CAPÍTULO 15

HILOS

Uno de los pasos importantes que la informática dio en favor de los desarrolladores de software fue colocar un nivel de software por encima del hardware de un ordenador. Este nivel de software, conocido como sistema operativo, es en esencia una interfaz fácil de utilizar que nos permite controlar todas las partes del hardware, en la mayoría de los casos, sin un profundo conocimiento del mismo.

A su vez, los sistemas operativos también han experimentado un gran avance, pasando de los sistemas de un único procesador a los actuales sistemas operativos distribuidos o de red, o a los sistemas operativos con multiprocesadores. Esta evolución ha desembocado en un mejor aprovechamiento de todos los recursos disponibles, permitiéndonos ejecutar cada vez más tareas en menos tiempo.

El concepto central de cualquier sistema operativo es el de *proceso*. Cualquier ordenador hoy en día es capaz de hacer varias cosas simultáneamente; por ejemplo, puede estar imprimiendo un documento por la impresora y ejecutando un programa del usuario. Esto requiere que la UCP (unidad central de proceso) alterne de un programa a otro en muy cortos espacios de tiempo, lo que conocemos como tiempo compartido. De esta forma, todos los programas, incluyendo los que componen el sistema operativo, que tengan que ejecutarse simultáneamente (multiprogramación) se organizan en varios procesos secuenciales.

CONCEPTO DE PROCESO

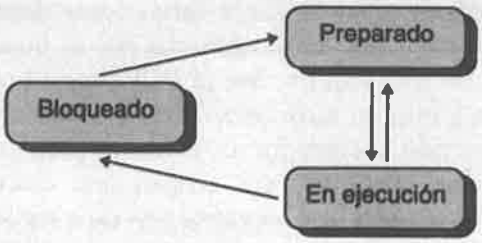
Un proceso es un ejemplar en ejecución de un programa. Cada proceso consta de bloques de código y de datos cargados desde un fichero ejecutable o desde una biblioteca dinámica. También es propietario de otros recursos que se crean durante la vida de dicho proceso y se destruyen cuando finaliza. Por ejemplo, un proceso posee:

- su propio espacio de direcciones,
- su memoria
- sus variables,
- ficheros abiertos,
- procesos hijo,
- contador de programa, registros, pila, señales, semáforos, etc.

Lo anterior, es equivalente a decir que cada proceso tiene su propia UCP virtual, lo que nos permite comprender mejor cómo un sistema puede ejecutar varios procesos simultáneamente, aunque la realidad sea que la UCP alterna entre esos procesos.

Según lo expuesto, sería un error confundir un programa con un proceso. Para evitar este posible malentendido, considere el siguiente ejemplo: cuando instalamos un juego en nuestro ordenador lo hacemos siguiendo las instrucciones adjuntas. En este caso, las instrucciones serían el programa, la actividad que hay que desarrollar para realizar la instalación (leer las instrucciones, introducir el CD-ROM, etc.) el proceso y nosotros la UCP. Si mientras estamos desarrollando esta actividad, alguien solicita nuestra colaboración para otra cosa, registramos el punto en el que nos encontramos y acudimos a resolver lo propuesto. En este caso, la UCP alterna de un proceso a otro.

De lo anterior se deduce que un proceso puede estar en *ejecución* (está utilizando la UCP), *preparado* (está detenido temporalmente para que se ejecute otro proceso) o *bloqueado* (no se puede ejecutar debido a que ocurrió algún evento al que hay que responder adecuadamente). Entre estos tres estados son posibles, como muestra la figura siguiente, cuatro transiciones:



Si un proceso en *ejecución* no puede continuar, pasa al estado de *bloqueado* o también, si puede continuar y el planificador decide que ya ha sido ejecutado el tiempo suficiente, pasa al estado de *preparado*. Si el proceso está *bloqueado* pasará a *preparado* cuando se dé el evento externo por el que se bloqueó y si está *preparado*, pasa a *ejecución* cuando el planificador lo decida porque los demás procesos ya han tenido su parte de tiempo de UCP.

En la UCP puede haber varios programas con varios procesos ejecutándose concurrentemente. En este caso se utilizan distintos mecanismos para la sincronización y comunicación entre procesos. Tales conceptos son parte del estudio de sistemas operativos.

HILOS

Un *hilo* (*thread* - llamado también proceso ligero o subproceso) es la unidad de ejecución de un proceso y está asociado con una secuencia de instrucciones, un conjunto de registros y una pila. Cuando se crea un proceso, el sistema operativo crea su primer hilo (hilo primario) el cual puede a su vez, crear hilos adicionales. Esto pone de manifiesto que un proceso no se ejecuta, sino que es sólo el espacio de direcciones donde reside el código que es ejecutado mediante uno o más hilos.

Según se ha expuesto en el apartado anterior, en un sistema operativo tradicional, cada proceso tiene un espacio de direcciones y un único *hilo de control*. Por ejemplo, considere un programa que incluya la siguiente secuencia de operaciones para actualizar el saldo de una cuenta bancaria cuando se efectúa un nuevo ingreso:

```
saldo = Cuenta.ObtnerSaldo();
saldo += ingreso;
Cuenta.EstablecerSaldo( saldo );
```

Este modelo de programación, en el que se ejecuta un solo *hilo*, es en el que estamos acostumbrados a trabajar habitualmente. Pero, continuando con el ejemplo anterior, piense en un banco real; en él, varios cajeros pueden actuar simultáneamente. Ejecutar el mismo programa por cada uno de los cajeros tiene un coste elevado (recuerde los recursos que necesita). En cambio, si el programa permitiera lanzar un hilo por cada petición de un cajero para actualizar una cuenta, estaríamos en el caso de múltiples hilos ejecutándose concurrentemente (*multithreading*). Esta característica ya es una realidad en los sistemas operativos modernos de hoy y como consecuencia contemplada en los lenguajes de programación actuales.

Como ya hemos indicado, cada hilo se ejecuta en forma estrictamente secuencial y tiene su propia pila, el estado de los registros de la UCP y su propio contador de programa. En cambio, comparten el mismo espacio de direcciones, lo que significa compartir también las mismas variables globales, el mismo conjunto de ficheros abiertos, procesos hijos (no hilos hijo), señales, semáforos, etc.

Entonces ¿qué ventajas aporta un hilo respecto a un proceso? Los hilos comparten un espacio de memoria, el código y los recursos, por lo que el lanzamiento

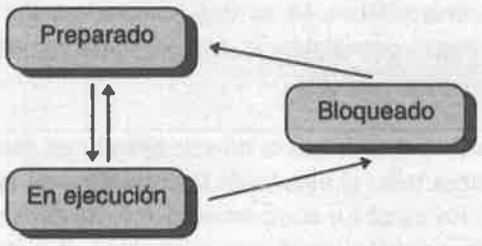
y la ejecución de un hilo es mucho más económica que el lanzamiento y la ejecución de un proceso. Por otra parte, muchos problemas pueden ser resueltos mejor con múltiples hilos; y si no, piense cómo escribiría un programa con un solo hilo de control para mostrar animación, sonido, visualizar documentos y traer ficheros de *Internet*, al mismo tiempo. No obstante, habrá situaciones en las que la mejor solución para ayudar en el trabajo sea crear un nuevo proceso (proceso hijo).

Los hilos comparten la UCP de la misma forma que lo hacen los procesos, pueden crear hilos hijo y se pueden bloquear. No obstante, mientras un hilo esté bloqueado se puede ejecutar otro hilo del mismo proceso, en el caso de hilos soportados por el *kernel* (núcleo del sistema operativo: programas en ejecución que hacen que el sistema funcione), no sucediendo lo mismo con los hilos soportados por una aplicación (por ejemplo, en Windows NT todos los hilos son soportados por el *kernel*). Un ejemplo, imaginemos que alguien llega a un cajero para depositar dinero en una cuenta y casi al mismo tiempo, un segundo cliente inicia la misma operación sobre la misma cuenta en otro cajero. Para que los resultados sean correctos, el segundo cajero quedará bloqueado hasta que el registro que está siendo actualizado por el primer cajero, quede liberado.

Resumiendo, sabemos que en la UCP puede haber varios programas con varios procesos ejecutándose concurrentemente, habilidad que se denomina multitarea, y a su vez, un proceso puede crear varios hilos y ejecutarlos de forma concurrente, lo que se traduce básicamente en una multitarea dentro de multitarea: el usuario sabe que puede ejecutar varias aplicaciones simultáneamente, y el programador sabe que cada aplicación puede ejecutar varios hilos a la vez.

Estados de un hilo

Igual que los procesos con un solo hilo de control, los *hilos* pueden encontrarse en uno de los siguientes estados:



- *Nuevo*. El hilo ha sido creado pero aún no ha sido activado. Cuando se active pasará al estado *preparado*.
- *Preparado*. El hilo está activo y está a la espera de que le sea asignada la UCP.

- *En ejecución*. El hilo está activo y le ha sido asignada la UCP (sólo los hilos activos, *preparados*, pueden ser ejecutados).
- *Bloqueado*. El hilo espera que otro elimine el bloqueo. Un hilo *bloqueado* puede estar:
  - ◊ *Dormido*. El hilo está bloqueado durante una cantidad de tiempo determinada, después de la cual despertará y pasará al estado *preparado*.
  - ◊ *Esperando*. El hilo está esperando a que ocurra alguna cosa: un mensaje **notify**, una operación de E/S o adquirir la propiedad de un método sincronizado. Cuando ocurra, pasará al estado *preparado*.
- *Muerto*. El hilo ha finalizado (está muerto) pero todavía no ha sido recogido por su padre. Los hilos *muertos* no pueden alcanzar ningún otro estado.

Observar que en la figura no se muestran los estados *nuevo* y *muerto* ya que no son estados de transición durante la vida del hilo; esto es, no se puede transitar al estado *nuevo* ni desde el estado *muerto*.

La transición entre estados está controlada por un planificador: parte del *kernel* encargada de que todos los hilos que esperan ejecutarse tenga su porción de tiempo de UCP. Si un hilo *en ejecución* no puede continuar, pasará al estado *bloqueado*; o también, si puede continuar y el planificador decide que ya ha sido ejecutado el tiempo suficiente, pasará al estado *preparado*. Si el proceso está *bloqueado* pasará a *preparado* cuando se dé el evento por el que espera; por ejemplo, puede estar esperando a que otro hilo elimine el bloqueo, o bien si está *dormido*, esperará a que pase el tiempo por el que fue enviado a este estado para ser activado; y si está *preparado*, pasará a *ejecución* cuando el planificador lo decida porque los demás hilos ya han tenido su parte de tiempo de UCP.

Cuándo se debe crear un hilo

Según lo expuesto anteriormente, cada vez que se crea un proceso, el sistema operativo crea un hilo primario. Para muchos procesos éste es el único hilo necesario. Sin embargo, un proceso puede crear otros hilos para ayudarse en su trabajo, utilizando la UCP al máximo posible. Por ejemplo, supongamos el diseño de una aplicación procesador de texto ¿Sería acertado crear un hilo separado para manipular cualquier tarea de impresión? Esto permitiría al usuario continuar utilizando la aplicación mientras se está imprimiendo. En cambio, ¿qué sucederá si los datos del documento cambian mientras se imprime? Éste es un problema que habría que resolver, quizás creando un fichero temporal que contenga los datos a imprimir.

Es evidente que los hilos son extraordinariamente útiles, pero también es evidente que si no se utilizan adecuadamente pueden introducir nuevos problemas mientras tratamos de resolver otros más antiguos. Por lo tanto, es un error pensar

que la mejor forma de desarrollar una aplicación es dividirla en partes que se ejecuten cada una como un hilo.

Cómo se crea un hilo

La mayoría del soporte que Java proporciona para trabajar con hilos reside en la clase **Thread** del paquete **java.lang**, aunque también la clase **Object**, la interfaz **Runnable** y la clases **ThreadGroup** y **ThreadDeath** del mismo paquete, así como la máquina virtual, proporcionan algún tipo de soporte.

Los hilos en Java se pueden crear de dos formas: escribiendo una nueva clase derivada de **Thread**, o bien haciendo que una clase existente implemente la interfaz **Runnable**.

