Indice Unidad III – Programación Concurrente

Introducción 2
Ventajas del procesamiento multihilos 2
Desafíos en programación multihilos . . . 2
Exclusión mutua 3
Sincronización 3
Calendarización de hilos, punto muerto . . . 3
Hilos en Java 4
class ParImpar extends Thread . . 5
public class BounceThread 6
La interfaz Runnable 9
Interrupción de threads 9
Estados de un thread 10
Cómo salir del estado bloqueado . . . 13
Threads muertos, Threads servidores, Grupos . 13
Prioridades de los threads 14
Threads egoístas. 16
Sincronización, Comunicación sin sincronización . . 17
public class UnsynchBankTest . . . 18
Bloqueo de objetos 21
Los métodos wait y notify . . . 22
public class SynchBankTest 25
Puntos muertos 27
CANCELACIÓN DE UN HILO . . . 30
ESPERA A QUE UN HILO FINALICE . . . 32
Bloques sincronizados . . . 33
Métodos estáticos sincronizados . . 34
Por qué están censurados los métodos stop() y suspend()? 35
El modelo productor/consumidor . . 37
Caso 1 – Sincronización a nivel de instancia . 37
Caso 2 – Sincronización combinada de instancia/clase 40
Usando Swing como interfaz de usuario (Barra de progreso) 43
Evaluacion Final 06-12-2006 . . . 47

Autor: Ing. Tymoschuk, Jorge

Introducción

En una computadora secuencial sólo se ejecuta un programa en un momento determinado. La computadora únicamente tiene un CPU, o elemento de procesamiento (PE, Processing Element) que ejecuta un programa a la vez, y es a éste al que se le llama proceso, el cual consta de rutinas, datos, pila, código y estructuras del sistema operativo. Cuando la Máquina Virtual de Java interpreta su programa, ejecuta un proceso.

Una computadora secuencial puede realizar multiprogramación al reasignar rápidamente el PE entre diversos procesos, dando el aspecto de paralelismo, al ejecutarse concurrentemente. Aunque el verdadero paralelismo se logra en un computadora con varios PEs

Dentro de un proceso, el control suele seguir a un sólo hilo de ejecución, que por lo general empieza con el primer elemento de main, recorriendo una secuencia de instrucciones y terminando cuando se regresa al main. Éste es el modelo familiar de programación de un solo hilo.

Java también soporta varios hilos de ejecución o multihilos. Un proceso de Java puede crear y manejar, dentro de sí mismo, **varias secuencias de ejecución concurrentes**. Cada una de estas secuencias es un **hilo independiente** y todos ellos comparten tanto el espacio de dirección como los recursos del sistema operativo; cada hilo puede accesar a todos los datos y procedimientos del proceso, pero además tiene su **propio contador de programa y su pila de llamado a procedimientos**. Como entidad de ejecución independiente, un hilo es mucho más fácil de crear que un nuevo proceso. Por ello a un hilo se le conoce a veces como **proceso ligero**. En lo que sigue hablaremos de **proceso ligero o hilo** indistintamente.

Ventajas del procesamiento multihilos

Los programas de un solo hilo son la herramienta ideal para procesamientos con mínima interacción con el usuario. Por ejemplo, un proceso de actualización de un archivo de personal con altas, bajas, modificaciones, que solo pide al operador de consola algún parámetro de control o una acción por única vez("Monte cinta Catastro Empleados mes anterior en Unidad X003") mientras que los procesamientos interactivos, controlados por eventos, suelen incluir varias partes activas que naturalmente se ejecutan de manera independiente e interactúan o cooperan de alguna manera para alcanzar las metas. Por ejemplo, veremos un programa de animación, una bola que rebota en los márgenes de su frame, está organizado en clases independientes para controles de eventos, generación de gráficos y movimiento, conservación de resultados y estadísticas, etc. Un juego de video con un solo hilo de ejecución sería enormemente complicado si no es que imposible. Un programa con multiprocesamiento puede modelar cada una de estas partes con un hilo diferente.

Es más, la generación de gráficos y el procesamiento del control del usuario ocurren simultáneamente. Resulta difícil un control con una buena respuesta a esta concurrencia en un programa de un solo hilo.

También puede utilizar hilos para desacoplar actividades con velocidades de procesamiento muy diferentes.

Desafíos en programación multihilos

Básicamente, un programa de procesamiento multihilos tiene que coordinar varias actividades independientes y evitar la posibilidad de que se encimen entre sí. Estos programas incluyen cuatro aspectos nuevos e importantes que no se encuentran presentes en los programas comunes de un solo hilo:

- exclusión mutua,
- sincronización,
- calendarización

- punto muerto.

Una buena comprensión de estos conceptos le ayudará a escribir programas correctos de varios hilos.

Exclusión mutua

Los hilos de un programa suelen necesitar ayuda para lograr una cierta tarea. Por lo general, esta cooperación implica el acceso de diferentes hilos a las mismas construcciones del programa. Cuando varios hilos comparten un recurso común (campo, arreglo u otro objeto), puede darse el acceso simultáneo de más de un hilo. Lo que puede dar como consecuencia resultados erróneos e impredecibles. Por ejemplo, si un hilo lee un campo **flag** y un segundo hilo le asigna un valor, entonces el valor leído puede ser el antiguo o el nuevo. Peor aún, si los dos hilos asignan valores a **flag** simultáneamente, uno de los dos se pierde. Debido a que nunca pueden conocerse las velocidades relativas de los hilos concurrentes, el resultado de programas con procesamiento multihilos puede depender de cuál hilo se obtiene primero. Esta **condición de carrera** debe evitarse si su programa con varios hilos habrá de funcionar siempre correctamente.

Para prevenir estas situaciones, es necesario organizar un **acceso mutuamente exclusivo a cantidades compartidas**. Programando correctamente, sólo un hilo puede accesar al mismo tiempo la cantidad protegida por exclusión mutua (mutex).

Considere un equipo de programadores que trabajan en un proyecto, que consta de muchos archivos, paquetes de clases. Si dos de ellos trabajan en el mismo archivo en diferentes estaciones simultáneamente sucederá un desastre. (Para nosotros, programadores Java, el acceso debe ser exclusivo a nivel clase, mientras que podemos compartir a nivel paquete). Entonces, la exclusión mutua debe organizarse a nivel .class, responsabilidad del entorno de programación. Y, por supuesto, para que ese bloqueo tenga lugar nadie debe estar editando/compilando esa clase en ese momento. Obtenido el bloqueo, nadie puede obtener acceso a él antes de desbloquearlo.

Con la POO el recurso compartido y las operaciones que contiene a menudo pueden encapsularse en un objeto. Todas las operaciones críticas se programarán bloqueando/desbloqueando el objeto anfitrión para obligar la exclusión mutua. El intento de acceso a un objeto ya bloqueado por el acceso desde un hilo hilo-1 causará que todos los demás hilos hilo-2, hilo-3, ,hilo-n con pretensión de actualizar el mismo objeto deban aguardar que hilo-1 lo libere, desbloquee.

Sincronización

La exclusión mutua evita que los procesos corridos en hilos se superpongan. Esto es muy bueno, pero aún se necesita de un medio para que ellos se comuniquen y coordinen sus acciones, trabajen como un equipo. Los hilos procesan su código a velocidades independientes, impredecibles. Nunca se puede armar una programación en base al tiempo total que el PE necesitará para correr la parte de la aplicación en un hilo específico. Ni siquiera sabemos el orden en que la cpu le asignará PEs a varios hilos que deben trabajar simultáneamente. Por tanto, es necesario coordinar el orden en que se realizan alguna tareas. Si una tarea no debe empezar antes de que otras tareas terminen, es imprescindible garantizar que esto así ocurra.

Por ejemplo, imagine que cada hilo es un trabajador en una fábrica, y están elaborando un producto que requiere del trabajo de todos, pero cada uno puede hacer su parte cuando el inmediato anterior concluyó la suya. (*Seguro ya escuchó algo de producción en serie …*). Entonces un hilo debe esperar hasta que otro haya terminado la parte que le corresponde elaborar. A tal coordinación, dependiente del tiempo de actividades concurrentes se le llama **sincronización**, y requiere del **retardo de un hilo** hasta que ciertas **condiciones se cumplan**.

Se dice que un hilo esta bloqueado si la ejecución de su parte activa está demorada, se retarda hasta un momento posterior. Hay varias maneras de hacerlo.

Por ejemplo, la programación del hilo puede contener un ciclo que verifica permanentemente una cierta condición. Mientras la condición se mantenga verdadera (Así en Java, los ciclos se mantienen vivos, ejecutando, mientras la condición de control es verdadera). Cuando la evaluación de la condición da resultado falso, salimos del ciclo y entramos en la "parte activa" del hilo.

Un pequeño ejemplo en código, un cliente necesita un par de zapatos.

```
while (noHayZapato) {sleep(t);} // Ciclo de espera
```

donde noHayZapato es una variable compartida. A esta forma de proceder, que le podemos llamar espera ocupada, (Lo de espera ocupada es porque gastamos los milisegundos del PE durmiendo y preguntando por zapato), no es lo mejor. Java, en su plataforma 1, recomendaba suspenderlo (suspend)y luego retomarlo (resume), pero en el uso de esta mecánica aparecieron problemas. La plataforma Java 2 los ha censurado. Volveremos sobre esto.

Calendarización de hilos

Cuando un proceso requiere de varios hilos el PE disponible los ejecuta en rápida sucesión. Dependerá de la política de calendarización del sistema operativo sobre el cual corre la JVM (Java Virtual Machina) el momento en que el hilo concurrente cederá el PE y cuál hilo se ejecutará luego. Además de esto los hilos de Java tienen un atributo de prioridad que afecta la calendarización, como se explicará más adelante. Ahora entremos en detalle sobre el tema de cómo depende la calendarización del sistema operativo.

En los sistemas operativos existen dos políticas de calendarización

- Se interrumpe el hilo sin ningún tipo de consulta; se le acabaron sus milisegundos, afuera. A esto se le suele llamar **multitarea apropiativa** (preemptive multitasking). Es la más adecuada. Y así trabajan los sistemas operativos UNIX/Linux, Windows NT, Windows 95/98/XP (32 bits)
- Se interrumpe el hilo cuando otros desean obtener el control y el que está corriendo está de acuerdo: **multitarea cooperativa** (cooperative multitasking) En esta modalidad un programa mal codificado puede acaparar todos los recursos del sistema y nunca permitir que otros ejecuten.

Punto muerto

En una situación en que varios hilos son interdependientes de muchas maneras y comparten recursos bajo exclusión mutua y subtareas bajo sincronización, existe la posibilidad de llegar a un **punto muerto**, el cual surge cuando los hilos están esperando eventos que nunca sucederán. Por ejemplo, el hilo A está esperando datos del hilo B antes de producir una salida para B. A su vez, B está esperando recibir alguna salida de A antes de producir una salida de datos para A. Cuando se escriben programas de varios hilos, se debe tener cuidado de evitar estos problemas.

Hilos en Java

Un programa de Java empieza la ejecución con un flujo principal (el main del hilo de control de la aplicación o el applet de Java). El flujo principal o hilo padre puede producir otros para ejecutarlos concurrentemente. Estos, en Java, son objetos de una clase que:

- extiende la clase Thread
- implantan la interfaz Runnable.

El punto de entrada de un hilo es siempre su método **public void run()**; este método puede invocar otros, como cualquier método Java. El hilo termina cuando en el método run finaliza. Y para entrar en código veamos una aplicación muy simple que usa hilos.

El cuerpo del método run() de la clase Thread no hace nada. Entonces, cuando se extiende Thread, debemos redefinirlo para que ejecute el comportamiento deseado. En el ejemplo abajo tenemos un par de hilos ejecutando el método run() de ParImpar. Lo que hace run() es desplegar enteros pares o impares con un punto de inicio y retardo (delay) específicos para cada hilo, ambos informados vía parámetros en la instanciación de los objetos thr01 y thr02, en el main().

```
package javaapplication13;
public class Main {
    public static void main(String[] args) throws InterruptedException{
        System.out.println("Arrancamos main");
        ParImpar thr01 = new ParImpar(1, 20);
        ParImpar thr02 = new ParImpar(0, 30);
        System.out.println("Objetos instanciados, arrancamos thr01");
        thr01.start();  //  Arrancamos hilo thr01
        System.out.println("                      Arrancamos thr02");
        thr02.start() ;
        thr01.join() ;  //  Necesitamos que la leyenda terminamos
        thr02.join() ;  //  salga al fin de todo ...salga al fin de todo ...
        System.out.println("");
        System.out.println("Terminamos!");
    }
}

class ParImpar extends Thread{
    private int iO, delay;
    public ParImpar(int first, int interval){
        iO = first; delay = interval;
    }
    public void run(){
        String aux;
        try{
            for (int i=iO; i <= 20; i += 2 ){
                aux = i+", ";
                System.out.print(aux);
                sleep(delay);
            }
        }catch (InterruptedException e){return;}
    }
}
```

