CLASE 9 THREADS

9.1 Introducción

Los procesadores y los Sistemas Operativos modernos permiten la multitarea, es decir, la realización simultánea de dos o más actividades (al menos aparentemente). En la realidad, un ordenador con una sola CPU no puede realizar dos actividades a la vez. Sin embargo los Sistemas Operativos actuales son capaces de ejecutar varios programas "simultáneamente" aunque sólo se disponga de una CPU: reparten el tiempo entre dos (o más) actividades, o bien utilizan los tiempos muertos de una actividad (por ejemplo, operaciones de lectura de datos desde el teclado) para trabajar en la otra. En ordenadores con dos o más procesadores la multitarea es real, ya que cada procesador puede ejecutar un hilo o thread diferente.

La Figura 9.1 muestra los esquemas correspondientes a un programa con uno o dos threads.

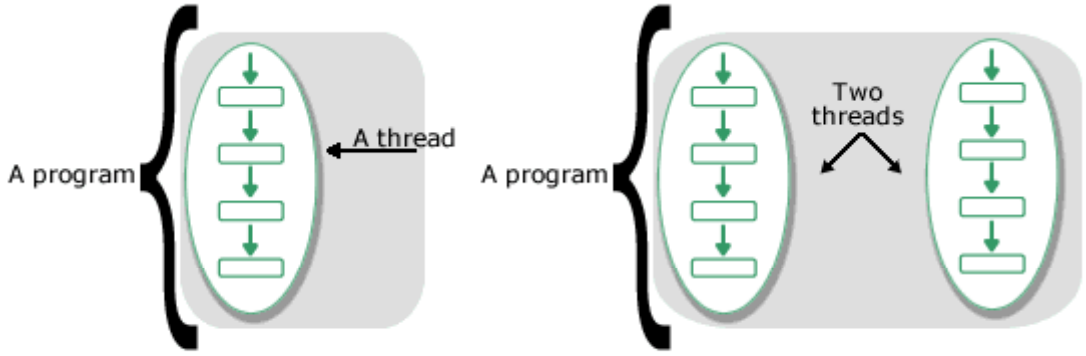


Figura 9.1. Programa con 1 y con 2 threads o hilos.

Un proceso es un programa ejecutándose de forma independiente y con un espacio propio de memoria. Un Sistema Operativo multitarea es capaz de ejecutar más de un proceso simultáneamente. Un thread o hilo es un flujo secuencial simple dentro de un proceso. Un único proceso puede tener varios hilos ejecutándose. Por ejemplo el programa IExplorer (Internet Explorer) sería un proceso, mientras que cada una de las ventanas que se pueden tener abiertas simultáneamente trayendo páginas HTML estaría formada por al menos un hilo.

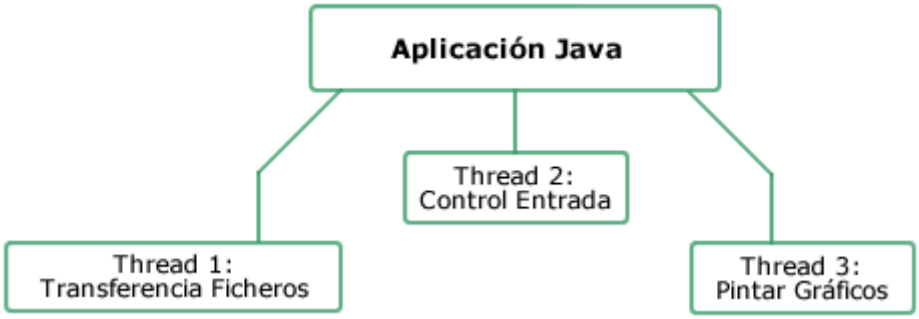
Un sistema multitarea da realmente la impresión de estar haciendo varias cosas a la vez y eso es una gran ventaja para el usuario. Sin el uso de threads hay tareas que son prácticamente imposibles de ejecutar, particularmente las que tienen tiempos de espera importantes entre etapas.

Los threads o hilos de ejecución permiten organizar los recursos del ordenador de forma que pueda haber varios programas actuando en paralelo. Un hilo de

ejecución puede realizar cualquier tarea que pueda realizar un programa normal y corriente. Bastará con indicar lo que tiene que hacer en el método run(), que es el que define la actividad principal de las threads.

Los threads pueden ser demonios o no demonios. Son demonios aquellos hilos que realizan en background (en un segundo plano) servicios generales, esto es, tareas que no forman parte de la esencia del programa y que se están ejecutando mientras no finalice la aplicación. Un thread demonio podría ser por ejemplo aquél que está comprobando permanentemente si el usuario pulsa un botón. Un programa de Java finaliza cuando sólo quedan corriendo threads de tipo demonio. Por defecto, y si no se indica lo contrario, los threads son del tipo no demonio.

Considerando el entorno multithread, cada thread (hilo, flujo de control del programa) representa un proceso individual ejecutándose en un sistema. A veces se les llama procesos ligeros o contextos de ejecución. Típicamente, cada thread controla un único aspecto dentro de un programa, como puede ser supervisar la entrada en un determinado periférico o controlar toda la entrada/salida del disco. Todos los threads comparten los mismos recursos, al contrario que los procesos en donde cada uno tiene su propia copia de código y datos (separados unos de otros). Gráficamente, los threads se parecen en su funcionamiento a lo que muestra la figura siguiente:



9.2 Flujo en Programas

9.2.1 Programas de flujo único

Un programa de flujo único o mono-hilvanado (single-thread) utiliza un único flujo de control (thread) para controlar su ejecución. Muchos programas no necesitan la potencia o utilidad de múltiples flujos de control. Sin necesidad de especificar explícitamente que se quiere un único flujo de control, muchos de los applets y aplicaciones son de flujo único.
Por ejemplo, en nuestra aplicación estándar de saludo:

```
public class HolaMundo {
   static public void main( String args[] ) {
```

```
    System.out.println( "Hola Mundo!" );
    }
}
```

Aquí, cuando se llama a main(), la aplicación imprime el mensaje y termina. Esto ocurre dentro de un único thread.

