

PROGRAMACIÓN MULTITHILO.

2.4. LA CLASE THREAD.

Los métodos para gestionar los hilos son:

| METODO | ACCION |
|---------------------------------|---|
| void start() | Provoca la llamada al método run() para que dé comienzo la ejecución del hilo. |
| void run() | El hilo comienza su ejecución tras un start(). Independientemente de que haya sido construido a partir de la interfaz Runnable o de la clase Thread. |
| String getName() | Devuelve el nombre del hilo |
| void setName(string nombre) | asigna el nombre al hilo |
| int getPriority() | Devuelve la prioridad de un hilo |
| void setPriority(int prioridad) | asigna la prioridad indicada al hilo. Cada hilo tiene una prioridad, que es un valor entero entre 1 y 10, de modo que cuanto mayor sea el valor, mayor es la prioridad. |
| boolean isAlive() | Devuelve true si está en ejecución y false en caso contrario. Un hilo está vivo si ha sido lanzado con start() y no ha muerto todavía. |
| void resume() | reanuda la ejecución de un hilo suspendido. Método obsoleto |
| void sleep(long milseg) | hace que el thread actual pase del estado ejecutable a dormido y permanezca en dicho estado durante los milisegundos especificados como parámetro. Una vez que se ha cumplido el tiempo, el thread despierta y pasa automáticamente al estado de ejecutable. Este método puede lanzar una InterruptedException , por lo tanto, las llamadas hacia él deben envolverse en un bloque try ... catch |
| void stop() | detiene la ejecución de un hilo. Método obsoleto |
| void suspend() | este método suspende un hilo, su estado pasa de ejecutable a suspendido inmediatamente y sólo puede ser reactivado (pasado al estado ejecutable) llamando a su método resume(). Método obsoleto |
| Thread currentThread() | devuelve una referencia al hilo que se está ejecutando actualmente. |

| | |
|--|---|
| <code>void isDaemon()</code> | devuelve verdadero si el hilo es daemon |
| <code>void join()</code> | Hace que el <i>thread</i> que se está ejecutando actualmente pase al estado “esperando” indefinidamente hasta que muera el <i>thread</i> sobre el que se realiza el <code>join()</code> . |
| <code>void join(long miliseg)</code> | espera como mucho los milisegundos indicados para que el hilo muera |
| <code>void setDaemon (boolean on)</code> | marca el 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. Los hilos demonio están supeditados a los hilos que los han creado, de tal manera que cuando el creador termina, sus hijos “demonio” también finalizan. |
| <code>String toString()</code> | devuelve una representación en forma de cadena del hilo, incluyendo su nombre, prioridad y grupo |
| <code>void yield()</code> | hace que el hilo que se está ejecutando actualmente pase al estado listo, permitiendo a otro hilo ganar el procesador. |
| <code>void destroy()</code> | destruye el hilo sin realizar ningún tipo de limpieza |
| <code>void interrupt()</code> | interrumpe la ejecución del hilo |
| <code>boolean interrupted()</code> | comprueba si el hilo actual ha sido interrumpido |
| <code>void wait()</code> <code>void wait(long miliseg)</code> | pondría el hilo en el estado “esperando” indefinidamente, hasta que el thread reciba un <i>notify()</i> o <i>notifyAll()</i> . si le indicamos un tiempo estará esperando durante ese tiempo. |

CONSTRUCTORES

| | |
|---|---|
| <code>public Thread()</code> | crea un nuevo objeto Thread. |
| <code>public Thread (String nombre)</code> | crea un nuevo objeto Thread asignándole el nombre indicado |
| <code>public Thread (Runnable target)</code> | crea un nuevo objeto Thread. target es el objeto que contiene el método run() que será invocado al lanzar el hilo con start() |
| <code>public Thread (Runnable target, String name)</code> | crea un nuevo objeto Thread, asignándole el nombre indicado. target es el objeto que contiene el método run() que será invocado al lanzar el hilo con start() |

ATRIBUTOS

| | |
|-------------------|---|
| int MIN_PRIORITY | la prioridad mínima que un hilo puede tener |
| int NORM_PRIORITY | la prioridad por defecto que se le asigna a un hilo |
| int MAX_PRIORITY | la prioridad máxima que un hilo puede tener. |

2.5. OBTENER EL HILO PRINCIPAL.

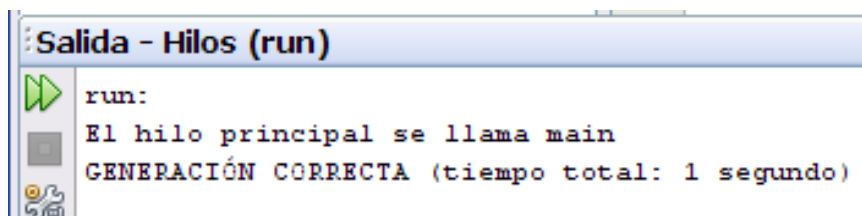
Todo programa Java tiene al menos un hilo, llamado *hilo principal* y que podemos observar con el método *currentThread*, que obtiene el hilo actual.

