Escuela Politécnica Nacional
Facultad de Ingeniería de Sistemas

Asignatura: Oracle Certified Java Programmer

Grupo: #7

Fecha: 28 de enero de 2017

Tema: Threads

Integrantes:
Cristhian Motoche
Evelin Zuñiga
Javier Utreras

Introducción

“El software se ralentiza más deprisa de lo que se acelera el hardware.” Frase que se conoce como **ley de Wirth** la proposición enunciada por Niklaus Wirth en 1995

El hardware se está volviendo, claramente, más rápido a medida que pasa el tiempo y parte de ese desarrollo está cuantificado por la Ley de Moore. Los programas tienden a hacerse más grandes y complicados con el paso del tiempo y a veces los programadores se refieren a la Ley de Moore para justificar la escritura de código lento o no optimizado, pensando que no será un problema porque el hardware sobre el que correrá el programa será cada vez más rápido.

Un ejemplo de la Ley de Wirth que se puede observar es que el tiempo que le toma a un PC actual arrancar su sistema operativo no es menor al que le tomaría a un PC de hace cinco o diez años con un sistema operativo de la época.

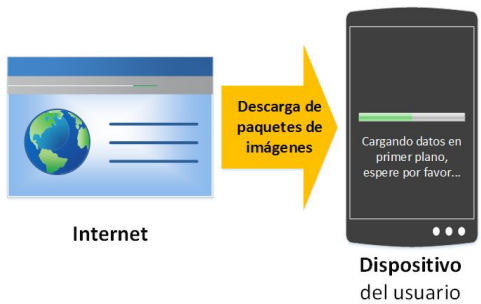
¿Para que la necesitamos?

Con la multitarea el usuario nunca quedará bloqueado -pudiendo seguir usando la aplicación mientras algo muy gordo se ejecuta debajo; se eliminan la mayoría de pantallas de carga o no interrumpirán la experiencia de uso de la aplicación; y se ejecutará de manera más óptima, haciendo que el procesador no esté esperando continuamente y con cuellos de botella por llegarle un montón de cosas a la vez. Claro está, si se hace bien.

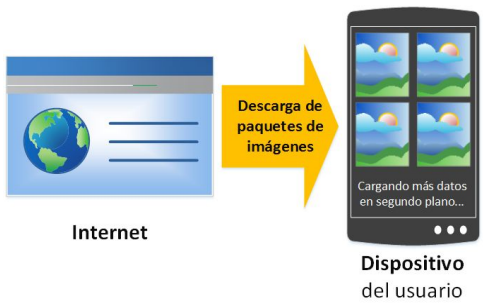
Que es un Hilo?

El concepto de Hilos (en inglés Thread) nos ayuda. Un Hilo es un trozo de código de nuestro programa que puede ser ejecutado al mismo tiempo que otro. Vamos a poner el siguiente ejemplo, imagina que queremos ver un listado de 100 imágenes que se descargan desde Internet, como usuario ¿Cuál de las dos opciones siguientes elegirías?:

A) Descargar las imágenes 100 imágenes, haciendo esperar al usuario con una pantalla de “cargando” hasta que se descargan todas. Luego podrá ver el listado con las imágenes.



B) Que mientras se descargan las 100 imágenes, el usuario pueda ir viendo y usando las que ya se han descargado.



Como desarrollador la opción A es más sencilla pero la B es lo que todo usuario quiere de una aplicación, tener que esperar no es una opción.

PLANOS DE EJECUCIÓN

Primer Plano

Aquí se ejecuta únicamente un hilo llamado “hilo principal”. Aquí se programa siempre, sin conocimiento de que estábamos trabajando ya con hilos. Es el hilo que trabaja con las vistas, es decir, con la interfaz gráfica que ve el usuario: botones, ventanas emergentes, campos editables, etc. También, puede ser usado para hacer cálculos u otros procesamientos complejos El primer plano influirá en la felicidad del usuario. Aquí es donde el usuario interacciona de manera directa, además todo lo que pase aquí lo ve y lo siente. La mala gestión del primer plano por parte del desarrollador, será castigada por el sistema operativo

Segundo plano

Se ejecuta todo el resto de hilos. El segundo plano tiene la característica de darse en el mismo momento que el primer plano. Aquí los hilos deberían de llevar las ejecuciones pesadas de la aplicación. El segundo plano el usuario no lo ve, es más, ni le interesa, para el usuario no existe. Por lo que comprenderás, que el desarrollador puede moverse libremente por este segundo plano.

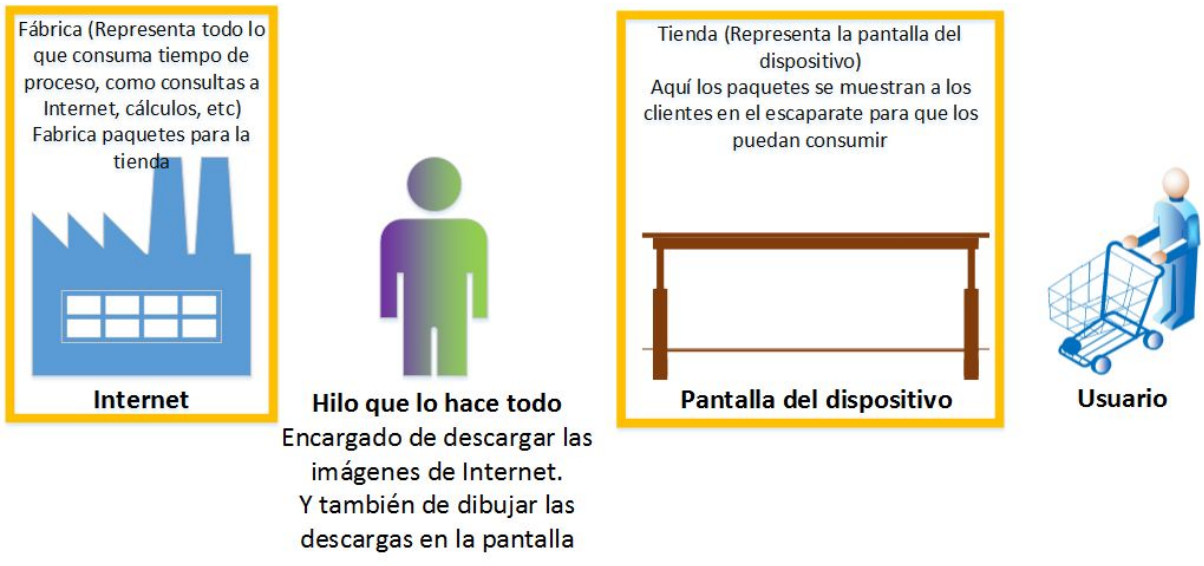
Aquí el desarrollador se puede resarcir y hacer que, por ejemplo, un método que tarde horas en ejecutarse, ya que el usuario ni lo sentirá.