9.2.2 Programas de flujo múltiple

En nuestra aplicación de saludo, no vemos el thread que ejecuta nuestro programa. Sin embargo, Java posibilita la creación y control de threads explícitamente. La utilización de threads en Java, permite una enorme flexibilidad a los programadores a la hora de plantearse el desarrollo de aplicaciones. La simplicidad para crear, configurar y ejecutar threads, permite que se puedan implementar muy poderosas y portables aplicaciones/applets que no se puede con otros lenguajes de tercera generación. En un lenguaje orientado a Internet como es Java, esta herramienta es vital.

Si se ha utilizado un navegador con soporte Java, ya se habrá visto el uso de múltiples threads en Java. Habrá observado que dos applet se pueden ejecutar al mismo tiempo, o que puede desplazar la página del navegador mientras el applet continúa ejecutándose. Esto no significa que el applet utilice múltiples threads, sino que el navegador es multithreaded.

Las aplicaciones (y applets) multithreaded utilizan muchos contextos de ejecución para cumplir su trabajo. Hacen uso del hecho de que muchas tareas contienen subtareas distintas e independientes. Se puede utilizar un thread para cada subtarea.

Mientras que los programas de flujo único pueden realizar su tarea ejecutando las subtareas secuencialmente, un programa multithreaded permite que cada thread comience y termine tan pronto como sea posible. Este comportamiento presenta una mejor respuesta a la entrada en tiempo real.

9.2.3 Ejemplo de un programa completo con Threads

Crearemos tres threads individuales, que imprimen cada uno de ellos su propio mensaje de saludo, este archivo lo llamaremos MultiHola.java:

```
// Definimos unos sencillos threads. Se detendrán un rato
// antes de imprimir sus nombres y retardos

class TestTh extends Thread {
    private String nombre;
    private int retardo;

// Constructor para almacenar nuestro nombre y el retardo
public TestTh( String s,int d ) {
    nombre = s;
```

```
    retardo = d;
    }

// El metodo run() es similar al main(), pero para threads.
//Cuando run() termina el thread muere
public void run() {
    // Retasamos la ejecución el tiempo especificado
    try {
       sleep( retardo );
    } catch( InterruptedException e ) {
       ;
       }

    // Ahora imprimimos el nombre
    System.out.println( "Hola Mundo! "+nombre+" "+retardo );
    }
}

public class MultiHola {
    public static void main( String args[] ) {
       TestTh t1,t2,t3;

       // Creamos los threads
       t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
       t2 = new TestTh( "Thread 2",(int)(Math.random()*2000) );
       t3 = new TestTh( "Thread 3",(int)(Math.random()*2000) );

       // Arrancamos los threads
       t1.start();
       t2.start();
       t3.start();
       }
    }
```

9.3 Creación y Control de Threads

9.3.1 Creación de un Thread

En Java hay dos formas de crear nuevos threads. La primera de ellas consiste en crear una nueva clase que herede de la clase java.lang.Thread y sobrecargar el método run() de dicha clase. El segundo método consiste en declarar una clase que implemente la interface java.lang.Runnable, la cual declarará el método run(); posteriormente se crea un objeto de tipo Thread pasándole como argumento al constructor el objeto creado de la nueva clase (la que implementa la interface Runnable). Como ya se ha apuntado, tanto la clase Thread como la interface Runnable pertenecen al package java.lang, por lo que no es necesario importarlas.

La implementación de la interface Runnable es la forma habitual de crear threads. Las 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. La interface define el trabajo y la clase, o clases, que implementan la interface realizan ese trabajo. Los diferentes grupos de clases que implementen la interface tendrán que seguir las mismas reglas de funcionamiento.

Hay una cuantas diferencias entre interface y clase. Primero, una interface 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, una interface no puede implementar cualquier método. Una clase que implemente una interface debe implementar todos los métodos definidos en esa interface. Una interface tiene la posibilidad de poder extenderse de otras interfaces y, al contrario que las clases, puede extenderse de múltiples interfaces. Además, una interface no puede ser instanciada 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 thread 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 sobrecarga 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 thread. Además, el método abstracto run() está definido en la interface Runnable tiene que ser implementado. La única diferencia entre los dos métodos es que este último es mucho más flexible. En el ejemplo anterior, todavía tenemos oportunidad de extender la clase MiThread, si fuese necesario. La mayoría de las clases

creadas que necesiten ejecutarse como un thread , implementarán la interface Runnable, ya que probablemente extenderán alguna de su funcionalidad a otras clases.

No pensar que la interface 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 vemos los fuentes de Java, podremos 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 la interface Runnable. Como se ve, una interface 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 nos 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 la interface Runnable. tarea.run() se asegura de que la clase con que trabaja (la clase que va a ejecutarse como un thread) 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 thread.

A continuación se presentan dos ejemplos de creación de threads con cada uno de los dos
métodos citados.

9.3.1.1 Creación de threads derivando de la clase Thread

Considérese el siguiente ejemplo de declaración de una nueva clase:

```
public class SimpleThread extends Thread {
      // constructor
      public SimpleThread (String str) {
        super(str);
      }
```

```
    // redefinición del método run()
    public void run() {
      for(int i=0;i<10;i++)
        System.out.println("Este es el thread : " + getName());
      }
    }
```

En este caso, se ha creado la clase SimpleThread, que hereda de Thread. En su constructor se utiliza un String (opcional) para poner nombre al nuevo thread creado, y mediante super() se llama al constructor de la super-clase Thread. Asimismo, se redefine el método run(), que define la principal actividad del thread, para que escriba 10 veces el nombre del thread creado.

