



HILOS (THREADS)

Víctor Custodio Ramírez

THREADS (HILOS)

- Conocidos también como procesos ligeros.
- Un thread es un flujo de ejecución secuencial dentro de un proceso.
- Un mismo proceso java puede tener
 - a). Un único thread (el thread principal) y por tanto se le llama monotarea.
 - b). Varios threads (por ejemplo el thread principal y el de gestión de eventos). Y por tanto se le llama multitarea.
- Casi todas las clases referentes al manejo de threads se encuentran en el paquete `java.lang.*`



MULTITAREA vs Multiproceso

- No hay que confundir los dos conceptos.
- Multiproceso significa que el equipo hardware cuenta con más de un procesador (CPU) y por tanto ejecuta varias tareas a la vez.
- Multitarea significa que varias tareas comparten el único procesador (CPU) dándonos la sensación de multiproceso.
- La multitarea se consigue mediante un planificador de tareas que van dando slots de CPU a cada tarea.

java.lang.Thread

- La clase principal es java.lang.Thread.
- Nos ofrece el API genérico de los threads así como la implementación de su comportamiento, incluyendo:
 - arrancar
 - dormirse
 - parar
 - ejecutarse
 - esperar
 - gestión de prioridades.

java.lang.Thread (contin..)

- La lógica que va a ejecutar un thread se incluye en el método:

```
public void run()
```

- Cuando termina la ejecución del método run() se termina el thread.
- La clase java.lang.Thread contiene un método run() vacío.

java.lang.Runnable

- Se trata de una interfaz.
- Simplemente fuerza la implementación de un método:

```
public void run();
```
- Existe para paliar la falta de herencia múltiple en el lenguaje java.

Implementando un thread

- Existen dos técnicas para crear un thread.
 - ✓ Heredar de la clase `java.lang.Thread` y sobrescribir el método `run()`.
 - ✓ Implementar la interfaz `java.lang.Runnable` (por tanto tenemos que implementar el método `run()`) y crear una instancia de la clase `java.lang.Thread` pasándole el objeto que implementa `java.lang.Runnable` como parámetro.
- Normalmente se usará la opción `Runnable` cuando la clase que va a contener la lógica del thread ya herede de otra clase (Swing, Applets,..)



Primera opción para ejecutar un hilo

```
public MiHilo extends Thread
{
    public void run()
    {
        // Aquí el código pesado que tarda mucho
    }
};

...
MiHilo elHilo = new MiHilo();
elHilo.start();
System.out.println("Yo sigo a lo mio");
```

