Unidad Didáctica 2:

PROGRAMACIÓN MULTIHILO

1. Conceptos Básicos sobre Hilos

El multihilo soportado en Java gira alrededor del concepto de hilo. La cuestión es, ¿qué es un hilo? De forma sencilla, un hilo es un flujo de ejecución dentro de un proceso.
Los hilos a menudo son conocidos o llamados procesos ligeros. Un hilo, en efecto, es muy similar a un proceso pero con la diferencia de que un hilo siempre corre dentro del contexto de otro programa. Por el contrario, los procesos mantienen su propio espacio de direcciones y entorno de operaciones. Los hilos dependen de un programa padre en lo que se refiere a recursos de ejecución.
La Maquina Virtual de Java (JVM) permite la ejecución concurrente de múltiples hilos. En la clase Thread se encapsula todo el control necesario sobre los hilos de ejecución o tareas. Un objeto Thread se lo puede entender como el panel de control sobre una tarea o hilo de ejecución.

2. Clases relacionadas con los hilos

El lenguaje de programación Java proporciona soporte para hilos a través de una simple interfaz y un conjunto de clases. La interfaz de Java y las clases que incluyen funcionalidades sobre hilos son las siguientes:

- Thread
- Runnable

Todas estas clases son parte del paquete Java.lang.

2.1.1 Thread

La clase Thread es la clase responsable de producir hilos funcionales para otras clases. Para añadir la funcionalidad de hilo a una clase simplemente se deriva la clase de Thread y se reemplaza el método run(). Es en este método run() donde el procesamiento de un hilo toma lugar, y a menudo se refieren a él como el cuerpo del hilo. La clase Thread también define los métodos start() y stop(), los cuales te permiten comenzar y parar la ejecución del hilo, además de un gran número de métodos útiles.

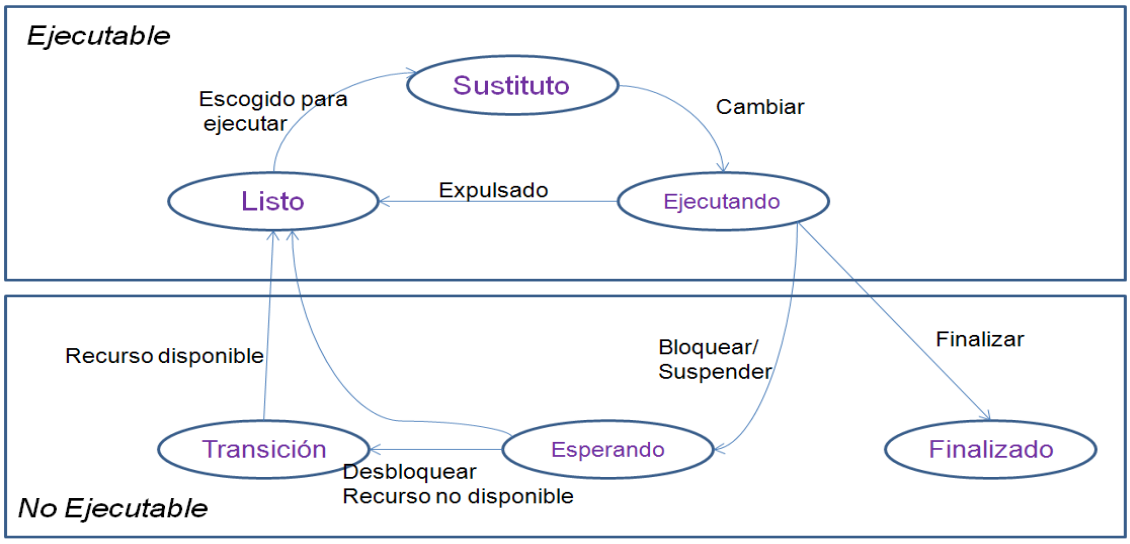
2.1.2 Runnable

Java no soporta herencia múltiple de forma directa, es decir, no se puede derivar una clase de varias clases padre. Esto nos plantea la duda sobre cómo podemos añadir la funcionalidad de Hilo a una clase que deriva de otra clase, siendo ésta

distinta de Thread. Para lograr esto se utiliza la interfaz Runnable. La interfaz Runnable proporciona la capacidad de añadir la funcionalidad de un hilo a una clase simplemente implementando la interfaz, en lugar de derivándola de la clase Thread. Las clases que implementan la interfaz Runnable proporcionan un método run() que es ejecutado por un objeto hilo asociado que es creado aparte. Esta es una herramienta muy útil y a menudo es la única salida que tenemos para incorporar multihilo dentro de las clases.

3.Ciclo de vida de un hilo

Un hilo tiene un ciclo de vida que va desde su creación hasta su terminación.



Durante su ciclo de vida cada uno de los hilos o tareas de una aplicación puede estar en diferentes estados, algunos de los cuales se indican a continuación:

• **Nuevo**: Cuando se acaba de crear un hilo, y continúa en ese estado hasta que se invoca el método start() del hilo. La siguiente sentencia crea un nuevo thread pero no lo arranca, por lo tanto deja el thread en el estado de nacido.
Thread miHilo = new MiClaseThread();
Cuando un thread está en este estado, es sólo un objeto Thread vacío o nulo. No se han asignado recursos del sistema todavía para el thread. Así, cuando un thread está en este estado, lo único que se puede hacer es arrancarlo con **start()**.

• **Listo**: Cuando se invoca el método start() del hilo, se dice que está en estado **listo**. El método se arranca con la siguiente instrucción, para el caso del hilo miHilo:
miHilo.start();

• **Sustituto**: cuando el método **start()** se ejecuta, crea los recursos del sistema necesarios para ejecutar el thread, programa el thread para ejecutarse, y llama al método **run()** del thread que se ejecuta en forma secuencial. En este punto el thread está en el estado **ejecutable**. Se denomina así puesto que todavía no ha empezado a ejecutarse.

• **Ejecutando:** Un hilo en estado de listo de la más alta prioridad, pasa al estado de ejecución, cuando se le asignan los recursos de un procesador, o sea cuando inicia su ejecución. Aquí el thread está en **ejecución**. Cada hilo tiene su prioridad, hilos con alta prioridad se ejecutan preferencialmente sobre los hilos de baja prioridad.