Para poner en marcha este nuevo thread se debe crear un objeto de la clase SimpleThread, y llamar al método start(),heredado de la super-clase Thread, que se encarga de llamar a run(). Por ejemplo:

```
SimpleThread miThread = new SimpleThread("Hilo de prueba");
miThread.start();
```

9.3.1.2 Creación de threads implementando la interface Runnable

Esta segunda forma también requiere que se defina el método run(), pero además es necesario crear un objeto de la clase Thread para lanzar la ejecución del nuevo hilo. Al constructor de la clase Thread hay que pasarle una referencia del objeto de la clase que implementa la interface Runnable. Posteriormente, cuando se ejecute el método start() del thread, éste llamará al método run() definido en la nueva clase. A continuación se muestra el mismo estilo de clase que en el ejemplo anterior implementada mediante la interface Runnable:

```
public class SimpleRunnable implements Runnable {
    // se crea un nombre
    String nameThread;

    // constructor
    public SimpleRunnable (String str) {
      nameThread = str;
    }

    // definición del método run()
    public void run() {
      for(int i=0;i<10;i++)
        System.out.println("Este es el thread: " + nameThread);
    }
}
```

El siguiente código crea un nuevo thread y lo ejecuta por este segundo procedimiento:

```
SimpleRunnable p = new SimpleRunnable("Hilo de prueba");
```

```
// se crea un objeto de la clase Thread pasándolo el objeto Runnable como
// argumento
Thread miThread = new Thread(p);
// se arranca el objeto de la clase Thread
miThread.start();
```

9.3.2 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 threads. La línea de código:

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

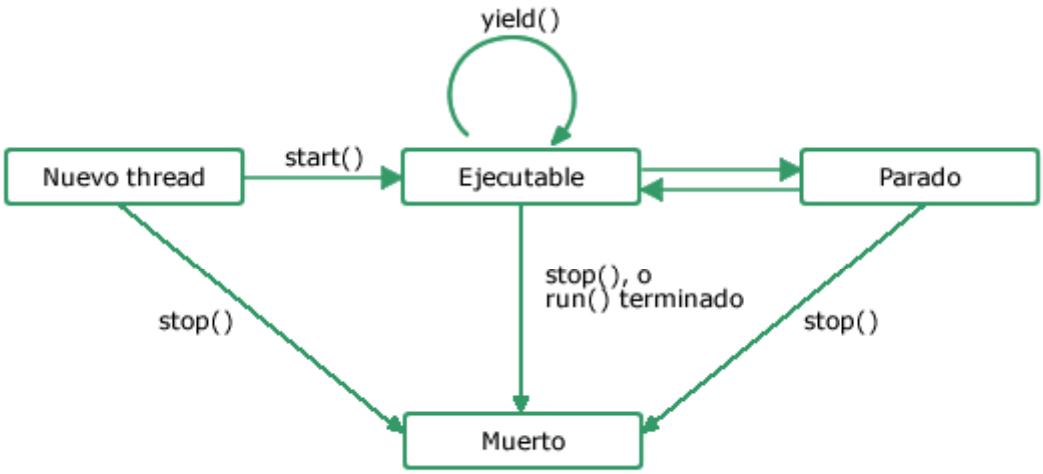
crea un nuevo thread. Los dos argumentos pasados representan el nombre del thread y el tiempo que queremos que espere antes de imprimir el mensaje.
Al tener control directo sobre los threads, tenemos que arrancarlos explícitamente. En nuestro ejemplo con:

```
    t1.start();
```

start(), en realidad es un método oculto en el thread que llama al método run().

9.4 Estados de un Thread

Durante el ciclo de vida de un thread, 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 thread .



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

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

Cuando un thread 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 veamos las dos línea 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 thread puede ejecutarse, lo incorpora a la lista de procesos disponibles para ejecución del sistema y llama al método run() del thread. En este momento nos encontramos en el estado "Ejecutable" del diagrama. Y este estado es Ejecutable y no En Ejecución, porque cuando el thread está aquí no esta corriendo. Muchos ordenadores tienen solamente un procesador lo que hace imposible que todos los threads 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 threads que se encuentran en la lista. Sin embargo, para nuestros propósitos, 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 nosotros es que todos los procesos se ejecutan al mismo tiempo.

Cuando el thread 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 thread entra en estado "Parado" cuando alguien llama al método suspend(), cuando se llama al método sleep(), cuando el thread está bloqueado en un proceso de entrada/salida o cuando el thread utiliza su método wait() para esperar a que se cumpla una determinada condición. Cuando ocurra cualquiera de las cuatro cosas anteriores, el thread 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 thread 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 thread 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 thread durmiendo no serviría para nada.

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

- Si un thread está dormido, pasado el lapso de tiempo
- Si un thread está suspendido, luego de una llamada al método resume()
- Si un thread está bloqueado en una entrada/salida, una vez que el comando E/S concluya su ejecución
- Si un thread está esperando por una condición, cada vez que la variable que controla esa condición varíe debe llamarse a notify() o notifyAll()

Muerto

Un thread se puede morir de dos formas: por causas naturales o porque lo maten (con stop()). Un thread 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 thread 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 thread, 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 thread MiThread, lo dormimos 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 thread que quiere detener. Así, cuando un thread es parado de este modo, muere asíncronamente. El thread morirá en el momento en que reciba la excepción ThreadDeath.

Los applets utilizarán el método stop() para matar a todos sus threads 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()

La interface de programación de la clase Thread incluye el método isAlive(), que devuelve true si el thread 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 nos devuelve true, sabemos que el thread se encuentra en estado "Ejecutable" o "Parado". No se puede diferenciar entre "Nuevo Thread" y "Muerto", ni entre un thread "Ejecutable" o "Parado".

9.5 Manipulación de un Thread

9.5.1 Detener un Thread temporalmente: Runnable - Not Runnable

El sistema operativo se ocupa de asignar tiempos de CPU a los distintos threads que se estén ejecutando simultáneamente. Aun en el caso de disponer de un ordenador con más de un procesador (2 ó más CPUs), el número de threads simultáneos suele siempre superar el número de CPUs, por lo que se debe repartir el tiempo de forma que parezca que todos los procesos corren a la vez (quizás más lentamente), aun cuando sólo unos pocos pueden estar ejecutándose en un instante de tiempo.

