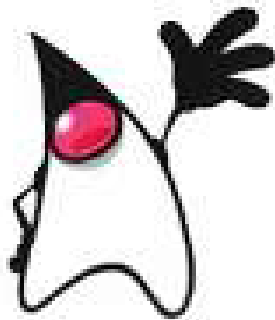


Conceptos Básicos de Concurrencia en Java



Ingeniería del Software II
Curso 2009/2010

Sergio Ilarri Artigas
silarri@unizar.es



Índice Detallado

- Programación Concurrente
 - Concepto de *Thread*
 - Crear y Arrancar *Threads*
 - Control de Threads: *stop, suspend/resume, sleep, join*, etc.
 - Planificación de *Threads*
 - Sincronización
 - Otros



Concepto de *Thread*

- CPU virtual
- Varias tareas al mismo tiempo
- Clase *Thread* (*java.lang*)
- Creación:
 - Clase *A* que implementa interface *Runnable*
 - Construir una instancia de *Thread* usando *A*
 - Llamar a *start()*, sólo 1 vez



Crear y Arrancar *Threads* (I)

```
public interface Runnable {  
    abstract public void run();  
}
```

Cualquier objeto puede convertirse en *thread*

No argumentos, no excepciones, no valor de retorno, vida del *thread*

```
class Animation implements Runnable { ... }
```

```
Animation happy = new Animation("Mr. Happy");  
Thread myThread = new Thread(happy);  
myThread.start();
```

Crear y Arrancar *Threads* (II)

O arrancando en el constructor

```
class Animation implements Runnable {  
    Thread myThread;  
    Animation (String name) {  
        myThread = new Thread(this);  
        myThread.start();  
    }  
}
```

...

class Animation implements Runnable

`myThread = new Thread()`

creates

Thread myThread

`myThread.start();`

`start()`

`run()`



Control de *Threads* (I)

- *stop()*
 - método complementario de *start()*
 - destruye el *thread*
 - *deprecated* (Java 1.2)
- *suspend()*, *resume()*
 - pausa y reanuda el *thread*

↙
¡Deprecated!
Utilizar en su lugar
Object.wait y Object.notify



Control de *Threads* (II)

- *sleep*(long millis)
- *join*() espera a que termine el *thread*
- *isAlive*()



Planificación de *Threads* (I)

- Si sólo hay una CPU, sólo se ejecuta 1 *thread* cada vez
- Llamar a *start()* pone al *thread* en estado ejecutable, no lo ejecuta
- Los *threads* son normalmente interrumpibles pero no se asegura un reparto por tiempo (no *time-slicing*)
- *setPriority(...)*



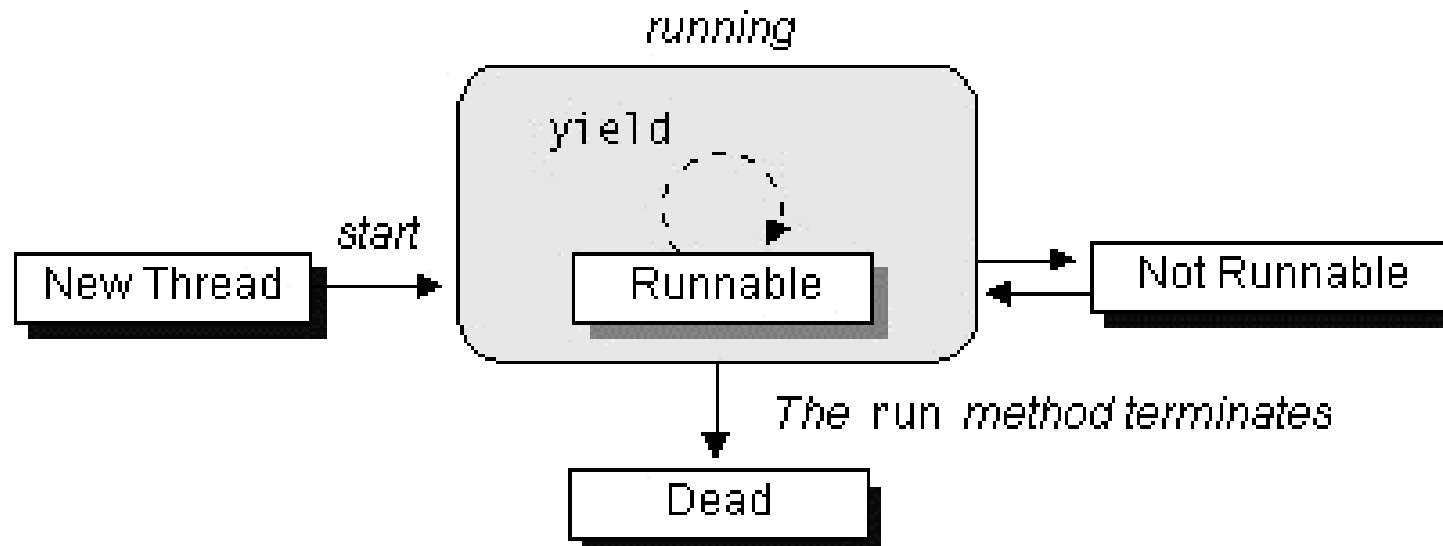
Planificación de *Threads* (II)

