**Academia Java – JAVA**

***Multitarea***

Tabla de contenido

[1. Threads y threads 3](#_Toc314826266)

[1.1. ¿Qué significa tener más de una pila de llamadas? 3](#_Toc314826267)

[1.2. ¿Cómo lanzar un nuevo hilo? 5](#_Toc314826268)

[1.3. Cada Thread necesita un trabajo que hacer. 6](#_Toc314826269)

[1.4. Para crear un job para tu thread, implementa la interface Runnable 6](#_Toc314826270)

[1.5. Los tres estados de un nuevo thread 7](#_Toc314826271)

[1.6. Un thread puede estar temporalmente no runnable 9](#_Toc314826272)

[1.7. El programador de procesos (o Thread Scheduler) 10](#_Toc314826273)

[1.8. Mandando a dormir al thread 16](#_Toc314826274)

[1.9. Creando e iniciando dos threads 17](#_Toc314826275)

[1.10. Un matrimonio en problemas 21](#_Toc314826276)

[1.11. El temido problema de “Lost Update” 28](#_Toc314826277)

[1.12. El lado mortal de la sincronización 30](#_Toc314826278)

# Threads y threads

|  |  |
| --- | --- |
| Podemos hablar de threads –si, con “t” minúscula, y de Threads con mayúscula.  Cuando hablamos de threads nos referimos a un hilo de ejecución o mejor dicho a una pila de llamadas.  Sin embargo, cuando hablamos de Thread, directamente pensamos en la convención de nombres de Java. Y son precisamente los nombres de clases e interfaces los que inician con mayúsculas. Por ello, cuando nos referimos a Thread, no es referimos al objeeto que representa un hilo de ejecución. |  |

|  |  |
| --- | --- |
|  |  |

# ¿Qué significa tener más de una pila de llamadas?

Con más de una pilla de llamadas, puedes dar la apariencias de hacer que varias cosas sucedan al mismo tiempo. En realidad, solo un verdadero sistema multiprocesador podria hacer más de una cosa al mismo tiempo, pero con los hilos en Java puedes aparentar hacer muchas cosas simultaneamente. Es decir, la ejecución se mueve tan rápido entre las pilas que da la impresión que todo está ejecutandose al mismo tiempo. Recuerda que Java es solo un proceso ejecutandose de forma subyacente al sistema operativo.

Cuando se usa varias pilas, se ejecuta primero lo que está en la cima de la pila, pero en 100 milisegundos, el código actual podria dejar de ser ejecutado y se pasa a un método diferente de una pila diferente.

Una de las cosas que un hilo debe hacer es no perder de vista la declaración que está ejecutando actualmente.

|  |  |
| --- | --- |
|  | La JVM llama al método main()  **public** **static** **void** main(String[] args) {  } |
|  |  |
|  | El main() inicia un nuevo hilo. El hilo principal es temporalmente detenido mientras que el nuevo hilo se empieza a ejecutar.  Runnable r = **new** MyThreadJob();  Thread t = **new** Thread(r);  t.start();  Dog d = **new** Dog(); |
|  |  |
|  | La JVM cambia entre el nuevo hilo (usuario hilo A) y el hilo principal original, hasta que ambos estén completos. |
|  |  |

# ¿Cómo lanzar un nuevo hilo?

|  |  |
| --- | --- |
|  | Crear un objeto Runnable (el hilo del trabajo)  Runnable threadJObj = **new** MyRunnable();  Runnable es la interface que revisarás más adelante. Por lo pronto, te comento que implementarás esta interface en una clase y que esa clase definirá el trabajo que un hilo ejecutará. Es decir, el método que se ejecutará desde la pila del nuevo hilo. |
|  |  |
|  | Crear un objeto de la clase Thread (el worker) y asignarle un Runnable (el job)  Thread t = **new** Thread(threadJObj);  Pásale el objeto Runnable al constructor de Thread. Esto le dice al nuevo objeto de Thread cual método debe colocar al fondo de la nueva pila –el método run() de Runnable. |
|  |  |
|  | Inicia el hilo  t.start();  Nada va a pasar hasta que llames al método start() de Thread. Aquí es cuando se pasa de tener una instancia de Thread a tener un nuevo hilo en ejecución. Cuando el nuevo hilo inicia, toma el método run() del objeto Runnable y lo coloca al fondo de la pila del nuevo hilo. |
|  |  |