Los tiempos de CPU que el sistema continuamente asigna a los distintos threads en estado runnable se utilizan en ejecutar el método run() de cada thread. Por diversos motivos, un thread puede en un determinado momento renunciar "voluntariamente" a su tiempo de CPU y otorgárselo al sistema para que se lo asigne a otro thread. Esta "renuncia" se realiza mediante el método yield().

Es importante que este método sea utilizado por las actividades que tienden a "monopolizar" la CPU. El método yield() viene a indicar que en ese momento

no es muy importante para ese thread el ejecutarse continuamente y por lo tanto tener ocupada la CPU. En caso de que ningún thread esté requiriendo la CPU para una actividad muy intensiva, el sistema volverá casi de inmediato a asignar nuevo tiempo al thread que fue "generoso" con los demás. Por ejemplo, en un Pentium II 400 Mhz es posible llegar a más de medio millón de llamadas por segundo al método yield(), dentro del método run(), lo que significa que llamar al método yield() apenas detiene al thread, sino que sólo ofrece el control de la CPU para que el sistema decida si hay alguna otra tarea que tenga mayor prioridad.

Si lo que se desea es parar o bloquear temporalmente un thread (pasar al estado Not Runnable), existen varias formas de hacerlo:

1. Ejecutando el método sleep() de la clase Thread. Esto detiene el thread un tiempo preestablecido. De ordinario el método sleep() se llama desde el método run().

2. Ejecutando el método wait() heredado de la clase Object, a la espera de que suceda algo que es necesario para poder continuar. El thread volverá nuevamente a la situación de runnable mediante los métodos notify() o notifyAll(), que se deberán ejecutar cuando cesa la condición que tiene detenido al thread.

3. Cuando el thread está esperando para realizar operaciones de Entrada/Salida o Input/Output (E/S ó I/O).

4. Cuando el thread está tratando de llamar a un método synchronized de un objeto, y dicho objeto está bloqueado por otro thread

Un thread pasa automáticamente del estado Not Runnable a Runnable cuando cesa alguna de las condiciones anteriores o cuando se llama a notify() o notifyAll().

La clase Thread dispone también de un método stop(), pero no se debe utilizar ya que puede provocar bloqueos del programa (deadlock). Hay una última posibilidad para detener un thread, que consiste en ejecutar el método suspend(). El thread volverá a ser ejecutable de nuevo ejecutando el método resume(). Esta última forma también se desaconseja, por razones similares a la utilización del método stop().

El método sleep() de la clase Thread recibe como argumento el tiempo en milisegundos que ha de permanecer detenido. Adicionalmente, se puede incluir un número entero con un tiempo adicional en nanosegundos. Las declaraciones de estos métodos son las siguientes:
public static void sleep(long millis) throws InterruptedException
public static void sleep(long millis, int nanosecons) throws InterruptedException
Considérese el siguiente ejemplo:
System.out.println ("Contador de segundos");

```
int count=0;
public void run () {
      try {
        sleep(1000);
        System.out.println(count++);
      }
      catch (InterruptedException e){}
}
```

Se observa que el método sleep() puede lanzar una InterruptedException que ha de ser capturada. Así se ha hecho en este ejemplo, aunque luego no se gestiona esa excepción.

La forma preferible de detener temporalmente un thread es la utilización conjunta de los métodos wait() y notifyAll(). La principal ventaja del método wait() frente a los métodos anteriormente descritos es que libera el bloqueo del objeto. por lo que el resto de threads que se encuentran esperando para actuar sobre dicho objeto pueden llamar a sus métodos. Hay dos formas de llamar a wait():

> **1.** Indicando el tiempo máximo que debe estar parado (en milisegundos y con la opción de indicar también nanosegundos), de forma análoga a sleep(). A diferencia del método sleep(), que simplemente detiene el thread el tiempo indicado, el método wait() establece el tiempo máximo que debe estar parado. Si en ese plazo se ejecutan los métodos notify() o notifyAll() que indican la liberación de los objetos bloqueados, el thread continuará sin esperar a concluir el tiempo indicado. Las dos declaraciones del método wait() son como siguen:
>
> public final void wait(long timeout) throws InterruptedException
> public final void wait(long timeout, int nanos) throws InterruptedException
>
> **2.** Sin argumentos, en cuyo caso el thread permanece parado hasta que sea reinicializado explícitamente mediante los métodos notify() o notifyAll().
>
> public final void wait() throws InterruptedException

Los métodos wait() y notify() han de estar incluidas en un método synchronized, ya que de otra forma se obtendrá una excepción del tipo IllegalMonitorStateException en tiempo de ejecución. El uso típico de wait() es el de esperar a que se cumpla alguna determinada condición, ajena al propio thread. Cuando ésta se cumpla, se utilizará el método notifyAll() para avisar a los distintos threads que pueden utilizar el objeto. Estos nuevos conceptos se explican con más profundidad en el apartado siguiente.
Si todo fue bien en la creación del thread, t1 debería contener un thread válido, que controlaremos en el método run().

Una vez dentro de run(), podemos comenzar las sentencias de ejecución como en otros programas. run() sirve como rutina main() para los threads; cuando run() termina, también lo hace el thread. Todo lo que queramos que haga el thread ha de estar dentro de run(), por eso cuando decimos que un método es Runnable, nos obliga a escribir un método run().

En este ejemplo, intentamos inmediatamente esperar durante una cantidad de tiempo aleatoria (pasada a través del constructor):
sleep(retardo);
El método sleep() simplemente le dice al thread que duerma durante los milisegundos especificados. Se debería utilizar sleep() cuando se pretenda retrasar la ejecución del thread. sleep() no consume recursos del sistema mientras el thread duerme. De esta forma otros threads pueden seguir funcionando. Una vez hecho el retardo, se imprime el mensaje "Hola Mundo!" con el nombre del thread y el retardo.

