# Material complementario 30/03/2020

Arno Formella

Este documento recoge más o menos lo que hubiese contado en clases presenciales a lo largo de la exposición paulatina de las transparencias. En clase normalmente descubro diferentes partes del contenido de cada transparencia poco a poco, para no sobrecargar con información de golpe, y para mantener un hilo de pensamiento. En las transparencias distribuidas todo viene de golpe, por eso recomiendo trabajar con ellas lentamente y ver este material complementario a la par.

Estarán incluidas preguntas (a veces sin respuestas) para animar a la reflexión sobre el tema y a la búsqueda de información adicional. Admás proporciono más referencias a la red y la bibliografía.

Ojo, las páginas mencionadas aquí siempre se refieren a los tochos que publico para cada semana. Puede ser (y es casi seguro) que en el documento global (que desarrollo en paralelo) la enumeración de las página va a variar, ya que se añaden transparencias en otros lugares.

Recapitulamos:

- Estamos en el proceso de implementar una multiplicación ($p * q$) con un programa concurrente.
- Os presenté un algoritmo (secuencial) y hemos comprobado su corrección.
- Si no hemos metido la pata en traducir el pseudo-código a Java el programa debe funcionar perfecto con un proceso.
- Desde la semana pasada he añadido tres líneas más al programa para que me imprima (antes de crear los hilos) lo que intenta calcular, así tendremos tal indicación en la salida.
- He puesto el código ahora en un archivo en la página web.

Proceso para conseguir el programita:

1. `Multi.java` que contiene la clase trabajador que implementa el algoritmo y la clase con el programa principal.
   (Sí, se puede meter varias clases en un fichero en Java!)

2. `Int_simple.java` que contiene la implementación de un entero simple usando internamente simplemente un `int`.
   Ojo, he nombrado el fichero (`Int_simple.java`) diferente que la clase `Int`, así puedo usar los enlaces simbólicos en linux para *implementar* las diferentes versiones en el futuro sin recurrir a la herencia.

3. He creado un enlace simbólico `Int.java` al fichero de implementación `Int_simple.java`.

4. Se compila con:
   `javac Multi.java`
   (y genera los ficheros de las clases: `Multi.class`, `Mul.class` y `Int.class`)

5. Se ejecuta, por ejemplo, con:
   `java Multi 1 1 1`
   (es decir calculamos $1 * 1$ con 1 hilo.)

**P158**

Eso es la misma transparencia de la última de la semana anterior. Si no has hecho lo que estaba pedido en el material complementario de la semana pasada: **¡mal vas!**

Con
`java Multi 1 1 1`
me sale como salida del programa:

```
1*1 with 1 threads
starting worker... 0
exiting... 0
1*1=1 ??
exiting...
```

¡correcto!

Con
`java Multi 1000 1000 1`
me sale como salida del programa:

```
1000*1000 with 1 threads
starting worker... 0
exiting... 0
1000*1000=1000000 ??
exiting...
```

¡correcto!

No me sorprende que es correcto, ya que hemos *comprobado* formalmente que el algoritmo es correcto, y la transformación a Java lo hemos hecho bien!

Vamos a ver si usamos ahora más de 1 hilo (nota que para eso no hemos comprobado nada!)

Con
`java Multi 1 1 2`
me sale como salida del programa:

```
1*1 with 2 threads
starting worker... 1
starting worker... 0
exiting... 1
exiting... 0
1*1=1 ??
exiting...
```

¡correcto!

Con
`java Multi 10 10 2`
me sale como salida del programa:

```
10*10 with 2 threads
starting worker... 0
starting worker... 1
exiting... 0
exiting... 1
10*10=100 ??
exiting...
```

¡correcto!

Que bien, ¿no? el programa funciona, dos pruebas y todo bien. Somos unos genios de la programación concurrente, ¿verdad? Hacemos otra prueba:

Con
`java Multi 1000 1000 2`
me sale como salida del programa:

```
1000*1000 with 2 threads
starting worker... 1
starting worker... 0
exiting... 0
exiting... 1
1000*1000=993000 ??
exiting...
```

¡incorrecto!

y otro vez otro intento con los mismos parámetros:

```
1000*1000 with 2 threads
starting worker... 0
starting worker... 1
exiting... 1
exiting... 0
1000*1000=1037000 ??
exiting...
```

¡incorrecto! ¿y diferente que antes?

entonces tenemos para el programa concurrente:

- Parece que funciona con factores de entrada pequeños.
- Si los factores son grandes, calcula un producto incorrecto, a veces menor a veces mayor que el resultado esperado.
- Observamos que dura (en la mayoria de las veces) mucho tiempo calculando.

