

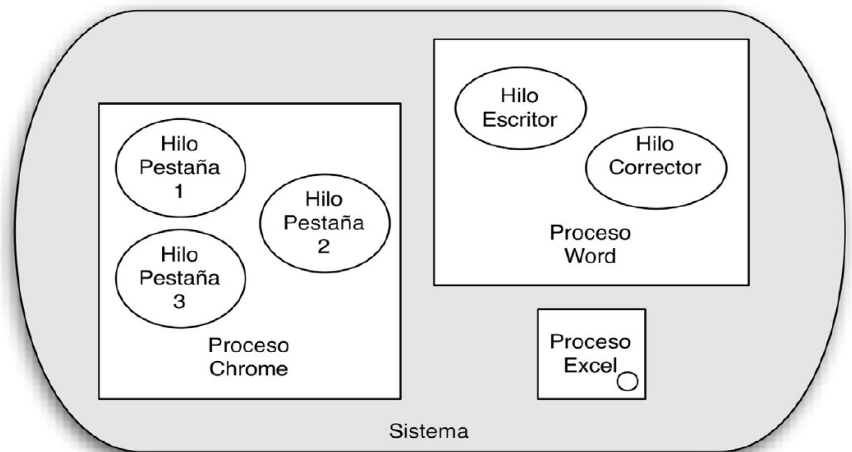
PROGRAMACIÓN MULTITHILO.

1. HILOS (THREAD).

Un proceso es un programa que se está ejecutando. Por lo tanto, la multitarea basada en procesos es la que permite que en un ordenador se ejecuten dos o más programas concurrentemente. Por ejemplo, la multitarea basada en procesos permite que se ejecute el compilador de Java al mismo tiempo que se está usando un editor de textos.

Pero, ¿qué es un hilo? De la misma manera que un S.O puede ejecutar varios procesos al mismo tiempo bien sea por concurrencia o paralelismo, dentro de **un proceso** puede haber **varios hilos** de ejecución. Por tanto, un **hilo** puede definirse como cada **secuencia de código dentro de un proceso** que ejecuta sus instrucciones de forma independiente. Los hilos no pueden ejecutarse ellos solos, **necesitan** supervisión de un **proceso padre**.

Los hilos son **procesos ligeros**, comparten el **mismo espacio de direccionamiento** y el mismo **proceso**. La multitarea basada en hilos permite escribir **programas muy eficientes que hacen una utilización óptima de la CPU**, ya que el tiempo que ésta está libre se reduce al mínimo. Además, los hilos son imprescindibles en una arquitectura cliente/servidor. Por ejemplo, supóngase un servidor de páginas web que recibe peticiones de varios clientes, si no utilizamos programación multihilo solamente se podría atender peticiones de una en una.



En la Figura puede verse cómo sobre el hardware (una o varias CPU's) se sitúa el Sistema Operativo. Sobre éste se sitúan los procesos (Pi) que pueden ejecutarse concurrentemente y dentro de estos se ejecutan los hilos (hj) que también se pueden ejecutar de forma concurrente dentro del proceso. Es decir, tenemos **concurrencia a dos niveles**, una entre procesos y otra entre hilos de un mismo proceso. Si por ej. tenemos dos procesadores, se podrían estar ejecutando al mismo tiempo el hilo 1 del proceso 1 y el hilo 2 del proceso 3. Otra posibilidad podría ser el hilo 1 y el hilo 2 del proceso 1.

Los **hilos comparten la información del proceso** (código, datos, etc). Si un **hilo modifica una variable** del proceso, el **resto** de hilos **verán esa modificación** cuando accedan a esa variable. Los **cambios de contexto** entre hilos **consumen poco** tiempo de procesador, de ahí su éxito.