9.5.2 Parar un Thread

El último elemento de control que se necesita sobre threads es el método stop(). Se utiliza para terminar la ejecución de un thread:
t1.stop();
Esta llamada no destruye el thread, 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 thread, el objeto thread (creado con new) quedará marcado para eliminarlo y el garbage collector se encargará de liberar la memoria que utilizaba.
En nuestro ejemplo, no necesitamos detener explícitamente el thread. Simplemente se le deja terminar. Los programas más complejos necesitarán un control sobre cada uno de los threads que lancen, el método stop() puede utilizarse en esas situaciones.
Si se necesita, se puede comprobar si un thread está vivo o no; considerando vivo un thread que ha comenzado y no ha sido detenido.
t1.isAlive();
Este método devolverá true en caso de que el thread 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.
Un thread finaliza cuando el método run() devuelve el control, por haber terminado lo que tenía que hacer (por ejemplo, un bucle for que se ejecuta un número determinado de veces) o por haberse dejado de cumplir una condición (por ejemplo, por un bucle while en el método run()).

9.6 Ejemplos del uso de Threads

9.6.1 Primer Ejemplo

Ahora que ya hemos visto por encima como se arrancan, paran y manipulan threads, vamos a mostrar un ejemplo un poco más gráfico, se trata de un contador, cuyo código (App1Thread.java) es el siguiente:

```
import java.awt.*;
import java.applet.Applet;
```

```
public class App1Thread extends Applet implements Runnable {
   Thread t;
   int contador;
   public void init() {
      contador = 0;
      t = new Thread( this );
      t.start();
      }
   public void run() {
      while( true )
      {
      contador++;
      repaint();
      try {
         t.sleep( 10 );
      } catch( InterruptedException e ) {
         ;
         };
      }
   }
   public boolean mouseDown( Event evt,int x,int y ) {
      t.stop();
      return( true );
      }
   public void paint( Graphics g ) {
      g.drawString( Integer.toString( contador ),10,10 );
      System.out.println( "Contador = "+contador );
      }
   public void stop() {
      t.stop();
      }
   }
```

Este applet 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, nos revelará su verdadera identidad.

En este caso, la clase App1Thread 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 thread ya conocerá al objeto que va a correr. En este caso this es una referencia a App1Thread. Después de que hayamos creado el thread, necesitamos arrancarlo. La llamada a start(), llamará a su vez al método run() de nuestra clase, es decir, a App1Thread.run(). La llamada a start() retornará con éxito y el thread 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 thread 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 thread, porque sino, el thread consumirá todo el tiempo de la CPU para su proceso y no permitirá que entren otros métodos de otros threads a ejecutarse. Otra forma de detener la ejecución del thread es hacer una llamada al método stop(). En el contador, el thread 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 convertirán en un único refresco. Así, mientras los refescos se van encolando, la variable contador se estará todavía incrementando, pero no se visualiza en pantalla.

9.6.2 Ejemplo para suspender y reanudar threads

Una vez que se para un thread, ya no se puede rearrancar con el comando start(), debido a que stop() concluirá la ejecución del thread. Por ello, en ver de parar el thread, lo que podemos hacer es dormirlo, llamando al método sleep(). El thread 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 thread reanude su ejecución ante la presencia de ciertos eventos. En estos casos, el método suspend() permite que cese la ejecución del thread y el método resume() permite que un método suspendido reanude su ejecución. En la siguiente versión de nuestra clase contador, App2Thread.java, modificamos el applet para que utilice los métodos suspend() y resume():

```
public class App2Thread extends Applet implements Runnable {
   Thread t;
   int contador;
   boolean suspendido;
   ...
   public boolean mouseDown( Event evt,int x,int y ) {
     if( suspendido )
       t.resume();
     else
       t.suspend();
     suspendido = !suspendido;

       return( true );
     }
   ...
```

Para controlar el estado del applet, hemos 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.

8.1 Scheduling

Java tiene un Scheduler, una lista de procesos, que monitoriza todos los threads 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 threads que el scheduler identifica en este proceso de decisión. Una, la más importante, es la prioridad del thread; la otra, es el indicador de demonio. La regla básica del scheduler es que si solamente hay threads demonio ejecutándose, la Máquina Virtual Java (JVM) concluirá.

Los nuevos threads heredan la prioridad y el indicador de demonio de los threads que los han creado. El scheduler determina qué threads deberán ejecutarse comprobando la prioridad de todos los threads, 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 threads que están corriendo en el sistema. El scheduler decide cual será el siguiente thread a ejecutarse y llama a resume() para darle vida durante un período fijo de tiempo. Cuando el thread ha estado en ejecución ese período de tiempo, se llama a suspend() y el siguiente thread en la lista de procesos será relanzado (resume()). Los schedulers no-preemtivos deciden que thread debe correr y lo ejecutan hasta que concluye. El thread tiene control total sobre el sistema mientras esté en ejecución. El método yield() es la forma en que un thread fuerza al scheduler a comenzar la ejecución de otro thread que esté esperando. Dependiendo del sistema en que esté corriendo Java, el scheduler será preemtivo o no-preemptivo.

En el siguiente ejemplo, SchThread.java, mostramos la ejecución de dos threads con diferentes prioridades. Un thread se ejecuta a prioridad más baja que el otro. Los threads incrementarán sus contadores hasta que el thread que tiene prioridad más alta alcance al contador que corresponde a la tarea con ejecución más lenta.

8.2 Prioridades y Demonios

8.2.1 Prioridades

El scheduler determina el thread 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 thread es Thread.NORM_PRIORITY, que tiene asignado un valor de 5. Hay otras dos variables estáticas disponibles, que son Thread.MIN_PRORITY, fijada a 1, y Thread.MAX_PRIORITY, aque tiene un valor de 10. El método getPriority() puede utilizarse para conocer el valor actual de la prioridad de un thread.