Se debe notar una cosa que suele suscitar dudas: **no confundamos el término proceso con hilo.** Un proceso es el programa o aplicación en ejecución (Extiendo un poco más para que queden claras las diferencias. Lo que llamamos aplicación es el fichero ejecutable almacenado en memoria. Varios procesos pueden ejecutar varias instancias del mismo programa, es decir, como cuando se abren varias ventanas de un Bloc de notas o un Word). Así, se deduce y es verdad que un proceso contiene un hilo, mínimo el hilo principal que corre en primer plano o varios hilos, El principal más algunos en segundo plano.

Vamos a ver un ejemplo para hilos retornando al ejemplo anterior: un listado de imágenes que se descargan desde Internet. Para hacerlo más cómodo y comprensible, vamos a ejemplificarlo con personas. Una persona es un hilo, como el siguiente:

Hilo que lo hace todo
encargado de descargar
los datos de Internet
Y también encargado
dibujar las descargas en
la pantalla

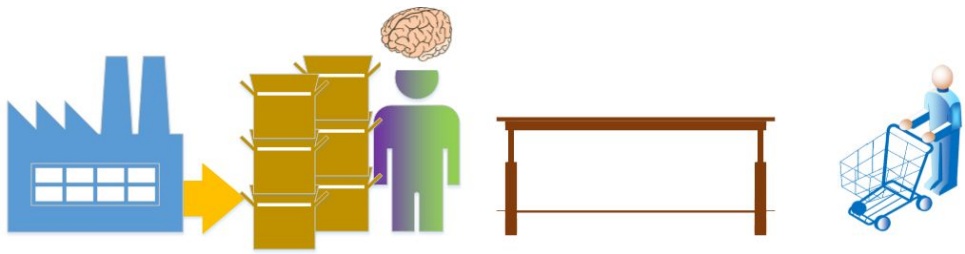
Este trabajador (nuestro hilo), hace su trabajo de la mejor manera que sabe, pero tiene demasiadas responsabilidades. Para lo que le han contratado es para llevar paquetes desde una fábrica hasta el repisade una tienda. Además, este trabajador/hilo es el hilo principal de la aplicación.



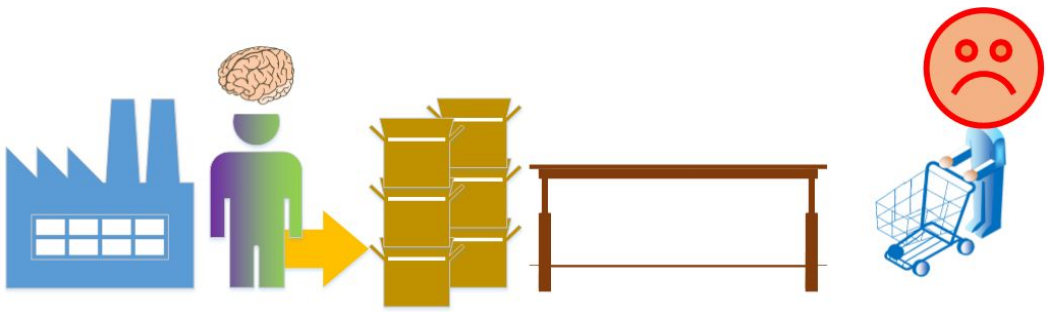
De todo el proceso de este trabajador/hilo, con sus correspondientes equiparaciones en la vida de un programa.. A continuación el ejemplo,

Ejemplo sin concurrencia ni paralelismo, es decir, sin multitarea

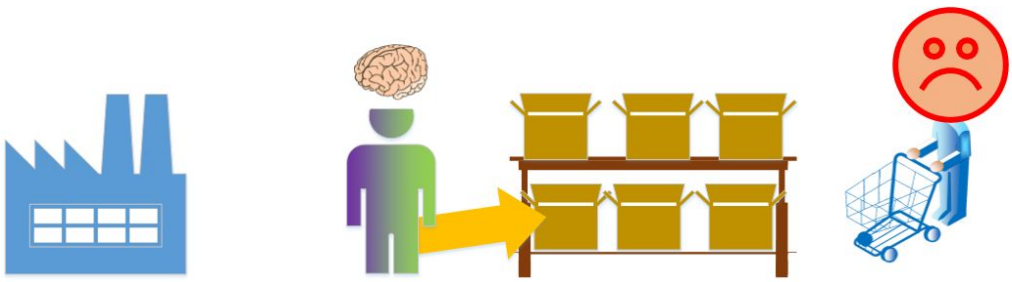
1) Primero busca todos los paquetes de datos con las “imágenes” de una fábrica que se llama “Internet”.



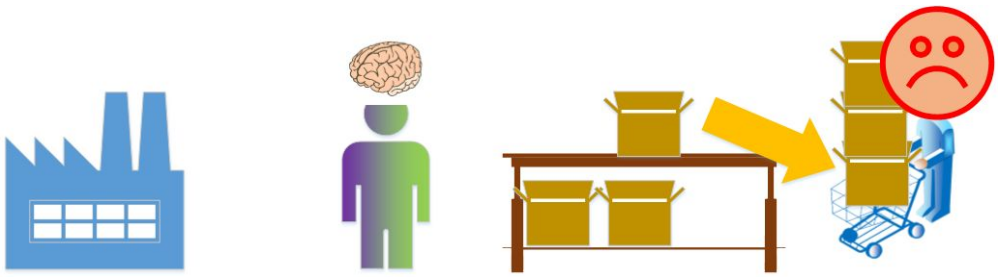
2) Lleva todos estos paquetes en camión y los descarga a una tienda llamada “dispositivo del usuario”.



3) Luego le toca colocar todo y cada uno de los paquetes que ha transportado al repisa de la tienda, que para aclarar también podemos llamar como “pantalla del dispositivo”.

