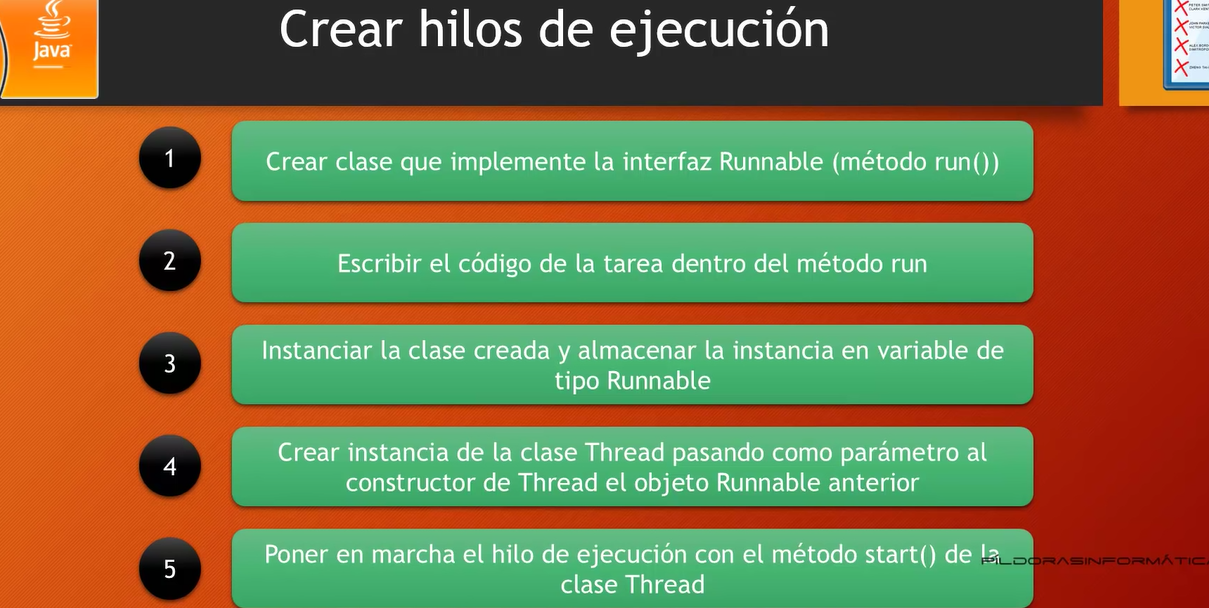
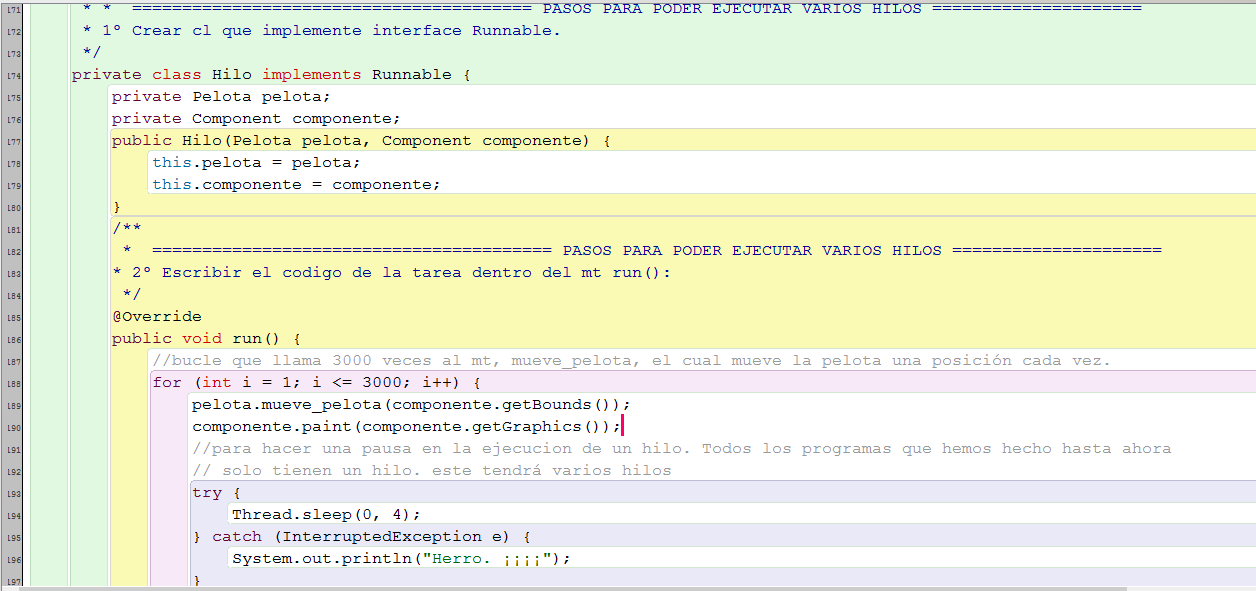
Video 168.

THREADS, programación concurrente, pueden realizar varias tareas simultaneamente.

