CLASE TEÓRICA 5 / 12

ÍNDICE DE CONTENIDO

5.1 MULTITAREA

Los procesadores y los Sistemas Operativos modernos permiten la multitarea, es decir, la realización simultánea de dos o más actividades (al menos aparentemente). En la realidad, una PC con una sola CPU no puede realizar dos actividades a la vez. Sin embargo los Sistemas Operativos actuales son capaces de ejecutar varios programas "simultáneamente" aunque sólo se disponga de una CPU: reparten el tiempo entre dos (o más) actividades, o bien utilizan los tiempos muertos de una actividad (por ejemplo, operaciones de lectura de datos desde el teclado) para trabajar en la otra. En PCs con dos o más procesadores la multitarea es real, ya que cada procesador puede ejecutar un hilo o thread diferente.

Un proceso es un programa ejecutándose de forma independiente y con un espacio propio de

memoria. Un Sistema Operativo multitarea es capaz de ejecutar más de un proceso simultáneamente. Un thread o hilo es un flujo secuencial simple dentro de un proceso. Un único proceso puede tener varios hilos ejecutándose. Por ejemplo un navegador web (Google Chrome, Mozilla Firefox, …) sería un proceso, mientras que cada una de las ventanas que se pueden tener abiertas simultáneamente trayendo páginas HTML estaría formada por al menos un hilo más.

Un sistema multitarea da realmente la impresión de estar haciendo varias cosas a la vez y eso es una gran ventaja para el usuario. Sin el uso de threads hay tareas que son prácticamente imposibles de ejecutar, particularmente las que tienen tiempos de espera importantes entre etapas.

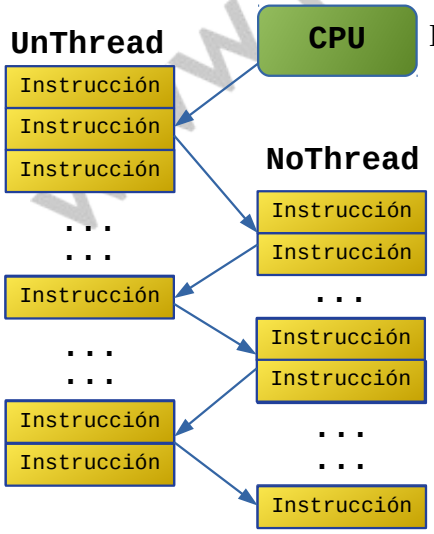
5.2 THREADS

Un thread es un flujo simple de ejecución dentro de un programa. Hasta el momento, todos los programas creados contenían un único thread, pero un programa (o proceso) puede iniciar la ejecución de varios de ellos concurrentemente. En los threads lanzados desde un mismo programa, la memoria se comparte, utilizando el mismo contexto y recursos asignados al programa (también disponen de variables y atributos locales al thread).

Un thread no puede existir independientemente de un programa, sino que se ejecuta dentro de un programa o proceso.

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

Los threads pueden ser daemon o no daemon. Son daemon aquellos hilos que realizan en background (en un segundo plano) servicios generales, esto es, tareas que no forman parte de la esencia del programa y que se están ejecutando mientras no finalice la aplicación. Un thread daemon podría ser por ejemplo aquél que está comprobando permanentemente si el usuario pulsa un botón. Un programa de JAVA finaliza cuando sólo quedan corriendo threads de tipo daemon. Por defecto, y si no se indica lo contrario, los threads son del tipo no daemon.



Ej.

```
public class NoThread extends Thread {
 public void run() {
        int i;
        for (i = 1; i <= 200; i++)
              System.out.print("NO ");
 }
}

public class UnThread {
 public static void main(String args[]) {
        int i;
        NoThread t = new NoThread();
        t.start();
        for (i = 1; i <= 200; i++)
              System.out.print("SI ");
 }
}
```

La salida de este programa es una serie alternativa de Síes y Noes:

SI SI SI SI NO NO NO NO NO NO NO NO NO NO NO SI SI SI SI SI SI SI SI SI SI NO NO NO SI SI SI SI SI SI NO NO ….

Una vez se inicia la ejecución del thread, el tiempo de la CPU se reparte entre todos los procesos y threads del sistema, con lo cuál, se intercalan instrucciones del método **main**() con instrucciones del método **run**() entre otras correspondientes a otros procesos (del sistema operativo y otros procesos de usuario que pudieran estar ejecutándose).

5.3 Creación de Threads

En JAVA hay dos formas de crear nuevos threads. La primera de ellas consiste en crear una nueva clase que herede de la clase **java.lang.Thread** y sobrecargar el método **run**() de dicha clase. El segundo método consiste en declarar una clase que implemente la interface **java.lang.Runnable**, la cual declarará el método **run**(); posteriormente se crea un objeto de tipo **Thread** pasándole como argumento al constructor el objeto creado de la nueva clase (la que implementa la interface **Runnable**). Como ya se ha apuntado, tanto la clase **Thread** como la interface **Runnable** pertenecen al package *java.lang*, por lo que no es necesario importarlas.

5.3.1 Creación de threads derivando de la clase Thread

Considérese el siguiente ejemplo de declaración de una nueva clase:

```
public class EjExtendsThread extends Thread {
    // constructor
    public EjExtendsThread(String str) {
          super(str);
    }

    // redefinición del método run()
    public void run() {
          for (int i = 0; i < 10; i++)
                System.out.println("Este es el thread : " + getName());
    }

    public static void main(String[] args) {
          //Creo dos objetos de la clase con diferente string
          EjExtendsThread miThread1 = new EjExtendsThread("Hilo de prueba");
          EjExtendsThread miThread2 = new EjExtendsThread("Prueba de hilo");
          //Ejecuto ambos threads
          miThread1.start();
          miThread2.start();
    }
}
```

En este caso, se ha creado la clase que hereda de **Thread**. En su constructor se utiliza un String (opcional) para poner nombre al nuevo thread creado, y mediante super() se llama al constructor de la super-clase **Thread**. Asimismo, se redefine el método **run**(), que define la principal actividad del thread, para que escriba 10 veces el nombre del thread creado.