• **No ejecutable** :Un hilo continúa la ejecución de su método run(), hasta que pasa al estado de **no ejecutable** originado cuando ocurre alguno de los siguientes cuatro eventos:

- Se invoca a su método **sleep()**.
- Se invoca a su su método **suspend()**.
- El thread utiliza su método **wait()** para esperar una condición variable.
- El thread está bloqueado durante una solicitud de entrada/salida.

Por ejemplo, en el siguiente fragmento de código se pone a dormir miHilo durante 10 segundos (10.000 milisegundos):

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

Durante los 10 segundos que miHilo está dormido, incluso si el proceso se vuelve disponible, miHilo no se ejecuta. Después de 10 segundos, miHilo se convierte en "Sustituto" de nuevo y, si el procesador está disponible se ejecuta. Para cada entrada en el estado "No Ejecutable", existe una ruta de escape distinta y específica que devuelve el thread al estado "Ejecutable". Por ejemplo, si un thread ha sido puesto a dormir durante un cierto número de milisegundos deben pasar esos milisegundos antes de volverse "Ejecutable" de nuevo.

● **Finalizado**: Un hilo pasa al estado de muerto cuando se termina su método **run()**, o cuando se ha invocado su método **stop()**. En algún momento el sistema dispondrá entonces del hilo muerto. Un hilo puede morir de dos formas:

- **Muerte natural**: se produce cuando su método **run()** sale normalmente. Por ejemplo, el bucle **while** en este método es un bucle que itera 100 veces y luego sale. Por tanto el hilo morirá naturalmente cuando se llegue al final de la iteración, es decir se termina su método run().

```
public void run() {
int i = 0;
while (i < 100) {
i++;
System.out.println("i = " + i);
}
}
```

- **Por muerte provocada:** en cualquier momento llamando a su método **stop()**. El siguiente código crea y arranca miHilo luego lo pone a dormir durante 10 segundos. Cuando el thread actual se despierta, lo mata con miHilo.stop(). El método stop() lanza un objeto ThreadDeath hacia al hilo a eliminar. El thread moriría cuando reciba realmente la excepción ThreadDeath.

```
Thread miHilo = new MiClaseThread();
miHilo.start();
try {
Thread.currentThread().sleep(10000);
} catch (InterruptedException e){
}miHilo.stop();
```

  El método **stop()** provoca una terminación súbita del método **run()** del hilo. Si el método **run()** estuviera realizando cálculos sensibles, **stop()** podría dejar el programa en un estado inconsistente. Normalmente, no se debería llamar al método **stop().**

● **Esperando**: un hilo se encuentra en el estado **bloqueado** cuando el hilo realiza una solicitud de entrada/salida. Cuando termina la entrada/salida que estaba esperando, un hilo bloqueado queda en el estado listo.

El método getState() de la clase thread, permite obtener en cualquier momento el estado en el que se encuentra un hilo. Devuelve por tanto: NEW, RUNNABLE, NO RUNNABLE o TERMINATED.

4. Control de un hilo

Arranque de un hilo
En el contexto de las aplicaciones, sabemos que es main la primera función que se invoca tras arrancar, y por tanto, lógicamente, es el lugar más apropiado para crear y arrancar otros hilos.
La línea de código:
t1 = new TestTh();
siendo TestTh una subclase de la clase Thread (o una clase que implemente la interfaz Runnable) crea un nuevo hilo.
Continuando con lo anterior existen diversos tipos de constructores a partir de los cuales se puede crear un hilo, algunos de los mas importantes son:

- Thread()
- Thread(ObjetoRunnable)
- Thread(objetoRunnable,String nombre)
- Thread(String nombre)

Al tener control directo sobre los hilos, tenemos que arrancarlos explícitamente. Como ya se comentó anteriormente, es la función miembro start la que nos permite hacerlo. En nuestro ejemplo sería:
t1.start();
start, en realidad es un método oculto en el hilo que llama al método run. Una vez que llamamos al método start() sucede lo siguiente:

- Un nuevo proceso de ejecución comienza(con su propia pila de información)
- el hilo o proceso cambia de estado nuevo a estado de ejecución.
- Cuando el hilo tenga su turno de ejecutarse, el método run() del objeto al que se refiere se ejecuta.

Para saber que hilo se encuentra en ejecución en un momento determinado, existe el método estático Thread.currentThread().getName() el cual te devuelve un valor tipo cadena con el nombre del hilo en ejecución, si no has definido un nombre con el método setName(), de igual manera el proceso lo tendrá, algo muy parecido a Thread-0.

Manipulación de un hilo
Si todo fue bien en la creación del objeto TestTh (t1), éste debería contener un hilo, una traza de ejecución válida, que controlaremos en el método run del objeto.

El cuerpo de esta función miembro viene a ser el cuerpo de un programa como ya los conocemos. Digamos que es la rutina main a nivel de hilo. Todo lo que queremos que haga el hilo debe estar dentro del método run. Cuando finalice run, finalizará también el hilo que lo ejecutaba.

Algunas consideraciones importantes que debes tener en cuenta son las siguientes:
Puedes invocar directamente al método run(), por ejemplo poner t1.run(); y se ejecutará el código asociado a run() dentro del hilo actual (como cualquier otro método), pero no comenzará un nuevo hilo como subproceso independiente.
Una vez que se ha llamado al método start() de un hilo, no puedes volver a realizar otra llamada al mismo método. Si lo haces, obtendrás una excepción IllegalThreadStateException.
El orden en el que inicies los hilos mediante start() no influye en el orden de ejecución de los mismos, lo que pone de manifiesto que el orden de ejecución de los hilos es no- determinístico (no se conoce la secuencia en la que serán ejecutadas la instrucciones del programa).