La clase **Thread**, que implementa la interfaz **Runnable**, de forma resumida, está definida así:

```
public class Thread extends Object implements Runnable
{
  // Atributos
  static int MAX_PRIORITY;
    // Prioridad máxima que un hilo puede tener.
  static int MIN_PRIORITY;
    // Prioridad mínima que un hilo puede tener.
  static int NORM_PRIORITY;
    // Prioridad asignada por omisión a un hilo.

  // Constructores
  Thread([argumentos])

  // Métodos
  static Thread currentThread()
    // Devuelve una referencia al hilo que actualmente está
    // en ejecución.
  void destroy()
    // Destruye este hilo, sin realizar ninguna operación de
    // limpieza.
  String getName()
    // Devuelve el nombre del hilo.
  int getPriority()
    // Devuelve la prioridad del hilo.
  void interrupt()
    // Envía este hilo al estado de preparado.
  boolean isAlive()
    // Verifica si este hilo está vivo (no ha terminado).
```

```
  boolean isDaemon()
    // Verifica si este hilo es un demonio. Se da este nombre a
    // un hilo que se ejecuta en segundo plano, realizando una
    // operación específica en tiempos predefinidos, o bien en
    // respuesta a ciertos eventos.
  boolean isInterrupted()
    // Verifica si este hilo ha sido interrumpido.
  void join([milisegundos[, nanosegundos]])
    // Espera indefinidamente o el tiempo especificado, a que este
    // hilo termine (a que muera).
  void run()
    // Contiene el código que se ejecutará cuando el hilo pase
    // al estado de ejecución. Por omisión no hace nada.
  void setDaemon(boolean on)
    // Define este hilo como un demonio o como un hilo de usuario.
  void setName(String nombre)
    // Cambia el nombre de este hilo.
  void setPriority(int nuevaPrioridad)
    // Cambia la prioridad de este hilo. Por omisión es normal
    // (NORM_PRIORITY).
  static void sleep(long milisegundos[, int nanosegundos])
    // Envía este hilo a dormir por el tiempo especificado.
  void start()
    // Inicia la ejecución de este hilo: la máquina virtual de Java
    // invoca al método run de este hilo.
  static void yield()
    // Detiene temporalmente la ejecución de este hilo para
    // permitir la ejecución de otros.

  // Métodos heredados de la clase Object: notify, notifyAll y wait
  void notify()
    // Despierta un hilo de los que están esperando por el
    // monitor de este objeto.
  void notifyAll()
    // Despierta todos los hilos que están esperando por el
    // monitor de este objeto.
  void wait([milisegundos[, nanosegundos]])
    // Envía este hilo al estado de espera hasta que otro hilo
    // invoque al método notify o notifyAll, o hasta que transcurra
    // el tiempo especificado.
}
```

Una clase que implemente la interfaz **Runnable** tiene que sobreescribir el método **run** aportado por ésta, de ahí que **Thread** proporcione este método aunque no haga nada. El método **run** contendrá el código que debe ejecutar el hilo.

Los métodos **stop**, **suspend**, **resume** y **runFinalizersOnExit** incluidos en versiones anteriores al *jdk1.2* han sido desaprobados porque son intrínsecamente inseguros. Para más detalles vea la ayuda suministrada por *Sun*.

Hilo derivado de Thread

Según hemos dicho anteriormente, un hilo puede ser un objeto de una subclase de la clase **Thread**. Entonces, para que una aplicación pueda lanzar un determinado hilo de ejecución, el primer paso es escribir la clase del hilo derivada de **Thread** y sobreescribir el método **run** heredado por ésta, con el fin de especificar la tarea que tiene que realizar dicho hilo.

Por ejemplo, supongamos que queremos escribir una aplicación elemental que en un instante determinado de su ejecución lance un hilo que realice un simple conteo. La clase del hilo puede ser la siguiente:

```
public class ContadorAdelante extends Thread
{
  public ContadorAdelante(String nombre) // constructor
  {
    if (nombre != null) setName(nombre);
    start(); // el hilo ejecuta su propio método run
  }
  public ContadorAdelante() { this(null); } // constructor

  public void run()
  {
    for (int i = 1; i <= 1000; i++)
    {
      System.out.print(getName() + " " + i + "\r");
    }
    System.out.println();
  }
}
```

La clase *ContadorAdelante* es una subclase de **Thread** y sobreescribe el método **run** heredado. Lo que hace este método es escribir el nombre del hilo seguido de un contador de 1 a 1000.

Para poder lanzar un hilo de la clase *ContadorAdelante*, primero tenemos que construir un objeto de esa clase y después enviar a dicho objeto el mensaje **start**; De esto último se encarga el constructor de la clase. La siguiente aplicación muestra un ejemplo:

```
public class Test
{
  public static void main(String[] args)
  {
    ContadorAdelante cuentaAdelante = new ContadorAdelante("Contador+");
  }
}
```

El operador **new** crea un hilo *cuentaAdelante* (el hilo está en el estado *nuevo*). El método **start** cambia el estado del hilo a *preparado*. De ahora en adelante y hasta que finalice la ejecución del hilo *cuentaAdelante*, será el *planificador de hilos* el que determine cuándo éste pasa al estado de *ejecución* y cuándo lo abandona para permitir que se ejecuten simultáneamente otros hilos.

Según lo expuesto, el método **start** no hace que se ejecute inmediatamente el método **run** del hilo, sino que lo sitúa en el estado *preparado* para que compita por la UCP junto con el resto de los hilos que haya en este estado. Sólo el *planificador* puede asignar tiempo de UCP a un hilo y lo hará con *cuentaAdelante* en cualquier instante después de que haya recibido el mensaje **start**. Por lo tanto, un hilo durante su tiempo de vida, gasta parte de él en ejecutarse y el resto en permanecer en alguno de los estados distintos al de *ejecución*. Más adelante aprenderá cómo un hilo transita entre los diferentes estados.

Lo que no se debe de hacer es llamar directamente al método **run**; esto ejecutaría el código de este método sin que intervenga el *planificador*. Quiere esto decir que es el método **start** el que registra el hilo en el *planificador de hilos*.

Hilo asociado con una clase

Cuando sea necesario que un hilo ejecute el método **run** de un objeto de cualquier otra clase que no esté derivada de **Thread**, los pasos a seguir son los siguientes:

1. El objeto debe ser de una clase que implemente la interfaz **Runnable**, ya que es esta la que aporta el método **run**.
2. Sobreescribir el método **run** con las sentencias que tiene que ejecutar el hilo.
3. Crear un objeto de esa clase.
4. Crear un objeto de la clase **Thread** pasando como argumento al constructor, el objeto cuya clase incluye el método **run**.
5. Invocar al método **start** del objeto **Thread**.

Por ejemplo, la siguiente clase implementa la interfaz **Runnable** y sobreescribe el método **run** proporcionado por ésta:

```
public class ContadorAtras implements Runnable
{
  private Thread cuentaAtrás;
  public ContadorAtras(String nombre) // constructor
  {
    cuentaAtrás = new Thread(this);   // objeto de la clase Thread
    if (nombre != null) cuentaAtrás.setName(nombre);
    cuentaAtrás.start(); // el hilo ejecuta el método run de
  }                      // ContadorAtras
```

```
  public ContadorAtras() { this(null); } // constructor

  public void run()
  {
    for (int i = 1000; i > 0; i--)
    {
      System.out.print("\t\t" + cuentaAtrás.getName() + " " + i + " \r");
    }
    System.out.println();
  }
}
```

La clase *ContadorAtras* no se deriva de **Thread**. Sin embargo, tiene un método **run** proporcionado por la interfaz **Runnable**. Por ello, cualquier objeto *ContadorAtras* puede ser pasado como argumento cuando se invoque al constructor de la clase **Thread** cuya sintaxis es:

```
Thread(Runnable objeto)
```

Para poder lanzar un hilo asociado con la clase *ContadorAtras*, primero tenemos que construir un objeto de la misma, después un objeto de la clase **Thread** pasando como argumento el objeto de la clase *ContadorAtras* y finalmente, enviar al objeto **Thread** el mensaje **start**; de estas dos últimas operaciones se encarga el constructor *ContadorAtras*. La siguiente aplicación muestra un ejemplo:

```
public class Test
{
  public static void main(String[] args)
  {
    ContadorAtras objCuentaAtrás = new ContadorAtras("Contador-");
  }
}
```

Esta forma de lanzar un hilo quizás sea un poco más complicada. Sin embargo, hay razones suficientes para hacer este pequeño esfuerzo. Si el método **run** es parte de la interfaz de una clase cualquiera, tiene acceso a todos los miembros de esa clase, cosa que no ocurre si pertenece a una subclase de **Thread**. Otra razón es que Java no permite la herencia múltiple; entonces, si escribimos una clase derivada de **Thread**, esa clase no puede ser a la vez una subclase de cualquier otra. Por ejemplo, para poder asociar un hilo con la clase *CCuentaAhorro* derivada de *CCuenta* (capítulo 10), la única forma de hacerlo es que *CCuentaAhorro* implemente la interfaz **Runnable**.

Finalmente, a pesar de que en ocasiones hablemos en términos similares a: "la clase *ContadorAtras* es un hilo", desde el punto de vista de la programación orientada a objetos no es correcto expresarse así, a pesar de entendernos. Lo único

que es correcto es: "la clase *ContadorAtras* está asociada con un hilo". Observe en el ejemplo anterior que el hilo es el objeto de la clase **Thread**, no el objeto de la clase *ContadorAtras*. Entonces, siempre que necesitemos que una clase tenga el comportamiento de un hilo, deberemos implementar en la misma la interfaz **Runnable** y sobreescribir el método **run**.

Como ejercicio, pruebe a ejecutar la aplicación siguiente y podrá observar como los dos hilos, *cuentaAdelante* y *cuentaAtrás*, se ejecutan simultáneamente.

```
public class Test
{
  public static void main(String[] args)
  {
    ContadorAdelante cuentaAdelante = new ContadorAdelante("Contador+");
    ContadorAtras objCuentaAtrás = new ContadorAtras("Contador-");
  }
}
```

Cuando ejecute la aplicación anterior, el sistema lanza la ejecución del hilo primario (hilo padre) el cual, al ejecutarse, lanza la ejecución del hilo *cuentaAdelante* y la ejecución del hilo *cuentaAtrás*, finalizando así la ejecución de **main**; no obstante, este método no retornará hasta que no hayan finalizado los hilos hijo; esto es, el hilo primario no termina mientras no terminen sus hilos hijo.

Demonios

Un *demonio*, a diferencia de los hilos tradicionales, no forma parte de la esencia del programa, sino de la máquina Java. Los *demonios* son usados generalmente para prestar servicios en segundo plano a todos los programas que puedan necesitar el tipo de servicio proporcionado. Por ejemplo, el recolector de basura de Java es un ejemplo de este tipo de hilos.

Para crear un hilo *demonio* simplemente hay que crear un hilo normal y enviarle el mensaje **setDaemon**:

```
hilo.setDaemon(true);
```

Si un hilo es un *demonio*, entonces cualquier hilo que el cree será automáticamente un *demonio*.

Para saber si un hilo es un *demonio* simplemente hay que enviarle el mensaje **isDaemon**. El método que se ejecuta devolverá **true** si el hilo es un *demonio* y **false** en caso contrario:

```
boolean b = hilo.isDaemon();
```

El intérprete Java normalmente permanece en ejecución hasta que todos los hilos en el sistema finalizan su ejecución. Sin embargo, los *demonios* son una excepción, ya que su labor es proporcionar servicios a otros programas. Por lo tanto, no tiene sentido continuar ejecutándolos cuando ya no haya programas en ejecución. Por esta razón, el intérprete Java finalizará cuando todos los hilos que queden en ejecución sean *demonios*. El siguiente ejemplo muestra cómo implementar un hilo demonio:

```
////////////////////////////////////////////////////////////////
// Hilo demonio. Suena "bip" aproximadamente cada segundo
//
public class CDemonio extends Thread
{
  public CDemonio()
  {
    setDaemon(true);
    start();
  }

  public void run()
  {
    char bip = '\u0007';
    while (true)
    {
      try
      {
        sleep(1000); // 1 segundo
      }
      catch (InterruptedException e) {}
      System.out.print(bip);
    }
  }
}
////////////////////////////////////////////////////////////////
```

Para iniciar un demonio, *dbip*, de la clase *CDemonio* basta con escribir una sentencia como la siguiente:

```
CDemonio dbip = new CDemonio();
```

Finalizar un hilo

Un hilo termina de forma natural cuando su método **run** devuelve el control. Cuando esto sucede el hilo pasa al estado *muerto* (ha terminado) y no hay forma de salir de este estado. Esto es, una vez que el hilo está muerto, no puede ser arrancado otra vez; si deseamos ejecutar otra vez la tarea desempeñada por el hilo

hay que construir un nuevo objeto hilo y enviarle el mensaje **start**, pero sí se puede invocar a sus métodos.

