Fíjate que para poder utilizar hilos, necesitamos usar muchos de los conceptos aprendidos de POO: extends, interface, sobreescritura de métodos, excepciones, etc.

Introducción y definiciones

viene muy bien en
http://docs.oracle.com/javase/tutorial/essential/concurrency/

Hilo, Multitarea, Concurrencia, Paralelismo …hay definiciones inevitables. Definiciones que dependen mucho del contexto, y que habría que matizar mucho. Se trata aquí simplemente de situarnos para entender dónde nos ubicamos al programar con hilos. LEE ESTOS CONCEPTOS RÁPIDAMENTE NO ES MÁS QUE UNA INTRODUCCIÓN.

Programa: Un programa es un conjunto de instrucciones. Se trata de algo estático. Para que el programa pueda hacer algo hay que ponerlo en ejecución. Por ejemplo, el código que hay dentro del fichero notepad.exe(el bloc de notas) es un programa editor de texto.

Proceso: A grandes rasgos, un proceso es un programa o parte de él en ejecución. También podemos decir que un proceso es una "instancia" de un programa. Si ejecuto el comando notepad.exe, el programa se carga en memoria y se convierte en un proceso. Observa que del mismo programa, por ejemplo notepad.exe puede haber muchas instancias abiertas y cada una de ellas es un proceso independiente, por ejemplo, con un notepad "abierto" trabajo en el fichero X y con otro notepad "abierto" trabajo con el fichero Y, ambos son procesos independientes. Esto demuestra que programa y proceso no es lo mismo pues un programa puede generar muchos procesos.

Proceso y tarea son sinónimos. Con el gestor de tareas de windows vemos los procesos o tareas actualmente en ejecución en el sistema.

La programación concurrente se ocupa de delimitar un conjunto de acciones(trozos de código) que pueden ser ejecutadas "simultáneamente" . Luego analizamos el significado de "simultáneamente".

La programación paralela es un tipo de programación concurrente diseñado para ejecutarse en un sistema multiprocesador

Lo de "simultáneamente": desde el punto de vista hardware, se consigue con dos técnicas:

    - Se comparte el "tiempo del procesador". Hay en este una ejecución en paralelo virtual, no es real. Los programas entran y salen tan rápido del procesador que aparentemente todos avanzan su ejecución en paralelo aunque realmente lo que ocurre es una ejecución solapada o entrelazada a toda leche que simula paralelismo. Idea similar a cuando varios ordenadores comparten la misma salida a internet.
    - El procesador tiene varios núcleos. Hay un paralelismo de ejecución real. En muchos textos se reserva el término concurrente para la ejecución entrelazada y el paralelismo real no se considera un subtipo de concurrencia. A nosotros no nos preocupa ahora mismo tanto rigor.

Estas dos capacidades del hardware son aprovechadas por el sistema operativo y las aplicaciones:

- Los sistemas operativos modernos son capaces de ejecutar concurrentemente dos o más procesos. Ejemplo: al mismo tiempo que navego por internet , puedo escanear los virus de un disco y escuchar música. Esto se llama concurrencia de procesos. El sistema operativo según el contexto puede aplicar concurrencia con tres técnicas (no exclusivas entre sí):
    - La multiprogramación es la forma de gestionar los procesos en un sistema monoprocesador.
    - El multiproceso es la gestión de varios procesos dentro de un sistema multiprocesador donde cada procesador puede acceder a una memoria común.
    - El procesamiento distribuido es la gestión de varios procesos en procesadores separados, cada uno con su memoria local
- Aplicaciones(nuestros programas por ejemplo). Nuestra aplicación al ejecutarse se convierte en un proceso, el sistema operativo procurará ejecutarla concurrentemente con otros procesos, pero por otro lado, un proceso se puede organizar en subprocesos, y estos subprocesos son trozos de código que se pueden ejecutar concurrentemente. Por ejemplo escribo un programa java y la interfaz gráfica y el acceso a BD puede ejecutarse concurrentemente de forma que mientras se actualiza algo en la BD puedo seguir haciendo operaciones en la ventanas del programa. **Los términos subproceso, lightweight processes, hebra e hilo(thread) son equivalentes.** Esto se llama por tanto concurrencia de hilos. Para la concurrencia de hilos de nuevo el sistema operativo puede aplicar a nuestros subprocesos las técnicas similares que aplica a los procesos: multiprogramación y multiproceso, pero esto es absolutamente transparente al programador de aplicaciones. Como programadores no tenemos que

preocuparnos si el equipo tiene uno o muchos núcleos, eso es trabajo del sistema operativo, nosotros simplemente disfrutamos de ejecutar "hilos concurrentemente" no nos importa cómo consigue la concurrencia el sistema operativo.