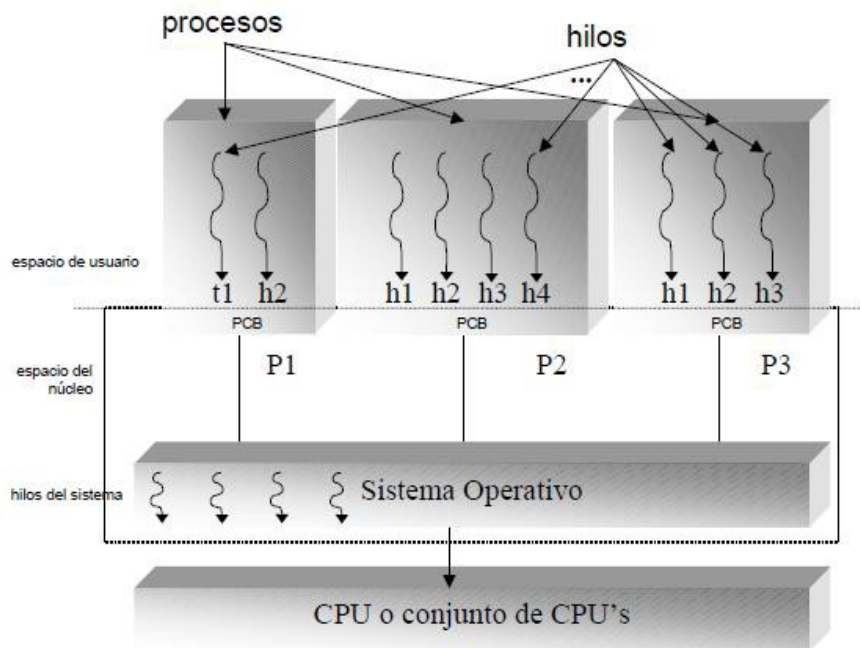


Figura 5 Concurrencia a dos niveles: procesos e hilos.

Resumiendo, las diferencias entre hilos y procesos son:

- Los **hilos** se ejecutan **dentro** del contexto de un **proceso**. Dependen de un proceso para ejecutarse.
- Los **procesos son independientes** y tienen espacios de memoria diferentes.
- Dentro de un mismo **proceso** pueden coexistir **varios hilos** ejecutándose que **compartirán la memoria** de dicho proceso.

2. LOS HILOS EN JAVA.

En Java un hilo es una clase que descende de la clase `java.lang.Thread` o bien que implementa a la interfaz `Runnable` (útil en los casos de que la clase ya forme parte de una jerarquía de clases esto es que ya derive de otra clase). La forma de construcción del hilo (derivación o implementación) es independiente de su utilización, una vez creados se usan de la misma forma sea cual sea el método de construcción utilizado.

El intérprete de Java utiliza las **prioridades** para determinar cómo debe tratar cada hilo con respecto a los demás. La prioridad de un hilo se utiliza para decidir cuándo se pasa a ejecutar otro hilo (*cambio de contexto*).

Java asigna a cada hilo una prioridad que determina cómo se debe tratar a ese hilo con relación a los demás. Las prioridades de los hilos son valores enteros. Cuando un hilo cede voluntariamente el uso de la CPU, ésta se asigna al hilo que estando preparado para ejecutarse tenga mayor prioridad. Además, un hilo puede ser desalojado por otro con prioridad más alta.

Para la sincronización de hilos, Java implementa monitores. En Java a cada objeto se asocia un monitor, y una vez que un hilo entra en el monitor todos los demás hilos deben esperar hasta que el hilo salga del monitor.

HILOS DEMONIO

Un **proceso demonio** es un proceso que debe **ejecutarse continuamente en modo *background*** (en segundo plano), y generalmente se diseña para responder a peticiones de otros procesos a través de la red.

La palabra “*daemon*” es propia de UNIX, pero no se utiliza de este mismo modo en Windows. En **Windows**, los demonios se denominan “**servicios**”. Cuando los servicios atienden peticiones, se conocen como la parte “Servidor” de una arquitectura Cliente/Servidor.