Por ejemplo, supongamos una clase *ContadorAdelante* que muestra un contador ascendente que será detenido cuando el atributo *continuar* sea **false**. La clase, además de este atributo y del método **run** que muestra la cuenta, tiene un método *terminar* que pone el atributo *continuar* a **false**, y dos constructores: el primero inicia el hilo con el nombre asignado por omisión y el segundo, también lo inicia pero con el nombre pasado como argumento.

```
//////////////////////////////////////////////////////////////////
// Clase que define un hilo que cuenta ascendentemente mientras
// que el atributo continuar sea true.
//
public class ContadorAdelante extends Thread
{
  private boolean continuar = true;

  public ContadorAdelante()
  {
    start();
  }

  public ContadorAdelante(String nombreHilo)
  {
    setName(nombreHilo);
    start();
  }

  public void run()
  {
    int i = 1;
    while (continuar)
    {
      System.out.print(getName() + " " + i++ + "\r");
    }
    System.out.println();
  }

  public void terminar()
  {
    continuar = false;
  }
}
//////////////////////////////////////////////////////////////////
```

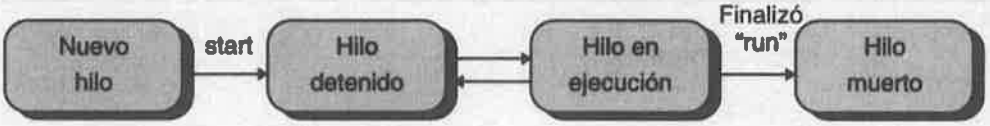
La siguiente aplicación inicia un demonio de la clase *CDemonio* expuesta anteriormente y un hilo de la clase *ContadorAdelante*. Mientras este hilo muestra

un contador ascendente en la pantalla, el demonio hace sonar un *bip* cada segundo. El contador se detendrá cuando el usuario pulse la tecla *Entrar*.

```
import java.io.*;
////////////////////////////////////////////////////////////////
// Terminar un hilo.
//
public class Test
{
  public static void main(String[] args)
  {
    // Lanzar el demonio dbip
    CDemonio dbip = new CDemonio();

    // Lanzar el hilo cuentaAdelante
    ContadorAdelante cuentaAdelante = new ContadorAdelante("Contador+");

    System.out.println("Pulse [Entrar] para finalizar");
    InputStreamReader is = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(is);
    try
    {
      br.readLine(); // ejecución detenida hasta pulsar [Entrar]
    }
    catch (IOException e) {}
    // Permitir al hilo cuentaAdelante finalizar
    cuentaAdelante.terminar();
  }
}
////////////////////////////////////////////////////////////////
```

Controlar un hilo

Ahora que ya hemos visto cómo realizar una determinada tarea utilizando un hilo, podemos deducir fácilmente que su ciclo de vida evoluciona según muestra la figura siguiente:



Un hilo, durante su ciclo de vida está transitando por los estados: *nuevo*, *preparado*, *en ejecución*, *bloqueado* y *muerto*, estudiados anteriormente. El estado *en ejecución* se corresponde en la figura con el bloque "hilo en ejecución", el cual se alcanza desde el estado *preparado* al que pasa el hilo después de que haya sido

creado y haber recibido el mensaje **start**, y cuando el hilo está vivo y no está en ejecución es que está detenido, bloque "hilo detenido".

Precisamente, el método **isAlive** de **Thread** devuelve **true** si el hilo que recibe este mensaje ha sido arrancado (**start**) y todavía no ha muerto.

Normalmente es el planificador el que controla cuándo un hilo debe estar en ejecución y cuándo pasa a estar detenido, pero en ocasiones tendremos que ser nosotros los que programemos las circunstancias bajo las cuales un hilo pueda pasar a ejecución, o bien deba pasar de ejecución a algunos de los estados *preparado* o *bloqueado* (bloqueado porque esté dormido, esté esperando a que otro hilo lo desbloquee, o esperando a que termine una operación de E/S, o bien esperando a apropiarse de un método sincronizado).

Preparado

A un hilo en ejecución se le puede enviar un mensaje **yield** para que se mueva al estado *preparado* y ceda así la UCP a otros hilos que estén compitiendo por ella (hilos que están en el estado *preparado*). Si el planificador observa que no hay ningún hilo esperando por la UCP, permitirá que el hilo que iba a ceder la UCP continúe ejecutándose.

El método **yield** es **static**, por lo tanto, opera sobre el hilo que actualmente se esté ejecutando. Cuando necesite invocarlo basta con que escriba: *Thread.yield()*.

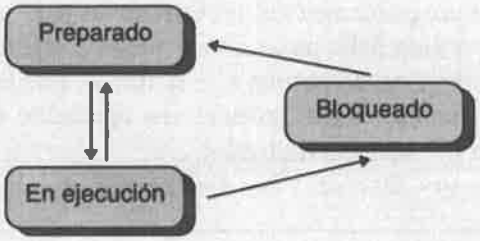
Bloqueado

Muchos métodos que ejecutan operaciones de entrada tienen que esperar por alguna circunstancia en el mundo exterior antes de que ellos puedan proseguir; este comportamiento se conoce como *bloqueo*. Por ejemplo, la sentencia siguiente lee un byte de la entrada estándar lanzando un hilo que ejecuta **read**:

```
n = System.in.read();
```

Si en la entrada estándar hay un byte disponible la sentencia anterior se ejecuta satisfactoriamente y la ejecución del método que la contiene continúa. Sin embargo, si no hay un byte disponible, **read** tiene que esperar hasta que haya uno.

Si el hilo que ejecuta **read** se mantuviera en el estado de ejecución, la UCP quedaría ocupada y no se podría realizar nada más. En general, si un método necesita esperar una cantidad de tiempo indeterminada hasta que la ocurrencia que lo detiene tenga lugar, el hilo en ejecución debe salir de este estado. Todos los métodos Java que permiten leer datos se comportan de esta forma. Un hilo que gentilmente abandona el estado de ejecución hasta que se dé la ocurrencia que lo detiene se dice que está *bloqueado*.



Java implementa muchos de los bloqueos que ocurren durante una operación de E/S llamando a los métodos **sleep** y **wait** que vemos a continuación.

Dormido

Un hilo dormido pasa tiempo sin hacer nada, por lo tanto, no utiliza la UCP.

Una llamada al método **sleep** solicita que el hilo actualmente en ejecución cese durante un tiempo especificado. Hay dos formas de llamar a este método:

```
Thread.sleep(milisegundos);
Thread.sleep(milisegundos, nanosegundos);
```

Se puede observar que el método **sleep**, igual que **yield**, es **static**. Ambos métodos operan sobre el hilo que actualmente se esté ejecutando.

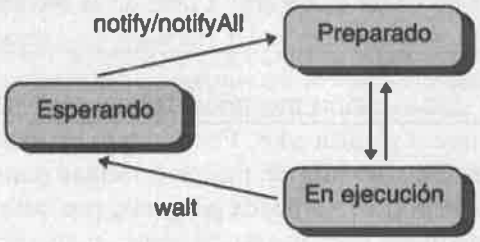
La figura anterior indica que cuando un hilo despierta (el tiempo que tenía que dormir ha transcurrido) no continúa la ejecución, sino que se mueve al estado

preparado. Pasará a ejecución cuando el planificador lo indique. Esto significa que una llamada a **sleep** bloqueará un hilo por un tiempo superior al especificado.

La clase **Thread** proporciona también un método **interrupt**. Cuando un hilo dormido recibe este mensaje, pasa automáticamente al estado *preparado*, y cuando pase a *ejecución*, ejecutará su manejador **InterruptedException**.

Esperando

El método **wait** mueve un hilo en ejecución al estado *esperando* y el método **notify** mueve un hilo que esté *esperando* al estado *preparado*; **notifyAll** mueve todos los hilos que estén *esperando* al estado *preparado*. Estos métodos, que veremos más adelante con más detalle, se utilizan para sincronizar hilos.



Planificación de hilos

Muchos ordenadores tienen sólo una UCP, así que los hilos que requieran ejecutarse deben compartirla. La ejecución de múltiples hilos sobre una única UCP, en cierto orden, es llamada *planificación*. La máquina Java (*Java Runtime Environment - JRE*: máquina virtual de Java, incluido el planificador, clases del núcleo central de Java y los ficheros de soporte) soporta un algoritmo de planificación determinista (en cualquier momento se puede saber qué hilo se está ejecutando o cuánto tiempo continuará ejecutándose) muy simple, conocido como *fixed priority scheduling* (planificación por prioridad: el hilo que se elige para su ejecución es el de prioridad más alta). Esto es, la planificación de la UCP es totalmente por derecho de prioridad (*preemptive*).

Lo anteriormente expuesto significa que cada hilo Java tiene asignado una prioridad definida por un valor numérico entre *MIN_PRIORITY* y *MAX_PRIORITY* (constantes definidas en la clase **Thread**), de forma que cuando varios hilos estén *preparados*, será elegido para su ejecución el de mayor prioridad. Solamente cuando la ejecución de ese hilo se detenga por cualquier causa, podrá ejecutarse un hilo de menor prioridad; y cuando un hilo con prioridad más alta que el

que actualmente se está ejecutando se mueva al estado *preparado*, pasará automáticamente a ejecutarse.

Según lo expuesto, la máquina Java no reemplazará el hilo actual en ejecución por otro hilo de la misma prioridad. En otras palabras, la máquina Java no aplica una planificación por *cuantos* (*time-slice* -- *cuanto* o rodaja de tiempo --: tiempo máximo que un hilo puede retener la UCP; esta planificación da lugar a un sistema no determinista), aunque la implementación del sistema de hilos que subyace en la clase **Thread** puede soportar *cuantos*. Por lo tanto, no escribir código que dependa de *cuantos* porque como se indica a continuación, en unas máquinas virtuales puede funcionar (en Windows y MacIntosh) y en otras no (en Solaris).

Después de lo dicho, sería bueno al programar que nuestros hilos cedieran voluntariamente el control algunas veces. Un hilo dado puede renunciar a su derecho de ejecutarse para ceder el control a otro de la misma prioridad llamando al método **yield**. Un intento de ceder la UCP a hilos de menor prioridad se ignorará.

La política de planificación por prioridades expuesta, puede verse en algún momento alterada por el planificador. Por ejemplo, el planificador de hilos puede elegir para su ejecución a un hilo de menor prioridad para evitar que quede completamente bloqueado porque no pueda progresar por falta de los recursos necesarios para ello (puede morir por falta de recursos: inanición -- *starvation*). Por esta razón, la exactitud de los algoritmos programados no debe basarse en la prioridad de los hilos.

¿Qué ocurre con los hilos que tengan igual prioridad?

Cuando todos los hilos que compiten por la UCP tienen la misma prioridad, el planificador elige para su ejecución al siguiente según el orden resultante de aplicar el algoritmo *round-robin* (no *preemptive*). En este caso, la cola de hilos listos para ejecutarse se trata como una cola circular *FIFO*. La UCP será cedida a otro hilo bien porque:

- un hilo de prioridad más alta ha alcanzado el estado de *preparado*;
- cede la UCP, o su método **run** finaliza;
- se supera el *cuanto* (*quantum*): tiempo máximo que un hilo puede retener la UCP. Esta tercera condición sólo es aplicable en sistemas que soporten la planificación por *cuantos*. En este aspecto, la especificación Java da mucha libertad. Cada máquina virtual implementa como quiere este agujero en la definición, cumpliendo perfectamente el estándar. Por ejemplo, las plataformas Windows 9x/NT/2000 y MacIntosh admiten planificación por *cuantos* para hilos con la misma prioridad; en cambio, Solaris planifica por *cooperación*, es decir, los hilos deben ceder voluntariamente la UCP.