Detener temporalmente un hilo.
¿Qué significa que un hilo se ha detenido temporalmente? Significa que el hilo ha pasado al estado "No Ejecutable". Y ¿cómo puede pasar un hilo al estado "No Ejecutable"? Un hilo pasará al estado "No Ejecutable" o "Detenido" por alguna de estas circunstancias:

- El hilo se ha dormido. Se ha invocado al método sleep() de la clase thread, indicando el tiempo que el hilo permanecerá deteniendo. Transcurrido ese tiempo, el hilo se vuelve "Ejecutable", en concreto pasa a "Preparado".
- El hilo está esperando. El hilo ha detenido su ejecución mediante la llamada al método wait(), y no se reanudará, pasará a "Ejecutable" (en concreto "Preparado") hasta que se produzca una llamada al método notify() o notifyAll() por otro hilo. Estudiaremos detalladamente estos métodos de la clase Object cuando veamos la sincronización y comunicación de hilos.
- El hilo se ha bloqueado. El hilo está pendiente de que finalice una operación de E/S en algún dispositivo, o a la espera de algún otro tipo de recurso; ha sido bloqueado por el sistema operativo. Cuando finaliza el bloqueo, vuelve al estado "Ejecutable", en concreto "Preparado".
- El método suspend() (actualmente en desuso o deprecated) también permite detener temporalmente un hilo, y en ese caso se reanudaría mediante el método resume() (también en desuso). No debes utilizar estos métodos, de la clase thread ya que no son seguros y provocan muchos problemas. Te lo indicamos simplemente porque puede que encuentres programas que aún utilizan estos métodos.

- *Ejemplo. Dormir un hilo con sleep.*

¿Por qué puede interesar dormir un hilo? Pueden ser diferentes las razones que nos lleven a dormir un hilo durante unos instantes. En este apartado veremos un ejemplo en el que si no durmiéramos unos instantes al hilo que realiza un cálculo, no le daría tiempo al hilo que dibuja el resultado a presentarlo en pantalla.
¿Cómo funciona el método sleep()? El método sleep() de la clase thread recibe como argumento el tiempo que deseamos dormir el hilo que lo invoca. Cuando transcurre el tiempo especificado, el hilo vuelve a estar "Ejecutable" ("Preparado") para continuar ejecutándose.
Hay dos formas de llamar a este método.
La primera le pasa como argumento un entero (positivo) que representa milisegundos: sleep(long milisegundos)
La segunda le agrega un segundo argumento entero (esta vez, entre 1 y 999999), que representa un tiempo extra en nanosegundos que se sumará al primer argumento: sleep(long milisegundos, int nanosegundos)
Cualquier llamada a sleep() puede provocar una excepción, que el compilador de Java nos obliga a controlar ineludiblemente mediante un bloque try-catch.

Parada de un Hilo

La forma natural de que muera o finalice un hilo es cuando termina de ejecutarse su método run(), pasando al estado 'Muerto'. Una vez que el hilo ha muerto, no lo puedes iniciar otra vez con start().
Si en tu programa deseas realizar otra vez el trabajo desempeñado por el hilo, tendrás que: Crear un nuevo hilo con new(). Iniciar el hilo con start().
Y ¿hay alguna forma de comprobar si un hilo no ha muerto? No exactamente, pero puedes utilizar el método isAlive() de la clase thread para comprobar si un hilo está vivo o no. Un hilo se considera que está vivo (alive) desde la llamada a su método start() hasta su muerte.
isAlive() devuelve verdadero (true) o falso (false), según que el hilo esté vivo o no. Devolverá true en caso de que el hilo t1 esté vivo, es decir, ya se haya llamado a su método run() y no haya sido parado con un stop() ni haya terminado el método run en su ejecución. En otro caso, lógicamente, devolverá false.
El método stop() de la clase thread (actualmente en desuso) también finaliza un hilo, pero es poco seguro. No debes utilizarlo. Te lo indicamos aquí simplemente porque puede que encuentres programas utilizando este método

5. Planificación y prioridades

5.1 Planificación (*Scheduling*)

Java tiene un Planificador (*Scheduler)*, una lista de procesos, que muestra por pantalla todos los hilos que se están ejecutando en todos los programas y decide cuáles deben ejecutarse y cuáles deben encontrarse preparados para su ejecución. Sus principales características son:

- Todos los hilos de Java tienen una prioridad y se supone que el planificador dará preferencia a aquel hilo que tenga una prioridad más alta. Sin embargo, no hay ningún tipo de garantía de que en un momento determinado el hilo de mayor prioridad se esté ejecutando.
- Los ciclos de tiempo pueden ser aplicados o no. Dependerá de la gestión de hilos que haga la librería sobre el que se implementa la máquina virtual java.

5.2 Prioridades

Las prioridades de cada hilo en Java están en el rango de 1 (MIN_PRIORITY) a 10 (MAX_PRIORITY). La prioridad de un hilo es inicialmente la misma que la del hilo que lo creó. Por defecto todo hilo tiene la prioridad 5 (NORM_PRIORITY). El planificador siempre pondrá en ejecución aquel hilo con mayor prioridad . Los hilos de prioridad inferior se ejecutarán cuando estén bloqueados los de prioridad superior. Las prioridades se pueden cambiar utilizando el método setPriority (nuevaPrioridad). La prioridad de un hilo en ejecución se puede cambiar en cualquier momento. El método getPriority() devuelve la prioridad de un hilo. El método yield()hace que el hilo actualmente en ejecución ceda el paso de modo que puedan ejecutarse otros hilos listos para ejecución. El hilo elegido puede ser incluso el mismo que ha dejado paso, si es el de mayor prioridad. En el siguiente programa se puede ver cómo se crean dos hilos (t1 y t2) y al primero de ellos se le cambia la prioridad. Esto haría que t2 acaparase el procesador hasta finalizar pues nunca será interrumpido por un hilo de menor prioridad.

```
public class A implements Runnable {
  String palabra;

 public A (String _palabra)  {
  palabra = _palabra;
 }
```

```
public void run () {
        for (int i=0;i<100;i++)
        System.out.println (palabra);
 }

public static void main (String args[]) {
        A a1 = new A("a1");
        A a2 = new A("a2");
        Thread t1 = new Thread (a1);
        Thread t2 = new Thread (a2);
        t1.start();
        t1.setPriority(1);
        System.out.println ("Prioridad de t1: "+t1.getPriority());
        t2.start();
        System.out.println ("Prioridad de t2: "+t2.getPriority());
}
}
```

6. Métodos importantes

6.1 Métodos de clase

Estos son los métodos estáticos que deben llamarse de manera directa en la clase Thread.

currentThread()
Este método devuelve el objeto thread que representa al hilo de ejecución que se esta ejecutando.

yield()
Este método hace que el interprete cambie de contexto entre el hilo actual y el siguiente hilo ejecutable disponible. Es una manera de asegurar que los hilos de menor prioridad no sufran inanición.

sleep(long)
El método sleep() provoca que el interprete ponga al hilo en curso a dormir durante el numero de milisegundos que se indiquen en el parámetro de invocación. Una vez transcurridos esos milisegundos, dicho hilo volverá a estar disponible para su ejecución.