En resumen: el programa concurrente es incorrecto.

Antes de seguir miramos que significa *correcto*.

**P159**

Fíjate que lo que expresamos con $x$ es el conjunto de todas las entradas del programa a lo largo de su ejecución, e idem, con $y$ expresamos todas las salidas.

Las propiedades $P$ y $Q$ que expresamos como funciones aquí (basicamente para manifestar que sean verificables), pueden ser cualquier cosa que nos interese.

Por ejemplo, para nuestro caso de multiplicación (caso $P$) sería que tanto $p$ como $q$, pero también $n$, sean números enteros positivos, donde $n$ adicionalmente tenga que ser mayor que 1.
Luego, en principio, deberíamos tener en cuenta que tanto los valores de $p$ y $q$ como el producto $p \cdot q$ sean de tal manera que sean representables con la precisión del programa en Java en la máquina que usemos. Para el caso de $Q$ pues sería que el producto sea correctamente calculado.

**P160**

Con eso tenemos las definiciones de funcionamiento correcto de un programa.

La diferencia entre las dos definiciones es que dado la teoría de computabilidad, no podemos asegurar que cualquier programa dado termina para cualquier entrada (teorema de parada).

En la práctica se suele usar el funcionamiento correcto total, es decir, normalmente queremos que el programa termina (menos los casos bucles infinitos a propósito, por ejemplo, sistemas operativos).

**P161**

En programas concurrente, como ya imaginabamos, el orden de la ejecución de las instrucciones atómicas no sigue un orden temporal estricto (o bien por hay varios procesadores físicos actuando, o bien por se conmutan en tiempo de forma casi aleatoria).

Admás, los procesadores realizan hoy en día en muchos casos una ejecución de sus instrucciones en orden diferente que escrito en memoria (siempre garantizando el mismo resultado).

(Observa esta práctica introduje varios fallos/brechas de seguridad en los sistemas, mira *meltdown* y *spectre* (`https://meltdownattack.com/`) o otros más recientemente (`https://en.wikipedia.org/wiki/Transient_execution_CPU_vulnerability`).

Lo importante aquí, es que definimos la corrección para todos los posibles ordenes de ejecución.

**P162**

Hay que ser consciente, que la conmutación de los hilos en Java en principio puede venir *en cualquier momento* (y según la lei de Murphy, siempre en *mal momento* :-)

Por el segundo punto, es importante también realizar pruebas en diferentes máquinas virtuales de Java y en diferentes sistemas hardware.

**P163**

Entonces, obviamente si encontramos una sola situación cuando el programa no funciona (porque lo estamos viendo en una ejecución, o porque lo estamos derivando teóricamente analizando el código), entonces el programa es incorrecto.

Nota, dado la complejidad de sistemas modernos puede ser que sea imposible reproducir un fallo con alguna máquina virtual de Java o en un sistema concreto.

**P164**

Mira por ejemplo en (`https://math.stackexchange.com/questions/382174/ways-of-merging-two-incomparable-sorted-lists-of-elements-keeping-their-relative`)

¿entonces por qué digo *exponentialmente*?

**P165+166**

Es un ejemplo donde la atomicidad de ciertas instrucciones a nivel hardware puede ser diferente. Por eso el aviso de arriba: si una máquina virtual de Java usa consistentemente tal `inc N` para la sentencia `N++` de java, no habría problema de atomicidad de tal incremento.

Por eso hay que mirar en la especificación del lenguaje cuales son las condiciones que se cumplen (y para Java se dice que el operador `++` no necesariamente es atómico, no he encontrado una referencia concreta).

Usamos: ¡si no está especificado claramente que una acción es atómica, mejor asumir que no lo es!

Observa la especificación del acceso a `double` y `long`: `https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.7`

**P167**

Implementa esta modificación, y observa la salida y el tiempo de cálculo.

Por ejemplo con:
`java Multi 100 400 2`
¿Observas diferencias a tus pruebas con la versión anterior?

**P168**

Seguimos la semana que viene, podéis pensar... ya que estamos confinado en casa, seguimos lentamente con las clases, para no aburrirnos en la semana santa.