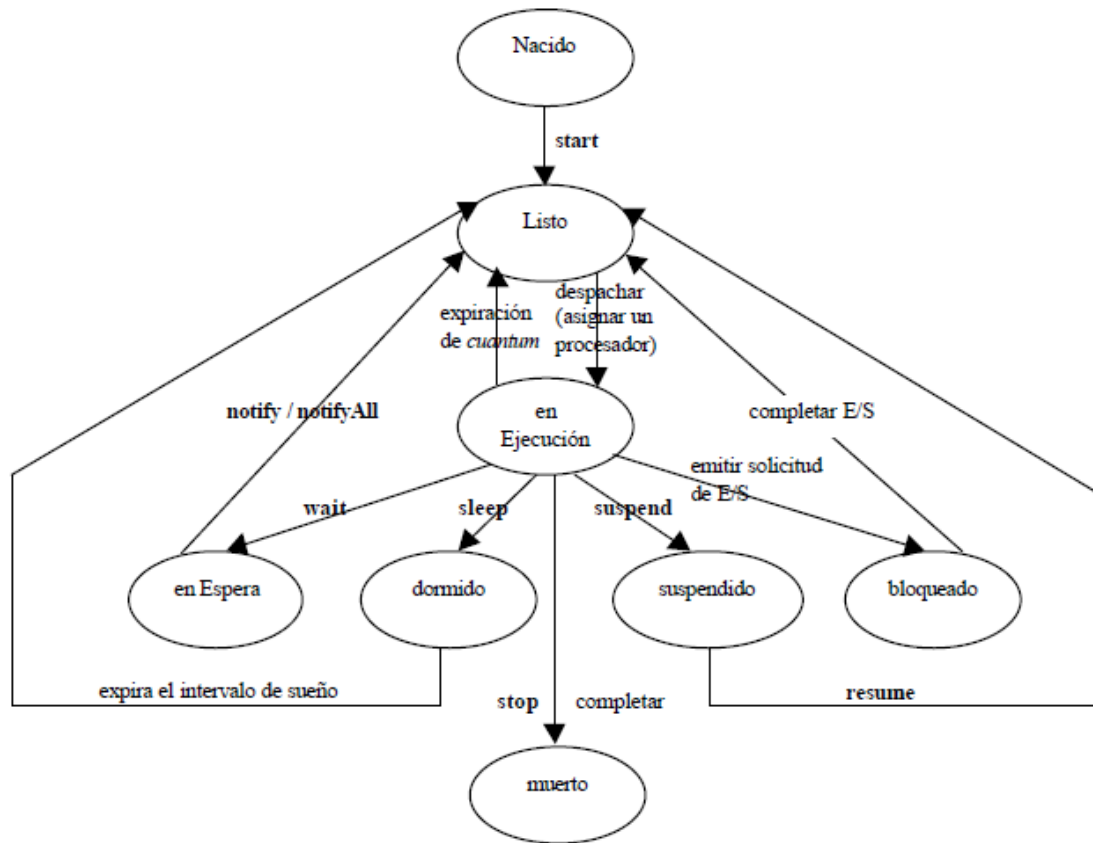
Los hilos demonio también se llaman servicios, porque se ejecutan, normalmente, con prioridad baja y proporcionan un **servicio básico a un programa o programas** cuando la actividad de la máquina es reducida. Un ejemplo de hilo demonio que está ejecutándose continuamente es el recolector de basura (*garbage collector*). Este hilo, proporcionado por la Máquina Virtual Java, comprueba las variables de los programas a las que no se accede nunca y libera estos recursos, devolviéndolos al sistema.

Un **hilo** puede fijar su **indicador de demonio** pasando un valor **true** al método **setDaemon()**. Si se pasa **false** a este método, el hilo será devuelto por el sistema como un hilo de usuario. No obstante, esto último debe realizarse antes de que se arranque el hilo con el método **start()**.

