**LENGUAJES DE PROGRAMACIÓN**

**(Sesión 6)**

**3. PROGRAMACIÓN CONCURRENTE**

3.3. Mecanismos de sincronización

3.4. Concurrencia en otros lenguajes de programación

Objetivo: Establecer el concepto de sincronización y comprender cómo trabaja la concurrencia en otros lenguajes de programación.

Métodos de Instancia

Aquí no están recogidos todos los métodos de la clase Thread, sino solamente los más interesantes, porque los demás corresponden a áreas en donde el estándar de Java no está completo, y puede que se queden obsoletos en la próxima versión del JDK, por ello, si se desea completar la información que aquí se expone se ha de recurrir a la documentación del interfaz de programación de aplicación (API) del JDK.

start()

Este método indica al intérprete de Java que cree un contexto del hilo del sistema y comience a ejecutarlo. A continuación, el método run() de este hilo será invocado en el nuevo contexto del hilo. Hay que tener precaución de no llamar al método start() más de una vez sobre un hilo determinado.

run()

El método run() constituye el cuerpo de un hilo en ejecución. Este es el único método del interfaz Runnable. Es llamado por el método start() después de que el hilo apropiado del sistema se haya inicializado. Siempre que el método run() devuelva el control, el hilo actual se detendrá.

stop()

Este método provoca que el hilo se detenga de manera inmediata. A menudo constituye una manera brusca de detener un hilo, especialmente si este método se ejecuta sobre el hilo en curso. En tal caso, la línea inmediatamente posterior a la llamada al método stop() no llega a ejecutarse jamás, pues el contexto del hilo muere antes de que stop() devuelva el

control. Una forma más elegante de detener un hilo es utilizar alguna variable que ocasione que el método run() termine de manera ordenada. En realidad, nunca se debería recurrir al uso de este método.

suspend()

El método suspend() es distinto de stop(). suspend() toma el hilo y provoca que se detenga su ejecución sin destruir el hilo de sistema subyacente, ni el estado del hilo anteriormente en ejecución. Si la ejecución de un hilo se suspende, puede llamarse a resume() sobre el mismo hilo para lograr que vuelva a ejecutarse de nuevo.

resume()

El método resume() se utiliza para revivir un hilo suspendido. No hay garantías de que el hilo comience a ejecutarse inmediatamente, ya que puede haber un hilo de mayor prioridad en ejecución actualmente, pero resume() ocasiona que el hilo vuelva a ser un candidato a ser ejecutado.

setPriority( int )

El método setPriority() asigna al hilo la prioridad indicada por el valor pasado como parámetro.

Hay bastantes constantes predefinidas para la prioridad, definidas en la clase Thread, tales como MIN_PRIORITY, NORM_PRIORITY y MAX_PRIORITY, que toman los valores 1, 5 y 10, respectivamente. Como guía aproximada de utilización, se puede establecer que la mayor parte de los procesos a nivel de usuario deberían tomar una prioridad en torno a NORM_PRIORITY. Las tareas en segundo plano, como una entrada/salida a red o el nuevo dibujo de la pantalla, deberían tener una prioridad cercana a MIN_PRIORITY. Con las tareas a las que se fije la máxima prioridad, en torno a MAX_PRIORITY, hay que ser especialmente cuidadosos, porque si no se hacen llamadas a sleep() o yield(), se puede provocar que el intérprete Java quede totalmente fuera de control.

getPriority()

Este método devuelve la prioridad del hilo de ejecución en curso, que es un valor comprendido entre uno y diez.

setName( String )

Este método permite identificar al hilo con un nombre nenmónico. De esta manera se facilita la depuración de programas multihilo. El nombre mnemónico aparecerá en todas

las líneas de trazado que se muestran cada vez que el intérprete Java imprime excepciones no capturadas.

getName()

Este método devuelve el valor actual, de tipo cadena, asignado como nombre al hilo en ejecución mediante setName().

Creación de un Thread

Hay dos modos de conseguir hilos de ejecución (threads) en Java. Una es implementando el interfaz Runnable, la otra es extender la clase Thread.

La implementación del interfaz Runnable es la forma habitual de crear hilos. Los interfaces proporcionan al programador una forma de agrupar el trabajo de infraestructura de una clase. Se utilizan para diseñar los requerimientos comunes al conjunto de clases a implementar. El interfaz define el trabajo y la clase, o clases, que implementan el interfaz para realizar ese trabajo. Los diferentes grupos de clases que implementen el interfaz tendrán que seguir las mismas reglas de funcionamiento.

Hay unas cuantas diferencias entre interfaz y clase, que ya son conocidas y aquí solamente se resumen. Primero, un interfaz solamente puede contener métodos abstractos y/o variables estáticas y finales (constantes). Las clases, por otro lado, pueden implementar métodos y contener variables que no sean constantes. Segundo, un interfaz no puede implementar cualquier método. Una clase que implemente un interfaz debe implementar todos los métodos definidos en ese interfaz.

Un interfaz tiene la posibilidad de poder extenderse de otros interfaces y, al contrario que las clases, puede extenderse de múltiples interfaces.

Además, un interfaz no puede ser instanciado con el operador new; por ejemplo, la siguiente sentencia no está permitida:

```
Runnable a = new Runnable(); // No se permite
```

El primer método de crear un hilo de ejecución es simplemente extender la clase Thread:

```
class MiThread extends Thread {
public void run() {
. . .
}
```

El ejemplo anterior crea una nueva clase MiThread que extiende la clase Thread y

sobreescribe el método Thread.run() por su propia implementación. El método run() es donde se realizará todo el trabajo de la clase. Extendiendo la clase Thread, se pueden heredar los métodos y variables de la clase padre. En este caso, solamente se puede extender o derivar una vez de la clase padre. Esta limitación de Java puede ser superada a través de la implementación de Runnable:

```
public class MiThread implements Runnable {
Thread t;
public void run() {
// Ejecución del thread una vez creado
}
}
```

En este caso necesitamos crear una instancia de Thread antes de que el sistema pueda ejecutar el proceso como un hilo. Además, el método abstracto run() está definido en el interfaz Runnable y tiene que ser implementado. La única diferencia entre los dos métodos es que este último es mucho más flexible. En el ejemplo anterior, todavía está la oportunidad de extender la clase MiThread, si fuese necesario. La mayoría de las clases creadas que necesiten ejecutarse como un hilo, implementarán el interfaz Runnable, ya que probablemente extenderán alguna de su funcionalidad a otras clases. No pensar que el interfaz Runnable está haciendo alguna cosa cuando la tarea se está ejecutando. Solamente contiene métodos abstractos, con lo cual es una clase para dar idea sobre el diseño de la clase Thread. De hecho, si se observan los fuentes de Java, se puede comprobar que solamente contiene un método abstracto:

```
package java.lang;
public interface Runnable {
public abstract void run() ;
}
```

Y esto es todo lo que hay sobre el interfaz Runnable. Como se ve, un interfaz sólo proporciona un diseño para las clases que vayan a ser implementadas.