IsAlive()
Devuelve true si el hilo ha sido arrancado(con start()) y no ha sido detenido (con stop()). Por ello, si el metodo isAlive() devuelve false, sabemos que estamos eante un Nuevo Thread o ante un thread Muerto. Si devuelve tru, se sabe que el hilo se encuentra en estado Ejecutable o Parado. No se puede diferenciar entre *Nuevo Thread* y *Muerto*,ni entre un hilo *Ejecutable* o *Parado*.

6.2 Métodos de Instancia

start()
Este método indica al interprete de Java que cree un contexto del hilo del sistema y comience a ejecutarlo. A continuación, el método run() de este hilo sera invocado en el nuevo contexto del hilo. Hay que tener precaución de no llamar al método start() mas de una vez sobre un hilo determinado.

run()
El método run() constituye el cuerpo de un hilo en ejecución. Este es el único método del interfaz Runnable. Es llamado por el método start() después de que el hilo apropiado del sistema se haya inicializado. Siempre que el método run() devuelva el control, el hilo actual se detendrá.

stop()
Este método provoca que el hilo se detenga de manera inmediata. A menudo constituye una manera brusca de detener un hilo, especialmente si este método se ejecuta sobre el hilo en curso. En tal caso, la linea inmediatamente posterior a la llamada al método stop() no llega a ejecutarse jamas, pues el contexto del hilo muere antes de que stop() devuelva el control. Una forma mas elegante de detener un hilo es utilizar alguna variable que ocasione que el método run() termine de manera ordenada. En realidad, nunca se debería recurrir al uso de este método.

suspend()
Este método es distinto al anterior. Suspend() toma el hilo y provoca que se detenga su ejecución sin destruir el hilo de sistema subyacente, ni el estado del hilo anteriormente en ejecución. Si la ejecución de un hilo se suspende, puede llamarse a resume() sobre el mismo hilo para lograr que vuelva a ejecutarse de nuevo.

resume()
el método resume() se utiliza para revivir un hilo suspendido. No hay garantías de que el hilo comience a ejecutarse inmediatamente, ya que puede haber un hilo de

prioridad mayor en ejecución actualmente, pero resume() ocasiona que el hilo vuelva a ser un candidato a ser ejecutado.

setPriority(int)
Este método asigna al hilo la prioridad indicada por el valor pasado como parámetro. Hay bastantes constantes predefinidas para la prioridad, definidas en la clase Thread, tales como MIN_PRIORITY, NORM_ PRIORITY y MAX_ PRIORITY, que toman los valores 1,5 y 10, respectivamente. Como guía aproximada de utilización, se puede establecer que la mayor parte de los procesos a nivel de usuario deberían tomar una prioridad en torno a NORM_ PRIORITY. Las tareas en segundo plano, como una entrada/salida a red o el nuevo dibujo de pantalla, deberían tener una prioridad cercana a MIN_ PRIORITY. Con las tareas a las que se fije máxima prioridad,hay que ser especialmente cuidadosos, porque si no se ,hacen llamadas a sleep() o yield(), se puede provocar que el interprete Java quede fuera de control.

getPriority()
Este método devuelve la prioridad del hilo de ejecución en curso, que es un valor comprendido entre uno y diez.

setName(String)
Permite identificar al hilo con un nombre nemonico. De esta manera se facilita la depuración de programas multihilo. El nombre aparecerá en todas las lineas de trazado que se muestran cada vez que el interprete de Java imprime excepciones no capturadas.

getName()
Devuelve el valor actual, de tipo cadena,asignado como nombre al hilo en ejecución mediante setName().

A continuación se muestran en la tabla los atributos y métodos de la clase Thread tratados en este capítulo y algunos más que considero de interés.

Atributos	
public static final int	**MIN_PRIORITY** La prioridad mínima que un hilo puede tener.
public static final int	**NORM_PRIORITY** La prioridad por defecto que se le asigna a un hilo**.**
public static final int	**MAX_PRIORITY** La prioridad máxima que un hilo puede tener.
Constructores	
public	**Thread** () Crea un nuevo objeto Thread. Este constructor tiene el mismo efecto que Thread (null, null, gname), donde ***gname*** es un nombre generado automáticamente y que tiene la forma "Thread-"+n, donde n es un entero asignado consecutivamente
public	**Thread** (String name) Crea un nuevo objeto Thread, asignándole el nombre ***name***.
public	**Thread** (Runnable target) Crea un nuevo objeto Thread. ***target*** es el objeto que contiene el método run() que será invocado al lanzar el hilo con start().
public	**Thread** (Runnable target, String name) Crea un nuevo objeto Thread, asignándole el nombre ***name***. ***target*** es el objeto que

	contiene el método run() que será invocado al lanzar el hilo con start().
Métodos	
public static Thread	**currentThread** () Retorna una referencia al hilo que se está ejecutando actualmente.
public static void	**dumpStack** () Imprime una traza del hilo actual. Usado sólo con propósitos de depuración..
public String	**getName** () Retorna el nombre del hilo.
int	**getPriority** () Retorna la prioridad del hilo.
public final boolean	**isAlive** () Chequea si el hilo está vivo. Un hilo está vivo si ha sido lanzado con start y no ha muerto todavía.
public final void	**isDaemon** () Devuelve verdadero si el hilo es daemon.
public final void	**join** () throws InterruptedException Espera a que este hilo muera.
public final void	**join** (long millis) throws InterruptedException Espera como mucho ***millis*** milisegundos para que este hilo muera.
public final void	**join** (long millis, int nanos) throws InterruptedException

