1.- Introducción

Multithreading es una característica de **Java** que permite la ejecución concurrente de dos o más partes de un programa para una utilización máxima de la CPU. Cada parte de dicho programa se llama hilo. Entonces, los hilos son **procesos livianos dentro de un proceso**.

A pesar de que Java contiene muchas características innovadoras, una de las más interesante es su soporte integrado para **programación multihilo**. Un programa multihilo contiene dos o más partes que se pueden ejecutar simultáneamente.

Cada parte de dicho programa se denomina **hilo** (thread) y cada hilo define una ruta de ejecución independiente. Por lo tanto, multihilo (Multithreaded) es una forma especializada de multitarea.

Fundamentos de Multihilo

Hay dos tipos distintos de multitareas: **basado en procesos** y **basado en hilos**. Es importante entender la diferencia entre los dos.

- Un **proceso** es, en esencia, un **programa que se está ejecutando**. Por lo tanto, la multitarea basada en procesos es la característica que le permite a su computadora ejecutar dos o más programas al mismo tiempo. Por ejemplo, es una **multitarea basada en procesos** que le permite ejecutar el compilador de Java al mismo tiempo que utiliza un editor de texto o navega por Internet. En la multitarea basada en procesos, **un programa es la unidad de código más pequeña que puede enviar el planificador del sistema operativo o bien ser gestionada con un sistema PVM.**
- En un entorno **multitarea basado en hilos**, **el hilo es la unidad más pequeña de código distribuible**. Esto significa que un solo programa puede realizar dos o más tareas a la vez. Por ejemplo, un editor de texto puede formatear texto al mismo tiempo que está imprimiendo, siempre que estas dos acciones se realicen mediante dos hilos separados. Aunque los programas Java utilizan entornos multitarea basados en procesos , **la multitarea basada en procesos no está bajo el control de Java**. La **multitarea multihilo** sí lo está bajo el control en este caso de Java.

Ventajas de Multihilo

Una ventaja principal del multihilo es que le **permite escribir programas muy eficientes** porque **le permite utilizar el tiempo de inactividad** que está presente en la mayoría de los programas. La mayoría de los dispositivos de E/S, ya sean puertos de red, unidades de disco o el teclado, son mucho más lentos que la CPU. Por lo tanto, un programa a menudo pasará la mayor parte de su tiempo de ejecución esperando para enviar o recibir información hacia o desde un dispositivo.

Al usar multihilo, el programa puede ejecutar otra tarea durante este tiempo de inactividad. Por ejemplo, mientras una parte del programa está enviando un archivo a través de Internet, otra parte puede leer la entrada del teclado, y otra puede almacenar el siguiente bloque de datos para enviar.

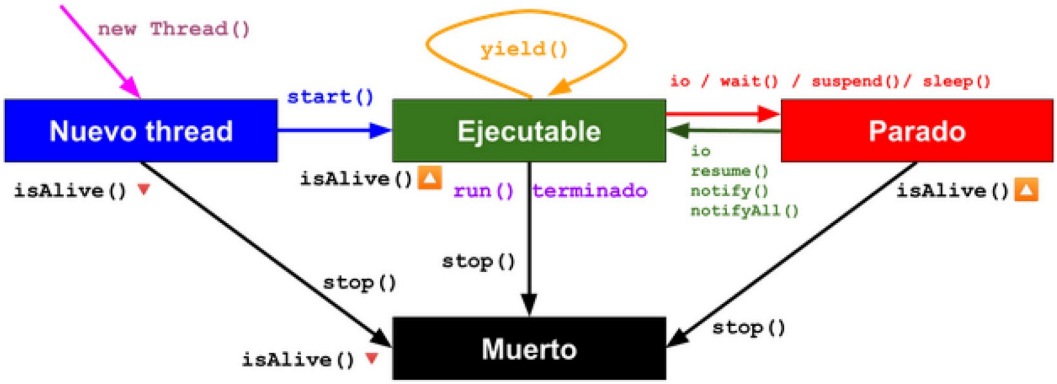
Como los sistemas **multiprocesador** y **multinúcleo** se han convertido en algo común. Es importante comprender que las características multihilo de Java funcionan en ambos tipos de sistemas. En un sistema de núcleo único, los hilos de ejecución simultáneas comparten la CPU, y cada hilo recibe un trozo de tiempo de CPU.

