**Estados de un Thread**

**Estado Nuevo**

El Thread ha sido creado pero no inicializado, es decir, no se ha ejecutado todavía el método start(). Se producirá un mensaje de error (IllegalThreadStateException) si se intenta ejecutar cualquier método de la clase Thread distinto de start().

**Estado Ejecutable**

El Thread puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado en beneficio de otro thread.

**Estado Bloqueado**

El Thread podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (E/S). Si un Thread está en este estado, no se le asigna tiempo de CPU.

**Estado Muerto**

La forma habitual de que un Thread muera es finalizando el método run(). También puede llamarse al método stop() de la clase Thread, dicho método es considerado “peligroso” y no debe utilizarse.

**Ampliación del diagrama de ciclo de vida todos los posibles estados**

Ampliando los conceptos de estado anteriores, un Thread desde su ejecución hasta su muerte pasa por diversos estados de vida. Estos estados y sus transiciones se muestran en forma más detallada en el siguiente diagrama:

**notify()**

**notifyAll()**

**wait()**

**sleep()**

**Expira el tiempo de sueño**

**\* stop()**

**Expiración de Quantum**

**Dispatcher asigna Quantum**

**\* suspend()**

**\* resume()**

**Inicia Operación de E/S**

**Termina Operación de E/S**

**start()**

**\*NOTA** Estos métodos ya están obsoletos (stop, suspend y resumen)

Ejecución -> En Espera -> wait()

Ejecución -> Dormido -> sleep()

Ejecución -> Muerto -> stop() No debe utilizarse

Ejecución -> Suspendido -> suspend()

Un Thread tras su creación pasa a estado LISTO de ahí puede pasar EN EJECUCION o no, esto dependerá de un Thread propio de Java denominado Dispatcher, que es el encargado de otorgarle a un Thread el recurso del procesador para que pueda ejecutarse.

Una vez EN EJECUCION procesara su código hasta que se le acabe el tiempo asignado (también llamado Quantum) por la JVM o bien hasta que termine (si le da tiempo), o también se puede dar el caso de que alguien lo elimine mediante stop().

Los Threads pasaran a ejecución dependiendo de sus prioridades y de acuerdo a la implementación de la JVM.

Una vez EN EJECUCION los Threads pueden, o pasar a LISTO de nuevo (si se acaba el Quantum) o pasar a otros estados, como: EN ESPERA, DORMIDO, SUSPENDIDO y BLOQUEADO. Esto dependerá de la ejecución de ciertos métodos sobre el Thread(o la ocurrencia de ciertos sucesos).

Un Thread pasara a DORMIDO cuando se invoque el método sleep(), permanecerá así (sin consumir recursos) hasta que se le acabe el tiempo de “siesta”, momento en el que volverá a LISTO. En este estado, el Thread no consume recursos, es decir, no es candidato a serle asignado el procesador hasta que no despierte.

El Thread pasara a BLOQUEADO cuando tenga que sufrir una espera debida a E/S, saldrá de este estado en cuanto termine la E/S (tampoco consume recursos mientras espera).

El estado SUSPENDIDO es para aquellos Threads que han sufrido la invocación del método suspend(), en este estado permanecerán hasta que alguien los llame mediante resume(). Por supuesto tampoco consume recursos en este estado.

Y por último el Thread pasara a EN ESPERA cuando alguien invoque un wait(), entonces el Thread pasara a esperar en un pool de Threads. Esto implica que la próxima vez que se ejecute un notify(), el Thread se despertara. También podemos ejecutar notifyAll(), de forma que sacamos a todos del pool.

Resumiendo, un Thread entra en el estado “No Ejecutable” cuando ocurre uno de estos cuatro eventos:

· Alguien llama a su método sleep()

· Alguien llama a su método suspend()

· El Thread utiliza su método wait() para esperar una condición variable.

· El Thread está bloqueado durante I/O

Esta lista indica la ruta de escape para cada entrada en el estado “No Ejecutable”:

· Si se ha puesto a dormir un Thread, deben pasar el número de milisegundos especificados.

· Si se ha suspendido un Thread, alguien debe llamar a su método resume ().

· Si un Thread está esperando una condición variable, siempre que el objeto propietario de la variable renuncie mediante notify () o notifyAll ().

