PSP -UT3:

Anexos

En este documento quiero explicar algunas características del lenguaje, ligadas a la concurrencia, que deberíamos conocer, pero que **no son imprescindibles ni evaluables en el módulo.**

Índice

JCONSOLE

Te permite visualizar los hilos que tiene un proceso, la memoria que ocupa etc.

Para hacerlo funcionar:

- Ejecuta JConsole (está desde la ver 1.5 en jdk/bin)
- En la parte de “Local Process” verás al menos 2 procesos (identificados por PID y opcionalmente por el nombre).
    - Conéctate al segundo proceso, que es del Netbeans, y haz click en “insecure connection”.
    - Nota: Si hay más de 2 comprueba el del Netbeans en el explorador de procesos de Windows (Dale a ver/añadir columna para sacar el PID).
- Ahora, si ejecutas tus programas en Netbeans podrás hacerles un seguimiento.

En este link tienes toda la información:
http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html

Esta herramienta se usa principalmente para monitorizar el rendimiento de los procesos ejecutados por servidores web.

Aquí tienes un artículo que te lo detalla: http://amunizmartin.wordpress.com/2009/06/14/monitorizacion-remota-en-java-jconsole/

BLOQUES VIGILADOS

En clase los usaremos continuamente por que son la forma más elemental y efectiva de garantizar el orden en la sincronización. En este apartado los definiré para su mejor comprensión.

Los guarded blocks o bloques de código vigilados son bloques controlados por una determinada condición. Tienen la característica de que hasta que la condición no se cumpla, no se ejecutará dicho bloque de código.

Nacen para evitar esto:

```
public void bloqueVigilado() {
        // No hacerlo así!
        while(!condicion) {} // Se detiene aquí,
        //comprobando iterativamente la condición
        System.out.println("La condición se ha cumplido");
}
```

Quiero dejar claro que esto es una bestialidad MUY ineficiente puesto que estaría ocupando la CPU durante la espera.

La forma correcta es invocar el método wait() bajo el while bloqueante.
Así el hilo se bloqueará hasta que otro hilo le haga una llamada a notify().

Sin embargo, hay que volver a hacer la comprobación de la condición, porque la notificación no significará necesariamente que se haya cumplido la condición. Por eso es necesario tener la llamada a wait() dentro de un bucle while que comprueba la condición.

```
public synchronized bloqueVigilado() {
        while(!condicion) {
        try {
                wait(); // desocupa la CPU
        } catch (InterruptedException e) {}
        }
        System.out.println("La condición se ha cumplido");
}
```

El bloque es synchronized porque las llamadas a los métodos wait() y notify(), por definición, siempre deben hacerse desde un bloque de código sincronizado.

REUTILIZACIÓN DE HILOS

From the Java API Specification for the Thread.start method:

> It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

So if you want a reusable "thread", you might do something like this:

```
Runnable myTask = new Runnable() {
    public void run() {
        // Do some task
    }
}

Thread t1 = new Thread(myTask);
t1.start();
t1.join();
Thread t2 = new Thread(myTask);
t2.start();
```

Or like this:

```
java.util.concurrent.ExecutorService threadPool = Executors.newFixedThreadPool(10);
threadPool.execute(new Runnable(){
                        public void run(){
                                    ///
                        }
                });
```

This will start a reusable thread, if you add more than 10 threads into the pool then the system will reuse the previously created thread (after it finishes executing).

Or like this:

```
Runnable doHelloWorld = new Runnable() {
    public void run() {
        // Put your UI update computations in here.
        // BTW - remember to restrict Swing calls to the AWT Event thread.
        System.out.println("Hello World on " + Thread.currentThread());
    }
};
SwingUtilities.invokeLater(doHelloWorld);
System.out.println("This might well be displayed before the other message.");
```

If you replace that println call with your computation, it might just be exactly what you need.
EDIT: following up on the comment, I hadn't noticed the Android tag in the original post. The equivalent to invokeLater in the Android work is Handler.post(Runnable).

LA INTERFAZ LOCK

El "lock" o cerrojo reentrante de Java es fácil de usar pero tiene muchas limitaciones.
Por eso el paquete java.util.concurrent.locks incluye una serie de utilidades relacionadas con lock. La interfaz más básica de éstas es Lock.