Por lo tanto, en un sistema de núcleo único, dos o más hilos no se ejecutan al mismo tiempo, pero se utiliza el tiempo de CPU inactivo. Sin embargo, en sistemas multiprocesador/multinúcleo, es posible que dos o más hilos se ejecuten de manera simultánea. En muchos casos, esto puede **mejorar aún más la eficiencia del programa** y aumentar la velocidad de ciertas operaciones.

2.- Estado de un hilo

Un hilo puede estar en uno de varios estados, en resumen:

- Puede estar ejecutándose.
- Puede estar listo para ejecutarse tan pronto como tenga tiempo de CPU.
- Se puede suspender un hilo en ejecución, que es un alto temporal a su ejecución.
- Puede luego reanudarse.
- Un hilo se puede bloquear cuando se espera un recurso.
- Un hilo puede terminar, en cuyo caso su ejecución finaliza y no puede reanudarse.



2.1.-Estado Nuevo Hilo

La siguiente sentencia crea un nuevo hilo de ejecución pero no lo arranca , lo deja en el estado de Nuevo Hilo

```
Thread Mihilo = new MiClaseHilo();
Thread Mihilo = new Thread ( new UnaClaseHilo , “hiloA”);
```

Cuando un hilo está en este estado, es simplemente un objeto Thread vacío. El sistema no ha destinado ningún recurso para él. Desde este estado solamente puede arrancarse llamando al método ***start(),*** o detenerse definitivamente, llamando al método ***stop();*** la llamada a cualquier otro método carece de sentido y lo único que provocará será la generación de una excepción de tipo ***IllegalThreadStateException***.

2.2.- Estado Ejecutable

Observamos las siguientes líneas de código:

```
Thread Mihilo=new MiClaseHilo();
Mihilo.start();
```

La llamada al método ***start()*** creará los recursos del sistema necesarios para que el hilo puede ejecutarse, lo incorpora a la lista de procesos disponibles para ejecución del sistema y llama al método run() del hilo de ejecución. En este momento se encuentra en el estado Ejecutable del diagrama. ***Y este estado es Ejecutable y no En Ejecución***, porque cuando el hilo está aquí no esta corriendo. Muchos ordenadores tienen solamente un procesador lo que hace imposible que todos los hilos estén corriendo al mismo tiempo. Java implementa un tipo de ***scheduling*** o ***lista de procesos***, que permite que el procesador sea compartido entre todos los procesos o hilos que se encuentran en la lista. Sin embargo, para el propósito que aquí se persigue, y en la mayoría de los casos, se puede considerar que este estado es realmente un estado ***En Ejecución***, porque la impresión que produce ante el usuario es que todos los procesos se ejecutan al mismo tiempo.

Cuando el hilo se encuentra en este estado, todas las instrucciones de código que se encuentren dentro del bloque declarado para el método run(), se ejecutarán secuencialmente.

2.3.- Estado Parado

El hilo de ejecución entra en estado **Parado** cuando:

- Alguien llama al método ***suspend().***
- Cuando se llama al método ***sleep().***
- Cuando el hilo está bloqueado en un proceso de ***entrada/salida.***
- Cuando el hilo utiliza su método ***wait()*** para esperar a que se cumpla una determinada condición.

Cuando ocurra cualquiera de las cuatro cosas anteriores, el hilo estará **Parado**.

Por ejemplo, en el trozo de código siguiente:

```
Thread Mihilo = new MiClaseHilo();
Mihilo.start();
try {
  Mihilo.sleep( 10000 );
} catch( InterruptedException e ) {
  ... ;
}
```

Esta línea de código "Mihilo.sleep(10000);" hace que el hilo se **duerma** durante **10 segundos**. Durante ese tiempo, incluso aunque el procesador estuviese totalmente libre, Mihilo no correría.