	Permite afinar con los nanosegundos ***nanos*** el tiempo a esperar.
public final void	**join** (long millis, int nanos) throws InterruptedException Permite afinar con los nanosegundos ***nanos*** el tiempo a esperar.
public void	**run()** Si este hilo fue construido usando un objeto que implementaba Runnable, entonces el método run de ese objeto es llamado. En cualquier otro caso este método no hace nada y retorna.
public final void	**setDaemon** (boolean on) Marca este hilo como daemon si el parámetro ***on*** es verdadero o como hilo de usuario si es falso. El método debe ser llamado antes de que el hilo sea lanzado.
public final void	**setName** (String name) Cambia el nombre del hilo por ***name***.
public final void	**setPriority** (int newPriority) Asigna la prioridad ***newPriority*** a este hilo.
public static void	**sleep** (long millis) throws InterruptedException Hace que el hilo que se está ejecutando actualmente cese su ejecución por los milisegundos especificados en ***millis***. Pasa al estado dormido. El hilo no pierde la propiedad de ningún cerrojo que tuviera adquirido con *synchronized*.
public static void	**sleep** (long millis, int nanos) throws InterruptedException

	Permite afinar con los nanosegundos ***nanos*** el tiempo a estar dormido.
public void	**start** () Hace que este hilo comience su ejecución. La JVM llamará al método run de este hilo.
public String	**toString** () Devuelve una representación en forma de cadena de este hilo, incluyendo su nombre, su prioridad y su grupo.
public static void	**yield** () Hace que el hilo que se está ejecutando actualmente pase al estado listo, permitiendo a otro hilo ganar el procesador.

7. Instrumentos de Sincronización

Cuando en un programa tenemos varios hilos corriendo simultáneamente es posible que varios hilos intenten acceder a la vez a un mismo sitio (un fichero, una conexión, un array de datos) y es posible que la operación de uno de ellos entorpezca la del otro. Para evitar estos problemas, hay que sincronizar los hilos.

La importancia de la sincronización

La programación concurrente puede dar lugar a muchos errores debido a la utilización de recursos compartidos que pueden ser alterados. Las secciones de código potencialmente peligrosas de provocar estos errores se conocen como secciones críticas.

En general los programas concurrentes deben ser correctos totalmente. En estos programas por lo tanto hay que evitar que varios hilos entren en una sección critica (exclusión mutua o mutex) y que los programas se bloqueen (deadlock). Además los programas que no terminan nunca (programas reactivos) deben cumplir la ausencia de inanición (starvation) esto es que tarde o temprano todos los hilos alcancen el control del procesador y ninguno quede indefinidamente en la lista de hilos listos y también deben cumplir la propiedad de equitatividad (fairwess) esto es repartir el tiempo de la forma mas justa entre todos los hilos.

Java proporciona un soporte único,el monitor, es un objeto que se utiliza como cerrojo exclusivo. Solo uno de los hilos puede ser el propietario de un monitor en un instante dado. Los restantes hilos que estuviesen intentando acceder al monitor bloqueado quedan en suspenso hasta que el hilo propietario salga del monitor. Todos los objetos de Java disponen de un monitor propio implicitamente asociado a ellos. La manera de acceder a un objeto monitor es llamando a un método marcado con la palabra clave synchronized.

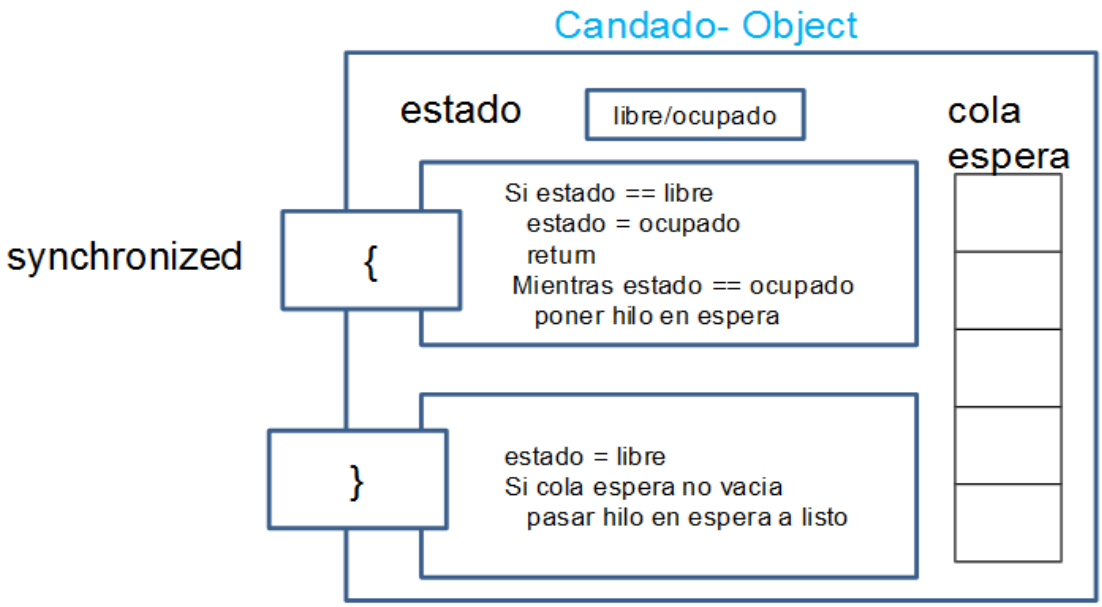
Un monitor impide que varios hilos accedan al mismo recurso compartido a la vez. Cada monitor incluye un protocolo de entrada y un protocolo de salida. En Java los monitores se consiguen mediante el uso de la palabra reservada synchronized bien como instrucción de bloque o bien como modificador de acceso de métodos. Cuando se utiliza la palabra clave synchronized se esta indicando una zona restringida para el uso de Threads, esta zona restringida para efectos prácticos puede ser considerada un candado (“lock”) sobre la instancia del objeto en cuestión.

Lo anterior implica que si es invocado un método synchronized únicamente el Thread que lo invoca tiene acceso a la instancia del objeto, y cualquier otro Thread

que intente acceder a esta misma instancia tendrá que esperar hasta que sea terminada la ejecución del método synchronized.