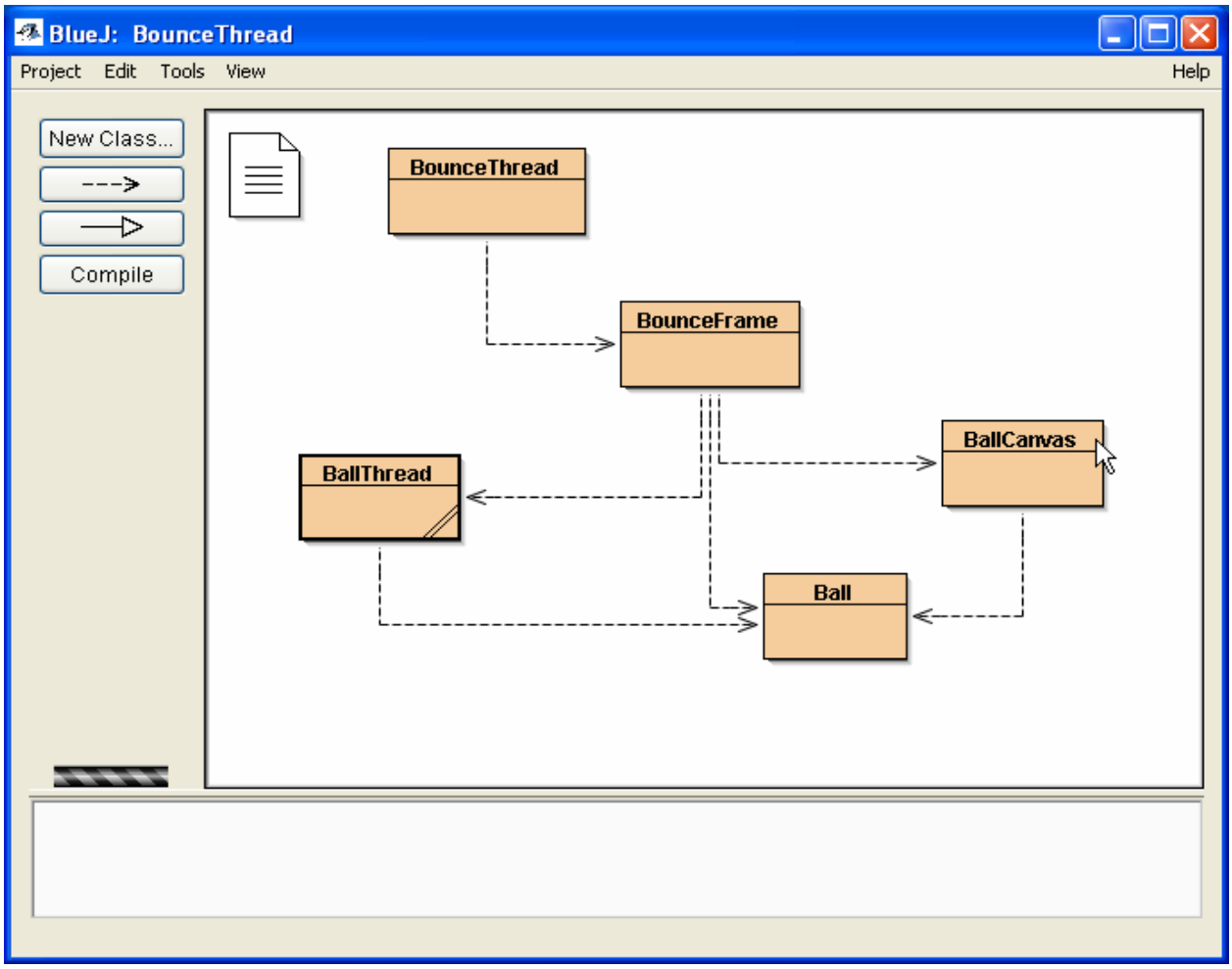
Copiamos/pegamos la ejecución:

```
run:
Arrancamos main
Objetos instanciados, arrancamos thr01
                      Arrancamos thr02
1, 0, 3, 2, 5, 4, 7, 9, 6, 11, 8, 13, 15, 10, 17, 19, 12, 14, 16, 18, 20,
Terminamos!
```

Como puede ver ambos hilos se ejecutan concurrentemente. La demora más corta utilizada para números impares hace que el hilo thr01 avance más rápido, por eso al final de la secuencia hay 5 pares seguidos.

Veamos ahora una aplicación donde el uso de hilos es imprescindible. Una animación. Una bola rebotando en los límites de su campo de acción.

Veamos como ve las relaciones entre clases BlueJ



La programación de esta animación requiere el concurso de varias clases.

- **Public class BounceThread:** Contiene el main() que instancia JFrame = new **BounceFrame**(), el hilo principal y ejecuta frame.show(). Nada mas.
- **class BounceFrame** extends JFrame), es el flujo o hilo principal, el cual se encarga de tomar nota de todos los eventos de la interfaz de usuario.
- **class BallThread** extends Thread, es el hilo que tiene el método run(), conteniendo un ciclo que comanda 1000 movimientos de la bola.
- **Class BallCanvas** extends JPanel, añade la bola al lienzo (JPanel)
- **Class Ball**, toda la lógica de movimiento de la bola, sus rebotes, …

A continuación el programa completo.

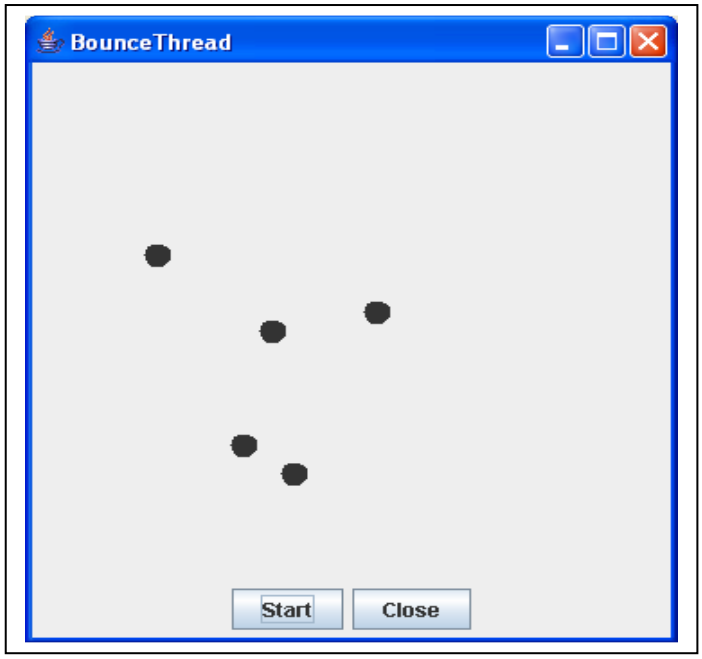
```
import javax.swing.*;
public class BounceThread{  //  "Rebote hilado..."
   public static void main(String[] args){
      JFrame frame = new BounceFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show(); //  Mostrando la estructura grafica
   }
}

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class BounceFrame extends JFrame{   //   Estructura con el canvas y botones
```

```
   private BallCanvas canvas;       //   Referencia a mi clase BallCanvas
   public static final int WIDTH = 450;
   public static final int HEIGHT = 350;

   /**
    * Construye la estructura con el area canvas para mostrar la bola
    * y los botones Start y Close.
    * La clase Canvas provee un area rectangular con capacidad para que
    * la aplicacion pueda dibujar graficos e imagenes
    * y atrapar eventos del usuario
   */

   public BounceFrame(){    //  Constructor
      setSize(WIDTH, HEIGHT);
      setTitle("BounceThread");

      Container contentPane = getContentPane(); // Defino contenedor
      canvas = new BallCanvas();            //    instancio BallCanvas
      // Incorporo canvas al contentPane, region CENTER
      contentPane.add(canvas, BorderLayout.CENTER);
      JPanel buttonPanel = new JPanel();    //    panel para botones
      // Incorporo boton start al buttonPanel (usando metodo add propio)
      addButton(buttonPanel, "Start",new ActionListener(){
         public void actionPerformed(ActionEvent evt){addBall();}});

      // Incorporo boton close al buttonPanel
      addButton(buttonPanel, "Close",new ActionListener(){
         public void actionPerformed(ActionEvent evt){System.exit(0);}});
      // Incorporo buttonPanel al contentPane, region SOUTH
      contentPane.add(buttonPanel, BorderLayout.SOUTH);
   }

   //   Mi metodo addButton propio
   public void addButton(Container c, String title, ActionListener listener){
      JButton button = new JButton(title);
      c.add(button);
      button.addActionListener(listener);
   }

   //   incorporo mi bola al canvas y arranco hilo de rebotes
   public void addBall(){
      Ball b = new Ball(canvas);
      canvas.add(b);
      BallThread thread = new BallThread(b);
      thread.start();
   }
}   //  class BounceFrame

//  El hilo para jugar con la bola...
class BallThread extends Thread{
   private Ball b;
   public BallThread(Ball aBall) { b = aBall; }

   public void run(){
      try{
         for (int i = 1; i <= 10000; i++){
            b.move(); sleep(5);
         }
      } catch (InterruptedException exception){}
   }
}   //  class BallThread

//  El canvas para dibujar la bola
import javax.swing.*;
```

```
import java.util.*;
import java.awt.*;
class BallCanvas extends JPanel{
    //   implementamos ArrayList, un array redimensionable vacio
    //  que puede contener objetos
   private ArrayList balls = new ArrayList();
   //  le apendamos una bola
   public void add(Ball b){balls.add(b);}

   public void paintComponent(Graphics g){  //  Dibujamos la bola
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D)g;
      for (int i = 0; i < balls.size(); i++){
         Ball b = (Ball)balls.get(i); b.draw(g2);
      }
   }
}   //  class BallCanvas

import java.awt.Graphics2D;
import java.awt.Component;
import java.awt.geom.*;
class Ball{  //  Clase bola
   private Component canvas;
   private static final int XSIZE = 15;  private static final int YSIZE = 15;
   private int x = 0; private int y = 0; private int dx = 2; private int dy = 2;

   public Ball(Component c) { canvas = c; } // Construimos una bola

   public void draw(Graphics2D g2){ // Dibujamos bola, posicion corriente
        g2.fill(new Ellipse2D.Double(x, y, XSIZE, YSIZE));
   }

   public void move(){   //  Movemos bola
      x += dx; y += dy;
      if (x < 0){x = 0; dx = -dx;}
      if (x + XSIZE >= canvas.getWidth()){
         x = canvas.getWidth() - XSIZE; dx = -dx;
      }
      if (y < 0){y = 0; dy = -dy;}
      if (y + YSIZE >= canvas.getHeight()){
         y = canvas.getHeight() - YSIZE; dy = -dy;
      }
      canvas.repaint();
   }
}
```

Cinco bolas rebotando…
Cada vez que se clickea Start estamos arrancando un nuevo hilo BallThread, que usa su propio objeto Ball para gestionar su bola. Todos los hilos comparten el mismo JPanel.

Es muy sencillo crear cualquier número de objetos autónomos que se estén ejecutando en paralelo.



La interfaz Runnable

Supongamos que necesitamos comportamiento de hilo en una clase que ya tiene otra superclase. Por tanto, no se puede derivar la clase Thread, pero si hacer que la clase implemente la interfaz Runnable. Aun cuando se haya derivado desde Thread, es posible colocar el código a ejecutar en el método run. Este es el caso de la clase **MasUnHilo**, que tiene su método run() sin derivar de Thread, arrancada dese el main() de class Main

```
package hiloscontando;
public class Main {
    public static void main(String[] args) {
        System.out.println("Startando hilos");
        for (int i=0; i<5; i++) {
            // Creamos 5 hilos, los
            // inicializamos p/intervalos 0/9, 10/19, ..., 90/99
            Thread t = new Thread(new MasUnHilo(i*10, (i+1)*10,i+1));
            t.start(); // start invoca a run de CountThreadTest
        }   // for
        System.out.println("Todos los hilos trabajando");
    }
}

class MasUnHilo implements Runnable{
    int from;   // Limite inferior del conteo.
    int to;     // Limite superior.
    int hilo;
    public MasUnHilo(int from, int to, int hilo){ // Constructor
        this.from   = from;
        this.to     = to;
        this.hilo   = hilo;
    }
    public void run(){
        System.out.print("Hilo # "+ hilo+", ");
        for (int i=from; i<to; i++){
            System.out.print(i + ", ");
        }
        System.out.println();
    }
}
```

Y una ejecución:

```
run:
Startando hilos
Hilo # 1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
Hilo # 2, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
Hilo # 3, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
Hilo # 4, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
Todos los hilos trabajando
Hilo # 5, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
BUILD SUCCESSFUL (total time: 0 seconds)
```

Tome nota de en que momento aparece la leyenda "Todos los hilos trabajando" Está bien? Era esperable? O hay algo errado?

Interrupción de threads

Un thread finaliza cuando su método run regresa. En la primera versión del entorno de programación Java existía también un método stop al que cualquier thread podía llamar para finalizar otro. Sin embargo, este método está censurado hoy en día. Veremos los motivos de ello más adelante.

No existe una forma fácil de forzar a que un thread finalice. Sin embargo, puede utilizarse el método interrupt para solicitar dicha finalización. Esto significa que el método **run de un thread tendrá que comprobar de vez en cuando si debe salir.** Algo así:

```
public void run(){
      while (no hay petición para terminar && hay más trabajo por hacer){
            seguir trabajando
      }
      // salir del método run y terminar el thread
}
```

Ademas, ya hemos dicho, un thread no debe estar continuamente trabajando, sino que debe "echarse a dormir" de vez en cuando para permitir que otros threads también trabajen. Pero cuando un thread está "durmiendo", no puede comprobar activamente si debe finalizar. Y es aquí donde entra en acción InterruptedException. Cuando en el thread thrd01 ejecutamos el método thrd02.interrupt() y thrd02 está durmiendo (bloqueado), dicho bloqueo (producido por sleep o wait) es finalizado en una InterruptedException.

No es un requisito del lenguaje que un thread que está interrumpido deba terminarse, la finalidad de la interrupción es llamar su atención. El thread puede decidir de qué modo tratar la interrupción situando las acciones pertinentes en la cláusula catch que manipula la excepción InterruptedException.

La alternativa es:

- ignorar la interrupción atrapando la excepción y continuando.
- Interpretar la interrupción como una solicitud de terminación.