8.2.2 Threads Demonio

Los threads 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. Un ejemplo de thread demonio que está ejecutándose continuamente es el recolector de basura (garbage collector). Este thread, 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 thread puede fijar su indicador de demonio pasando un valor true al método setDaemon(). Si se pasa false a este método, el thread será devuelto por el sistema como un thread de usuario. No obstante, esto último debe realizarse antes de que se arranque el thread (start()).

Diferencia de threads con fork()
fork() en Unix crea un proceso hijo que tiene su propia copia de datos y código del padre. Esto funciona correctamente si estamos sobrados de memoria y disponemos de una CPU poderosa, y siempre que mantengamos 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 threads.

La multi-tarea pre-emptiva tiene sus problemas. Un thread puede interrumpir a otro en cualquier momento, de ahí lo de pre-emptive. Imaginarse lo que pasaría si un thread está escribiendo en un array, mientras otro thread 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 threads 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.

8.3 Sincronización

La sincronización nace de la necesidad de evitar que dos o más threads traten de acceder a los mismos recursos al mismo tiempo. Así, por ejemplo, si un thread tratara de escribir en un fichero, y otro thread estuviera al mismo tiempo tratando de borrar dicho fichero, se produciría una situación no deseada. Otra situación en la que hay que sincronizar threads se produce cuando un thread

debe esperar a que estén preparados los datos que le debe suministrar el otro thread. Para solucionar estos tipos de problemas es importante poder sincronizar los distintos threads.

Las secciones de código de un programa que acceden a un mismo recurso (un mismo objeto de una clase, un fichero del disco, etc.) desde dos threads distintos se denominan secciones críticas (critical sections). Para sincronizar dos o más threads, hay que utilizar el modificador synchronized en aquellos métodos del objeto-recurso con los que puedan producirse situaciones conflictivas. De esta forma, Java bloquea (asocia un bloqueo o lock) con el recurso sincronizado. Por ejemplo:

```
public synchronized void metodoSincronizado() {
...// accediendo por ejemplo a las variables de un objeto
...
}
```

La sincronización previene las interferencias solamente sobre un tipo de recurso: la memoria reservada para un objeto. Cuando se prevea que unas determinadas variables de una clase pueden tener problemas de sincronización, se deberán declarar como private (o protected). De esta forma sólo estarán accesibles a través de métodos de la clase, que deberán estar sincronizados.

Es muy importante tener en cuenta que si se sincronizan algunos métodos de un objeto pero otros no, el programa puede no funcionar correctamente. La razón es que los métodos no sincronizados pueden acceder libremente a las variables miembro, ignorando el bloqueo del objeto.

Sólo los métodos sincronizados comprueban si un objeto está bloqueado. Por lo tanto, todos los métodos que accedan a un recurso compartido deben ser declarados synchronized. De esta forma, si algún método accede a un determinado recurso, Java bloquea dicho recurso, de forma que el resto de threads no puedan acceder al mismo hasta que el primero en acceder termine de realizar su tarea. Bloquear un recurso u objeto significa que sobre ese objeto no pueden actuar simultáneamente dos métodos sincronizados.

Existen dos niveles de bloqueo de un recurso. El primero es a nivel de objetos, mientras que el segundo es a nivel de clases. El primero se consigue declarando todos los métodos de una clase como synchronized. Cuando se ejecuta un método synchronized sobre un objeto concreto, el sistema bloquea dicho objeto, de forma que si otro thread intenta ejecutar algún método sincronizado de ese objeto, este segundo método se mantendrá a la espera hasta que finalice el anterior (y desbloquee por lo tanto el objeto). Si existen varios objetos de una misma clase, como los bloqueos se producen a nivel de objeto, es posible tener distintos threads ejecutando métodos sobre diversos objetos de una misma clase.

El bloqueo de recursos a nivel de clases se corresponde con los métodos de clase o static, y por lo tanto con las variables de clase o static. Si lo que se

desea es conseguir que un método bloquee simultáneamente una clase entera, es decir todos los objetos creados de una clase, es necesario declarar este método como synchronized static. Durante la ejecución de un método declarado de esta segunda forma ningún método sincronizado tendrá acceso a ningún objeto de la clase bloqueada.

La sincronización puede ser problemática y generar errores. Un thread podría bloquear un determinado recurso de forma indefinida, impidiendo que el resto de threads accedieran al mismo.

Para evitar esto último, habrá que utilizar la sincronización sólo donde sea estrictamente necesario.

Es necesario tener presente que si dentro un método sincronizado se utiliza el método sleep() de la clase Thread, el objeto bloqueado permanecerá en ese estado durante el tiempo indicado en el argumento de dicho método. Esto implica que otros threads no podrán acceder a ese objeto durante ese tiempo, aunque en realidad no exista peligro de simultaneidad ya que durante ese tiempo el thread que mantiene bloqueado el objeto no realizará cambios. Para evitarlo es conveniente sustituir sleep() por el método wait() de la clase java.lang.Object heredado automáticamente por todas las clases. Cuando se llama al método wait() (siempre debe hacerse desde un método o bloque synchronized) se libera el bloqueo del objeto y por lo tanto es posible continuar utilizando ese objeto a través de métodos sincronizados. El método wait() detiene el thread hasta que se llame al método notify() o notifyAll() del objeto, o finalice el tiempo indicado como argumento del método wait(). El método unObjeto.notify() lanza una señal indicando al sistema que puede activar uno de los threads que se encuentren bloqueados esperando para acceder al objeto unObjeto. El método notifyAll() lanza una señal a todos los threads que están esperando la liberación del objeto.

Los métodos notify() y notifyAll() deben ser llamados desde el thread que tiene bloqueado el objeto para activar el resto de threads que están esperando la liberación de un objeto. Un thread se convierte en propietario del bloqueo de un objeto ejecutando un método sincronizado del objeto.