Si los programas son java, a la discusión anterior hay que añadirle que cada aplicación java se ejecuta en una instancia individual de la MVJ, es decir, la MVJ sería un proceso del sistema operativo, y los threads de la aplicación java son manejados por la MVJ y luego la MVJ se entiende con el sistema operativo.

Conclusión, lo que nos concierne a nosotros ahora es descomponer un programa en hilos para conseguir que sea más eficiente

Tutorial de oracle.
https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
Las explicaciones iniciales son francamente buenas. Incluimos algún ejemplo de dicho tutorial a continuación

Definir y arrancar(start) un hilo.
https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html
Dos formas: implementando Runnable o extendiendo a Thread. En ambos casos:

- hay que sobreescribir run()
- El hilo se arranca con start(), que a su vez se encarga de invocar a run(). Si invocamos directamente a run() se ejecutan sus instrucciones pero como un método normal, no como en un hilo.

Definir un hilo a partir de un objeto Runnable
Con Runnable.Observa cómo se crea el hilo. Al constructor de Thread se le pasa un objeto Runnable

```
class HelloRunnable implements Runnable {
   public void run() {
      System.out.println("Hello from a thread!");
   }
}

public class App {
   public static void main(String[] args) {
       new Thread(new HelloRunnable()).start();
   }
}
```

debes de tener claro que la instrucción del main,muy compacta, se corresponde con:

```
public static void main(String[] args) {
      HelloRunnable r= new HelloRunnable();
      Thread t = new Thread(r);
      t.start();
   }
```

Definir un hilo extendiendo a Thread

El constructor de Thread no necesita obligatoriamente un Runnable, hay una versión del constructor sin parámetros.

```
class HelloThread extends Thread {
   public void run() {
      System.out.println("Hello from a thread!");
   }
}

public class App {
   public static void main(String[] args) {
       new HelloThread().start();
   }
}
```

En el ejemplo anterior se aprecia que "desapareció del mapa" la interface Runnable, pero realmente la interface Runnable se sigue utilizando ya que la clase Thread implementa Runnable(ver API). De una forma u otra siempre va a haber un método run() que sobreescribe al run() de Runnable y encapsula el código que se lanza como un hilo.

En ambos casos hacemos:

1. las instrucciones que queremos que se ejecuten concurrentemente se escriben en el método run()
2. Hay que crear un objeto Thread que contiene de alguna manera ese run()
3. Hay que lanzar al objeto Thread con start()

La diferencia es como escribo el run() para que finalmente esté dentro del objeto Thread:

1. pasándole un objeto Runnable al constructor de Thread
2. o bien, extendiendo el Thread y sobrescribiendo run()

¿Qué forma de definir hilos es mejor?

Recuerda que en Java: una clase puede tener un extends como mucho pero varios implements. Es por esto que lo más habitual es utilizar el esquema de implementar Runnable, de esta manera la clase HelloRunnable del ejemplo se reserva la posibilidad de extender a otra clase por ejemplo podríamos hacer

```
public class HelloRunnable extends OtraClase implements Runnable
```

Pausar la ejecución con sleep()

https://docs.oracle.com/javase/tutorial/essential/concurrency/sleep.html

De la clase Thread ya vimos el método imprescindible start(). Otro método de esta clase es sleep() que detiene la ejecución del hilo los milisegundos que le indiquemos. sleep() es static y lanza *InterruptedException*

Ejemplo:

```
public class App {
   public static void main(String[] args) throws InterruptedException {
```

```
        String importantInfo[] = {
           "Mares eat oats",
           "Does eat oats",
           "Little lambs eat ivy",
           "A kid will eat ivy too"
        };

        for (int i = 0; i < importantInfo.length;i++) {
           //Pause for 4 seconds
           Thread.sleep(4000);
           //Print a message
           System.out.println(importantInfo[i]);
        }
    }
}
```

En este ejemplo sleep() lo usamos como método de clase, no como método de instancia.

Uno de los estados de un hilo es "bloqueado". El método sleep() hace que un hilo en ejecución pase a bloquearse durante los segundos indicados tras los cuales regresa al estado de "ejecución".