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

5.3.2 Creación de threads implementando la interface Runnable

Esta segunda forma también requiere que se defina el método **run**(), pero además es necesario crear un objeto de la clase **Thread** para lanzar la ejecución del nuevo hilo. Al constructor de la clase **Thread**

hay que pasarle una referencia del objeto de la clase que implementa la interface **Runnable**.

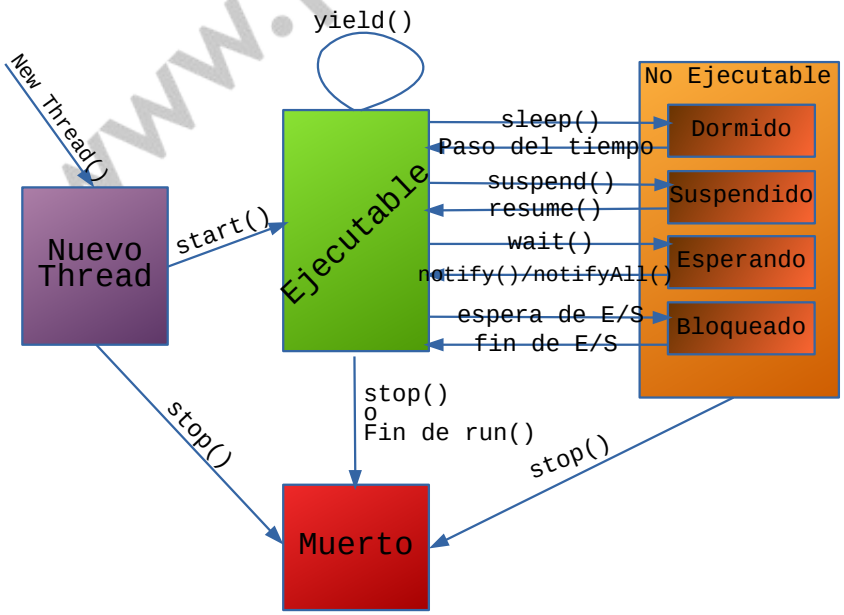
Posteriormente, cuando se ejecute el método *start*() del thread, éste llamará al método **run**() definido en la nueva clase. A continuación se muestra el mismo estilo de clase que en el ejemplo anterior implementada mediante la interface **Runnable**:

```
public class EjRunnable implements Runnable {
    // se crea un nombre
    String nameThread;
    // constructor
    public EjRunnable(String str) {
          nameThread = str;
    }
    // definición del método run()
    public void run() {
          for (int i = 0; i < 10; i++)
                System.out.println("Este es el thread: " + nameThread);
    }
}
public class PruebaRunnable {

    public static void main(String[] args) {
          // El siguiente código crea dos nuevos threads y los ejecuta
          EjRunnable p1 = new EjRunnable("Hilo de prueba");
          EjRunnable p2 = new EjRunnable("Prueba de hilo");
          // se crean objetos de la clase Thread pasándolo el objeto Runnable como
argumento
          Thread miThread1 = new Thread(p1);
          Thread miThread2 = new Thread(p2);
          // se arranca los objetos de la clase Thread
          miThread1.start();
          miThread2.start();
    }
}
```

La elección de una u otra forma -derivar de **Thread** o implementar **Runnable**- depende del tipo de clase que se vaya a crear. Así, si la clase a utilizar ya hereda de otra clase (por ejemplo un applet, que siempre hereda de Applet), no quedará más remedio que implementar **Runnable**, aunque normalmente es más sencillo heredar de **Thread**.

5.4 Ciclo de vida de un Thread

En la Figura se muestran los distintos estados por los que puede pasar un thread a lo largo de su vida. Un thread puede presentar cuatro estados distintos:



1. Nuevo (New): El thread ha sido creado pero no inicializado, es decir, no se ha ejecutado todavía el método *start*(), por ende aun no se encuentra en cola de ejecución del CPU. Se producirá un mensaje de error (*IllegalThreadStateException*) si se intenta ejecutar cualquier método de la clase **Thread** distinto de *start*().

2. Ejecutable (Runnable): El thread puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado en beneficio de otro thread, pero sigue estando en cola de ejecución.
3. Bloqueado (Blocked o Not Runnable): Un thread en estado ejecutable puede pasar al estado no ejecutable por alguna de las siguientes razones: que sean invocados alguno de sus métodos *sleep*() o *suspend*(), que el thread haga uso de su método *wait*() o que el thread esté bloqueado esperando una operación de entrada/salida o que se le asigne algún recurso.
4. Muerto (Dead): La forma habitual de que un thread muera es finalizando el método **run**(). También puede llamarse al método *stop*() de la clase Thread, aunque dicho método es considerado "peligroso" y no se debe utilizar.

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

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

5.4.1 Ejecución de un nuevo thread

La creación de un nuevo thread no implica necesariamente que se empiece a ejecutar algo. Hace falta iniciarlo con el método *start*(), ya que de otro modo, cuando se intenta ejecutar cualquier método del thread -distinto del método *start*() - se obtiene en tiempo de ejecución el error *IllegalThreadStateException*.

El método *start*() se encarga de llamar al método **run**() de la clase **Thread**. Si el nuevo thread se ha creado heredando de la clase **Thread** la nueva clase deberá redefinir el método **run**() heredado. En el caso de utilizar una clase que implemente la interface **Runnable**, el método **run**() de la clase **Thread** se ocupa de llamar al método **run**() de la nueva clase.