Este hilo es especial por dos razones:

- Desde él se crearán el resto de hilos del programa.
- Debe ser el último hilo que termine su ejecución. (Si un hilo principal finaliza antes que un hijo Java puede bloquearse, hang).

El método *run()* es el punto de entrada de un nuevo hilo de ejecución concurrente dentro de un programa. El hilo termina cuando finalice el método *run()*.

```
public class HiloPrincipal {  
    public static void main (String args[]) {  
        Thread hilo = Thread.currentThread();  
  
        System.out.println("El hilo principal se llama "+hilo.getName());  
    }  
}
```



2.6. MÚLTIPLES PROCESOS.

Podemos crear y ejecutar múltiples hilos en un mismo programa: basta con dar a cada hilo un nuevo objeto.

A continuación tienes un ejemplo en el que se crean 4 hilos y esperamos a que termine cada uno antes de finalizar la aplicación principal, cada uno imprima su nombre una vez por segundo, obtenemos además una ejecución secuencial ordenada.

```

Primero está ejecutándose...
Primero está ejecutándose...
Primero está ejecutándose...
Primero está ejecutándose...
Primero ha finalizado.
Segundo está ejecutándose...
Segundo está ejecutándose...
Segundo está ejecutándose...
Segundo está ejecutándose...
Segundo ha finalizado.
Tercero está ejecutándose...
Tercero está ejecutándose...
Tercero está ejecutándose...
Tercero está ejecutándose...
Tercero ha finalizado.
Cuarto está ejecutándose...
Cuarto está ejecutándose...
Cuarto está ejecutándose...
Cuarto está ejecutándose...
Cuarto ha finalizado.
FIN DE LA APLICACION PRINCIPAL

```

```

class Multiples extends Thread{
    //constructor
    public Multiples (String nombre){
        super(nombre);
    }

    //redefinición del método run(), que es el que contiene
    //las indicaciones de lo que hará el hilo
    public void run(){
        try{
            for (int i=0;i<4;i++){
                //Mostrará el nombre del hilo actual
                System.out.println((Thread.currentThread()).getName()+" está ejecutándose.....");
                //Duermo el hilo actual un segundo
                Thread.sleep(1000);
            }
        }catch (InterruptedException ex){}
        System.out.println((Thread.currentThread()).getName()+" ha finalizado.");
    }
}

```

```

public class MainMultiples{
    public static void main(String args[]){

        Multiples hilo1 = new Multiples("Primero");
        Multiples hilo2 = new Multiples("Segundo");
        Multiples hilo3 = new Multiples("Tercero");
        Multiples hilo4 = new Multiples("Cuarto");

        try{

            hilo1.start();
            hilo1.join();

            hilo2.start();
            hilo2.join();

            hilo3.start();
            hilo3.join();

            hilo4.start();
            hilo4.join();

        }catch (InterruptedException ex){}

        System.out.println("Fin de la aplicación principal");
    }
}

```

3.AGRUPAMIENTO DE HILOS.

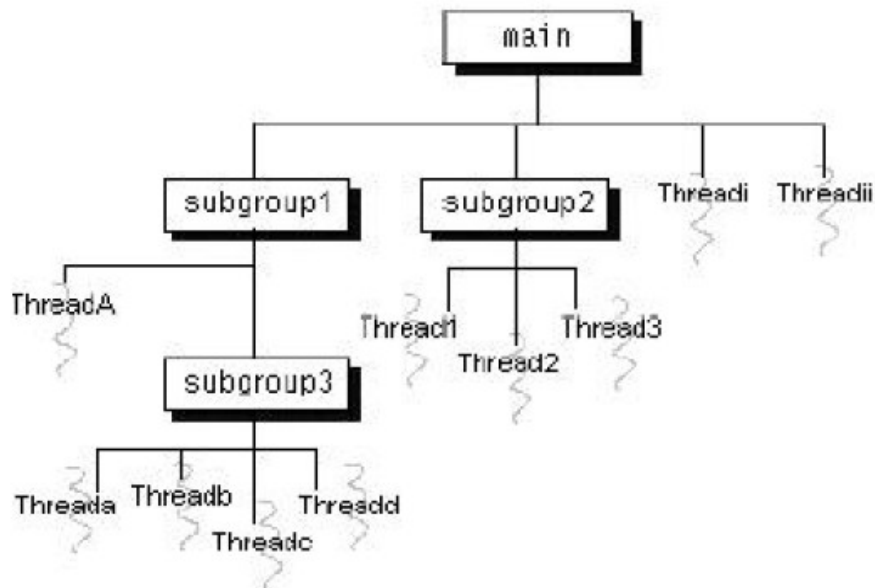
3.1 GRUPOS DE HILOS.

Todo hilo de Java es un miembro de un grupo de hilos. Los grupos de hilos proporcionan un mecanismo de reunión de múltiples hilos dentro de un único objeto y de manipulación de dichos hilos en conjunto, en lugar de una forma individual.

Por ejemplo, se pueden arrancar o suspender todos los hilos que están dentro de un grupo con una única llamada al método. Los grupos de hilos de Java están implementados por la clase `ThreadGroup` en el paquete `java.lang`.

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

Una vez que un hilo ha sido asociado a un grupo de hilos, no puede cambiar de grupo.