- **CANDADO (Object)**

Todos los objetos tienen un "candado" (lock). Solo un hilo puede tener bloqueado el candado de un objeto en un momento dado. Si un hilo intenta obtener un candado ocupado, quedará suspendido hasta que éste se libere y pueda obtenerlo.



Como trabajar con candados:

- Cada objeto tiene solo un candado.
- Solo métodos (o bloques) pueden ser sincronizados, ni las variables ni las clases.
- No todos los métodos en una clase tiene que ser sincronizados.
- Si dos hilos están a punto de ejecutar un método sincronizado en una clase, solo un hilo podrá ejecutar el método. El otro método tendrá que esperar hasta que el primero termine su invocación. En otras palabras, una vez que un hilo adquiere el candado de un objeto, otro hilo no podrá entrar a ningún método sincronizado de esa clase (de ese objeto).

- Si una clase tiene métodos sincronizados y no sincronizados, múltiples hilos podrán acceder a sus métodos no sincronizados. Si tienes métodos que no acceden a los datos que estas intentando proteger, entonces no necesitas protegerlos porque ellos no necesitan ser protegidos.

Ya que la sincronización daña la concurrencia, no interesa sincronizar el código mas de lo necesario para proteger los datos.

```
class sincro {
   public void Ejemplo() {
      System.out.println("no sincro");
      synchronized(this) {
         System.out.println("sincro");
      }
   }
}
```

Cuando en un hilo se ejecuta código de un bloque sincronizado, incluyendo cualquier método invocado de ese método sincronizado, al código se le dice que se ejecuta en un contexto sincronizado. La pregunta real es, sincronizado dónde? O sincronizado en el candado de qué objeto?

Cuando se sincroniza un método, el objeto usado para invocar el método es el objeto cuyo candado debe ser adquirido. Pero cuando se sincroniza un bloque de código, se especifica el objeto que se usa como candado, así por ejemplo, se puede usar un tercer objeto como candado para esa pieza de código.

También se puede sincronizar la instancia actual (this) en el código de arriba. Ya que esa es la misma instancia que la del método sincronizado, significa que se puede siempre reemplazar un método sincronizado con un método no sincronizado contenido en el método sincronizado, en otras palabras, esto:

```
public synchronized void Ejemplo() {
System.out.println("sincro");
}
```

es lo mismo que esto:

```
public void Ejemplo() {
synchronized(this) {
System.out.println("sincro");
}
}
```

Resumiendo y agregando algunas cosas mas podemos decir:
•Es posible garantizar la ejecución en exclusión mutua de un método definiéndolo como synchronized.
• Los métodos synchronized bloquean el cerrojo del objeto actual, o del objeto Class si el método es estático.
• Si el cerrojo está ocupado, el hilo se suspende hasta que éste es liberado.
• No se ejecutarán simultáneamente dos métodos synchronized de un mismo objeto, pero sí uno que lo sea y cualquier número de otros que no.
•Puede bloquearse el cerrojo de un objeto dentro de una porción de código mediante synchronized(objeto) { }.

•SEMÁFOROS

El nombre de semáforos es como en la vida real, un semáforo cerrado no podrán pasar coches hacia un lado, y un semáforo abierto sí podrán. Los semáforos garantizan la exclusión mutua y la sincronización (para que los coches no se choquen en la región crítica que en este caso es el cruce). En semáforos nos ayudamos de varios métodos como por ejemplo:

- •acquire(): Para adquirir el semáforo (ponerlo en verde para la cola de coches A) una vez que lo hemos adquirido pueden pasar los coches porque está en verde.
- •release(): El último coche en pasar hace un release() para que los coches que están esperando del otro semáforo puedan pasar ya que nosotros hemos terminado.

Los semáforos se usan para controlar el número de hilos que pueden acceder a un recurso.

Los semáforos son una de las soluciones clásicas al problema de concurrencia. Concebidos en 1965 por Dijkstra, son implementados en java a través del objeto ***java.util.concurrent.Semaphore***.

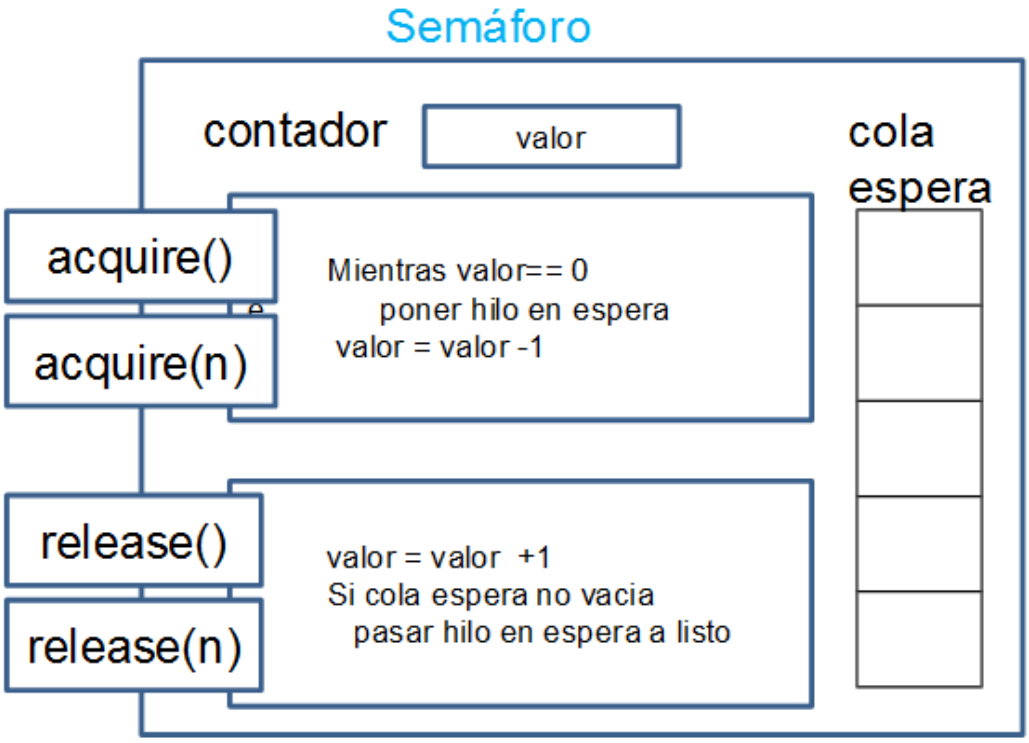
Formalmente un semáforo es un elemento que lleva en su interior un contador que indica la cantidad inicial de autorizaciones de paso o "banderitas". Un proceso al querer entrar a una *región crítica* adquiere una banderita del semáforo y cuando la deja lo libera. Si existen banderitas disponibles, entonces el proceso continua su ejecución, de lo contrario queda bloqueado a la espera que otro proceso libere alguna banderita y se pueda continuar.