# Cada Thread necesita un trabajo que hacer.

**Un método que debe ponerse en la pila del nuevo hilo.**

|  |  |
| --- | --- |
|  | Runnable es a un Thread como un job es a un worker. Un Runnable es el trabajo (job) del hilo que se va a ejecutar.  Un Runnable toma el método que va al fondo de la pila del nuevo thread: run(). |

Un objeto Thread necesita un job. Un job que el thread ejecutará cuando é inicie. Ese trabajo es realmente el primer método que va a la pila del nuevo thread y siempre debe ser un método que se vea como:

**public** **void** run (){

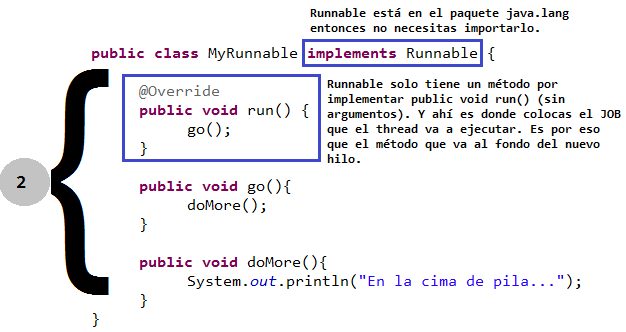
//codigo que se ejecutará por el nuevo hilo

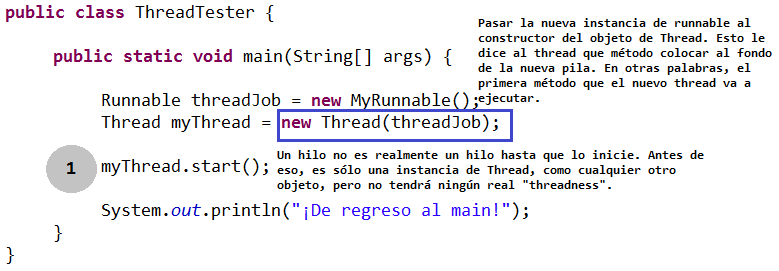
}

Pero ¿cómo sabe el thread que método colocar en el fondo de la pila? Porque Runnable define un contrato. Porque Runnable es una interface. El job de un thread puede ser definido en cualquier clas que implemente la interface Runnable. El thread se preocupa que solo le pase a su constructor un objeto de una clase que implemente Runnable.

Cuando le pasas un objeto Runnable al constructor de un Thread, realmente le estás dado a Thread una forma de obtener el método run(). Le estás dando al Thread el trabajo por ejecutar.

# Para crear un job para tu thread, implementa la interface Runnable





|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

# Los tres estados de un nuevo thread



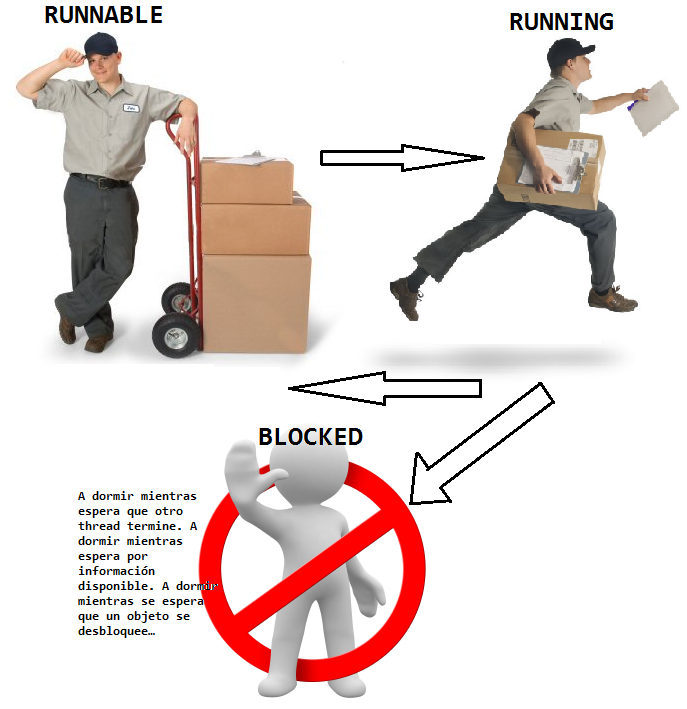
|  |  |  |
| --- | --- | --- |
|  | | |
| Una instancia de Thread que ha sido creado pero no iniciado. En otras palabras, hay un objeto Thread, pero no hay un thread en ejecución | Cuando inicias el thread, pasa a un estado runnable. Esto significa que el thread está listo para ejecutarse y solo esperando la “gran oportunidad” de ser elegido para ejecutarse. En este punto, ya hay una nueva pila de llamadas que le pertenece a este thread. | Aquí todos los threads quieren llegar. Ser el elegido. El hilo actual ejecutándose. Solo el programador de procesos de la JVM puede tomar esta decisión. A veces podrás influir en la decisión, pero no puede forzar a un thread a pasar de runnable a running. En el estado running, un thread (y UNICAMENTE este thread) tiene la pila de llamadas activa, y el método en la cima de la pila está ejecutándose. |