3.2. CREACIÓN DE UN HILO EN UN GRUPO DE FORMA EXPLÍCITA.

Como hemos mencionado anteriormente, un hilo es un miembro permanente de aquel grupo de hilos al cual se unió en el momento de su creación (no tenemos la posibilidad de cambiarlo posteriormente). De este modo, si quieres poner tu nuevo hilo en un grupo de hilos distinto del grupo por defecto, debes especificarlo explícitamente cuando lo creas.

Para conseguir que un hilo pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo hilo, 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)`

Cada uno de estos constructores crea un nuevo hilo, lo inicializa en base a los parámetros `Runnable` y `String`, y hace al nuevo hilo miembro del grupo especificado.

Por ejemplo, la siguiente muestra de código crea un grupo de hilos (`myThreadGroup`) y entonces crea un hilo (`myThread`) en dicho grupo

```
ThreadGroup miGrupoHilo = new ThreadGroup("Mi grupo hilos ");
```

```
Thread miHilo = new Thread(miGrupoHilos, "un hilo para mi grupo" );
```

El `ThreadGroup` pasado al constructor `Thread` no tiene que ser necesariamente un grupo que hayas creado tú, puede tratarse de un grupo creado por el sistema de ejecución de Java, o un grupo creado por la aplicación en la cual se está ejecutando un *applet*.

3.3 LA CLASE THREADGROUP

La clase `ThreadGroup` es la implementación del concepto de grupo de hilos en Java. Ofrece, por tanto, la funcionalidad necesaria para la manipulación de grupos de hilos para las aplicaciones Java. Un objeto `ThreadGroup` puede contener cualquier número de hilos. Los hilos de un mismo grupo generalmente se relacionan de algún modo, ya sea por quién los creó, por la función que llevan a cabo, o por el momento en que deberían arrancarse y parar.

El grupo de hilos de más alto nivel en una aplicación Java es el grupo de hilos denominado `main`. La clase `ThreadGroup` tiene métodos que pueden ser clasificados como sigue:

- *Collection Management Methods* (Métodos de administración del grupo): métodos que manipulan la colección de hilos y subgrupos contenidos en el grupo de hilos.
- *Methods That Operate on the Group* (Métodos que operan sobre el grupo): estos métodos establecen u obtienen atributos del objeto `ThreadGroup`.
- *Methods That Operate on All Threads within a Group* (Métodos que operan sobre todos los hilos dentro del grupo): este es un conjunto de métodos que desarrollan algunas operaciones, como inicio y reinicio, sobre todos los hilos y subgrupos dentro del objeto `ThreadGroup`.
- *Access Restriction Methods* (Métodos de restricción de acceso): `ThreadGroup` y `Thread` permiten al administrador de seguridad restringir el acceso a los hilos en base a la relación de miembro/grupo con el grupo

Métodos de administración del grupo

La clase `ThreadGroup` proporciona un conjunto de métodos que manipulan los hilos y los subgrupos que pertenecen al grupo y permiten a otros objetos solicitar información sobre sus miembros. Por ejemplo, puedes llamar al método `activeCount` de `ThreadGroup` para conocer el número de hilos activos que actualmente hay en el grupo. El método `activeCount` se usa generalmente con el método `enumerate` para obtener un vector (array) que contenga las referencias a todos los hilos activos en un `ThreadGroup`.

Métodos que operan sobre el grupo

La clase `ThreadGroup` da soporte a varios atributos que son establecidos y recuperados de un grupo de forma global (hacen referencia al concepto de grupo, no a los hilos individualmente).

Se incluyen atributos como la prioridad máxima que cualquiera de los hilos del grupo puede tener, el carácter “*daemon*” o no del grupo, el nombre del grupo, y el padre del grupo.

Los métodos que recuperan y establecen los atributos de `ThreadGroup` operan a nivel de grupo. Consultan o cambian el atributo del objeto de la clase `ThreadGroup`, pero no hacen efecto sobre ninguno de los hilos pertenecientes al grupo. La siguiente es una lista de métodos de `ThreadGroup` que operan a nivel de grupo:

- `getMaxPriority` y `setMaxPriority`
- `getDaemon` y `setDaemon`
- `getName`
- `getParent` y `parentOf`
- `toString`

Métodos que operan sobre todos los hilos de un grupo

La clase `ThreadGroup` tiene tres métodos que te permiten modificar el estado actual de todos los hilos pertenecientes al grupo:

- `resume`
- `stop`
- `suspend`

Estos métodos suponen el cambio correspondiente de estado para todos y cada uno de los hilos del grupo, así como los de sus subgrupos. No se aplican, por tanto, a un nivel de grupo, sino que se aplican individualmente a todos los miembros.

Métodos de restricción de acceso

La clase `ThreadGroup` no impone ninguna restricción de acceso por sí sola, como permitir a los hilos de un grupo consultar o modificar los hilos de un grupo diferente. En lugar de esto, las clases `Thread` y `ThreadGroup` cooperan con los administradores de seguridad (subclases de la clase `SecurityManager`), la cual impone las restricciones de acceso basándose en la pertenencia de los hilos a los grupos.