2.1. LOS ESTADOS DE UN HILO.

En Java un hilo puede encontrarse básicamente en uno de los siguientes estados:

- **Nuevo (*New*)**: El **thread** ha sido creado pero no inicializado, es decir, no se ha ejecutado todavía el método **start()**. Se producirá un mensaje de error (**IllegalThreadStateException**) si se intenta ejecutar cualquier método de la clase **Thread** distinto de **start()**.
- **Ejecutable (*Runnable*)**: El **thread** puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado en beneficio de otro **thread**.
- **Bloqueado (*Blocked* o *Not Runnable*)**: El **thread** podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (E/S). Si un **thread** está en este estado, no se le asigna tiempo de CPU.
- **Muerto (*Dead*)**: La forma habitual de que un **thread** muera es finalizando el método **run()**. También puede llamarse al método **stop()** de la clase **Thread**, aunque dicho método es considerado “peligroso” y no se debe utilizar.



Ciclo de vida de un hilo

Cuando se **instancia** un **thread**, se inicializa sin asignarle recursos. Está en el estado **nuevo Thread (Nacido)**. Un **thread** en este estado únicamente acepta las llamadas a los métodos **start()** o **stop()**.

La llamada al método **start()** asigna los recursos necesarios al objeto, lo sitúa en el estado **Listo** y llama al método **run()** del objeto. Esto no significa que el **thread** esté ejecutándose sino que está en **disposición de ejecutarse** en cuanto la CPU le conceda su tiempo.

Un **thread** en estado **ejecutable** puede pasar al estado **no ejecutable** por alguna de las siguientes razones:

- que sean invocados alguno de sus métodos **sleep()** → pasará al **estado dormido**
- o que se invoque su método **suspend()**, pasará al **estado suspendido**
- o que el **thread** haga uso de su método **wait()**, con lo que estaría **En Espera**
- o que el **thread** esté **bloqueado** esperando una operación de entrada/salida o que se le asigne algún recurso.

Un **thread** puede pasar al estado **muerto** por dos motivos: que finalice normalmente su método **run()** o que se llame a su método **stop()** desde cualquiera de sus posibles estados (nuevo **thread**, ejecutable o no ejecutable).

Un **thread** pasa del estado **no ejecutable** a **ejecutable** por alguna de las siguientes razones:

- **dormido:** que pase el **tiempo de espera** indicado por su método **sleep()**, momento en el cual, el **thread** pasará al estado **ejecutable** y, si se le asigna la CPU, proseguirá su ejecución.
- **suspendido:** que, después de haber sido suspendido mediante el método **suspend()**, sea continuado mediante la llamada a su método **resume()**.
- **esperando:** que después de una llamada a **wait()** se continúe su ejecución con **notify()** o **notifyAll()**.
- **bloqueado:** una vez **finalizada una espera** sobre una operación de E/S sobre algún recurso.

2.2. EL HILO PRINCIPAL.

Cuando un programa Java comienza su ejecución, ya hay un hilo ejecutándose, es el indicado por el método `main`. Este hilo se denomina *hilo principal* del programa. El hilo principal es importante por dos razones:

- Es el *hilo a* partir del cual se crean el resto de los hilos del programa.
- Debe ser el *último que finaliza* su ejecución. Cuando el hilo principal finaliza, el programa termina. (Si un hilo principal finaliza antes que un hijo, Java puede bloquearse, *hang*)

Si no se crean más hilos desde el hilo principal tendremos algo como lo representado en la Figura inferior donde sólo existe un hilo de ejecución que va recorriendo los distintos objetos participantes en el programa según se vayan produciendo las distintas llamadas entre métodos de los mismos.

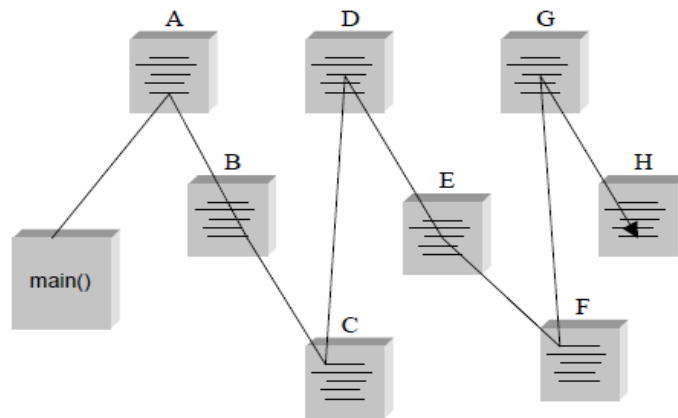


Figura Varios objetos y un solo hilo.