Una vez que el método *start*() ha sido llamado, se puede decir ya que el thread está "corriendo" (running), lo cual no quiere decir que se esté ejecutando en todo momento, pues ese thread tiene que compartir el tiempo de la CPU con los demás threads que también estén running. Por eso más bien se dice que dicha thread es runnable.

5.4.2 Detener un Thread temporalmente: Runnable - Not Runnable

El Sistema Operativo se ocupa de asignar tiempos de CPU a los distintos threads que se estén ejecutando simultáneamente. Aun en el caso de disponer de un ordenador con más de un procesador (2 ó más CPUs), el número de threads simultáneos suele siempre superar el número de CPUs, por lo que se debe repartir el tiempo de forma que parezca que todos los procesos corren a la vez (quizás más lentamente), aun cuando sólo unos pocos pueden estar ejecutándose en un instante de tiempo.

Los tiempos de CPU que el Sistema continuamente asigna a los distintos threads en estado runnable se utilizan en ejecutar el método **run**() de cada thread. Por diversos motivos, un thread puede en un determinado momento renunciar "voluntariamente" a su tiempo de CPU y otorgárselo al Sistema para que se lo asigne a otro thread. Esta "renuncia" se realiza mediante el método *yield*().

Es importante que este método sea utilizado por las actividades que tienden a "monopolizar" la CPU. El método *yield*() viene a indicar que en ese momento no es muy importante para ese thread el ejecutarse continuamente y por lo tanto tener ocupada la CPU. En caso de que ningún thread esté requiriendo la CPU para una actividad muy intensiva, el Sistema volverá casi de inmediato a asignar nuevo tiempo al thread que fue "generoso" con los demás.

Si lo que se desea es parar o bloquear temporalmente un thread (pasar al estado Not Runnable), existen varias formas de hacerlo:

1. Ejecutando el método *sleep*() de la clase **Thread**. Esto detiene el thread un tiempo preestablecido. De ordinario el método *sleep*() se llama desde el método **run**().
2. Ejecutando el método *wait*() heredado de la clase **Object**, a la espera de que suceda algo que es necesario para poder continuar. El thread volverá nuevamente a la situación de runnable mediante los métodos *notify*() o *notifyAll*(), que se deberán ejecutar cuando cesa la condición que tiene detenido al thread.
3. Cuando el thread está esperando para realizar operaciones de Entrada/Salida o Input/Output (E/S ó I/O).
4. Cuando el thread está tratando de llamar a un método synchronized de un objeto, y dicho objeto está bloqueado por otro thread.

Un thread pasa automáticamente del estado Not Runnable a Runnable cuando cesa alguna de las condiciones anteriores o cuando se llama a *notify*() o *notifyAll*().

La clase **Thread** dispone también de un método *stop*(), pero no se debe utilizar ya que puede provocar bloqueos del programa (deadlock). Hay una última posibilidad para detener un thread, que consiste en ejecutar el método *suspend*(). El thread volverá a ser ejecutable de nuevo ejecutando el método *resume*(). Esta última forma también se desaconseja, por razones similares a la utilización del método *stop*().

El método *sleep*() de la clase **Thread** recibe como argumento el tiempo en milisegundos que ha de permanecer detenido. Adicionalmente, se puede incluir un número entero con un tiempo adicional en nanosegundos. Las declaraciones de estos métodos son las siguientes:

La forma preferible de detener temporalmente un thread es la utilización conjunta de los métodos *wait*() y *notifyAll*(). La principal ventaja del método *wait*() frente a los métodos anteriormente descritos es que libera el bloqueo del objeto. por lo que el resto de threads que se encuentran esperando para actuar sobre dicho objeto pueden llamar a sus métodos. Hay dos formas de llamar a *wait*():

1. Indicando el tiempo máximo que debe estar parado (en milisegundos y con la opción de indicar también nanosegundos), de forma análoga a *sleep*(). A diferencia del método *sleep*(), que simplemente detiene el thread el tiempo indicado, el método *wait*() establece el tiempo máximo que debe estar parado. Si en ese plazo se ejecutan los métodos *notify*() o *notifyAll*() que indican la liberación de los objetos bloqueados, el thread continuará sin esperar a concluir el

tiempo indicado.

2. Sin argumentos, en cuyo caso el thread permanece parado hasta que sea reinicializado explícitamente mediante los métodos *notify*() o *notifyAll*().

Los métodos *wait*() y *notify*() han de estar incluidas en un método synchronized, ya que de otra forma se obtendrá una excepción del tipo *IllegalMonitorStateException* en tiempo de ejecución. El uso típico de *wait*() es el de esperar a que se cumpla alguna determinada condición, ajena al propio thread. Cuando ésta se cumpla, se utilizará el método *notifyAll*() para avisar a los distintos threads que pueden utilizar el objeto.

5.4.3 Finalizar un Thread

Un thread finaliza cuando el método **run**() devuelve el control, por haber terminado lo que tenía que hacer (por ejemplo, un bucle for que se ejecuta un número determinado de veces) o por haberse dejado de cumplir una condición (por ejemplo, por un bucle while en el método **run**()).

Para saber si un thread está "vivo" o no, es útil el método *isAlive*() de la clase **Thread**, que devuelve **true** si el thread ha sido inicializado y no parado, y **false** si el thread es todavía nuevo (no ha sido inicializado) o ha finalizado.