SUSPENSIÓN Y PARADA DE HILOS

Existen los métodos **suspend()** y **resume()** para la suspensión y reactivación de hilos. No obstante, están obsoletos y en desuso porque puede generar situaciones de interbloqueo. Por ejemplo, si un hilo está bloqueando un recurso y es suspendido, puede dar lugar a que el resto de hilos que necesitan el recurso, se queden esperando de forma indefinida.

Para suspender de forma segura un hilo, se deben emplear variables de control, cuyo valor se comprobará en el método `run()`. El valor de dicha variable será modificado en función del estado en el que se quiera dejar el hilo.

```
class MyHilo extends Thread {
    private SolicitaSuspende suspender = new SolicitaSuspende();

    public void Suspende() {
        suspender.set(true);
    }

    public void Reanuda() {
        suspender.set(false);
    }

    public void run () {
        try {
            while (no termine la condición) {
                ...
                suspender.esperandoParaReanudar();
            }
        }
    }
}
```

```

class SolicitaSuspend {
    private boolean suspender;
    public synchronized void set (boolean b) {
        suspender = b;
        notifyAll();
    }

    public synchronized void esperandoParaReanudar() throws InterruptedException{
        while (suspender) {
            wait(); // SUSPENDE HILO HASTA RECIBIR NOTIFY O NOTIFYALL
        }
    }
}

```

La variable de control se envuelve en la clase SolicitaSuspend, modificando su valor a través del método set(). Cuando esto ocurre, se realiza un notifyAll para notificar del cambio de estado a los hilos que están esperando (wait()).

El método wait() únicamente puede ser llamado desde un método sincronizado.

Para realizar la parada de un hilo, a pesar de que existe el método **stop()**, éste también está obsoleto y en desuso, por lo que se deberá gestionar mediante variables, como lo hemos visto hasta ahora.

Con el método **interrupt()**, se envía una petición de interrupción a un hilo. Si éste se encuentra bloqueado por una llamada a sleep() o a wait(), se lanzará una excepción *InterruptedException*. El método **isInterrupted()** devuelve true si el hilo ha sido interrumpido, o false en caso contrario.

```

public class HiloEjemploInterrup extends Thread {
    public void run() {
        try {
            while (!isInterrupted()) {
                System.out.println("En el Hilo");
                Thread.sleep(10);
            }
        } catch (InterruptedException e) {
            System.out.println("HA OCURRIDO UNA EXCEPCIÓN");
        }

        System.out.println("FIN HILO");
    } // run

    public void interrumpir() { interrupt(); } // interrumpir

    public static void main(String[] args) {
        HiloEjemploInterrup h = new HiloEjemploInterrup();
        h.start();
        for(int i=0; i<1000000000; i++) ; // no hago nada
        h.interrumpir();
    } //
} //

```

Cuya salida es:

```
En el hilo
En el hilo
HA OCURRIDO UNA EXCEPCIÓN
FIN HILO
```

4. PLANIFICACIÓN Y PRIORIDAD DE LOS HILOS.

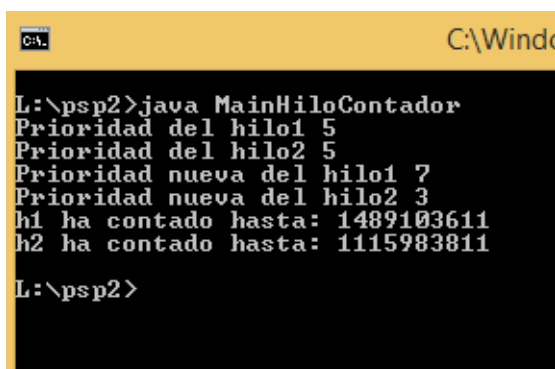
En teoría, los hilos de prioridad más alta disponen de más tiempo de CPU que los de prioridad más baja. Y un hilo de prioridad más alta puede desalojar a un hilo de prioridad más baja. También en teoría, hilos de la misma prioridad deben tener el mismo acceso a la CPU. Sin embargo, todos estos factores dependen de cómo el sistema operativo implemente la multitarea.

Características de las prioridades de los hilos:

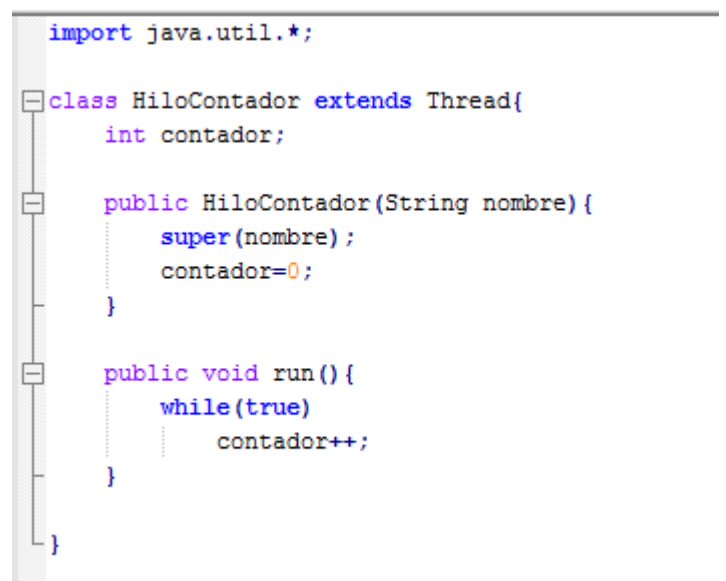
- Cada hilo tiene una prioridad, que no es más que un valor entero entre 1 y 10.
- Cuanto mayor el valor, mayor es la prioridad.
- El planificador determina el hilo que debe ejecutarse en función de la prioridad asignada a cada uno de ellos.
- Cuando se crea un hilo en Java, éste hereda la prioridad de su padre, el hilo que lo ha creado.

