario esperar a que un determinado hilo haya finalizado su tarea para continuar. Esto lo podremos hacer llamando al método join() de dicho hilo, que nos bloqueará hasta que el hilo haya finalizado.

## Sincronización reentrante

Se dice que en Java la sincronización es reentrante porque una sección crítica sincronizada puede contener dentro otra sección sincronizada sobre el mismo cerrojo y eso no causa un bloqueo. Por ejemplo el siguiente código funciona sin bloquearse:

```
1  class Reentrant {
2      public synchronized void a() {
3          b();
4          System.out.println(" estoy en a() ");
5      }
6      public synchronized void b() {
7          System.out.println(" estoy en b() ");
8      }
   }
```

9
10

La salida sería estoy en b() \n estoy en a() .