Después de esos 10 segundos. Mihilo volvería a estar en estado **Ejecutable** y ahora sí que el procesador podría hacerle caso cuando se encuentre disponible.

Para cada una de los cuatro modos de entrada en estado **Parado**, hay una forma **específica** de volver a estado **Ejecutable**. Cada forma de recuperar ese estado es exclusiva; por ejemplo, si el hilo ha sido puesto a dormir, una vez transcurridos los milisegundos que se especifiquen, él solo se despierta y vuelve a estar en estado Ejecutable. Llamar al método **resume()** mientras esté el hilo durmiendo no serviría para nada.

Los **métodos de recuperación** del estado **Ejecutable**, en función de la forma de llegar al estado **Parado** del hilo, son los siguientes:

- Si un hilo está dormido, pasado el lapso de tiempo
- Si un hilo de ejecución está suspendido, después de una llamada a su método **resume()**
- Si un hilo está bloqueado en una entrada/salida, una vez que el comando de entrada/salida concluya su ejecución
- Si un hilo está esperando por una condición, cada vez que la variable que controla esa condición varíe debe llamarse al método **notify()** o **notifyAll()**

2.4.- Estado Muerto

Un hilo de ejecución se puede morir de dos formas:

- por causas naturales
- o porque lo maten (constop()).

Un hilo muere normalmente cuando concluye de forma habitual su método ***run()***. Por ejemplo, en el siguiente trozo de código, el bucle while es un bucle finito -realiza la iteración 20 veces y termina-:

```
public void run() {
  int i=0;
  while( i < 20 ) {
    i++;
    System.out.println( "i = "+i );
  }
}
```

Un hilo que contenga a este método **run()**, morirá naturalmente después de que se complete el bucle y ***run()*** concluya.

También se puede matar en cualquier momento un hilo, invocando a su método ***stop()***. En el trozo de código siguiente:

```
Thread Mihilo = new MiClaseHilo();
Mihilo.start();
try {
  Mihilo.sleep( 10000 );
} catch( InterruptedException e ) {
  ...;
}
Mihilo.stop();
```

Se crea y arranca el hilo Mihilo, se duerme durante 10 segundos y en el momento de despertarse, la llamada a su método ***stop(),*** lo mata.

El método ***stop()*** envía un objeto ***ThreadDeath*** al hilo de ejecución que quiere detener. Así, cuando un hilo es parado de este modo, muere asíncronamente. El hilo morirá en el momento en que reciba ese objeto ***ThreadDeath***.

Los ***applets*** utilizarán el método ***stop()*** para matar a todos sus hilos cuando el navegador con soporte Java en el que se están ejecutando le indica al ***applet*** que se detengan, por ejemplo, cuando se **minimiza** la ventana del navegador o cuando se **cambia de página**.

2.5.- El método isAlive()

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. N**o se puede diferenciar entre Nuevo Thread y Muerto, ni entre un hilo Ejecutable o Parado.**

Junto con la multitarea basada en hilos viene la necesidad de un tipo especial de característica llamada sincronización (synchronization), que permite coordinar la ejecución de hilos de determinadas maneras bien definidas. Java tiene un subsistema completo dedicado a la sincronización.

2.6.- El método yield()

yield() es un método de buena educación. Si tienes código que sabes que se consume mucha CPU puedes meter una llamada ocasional a yield() para que los demás procesos (dentro y fuera de la JVM (maquina virtual java)) tengan oportunidad de hacer algo. Java en principio junto con el sistema operativo será el encargado de repartir la CPU, pero podemos forzar a que el programa solicite la acción del manejador.

3.- Clases para la creación de hilos: Thread

La clase Thread y la interfaz Runnable

El sistema multihilo de Java se basa en la clase **Thread** y su interfaz complementaria, **Runnable**. Ambos están empaquetados en **java.lang**. El hilo encapsula un hilo de ejecución. Para crear un nuevo hilo, su programa extenderá **Thread** o implementará la interfaz **Runnable**.