Hasta ahora hemos programado monotareas,





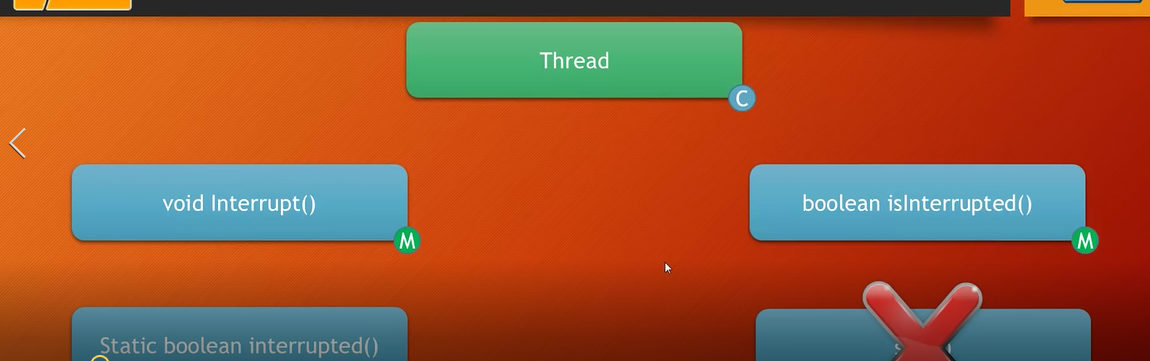
Video 169.

**DETENER LA EJECUCIÓN DE UN THREAD** en un programa de java. Se utilizan varios mt. Los más destacables serían,

, Interrupt(), 🡪 solicita la interrupción de un hilo.

, boolean isInterrupted(), 🡪chequea si este hilo ha sido interrumpido, este mt se aplica a un hilo en concreto.

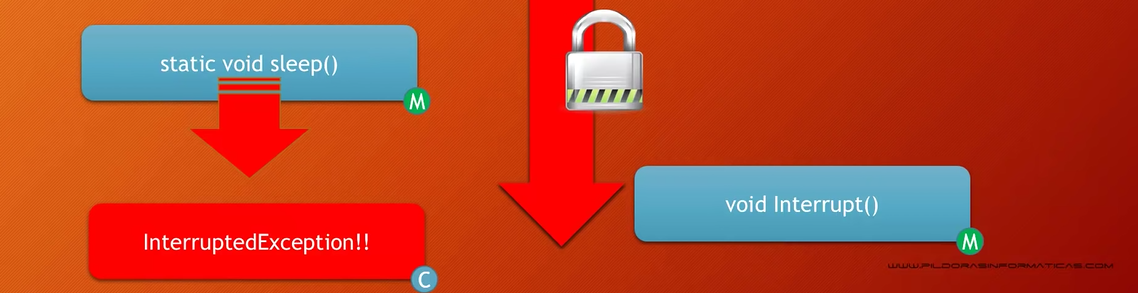
, static boolean interrupted(); 🡪informa si el hilo actual ha sido interrumpido, se aplica a cualquier hilo.



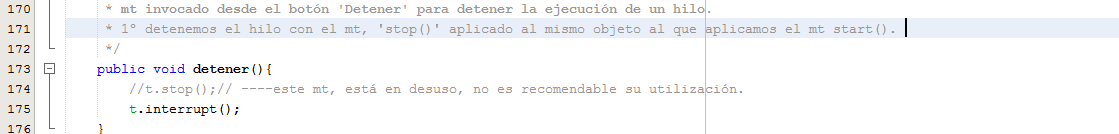
**.DATO¡¡¡**.

Cuando un hilo está ejecutando el mt, **sleep()**, ese hilo está completamente bloqueado, mientras se encuentra en la pausa que produce este mt, no se puede hacer nada con ese hilo, ni siquiera solictar su interrupción.

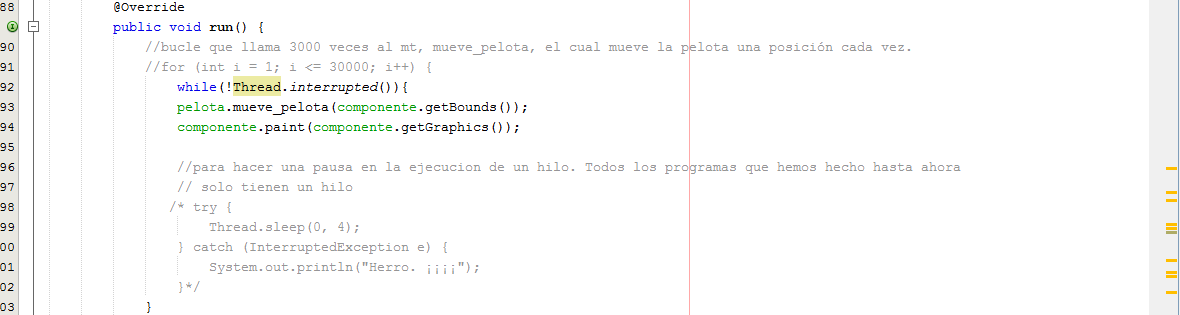
Y qué ocurre si solicitamos una interrupción con el mt, Interrupt(), pues es en ese instante en el que se produce una Excepcion,



**Teniendo esto encuenta, vamos a ver si somos capaces de detener a un Thread en tiempo de ejecución**.



Podemos eliminar el ‘try catch’ lo que nos permitiría ver el movimiento de la bola, pero sí ver que funciona y que detiene la ejecución del hilo. Cambiamos el bucle for por un while que será infinito hasta pulsar el botón.