· Si un Thread está bloqueado durante I/O, cuando se complete la I/O

Un Thread puede morir de dos formas: por causas naturales o siendo asesinado (parado). Una muerte natural se produce cuando su método run () sale normalmente. Se puede matar a un Thread en cualquier momento llamando a su método stop ().

El método stop () provoca una terminación súbita del método run () del Thread. Si el método run () estuviera realizando cálculos sensibles, stop () podría dejar el programa en un estado inconsistente. Normalmente, no se debería llamar al método stop ().

**NOTA: subproceso == hilo**

**notify() .- public final void notify()**

Es un método de la clase Object

Despierta un solo Thread (hilo) que está esperando en el monitor de este objeto. Si algún hilo está esperando en este objeto, se elige uno de ellos para despertarlo. La elección es arbitraria y ocurre a discreción de la implementación. Un hilo espera en el monitor de un objeto llamando a uno de los métodos de espera.

El hilo activado no podrá continuar hasta que el hilo actual renuncie al bloqueo de este objeto. El hilo activado competirá de la manera habitual con cualquier otro hilo que pueda estar compitiendo activamente para sincronizar en este objeto; por ejemplo, el hilo activado no disfruta de ningún privilegio o desventaja confiable al ser el siguiente hilo en bloquear este objeto.

Este método solo debe ser llamado por un hilo que sea el propietario del monitor de este objeto.

Un hilo se convierte en el propietario del monitor del objeto de una de estas tres formas:

* Ejecutando un método de instancia sincronizado de ese objeto.
* Ejecutando el cuerpo de una declaración sincronizada que se sincroniza en el objeto.
* Para objetos de tipo Clase, ejecutando un método sincronizado estático de esa clase.

Solo un hilo a la vez puede poseer el monitor de un objeto.

Lanza: IllegalMonitorStateException: si el hilo actual no es el propietario del monitor de este objeto.

**notifyAll() .- public final void notifyAll()**

Es un método de la clase Object

Despierta todos los threads (hilos) que están esperando en el monitor de este objeto. Un hilo espera en el monitor de un objeto llamando a uno de los métodos de espera.

Los hilos activados no podrán continuar hasta que el hilo actual renuncie al bloqueo de este objeto. Los hilos activados competirán de la manera habitual con cualquier otro hilo que pueda estar compitiendo activamente para sincronizar en este objeto; por ejemplo, los hilos activados no disfrutan de ningún privilegio o desventaja confiable al ser el siguiente hilo para bloquear este objeto.

Este método solo debe ser llamado por un hilo que sea el propietario del monitor de este objeto. Consulte el método de notificación para obtener una descripción de las formas en que un hilo puede convertirse en propietario de un monitor.

Lanza:

IllegalMonitorStateException: si el hilo actual no es el propietario del monitor de este objeto.

**public final void wait() throws InterruptedException**

Es un método de la clase Object

Hace que el subproceso actual espere hasta que se despierte, normalmente mediante una notificación o una interrupción.

En todos los aspectos, este método se comporta como si se hubiera llamado a wait(0L, 0). Consulte la especificación del método wait(long, int) para obtener más información.

Lanza:

IllegalMonitorStateException: si el subproceso actual no es el propietario del monitor del objeto InterruptedException: si algún subproceso interrumpió el subproceso actual antes o mientras el subproceso actual estaba esperando. El estado interrumpido del subproceso actual se borra cuando se lanza esta excepción.

**public final void wait(long timeoutMillis, int nanos) throws InterruptedException**

Es un método de la clase Object

Hace que el subproceso actual espere hasta que se despierte, normalmente mediante notificación o interrupción, o hasta que haya transcurrido una cierta cantidad de tiempo real.

El subproceso actual debe poseer el bloqueo del monitor de este objeto. Consulte el método de notificación para obtener una descripción de las formas en que un subproceso puede convertirse en propietario de un bloqueo de monitor.

Este método hace que el subproceso actual (al que aquí se hace referencia como T) se coloque en el conjunto de espera para este objeto y luego renuncie a todas y cada una de las reclamaciones de sincronización sobre este objeto. Tenga en cuenta que solo se cederán los bloqueos de este objeto; cualquier otro objeto en el que se pueda sincronizar el subproceso actual permanece bloqueado mientras el subproceso espera.