Thread

3.1.- Crear un hilo

Lo más sencillo para crear hilos es extender la clase Thread, es decir crear una subclase. Esta subclase debe sobreescribir el método ***run()***, ya que aquí pondremos lo que queremos que haga el hilo. Esta clase también tiene dos métodos que están en desuso start() y stop()

```
Public class MiHilo extends Thread
{
  public void run()
  {
    // Aquí el código pesado que tarda mucho
  }
};
...
MiHilo elHilo = new MiHilo();
elHilo.start();
System.out.println("Yo sigo a lo mio");
```

Se ha creado una clase **MiHilo** que hereda de **Thread** y con un método **run()**. En el método **run()** pondremos el código que queremos que se ejecute en un **hilo** separado. Luego instanciamos el **hilo** con un **new MiHilo()** y lo arrancamos con **elHilo.start()**. El **System.out** que hay detrás se ejecutará inmediatamente después del **start()**, haya terminado o no el código del hilo.

3.2.- Detener un hilo

Suele ser una costumbre bastante habitual que dentro del método **run()** haya un bucle infinito, de forma que el método **run()** no termina nunca. Por ejemplo, supongamos un chat. Cuando estás chateando, el programa que tienes entre tus manos está haciendo dos cosas simultáneamente. Por un lado, lee el teclado para enviar al servidor del chat todo lo que tú escribas. Por otro lado, está leyendo lo que llega del servidor del chat para escribirlo en tu pantalla. Una forma de hacer esto es "por turnos".

```
while (true)
{
  leeTeclado();
  enviaLoLeidoAlServidor();
  leeDelServidor();
  muestraEnPantallaLoLeidoDelServidor();
}
```

Esta, desde luego, es una opción, pero sería bastante "mala", tendríamos que hablar por turnos. Yo escribo algo, se le envía al servidor, el servidor me envía algo, se pinta en pantalla y me toca a mí otra vez. Si no escribo nada, tampoco recibo nada.

Lo normal es hacer dos hilos, ambos en un bucle infinito, leyendo (del teclado o del servidor) y escribiendo (al servidor o a la pantalla). Por ejemplo, el del teclado puede ser así

```
public void run()
{
  while (true)
  {
    String texto = leeDelTeclado();
    enviaAlServidor(texto);
  }
}
```

Esta opción es mejor, dos hilos con dos bucles infinitos, uno encargado del servidor y otro del teclado.

Ahora viene la pregunta . Si queremos detener este hilo, ¿qué hacemos?. Los **Thread** de java tienen muchos métodos para parar un hilo: **detroy()**, **stop()**, **suspend()** … Pero, si nos paramos a mirar la API de **Thread**, vemos que todos esos métodos son inseguros, están obsoletos, desaconsejados .

¿Cómo paramos entonces el hilo?

La mejor forma de hacerlo es implementar nosotros mismos un mecanismo de parada, que lo único que tiene que hacer es terminar el método **run()**, saliendo del bucle.

Un posible mecanismo es el siguiente:

```
public class MiHilo extends Thread
{
  // boolean que pondremos a false cuando queramos parar el hilo
  private boolean continuar = true;
  // metodo para poner el boolean a false.
  public void detenElHilo()
  {
    continuar=false;
  }
  // Metodo del hilo
  public void run()
  {
    // mientras continuar ...
    while (continuar)
    {
       String texto = leeDelTeclado();
       enviaAlServidor(texto);
    }
  }
}
```

Simplemente hemos puesto en la clase un **boolean** para indicar si debemos seguir o no con el bucle infinito. Por defecto a **true**. Luego hemos añadido un método para cambiar el valor de ese **boolean** a **false**. Finalmente hemos cambiado la condición del bucle que antes era **true** y ahora es **continuar**.

Para parar este hilo, sería:

```
MiHilo elHilo = new MiHilo();
elHilo.start();
// Ya tenemos el hilo arrancado
...
// Ahora vamos a detenerlo
elHilo.detenElHilo();
```

Ejemplo de hilo al extender la clase Thread

La implementación de Runnable es una forma de crear una clase que pueda instanciar objetos hilos. Extender de Thread es la otra. En este ejemplo, se verá cómo extender Thread .