El método run de estos threads tiene el siguiente aspecto:

```
public void run(){
      try{
            while (mientras hay trabajo por hacer){
                  seguir trabajando
            }
      }
      catch(InterruptedException exception){

            // Entramos aquí si "nos perturban la siesta",
            // o sea que el interrupt proveniente de thrd01
            // nos encuentra en medio de un sleept o wait.
      }
      finally{
            limpiar, en caso de ser necesario
      }
      // salir del método run y terminar el thread
}
```

Sin embargo, todavía hay un problema. Si el método interrupt no encuentra al thread "durmiendo" o "esperando", la InterruptedException no se generaría. Previendo esta situación el thread necesita llamar al método **interrupted** para determinar si ha sido interrumpido recientemente. Entonces modificamos la condición de permanencia del while

```
while (¡interrupted() && hay trabajo por hacer){
      seguir trabajando
}
```

En particular, si un thread fue bloqueado mientras esperaba la finalización de una operación de entrada/salida, esas operaciones **no** serían finalizadas por la llamada a interrupt. (A diferencia de sleep() o wait())

Cuando se vuelve de la operación de bloqueo, es necesario llamar a interrupted para saber si el thread actual ha sido interrumpido.

NOTA: existen dos métodos de nombre parecido:

- **interrupted:** es un método estático que comprueba si el thread actual ha sido interrumpido por solicitud de otro thread. Además, la llamada a interrupted reinicializa el thread, **lo saca de su estado interrumpido.**
- **islnterrupted:** es un método instanciado que puede usar para determinar si algún thread ha sido interrumpido. La llamada al mismo **no modifica el estado "interrumpido"** de su argumento.

En definitiva, existen dos formas de tratar la interrupción de un thread:

- comprobar el indicador de "interrumpido"
- capturar una InterruptedException.

Y como actúan en situaciones distintas que se dan el mismo hilo, debemos preocuparnos por ambas, lo que es bastante molesto.

Estados de un thread

Los threads pueden estar en uno de estos cuatro estados:

· nuevo

· ejecutable

· bloqueado

· muerto

Threads nuevos

Cuando se crea un thread con el operador new [por ejemplo, new Ball()), dicho thread todavía no está ejecutándose. Es un thread nuevo.

Threads ejecutables

Una vez que se ha llamado al método start, el thread es ejecutable, está en condiciones de ejecutarse, lo cual no implica que esté ejecutándose. Es el sistema operativo el que debe dar a ese thread ocasión de activarse. Cuando lo hace el código contenido en el thread está en ejecución. Java no distingue entre ejecutable y en ejecución, lo considera ejecutable.

Una vez que un thread está en ejecución, no es necesario que se mantenga en este estado siempre. De hecho, es aconsejable que se detenga de vez en cuando para dar oportunidad a otros threads de llevar a cabo sus tareas. Los detalles de planificación del thread dependen de los servicios que ofrezca el sistema operativo. (Ver calendarización de hilos). Si éste no coopera, la implementación de] thread hará lo mínimo para nevar a cabo e] trabajo multithreading. Sirva como ejemplo lo que se conoce como modelo de threads verdes (green threads) usados por Java 1.x sobre Solaris. Éste mantiene en ejecución un thread hasta que otro de una prioridad mayor "despierta" y toma el control. Otros sistemas, como el empleado por Windows 95 y Windows NT, otorgan a cada thread ejecutable una porción de tiempo para realizar su tarea.
Cuando dicha porción termina, el sistema operativo da el control a otro thread. Este planteamiento recibe el nombre de **división de tiempo**. La división de tiempo presenta una ventaja importante: un thread no cooperativo no puede evitar que otros threads se ejecuten. Las versiones actuales de la plataforma Java para Solaris pueden configurarse de modo que permitan el uso de los threads nativos de Solaris, o emplear la división de tiempo.

Tenga siempre en cuenta que un thread ejecutable puede, o no, ejecutarse en cualquier momento (ésta es la razón por la que este estado recibe el nombre de "ejecutable" y no el de "en ejecución").

Threads bloqueados

Un thread entra en estado bloqueado cuando se produce una de estas situaciones:

1. El thread realiza una llamada al método sleep() del thread.

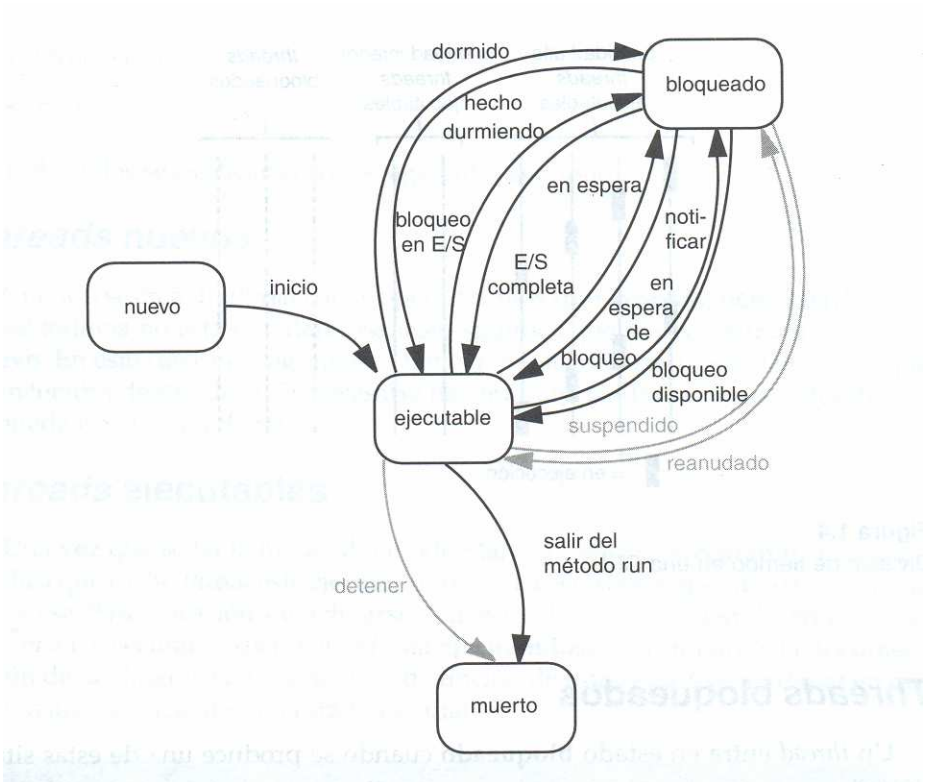
2. El thread llama a una operación de entrada/salida

3. El thread llama al método wait().

4. El thread intenta bloquear un objeto que ya se encuentra bloqueado por otro thread. Estudiaremos el bloqueo de objetos más adelante, en el punto Sincronización.

5. Otro thread ejecuta sobre el actual el método suspend(). Sin embargo, este método está censurado y no debería usarse. Veremos el porqué de ello más adelante.

La figura siguiente muestra todos los posibles estados de un thread y las transiciones entre ellos. Cuando un thread bloqueado se reactiva (debido, por ejemplo, a que ha "dormido" el número de milisegundos especificado o ha finalizado la operación de entrada/ salida de datos), el planificador comprueba si éste dispone de una prioridad más alta que la del thread que se está ejecutando en ese momento. En caso afirmativo, éste prevalece sobre el thread actual y comienza a ejecutarse. En computadoras con múltiples procesadores, cada uno de ellos puede ejecutar un thread, por lo que es posible tener varios ejecutándose en paralelo.



Por ejemplo, el método run() de BallThread se bloquea a sí mismo después de haber completado un movimiento, llamando al método sleep(). Esto posibilita a otros threads (en nuestro caso, a otras bolas y al thread principal) la posibilidad de ejecutarse.

Cómo salir del estado bloqueado

Un thread puede salir del estado bloqueado y regresar al de ejecutable siguiendo el camino inverso al recorrido para llegar al estado bloqueado:

1. Si un thread se ha puesto a "dormir", debe expirar el número de milisegundos especificado.

2. Si un thread está esperando una operación de entrada o de salida, ésta operación debe finalizar.

3. Si un thread llamó a wait, otro thread debe invocar a notifyAll o notify (trataremos los métodos wait y notifyAll/notify un poco más adelante).

4. Si un thread está esperando por un objeto bloqueado que es propiedad de otro thread, este último debe liberar ese bloqueo (verá los detalles de esta operación más adelante en este capítulo).

5. Si un thread ha sido suspendido, es necesario que se llame al método resume. Sin embargo, y ya que dicho método está censurado, es mejor no usarlo en los programas.

Si se invoca a un método de un thread que es incompatible con su estado, la máquina virtual dispara una IllegalThreadStateException.

Threads muertos

Un thread muere por alguna de estas dos razones:

. Naturalmente, debida a una salida normal del método run().

. Inesperadamente, una excepción no capturada que finaliza el método run().

Es posible matar un thread invocando al método stop. Dicho método dispara un error de tipo ThreadDeath que elimina dicho thread. Sin embargo, stop está censurado y no debería utilizado en los programas. Mas adelante explicaremos por qué el método stop es inherentemente peligroso.

Para determinar si un thread está vivo (es decir, si se encuentra en estado ejecutable o bloqueado), use el método isAlive, el cual devuelve true si está en cualquiera de estos dos estados, o false si el thread es nuevo o si ha muerto.

Existen algunas limitaciones en la capacidad de Java para determinar exactamente el estado de un thread. No es posible determinar si:

- Un thread vivo es ejecutable o está bloqueado.
- Un thread ejecutable está en ese momento ejecutándose.
- Un thread nuevo y otro que ya está muerto.

Threads servidores

Un thread puede convertirse en servidor (o daemon) invocandollamando a:

```
thrd.setDaemon(true);
```

Un servidor no es más que un thread cuya única finalidad es servir a otros. Ejemplos de ellos son los contadores, que envían "señales de reloj" a otros threads. O el recolector de basura…

Grupos de threads

Algunos programas pueden contener muchos hilos. Es útil categorizarlos en función de su cometido. Por ejemplo, considere un navegador de Internet. Si

varios threads intentan obtener imágenes desde un servidor y el usuario hace click en el botón Detener para interrumpir la carga de la página actual, sería muy útil disponer de alguna forma de interrumpir todos estos threads simultáneamente. Java permite la construcción de **grupos de threads**.

Para construir uno de estos grupos, se utiliza el constructor:

```
String groupName = . . .¡
ThreadGroup grp = new ThreadGroup(groupName)
```

El argumento de tipo cadena del constructor ThreadGroup identifica al grupo y debe ser único. A continuación, se pueden incorporar threads al grupo.

```
Thread t = new Thread(grp, threadName)¡
```

Para determinar aquellos threads de un grupo particular que aún permanecen ejecutables, utilice el método **activeCount**.

```
if (grp.activeCount() == 0){
      // Todos los threads del grupo grp están detenidos
}
```

Para detener todos los threads de un grupo, basta con llamar a interrupt en el objeto de dicho grupo.

```
grp.interrupt()¡ // Detiene todos los threads del grupo grp
```

Los grupos de threads pueden disponer de subgrupos hijo. Por defecto, cualquier nuevo grupo creado se convierte en hijo del grupo actual. Los métodos como activeCount() e interrupt() hacen referencia a todos los threads de su grupo y a los incluidos en los grupos hijo.

Una característica interesante de los grupos de threads es que se puede obtener una notificación en el caso de que uno de sus hilos muera debido a una excepción. Para ello, es necesario definir una subclase de la clase ThreadGroup y reemplazar el método uncaughtException. A partir de ese momento, puede sustituir la acción predeterminada (que muestra un volcado de la pila en el dispositivo estándar de error) con algo más personalizado, lo que Ud necesite.

Prioridades de los threads

En Java, cada thread tiene una prioridad, la cual, por defecto, se hereda de su thread padre. Se puede aumentar o disminuir esta prioridad a través del método **setPriority**. El valor de la misma puede oscilar entre

- MIN_PRIORITY (definido como 1 en la clase Thread)
- MAX_PRIORITY (definido como 10).
- NORM_PRIORITY tiene el valor 5.

Importante: En realidad, las reglas que se aplican a las prioridades son muy dependientes del sistema operativo. Cuando la máquina virtual cuenta con la implementación del thread de la plataforma anfitriona, el planificador está a merced de dicha implementación. La máquina virtual asigna las prioridades de los threads a los niveles de prioridad de la maquina anfitriona, los cuales pueden ser superiores o inferiores a los primeros. Lo que se describe en esta sección es una situación ideal a la que debería aproximarse cada implementación de la máquina virtual.

El thread ejecutable de mayor prioridad sigue ejecutándose hasta que:

- cede el control, invocando al método yield()
- deje de ser ejecutable (muerto, bloqueado)
- un thread de mayor prioridad ha pasado al estado ejecutable, porque:

- ha finalizado su tiempo de suspensión,
- se ha completado su operación de entrada/salida,
- se ha desbloqueado por la llamada a los métodos notifyAll / notify en el objeto por el que el thread estaba esperando.

A continuación, el planificador selecciona el nuevo thread a ejecutar, que será el de mayor prioridad de todos los que están en estado ejecutable.

¿Qué ocurre si existen varios threads ejecutables con la misma máxima prioridad? El planificador selecciona uno de ellos. Java no garantiza que todos los threads sean tratados imparcialmente. Desde luego, lo deseable sería que todos los que tuvieran la misma prioridad fueran servidos por orden, para garantizar que tuvieran la posibilidad de progresar. Pero es teóricamente posible que en algunas plataformas el planificador elija los threads de forma aleatoria, o que elija el primero de ellos disponible. Éste es uno de los puntos débiles del lenguaje de programación Java, precio a pagar por su portabilidad, por lo que resulta complicado escribir programas multithreaded que funcionen de forma idéntica en todas las máquinas virtuales.

Por ejemplo, Windows NT dispone de menos niveles de prioridad que los 10 especificados por Java. En dichas plataformas no importa el mapeo de los niveles de prioridad que se elija, ya que habrá alguno _de los 10 JVM que apuntará a los mismos niveles de dicha plataforma. En la JVM Sun para Linux, las prioridades de los threads son ignoradas por completo. (Que tal?)