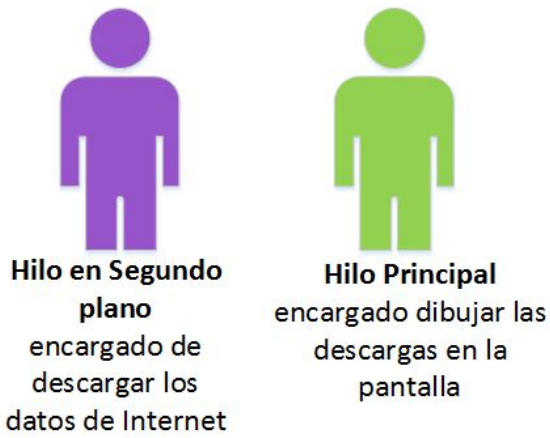


4) Todo esto se hace con la finalidad de que el cliente o “usuario” los consuma. Eso sí, solo en este momento se le permite al usuario poder consumirlos , ya que es el momento que tiene algo que consumir, antes no había nada- con lo que el resto del tiempo estuvo esperando.



Si, el usuario lleva bastante tiempo enfadado con nuestra aplicación. Ha tenido que esperar demasiado y además cuando no hacía falta. Un usuario descontento se traducirá en menos estrellas en nuestra aplicación, en malas críticas, o en otras maneras de expresar una opinión negativa hacia nuestro programa.

Seguro que te has fijado en el doble color de nuestro trabajador/hilo que es un hilo principal todo él. Tiene dos colores para representar la gran carga de trabajo que tiene este hilo. Habrá que investigar si se puede hacer de él dos hilos diferentes, pues es algo que podría hacerse en dos hilos diferentes, uno principal el encargado de dibujar en pantalla y otro en segundo plano encargado de descargar los datos desde Internet.

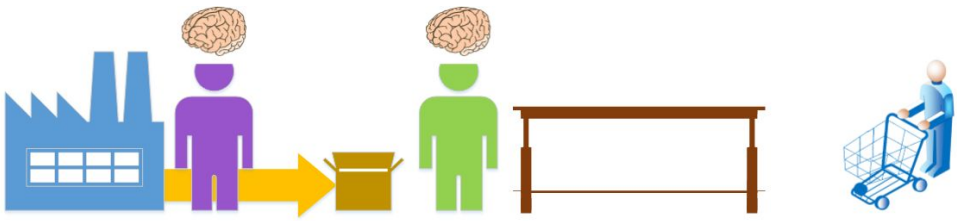


Teniendo dos trabajadores/hilos es lógico pensar que todo vaya mejor y más rápido. Dispondremos de un trabajador/hilo principal cuya dedicación es exclusiva para el usuario. Y otro hilo/trabajador en segundo plano, que se dedica a descargar los paquetes de datos, cuya función no la siente el usuario y nunca sabrá que está ahí trabajando para él.

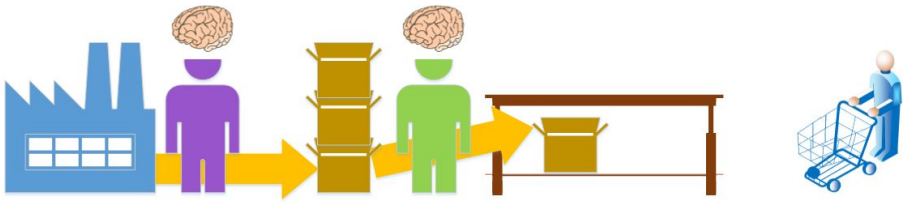
Ejemplo de concurrencia y paralelismo, la multitarea en todo su esplendor

1) Para empezar, el trabajador/hilo en segundo plano toma algún paquete de datos de la fábrica que llamamos “Internet”, lo lleva en camión y lo descarga en la tienda. Donde el

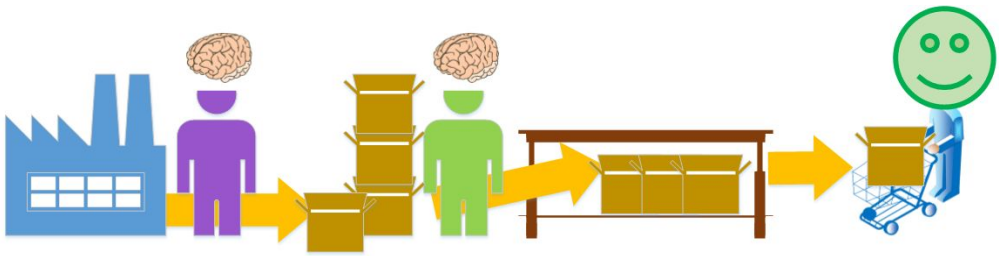
trabajador/hilo principal está esperando a que le llegue algún paquete para poder realizar su labor.



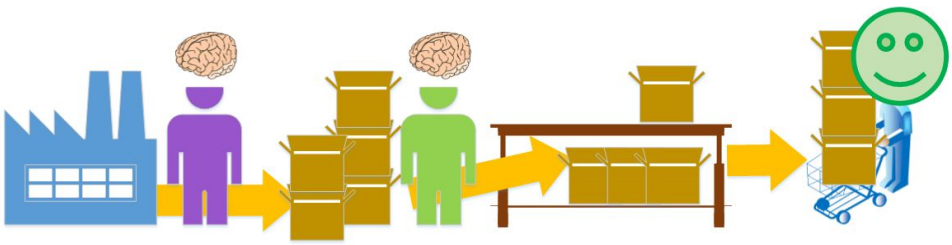
2) El trabajador/hilo principal ya tiene algo que hacer. Toma el paquete que le ha llegado y lo coloca en el repisa de la tienda, el trabajador/hilo en segundo plano no ha terminado su trabajo, con lo que continúa llevando paquetes a la tienda desde la fábrica. El cliente/usuario ya tiene al menos un paquete que consumir en la tienda.



3) El proceso continúa. El trabajador/hilo en segundo plano lleva a la tienda los paquetes, para que el trabajador/hilo en primer plano los coloque en la repisa. Por lo que, el cliente/usuario puede ir consumiendo cuanto desee de lo que exista ya en la repisa.



4) Esto durará hasta que se cumpla alguna condición de finalización o interrupción de algún trabajador/hilo, las cuales pueden ser muchas (por ejemplo: apagar el dispositivo, que se hayan acabado los paquetes que descargar, etc).