**Pero una vez más: una vez que el thread se convierte en runnable, puede ir y venir entre runnable y running…y un estado adicional: temporalmente no runnable (conocido como blocked).**



# Un thread puede estar temporalmente no runnable

El programador de procesos puede cambiar a un thread de estado running a blocked, por muchas razones. Por ejemplo, el thread podrías estar ejecutando código para leer desde un Socket pero no hay nada que leer. El programador de la JVM decidirá inteligentemente ponerlo en blocked hasta que haya data disponible. Y así, sucesivamente podríamos encontrar razones para poner a un thread temporalmente no runnable.



# El programador de procesos (o Thread Scheduler)

El thread scheduler toma todas las decisiones acerca de quién va de runnable a running, y acerca de cuándo (bajo determinadas circunstancias) un thread dej el estado running. El scheduler decide quien se ejecuta y por cuanto, y a dónde van los threads cuando decide sacarlos del estado running.

No vas a poder controlar el scheduler. No hay forma de llamar métodos para hacerlo. ¡Lo más importante es que no hay garantías acerca de la programación! (aunque hay formas que tampoco garantizan mucho y que son algo confusas)

El fondo de todo esto es: no bases la implementación de tu programa en el shceduler de forma particular! Las implementaciones del schedules son diferentes para diferentes JVM, e incluso ejecutar el mismo programa en la misma máquina puede dar resultados diferentes. Uno de los errores más comunes cuando iniciamos a programar con Threads es probar un programa multihilos en una sola máquina y asumir que el comportamiento obtenido será el de siempre, en cualquier momento y cualquier lugar. Esto no es así.

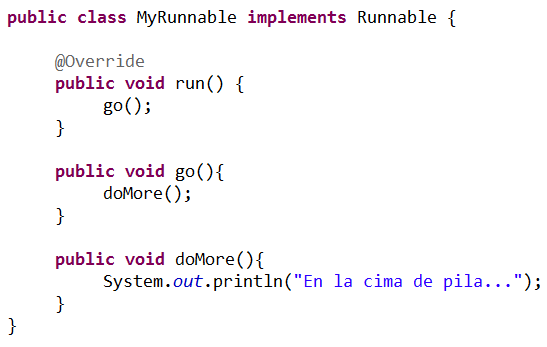
Entonces ¿qué significó eso de escribe-una-vez-y-ejecútalo-donde-sea? Significa que para escribir código independiente de plataforma Java, u programa multihilos debe trabajar sin importar como el scheduler se comporte. Esto quiere decir que no puedes depender, por ejemplo, que el scheduler se asegure que todos los threads estén en estado running. Esto no sucede frecuentemente hoy en día, tu programa podría terminar con la JVM diciendo “OK, thread 5, qué estás haciendo, ahhh pues entonces puedes continuar hasta que termines de ejecutar tu método run()”.

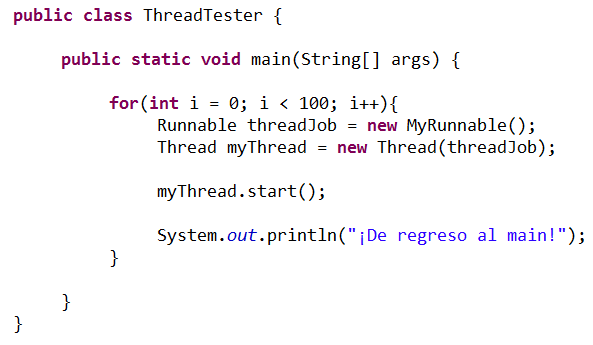
El secreto de todo es *dormir*. Si, como lo lees *dormir*.