Entonces, el subproceso T se deshabilita para fines de programación de subprocesos y permanece inactivo hasta que ocurre uno de los siguientes:

Algún otro subproceso invoca el método de notificación para este objeto y el subproceso T se elige arbitrariamente como el subproceso que se despertará.

Algún otro subproceso invoca el método de notificación a todos para este objeto.

Algún otro hilo interrumpe el hilo T.

La cantidad especificada de tiempo real ha transcurrido, más o menos. La cantidad de tiempo real, en nanosegundos, viene dada por la expresión 1000000 \* timeoutMillis + nanos. Si timeoutMillis y nanos son cero, entonces el tiempo real no se tiene en cuenta y el subproceso espera hasta que una de las otras causas lo despierte.

El hilo T se despierta espuriamente. (Vea abajo.)

Luego, el subproceso T se elimina del conjunto de espera para este objeto y se vuelve a habilitar para la programación de subprocesos. Compite de la manera habitual con otros subprocesos por el derecho a sincronizar en el objeto; una vez que ha recuperado el control del objeto, todas sus reclamaciones de sincronización sobre el objeto se restauran al statu quo anterior, es decir, a la situación en el momento en que se invocó el método de espera. El subproceso T luego regresa de la invocación del método de espera. Por lo tanto, al regresar del método de espera, el estado de sincronización del objeto y del subproceso T es exactamente igual que cuando se invocó el método de espera.

Un subproceso puede activarse sin ser notificado, interrumpido o agotado, lo que se conoce como activación espuria. Si bien esto rara vez ocurrirá en la práctica, las aplicaciones deben protegerse comprobando la condición que debería haber provocado que se despertara el subproceso y continuar esperando si la condición no se cumple. Vea el ejemplo a continuación.

Para obtener más información sobre este tema, consulte la sección 14.2, "Colas de condiciones", en Java Concurrency in Practice de Brian Goetz y otros (Addison-Wesley, 2006) o el artículo 69 en Java efectivo de Joshua Bloch, segunda edición (Addison-Wesley, 2008).).

Si el subproceso actual es interrumpido por cualquier subproceso antes o mientras está esperando, se lanza una InterruptedException. El estado interrumpido del subproceso actual se borra cuando se lanza esta excepción. Esta excepción no se lanza hasta que el estado de bloqueo de este objeto se haya restaurado como se describe anteriormente.

Nota API:

El enfoque recomendado para la espera es verificar la condición que se espera en un bucle while alrededor de la llamada a esperar, como se muestra en el ejemplo a continuación. Entre otras cosas, este enfoque evita problemas que pueden ser causados por activaciones falsas.

sincronizado (obj) {

while (<la condición no se cumple> y <el tiempo de espera no se supera>) {

mucho tiempo de espera Millis = ... ; // volver a calcular los valores de tiempo de espera

int nanos = ... ;

obj.wait(timeoutMillis, nanos);

}

... // Realiza la acción apropiada para la condición o el tiempo de espera

}

Parámetros:

timeoutMillis - el tiempo máximo de espera, en milisegundos nanos - tiempo adicional, en nanosegundos, en el rango 0-999999 inclusive

Lanza:

IllegalMonitorStateException: si el subproceso actual no es el propietario del monitor del objeto InterruptedException: si algún subproceso interrumpió el subproceso actual antes o mientras el subproceso actual estaba esperando. El estado interrumpido del subproceso actual se borra cuando se lanza esta excepción. IllegalArgumentException: si timeoutMillis es negativo o si el valor de nanos está fuera de rango.

**public final void sleep(long long millis) throws InterruptedException**

Es un método de la clase Object

Hace que el subproceso que se está ejecutando entre en suspensión (deje de ejecutarse temporalmente) durante el número especificado de milisegundos, sujeto a la precisión y exactitud de los temporizadores y programadores del sistema. El subproceso no pierde la propiedad de ningún monitor.

Parámetros:

millis - la cantidad de tiempo para dormir en milisegundos

Lanza:

InterruptedException: si algún subproceso ha interrumpido el subproceso actual. El estado interrumpido del subproceso actual se borra cuando se lanza esta excepción. IllegalArgumentException: si el valor de millis es negativo.