- Por tanto, hay que llamar a *sleep()* de vez en cuando
 - No necesariamente se reanuda el *thread* al terminar el período

```
try {Thread.sleep(1000);}  
catch (InterruptedException e) {...}
```

- El método *yield()* –pausa temporal- da paso a otros *threads* de igual prioridad

Ciclo de Vida





Otra Forma de Crear Threads

```
class MyThread extends Thread {  
    ...  
    public void run() {  
        ...  
    }  
}  
MyThread t = new MyThread();  
t.start();
```

Timer (en *java.util*):

- Ejecuta una *TimerTask* (implementa *Runnable*) tras un retardo



Resumiendo: Dos Métodos de Creación de *Threads*

```
public class HelloRunnable implements Runnable {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

```
public class HelloThread extends Thread {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}
```



Ejemplo: *SleepMessages*

```
public class SleepMessages {  
    public static void main(String args[]) throws InterruptedException {  
        String importantInfo[] = {  
            "Mares eat oats",  
            "Does eat oats",  
            "Little lambs eat ivy",  
            "A kid will eat ivy too"  
        };  
        for (int i = 0; i < importantInfo.length; i++) {  
            //Pause for 4 seconds  
            Thread.sleep(4000);  
            //Print a message  
            System.out.println(importantInfo[i]);  
        }  
    }  
}
```



Ejemplo: *SimpleThreads*

```
public class SimpleThreads {

    //Display a message, preceded by the name of the current thread
    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n", threadName, message);
    }

    private static class MessageLoop implements Runnable {
        public void run() {
            String importantInfo[] = {
                "Mares eat oats",
                "Does eat oats",
                "Little lambs eat ivy",
                "A kid will eat ivy too"
            };
            try {
                for (int i = 0; i < importantInfo.length; i++) {
                    //Pause for 4 seconds
                    Thread.sleep(4000);
                    //Print a message
                    threadMessage(importantInfo[i]);
                }
            } catch (InterruptedException e) {
                threadMessage("I wasn't done!");
            }
        }
    }
}
```

...

<http://java.sun.com/docs/books/tutorial/essential/concurrency/simple.html>



Ejemplo: *SimpleThreads*

```
...
public static void main(String args[]) throws InterruptedException {

    //Delay, in milliseconds before we interrupt MessageLoop
    //thread (default one hour).
    long patience = 1000 * 60 * 60;

    //If command line argument present, gives patience in seconds.
    if (args.length > 0) {
        try {
            patience = Long.parseLong(args[0]) * 1000;
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer.");
            System.exit(1);
        }
    }

    ...
}
```

<http://java.sun.com/docs/books/tutorial/essential/concurrency/simple.html>



Ejemplo: *SimpleThreads*

```
...  
  
threadMessage("Starting MessageLoop thread");  
long startTime = System.currentTimeMillis();  
Thread t = new Thread(new MessageLoop());  
t.start();  
  
threadMessage("Waiting for MessageLoop thread to finish");  
//loop until MessageLoop thread exits  
while (t.isAlive()) {  
    threadMessage("Still waiting...");  
    //Wait maximum of 1 second for MessageLoop thread to  
    //finish.  
    t.join(1000);  
    if (((System.currentTimeMillis() - startTime) > patience) &&  
        t.isAlive()) {  
        threadMessage("Tired of waiting!");  
        t.interrupt();  
        //Shouldn't be long now -- wait indefinitely  
        t.join();  
    }  
}  
  
threadMessage("Finally!");  
}  
}
```

<http://java.sun.com/docs/books/tutorial/essential/concurrency/simple.html>



Problemas de Concurrency

```
public class MyStack {  
    int index=0;  
    char[] data=new char[10];  
    public void push(char c) {  
        data[index]=c;  
        index++;  
    }  
    public char pop() {  
        index--;  
        return data[index];  
    }  
}
```

¡¡PROBLEMA!!

Sincronización: Uso de Cerrojos

- Cada objeto en Java tiene un *lock* (cerrojo) asociado
- La palabra clave *synchronized* permite el acceso al lock:

Espera en un *pool* por el cerrojo

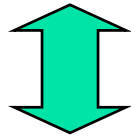
```
synchronized(objeto) {  
    // Aquí se dispone ya del cerrojo  
    ...
```

Se libera de nuevo el
cerrojo



Ejemplo de Uso

```
public void sampleMethod {  
    synchronized(this) {  
        ...  
    }  
}
```