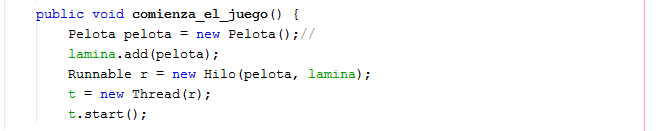


Video 170.

**DETENER**  varios hilos.

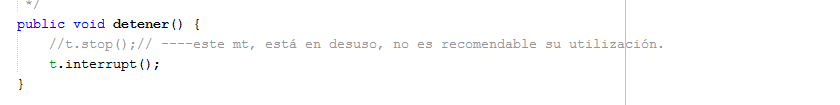
Con el código anterior, si creamos varias pelotas en movimiento, y activamos el mt ‘deterner()’, lo que ocurre es que solo detiene una pelota, la última.

Por qué?

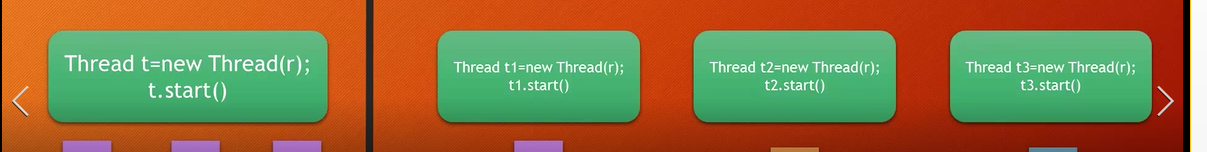


, porque al invocar el mt, ‘comienza\_el\_juego()’ cada vez que creamos una Pelota, todas son la misma. ‘t.start(), y la última que creamos machaca a todas las anteriores quedandose con el nombre ‘t’.

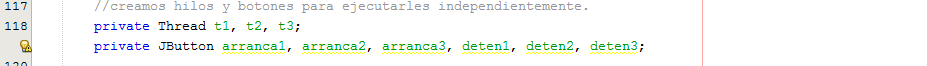
**Y luego el mt, deterner(), solo detiene a la última ‘t’**.



Podemos crear varios nombres para varias ejecuciones.



SOLUCIÓN.



Video 171.

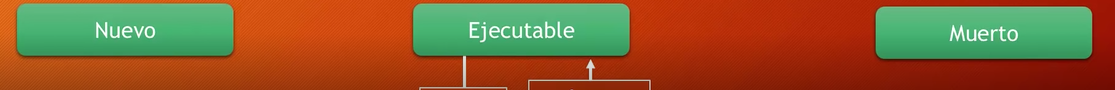
**ESTADO Y SINCRONIZACIÓN DE THREADS.**

**Estados de los hilos**.

**Nuevo** 🡪 el hilo está en estado de ‘nuevo’ cuando todavía no hemos llamado al mt, start().

**Ejecutable** 🡪cuando hemos llamado al mt ‘start()’ y comienza la tarea.’

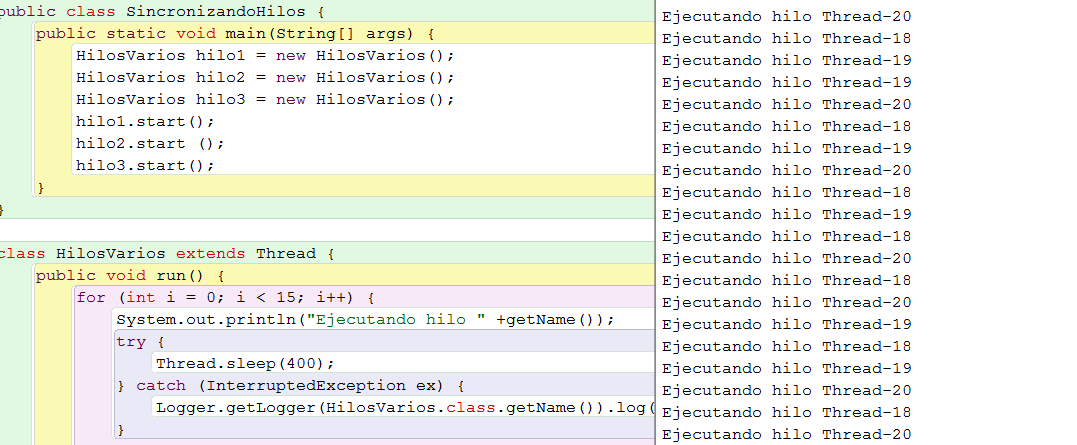
**Muerto** 🡪cuando finaliza el mt, ‘run()’ o cuando salta una Excepcion.