Ejemplo: extraído del libro *java a fondo (snazjdleder).*

```
class DemoThread extends Thread{
   private String nombre;
   public DemoThread(String nombre){
      this.nombre=nombre;
   }
   public void run(){
      try{
         int x=(int)(Math.random()*5000);
         Thread.sleep(x);
         System.out.println("Soy: "+nombre+"("+x+")");
      }catch(Exception ex){
         ex.printStackTrace();
      }
   }


}

class App{
   public static void main(String[] args) {
     DemoThread t1 = new DemoThread("Pedro");
     DemoThread t2 = new DemoThread("Pablo");
     DemoThread t3 = new DemoThread("Juan");

     t1.start();
     t2.start();
     t3.start();
   }
}
```

aunque t1, t2 y t3 son objetos hilos, si invocamos a run() directamente, los hilos no entran en su ciclo de vida y no es más que código que se ejecuta secuencialmente. Para

que se ejecuten concurrentemente es necesario iniciar su ciclo de vida con start(). Comprobamos esta afirmación sustituyendo en el código anterior

```
t1.start();
t2.start();
t3.start();
```

por

```
t1.run();
t2.run();
t3.run();
```

¡con run() la ejecución es secuencial no hay concurrencia!

Ejercicio U5_B9A_E1

Como ya sabemos si a un método le llega una excepción verificada tendrá que capturarla (catch) o bien propagarla (throws). En el ejemplo anterior run() captura la excepción y en este caso no es posible propagarla. Es decir no es posible escribir:

```
public void run() throws InterruptedException{

        int x=(int)(Math.random()*5000);
        Thread.sleep(x);
        System.out.println("Soy: "+nombre+"("+x+")");

   }
```

Comprueba que esto da error e ¡Intenta explicar porqué!

Ejercicio U5_B9A_E2

Escribir el ejemplo de pedro,pablo y juan pero ahora implementando Runnable

El hilo principal se llama por defecto main.

Al ejecutar un programa, automáticamente java crea un hilo. Es el hilo principal que contiene el método main(). A partir de este hilo, como ya sabemos podemos crear otros hilos. Para demostrar esto, podemos usar el método currentThread() que devuelve una referencia al Thread en ejecución. Probamos el siguiente ejemplo:

```
class App{
   public static void main(String[] args) throws Exception {
      Thread t1= Thread.currentThread();
      System.out.println(t1.getName());
      t1.setName("mi gran hilo");
      Thread t2= Thread.currentThread();
      System.out.println(t2.getName());
   }
}
```

Fin de la aplicación

Una aplicación se puede ver como un conjunto de hilos en ejecución. El primer hilo del que parten todos es el hilo main, que llamamos hilo principal, pero ojo, una cosa es que el hilo main termine de ejecutar sus instrucciones y otra que la aplicación termine. Al haber varios hilos en ejecución todos deben terminar para que la aplicación termine.

```
class HiloHijo extends Thread {

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("soy hilo hijo");
        }
    }
}

public class App  {

    public static void main(String[] args) {
        System.out.println("principio de main");
        new HiloHijo().start();
        System.out.println("fin  de main");
    }
}
```

Ejecuta y observa cómo se llega al final del main pero la aplicación no acaba hasta que acaba el hilo hijo

Ahora con System.exit(0), fuerzo la salida de la aplicación y no se espera al fin del hilo hijo. Habrá ejecuciones con 0 impresiones del hilo hijo, otras con 1, 2, … El tiempo que el procesador concede a cada hilo es variable según el estado del equipo, cuanto antes se llegue al exit(0) antes se acaba.

```
class HiloHijo extends Thread {

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("soy hilo hijo");
        }
    }
}

public class App {
    public static void main(String[] args) {
        System.out.println("principio de main");
        new HiloHijo().start();
        System.out.println("fin  de main");
        System.exit(0);
    }
}
```

Por lo tanto, una aplicación acaba cuando todos sus hilos acaban o se provoca una salida forzada con exit(0). A esto hay que añadir el matiz de "hilos demonio" que veremos más adelante.