Si la plataforma anfitriona usa niveles de prioridad inferiores a los de Java, un thread puede ser desalojado por otro con una prioridad aparentemente menor. O sea que si lo que Ud desea es diseñar uan aplicación “super portable”, no use mucho niveles de prioridad.

Considere, por ejemplo, una llamada a yield(). En algunas implementaciones, el planificador puede optar por continuar con el mismo thread. Nada se lo impide. Mejor usar sleep(), ya que al menos sabrá que el thread actual no será seleccionado de nuevo.

Muchas máquinas virtuales tienen varias prioridades (aunque no necesariamente 10), y los planificadores de threads realizan grandes esfuerzos para rotar alrededor de los que tienen las mismas prioridades, sin garantizarlo. Si UD. necesita una política de planificación concreta, deberá implementarla usted mismo.

Sugerimos el siguiente programa como ejemplo; si hace clic en el botón Start, se lanza un thread con una prioridad normal que anima la bola negra. Si hace clic en el botón Express, lanzará la bola roja, cuyo thread se ejecuta con una prioridad mayor que el resto.

```
public class BounceFrame{
      public BounceFrame(){
            addButton(buttonPanel, "Start",
                  new ActionListener(){
                        public void actionPerformed(ActionEvent evt){
                              addBall(Thread.NORM_PRIORITY, Color.black);
                        }
                  });

            addButton(buttonPanel, "Express",
                  new ActionListener(){
                        public void actionPerformed(ActionEvent evt){
                              addBall(Thread.NORM_PRIORITY + 2, Color.red);
                        }
                  });
}

public void addBall(int priority, Color color){
```

```
      Ball b = new Ball(canvas, color);
      canvas.add(b);
      BallThread thread = new BallThread(b);
      thread.setPriority(priority);
      thread.start();

}
}
```

Pruebe de arrancar un conjunto de bolas normal, y otro express. Observará que estas últimas parecen ejecutarse más deprisa. Esto es cuando el planificador hace lo esperado. Ahora pruebe en una máquina Linux y nos cuenta…

Si UD recuerda el código de este ejemplo: cinco milisegundos después de que un thread express se va a dormir, el planificador le despierta. Entonces:

- el planificador evalúa de nuevo las prioridades de todos los threads ejecutables;
- determina que el thread express tiene la mayor prioridad.

Cualquiera de los threads express consigue otro turno enseguida. Puede ser el único que se despierte, o quizá otro (no hay forma de saberlo). Los threads express toman turnos, y sólo cuando todos ellos están dormidos puede el planificador dar permiso a los threads de menor prioridad para que ejecuten sus labores.

La cosa podría ser peor. Observe que los threads de menor prioridad podrían no ejecutarse nunca si los express hubieran llamado a yield() en lugar de sleep().

Threads egoístas

Los threads de las bolas botando tenían un buen comportamiento y permitían que otros se ejecutasen. Para ello, llamaban al método sleep() para esperar sus turnos. Dicho método bloquea el thread y ofrece la oportunidad a los demás de ser planificados. Incluso, aunque un thread no se vaya a dormir por sí mismo, puede llamar a yield() siempre que no tenga inconveniente en ser interrumpido. Un thread siempre debe llamar a sleep() o yield() cuando se ejecuta durante un periodo largo de tiempo, para asegurar que no monopoliza el sistema. Aquellos thread que no siguen estas reglas reciben el nombre de **egoístas**.

El siguiente programa ilustra lo que ocurre cuando un thread contiene un bucle que monopoliza recursos de la CPU un montón de tiempo sin dar la posibilidad a otros threads de realizar el suyo. Cuando un evento torna **flag** verdadera, vea que ocurre.

```
class Cualquiera extends Thread{
      …
   private boolean flag;
   public void run(){
      try{
         for (int i = 1; i <= 1000; i++){
            b.move();
            if (flag){
               long t = System.currentTimeMillis();
               while (System.currentTimeMillis() < t + 5);
            }else sleep(5);
         }
      }
      catch (InterruptedException exception){}
   }
}
```

Cuando se activa **flag**, el método run durará alrededor de cinco segundos antes de volver, finalizando el thread. Mientras tanto, nunca llama a sleep o a yield.

Lo que ocurra entonces al ejecutar este programa depende del sistema operativo y de la elección en la implementación del thread. Por ejemplo, si ejecuta este programa bajo Solaris o Linux con la implementación "green threads" en lugar de los "threads nativos", el hilo "captura" la CPU. No habrá respuesta a eventos, en por lo menos 5 segundos. Sin embargo, si ejecuta el mismo programa en Windows o con la implementación de threads nativa, no ocurrirá nada inmanejable. El sistema operativo le está dando "oportunidades" a su hilo de eventos.

La razón de esta diferencia en el comportamiento reside en que el paquete subyacente de threads del sistema operativo lleva a cabo división de tiempo. Gracias a ello, se interrumpe periódicamente un thread en medio de un proceso. (para un buen sistema operativo no hay egoísmo que valga). El planificador activa otro hilo eligiéndolo de la lista de los ejecutables que tienen la prioridad mayor. La implementación green threads de Solaris y Linux no realiza división de tiempo, aun cuando el paquete nativo, de threads sí lo hagan.

Si sabe que el programa se ejecutará en una máquina cuyo sistema de threads realiza división de tiempo, no tendrá que preocuparse de construir políticas propias de threads. Pero si quiere que lo suyo sea portable a cualquier plataforma deberá prepararse para lo peor y efectuar llamadas a sleep o a yield en cada bucle.

Sincronización

En la mayoría de las aplicaciones multithreaded, no resulta extraño el que dos o más hilos deban compartir el acceso a los mismos objetos. Pero, ¿qué ocurriría si esos dos hilos que tienen acceso al mismo objeto llaman a un método que modifica el estado del mismo? Como puede imaginar, los threads pisan los pies de otro. Según el orden en el que se acceda a los datos, se pueden corromper los objetos. Esta situación suele recibir el nombre de condición de competencia.

Comunicación de threads sin sincronización

Para evitar accesos simultáneos a objetos compartidos por varios threads, debe aprender cómo sincronizar ese acceso.

Primero veremos que ocurre **si no sincronizamos**

Simularemos un banco que dispone de 10 cuentas, y generaremos transacciones aleatorias que muevan dinero entre ellas. Cada cuenta es gestionada por su propio thread, entonces tenemos 10. Cada transacción mueve una cantidad aleatoria de dinero desde la cuenta gestionada por uno de los thread a cualquier otra.

Disponemos de una clase Bank con un método transfer. Dicho método transfiere una cantidad de dinero de una cuenta a otra.

El método run de la clase TransferThread comanda movimientos de dinero desde una cuenta bancaria fija, la que se le asigna al hilo en el momento de su creación, y en cada iteración aleatoria mente elige una cuenta destino y una monto de dinero, llama a banco.transfer y se pone a dormir.

Cuando se ejecuta esta simulación, no conocemos la cantidad de dinero existente en cada cuenta. Pero lo que sí sabemos es que al final de todo el proceso la cantidad total **sum** no debe haber cambiado, ya que **todo el movimiento se realiza entre las 10 cuentas del banco.**

Cada 10.000 transacciones, el método transfer llama al método **test** para que recalcule el total y lo muestre. Este último contabiliza cuantas veces actúa. Todos los hilos terminan una vez que el test se ha realizado 15 veces.

```
/**
 *  Este programa muestra corrupcion de datos cuando
 * multiples hilos accesan la misma estructura de datos
 * Adecuaciones. Ing. Tymoschuk, Jorge
*/
public class UnsynchBankTest{
   public static final int NACCOUNTS = 10;  //  Cantidad de cuentas
   public static final int INITIAL_BALANCE = 10000; //  Balance inicial
   public static void main(String[] args){
      Bank bank = new Bank(NACCOUNTS, INITIAL_BALANCE);
      int i;
      for (i = 0; i < NACCOUNTS; i++){
         TransferThread t = new TransferThread(bank, i,
            INITIAL_BALANCE);
         t.setPriority(Thread.NORM_PRIORITY + i % 2);
         t.start();
      } //  for
   }    //  main
}       //  class

class Bank{     //  Un banco y sus cuentas
   public static final int NTEST = 10000;
   private final int[] accounts;
   private long nTransacts = 0;
   private int nTests     = 0;

   public Bank(int n, int initialBalance){  //
      accounts = new int[n];
      int i;
      for (i = 0; i < accounts.length; i++)
         accounts[i] = initialBalance;
      nTransacts = 0;
   }

   public void transfer(int origen, int destino, int monto)
      throws InterruptedException{  //  transferencia de dinero
      accounts[origen] -= monto;
      int cont = 0;
      for (int i=0;i<85000;i++)cont++; //  Separando debito/crédito
      nTransacts++;
      accounts[destino] += monto;
      if (nTransacts % NTEST == 0) test();
   }

   public void test(){    //  Verificando integridad del objeto Bank
      int sum = 0, ctasNeg=0;String texto;
      for (int i = 0; i < accounts.length; i++){
         sum += accounts[i];
         if(accounts[i]<0)ctasNeg++;
      }
      nTests++;
      texto = "Banco chequeado " + nTests + " veces, Suma: " + sum;
      texto+=", Ctas sdo neg. "+ctasNeg;
      System.out.println(texto);
   }

   public int size(){return accounts.length;}

   public int getTests(){return nTests;}
}

class TransferThread extends Thread{  //  Hilo que comanda transferencias
   private Bank banco;
   private int montoMax;
```

```
   private static final int REPS = 1000;

   public TransferThread(Bank banco, int ctaOrigen, int montoMax){
      this.banco = banco;
      this.montoMax = montoMax;
   }

   public void run(){
      try{
         while (banco.getTests() < 15 ){
            for (int i = 0; i < REPS; i++){
               int ctaOrigen  = (int)(banco.size() * Math.random());
               int ctaDestino = (int)(banco.size() * Math.random());
               int monto = (int)(montoMax * Math.random());
               banco.transfer(ctaOrigen, ctaDestino, monto);
               sleep(1);
            }   //  for
         }      //  while
      }         //  try
      catch(InterruptedException e) {}
   }            //  run
}
```

```
run:
Banco chequeado 1 veces, Suma: 96553, Ctas sdo neg. 6
Banco chequeado 2 veces, Suma: 99088, Ctas sdo neg. 6
Banco chequeado 3 veces, Suma: 100000, Ctas sdo neg. 6
Banco chequeado 4 veces, Suma: 99511, Ctas sdo neg. 6
Banco chequeado 5 veces, Suma: 96294, Ctas sdo neg. 5
Banco chequeado 6 veces, Suma: 100000, Ctas sdo neg. 6
Banco chequeado 7 veces, Suma: 93027, Ctas sdo neg. 5
Banco chequeado 8 veces, Suma: 100000, Ctas sdo neg. 4
Banco chequeado 9 veces, Suma: 93970, Ctas sdo neg. 5
Banco chequeado 10 veces, Suma: 100000, Ctas sdo neg. 4
Banco chequeado 11 veces, Suma: 100000, Ctas sdo neg. 3
Banco chequeado 12 veces, Suma: 91131, Ctas sdo neg. 3
Banco chequeado 13 veces, Suma: 100000, Ctas sdo neg. 4
Banco chequeado 14 veces, Suma: 99091, Ctas sdo neg. 4
Banco chequeado 15 veces, Suma: 92143, Ctas sdo neg. 3
BUILD SUCCESSFUL (total time: 35 seconds)
```

Como puede comprobar, hay algo raro. En casi la mitad de los tests la suma de los saldos es de 100.000 dólares, lo cual es correcto, ya que partimos de 10 cuentas con 10.000 dólares en cada una de ellas, y los movimientos se hicieron entre ellas. Algunas tendrán más de 10.000, otras menos. Pero después, esa suma de saldos cambia. Cuando ejecute este programa, puede ocurrir que esos errores ocurran rápidamente o que tarden bastante tiempo en "aparecer". Y, mas gracioso todavía, luego pueden "compensarse". Es como que el gerente lleva plata a casa de tanto en tanto, luego la repone… Desde luego, esta situación no inspira ninguna confianza, y es probable que no desee depositar el dinero que tan duramente ha ganado en este sospechoso banco (Si es que Ud confía en alguno…).

Esto, por supuesto, ocurre porque no nos hemos preocupado de sincronizar lo que hacen los hilos. Es mas, hemos "propiciado" en que estas irregularidades ocurrieran, demorando bastante en medio de la transferencia.

Como es esto?

Si Ud. analiza el método **run** del hilo **TransferThread** verá que muchísimas veces llama a banco.transfer(ctaOrigen, ctaDestino, monto). Por otra parte, el método **transfer(int origen, int destino, int monto)** de la clase **Bank** hace las transferencias. La situación irregular puede ocurrir si el método **test** muestra

un estado de suma de saldos de cuentas irregular cuando hay transferencias sin completar, o sea que algunos hilos han sido interrumpidos por tiempo por el sistema operativo sin dejarles terminar. (Esta situación ha sido "propiciada" con el bucle **for (int i=0;i<85000;i++)cont++;)** Claro que luego, cuando este hilo retome, hace la parte de la transferencia que faltaba y todo queda bien, en principio.

En realidad, en este ejemplo la situación tiende a compensarse, pero en otros casos puede ser mucho mas serio. Supongamos que haya muchos hilos "consumidores" que atienden clientes que compran vía Internet en un supermercado. Si el estoque es suficiente, autorizamos la compra, pero supongamos que el hilo se interrumpe. Otro cliente, usando otro hilo, compra el mismo producto, hay estoque suficiente, restamos la compra. Cuando vuelva a entrar en acción el hilo interrumpido, **no verifica de nuevo**, termina la operación, listo.