En el caso de Runnable, fuerza a la definición del método run(), por lo tanto, la mayor parte del trabajo se hace en la clase Thread. Un vistazo un poco más profundo a la definición de la clase Thread da idea de lo que realmente está pasando:

```
public class Thread implements Runnable {
...
public void run() {
if( tarea != null )
tarea.run() ;
}
}
...
}
```

De este trocito de código se desprende que la clase Thread también implemente el interfaz Runnable. tarea.run() se asegura de que la clase con que trabaja (la clase que va a ejecutarse como un hilo) no sea nula y ejecuta el método run() de esa clase. Cuando esto suceda, el método run() de la clase hará que corra como un hilo.

A continuación se presenta un ejemplo, java1001.java, que implementa el interfaz Runnable para crear un programa multihilo.

```
class java1001 {
static public void main( String args[] ) {
// Se instancian dos nuevos objetos Thread
Thread hiloA = new Thread( new MiHilo(),"hiloA" );
Thread hiloB = new Thread( new MiHilo(),"hiloB" );
// Se arrancan los dos hilos, para que comiencen su ejecución
hiloA.start();
hiloB.start();
// Aquí se retrasa la ejecución un segundo y se captura la
// posible excepción que genera el método, aunque no se hace
// nada en el caso de que se produzca
try {
Thread.currentThread().sleep( 1000 );
}catch( InterruptedException e ){}
// Presenta información acerca del Thread o hilo principal
// del programa
```

```
System.out.println( Thread.currentThread() );
// Se detiene la ejecución de los dos hilos
hiloA.stop();
hiloB.stop();
}
}
class NoHaceNada {
// Esta clase existe solamente para que sea heredada por la clase
// MiHilo, para evitar que esta clase sea capaz de heredar la clase
// Thread, y se pueda implementar el interfaz Runnable en su
// lugar
}
class MiHilo extends NoHaceNada implements Runnable {
public void run() {
// Presenta en pantalla información sobre este hilo en particular
System.out.println( Thread.currentThread() );
}
}
```

Como se puede observar, el programa define una clase MiHilo que extiende a la clase NoHaceNada e implementa el interfaz Runnable. Se redefine el método run() en la clase MiHilo para presentar información sobre el hilo.

La única razón de extender la clase NoHaceNada es proporcionar un ejemplo de situación en que haya que extender alguna otra clase, además de implementar el interfaz.

En el ejemplo java1002.java muestra el mismo programa básicamente, pero en este caso extendiendo la clase Thread, en lugar de implementar el interfaz Runnable para crear el programa multihilo.

```
class java1002 {
static public void main( String args[] ) {
// Se instancian dos nuevos objetos Thread
Thread hiloA = new Thread( new MiHilo(),"hiloA" );
Thread hiloB = new Thread( new MiHilo(),"hiloB" );
```

```
// Se arrancan los dos hilos, para que comiencen su ejecución
hiloA.start();
hiloB.start();
// Aquí se retrasa la ejecución un segundo y se captura la
// posible excepción que genera el método, aunque no se hace
// nada en el caso de que se produzca
try {
Thread.currentThread().sleep( 1000 );
}catch( InterruptedException e ){}
// Presenta información acerca del Thread o hilo principal
// del programa
System.out.println( Thread.currentThread() );
// Se detiene la ejecución de los dos hilos
hiloA.stop();
hiloB.stop();
}
}
class MiHilo extends Thread {
public void run() {
// Presenta en pantalla información sobre este hilo en particular
System.out.println( Thread.currentThread() );
}
}
```

En ese caso, la nueva clase MiHilo extiende la clase Thread y no implementa el interfaz Runnable directamente (la clase Thread implementa el interfaz Runnable, por lo que indirectamente MiHilo también está implementando ese interfaz). El resto del programa es similar al anterior.

Y todavía se puede presentar un ejemplo más simple, utilizando un constructor de la clase Thread que no necesita parámetros, tal como se presenta en el ejemplo java1003.java. En los ejemplos anteriores, el constructor utilizado para Thread necesitaba dos parámetros, el primero un objeto de cualquier clase que implemente el interfaz Runnable y el segundo

una cadena que indica el nombre del hilo (este nombre es independiente del nombre de la variable que referencia al objeto Thread).

```
class java1003 {
static public void main( String args[] ) {
// Se instancian dos nuevos objetos Thread
Thread hiloA = new MiHilo();
Thread hiloB = new MiHilo();
// Se arrancan los dos hilos, para que comiencen su ejecución
hiloA.start();
hiloB.start();
// Aquí se retrasa la ejecución un segundo y se captura la
// posible excepción que genera el método, aunque no se hace
// nada en el caso de que se produzca
try {
Thread.currentThread().sleep( 1000 );
}catch( InterruptedException e ){}
// Presenta información acerca del Thread o hilo principal
// del programa
System.out.println( Thread.currentThread() );
// Se detiene la ejecución de los dos hilos
hiloA.stop();
hiloB.stop();
}
}
class MiHilo extends Thread {
public void run() {
// Presenta en pantalla información sobre este hilo en particular
System.out.println( Thread.currentThread() );
}
}
```

Las sentencias en este ejemplo para instanciar objetos Thread, son mucho menos

complejas, siendo el programa, en esencia, el mismo de los ejemplos anteriores.