**Sincronización de los hilos**.

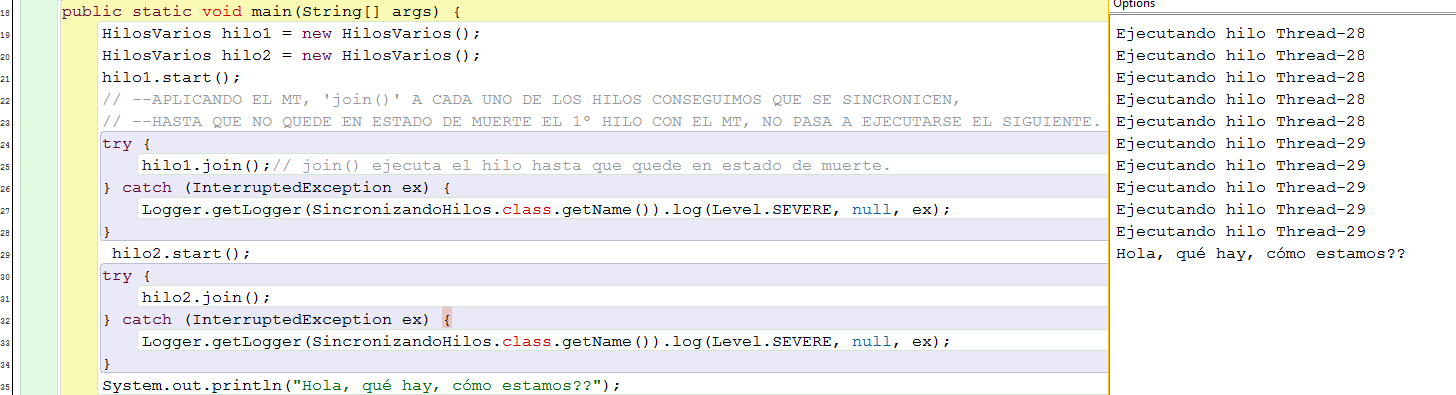
--otra forma de crear un hilo sin que implemente la interface ‘Runnable’ es crear un cl que herede de la cl ‘Thread’,

EN LA IMAGEN **SE VE QUE NO IMPRIME** LOS HILOS **EN EL ORDEN EN QUE ESTÁN COLOCADOS** EN EL CÓDIGO. (el orden de mostrar los hilos en pantalla depende del SO y de los recursos que esté utilizando el ordenador en ese momento.)



HABRA OCOASIONES EN QUE TENGAMOS QUE UTILIZAR UN HILO ANTES QUE OTRO, PARA ELLO SE NECESITA --------🡪 **SINCRONIZAR LOS HILOS**. 🡨-------

PARA SINCRONIZAR LOS HILOS UTILIZMOS EL MT ‘join()’



Video 172.

**ESTADO Y SINCRONIZACIÓN DE THREADS II.**

Para **sincronizar hilos entre si**,

En el vd, anterior se consigue sincronizar los hilos de forma manual, es decir, ordenamos que se ejecute un hilo hasta su muerte y después que se ejecute el siguiente.

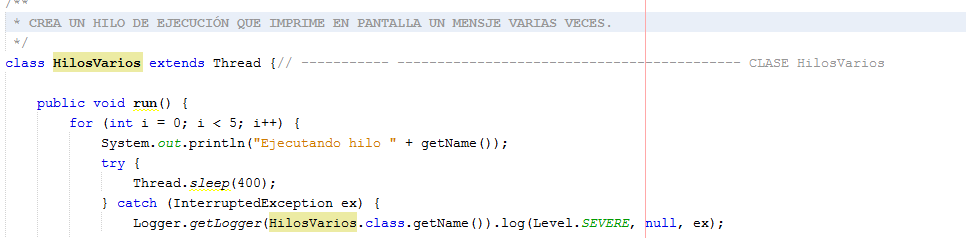
Ocurre que habrá ocasiones en que mientras se esté ejecutando uno de los hilos, tenga que haber otro que esté ejecutando su tarea a la vez.

Tenemos que sincronizar los hilos entre sí sin necesidad de utilizar el mt ‘join()’,

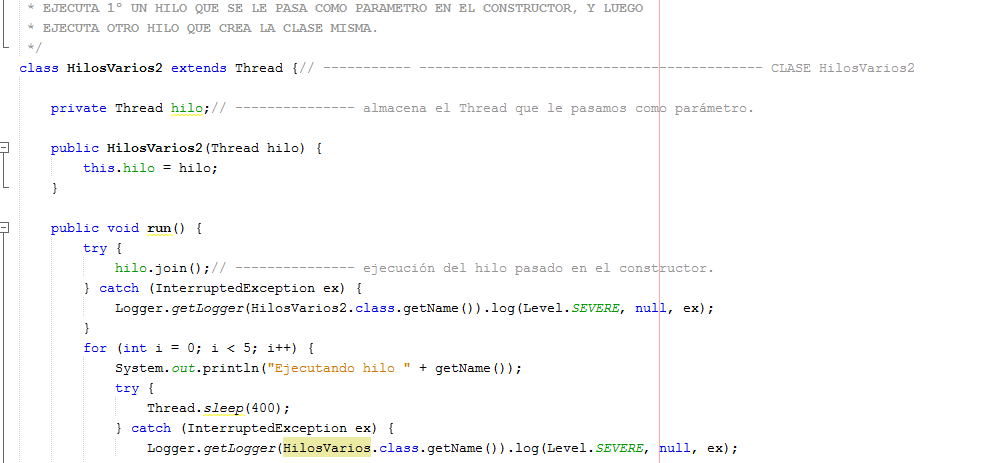
**EJEMPLO**;

