
PRESENTACIÓN

Parciales	Ejercicios Semanales (individuales)	Prácticas (por parejas)
Parcial 1 (25%) → 05/04/22	10 ejercicios en total (10%) Un ejercicio por semana	Práctica 1 (20%) → Semana 16 MONITORES
Parcial 2 (25%) → 13/06/22		Práctica 2 (20%) → Semana 17 JCSP
Nota mínima* → 4 por parcial 5 de media	Nota mínima* → 3 (OJO: no son obligatorios)	Nota mínima* → 3 por práctica

***NOTA:** Para aprobar la asignatura la nota media de los parciales debe ser igual o superior a 5; y la nota media de las dos prácticas y los ejercicios individuales debe ser igual o superior a 5 también.

Cualquier calificación (parciales 1 y 2; prácticas 1 y 2; o ejercicios semanales) se guarda hasta la convocatoria de JULIO. No se guardan calificaciones de un año para otro.

INTRODUCCIÓN

(1) Introducción. Concurrencia e interacción:

(a) Conceptos fundamentales de la concurrencia

¿Qué es la **concurrencia**? Según la RAE es la acción de concurrir distintas personas, sucesos o cosas en un mismo lugar o tiempo; pero ¿y qué es la concurrencia en el software? ¿Qué es un proceso? ¿Es lo mismo un proceso que un programa?

Un **programa secuencial** es aquel que ejecuta de forma secuencial una lista de instrucciones.

Un **proceso** es la ejecución de un programa secuencial.

Un **programa concurrente** es aquel que define dos o más programas secuenciales que pueden ejecutarse concurrentemente en forma de **procesos paralelos**.

Entonces... ¿es lo mismo **concurrencia** que *paralelismo*? La respuesta es **NO**, ya que si únicamente disponemos de un procesador, puede haber concurrencia, pero no paralelismo; en el que la ejecución de los diferentes procesos se va intercalando.

La existencia de varios procesos ejecutando multiplica las posibles ejecuciones de los programas (**ejecución simultánea**). Dependiendo de cada posible ejecución se podrán obtener resultados diferentes (**indeterminismo**). Los procesos pueden interactuar entre sí (**interacción**).

La **concurrencia** se compone de estos tres conceptos básicos, es decir:

CONCURRENCIA = EJECUCIÓN SIMULTÁNEA + INDETERMINISMO + INTERACCIÓN

La **comunicación** permite a la ejecución de un proceso influenciar en la ejecución del otro gracias a las variables compartidas y/o el paso de mensajes; mientras que la **sincronización** permite establecer cierto orden de ejecución de diferentes partes del programa utilizando para ello la exclusión mutua o la sincronización por condición.

La **interacción** se compone de estos dos conceptos básicos, es decir:

INTERACCIÓN = COMUNICACIÓN + SINCRONIZACIÓN

Dificultades que presenta la concurrencia:

La **velocidad** de ejecución de un programa depende de muchos factores; no se puede garantizar que programas concurrentes idénticos y ejecutados en el mismo procesador se ejecuten exactamente igual.

Lo único que podemos asumir es que el programa **progres**a; pero no podemos asumir nada sobre los posibles **ritmos** de ejecución.

Incluso los programas más sencillos se componen de instrucciones más pequeñas, **por ejemplo**:

Programa Java	Instrucciones Bytecode
<code>y = x + 1;</code>	0: load x
<code>x = y;</code>	1: const 1
	2: add
	3: store y
	4: load y
	5: store x

Ejemplo: ¿Cuál será el valor final de x si ejecutamos el programa en dos procesadores con un valor inicial de x = 0?