Cuando una clase extiende de **Thread**, debe anular el método **run()**, que es el punto de entrada para el nuevo hilo. También debe llamar a **start()** para comenzar la ejecución del nuevo hilo. Es posible anular otros métodos Thread, pero no es necesario.

Crea un archivo ExtendHilo.java.

```
class MiHilo extends Thread{
	//Construye un nuevo hilo.
	MiHilo(String nombre){
	//super se usa para llamar a la versión del constructor de Thread
	super(nombre);
	}

	//Punto de entrada del hilo
	public void run(){
		System.out.println(getName()+" iniciando.");
		//Como ExtendThread extiende de Thread, puede llamar directamente
		//a todos los métodos de Thread, incluido el método getName().
		try {
			for (int cont=0;cont<10;cont++){
			Thread.sleep(400);
			System.out.println("En "+getName()+ ", el recuento es "+cont);
			}
		}catch (InterruptedException exc){
			System.out.println(getName()+ "interrumpido.");
		}
		System.out.println(getName()+ "finalizando.");
	}
}

class ExtendHilo{
	public static void main(String[] args) {
		System.out.println("Iniciando hilo principal.");
		MiHilo mh=new MiHilo("#1");
		mh.start();
		for (int i=0;i<50;i++){
			System.out.print(".");
			try {
				Thread.sleep(100);
			}catch (InterruptedException exc) {
			System.out.println("Hilo principal interrumpido");
			}
		}
			System.out.println("Hilo principal finalizado");
	}
}
```

La clase ***Thread*** define varios métodos que ayudan a administrar los hilos. Estos son algunos de los más utilizados.

Tabla de métodos de la clase Thread.

Método	Significado
final String getName()	Obtiene el nombre de un hilo.
final int getPriority	Obtiene la prioridad de un hilo.
final boolean isAlive()	etermina si un hilo todavía se está ejecutando.
final void join()	Espera a que termine un hilo.
void run()	Punto de entrada para el hilo.
static void sleep(long milisegundos)	Suspende un hilo durante un período específico de milisegundos.
void start()	Inicia un hilo llamando a su método run().

Todos los procesos tienen al menos un hilo de ejecución, que generalmente se denomina hilo principal (*main thread*), porque es el que se ejecuta cuando comienza el programa.. Desde el hilo principal, se pueden crear otros hilos.

Ejercicio 1: TIC TAC

Crea dos clases (hilos) Java que extiendan la clase Thread. Uno de los hilos debe visualizar en pantalla en un buble infinito la palabra TIC y el otro la palabra TAC. Dentro del bucle utiliza el método sleep() para que de tiempo a ver las palabras que se visualizan cuando se ejecute. Tendrás que añadir el bloque try-catch (para capturar la excepción InterruptedException). Crea después la función main() que haga uso de los hilos anteriores. ¿Se visualizan alternadamente?

Runnable

3.3. Creando un hilo con la interfaz Runnable

Hasta ahora sabemos que se crea un hilo instanciando un objeto de tipo **Thread**. La clase **Thread** encapsula un objeto que se puede ejecutar.

¿Qué ocurre si queremos hacer concurrente los objetos de una clase que hereda de otra que no es Thread?

> JAVA NO PERMITE LA HERENCIA MÚLTIPLE
>
> Java permite heredar sólo de una clase pero implementar múltiples interfaces.

Una **interfaz** en **Java** es una colección de métodos abstractos y propiedades constantes. En las **interfaces** se especifica qué se debe hacer pero no su implementación. Serán las **clases** que implementen estas **interfaces** las que describen la lógica del comportamiento de los métodos.

JAVA proporciona dos palabras reservadas para trabajar con interfaces: **interface** e **implements.**

Para declarar una interfaz se utiliza:

```
modificador_acceso interface NombreInterfaz {
    ....
}
```

modificador_acceso puede ser una clase de objetos que nos permite utilizar herencia en abstracción constante en las clases en las que se implemente.

