7. Hilos

Objetivos

Este capítulo cubre los siguientes aspectos, de los objetivos del examen de certificación en Java:

- Escribir código para definir, instanciar e iniciar nuevos hilos usando **java.lang.Thread** y **java.lang.Runnable**.
- Reconocer condiciones que podrían impedir la ejecución de un hilo.
- Escribir código usando **synchronized**, **wait**, **notify**, y **notifyAll** para proteger contra problemas de acceso concurrente y para comunicación entre hilos. Definir la interacción entre hilos y entre hilos y objetos cuando se ejecuta **synchronized**, **wait**, **notify**, o **notifyAll**.

Los Hilos son una forma que tiene Java de hacer ver una sola Maquina Virtual de Java como muchas maquinas, todos corren a la vez. Este efecto, usualmente, es una ilusión: hay solamente una MVJ y muy a menudo solo una CPU; pero la CPU alterna entre las MVJ de varios proyectos para dar la impresión de muchas CPUs.

Java suministra un buen numero de herramientas para crear y manejar hilos. Los hilos son una herramienta muy valiosa para simplificar el diseño de programas, permitiendo que el trabajo no relacionado o relacionado muy poco sea programado separadamente, mientras se ejecutan concurrentemente. Hay sistemas de hilos que trabajan detrás de las aplicaciones, escuchando entradas de usuarios y administrando la recolección de basura. La mejor forma para contribuir con estas facilidades es entender que son realmente los hilos.

Los objetivos de la Certificación requieren que usted se familiarice con el soporte de los hilos de Java, incluyendo los mecanismos para crear, controlar, y comunicarse entre ellos.

Fundamentos sobre hilos

El soporte de hilos de Java reside en tres lugares:

- La clase **java.lang.Thread**
- La clase **java.lang.Object**
- El lenguaje Java y la maquina virtual

Muchos soportes (pero no definitivamente todos) residen en la clase **Thread** . En Java, cada hilo corresponde a una instancia de la clase **Thread**. Estos objetos pueden estar en varios estados: en cualquier momento, al menos uno de estos se esta ejecutando por CPU, mientras otros podrían estar

esperando por recursos, o esperando por un cambio para ejecutarse, o dormirse, o morir.

En orden para demostrar una comprensión de hilos, usted necesita ser capaz de responder una pocas preguntas:

- Cuándo un hilo se ejecuta, que código ejecuta este?
- En que estados puede estar un hilo?
- Como cambia un hilo su estado? Las próximas secciones verá cada una de estas preguntas.

Como se ejecuta un hilo

Para hacer que un hilo se ejecute, usted debe llamar su método **start()**. Este registra el hilo con una pieaza del código del sistema llamada el programador de hilos (*thread scheduler)*. El programador podría ser parte de la MVJ o ser parte del sistema operativo . El programador determina cual hilo esta actualmente corriendo en cada CPU disponible en cualquier momento dado. Note que llamar el método **start()** no hace que el hilo corra inmediatamente; este solo hace que esté elegible para correr. El hilo podría aun competir por un tiempo de CPU con todos los otros hilos.

Si todo está bien, en algún momento el programador de hilos permitirá que su hilo se ejecute. Durante su tiempo de vida, un hilo gasta algún tiempo ejecutándose y otro en cualquiera de los diversos estados de no-ejecución . En esta sección, usted puede ignorar (de momento) las preguntas de como el hilo cambie entre estados. La pregunta de hecho es: cuándo el hilo consigue ejecutarse, que hace esta ejecución?

La respuesta más simple es que este ejecuta un método llamado **run()**. Pero el método **run()** de cual objeto? Usted tiene dos elecciones:

- El hilo puede ejecutar su propio método **run()**.
- El hilo puede ejecutar el método **run()** de algunos otros objetos.

Si usted quiere que el hilo ejecute su propio método **run()**, usted necesita para la subclase la clase **Thread** y darle a su subclase un **run()**. Por ejemplo

```
1. public class CounterThread extends Thread {
2.         public void run() {
3.                 for ( int i = 1; i <= 10; i++ ) {
4.                         System.out.println( "Counting: " + i );
5.                 }
6.         }
7. }
```

Este método **run()** solamente imprime los números de 1 a 10. Para hacer esto en un hilo, usted primero construye una instancia del CounterThread y

entonces invoca su método **start()**:

```
1. CounterThread ct = new CounterThread();
2. ct.start();       // start(), no run()
```

Que no llama directamente **run()** ; que podría solo contar hasta 10 en el actual thread. En su lugar, usted llama **start()**, en el cual la clase CounterThread hereda de su clase padre , **Thread**. El método **start()** registra el hilo (que es, ct) con el programador de hilos; eventualmente el hilo se ejecutará, y a la vez su método **run()** será llamado.

Si usted quiere que su hilo ejecute el método **run()** de cualquier otro objeto , se necesitará construir una instancia de la clase **Thread** . La única diferencia es que cuando se llama el constructor del **Thread** , se tiene que especificar cual objeto es el dueño del método **run()** que usted quiere. Para hacer esto, se invoca una forma alterna del constructor del **Thread** :

```
public Thread( Runnable target );
```

La interface Runnable describe un único método:

```
public void  run();
```

Así se puede pasar cualquier objeto que quiera dentro del constructor, suministrado este implementa la interface **Runnable** (dado que este realmente tiene un método **run()** para el programador de hilos que invoca).

Habiendo construido una instancia del **Thread**, se procede como antes : Usted invoca el método **start()** . Como antes, este registre el hilo con el programador, y eventualmente el método **run()** del objetivo será llamado.

Por ejemplo, la siguiente **class** tiene un método **run()** que cuenta en forma descendente de 10 a 1:

```
1. public class DownCounter implements Runnable {
2.         public void run() {
3.                 for ( int i = 10; i >= 1;   i-- ) {
4.                         System.out.println( "Counting Down: " + i );
5.                 }
6.         }
7. }
```

Esta **clase** no extiende de **Thread**. Sin embargo, esta tiene un método **run()** , y este declara que implementa la interface **Runnable** . Así cualquier instancia de la clase DownCounter es candidata para ser pasada en el constructor alterno para el **Thread**:

```
1. DownCounter dc = new DownCounter();
2. Thread t = new Thread( dc );
```

3. t.start();

Esta sección ha presentado dos estrategias para construir hilos. Superficialmente, la única diferencia entre estas dos estrategias es la ubicación del método **run()**. La segunda estrategia es quizás un poco mas complicada que en el caso de los ejemplos sencillos que hemos considerado. Sin embargo, hay muy buenas razones por la que usted debería hacer un esfuerzo extra.

El método **run()** , como cualquier otro método miembro, tiene permiso de accesar los datos privados, y llamar los métodos privados, de la clase de la cual es miembro. Colocando **run()** en una subclase del **Thread** principal significa que el método no puede obtener las características que este necesita (o no puede obtener estas características de una manera razonable).

Otra razón que le podría persuadir para implementar sus hilos usando **Runnable** mejor que hacer una subclase **Thread** es la única implementación de la regla de herencia. Si usted escribe una subclase de **Thread**, entonces esta no puede ser una subclase de cualquier otra.

Mientras que si usa **Runnable**, usted puede hacer subclases de cualquier otra clase padre que elija. De esta manera implementar **Runnable** es mucho mas aplicable generalmente, esto podría tener mas sentido si lo tomara como habito, mejor que complicar su código con hilos creados en diferentes formas en diferentes lugares. Finalmente, desde un punto de vista orientado a objetos, la "thread-iness" de una clase es usualmente periférica para la naturaleza esencial de cualquiera que usted este creando, y por lo tanto hacer una subclase **Thread**, la cual dice en efecto "Esta clase 'es un hilo" es probablemente una elección de diseño inapropiada. Usando **Runnable**, el cual dice "esta clase esta asociada con un hilo," tiene mas sentido.