Proceso 1: (x = x + 1)

i_0 : load x
 i_1 : const 1
 i_2 : add
 i_3 : store x

Proceso 2: (x = x + 2)

i'_0 : load x
 i'_1 : const 2
 i'_2 : add
 i'_3 : store x

Algunas posibles ejecuciones son:

$i_0 \rightarrow i_1 \rightarrow i_2 \rightarrow i_3 \rightarrow i'_0 \rightarrow i'_1 \rightarrow i'_2 \rightarrow i'_3$	\rightarrow	x = 
$i_0 \rightarrow i'_0 \rightarrow i'_1 \rightarrow i'_2 \rightarrow i'_3 \rightarrow i_1 \rightarrow i_2 \rightarrow i_3$	\rightarrow	x = 
$i_0 \rightarrow i'_0 \rightarrow i_1 \rightarrow i'_1 \rightarrow i_2 \rightarrow i'_2 \rightarrow i_3 \rightarrow i'_3$	\rightarrow	x = 
$i'_0 \rightarrow i'_1 \rightarrow i'_2 \rightarrow i'_3 \rightarrow i_0 \rightarrow i_1 \rightarrow i_2 \rightarrow i_3$	\rightarrow	x = 
...		

Operaciones atómicas: son operaciones indivisibles, es decir, deben realizarse de manera completa (o poder deshacerse de manera completa en caso de fallar o ser interrumpidas). Pueden estar formada por una o varias instrucciones y ningún proceso puede acceder a la información modificada hasta que no se hayan completado.

Multiprocessing: varios procesos comparten uno o más procesadores.

Multiprogramming: cada uno de los procesos tiene su propio procesador y se comunican mediante memoria compartida.

Procesos distribuidos: cada uno de los procesos tiene su propio procesador, pero se comunican mediante una red de comunicaciones.

Objetivos de la asignatura:

Los aspectos claves de la concurrencia son:

- ¿Cómo indicamos la ejecución concurrente?
- ¿Qué tipo de comunicación entre procesos debemos utilizar?
- ¿Qué mecanismo de sincronización utilizamos?

Por ello, intentamos dar respuesta a dichas preguntas a través del estudio de los siguientes puntos:

- (1) Diseño y especificación de Programas Concurrentes:
 - (a) Definir las interacciones entre los procesos que componen el sistema
 - (b) Usar técnicas formales para especificar un programa concurrente
 - (c) Detectar las partes del programa que necesitan control de acceso
- (2) Estudiar los modelos de programación concurrente:
 - (a) Programas con concurrencia explícita (comunicación y sincronización)
 - (b) Programación a través de paso de mensajes

(b) Manejo básico de procesos en Java

PROCESOS	THREADS
Se conocen como procesos <i>pesados</i> (heavys)	Se conocen como procesos <i>ligeros</i> (lightweight)
Disponen de su propio entorno de ejecución con sus propios recursos y espacio de memoria	Se crean en el mismo proceso compartiendo recursos (p.e. memoria)
El cambio entre procesos es lento	El cambio de contexto es rápido
Se suelen identificar con programas o apps. El SSOO es el encargado de su gestión	Habitualmente su creación y ejecución se realiza en el propio lenguaje de programación

¿Cómo se implementan los threads en Java? Es necesario implementar una clase donde se implemente el código que ejecutará el thread. Se puede hacer de dos formas:

(1) Extender la clase Thread y sobrescribir el método run

```
public class MiPrimerThread extends Thread {
    public void run () { /* CODIGO AQUI */ }
}
```

(2) Implementar el interfaz Runnable, que requiere implementar el método run:

```
public class MiPrimerRunnable implements Runnable {
    public void run () { /* CODIGO AQUI */ }
}
```

¿Cómo se lanzan los threads en Java?

(1) Crear el objeto de tipo Thread o Runnable*

* **¡OJO!** Solamente se instancia, pero no se “arranca”.

(2) Ejecutar el método start() para lanzar el nuevo thread ejecutando el método run

Extendiendo de Thread	Implementando Runnable
<pre>MiPrimerThread t = new MiPrimerThread(); t.start();</pre>	<pre>Runnable runnable = new MiPrimerRunnable(); Thread t = new Thread(runnable); t.start();</pre>

NOTA: No es lo mismo llamar a start() que a run(). Dado que run() es un método de los de siempre, entonces no se produce concurrencia; mientras que si llamamos a start() sí que se produce concurrencia, ya que estamos arrancando un nuevo hilo de ejecución.