Arranque de un Thread

Las aplicaciones ejecutan main() tras arrancar. Esta es la razón de que main() sea el lugar natural para crear y arrancar otros hilos. La línea de código:

t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );

crea un nuevo hilo de ejecución. Los dos argumentos pasados representan el nombre del hilo y el tiempo que se desea que espere antes de imprimir el mensaje.

Al tener control directo sobre los hilos, hay que arrancarlos explícitamente. En el ejemplo con:

t1.start();

start(), en realidad es un método oculto en el hilo de ejecución que llama a run().

Manipulación de un Thread

Si todo fue bien en la creación del hilo, t1 debería contener un thread válido, que controlaremos en el método run(). Una vez dentro de run(), se pueden comenzar las sentencias de ejecución como en otros programas. run() sirve como rutina main() para los hilos; cuando run() termina, también lo hace el hilo. Todo lo que se quiera que haga el hilo de ejecución ha de estar dentro de run(), por eso cuando se dice que un método es Runnable, es obligatorio escribir un método run().

En este ejemplo, se intenta inmediatamente esperar durante una cantidad de tiempo aleatoria (pasada a través del constructor):

sleep( retardo );

El método sleep() simplemente le dice al hilo de ejecución que duerma durante los milisegundos especificados. Se debería utilizar sleep() cuando se pretenda retrasar la ejecución del hilo. sleep() no consume recursos del sistema mientras el hilo duerme. De esta forma otros hilos pueden seguir funcionando. Una vez hecho el retardo, se imprime el mensaje "Hola Mundo!" con el nombre del hilo y el retardo.

Suspensión de un Thread

Puede resultar útil suspender la ejecución de un hilo sin marcar un límite de tiempo. Si, por ejemplo, está construyendo un applet con un hilo de animación, seguramente se querrá permitir al usuario la opción de detener la animación hasta que quiera continuar.

No se trata de terminar la animación, sino desactivarla. Para este tipo de control de los

hilos de ejecución se puede utilizar el método suspend().

t1.suspend();

Este método no detiene la ejecución permanentemente. El hilo es suspendido indefinidamente y para volver a activarlo de nuevo se necesita realizar una invocación al método resume():

t1.resume();

Parada de un Thread

El último elemento de control que se necesita sobre los hilos de ejecución es el método stop(). Se utiliza para terminar la ejecución de un hilo:

t1.stop();

Esta llamada no destruye el hilo, sino que detiene su ejecución. La ejecución no se puede reanudar ya con t1.start(). Cuando se desasignen las variables que se usan en el hilo, el objeto Thread (creado con new) quedará marcado para eliminarlo y el garbage collector se encargará de liberar la memoria que utilizaba.

En el ejemplo, no se necesita detener explícitamente el hilo de ejecución. Simplemente se le deja terminar. Los programas más complejos necesitarán un control sobre cada uno de los hilos que lancen, el método stop() puede utilizarse en esas situaciones.

Si se necesita, se puede comprobar si un hilo está vivo o no; considerando vivo un hilo que ha comenzado y no ha sido detenido.

t1.isAlive();

Este método devolverá true en caso de que el hilo t1 esté vivo, es decir, ya se haya llamado a su método run() y no haya sido parado con un stop() ni haya terminado el método run() en su ejecución.

En el ejemplo no hay problemas de realizar una parada incondicional, al estar todos los hilos vivos. Pero si a un hilo de ejecución, que puede no estar vivo, se le invoca su método stop(), se generará una excepción. En este caso, en los que el estado del hilo no puede conocerse de antemano es donde se requiere el uso del método isAlive().

Grupos de Hilos

Todo hilo de ejecución en Java debe formar parte de un grupo. La clase ThreadGroup define e implementa la capacidad de un grupo de hilos.

Los grupos de hilos permiten que sea posible recoger varios hilos de ejecución en un solo

objeto y manipularlo como un grupo, en vez de individualmente. Por ejemplo, se pueden regenerar los hilos de un grupo mediante una sola sentencia.

Cuando se crea un nuevo hilo, se coloca en un grupo, bien indicándolo explícitamente, o bien dejando que el sistema lo coloque en el grupo por defecto. Una vez creado el hilo y asignado a un grupo, ya no se podrá cambiar a otro grupo.

Si no se especifica un grupo en el constructor, el sistema coloca el hilo en el mismo grupo en que se encuentre el hilo de ejecución que lo haya creado, y si no se especifica en grupo para ninguno de los hilos, entonces todos serán miembros del grupo "main", que es creado por el sistema cuando arranca la aplicación Java.

En la ejecución de los ejemplos de esta sección, se ha podido observar la circunstancia anterior. Por ejemplo, el resultado en pantalla de uno de esos ejemplos es el que se reproduce a continuación:

```
% java java1002
Thread[hiloA,5,main]
Thread[hiloB,5,main]
Thread[main,5,main]
```

Como resultado de la ejecución de sentencias del tipo:

```
System.out.println( Thread.currentThread() );
```

Para presentar la información sobre el hilo de ejecución. Se puede observar que aparece el nombre del hilo, su prioridad y el nombre del grupo en que se encuentra englobado.

La clase Thread proporciona constructores en los que se puede especificar el grupo del hilo que se esta creando en el mismo momento de instanciarlo, y también métodos como setThreadGroup(), que permiten determinar el grupo en que se encuentra un hilo de ejecución.

Arrancar y Parar Threads

Ahora que ya se ha visto por encima como se arrancan, paran, manipulan y agrupan los hilos de ejecución, el ejemplo un poco más gráfico, java1004.java, implementa un contador.

El programa arranca un contador en 0 y lo incrementa, presentando su salida tanto en la pantalla gráfica como en la consola. Una primera ojeada al código puede dar la impresión de que el programa empezará a contar y presentará cada número, pero no es así. Una

revisión más profunda del flujo de ejecución del applet, revelará su verdadera identidad. En este caso, la clase java1004 está forzada a implementar Runnable sobre la clase Applet que extiende. Como en todos los applets, el método init() es el primero que se ejecuta. En init(), la variable contador se inicializa a cero y se crea una nueva instancia de la clase Thread. Pasándole this al constructor de Thread, el nuevo hilo ya conocerá al objeto que va a correr. En este caso this es una referencia a java1004.