Para resumir, hay dos enfoques para especificar cual método **run()** será ejecutado por un hilo:

- Subclase **Thread**. Define su método **run()** en la subclase.
- Escribe una clase que implementa **Runnable**. Define su método **run()** en la clase. Pasa una instancia de la clase dentro de su llamada al constructor **Thread**.

Cuando finaliza una Ejecución

Cuando el método **run()** retorna, el hilo ha finalizado su tarea y es considerado muerto. No hay forma fuera de este estado. Una vez un hilo esta muerto, este no podría inicializarce de nuevo; si usted quiere que la tarea del hilo sea ejecutada de nuevo, usted tiene que construir e inicializar una nueva instancia del hilo. El hilo muerto está listo para existir; este es un objeto como cualquier otro objeto, y usted puede aun accesar sus datos y

llamar su método. Lo que no puede hacer es correrlo de nuevo. En otras palabras,

- Usted no puede reiniciar un hilo muerto.
- Usted puede llamar los métodos de un hilo muerto.

Los métodos de **Thread** incluyen un método llamado **stop()**, cual forzosamente termina un hilo, colocando este en el estado muerto Este método es desaprobado desde JDK 1.2, porque este puede causar corrupción de datos o un punto si salida si usted mata un hilo que esta en una sección critica del código. El método **stop()** por lo tanto no se cubre en gran parte del examen de Certificación. En lugar de usar **stop()**, Si un hilo necesitara terminar su actividad desde otro hilo, debería mandarle a este un **interrupt()** desde el método anulador .

Aunque se puede reiniciar un hilo muerto, si usted usa runnables, usted puede ceder la instancia antigua **Runnable** al nuevo hilo. Sin embargo, este es generalmente un diseño pobre para constantemente crear, usar, y descartar hilos, porque construir un **Thread** es una operación relativamente pesada, involucrando fuentes significativas del kernel . Es mejor crear un conjunto de hilos reutilizables que puedan ser asignados a tareas cada que sea necesario . Esto se discute mejor en el capitulo 17 (en la sección examen del desarrollador).

Estados de Hilos

Cuando se llama el **start()** de un hilo, el hilo no corre inmediatamente. Este va al estado "listo-para-correr" y permanece ahí hasta que el programador cambia este al estado "Ejecutándose". Luego el método **run()** es llamado. En el curso de ejecución **run()**, el hilo podría temporalmente ganar la CPU y entrar a algún otro estado por un while. Es importante estar enterado de los posibles estados en los que un hilo podría estar y de los disparadores que pueden causar que el hilo cambie.

Los estados de los hilos son:

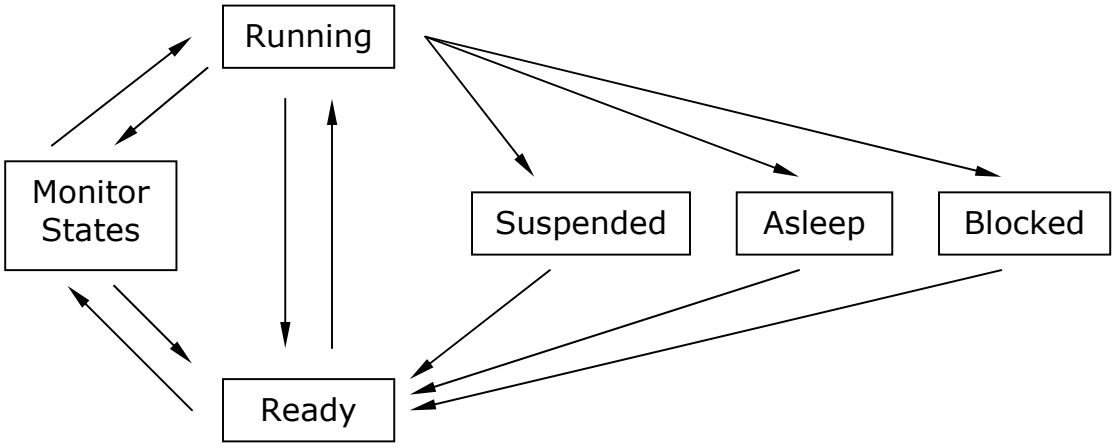
Ejecutándose	El estado a el que todos los hilos aspiran
Varios estados de espera	Waiting, Sleeping, Suspended, Blocked
Listo	No espera nada excepto la CPU
Muerto	Todo hecho

La Figura 7.1 muestra los estados . Note que la figura no muestra como el estado muerto.

En la parte superior de la figura 7.1 esta el estado Ejecutándose . En la parte inferior esta el estado Listo. Y entre ellos están los demás estados. Un hilo en uno de estos estados intermedios espera hasta que algo pase; cuando algo eventualmente pasa, el hilo cambia al estado Ready, y eventualmente el

programador de hilos permitirá que este corra de nuevo. Note que los métodos asociados con el estado suspendido son ahora ignorados; usted no tendrá que probar con este estado o sus métodos asociados en el examen. Por esta razón, nosotros no los discutiremos en detalle en este libro.

Las flechas entre los cuadros en la Figura 7.1 representa las transiciones de los estados .Note que solamente el programador de hilos puede cambiar a hilo a listo dentro de la CPU.
FIGURA 7.1
Estados de los hilos



Después en este capitulo, usted examinara en detalle los diversos estados de espera. Por ahora, lo importante para observar en la Figura 7.1 es el flujo general :

Un hilo ejecutándose entra en un estado intermedio por alguna razón; después el hilo espera a que algo pase, y el hilo entra en el estado Ready ; después, El programador concede la CPU al hilo. La excepción para este flujo general involucra código **synchronized** y la secuencia **wait**/**notify** ; la parte correspondiente de la Figura 7. 1 describe como a un cuadro no definido marcado como "Monitor States." Este monitor de estados se discute después en la sección "**Monitors**, **wait**(), y **notify**()".

Prioridades de los hilos

Cada hilo tiene una prioridad. La prioridad es un entero de 1 a 10. Los hilos con mas alta prioridad tienen preferencia sobre los hilos con mas baja prioridad es considerado por el programador de hilos cuando este decide este podría ejecutarse. El programador generalmente elige el hilo mas alto. Si hay mas de un hilo esperando, El programador elige uno de ellos. No hay garantía que el hilo elegido sea el que mas tiempo ha estado esperando.

La prioridad por defecto es 5, pero todos los nuevos hilos creados tienen que establecer su prioridad en la creación del hilo. Para definir la prioridad de un hilo, llame el método **setPriority()**, pasándole la prioridad deseada. El método **getPriority()** retorna la prioridad de un hilo.

El fragmento de código abajo incrementa la prioridad de un hilo , suministrando la prioridad menor que 10. En lugar de colocar con código duro el valor 10, el fragmento usa la constante MAX_PRIORITY. La clase **Thread** también define constantes para MIN_PRIORITY (la cual es 1), y NORM_PRIORITY (la cual es 5).

```
1. int oldPriority = theThread.getPriority();
2. int newPriority = Math.min( oldPriority+1, Thread.MAX_PRIORITY );
3. theThread.setPriority( newPriority );
```

Los detalles específicos de cómo las prioridades de los hilos afectan la programación son dependientes de la plataforma. la especificación de Java declara que hilos podrían tener prioridades ,pero esta no fija precisamente que debería hacer el programador con ellas. Esta ambigüedad es un problema: Los algoritmos que confían en la manipulación de las prioridades de los hilos podrían quizás no correr consistentemente sobre todas las plataformas.