**public final void suspend()**

Es un método de la clase java.lang.Thread

Obsoleto. Este método ha quedado obsoleto, ya que es inherentemente propenso a interbloqueos. Si el subproceso de destino mantiene un bloqueo en el monitor que protege un recurso crítico del sistema cuando se suspende, ningún subproceso puede acceder a este recurso hasta que se reanude el subproceso de destino. Si el subproceso que reanudaría el subproceso de destino intenta bloquear este monitor antes de llamar a reanudar, se produce un interbloqueo. Estos interbloqueos suelen manifestarse como procesos "congelados".

Para obtener más información, consulte ¿Por qué Thread.stop, Thread.suspend y Thread.resume están en desuso?

Suspende este hilo.

Primero, se llama al método checkAccess de este subproceso sin argumentos. Esto puede resultar en lanzar una SecurityException (en el hilo actual).

Si el subproceso está activo, se suspende y no avanza a menos que se reanude.

Lanza:

SecurityException: si el subproceso actual no puede modificar este subproceso.

**public final void stop()**

Es un método de la clase java.lang.Thread

Obsoleto. Este método es intrínsecamente inseguro. Detener un subproceso con Thread.stop hace que desbloquee todos los monitores que ha bloqueado (como consecuencia natural de la excepción ThreadDeath no verificada que se propaga hacia arriba en la pila). Si alguno de los objetos previamente protegidos por estos monitores se encontraba en un estado inconsistente, los objetos dañados se vuelven visibles para otros subprocesos, lo que podría generar un comportamiento arbitrario. Muchos usos de stop deben reemplazarse por código que simplemente modifica alguna variable para indicar que el subproceso de destino debe dejar de ejecutarse. El subproceso de destino debe verificar esta variable regularmente y regresar de su método de ejecución de manera ordenada si la variable indica que debe dejar de ejecutarse. Si el subproceso de destino espera durante largos períodos (en una variable de condición, por ejemplo), se debe utilizar el método de interrupción para interrumpir la espera. Para obtener más información, consulte ¿Por qué Thread.stop, Thread.suspend y Thread.resume están en desuso?

Obliga al subproceso a dejar de ejecutarse.

Si hay un administrador de seguridad instalado, se llama a su método checkAccess con este como argumento. Esto puede dar lugar a que se genere una SecurityException (en el subproceso actual).

Si este subproceso es diferente del subproceso actual (es decir, el subproceso actual está intentando detener un subproceso que no sea él mismo), se llama además al método checkPermission del administrador de seguridad (con un argumento RuntimePermission("stopThread")). Nuevamente, esto puede resultar en el lanzamiento de una SecurityException (en el hilo actual).

El subproceso representado por este subproceso se ve obligado a detener cualquier cosa que esté haciendo de manera anormal y lanzar un objeto ThreadDeath recién creado como una excepción.

Está permitido detener un hilo que aún no se ha iniciado. Si finalmente se inicia el subproceso, finaliza inmediatamente.

Normalmente, una aplicación no debería intentar atrapar ThreadDeath a menos que deba realizar alguna operación de limpieza extraordinaria (tenga en cuenta que el lanzamiento de ThreadDeath hace que las cláusulas finalmente de las declaraciones de prueba se ejecuten antes de que el hilo muera oficialmente). Si una cláusula catch atrapa un objeto ThreadDeath, es importante volver a lanzar el objeto para que el hilo realmente muera.

El controlador de errores de nivel superior que reacciona a las excepciones no detectadas no imprime un mensaje ni notifica a la aplicación si la excepción no detectada es una instancia de ThreadDeath.

Lanza:

SecurityException: si el subproceso actual no puede modificar este subproceso.

**public final void reseme()**

Es un método de la clase java.lang.Thread

Obsoleto. Este método existe únicamente para su uso con suspensión, que ha quedado obsoleto porque es propenso a interbloqueos. Para obtener más información, consulte ¿Por qué Thread.stop, Thread.suspend y Thread.resume están en desuso?

Reanuda un hilo suspendido.

Primero, se llama al método checkAccess de este subproceso sin argumentos. Esto puede resultar en lanzar una SecurityException (en el hilo actual).

Si el hilo está vivo pero suspendido, se reanuda y se le permite avanzar en su ejecución.

Lanza:

SecurityException: si el subproceso actual no puede modificar este subproceso.