Para implementarla en una clase, se utiliza la forma:

```
modificador_acceso class NombreClase implements NombreInterfaz1 , NombreInterfaz2
```

Una clase puede implementar varias interfaces de los paquetes que se han importado dentro del programa, separando los nombres por comas.

Ejemplo

Definición de una interfaz

```
interface Nave {
    public  moverPosicion (int x, int y);
    public  disparar();
    .....
}
```

Uso de la interfaz definida

```
public class NaveJugador implements Nave {
    public void moverPosicion (int x, int y) {
        //Implementación del método
        posActualx = posActualx - x;
        posActualy = posActualy - y;
    }
    public void disparar() {
        //Implementación del método
    }
    ...
}
```

Con esto solucionamos el problema del Thread

```
class ObjetoHijo extends ObjetoPadre implements Runnable
```

Luego tenemos que implementar el método ***run()*** al igual que hacíamos con el Thread. De esta manera hemos evitado el problema de la herencia múltimple.

```
ObjetoHijo OH = new ObjetoHijo;
// No podemos acceder a OH.start() porque no es un Thread
// Se crea un hilo de ejecución y se indica que el objeto OH se va a ejecutar en él.
Thread HiloOH = new Thread (OH);
HiloOH.start(); // Se ejecuta concurrentemente en método run() asociado con HiloOH
```

Creando un hilo Runnable.

Java define dos formas en las que puede crear un objeto ejecutable:

Los hilos se pueden crear utilizando dos mecanismos:

1. Extender la clase Thread
2. Implementar la interfaz Runnable

Recuerde: ambos enfoques aún usan la clase Thread para crear instancias, acceder y controlar el hilo. La única diferencia es cómo se crea una clase habilitada para hilos.

La interfaz **Runnable** abstrae una unidad de código ejecutable. Puede construir un hilo en cualquier objeto que implemente la interfaz Runnable. Runnable define solo un método llamado **run()**, que se declara así:

public void run()

Dentro de **run()**, se definirá el código que constituye el nuevo hilo. Es importante entender que *run()* puede llamar a otros métodos, usar otras clases y declarar variables como el hilo principal. La única diferencia es que *run()* establece el punto de entrada para otro hilo de ejecución concurrente dentro de su programa. Este hilo terminará cuando retorne *run()*.

Después de crear una clase que implemente **Runnable**, instanciará un objeto del tipo Thread en un objeto de esa clase. El hilo define varios constructores. El que usaremos primero será el siguiente:

Thread(Runnable threadOb)

En este constructor, ***threadOb*** es una instancia de una clase que implementa la interfaz *Runnable*. Esto define dónde comenzará la ejecución del hilo.

Una vez creado, el nuevo hilo no comenzará a ejecutarse hasta que llame a su método **start()**, que se declara dentro de Thread. En esencia, *start()* ejecuta una llamada a *run()*. El método *start()* se muestra de la siguiente forma:

void start ()

Ejemplo Runnable.

Creamos un programa Java con la clase principal y una segunda clase que llamaremos Hilo2.java

```
package informatica.dam;
public class Main {
   public static void main(String[] args) {
      Hilo2 uno, dos;
      uno = new Hilo2("Jamaica");
      dos = new Hilo2("Fiji");
      System.out.println("Main no hace nada");
   }
}
```

```
package informatica.dam;
public class Hilo2 implements Runnable {
   private Thread miHilo = null;
   public Hilo2 (String str) {
      miHilo = new Thread(this,str );
      miHilo.start();
   }
   public void run() {
     for (int i =0; i<10; i++) {
        System.out.println(i+" "+miHilo.getName());
        try {
         Thread.sleep((long)(Math.random()*1000));
        } catch (InterruptedException e) {
        }
     }
     System.out.println("Fin! "+miHilo.getName());
   }
}
```

Salida:

Main no hace nada

```
0 Fiji
0 Jamaica
1 Jamaica
1 Fiji
2 Jamaica
3 Jamaica
2 Fiji
3 Fiji
4 Fiji
4 Jamaica
5 Fiji
5 Jamaica
6 Fiji
7 Fiji
6 Jamaica
8 Fiji
7 Jamaica
9 Fiji
8 Jamaica
9 Jamaica
Fin! Fiji
Fin! Jamaica
```