Control de Hilos

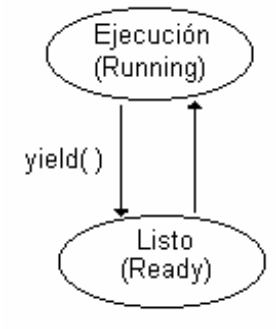
El control de hilos, es el arte de mover hilos de estado a estado. Usted controla los hilos disparando estados de transición. Esta sección examina varias formas del estado corriendo(Running) . Estas formas son

- Yielding
- Suspendido y luego resumido
- Dormido y luego waking up
- Bloqueado y luego continuando
- Esperando y luego ser anunciado

Yielding

Un hilo puede ofrecerse para moverse fuera de la CPU virtual por yielding.Una llamada al método **yield( )** causa que la corriente ejecución del hilo pase al estado listo(Ready) si el programador esta pendiente para correr cualquier otro hilo en lugar del hilo yielding . La transición del estado es mostrada en la Figura 7.2.
Un hilo que ha sucumbido va al estado listo (Ready) . Hay dos posibles escenarios. Si cualquier otro hilo esta en estado Ready , entonces el hilo que solo sucumbe podría esperar un tiempo mientras este se ejecuta de nuevo .

FIGURA 7.2



Si embargo, si ningún otro hilo esta esperando, entonces el hilo que solo sucumbe se quedará para continuar ejecutándose inmediatamente. Note que la mayor parte de los programadores no paran el hilo que sucumbe de correr en favor de un hilo de baja prioridad.

El método **yield()** es un método estático de la clase **Thread** . Este siempre origina que el hilo ejecutándose corrientemente sucumba.
Al ceder se le permite a un hilo consumir un tiempo para permitir que otro hilo se ejecute. Por ejemplo, considere un applet que calcula una imagen de pixeles 300 x 300 usando un algoritmo ray-tracing. El applet podría tener un botón "Calcular" y un botón de "Interrupir". El evento generado por el botón "Calcular" crearía e iniciaría un hilo separado, el cual llamaría un método traceRaysQ. Una primera parte para este código podría verse como sigue:

```
1. private void traceRays()  {
2.   for (int j=0;   j<300;  j++)  {
3.       for (int i=0;   i<300;i++)  {
4.           computeOnePixel(i, j);
5.       }
6.    }
7. }
```

Hay 90,000 valores color de pixeles que calcular. Si toma 0.1 segundo calcular el valor del color de un pixel, entonces tomaría dos horas y media calcular la imagen completa.

Suponga que después de media hora el usuario ve la imagen parcialmente y se da cuenta que algo esta mal. (Quizás el punto de vista o el factor de zoom son incorrectos.) El usuario hará click en el botón "Interrumpir" de esta forma no tiene sentido seguir calculando la imagen inservible . Desafortunadamente, el hilo que maneja la entrada GUI podría quizás no conseguir una oportunidad de ejecutarse hasta que el hilo que está ejecutando traceRays() suelte la CPU. De esta manera el botón "Interrumpir"

podría no tener efecto durante las otras dos horas.

Si las prioridades son implementadas en el programador, entonces bajando la prioridad del hilo ray-tracing tendrá el efecto deseado, asegurando que el hilo GUI correrá cuando este tenga algo útil que hacer. Sin embargos, esto no es confiable entre plataformas (aunque es una buena acción de cualquier forma, dado que este trabajará en muchas plataformas). La aproximación confiable es tener el hilo ray-tracing periódicamente cediendo. Si no ha7y entradas pendientes cuando el hilo yield es ejecutado, entonces el hilo ray-tracing no dejará la CPU. Si,de otra parte, hay una entrada para ser procesada, el hilo input-listening tendrá la oportunidad de ejecutarse.

El hilo ray-tracing puede obtener su prioridad como se muestra a continuación:

```
rayTraceThread. setPriority(Thread. NORM_PRIORITY-1);
```

El método traceRaysQ listado anteriormente puede ceder despues que cada valor de pixel es calculado, despues de la línea 4. La version revisada se ve como sigue:

```
1. private void traceRays()  {
2.    for (int j=0;   j<300;{
3.       for (int i=0;   i<300;{
4.         computeOnePixel(i,  j);
5.         Thread.yield();
6.      }
7.    }
8. }
```

Suspendido (Suspending)

Suspender un hilo es un mecanismo que le permite a cualquier hilo arbitrario hacer que otro hilo pase a no-ejecución (un-runnable) por un periodo de ,tiempo indefinido. El hilo suspendido vuelve estar en ejecución cuando algún otro hilo reanude este. Esto podría considerarse como un mecanismo útil, pero hace que sea muy fácil causar un punto muerto en un programa usándolo, ya que un hilo no tiene control sobre si cuando este es suspendido (el control se obtiene de un hilo externo) y este podría estar en una sección critica . El efecto exacto de suspender y resumir es mejor implementarlo usando **wait** y **notify** y quizás un pequeño cambio para el diseño. Ya que suspend() y resume() no son tenidos en cuenta y no se evalúan en el examen de certificación, no trataremos estos mas adelante.

Dormido(Sleeping)

Un hilo dormido pasa el tiempo sin hacer nada y sin usar la CPU. Una llamada al método sleep() requiere que el hilo se este ejecutando

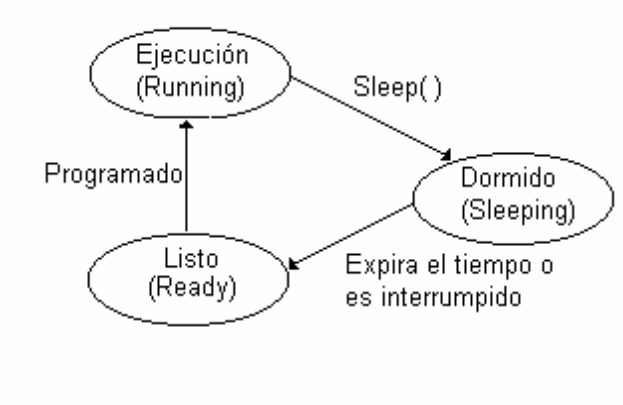
corrientemente para cesar su ejecución por (aproximadamente) una cantidad especifica de tiempo. Hay dos formas de llamar este método , dependiendo si se quiere especificar el periodo para dormir con una precisión en milisegundos o nanosegundos:

- **public** static **void** sleep(long milliseconds) throws InterruptedException
- **public** static **void** sleep(long milliseconds, **int** nanoseconds) throws InterruptedException

Note que sleep(), como yield(), es estático. ambos métodos operan sobre el hilo que se esta ejecutando corrientemente.
El diagrama de estados para dormido(sleeping) es mostrado en la Figura 7.3. Note que cuando el hilo ha finalizado el estado sleeping, este no continua la ejecución. Como podría esperarse, este entra al estado listo y solamente se ejecutará cuando el programador de hilos se lo permita. Por esta razón, podría esperarse que una llamada a sleep() bloqueará un hilo al menos por el tiempo requerido, pero este podría bloquearse por mucho mas tiempo. Esto sugiere piense con cuidado lo que haría en dado momento su diseño antes de esperar cualquier resultado de la precisión de la versión nanosegundos del método sleep().

La clase **Thread** tiene un método llamado interrupt(). Un hilo dormido(sleeping) que recive una llamada interrupt() cambia inmediatamente al estado listo(Ready); cuando este logra correr, ejecutara su manejador InterruptedException.