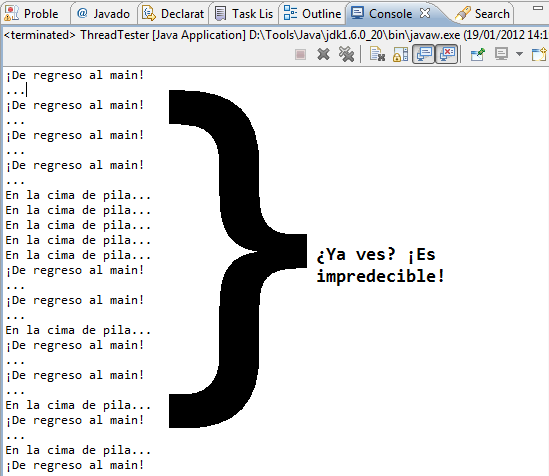
Colocar el thread a dormir, por unos milisegundos, fuerza al hilo actual a dejar el estado running, esto le da chance a otro thread para ejecutarse. El método sleep() del thread viene con una garantía: un thread durmiendo no se ejecutará (no será el actual) hasta que termine el tiempo asignado de *sueño*.

**Un ejemplo de lo impredecible que el scheduler puede ser…**





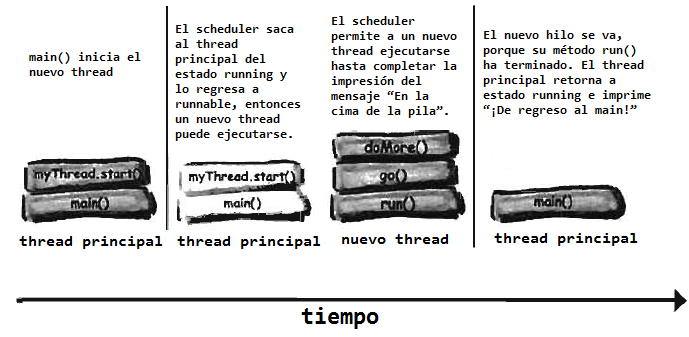
Y obtenemos:



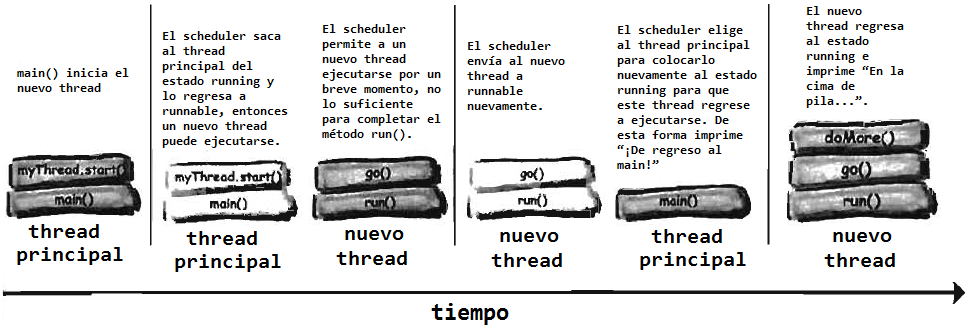
Y si lo vuelves a ejecutar verás que el resultado será diferente.

**¿Cómo es posible que se obtengan diferentes resultados?**

A veces pasa esto…



Y a veces esto…

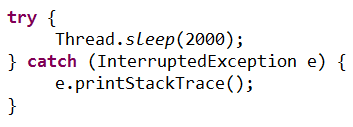
****

# Mandando a dormir al thread

|  |  |
| --- | --- |
| Una de las mejores formas de ayudar a tus threads a tomar turnos es poniéndolos a dormir periódicamente. Todo lo que necesitas hacer es llamar al método estático sleep(), y darle la duración del descanso en milisegundos.  Por ejemplo:  Thread.*sleep*(2000); |  |

Esto saca de juego al thread, es decir, lo saca del estado running y lo manda al estado runnable por dos segundos. El thread no puede regresar al estado running hasta que se cumplan los dos segundos indicados.