Resumiendo, cuando se ejecuta un proceso que tiene varios hilos *preparados*, la máquina Java asigna la UCP en función de la prioridad que tenga asignada el hilo activo: de mayor prioridad a menor prioridad. En Java, los hilos tienen asignadas prioridades de 1 a 10 (10 es la prioridad más alta: *MAX_PRIORITY*). Por otra parte, cuando el sistema asigna la UCP a un hilo, trata de igual forma a todos los hilos de la misma prioridad. Esto es, asigna un *cuanto* al primer hilo *preparado* de prioridad 10, cuando éste finaliza su intervalo de tiempo, asigna otro *cuanto* al siguiente hilo *preparado* de prioridad 10 y así sucesivamente. Cuando todos los hilos de prioridad 10 han tenido su intervalo de tiempo, se empieza otra vez por el primero.

Según lo expuesto ¿cómo permitir la ejecución de hilos con prioridad inferior? La respuesta está en saber que muchos hilos del sistema son detenidos de vez en cuando, por motivos diferentes. Así, cuando todos los hilos de prioridad 10 estén detenidos, el sistema asigna *cuantos* a los hilos *preparados* de prioridad 9. Un razonamiento análogo nos conduce a pensar que los hilos de prioridad 8 sólo pueden ejecutarse cuando los hilos de prioridades 10 y 9 estén detenidos. Parece entonces, que los procesos de prioridad 1 nunca se ejecutarán, o que se ejecutarán de tarde en tarde. Pero, la verdad es que no es así. La mayoría de los hilos consumen su tiempo durmiendo, lo que permite la ejecución de los hilos de prioridades bajas con una frecuencia, probablemente, un poco inferior.

Asignar prioridades a los hilos

Cuando se crea un hilo Java, éste hereda su prioridad del hilo que lo crea. No obstante, es posible aumentar o disminuir esta prioridad. Para modificar la prioridad de un hilo utilice el método **setPriority**:

```
hilo.setPriority(nuevaPrioridad);
```

La siguiente tabla muestra las constantes correspondientes a las prioridades definidas en la clase **Thread**:

Constante	*Valor*
MIN_PRIORITY	1
NORM_PRIORITY	5 (valor por omisión)
MAX_PRIORITY	10

Para obtener la prioridad que tiene un hilo utilice el método **getPriority**:

```
int p = getPriority();
```

El valor que devuelve este método está comprendido entre *MIN_PRIORITY* y *MAX_PRIORITY*.

El siguiente ejemplo implementa una aplicación que visualiza *n* contadores. Por ejemplo, para 2 contadores la pantalla mostraría una línea con el siguiente formato: *nombre del hilo*, *prioridad* y *cuenta*.

```
Thread-1, P-2 2410 Thread-2, P-3 465771
```

Cada contador es un hilo. Las prioridades asignadas a cada hilo son diferentes (2, 3, etc.). La clase que da lugar a cada objeto hilo contador es la siguiente:

```
public class Contador extends Thread
{
  public int cuenta;
  private double suma = 0;

  public void run()
  {
    for (cuenta = 0; cuenta < 500000; cuenta++)
    {
      // Realizar algunos cálculos
      suma += Math.random();
    }
  }

  // Otros métodos
}
```

El método **run** de la clase simplemente genera, cuenta y acumula 500000 números aleatorios. El miembro *cuenta* es público porque otro hilo *Cuentas* utilizará ese valor para mostrar el progreso de cada uno de los contadores.

La clase *Cuentas* se implementa también como un hilo encargado de lanzar las cuentas. Su método **run** contiene un bucle que se ejecutará mientras los hilos contadores estén vivos; en cada iteración mostrará por cada hilo contador su nombre, prioridad y estado de la cuenta, y esperará durante *nMilisegundos*. La prioridad del hilo *Cuentas* será *(nCuentas+2)%Thread.MAX_PRIORITY*, donde *nCuentas* es el número de hilos contador. Por ejemplo, para 2 hilos contador la prioridad del primer hilo, *Contador[0]*, será 2, la del segundo, *Contador[1]*, será 3 y la del hilo de la clase *Cuentas*, 4. De esta forma, mientras el hilo *Cuentas* duerme los hilos contador compiten por la UCP (lógicamente finalizará antes la cuenta el de mayor prioridad) y cuando despierte, por ser el hilo de mayor prioridad obtendrá inmediatamente la UCP y mostrará los resultados actuales de las cuentas. A continuación se muestra el código correspondiente a esta clase:

```
public class Cuentas extends ...ead
{
  private static int nCuentas;
  private Contador[] cuenta;

  public Cuentas(int n)
  {
    nCuentas = n; // número de hilos contadores

    // Establecer la prioridad de este hilo
    setPriority((nCuentas+2)%Thread.MAX_PRIORITY);

    // Crear y establecer las prioridades de los hilos contador
    cuenta = new Contador[nCuentas];
    for (int i = 0; i < nCuentas; i++)
    {
      cuenta[i] = new Contador();
      cuenta[i].setPriority((i+3)%Thread.MAX_PRIORITY-1);
    }
  }

  public void run()
  {
    int i;
    boolean hayaHilosVivos;

    // Mostrar el nombre y la prioridad de este hilo
    System.out.println(this.getName() + ", P-" +
                       this.getPriority());
    // Lanzar los hilos contadores para su ejecución
    for (i = 0; i < nCuentas; i++)
      cuenta[i].start();

    do
    {
      // Mostrar nombre del hilo, prioridad y estado de la cuenta
      for (i = 0; i < nCuentas; i++)
        System.out.print(cuenta[i].getName() +
                         ", P-" + cuenta[i].getPriority() + " " +
                         cuenta[i].cuenta + " ");
      System.out.print("\r");
      // ¿Hay hilos vivos?
      hayaHilosVivos = cuenta[0].isAlive();
      for (i = 1; i < nCuentas; i++)
        hayaHilosVivos = hayaHilosVivos || cuenta[i].isAlive();
      // Ahora el hilo dormirá nMilisegundos, mientras los hilos
      // contadores siguen su curso.
      try
      {
        int nMilisegundos = (int)(10 * Math.pow(2,nCuentas));
```

```
            sleep(nMilisegundos);
          }
          catch (InterruptedException e) { }
        }
        while (hayaHilosVivos);
      }
    }
```

La aplicación *Test* que muestre los resultados perseguidos puede ser la siguiente:

```
public class Test
{
  public static void main(String[] args)
  {
    int nCuentas = 2; // número de contadores
    // Crear y lanzar el hilo Cuentas
    Cuentas hiloCuantas = new Cuentas(nCuentas);
    hiloCuantas.start();
  }
}
```

En este ejemplo que acabamos de realizar, la política de planificación es por derecho de prioridad.

¿Qué pasará si eliminamos la sentencia *sleep(nMilisegundos)*? Pues que la política de planificación seguida se ve alterada por el planificador para evitar que el hilo de mayor prioridad se apodere de la UCP (hilo egoísta); pruébelo (evitar *starvation*). Sistemas como Windows 9x/NT/2000 pelean contra los "hilos egoístas" con la estrategia de asignar la UCP por *cuantos* (*time-slicing*).

¿Qué sucede si asignamos a todos los hilos contador la misma prioridad? En esta situación, el planificador elegirá el siguiente para ejecución según el modelo *round-robin* y en el caso de Windows asignará, además, la UCP por *cuantos*.

SINCRONIZACIÓN DE HILOS

En los ejemplos que hemos visto hasta ahora cada hilo contenía todo lo que necesitaba para su ejecución: datos y métodos. Además, cada uno de ellos se podía ejecutar sin que interfiriera en la ejecución de cualquier otro hilo que se ejecutara concurrentemente con él. Estamos en el caso de *hilos independientes*.

Sin embargo, hay muchas situaciones en las que dos o más hilos ejecutándose concurrentemente deben acceder a los mismos recursos y/o datos. Como ejemplo, imagine la situación donde dos hilos acceden al mismo fichero de datos; un hilo

puede escribir en el fichero mientras el otro simultáneamente lee del mismo. Estamos en el caso de *hilos cooperantes*. Este tipo de situación puede crear resultados impredecibles, además de indeseables. En estos casos, simplemente se debe tomar el control de la situación y asegurar que cada hilo acceda a los recursos de una manera previsible, sincronizando las actividades que desarrollan cada uno de ellos. Para realizar operaciones de sincronización Java proporciona los siguientes elementos de sincronización: *secciones críticas*, **wait** y **notify**.

En general un hilo se sincroniza con otro hilo poniéndose él mismo a dormir. No obstante, antes de ponerse a dormir, debe poner en conocimiento del sistema qué evento debe ocurrir para reanudar su ejecución. De esta forma, cuando se produzca ese evento, el sistema despertará al hilo permitiéndole continuar la ejecución. Por ejemplo, si un hilo padre necesita esperar hasta que uno o más hilos hijo finalicen, se pone él mismo a dormir hasta que el hilo o hilos hijo pasen al estado *muerto*.

Cuando un hilo se pone a dormir (se bloquea), no entra en la planificación del sistema; esto es, el planificador no le asigna tiempo de UCP y, por consiguiente, detiene su ejecución.

Secciones críticas

Supongamos una aplicación en la que dos hilos de un proceso acceden a una única matriz de datos con la intención de registrar los resultados obtenidos durante un experimento. El programa podría simularse así:

- Creamos un objeto *datos* que envuelva una matriz unidimensional con el propósito de almacenar los datos adquiridos a través de una tarjeta que actúa como interfaz entre nuestra aplicación y el medio utilizado para realizar el experimento. En nuestro ejemplo, simularemos cada uno de los datos adquiridos con un valor obtenido a partir de unos sencillos cálculos.

- Creamos uno o más hilos para que tomen los datos y los vayan almacenando en la matriz hasta llenarla, instante en el que su ejecución finalizará.

Implementemos una clase *CDatos* para manipular una matriz unidimensional de tipo **double** con *n* elementos. Dicha clase incluirá los atributos:

- *datos*: matriz de tipo **double**.
- *ind*: índice del siguiente elemento vacío.
- *tamaño*: número de elementos de la matriz.

y los métodos:

- *CDatos*: es el constructor de la clase. Crea la matriz con *n* elementos, valor que se pasa como argumento, o con 10 si el valor pasado no es válido; también inicia *tamaño* con el número de elementos.
- *obtener*: devuelve el valor de un determinado elemento.
- *asignar*: asigna un valor a un determinado elemento.
- *cálculos*: obtiene el siguiente valor a almacenar en la matriz. Recibe como argumento el nombre del hilo en ejecución y devuelve el índice del siguiente elemento vacío.

```
public class CDatos
{
  // Atributos
  private double[] dato;
  private int ind = 0;
  public int tamaño;

  // Métodos
  public CDatos(int n)
  {
    if (n < 1) n = 10;
    tamaño = n;
    dato = new double[n];
  }

  public double obtener(int i)
  {
    return dato[i];
  }

  public void asignar(double x, int i)
  {
    dato[i] = x;
  }

  public int cálculos(String hilo)
  {
    if (ind >= tamaño) return tamaño;
    double x = Math.random();
    System.out.println(hilo + " muestra " + ind);
    asignar(x, ind);
    ind++;
    return ind;
  }
}
```

Uno o más hilos serán los encargados de adquirir los datos. Quiere esto decir que cuando se lancen estos hilos, el constructor de cada uno de ellos debe de recibir como argumento el objeto *CDatos* donde serán almacenados los datos que se

adquirirán, ejecutando el método *cálculos* del objeto *CDatos*. La clase de los hilos aludidos puede ser así:

```
public class CAdquirirDatos extends Thread
{
  private CDatos m;    // objeto para almacenar los datos
  public CAdquirirDatos(CDatos mdatos) // constructor
  {
    m = mdatos;
  }

  public void run()
  {
    int i = 0;

    do
    {
      i = m.cálculos(getName()); // adquirir datos
    }
    while (i < m.tamaño);
  }
}
```

Para lanzar los hilos que adquirirán los datos, implementaremos una aplicación como la siguiente:

```
public class Test
{
  public static void main(String[] args)
  {
    CDatos datos = new CDatos(10);
    CAdquirirDatos adquirirDatos_0 = new CAdquirirDatos(datos);

    adquirirDatos_0.start();
  }
}
```