Sin embargo, si desde el *hilo principal se crean* por ejemplo *dos hilos* tendremos algo como lo representado en la Figura de abajo, en la que se puede ver cómo el programa se está *ejecutando* al mismo tiempo *por tres sitios* distintos: el hilo principal más los dos hilos creados. Los hilos pueden estar ejecutando código en diferentes objetos, código diferente en el mismo objeto o incluso el mismo código en el mismo objeto y al mismo tiempo.

Es necesario ver la diferencia entre objeto e hilo. Un objeto es algo estático, tiene una serie de atributos y métodos. Quien realmente ejecuta esos métodos es el hilo de ejecución. En ausencia de hilos sólo hay un hilo que va recorriendo los objetos según se van produciendo las llamadas entre métodos de los objetos. Podría darse el caso del objeto E en la Figura , donde es posible que tres hilos de ejecución distintos estén ejecutando el mismo método al mismo tiempo.

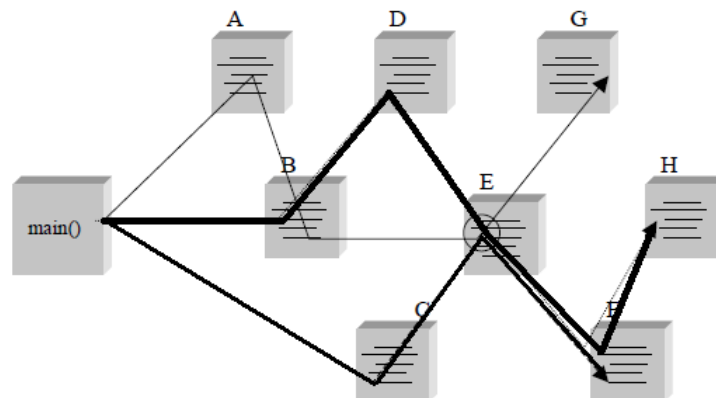


Figura Tres hilos "atravesando" varios objetos.

El método `run()` es el punto de entrada de un nuevo hilo de ejecución concurrente dentro de un programa. El hilo termina cuando finalice el método `run()`. Para que un hilo comience su ejecución se debe llamar a su método `start()`.

2.3. CREACIÓN DE UN HILO.

Los threads o hilos de ejecución permiten organizar los recursos del ordenador de forma que pueda haber varios programas actuando en paralelo. Un hilo de ejecución puede realizar cualquier tarea que pueda realizar un programa normal y corriente. Bastará con indicar lo que tiene que hacer en el método `run()`, que es el que define la actividad principal de los threads.

En **Java** hay dos formas de crear nuevos *threads*:

1. La primera consiste en crear una nueva clase que herede de la clase `java.lang.Thread` y sobrecargar el método `run()` de dicha clase
2. El segundo método consiste en declarar una clase que implemente la interface `java.lang.Runnable`, la cual declarará el método `run()`; posteriormente se crea un objeto de tipo `Thread` pasándole como argumento al constructor el objeto creado de la nueva clase (la que implementa la interface `Runnable`). Tanto la clase `Thread` como la interface `Runnable` pertenecen al *package* `java.lang`, por lo que no es necesario importarlas.

2.3.1. Heredando de la clase Thread.

Para crear un hilo de esta forma simplemente hay que crear una clase que extienda a la clase `Thread`. Debemos sobrescribir el método `run()` para poder indicar el código que deseamos que ejecute nuestro hilo. Esta clase ya será un hilo, por lo tanto no estaría demás el crear un constructor que llamase al constructor de `Thread` y le diese un nombre al hilo (`super(nombre)`). Podemos crear hilos desde cualquier parte de un programa creando objetos de esta nueva clase, y tratarlos como tales: `start()`, `sleep()`, `setName()`, etc.