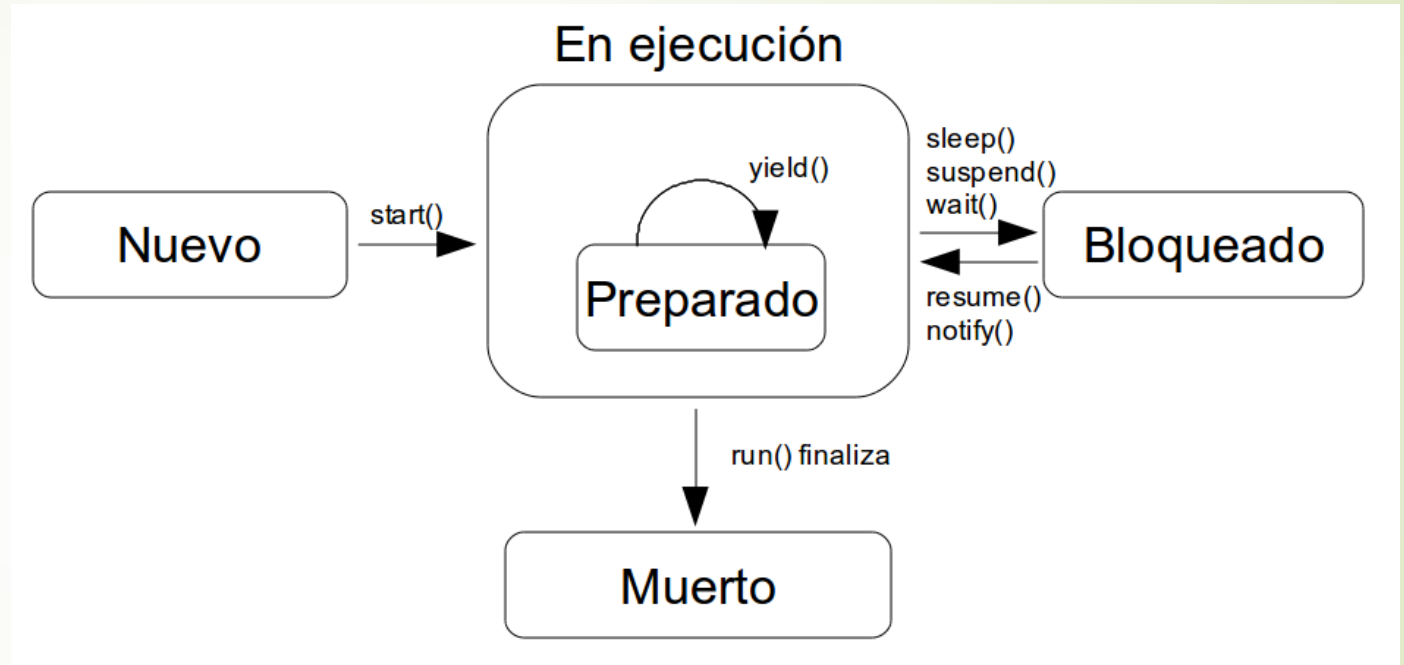



Segunda opción para ejecutar un hilo

```
public class MyRunnable implements Runnable {  
  
    public void run(){  
        System.out.println("MyRunnable running");  
    }  
}  
  
Thread thread = new Thread(new MyRunnable());  
thread.start();
```

CICLO DE VIDA

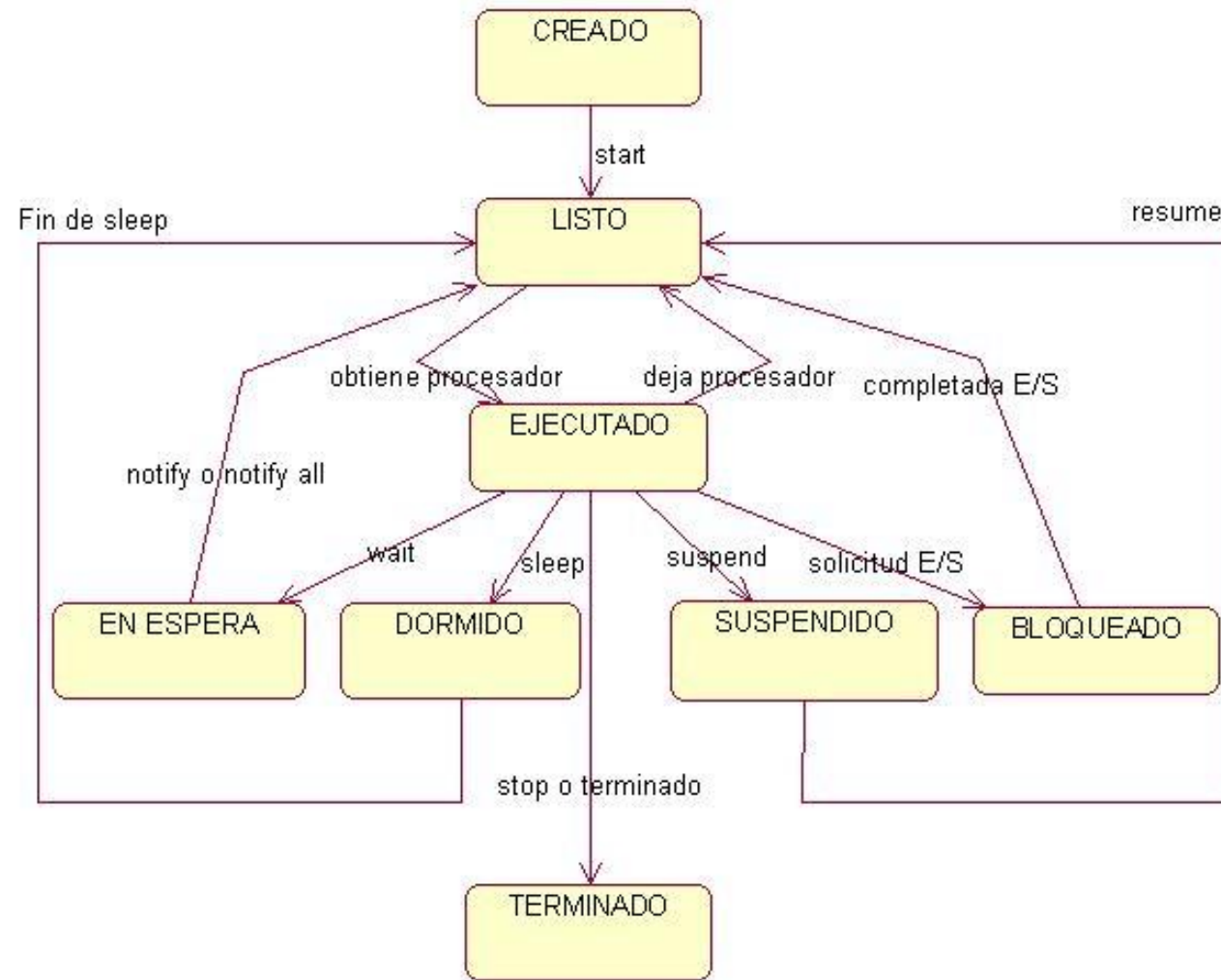
- Un thread puede pasar por varios estados durante su vida.