Después de que se haya creado el hilo, necesitamos arrancarlo. La llamada a start(), llamará a su vez al método run() de la clase, es decir, a java1004.run(). La llamada a start() retornará con éxito y el hilo comenzará a ejecutarse en ese instante. Observar que el método run() es un bucle infinito. Es infinito porque una vez que se sale de él, la ejecución del hilo se detiene. En este método se incrementará la variable contador, se duerme 10 milisegundos y envía una petición de refresco del nuevo valor al applet.

Es muy importante dormirse en algún lugar del hilo, porque sino, el hilo consumirá todo el tiempo de la CPU para su proceso y no permitirá que entren métodos de otros hilos a ejecutarse. Otra forma de detener la ejecución del hilo sería hacer una llamada al método stop(). En el contador, el hilo se detiene cuando se pulsa el ratón mientras el cursor se encuentre sobre el applet.

Dependiendo de la velocidad del ordenador, se presentarán los números consecutivos o no, porque el incremento de la variable contador es independiente del refresco en pantalla. El applet no se refresca a cada petición que se le hace, sino que el sistema operativo encolará las peticiones y las que sean sucesivas las convertirá en un único refresco. Así, mientras los refrescos se van encolando, la variable contador se estará todavía incrementando, pero no se visualiza en pantalla.

El uso y la conveniencia de utilización del método stop() es un poco dudoso y algo que quizá debería evitarse, porque puede haber objetos que dependan de la ejecución de varios hilos, y si se detiene uno de ellos, puede que el objeto en cuestión estuviese en un estado no demasiado consistente, y si se le mata el hilo de control puede que definitivamente ese objeto se dañe. Una solución alternativa es el uso de una variable de control que permita saber si el hilo se encuentra en ejecución o no, por ello, en el ejemplo se utiliza la variable miThread que controla cuando el hilo está en ejecución o parado.

La clase anidada ProcesoRaton es la que se encarga de implementar un objeto receptor de

los eventos de ratón, para detectar cuando el usuario pulsa alguno de los botones sobre la zona de influencia del applet.

Suspender y Reanudar Threads

Una vez que se para un hilo de ejecución, ya no se puede rearrancar con el comando start(), debido a que stop() concluirá la ejecución del hilo. Por ello, en ves de parar el hilo, lo que se puede hacer es dormirlo, llamando al método sleep(). El hilo estará suspendido un cierto tiempo y luego reanudará su ejecución cuando el límite fijado se alcance. Pero esto no es útil cuando se necesite que el hilo reanude su ejecución ante la presencia de ciertos eventos. En estos casos, el método suspend() permite que cese la ejecución del hilo y el método resume() permite que un método suspendido reanude su ejecución.

En la versión modificada del ejemplo anterior, java1005.java, se modifica el applet para que utilice los métodos suspend() y resume():

El uso de suspend() es crítico en ocasiones, sobre todo cuando el hilo que se va a suspender está utilizando recursos del sistema, porque en el momento de la suspensión los va a bloquear, y esos recursos seguirán bloqueados hasta que no se reanude la ejecución del hilo con resume(). Por ello, deben utilizarse métodos alternativos a estos, por ejemplo, implementando el uso de variables de control que vigiles periódicamente el estado en que se encuentra el hilo actual y obren el consecuencia.

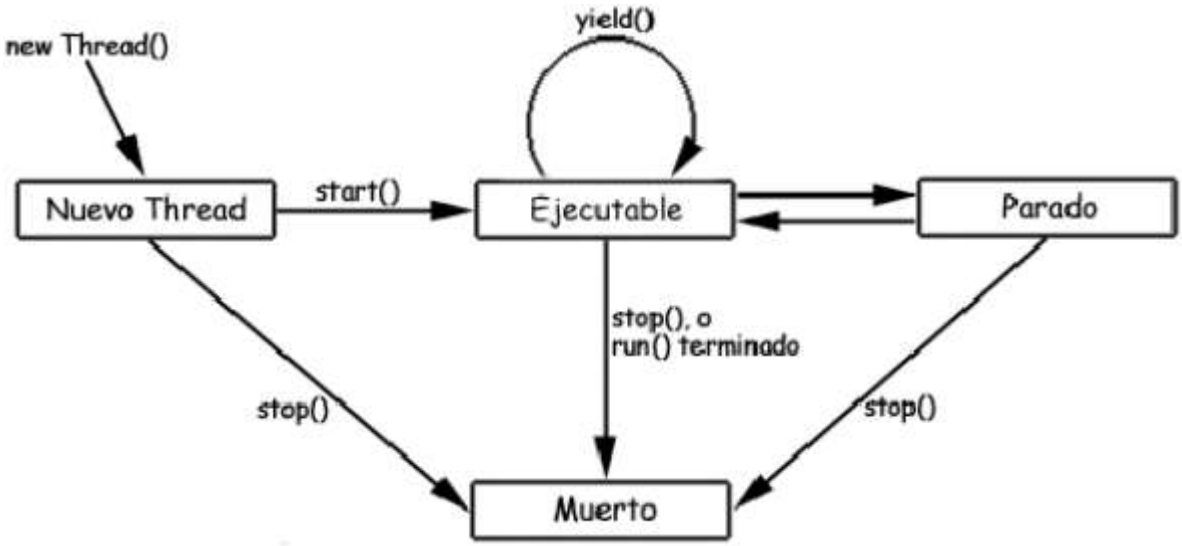
```
public class java1005 extends Applet implements Runnable {
...
class ProcesoRaton extends MouseAdapter {
boolean suspendido;
public void mousePressed( MouseEvent evt ) {
if( suspendido )
t.resume();
else
t.suspend();
suspendido = !suspendido;
}
}
...
```

Para controlar el estado del applet, se ha modificado el funcionamiento del objeto Listener que recibe los eventos del ratón, en donde se ha introducido la variable suspendido. Diferenciar los distintos estados de ejecución del applet es importante porque algunos métodos pueden generar excepciones si se llaman desde un estado erróneo. Por ejemplo, si el applet ha sido arrancado y se detiene con stop(), si se intenta ejecutar el método start(), se generará una excepción IllegalThreadStateException.