Ejemplo de hilo mediante la implementación de la interfaz Runnable

Ejemplo que crea un nuevo hilo y lo ejecuta:

```
//Crea un hilo implementando Runnable.
//Los objetos de MiHilo se pueden ejecutar en sus propios hilos
// porque MiHilo implementa Runnable.
 class MiHilo implements Runnable {
    String nombreHilo;
    MiHilo(String nombre){
       nombreHilo=nombre;
    }
    //Punto de entrada del hilo
    //Los hilos comienzan a ejecutarse aquí
    public void run(){
       System.out.println("Comenzando "+nombreHilo);
       try {
          for (int contar=0; contar<10; contar++){
             Thread.sleep(400);
             System.out.println("En "+nombreHilo+", el recuento "+contar);
          }
       }catch (InterruptedException exc){
          System.out.println(nombreHilo + "Interrumpido.");
       }
       System.out.println("Terminando "+nombreHilo);
    }
}
```

```
class UsoHilos{
   public static void main(String[] args) {
      System.out.println("Hilo principal iniciando.");
      //Primero, construye un objeto MiHilo.
      MiHilo mh=new MiHilo("#1");
      //Luego, construye un hilo de ese objeto.
      Thread nuevoh=new Thread(mh);
      //Finalmente, comienza la ejecución del hilo.
      nuevoh.start();
      for (int i=0; i<50;i++){
         System.out.print(" .");
      }try{
         Thread.sleep(100);
      }catch (InterruptedException exc){
         System.out.println("Hilo principal interrumpido.");
      }
      System.out.println("Hilo principal finalizado.");
   }
}
```

¿Qué hace este programa?

Primero, *MiHilo* implementa *Runnable*. Esto significa que un objeto de tipo *MiHilo* es adecuado para usar como un hilo y se puede pasar al constructor de Thread.

Dentro de *run(),* se establece un bucle que cuenta de 0 a 9. Observar la llamada a *sleep()*. El método *sleep()* hace que el hilo del que se llama suspenda la ejecución durante el período especificado de milisegundos. Su forma general es:

static void sleep(long milisegundos) throws InterruptedException

La cantidad de milisegundos para suspender se especifica en *milisegundos*. Este método puede lanzar una **InterruptedException**. Por lo tanto, las llamadas a ella deben estar envueltas en un bloque **try**.

El método **sleep()** también tiene una segunda forma, que le permite especificar el período en términos de milisegundos y nanosegundos si necesita ese nivel de precisión. En run(), sleep() pausa el hilo durante 400 milisegundos cada vez a través del bucle. Esto permite que el hilo se ejecute con la lentitud suficiente para que pueda verlo ejecutar.

Dentro de main(), se crea un nuevo objeto Thread mediante la siguiente secuencia de instrucciones:

```
//Primero, construye un objeto MiHilo.
 MiHilo mh=new MiHilo("#1");
//Luego, construye un hilo de ese objeto.
 Thread nuevoh=new Thread(mh);
//Finalmente, comienza la ejecución del hilo.
 nuevoh.start();
```

Primero se crea un objeto de *MiHilo*. Este objeto luego se usa para construir un objeto *Thread*. Esto es posible porque *MiHilo* implementa Runnable. Finalmente, la ejecución del nuevo hilo se inicia llamando a **start()**. Esto hace que comience el método run() del hilo hijo.

Después de llamar a *start()*, la ejecución vuelve a *main()*, y entra *main()* para el ciclo. Observe que este ciclo itera 50 veces, pausando 100 milisegundos cada vez a través del ciclo. Ambos hilos continúan ejecutándose, compartiendo la CPU en sistemas de una sola CPU, hasta que terminan sus bucles. El resultado producido por este programa es el siguiente.

```
Hilo principal iniciando.
.Comenzando #1
.....En #1, el recuento 0
....En #1, el recuento 1
....En #1, el recuento 2
....En #1, el recuento 3
....En #1, el recuento 4
....En #1, el recuento 5
....En #1, el recuento 6
....En #1, el recuento 7
....En #1, el recuento 8
....En #1, el recuento 9
Terminando #1
........Hilo principal finalizado.
```

Explicación de hilo en Java

El hecho de que el hilo *main* y *mh* se ejecutan simultáneamente, es necesario evitar que *main()* termine hasta que no termine *mh*.