5.4 Sincronización

La sincronización nace de la necesidad de evitar que dos o más threads traten de acceder a los mismos recursos al mismo tiempo. Así, por ejemplo, si un thread tratara de escribir en un fichero, y otro thread estuviera al mismo tiempo tratando de borrar dicho fichero, se produciría una situación no deseada. Otra situación en la que hay que sincronizar threads se produce cuando un thread debe esperar a que estén preparados los datos que le debe suministrar el otro thread (paradigma productor/consumidor). Para solucionar estos tipos de problemas es importante poder sincronizar los distintos threads.

Las secciones de código de un programa que acceden a un mismo recurso (un mismo objeto de una clase, un fichero del disco, etc.) desde dos o más threads distintos se denominan secciones críticas (critical sections). Para sincronizar dos o más threads, hay que utilizar el modificador **synchronized** en aquellos métodos del objeto-recurso con los que puedan producirse situaciones conflictivas. De esta forma, JAVA bloquea (asocia un bloqueo o lock) con el recurso sincronizado.

```
public synchronized void metodoSincronizado() {
        ...// accediendo por ejemplo a las variables de un objeto
        ...
}
```

La sincronización previene las interferencias solamente sobre un tipo de recurso: la memoria reservada para un objeto. Cuando se prevea que unas determinadas variables de una clase pueden tener problemas de sincronización, se deberán declarar como **private** (o **protected**). De esta forma sólo estarán accesibles a través de métodos de la clase, que deberán estar sincronizados.

Es muy importante tener en cuenta que si se sincronizan algunos métodos de un objeto pero otros no, el programa puede no funcionar correctamente. La razón es que los métodos no sincronizados pueden acceder libremente a las variables miembro, ignorando el bloqueo del objeto.

Sólo los métodos sincronizados comprueban si un objeto está bloqueado. Por lo tanto, todos los métodos que accedan a un recurso compartido deben ser declarados **synchronized**. De esta forma, si

algún método accede a un determinado recurso, JAVA bloquea dicho recurso, de forma que el resto de threads no puedan acceder al mismo hasta que el primero en acceder termine de realizar su tarea.

Bloquear un recurso u objeto significa que sobre ese objeto no pueden actuar simultáneamente dos métodos sincronizados.

Existen dos niveles de bloqueo de un recurso. El primero es a nivel de objetos, mientras que el segundo es a nivel de clases. El primero se consigue declarando todos los métodos de una clase como **`synchronized`**. Cuando se ejecuta un método **`synchronized`** sobre un objeto concreto, el sistema bloquea dicho objeto, de forma que si otro thread intenta ejecutar algún método sincronizado de ese objeto, este segundo método se mantendrá a la espera hasta que finalice el anterior (y desbloquee por lo tanto el objeto). Si existen varios objetos de una misma clase, como los bloqueos se producen a nivel de objeto, es posible tener distintos threads ejecutando métodos sobre diversos objetos de una misma clase.

El bloqueo de recursos a nivel de clases se corresponde con los métodos de clase o **`static`**, y por lo tanto con las variables de clase o **`static`**. Si lo que se desea es conseguir que un método bloquee simultáneamente una clase entera, es decir todos los objetos creados de una clase, es necesario declarar este método como **`synchronized static`**. Durante la ejecución de un método declarado de esta segunda forma ningún método sincronizado tendrá acceso a ningún objeto de la clase bloqueada.

La sincronización puede ser problemática y generar errores. Un thread podría bloquear un determinado recurso de forma indefinida, impidiendo que el resto de threads accedieran al mismo.

Para evitar esto último, habrá que utilizar la sincronización sólo donde sea estrictamente necesario.

Es necesario tener presente que si dentro un método sincronizado se utiliza el método *`sleep`*() de la clase **`Thread`**, el objeto bloqueado permanecerá en ese estado durante el tiempo indicado en el argumento de dicho método. Esto implica que otros threads no podrán acceder a ese objeto durante ese tiempo, aunque en realidad no exista peligro de simultaneidad ya que durante ese tiempo el thread que mantiene bloqueado el objeto no realizará cambios. Para evitarlo es conveniente sustituir *`sleep`*() por el método *`wait`*() de la clase **`java.lang.Object`** heredado automáticamente por todas las clases. Cuando se llama al método *`wait`*() (siempre debe hacerse desde un método o bloque **`synchronized`**) se libera el bloqueo del objeto y por lo tanto es posible continuar utilizando ese objeto a través de métodos sincronizados. El método *`wait`*() detiene el thread hasta que se llame al método *`notify`*() o *`notifyAll`*() del objeto, o finalice el tiempo indicado como argumento del método *`wait`*(). El método *`unObjeto.notify`*() lanza una señal indicando al sistema que puede activar uno de los threads que se encuentren bloqueados esperando para acceder al objeto *`unObjeto`*. El método *`notifyAll`*() lanza una señal a todos los threads que están esperando la liberación del objeto.

Los métodos *`notify`*() y *`notifyAll`*() deben ser llamados desde el thread que tiene bloqueado el objeto para activar el resto de threads que están esperando la liberación de un objeto. Un thread se convierte en propietario del bloqueo de un objeto ejecutando un método sincronizado del objeto.

```
public class Contador {
    private long valor = 0;

    public void incrementa() {
          long aux;
          aux = valor;
          aux++;
          valor = aux;
    }

    public long getValor() {
          return valor;
    }
```