FIGURE 7.3
El estado Dormido (Sleeping)



Bloqueado(Blocking)

Muchos métodos que ejecutan entrada o salida deben esperar por alguna ocurrencia en el mundo exterior antes que ellos puedan proceder; este comportamiento es conocido como bloqueado(blocking). Un buen ejemplo es la lectura de un socket:

```
1. try  {
2. Socket sock = new SocketC"magnesium", 5505);
3. InputStream istr = sock.getlnputStreamQ;
4. int b = istr.readQ;
5. }
6. catch (lOException ex) {
7. // Manejador para la excepción
8. }
```

Si no está familiarizado con sockets en Java y funcionalidad de flujo, no se preocupe: Esta es cubierta en el capitulo 13. El problema no es complicada .

Esta muestra como en la Linea 4 lee un byte de una entrada de flujo que es conectada al puerto 5505 sobre una maquina llamada "magnesium." Actualmente, la línea 4 trata de leer un byte. Si un byte está disponible (Si magnesium tiene previamente escrito un byte), entonces la línea 4 puede retornar inmediatamente y la ejecución puede continuar. Si magnesium no tiene escrito nada, sin embargo, la llamada a read() tiene que esperar. Si magnesium esta ocupada haciendo otras cosas y le toma una hora y media para escribir un byte de nuevo, entonces el llamado read() tendrá que esperar por una hora y media.

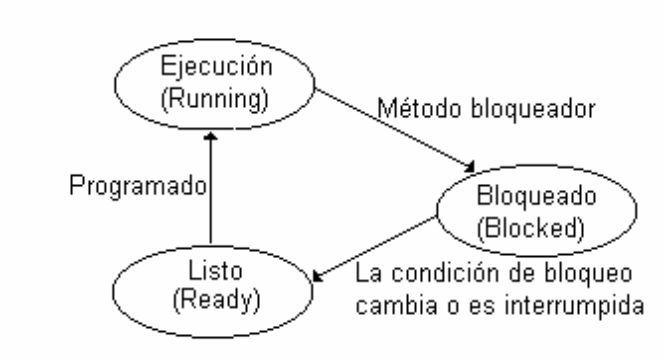
Claramente, esto podría ser un serio problema si el hilo en ejecución llama a read() en la línea 4 permaneciendo en el estado ejecución (Running) por toda la media hora . Note que esto pudría pasar. En general, si un método necesita esperar una cantidad de tiempo indeterminada hasta que algún evento de I/O tome lugar, entonces un hilo en ejecución podría graciosamente pasar fuera del estado de ejecución. Todos los métodos de I/O de Java tienen este comportamiento. Un hilo que tiene este gracioso paso de esta manera pasa al estado bloqueado. La Figura 7.4 muestra estas transiciones.

Estado Bloqueado (Blocked)

En general, si usted ve un método con un nombre que sugiere que este podría no hacer nada hasta que algo este listo, por ejemplo esperar por una entrada, o esperar tForImages( ), se esperaría que la llamada al hilo quedara bloqueada, llegando a estar fuera de ejecución y perdiendo la CPU, cuando el método es llamado. No necesita saber acerca de las APIs para suponer esto; este es un principio general de las APIs, ambas son el centro , en un medio Java.

FIGURE 7.4

El estado Bloqueado

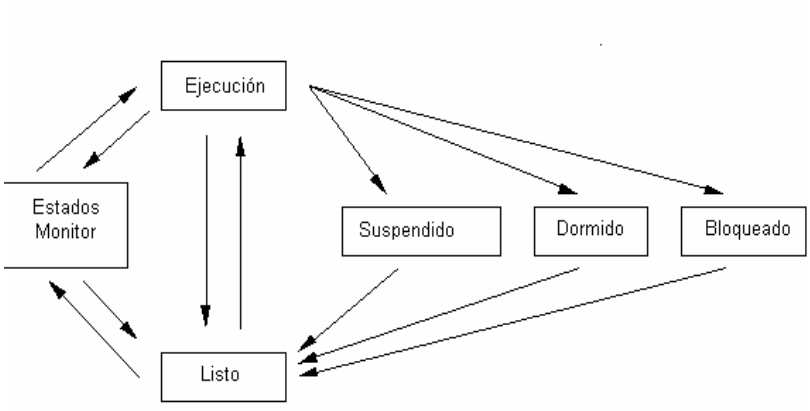


Un hilo puede tambien llegar a quedar bloqueado si este falla para adquirir el seguro para un monitor o si este emite un llamado al método **wait**(). Los seguros y monitores son explicados en detalle mas adelante en este capitulo, en la seccion "Monitores, **wait**(), y **notify**()." Internamente, muchos bloqueos para I/O, como llamadas a read() son discutidas, se implementan usando llamadas a **wait**() y **notify()**.

Estado Monitor

La Figura 7.5 muestra todos los estados de transición de los hilos . Los estados intermedios del lado derecho de la figura (Suspendido, Dormido, y Bloqueado) han sido discutidos en las secciones previas. Los estados de Monitor son dibujados solos en el lado izquierdo de la figura para hacer énfasis en que ellos son muy diferentes de los otros estados intermedios .

FIGURE 7.5
Estados de los hilos (reprise)



EL método **wait**() coloca un hilo en ejecución en el estado de espera (**Wait**ing), y los métodos **notify**( ) y **notifyAll**() colocan los hilos que están en espera al estado listo (Ready). Sin embargo, Estos método son muy diferentes de suspend(), resume(), y yield( ). Por lo siguiente, Ellos esrtan implementados en la claseObject , no en la clase **Thread**. Por otro lado, ellos pueden solamente ser llamados en el código **synchronized** . el estado de espera (**Wait**ing), y lo asuntos asociados, son discutidos en la sección final de este capitulo. Pero primero, hay un tópico mas concerniente al control de hilos que tratar.

Implementaciones de la Programación

Históricamente , dos aproximaciones han surgido para implementar los programadores de hilos:

- Programación Preferencial
- Divisiones de tiempo o programación round-robin

Hasta ahora, las facilidades descritas en este capitulo han sido preferenciales. En programación preferencial , hay solo dos formas para que un hilo salga del estado de ejecución (Running) sin llamar explícitamente un método programador de hilos al como **wait**Q or suspendQ:

- Este puede cesar de estar listo para ejecutarse (llamando un método de bloqueo I/O , por ejemplo).
- Este puede conseguir ser cambiado fuera de la CPU por un hilo de mas alta prioridad que este listo para ejecutarse.

Con divisiones de tiempo, un hilo puede solamente ejecutarse por una cantidad de tiempo limitada. Este luego es cambiado al estado listo, donde este competiría con todos los otros hilos en este mismo estado. Las divisiones de tiempo aseguran de nuevo la posibilidad de que un único hilo de alta prioridad consiga el estado de ejecución y nunca estén fuera, previniendo a todos los otros hilos de hacer estas tareas. desafortunadamente, las divisiones de tiempo crean un sistema no-determinístico; en cualquier momento no se puede saber con certeza cual hilo se está ejecutando o durante cuanto tiempo este continuará la ejecución. Es natural preguntar que implementación usa Java. La respuesta es que esto depende de la plataforma; la especificación Java da implementaciones un poco libres. Las maquinas Solaris son preferenciales. Las Macintosh son por intervalos de tiempo. Las plataformas Windows fueron originalmente preferenciales, pero cambiaron a intervalos de tiempo con la versión 1.0.2 del JDK.