Aquí, esto se hace a través de las diferencias de tiempo entre los dos hilos. Porque las llamadas a **sleep()** dentro del bucle **for** de **main()** causan un retraso total de 5 segundos (50 iteraciones por 100 milisegundos), pero el retardo total dentro del bucle **run()** es de solo 4 segundos (10 iteraciones por 400 milisegundos), **run()** finalizará aproximadamente 1 segundo antes de *main()*. Como resultado, tanto el hilo *main* como *mh* se ejecutarán simultáneamente hasta que termine *mh*. Luego, aproximadamente 1 segundo más tarde, *main()* finaliza.

Aunque este uso de las diferencias de tiempo para asegurar que *main()* termina al final es suficiente para este simple ejemplo, no es algo que normalmente se usaría en la práctica. Java proporciona formas mucho mejores de esperar a que termine un hilo. Sin embargo, es suficiente para los próximos programas.

Otro punto: en un programa multihilo, a menudo querrás que **el hilo principal sea el último hilo que termine ejecutando**. Como regla general, un programa continúa ejecutándose hasta que todos sus hilos hayan finalizado. Por lo tanto, no es obligatorio finalizar el hilo principal al final. Sin embargo, a menudo es una buena práctica.

Crear múltiples hilos

Los ejemplos anteriores han creado solo un hilo hijo. Sin embargo, su programa puede engendrar tantos hilos como necesite. Por ejemplo, el siguiente programa crea tres hilos hijos:

```
class MiHilo implements Runnable{
   Thread hilo;
   //Construye un nuevo hilo.
    MiHilo(String nombre){
       hilo= new Thread(this,nombre);
    }
    //Un método de fábrica que crea e inicia un hilo.
    public static MiHilo crearYComenzar (String nombre){
       MiHilo miHilo=new MiHilo(nombre);
       miHilo.hilo.start(); //Inicia el hilo
       return miHilo;
    }
    //Punto de entrada de hilo.
    public void run(){
       System.out.println(hilo.getName()+" iniciando.");
       try {
          for (int count=0; count<10;count++){
             Thread.sleep(400);
             System.out.println("En "+hilo.getName()+ ", el recuento es "+count);
          }
       }catch (InterruptedException exc){
          System.out.println(hilo.getName()+ " interrumpudo.");
       }
       System.out.println(hilo.getName()+" terminado.");
    }
}
```

```
class MasHilos {
    public static void main(String[] args) {
       System.out.println("Hilo principal iniciando.");
       MiHilo miHilo1 = MiHilo.crearYComenzar("#1");
       MiHilo miHilo2 = MiHilo.crearYComenzar("#2");
       MiHilo miHilo3 = MiHilo.crearYComenzar("#3");
       for (int i = 0; i < 50; i++) {
          System.out.print(".");
          try {
             Thread.sleep(100);
          } catch (InterruptedException exc) {
             System.out.println("Hilo principal interrumpido.");
          }
       }
       System.out.println("Hilo principal finalizado");
    }
}
```

Salida:

```
Hilo principal iniciando.
.#2 iniciando.
#1 iniciando.
#3 iniciando.
....En #1, el recuento es 0
En #2, el recuento es 0
...
En #3, el recuento es 9
#3 terminado.
.........Hilo principal finalizado
```

Una vez iniciados, los tres hilos hijos comparten la CPU. Tener en cuenta que en esta ejecución **los hilos se inician en el orden en que se crean**. Sin embargo, esto puede no ser siempre el caso.

Java es libre de programar la ejecución de los hilos a su manera. Por supuesto, debido a las diferencias en el tiempo o el entorno, el resultado preciso del programa puede variar.