Aquí podemos poner de nuevo en cuarentena la idoneidad del uso de estos métodos para el control del estado del hilo de ejecución, tanto por lo comentado del posible bloqueo de recursos vitales del sistema, como porque se puede generar un punto muerto en el sistema si el hilo de ejecución que va a intentar revivir el hilo suspendido necesita del recurso bloqueado. Por ello, es más seguro el uso de una variable de control como suspendido, de tal forma que sea ella quien controle el estado del hilo y utilizar el método notify() para indicar cuando el hilo vuelve a la vida.

Estados de un Hilo de Ejecución

Durante el ciclo de vida de un hilo, éste se puede encontrar en diferentes estados. La figura siguiente muestra estos estados y los métodos que provocan el paso de un estado a otro. Este diagrama no es una máquina de estados finita, pero es lo que más se aproxima al funcionamiento real de un hilo.



Nuevo Thread

La siguiente sentencia crea un nuevo hilo de ejecución pero no lo arranca, lo deja en el estado de Nuevo Thread:

```
Thread MiThread = new MiClaseThread();
Thread MiThread = new Thread( new UnaClaseThread,"hiloA" );
```

Cuando un hilo está en este estado, es simplemente un objeto Thread vacío. El sistema no ha destinado ningún recurso para él. Desde este estado solamente puede arrancarse llamando al método start(), o detenerse definitivamente, llamando al método stop(); la llamada a cualquier otro método carece de sentido y lo único que provocará será la generación de una excepción de tipo IllegalThreadStateException.

Ejecutable

Ahora obsérvense las dos líneas de código que se presentan a continuación:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
```

La llamada al método start() creará los recursos del sistema necesarios para que el hilo puede ejecutarse, lo incorpora a la lista de procesos disponibles para ejecución del sistema y llama al método run() del hilo de ejecución. En este momento se encuentra en el estado Ejecutable del diagrama. Y este estado es Ejecutable y no En Ejecución, porque cuando el hilo está aquí no esta corriendo.

Muchos ordenadores tienen solamente un procesador lo que hace imposible que todos los hilos estén corriendo al mismo tiempo. Java implementa un tipo de scheduling o lista de procesos, que permite que el procesador sea compartido entre todos los procesos o hilos que se encuentran en la lista. Sin embargo, para el propósito que aquí se persigue, y en la mayoría de los casos, se puede considerar que este estado es realmente un estado En Ejecución, porque la impresión que produce ante el usuario es que todos los procesos se ejecutan al mismo tiempo.

Cuando el hilo se encuentra en este estado, todas las instrucciones de código que se

encuentren dentro del bloque declarado para el método run(), se ejecutarán secuencialmente.

Parado

El hilo de ejecución entra en estado Parado cuando alguien llama al método suspend(), cuando se llama al método sleep(), cuando el hilo está bloqueado en un proceso de entrada/salida o cuando el hilo utiliza su método wait() para esperar a que se cumpla una determinada condición. Cuando ocurra cualquiera de las cuatro cosas anteriores, el hilo estará Parado.

Por ejemplo, en el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
;
}
```

la línea de código que llama al método sleep():

```
MiThread.sleep( 10000 );
```

hace que el hilo se duerma durante 10 segundos. Durante ese tiempo, incluso aunque el procesador estuviese totalmente libre, MiThread no correría. Después de esos 10 segundos. MiThread volvería a estar en estado Ejecutable y ahora sí que el procesador podría hacerle caso cuando se encuentre disponible.

Para cada una de los cuatro modos de entrada en estado Parado, hay una forma específica de volver a estado Ejecutable. Cada forma de recuperar ese estado es exclusiva; por ejemplo, si el hilo ha sido puesto a dormir, una vez transcurridos los milisegundos que se especifiquen, él solo se despierta y vuelve a estar en estado Ejecutable. Llamar al método resume() mientras esté el hilo durmiendo no serviría para nada.

Los métodos de recuperación del estado Ejecutable, en función de la forma de llegar al estado Parado del hilo, son los siguientes:

- Si un hilo está dormido, pasado el lapso de tiempo
- Si un hilo de ejecución está suspendido, después de una llamada a su método resume()

 Si un hilo está bloqueado en una entrada/salida, una vez que el comando de entrada/salida concluya su ejecución

 Si un hilo está esperando por una condición, cada vez que la variable que controla esa condición varíe debe llamarse al método notify() o notifyAll()

Muerto

Un hilo de ejecución se puede morir de dos formas: por causas naturales o porque lo maten (con stop()). Un hilo muere normalmente cuando concluye de forma habitual su método run(). Por ejemplo, en el siguiente trozo de código, el bucle while es un bucle finito -realiza la iteración 20 veces y termina-:

```
public void run() {
int i=0;
while( i < 20 ) {
i++;
System.out.println( "i = "+i );
}
}
```

Un hilo que contenga a este método run(), morirá naturalmente después de que se complete el bucle y run() concluya.