En Java, para establecer la prioridad de un hilo se utiliza el método *setPriority()*. La prioridad que se le puede asignar a un hilo está entre el rango comprendido por dos constantes: *Thread.MIN_PRIORITY* y *Thread.MAX_PRIORITY*. Actualmente estos valores son **1** y **10**, respectivamente. La prioridad por defecto de un hilo es igual a *Thread.NORM_PRIORITY*, que actualmente es **5**. Para obtener la prioridad de un hilo puede usarse el método *getPriority()*.

El siguiente ejemplo presenta dos hilos con distintas prioridades. Cada hilo ejecuta un bucle en el que va incrementando un contador. Después de 3 segundos, el hilo principal detiene a ambos hilos y se visualiza el contenido del contador de cada hilo. En teoría, el hilo de mayor prioridad se habrá ejecutado durante más tiempo y su contador tendrá un número mayor.



```
C:\Wind...
L:\psp2>java MainHiloContador
Prioridad del hilo1 5
Prioridad del hilo2 5
Prioridad nueva del hilo1 7
Prioridad nueva del hilo2 3
h1 ha contado hasta: 1489103611
h2 ha contado hasta: 1115983811
L:\psp2>
```



```
import java.util.*;

class HiloContador extends Thread{
    int contador;

    public HiloContador(String nombre){
        super(nombre);
        contador=0;
    }

    public void run(){
        while(true)
            contador++;
    }
}
```

```

public class MainHiloContador{
    public static void main(String args[]){
        //Creo dos hilos
        HiloContador h1= new HiloContador("Mi hilo 1");
        HiloContador h2= new HiloContador("Mi hilo 2");

        //Muestro las prioridades actuales de cada hilo
        System.out.println("Prioridad del hilo1 "+h1.getPriority());
        System.out.println("Prioridad del hilo2 "+h2.getPriority());

        //Cambiamos las prioridades
        h1.setPriority(Thread.NORM_PRIORITY+2); //Prioridad 7
        h2.setPriority(Thread.NORM_PRIORITY-2); //Prioridad 3

        //Muestro las prioridades actuales de cada hilo
        System.out.println("Prioridad nueva del hilo1 "+h1.getPriority());
        System.out.println("Prioridad nueva del hilo2 "+h2.getPriority());

        h2.start(); //arrancamos los hilos, primero el de menor prioridad
        h1.start();

        try{
            Thread.sleep(3000); //dormimos el hilo principal 3 segundos

            h1.stop(); //Paramos o fío de mayor prioridad
            h2.stop(); //Paramos o fío de menor prioridad

        }catch (InterruptedException ex){
            System.out.println("Hilo principal interrumpido");
        }

        System.out.println("h1 ha contado hasta: "+h1.contador);
        System.out.println("h2 ha contado hasta: "+h2.contador);
    }
}

```

5.SINCRONIZACIÓN DE HILOS.

Cuando trabajamos con varios hilos a la vez pueden darse problemas como:

- **CONDICIONES DE CARRERA:** Si el resultado de la ejecución de un programa depende del orden concreto en que se realicen los accesos a memoria.
- **INCONSISTENCIA DE MEMORIA:** Se produce cuando diferentes hilos tienen una visión diferente de lo que debería ser el mismo dato.
- **INANICIÓN:** Cuando un proceso nunca llega a tomar el control de un recurso debido a que el resto siempre toman el control antes que él por diferentes motivos.
- **INTERBLOQUEO:** Se produce cuando dos o más procesos o hilos están esperando indefinidamente por un evento que solo puede generar un proceso o hilo bloqueado
- **BLOQUEO ACTIVO:** Es similar a un interbloqueo, excepto que el estado de los dos procesos envueltos en el bloqueo activo cambia constantemente con respecto al otro.

Las condiciones de carrera y las inconsistencias de memoria se producen porque se ejecutan varios hilos concurrentemente pudiendo ser ordenados de forma diferente a la esperada.

- La solución pasa por provocar que cuando los hilos accedan a datos compartidos, los accesos se produzcan de forma ordenada o síncrona.
- Cuando estén ejecutando código que no afecte a datos compartidos, podrán ejecutarse libremente en paralelo, proceso también denominado ejecución asíncrona.

Frecuentemente, los *threads* necesitan compartir datos. Por ejemplo, supongamos que existe un *thread* que escribe datos en un fichero mientras, al mismo tiempo, otro *thread* está leyendo el mismo fichero. Cuando los *threads* comparten información necesitan sincronizarse para obtener los resultados deseados.