Ejecutándose
Parado
Muerto

- Existen distintos métodos que provocan las transiciones entre estos estados.

CICLO DE VIDA (Cont)




Arrancar un thread

- Para arrancar un thread hay que llamar al método `start()`
- El método `start()` registra al thread en el planificador de tareas del sistema y llama al método `run()` del thread.
- Ejecutar este método no significa que de forma inmediata comience a ejecutarse. Esto ya dependerá del planificador de tareas (Sistema Operativo) y del número de procesadores (Hardware)

Ejemplo Hilos

```
public class TortugaThread extends Thread
{
    public void run()
    {
        int i=0;
        System.out.println("Comienza la tortuga..");
        while(i<5)
        {
            try
            {
                Thread.sleep(5000);
                System.out.println("Tortuga..");
            }
            catch(InterruptedException ex)
            {
            }
            i++;
        }
        System.out.println("Termina la tortuga");
    }
}
```



```
import java.io.*;
import java.lang.*;
```

Ejemplo (cont..)

```
public class LiebreThread implements Runnable
{
    public void run()
    {
        int i=0;
        System.out.println("Comienza la liebre..");
        while(i<5)
        {
            try
            {
                Thread.sleep(2000);
                System.out.println("Liebre..");
            }
            catch(InterruptedException ex)
            {
            }
            i++;
        }
        System.out.println("Termina la liebre");
    }
}
```