Para poner en marcha este nuevo thread se debe crear un objeto de la clase creada, y llamar al método `start()`, heredado de la super-clase `Thread`, que se encarga de llamar a `run()`.

```
class NombreHilo extends Thread{
    //propiedades, constructores y métodos de la clase.

    public void run(){
        //acciones que lleva a cabo el hilo
    }
}
```

Veamos un ejemplo:

1. Creo mi hilo como subclase de `Thread`
2. En la aplicación principal (main) creo un objeto hilo e invoco al método `start()` para que comience a ejecutarse el hilo llamando al método `run()`

```

//Se crea la clase EjemHilo que hereda de Thread

class EjemHilo extends Thread{

    //constructor, se asigna un nombre al nuevo hilo
    public EjemHilo (String nom){
        super(nom); //Llamada al constructor de la superclase.
    }

    //redefinición del método run(), que es el que contiene
    //las indicaciones de lo que hará el hilo
    public void run(){
        for (int i=0;i<10;i++){
            //Mostrará 10 veces el nombre del hilo
            System.out.println("Este es el thread: "+getName());
        }
    }
}

public class MainEjemHilo{
    public static void main(String args[]){
        /*Para poner en marcha el Thread, creamos un objeto de la clase
        EjemHilo y llamamos al método start() heredado de la superclase
        Thread, que se encarga de llamar a run().

        */

        EjemHilo miHilo = new EjemHilo("HiloEjemplo");
        miHilo.start();
    }
}

```

2.3.2. Implementando la interfaz Runnable.

Para crear un hilo de esta forma simplemente hay que crear una clase que implemente ¹la interfaz *Runnable*, y luego redefinimos el método *run()*. Este método define el código que constituye el nuevo hilo, es importante entender que desde aquí se puede llamar a otros métodos, usar otras clases, declarar variables,... La única diferencia es que el método *run()* establece el punto de entrada para otro hilo de ejecución concurrente dentro del programa, es decir, puede haber dos hilos ejecutando a la vez código. El hilo finaliza cuando su método *run()* acaba. Después de crear una clase que implemente la interfaz *Runnable* podemos crear varios hilos que ejecuten su código, para ello desde cualquier parte del programa crearemos objetos de tipo *Thread*.

¹Extends es para declarar una relación de herencia, es decir, establecer una relación “es un”

Implements es para implementar clases abstractas, las clases abstractas tienen declaración de funciones sin ser implementadas, por esa razón cuando usas implements tienes que sobreescribir todas las funciones de la clase abstracta. Una clase puede implementar más de una interfaz en Java, pero sólo puede extender una clase. Esto es lo más parecido que tiene Java a la herencia múltiple.


```

class NombreHilo implements Runnable{
    //propiedades, constructores y métodos de la clase.
    public void run(){
        //acciones que lleva a cabo el hilo
    }
}

```

La 2ª forma tiene como ventaja el hecho de que como en Java no hay herencia múltiple, esta forma de implementar hilos permite hacer que el método `run()` de una clase se ejecute en un hilo y además, esta clase herede el comportamiento de otra clase. En la 1ª forma nuestra clase no podría heredar de otras clases.

Un ejemplo sería el siguiente:

```

class HiloRunnable implements Runnable{
    //atributos
    String palabra;

    //constructor
    public HiloRunnable (String pala){
        palabra=pala;
    }

    //redefinición del método run()
    public void run(){
        for (int i=0;i<10;i++){
            //Mostrará 10 veces la palabra
            System.out.println(palabra);
        }
    }
}

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

        /*Se crean dos objetos de la clase HiloRunnable */
        HiloRunnable objeto1= new HiloRunnable("Hola");
        HiloRunnable objeto2= new HiloRunnable("Adiós");

        /*En HiloRunnable sólo hemos creado una clase. Los objetos de esa clase
        no son hilos ya que no heredan de la clase Thread. Pero si queremos
        que un objeto de esta nuestra clase se ejecuten en un hilo independiente,
        entonces tenemos que crear un objeto de la clase Thread y le pasamos
        como parámetro el objeto donde queremos que empiece la ejecución del hilo*/

        Thread hilo1 = new Thread(objeto1);
        Thread hilo2 = new Thread(objeto2);

        /*Invocamos el método start() de la clase Thread que se encarga de llamar
        al método run() de los objetos objeto1 y objeto2 */

        hilo1.start();
        hilo2.start();
    }
}