Cuando utilizamos una interface gráfica y está realizando tareas, la interface gráfica debe quedar libre, (***mientras realiza una tarea tiene que poder manejarse los menús de la Inte\_G).***

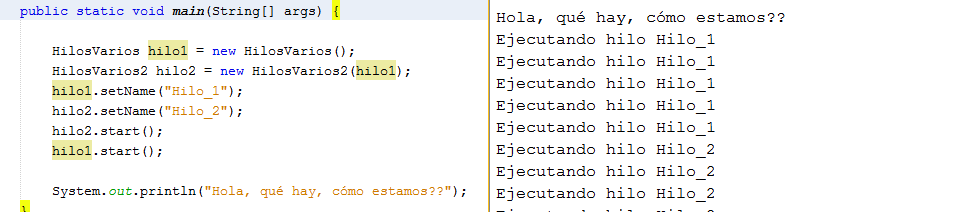
Una manera sencilla de hacerlo sería trabajar con dos hilos, y decirle a uno de los hilos que espere a que el hilo que le pasemos por parámetro termine su tarea.



////////////////////////////////////////////////////////////////////////////////////////////////



//////////////////////////////////////////////////////////////////////////////////////



Video 173.

**ESTADO Y SINCRONIZACIÓN**

Habrá programas en los que los Threads han de estar sincronizados, en otros casos no hará falta.

EJEMPLO.

El siguiente ejercicio representa a un banco con cien cuentas corrientes. Cada cuenta tiene un saldo inicial de 200€, por lo que el saldo total de todas las cuentas es de 200000€.

--La cuenta nº 1 hace una transferencia de 500€ a la cuenta nº 4, por lo que una cuenta se queda en 1500€ y otra cuenta aumenta hasta 2500€, el saldo total del banco sigue siendo de 200000€. Esta operarción la realizará un Thread.

--Lo mismo ocurre con la cuenta nº 2 y la cuenta nº 6, pero con una cantidad de 725€.

Esta operarción la realizará otro Thread.

---Eso es lo que hará el programa. Crea un banco con 100 cuentas, y dentro de un bucle infinito realizará transferencias entre cuentas.



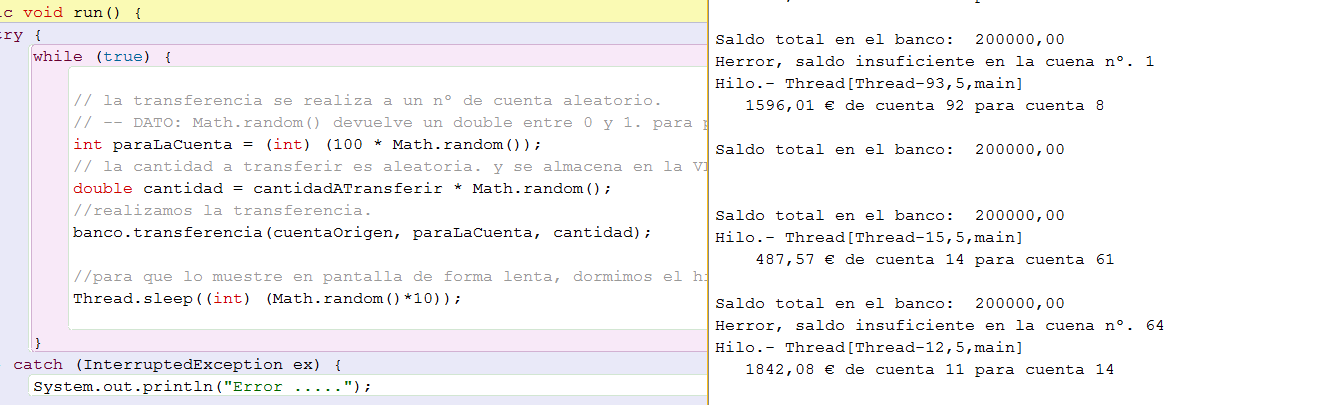
Esta operarción la realizará un Thread.



Video 174.

**CREANDO THREADS.**

Para que realice transferencias de forma infinta.



Video 175.