En la aplicación anterior observamos que **main** crea el objeto *datos* donde el hilo *AdquirirDatos_0* almacenará los datos. Si ejecuta esta aplicación el resultado será el siguiente:

```
Thread-0 tomó la muestra 0
Thread-0 tomó la muestra 1
Thread-0 tomó la muestra 2
Thread-0 tomó la muestra 3
Thread-0 tomó la muestra 4
Thread-0 tomó la muestra 5
Thread-0 tomó la muestra 6
Thread-0 tomó la muestra 7
```

```
Thread-0 tomó la muestra 8
Thread-0 tomó la muestra 9
```

Observando los resultados vemos que todo se ha desarrollado normalmente. Modifiquemos la aplicación *Test* para que ahora utilice dos hilos en lugar de uno, para adquirir los datos:

```
public class Test
{
  public static void main(String[] args)
  {
    CDatos datos = new CDatos(10);

    CAdquirirDatos adquirirDatos_0 = new CAdquirirDatos(datos);
    CAdquirirDatos adquirirDatos_1 = new CAdquirirDatos(datos);

    adquirirDatos_0.start();
    adquirirDatos_1.start();
  }
}
```

Ahora el método **main** de la aplicación *Test* lanza dos hilos: *adquirirDatos_0* y *adquirirDatos_1*. Cuando se lanza un hilo, el retorno al proceso padre es inmediato. Por eso podemos suponer que la ejecución del hilo *AdquirirDatos_1* se inicia paralelamente a la de *adquirirDatos_0*. Si ahora ejecutamos la aplicación, el resultado obtenido será similar al siguiente:

```
Thread-0 tomó la muestra 0
Thread-0 tomó la muestra 1
Thread-0 tomó la muestra 2
Thread-1 tomó la muestra 0
Thread-1 tomó la muestra 4
Thread-0 tomó la muestra 4
Thread-1 tomó la muestra 5
Thread-0 tomó la muestra 6
Thread-1 tomó la muestra 7
Thread-0 tomó la muestra 9
Thread-1 tomó la muestra 9
java.lang.ArrayIndexOutOfBoundsException: 10
        at CDatos.asignar(CDatos.java:21)
        at CDatos.cálculos(CDatos.java:29)
        at CAdquirirDatos.run(CAdquirirDatos.java, Compiled Code)
```

Analicemos los resultados. Cuando se ejecutó sólo un hilo, la matriz se llenó totalmente sin problemas; esto es, no faltaron muestras, tampoco se perdieron por realizar almacenamientos consecutivos en el mismo elemento y no hubo accesos a elementos fuera de los límites establecidos (el número de muestra coincide con el

índice del elemento de la matriz donde está almacenada). En cambio, al ejecutarse los dos hilos concurrentemente, sí se han dado esos problemas.

En sistemas que soporten la planificación por *cuantos*, un hilo en ejecución puede ser interrumpido después de cualquier línea del método siguiente; por ejemplo, supongamos según el código siguiente que uno de los hilos se interrumpe después de la línea 6; no se incrementó *ind*. Si esto ocurre, cuando se ejecute el otro hilo, almacenará la muestra adquirida en el último elemento utilizado; y si suponemos que este hilo es interrumpido después de la línea 7, se incrementa el índice, cuando se ejecute de nuevo el hilo que se interrumpió en la línea 6, volverá a incrementar el índice dejando un elemento vacío.

```
1. public int cálculos(String hilo)
2. {
3.   if (ind >= tamaño) return tamaño;
4.   double x = Math.random();
5.   System.out.println(hilo + " muestra " + ind);
6.   asignar(x, ind);
7.   ind++;
8.   return ind;
9. }
```

Lógicamente los problemas expuestos aparecen porque dos hilos están accediendo a un mismo objeto de datos sin ningún sincronismo. Por lo tanto, la forma de evitar los problemas planteados es que cuando un hilo esté accediendo a ese objeto de datos, no pueda hacerlo el otro y viceversa. Esta sección de código que en un instante determinado tiene que acceder exclusivamente a un objeto de datos compartido, recibe el nombre de *sección crítica*.

Crear una sección crítica

En Java, cada "objeto" tiene un *monitor* (también llamado cerrojo -- *lock*). En un instante determinado, ese monitor es controlado, como mucho, por un solo hilo. El monitor controla el acceso al código sincronizado del objeto; en otras palabras, a la *sección crítica*.

Y ¿cómo se crea una *sección crítica*? La forma más sencilla de crear una *sección crítica* es agrupando el código definido como crítico en un método declarado **synchronized** (sincronizado).

En el ejemplo anterior, la sección de código crítica es el método *cálculos*. Esto quiere decir que un hilo no debe acceder a *cálculos* cuando otro hilo lo está ejecutando, para lo cual, el método *cálculos* de la clase *CDatos* debe ser declarado **synchronized**:

```
public class CDatos
{
  // ...
  public synchronized int cálculos(String hilo)
  {
    if (ind >= tamaño) return tamaño;
    double x = Math.random();
    System.out.println(hilo + " tomó la muestra " + ind);
    asignar(x, ind);
    ind++;
    return ind;
  }
}
```

Si ahora ejecuta de nuevo la aplicación *Test* comprobará que todo funciona como esperábamos.

Un hilo que quiera ejecutar el código sincronizado de un objeto debe primero intentar adquirir el control del monitor de ese objeto. Si el monitor está disponible, esto es, si no está controlado por otro hilo, entonces lo adquirirá y ejecutará el código sincronizado y cuando finalice liberará el monitor. En cambio, si el monitor está controlado por otro hilo, entonces el hilo que lo intentó se bloqueará y sólo retornará al estado *preparado* cuando el monitor esté disponible.

Las secciones de código sincronizadas, llamadas secciones críticas, denotan que el acceso a ellas es crítico para el éxito de la ejecución de los hilos del programa. Por ello, en ocasiones, nos referimos a las secciones críticas como *operaciones atómicas*, significando que ellas representan para cualquier hilo una operación que debe ejecutarse de una sola vez.

Una sección crítica puede ser también un bloque de código que se ejecuta sobre un determinado objeto. En este caso, la sección crítica se delimita así:

```
synchronized (objeto)
{
  // Código que se ejecuta sobre objeto
}
```

Si aplicamos esta segunda técnica sobre el ejemplo anterior, podemos eliminar el método *cálculos* de la clase *CDatos* y reescribir el método **run** de la clase *CAdquirirDatos* así:

```
public class CAdquirirDatos extends Thread
{
  private CDatos m;    // objeto para almacenar los datos

  public CAdquirirDatos(CDatos mdatos) // constructor
  {
    m = mdatos;
  }

  public void run()
  {
    double x;
    do
    {
      synchronized (m)
      {
        if (m.ind >= m.tamaño) return;
        x = Math.random();
        System.out.println(getName() + " tomó la muestra " + m.ind);
        m.asignar(x, m.ind);
        m.ind++;
      }
    }
    while (m.ind < m.tamaño);
  }
}
```

Observe que el código anterior exige que el atributo *ind* de *CDatos* sea público. En la versión anterior era privado.

Lo que no se debe hacer es lo que se muestra a continuación, ya que si el bucle **while** pertenece a la sección crítica, el planificador no podrá bloquear el hilo hasta que no termine de ejecutarse y por lo tanto, no podrá asignar tiempo de UCP al otro hilo. Cuando el bucle **while** finalice, la matriz ya estará llena, lo que supone que el bucle **while** para el otro hilo nunca se ejecutará.

```
synchronized (m)
{
  do
  {
    if (m.ind >= m.tamaño) return;
    x = Math.random();
    System.out.println(getName() + " tomó la muestra " + m.ind);
    m.asignar(x, m.ind);
```

```
      m.ind++;
    }
    while (m.ind < m.tamaño);
  }
```

En general es mejor aplicar la sincronización a nivel del método que a un bloque de código. La primera técnica facilita más el diseño orientado a objetos y proporciona un código más fácil de interpretar y por lo tanto, más fácil de depurar y de mantener.

Monitor reentrante

Supongamos que en alguna ocasión necesitamos que un método sincronizado tenga que llamar a otro método también sincronizado de la misma clase. Por ejemplo, para facilitar la comprensión de lo que se trata de explicar, vamos a suponer que el método *asignar* también está sincronizado:

```
public class CDatos
{
  // ...

  public synchronized void asignar(double x, int i)
  {
    dato[i] = x;
  }

  public synchronized int cálculos(String hilo)
  {
    if (ind >= tamaño) return tamaño;
    double x = Math.random();
    System.out.println(hilo + " tomó la muestra " + ind);
    asignar(x, ind);
    ind++;
    return ind;
  }
}
```

La clase *CDatos* contiene ahora dos métodos sincronizados: *asignar* y *cálculos*. El segundo llama al primero. Cuando un hilo trata de ejecutar el método *cálculos* primero toma el control del monitor del objeto *CDatos*. A continuación ejecuta este método, el cual llama al método *asignar*. Como *asignar* está también sincronizado, el hilo intenta adquirir otra vez el control del monitor del objeto *CDatos*. Parece lógico que el hilo debe bloquearse asimismo puesto que trata de adquirir un monitor que él mismo debe ceder, cosa que no puede hacer hasta que no finalice la ejecución de *cálculos*. En cambio no sucede así ¿por qué?

La máquina Java permite a un hilo volver a tomar el control de un monitor del que ya lo tiene, porque los monitores Java son reentrantes. Esto sólo funcionará en sistemas que soporten monitores reentrantes.

Utilizar wait y notify

Hemos visto que las secciones críticas son muy fáciles de utilizar, pero sólo se pueden emplear para sincronizar hilos involucrados en una única tarea; en el ejemplo anterior la tarea era única: almacenar datos en una matriz. Los métodos **wait** y **notify** proporcionan una alternativa más para compartir un objeto, pero con la diferencia de que permiten sincronizar hilos involucrados en tareas distintas, una dependiente de la otra. Piense, por ejemplo, en un sistema que cada vez que genera un mensaje lo encapsula en un objeto *CMensaje* con el fin de manipularlo. En este caso, las tareas involucradas sobre el objeto *CMensaje* son: una, almacenar el mensaje generado y otra, obtener el mensaje almacenado para mostrarlo. Claramente se ve que una tarea depende de la otra; evidentemente, un mensaje no puede ser mostrado si antes no se ha producido.

Supongamos la clase *CMensaje* según se muestra a continuación:

```
public class CMensaje
{
  private String textoMensaje;
  private int númeroMensaje;

  public synchronized void almacenar(int nmsj)
  {
    númeroMensaje = nmsj;
    // Suponer operaciones para buscar el mensaje en una tabla
    // de mensajes; resultado:
    textoMensaje = "mensaje";
  }

  public synchronized String obtener()
  {
    // Componer el mensaje bajo un determinado formato
    String mensaje;
    mensaje = textoMensaje + " #" + númeroMensaje;
    return mensaje;
  }
}
```

Se ha considerado que los métodos *almacenar* y *obtener* son *operaciones atómicas*, significando que para cualquier hilo deben ejecutarse de una sola vez; dicho de otra forma, se han definido como secciones críticas.