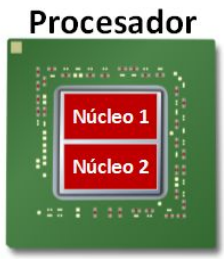


Con los hilos aparece la concurrencia y con ello se posibilita la programación en paralelo

En primer ejemplo de la fábrica y la tienda en el que solo había un hilo principal que lo hace todo. En este ejemplo tenemos que todo se hace en un tiempo X que se estima largo. Podemos representar por la siguiente barra de tiempo desde el principio de la ejecución del hilo hasta el fin:

En comparación con este segundo ejemplo, en el que ya tenemos dos hilos, uno principal y otro secundario, que trabajan de manera simultánea. Es deducible que el tiempo que tarda en hacerse todo el proceso es bastante menor, ya que se sobreexponen los tiempos de ambos hilos. En la siguiente representación vemos como el hilo principal se ejecuta de manera “paralela” al hilo en segundo plano y que es menos a la anterior barra de tiempo:

Esto nos concluye en la definición de paralelo: dos o más hilos que se ejecutan de manera simultánea.
Depende de un factor delimitador más a tener en cuenta, que aunque no lo controla el desarrollador de aplicaciones existe siempre, es más, al desarrollador de aplicaciones no le importa demasiado, pero conocerlo explica muchas cosas y es muy útil para entender bastante.
Los núcleos de un procesador, la mayoría de las aplicaciones no los aprovechan porque los desarrolladores de aplicaciones no los han tenido en cuenta, no han sabido aprovechar la tecnología. Bueno, sabiendo que un procesador es el encargado de ejecutar los programas. Y que un procesador puede tener entre un núcleo y muchos.

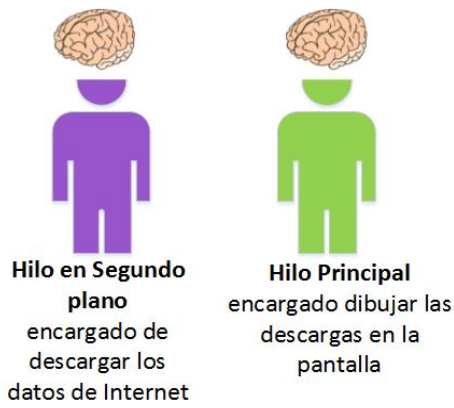


Ejemplo de 1 procesador
que contiene 2 núcleos

Cada núcleo ejecuta un hilo a la vez. Por lo que si tenemos un procesador con dos núcleos, podremos ejecutar dos hilos a la vez. Si tenemos uno de cuatro núcleos podrá ejecutar un máximo de cuatro hilos de manera simultánea. Y así con todos los núcleos de los que dispongamos.
Para seguir con el ejemplo, con los trabajadores que son hilos, el núcleo sería el cerebro de cada trabajador/hilo. Un trabajador/hilo siempre tendrá una misión que hacer determinada, pero sin un cerebro/núcleo no puede hacer nada.
Se repasa el primer ejemplo de la fábrica y la tienda con un solo trabajador/hilo, con un cerebro/núcleo, le es más que suficiente para ejecutar todo de manera correcta. Le sobraría con un procesador con un núcleo, pues más núcleos no los aprovecharía.

Por lo que un cerebro/núcleo procesa el trabajo tan rápido como pueda en el tiempo y para representar el tiempo que tarda este en procesarlo todo.

El segundo ejemplo es más complejo. Estamos suponiendo que tenemos dos núcleos. Si lo volvemos a repasar, fijándonos en los cerebros/núcleos, podemos llegar a la conclusión que existen dos núcleos, uno por cada hilo que se ejecuta.



Así, con dos cerebros/núcleos se puede trabajar en paralelo como indicamos anteriormente.

Pero también se puede dar lo siguiente con el segundo ejemplo ¿Qué ocurriría si tenemos un procesador con un único núcleo? ¿Qué hilo se ejecutaría? ¿Uno primero y otro después? ¿Dará error la aplicación? ¿Qué hará?

Es evidente que no hay suficientes cerebros/núcleos para tantos trabajadores/hilos. En algún momento uno de ellos no tendrá cerebro/núcleo y se tendrá que quedar detenido completamente mientras el otro con cerebro/núcleo realiza sus labores. Lo razonable y justo es que compartan el cerebro/núcleo un rato cada uno para poder trabajar los dos.

Y así es como se hace. El procesador se encarga de decidir cuánto tiempo va a usar cada hilo el único núcleo que existe. Dicho de otro modo, el procesador partirá los hilos en cachos muy pequeños y los juntará saltando en una línea de tiempo. Al procesarse los dos hilos de manera saltada, y gracias a la gran velocidad con la que procesan datos los procesadores hoy día, dará la sensación de paralelismo, pero no es paralelismo. Si nos fijamos en la siguiente línea de tiempo, la longitud es tan larga como si lo hiciéramos todo en un hilo.

No es paralelo pero sí es concurrente. La definición de concurrencia: dos o más componentes entre ellos independientes (por ejemplo, hilos con funcionalidades separadas)

se ayudan para solucionar un problema común (en estos ejemplos, los hilos trabajan juntos para mostrar al usuario unos paquetes descargados de Internet). Lo concurrente se recomienda ejecutarse en paralelo para sacar el máximo partido de la concurrencia, pero como vemos no es obligatorio. la pregunta ha realizarse es ¿Si no existen varios núcleos en el procesador, entonces no tiene ninguna ventaja haber desarrollado la aplicación con hilos? Pero sí las tiene. Si lo ejecutamos todo en el hilo principal, siempre se va a bloquear al usuario. Pero si lo ejecutamos con hilos, aunque solo dispongamos de un núcleo, habrá en algún instante en el que el usuario tenga con qué actuar, y el procesador ya se encargará de darle preferencia al usuario para no bloquear.

A veces necesitamos que nuestro programa **Java** realice varias cosas simultáneamente. Otras veces tiene que realizar una tarea muy pesada, por ejemplo, consultar en la lista telefónica todos los nombres de chicas que tengan la letra n, que tarda mucho y no deseamos que todo se quede parado mientras se realiza dicha tarea. Para conseguir que **Java** haga varias cosas a la vez o que el programa no se quede parado mientras realiza una tarea compleja, tenemos los **hilos** (**Threads**) (Abellan, 2017).

Un hilo es un proceso que se está ejecutando en un momento determinado en nuestro sistema operativo, como cualquier otra tarea, esto se realiza directamente en el procesador. Existen los llamados 'demonios' que son los procesos que define el sistema en sí para poder funcionar y otros que llamaremos los hilos definidos por el usuario o por el programador, estos últimos son procesos a los que el programador define un comportamiento e inicia en un momento específico.