Ejemplo (contin.)

```
import java.awt.*;
import java.awt.event.*;
import java.lang.*;
import java.io.*;

public class AplicHilo1 {

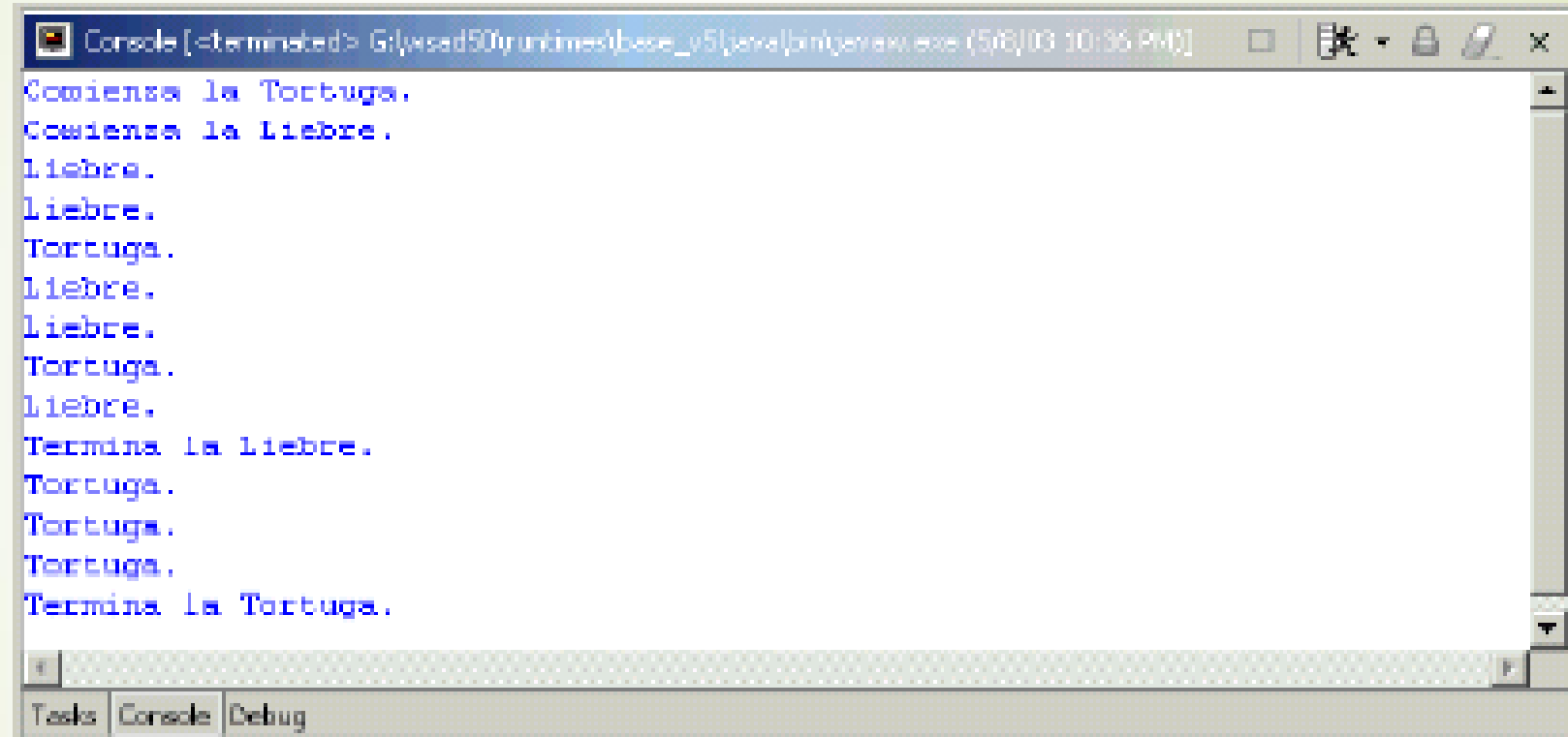
    public static void main(String args[])
    {

        TortugaThread tortuga=new TortugaThread();
        Thread liebre=new Thread(new LiebreThread());

        tortuga.start();
        liebre.start();

    }
}
```

Ejecución



```
Console [terminated] G:\wsad50\runtimes\base_v5\java\bin\java.exe (5/8/03 10:36 PM)
Comienza la Tortuga.
Comienza la Liebre.
Liebre.
Liebre.
Tortuga.
Liebre.
Liebre.
Tortuga.
Liebre.
Termina la Liebre.
Tortuga.
Tortuga.
Tortuga.
Termina la Tortuga.
```


Pausar un thread

- Existen distintos motivos por los que un thread puede detenerse temporalmente su ejecución o lo que es lo mismo, pasar a un estado de pausa.
 - Se llama a su método `sleep()`. Recibe un long con el número de milisegundos de la pausa.
 - Se llama al método `wait()` y espera hasta recibir una señal (`notify`) o cumplirse un timeout definido por un long con el número de milisegundos.
 - Se realiza alguna acción de entrada/salida.
 - Se llama al método `yield()`. Este método saca del procesador al thread hasta que el sistema operativo lo vuelva a meter.

Reanudar un thread

- Existen distintos motivos por los que un thread puede reanudar su ejecución:
 - Se consumen los milisegundos establecidos en una llamada al método sleep.
 - Se recibe una llamada (notify) o se consumen los milisegundos en una llamada al método wait.
 - Se termina alguna acción de entrada/salida.