```


Metodología para la creación del *thread* mediante *implements*:

- 1) La clase creada debe implementar el interface `Runnable`: `class PrimerThread implements Runnable {`
- 2) Redefinir el método `run()` tal y como se hace en la otra alternativa de creación de *threads* (mediante subclases de `Thread`).
- 3) En el main, hay que
 - Instanciar un objeto de nuestra clase implementada, teniendo en cuenta si necesita parámetros el constructor o no.

```
PrimerThread objeto = new PrimerThread(parametros)
```

- Crear un objeto, un hilo, de la clase `Thread` llamando a su constructor y pasándole como parámetro el objeto creado

```
Thread hilo = new Thread(objeto);
```

Una vez declarada la clase que implementa la interface `Runnable`, ya puede ser instanciada e iniciada como cualquier *thread*. Es más, cualquier subclase descendiente de esta clase poseerá también las características propias de los *threads*.

- Iniciar el thread

```
Hilo.start();
```

Ahora mostramos un ejemplo en el que ya se crea el hilo (de la clase `Thread`) en el constructor de nuestra clase implementada, con lo cual, cuando creamos un nuevo objeto de la clase implementada ya creamos un hilo (objeto de la clase `Thread`) y por lo tanto por eso ya no lo creamos en el programa principal.

```

class HelloThread implements Runnable {
    Thread t; //La clase tiene un atributo hilo

    HelloThread () { //Constructor
        t = new Thread(this, "Nuevo Thread");
        System.out.println("Creado hilo: " + t);
        t.start(); // Arranca el nuevo hilo de ejecución. Ejecuta run
    }

    public void run() {
        System.out.println("Hola desde el hilo creado!");
        System.out.println("Hilo finalizando.");
    }
}

class RunThread { //Programa principal
    public static void main(String args[]) {
        new HelloThread(); // Crea un nuevo hilo de ejecución
        System.out.println("Hola desde el hilo principal!");
        System.out.println("Proceso acabando.");
    }
}

```

EJERCICIO → Prueba a realizar el ejemplo del apartado 2.3.1 “creación de hilo heredando de Thread” (donde se mostraba el nombre del hilo) pero ahora implementando la interface Runnable.

```

public class EjemHiloRunnable implements Runnable {
    String nombre;

    public EjemHiloRunnable (String nom) {
        nombre=nom;
    }

    public void run() {
        for (int i=0; i<10; i++)
            System.out.println("Nombre del hilo: " + nombre);
    }
}

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

        //Creo un objeto de la clase implementada
        EjemHiloRunnable objetoRun = new EjemHiloRunnable("Hilito");

        /*Creo un objeto de la clase Thread, un hilo, pasándole
        el objeto Runnable como argumento */
        Thread hilo = new Thread(objetoRun);

        hilo.start(); //Arranco el hilo
    }
}

```

De las dos alternativas, ¿cuál utilizar? Depende de la necesidad:

- La utilización de la interfaz *Runnable* es más general, ya que el objeto puede ser una subclase de una clase distinta de *Thread*.
- La utilización de la interfaz *Runnable* no tiene ninguna otra funcionalidad además de *run()* que la incluya por el programador.
- La extensión de la clase *Thread* es más fácil de utilizar, ya que está definida una serie de métodos útiles para la administración de hilos.
- La extensión de la clase *Thread* está limitada porque las clases creadas como hilos deben ser descendientes únicamente de dicha clase.