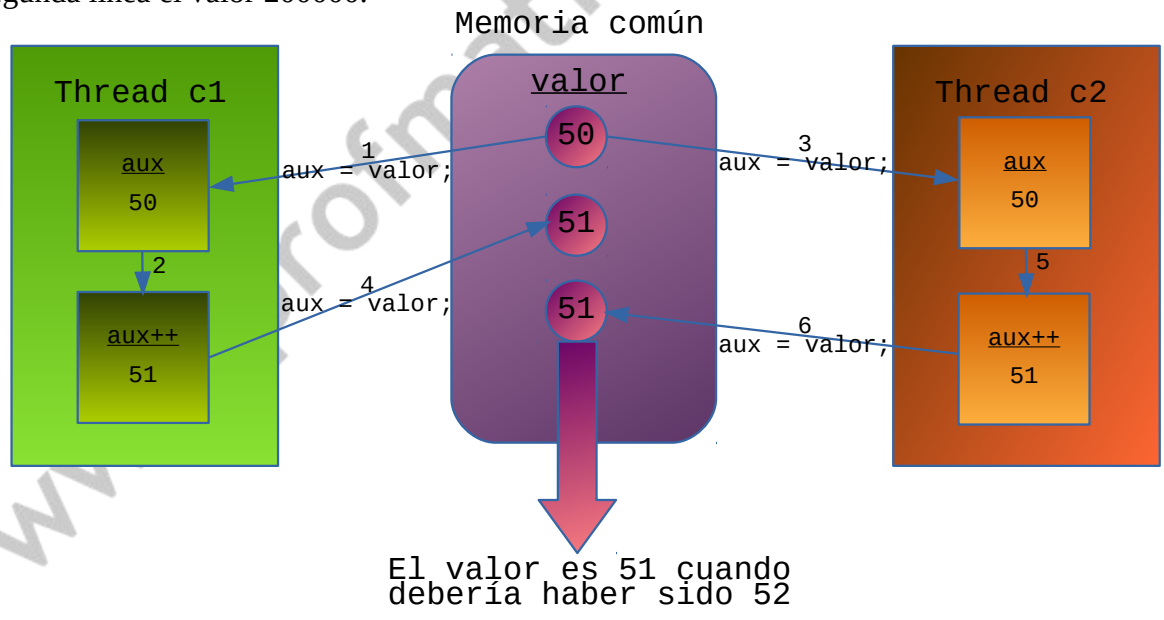
```
}
public class Contable extends Thread {
    Contador contador;

    public Contable(Contador c) {
          contador = c;
    }

    public void run() {
          int i;
          long aux;
          for (i = 1; i <= 100000; i++) {
                contador.incrementa();
          }
          System.out.println("Contado hasta ahora: " + contador.getValor());
    }
}

public class ThreadsSincronizados {
    public static void main(String arg[]) {
          Contable c1 , c2 ;
          Contador c = new Contador();
          c1 = new Contable(c);
          c2 = new Contable(c);
          c1.start();
          c2.start();
          }
}
```

En este caso, la sección crítica es la línea: `contador.incrementa();` ya que desde esta instrucción se accede a un objeto (**Contador**) que es compartido por ambos threads (`c1` y `c2`).

La salida por pantalla es:

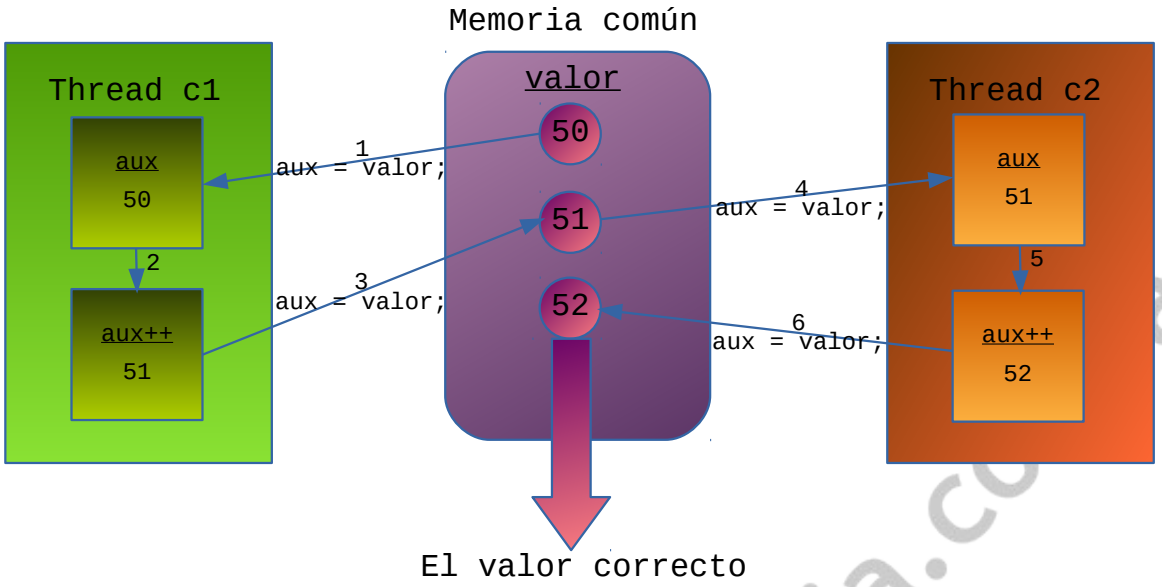
```
Contado hasta ahora: 124739
Contado hasta ahora: 158049
```

cuando debería haber sido: en la primera línea un número comprendido entre 100000 y 200000; y en la segunda línea el valor 200000.



Evidentemente, esto mismo ha ocurrido un gran número de veces durante la ejecución de los threads.

Simplemente cambiando la línea de código **`public void incrementa()`** por **`public synchronized void incrementa()`** se habría conseguido bloquear el método incrementa, de forma que no pudieran ejecutarlo simultáneamente dos threads sobre el mismo objeto (**contador**).



La salida del programa habría sido (correcta):

```
Contado hasta ahora: 186837
Contado hasta ahora: 200000
```

Existe también la posibilidad de sincronizar una parte del código de un método sin necesidad de mantener bloqueado el objeto desde el comienzo hasta el final del método. Para ello se utiliza la palabra clave **`synchronized`** indicando entre paréntesis el objeto que se desea sincronizar (**`synchronized`** (*`objetoASincronizar`*)).