En Java, el proceso que siempre se ejecuta es el llamado ***main*** que es a partir del cual se inicia prácticamente todo el comportamiento de nuestra aplicación, y en ocasiones a la aplicación le basta con este solo proceso para funcionar de manera adecuada, sin embargo, existen algunas aplicaciones que requieren más de un proceso (o hilo) ejecutándose al mismo tiempo (multithreading) (Navarro, 2017).

Marco teórico

Hilos (Threads)

Definición

Un hilo o subproceso en Java comienza con una instancia de la clase **java.lang.Thread**, si analizamos la estructura de dicha clase podremos encontrar bastantes métodos que nos ayudan a controlar el comportamiento de los hilos, desde crear un hilo, iniciarlo, pausar su ejecución, etc.

a) Los métodos propios de la clase **Thread** son:
   1) start()
   2) notify()
   3) sleep()
   4) run()

5) join()

b) Los objetos Thread se puede crear extendiendo de la clase **Thread** y sobrescribiendo el método: **public void run().**

c) También los objetos **Thread** pueden ser creados implementando la interface **Runnable**.

Para definir e instanciar un nuevo Thread (hilo o subproceso) existen 2 formas:

- Extendiendo (o heredando) a la clase **java.lang.Thread**
- Implementando la interfaz **Runnable**

Uso de Subclase
Cuando se crea una subclase de Thread, la subclase debería definir su propio método run() para sobremontar el métoddo run() de la clase Thread. La tarea concurrente es desarrollada en este método run().

Ejecución del método run()
Una instancia de la subclase es creada con new, luego llamamos al método start() de la thread para hacer que la máquina virtual Java ejecute el método run().

Implementación de la Interface Runnable
La interfaz Runnable requiere que sólo un método sea implementado, el método run(). Primero creamos una instancia de esta clase con new, luego creamos una instancia de Thread con otra sentencia new y usamos nuestra clase en el constructor. Finalmente, llamamos el método start() de la instancia de Thread para iniciar la tarea definida en el

método run().

Instanciación

Debemos recordar que cada hilo de ejecución es una instancia de la clase **Thread**, independientemente de si tu método **run()** está dentro de una subclase de **Thread** o en una implementación de la interfaz **Runnable**, se necesita un objeto tipo **Thread** para realizar el trabajo.

Si has extendido la clase Thread, el instanciar el hilo es realmente simple:

```
MiHilo h = new MiHilo();
```

Si has implementado la interfaz Runnable, es un poquito más complicado, pero solo un poco:

```
MiHilo h = new MiHilo();
Thread t = new Thread(h); // Pasas tu implementación de Runnable
al nuevo Thread
```

MÉTODOS O EJECUCIONES DE LOS HILOS

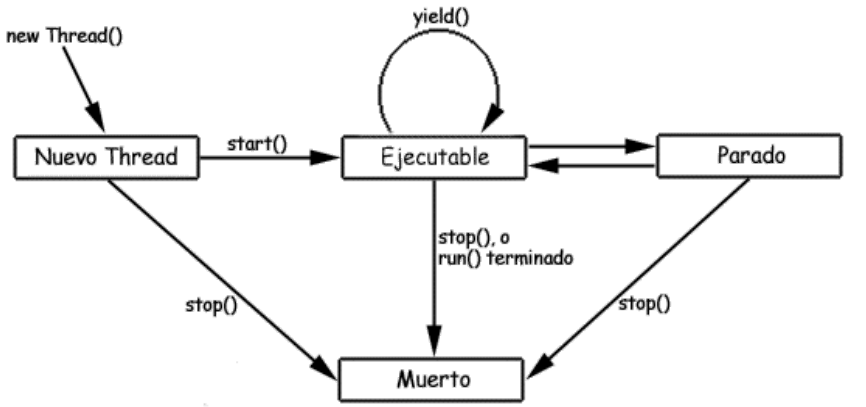
Métodos

Es la clase que encapsula todo el control necesario sobre los hilos de ejecución (threads). Hay que distinguir claramente un objeto Thread de un hilo de ejecución o thread. Esta distinción resulta complicada, aunque se puede simplificar si se considera al objeto Thread como el panel de control de un hilo de ejecución (thread). La clase Thread es la única forma de controlar el comportamiento de los hilos y para ello se sirve de los métodos comunes entre los que están

void start():
Usado para iniciar el cuerpo de la thread definido por el método run().
void sleep():
Pone a dormir una thread por un tiempo mínimo especificado.
void join():
Usado para esperar por el término de la thread, por ejemplo por término de método run().
void yield():
Mueve a la thread desde el estado de corriendo al final de la cola de procesos en espera por la CPU.
Notify() despierta un hilo

Estados de los Hilos

Durante el ciclo de vida de un hilo, éste se puede encontrar en diferentes estados. La figura siguiente muestra estos estados y los métodos que provocan el paso de un estado a otro.



Nuevo Thread

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

```
Thread MiThread = new MiClaseThread();
Thread MiThread = new Thread( new UnaClaseThread,"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.

Ejecutable

Ahora obsérvense las dos líneas de código que se presentan a continuación:

```
Thread MiThread = new MiClaseThread();
MiThread.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.

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 o 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 MiThread = new MiClaseThread();
MiThread.start();
try {
      MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
}
```

la línea de código que llama al método sleep():

```
      MiThread.sleep( 10000 );
```

hace que el hilo se duerma durante 10 segundos. Durante ese tiempo, incluso aunque el procesador estuviese totalmente libre, MiThread no correría. Después de esos 10 segundos. MiThread 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()

Muerto

Un hilo de ejecución se puede morir de dos formas: por causas naturales o porque lo maten (con stop()). 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 MiThread = new MiClaseThread();
MiThread.start();

try {
       MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
}

MiThread.stop();
```

se crea y arranca el hilo MiThread, 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.