Los objetos cuyas clases implementan la interfaz Lock funcionan de manera muy similar a los locks implícitos que se utilizan en código sincronizado, de manera que sólo un hilo puede poseer el Lock al mismo tiempo.

La ventaja de los objetos Lock es que posibilitan rechazar un intento de adquirir el cerrojo.

El método tryLock() rechaza darnos el lock si éste no está disponible inmediatamente, o bien tras un tiempo máximo de espera, si se especifica así.

El método lockInterruptibly rechaza darnos el lock si otro hilo envía una interrupción antes de que el lock haya sido adquirido.

Un ejemplo de sincronización utilizando Lock en lugar de synchronized sería:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
//...
Lock l = new ReentrantLock();
l.lock();
try {
        // acceder al recurso protegido por l
} finally {
        l.unlock();
}
```

Los objetos Lock también dan soporte a un mecanismo de wait/notify a través de objetos Condition.

```
class BufferLimitado {
    final Lock lock = new ReentrantLock();
    //Dos condiciones para notificar a los hilos que deban hacer put o take, respectivamente
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
            notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
```

```
        }

        public Object take() throws InterruptedException {
                lock.lock();
                try {
                        while (count == 0)
                        notEmpty.await();
                        Object x = items[takeptr];
                        if (++takeptr == items.length)
                                takeptr = 0;
                        --count;
                        notFull.signal();
                        return x;
                } finally {
                        lock.unlock();
                }
        }
}
```

COLECCIONES CONCURRENTES

Java proporciona algunas estructuras con métodos sincronizados, como por ejemplo Vector.

A partie de la 1.5 Java también proporciona una serie de clases de colecciones que facilitan la concurrencia, y se encuentran en el paquete java.util.concurrent, como ya sabemos.

Se pueden clasificar según las interfaces que implementan:

- **BlockingQueue** define una estructura de datos FIFO que bloquea o establece un tiempo máximo de espera cuando se intenta añadir elementos a una cola llena o cuando se intenta obtener de una cola vacía.

- **ConcurrentMap** es una subinterfaz de java.util.Map que define operaciones atómicas útiles: por ejemplo eliminar una clave-valor sólo si la clave está presente, o añadir una clave valor sólo si la clave no está presente.
  Al ser operaciones atómicas, no es necesario añadir otros mecanismos de sincronización.
  La implementación concreta es la clase **ConcurrentHashMap**.

- La interfaz **ConcurrentNavigableMap** es para coincidencias aproximadas, con implementación concreta en la clase **ConcurrentSkipListMap**, que es el análogo concurrente de **TreeMap**.

ATOMIC ACCESS

In programming, an atomic action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

We have already seen that an increment expression, such as c++, does not describe an atomic action.

Even very simple expressions can define complex actions that can decompose into other actions. However, there are actions you can specify that are atomic:

- Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
- Reads and writes are atomic for all variables declared volatile (including long and double variables).

Atomic actions cannot be interleaved, so they can be used without fear of thread interference.
However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible.

Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable.

This means that changes to a volatile variable are always visible to other threads. What's more, it also means that when a thread reads a volatile variable, it sees not just the latest change to the volatile, but also the side effects of the code that led up the change.

Using simple atomic variable access is more efficient than accessing these variables through synchronized code, but requires more care by the programmer to avoid memory consistency errors. Whether the extra effort is worthwhile depends on the size and complexity of the application.

Some of the classes in the java.util.concurrent package provide atomic methods that do not rely on synchronization. We'll discuss them in the section on High Level Concurrency Objects.

ATOMIC VARIABLES

The java.util.concurrent.atomic package defines classes that support atomic operations on single variables.

All classes have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables.

To see how this package might be used, let's return to the Counter class we originally used to demonstrate thread interference:

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

One way to make Counter safe from thread interference is to make its methods synchronized, as in SynchronizedCounter:

```
class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

For this simple class, synchronization is an acceptable solution. But for a more complicated class, we might want to avoid the liveness impact of unnecessary synchronization.