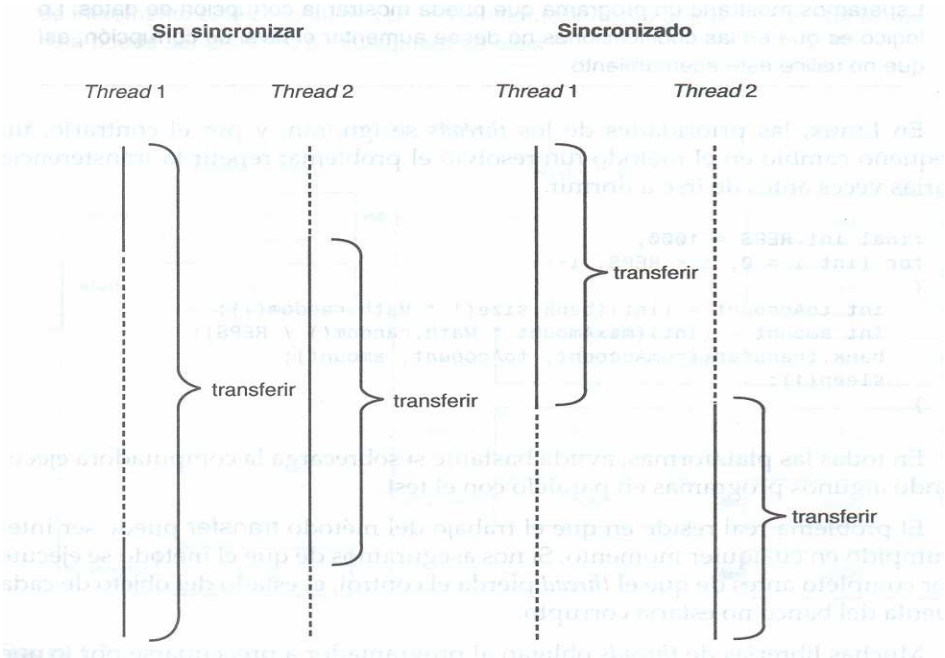
La pregunta es: Cuantas veces Ud compraría a un Super que le vende todo lo que quiere, le debita en su cuenta, pero a veces (Por fallas de sistema, claro …) no le envía todo?

El problema real reside en que el trabajo del método transfer puede ser interrumpido en cualquier momento. Si nos aseguramos de que el método se ejecute por completo antes de que el thread pierda el control, el estado del objeto de cada cuenta del banco no estaría corrupto.

Muchas librerías de threads obligan al programador a preocuparse por lo que se conoce como semáforos y secciones críticas para obtener acceso no interrumpido a un recurso. Esto basta para una programación procedural, pero es algo más difícil en la orientación a objetos. Java dispone de un bonito mecanismo inspirado por los monitores inventados por Tony Hoare.

Basta con agregar la cláusula synchronized en cada operación que no deba ser interrumpida. Entonces:

public **synchronized** void transfer(int origen, int destino, int monto)
Cuando un thread llama a un método sincronizado, existe la garantía de que dicho método terminará su cometido antes de que otro thread pueda ejecutar cualquier otro método sincronizado del mismo objeto. Cuando un thread llama a transfer y después otro intenta hacer lo mismo, el segundo no podrá continuar, y será desactivado y puesto en espera hasta que el primero termine.



En general, se etiquetarán como synchronized aquellos métodos que realicen múltiples operaciones de actualización en una estructura de datos, así como

aquellos que recuperen valores desde dichas estructuras. Esto le garantiza que esas operaciones se ejecutarán completamente antes de que otro thread pueda usar el mismo objeto.

Desde luego, el mecanismo de sincronización no es gratis. Se ejecuta algo de código de contabilidad cada vez que se llama a un método sincronizado. Por tanto, no resulta aconsejable sincronizar todos los métodos de una clase. Si los objetos no están compartidos por varios threads, no es necesario usar la sincronización. Si un método siempre devuelve el mismo valor para un objeto concreto, tampoco es necesario usar la sincronización en ese método. Por ejemplo, el método size de nuestra clase Bank no necesita ser sincronizado: el tamaño del objeto banco se fija una única vez, cuando es instanciado.

Ya que es caro utilizar la sincronización en cada clase, resulta más adecuado construir clases personalizadas para la comunicación de threads. Por ejemplo, suponga que dispone de un navegador que carga imágenes en paralelo y desea conocer el número de ellas que se han cargado. Puede definir una clase ProgressTracker con métodos sincronizados para actualizar y solicitar el estado de la carga.

Algunos programadores que tienen experiencia con otros modelos de threading se quejan de la sincronización Java, y la encuentran engorrosa e ineficiente.
Si nos ceñimos a una programación a nivel de sistema, esas quejas son comprensibles. Pero para los programadores de aplicaciones, el modelo Java funciona a las mil maravillas. Sólo recuerde utilizar clases que soporten la comunicación thread: no intente recortar código que ya exista rociándolo de algunas palabras synchronized.

NOTA

A veces, los programadores intentan evitar el coste de la sincronización cargando o almacenando operaciones independientes. Sin embargo, esto puede resultar peligroso por dos razones. La primera, la carga o el almacenamiento de un valor de 64 bits no está totalmente garantizado. Si hace una asignación a un campo double o long, podría efectuarse la mitad de esa asignación y después desalojar el thread.
El siguiente thread vería ese campo en un estado incoherente. Por otra parte, en una máquina multiprocesador, cada uno de ellos puede trabajar en una caché de datos separada de la memoria principal. La palabra reservada synchronized garantiza que las cachés locales sean coherentes con la memoria principal, cosa que una desincronización de métodos no hace. Esto puede provocar que un thread no vea una modificación efectuada por otro thread a una variable compartida.

La palabra reservada volatile está diseñada para controlar estas situaciones. Pero se tiene noticia de implementaciones de máquinas virtuales que no manipulan correctamente las variables volátiles. Entonces, para garantizar un thread seguro utilice la **sincronización** y no las variables volátiles.

Bloqueo de objetos

Cuando un thread llama a un método sincronizado, su objeto se "bloquea".
Piense en cada objeto como en una puerta que tiene un cerrojo en su interior.
Imagine una tradicional y cerrada cabina de teléfonos y suponga que tiene un pestillo en el interior. Cuando un thread introduce un método sincronizado, éste cierra la puerta y la bloquea. Cuando otro thread intenta llamar a un método sincronizado que referencia al mismo objeto, no puede abrir la puerta, así que detiene la ejecución. Finalmente, el primer thread termina su método sincronizado y desbloquea la puerta.

Periódicamente, el planificador de threads activa aquellos que están esperando debido al bloqueo de la puerta, usando las reglas de activación que ya hemos visto anteriormente. Siempre que uno de los threads que quiere llamar a un método sincronizado del objeto vuelve a ejecutarse, comprueba si dicho objeto

aún continúa bloqueado. En caso de no estado, ese thread lo captura para ser el siguiente que gane acceso exclusivo al mismo.

Por otro lado, otros threads son libres de llamar a los métodos no sincronizados de un objeto bloqueado. Por ejemplo, el método size de la clase Bank no está sincronizado y puede ser invocado (y ejecutado) sobre un objeto bloqueado.

Cuando un thread abandona un método sincronizado disparando una excepción, también cesa el bloqueo del objeto. Esto es bueno, ya que no queremos un thread que acapare el objeto una vez que haya terminado con el método sincronizado.

Si un thread tiene el bloqueo de un objeto y llama a otro método sincronizado del mismo objeto, se otorga inmediatamente el acceso a dicho método. El thread sólo anula el bloqueo cuando sale del último método sincronizado.

NOTA: Técnicamente, cada objeto dispone de un contador de bloqueo que registra cuantos métodos sincronizados ha llamado el "bloqueador". Cada vez que se efectúa una nueva llamada a un método, dicho contador se incrementa, y cuando finaliza un método sincronizado (ya sea por una terminación normal del mismo o por el lanzamiento de una excepción), el contador disminuye. Cuando se alcanza un valor cero, el thread libera el bloqueo.

Observe que es posible tener dos objetos diferentes de una misma clase bloqueados por distintos threads. Dichos threads pueden, incluso, ejecutar el mismo método sincronizado. Es el objeto el que está bloqueado, no el método. En nuestra analogía de la cabina de teléfono, se puede tener a dos personas en cabinas separadas. Cada uno de ellos puede ejecutar el mismo método sincronizado, o métodos diferentes. Esto no tiene inconvenientes.

Por supuesto, el bloqueo de un objeto sólo puede ser propiedad de un thread en un momento de tiempo concreto. Pero ese thread puede tener el bloqueo de varios objetos a la vez simplemente llamando al método sincronizado de un segundo objeto mientras se ejecuta el método sincronizado de otro objeto.

Los métodos wait y notify

Qué hacemos cuando no hay suficiente dinero en la cuenta? Pues esperamos hasta que otro thread añada fondos. Pero el método transfer está synchronized. Este thread tiene el acceso exclusivo al objeto banco, por lo que ningún otro tendrá la posibilidad de efectuar un depósito. Es en este punto donde entra en juego la segunda característica de los métodos sincronizados. Use el método wait del objeto Object si necesita esperar dentro de un método sincronizado.

Cuando se llama a wait dentro de uno de estos métodos, **el thread actual se bloquea y libera el bloqueo del objeto**. Esto permite a otro thread que pueda, si lo desea, aumentar el saldo de la cuenta.

El método wait puede disparar una InterruptedException cuando el thread es interrumpido mientras está esperando. En este caso, se puede activar el indicador de "interrumpido" o propagar la excepción (después de todo, es el thread y no el objeto bank el que debe decidir qué hacer ante una interrupción). En nuestro caso, simplemente propagamos la excepción y añadimos un especificador throws al método transfer.

```
public synchronized void transfer(int origen, int destino int monto)
      throws InterruptedException{

      while (banco[origen] < monto) wait();     //    ciclo de espera
      // transferir fondos
      …
}
```

Observe que wait es un método de la clase Object y no de Thread. Cuando llame a wait, bloquea el thread actual (Donde está corriendo el método que lo invoca) y el objeto banco se desbloquea.

ADVERTENCIA

Si un thread mantiene el bloqueo de varios objetos, la llamada a wait sólo desbloquea aquél en el cual se efectúa dicha llamada. Esto significa que el thread bloqueado puede mantener aún el bloqueo sobre otros objetos, los cuales no se desbloquearán hasta que se desbloquee el propio thread. Ésta es un situación peligrosa que debe evitar.

Existe una diferencia esencial entre un thread que está esperando para obtener un método sincronizado y otro que ha llamado a wait. Una vez que un thrend llama al método wait, entra en una lista de espera de ese objeto y se bloquea. Hasta que ese thread se elimine de esa lista, el planificador lo ignora y no tiene la posibilidad de seguir ejecutándolo.

Para eliminar un thread de la lista de espera, es necesario que otro thread llame a los métodos notifyAll o notify en el mismo objeto; notifyAll elimina todos los threads de la lista de espera del objeto; notify elimina uno elegido arbitrariamente.
Una vez que un thread ha sido eliminado de la lista de espera, vuelve a ser ejecutable, y el planificador podrá activarlo de nuevo. A su vez, podrá intentar entrar de nuevo en el objeto. Tan pronto como el objeto bloqueado esté disponible, cualquier thread intentará bloquearlo y continuar donde lo dejó tras la llamada a wait.

Para entender las llamadas wait y notifyAll/notify, hagamos una analogía con la cabina telefónica. Supongamos que un thread bloquea un objeto y después comprueba que no puede proceder debido a que el teléfono está roto.
Lo primero de todo, no tendría sentido para el thread el que se echara a dormir mientras bloquea el interior de la cabina. Si un operario de mantenimiento quisiera entrar para reparar el equipo, no podría. Llamando a wait, el thread desbloquea el objeto y espera fuera. Eventualmente, alguien del personal de mantenimiento entra en la cabina, la bloquea desde dentro y realiza sus tareas. Una vez que esta persona abandona la cabina, los threads que están esperando no estarán al tanto de esa situación (puede que la persona de mantenimiento sólo haya vaciado el recipiente de las monedas), y seguirán esperando, y esperando,... Llamando a notifyAll, el encargado informa a todos los threads que esperan que el estado del objeto ha cambiado. Con notify, el encargado elige uno de los threads de forma aleatoria y sólo le informa a él del cambio de estado, dejando a los demás en su desalentadora situación de espera.

Es fundamentalmente importante que algún otro thread llame a los métodos notify o notifyAll de forma periódica. Cuando un thread llama a wait, éste no tiene forma de desbloquearse, y debe depositar todas sus esperanzas en los otros threads. Si ninguno de ellos se preocupa de desbloquearlo, nunca más volverá a ejecutarse. Esto puede conducir a desagradables situaciones de bloqueos totales. Si todos los demás threads están bloqueados y el último que permanece activo llama al método wait sin desbloquear previamente alguno de los otros, entonces él también se bloqueará. En este caso, no existe ningún thread que pueda desbloquear a los demás, y el programa se "colgará". Los threads que esperan no son reactivados automáticamente cuando ningún otro thread está trabajando con el objeto. Veremos los bloqueos más adelante.

Como consejo práctico, es peligroso llamar a notify porque no tendrá control sobre el único thread que será desbloqueado. Si la operación se efectúa sobre el erróneo, ese thread puede que no sea capaz de proceder de forma adecuada. Como norma le recomendamos:

- Emplear el método **notify** si **cualquiera** de los hilos que están bloqueados por ese objeto podrá trabajar al ser notificado.
- Emplear el método **notifyAll** si **alguno** de los hilos que están bloqueados por ese objeto podrá trabajar al ser notificado.

Tanto notify como notifyAll no activan inmediatamente threads que esperan. Sólo desbloquean los threads que esperan para que éstos **compitan por la entrada** en el objeto una vez que el thread actual haya terminado con el método sincronizado.

Ninguno de los dos tipos de notificación tiene efecto retroactivo. Una notificación para un hilo que no está esperando no significa nada. Sólo las notificaciones que ocurran después de comenzar walt afectarán a un hilo en espera.

Estos métodos sólo se pueden invocar desde el interior de un código sincronizado, o desde un método previamente invocado desde otro sincronlzado.

Hay dos formatos "temporizados" para el método wait; uno es:

public final void **wait**(long timeout) throws Interrupted Exception.

El thread corriente debe poseer su objeto monitor.
Este método causa que el thread corriente se disponga a si mismo en un estado de espera para ese objeto, abandonando todos los requerimientos de métodos sincronizados sobre él. El thread corriente queda deshabilitado a los efectos de propósitos de planificación (*thread scheduling purposes*) y yace durmiente (*lies dormant*) hasta que ocurre algo de la siguiente:

- Algún otro thread invoca el método notify sobre este objeto y ocurre que este hilo es arbitrariamente elegido como el que corresponde despertar.
- Algún otro hilo invoca el método notifyAll sobre este objeto
- Algún otro thread interrumpe (*interrupts*) el thread corriente.
- El tiempo especificado ha transcurrido. Si timeout es cero, (O no informado) el tiempo no se toma en consideración y el thread corriente simplemente espera ser notificado.

Una vez que ha ocurrido alguno de estos cuatro puntos, el thread corriente es retirado del estado yaciente (o durmiente) respecto a su objeto y rehabilitado para la planificación de hilos. Entonces pasa a competir en la manera usual con los otros hilos por el derecho de sincronizar sobre el objeto (Bloquearlo); una vez que ha ganado control sobre el objeto, todos sus requerimientos de métodos sincronizados son restaurados al statu quo anterior, al existente al momento en que el método wait fue invocado. Entonces el thread corriente retorna de la invocación del método wait. Así, en el retorno del método wait, el estado de la sincronización es exactamente como era en el momento que wait fue invocado.

Si el thread corriente es interrumpido (interrumped) por otro thread mientras está aguardando, entonces se dispara una InterruptedException. La excepción no es disparada hata que el bloqueo sobre el objeto ha sido restaurado, como descrito anteriormente.

Note que el método wait, cuando setea el thread corriente en el estado de espera para su objeto, desbloquea únicamente este objeto; cualquiera otros objetos sobre los cuales el thread corriente puede estar sincronizado permanecen bloqueados mientras el thread aguarda.

CONSEJO

Si el programa multithreaded se "cuelga", revíselo y compruebe que cada wait tiene su correspondiente notify/notifyAll.

Cuando ejecutamos el programa de ejemplo con la versión sincronizada del método transfer, no se produce ningún error. El saldo total permanece siempre en 100.000 dólares.

También se puede observar que el programa con método synchronized funciona un poco más despacio; éste es el precio que se debe pagar por añadir la contabilización involucrada en el mecanismo de sincronización.

Forma de trabajo del mecanismo de sincronización. Resumen.

- Para llamar a un método sincronizado, el parámetro implícito no debe ser un objeto bloqueado. La propia llamada al método bloquea el objeto. El retorno desbloquea el objeto. Sólo un thread puede ejecutar un método sincronizado de un objeto particular cada vez.
- Cuando un thread ejecuta una llamada a wait, entrega el bloqueo del objeto y entra en la lista de espera de dicho objeto.
- Para eliminar un thread de esa lista de espera, es necesario que algún otro efectúe una llamada a notifyAll o a notify en el mismo objeto.

Las reglas de planificación son complejas, aunque comparativamente en la actualidad resulta muy sencillo ponerlas en práctica. Basta con seguir estas cinco recomendaciones:

- Si dos o más threads modifican un objeto, declare los métodos objeto de dicha modificación como sincronizados. Los métodos de sólo lectura que se **vean afectados** por la modificación del objeto también deben estar sincronizados.
- Si un thread debe esperar a que el estado de un objeto cambie, debe esperar dentro de dicho objeto, y no fuera, introduciendo un método sincronizado y llamando a wait.
- No gaste demasiado tiempo en la sincronización de un método. La mayor parte de las operaciones suelen actualizar una estructura de datos y volver inmediatamente. Si no quiere completar un método sincronizado inmediatamente, llame a wait para dejar ese objeto bloqueado mientras espera.
- Siempre que un método cambie el estado de un objeto, debería llamar a notifyAll. Esto da la oportunidad a los threads que están esperando de comprobar si la situación ha cambiado.
- Recuerde que wait y notifyAll/notify son métodos de la clase Object, y no de Thread. Compruebe cuidadosamente que las llamadas que realice a wait tienen su notificación en el mismo objeto.

Bueno, como quedaría finalmente nuestro método tranfer?. Le incorporamos:

- Cláusula **Synchronized**
- Ciclo de espera aguardando por saldo suficiente
- wait(), mientras no haya saldo
- notifyAll()

Y le hacemos algunos toques más, para conseguir que funcione. **En negrita**

```
/**
 *  Este programa resuelve la corrupcion de datos
 * que teníamos en UnSinchBankTest
 * Adecuaciones. Ing. Tymoschuk, Jorge
*/
public class SynchBankTest{
   public static final int NACCOUNTS = 10;  //  Cantidad de cuentas
   public static final int INITIAL_BALANCE = 10000; //  Balance inicial
   public static void main(String[] args){
      Bank bank = new Bank(NACCOUNTS, INITIAL_BALANCE);
      int i;
      for (i = 0; i < NACCOUNTS; i++){
         TransferThread t = new TransferThread(bank,
            INITIAL_BALANCE/100);
         t.setPriority(Thread.NORM_PRIORITY + i % 2);
         t.start();
      } //  for
```

```
   }    //  main
}       //  class

class Bank{     //  Un banco y sus cuentas
   public static final int NTEST = 10000;
   private final int[] accounts;
   private long nTransacts = 0;
   private int nTests     = 0;

   public Bank(int n, int initialBalance){  //
      accounts = new int[n];
      int i;
      for (i = 0; i < accounts.length; i++)
         accounts[i] = initialBalance;
      nTransacts = 0;
   }

    public synchronized void transfer(int origen, int destino, int monto)
      throws InterruptedException{  //  transferencia de dinero
        while(accounts[origen] < monto)
            wait();
        accounts[origen] -= monto;
        int cont = 0;
        // for (int i=0;i<85000;i++)cont++; //  Separando debito/crédito
        nTransacts++;
        accounts[destino] += monto;
        notifyAll();
        if (nTransacts % NTEST == 0) test();
   }

   public void test(){    //  Verificando integridad del objeto Bank
      int sum = 0, ctasNeg=0;String texto;
      for (int i = 0; i < accounts.length; i++){
         sum += accounts[i];
         if(accounts[i]<0)ctasNeg++;
      }
      nTests++;
      texto = "Banco chequeado " + nTests + " veces, Suma: " + sum;
      texto+=", Ctas sdo neg. "+ctasNeg;
      System.out.println(texto);
   }

   public int size(){return accounts.length;}

   public int getTests(){return nTests;}
}

class TransferThread extends Thread{  //  Hilo que comanda transferencias
   private Bank banco;
   private int montoMax;
   private static final int REPS = 1000;

   public TransferThread(Bank banco, int montoMax){
      this.banco = banco;
      this.montoMax = montoMax;
   }

   public void run(){
      try{
         while (banco.getTests() < 15 ){
            for (int i = 0; i < REPS; i++){
               int ctaOrigen  = (int)(banco.size() * Math.random());
               int ctaDestino = (int)(banco.size() * Math.random());
               int monto = (int)(montoMax * Math.random());
```

```
                banco.transfer(ctaOrigen, ctaDestino, monto);
                sleep(1);
             }   //  for
          }      //  while
       }         //  try
       catch(InterruptedException e) {}
   }             //  run
}
```

```
run:
Banco chequeado 1 veces, Suma: 100000, Ctas sdo neg. 0
Banco chequeado 2 veces, Suma: 100000, Ctas sdo neg. 0
Banco chequeado 3 veces, Suma: 100000, Ctas sdo neg. 0
Banco chequeado 4 veces, Suma: 100000, Ctas sdo neg. 0
Banco chequeado 5 veces, Suma: 100000, Ctas sdo neg. 0
Banco chequeado 6 veces, Suma: 100000, Ctas sdo neg. 0

Y aquí el programa se detiene …
```

Puede correrlo varias veces, nunca llega a 15 tests. Si reducimos el monto máximo a transferir, de **INITIAL_BALANCE/100**) a **INITIAL_BALANCE/500**, en varias pruebas realizadas si terminó la corrida. Analizaremos este problema a continuación.

Puntos muertos

La sincronización no resuelve todos los problemas que pueden producirse en el multithreading. Considere la siguiente situación:

Cuenta 1: saldo 2.000 dólares Cuenta 2: saldo 3.000 dólares

Thread 1: banco.transfer(1,2,3000); wait();

Thread 2: banco.transfer(2,1,4000);wait();

Supongamos que 1 y 2 son los únicos hilos que pueden correr. Pero ninguno de ellos puede transferir porque los saldos de las cuentas 1 y 2 son insuficientes. Eso es lo que nos pasaba en **SynchBankTest**. El programa no llega a las 70000 transferencias, parece tildado, en realidad estamos en un **punto muerto**, **abrazo mortal**, **deadlock**, hay muchos nombres y un solo problema.

En el lenguaje Java UD es el responsable de **eludir estos puntos muertos**. Por tanto, debe diseñar los threads de modo que estas situaciones no puedan producirse, y analizar el programa para asegurarse de que cada thread bloqueado pueda ser notificado por otro, y de esta manera siempre pueda proceder.

En el caso del ejemplo que estamos tratando, **SynchBankTest**, como podemos proceder? Normalmente un banco no atiende retiradas de sus cuentas cuando estas superan su saldo. Existen cuentas especiales que si autorizan el giro por un importe superior al saldo, pero limitándolo a un determinado importe, (Y cobran buenos intereses por el saldo rojo). En definitiva, estamos en el mismo caso. A modo de ejemplo modificamos los métodos transfer() y test():

- Transfer(): Transferirá si hay saldo suficiente; en caso contrario contabilizará como transacción no atendida (**contTransNoAte++**).
- Test(). Informará las transacciones no atendidas en el último intervalo.

Los dos métodos modificados quedan:

```
    public synchronized void transfer(int origen, int destino, int monto)
      throws InterruptedException{  //  transferencia de dinero
        if(accounts[origen] < monto)
            contTransNoAte++;
```

```
        else{
            accounts[origen] -= monto;
            // int cont = 0;
            // for (int i=0;i<85000;i++)cont++; //  Separando debito/crédito
            accounts[destino] += monto;
        }
        nTransacts++;
        notifyAll();
        if (nTransacts % NTEST == 0) test();
    }

    public void test(){    //  Verificando integridad del objeto Bank
        int sum = 0, ctasNeg=0;String texto;
        for (int i = 0; i < accounts.length; i++){
            sum += accounts[i];
            if(accounts[i]<0)ctasNeg++;
        }
        nTests++;
        texto = "Banco chequeado " + nTests + " veces, Suma: " + sum;
        texto+=", Ctas sdo neg. "+ctasNeg;
        texto+=", Trans no atend. "+contTransNoAte;
        contTransNoAte = 0;
        System.out.println(texto);
    }
```

Además hemos modificado el monto máximo a transferir, lo llevamos a **INITIAL_BALANCE/50**, porque con INITIAL_BALANCE/100 el contador de transacciones no atendidas fue siempre 0. El resultado de esta rodada:

```
run:
Banco chequeado 1 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 14
Banco chequeado 2 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 44
Banco chequeado 3 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 22
Banco chequeado 4 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 89
Banco chequeado 5 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 31
Banco chequeado 6 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 29
Banco chequeado 7 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 62
Banco chequeado 8 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 73
Banco chequeado 9 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 79
Banco chequeado 10 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 60
Banco chequeado 11 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 65
Banco chequeado 12 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 51
Banco chequeado 13 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 0
Banco chequeado 14 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 5
Banco chequeado 15 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 81
BUILD SUCCESSFUL (total time: 31 seconds)
```

Nota: Normalmente el software de gestión de bases de datos tiene capacidad de resolver puntos muertos y hacerlo en forma transparente para el usuario. Son sistemas que "loguean" las transacciones y en una situación como la descripta pueden retroceder lo actuado por la transacción de un cliente, terminar la del otro y relanzar la transacción retrocedida. El usuario solo percibe que una transacción suya tardó bastante más que lo habitual en algunos casos. Pero esto ya es tema de la asignatura Gestión de Datos.

ADVERTENCIA

Además de lo anteriormente descrito debe evitar las llamadas que bloquean el thread dentro de un método sincronizado, como, por ejemplo, una **llamada a una operación de entrada/salida**. En esta operación se libera el objeto, y se **bloquea el thread**. Si, eventualmente, todos los demás threads llaman a un método sincronizado de este objeto, dichos threads también se bloquearán y se producirá un punto muerto. Esto es lo que se conoce como un "agujero negro" (piense en

esto como en alguien que está dentro de una cabina de teléfono durante mucho tiempo mientras que otras personas esperan fuera para hacer sus llamadas).

Que pasa con un método no sincronizado que trata con el mismo objeto que otro sincronizado?. Es bloqueado o puede acceder al objeto mientras lo tiene transfer()?. Investiguemos. Tenemos en método test(), que es sincronizado y cada 10000 transacciones es llamado desde transfer(), también sincronizado. Podemos hacer lo siguiente.

- Le sacamos el carácter de sincronizado a test().
- Lo llamamos desde un método no sincronizado, desde el run() de TransferThread, inmediatamente después de la llamada al método transfer()

Las partes modificadas de la codificación anterior, en negrita.

```
 public synchronized void transfer(int origen, int destino, int monto)
   throws InterruptedException{  //  transferencia de dinero
     if(accounts[origen] < monto)
         contTransNoAte++;
     else{
         accounts[origen] -= monto;
         Thread.sleep(1);  //Tiempo para test() encontrar "corrupción"
         accounts[destino] += monto;
     }
     nTransacts++;
     notifyAll();
     // if (nTransacts % NTEST == 0) test(); la trasladamos a
}
public void run(){
   try{
      while (banco.getTests() < 15 ){
         for (int i = 0; i < REPS; i++){
            int ctaOrigen  = (int)(banco.size() * Math.random());
            int ctaDestino = (int)(banco.size() * Math.random());
            int monto = (int)(montoMax * Math.random());
            banco.transfer(ctaOrigen, ctaDestino, monto);
            if (banco.nTransacts % Bank.NTEST == 0)
                banco.test();
            sleep(1);
         }   //  for
      }      //  while
   }         //  try
   catch(InterruptedException e) {}
}            //  run
```

Y el resultado obtenido:

```
run:
Banco chequeado 1 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 12
Banco chequeado 2 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 0
Banco chequeado 3 veces, Suma: 99922, Ctas sdo neg. 0, Trans no atend. 92
Banco chequeado 4 veces, Suma: 99924, Ctas sdo neg. 0, Trans no atend. 74
Banco chequeado 5 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 92
Banco chequeado 6 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 92
Banco chequeado 7 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 180
Banco chequeado 8 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 35
Banco chequeado 9 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 58
Banco chequeado 10 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 106
Banco chequeado 11 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 87
Banco chequeado 12 veces, Suma: 99842, Ctas sdo neg. 0, Trans no atend. 67
Banco chequeado 13 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 101
Banco chequeado 14 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 90
Banco chequeado 15 veces, Suma: 100000, Ctas sdo neg. 0, Trans no atend. 106
BUILD SUCCESSFUL (total time: 5 minutes 1 second)
```

CANCELACIÓN DE UN HILO

A menudo necesitamos cancelar un trabajo antes de que se complete. El ejemplo más obvió es la pulsación de un botón de cancelar en una interfaz de usuario. Hacer que un hilo sea cancelable complica la programación, pero es un mecanismo limpio y seguro para finalizar un hilo. La cancelación se solicita interrumpiendo el hilo y escribiendo dicho hilo de forma que espere y responda a las interrupciones. Por ejemplo:

Hilo 1
```
thread2.interrupt();
```

Hilo 2
```
      while (!interrumpido()){
      // trabajando ...
}
```

Al interrumpir al hilo le avisamos de que deseamos que ponga atención, generalmente para detener su ejecución. Una interrupción no fuerza a un hilo a detenerse, aunque interrumpe el sueño de un hilo durmiente o en espera.

Las interrupciones son también útiles cuando deseamos dar al hilo en ejecución algún control sobre cuándo gestionará un evento. Por ejemplo, un bucle de actualización de pantalla podría necesitar acceder a la información de alguna base de datos utilizando una transacción y eventualmente necesitaría gestionar una acción de "cancelar" por parte del usuario a esperar a que la transacción se complete normalmente. El hilo de la interfaz de usuario podría implementar un botón de "Cancelar" interrumpiendo al hilo de presentación en pantalla para dar el control a dicho hilo. Esta solución funcionará correctamente siempre que el hilo de presentación en pantalla se comporte adecuadamente: Compruebe al final de cada transacción si ha sido interrumpido, deteniéndose si ha sido así.

Los métodos que se relacionan con la interrupción de los hilos son:

- **interrupt()**, que envía una interrupción a un hilo;
- **isInterrupted()**, que comprueba si un hilo ha sido interrumpido;
- **interrupted()**, un método static que comprueba si el hilo actual ha sido interrumpido y borra el estado de "interrumpido" de dicho hilo.

El estado "interrumpido" de un hilo sólo puede ser borrado por dicho hilo. No hay forma de borrar el estado "interrumpido" de otro hilo. Generalmente, no tiene mucha utilidad preguntar por el estado respecto a las interrupciones de otro hilo, por lo que estos métodos son normalmente utilizados por el hilo en curso sobre sí mismo. Cuando un hilo detecta que ha sido interrumpido, es frecuente que necesite realizar algún trabajo de limpieza antes de responder realmente a la interrupción. Este trabajo de limpieza podría involucrar acciones que podrían verse afectadas si el hilo se deja en un estado interrumpido. Por lo tanto el hilo comprobará y borrará su estado "interrumpido" utilizando interrupted().

Interrumpir a un hilo normalmente no afectará a lo que está haciendo, pero algunos métodos, como sleep() y wait(), lanzarán una excepción InterruptedException. Si nuestro hilo está ejecutando uno de estos

métodos cuando es interrumpido, el método lanzará la InterruptedException. El lanzamiento de esa excepción borra el estado "interrumpido" del hilo. El código de gestión de esta excepción podría ser algo como esto:

```
class prueCiclin{
     static void ciclin(int cuenta, long pausa){
          try {
               for (int i = O; i < cuenta; i++){
                    System.out.print(" ");
                    System.out.print(i) ;
                    System.out.flush() ;
                    Thread.sleep(pausa) ;
               }
          catch(InterruptedException e)(
               Thread.currentThread().interrupt();
          }

     }
     public static void main(String args[]){
          ciclin(lO,lOOO);
     }
}    // prueCiclin
```

```
Un capture
Class Path - .;C:\kava4.01'kawaclasses.zip;f:\

0 1 2 3 4 5 6 7 8 9
Process Exit. .
```

El método ciclin imprime el valor de i cada pausa milisegundos, hasta un máximo de cuenta veces. Si algo interrumpe al hilo donde se lo ejecuta, sleep lanzará una InterruptedException. La presentación sucesiva de valores finalizará, y la cláusula catch volverá a interrumpir el hilo. Alternativamente, podemos declarar que el propio prueCiclin lanza una InterruptedException y dejar simplemente que la excepción se filtre hacia arriba, pero en este caso cualquiera que invoque al método tendría que gestionar la misma posibilidad. La reinterrupción del hilo permite que prueCiclin limpie su propio comportamiento y permite después a otro código gestionar la interrupción como haría normalmente.

En general, cualquier método que realice una operación de bloqueo (directa o indirectamente) debería permitir que dicha operación de bloqueo se cancelara con interrupt() y debería lanzar una excepción apropiada si eso ocurre. Esto es lo que hacen sleep() y wait(). En algunos sistemas, las operaciones de E/S bloqueadas responderán a una interrupción lanzando InterruptedException (que es una subclase de la clase más general IOException, que la mayoría de los métodos de E/S pueden lanzar). Incluso si no se puede responder a la interrupción durante la operación de E/S, los sistemas pueden comprobar la interrupción al principio de la operación y lanzar la excepción. De aquí la necesidad de que un hilo interrumpido borre su estado "interrumpido" si necesita realizar operaciones de E/S como parte de sus operaciones de limpieza. Sin embargo, en general no podemos suponer que interrupt() desbloqueará a un hilo que está realizando E/S.

Todos los métodos de todas las clases se ejecutan en algún hilo, pero el comportamiento definido por esos métodos concierne generalmente al

estado del objeto involucrado, no al estado del hilo que ejecuta el método. Teniendo esto en cuenta, ¿cómo escribiríamos métodos que permitan que los hilos respondan a interrupciones y sean cancelables? Si nuestro método se puede bloquear, debería responder a una interrupción, como se ha comentado. En otro caso debemos decidir qué interrupción, o cancelación, tendría sentido para nuestro método y hacer que ese comportamiento sea parte su contrato. En la mayoría de los casos los métodos no necesitan preocuparse en absoluto de las interrupciones. Sin embargo, la regla de oro es no ocultar nunca una interrupción borrándola explícitamente, y nunca capturar una InterruptedException y continuar normalmente (esto impide que los hilos sean cancelables cuando ejecutemos nuestro código).

El mecanismo de interrupciones es una herramienta que puede utilizar el código que coopera para hacer que el uso de múltiples hilos sea efectivo.

///////// Aquí necesitamos un ejemplo donde un hilo interrumpa a otro, verifique si efectiva mente logró interrumpirlo y luego el hilo interrumpido se ponga en marcha nuevamente.
Usar interrupt(), isInterrumped(), interrumped(). /////////

ESPERA A QUE UN HILO FINALICE

Un hilo puede esperar a que otro finalice utilizando el método join(), en cualquiera de sus variantes. La forma no parametrizada espera indefinidamente a que el hilo finalice.

```
public final void join(long milis) throws InterruptedException

public final vold join(long milis, int nanos) throws InterruptedException

public final void join() throws InterruptedException
```

```
package demojoin;
public class Main {
    public Main() {
    }
    public static void main(String[] args) {
        Factorial fact = new Factorial(5);
        fact.start();
        try{
            fact.join();
            int aux=fact.getResultado();
            System.out.println("el factorial de 5 es "+aux);
        }catch (InterruptedException e){
            System.out.println("problema método join()");
        } // bloque try
    }// main ()
}         // class Main

class Factorial extends Thread {
    int resultado,numero;
    Factorial(int num){
        numero=num;
    }
    public void run(){
```

```
        try{
            factorial(numero);
        }catch (InterruptedException e){
            System.out.println("problema método wait");
        }   // bloque try
    }
    public synchronized int getResultado() throws InterruptedException{
        return resultado;
    }

    public synchronized void factorial(int lim) throws
                                        interruptedException{
        for(int i=1; i<=lim; ++i)
           resultado = resultado * i;
    }
}
```

Corriendolo tal cual

```
run:
el factorial de 5 es 120
BUILD SUCCESSFUL (total time: 0 seconds)
```

Comentarizando: // fact.join()

```
run:
el factorial de 5 es 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

En este ejemplo, en la primera corrida **fact.join()** indica que el código a seguir se ejecutará luego de la finalización del metodo fact(), que corre en el hilo Factorial. El método getResultado() nos retornará el resultado del calculo y todo ocurre como queremos.
En la segunda corrida, el main() arranca fact() y continúa. No espera que concluya. Entonces getResultado() puede retornar cualquier valor: inicial(en este caso), intermedio, final…

Ocasionalmente, puede resultar útil bloquear un objeto y obtener acceso exclusivo a él en algunas instrucciones sin tener que escribir un nuevo método sincronizado. Para ello, use los bloques sincronizados. Un bloque sincronizado consiste en una secuencia de instrucciones encerradas entre { . . . } y prefijadas con synchronized (objeto), donde objeto es el objeto que debe ser bloqueado. Un ejemplo de la sintaxis:

```
public void run(){
      . . .
      synchronized (banco){ // bloquea el objeto banco
      if (banco.getBalance(origen) >= monto)
            banco.transfer(origen, destino, monto);
      }
}
```

En este fragmento de código, el bloque sincronizado se ejecutará por completo antes de que cualquier otro thread llame a un método sincronizado del objeto banco. La idea es que el código sincronizado sea el mas breve posible.

Métodos estáticos sincronizados

Una **singleton** es una clase con un solo objeto. Las singletons suelen emplearse normalmente para el mantenimiento de objetos que deben ser únicos globalmente, como colas de impresión, gestores de conexiones a bases de datos, etc.

Una implementación típica:

```
public class Singleton{
      private Singleton (. . .) { . . . }
      private static Singleton instance;
      public static Singleton getInstance(){
            if (instance == null) instance = new Singleton(. . .);
            return instance;
      }
}
```

Sin embargo, el método **getlnstance** no es seguro desde el punto de vista de los threads. Supongamos un thread llamando a getlnstance que es desalojado en mitad del constructor, antes de que el campo instance haya sido definido. A continuación, un segundo thread obtiene el control y llama a getlnstance. Como instance aún sigue siendo null, dicho thread construirá un segundo objeto. Esto es precisamente lo que un singleton se supone que evita.

El remedio es sencillo: defina el método getlnstance como sincronizado:

```
public static synchronized Singleton getlnstance(){
      private Singleton (. . .) { . . . } //    Constructor
      private static Singleton instance;
      if (instance == null) instance = new Singleton(. . .);
      return instancej
}
```

Ahora, ese método se ejecuta por completo antes de que cualquier otro thread pueda llamarlo.

Y adonde está el objeto, si estamos trabajando con métodos estáticos? Sorprendido? Cuando un thread llama a un método sincronizado, éste obtiene el bloqueo del objeto. Pero este método es estático. Entonces, ¿qué objeto otorga el bloqueo al thread cuando se invoca a Singleton.getlnstance()?

La llamada a un método estático bloquea el objeto clase Singleton.class (Aprovechamos para refrescarle que existe un único objeto de tipo Class que describe cada clase que la máquina virtual puede cargar). A pesar de todo, si un thread llama a un método estático sincronizado de una clase, todos los métodos estáticos sincronizados de esa clase son bloqueados hasta que se retorna de la primera llamada.

Recapitulando, métodos de JAVA.LANG.OBJECT

Los tres métodos abajo descriptos sólo pueden ser llamados desde dentro de un método sincronizado o bloqueado, disparando una IIlegalMonitorStateException si el thread actual no tiene el bloqueo del objeto.

- void wait() // Provoca que un thread espere hasta que se le notifique.
- void notifyAll() //Desbloquea todos los threads que llamaron a wait en este objeto.
- void notify() // Desbloquea un thread elegido aleatoriamente de entre todos los que usaron wait en este objeto.

¿Por qué están censurados los métodos stop() y suspend()?

La plataforma Java 1.0 definió un método **stop**, que simplemente daba por finalizado un thread, y otro **suspend**, que lo bloqueaba hasta que otro thread lla maba a resume. Ambos están censurados en Java 2. El método stop() es inherentemente inseguro, y la experiencia ha demostrado que suspend() suele producir puntos muertos con bastante frecuencia.

Cuando stop() detiene un thread, el método que en ese thread corría desbloquea todos sus objetos, sin chequear la coherencia de su estado. Por ejemplo, suponga que TransferThread es detenido en la mitad de la transferencia de dinero de una cuenta a otra, después del reembolso pero antes del depósito. En este momento, el objeto banco está dañado, hay una operación de transferencia no terminada, hay dinero que ha desaparecido. Dicho daño afecta a los otros threads no detenidos.

ADVERTENCIA

Técnicamente hablando, el método stop() provoca la detención del thread mediante el lanzamiento de una excepción de tipo ThreadDeath, que finaliza todos los métodos pendientes incluyendo run.

Por la misma razón, cualquier excepción sin capturar en un método sincronizado puede provocar que ese método finalice prematuramente y provoque daño en el objeto.

Cuando un thread procede a detener a otro simplemente usando **stop()**, no tiene conocimiento de la coherencia del estado de su objeto; **stop() no toma ningún cuidado, puede provocar daños** en los objetos. Ésta es la razón por la que este método ha sido censurado.

Si necesita detener un thread de una forma segura, tendrá que hacer que dicho thread compruebe periódicamente una variable que indique si se ha solicitado su detención. Es como decir que se requiere del consenso del thread a ser detenido para efectivamente concretar esta detención.