Ejemplo:

```
public class HolaMundos {
    private static class HolaMundo extends Thread {
        int id;
        public HolaMundo(int id) {
            this.id = id;
        }

        public void run() {
            System.out.println("Hola mundo " + id);
        }
    }

    public static void main(String[] args) {
        HolaMundo hola1 = new HolaMundo(1);
        HolaMundo hola2 = new HolaMundo(2);
        hola1.start();
        hola2.start();
        try {
            hola1.join();
            hola2.join();
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Hola, soy 'el main'");
    }
}
```

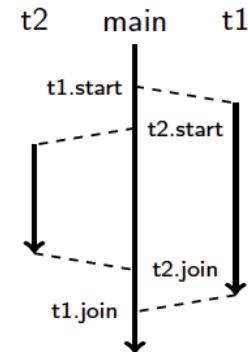
¿Cuántos procesos hay?**Otros métodos:**

(1) El método estático `Thread.sleep(tiempo)` suspende la ejecución del *thread* que está ejecutando un tiempo determinado, donde `tiempo` se expresa en ms y puede lanzar la excepción `InterruptedException`

```
class ThreadDormido extends Thread {
    public void run() {
        try {
            Thread.sleep(5000) ;    // A dormir 5 segundos
        } catch(InterruptedException e) {
            System.out.println ("¡Me han interrumpido!");
        }
    }
}
```

(2) El método `join()` hace que el *thread* en ejecución espere a que el *thread* `t` termine su ejecución. También se puede especificar un tiempo máximo de espera y puede lanzar la excepción `InterruptedException`

```
Thread t1 = new Thread ();
Thread t2 = new Thread ();
t1.start();
t2.start();
try {
    t2.join();
    t1.join();
} catch (InterruptedException e) {
    ...
}
```



Interrupciones:

- Las interrupciones son una indicación de que un hilo debe detener lo que está haciendo para hacer otra cosa
- Llamando al método `t.interrupt()` se puede intentar interrumpir lo que está haciendo *thread* `t`
- Las interrupciones sólo interrumpen el *thread* cuando se está ejecutando un método que lanza la excepción `InterruptedException`
- El programador del *thread* interrumpido es quien decide qué hacer para tratar las interrupciones capturando `InterruptedException`
- Si estamos ejecutando, siempre podemos comprobar si nos han interrumpido a través de `isInterrupted()`

La clase Thread (simplificada):

```
public class Thread extends Object implements Runnable {
    public Thread();
    public Thread(String name);
    // Más constructores

    public String getName();
    public void run();
    public void start();
    public void interrupt();
    public void sleep(long ms) throws InterruptedException;
    public void join() throws InterruptedException;
    public boolean isAlive();
    public final void suspend();
    public final void resume();
    ...
}
```

ENTREGABLE 1

1. Creación de threads en Java:

Con lo visto en clase y la documentación sobre concurrencia en los tutoriales de Java (<http://docs.oracle.com/javase/tutorial/essential/concurrency/>), se pide escribir un programa concurrente en Java que arranque N threads y termine cuando los N threads terminen. Todos los threads deben realizar el mismo trabajo: imprimir una línea que los identifique y distinga (no se permite el uso de `Thread.currentThread()` ni los métodos `getId()` o `toString()` o `getName()` de la clase `Thread`), dormir durante T milisegundos y terminar imprimiendo una línea que los identifique y distinga. El thread principal, además de poner en marcha todos los procesos, debe imprimir una línea avisando de que todos los threads han terminado una vez que lo hayan hecho.

Es un ejercicio muy sencillo que debe servir para jugar con el concepto de proceso intentando distinguir lo que cada proceso hace y el momento en el que lo hace. Además, se podrá observar cómo cada ejecución lleva a resultados diferentes. Se sugiere jugar con los valores de N y T e incluso hacer que T sea distinto para cada proceso.

Material a entregar: el fichero fuente a entregar debe llamarse `CC_01_Threads.java`

EJERCICIOS PARA PRACTICAR

1. Ingresar los nombres de tres corredores y simular una carrera de 100 metros (indicar el paso cada 20 kilómetros) e indicar cuando llegan todos a la meta.
2. Queremos programar dos procesos, uno imprime los números pares del 1 al 10 y la suma de dichos números, y otro imprime los números impares del 1 al 10.
3. Simular con procesos diferentes los estudiantes de una clase que llegan al aula (indicarlo) y se sientan (indicarlo). ¿Cómo implementaríamos la llegada del profesor (otro proceso) después de llegar todos los estudiantes?