Esperar a que finalice un Thread

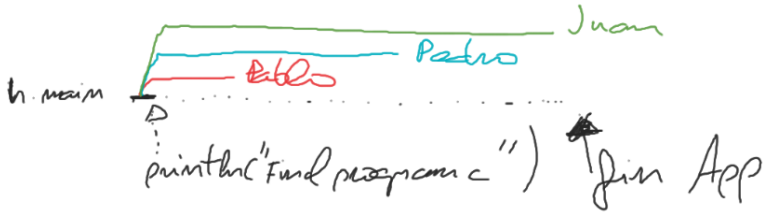
Como cada hilo avanza en paralelo a su ritmo muchas veces queremos que uno espere por otro. La forma de espera más básica se consigue con join().

Observa el siguiente código. Es una variante del primer ejemplo en el queríamos ejecutar los 3 hilos pero además ahora queremos que al final del programa se imprima "Final del programa!".

Observa que el método main() también forma parte de un hilo, del *Thread main*, que crea la MVJ para la aplicación. El Thread main también se ejecuta concurrentemente con t1, t2 y t3 y como no tiene sleep() siempre acaba primero.

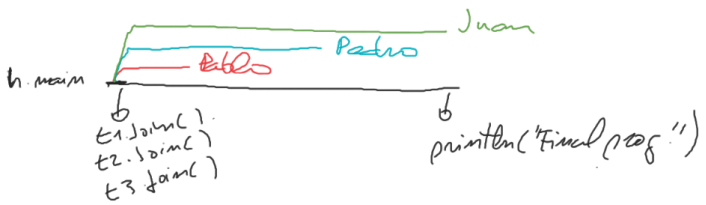
```
class DemoThread implements Runnable{
   private String nombre;
   public DemoThread(String nombre){
      this.nombre=nombre;
   }
   public void run(){
      try{
         int x=(int)(Math.random()*5000);
         Thread.sleep(x);
         System.out.println("Soy: "+nombre+"("+x+")");
      }catch(Exception ex){
         ex.printStackTrace();
      }
   }


}

class App{
   public static void main(String[] args) {
     Thread t1 = new Thread( new DemoThread("Pedro"));
     Thread t2 = new Thread( new DemoThread("Pabl0"));
     Thread t3 = new Thread( new DemoThread("Juan"));


     t1.start();
     t2.start();
     t3.start();

     //quería que esto se imprimiera cuando acabaran los hilos  pero se imprime antes de que acaben los hilos
     System.out.println("Final del programa!!");
   }
}
```

```
Final del programa!!
Soy: Pabl0(680)
Soy: Pedro(1445)
Soy: Juan(3299)
```



Para solucionar esto usamos el método join().

```
class DemoThread implements Runnable{
    private String nombre;
    public DemoThread(String nombre){
        this.nombre=nombre;
    }
    public void run(){
        try{
            int x=(int)(Math.random()*5000);
            Thread.sleep(x);
            System.out.println("Soy: "+nombre+"("+x+")");
        }catch(Exception ex){
            ex.printStackTrace();
        }
    }


}

class App{
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread( new DemoThread("Pedro"));
        Thread t2 = new Thread( new DemoThread("Pablo"));
        Thread t3 = new Thread( new DemoThread("Juan"));


        t1.start();
        t2.start();
        t3.start();

          //forzamos a esperar a que los 3 hilos acaben antes de imprimir Final del programa
        t1.join();//hasta que acaba t1 main() espera aquí
        t2.join();//hasta que acaba t2 main() espera aquí
        t3.join();//hasta que acaba t3 main() espera aquí



        //ahora si que se imprime al final
        System.out.println("Final del programa!!");
    }
}
```



join() obliga al Thread actual, es decir, el que contiene a la instrucción join(), en este caso el Thread main, a esperar. Observa este ejemplo:

Ejemplo: comenta las instrucciones t1.join() y t2.join() y observa que
si sólo hago t3.join() main sólo está obligado a esperar a que acabe t3 con lo cual el println("Final del programa!!") se puede hacer cuando acaba t3. Lo que no sabemos es si se va a hacer antes o después que acaben t1 y t2 pues con estos compite concurrentemente por el tiempo de CPU y depende del tiempo de sleep() de t1 y t2

```
//forzamos a esperar a que los 3 hilos acabe
//t1.join();
//t2.join();
t3.join();
System.out.println("Final del programa!!");

run:
Soy: Juan(198)
Final del programa!!
Soy: Pablo(838)
Soy: Pedro(4966)
```

Por cierto el método join() también nos obliga a tratar *InterruptedException* y en el código ejemplo anterior Y para simplificar la propagamos a MVJ

Thread en bucle infinito

Que un thread contenga un bucle infinito suele ser algo habitual, por ejemplo, un thread que reproduce contínuamente una canción como música de fondo.