Cuando dos o más hilos tienen que acceder a un recurso compartido, es necesario asegurar de alguna manera que sólo uno de ellos accede a ese recurso en cada instante. El proceso mediante el que se consigue se denomina **sincronización**. Java proporciona un soporte para la sincronización a nivel de lenguaje, para ello emplea **monitores**.

Un monitor es un objeto que se utiliza como cerrojo exclusivo. Sólo uno de los hilos puede poseer el monitor en un determinado instante. Cuando un hilo adquiere un cerrojo, se dice que ha entrado en el monitor. Todos los demás hilos que intenten acceder al monitor quedarán suspendidos hasta que el primero salga del monitor.

Los monitores se utilizan para proteger un recurso compartido y evitar que sea manipulado por más de un hilo simultáneamente.

En Java, cada objeto tiene un monitor implícito. Para entrar en el monitor de un objeto, basta con llamar a un método modificado con la palabra clave **synchronized**. Mientras un hilo esté dentro de un método sincronizado, todos los demás hilos que traten de llamar a ese método, o a otro método sincronizado del mismo objeto, tendrán que esperar. Cuando un hilo acaba de ejecutar el código de un método sincronizado, entonces se dice que abandona el monitor y otro hilo puede entrar.

5.1. MÉTODOS SYNCHRONIZED.

Una clase cuyos objetos se deben proteger de interferencias en un entorno con múltiples hilos declara generalmente sus métodos apropiados como **synchronized**. Si un hilo invoca a un método **synchronized** sobre un objeto, en primer lugar, se adquiere el bloqueo de ese objeto, se ejecuta el cuerpo del método y después se libera el bloqueo. Otro hilo que invoque un método **synchronized** sobre ese mismo objeto se bloqueará hasta que el bloqueo se libere.

Los constructores no necesitan ser **synchronized** porque se ejecutan sólo cuando se crea un objeto, y

eso sólo puede suceder en un hilo para un objeto dado. De hecho, los constructores no pueden ser declarados synchronized.

Cuando una clase extendida redefine a un método synchronized, el nuevo método puede ser synchronized o no. El método de la superclase será synchronized cuando se invoque.

Si el método no sincronizado de la subclase utiliza super para invocar al método de la superclase, el bloqueo del objeto se adquirirá en ese momento y se liberará cuando se vuelva del método de la superclase.

Para entender mejor todo esto veamos un ejemplo:

SIN SINCRONIZACION (TAL COMO HEMOS ESTADO HACIENDO HASTA AHORA)

```
class ImprimirMensaje{
    void visualizar(String msg){//Imprime un mensaje entre corchetes.
        System.out.print("[ "+msg); //Imprime el primer corchete y el mensaje
        try{
            Thread.sleep(1000); //se duerme durante un segundo
        } catch (InterruptedException ex){
            System.out.println("Hilo interrumpido");
        }
        System.out.print("]");
    }
}
```

```
class MiHilo extends Thread{
    String mensaje; //Cadena a visualizar entre corchetes.
    ImprimirMensaje obj; //Objeto de ImprimirMensaje para llamar a "visualizar".

    public MiHilo(String nombre, ImprimirMensaje obj, String cadena){
        super(nombre); //Llamada al constructor de Thread.
        mensaje=cadena; //inicialización de los atributos.
        this.obj=obj;
    }

    public void run(){ //codigo del hilo
        obj.visualizar(mensaje);
        //Se usa obj para visualizar mensaje entre corchetes.
    }
}
```

```

public class EjSincronizacion{
    public static void main(String args[]) throws InterruptedException{
        //objeto para imprimir mensajes entre corchetes.
        ImprimirMensaje objeto=new ImprimirMensaje();
        //hilo para visualizar "Hola" entre corchetes usando objeto
        MiHilo a = new MiHilo("hilo a", objeto, "Hola");
        //hilo para visualizar "Mundo" entre corchetes usando objeto
        MiHilo b = new MiHilo("hilo n", objeto, "Mundo");
        //hilo para visualizar "Sincronizado" entre corchetes usando objeto
        MiHilo c = new MiHilo("hilo c", objeto, "Sincronizado");

        //Arrancar los hilos
        a.start();
        b.start();
        c.start();
    }
}

```

Este programa utiliza una clase llamada *ImprimirMensaje* que tiene un método que recibe una cadena y la muestra entre corchetes. El método *visualizar* imprime una cadena entre corchetes. Sin embargo, antes de mostrar el último corchete de cierre se duerme durante un segundo, esta pausa puede ser aprovechada por otro hilo para empezar a ejecutar el mismo código antes de que el primero haya finalizado.

Se crean tres hilos, cada uno de ellos contiene una cadena y el mismo objeto de la clase *ImprimirMensaje*, los hilos utilizarán este objeto para llamar al método *visualizar* y mostrar sus cadenas entre corchetes. Como el método *visualizar* no está sincronizado, los tres hilos pueden entrar a la vez en el método y producir un resultado inesperado.

