Java Avanzado

Threads

Copyright

Threads

- También conocidos como procesos ligeros.
- Un thread es un flujo de ejecución secuencial dentro de un proceso.
- Un mismo proceso Java puede tener:
  - Un único thread (el thread principal) y por tanto se le llama monotarea.
  - 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.*

Multiproceso vs. Multitarea

- No hay que confundir los dos conceptos.
- Multiproceso significa que el equipo hardware cuenta con mas 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 va 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 (cont.)

- La lógica que va a ejecutar un thread se introduce 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 un interfaz.
- Simplemente fuerza la implementación de un método:

```
public void run();
```

- Existe para paliar la falta de herencia múltiple en el lenguaje Java. Ya veremos cómo en las siguientes transparencias.

Implementando un thread

- Existen dos técnicas para crear un thread:
  - Heredar de la clase java.lang.Thread y sobrescribir el método run().
  - Implementar el 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 de Runnable cuando la clase que va a contener la lógica del thread ya herede de otra clase (Swing, Applets,...).

Ejemplo

```
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.");
  }
}
```

Ejemplo

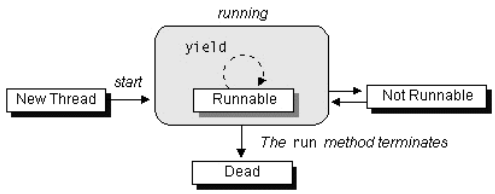
```
public class LiebreThread implements Runnable
{
  public void run()
  {
    int i=0;
    System.out.println("Comienza la Liebre.");
    while(i<5)
    {
      try
      {
        Thread.sleep(2000);
        System.out.println("Liebre.");
      }
      catch(InterruptedException ex)
      {
     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

Ejemplo

```
public class Carrera
{
  public static void main(String[] args)
  {
    TortugaThread tortuga = new TortugaThread();
    Thread liebre = new Thread(new LiebreThread());
  }
}
```

Ciclo de vida

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



  - Ejecutándose.
  - Pausado o parado.
  - Muerto.
- Existen distintos métodos que provocan las transiciones entre estos estados.

Crear un thread

- Para crear un thread hay que instanciarlo llamando al constructor como con el resto de clases Java.
- Dependiendo de cómo hayamos implementado el thread se actuará de una forma u otra:
  - Si hereda de la clase java.lang.Thread, simplemente se instancia nuestra clase.
  - Si implementa el interfaz java.lang.Runnable, se instancia la clase java.lang.Thread pasándole como parámetro del un constructor una instancia de nuestra clase.

Ejemplo

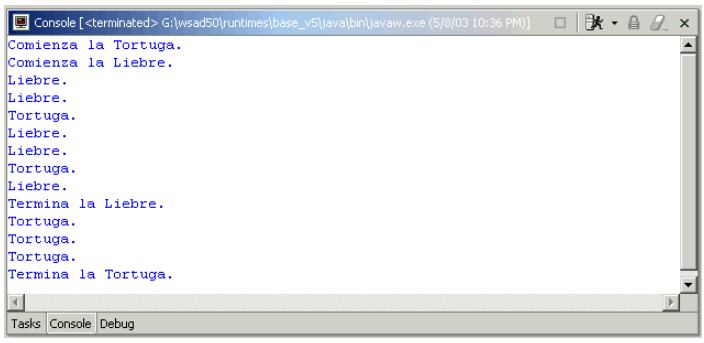
```
public class Carrera
{
  public static void main(String[] args)
  {
    TortugaThread tortuga = new TortugaThread();
    Thread liebre = new Thread(new LiebreThread());
  }
}
```

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

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



Pausar un thread

- Existen distintos motivos por los que un thread puede detener 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 le 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 señal (*notify*) o se consumen los milisegundos en una llamada al método wait.
  - Se termina alguna acción de entrada/salida.

Ejemplo

```
public class MiThread extends Thread
{
  public void run()
  {
    try
    {
      Thread.sleep(10000);
    }
    catch (InterruptedException ex)
    {
    }
  }
}
```

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.

Ejemplo

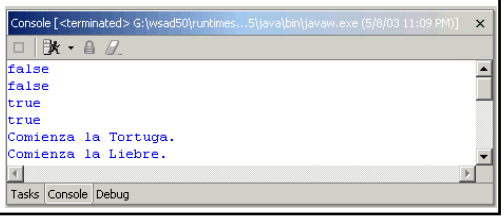
```
public class MiThread extends Thread
{
  public boolean sw = true;

  public void run()
  {
    while(sw)
    {
      .....
      .....
    }
  }
}
```

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());
  }
}
```



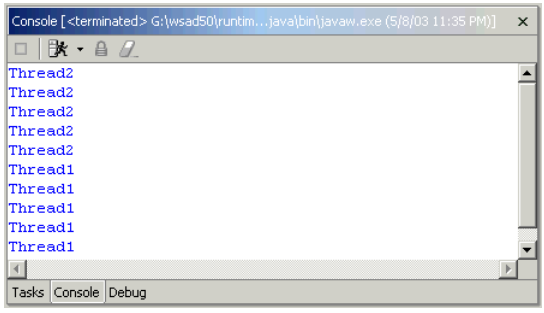
Prioridades

- 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

Ejemplo

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

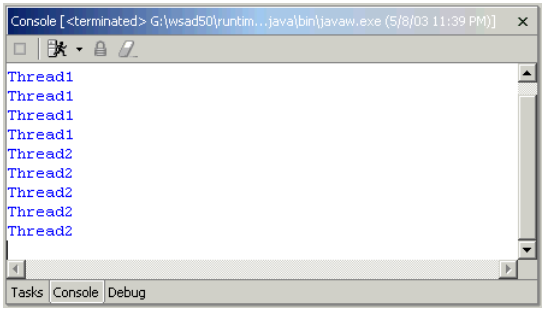
```
public class Test
{
  public static void main(String[] args)
  {
    MiThread t1 = new MiThread();
    t1.setName("Thread1");
    t1.setPriority(Thread.MIN_PRIORITY);
    MiThread t2 = new MiThread();
    t2.setName("Thread2");
    t2.setPriority(Thread.MAX_PRIORITY);
    t1.start();
    t2.start();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



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.MAX_PRIORITY);
    MiThread t2 = new MiThread();
    t2.setName("Thread2");
    t2.setPriority(Thread.MIN_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 (cont.)

- 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 (cont.)

- 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.
- ¿Qué 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 (cont.)

- 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.
- 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

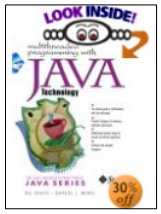
- Este es un sistema mediante el cual un thread puede detener su ejecución a la espera de una señal lanzada por otro thread.
- Para detener la ejecución y esperar a que otro thread nos envie una señal se utiliza el método:
  - public void wait();
  - public void wait(long timeout);
- Para enviar una 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();

Bibliografía

- Java Threads (2nd edition)
  Scott Oaks, Henry Wong.
  O'Reilly.

- Concurrent programming in Java (2nd edition)
  Doug Lea.
  Addison-Wesley.

- Multithreaded programming with Java
  Bil Lewis, Daniel J. Berg y Bill Lewis.
  Prentice Hall.



- The Java tutorial (on-line)
  http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html