Monitores, wait(), y notify()

Un monitor es un objeto que puede bloquear y revivir hilos. El concepto es simple, pero toma un poco de trabajo entender para que son buenos los

monitores y como usarlos eficientemente.
La razón de tener monitores es que algunas veces un hilo no puede ejecutar su tarea hasta que un objeto alcance un determinado estado. Por ejemplo, considere una clase que maneja peticiones para escribir por una salida estandar:

```
1.  class Mailbox  {
2.      public boolean  request;
3.      public String     message;
4.  }
```

La intención de esta clase es que un cliente pueda colocar algún mensaje de algún valor, luego colocar la solicitud a verdadero:

```
1. myMailbox.message = "Hello everybody.";
2. myMailbox.request = true;
```

Aquí podría haber un hilo que chequee la solicitud; al encontrar este en verdadero, el hilo podría escribir un mensaje para el sistema, fuera y luego colocar la solicitud a falso. (Colocando la solicitud a falso indica que el objeto mailbox esta listo para manejar otra solicitud) es atractivo para implementar el hilo como esto:

```
1. public class Consumer extends Thread {
2.   private Mailbox myMailbox;
3.
4. public Consumer(Mailbox box)  {
5.     this.myMailbox = box;
6.  }
7.
8. public void run()   {
9.        while (true)  {
10.         if (myMailbox.request)   {
11.         System.out.pri ntln(myMai1 box.message);
12.         myMailbox.request = false;
13.    }
14.
15. try  {
16.       sleep(50);
17.        }
18. catch (InterruptedException e)   {   }
19.    }
20.}
```

El hilo consumidor entra en un ciclo infinito, chequeando las peticiones cada 50 milisegundos. Si hay una solicitud (línea 10), el consumidor escribe el mensaje a la salida estandar (línea 11) y luego coloca las solicitudes a falso

para mostrar que esta listo para mas solicitudes.

La clase Consumer puede verse bien aparentemente, pero esta tiene dos problemas serios:

• La clase Consumer accesa datos internos de la clase Mailbox , introduciendo la posibilidad de corrupción. En un sistema time-sliced, el hilo consumidor podría solo posiblemente ser interrumpido entre las líneas 10 y 11. El hilo interruptor podría solo posiblemente ser un cliente que coloca mensajes para sus propios mensajes (ignorando la convención de chequear solicitudes para ver si el manejador esta disponible). El hilo consumidor podría enviar un mensaje equivocado.

• La elección de 50 milisegundos para el retardo puede no ser ideal. Algunas veces 50 milisegundos será demasiado tiempo, y los clientes recibirán un servicio lento; algunas veces 50 milisegundos será muy frecuente, y los ciclos serán desaprovechados. Un hilo que quiere enviar un mensaje tiene un dilema similar si este encuentra las solicitudes colocadas por solicitud de bandera: El hilo estaría off por un rato , pero durante cuanto tiempo? Idealmente, Estos problemas podrían ser solucionados haciendo algunas modificaciones a la clase Mailbox:

• La clase mailbox podría ser capaz de proteger sus datos de clientes irresponsables.

• Si el mailbox no esta disponible—lo que significa, si la bandera de solicitud ya esta colocada—entonces un cliente consumidor no tendría que adivinar cuanto tiempo tener que esperar antes de chequear la bandera de nuevo . El manejador llamaría el cliente cuando el tiempo este bien.

Los monitores de Java soportan direcciones estos asuntos lo suministran el siguiente código:

• Una clave para cada objeto

• La palabra clave **synchronized** para accesar un objeto llave

• Los métodos **wait()**, **notify()**, y **notifyAll()**, los cuales le permiten al objeto controlar los hilos cliente

La sección siguiente describe las llaves, código **synchronized** ,y los métodos **wait**Q, **notify**Q, y **notifyAll**() ,y muestra como estos pueden ser usados para hacer el código del hilo mas robusto.

Los Objetos llave y Sincronización

Cada objeto tiene una llave. En cualquier momento, la llave es controlada por, al menos , un único hilo. La llave controla el acceso a los objetos de

código sincronizado. Un hilo que quiere ejecutar un código sincronizado de un objeto debe primero tratar de adquirir la llave del objeto. Si la llave está disponible—lo que significa, si este no está ya controlado por otro hilo —entonces todo esta bien. Si la llave esta bajo el control de otro hilo, entonces el hilo trata de ir a buscar en el estado llave y solo estará listo cuando la llave este disponible. Cuando un hilo que posee una llave pasa fuera del código sincronizado, el hilo automáticamente abandona la llave . Todo esto chequeo de llaves y cambios de estados es hecho detrás de la escena; La única programación explícita que usted necesita es declarar el código sincronizado**(synchronized)**.

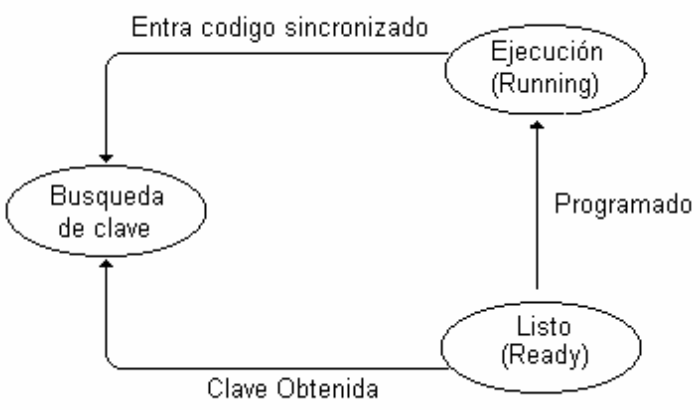
La figura 7.6 muestra el estado de búsqueda de llaves. Esta figura es el primer estado en nuestra expansión de los estados monitores, como se describe en la Figura 7.5.

Hay dos formas para marcar un código como sincronizado:

• Sincronizar un método entero colocando el modificador synchronized en la declaración del método. Para ejecutar el método, un hilo debe adquirir la llave del objeto que posee el método.

• Sincronizar un grupo de métodos encerrando las líneas deseadas del código con paréntesis ({})e insertando la expresión **synchronized**(someOb)fCt) antes de abrir el paréntesis.
Esta técnica le permite sincronizar el bloque sobre la llave de cualquier objeto, no necesariamente el objeto que posee el código.

La primera técnica esta muy lejos de ser común; la sincronización sobre otros objetos a los que poseen el código **synchronized** puede ser explícita. peligrosa
FIGURA 7.6
Los estados de búsqueda de llaves



El examen de certificación requieres que usted sepa como aplicar la segunda técnica, pero el examen no hace que usted piense en escenarios complicados de sincronización sobre objetos externos. La segunda técnica es discutida al final de este capitulo.
La sincronización hace fácil poner en orden algunos de los problemas con el mailbox

```
1.class Mailbox  {
2.  private boolean      request;
3.  private String       message;
4.
5.   public synchronized void storeMessage(String message)  {
6.     request = true;
7.     this. message = message;
8.    }
9.
10. public synchronized String retrieveMessage()  {
11.   request = false;
12.   return message;
13.   }
14. }
```

Ahora la bandera de solicitud y el mensaje string son privados , así ellos pueden solamente ser modificados por medio de los métodos **públicos** de la clase. Dado que storeMessage() y retrieveMessage() son **synchronized**, no hay peligro de que un hilo produzca mensajes que dañe la bandera y eche a perder cosas como hilos consumidores de mensajes o viceversa.