Terminar un thread

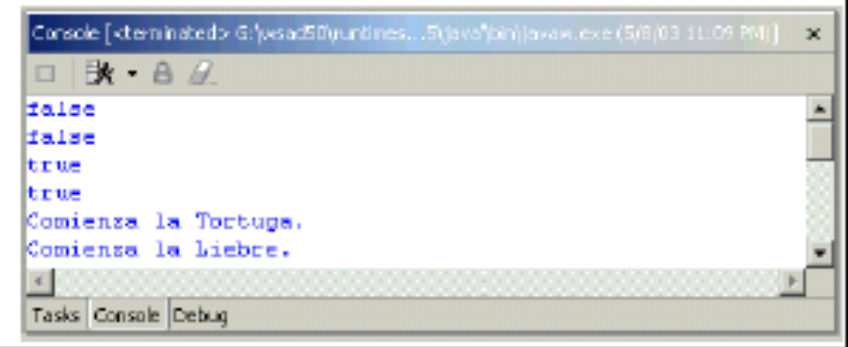
- Un thread, por defecto, termina cuando finaliza la ejecución de su método `run()`.
- En las primeras versiones de JDK existía el método `stop()`. Pero con el tiempo se deprecó (deprecated) desaconsejando su uso.
- La manera correcta de terminar un thread es conseguir que finalice la ejecución del método `run()` mediante la implementación de algún tipo de bucle gobernado por una condición controlable.
- El método `System.exit()` termina la JVM, terminando también todos los threads.

Conocer el estado del thread



Se puede conocer el estado mediante el método `isAlive()`.

```
public class Carrera
{
    public static void main(String[] args)
    {
        TortugaThread tortuga = new TortugaThread();
        Thread liebre = new Thread(new LiebreThread());
        System.out.println(tortuga.isAlive());
        System.out.println(liebre.isAlive());
        tortuga.start();
        liebre.start();
        System.out.println(tortuga.isAlive());
        System.out.println(liebre.isAlive());
    }
}
```



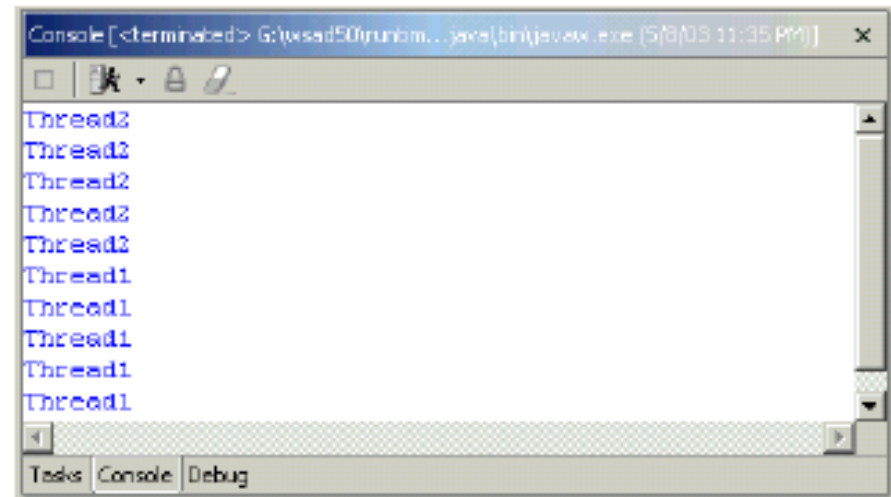
Prioridades

- Ya hemos comentado que cuando existe un único procesador (CPU) no existe multiproceso real. Los distintos threads van compartiendo dicho procesador (CPU) siguiendo las políticas o algoritmos del Sistema Operativo.
- Pero esas políticas o algoritmos pueden tener en cuenta prioridades cuando realiza sus cálculos.
- La prioridad de un thread se establece mediante el método `setPriority` pasándole un `int` entre:
`Thread.MAX_PRIORITY`
`Thread.MIN_PRIORITY`
- El método `join()` sobre un hilo hijo hace que el hilo padre, espere a que el hilo hijo termine, para seguir ejecutandose.