Ejemplo: Ejecuta y entiende el siguiente ejemplo.
(adaptación de http://www.ctr.unican.es/asignaturas/procodis_3_ii/doc/procodis_3_01.pdf)

```
class PingPong extends Thread{
   private String word; // Lo que va a escribir.
   private int delay; // Tiempo entre escrituras
   public PingPong(String queDecir,int cadaCuantosMs){
      word = queDecir;
      delay = cadaCuantosMs;
   };
   public void run(){ //Se sobrescribe run() de Thread
      while(true){
         System.out.print(word + " ");
         try{
            sleep(delay);
         } catch(InterruptedException e){
         }
      }
   }
}
public class App{
   public static void main(String[] args) throws InterruptedException {
      // Declaración de 2 threads
      PingPong t1 =new PingPong("PING",100);
      PingPong t2= new PingPong("PONG",100);
      // Activación
      t1.start();
      t2.start();
      Thread.sleep(5000);
      System.exit(0);
   }
}
```

Observa que los dos hilos se ejecutan en un bucle infinito. El programa acaba ya que tras 5 seg(Thread.sleep(5000)) hacemos en main() System.exit(0);

Si varías el tiempo de espera de uno y de otro verás como uno escribe más que otro.

Si sustituyes
System.exit(0);
por
return;
El programa nunca acaba ya que aunque el método main() acabó su ejecución, el hilo que contiene a main(), el Thread main, tiene que esperar a que acaben dos hilos en ejecución. Con System.exit(0) se ordena que se finalice la aplicación saliendo de todo hilo

Ejercicio U5_B9A_E3
Modifica el ejemplo anterior de forma que en lugar de extender a Thread se implemente Runnable.

PARAR UN HILO

Es habitual que un hilo se ejecute en un bucle infinito y que cuando se dan determinadas condiciones queramos finalizarlo. En la clase Thread hay método stop() pero está deprecated(no se debe usar).

Ejemplo: para un hilo con stop(). No recomendado por ser deprecated()

```
class HiloHijo extends Thread {
    @Override
    public void run() {
        while(true){
            System.out.println("soy hilo hijo");
        }
    }
}

public class App extends Thread {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("principio de main");
        HiloHijo h1=new HiloHijo();
        h1.start();
        Thread.sleep(5);
        //despues de 5 milisegundos  quiero que pare el hilo
        h1.stop(); //no se debe usar
        System.out.println("otras cosas de main ");
    }
}
```

Ejemplo: parar un hilo finalizando la aplicación
Su funcionamiento es obvio ya que se finalizan la aplicación y se finalizan todos sus hilos. El problema de este método, es que puedo querer parar un hilo pero que la aplicación no termine.

```
class HiloHijo extends Thread {
    @Override
    public void run() {
        while(true){
            System.out.println("soy hilo hijo");
        }
    }
}

public class App extends Thread {
    public static void main(String[] args) throws InterruptedException {
```

```
        System.out.println("principio de main");
        new HiloHijo().start();
        Thread.sleep(5);
        //despues de 5 milisegundos  quiero que pale el hilo
        System.exit(0);//pero así se acaba aplicación
        System.out.println("otras cosas de main que no se harán");
    }
}
```

parar un hilo modificando su estado a través de una variable

Esta es la solución más flexible y habitual.

```
class HiloHijo extends Thread {
    private boolean parado=false;
    @Override
    public void run() {
        int i=0;
        while(!parado){
            System.out.println("soy hilo hijo, iteracion "+i);
            i++;
        }
    }
    public void pararHilo(){
        parado=true;
    }
}

public class App extends Thread {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("principio de main");
        HiloHijo h1=new HiloHijo();
        h1.start();
        Thread.sleep(500);
        //despues de 5 milisegundos  quiero que pale el hilo
        h1.pararHilo();
        System.out.println("otras cosas de main ");
    }
}
```

Recuerda que en POO los objetos intercambian información enviándose mensajes. Un mensaje en java es una llamada a un método. Los hilos son objetos. Por tanto el hilo main manda un mensaje al hilo *HiloHijo* a través del método pararHijo()

EXTENDS VS RUNNABLE

Fíjate que es un caso concreto de Herencia(extends) vs Composición (el objeto que queremos ejecutar en hilo se pasa por parámetro a Thread).