EJERCICIO

```
package estadohilo;

import java.util.logging.Level;
```

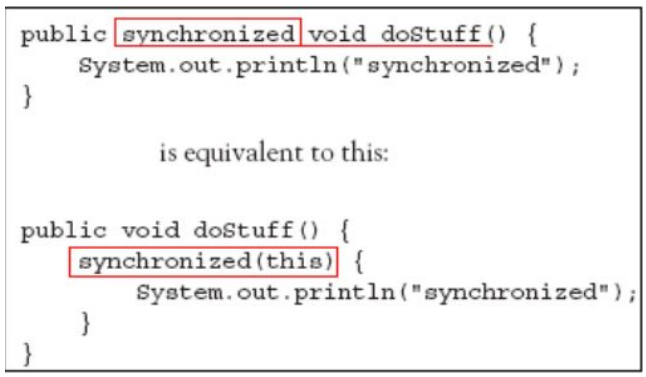
```
import java.util.logging.Logger;


public class HiloEjemplo extends Thread {
    public HiloEjemplo(String nombre){
        super(nombre);
    }
    public void run(){
        for (int i = 0; i < 10; i++) {
            System.out.println(i+"  "+getName());

            try {
                Thread.sleep(2000);
            } catch (InterruptedException ex) {

Logger.getLogger(HiloEjemplo.class.getName()).log(Level.SEVERE, null, ex);
            }

        }
        System.out.println("Finaliza "+getName());


    }

}




public class Principal {


    public static void main(String[] args) {
        Thread miHilo1 = new HiloEjemplo("Hilo Uno");
        Thread miHilo2 = new HiloEjemplo("Hilo Dos");
        miHilo1.start();
        miHilo2.start();
        System.out.println("Fin del Proceso");



    }

}
```

Sincronización e Interacción entre hilos

¿Qué podría ocurrir si dos hilos diferentes llaman a un método que altera el estado del objeto (por ejemplo, un *set*) al mismo tiempo? Un caso como este podría dejar al objeto en un estado inconsistente y si ese objeto es usado por otras partes en el código, sería un total desastre. Para solucionar este tipo de problemas se debe garantizar que siempre se realice una operación, esto se le conoce como una “operación atómica”. Y se llama así porque la operación, a pesar del número de sentencias actuales, es completada **antes** de que cualquier otro hilo pueda actuar sobre sobre los mismo datos.

Sincronización y bloqueos

¿Cómo funciona la sincronización? Con bloqueos. Cada objeto en Java tiene un bloqueo propio que solo funciona cuando el objeto tiene código de un método sincronizado. Cuando se ingresa a un método *no-estático sincronizado*, automáticamente se adquiere el bloqueo con la instancia actual de la clase cuyo código se está ejecutando (la instancia *this*).



Los siguientes puntos son importantes con respecto al bloqueo y sincronización:

- Solo método (o bloques) pueden ser sincronizados, no variables ni clases.
- Cada clase solo tiene un bloqueo.
- Una clase puede tener método sincronizados y no sincronizados. Pero solo los métodos sincronizados podrán tener bloqueo, los otros no.
- Si dos hilos están a punto de ejecutar un código sincronizado de una misma instancia de una clase, solo un hilo a la vez podrá ejecutar el método. El otro hilo deberá esperar a que el primero termine para realizar su operación.
- Si un hilo pasa al estado de dormido, mantiene cualquier bloqueo que posea y no lo libera.

¿Se pueden sincronizar los métodos estáticos?

Sí se lo puede realizar, pero el bloqueo se realiza a nivel de *instancia de clase*. Lo que significa que si un hilo obtiene el bloqueo para ejecutar un método *estático sincronizado* todos los hilos que intenten ejecutar otro método *estático* de esa clase tendrán que esperar a obtener el bloqueo.

¿Qué ocurre cuando un objeto no puede obtener el bloqueo?

Cuando un hilo intenta acceder a un método o bloque que ya ha otorgado su bloqueo a algún otro, se dice que este hilo está bloqueado en el bloqueo del objeto. Este hilo tiene que esperar a que el bloqueo del objeto se haya liberado. Se debe tener en cuenta que no existe garantía de que el hilo que haya esperado más tiempo será el próximo a obtener el bloqueo, no existe un orden particular. Se debe tener en cuenta los siguientes puntos:

- Los hilos que llamen a un método no estático sincronizado de una misma clase se bloquearán entre ellos si son invocados utilizando la misma instancia.
- Los hilos que llamen a método estáticos sincronizados en la misma clase siempre se bloquearán entre ellos, ya que están bloqueados en la misma instancia de clase.
- Un método estático sincronizado y un método no estático sincronizado nunca se bloquearán.
- Para bloques sincronizados, se tiene que ver a qué objeto exactamente se está utilizando para el bloqueo (aquello que se encuentra dentro de los paréntesis del bloque sincronizado).

Interacción de hilos

El siguiente tema es la interacción entre hilos para poder comunicarse entre ellos sobre su estado y otras cosas más. La clase *Object* tiene tres métodos, *wait(), notify()* y *notifyAll()*, que ayudarán a comunicar el estado de un evento que es de interés para los otros hilos. Los métodos *wait* y *notify* ponen a un hilo en modo de espera hasta que **otro** hilo le notifique que existe una razón para salir de la espera. Otra cosa a tener en cuenta es que estos tres métodos solo pueden ser llamados desde un método sincronizado.

Cada objeto puede tener una lista de hilos que están esperando una señal (una notificación) del objeto. Un hilo entra en esta lista cuando ejecuta el método *wait()* en el objeto objetivo. Desde ese momento, no se ejecuta ninguna otra instrucción hasta que se ejecuta el método *notify()* del objeto objetivo. Si muchos hilos están esperando en el mismo objeto sólo uno será escogido (sin un orden garantizado) para proceder con su ejecución. Si no hay hilos en espera, no se realiza ninguna acción en particular.

Cuando se ejecuta el método *wait* puede surgir una excepción, de la misma forma en la que puede ocurrir con un hilo en estado dormido (*sleeping*), por lo tanto se debe manejar la excepción debidamente.

```
1.  class ThreadA {
2.     public static void main(String [] args) {
3.        ThreadB b = new ThreadB();
4.        b.start();
5.
6.        synchronized(b) {
7.           try {
8.             System.out.println("Waiting for b to complete...");
9.             b.wait();
10.          } catch (InterruptedException e) {}
11.          System.out.println("Total is: " + b.total);
12.       }
13.    }
14. }
15.
16. class ThreadB extends Thread {
17.    int total;
18.
19.    public void run() {
20.       synchronized(this) {
21.          for(int i=0;i<100;i++) {
22.             total += i;
23.          }
24.          notify();
25.       }
26.    }
27. }
```

Utilizar *notifyAll()* cuando varios hilos estén en espera

En ocasiones, es preferible notificar a todos los hilos que están a la espera de un objeto para pasar del modo espera a ejecución. El método *notifyAll()* permite que esto ocurra. Todos los hilos serán notificados y comenzarán a competir para conseguir el bloqueo.