La sincronización es un proceso que lleva bastante tiempo a la CPU, luego se debe minimizar su uso, ya que el programa será más lento cuanta más sincronización incorpore.

5.5 Prioridades

Con el fin de conseguir una correcta ejecución de un programa se establecen prioridades en los threads, de forma que se produzca un reparto más eficiente de los recursos disponibles. Así, en un determinado momento, interesará que un determinado proceso acabe lo antes posible sus cálculos, de forma que habrá que otorgarle más recursos (más tiempo de CPU). Esto no significa que el resto de procesos no requieran tiempo de CPU, sino que necesitarán menos. La forma de llevar a cabo esto es gracias a las prioridades.

Cuando se crea un nuevo thread, éste hereda la prioridad del thread desde el que ha sido inicializado. Las prioridades viene definidas por variables miembro de la clase **`Thread`**, que toman valores enteros que oscilan entre la máxima prioridad `MAX_PRIORITY` (normalmente tiene el valor 10) y la mínima prioridad `MIN_PRIORITY` (valor 1), siendo la prioridad por defecto `NORM_PRIORITY` (valor 5). Para modificar la prioridad de un thread se utiliza el método *`setPriority`*(). Se obtiene su valor con *`getPriority`*().

El algoritmo de distribución de recursos en JAVA escoge por norma general aquel thread que tiene una prioridad mayor, aunque no siempre ocurra así, para evitar que algunos procesos queden “dormidos”. Cuando hay dos o más threads de la misma prioridad (y además, dicha prioridad es la más

elevada), el Sistema no establecerá prioridades entre los mismos, y los ejecutará alternativamente dependiendo del Sistema Operativo en el que esté siendo ejecutado. Este algoritmo tiene un peligro: que un thread nunca se ejecute porque siempre existan threads de mayor prioridad. JAVA evita esto asignando la CPU también a threads de menor prioridad aunque cada mucho menos tiempo que a los threads de mayor prioridad.

Un thread puede en un determinado momento renunciar a su tiempo de CPU y otorgárselo a otro thread de la misma prioridad, mediante el método *yield*(), aunque en ningún caso a un thread de prioridad inferior.

5.6 Grupos de Threads

Todo hilo de JAVA debe formar parte de un grupo de hilos (ThreadGroup). Puede pertenecer al grupo por defecto o a uno explícitamente creado por el usuario. Los grupos de threads proporcionan una forma sencilla de manejar múltiples threads como un solo objeto. Así, por ejemplo es posible parar varios threads con una sola llamada al método correspondiente. Una vez que un thread ha sido asociado a un threadgroup, no puede cambiar de grupo.

Cuando se arranca un programa, el Sistema crea un ThreadGroup llamado *main*. Si en la creación de un nuevo thread no se especifica a qué grupo pertenece, automáticamente pasa a pertenecer al threadgroup del thread desde el que ha sido creado (conocido como current thread group y current thread, respectivamente). Si en dicho programa no se crea ningún ThreadGroup adicional, todos los threads creados pertenecerán al grupo *main* (en este grupo se encuentra el método **main**()).

Para conseguir que un thread pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo thread, según uno de los siguientes constructores:

```
public Thread (ThreadGroup grupo, Runnable destino)
public Thread (ThreadGroup grupo, String nombre)
public Thread (ThreadGroup grupo, Runnable destino, String nombre)
```

A su vez, un ThreadGroup debe pertenecer a otro ThreadGroup. Como ocurría en el caso anterior, si no se especifica ninguno, el nuevo grupo pertenecerá al ThreadGroup desde el que ha sido creado (por defecto al grupo *main*). La clase **ThreadGroup** tiene dos posibles constructores:

```
ThreadGroup (ThreadGroup parent, String nombre);
ThreadGroup (String nombre);
```

el segundo de los cuales toma como parent el threadgroup al cual pertenezca el thread desde el que se crea (**Thread**.*currentThread*()).

En la práctica los ThreadGroups no se suelen utilizar demasiado. Su uso práctico se limita a efectuar determinadas operaciones de forma más simple que de forma individual. En cualquier caso, véase el siguiente ejemplo:

```
ThreadGroup miThreadGroup = new ThreadGroup("Mi Grupo de Threads");
Thread miThread = new Thread(miThreadGroup, "un thread para mi grupo");
```

donde se crea un grupo de threads (`miThreadGroup`) y un thread que pertenece a dicho grupo (`miThread`).

5.7 Multitarea en Swing

En las aplicaciones JAVA que usan Swing es particularmente importante manejar con cuidado la concurrencia. Una aplicación JAVA que usa Swing y que está bien desarrollada usa la concurrencia para crear una interfaz de usuario que nunca se bloquea, de modo que la aplicación siempre es capaz de responder a la interacción del usuario en un momento dado con independencia de las tareas que esté llevando a cabo la aplicación en ese momento. Para desarrollar una aplicación que tiene esta capacidad de respuesta, el desarrollador debe aprender cómo Swing emplea los hilos.

Las aplicaciones JAVA que usan Swing tienen tres tipos de hilos:

- Un hilo inicial o principal.
- Un hilo de despacho o expedición de eventos.
- Varios hilos trabajadores, también conocidos como hilos en segundo plano.

En toda aplicación JAVA, la máquina virtual arranca la aplicación creando el hilo principal de dicha aplicación y, a continuación, le cede el control para que se empiece a ejecutar. Este thread invoca al método `main()` de la aplicación, el cual constituye el punto de entrada o de inicio de la aplicación.