Synchronized como modificador de método

Un método en Java puede llevar el modificador *synchronized*. Todos los métodos que lleven ese modificador se ejecutarán en exclusión mutua. Cuando un método sincronizado se está ejecutando, se garantiza que ningún otro método sincronizado podrá ejecutarse. Sin embargo, cualquier número de métodos no sincronizados puede estar ejecutándose dentro del objeto. Sólo puede haber un *thread* ejecutando el método 2 o 4 (métodos sincronizados) mientras que puede haber cualquier número de *threads* ejecutando el resto de métodos. Aquellos *threads* que quieran ejecutar un método sincronizado mientras otro *thread* está dentro de él tendrán que esperar a que éste último abandone la exclusión mutua.

EL MISMO EJEMPLO, PERO AHORA USANDO SINCRONIZACIÓN

Para corregir el problema anterior, se debe producir un acceso en serie al método *visualizar*, es decir, se debe **restringir** el acceso a un único hilo en cada instante. Ello es lo que nos permiten los monitores. Sustitúyase el método *visualizar* anterior, por el siguiente:

synchronized void visualizar(String msg){ ... }

```
class ImprimirMensajeSincro{
    synchronized public void visualizar(String msg){//Imprime un mensaje entre corchetes.
        System.out.print("[ "+msg); //Imprime el primer corchete y el mensaje
        try{
            Thread.sleep(1000); //se duerme durante un segundo
        }catch (InterruptedException ex){
            System.out.println("Hilo interrumpido");
        }
        System.out.print("]");
    }
}
```

Añadiendo la palabra reservada *synchronized* delante de la definición de un método se impide que otros hilos accedan a él mientras un determinado hilo lo está utilizando.

Una vez hecho esto, al ejecutar el programa, el primero que entraría en el método sincronizado sería el *a*, que imprimiría la palabra “*Hola*” entre corchetes, y luego *b* y *c* deberían esperar hasta que *a* finalizase. Cuando *a* finaliza entra el siguiente en llegar al monitor, es decir *b* que imprimiría “*Mundo*” entre corchetes, para posteriormente entrar *c* e imprimir “*Sincronizado*” entre corchetes.

[Hola][Mundo][Sincronizado]

5.2. SINCRONIZACIÓN DE OBJETOS O A UN BLOQUE DE CÓDIGO

La creación de métodos sincronizados en clases creadas por el programador es una forma fácil y efectiva de conseguir la sincronización. Sin embargo, a veces es necesario sincronizar el acceso a objetos de una clase que no fue diseñada para el acceso de múltiples hilos. Por lo tanto, ¿cómo podemos añadir la palabra *synchronized* delante de cada método que queramos sincronizar? Afortunadamente Java proporciona otra solución, simplemente hay que poner las llamadas a los objetos que se quieren sincronizar dentro de un bloque sincronizado:

```
synchronized (objeto){
    //instrucciones que deben ser sincronizadas
}
```

donde *objeto* es una referencia al objeto que se quiere sincronizar. Para comprender mejor esto, veamos cómo se podía conseguir la sincronización en el ejemplo anterior sin añadir *synchronized* delante del método *visualizar* de la clase *ImprimirMensaje*.


```

class MiHilo extends Thread{
    String mensaje;//Cadena a visualizar entre corchetes.
    ImprimirMensaje obj; //Objeto de ImprimirMensaje para llamar a "visualizar".

    public MiHilo(String nombre, ImprimirMensaje obj, String cadena){
        super(nombre);//Llamada al constructor de Thread.
        mensaje=cadena; //inicialización de los atributos.
        this.obj=obj;
    }

    public void run(){//codigo del hilo
        synchronized(obj){//Se sincronizan las llamadas al objeto
            obj.visualizar(mensaje);
            //Se usa obj para visualizar mensaje entre corchetes.
        }
    }
}

```

En el ejemplo no se ha modificado con la palabra *synchronized* el método *visualizar*. En su lugar, se utiliza la sentencia *synchronized* dentro del método *run* de los hilos llamantes. La salida que se obtiene es la misma que la anterior, ya que cada hilo espera a que el anterior termine antes de proceder.

Sin embargo, existen muchas situaciones interesantes donde ejecutar threads concurrentes que compartan datos y deban considerar el estado y actividad de otros threads. **Este conjunto de situaciones de programación son conocidos como escenarios 'productor/consumidor'; donde el productor genera un canal de datos que es consumido por el consumidor.**

6. BLOQUEOS. DEADLOCK.

Este tipo de error está relacionado con la multitarea y es necesario evitarlo. Se produce cuando dos hilos tienen una **dependencia circular** en un par de objetos sincronizados.

Siempre que tenemos **2 hilos y 2 objetos con bloqueo**, puede producirse un deadlock. Se trata de una situación en la que **cada uno de los hilos tiene el bloqueo** de uno de los objetos **y está esperando** por el bloqueo del otro objeto. Si un objeto X tiene un método *synchronized* que invoca a un método *synchronized* del objeto Y, y éste a su vez tiene un método *synchronized* que invoca a un método *synchronized* del objeto X, puede suceder que dos hilos estén esperando a que el otro finalice para obtener el bloqueo, y ninguno de los dos podrá ejecutarse. Esta situación se denomina también **abrazo mortal**.