La clase Mailbox es hora segura para sus clientes, pero los clientes aun tienen problemas. Un cliente productor de mensajes debería solamente llamar storeMessage() cuando la solicitud de bandera este en falso; un cliente consumidor de mensajes debería solamente llamar retrieve-MessageQ cuando la solicitud de bandera este en verdadero. En la clase Consumer de la sección previa, los hilos consumidores principales verifican la solicitud de bandera cada 50 segundos. (Presumiblemente un hilo productor de mensajes debería hacer algo similar.) ahora la solicitud de bandera es privada, así usted debería encontrar otra forma.

Es posible traer a colación un numero de formas ágiles para los hilos que están continuamente verificando el correo, pero todas las aproximaciones es volver atrás. El correo llega a estar disponible o no disponible basado en los cambios de su propio estado.

El correo debería estar encargado del progreso de los clientes. Los métodos de Java **wait()** y **notify()** suministran los controles necesarios, como se vera en la próxima sección.

wait() y notify()

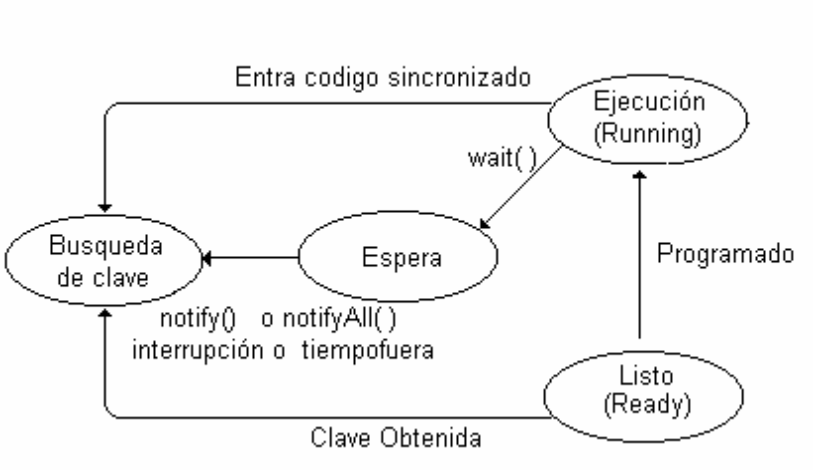
Los métodos **wait**() y **notify()** suministran una forma para que un objeto compartido coloque en pausa un hilo cuando este no disponible para aquel hilo, y le permita al hilo continuar cuando sea apropiado. Los hilos en si mismos no tiene nunca que chequear el estado de los objetos compartidos.

Un objeto que controla sus hilos clientes de esta manera es conocido como un monitor. En la estricta terminología de Java, un monitor es cualquier objeto que tiene algún código **sincronizado**. Para ser realmente útil, muchos monitores hacen uso de los métodos **wait**() y **notify()**. De esta manera la clase Mailbox es un monitor.

La Figura 7.7 muestra los estados de transición de **wait** () y **notify**O.

Ambos **wait**() y **notify()** tiene que ser llamados en un código **sincronizado** . Un hilo que llama a **wait**() suelta la CPU virtual; al mismo tiempo este suelta la llave. Este entra a un grupo de hilos en espera, el cual es administrado por el objeto cuyo método **wait**() fue llamado. Cada objeto tiene tal grupo. El código siguiente muestra como el método retrieveMessageO de la clase Mailbox podría ser modificado para comenzar a tener ventaja de la llamada a **wait**().

FIGURA 7.7
Estados Monitor




```
1.  public synchronized String  retrieveMessage()  {
2.     while (request == false)  {
3.  try{
4.    wait();
5. catch (InterruptedException e) { }
6.}
```

```
7. request = false;
8. return message;
9. }
```

Ahora considere que pasa cuando un hilo consumidor de mensajes llama este método. La llamada podría verse como esto:

```
myMailbox. retrieveMessage();
```

Cuando un hilo consumidor de mensajes llama este método, el hilo debería primero obtener la clave para myMaiIbox. Adquiriendo la clave podría pasar inmediatamente o este podría incurrir en una espera si cualquier otro hilo esta ejecuntado cualquier código **sincronizado** de myMaiIbox. De una u otra forma, eventualmente el hilo consumidor tiene la clave y comienza a ejecutarse en la línea 2. El código primero verifica la bandera de solicitud. Si la bandera no se ha colocado, entonces myMaiIbox no tiene mensajes que recuperar para el hilo. En este caso el método **wait**() es llamado en la línea 4 (este puede lanzar un InterruptedException,de esta manera el código try/catch es requerido, y el while verificará la condición). Cuando la línea 4 se ejecuta, el hilo consumidor suspende la ejecución; Esto también libera la clave para myMailbox y entra al grupo de hilos en espera administrados por myMailbox.

El hilo consumidor ha estado exitosamente evitando de dañar el monitor myMailbox. Desafortunadamente, Este esta introducido en el grupo de monitores de hilos en espera. Cuando el monitor cambia a un estado donde este puede suministrar el consumidor con algo que hacer, entonces algo tendrá que haber hecho para llevar el consumidor fuera del estado de espera. Esto se hace llamando a notify() cuando la bandera de solicitud del monitor se coloca a verdadero, lo cual solamente pasa en el método storeMessage() .El método storeMessage() se ve como sigue:

```
1. public synchronized void storeMessage(String message)  {
2.    this.message = message;
3.    request = true;
4.    notifyO;
5. }
```

En la línea 4, el código llama a **notify()** solo después de cambiar el estado del monitor. Lo que **notify()** hace es seleccionar uno de los hilos del grupo de monitores en espera y cambiar este al estado de búsqueda de clave. Eventualmentel el hilo obtendrá la clave de mailbox y proceder con la ejecución.

Ahora imagine un escenario completo. A hilo consumidor llama retrieve-Message() en un mailbox que no tiene mensajes. Este obtiene la clave y comienza a ejecutar el método. Este observa que la bandera de solicitud esta en falso, así este llama **wait**() y lo inserta al grupo de espera del correo. (En

este ejemplo sencillo, no hay otros hilos en el grupo). Dado que el consumidor ha llamado a **wait**(), este abandona la clave.

Después , un hilo productor de mensajes llama storeMessage() en el mismo correo. Este adquiere la clave, almacena su mensaje en una variable del monitor, y coloca la bandera de solicitud a verdadero. El produce entonces **notify**(). En este momento hay solo un hilo en el grupo de espera del monitor.

Así el consumidor consigue salir del estado de espera y va al estado de búsqueda de clave. Ahora el productor retorna del mensaje almacenado; Así el productor ha salido del código **sincronizado**, este abandona la clave del monitor. Después el consumidor pacientemente adquiere la clave y consigue ejecutarse; una vez esto pasa, este chequea la bandera de solicitud y (finalmente!) observa que hay un mensaje disponible. El consumidor retorna el mensaje; después de retornar este automáticamente suelta la clave.

Para resumir: Un consumidor trata de consumir nada, pero no hubo nada que consumir, entonces espera. El productor genera algo. De tal forma que hay algo que consumir por el consumidor, de esta forma el consumidor será notificado; una vez el monitor estuvo listo el monitor, el consumidor consumió el mensaje.

Como se muestra en la Figura 7.7 , un hilo que esta en espera tiene varias formas de salir de este estado donde no requiere ser notificado. Una versión de la llamada a **wait**() toma un argumento que especifica un tiempo fuera en milisegundos; si este expira, el hilo cambia al estado de búsqueda de clave, aun este no haya sido notificado.