Replacing the int field with an Atomic Integer allows us to prevent thread interference without resorting to synchronization, as in AtomicCounter:

```
import java.util.concurrent.atomic.AtomicInteger;
class AtomicCounter {
   private AtomicInteger c = new AtomicInteger(0);

   public void increment() {
      c.incrementAndGet();
   }

   public void decrement() {
      c.decrementAndGet();
   }

   public int value() {
      return c.get();
   }

}
```

VOLATILE

La keyword volatile de Java es seguramente la palabra reservada peor documentada, entendida y menos utilizada de entre todas las disponibles en Java.
Para inri, **en la versión 5**, en la que Java añade una nueva API (java.util.concurrent), **su significado cambia radicalmente**.

Lo primero que hay que entender es que **volatile** es, junto con synchronized, **uno de los mecanismos de sincronización básicos de Java**.
Se utiliza sobre los atributos de los objetos para indicar al compilador que es posible que dicho atributo vaya a ser modificado por varios threads de forma simultanea y asíncrona, y que no queremos guardar una copia local del valor para cada thread a modo de caché, sino que queremos que los valores de todos los threads estén sincronizados en todo momento, **asegurando así la visibilidad** del valor actualizado a costa de un pequeño impacto en el rendimiento (respecto a variables normales).

volatile es más simple y más sencillo que synchronized, lo que implica también un mejor rendimiento.
Sin embargo volatile, a diferencia de synchronized, no proporciona atomicidad, lo que puede hacer que sea más complicado de utilizar.
Una operación como el incremento, por ejemplo, no es atómica (El operador de incremento se divide en realidad en 3 instrucciones distintas: primero se lee la variable, después se incrementa, y por último se actualiza el valor) por lo que algo como lo siguiente podría causarnos problemas a pesar de que la variable sea volatile:

```
volatile int contador;
public void aumentar() {
  contador++;
}
```

En caso de que necesitemos atomicidad podemos recurrir a synchronized o a cosas más avanzadas, como las clases de atomización vistas en el punto anterior (solo a partir del API 1.5).

Para entenderlo mejor puedes leer este artículo:

> Declaring a **static** variable in Java, means that there will be only one copy, no matter how many objects of the class are created. The variable will be accessible even with no Objects created at all.
> However, threads may have locally cached values of it.
>
> When a variable is **volatile** and not **static**, there will be one variable for each Object. So, on the surface it seems there is no difference from a normal variable but totally different from static. However, even with Object fields, a thread may cache a variable value locally.
>
> This means that if two threads update a variable of the same Object concurrently, and the variable is not declared volatile, there could be a case in which one of the thread has in cache an old value.
> Even if you access a static value through multiple threads, each thread can have its local cached copy!
> To avoid this you can declare the variable as **static volatile** and this will force the thread to read each time the global value.

However, volatile is not a substitute for proper synchronisation! For instance:

```
private static volatile int counter = 0;

private void concurrentMethodWrong() {
    counter = counter + 5;
    //do something
    counter = counter - 5;
}
```

Executing concurrentMethodWrong concurrently many times may lead to a final value of counter different from zero! To solve the problem, you have to implement a lock:

```
private static final Object counterLock = new Object();

private static volatile int counter = 0;

private void concurrentMethodRight() {
    synchronized (counterLock) {
        counter = counter + 5;
    }
    //do something
    synchronized (counterLock) {
        counter = counter - 5;
    }
}
```

Or use the AtomicInteger class.

Java 5 definition of volatile

As of Java 5, accessing a volatile variable creates a memory barrier: it effectively synchronizes all cached copies of variables with main memory, just as entering or exiting a synchronized block that synchronizes on a given object. Generally, this doesn't have a big impact on the programmer, although it does occasionally make volatile a good option for safe object publication.

Perhaps more significant for most programmers is that as of Java 5, the VM exposes some additional functionality for accessing volatile variables:

- truly atomic get-and-set operations are permitted;
- an efficient means of accessing the nth element of a volatile array (and performing atomic get-and-set on the element) is provided.

RESUMEN

Se podría definir, a grandes rasgos, **una variable volatile como una variable independiente que usa una especie de semáforo interno** que garantiza la exclusión mutua **cada vez que se accede a su valor siempre y cuando las instrucciones sobre ella sean atómicas.**

Es decir, que sobre una instrucción como i++ no se garantiza su atomicidad.