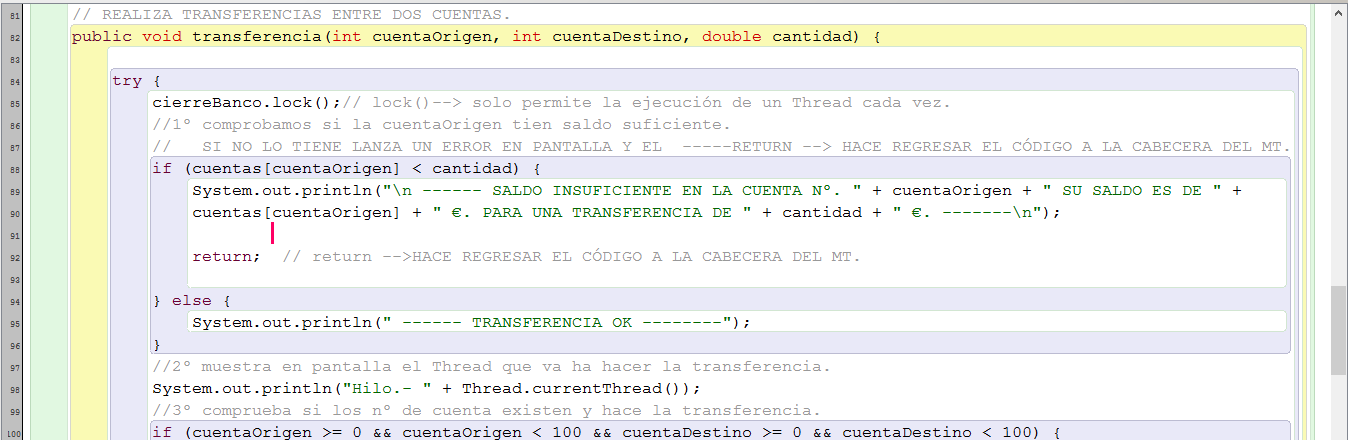
**Cl**  ReentrantLock.

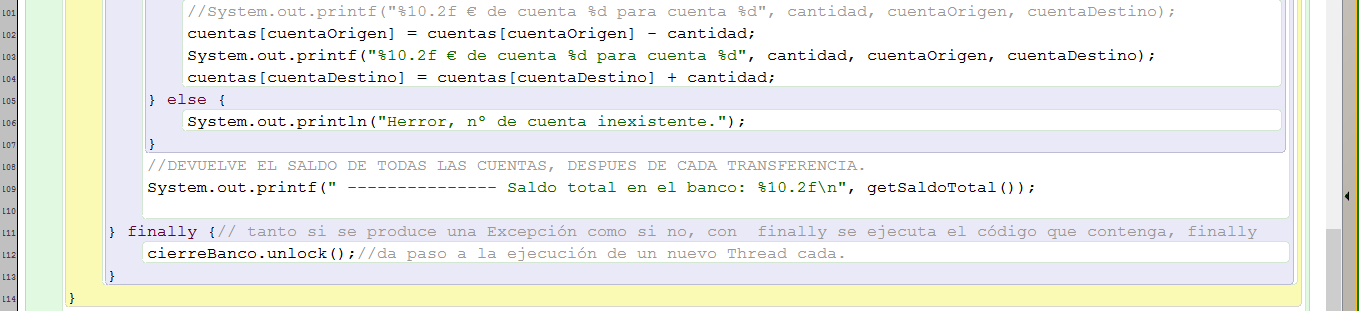
\* 4º Al ser programación concurrente ocurre que se están ejecutando varios Threads a la vez, con lo que uno de ellos puede estar sacando dinero de una cuenta y antes de haberlo ingresado en otra cuenta, otro de los hilos haya imprimido el saldo total de todas las cuenta, con lo que ocurrirá que en algúm momento falte dinero en el total del Banco.

Para solucionar este problema, utilizamos la cl, '**ReentrantLock**' la cual implementa dos interfaces, --la interface **'Lock**' implementa varios mt,

------ el mt ‘**lock()**’ se encarga de bloquear un trozo de código hasta que el hilo que lo esté ejecutando termine de recorrer ese código.

------- el mt **‘unLock()**’ desbloquea el código antes bloqueado.





Video 176.

**Como establecer condiciones a los bloqueos**.

Ocurre que si la cantidad de la transferencia es mayor que el saldo de la cuenta, el ‘**return**’ que tenemos en ese **if** HACE REGRESAR AL PROGRAMA AL FLUJO DEL QUE SE LANZÓ ESTE MT.

Y el ‘Thread’ se pierde.

NO NOS PODEMOS PERMITIR PERMITIR LA PERDIDA DE NINGÚN HILO, Y TODAS LAS TRANSFERENCIAS DEBEN HACERSE

---con el mt, ‘**await()**’, podemos poner el ‘**Thread**’ a la espera, hasta que pueda realizar la operación, además de poner el hilo a la espera, libera el bloqueo del programa para que se ponga en ejecución otro ‘**Thread**’

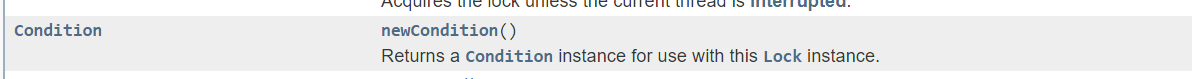
**DATO**. Cada vez que es ha ejecutado un ‘Thread’.

1º este libera el bloqueo para que pueda seguir ejecutandose más hilos.

2º informa a los hilos que están a la espera de la operación que ha realizado, para que conozcan si esta operación les permite ponerse en funcionamiento.