Los bloqueos de tipo clase, se consiguen ejecutando un método de clase sincronizado (synchronized static). Véanse las dos funciones siguientes, de las que put() inserta un dato y get() lo recoge:

```
public synchronized int get() {
      while (available == false) {
        try {
          // Espera a que put() asigne el valor y lo comunique con notify()
          wait();
        } catch (InterruptedException e) { }
      }
      available = false;
      // notifica que el valor ha sido leído
      notifyAll();
```

```
        // devuelve el valor
        return contents;
}

public synchronized void put(int value) {
        while (available == true) {
          try {
            // Espera a que get() lea el valor disponible antes de darle otro
            wait();
          } catch (InterruptedException e) { }
        }
        // ofrece un nuevo valor y lo declara disponible
        contents = value;
        available = true;
        // notifica que el valor ha sido cambiado
        notifyAll();
}
```

El bucle while de la función get() continúa ejecutándose (avalaible == false) hasta que el método put() haya suministrado un nuevo valor y lo indique con avalaible = true. En cada iteración del while la función wait() hace que el hilo que ejecuta el método get() se detenga hasta que se produzca un mensaje de que algo ha sido cambiado (en este caso con el método notifAll() ejecutado por put()). El método put() funciona de forma similar.

Existe también la posibilidad de sincronizar una parte del código de un método sin necesidad de mantener bloqueado el objeto desde el comienzo hasta el final del método. Para ello se utiliza la palabra clave syncronized indicando entre paréntesis el objeto que se desea sincronizar (synchronized(objetoASincronizar)). Por ejemplo si se desea sincronizar el propio thread en una parte del método run(), el código podría ser:

```
public void run() {
        while(true) {
          ...
          syncronized(this) { // El objeto a sincronizar es el propio thread
            ...                      // Código sincronizado
          }
          try {
            sleep(500);         // Se detiene el thread durante 0.5 segundos pero el
objeto
                                // es accesible por otros threads al no estar
sincronizado
          } catch(InterruptedException e) {}
        }
}
```

Un thread puede llamar a un método sincronizado de un objeto para el cual ya posee el bloqueo, volviendo a adquirir el bloqueo. Por ejemplo:

```
public class VolverAAdquirir {
      public synchronized void a() {
        b();
        System.out.println("Estoy en a()");
      }

      public synchronized void b() {
        System.out.println("Estoy en b()");
      }
}
```

El anterior ejemplo obtendrá como resultado:
Estoy en b()
Estoy en a()

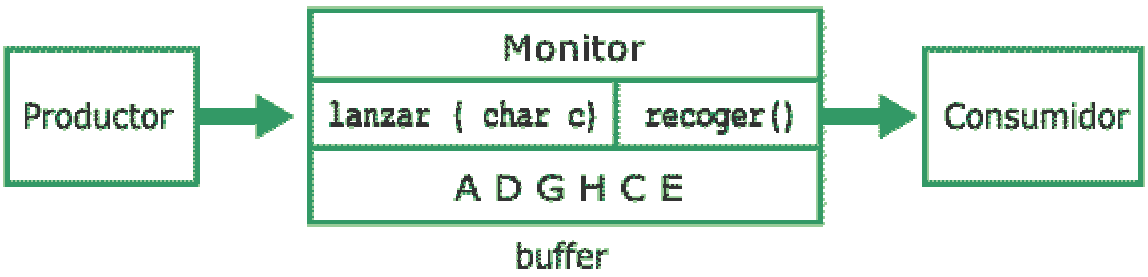
debido a que se ha podido acceder al objeto con el método b() al ser el thread que ejecuta el método a() "propietario" con anterioridad del bloqueo del objeto.

La sincronización es un proceso que lleva bastante tiempo a la CPU, luego se debe minimizar su uso, ya que el programa será más lento cuanta más sincronización incorpore.

8.4 Comunicación entre Threads

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

El ejemplo clásico de comunicación de threads es un modelo productor/consumidor. Un thread produce una salida, que otro thread usa (consume), sea lo que sea esa salida. Vamos entonces a crear un productor, que será un thread que irá sacando caracteres por su salida; crearemos también un consumidor que ira recogiendo los caracteres que vaya sacando el productor y un monitor que controlará el proceso de sincronización entre los threads. Funcionará como una tubería, insertando el productor caracteres en un extremos y leyéndolos el consumidor en el otro, con el monitor siendo la propia tubería.



8.4.1 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 );
         // Espera un poco antes de añadir más letras
         try {
            sleep( (int)(Math.random() * 100 ) );
         } catch( InterruptedException e ) {
            ;
            }
         }
    }
}
```

Notar que creamos 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.

8.4.2 Consumidor

Veamos ahora 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();
        // 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, contamos con un método en la clase Tuberia, tuberia.recoger(), para manejar la información.

8.4.3 Monitor

Una vez vistos el productor de la información y el consumidor, nos 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 threads, el productor y el consumidor. Los monitores, en general, son piezas muy importantes de las aplicaciones multithreaded, porque mantienen el flujo de comunicación entre los threads.

```
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;
    // Imprime las letras retiradas
    System.out.println( "Recogido el caracter "+ buffer[siguiente] ); 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;
    // Imprime un registro con lo añadido
    System.out.println( "Lanzado "+c+" a la tuberia." );
    notify();
    }
  }
```

En la clase Tuberia vemos dos características importantes: los miembros dato (buffer[]) son privados, y los métodos de acceso (lanzar() y recoger()) son sincronizados.

Aquí vemos 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 threads 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:

```
synchronized( 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() hacemos que el thread se quede a la espera de que le llegue un notify(), ya sea enviado por el thread o por el sistema.