Ejemplo

```
public class MiThread extends Thread
{
    public void run()
    {
        int i = 0;
        while(i<5)
        {
            System.out.println(this.getName());
            i++;
        }
    }
}
```

```
public class Test
{
    public static void main(String[] args)
    {
        MiThread t1 = new MiThread();
        t1.setName("Thread1");
        t1.setPriority(Thread.MIN_PRIORITY);
        MiThread t2 = new MiThread();
        t2.setName("Thread2");
        t2.setPriority(Thread.MAX_PRIORITY);
        t1.start();
        t2.start();
    }
}
```



Grupo de Threads

- Todo thread es miembro de un grupo de threads.
- La clase `java.lang.ThreadGroup` implementa los grupos de threads.
- El grupo de threads al que pertenece un thread se establece en su construcción. Luego es inmutable.
- Por defecto, un thread pertenece al grupo al que pertenece el thread desde donde se le creo.
- El grupo del thread principal se llama “main”.

Grupo de Threads (contin..)

- Para crear un thread en un grupo distinto al seleccionado por defecto, hay que añadir como parámetro del constructor la instancia del grupo:

```
ThreadGroup tg=new ThreadGroup("Mis threads")  
Thread t=new Thread(tg);
```

- Para conocer el grupo al que pertenece un thread:

```
t.getThreadGroup();
```


Grupo de Threads (contin..)

- Los grupos de threads permiten actuar sobre todos los threads de ese grupo como una unidad.
- Pudiendo con una sola llamada:
 - ✓ Cambiarles el estado a todos
 - ✓ Cambiarles la prioridad a todos.
 - ✓ Acceder a la colección de threads.
 - ✓ Saber si un thread pertenece al grupo o no.

Sincronización de threads

- Hasta ahora hemos visto threads totalmente independientes. Pero podemos tener el caso de dos threads que ejecuten un mismo método o accedan a un mismo dato.
- Que pasa si un thread está trabajando con un dato y llega otro y se lo cambia?.
- Para evitar estos problemas existe la sincronización de threads que regula estas situaciones.



Sincronización de threads (contin..)

- Existen dos mecanismos de sincronización;
Bloqueo del objeto: synchronized;
Uso de señales: wait y notify.
- El tema de la sincronización de threads es muy delicado y peligroso. Se pueden llegar a provocar un dead-lock y colgar la aplicación.
- La depuración de problemas provocados por una mala sincronización es muy compleja.

Bloqueo de Objetos

- Para poder bloquear un objeto e impedir que otro thread lo utilice mientras está este, se emplea la palabra `synchronized` en la definición de los métodos susceptibles de tener problemas de sincronización.

```
public synchronized int getNumero();
```

- Cuando un thread está ejecutando un método `synchronized` en un objeto, se establece un bloqueo en dicho objeto.

Bloqueo de Objetos (cont..)

- Cualquier otro thread que quiera ejecutar un método marcado como synchronized en un objeto bloqueado, tendrá que esperar a que se desbloquee.
- El objeto se desbloquea cuando el thread actual termina la ejecución del método synchronized o pasa al estado de espera de entrada y salida o pausado por un wait.
- Se creará una lista de espera y se irán ejecutando por orden de llegada.
- El sistema de bloqueo/desbloqueo es algo gestionado de forma automática por la JVM.

Uso de Señales(wait y notify)



- Este es un sistema mediante el cual un thread puede detener su ejecución a la espera de una señal lanzada por otro thread.
- Para detener la ejecución y esperar a que otro thread nos envíe una señal se utiliza el método:

```
public void wait();  
public void wait(long timeout);
```
- Para enviar una señal a los threads que están esperando en el objeto desde donde enviamos la señal se utiliza el método:

```
public void notify();  
public void notifyAll();
```