Esto es, cuando quieras proteger sólo una variable primitiva compartida, sobre la que solo vas a hacer operaciones de asignación (como por ejemplo sobre un sw) te aconsejo que la precedas de volatile en lugar de controlarla a través de monitores o semáforos (más simple y efectivo).

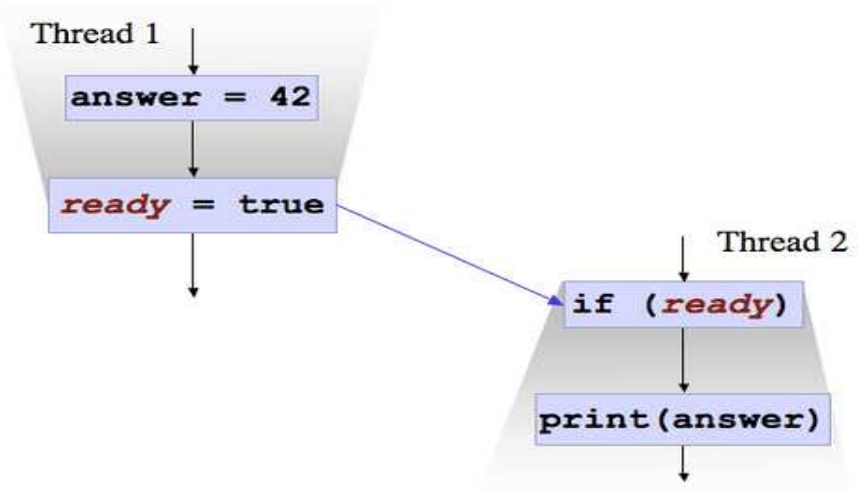
DETALLADO

First, you have to understand a little something about the Java memory model. I've struggled a bit over the years to explain it briefly and well. As of today, the best way I can think of to describe it is if you imagine it this way:

- Each thread in Java takes place in a separate memory space (this is clearly untrue, so bear with me on this one).
- You need to use special mechanisms to guarantee that communication happens between these threads, as you would on a message passing system.
- Memory writes that happen in one thread can "leak through" and be seen by another thread, but this is by no means guaranteed. Without explicit communication, you can't guarantee which writes get seen by other threads, or even the order in which they get seen.

The Java volatile modifier is an example of a special mechanism to guarantee that communication happens between threads. When one thread writes to a volatile variable, and another thread sees that write, the first thread is telling the second about all of the contents of memory up until it performed the write to that volatile variable.

At this point, I usually rely on a visual aid, which we call the "two cones" diagram, but which my officemate insists on calling the "two trapezoids" diagram, because he is picky. ready is a volatile boolean variable initialized to false, and answer is a non-volatile int variable initialized to 0.



The first thread writes to ready, which is going to be the sender side of the communications. The second thread reads from ready and sees the value the first thread wrote to it. It therefore becomes a receiver. Because this communication occurs, all of the memory contents seen by Thread 1, before it wrote to ready, must be visible to Thread 2, after it reads the value true for ready.

This guarantees that Thread 2 will print "42", if it prints anything at all.

If ready were not volatile, what would happen? Well, there wouldn't be anything explicitly communicating the values known by Thread 1 to Thread 2. As I pointed out before, the value written to the (now non-volatile) ready could "leak through" to Thread 2, so Thread 2 might see ready as true. However, the value for answer might **not** leak through. If the value for ready **does** leak through, and the value for answer **doesn't** leak through, then this execution will print out 0.

We call the communications points "happens-before" relationships, in the language of the Java memory model.

(Minor niggle: The read of ready doesn't just ensure that Thread 2 sees the contents of memory of Thread 1 up until it wrote to ready, it also ensures that Thread 2 sees the contents of memory of any other thread that wrote to ready up until that point.)

With this in mind, let's look at the Double-Checked Locking example again. To refresh your memory, it goes like this:

```
class Foo {
  private volatile Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      synchronized(this) {
        if (helper == null) {
          helper = new Helper();
        }
      }
    }
  return helper;
}
```

The object of the double-checked locking pattern is to avoid synchronization when reading a lazily constructed singleton that is shared between threads. If you have already constructed the object, the helper field will not be null, so you won't have to perform the synchronization.