Pero tenemos el inconveniente que el método sleep lanza una excepción llamada InterruptedException (una excepción verificada), entonces esto nos obliga a declararlo dentro de un bloque try/catch:

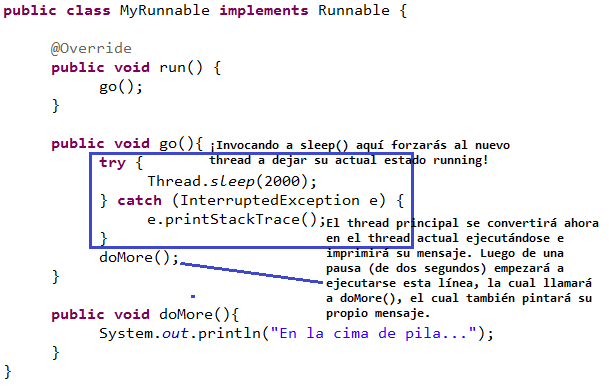


Ahora lo sabes: tu thread no despertará hasta que cumpla el tiempo indicado. Pero también debes tomar en cuenta que es posible que al terminar ese tiempo el thread no pasará directamente al estado running. Sino que tomará el estado runnable nuevamente y como siempre quedará a criterio del scheduler cuando regresa al estado running.

**Usando sleep() para hacer nuestro programa un poquito más predecible**

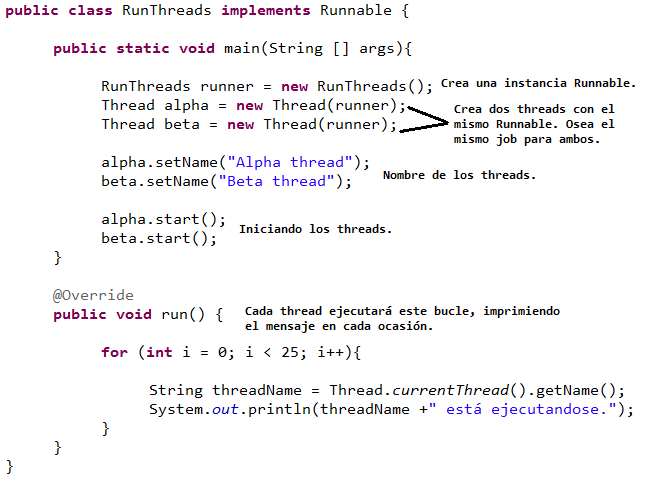
Regresando un poquito hacia nuestro ejemplo donde obteníamos diferentes resultados…a veces el principal debe esperar hasta que el nuevo thread termine, otras veces el nuevo thread regresa a runnable antes de terminar, permitiendo que el principal regrese a terminar de ejecutarse. ¿Cómo podemos corregirlo? Detente un ratito y responde lo siguiente ¿Dónde puedes colocar un sleep() para asegurarte que antes de “regresar al main” siempre se termina de ejecutar el nuevo thread?

Mientras piensas tu respuesta (o respuestas), mira lo siguiente y analízalo.

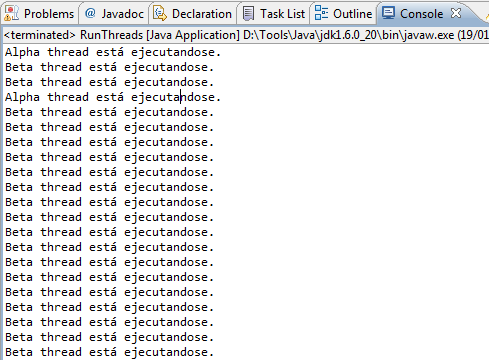


# Creando e iniciando dos threads

Los threads tienen nombres. Puedes darles nombres o aceptar nombres por defecto. Lo cool de esto es que puedes usar esos nombres para saber quien está ejecutándose.