También se puede matar en cualquier momento un hilo, invocando a su método stop(). En el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
;
}
MiThread.stop();
```

se crea y arranca el hilo MiThread, se duerme durante 10 segundos y en el momento de despertarse, la llamada a su método stop(), lo mata.

El método stop() envía un objeto ThreadDeath al hilo de ejecución que quiere detener. Así,

cuando un hilo es parado de este modo, muere asíncronamente. El hilo morirá en el momento en que reciba ese objeto ThreadDeath.

Los applets utilizarán el método stop() para matar a todos sus hilos cuando el navegador con soporte Java en el que se están ejecutando le indica al applet que se detengan, por ejemplo, cuando se minimiza la ventana del navegador o cuando se cambia de página.

El método isAlive()

El interfaz de programación de la clase Thread incluye el método isAlive(), que devuelve true si el hilo ha sido arrancado (con start()) y no ha sido detenido (con stop()). Por ello, si el método isAlive() devuelve false, sabemos que estamos ante un Nuevo Thread o ante un thread Muerto. Si devuelve true, se sabe que el hilo se encuentra en estado Ejecutable o Parado. No se puede diferenciar entre Nuevo Thread y Muerto, ni entre un hilo Ejecutable o Parado.

Scheduling

Java tiene un Scheduler, una lista de procesos, que monitoriza todos los hilos que se están ejecutando en todos los programas y decide cuales deben ejecutarse y cuales deben encontrarse preparados para su ejecución. Hay dos características de los hilos que el scheduler identifica en este proceso de decisión. Una, la más importante, es la prioridad del hilo de ejecución; la otra, es el indicador de demonio. La regla básica del scheduler es que si solamente hay hilos demonio ejecutándose, la Máquina Virtual Java (JVM) concluirá. Los nuevos hilos heredan la prioridad y el indicador de demonio de los hilos de ejecución que los han creado. El scheduler determina qué hilos deberán ejecutarse comprobando la prioridad de todos ellos, aquellos con prioridad más alta dispondrán del procesador antes de los que tienen prioridad más baja.

El scheduler puede seguir dos patrones, preemptivo y no-preemptivo. Los schedulers preemtivos proporcionan un segmento de tiempo a todos los hilos que están corriendo en el sistema. El scheduler decide cual será el siguiente hilo a ejecutarse y llama al método resume() para darle vida durante un período fijo de tiempo. Cuando el hilo ha estado en ejecución ese período de tiempo, se llama a suspend() y el siguiente hilo de ejecución en la lista de procesos será relanzado (resume()). Los schedulers no-preemtivos deciden que hilo debe correr y lo ejecutan hasta que concluye. El hilo tiene control total sobre el sistema mientras esté en ejecución. El método yield() es la forma en que un hilo fuerza al scheduler

a comenzar la ejecución de otro hilo que esté esperando. Dependiendo del sistema en que esté corriendo Java, el scheduler será de un tipo u otro, preemtivo o no-preemptivo.

Prioridades

El scheduler determina el hilo que debe ejecutarse en función de la prioridad asignada a cada uno de ellos. El rango de prioridades oscila entre 1 y 10. La prioridad por defecto de un hilo de ejecución es NORM_PRIORITY, que tiene asignado un valor de 5. Hay otras dos variables estáticas disponibles, que son MIN_PRORITY, fijada a 1, y MAX_PRIORITY, que tiene un valor de 10. El método getPriority() puede utilizarse para conocer el valor actual de la prioridad de un hilo.

Hilos Demonio

Los hilos de ejecución demonio también se llaman servicios, porque se ejecutan, normalmente, con prioridad baja y proporcionan un servicio básico a un programa o programas cuando la actividad de la máquina es reducida.

Los hilos demonio son útiles cuando un hilo debe ejecutarse en segundo plano durante largos períodos de tiempo.Un ejemplo de hilo demonio que está ejecutándose continuamente es el recolector de basura (garbage collector).

Este hilo, proporcionado por la Máquina Virtual Java, comprueba las variables de los programas a las que no se accede nunca y libera estos recursos, devolviéndolos al sistema.

Un hilo puede fijar su indicador de demonio pasando un valor true al método setDaemon(). Si se pasa false a este método, el hilo de ejecución será devuelto por el sistema como un hilo de usuario. No obstante, esto último debe realizarse antes de que se arranque el hilo de ejecución (start()). Si se quiere saber si un hilo es un hilo demonio, se utilizará el método isDaemon().

Diferencia entre hilos y fork()

fork() en Unix crea un proceso hijo que tiene su propia copia de datos y código del padre. Esto funciona correctamente si no hay problemas de cantidad de memoria de la máquina y se dispone de una CPU poderosa, y siempre que se mantenga el número de procesos hijos dentro de un límite manejable, porque se hace un uso intensivo de los recursos del sistema.

Los applets Java no pueden lanzar ningún proceso en el cliente, porque eso sería una fuente de inseguridad y no está permitido. Las aplicaciones y los applets deben utilizar hilos de ejecución.

La multitarea pre-emptiva tiene sus problemas. Un hilo puede interrumpir a otro en cualquier momento, de ahí lo de pre-emptive. Fácilmente puede el lector imaginarse lo que pasaría si un hilo de ejecución está escribiendo en un array, mientras otro hilo lo interrumpe y comienza a escribir en el mismo array. Los lenguajes como C y C++ necesitan de las funciones lock() y unlock() para antes y después de leer o escribir datos. Java también funciona de este modo, pero oculta el bloqueo de datos bajo la sentencia synchronized:

```
synchronized int MiMetodo();
```

Otro área en que los hilos son muy útiles es en los interfaces de usuario. Permiten incrementar la respuesta del ordenador ante el usuario cuando se encuentra realizando complicados cálculos y no puede atender a la entrada de usuario. Estos cálculos se pueden realizar en segundo plano, o realizar varios en primer plano (música y animaciones) sin que se dé apariencia de pérdida de rendimiento.

Ejemplo de animación

Este es un ejemplo de un applet, java1006.java, que crea un hilo de animación que nos presenta el globo terráqueo en rotación. Aquí se puede ver que el applet crea un hilo de ejecución de sí mismo, concurrencia. Además, animacion.start() llama al start() del hilo, no del applet, que automáticamente llamará a run():

```
import java.awt.*;
import java.applet.Applet;
public class java1006 extends Applet implements Runnable {
Image imagenes[];
MediaTracker tracker;
int indice = 0;
Thread animacion;
int maxAncho,maxAlto;
Image offScrImage; // Componente off-screen para doble buffering
Graphics offScrGC;
// Nos indicará si ya se puede pintar
boolean cargado = false;
// Inicializamos el applet, establecemos su tamaño y
```

```
// cargamos las imágenes
public void init() {
// Establecemos el supervisor de imágenes
tracker = new MediaTracker( this );
// Fijamos el tamaño del applet
maxAncho = 100;
maxAlto = 100;
imagenes = new Image[33];
// Establecemos el doble buffer y dimensionamos el applet
try {
offScrImage = createImage( maxAncho,maxAlto );
offScrGC = offScrImage.getGraphics();
offScrGC.setColor( Color.lightGray );
offScrGC.fillRect( 0,0,maxAncho,maxAlto );
resize( maxAncho,maxAlto );
} catch( Exception e ) {
e.printStackTrace();
}
// Cargamos las imágenes en un array
for( int i=0; i < 33; i++ )
{
String fichero =
new String( "Tierra"+String.valueOf(i+1)+".gif" );
imagenes[i] = getImage( getDocumentBase(),fichero );
// Registramos las imágenes con el tracker
tracker.addImage( imagenes[i],i );
}
try {
// Utilizamos el tracker para comprobar que todas las
// imágenes están cargadas
tracker.waitForAll();
```