However, this is only part of the solution. If one thread creates the object, it has to communicate the contents of its memory to another thread. Otherwise, the object will just sit in the first thread's memory. How do we communicate the contents of memory to another thread? Well, we can use volatile variables. That's why helper has to be volatile -- so that other threads see the fully constructed object.

Locking in Java also forms these "happens-before" communication points. An unlock is the sender side, and a lock on the same variable is the receiver side. The reason that doesn't work for (non-volatile) double-checked locking is that only the writing thread ever performs the locking. The whole point of the idiom is that the reader side doesn't do the locking. Without the explicit communication in the form of the volatile variable, the reading thread will never see the update performed by the writer thread.

Static vs Volatile

Static Variable: If two Threads(suppose t1 and t2) are accessing the same object and updating a variable which is declared as static then it means t1 and t2 can make their own local copy of the same object(including static variables) in their respective cache, so update made by t1 to the static variable in its local cache wont reflect in the static variable for t2 cache .

Static variables are used in the **Object Context** where update made by one object would reflect in all the other objects of the same class **but not in the Thread context** where update of one thread to the static variable will reflect the changes immediately to all the threads (in their local cache).

Volatile variable: If two Threads(suppose t1 and t2) are accessing the same object and updating a variable which is declared as volatile then it means t1 and t2 can make their own local cache of the Object **except the variable which is declared as a volatile** . So the volatile variable will have only one main copy which will be updated by different threads and update made by one thread to the volatile variable will immediately reflect to the other Thread.

Synchronized vs Volatile

Characteristic	Synchronized	Volatile
Type of variable	Object	Object or primitive
`Null` allowed?	No	Yes
Can block?	Yes	No
All cached variables synchronized on access?	Yes	From Java 5 onwards
When synchronization happens	When you explicitly enter/exit a synchronizedblock	Whenever a volatile variable is accessed.
Can be used to combined several operations into an atomic operation?	Yes	Pre-Java 5, no. Atomic get-set of volatiles possible in Java 5.

EJECUTORES

El manejo de la ejecución de hilos puede llevarse a cabo por el programador, o bien, **en aplicaciones más complejas, la creación y manejo de los hilos se pueden separar en clases especializadas**.
Estas clases se conocen como ejecutores, o Executors.

INTERFACES DE EXECUTOR

La **interfaz Executor** nos obliga a implementar un único método, **execute()**.

Si 'r' es un objeto Runnable y 'e' un objeto Executor, entonces en lugar de iniciar el hilo con (new Thread(r)).start(), lo iniciaremos con **e.execute(r)**.

De la segunda manera no sabemos si se creará un nuevo hilo para ejecutar el método run() del Runnable, o **si se reutilizará un hilo** "worker thread" que ejecuta distintas tareas. Es más probable lo segundo.

El Executor está diseñado para ser utilizado a través de las siguientes subinterfaces (aunque también se puede utilizar sólo):

ExecutorService

La interfaz ExecutorService es subinterfaz de la anterior y proporciona un método más versátil, **submit()** (significa enviar), que acepta objetos Runnable, pero también acepta objetos **Callable**, que permiten a una tarea devolver un valor.

El método **submit() devuelve un objeto Future** a través del cuál se obtiene el valor devuelto, y a través del cuál se obtiene el estado de la tarea a ejecutar.
También se permite el envío de colecciones de objetos Callable.

ExecutorService tiene métodos para para el ejecutor pero las tareas deben estar programadas manejar las interrupciones de manera adecuada (no capturarlas e ignorarlas).

ScheduledExecutorService
Esta interfaz es a su vez subinterfaz de la última, y aporta el método schedule() que ejecuta un objeto Runnable o Callable después de un retardo determinado.

Además define scheduleAtFixedRate y scheduleWithFixedDelay que ejecutan tareas de forma repetida a intervalos de tiempo determinados.

POOLS DE HILOS

La mayoría de implementaciones de java.util.concurrent utilizan pools de hilos que consisten en "worker threads" que existen de manera separada de los Runnables y Callables.

El uso de estos working thread minimiza la carga de CPU evitando creaciones de hilos nuevos.
La carga consiste sobre todo en liberación y reseva de memoria, ya que los hilos utilizan mucha.