**¿Qué pasará?**









*Los problemas de concurrencia llevan a los race conditions.*

*Los race conditions llevan hacia la corrupción de información.*

*La información corrupta lleva al miedo…y ya te sabes el resto.*

|  |  |
| --- | --- |
| Todo se reduce a un escenario potencialmente mortal: dos o más threads tienen acceso a los datos de un mismo objeto. Es otras palabras, métodos de dos pilas diferentes llamando, por ejemplo, a getters y setter de un mismo objetos en el heap. |  |

Esto pasa por un tema de “que-tu-mano-izquierda-no-sepa-lo-que-tu-derecha-esta-haciendo”. Dos threads, sin más preocupación en el mundo, cada uno marcha bien en sus ejecuciones y trabajos, cada uno pensando que es “EL THREAD”. El único que importa. Después de todo, cuando un thread no está en running, está en runnable (o en blocked).

Un ejemplo de la vida real para entender un poco más este caso.

# Un matrimonio en problemas

|  |  |
| --- | --- |
| La historia de hoy se centrará en una de las primeras razones por las cuales las parejas se separan: finanzas y sueño.  La pareja con problemas el día de hoy está conformada por Ryan y Mónica, comparten la casa y la cuenta bancaria. Lo cual no será por mucho tiempo más si no los ayudamos. ¿El problema? Clásico: dos personas – una cuenta bancaria.  Mónica lo describe de la siguiente forma: |  |

*Ryan y yo acordamos que ninguno iba a sobregirar la cuenta corriente. Por lo que acordamos un procedimiento: el que desee retirar dinero debe comprobar el saldo de la cuenta antes de proceder a retirar. Todo parecía simple hasta ese momento. Pero de pronto nos llenamos de cheques sin fondos y estábamos sobregirados. ¡Pensé que era imposible! ¡Nuestro procedimiento era seguro! Pero entonces me di cuenta que había pasado.*

*Ryan necesitaba $ 50, por lo que luego de comprobar el saldo de la cuenta y ver que había $ 100, se dio cuenta que no habría problemas. Por lo tanto iba a retirar el dinero cuando se quedó ¡dormido!*

*En ese momento entré yo. Mientras Ryan estaba durmiendo necesité retirar $ 100. Luego de consultar el saldo, que era de $ 100 (pues Ryan estaba durmiendo y no había hecho su retiro), noté que no habrían problemas. Así que procedí retirar el dinero y todo estaba bien. Pero entonces Ryan se despertó, completó tu retiro y aquí vino el problema. Él ni siquiera se dio cuenta que se había quedado dormido, por lo que simplemente terminó su transacción y no verificó el saldo de nuevo.*

*¡Tiene que ayudarnos Dr. Steve!*

¿Existe solución? ¿Están condenados? No podemos detener el descanso de Ryan, pero podemos asegurarnos que Mónica no ponga sus manos en la cuenta bancaria hasta que él se despierte?

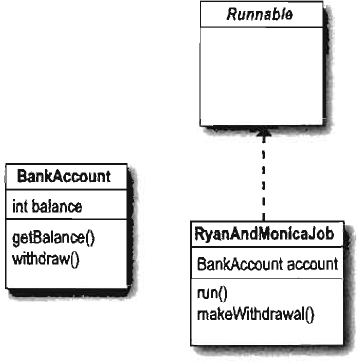
Tomate un momento y piensa que podemos aconsejarle a esta pareja. Mientras vamos a una pausa comercial.

**El problema de Ryan y Mónica…en código**

El siguiente ejemplo va a mostrar que puede pasar cuando dos threads (Ryan y Mónica) comparten un mismo objeto (la cuenta de banco).

El código tiene dos clases, BankAccount y MonicaAndRyanJob. La clase MonicaAndRyanJob implementa Runnable, y representa el comportamiento que Ryan y Mónica tiene –verificar el saldo y hacer retiros. Pero, cada thread cae dormido entre la verificación del balance y el retiro.

Además, la clase MonicaAndRyanJob tiene una variable de instancia del tipo BankAccount, que representa la cuenta compartida.



Veamos que hace este código:

|  |  |
| --- | --- |
|  | Crea una instancia de RyanAndMonicaJob  La clase RyanAndMonicaJob es la clase Runnable (el trabajo por hacer). Solo necesitaremos una sola instancia. |
|  | RyanAndMonicaJob theJob = **new** RyanAndMonicaJob(); |
|  | Crear dos threads con la misma clase Runnable |
|  | Thread one = **new** Thread(theJob);  Thread two = **new** Thread(theJob); |
|  | Dale nombre a los threads e inicializalos: |
|  | one.setName("Ryan");  two.setName("Monica");  one.start();  two.start(); |
|  | Observa ambos threads ejecutar su método run() |
|  | **if** (account.getBalance() >= amount) {  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  } |