Ejercicios

Ejecución de un programa sin Hilos

```
package ASinHilo;

/**
 *
 * @author User-WL20
 */
//importacion del paquete de Ingreso y salida de teclado
import java.io.*;
//creacion de la clase principal
public class WithoutThread {
    //Instancia de un objeto de escritura
    static PrintWriter out = new PrintWriter(System.out, true);
    //funcion Principal ejecutable
    public static void main (String args[]) {
        // Primera Tarea: algunos operacion de seugo entrada y salida
        NoThreadPseudoIO pseudo = new NoThreadPseudoIO();
```

```
        pseudo.start();
        // Segunda Tarea: una tarea Randomica
        showElapsedTime("Una segunda tarea Inicia");
    }

    static long baseTime = System.currentTimeMillis();

    // Muestra el tiempo que ha pasado desde que el programa inicio

    static void showElapsedTime(String message) {
        long elapsedTime = System.currentTimeMillis() -baseTime;
        out.println(message + " a " + (elapsedTime/1000.0) + " segundos");
    }
}

// Operacion pseudo I/O  corre en  un hilo principal
class NoThreadPseudoIO {
    int data = -1;
    NoThreadPseudoIO() {  // constructor
        WithoutThread.showElapsedTime("Simulacion IO creado tiempo: ");
    }
    //Inicio del metodo run del hilo
    public void run() {
        WithoutThread.showElapsedTime("Comienza Simulacion IO sin Hilo ");
        try {
            Thread.sleep(10000);  // 10 segundos
            data = 999;             // data lista
            WithoutThread.showElapsedTime("finaliza Simulacion IO sin hilo");
        } catch (InterruptedException e) {}
    }
    //inicio del hilo
    public void start()
    {
        run();
    }
}
```

Ejecución de un programa con hilos en forma heredada

```
package Bhilo;
//importacion del paquete de Ingreso y salida de teclado
import java.io.*;
/**
 *
 * @author User-WL20
 */
//creacion de la clase principal
```

```
public class conHilo {
   //Instancia de un objeto de escritura
   static PrintWriter out = new PrintWriter(System.out, true);
//funcion Principal ejecutable
   public static void main (String args[]) {
         // Primera Tarea: algunos operacion de seugo entrada y salida
         ThreadedPseudoIO pseudo = new ThreadedPseudoIO();
         pseudo.start();
         // Segunda Tarea: una tarea Randomica
         showElapsedTime("Una segunda tarea Inicia");
   }

   static long baseTime = System.currentTimeMillis();
    // Muestra el tiempo que ha pasado desde que el programa inicio

   static void showElapsedTime(String message) {
         long elapsedTime = System.currentTimeMillis() -baseTime;
         out.println(message + " a " + (elapsedTime/1000.0) + " segundos");
   }
}

// Operacion de IO se ejecuta en un hilo separado
class ThreadedPseudoIO extends Thread {
   int data = -1;
   ThreadedPseudoIO() {  // constructor
         conHilo.showElapsedTime("Simulacion de IO creada en un hilo separado");
   }
  // Metodo de ejecucion de Hilo
   public void run () {
         conHilo.showElapsedTime("Simulacion de IO en Hilo comienza");
         try {
            Thread.sleep(10000);  // 10 segundos
            data = 999;             // data lista
            conHilo.showElapsedTime("Simulacion de IO en Hilo Finaliza");
         } catch (InterruptedException e) {}
   }
}
```

Ejecución de un Programa con Hilos en interfaz Runnable

```
package CIntrefazRunnable;
import java.io.*;
/**
 *
 * @author User-WL20
 */
//creacion de la clase principal
```

```
public class RunnableThread {
     //Instancia de un objeto de escritura
    static PrintWriter out = new PrintWriter(System.out, true);
//funcion Principal ejecutable
    public static void main (String args[]) {
            // Primera Tarea: algunos operacion de seugo entrada y salida
            RunnablePseudoIO pseudo = new RunnablePseudoIO();
            Thread thread = new Thread (pseudo);
            thread.start();
            // Segunda Tarea: una tarea Randomica
            showElapsedTime("Una segunda tarea Inicia");
    }

    static long baseTime = System.currentTimeMillis();

    // Muestra el tiempo que ha pasado desde que el programa inicio
    static void showElapsedTime(String message) {
            long elapsedTime = System.currentTimeMillis() -baseTime;
            out.println(message + " a " + (elapsedTime/1000.0) + " segundos");
    }
}

// Operacion de IO se ejecuta en un hilo separado
class RunnablePseudoIO implements Runnable {
    int data = -1;

    RunnablePseudoIO() {  // constructor
            RunnableThread.showElapsedTime("Simulacion de IO en Runnable creado");
    }
 // Metodo de ejecucion de Hilo
    public void run() {
            RunnableThread.showElapsedTime("Simulacion de IO en Runnable Inicia");
            try {
                Thread.sleep(10000);  // 10 segundos
                data = 999;             // data lista
                RunnableThread.showElapsedTime("Simulacion de IO en Runnable Finaliza");
            } catch (InterruptedException e) {}
    }
}
```

Programa de prueba de metodos de Hilos

```
package Dmetodos;
/**
 *
 * @author User-WL20
```

```
 */
import java.io.*;
//creacion de la clase principal
public class MethodTest {
    //Instancia de un objeto de escritura
   static PrintWriter out = new PrintWriter(System.out, true);
   //funcion Principal ejecutable
   public static void main (String args[]) {
      //instaciacion de objetos
         FirstThread first = new FirstThread();
         SecondThread second = new SecondThread();
      //ejecucion de los hilos
         first.start();//metodo start
         second.start();
      //try cathc para capturas de errores
         try {
            out.println("Esperando para que el primer hilo finalize...");
            first.join();//Metodo Join
            out.println("Es una espera larga!!");
            out.println("Despertando el segundo Hilo ...");
            synchronized(second) {
              second.notify();//metodo notify
            }
            out.println("Esperando que el segundo Hilo Finalize ...");
            second.join();
         } catch (InterruptedException e) {}
         out.println("Estoy listo para finalizar tambien.");
   }
}
// clase primer hilo heredada
class FirstThread extends Thread {
   //metodo run de hilo
   public void run () {
         try {
            MethodTest.out.println("  Primer hilo comienza a ejecutar.");
            sleep(10000);//metodo sleeo
            MethodTest.out.println(" Primer hilo finaliza la ejecucion .");
         } catch (InterruptedException e) {}
   }
}
// clase segundo hilo heredada
class SecondThread extends Thread {
   public synchronized void run () {
         try {
            MethodTest.out.println("  Segundo hilo comienza la ejecucion.");
            MethodTest.out.println("  Segundo hilo se suspende el mismo.");
```

```
            wait();//metodo waith de espera
            MethodTest.out.println("  Segundo Hilo se ejecuta de nuevo y finaliza.");
        }catch (InterruptedException e) {}
    }
}
```

```
public class Principal {
    public static void main(String[] args) {
        HiloEjemplo miHilo= new HiloEjemplo();/*Se instancia*/
         HiloEjemplo miHilo2= new HiloEjemplo();
        miHilo.start();/*Metodo star que permite ejecutar el hilo*/
        miHilo2.start();
        System.out.println("Fin del proceso");
    }
}

package hiloejemplo;

public class HiloEjemplo extends Thread {
    public void run(){  /*hasta aqui se construye un hilo*/
       for (int i = 0; i <10; i++) {
            System.out.println(i+" "+getName());
        }
        System.out.println("Inicia Hilo"+getName());
    }
}
```