Ejemplo Aplicación de Sincronización de Hilos

```
import java.io.* ;
public class escuela {
    public static void main ( String args[] )
    {
        try {
            Nota laNota = new Nota ();
            Profesor p = new Profesor ( laNota );
            Alumno a = new Alumno ( "Javier", laNota);
            Alumno b = new Alumno ( "Jose", laNota );
            // Empezamos la ejecución
            a.start();
            b.start();
            p.start();
            a.join();// El join hace que esperemos a que "a" acabe para continuar nuestra ejecución.
            b.join();
            p.join();
            System.out.println("Se cierra la escuela");
        }
        catch ( Exception e ){
            System.out.println ( e.toString() );
        }
    }
}
```



```
class Alumno extends Thread
{
    Nota na ; // nota del alumno
    String nom ; // nombre
    Alumno ( String nombre , Nota n )
    {
        na = n ;
        nom = nombre ;
    }

    public void run () {
        System.out.println ( nom + " Esperado su nota" );
        na.esperar(); // el alumno espera la nota
        System.out.println ( nom + " recibio su nota");  }
}
```

```
class Profesor extends Thread{
    Nota na ;
    Profesor ( Nota n ){
        na = n ;
    }
    public void run () {
        System.out.println ( " Voy a corregir los examenes");
        Thread.sleep(500);
        System.out.println ( " Voy a poner la nota ");
        na.dar (); // el profesor pone la nota del alumno
    }
}

class Nota {
    synchronized void esperar () {
        try { wait();
            }catch (InterruptedException e ){}
    }
    synchronized void dar (){
        notifyAll();}
}
```



C:\Archivos de programa\Xinox Software\JCreator LE\GE2001.exe

```
Jose Esperado su nota  
Javier Esperado su nota  
Voy a poner la nota  
Javier recibio su nota  
Jose recibio su nota  
Press any key to continue...
```

Ejercicio 4x100

- Implementar una carrera por relevos:
 - Tenemos 4 Atletas dispuestos a correr
 - Tenemos una clase principal Carrera
 - Tenemos un objeto estático testigo
 - Todos los atletas empiezan parados, uno comienza a correr (tarda aleatoriamente entre 9 y 11s) y al terminar su carrera pasa el testigo a Otro que comienza a correr, y así sucesivamente
- Pistas:
 - Thread.sleep y Math.random para simular la carrera
 - synchronized, wait y notify para el paso del testigo
 - System.currentTimeMillis o Calendar para ver tiempos

Ejercicio carrera 100 m lisos

Implementar una carrera de 100m lisos:

- Tenemos 8 Atletas dispuestos a correr
 - Cada uno tiene un atributo dorsal
- Tenemos una clase principal Carrera que indica el pistoletazo de salida y el resultado de la carrera
- Todos los Atletas comienzan pero se quedan parados esperando el pistoletazo de salida
- Luego comienzan a correr (tardan aleatoriamente entre 9 y 11s)
- Al llegar a meta notifican a la carrera su dorsal y terminan
- La Carrera escribe "preparados" y espera 1s, luego escribe "listos" y espera 1s, finalmente escribe "ya!" y notifica a los hilos de los Atletas
- Cada vez que un atleta le notifica su dorsal, escribe por pantalla: dorsal+" tarda "+System.currentTimeMillis()

Simulación Wa-tor

- El planeta Wa-Tor es un mundo acuático, toroidal y reticulado de dimensión 20x20. Este mundo está habitado por peces y tiburones de ambos sexos.

Inicialmente sitúa en forma aleatoria 100 peces (50 machos y 50 hembras) y 10 tiburones (5 machos y 5 hembras) en el planeta de manera que:

- Cada habitante del mundo (pez o tiburón) esté representado por **un hilo concurrente**.

- Cada habitante del mundo nada desde la posición en que se encuentra **una posición** hacia el norte, sur, este u oeste de forma aleatoria (recuerda que **el mundo es un toroide**) cada X milisegundos (el valor de x debe ser parametrizable).

- Si habitantes del mundo se encuentran en la misma posición ocurre lo siguiente:

- a) Si son del mismo sexo y de la misma especie, uno aniquila al otro.
- b) Si son de especies diferentes, el tiburón siempre aniquila al pez.
- c) Si son de la misma especie y de distinto sexo, se reproducen, generando un nuevo individuo cuyo sexo será macho o hembra con igual probabilidad, y luego continúan su camino.

- Muestra la evolución de Wa-tor



Ejemplo de una matriz toroidal de 4x4

➤

00	01	02	03	04
05	06	07	08	09
10	11	12	13	14
15	16	17	18	19

➤ la casilla al este de la 04 es la 00, al oeste de la 00 esta 04, al norte de la 00 esta la 15 y al sur de la 00 esta la 05.