El código seria:

**public** **class** BankAccount {

**private** **int** balance = 100;

**public** **int** getBalance() {

**return** balance;

}

**public** **void** withdraw(**int** amount){

balance = balance - amount;

}

}

**public** **class** RyanAndMonicaJob **implements** Runnable {

**private** BankAccount account = **new** BankAccount();

**public** **static** **void** main(String[] args) {

RyanAndMonicaJob theJob = **new** RyanAndMonicaJob();

Thread one = **new** Thread(theJob);

Thread two = **new** Thread(theJob);

one.setName("Ryan");

two.setName("Monica");

one.start();

two.start();

}

@Override

**public** **void** run() {

**for** (**int** x = 0; x < 10; x++) {

makeWithdrawal(10);

**if**(account.getBalance() < 0){

System.*out*.println("Overdrawn!");

}

}

}

**private** **void** makeWithdrawal (**int** amount) {

**if** (account.getBalance() >= amount) {

System.*out*.println(Thread.*currentThread*().getName() + " is about to withdraw.");

**try** {

System.*out*.println(Thread.*currentThread*().getName() + " is going to sleep.");

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.*out*.println(Thread.*currentThread*().getName() + " woke up.");

account.withdraw(amount);

System.*out*.println(Thread.*currentThread*().getName() + " complete the withdraw.");

} **else** {

System.*out*.println("Sorry! Not enough for " + Thread.*currentThread*().getName());

}

}

}

|  |  |
| --- | --- |
| Ryan is about to withdraw.  Ryan is going to sleep.  Monica is about to withdraw.  Monica is going to sleep.  Ryan woke up.  Ryan complete the withdraw.  Ryan is about to withdraw.  Ryan is going to sleep.  Monica woke up.  Monica complete the withdraw.  Monica is about to withdraw.  Monica is going to sleep.  Ryan woke up.  Ryan complete the withdraw.  Ryan is about to withdraw.  Ryan is going to sleep.  Monica woke up.  Monica complete the withdraw.  Monica is about to withdraw.  Monica is going to sleep.  Ryan woke up.  Ryan complete the withdraw.  Ryan is about to withdraw.  Ryan is going to sleep.  Monica woke up.  Monica complete the withdraw.  Monica is about to withdraw.  Monica is going to sleep.  Ryan woke up.  Ryan complete the withdraw.  Ryan is about to withdraw.  Ryan is going to sleep.  Monica woke up.  Monica complete the withdraw.  Monica is about to withdraw.  Monica is going to sleep.  Ryan woke up. | Ryan complete the withdraw.  Ryan is about to withdraw.  Ryan is going to sleep.  Monica woke up.  Monica complete the withdraw.  Sorry! Not enough for Monica  Sorry! Not enough for Monica  Sorry! Not enough for Monica  Sorry! Not enough for Monica  Sorry! Not enough for Monica  Ryan woke up.  Ryan complete the withdraw.  Overdrawn!  Sorry! Not enough for Ryan  Overdrawn!  Sorry! Not enough for Ryan  Overdrawn!  Sorry! Not enough for Ryan  Overdrawn!  Sorry! Not enough for Ryan  Overdrawn! |

**¡Ellos necesitan bloquear el acceso a su cuenta!**

El bloqueo debe funcionar así:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Debe haber un bloqueo asociado con cada transacción en la cuenta bancaria (verificación del balance y retiro de dinero). Sólo debe existir una cerradura, y se queda bloqueada hasta que alguien quiere acceder a la cuenta. |  | La operación cuenta bancaria se desbloquea cuando nadie está utilizando la cuenta. |
|  | Cuando Ryan quiera acceder a la cuenta (para verificar el saldo y retirar dinero), el bloquea la cerradura y se lleva la llave al bolsillo. De esa forma, nadie podrá acceder a la cuenta, ya que la llave no está a la mano. |  | Cuando Ryan quiere acceder a la cuenta, el asegura la cerradura y se lleva la llave. |
|  | Ryan se queda con la llave en su bolsillo hasta que termine la transacción. Solo el tiene la llave, entonces Mónica no podrá acceder a la cuenta hasta que Ryan bosbloquee la cuenta y regrese la llave.  Ahora, si Ryan se quedara dormido después de la verificación del saldo, está garantizado que este seguirá siendo el mismo cuando despierte, porque el se queda con la llave mientras duerme. |  | Cuando Ryan termina la transacción, desbloquea la cerradura y regresa la llave a su lugar. Y ahora esta se encontrará disponible cuando Mónica (o Ryan nuevamente) desee acceder a la cuenta. |