No hay duda al respecto: por varias razones es mejor el enfoque de implementar el interface. Aquí nos dan 5 razones contundentes :
http://javahungry.blogspot.com/2015/05/implements-runnable-vs-extends-thread-in-java-example.html

Lo más importante del artículo anterior: El uso de Extends debe reservarse para una relación clara padre/hijo en el que el hijo es una especialización clara del padre. Piensa en una clase *Circulo* que quieres ejecutar en hilo. Hay un sentido "semántico" evidente cuando decimos que Circulo es una especialización de Figura, pero decir que Circulo es una especialización de la clase Hilo es forzado. Es más claro decir
Thread t = new Thread(new Circulo());
que un objeto Thread encapsula un Circulo y lo ejecuta como un hilo

HILOS QUE SON DEMONIOS

El término demonio es diferente en el contexto de los sistemas operativos que el contexto de la programación multihilo

En el contexto de los sistemas operativos, un demonio o servicio es un programa que se ejecuta en segundo plano, fuera del control interactivo de los usuarios del sistema ya que carecen de interfaz con estos. El término demonio se usa fundamentalmente en sistemas UNIX y basados en UNIX, como GNU/Linux o Mac OS X. En Windows y otros sistemas operativos se denominan servicios porque fundamentalmente son programas que ofrecen servicios al resto del sistema.

En el contexto de la programación multihilo un demonio *d,* es un hilo tal que si el hilo que lo invocó acabó, se acaba también la ejecución de *d*. Para declarar un hilo demonio se usa el método setDaemon().

Ejemplo:

```
class DemoThread implements Runnable{
   private String nombre;
   public DemoThread(String nombre){
      this.nombre=nombre;
   }
   public void run(){
      try{
         int x=(int)(Math.random()*5000);
         Thread.sleep(x);
         System.out.println("Soy: "+nombre+"("+x+")");
      }catch(Exception ex){
         ex.printStackTrace();
      }
   }


}

class App{
   public static void main(String[] args) {
     Thread t1 = new Thread( new DemoThread("Pedro"));
     Thread t2 = new Thread( new DemoThread("Pablo"));
     Thread t3 = new Thread( new DemoThread("Juan"));
     //t1.setDaemon(true);
     //t2.setDaemon(true);
     //t3.setDaemon(true);

     t1.start();
     t2.start();
     t3.start();

     System.out.println("Fin código main");
   }
}
```

Como el método main no tiene retardos llega a su última instrucción antes que los hilos. Si descomentamos las instrucciones que declaran a los hilos como demonios observamos que a los hilos ya no les da tiempo a imprimir nada pues al acabar main se fuerza su finalización.

DEMONIOS EN EL API

en el API hay muchas clases que usan hilos y muchos de estos hilos son demonios. Por ejemplo observa cómo demuestro que la clase Clip efectivamente se ejecuta en un hilo pues no bloquea la secuencia del main() y al mismo tiempo demuestro que es un demonio ya que cuando acaba el main también se da por finalizado el hilo

```
import java.io.File;
import java.io.IOException;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
import javax.sound.sampled.LineUnavailableException;
import javax.sound.sampled.UnsupportedAudioFileException;



public class App {

    public static void main(String[] args) throws LineUnavailableException, UnsupportedAudioFileException, IOException,
InterruptedException {
        AudioInputStream audioIn = AudioSystem.getAudioInputStream(new File("micarro.wav"));
        Clip clip = AudioSystem.getClip();
        clip.open(audioIn);
        clip.start();
        System.out.println("start no bloquea  y al  clip ni le da tiempo a sonar");


    }

}
```

lo volvemos a comprobar ahora esperando 15 segundos pero vemos como se corta tras el sleep. Tenemos claro pues que es un hilo demonio pues cuando se acaba el hilo main se acaba el hilo clip.

```
import java.io.File;
import java.io.IOException;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
import javax.sound.sampled.LineUnavailableException;
import javax.sound.sampled.UnsupportedAudioFileException;



public class App {

    public static void main(String[] args) throws LineUnavailableException, UnsupportedAudioFileException, IOException,
InterruptedException {
        AudioInputStream audioIn = AudioSystem.getAudioInputStream(new File("micarro.wav"));
        Clip clip = AudioSystem.getClip();
        clip.open(audioIn);
        clip.start();
        Thread.sleep(15000);

    }

}
```