En las aplicaciones JAVA que usan Swing, la principal tarea del hilo principal es proveer la creación e inicialización de la interfaz gráfica de usuario (GUI) de la aplicación. Una vez que se finaliza esta tarea, generalmente termina la ejecución del método `main()` de la aplicación y, por tanto, también termina de ejecutarse el hilo principal.

También existe un hilo de despacho de eventos dedicado a la GUI. Este hilo es el que dibuja y actualiza los componentes, y también es el que responde a las interacciones del usuario con la GUI invocando los correspondientes manejadores de eventos de la aplicación. De esta explicación podemos sacar dos conclusiones:

- Todos los manejadores de eventos son ejecutados por el hilo de despacho de eventos.
- Toda interacción con los componentes de la GUI y con sus modelos de datos asociados debe realizarse únicamente por el hilo de despacho de eventos.

Por tanto, acceder a los componentes de la GUI o a sus manejadores de eventos desde otros hilos distintos al hilo de despacho de eventos puede causar errores de dibujo y actualización de la GUI y, en el peor de los casos, interbloqueo.

Por todo ello, llegamos a la primera regla para trabajar con Swing:

Regla 1

No se debe interactuar con componentes Swing excepto desde el hilo de despacho de eventos.

Veamos una aplicación práctica de esta regla. Es frecuente encontrar que el hilo principal de una aplicación JAVA que usa Swing tiene una forma parecida a la siguiente:

```
public class SwingMultitarea extends javax.swing.JFrame {
    ...
    public static void main(String[] args) {
          new SwingMultitarea().setVisible(true);
    }
}
```

Aunque pueda parecer un código inocuo, viola la primera regla para trabajar con Swing, ya que se está interaccionando con componentes Swing desde un hilo distinto del hilo de despacho de eventos.

Este error es fácil de cometer y, además, los problemas de sincronización que se pueden presentar no son inmediatamente obvios. Para evitar este error, el hilo principal debe tener la forma siguiente:

```
public class SwingMultitarea extends javax.swing.JFrame {

    public static void main(String[] args) {
          SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                      new SwingMultitarea().setVisible(true);
                }
          });
    }
}
```

El hilo principal provee la creación e inicialización de la GUI creando un objeto **`Runnable`** que crea e inicializa la GUI pero cuyo método **`run`**() no es ejecutado por el hilo principal sino por el hilo de despacho de eventos. Para que esto ocurra, se pasa dicho objeto como argumento al método estático *`invokeLater`*() o al método estático *`invokeAndWait`*(), donde ambos pertenecen a la clase **`javax.swing.SwingUtilities`**.

De este modo, se actúa directamente sobra la cola de despacho de eventos que es procesada por el hilo de despacho de eventos, ya que estos métodos estáticos colocan el objeto **`Runnable`** que reciben como argumento al final de la cola de despacho de eventos existentes en ese momento. De este modo, el código del método **`run`**() de dicho objeto será ejecutado por el hilo de despacho de eventos después de que haya ejecutado los demás eventos pendientes que están en la cola de despacho de eventos.

El método *`invokeLater`*() es asíncrono de modo que, una vez que ha insertado el objeto que recibe como argumento en la cola de despacho de eventos, retorna inmediatamente sin esperar a que el hilo de despacho de eventos ejecute el código, de tal forma que el hilo que invocó dicho método puede seguir ejecutándose. Sin embargo, el método *`invokeAndWait`*() es síncrono de modo que, una vez que ha insertado el objeto que recibe como argumento en la cola de despacho de eventos, no retorna hasta que el hilo de despacho de eventos haya ejecutado el método **`run`**() de ese objeto, permaneciendo a la espera mientras tanto el hilo que invocó dicho método.

Como norma general, se debería usar el método *`invokeAndWait`*() para leer el valor de componentes Swing, para actualizar la GUI o para asegurar que algo se muestra por pantalla antes de continuar la ejecución de la aplicación. En otro caso, se puede usar el método *`invokeLater`*().

El método *`invokeAndWait`*() puede lanzar una excepción *`InterruptedException`* si el hilo que ejecuta este método es interrumpido mientras espera a que el hilo de despacho de eventos termine de ejecutar el método **`run`**(), o bien una excepción *`InvocationTargetExcepcion`* si el objeto **`Runnable`** lanza una excepción *`java.lang.Error`* o *`java.lang.RuntimeException`* mientras el hilo de despacho de eventos está ejecutando su método **`run`**(). El método *`invokeLater`*() no lanza ninguna excepción.

Una aplicación JAVA que usa Swing está fundamentalmente guiada por eventos producidos por las interacciones del usuario con la GUI, cada uno de los cuales provoca que el hilo de despacho de eventos ejecute el correspondiente manejador de eventos asociado.

Es importante recordar cuando se escribe código para manejar eventos que todo ese código se ejecuta en el mismo hilo: el hilo de despacho de eventos. Esto implica que mientras el código de manejo de un cierto evento se está ejecutando, ningún otro evento puede ser procesado. El código para manejar el siguiente evento empezará a ejecutarse sólo cuando termine de ejecutarse el manejador de eventos que se está ejecutando actualmente. Por tanto, la capacidad de respuesta de la aplicación a las interacciones del usuario con la GUI es dependiente de cuánto tiempo cuesta ejecutar los manejadores de eventos. Por tanto, los manejadores de eventos deben ejecutarse en tan poco tiempo como sea posible.

Por ello, los manejadores de eventos ejecutados por el hilo de despacho de eventos deben finalizar rápidamente de modo que la GUI pueda tener una buen tiempo de respuesta ante las interacciones del usuario. Si el hilo de despacho de eventos realiza tareas de larga duración, los eventos se acumulan en el hilo de despacho de eventos y la aplicación puede parecer bloqueada ya que no puede responder a evento alguno.