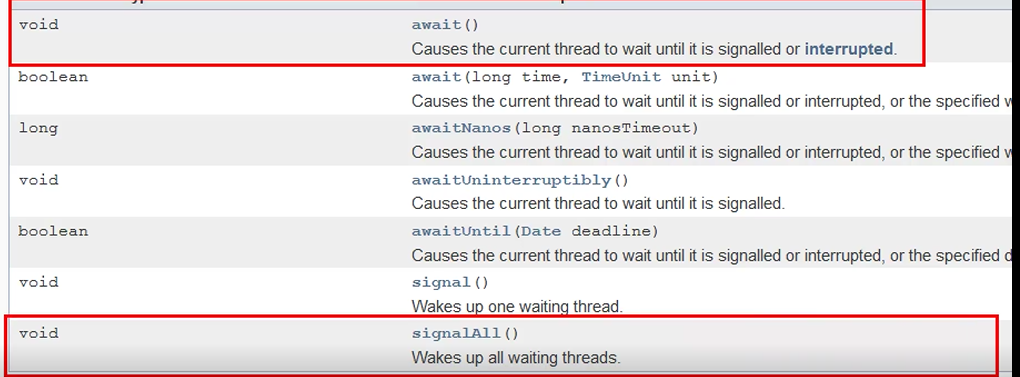
.Si nos vamos a la cl ‘**ReentrantLock**’.

Vemos que es capaz de establecer un bloqueo pero con una condición gracias al mt, ‘newCondition()’



Devuelve un objeto de tipo ‘**Condition**’ que es una inerface.

**Tiene dos mt importantes**.

  
, ‘**await()**’, pone el ‘**Thread**’ a la espera, hasta que pueda realizar la operación, además de poner el hilo a la espera, libera el bloqueo del programa para que se ponga en ejecución otro ‘**Thread**’.

, ‘**signalAll()**’, informa a todos los Threads que están a la espera despierten.

Video 177.

**Aplicamos la cl**  ‘**Condition**’

\* 5º Ponemos una condición al bloqueo producido por al interface 'Lock' utilizando su mt

\* 'newCondition()' que devuelve un objeto de tipo 'Condition' por lo que declaramos

\* una 'variable de campo' de tipo 'Condition' --> 'private Condition saldoSuficiente' para almacenar el objeto

\* que devuelve 'saldoSuficiente = cierreBanco.newCondition()'

\*

\* Luego establecemos la condición en el mt, donde se produce el bloqueo,"mt transferencia()" La condición es

\* que si la cantidad de la transferencia es mayor que el saldo que se tiene, la transferencia

\* no se realice. Y en vez de perder el hilo por esto, que el hilo quede a la espera, hasta que

\* tenga saldo.

\* Para esto eliminamos el return, y utilizamos el mt 'await()' aplicado al objeto 'Condicion' dentro de un bucle while.

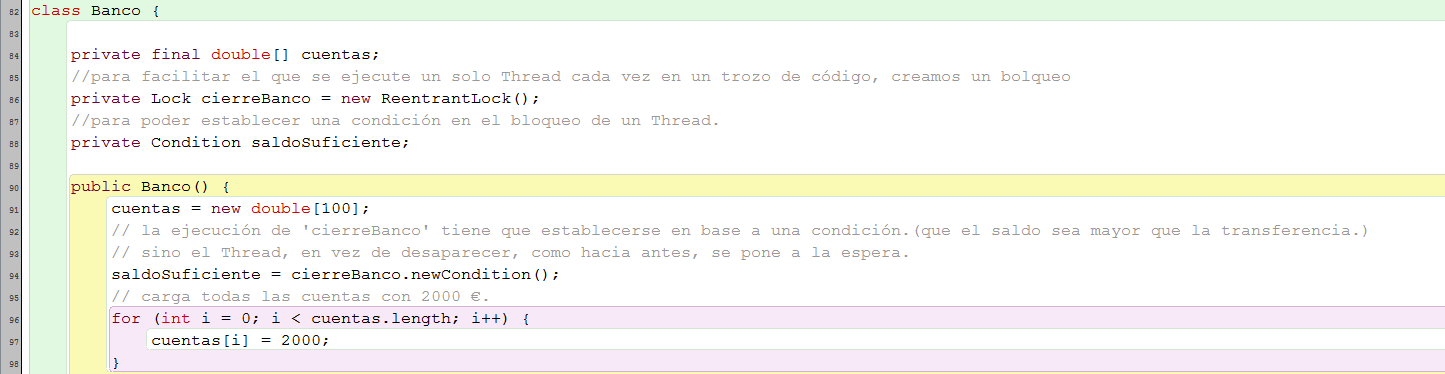
\*

\* Después decimos a los hilos que se han ejecutado correctamente, al terminar su labor que informe a los hilos que