No importa que versión de **wait**() es invocada, si el hilo que espera recibe un llamado de una interrupción este cambia inmediatamente al estado de busqueda de clave.
Este ejemplo protege el consumidor contra la posibilidad de que el monitor pueda estar vacío; la protección fue implementada con una llamada a **wait**() en retrieveMessage() y una llamada a **notify()** en storeMessage(). Una precaución similar debe tenerse en cuenta en caso que un hilo productor quiera producir dentro de un monitor que ya contenga un mensaje. Para ser mas robusto, storeMessage() necesita llamara **wait()**, y retrieveMessage() necesita llamar al **notify()**. La clase Mailbox completa se vería como esto :

```
1.class Mailbox {
2.private boolean  request;
3.private String     message;
4.
5.public synchronized void storeMessage(String message) {
6.while( request == true) { // No room for another message
7.try {
8.wait();
```

```
9.    } catch (InterruptedException e) { }
10. }
11. request = true;
12. this. message = message;
13.  notifyO;
14.  }
15.
16. public synchronized String retrieveMessage() {
17. while( request == false) { // No hay mensajes para recuperar
18. try {
19. waitO;
20.     } catch (InterruptedException e) { }
21. }
22.  request = false;
23.  notify();
24.  return message;
25.  }
26.  }
```

Sincronizando el código y llamando cuidadosamente a wait() y **notify()** los monitores tales como la clase MaiIbox pueden asegurar la propia interacción de hilos clientes y proteger los datos compartidos de daño.

Aquí están los principales puntos para recordar acerca de wait():

• La llamada al hilo abandona la CPU.

• La llamada al hilo abandona la clave.

• La llamada al hilo va dentro del grupo de monitores en espera.

Aquí están los principales puntos para recordar acerca de **notify()**:

• Un hilo consigue cambiar y estar fuera del grupo de monitores en espera y pasa al estado listo (Ready) .

• El hilo que fue notificado debe readquirir la clave del monitor antes que este pueda ser procesado.

Mas alla del Modelo Puro

El ejemplo del correo de las secciones previas ha sido muy simple son ejemplos de situaciones que involucran un productor y un consumidor. En situaciones reales no son siempre tan simples. Usted debe tener un monitor que tiene métodos que no producen puramente o consumen puramente. Todo lo que usted puede decir en general acerca de tales métodos es que ellos no pueden proceder a menos que el monitor este en un cierto estado y

ellos mismos pueden cambiar el estado del monitor de diversas formas que podrían ser de vital interés para los otro métodos

El método **notify**() no es preciso: Usted no puede especificar cual hilo va a ser notificado. En un escenario confuso tal como el descrito arriba , un hilo puede alterar el estado del monitor en una forma que es inútil para el hilo en particular que será anunciado. En tal caso, Los métodos del monitor tomarían dos precauciones:

• Siempre verifique el estado del monitor en un while antes de una declaración if.

• Después de cambiar el estado del monitor, llame notifyAll( ) antes que **notify()**.

La primera precaución significa que usted no debería hacer lo siguiente:

```
1. public synchronized void mixedUpMethod()  {
2. if (i<16    ||    f>4.3f    ||    message.equals("UH-OH")  {
3. try  {  wait();  }  catch  (InterruptedException e)  {  ]
4. }
5.
6. // Proceder de una forma que cambia el estado , y luego..
7. notifyO;
8. }
```

El peligro es que algunas veces un hilo podría ejecutar la prueba en la línea 2, entonces note que este es (por ejemplo) 15, y tiene que esperar .después otro , hilo puede cambiar el estado del monitor colocando i a -23444, y entonces llamar a **notify()**. Si el hilo original es el único que consigue ser anunciado, este continuará donde se detuvo, aun cuando el monitor no este en un estado donde este listo para mixedllpMethod().

La solución es cambiar mixedUpMethod() como sigue:

```
1. public synchronized void mixedUpMethod()  {
2. while  (i<16    ||    f>4.3f    ||    message.equalsC'UH-OH11)  {
3. try  {  wait();  }  catch (InterruptedException e)  {  }
4.       }
5.
6.  // Proceed in a way that changes state, and then...
7.  notifyAll();
8.  }
```

Los métodos **sincronizados** del otro monitor deberían ser modificados en 1 de una manera similar.

Ahora cuando un hilo en espera consigue ser anunciado , este no asume que el estado del monitor es aceptable. Este verifica de nuevo, en un ciclo while chequea en la línea 2. Si el estado aun no es conducente, El hilo espera de nuevo.

En la línea 7, Se ha hecho sus propias modificaciones al estado del monitor, El código llama a **notifyAll**(); esta llamada es como **notify()**, pero este cambia cada hilo en el grupo de monitores en espera al estado listo. Presumiblemente cada hilo llama a wait() dentro de un ciclo como en las líneas 2-4, así cada hilo conseguirá verificar el estado del monitor y esperar o proceder.

Usando un ciclo while para verificar el estado del monitor es una buena idea aun cuando usted este codificando un modelo puro de un productor o un consumidor, después de todo, usted nunca puede estar seguro que alguien no querrá tratar de adicionar un productor extra o un consumidor.

Formas extrañas para Sincronizar

Hay dos formas de sincronizar código que todavía no han sido explicadas. Ellas son poco comunes y generalmente no deberían ser usadas sin una razón obligatoria. Estas dos formas son

- Sincronizando la clave de un objeto diferente
- Sincronizando la clave de una clase

Esto fue brevemente mencionado en una sección anterior ("La clave del objeto y Sincronización") que usted puede sincronizar la clave de un objeto. Suponga, por ejemplo, que usted tiene la siguiente clase, la cual tiene un buen grado de creatividad.

```
1. class StrangeSync {
2. Rectangle rect = new Rectangledl,  13,  1100,  1300);
3. void doitQ  {
4.   int x =  504;
5.   int y = x / 3;
6.   rect.width -= x;
7.   rect.height -= y;
8.  }
9 }
```

Si usted adiciona la clave sincronizada en la línea 3, entonces un hilo que quiere ejecutar el método doit() de alguna instancia de StrangeSync debe primero adquirir la clave para la instancia. Que puede ser exactamente lo que usted quiere. Sin embargo, quizás usted solamente quiera sincronizar las líneas 6 y 7, y quizás usted quiera que un hilo intentar ejecutar estas líneas para sincronizar la clave de rect, antes que la clave del objeto en ejecución. La forma de hacer esto se muestra a continuación:

```
1.class StrangeSync {
2.  Rectangle rect = new Rectangle(ll, 13, 1100, 1300);
3.  void doitQ {
4.     int x = 504;
5.     int y = x / 3;
6.     synchronized(rect) {
7.       rect.width -= x;
8.       rect.height -= y;
9.      }
10.  }
11. }
```

El código de arriba sincroniza la clave de, algun objeto arbitrario (especificado en el paréntesis después de la palabra clave **sincronizada** en la línea 6), antes de sincronizar la clave del objeto actual. También , el código de arriba sincroniza solo dos líneas, en cambio del método entero.

Es difícil encontrar una buena razón para sincronizar un objeto arbitrario . Sin embargo, sincronizar solo un subconjunto de un método puede ser útil; algunas veces usted quiere obtener la clave tan pronto como sea posible, dado que los otros hilos pueden conseguir sus turnos tan pronto como sea posible. El compilador de Java insiste que cuando usted sincroniza una porción de un método (no todo el método), usted tiene que especificar un objeto en paréntesis después de la clave **sincronizada.** Si usted coloca esta en paréntesis , Entonces logra el objetivo: Usted ha **sincronizado** una porción de un método, con la clave usando la clave del objeto que es dueño del método.

De esta manera sus opciones son

• Para sincronizar un método entero, usando la clave del objeto que es dueño del método. Para hacer esto ,coloque la palabra clave **sincronizada** en la declaración del método.