**Necesitamos que el método makeWithDrawal() se ejecute atómicamente!**

|  |  |
| --- | --- |
| Hay algo importante que agregar y es que necesitamos asegurarnos completamente que el método makeWithdrawal() se ejecute completo. Este debe terminar de ejecutarse antes que otro thread pueda ejecutarlo.  Es decir, necesitamos asegurarnos que una vez el thread ha verificado el saldo, el mismo thread garantice que va a estar despierto para terminar el retiro antes que cualquier otro thread pueda verificar el saldo. |  |

Para ellos usaremos el keywork **synchronized** para modificar el método y lograr que un solo thread a la vez pueda accederlo.

Así un solo thread obtendrá el control completo de la transacción, de inicio a fin, y nunca se dormirá en medio de la ejecución del método.

**private** **synchronized** **void** makeWithdrawal (**int** amount) {

…

}

Cada objeto en Java tiene una cerradura. La cerradura tiene una sola clave. La mayoría de las veces, la cerradura está desbloqueada y a nadie le importa. Pero si un objeto tiene métodos sincronizados, un thread puede entrar en uno de los métodos de sincronización sólo si la llave para bloquear el objeto está disponible. En otras palabras, sólo si otro thread no ha tomado ya la llave de una.

# El temido problema de “Lost Update”

Este es otro problema clásico y más o menos gira en torno a:

Paso 1: Obtener el saldo de la cuenta

int i = balance;

Paso 2: aumentar en uno el saldo:

balance = i + 1;

Como ves aquí se fuerza a que la computadora haga en dos pasos el cambio de saldo. En el mundo real, podrías hacerlo en uno solo:

balance++;

Entonces forzando a hacer esto en dos pasos, el problema con el proceso no atómico se vuelve claro. Imagínate que no sea algo tan trivial como aumentar el saldo en uno, sino que sea algo más complejo y no puedas hacerlo en una sola sentencia.

En el problema “Lost Update” tenemos dos threads intentando incrementar el saldo:

**public** **class** TestSync **implements** Runnable {

**private** **int** balance;

@Override

**public** **void** run() {

**for**(**int** i = 0; i < 50; i++){

increment();

System.*out*.println("balance is "+balance);

}

}

**public** **void** increment(){

**int** i = balance;

balance = i + 1;

}

}

**public** **class** TestSyncTest {

**public** **static** **void** main(String[] args) {

TestSync job = **new** TestSync();

Thread a = **new** Thread(job);

Thread b = **new** Thread(job);

a.start();

b.start();

}

}

Podríamos obtener como resultado:

|  |  |
| --- | --- |
|  | balance is 1  balance is 2  balance is 3  balance is 5  balance is 6  balance is 7  balance is 8  balance is 9  balance is 10  **balance is 11**  **balance is 4**  **balance is 12** |

Bueno no te alteres tanto ☺ esto se puede solucionar usando synchronized, lo cual hará indestructible la unidad del método:

**public** **synchronized** **void** increment(){

**int** i = balance;

balance = i + 1;

}

# El lado mortal de la sincronización

|  |  |
| --- | --- |
| Hay que ser sumamente cuidadosos cuando usamos código sincronizado, porque nada va a salvar a tu programa de un thread en deadlock.  Un deadlock sucede cuando tienes dos hilos y ambos están tomando la llave que el otro quiere. No hay forma de salir de este escenario, porque los dos hilos esperan… esperan… esperan… esperan…esperan… |  |

Si este escenario te es familiar con las bases de datos u otros servidores de aplicaciones, puedes reconocer el problema. Sin embargo, sabes que las bases de datos ofrecen un mecanismo para el bloqueo ante algo como la sincronización. Podrían aplicar un transaction rollback y así salvar la situación.

Pero Java no tiene esto. Entonces uno de los tips para contrarrestar esto es prestar mucha atención al orden en el que tus threads se inician.

Algo así…

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  | |
|  |  |  |