```
public synchronized void sampleMethod {  
    ...  
}
```



Métodos *notify* y *wait* (*Object*)



- *wait()*:

- libera cerrojo hasta que alguien avise (con *notify* o *notifyAll*) y entonces lo retoma

- *notify()*:

- despierta a 1 arbit. esperando en el cerrojo

- *notifyAll()*:

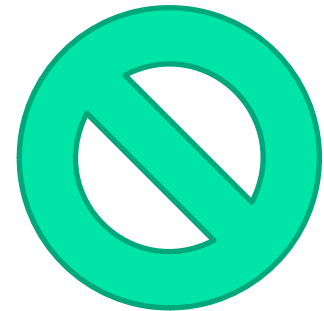
- despierta a todos
- sólo 1 conseguirá el cerrojo

Para llamarlos, hay que estar en posesión del cerrojo

Ejemplo de Uso de *Wait* y *Notify*

Espera activa

```
public void guardedJoy() {  
    //Simple loop guard. Wastes processor time. Don't do this!  
    while(!joy) {}  
    System.out.println("Joy has been achieved!");  
}
```



Ejemplo de Uso de *Wait* y *Notify*

Espera "inteligente"

```
public synchronized guardedJoy() {
    //This guard only loops once for each special event, which may not
    //be the event we're waiting for.
    while (!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}

public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```





Otros Métodos de Interés

- Variantes de *wait()*:
 - *wait(long millis)*
 - *wait(long millis, int nanos)*



Deadlocks

- Un *thread* A espera la liberación de un *lock* por parte de otro *thread* B que a su vez espera la liberación de un *lock* mantenido por A
- Es responsabilidad del programador
- Obtener siempre los cerrojos en el mismo orden y liberarlos en orden inverso



Ejemplo de *Deadlock*

```
public class Deadlock {  
    static class Friend {  
        private final String name;  
        public Friend(String name) {  
            this.name = name;  
        }  
        public String getName() {  
            return this.name;  
        }  
        public synchronized void bow(Friend bower) {  
            System.out.format("%s: %s has bowed to me!%n",  
                this.name, bower.getName());  
            bower.bowBack(this);  
        }  
        ...  
    }  
}
```

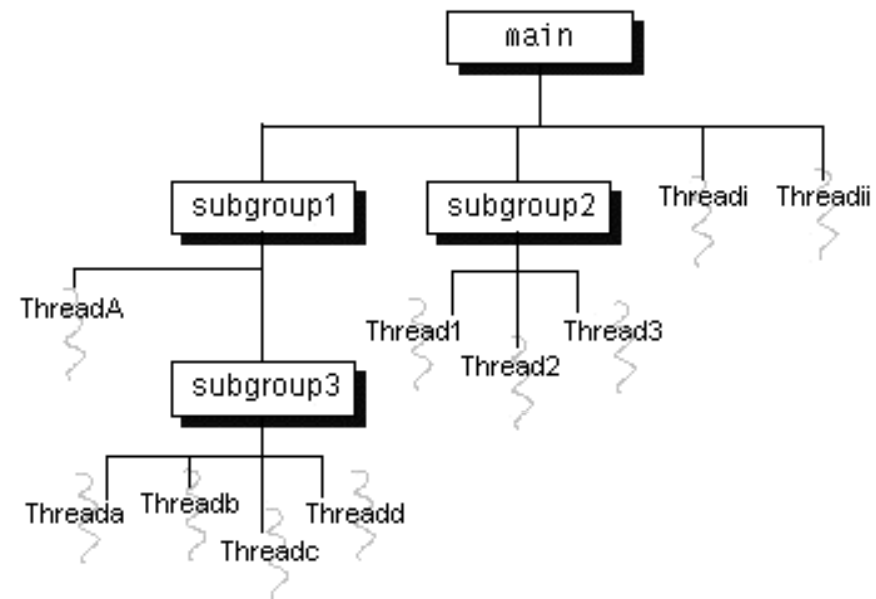


Ejemplo de *Deadlock*

```
...
    public synchronized void bowBack(Friend bower) {
        System.out.format("%s: %s has bowed back to me!%n",
            this.name, bower.getName());
    }
}
public static void main(String[] args) {
    final Friend alphonse = new Friend("Alphonse");
    final Friend gaston = new Friend("Gaston");
    new Thread(new Runnable() {
        public void run() { alphonse.bow(gaston); }
    }).start();
    new Thread(new Runnable() {
        public void run() { gaston.bow(alphonse); }
    }).start();
}
}
```

Otros

- *ThreadGroup*
 - *activeCount*
 - *getMaxPriority*
 - *setMaxPriority*
 - *resume*
 - *stop*
 - *suspend*





Referencias

- Concurrency in Java - Tutorial.
<http://java.sun.com/docs/books/tutorial/essential/concurrency/>
- Thread Programming, Paul Hyde, Sams, ISBN 0672315858, 528 pages.
- Thinking in Java, Bruce Eckel, Prentice Hall, 4th edition, ISBN 0131002872.
- Thinking in Java, 3th edition. Free download:
<http://mindview.net/Books/TIJ/DownloadSites>