• Para sincronizar parte de un método, usando la clave de un objeto arbitrario. Para hacer esto, coloque corchetes alrededor del código que será **sincronizado**, precedido por **sincronizado**(theArbitraryObject).

Para sincronizar parte de un método, use la clave del objeto que es dueño del método. Para hacer esto, coloque corchetes alrededor del código que será **sincronizado**, precedido por **synchronized**(this).
Las clases, así como los objetos, tienen claves. Una clave de una clase es usada para sincronizar los métodos estáticos de la clase. Los objetivos de la certificación no hacen gran énfasis sobre las claves de las clases.

Resumen del capitulo

La programación de hilos en Java pueden ser con preferencia o con divisiones de tiempo, dependiendo del diseño del JVM. No importa cual diseño sea usado, un hilo llega a ser elegible para ser ejecutado ("Ready") cuando su método start() es invocado. Cuando un hilo comienza la ejecución, el programador llama el método **run()** del hilo objetivo (si hay un objetivo) o el método **run()** del hilo en si mismo (si no hay objetivo).

En el curso de la ejecución, un hilo puede llegar a ser no elegible por un numero de razones : Un hilo puede suspender, o dormir, o bloquear, o esperar. A su debido tiempo (una esperanza!), las condiciones cambiaran dado que el hilo una vez mas llega a ser elegible para la ejecución; entonces el hilo entra al estado listo y eventualmente puede ejecutarse. ,

Cuando un hilo retorna de su método(), este entra al estado muerto y no puede ser reinicializado.

Usted puede encontrar la siguiente lista como un resumen útil de los hilos de Java.

Implementaciones del programador:
• Preferencia

• Divisiones de tiempo en la construcción de un hilo:

- **new** Thread(): sin objetivo;el método **run()** del propio hilo se ejecuta

- **new Thread** (**Runnable** target): el método **run()** del hilo se ejecuta

Estados de no ejecución:

•Suspendido(Suspended): causado por suspend(), espera a resume()

•Dormido (Sleeping): causado by sleep(), espera por pausa

• Bloqueado (Blocked): causado por varias llamadas a I/O o por una falla para conseguir una clave de un monitor, espera por una I/O o por la clave del monito

- Espera (**Wait**ing): causado por **wait**(), **wait**s por **notify()** o **notifyAll**()

- Muerto: causado por stop() o retornando de **run()**

Test Yourself

1. Cual de las siguientes declaraciones abajo respecto al siguiente código son verdaderas?

```
1.  class Greebo extends java.util.Vector implements Runnable {
2.          public void  run( String message ) {
3.                  System.out.println( "in run() method: " + message );
4.          }
5.  }
6.
7.  class GreeboTest {
8.          public static void main( String[] args ) {
9.                  Greebo g = new Greebo();
10.                 Thread t = new Thread( g );
11.                 t.start();
12.         }
13. }
```

A. Habrá un error del compilador, porque la clase Greebo no implementa la interface **Runnable**.
B. Habrá un error del compilador en la línea 10, porque no se pueden pasar parámetros al constructor de un hilo.
C. El código compilará correctamente pero lanzará una excepción en al línea 10.
D. El código compilará correctamente pero lanzará una excepción en línea 11.
E. El código compilará correctamente y se ejecutará sin lanzar ninguna excepción.

2. Cual declaración abajo es siempre verdadera acerca de la siguiente aplicación?

```
1.  class HiPri extends Thread {
2.          HiPri() {
3.                  setPriority( 10 );
4.          }
5.
6.          public void run() {
7.                  System.out.println( " Another thread starting up." );
8.                  while ( true ) { }
9.          }
10.
11.         public static void main( String[] args ) {
12.                 HiPri   hp1 = new HiPri();
```

```
13.                 HiPri   hp2 = new HiPri();
14.                 HiPri   hp3 = new HiPri();
15.                 hp1.start();
16.                 hp2.start();
17.                 hp3.start();
18.         }
19. }
```

A. Cuando la aplicación corre, el hilo hp1 se ejecutará; los hilos hp2 y hp3 nunca conseguirán la CPU.
B. Cuando la aplicación corre , todos los hilos (hp1, hp2, y hp3) conseguirán ejecutarse, tomando intervalos de tiempo en la CPU.
C. A o B serán verdadera, dependiendo de la plataforma que corra.

3.Un hilo quiere que un segundo hilo no sea elegible para la ejecución. Para Hacer esto, el primer hilo puede llamar el método yield() sobre el segundo hilo.

A. Verdadero
B. Falso

4. El método **run()** de un hilo incluye las siguientes líneas de código:

```
1.  try  {
2.          sleep( 100 );
3.  }  catch ( InterruptedException e )  { }
```

Asumiendo que el hilo no es interrumpido, cual de las siguientes declaraciones es correcta?

A. El código no compilará, porque las excepciones no pueden ser atrapadas en el método **run()** de un hilo.
B. En la línea 2,el hilo parará de correr. La ejecución comenzará de nuevo al menos en 100 milisegundos.
C. En la línea 2, el hilo parará de correr . Este comenzará de nuevo a correr en exactamente 100 milisegundos.
D. En la línea 2, el hilo parará de correr . Este comenzará de nuevo a correr en algún momento después que hayan transcurrido 100 milisegundos .

5. Un monitor llamado mon tiene 10 hilos en su grupo de espera; todos estos hilos en espera tienen la misma prioridad. Uno de los hilos es thr1. Como se puede anunciar thr1 dado que este solo cambia del estado en espera al estado listo?

A. Ejecute **notify**(thr1); Desde adentro del código **synchronized** del monitor.
B. Ejecute mon.**notify**(thr1); desde el código **synchronized** de cualquier objeto.

C. Ejecute thr1.**notify**();desde el código **synchronized** de cualquier objeto.
D. Ejecute thr1.**notify**();desde cualquier código **synchronized** o no, de cualquier objeto.
E. Usted no puede especificar cual hilo será anunciado.

6. Cual de las declaraciones abajo esta verdaderamente relacionada con la siguiente aplicación?

```
1.  class TestThread2 extends Thread  |
2.          public void run() {
3.                  System.out.println( "Starting" );
4.                  yield();
5.                  resume();
6.                  System.out.println( "Done" );
7.          }
8.
9.          public static void main( String[] args ) {
10.                 TestThread2 tt = new TestThread2();
11.                 tt.start();
12.         }
13. }
```

A. La compilación fallará en la línea 4, porque yied() debe ser llamado en un código **sincronizado**.
B. La compilación fallará en la línea 5, porque resume() debe ser llamado en código **sincronizado**.
C. La compilación será exitosa. Durante la ejecución, nada se imprimirá.
D. La compilación será exitosa . Durante la ejecución, solamente una línea de la salida (Starting) se imprimirá.
E. La compilación será exitosa . Durante la ejecución, ambas líneas de la salida (Starting and Done) serán impresas.

7. Si usted trata de compilar y ejecutar la aplicación listada abajo, imprimirá el mensaje In xxx?

```
1.  class TestThread3 extends Thread {
2.          public void  run()   {
3.                  System.out.println( "Running" );
4.                  System.out.println( "Done" );
5.          }
6.
7.          private void xxx() {
8.                  System.out.println( "In xxx" );
9.          }
10.
11.         public static void main( String[] args ) {
12.                 TestThread3 ttt = new TestThread3();
13.                 ttt.xxx();
14.                 ttt.start();
```

```
15.         }
16. }
```

8. Un monitor en Java debe heredar de **Thread** o implementar **Runnable**.
A. Verdadero
B. Falso