Sincronización

El siguiente código demuestra un problema de condición de carrera con dos hilos intentando acceder a un mismo recurso de un objeto.

```
public class AccountDanger implements Runnable{
  private Account acct = new Account();
  public static void main (String [] args){
          AccountDanger r = new AccountDanger();
          Thread one = new Thread(r);
          Thread two = new Thread(r);
          one.setName("Fred");
          two.setName("Lucy");
          one.start();
          two.start();
  }

  public void run(){
          for (int x = 0; x < 5; x++) {
          makeWithdrawal(10);
```

```
        if (acct.getBalance() < 0){
        System.out.println("La cuenta esta sobregirada!");
        }
        }
  }

  private synchronized void makeWithdrawal(int amt){
  //private void makeWithdrawal(int amt){
        if (acct.getBalance() >= amt){
        System.out.println(Thread.currentThread().getName() + " va a hacer un retiro");
        try{
        Thread.sleep(1000);
        }
        catch(InterruptedException ex){ }

        acct.retirada(amt);
        System.out.println(Thread.currentThread().getName() + " completo el retiro");
        }
        else {
        System.out.println("No hay suficiente en la cuenta " +
Thread.currentThread().getName() + " para retirar " + acct.getBalance());
        }
  }
}

class Account {
  private int balance = 50;
  public int getBalance() {
        return balance;
  }
  public void retirada(int amount){
        balance = balance - amount;
  }
}
```

Interacción 1

```
public class ThreadA {
  public static void main(String [] args) {
        ThreadB b = new ThreadB();
        b.start();
        System.out.println("Waiting for b to complete...");
        /*
        synchronized(b){
        try {
```

```
        System.out.println("Waiting for b to complete...");
        b.wait();
        } catch(InterruptedException e){}

        System.out.println("Total is: " + b.total);
        }
        */
  }
}

class ThreadB extends Thread {
  int total;
  public void run() {
        synchronized(this) {
        for(int i=0;i<100;i++) {
        total += i;
        }
        notify();
        }
  }
}
```

Interacción 2

```
class Reader extends Thread {
  Calculator c;
  public Reader(Calculator calc) {
        c = calc;
  }
  public void run() {
        synchronized(c) {
        try {
        System.out.println("Waiting for calculation...");
        c.wait();
        }
        catch (InterruptedException e) {
        }
        System.out.println("Total is: " + c.total);
        }
  }
  public static void main(String [] args) {
        Calculator calculator = new Calculator();
        new Reader(calculator).start();
        new Reader(calculator).start();
        new Reader(calculator).start();
```

```
        new Thread(calculator).start();
  }
}

class Calculator implements Runnable {
  int total;
  public void run() {
        synchronized(this) {
        for(int i = 0; i < 100; i++) {
        total += i;
        }
        notifyAll();
        }
  }
}
```

Conclusiones

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

Para evitar problemas de condiciones de carrera se debe sincronizar el código. En Java se utiliza la palabra reservada *syncronized* solo a los métodos de una clase (estáticos y no estáticos) para que estas puedan bloquear el acceso para un solo hilo a la vez.

Para coordinar actividades entre diferentes hilos se utiliza los métodos *wait(), notify()* y *notifyAll()*.

El concepto de multitarea o multiprocesamiento es bastante sencillo de entender ya que solo consiste en hacer varias cosas a la vez sin que se vea alterado el resultado final. Como ya se ha dicho en la entrada no todo se puede paralelizar y en muchas ocasiones suele ser complicado encontrar la manera de paralelizar procesos dentro de una aplicación sin que esta afecte al resultado de la misma, por tanto aunque el concepto sea fácil de entender el aplicarlo a un caso práctico puede ser complicado para que el resultado de la aplicación no se vea afectado.

Recomendaciones

Puedes llegar a ser mejor programador, tienes que ser realmente bueno en lo que se refiere a la estructura de los datos, algoritmos, diseño usando OOPS, multi-hilo y varios otros conceptos de programación, por ejemplo, recursión, divide y vencerás, prototipado y pruebas de unidad.

Se recomienda no usar el método stop() al menos que sea necesario ya que podría causar inconsistencias, es mejor dejar que ejecute cuando el método run() haya finalizado.

Es recomendable agregar tus propios controles de bloqueos a tu código, a pesar de que se esté utilizando una clase que sea *"thread-safe"* ya que podrían ocurrir llamadas que alteren el estado del objeto durante el uso de estas clases.

Utilizar *notifyAll()* cuando existan varios hilos en espera de un objeto.

Bibliografía

Sierra K. & Bates B. (2015) OCA/OC Java SE 7 Programmer I & II Study Guide. McGraw Hill Education (Publisher).

Abellan, J. (2017). *Multitarea e Hilos en Java con ejemplos (Thread & Runnable) - Jarroba*. *Jarroba*. Retrieved 28 December 2016, from http://jarroba.com/multitarea-e-hilos-en-java-con-ejemplos-thread-runnable/

Navarro, L. (2017). *Hilos en Java(Threads) parte 1*. *Monillo007.blogspot.com*. Retrieved 3 January 2017, from http://monillo007.blogspot.com/2008/01/hilos-en-java-threads-parte-1.html

Moya, R. (2017). *Multitarea e Hilos en Java con ejemplos (Thread & Runnable) - Jarroba*. *Jarroba*. Retrieved 9 January 2017, from http://jarroba.com/multitarea-e-hilos-en-java-con-ejemplos-thread-runnable/

Abellan, J. (2017). Multitarea e Hilos, fácil y muchas ventajas. *Jarroba*. Retrieved 9 January 2017, from http://jarroba.com/multitarea-e-hilos-facil-y-muchas-ventajas/