El programador es el responsable de evitar que se produzcan deadlocks. El sistema en tiempo de ejecución no los detecta, ni los evita.

Los deadlocks son un tipo de error difícil de resolver por dos razones:

- Ocurre en raras ocasiones, cuando los dos hilos intentan entrar a la vez.
- En el bloqueo pueden estar involucrados más de dos hilos y dos objetos sincronizados.

Un ejemplo:

```
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();

        System.out.println(name + " dentro A.foo");

        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            System.out.println("A Interrupted");
        }

        System.out.println(name + " Intentando llamar a B.last()");
        b.last();
    }

    synchronized void last() { System.out.println("Dentro de A.last"); }
}
```

```
class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " dentro B.bar");

        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            System.out.println("B Interrupted");
        }

        System.out.println(name + " Intentando llamar a a A.last()");
        a.last();
    }

    synchronized void last() { System.out.println("Dentro de A.last"); }
}
```

```

class Deadlock implements Runnable {
    A a = new A();
    B b = new B();

    Deadlock() {
        Thread.currentThread().setName("MainThread");
        Thread t = new Thread(this, "RacingThread");
        t.start();

        a.foo(b); // get lock on a in this thread.
        System.out.println("Back in main thread");
    }

    public void run() {
        b.bar(a); // get lock on b in other thread.
        System.out.println("Back in other thread");
    }

    public static void main(String args[]) { new Deadlock(); }
}

```

Un hilo se bloquea intentando entrar en `a.last()`, ya que el otro hilo tiene ya el bloqueo sobre el objeto `a` (al estar ejecutando un método `synchronized` de `a`). El otro hilo se bloquea intentando entrar en `b.last()`, ya que el otro hilo tiene ya el bloqueo sobre el objeto `b` (al estar ejecutando un método `synchronized` de `b`).

7.COMUNICACIÓN ENTRE HILOS.

En los ejemplos anteriores se bloqueaba el acceso asíncrono a ciertos métodos para los demás hilos. Esta utilización de los monitores de Java es bastante eficaz, pero puede conseguirse un nivel más refinado de control mediante la comunicación entre hilos.

La comunicación entre hilos la podemos ver como un mecanismo de auto-sincronización, que consiste en lograr que **un hilo actúe solo cuando otro ha concluido cierta actividad** (y viceversa).

Java soporta **comunicación entre hilos** mediante los siguientes métodos de la clase **java.lang.Object**.

- **wait()**. Detiene el hilo (pasa a "no ejecutable"), el cual no se reanudará hasta que otro hilo notifique que ha ocurrido lo esperado.
- **wait(long tiempo)**. Como el caso anterior, solo que ahora el hilo también puede reanudarse

- (pasar a "ejecutable") si ha concluido el tiempo pasado como parámetro.
- **notify()**. Notifica a uno de los hilos puestos en espera para el mismo objeto, que ya puede continuar.
- **notifyAll()**. Notifica a todos los hilos puestos en espera para el mismo objeto que ya pueden continuar.

La llamada a estos métodos se realiza dentro de bloques *synchronized*.

Vamos a ver con más calma cómo funcionan estos métodos.

- **wait()**. Detiene al hilo que lo invoca hasta que le sea notificada la posibilidad de continuar.
 - El método **wait()** se debe invocar sobre un **objeto compartido** por los hilos a sincronizar.
 - Para poder invocar a **wait()** el hilo debe tener la **exclusión mutua** (el cerrojo) del objeto compartido.
 - La invocación de **wait()** detiene al hilo, **pasa a "no ejecutable"**, lo pone en una cola de espera asociada al objeto (**cola wait del objeto**), y libera el cerrojo del objeto.
 - El método **wait()** puede provocar una **InterruptedException**.
- **notify()**. Notifica a un hilo que invocó **wait()** sobre el mismo objeto y que está en la cola de espera del objeto (cola wait), que ya puede continuar.
 - Un hilo **sale de la cola de espera** del objeto (cola wait) pasando al estado "ejecutable", y se bloquea hasta conseguir el cerrojo del objeto para continuar su ejecución.
 - Si hay más de un hilo en la cola de espera (cola wait), **notify()** reactivará **solo a uno** de ellos. El criterio de selección del hilo a reactivar o pasar a "ejecutable" depende de la implementación de Java.
 - Una vez re-obtenido el cerrojo del objeto, el hilo que salió de la lista de espera (cola wait) **continuará la ejecución** del método en la instrucción siguiente a la llamada a **wait()**.
- **notifyAll()**. **Notifica a todos** los hilos puestos en espera para el mismo objeto (cola wait) que ya pueden continuar.

El hilo que invoca **notify()** no tiene ninguna referencia del hilo que está en espera por haber invocado **wait()** sobre el mismo objeto. Cuando un hilo invoca a **notify()**, otro hilo (no se sabe cuál) de los que están en espera es reactivado (pasa a "ejecutable").

Con **notifyAll()** se reactivan, volverán "ejecutables" todos los hilos que estaban bloqueados en la cola de espera del objeto. Sin embargo, el cerrojo, solo podrán tomarlo de uno en uno.