```
} catch( InterruptedException e ) {
;
}
cargado = true;
}
// Pintamos el fotograma que corresponda
public void paint( Graphics g ) {
if( cargado )
g.drawImage( offScrImage,0,0,this );
}
// Arrancamos y establecemos la primera imagen
public void start() {
if( tracker.checkID( indice ) )
offScrGC.drawImage( imagenes[indice],0,0,this );
animacion = new Thread( this );
animacion.start();
}
// Aquí hacemos el trabajo de animación
// Muestra una imagen, para, muestra la siguiente...
public void run() {
// Obtiene el identificador del thread
Thread thActual = Thread.currentThread();
// Nos aseguramos de que se ejecuta cuando estamos en un
// thread y además es el actual
while( animacion != null && animacion == thActual )
{
if( tracker.checkID( indice ) )
{
// Obtenemos la siguiente imagen
offScrGC.drawImage( imagenes[indice],0,0,this );
indice++;
```

```
// Volvemos al principio y seguimos, para el bucle
if( indice >= imagenes.length )
indice = 0;
}
// Ralentizamos la animación para que parezca normal
try {
animacion.sleep( 200 );
} catch( InterruptedException e ) {
;
}
// Pintamos el siguiente fotograma
repaint();
}
}
}
```

En el ejemplo se pueden observar más cosas. La variable thActual es propia de cada hilo que se lance, y la variable animación la estarán viendo todos los hilos. No hay duplicidad de procesos, sino que todos comparten las mismas variables; cada hilo de ejecución, sin embargo, tiene su pila local de variables, que no comparte con nadie y que son las que están declaradas dentro de las llaves del método run().

La excepción InterruptedExcepcion salta en el caso en que se haya tenido al hilo parado más tiempo del debido.

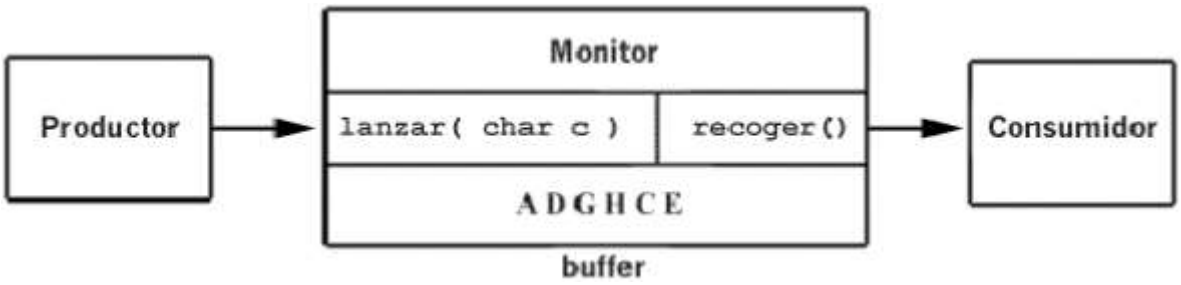
Es imprescindible recoger esta excepción cuando se están implementando hilos de ejecución, tanto es así, que en el caso de no recogerla, el compilador generará un error.

Comunicación entre Hilos

Otra clave para el éxito y la ventaja de la utilización de múltiples hilos de ejecución en una aplicación, o aplicación multithreaded, es que pueden comunicarse entre sí. Se pueden diseñar hilos para utilizar objetos comunes, que cada hilo puede manipular independientemente de los otros hilos de ejecución.

El ejemplo clásico de comunicación de hilos de ejecución es un modelo productor/consumidor. Un hilo produce una salida, que otro hilo usa (consume), sea lo que sea esa salida.

Entonces se crea un productor, que será un hilo que irá sacando caracteres por su salida; y se crea también un consumidor que irá recogiendo los caracteres que vaya sacando el productor y un monitor que controlará el proceso de sincronización entre los hilos de ejecución. Funcionará como una tubería, insertando el productor caracteres en un extremo y leyéndolos el consumidor en el otro, con el monitor siendo la propia tubería.




Productor

El productor extenderá la clase Thread, y su código es el siguiente:

```
class Productor extends Thread {
private Tuberia tuberia;
private String alfabeto = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
public Productor( Tuberia t ) {
// Mantiene una copia propia del objeto compartido
tuberia = t;
}
public void run() {
char c;
// Mete 10 letras en la tubería
```

```
for( int i=0; i < 10; i++ )
{
c = alfabeto.charAt( (int)(Math.random()*26 ) );
tuberia.lanzar( c );
// Imprime un registro con lo añadido
System.out.println( "Lanzado "+c+" a la tuberia." );
// Espera un poco antes de añadir más letras
try {
sleep( (int)(Math.random() * 100 ) );
} catch( InterruptedException e ) {;}
}
}
}
```

Notar que se crea una instancia de la clase Tuberia, y que se utiliza el método tuberia.lanzar() para que se vaya construyendo la tubería, en principio de 10 caracteres.

Consumidor

Ahora se reproduce el código del consumidor, que también extenderá la clase Thread:

```
class Consumidor extends Thread {
private Tuberia tuberia;
public Consumidor( Tuberia t ) {
// Mantiene una copia propia del objeto compartido
tuberia = t;
}
public void run() {
char c;
// Consume 10 letras de la tubería
for( int i=0; i < 10; i++ )
{
c = tuberia.recoger();
// Imprime las letras retiradas
System.out.println( "Recogido el caracter "+c );
```

```
// Espera un poco antes de coger más letras
try {
sleep( (int)(Math.random() * 2000 ) );
} catch( InterruptedException e ) {;}
}
}
}
```

En este caso, como en el del productor, se cuenta con un método en la clase Tuberia, tuberia.recoger(), para manejar la información.

Monitor

Una vez vistos el productor de la información y el consumidor, solamente queda por ver qué es lo que hace la clase Tuberia.

Lo que realiza la clase Tuberia, es una función de supervisión de las transacciones entre los dos hilos de ejecución, el productor y el consumidor. Los monitores, en general, son piezas muy importantes de las aplicaciones multihilo, porque mantienen el flujo de comunicación entre los hilos.