Ahora que ya tenemos un productor, un consumidor y un objeto compartido, necesitamos una aplicación que arranque los threads 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 TubTest.java:

```
class TubTest {
   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, podremos observar nuestro modelo el pleno funcionamiento.

8.4.4 Monitorización del Productor

Los programas productor/consumidor a menudo emplean monitorización remota, que permite al consumidor observar el thread del productor interaccionando con un usuario o con otra parte del sistema. Por ejemplo, en una red, un grupo de threads productores podrían trabajar cada uno en una workstation. Los productores imprimirían documentos, almacenando una entrada en un registro (log). Un consumidor (o múltiples consumidores) podría procesar el registro y realizar durante la noche un informe de la actividad de impresión del día anterior.

Otro ejemplo, a pequeña escala podría ser el uso de varias ventanas en una

workstation. Una ventana se puede usar para la entrada de información (el productor), y otra ventana reaccionaría a esa información (el consumidor).

8.5 Grupos de Threads

Todo hilo de Java debe formar parte de un grupo de hilos (ThreadGroup). Puede pertenecer al grupo por defecto o a uno explícitamente creado por el usuario. Los grupos de threads proporcionan una forma sencilla de manejar múltiples threads como un solo objeto. Así, por ejemplo es posible parar varios threads con una sola llamada al método correspondiente. Una vez que un thread ha sido asociado a un threadgroup, no puede cambiar de grupo.

Cuando se arranca un programa, el sistema crea un ThreadGroup llamado main. Si en la creación de un nuevo thread no se especifica a qué grupo pertenece, automáticamente pasa a pertenecer al threadgroup del thread desde el que ha sido creado (conocido como current thread group y current thread, respectivamente). Si en dicho programa no se crea ningún ThreadGroup adicional, todos los threads creados pertenecerán al grupo main (en este grupo se encuentra el método main()). La siguiente figura presenta una posible distribución de threads distribuidos en grupos de threads.

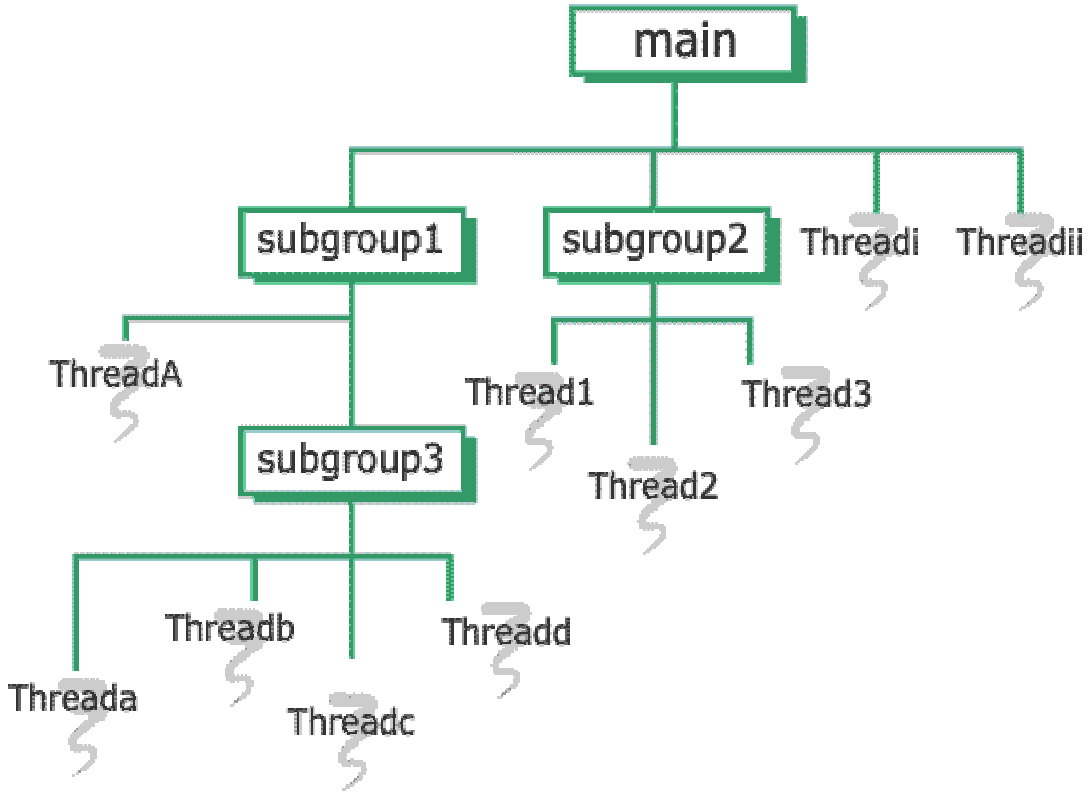


Figura 8.1: Representación de los grupos de Threads.

Para conseguir que un thread pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo thread, según uno de los siguientes constructores:

public Thread (ThreadGroup grupo, Runnable destino)
public Thread (ThreadGroup grupo, String nombre)
public Thread (ThreadGroup grupo, Runnable destino, String nombre)

A su vez, un ThreadGroup debe pertenecer a otro ThreadGroup. Como ocurría en el caso anterior, si no se especifica ninguno, el nuevo grupo pertenecerá al ThreadGroup desde el que ha sido creado (por defecto al grupo main). La clase ThreadGroup tiene dos posibles constructores:

ThreadGroup(ThreadGroup parent, String nombre);
ThreadGroup(String name);

el segundo de los cuales toma como parent el threadgroup al cual pertenezca el thread desde el que se crea (Thread.currentThread()). Para más información acerca de estos constructores, dirigirse a la documentación del API de Java donde aparecen numerosos métodos para trabajar con grupos de threads a disposición del usuario (getMaxPriority(), setMaxPriority(), getName(), getParent(), parentOf()).

En la práctica los ThreadGroups no se suelen utilizar demasiado. Su uso práctico se limita a efectuar determinadas operaciones de forma más simple que de forma individual. En cualquier caso, véase el siguiente ejemplo:

```
ThreadGroup miThreadGroup = new ThreadGroup("Mi Grupo de Threads");
Thread miThread = new Thread(miThreadGroup, "un thread para mi grupo");
```

donde se crea un grupo de threads (miThreadGroup) y un thread que pertenece a dicho grupo (miThread).