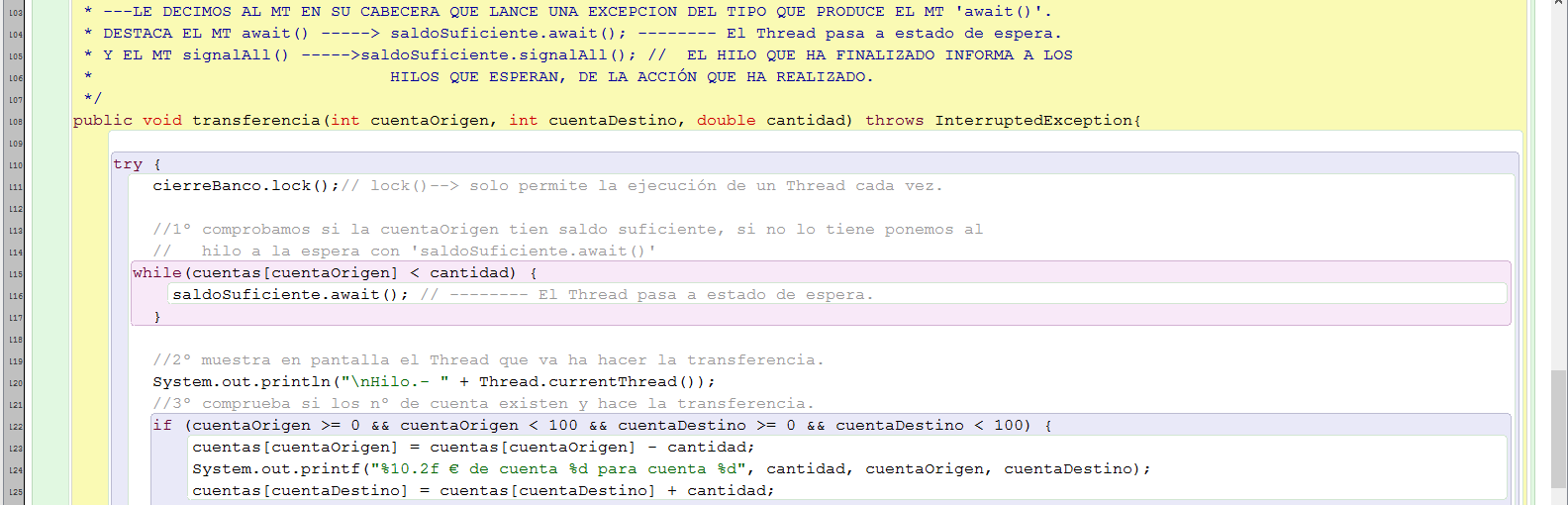
\* están a la espera de la acción que han realizado, utilizando el mt signalAll() ---> saldoSuficiente.signalAll();

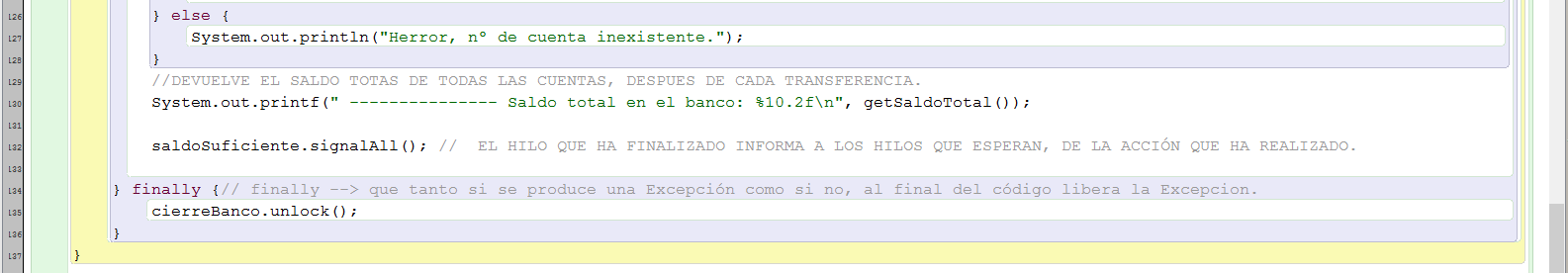
\* =============== DE ESTA NO SE PIERDE NINGÚN HILO, TODOS REALIZARÁN SU FUNCIÓN. ===============================

**Declaramos variables de campo**.



**Las ejecutamos en el mt transferencia()**.

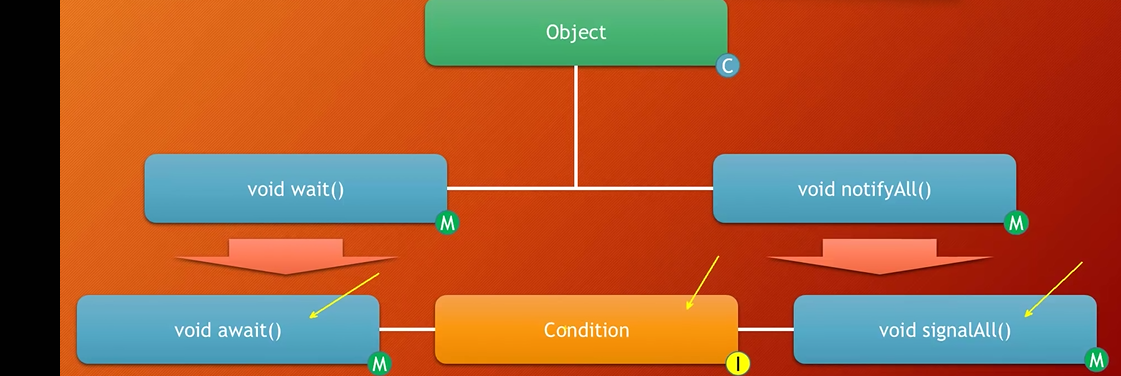




Video 178.

**Uso de la palabra reservada Synchronized**

Lo que hace es establecer bloqueos y una condición, de una forma más sencilla, aunque tiene alguna limitación. A veces merece la pena utilizarla.



v