Se empieza por inicializar la posición de memoria a 1 (o al valor correspondiente si ese recurso concreto admite más de un acceso simultáneo). Esto se hace en el inicio del programa principal.

A continuación, cada vez que un thread o un proceso quiera acceder a dicho recurso (por ejemplo, un fichero), hará primero una petición con la primera de las llamadas disponibles. Cuando el S.O. ejecuta esa llamada, comprueba el valor que hay en la posición de memoria del semáforo, y si es distinta de cero, se limita a restarle 1 y devolver el control al programa; sin embargo, si ya es cero, duerme al proceso que hizo la petición y lo mete en la cola de procesos, en espera de que el semáforo se ponga a un valor distinto de cero.

Por último, cuando el proceso ha terminado el acceso al recurso, usa la segunda llamada para liberar el semáforo. Cuando el S.O. la ejecuta, comprueba si la cola del semáforo está vacia, en cuyo caso se limita a incrementar el valor del semáforo, mientras que si tiene algún proceso, lo despierta, de modo que vuelve a recibir ciclos de CPU y sigue su ejecución. Si había varios procesos en espera, se irán poniendo en marcha uno tras otro a medida que el anterior va liberando el semáforo. Cuando termina el último, el semáforo se vuelve a poner a 1. Se trata, por tanto, del mismo proceso que seguiríamos con la variable, pero con la ventaja de que es un mecanismo estandar para todos los procesos, y como es una operacion *atómica* (esto es, que durante su ejecución no se admiten cambios de tarea), no surje el problema de que una conmutación pueda producir errores aleatorios.

Vemos que la primera vez que un proceso usa el semáforo, este tiene valor 1, por lo que pasa a cero y el proceso puede acceder al recurso. Si durante ese tiempo otro proceso quiere acceder también, al usar el semáforo, este tiene valor cero, por lo que el S.O. deja de darle ciclos de CPU. Cuando el primer proceso ha terminado, libera el recurso, con lo que el S.O. puede comprobar que el segundo proceso está esperando, por lo que le vuelve a dar ciclos. En este punto, el proceso sigue como si nunca hubiese sido detenido.



Veamos un ejemplo de utilizacion de los semaforos:

```
class MiHebra extends Thread {
private static Semaphore mutex = new Semaphore(1);
private static int i = 0;
public void run() {
while(running) {
try {
mutex.acquire();
i++;
mutex.release();
}
catch(InterruptedException e) { }
}
}
}
```

·COMUNICACIÓN ENTRE HILOS

Cuando dos o más hilos son interdependientes, deben decidir entre ellos cuándo esperar y cuándo avanzar. Los mecanismos anteriores sirven para evitar la interferencia entre hilos. Es necesario algún método de comunicación entre ellos. Todos los objetos implementan los métodos wait () y notify (). Mediante wait () se suspende el hilo actual hasta que algún otro llame al método notify () del mismo objeto.

La clase Object tiene tres métodos, wait(), notify() y notifyAll(), que ayuda a los hilos para que puedan comunicar el estatus de un evento al que el hilo esta atendiendo .wait(), notify() y notifyAll() deben ejecutarse en exclusión mutua (en métodos sincronizados).

```
synchronized void hacerMientrasCondicion() {
while(!condicion) wait();
/* ... */
}
synchronized void cambiarCondicion() {
/*...*/
notify();
}
```

Por ejemplo, si un hilo es un reparto de email y otro es un procesador de email, el procesador de mail tiene que esperar para ver si se procesa el email. Usando el método notify() y wait(), el hilo que podría procesar la verificación del email, y si no encuentra ninguno, puede decir "No voy a malgastar mi tiempo verificando un email cada dos segundos. Cuando el email llegue, que me avisen, y asi voy a verificarlo". En otras palabras usar wait() y notify() te permite poner el hilo en la sala de espera, hasta que otro hilo notifique la razón para salir.

Sistema de parada y avance
• Cuando un hilo A depende de que otro B complete una tarea, se para a esperar
• Cuando el hilo B completa su tarea, B avisa a A para que continúe su ejecución

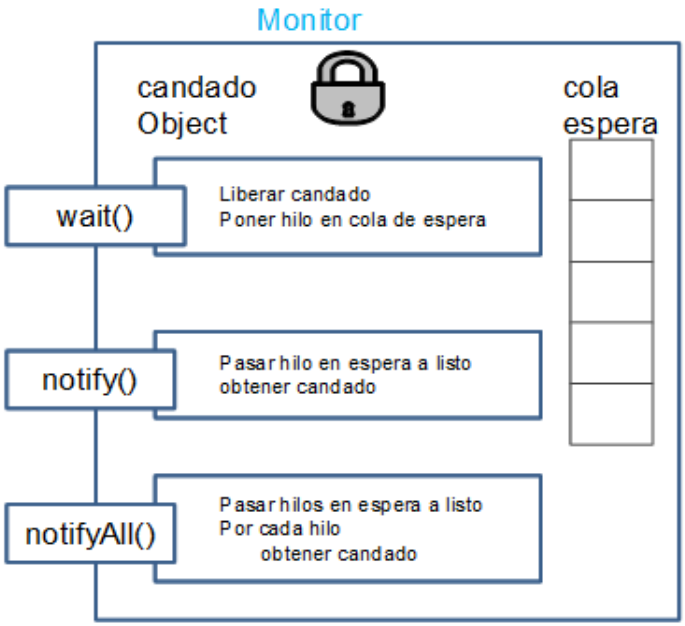
Todos los objetos implementan los siguientes métodos referidos a su monitor interno:

• **wait()** Detiene la ejecución del hilo hasta recibir una notificación. Bloquea al hilo que lo invoca hasta que el hilo sea interrumpido o hasta que otro hilo notifique que algún evento ha ocurrido (con notify() o notifyAll()). Aunque no necesariamente el evento que espera el hilo que invocó **wait().** También puede ocurrir que la condición que esperara haya dejado de cumplirse.