```
class Tuberia {
private char buffer[] = new char[6];
private int siguiente = 0;
// Flags para saber el estado del buffer
private boolean estaLlena = false;
private boolean estaVacia = true;
// Método para retirar letras del buffer
public synchronized char recoger() {
// No se puede consumir si el buffer está vacío
while( estaVacia == true )
{
try {
wait(); // Se sale cuando estaVacia cambia a false
} catch( InterruptedException e ) {
;
```

```
}
}
// Decrementa la cuenta, ya que va a consumir una letra
siguiente--;
// Comprueba si se retiró la última letra
if( siguiente == 0 )
estaVacia = true;
// El buffer no puede estar lleno, porque acabamos
// de consumir
estaLlena = false;
notify();
// Devuelve la letra al thread consumidor
return( buffer[siguiente] );
}
// Método para añadir letras al buffer
public synchronized void lanzar( char c ) {
// Espera hasta que haya sitio para otra letra
while( estaLlena == true )
{
try {
wait(); // Se sale cuando estaLlena cambia a false
} catch( InterruptedException e ) {
;
}
}
// Añade una letra en el primer lugar disponible
buffer[siguiente] = c;
// Cambia al siguiente lugar disponible
siguiente++;
// Comprueba si el buffer está lleno
if( siguiente == 6 )
```

```
estaLlena = true;
estaVacia = false;
notify();
}
}
```

En la clase Tuberia se pueden observar dos características importantes: los miembros dato (buffer[]) son privados, y los métodos de acceso (lanzar() y recoger()) son sincronizados. Aquí se observa que la variable estaVacia es un semáforo, como los de toda la vida. La naturaleza privada de los datos evita que el productor y el consumidor accedan directamente a éstos. Si se permitiese el acceso directo de ambos hilos de ejecución a los datos, se podrían producir problemas; por ejemplo, si el consumidor intenta retirar datos de un buffer vacío, obtendrá excepciones innecesarias, o se bloqueará el proceso.

Los métodos sincronizados de acceso impiden que los productores y consumidores corrompan un objeto compartido. Mientras el productor está añadiendo una letra a la tubería, el consumidor no la puede retirar y viceversa. Esta sincronización es vital para mantener la integridad de cualquier objeto compartido.

No sería lo mismo sincronizar la clase en vez de los métodos, porque esto significaría que nadie puede acceder a las variables de la clase en paralelo, mientras que al sincronizar los métodos, sí pueden acceder a todas las variables que están fuera de los métodos que pertenecen a la clase.

Se pueden sincronizar incluso variables, para realizar alguna acción determinada sobre ellas, por ejemplo:

```
sincronized( p ) {
// aquí se colocaría el código
// los threads que estén intentando acceder a p se pararán
// y generarán una InterruptedException
}
```

El método notify() al final de cada método de acceso avisa a cualquier proceso que esté esperando por el objeto, entonces el proceso que ha estado esperando intentará acceder de nuevo al objeto. En el método wait() se hace que el hilo se quede a la espera de que le llegue un notify(), ya sea enviado por el hilo de ejecución o por el sistema. Ahora que ya se

dispone de un productor, un consumidor y un objeto compartido, se necesita una aplicación que arranque los hilos y que consiga que todos hablen con el mismo objeto que están compartiendo. Esto es lo que hace el siguiente trozo de código, del fuente java1007.java:

```
class java1007 {
public static void main( String args[] ) {
Tuberia t = new Tuberia();
Productor p = new Productor( t );
Consumidor c = new Consumidor( t );
p.start();
c.start();
}
}
```

Compilando y ejecutando esta aplicación, se podrá observar en modelo que se ha diseñado en pleno funcionamiento.

### Bibliografía Recomendada:


- A. Burns, A. Wellings: Concurrency in Ada. Cambridge University Press, 1998. *Fácil de leer, es una buena introducción a la concurrencia en Ada. Incluye temas que no se tocarán en el curso.*
- G. Andrews, F. Schneider: Concepts and Notations for Concurrent Programming. ACM Computing Surveys, vol. 15, n. 1, 1983, pp. 3-43. *Una revisión de conceptos y lenguajes para expresar concurrencia. Resume las propuestas más importantes en el área. Se dejará en fotocopiadora.*
- M. Ben-Ari: Principles of Concurrent Programming. Prentice-Hall, 1982. *Escueto, con poca orientación metodológica, pero con un contenido apreciable. Incluye soluciones a los problemas clásicos de concurrencia y un capítulo sobre Rendez-Vous en Ada.*


# Adicional

- M. Ben-Ari: Ada for software engineers. John Wiley & Sons, 1998. *Una somera revisión de Ada, apropiada para alguien que conoce otros lenguajes y quiere introducirse en Ada, que incluye capítulos sobre concurrencia.*
- Gregory Andrews: Concurrent Programming, Principles and Practice. Benjamin Cummings, 1990. *Cubre casi todos los conceptos dados en la asignatura, más otros muchos relativos a algoritmos distribuidos, no necesarios en este nivel. Utiliza un lenguaje de programación propio.*
- Alan Burns, Geoff Davies: Concurrent programming. Addison-Wesley, 1993. *Una introducción a la concurrencia usando Pascal FCP. Adolece de una falta de metodología uniforme a la hora de afrontar los problemas.*
- N. H. Cohen: Ada as a Second Language. McGraw Hill. *El libro de referencia definitivo de Ada.*


# Sitios consultados


- http://babel.ls.fi.upm.es/teaching/
- http://djaramillo2dani.blogspot.mx/2011/04/guia-practica-uno-1-estructura-228106.html
- http://marcelo-trabajo.blogspot.mx/
- http://cursos.aiu.edu/Lenguages%20de%20Programacion/PDF/Tema%201.pdf
- http://algoritmosylenguajes.blogspot.mx/2008/05/unidad-iii_31.html