Un tipo de pool común es el "**fixed thread pool**" que tiene un número prefijado de hilos en ejecución. Si un hilo acaba mientras todavía está en uso, éste es automáticamente reemplazado por otro.

Las tareas se envían al pool a través de una cola interna que mantiene las tareas extra que todavía no han podido entrar en un hilo de ejecución. De esta manera, si hubiera más tareas de lo normal, el número de hilos se mantendría fijo sin degradar el uso de CPU, aunque lógicamente, habrá tareas en espera y eso podrá repercutir, dependiendo de la aplicación.

Una manera sencilla de crear un ejecutor que utiliza un fixed thread pool es invocando el método estático **newFixedThreadPool (int nThreads)** de la clase **java.util.concurrent.Executors**.

Esta misma clase también tiene los métodos **newCachedThreadPool** que crea un ejecutor con un pool de hilos ampliable, y el método **newSingleThreadExecutor** que crea un ejecutor que ejecuta una única tarea al mismo tiempo.

Alternativamente se pueden crear instancias de **java.util.concurrent.ThreadPoolExecutor** o de **java.util.concurrent.ScheduledThreadPoolExecutor** que cuentan con más opciones.

Ejemplo de cómo crear un una instancia de java.util.concurrent.ThreadPoolExecutor:

```
int poolSize = 2;
int maxPoolSize = 2;
long keepAliveTime = 10;
final ArrayBlockingQueue<Runnable> queue = new ArrayBlockingQueue<Runnable>(5);
ThreadPoolExecutor threadPool = new ThreadPoolExecutor(poolSize, maxPoolSize,
                                                       keepAliveTime, TimeUnit.SECONDS, queue);
Runnable myTasks[3] = ....; // y le asignamos tareas

//Poner a ejecutar dos tareas y una que quedará en cola:
for(int i=0; i<3; i++){
        threadPool.execute(task);
        System.out.println("Tareas:" + queue.size());
}

//Encolar otra tarea más que declaramos aquí mismo:
threadPool.execute( new Runnable() {
                        public void run() {
                                for (int i = 0; i < 10; i++) {
                                        try {
                                                System.out.println("i = " + i);
                                                Thread.sleep(1000);
                                        } catch (InterruptedException ie){ }
                                }//for
                            }//run
                        });
//Ejecuta las tareas que queden pero ya no acepta nuevas:
threadPool.shutdown();
```

También es frecuente sobrecargar ThreadPoolExecutor para añadir algún comportamiento adicional. Por ejemplo, hacer que sea pausable:

```
class PausableThreadPoolExecutor extends ThreadPoolExecutor {

        private boolean isPaused;
        private ReentrantLock pauseLock = new ReentrantLock();
        private Condition unpaused = pauseLock.newCondition();

        //Constructor: utilizamos el del padre
        public PausableThreadPoolExecutor(...) { super(...); }

        //Sobrecargamos el método:
        protected void beforeExecute(Thread t, Runnable r) {
                super.beforeExecute(t, r);
                pauseLock.lock(); //Sección sincronizada
                try {
                        while (isPaused) unpaused.await(); //Bloquearlo
                } catch (InterruptedException ie) {
                        t.interrupt();
                } finally {
                        pauseLock.unlock();
                }
        }
        //Método nuevo:
```

```
        public void pause() {
                pauseLock.lock(); //Sección sincronizada
                try {
                        isPaused = true;
                } finally {
                        pauseLock.unlock();
                }
        }
        //Método nuevo:
        public void resume() {
                pauseLock.lock(); //Sección sincronizada
                try {
                        isPaused = false;
                        unpaused.signalAll(); //Desbloquear hilos bloqueados
                } finally {
                        pauseLock.unlock();
                }
        }
}//Class
```

Nota que en el anterior código se pausa la ejecución de nuevas tareas pero no se pausan las que ya están ejecutándose. El problema aquí es que cada hilo debe comprobar por si mismo si debe seguir ejecutándose o no. Es decir, es responsabilidad del programador programar un mecanismo de pausado en sus hilos.

BIBLIOGRAFÍA

- https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- http://jeremymanson.blogspot.com.es/2008/11/what-volatile-means-in-java.html