Para invocar el método **wait** es necesario tener el control del monitor… por tanto, siempre debe activarse dentro de un método sincronizado.

• **notify()** Despierta a uno de los hilos detenidos. ¡No se puede controlar especificamente a cual!

• **notifyAll()** Despierta a todos los hilos detenidos. Todos pasan a estar "en espera" hasta que el planificador les ceda el turno.



Como implementar la comunicación entre hilos.

No se garantiza que, cuando un hilo despierta, lo hace porque ya se cumple la condición a la que estaba esperando, por lo que la llamada a wait se debe hacer en un while, en vez de en un if.

En el hilo que espera:
```
while(!condicion) {
try {
[this.]wait();
} catch (InterruptedException e) {}
}
```
En el otro hilo:
```
t.notifyAll();
```

En este ejemplo vamos a tener 3 clases: una clase Main, una clase Saludo y una clase Personal (Hilo).

Clase Main:

```
public class Main {

	public static void main(String[] args) {

		// Objeto en comun, se encarga del wait y notify
			Saludo s = new Saludo();
		 /*Instancio los hilos y le paso como parametros:
		* El Nombre del Hilo(en este caso es el nombre del personal)
		* ----El objeto en comun (Saludo)----
		 * Un booleano para verificar si es jefe o empleado
		*
		*/
		Personal Empleado1 = new Personal("Pepe", s, false);
		Personal Empleado2 = new Personal("José", s, false);
		Personal Empleado3 = new Personal("Pedro", s, false);
		Personal Jefe1 = new Personal("JEFE", s, true);

		//Lanzo los hilos
		 Empleado1.start();
		 Empleado2.start();
		Empleado3.start();
		Jefe1.start();

	}
}
```

Clase Personal:

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

public class Personal extends Thread{
 String nombre;
	 Saludo saludo;
	boolean esJefe;

	public Personal(String nombre, Saludo salu, boolean esJefe){
		this.nombre = nombre;
```

```
                this.saludo = salu;
                this.esJefe = esJefe;
                 }
         public void run(){
        System.out.println(nombre + " llegó.");
        try {
                Thread.sleep(1000);
                //Verifico si es personal que esta es jefe o no
        if(esJefe){
         saludo.saludoJefe(nombre);
        }else{
        saludo.saludoEmpleado(nombre);
        }

         } catch (InterruptedException ex) {
                 Logger.getLogger(Personal.class.getName()).log(Level.SEVERE, null, ex);
        }
        }
}
```

Clase Saludo:

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

public class Saludo {

 public Saludo(){
 }

 /* Si no es jefe, el empleado va a quedar esperando a que llegue el jefe
 Se hace wait de el hilo que esta corriendo y se bloquea, hasta que se le avise que ya puede saludar*/
 public synchronized void saludoEmpleado(String nombre){
 try {
 wait();
 System.out.println("\n"+nombre.toUpperCase() + "-: Buenos días jefe.");
 } catch (InterruptedException ex) {
 Logger.getLogger(Saludo.class.getName()).log(Level.SEVERE, null, ex);
}
 }

 //Si es jefe, saluda y luego avisa a los empleados para que saluden     // El notifyAll despierta a todos los
hilos que esten bloqueados
 public synchronized void saludoJefe(String nombre){
```

```
 System.out.println("\n****** "+nombre + "-: Buenos días empleados. ******");
notifyAll();
}
}
```

Salida del programa por consola:

```
Pepe llegó.
José llegó.
Pedro llegó.
JEFE llegó.

****** JEFE-: Buenos días empleados. ******

PEPE-: Buenos días jefe.

JOSÉ-: Buenos días jefe.

PEDRO-: Buenos días jefe.
```

Funcionamiento de un monitor
Respecto a la sincronización:

- la exclusión mutua se asegura por definición
  - por lo tanto, sólo un proceso puede estar ejecutando acciones de un monitor en un momento dado
  - aunque varios procesos pueden en ese momento ejecutar acciones que nada tengan que ver con el monitor
- la sincronización condicionada
  - con frecuencia es necesaria una sincronización explícita entre procesos
  - para ello, se usarán las variables "condición"

El paquete *java.util.concurrent* dispone de constructores de concurrencia más potentes e, incluso, de variables "condición".

- Todo objeto Java tiene asociado un *lock* que a su vez tiene asociado un *wait set (*conjunto de threads bloqueados a nivel de objeto)
- Un método *synchronized* automáticamente toma y deja el *lock*
- Semántica de las operaciones:
  - *wait()* :: el proceso que invoca el método se bloquea y es añadido al *wait set* del objeto liberando inmediatamente su *lock* (*Interrupted exception* debe ser capturada)
  - *notify()* :: desbloquea un proceso cualquiera del *wait set* del objeto
  - *notifyAll()* :: desbloquea todos los procesos del *wait set* del objeto
- Para invocar cualquiera de las operaciones anteriores el proceso invocador debe tener el *lock* del objeto, sólo desde métodos *synchronized .*

Cuando un proceso es desbloqueado por una operación *notify()* o *notifyAll()* debe volver a evaluar la condición que provocó su bloqueo
•isu valor ha podido cambiar desde que se produjo el desbloqueo!

```
synchronized método(…) {
… while(!condición)
        wait();
 //condición TRUE … }
```

```
class Buffer {
private Object dato = null;
 private boolean hayDato = false;

public synchronized void poner(Object nuevoDato) throws
InterruptedException
{
 while(hayDato) {
 wait(); }
 // !hayDato
dato = nuevoDato;
 hayDato = true;
 notifyAll(); }
```

```
public synchronized Object quitar() {
 throws InterruptedException
 {
 while(!hayDato)
{ wait(); }
// hayDato
notifyAll();
hayDato = false;
 return dato; }
```