Por todo ello, llegamos a la segunda regla para trabajar con Swing:

Regla 2

Si un manejador de eventos debe realizar una tarea que requiere mucho tiempo o que se puede bloquear, dicha tarea debe ser ejecutada por un nuevo hilo, que recibe el nombre de hilo trabajador o hilo en segundo plano.

```
import javax.swing.SwingUtilities;

public class MultiSwing {

    public static void main(String[] args) {
          SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                      new VentanaPrincipal();
                }
          });
    }
}


import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.DefaultListModel;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JProgressBar;
import javax.swing.JScrollPane;

public class VentanaPrincipal extends JFrame implements ActionListener {
    private JButton botonInicio, botonParada;
    private JScrollPane panelDesplazamiento;
    private JList lista;
    private DefaultListModel modeloLista;
    private JProgressBar barraProgreso;
    private GeneraPrimos generaPrimos;

    public VentanaPrincipal() {
          super("Multitarea Swing");
          this.panelDesplazamiento = new JScrollPane();
          this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          this.getContentPane().setLayout(new FlowLayout());
          this.botonInicio = this.construyeBoton("Iniciar");
          this.botonParada = this.construyeBoton("Parar");
          this.botonParada.setEnabled(false);
          this.barraProgreso = this.construyeBarraProgreso(0, 99);
          this.modeloLista = new DefaultListModel();
          this.lista = new JList(this.modeloLista);
          this.panelDesplazamiento.setViewportView(this.lista);
          this.getContentPane().add(this.panelDesplazamiento);
          this.pack();
          this.setVisible(true);
    }

    private JButton construyeBoton(String titulo) {
          JButton b = new JButton(titulo);
          b.setActionCommand(titulo);
          b.addActionListener(this);
          this.getContentPane().add(b);
          return b;
    }

    private JProgressBar construyeBarraProgreso(int min, int max) {
          JProgressBar progressBar = new JProgressBar();
          progressBar.setMinimum(min);
          progressBar.setMaximum(max);
```

```
            progressBar.setStringPainted(true);
            progressBar.setBorderPainted(true);
            this.getContentPane().add(progressBar);
            return progressBar;
      }

      public void actionPerformed(ActionEvent e) {
            if ("Iniciar".equals(e.getActionCommand())) {
                  this.modeloLista.clear();
                  this.botonInicio.setEnabled(false);
                  this.botonParada.setEnabled(true);
                  this.generaPrimos = new GeneraPrimos(this.modeloLista,
this.barraProgreso, this.botonInicio,
                              this.botonParada);
                  this.generaPrimos.execute();
            } else if ("Parar".equals(e.getActionCommand())) {
                  this.botonInicio.setEnabled(true);
                  this.botonParada.setEnabled(false);
                  this.generaPrimos.cancel(true);
                  this.generaPrimos = null;
            }
      }
}


import java.util.ArrayList;
import java.util.concurrent.ExecutionException;
import javax.swing.DefaultListModel;
import javax.swing.JButton;
import javax.swing.JProgressBar;
import javax.swing.SwingWorker;

public class GeneraPrimos extends SwingWorker<ArrayList<Integer>, Integer> {
      private DefaultListModel modeloLista;
      private JProgressBar barraProgreso;
      private JButton botonInicio, botonParada;

      public GeneraPrimos(DefaultListModel modeloLista, JProgressBar barraProgreso,
JButton botonInicio, JButton botonParada) {
            this.modeloLista = modeloLista;
            this.barraProgreso = barraProgreso;
            this.botonInicio = botonInicio;
            this.botonParada = botonParada;
      }

      protected ArrayList<Integer> doInBackground() {
            Integer valorTemp = new Integer(1);
            ArrayList<Integer> lista = new ArrayList<Integer>();
            for (int i = 0; i < 100; i++) {
                  for (int j = 0; j < 100 && !isCancelled(); j++) {
                        valorTemp = encuentraSiguientePrimo(valorTemp.intValue());
                  }
                  publish(new Integer(i));
                  lista.add(valorTemp);
            }
            return lista;
      }

      @Override
      protected void process(java.util.List<Integer> lista) {
            if (!isCancelled()) {
                  Integer parteCompletada = lista.get(lista.size() - 1);
                  barraProgreso.setValue(parteCompletada.intValue());
            }
      }

      @Override
      protected void done() {
            if (!isCancelled()) {
                  try {
                        ArrayList<Integer> results = get();
                        for (Integer i : results)
                              modeloLista.addElement(i.toString());
                        this.botonInicio.setEnabled(true);
                        this.botonParada.setEnabled(false);
                  } catch (ExecutionException ex) {
                        ex.printStackTrace();
                  } catch (InterruptedException ex) {
                        ex.printStackTrace();
                  }
            }
      }

      private Integer encuentraSiguientePrimo(int num) {
```

```
            do {
                  if (num % 2 == 0)
                        num++;
                  else
                        num = num + 2;
            } while (!esPrimo(num));
            return new Integer(num);
      }

      private boolean esPrimo(int num) {
            int i;
            for (i = 2; i <= num / 2; i++) {
                  if (num % i == 0)
                        return false;
            }
            return true;
      }

}
```

BIBLIOGRAFÍA

- Ceballos, Fco. Javier, “JAVA 2 Curso de programación” 4ta Ed. (Ra-Ma 2010)
- Gosling, James; Holmes, David, “El lenguaje de programación JAVA” (Addison-Wesley 2001)
- Otero, Abraham, “Tutorial básico de JAVA” 3ra Ed. (javahispano.org 2007)
- Sánchez, Jorge, “JAVA 2” (2004)
- Vegas Gertrudix, José Maria, “Multitarea en Swing” (2008)

LICENCIA

Matías E. García

Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar