Threads

# Temas

1. Crear y usar la clase Thread y la interface Runnable
2. Manejar y controlar el ciclo de vida de un Thread
3. Sincronizar acceso a threads para compartir datos
4. Identificar problemas potenciales en el uso de threads

# Introducción

Estos días, cuando compras una computadora, puedes ver las etiquetas como *dual core*, *quad core*, etc., para describir el tipo de procesador dentro del sistema. Los procesadores estos días cuentan con múltiples núcleos, que son múltiples unidades de ejecución en el mismo procesador. Para hacer un mejor uso de los procesadores multi-núcleo, necesitamos ejecutar tareas en threads (hilos) en paralelo. En otras palabras, necesitamos hacer nuestros programas multi-hilo o concurrentes. En esencia, la concurrencia está ganando importancia con un uso más generalizado en estos días. Afortunadamente, Java tiene soporte integrado para concurrencia. En este capítulo aprenderás los conceptos básicos de programación multi-threading y hacer aplicaciones y programas concurrentes. Temas más avanzados acerca de la concurrencia serán cubiertos en el siguiente capítulo.

# Introducción a la programación concurrente

En una aplicación típica como un procesador de textos, muchas tareas necesitan ser ejecutadas al mismo tiempo, digamos responder al usuario, corregir ortografía, llevar a cabo tareas de formato y tareas asociadas en segundo plano, etc. Ejecutar múltiples tareas al mismo tiempo es esperado para una aplicación interactiva. Esto es posible de hacer con tareas secuenciales; sin embargo, la experiencia del usuario no será la misma. Por ejemplo, muchos procesadores de texto tienen la característica de auto-guardar, si esta se ejecuta cada 60 segundos, y si durante este tiempo la aplicación no responde a las acciones del usuario, el usuario sentirá que la aplicación está fallando. En lugar de ejecutar esta tarea de manera secuencial, si la característica de auto-salvar es ejecutada automáticamente en el segundo plano sin afectar la tarea principal de responder al usuario, la experiencia de usuario será mucho mejor. Un escenario similar es llevar a cabo las tareas del diccionario en el segundo plano como cuando el usuario escribe alguna palabra y entonces se sugieren alternativas para el error ortográfico cometido. Ejecutar actividades como estas en paralelo mejora capacidad de respuesta y la experiencia del usuario. Estas actividades paralelas pueden ser implementadas como Threads: ejecutar múltiples threads en paralelo al mismo tiempo es llamado multi-hilo o concurrente.

La programación multi-hilo es muy útil para aplicaciones de internet también. Por ejemplo, un applet que muestra la actualización del mercado de valores intenta obtener la última información y mostrar las actualizaciones en textos y gráficos. Puedes escribir directamente un loop infinito que se quedará esperando por las actualizaciones y después recargar los gráficos y el texto. Esta aproximación desperdicia ciclos de procesador; también el usuario sentirá que el applet se cuelga cuando la actualización ocurre. Una mejor aproximación es hacer a un thread esperar por las actualizaciones y cuando ocurran informar al thread principal. Entonces threads separados pueden refrescar las gráficas y el texto del applet.

Las clases Object y Thread y la interface Runnable proveen el soporte necesario para concurrencia en Java. La clase Object tiene métodos como wati(), notify()/notifyAll() etc., que son muy útiles para concurrencia. Ya que cada clase en java deriva de la clase Object, todos los objetos tienen capacidades básicas de multi-threading. Por ejemplo, puedes adquirir un aspecto de cualquier objeto en java (se explicará más tarde). Sin embargo, para crear un thread, este soporte básico de la clase Object no es muy útil. Para esto, una clase debe extender a la clase Thread o implementar la interface Runnable. Ambas están en el paquete java.lang, por lo que no tienes que importar estas clases explícitamente.

## Métodos importantes relacionados con Threading

La siguiente tabla lista importantes métodos en la clase Thread, los cuales usarás en este capítulo.

|  |  |  |
| --- | --- | --- |
| Método | Tipo | Descripción corta |
| Thread currentThread() | static | Retorna la referencia al thread actual |
| String getName() | instance | Retorna el nombre del thread actual |
| int getPriority() | instance | Retorna el valor de la prioridad del thread actual |
| void join()  void join(long)  void join(long, int) | instance and overloaded | El thread actual invoca join en otro thread espera a que el otro thread muera. Puedes opcionalmente dar un tiempo de término en milisegundos (long) o bien en nanosegundos (long, int) |
| run() | instance | Ya que inicias un thread mediante el método start() el método run() será llamado cuando el thread esté listo para ejecutarse |
| void setName(String) | instance | Cambia el nombre del thread al nombre dado en el argumento |
| void setPriority(int) | instance | Ajusta la prioridad del thread al valor del argumento dado |
| void sleep(long)  void sleep(long, int) | static overloaded | Hace que el thread actual se duerma por los milisegundos dados (long) o milisegundos y nanosegundos(long, int) |
| void start() | instance | Comienza el thread; JVM llama al método run() en el thread |
| String toString() | instance | Retorna la representación String del thread; esta string contiene el nombre del thread, la prioridad y su grupo. |

En este capítulo también usarás algunos métodos para threads en la clase Object como sigue:

|  |  |  |
| --- | --- | --- |
| Método | Tipo | Descripción corta |
| void wait()  void wait(long)  void wait(long, int) | instance overload | El thread actual debe adquirir un bloqueo en este objeto antes de llamar cualquiera de estos métodos.  Si wait() es llamado, el thread espera infinitamente hasta que otro thread notifica (llamando a notify()/notifyAll()) para este bloqueo.  El método wait(long) toma los milisegundos como argumento. El thread espera hasta que sea notificado o hasta que el tiempo pase.  El método wait(long, int) es similar que wait(long) y adicionalmente toma nanosegundos como argumento. |
| void notify() | instance | El thread actual debe adquirir un bloqueo en este objeto antes de llamar a este método. La JVM escoge un único thread que está esperando en el bloqueo y lo despierta. |
| void notifyAll() | instance | El thread actual debe adquirir un bloqueo antes de llamar a este método. La JVM despierta todos los threads esperando en un bloqueo |

## Crear Threads

Un thread en Java puede ser creado de dos formas: extendiendo a la clase Thread o también implementando a la interface Runnable. Ambos tienen un método llamado run(). La JVM llamará a este método cuando el thread inicia a ejecutarse. Puedes pensar en el método run() como el punto de inicio para la ejecución de un thread, como el método main(), que es el punto de partida de la ejecución de un programa. Primero verás dos ejemplos de crear threads extendiendo a Thread e implementando Runnable, antes de aprender las diferencias entre ellos.

### Extendiendo la clase Thread

Primero considerarás cómo extender la clase Thread. Necesitas sobre-escribir el método run() cuando quieres extender la clase Thread. Si tu no sobre-escribes el método run(), el método por default de la clase Thread será llamado y el cual no hace nada.

Para sobre-escribir el método run() necesitas declararlo público; este no tiene argumentos y es de tipo void. En otras palabras lo declaras public void run().

Un thread puede ser creado invocando al método start() sobre el objeto de la clase Thread o sus clases derivadas. Cuando la JVM programa el thread, lo moverá al estado runnalble y después ejecutará el método run(). Cuando el método run() completa su ejecución y retorna, el thread se habrá terminado. El siguiente programa es el primer ejemplo de multi-threading.

[MyThread1.java](src/java/oca2/cap13/MyThread1.java)

En este ejemplo, la clase MyThread1 extiende a la clase Thread. Sobre-escribes el método run() en esta clase. Este método será llamado cuando el thread se ejecuta. En el método main() creas un thread nuevo y lo inicias usando el método start(). Una nota importante: no invocas al método run() directamente. En su lugar, invocas al thread usando el método start(); el método run() es invocado automáticamente por la JVM. Analizaremos este tema más tarde.

Para imprimir el nombre del thread, puedes usar el método de instancia getName(), que retorna un String. Ya que el método main() es estático, no tienes acceso a la referencia this. Por lo tanto obtienes el nombre del thread usando el método estático currentThread() en la clase Thread (el cual retorna un objeto Thread). Ahora puedes llamar al método getName() en este objeto Thread devuelto. Como verás más tarde, el método main() es también ejecutado como un thread! Sin embargo, dentro del método run() puedes llamar directamente al método getName(): MyThread1 extends Thread, entonces todos los miembros de la clase base están disponibles en MyThread1.

El programa imprime mensajes de los dos: el método main() y la instancia myThread (que creaste en el main). El nombre del thread impreso es *Thread-0*. Verás la convención de nombres por default después.

La figura que sigue muestra cómo se ejecuta el programa e imprime la salida. Nota que el thread main y e myThread thread se ejecutan al mismo tiempo (concurrentemente), como se ve en el diagrama. Si intentas este programa un par de veces, obtendrás la salida mostrada anteriormente, o el orden de las dos sentencias estará invertido (dependiendo qué thread se programa primero para ejecutar esta sentencia). Estudiarás este comportamiento no-determinístico después en este capítulo.



### Implementando la interface Runnable

La clase Thread por sí misma implementa la interface Runnable. En lugar de extender a la clase Thread puedes implementar la interface Runnable. La interface Runnable declara un solo método: run().

Cuando implementas la interface Runnable, necesitas definir el método run(). Recuerda que Runnable no declara el método start(). Entonces, ¿Cómo puedes crear un Thread con Runnable?.

La clase Thread tiene un constructor sobrecargado que toma como argumento a un objeto Runnable.

Thread(Runnable target){//...}

Puedes usar este constructor para crear un Thread de una clase que implemente la interface Runnable.

Primero vamos a cambiar el programa anterior implementando la interface Runnable. Si tu cambias “class MyThread1 extends Thread” por “class MyThread1 implements Runnable” y compilas el código obtendrás dos errores de compilación: el método getName() no existe para la interface Runnable, y la asignación Thread myThread = new MyThread1() ya no es válida, ya que el tipo de la referencia myThread es la clase Thread y el tipo de MyThread1 es MyThread1. Tampoco está disponible el método start() ya que este está disponible para la clase Thread.

El siguiente programa contiene la versión mejorada del programa implementando la interface Runnable después de arreglar los problemas que acabamos de analizar.

[MyThread2.java](src/java/oca2/cap13/MyThread2.java)

Estás implementando el método run() como en el programa anterior. Sin embargo, para obtener el nombre del thread debes seguir la ruta redundante y obtener el nombre del thread con Thread.currentThread().getName(), como lo hiciste para el método main(). Similarmente, en el método main(), para crear un thread debes pasar el objeto de la clase al constructor de la clase Thread. Fue más fácil crear MyThread1 y llamar el método start().

### ¿Debes extender a Thread o implementar Runnable?

Puedes hacerlo de cualquiera de las dos formas para crear un Thread. Entonces ¿Cuál forma es la que debes escoger?

La clase Thread tiene una implementación por default del método run(), por lo que si no provees una definición de este método, el compilador no se quejará. Sin embargo, la implementación por default en la clase Thread NO HACE NADA, entoces si quieres que tu thread haga trabajo útil, necesitas definir el método run(). La interface Runnable declara el método run(), por lo tanto debes definirlo en tu clase si implementas la interface Runnable. Por lo tanto no importa si implementas la interface Runnable o extiendes a la clase Thread. Debes de definir el método run() por razones prácticas. En resumen, no hay mayor diferencia entre las dos formas. ¿Qué pasa con la relación de herencia?

Ya que java soporta solo herencia única, si extiendes de Thread, no puedes extender de otra clase. Ya que la herencia es una relación is-a (es-un), raramente necesitarás que la clase con una relación is-a con la clase Thread. Por lo tanto, en un diseño OO puro se argumenta que no debes extender a la clase Thread. Por otro lado, si implementas a Runnable, puedes extender a otra clase. Por lo que muchos expertos en Java sugieren que esta es la mejor opción: Implementar Runnable, al menos que haya razones poderosas para extender a Thread.

Sin embargo, extender a la clase Thread es más conveniente en muchos casos. En el ejemplo que viste para obtener el nombre del Thread, necesitaste usar Thread.currentThread().getName() cuando implementaste Runnable mientras que cuando extendiste a Thread solo invocaste el método miembro getName(). En este caso, extender a Thread es ligeramente más conveniente.

Ambas técnicas son útiles y muy equivalentes para resolver problemas. Entonces toma una perspectiva práctica: usa cualquiera de las dos como necesites para un problema específico que intentes resolver. Para el examen, debes de saber cómo crear threads de cualquiera de las dos formas, así como la diferencia entre estas dos aproximaciones.

# Los métodos start() y run()

Sobre-escribes el método run() pero invocas al método start(). ¿Por qué no puedes llamar al método run() directamente? Si cambias el programa previo por myThread.run(), ¿Qué pasa? El siguiente programa muestra esta modificación.

[MyThread3.java](src/java/oca2/cap13/MyThread3.java)

Ahora la salida es diferente. Si llamas al método run () directamente, simplemente se ejecuta como parte del thread que lo llama; significa que no se ejecuta como otro Thread: este no es programado y llamado por la JVM. Por esta razón, el método getName() en el método run() retorna el nombre del thread como “main” en lugar de “Thread-0”. Cuando llamas al método start(), el thread es programado y el método run() es invocado por la JVM cuando es tiempo de ejecutar el thread.

NOTA: Nunca llames al método run() directamente para invocar un thread. Usa el método start() y deja que la JVM invoque implícitamente al método run(). Llamando al método run() directamente, es un error y un bug común.

# Nombre del Thread, prioridad y grupo

Necesitas entender tres aspectos importantes asociados con cada Thread: su nombre, prioridad y el grupo de threads al que pertenece.

Cada thread tiene un nombre, que puedes usar para identificar al thread. Si no provees un nombre explícitamente, el thread obtendrá un nombre por default. La prioridad puede variar de 1 a 10 (de baja a alta). La prioridad de un thread normal es 5 por default y pues cambiarla proveyendo un valor explícito. Cada thread es parte de un grupo de threads. Es una característica raramente usada, por lo que no la cubriremos en este libro. El método toString() de un thread imprime estos tres detalles. Vamos a ver el siguiente programa para ver estos detalles.

[SimpleThread.java](src/java/oca2/cap13/SimpleThread.java)

La salida: Thread[Thread-0,5,main]

Thread es el nombre de la clase. Entre [] está el nombre del thread, su prioridad y el grupo al que pertenece. No diste ningún nombre al thread, por lo que obtiene el nombre por default Thread-0 (si creas más threads, se les asignarán nombres como: Thread-1, Thread-2...). La prioridad por default es 5. Creas el Thread en el método main(), entonces el grupo por default es “main”.

Ahora vamos a intentar cambiar el nombre y la prioridad usando los métodos setName() y setPriority():

[SimpleThread2.java](src/java/oca2/cap13/SimpleThread2.java)

La salida: Thread[HiloSimple,9,main]

El thread tiene el nombre y la prioridad que le diste. Puedes cambiar el nombre de los threads como quieras y no cambiará la ejecución del programa. Sin embargo, necesitas ser cuidadoso cuando cambias la prioridad, ya que esta puede afectar la programación de threads. Puedes acceder programáticamente a la prioridad minima, normal y máxima de los threads, usando los miembros estáticos MIN\_PRIORITY, NORM\_PRIORITY y MAX\_PRIORITY, como se observa en el siguiente código.

[ThreadPriorities.java](src/java/oca2/cap13/ThreadPriorities.java)

# Usar el método Thread.sleep()

Digamos que quieres implementar una cuenta regresiva para una bomba de tiempo que cuente de diez a cero en pasos de un segundo. Después de llegar a cero debe imprimir “Boom!!!!!” Puedes implementar esta funcionalidad creando un thread que ejecute la cuenta regresiva. Con el objetivo de pausar cada segundo, puedes llamar al método Thread.sleep(). Observa el siguiente código.

[TimeBomb.java](src/java/oca2/cap13/TimeBomb.java)

El programa no funciona correctamente. El mensaje “Boom!!!” será impreso incluso antes de que el count down empiece. Antes discutimos la causa de este extraño comportamiento, vamos a ver lo básico del método sleep().

Usas el método estático sleep() disponible en la clase Thread para dormir al thread actual (o pausarlo) por un cierto periodo de tiempo. Hay dos métodos sobrecargados sleep() en la clase Thread:

1. void sleep(long)
2. void sleep(long, int)

La primera versión del método toma milisegundos como argumento. La segunda versión, en adición a los milisegundos, toma nanosegundos como segundo argumento.

El método sleep() lanza a InterruptedException. Ya que InterruptedException es una excepción checada (Extiende a la clase Exception), necesitas proveer un bloque try-catch cubriendo el método sleep() o relanzar la excepción en la cláusula throws del método. Sin embargo, si declaras void run() throws InterruptedException, no estás sobre-escribiendo el método run() ya que la especificación de la excepción es diferente (**el método run() no declara ninguna checked exception**)**.** Por lo que deberás proveer en su lugar un bloque try-catch para manejar esta excepción dentro del método run(). ¿Qué puedes hacer para manejar una InterruptedException?

Primero, necesitas entender que significa InterruptedException y cuando es lanzada. Un thread puede interrumpir a otro thread, digamos, para solicitarle que pare de trabajar. En este caso, el thread recibe la interrupción, si está en sleep() o wait() (que revisaremos después), el resultado es lanzar una InterruptedException. El thread recibe la interrupción puede ignorar la interrupción y continuar ejecutándose (que no es una buena idea, pero es posible de hacer), o puede detener la ejecución. Puedes no interrumpir otros threads en programas multi-thread que cubriremos en este libro. Entonces vamos a asumir que tus threads no serán interrumpidos, y tú ignoras la excepción y solicitas al thread que se siga ejecutando. En otras palabras, puedes conscientemente ignorar la excepción (después de llamar al método printStackTrace()); sin embargo, en programas reales, debes de manejar esta excepción si usas la característica de interrumpir threads.

Regresemos a la salida del programa, el mensaje “Boom!!!” es impreso justo antes de iniciar la cuenta regresiva y no después de alcanzar el cero. ¿Por qué pasa esto?

El thread principal inicia la ejecución del timer, entonces este ejecuta la siguiente sentencia, que imprime “Boom!!!” a la consola.

Pero recuerda que quieres que el método main() espere hasta que el timer sea completado. ¿Cómo puedes hacer esto? Para esto denes de aprender cómo usar el método join() provisto en la clase Thread.

# Usar el método Join de la clase Thread

La clase Thread tiene el método de instancia join() para esperar que el thread muera. En el programa anterior “Time bomb”, quieres que el thread main() espere que el thread timer complete su ejecución. Puedes usar el método join() de la clase Thread para lograr esto. Aquí está una versión mejorada del programa, con cambios sólo en el método main().

[TimeBomb.java](file:///C:\Users\toshiba2\Documents\javaEx\1z0-804\src\java\oca2\cap13\TimeBomb.java)

La clase Thread tiene tres versiones sobrecargadas del método join():

1. join();
2. join(long);
3. join(long,int);

Si el thread actual invoca al método join() en una instancia de otro thread, entonces el thread actual espera indefinidamente hasta que el otro thread muera. Los otros dos métodos sobrecargados toman un tiempo límite como argumento; el actual thread esperará al otro thread, solamente mientras el tiempo límite expira. El thread actual continuará su ejecución en caso de que el otro thread no complete antes de este tiempo fuera. El segundo método toma un periodo de tiempo fuera con milisegundos y nanosegundos.

El método join() también lanza una InterruptedException; ignoras esta excepción por las mismas razones discutidas en el método sleep() visto anteriormente en este capítulo.

## Ejecución asíncrona

En el programa previo viste que el thread principal y el thread que creaste se ejecutan de manera independiente. En otras palabras, los threads corren de manera asincrónica. Los threads no corren secuencialmente (como llamadas a funciones), entonces el orden de la ejecución de los threads no es predecible. En otras palabras, el comportamiento de los threads es no-determinístico por naturaleza. Para entender esto considera el siguiente programa.

[AsyncThread.java](src/java/oca2/cap13/AsyncThread.java)

Como puedes ver, la salida de estos dos threads es ligeramente diferente ¿Por qué? Los threads Thread-0 y Thread-1 son ejecutados independientemente. La salida no es fija y el orden de ejecución de las iteraciones en los threads no es predecible. Un programador no puede determinar el orden de ejecución de los threads. La plataforma subyacente donde corre el programa puede usar cualquiera delos múltiples procesadores o usar un intervalo de tiempo asignado a un solo CPU para el thread. Esto no será controlado por la JVM o el programador. Este es uno de los conceptos más importantes que entender en la programación multi-threading. TU NO PUEDES PREDECIR NI CONTROLAR EL ORDEN DE EJECUCIÓN DE LOS THREADS.

NOTA: Ya que el comportamiento de los programas multi-thread es no determinístico, debes de ser cuidadoso en escribirlo. No puedes esperar una salida predeterminada basada en el orden de ejecución de los threads.

# Los estados de un thread

Un Thread tiene varios estados durante su ciclo de vida. Es importante entender los diferentes estados de un thread y aprender a escribir código robusto basado en este entendimiento. Tú verás tres estados: new, runnable y terminated, que son aplicables a casi todos los threads que crearás en esta sección. Discutiremos más estados después.

Un programa puede tener acceso al estado usando el enum Thread.State. La clase Thread tiene el método de instancia getState(), que retorna el estado actual del thread; observa el siguiente código que es un ejemplo de esto.

[BasicThreadStates.java](src/java/oca2/cap13/BasicThreadStates.java)

Justo después de la creación del thread y antes de llamar al método start() de este thread, este está en estado “new”. Después de llamar al método start(), el thread está listo para correr o corriendo (que no puedes determinar), entonces está en estado “runnable”. En el método main() llamas al método t.join(). El método main() espera a que se termine el thread. Entonces, ya que se ejecutó el método join() satisfactoriamente significa que el thread se murió o ha terminado. Entonces el thread está en estado “terminated”.

Un aviso: ten cuidado cuando accedes al estado de los threads usando el método getState(). ¿Por qué? En el tiempo que adquieres la información del estado del thread y la imprimes, esta puede haber cambiado. Sabemos que esto último puede ser confuso. Para entender el problema con obtener el estado del thread usando el método getState(), considera el ejemplo previo. En uno de los ejemplos, las dos últimas impresiones imprimen TERMINATED, aunque esperas que la penúltima imprima RUNNABLE ¿Por qué? En este caso, el thread está muerto antes de que tengas chance de checarlo e imprimir el estatus (nota que no implementas el método run() en la clase BasicThreadStates, entonces la implementación por default del método hace nada y termina rápido).

Cada thread en java atraviesa por estos tres estados, como se observa en la siguiente figura. Entre estos, el estado RUNNABLE de hecho consiste en dos estados separados en el nivel de sistema operativo. Esto lo discutiremos ahora.



## Dos estados en el estado RUNNABLE

Ya que un thread hace la transición de estado de NEW a RUNNABLE, puedes pensar que el thread tiene dos estados en el nivel del sistema operativo: el estado READY y el estado RUNNING. Un thread está en estado READY cuando está esperando a que el sistema operativo le asigne un tiempo en el procesador para correr. Cuando el procesador está corriendo el thread está en estado RUNNING. Puede haber muchos threads esperando un tiempo en el procesador. El thread actual puede terminar tomando una gran cantidad de tiempo de procesador y finalmente renunciar al CPU voluntariamente. En este caso, el thread regresa al estado READY. Estos dos estados se muestran en el siguiente diagrama.



# Problemas de acceso concurrentes

La programación concurrente en threads está plagada de dificultades y problemas. Discutiremos dos principales problemas de acceso concurrente en esta sección: data-races y deadlocks.

## Data Races

Los threads comparten memoria y pueden modificar datos concurrentemente. Ya que esta modificación puede estar ocurriendo al mismo tiempo sin seguridad, esto puede conllevar a resultados no intuitivos.

Cuando dos o más threads están intentando tener acceso a una variable y uno quiere modificarla, tienes un problema llamado data-race (también llamado race-condition o race-hazard) El siguiente programa es un ejemplo de esto.

[DataRace.java](src/java/oca2/cap13/DataRace.java)

En este programa hay una clase Counter que tiene una variable estática count. En el método run() de la clase UseCounter, incrementas el contador tres veces llamando al método increment(). Creas tres threads en el método main() en la clase DataRace y los inicias con start(). Esperas que se imprima secuencialmente del 1 al 9 como los threads corren e incrementan el contador. Sin embargo, cuando corres este programa, imprime nueve valores enteros, pero la salida parece basura. Cada vez que ejecutamos el programa hay una salida distinta. Entonces ¿Cuál es el problema?

La expresión Counter.count++ es una operación de escritura y la siguiente sentencia print() lee la operación de Counter.count. Cuando los tres threads se ejecutan, cada uno de ellos hace una copia local del valor Counter.count y cuando se actualiza el couter con Counter.count++, estos no reflejan inmediatamente este valor en la memoria principal. En la siguiente operación de lectura de Counter.count, el valor local de Counter.count es impreso.



Este diagrama es un ejemplo de data-race.

Por lo tanto este programa tiene problemas de data-race. Para evitar este problema, necesitas asegurar que un único thread escriba y lea de manera conjunta (atómica). La sección del código que es de acceso común y modificado por más de un thread es conocida como *sección crítica*. Para evitar este problema necesitas asegurar que esta sección crítica por un único thread a la vez.

¿Cómo puedo hacer esto? Adquiriendo un bloqueo en el objeto. Solo un único thread puede adquirir un bloqueo en el objeto a la vez, y solo este thread puede ejecutar el bloque de código (digamos la sección crítica) protegido por el bloqueo. Mientras esto pasa, los otros threads tienen que esperar. Internamente, esto es implementado con monitores y el proceso es llamado locking y unlocking (digamos thread synchronization). Vamos a discutir esto con más detalle.

## Sincronización de Threads

Java cuenta con la palabra clave *synchronized* que ayuda a la sincronización de threads. Puedes usarla en dos formas: bloques sincronizados o métodos sincronizados.

### Bloques sincronizados

En los bloques sincronizados usas la palabra clave *synchronized* para una variable de referencia y es seguida por un bloque de código. El thread que adquiera el bloqueo a la variable sincronizada entrará al bloque; cuando la ejecución del bloque sea completada, el thread liberará el bloqueo. Por ejemplo, puedes adquirir un bloqueo en la referencia “this” si el bloque de código está dentro de un método no-estático.

synchronized(this) { // código custodiado por el bloqueo excluyente}

¿Qué pasa si es lanzada una excepción en el bloque sincronizado? ¿El bloqueo será liberado? Sí, no importa si el bloqueo es ejecutado completo o lanza una excepción, el bloqueo será liberado automáticamente por la JVM.

Con bloques sincronizados, puedes adquirir bloqueos solamente para variables de referencia. Si usas tipos primitivos obtendrás un error de compilación.

Aquí está una versión mejorada del programa anterior que ejecuta acceso sincronizado a Counter.count y hace ambas operaciones (escribir y leer) en esta sección crítica. Para esto necesitas hacer un cambio en el método increment():

[DataRace2.java](src/java/oca2/cap13/DataRace2.java)

Ahora el programa imprime la salida correctamente.

En el método increment() adquieres un bloqueo en la referencia “this” antes de leer y escribir a Counter.count. Entonces no es posible que más de un thread ejecute estas sentencias al mismo tiempo. Ya que un único thread puede adquirir el bloqueo y ejecutar el bloque de código sección crítica, el contador es incrementado solo por un thread a la vez; como resultado, el programa imprime los valores del 1 al 9 correctamente (sin problemas de data-race).

### Métodos sincronizados

Un método entero puede ser declarado como sincronizado. En este caso, cuando el método es declarado como sincronizado, un bloqueo es obtenido en el objeto donde el método es llamado, y es liberado cuando el método retorna al llamador:

public synchronized void assign(int i) { val = i;}

Ahora el método assign() es un método sincronizado. Si tu llamas al método assign(), este adquirirá un bloqueo en la referencia this implícitamente y después ejecutará la sentencia val = i;. ¿Qué pasa si algún otro thread intenta adquirir el bloqueo al mismo tiempo? Justo como los bloques sincronizados, si el thread no puede obtener el bloqueo, este será bloqueado y el thread esperará hasta que el bloqueo esté disponible.

Un método sincronizado es equivalente a un bloque sincronizado si tú encierras el cuerpo completo del método con un bloque synchronized(this). Por lo tanto, el método assign() equivalente usando un bloque sincronizado es:

public void assign(int i) { synchronized(this) {val = i;}}

Puedes declarar métodos estáticos sincronizados. Sin embargo, ¿Cuál es la variable de referencia de donde el bloqueo es obtenido? Recuerda que los métodos estáticos no tienen implícita la referencia this. Los métodos estáticos sincronizados adquieren los bloqueos en la clase de objeto. Cada clase es asociada con un objeto de tipo Class, y puedes tener acceso a ella usado la sintaxis ClassName.class. Por ejemplo:

[SomeClass.java](src/java/oca2/cap13/SomeClass.java)

No puedes declarar constructores sincronizados; esto resultará en un error de compilación. ¿Por qué? La JVM asegura que solo un thread pueda invocar al constructor en un punto en el tiempo. Por lo tanto no es necesario declarar un constructor sincronizado. Sin embargo, si quieres, puedes utilizar bloques sincronizados dentro de los constructores.

Vamos a regresar al ejemplo de la clase Counter. El método increment() puede ser re-escrito como un método sincronizado también.

[DataRace3.java](src/java/oca2/cap13/DataRace3.java)

Los programadores principiantes olvidan que un bloque sincronizado obtiene un bloqueo para un bloque de código. De hecho, el bloqueo es obtenido para el objeto, mas no para una pieza de código. El bloqueo obtenido se lleva a cabo mientras todas las sentencias en el bloque se completen o, en su defecto, ocurra una excepción.

### Bloques sincronizados vs Métodos sincronizados

Como puedes ver de la discusión previa de bloques y métodos sincronizados, puedes usar cualquiera de los dos para resolver problemas de data-race. Entonces ¿Cuál escoger? Como en otras características del lenguaje, necesitas escoger dependiendo de las necesidades de una situación particular. Aquí hay algunos factores a considerar.

Si quieres adquirir un bloqueo en un objeto para solo un pequeño pedazo de código y no el método entero, entonces un bloque sincronizado es suficiente; usando métodos sincronizados es una exageración. En general, es mejor obtener bloqueos para segmentos pequeños de código en lugar de sincronizar métodos completos innecesariamente, entonces los bloques sincronizados son útiles aquí. En bloques sincronizados puedes proveer la referencia al objeto de manera explícita, al cual quieres que adquiera el bloqueo. Sin embargo, en el caso de los métodos sincronizados, no provees una referencia explícita para adquirir el bloqueo. Un método sincronizado adquiere un bloqueo implícito a la referencia this (para métodos de instancia) y al objeto class (para métodos estáticos).

Por otro lado, si quieres adquirir un bloqueo para el cuerpo completo de un método pequeño, entonces usar un método sincronizado es más elegante y conveniente que un bloque sincronizado; con bloques sincronizados necesitas leer la documentación u observar dentro del código para entender que se está ejecutando alguna sincronización.

## Deadlocks

Obtener y usar bloqueos es confuso y puede conllevar un montón de problemas. Una de las dificultades (y común) es conocida como Deadlock. Hay otros problemas como Livelocks y lock starvation, que discutiremos en la siguiente sección.

Un Deadlock se alcanza cuando threads bloqueados resultan en una situación donde no proceden y esperan indefinidamente por otros para terminar. Digamos, un thread adquiere un bloqueo para el recurso 1 r1 y espera para adquirir otro para el recurso 2 r2. al mismo tiempo, digamos que hay otro thread que ya ha adquirido r2 y está esperando para obtener un bloqueo en r1. Ninguno de los threads pueden proceder mientras el otro no libere el bloqueo, que nunca pasa. Entonces estos están atrapados en un Deadlock.

El siguiente código muestra cómo se puede alcanzar esta situación.

[DeadLock.java](src/java/oca2/cap13/DeadLock.java)

En este ejemplo hay dos clases, Balls y Runs con miembros estáticos balls y runs. La clase Counter tiene dos métodos, IncrementBallAfterRun() e IncrementRunAfterBall(). Estos adquieren bloqueos en las clases Balls.class() y Runs.class() en orden opuesto. El método run() llama a estos dos métodos de manera consecutiva. El método main() en la clase DeadLock crea dos threads y los inicia.

Cuando los threads t1 y t2 se ejecutan, estos invocan los dos métodos. En los métodos, los bloqueos son obtenidos en orden opuesto. Entonces puede pasar que t1 adquiera un bloqueo en Runs.class() y después espere para adquirir el bloqueo en Balls.class(). Mientras tanto, t2 puede adquirir el bloqueo en Balls.class() y ahora esperar para adquirir el bloqueo en Runs.class(). Por lo tanto, este programa puede resultar en un Deadlock. Observa el siguiente diagrama.



No se puede asegurar que este programa caiga en un deadlock cada vez que se ejecute. ¿Por qué? Tú nunca sabes la secuencia en que los threads se ejecutan y el orden en que los bloqueos son adquiridos y liberados. Por esta razón, estos problemas son no-determinísticos, y estos problemas no pueden ser reproducidos conscientemente.

Hay diferentes estrategias para tratar con deadlocks, como deadlock prevention, avoidance o detection. Para los propósitos del examen esto es lo que debes saber acerca de deadlocks:

* Pueden alcanzarse en contextos de múltiples bloqueos.
* Si múltiples bloqueos son adquiridos en el mismo orden, entonces el deadlock no ocurrirá, si los adquieres en diferente orden, los deadlocks pueden ocurrir.
* Deadlocks (como otros problemas multi-threading) son no determinísticos; no puedes reproducir deadlocks de manera consciente.

NOTA: Evita adquirir múltiples bloqueos. Si quieres adquirir múltiples bloqueos, asegúrate de que sean adquiridos en el mismo orden siempre, para evitar los deadlocks.

## Otros problemas con Threading

Por mucho, hemos discutido data-races y deadlocks con ejemplos. Ahora discutiremos de manera breve dos problemas más: livelocks y lock starvation.

### Livelocks

Para ayudar a entender livelocks, considera una analogía. Asume que aquí hay dos carros robóticos que están programados para automáticamente manejar en el camino. Esta es la situación cuando los dos carros alcanzan las dos entradas opuestas a un delgado puente. El puede es tan delgado que solo un carro puede pasar a la vez. Los carros están programados para que el otro carro pase primero por el puente. Cuando ambos carros intentar entrar en el puente al mismo tiempo, la siguiente situación pasa: cada carro inicia su entrada al puente, se da cuenta que otro carro está intentando hacer lo mismo, y se regresa. Nota que estos carros se siguen moviendo hacia adelante y hacia atrás, por lo tanto parece que están haciendo un montón de trabajo, pero no hay progreso de ninguno de los coches. Esta situación es llamada livelock.

Considera dos threads t1 y t2. Asume que t1 hace un cambio y t2 lo deshace. Cuando ambos threads trabajan, puede parecer que se está haciendo un montón de trabajo pero no hay progreso. Esto es llamado livelock en threads.

La similitud entre livelocks y deadlocks es que el proceso se cuelga y el programa nunca termina. Sin embargo, en un deadlock, los threads se quedan en el mismo estado esperando por el otro thread para liberar un recurso compartido; en un livelock los threads siguen ejecutando una tarea y tienen cambios continuos en los estados de los threads, pero la aplicación no muestra progreso.

### Lock Starvation (inanición)

Considera la situación en donde numerosos threads tienen diferentes prioridades asignadas (en el rango de 1 a 10, que son los rangos permitidos en java). Cuando un bloqueo mutuo es disponible el programador de threads le dará prioridad a los threads con la mayor prioridad, sobre threads con menor prioridad. Si hay muchos threads de mayor prioridad que quieren obtener el bloqueo y lo mantienen por periodos largos de tiempo, ¿Cuándo los threads de menor prioridad podrán obtener el bloqueo? En otras palabras, en una situación donde threads de baja prioridad se mueren de hambre por un largo tiempo intentando obtener el bloqueo es conocido como lock-starvation.

Hay muchas técnicas disponibles para detectar o evitar problemas con threading como livelocks o lock-starvation, pero están fuera del examen. Desde la perspectiva del examen, se espera que conozcas los diferentes problemas de threading y que ya han sido cubiertos en este capítulo.

## El mecanismo wait/notify

En programas multi-thread a menudo, hay la necesidad de un thread para comunicarse con otro thread. El mecanismo wait/notify es útil cuando threads deben comunicarse para proveer cierta funcionalidad.

Vamos a tomar el ejemplo de la tienda de café. Un mesero está usando la máquina de café y enviando café a los clientes. La máquina de café en esta tienda es una máquina antigua: hace solo una taza de café a la vez, y toma de cinco a diez minutos hacerla. El mesero no tiene que estar de ocioso mientras espera a la máquina de café; el puede repartir a los clientes el café hecho con anterioridad. Este ejemplo es un poco artificial, aunque: asume que la máquina de café se mantiene haciendo el café y el mesero se mantiene sirviendo el café.

El método wait() permite que el thread que llama esperar al objeto que espera (donde el método es llamando). En otras palabras, si quieres hacer que un thread espere por otro thread, puedes pedirle que espere al objeto que usa el método wait(). El thread se mantiene en el estado wait mientras otro thread llama al método notify o notifyAll() en el objeto que espera. Para entender este mecanismo, estás por simular esta tienda de café en un programa. Puedes implementar la máquina de café como un thread y el mesero como otro thread en dos diferentes clases. La máquina de café puede notificar al mesero para tomar el café, y puede esperar mientras el mesero toma el café de la bandeja. Similarmente, el mesero puede tomar el café si está disponible y notificar a la máquina de café que haga otra copa.

Explicar este mecanismo en un ejemplo envuelve un poco de código. Pero es un ejemplo interesante para ilustrar este concepto. El siguiente código contiene la clase CoffeMachine.

[CoffeeMachine.java](src/java/oca2/cap13/CoffeeMachine.java)

El objeto CoffeeMachine está por correr como un thread, por lo que extiende a la clase Thread e implementa el método run(). El método run() corre por siempre y mantiene llamando al método makeCoffee(). Para cada iteración, este llama al método sleep() por diez segundos para simular el tiempo que le toma a la máquina hacer un café.

La clase CoffeeMachine tiene tres miembros estáticos. El miembro coffeeMade tiene una descripción del café que está haciendo. El miembro lock está para la sincronización entre los objetos threads CoffeeMachine y Waiter. El numOfCoffees es usado internamente por el método makeCoffee(), para obtener la descripción del café hecho.

El método makeCoffee() hace la mayor parte del trabajo. La primera cosa que hace es adquirir un bloqueo a CoffeeMachine.lock con la palabra synchronized. Dentro del bloque, esta checa se coffeeMade es null o no. La primera vez que el thread CoffeeMachine llama al método makeCoffee(), coffeeMade será null. En los otros casos, es el thread Waiter el que hará a coffeeMade null y notificará (usando el método notifyAll()) al thread CoffeeMade. Si el thread Waiter no ha sido creado todavía, este ira al estado wait() e imprimirá el mensaje “Waiting for waiter notification to deliver the coffee”.

Ya que el Waiter notifica al thread CoffeeMachine, la máquina envía el siguiente café al Waiter e imprime el mensaje notificando al waiter para que recoja el café. Ahora vamos a ver la clase Waiter. Observa el siguiente código.

[Waiter.java](src/java/oca2/cap13/Waiter.java)

La clase Waiter también extiende a Thread ya que esta clase también correrá como un thread. Sobre-escribe el método run() que hace algo muy simple: se mantiene llamando al método getCoffee() para siempre.

La clase Waiter tiene el método getCoffee() donde todo el trabajo está hecho. La primera cosa que el método hace es intentar adquirir un bloqueo en CoffeeMachine.lock. Ya que obtiene este bloqueo, checa si coffeeMade es null. Si la variable es null, significa que el thread CoffeeMachine todavía está preparando el café. En este caso, el Waiter llama wait() e imprime el mensaje “Will get orders till coffee machine notifies me”. Cuando CoffeeMachine ha hecho el café, fijará la variable coffeeMade, y esta será no null entonces; este thread también notificará al Waiter usando el método notifyAll().

Ya que el thread Waiter es notificado, puede enviar el café al cliente; este imprime el mensaje “Delivering coffee”. Después de esto, Waiter limpia la variable coffeeMade() a null y notifica a CoffeeMachine para hacer otro café (“Notifying cofee machine to make another one”). El siguiente código muestra la clase CoffeeShop.

[CoffeeShop.java](src/java/oca2/cap13/CoffeeShop.java)

Entonces: ¿DEBES USAR NOTIFY O NOTIFYALL?

Tienes dos métodos notify() y notifyAll() para notificar (despertar un thread que está esperando en la clase Thread). Pero ¿Cuál usar?

Vamos a examinar la sutil diferencia entre estas dos llamadas. El método notify() despierta un thread que está esperando por el bloqueo (el primer thread que llama a wait() en ese bloqueo). El método notifyAll() despierta todos los threads esperando por el bloqueo; la JVM selecciona uno de los threads de la lista de threads esperando por el bloqueo y despierta este thread.

En el caso de que un único thread esté esperando por el bloqueo, no hay mucha diferencia entre notify() y notifyAll(), el thread exacto es despertado bajo el control de la JVM y tú no puedes programáticamente controlar despertar un thread específico.

A primera vista, parece buena idea solo llamar a notify() para despertar un thread; parece poco necesario despertar todos los threads. Sin embargo, el problema con notify() es que el thread despertado puede no ser el thread que debe despertarse (el thread puede estar esperando por otra condición, o la condición no está satisfecha todavía para ese thread, etc.) En este caso, el método nofity() puede perderse y no despertar a otro thread que potencialmente podría llevarnos a un deadlock (la notificación se pierde y todos los otros threads esperan para siempre a ser notificados).

Para evitar este problema, siempre es mejor llamar a notifyAll() cuando hay más de un thread esperando por un bloqueo (o más de una condición en el que está esperando está heha). El método notifyAll() despierta a todos los threads, entonces no es muy eficiente. Sin embargo esta pérdida de eficiencia es imperceptible en programas reales.

NOTA: Usando notify()/notifyAll() despertará solo a los threads esperando el bloqueo donde este es llamado; no despertará ningún otro thread. Si por error usas wait() en un bloqueo y notify()/notifyAll() en otro bloqueo, el thread que está esperando nunca será notificado y el programa se colgará (deadlock).

### Vamos a resolver el problema

Ya que el mecanismo wait/notify es importante de entender, vamos a tomar otro ejemplo e intentar entenderlo más rigurosamente.

**Problema**: Asume que necesitas implementar un juego de dados. Es un juego de dos jugadores (digamos que los jugadores son joe y jane) donde los jugadores lanzan el dado en sus turnos. Cuando un jugador lanza el dado el otro jugador espera. Ya que el jugador completa el lanzamiento informa al otro jugador para jugar; después de esto espera a que el otro jugador lance el dado. Necesitas implementar estos dos jugadores como dos threads trabajando juntos. El juego termina después de que ambos jugadores tiraron el dado 6 veces, para un total de 12 lanzamientos en el juego.

Ya que el problema dice “implementar estos dos jugadores como threads trabajando juntos”, tu solución es un programa multi-thread en donde cada jugador es implementado como un thread. El problema también establece que cuando un jugador tira el dado el otro espera. Por lo que deberás usar quizá el mecanismo wait/notify. El valor del dado resultará en un valor aleatorio, entonces puedes usar la clase Random para crear números aleatorios del 1 al 6.

Aquí está la solución. Primero vamos a revisar todo el programa y después verás la explicación de cómo funciona.

[DiceGame.java](src/java/oca2/cap13/DiceGame.java)

NOTA: si quieres un mecanismo para esperar a un evento en particular, wait/notify es la mejor opción. A veces los programadores solventan este problema usando una llamada a sleep() y de manera repetida checan si la condición ya está cumplida. Esta es una solución poco efectiva. Además, llamando a sleep no libera el bloqueo como wait(), por lo que llamar a sleep produce deadlocks. No uses sleep().

## Más estados de un Thread

Antes, en este capítulo discutimos tres estados básicos de un thread: new, runnable y terminated. En adición a estos estados, un thread también puede estar: blocked, waiting, time-waiting. La siguiente figura muestra cómo y dónde estos estados pásan.



El siguiente ejemplo es para ejemplificar de manera simple time\_waiting y blocked

[MoreThreadStates.java](src/java/oca2/cap13/MoreThreadStates.java)

Tienes la clase SleepyThread con el método run() que solo adquiere un bloqueo y se va a dormir. Creas dos threads en el método main().

Cuando t1 corre, adquiere el bloqueo a (SleepyThread.class) y se va a dormir. Recuerda, cuando un thread duerme no suelta el lock: solo lo mantiene. Por tanto sleep() es llamado un segundo, entonces t1 está en estado timed\_waiting.

Mientras tanto, el thread main() inicia el thread t2. Cuando su método run() es llamado, este va a intentar adquirir el bloqueo (SleepyThread.class). Sin embargo, tú sabes que el bloqueo ya lo tomó el thread t1 que sigue durmiendo (en timed\_waiting). Por lo tanto, t2 espera para adquirir el bloqueo, entonces está en estado de blocking. El método main() solo imprime los estados de estos dos threads llamando al método getState() después de iniciar los threads.

### Estado waiting

Este estado típicamente pasa cuando un thread espera por una condición, llamando al método wait(). El siguiente programa es un ejemplo sencillo de esto.

[WaitingThreadState.java](src/java/oca2/cap13/WaitingThreadState.java)

Este programa espera por siempre ya que no hay otro thread que lo saque de la espera con notify o notifyAll. ¿Qué pasa si cambiamos el método wait() por wait(1000)? ahora el programa imprime TIMED\_WAITING. El estado timed\_waiting pasa diferente que como sleep con timeout que viste antes; este también trabaja para el método wait() con valor timeout.

### Usar el enum Thread.State

La clase Thread define el enim Thread.State, que tiene una lista de posibles estados de un thread. El siguiente programa es un ejemplo simple que imprime el valor de los estados en este enum.

[ThreadStatesEnumeration.java](src/java/oca2/cap13/ThreadStatesEnumeration.java)

### Entender IllegalThreadStateException

Debes de ser cuidadoso siempre que escribas código para threads, siempre mantén en mente los estados de los threads. Si no ejercitas el cuidado de estos estados ¿Qué pasará? Vamos a ver un simple ejemplo en el siguiente programa.

[ThreadStateProblem.java](src/java/oca2/cap13/ThreadStateProblem.java)

Aquí estás intentando iniciar un thread que ya está iniciado (start()). Cuando llamas al método start(), el thread se mueve del estado new. Esta no es una transición si llamas al método start() otra vez. Entonces la JVM lanzará una IllegalThreadStateException.

¿Puedes arreglar el problema añadiendo un bloque try-catch alrededor del segundo start()? Esta es una mala solución ya que IllegalThreadStateException es una excepción RuntimeException, lo que significa que es un error de programación. Entonces necesitas corregir el problema en lugar de intentar manejar la excepción; en vez de eso, asegúrate de que el método start() no sea llamado dos veces por el mismo thread.

Veamos otro ejemplo

[ThreadStateProblem.java](src/java/oca2/cap13/ThreadStateProblem2.java)

El método wait(int) (con o sin timeout) debe ser llamado solo cuando se adquiere un bloqueo: una llamada a wait() agrega el thread a una cola del lock adquirido. Si no haces esto, esta no es una transición propia del estado running a timed\_waiting o waiting si el timeout no es proporcionado. Entonces el programa falla lanzando una IllegalMonitorStateException exception. Para arreglar este problema primero hay que adquirir un lock para después ponerlo en estado waiting; para esto puedes agregar el modificador synchronized al método run(). Ya que el método run() es sincronizado, wait() se agregará a sí mismo un bloqueo a su objeto referencia. Ya que no hay llamada a notify()/notifyAll() el wait(1000) retornará después de un segundo al método run(); puede hacerse esta espera con el método sleep(), la diferencia es que este último no libera el bloqueo, y wait() sí.

# Resumen

Introducción a la programación concurrente

* Puedes crear clases que sean multi-threading implementando la interface Runnable o extendiendo a la clase Thread.
* Siempre implementa el método run(). La implementación por default de run() no hace nada.
* Llama al método start() y no al método run(), ya que de esta manera no se ejecutará como multi-thread.
* Cada thread tiene un nombre, prioridad y thread-group asociado con él; la implementación por default de toString() imprime estos valores.
* Si llamas al método sleep() de un thread, el thread no libera el bloqueo.
* Puedes usar el método join() para esperar a que otro thread termine.
* En general, si no estás usando la característica interrupt, es seguro ignorar a InterruptedException; sin embargo está bien mantener la impresión del stackTrace si ocurriera una excepción.
* Los threads se ejecutan asincrónicamente: no puedes predecir el orden en que el thread corre.
* Los threads son también no-deterministicos: en muchos casos, no puedes reproducir problemas como deadlocks o data-races todo el tiempo.

Estados de un thread

* Aquí hay tres básicos estados: new, runnable y terminated. Cuando un thread está solo creado, está en estado new; cuando está listo para correr o corriendo, está en estado runnable. Cuando el thread muere está en estado terminated.
* El estado runnable tiene dos estados internamente (al nivel de sistema operativo): ready y running.
* Un thread puede entrar en estado de blocked cuando espera a adquirir un bloqueo. El thread estará en estado timed\_waiting cuando un timeout es dado para llamadas como wait(int). El thread entrará en estado waiting cuando se llama al método wait() sin timeout.
* Obtendrás una IllegalThreadStateException si tus operaciones caen en una transición de estado del thread inválido.

Problemas de acceso concurrente

* Escrituras y lecturas concurrentes a recursos pueden cargar problemas de data-race.
* Debes de usar sincronización de threads para proveer acceso protegido a recursos compartidos; llamemos bloques o métodos sincronizados.
* Usando bloqueos puede introducir un problema como deadlocks. Cuando un deadlock pasa el proceso se cuelga y nunca termina.
* Un deadlock típicamente pasa cuando dos threads adquieren múltiples bloqueos. Cuando un thread ha adquirido un bloqueo y espera por otro bloqueo, otro thread adquiere ese otro bloqueo y espera a que el primer bloqueo sea liberado. Entonces no habrá progreso en el programa y caerá en deadlocks.
* Para evitar deadlocks, es mejor evitar adquirir múltiples bloqueos. Cuando tienes que adquirir múltiples bloqueos, asegúrate que estos sean adquiridos en el mismo orden en todos los lugares del programa.

El mecanismo wait/notify

* Cuando un thread tiene que esperar a una condición particular o un evento de otro thread, puedes usar el mecanismo wait/notify como método de comunicación entre threads.
* Cuando un thread necesita esperar a una situación particular o evento, puedes llamar a wait() con o sin valor de timeout.
* Para evitar objetos que esperan el bloqueo sean olvidados, es mejor llamar a notifyAll() en lugar de notify().