En el ejemplo expuesto, estamos pensando en un sistema que tendrá un productor de mensajes (mensajes de aviso, de error, etc.) para generar y almacenar los mensajes producidos (sólo se recuerda el último mensaje) y un consumidor de mensajes que mostrará al usuario el texto de cada mensaje que se produzca. Tanto el productor como el consumidor serán hilos que se suponen están alertas para desempeñar su función cuando sea requerida.

```
public class Productor extends Thread
{
  private CMensaje mensaje;      // último mensaje producido
  // Cuando se produce un mensaje, el productor
  // almacena el texto en su miembro "mensaje"
}

public class Consumidor extends Thread
{
  private CMensaje mensaje;      // mensaje a mostrar
  // Cuando se ha producido un mensaje, el consumidor
  // lo obtiene de su miembro "mensaje" y lo muestra
}
```

Completemos el código del productor. La clase *Productor* tiene un constructor que inicia el atributo *mensaje* con el objeto *CMensaje* pasado como argumento; este mismo objeto será el que utilice el consumidor para mostrar el último mensaje producido. Como se puede observar, ésta es una forma sencilla de hacer que dos o más hilos compartan datos. Asimismo, sobreescribe el método **run** para almacenar el mensaje que se produzca en el objeto *CMensaje*; este método simula que cada *msegs* milisegundos, valor generado aleatoriamente para cada mensaje, se produce el mensaje de número *númeroMsj*, valor generado también aleatoriamente. El hilo permanece dormido y despierta cada vez que se produce un mensaje. Según lo expuesto, una aproximación a la implementación de esta clase puede ser la siguiente:

```
public class Productor extends Thread
{
  private CMensaje mensaje;    // último mensaje producido

  public Productor(CMensaje c) // constructor
  {
    mensaje = c;
  }

  public void run()
  {
    int númeroMsj; // número de mensaje
    while (true)
    {
```

```
      númeroMsj = (int)(Math.random() * 100);
      mensaje.almacenar(númeroMsj); // almacena el mensaje
      System.out.println("Productor  " + getName() +
                         " almacena el mensaje #" + númeroMsj);
      try
      {
        int msegs = (int)(Math.random() * 100);
        // Poner a dormir el hilo hasta que se produzca el
        // siguiente mensaje.
        sleep(msegs);
      }
      catch (InterruptedException e) { }
    }
  }
}
```

Completemos a continuación el código del consumidor. La clase *Consumidor* tiene un constructor que inicia el atributo *mensaje* con el objeto *CMensaje* que comparte con el productor. Asimismo, sobreescribe el método **run** para obtener el mensaje almacenado en el objeto *mensaje* y mostrarlo. Según esto, la implementación de esta clase puede ser la siguiente:

```
public class Consumidor extends Thread
{
  private CMensaje mensaje;      // mensaje a mostrar

  public Consumidor(CMensaje c)  // constructor
  {
    mensaje = c;
  }

  public void run()
  {
    String msj;

    while (true)
    {
      msj = mensaje.obtener(); // obtiene el último mensaje
      System.out.println("Consumidor " + getName() +
                         " obtuvo:   " + msj);
    }
  }
}
```

Una aplicación que lance los hilos productor y consumidor y muestre los resultados que producen puede ser la siguiente:

```
public class Test
{
```

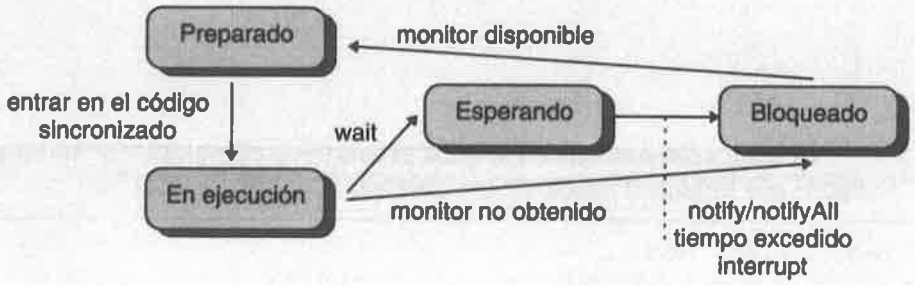
```
    public static void main(String[] args)
    {
      CMensaje mensaje = new CMensaje();
      Productor productor1 = new Productor(mensaje);
      Consumidor consumidor1 = new Consumidor(mensaje);

      productor1.start();
      consumidor1.start();
    }
}
```

Cuando ejecute la aplicación anterior, tenga presente que los hilos productor y consumidor trabajarán indefinidamente. Por lo tanto, para detener la ejecución tendrá que pulsar las teclas *Ctrl+C*. En lugar de esto, podríamos haber utilizado la técnica mostrada en el apartado "Finalizar un hilo". No lo hemos hecho para no complicar el código y centrarnos en el tema de sincronización. Una vez que haya ejecutado la aplicación, observará resultados análogos a los siguientes:

```
Consumidor Thread-1 obtuvo:    null #0
Productor  Thread-0 almacena: mensaje #15
Consumidor Thread-1 obtuvo:    mensaje #15
Consumidor Thread-1 obtuvo:    mensaje #15
Consumidor Thread-1 obtuvo:    mensaje #15
Consumidor Thread-1 obtuvo:    mensaje #15
Productor  Thread-0 almacena: mensaje #19
Consumidor Thread-1 obtuvo:    mensaje #19
Consumidor Thread-1 obtuvo:    mensaje #19
```

Un análisis sencillo nos conduce a la conclusión de que independientemente de que los métodos se hayan definido como secciones críticas, no existe una sincronización entre el productor y el consumidor. El consumidor muestra el último mensaje producido cada vez que el planificador le asigna tiempo de UCP, en lugar de hacerlo única y exclusivamente cada vez que se produzca un mensaje.

Para conseguir la sincronización deseada, el monitor asociado con el objeto *CMensaje* tiene que auxiliarse de los métodos **wait** y **notify**. La siguiente figura muestra las transiciones de estados cuando intervienen estos métodos:



Dijimos que en Java, cada "objeto" tiene un *monitor*. El monitor controla el acceso al código sincronizado del objeto; en otras palabras, a la *sección crítica*. Y según hemos visto, un objeto *CMensaje* presenta dos secciones críticas. Pues bien, el método **notify** despierta sólo un hilo de los que estén esperando por ese monitor. Esto es, si hay varios hilos esperando, se elige uno arbitrariamente. El hilo despertado competirá de la manera habitual con el resto de los hilos que estén en el estado *preparado*, por adquirir la UCP. Según lo expuesto, el método **notify** sólo puede ser llamado por un hilo que haya adquirido el control del monitor.

Un hilo se pone a esperar por el monitor de un determinado objeto invocando al método **wait**. Además, el hilo cede el control del monitor.

```
void wait([milisegundos[, nanosegundos]])
```

El método **wait** envía al hilo actualmente en ejecución al estado de espera, hasta que otro hilo, el que tiene el control del monitor, invoque al método **notify**, **notifyAll** o **interrupt**, o bien hasta que transcurra el tiempo especificado. Cuando el hilo se pone a dormir, cede el control sólo del monitor que controla el acceso al código sincronizado del objeto que lo ha invocado, lo que permitirá a otro hilo que esté esperando por él, adquirirlo; esto es, cualquier otro objeto actualmente controlado por el hilo que se pone a dormir permanecerá bloqueado mientras éste esté dormido.

Precisamente, una de las diferencias entre **sleep** y **wait** es que el primero, cuando es llamado, no cede el control del monitor, mientras que el segundo sí.

El método **notifyAll**, a diferencia de **notify**, despierta todos los hilos que están esperando por el monitor que controla el acceso al código sincronizado de un objeto. Igualmente, los hilos despertados competirán de la manera habitual por adquirir la UCP con el resto de los hilos que estén en el estado *preparado*.

Evidentemente, el método **notify** es más rápido que **notifyAll**, pero su forma de proceder nos puede conducir a situaciones no deseadas cuando hay varios hilos esperando en el mismo objeto. En este caso, rara vez se utiliza, y por seguridad se sugiere utilizar **notifyAll**.

Lo anteriormente expuesto conduce a la conclusión de que los métodos **wait**, **notify**, **notifyAll** e **interrupt**, deben ser invocados desde código sincronizado.

Aplicando la teoría expuesta vamos a continuación a sincronizar los hilos productor y consumidor del ejemplo que estamos desarrollando. Para ello, hay que tener presente que no se puede mostrar un mensaje que aún no se ha generado (por supuesto, los mensajes se muestran una sola vez) y no se puede almacenar un mensaje, si aún no se ha mostrado el último generado.

Entonces, añadiremos a la clase *CMensaje* un atributo *disponible* de tipo **boolean**, que valga **false** cuando no haya ningún mensaje que mostrar, y **true** en caso contrario.

```
public class CMensaje
{
  private String textoMensaje;
  private int númeroMensaje;
  private boolean disponible = false;

  // ...
}
```

Ahora, cuando el hilo productor adquiera el control del monitor del objeto *CMensaje* y ejecute el método sincronizado *almacenar*, lo primero que hará será interrogar el atributo *disponible*. Si su valor es **true**, el hilo se pondrá a dormir hasta que se muestre el último mensaje producido y cede el control del monitor, y si vale **false**, almacena el nuevo mensaje, cambia el atributo *disponible* a **true**, e invoca a **notifyAll** para despertar a todos los hilos que estén esperando por este monitor.

```
public synchronized void almacenar(int nmsj)
{
  while (disponible == true)
  {
    // El último mensaje aún no ha sido mostrado
    try
    {
      wait(); // el hilo se pone a dormir y cede el monitor
    }
    catch (InterruptedException e) { }
  }
  númeroMensaje = nmsj;
  // Suponer operaciones para buscar el mensaje en una tabla
  // de mensajes; resultado:
  textoMensaje = "mensaje";
  disponible = true;
  notifyAll();
}
```

Asimismo, cuando el hilo consumidor adquiera el control del monitor del objeto *CMensaje* y ejecute el método sincronizado *obtener*, lo primero que hará será interrogar el atributo *disponible*. Si su valor es **false**, el hilo esperará hasta que haya un mensaje cediendo el control del monitor, y si su valor es **true**, cambia el atributo *disponible* a **false**, invoca a **notifyAll** para despertar a todos los hilos que estén esperando por este monitor y retorna el mensaje.

```
public synchronized String obtener()
{
  while (disponible == false)
  {
    // No hay mensaje
    try
    {
      wait(); // el hilo se pone a dormir y cede el monitor
    }
    catch (InterruptedException e) { }
  }
  disponible = false;
  notifyAll();
  // Componer el mensaje bajo un determinado formato
  String mensaje;
  mensaje = textoMensaje + " #" + númeroMensaje;
  return mensaje;
}
```

Una vez modificados los métodos *almacenar* y *obtener* de la clase *CMensaje*, ejecute de nuevo la aplicación *Test*. Observará que ahora los resultados sí son los esperados:

```
Productor  Thread-0 almacena: mensaje #74
Consumidor Thread-1 obtuvo:   mensaje #74
Productor  Thread-0 almacena: mensaje #17
Consumidor Thread-1 obtuvo:   mensaje #17
Productor  Thread-0 almacena: mensaje #85
Consumidor Thread-1 obtuvo:   mensaje #85
Productor  Thread-0 almacena: mensaje #3
Consumidor Thread-1 obtuvo:   mensaje #3
Productor  Thread-0 almacena: mensaje #91
Consumidor Thread-1 obtuvo:   mensaje #91
```

¿Por qué los métodos almacenar y obtener utilizan un bucle?

Antes de proceder a la explicación, añada más consumidores y observar los resultados. Por ejemplo:

```
public class Test
{
  public static void main(String[] args)
  {
    CMensaje mensaje = new CMensaje();

    Productor productor1 = new Productor(mensaje);
    Consumidor consumidor1 = new Consumidor(mensaje);
    Consumidor consumidor2 = new Consumidor(mensaje);
```

```
        productor1.start();
        consumidor1.start();
        consumidor2.start();
    }
}
```

A continuación edite los métodos *almacenar* y *obtener*, y cambie las sentencias **while** por **if**:

```
1. if (disponible == false) // antes: while (disponible == false)
2. {
3.   try
4.   {
5.     wait(); // el hilo se pone a dormir y cede el monitor
6.   }
7.   catch (InterruptedException e) { }
8. }
9. disponible = false;
10. notifyAll();
11. String mensaje;
12. mensaje = textoMensaje + " #" + númeroMensaje;
13. return mensaje;
```

Compile y ejecute de nuevo la aplicación. Compare los resultados con los obtenidos anteriormente ¿Qué ha ocurrido?

```
Productor  Thread-0 almacena: mensaje #14
Consumidor Thread-1 obtuvo:   mensaje #14
Consumidor Thread-2 obtuvo:   mensaje #14
```

El peligro de utilizar una sentencia **if** en lugar de **while** es que algunas veces el hilo que adquiere el control del monitor, *Thread-2*, ejecuta la línea *1* y suponiendo que la condición es cierta se pone a esperar, además de ceder el monitor. Más tarde, otro hilo adquiere el monitor, *Thread-1*, suponiendo que la condición es falsa ejecuta la línea 10 y despierta a los hilos que esperan por este monitor. Los hilos en el estado *preparado* compiten por la UCP. Supongamos que el planificador se la adjudica al hilo original, *Thread-2*; éste continuará donde lo dejó, a partir de la línea 5, independientemente del estado del monitor. El resultado es que retorna el mismo mensaje que *Thread-1*. Utilizando una sentencia **while** en lugar de **if**, cuando *Thread-2* continúe donde lo dejó, a partir de la línea 5, volverá a ejecutar la línea *1* y se pondrá de nuevo a esperar por ser la condición cierta.

Interbloqueo

Anteriormente dijimos que la inanición (*starvation*) ocurre cuando un hilo se queda complemente bloqueado y no puede progresar porque no puede acceder a los

recursos que necesita; si esto ocurre entre dos o más hilos porque esperan por una condición recíproca que nunca puede ser satisfecha, estamos en un caso de *interbloqueo* (*deadlock*; algunos autores prefieren denominarlo *abrazo mortal*). Por ejemplo dos hilos necesitan imprimir un documento almacenado en el disco, para lo que necesitan los recursos disco e impresora. Puesto que los hilos se están ejecutando paralelamente, suponga que uno ya ha adquirido el disco y el otro la impresora. Esto significa que ambos hilos quedarán bloqueados, cada uno de ellos esperando por el recurso que tiene el otro.

Para la mayoría de los programadores Java, la mejor de evitar el interbloqueo es prevenirlo, mejor que probar y detectarlo. En cualquier caso, cualquiera de las técnicas existentes para manejar los interbloqueos se sale fuera del objetivo de este capítulo.

GRUPO DE HILOS

Cada hilo Java es un miembro de un *grupo de hilos*. Este grupo puede ser el predefinido por Java o uno especificado explícitamente. Los grupos de hilos proporcionan un mecanismo para agrupar varios hilos en un único objeto con el fin de poder manipularlos todos de una vez; por ejemplo, poder interrumpir un grupo de hilos invocando una sola vez al método **interrupt**. A su vez, un grupo de hilos también puede pertenecer a otro grupo, formando una estructura en árbol. Desde el punto de vista de esta estructura, un hilo sólo tiene acceso a la información acerca de su grupo, no a la de su grupo padre o de cualquier otro grupo.

Java proporciona soporte para trabajar con grupos de hilos a través de la clase **ThreadGroup** del paquete **lang**.

Grupo predefinido

Cuando creamos un hilo sin especificar su grupo en el constructor, Java lo coloca en el mismo grupo (*grupo actual*) del hilo bajo el cual se crea (*hilo actual*).

Por ejemplo, la siguiente aplicación obtiene una referencia al *grupo actual* (grupo predefinido) al cual pertenece el *hilo actual* (en este caso el hilo primario) y la almacena en *consumidores*.

Después, cada hilo consumidor que es creado es añadido al grupo actual que hemos denominado *consumidores*.

Finalmente, más adelante, se envía al grupo actual el mensaje **list** con el objetivo de escribir información acerca del grupo de hilos. Otros métodos puede verlos en la documentación proporcionada con el JDK.

```
public class Test
{
  public static void main(String[] args)
  {
    ThreadGroup consumidores =
                        Thread.currentThread().getThreadGroup();
    CMensaje mensaje = new CMensaje();
    Productor productor1 = new Productor(mensaje);
    Consumidor consumidor1 = new Consumidor(mensaje, consumidores,
                                            "consumidor1");
    Consumidor consumidor2 = new Consumidor(mensaje, consumidores,
                                            "consumidor2");
    consumidores.list();

    // ...
  }
}
```

¿Cómo se añaden los hilos a un grupo? Pues utilizando alguno de los constructores que la clase **Thread** proporciona para ello. Por ejemplo:

Thread(ThreadGroup *grupo*, **String** *nombreHilo*)

Siguiendo con el ejemplo anterior, vemos que cuando se invocó al constructor *Consumidor* se pasaron tres argumentos: un objeto *CMensaje*, el grupo de hilos y el nombre del hilo que se desea añadir al grupo. Según esto, el constructor de esta clase será como se indica a continuación:

```
public class Consumidor extends Thread
{
  private CMensaje mensaje;       // mensaje a mostrar

  public Consumidor(CMensaje msj, ThreadGroup grupo, String nombre)
  {
    super(grupo, nombre);
    mensaje = msj;
  }

  public void run()
  {
    // ...
  }
}
```

Se puede observar que el constructor de la clase *Consumidor* invoca al constructor de su clase base (**Thread**) pasándole como argumento el grupo al cual se quiere añadir el hilo, el cual está referenciado por **this**, y el nombre del hilo.

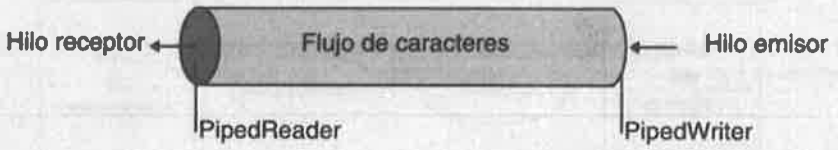
Grupo explícito

Para añadir un hilo a un determinado grupo primero crearemos el grupo y después procederemos de la misma forma explicada en el apartado anterior. Por ejemplo, si en el ejemplo anterior en lugar de utilizar el grupo predefinido por Java quisiéramos definir explícitamente un grupo referenciado por la variable *consumidores* y denominado también *consumidores*, la primera línea del método **main** la sustituiríamos por la sombreada en el código mostrado a continuación:

```
public class Test
{
  public static void main(String[] args)
  {
    ThreadGroup consumidores = new ThreadGroup("consumidores");
    // ...
  }
}
```

TUBERÍAS

Básicamente una tubería es utilizada para canalizar la salida de un hilo (puede ser el hilo principal de un programa en ejecución) hacia la entrada de otro. De esta forma los hilos pueden compartir datos sin tener que recurrir a otros elementos como, por ejemplo, ficheros temporales o matrices.



Java proporciona las clases **PipedReader** y **PipedWriter** (y sus homólogas para bytes, **PipedInputStream** y **PipedOutputStream**) para trabajar con tuberías a través de las cuales circularán flujos de caracteres. La primera representa el extremo de la tubería del cual un hilo obtiene los datos y la segunda el extremo de la tubería por el cual un hilo envía los datos al otro. Por lo tanto, estas clases trabajan conjuntamente para proporcionar un flujo de datos a través de una tubería de forma muy similar a como una tubería real proporciona un flujo de agua; en ésta,

si se cerrara un extremo se interrumpiría el flujo. Esto mismo ocurre con los flujos que denominamos tuberías.

Para crear la estructura de la figura anterior, primero crearíamos un extremo de la tubería (extremo sobre el que trabajará el emisor) y después el otro conectado al anterior para formar la tubería (extremo sobre el que trabajará el receptor). El código necesario para realizar lo expuesto es el siguiente:

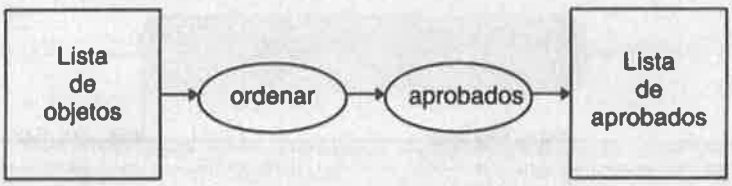
```
PipedWriter emisor = new PipedWriter();
PipedReader receptor = new PipedReader(emisor);
```

o bien:

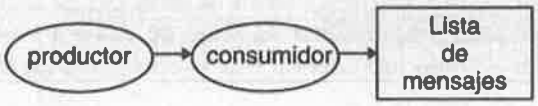
```
PipedReader receptor = new PipedReader();
PipedWriter emisor = new PipedWriter(receptor);
```

Por ejemplo, pensemos en una lista de objetos, relacionados con alumnos que cursan una determinada asignatura, que deseamos ordenar para después obtener una lista de los aprobados. Sin tuberías, el programa tendría que almacenar los resultados entre cada paso en algún lugar, por ejemplo, en matrices:

Con tuberías, la salida de un proceso se conecta directamente a la entrada del siguiente, según muestra la figura siguiente:



Dejamos este problema para que lo resuelva el lector. Nosotros vamos a resolver uno más breve que muestre simplemente cómo se utilizan las tuberías. Un hilo productor produce mensajes que pasa a través de una tubería a otro hilo consumidor para que los muestre en pantalla. La figura siguiente resume lo expuesto:



En primer lugar vamos a mostrar la aplicación que lanzará los hilos productor y consumidor:

```
import java.io.*;
public class Test
{
  public static void main(String[] args)
  {
    try
    {
      PipedWriter emisor = new PipedWriter();
      PipedReader receptor = new PipedReader(emisor);

      Productor productor1 = new Productor(emisor);
      Consumidor consumidor1 = new Consumidor(receptor);

      productor1.start();
      consumidor1.start();
    }
    catch (IOException ignorada) {}
  }
}
```

Se puede observar que el método **main** crea una tubería *emisor-receptor*. A continuación crea el hilo *productor1* y le pasa como argumento el extremo de la tubería por el cual debe de enviar los mensajes al hilo consumidor. Después crea el hilo *consumidor1* y le pasa como argumento el extremo de la tubería por el cual debe obtener los mensajes enviados por el hilo productor. Finalmente, lanza los dos hilos para su ejecución.

Mostramos a continuación la clase correspondiente al hilo productor. Esta clase tiene un atributo *emisor*, que referenciará el extremo de la tubería por lo que se enviarán los mensajes al hilo consumidor. Este atributo será establecido por el constructor de la clase.

Para enviar mensajes al consumidor, el método **run** del productor crea un flujo (*flujoS*) de la clase **PrintWriter** hacia el extremo *emisor*. Este flujo permitirá utilizar el método **println**, que **PipedWriter** no tiene, para enviar los mensajes por la tubería. La generación de los mensajes se simula igual que hicimos en la versión de la aplicación productor consumidor anterior. En este caso, por tratarse de una tubería, los mensajes producidos son enviados por la misma y puestos en cola mientras el consumidor los va recuperando.

```
import java.io.*;
public class Productor extends Thread
{
  private PipedWriter emisor = null;
```

```
  private PrintWriter flujoS = null;

  public Productor(PipedWriter em) // constructor
  {
    emisor = em;
    flujoS = new PrintWriter(emisor);
  }

  public void run()
  {
    while (true)
    {
      almacenarMensaje();
      try
      {
        int msegs = (int)(Math.random() * 100);
        // Poner a dormir el hilo hasta que se produzca el
        // siguiente mensaje.
        sleep(msegs);
      }
      catch (InterruptedException e) { }
    }
  }

  public synchronized void almacenarMensaje()
  {
    int númeroMsj;        // número de mensaje
    String textoMensaje; // texto mensaje

    númeroMsj = (int)(Math.random() * 100);
    // Suponer operaciones para buscar el mensaje en una tabla
    // de mensajes; resultado:
    textoMensaje = "mensaje #" + númeroMsj;
    flujoS.println(textoMensaje); // enviar mensaje por la tubería
    System.out.println("Productor  " + getName() +
                       " almacena: " + textoMensaje);
  }

  protected void finalize() throws IOException
  {
    if (flujoS != null) { flujoS.close(); flujoS = null; }
    if (emisor != null) { emisor.close(); emisor = null; }
  }
}
```

Finalmente, mostramos la clase correspondiente al hilo consumidor. Esta clase tiene un atributo *receptor*, que referenciará el extremo de la tubería desde el cual el hilo consumidor obtendrá los mensajes. Este atributo será establecido por el constructor de la clase.

Para obtener los mensajes, el método run del hilo consumidor crea un flujo (*flujoE*) de la clase BufferedReader desde el extremo *receptor*. Este flujo permitirá utilizar el método readLine, que PipedReader no tiene, para obtener los mensajes enviados por el productor. Cuando no haya ningún mensaje, simplemente el hilo que ejecuta el método readLine queda bloqueado.

```
import java.io.*;
public class Consumidor extends Thread
{
  private PipedReader receptor = null;
  private BufferedReader flujoE = null;

  public Consumidor(PipedReader re)  // constructor
  {
    receptor = re;
    flujoE = new BufferedReader(receptor);
  }

  public void run()
  {
    while (true)
    {
      obtenerMensaje();
    }
  }

  public synchronized void obtenerMensaje()
  {
    String msj = null;

    try
    {
      msj = flujoE.readLine(); // obtener mensaje de la tuberia
      System.out.println("Consumidor " + getName() +
                         " obtuvo:    " + msj);
    }
    catch (IOException ignorada) {}
  }

  protected void finalize() throws IOException
  {
    if (flujoE != null) { flujoE.close(); flujoE = null; }
    if (receptor != null) { receptor.close(); receptor = null; }
  }
}
```

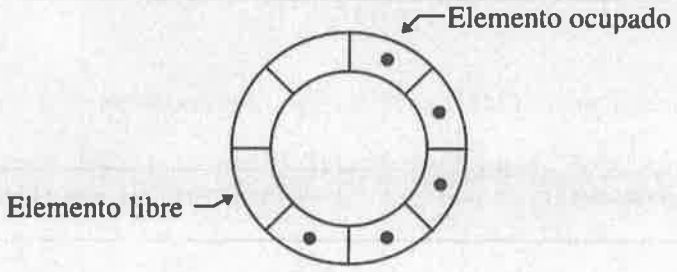
ESPERA ACTIVA Y PASIVA

Cuando se diseña un hilo, lógicamente no sólo pensamos en el trabajo que tiene que desempeñar, sino en cómo su trabajo puede verse afectado por otros hilos. De ahí el estudio de la sincronización de hilos. Pero no es menos importante pensar cómo tiene que comportarse el hilo como unidad individual de ejecución; esto es, poniéndonos en el caso de que durante espacios más o menos cortos, su ejecución no va a ser interferida por otros hilos. Si esto es así, los objetos de sincronización no serán requeridos por otros hilos y puede haber tiempo suficiente para que el hilo finalice su trabajo a la espera de que se den otros eventos que lo requieran de nuevo. En un caso como éste, la espera debe ser pasiva y no activa; es decir, no debe consumir tiempo de UCP.

En general un hilo realiza una espera pasiva poniéndose él mismo a dormir porque cuando está durmiendo, no entra en la planificación del sistema operativo; esto es, el sistema operativo no le asigna tiempo de UCP y, por consiguiente, detiene su ejecución. Tanto **wait** como **sleep** ponen un hilo a dormir; en el primer caso, para despertado hay que invocar a **notify** o **notifyAll**, o bien esperar a que transcurra el tiempo si se especificó, y en el segundo caso despierta cuando transcurra el tiempo especificado.

EJERCICIOS RESUELTOS

1. Realizar una aplicación que utilice dos hilos, un productor y un consumidor, trabajando sobre una única matriz de enteros positivos. Esto es, un hilo productor generará enteros que almacenará en una matriz circular y un hilo consumidor obtendrá de esa matriz los enteros generados por el productor. Muchas aplicaciones de la vida ordinaria reproducen este problema. Un ejemplo es el administrador de impresión en un servidor de red; los productores son los usuarios de la red y el consumidor la impresora o impresoras.



La figura anterior muestra un esquema del problema del *productor* y el *consumidor*. Para un correcta sincronización entre los hilos, el productor deberá blo-

quear la matriz sólo mientras se esté insertando un dato y el consumidor lo hará sólo mientras se esté extrayendo. Cuando la matriz esté vacía, el hilo consumidor se pondrá a esperar hasta que haya datos. Asimismo, cuando la matriz esté llena, el hilo productor se pondrá a esperar hasta que haya elementos libres.

La matriz que almacenará los datos será un objeto de la clase *CMatriz*. Esta clase estará formada por los atributos:

m	Matriz de *n* enteros positivos.
indProd	Índice del elemento donde el productor debe insertar el siguiente elemento. Su valor será: *0, 1, 2, ..., n-1, 0, 1, 2, ...*
indCons	Índice del elemento donde el consumidor debe obtener el siguiente elemento. Su valor será: *0, 1, 2, ..., n-1, 0, 1, 2, ...*
elementosVacíos	Número de elementos vacíos en un instante determinado.
elementosLlenos	Número de elementos llenos en un instante determinado.

y por los métodos:

almacenar	Almacena un dato en el siguiente elemento vacío.
obtener	Obtiene el siguiente dato aún no extraído.

El código correspondiente a esta clase se muestra a continuación:

```
//////////////////////////////////////////////////////////////////
// Sincronización de hilos: wait y notify.
//
public class CMatriz
{
  private int[] m;
  private int indProd = 0; // índice productor
  private int indCons = 0; // índice consumidor
  private int elementosVacíos, elementosLlenos;

  public CMatriz(int n)
  {
    if (n < 1) n = 10;
    m = new int[n];
    elementosVacíos = m.length;
    elementosLlenos = 0;
  }

  public synchronized void almacenar(int num)
  {
    // Esperar a que haya elementos vacíos
    while (elementosVacíos == 0)
    {
      try
      {
        wait(); // el hilo se pone a dormir y cede el monitor
```

```
        }
        catch (InterruptedException e) { }
      }
      elementosVacios--;
      elementosLlenos++;
      System.out.print("vacios: " + elementosVacios + ", llenos: " +
                       elementosLlenos + "  \r");
      m[indProd] = num;
      indProd = (indProd + 1) % m.length;
      // Despertar hilos;
      notifyAll();
    }

    public synchronized int obtener()
    {
      // Esperar a que haya elementos llenos
      while (elementosLlenos == 0)
      {
        try
        {
          wait(); // el hilo se pone a dormir y cede el monitor
        }
        catch (InterruptedException e) { }
      }
      elementosVacios++;
      elementosLlenos--;
      System.out.print("vacios: " + elementosVacios + ", llenos: " +
                       elementosLlenos + "  \r");
      int num =  m[indCons];
      indCons = (indCons + 1) % m.length;
      notifyAll();
      return num;
    }
}
//////////////////////////////////////////////////////////////////
```

En el problema del productor y del consumidor los recursos que estos hilos deben adquirir para poder ejecutarse son los elementos vacíos y los elementos llenos de la matriz, respectivamente. Cada uno de estos tipos de recursos los representaremos por sendas variables que actuarán como semáforos: *elementosLlenos* y *elementosVacíos*. Un valor cero equivale a semáforo en *rojo* y un valor distinto de cero a semáforo en *verde*.

elementosLlenos es un semáforo inicialmente en *rojo* para el consumidor (porque no hay ningún elemento lleno, esto es, no se puede *obtener*) que representa los elementos actualmente llenos de la matriz y *elementosVacíos* es un semáforo inicialmente en *verde* para el productor (porque todos los elementos están vacíos, esto es, se puede *almacenar*) que representa los elementos actualmente

vacíos de la matriz. Por lo tanto, el contador de *elementosLlenos* debe valer inicialmente cero y el de *elementosVacíos* debe valer *m.length*.

Cuando un hilo necesita un recurso de un tipo particular, decrementa el contador del semáforo correspondiente, y cuando lo libera lo incrementa. Por ejemplo, cuando el productor quiere *almacenar* un dato necesita el recurso "elementos vacíos", de tal forma que cada vez que lo adquiere lo decrementa; cuando llegue a cero implica semáforo en rojo indicando que el recurso "elementos vacíos" está ocupado. Lógicamente decrementar *elementosVacíos* implica incrementar *elementosLlenos*.

```
// elementosVacios es el semáforo para el productor
while (elementosVacios == 0)
{
  try
  {
    wait(); // el hilo se pone a dormir y cede el monitor
  }
  catch (InterruptedException e) { }
}
elementosVacios--;
elementosLlenos++;
m[indProd] = num;
indProd = (indProd + 1) % m.length;
notifyAll();
```

El código anterior, que pertenece al hilo productor, decrementa el contador del semáforo *elementosVacíos*, inserta un dato en el siguiente elemento vacío de la matriz y, lógicamente, incrementa el contador del semáforo *elementosLlenos*. Si el contador de *elementosVacíos* fuera cero, el hilo productor pasaría al estado bloqueado hasta que el hilo consumidor extraiga uno o más datos y, por consiguiente, incremente *elementosVacíos*. Un razonamiento análogo haríamos para el consumidor.

Según los expuesto, el hilo productor básicamente se limitará a llamar al método *almacenar* de la clase *CMatriz*. Esto es:

```
////////////////////////////////////////////////////////////////
// Sincronización de hilos. Hilo productor.
//
public class Productor extends Thread
{
  private CMatriz matriz;
  private boolean continuar = true;

  public Productor(CMatriz m) // constructor
  {
```

```
  public void terminar()
  {
    continuar = false;
  }
}
//////////////////////////////////////////////////////////////////
```

Para probar el comportamiento de ambos hilos puede servir la aplicación siguiente:

```
import java.io.*;
//////////////////////////////////////////////////////////////////
// Sincronización de hilos.
//
public class Test
{
  public static void main(String[] args)
  {
    CMatriz matriz = new CMatriz(10);
    Productor productor1 = new Productor(matriz);
    Consumidor consumidor1 = new Consumidor(matriz);

    System.out.println("Pulse [Entrar] para continuar y");
    System.out.println("vuelva a pulsar [Entrar] para finalizar.");

    InputStreamReader is = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(is);
    try
    {
      br.readLine(); // ejecución detenida hasta pulsar [Entrar]
      // Iniciar la ejecución de los hilos
      productor1.start();
      consumidor1.start();
      br.readLine(); // ejecución detenida hasta pulsar [Entrar]
    }
    catch (IOException e) {}
    // Permitir a los hilos finalizar
    productor1.terminar();
    consumidor1.terminar();
  }
}
//////////////////////////////////////////////////////////////////
```

EJERCICIOS PROPUESTOS

1. Escribir una aplicación que lance tres hilos que ordenen otras tantas matrices, todas de la misma dimensión, utilizando, el primero el método de ordenación de la burbuja, el segundo el de inserción y el tercero el método *quicksort*. Visualizar

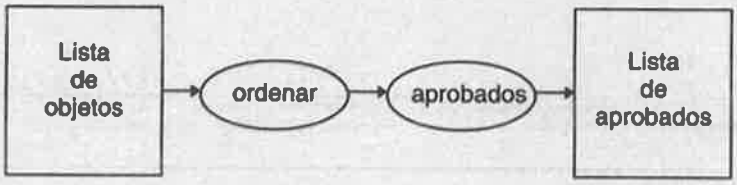
```
    matriz = m;
  }

  public void run()
  {
    int número; // número producido

    while (continuar)
    {
      número = (int)(Math.random() * 100);
      matriz.almacenar(número); // almacena el número
      //System.out.println("Productor  " + getName() +
      //                   " almacena: número " + número);
    }
  }

  public void terminar()
  {
    continuar = false;
  }
}
////////////////////////////////////////////////////////////////////
```

Análogamente, el hilo consumidor básicamente se limitará a llamar al método *obtener* de la clase *CMatriz*. Esto es:

```
////////////////////////////////////////////////////////////////////
// Sincronización de hilos. Hilo consumidor.
//
public class Consumidor extends Thread
{
  private CMatriz matriz;
  private boolean continuar = true;

  public Consumidor(CMatriz m)  // constructor
  {
    matriz = m;
  }

  public void run()
  {
    int número;
    while (continuar)
    {
      número = matriz.obtener();
      //System.out.println("Consumidor " + getName() +
      //                   " obtuvo:   número " + número);
    }
  }
```

como resultado el nombre de los métodos de ordenación colocados de más rápido a menos rápido.

2. Supongamos una lista de objetos, relacionados con alumnos que cursan una determinada asignatura, que deseamos ordenar para después obtener una lista de los aprobados. Utilizando tuberías, podemos plantear la solución del problema según muestra la figura siguiente:



Cada objeto alumno almacenará información relativa al nombre del alumno, al nombre de la asignatura y a la nota.

Para obtener el resultado solicitado, los pasos a seguir básicamente pueden ser los siguientes:

1. Crear el fichero con la información de los alumnos ordenada por el *nombre* del alumno.

2. Abrir un flujo desde el fichero que permita leer la información del mismo.

3. Invocar a un método *ordenar* que reciba como parámetro el flujo abierto en el punto 2 y devuelva una referencia a un objeto **PipedInputStream** (o a su superclase), que se corresponda con el extremo de una tubería en la que un hilo lanzado por este método coloque los alumnos clasificados por la *nota*. Para realizar la ordenación, el hilo cargará la información en una matriz, la ordenará y después la volcará en la tubería.

4. Invocar a un método *aprobados* que reciba como argumento el flujo de datos resultante del punto 3 y devuelva una referencia a un objeto **PipedInputStream** (o a su superclase), que se corresponda con el extremo de una tubería en la que otro hilo lanzado por este método coloque los alumnos aprobados.

5. Grabar el resultado obtenido en el punto 4 en otro fichero. Después, visualizar el fichero para comprobar el resultado obtenido.