```
public class MyThread extends Thread{
      private boolean stopRequested;
      public void run(){
            while (!stopRequested && haya trabajo por hacer){
                  seguir trabajando …
            }
      }
      public void requestStop(){stopRequested = true;}
}
```

La comprobación de la variable stopRequested en el bucle principal está perfecto, excepto si el thread se encuentra en ese momento bloqueado. En ese caso, dicho thread sólo terminará si es desbloqueado, situación ésta que puede provocarse interrumpiéndolo. De esta manera, deberá definir el método requestStop para que llame a interrupt:

```
public void requestStop(){
      stopRequested = true;
      interrupt();
}
```

Se puede comprobar la variable stopRequested en la cláusula catch de InterruptedException. Por ejemplo:

```
try{
      wait();
```

```
}
catch (InterruptedException e){
      if (stopRequested) return; // salir del método run
}
```

El método interrupt no genera una InterruptedException cuando se detiene un thread que está trabajando. En su lugar, activa el indicador de "interrumpido". De esta forma, la interrupción de un thread no corrompe el objeto de datos. Esto permite que el thread compruebe el indicador de "interrumpido" después de haberse completado todos los cálculos críticos.

A continuación, vamos a ver el problema que se plantea con el método **suspend()** A diferencia de stop(), suspend() no daña los objetos. Pero pueden generarse situaciones de punto muerto. Veamos un caso.

- En el thread thr01 corre sincronizado sMet01 que bloquea obj01.
- En el thread thr02 corre sincronizado sMet02 que quiere suspender a thr01y acceder a obj01.
- thr02 suspende a thr01, todo bien, pero thr01 no liberó obj01, y no lo hará mientras esté suspendido. Solo si se lo reactiva podrá hacerlo. Si el thread thr02 intenta bloquear obj01 el programa sufrirá un punto muerto:
  - thr01 mantiene bloqueado obj01, solo si reactivado lo liberará.
  - Thr02 quiere el objeto obj01, bloqueado por thr01.

Tenemos un punto muerto.
Esta situación puede ocurrir en interfaces gráficas de usuario mal programadass. Supongamos una simulación gráfica de nuestro banco. Tenemos un botón etiquetado como Pause que suspende los threads de transferencias, y otro con el rótulo Resume que las reinicia. La codificación pertinente pdría ser:

```
pauseButton.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event){
            for (int i = 0; i < threads.lengthj i++)
                  threads[i].suspend()j // No haga esto!!!
      }
});

resumeButton.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event){
            for (int i = 0; i < threads.lengthj i++)
                  threads[i].resume(); // No haga esto!!!
      }
});
```

Supongamos también que un método paintComponent pinta una representación de cada cuenta llamando al método bank.getBalance, y que el método es sincronizado.

Ocurre que tanto las acciones de ambos botones como la operación de pintado **se producen en el mismo thread**, el thread de expedición de eventos. Y puede darse una situación como la siguiente:

1. Uno de los threads de transferencia toma el control del objeto bank.

2. El usuario hace clic en el botón Pause.

3. Se suspenden todos los threads de transferencias; uno de ellos aún mantiene el bloqueo sobre el objeto bank.

4. Por alguna razón, la representación de la cuenta debe pintarse de nuevo.

5. El método paintComponent llama al método sincronizado bank.getBalance.

El programa en este momento no puede actuar, porque el thread de expedición de eventos encuentra al objeto bank bloqueado por uno de los threads suspendidos.

En consecuencia, por mas que el usuario haga clic en el botón Resume, los threads no se reanudarán.

Si desea suspender de forma segura un thread, debe introducir una variable suspendRequested y comprobarla en un lugar seguro de su método run (en alguna parte en la que su thread no bloquee los objetos que otros threads necesiten). Cuando su thread detecte que suspendRequested ha sido activada, se mantendrá a la espera hasta que vuelva a estar disponible.

Podemos hacer esto en una clase específica, SuspendRequestor como ésta:

```
class SuspendRequestor{
      private boolean suspendRequestedj
      public synchronized void set(boolean b){
            suspendRequested = b;
            notifyAll();
      }

      public synchronized void waitForResume()
            throws InterruptedException{
            while (suspendRequested) wait () ;
      }
}


class MyThread extends Thread{
      private SuspendRequestor suspender = new SuspendRequestor();

      public void requestSuspend(){
            suspender.set(true);
      }

      public void requestResume(){
            suspender.set(false);
      }

      public void run(){
            try{
                  while (haya trabajo por hacer){
                        suspender.waitForResume();
                        seguir trabajando
                  }
            }
            catch (InterruptedException exception){…}
      }
}
```

Ahora, la llamada al bloque suspender.waitForResumeO se realizará cuando se haya solicitado la suspensión. Para desbloquear, algún otro thread tiene que reasumir la petición.

El modelo productor/consumidor

A continuación un ejemplo clásico de sincronización y comunicación de hilos en un modelo productor/consumidor. Un hilo produce algo, que otro consume.

Caso 1 – Sincronización a nivel de instancia

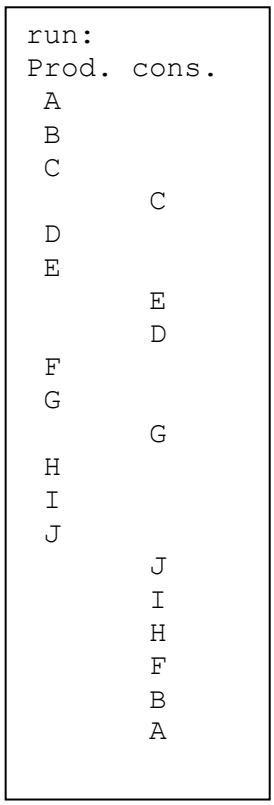
Conceptualmente tenemos la figura de un productor, que será (por ejemplo) un hilo que irá generando caracteres; en otro hilo instalamos un consumidor que irá usando estos caracteres. Necesitamos también un monitor que controlará el proceso de sincronización entre estos hilos de

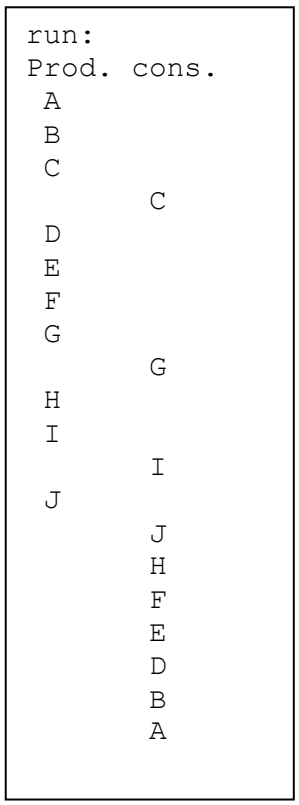
ejecución. Una estructura de datos adecuada para esta gestión del monitor es un buffer implementado sobre una pila simple:

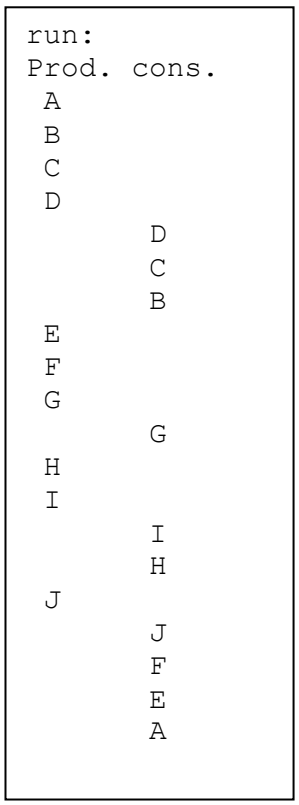
. El productor inserta caracteres en la pila.
. El consumidor los extrae.
. El monitor se ocupa de que esto se haga sin interferencias.

```
package prodcons01;
public class Main {
    public Main() {}

    public static void main(String[] args) {
           Pila.encabe();
           Pila pila = new Pila();
           Productor prod = new Productor(pila);
           Consumidor cons = new Consumidor(pila);
           prod.start();
           cons.start();
    }
}

class Productor extends Thread{ // PRODUCTOR
     private Pila pila;
     private String alfabeto = "ABCDEFGHIJ";
     public Productor( Pila letra){pila = letra;}

public void run(){
           char letra;
           for( int i = 0; i < alfabeto.length(); i++ ){// Apila letras
                letra = alfabeto.charAt(i);
                       pila.apilar( letra );
                       System.out.println(" "+letra);
                       try{ // Da una chance al hilo consumidor
                           sleep( (int) (Math.random()*200 ) );
                       }catch( InterruptedException e ) {;}
           }
           pila.FinPro(); // Fin de la producción
     }
}

// Creamos una instancia de la clase Pila, y utilizamos
// pila.apilar() para ir produciendo.

class Consumidor extends Thread{// CONSUMIDOR
    private Pila pila;
    public Consumidor( Pila t){pila = t;}
    public void run(){
        char letra;
        try { sleep (200 );
        } catch( InterruptedException e ){;}
        while(pila.HayStock() || pila.HayProd()){
            letra = pila.sacar();
            System.out.println("       "+letra);
            try{ //Da una chance al hilo productor
                sleep( (int) (Math.random() *400 ));
            } catch( InterruptedException e ){;}
```

```
        }  // while
    }  // run
} // class consumidor

class Pila{                        // MONITOR
    private char buffer[] = new char[10];
    private int siguiente = 0;
    // Sigen flags para saber el estado del buffer
    private boolean estaLlena = false;
    private boolean estaVacia = true;
    private boolean esperanza = true; // (De producción)
    public static void encabe (){
        System.out.println ("Prod. cons.");
    }

    public synchronized char sacar(){ // Método para consumir
        while(estaVacia){ // esperando para consumir …
            try{wait() ;
            }catch(InterruptedException e){;}
        }
        // podemos consumir, entonces:
        siguiente--; // decrementamos, vamos consumir
        if(siguiente == 0) // Fué la última, ya no quedan
            estaVacia = true; // mas letras
        estaLlena = false;  // acabamos de consumir)
        notify() ;
        return(buffer[siguiente]); // consumimos
    }

    public synchronized void apilar( char letra) { // producir
        while( estaLlena){ // Ciclo de espera por sitio donde
                           // almacenar esta nueva producción.
            try{ // Lo habrá cuando sacar() cambie esta
                wait(); // bandera a false
            }catch( InterruptedException e){;}
        }
        buffer[siguiente++] = letra;
        if( siguiente == 10 ) // Comprueba si el buffer está lleno
            estaLlena = true;
        estaVacia = false;
        notify();
    }

    public synchronized void FinPro(){esperanza=false;}

    public synchronized boolean HayProd(){return esperanza;}

    public synchronized boolean HayStock(){return !estaVacia;}
}
```








```
/* En la clase Pila se pueden observar dos características importantes:
los miembros dato (buffer[]) son privados, y los métodos de acceso
(apilar() y sacar() son sincronizados.
Observamos que la variable estaVacia es un semáforo. La naturaleza
privada de los datos evita que el productor y el consumidor accedan
directamente a éstos. Si se permitiese el acceso directo de ambos hilos
de ejecución a los datos, se podrían producir problemas; por ejemplo, si
el consumidor intenta retirar datos de- un buffer vacío, obtendrá
excepciones innecesarias, o se bloqueará el proceso. Los métodos de
acceso sincronizados impiden que los productores y consumidores
corrompan un objeto compartido. Mientras el productor está apilando una
letra, el consumidor no la puede retirar y viceversa. Está asegurada la
integridad de cualquier objeto compartido. El método notlfy() al final
de cada método de acceso avisa a cualquier proceso que esté esperando
por el objeto, entonces el proceso que ha estado esperando intentará
acceder de nuevo al objeto. En el método wait() se hace que el hilo se
quede a la espera de que le llegue un notify(), ya sea enviado por el
hilo de ejecución o por el sistema.                                  */
```

Caso 2 – Sincronización combinada de instancia/clase

El siguiente ejemplo implementa una clase Cola que utilizaremos para servidores de impresión. La clase tiene métodos para insertar y quitar elementos. Está implementada sobre una lista lineal simplemente vinculada y el nodo tiene tipo de dato String.

Cuando se añade un elemento a la cola, (método inclUIt) los hilos que esperan reciben la notificación. Y en lugar de devolver null cuando la cola está vacía, el método exclPri() espera a que algún otro hilo inserte algo. Puede haber muchos hilos añadiendo elementos en la cola, y muchos hilos que pueden estar tomando elementos de la cola. Como wait puede lanzar una InterruptedException, lo declaramos(el compilador exige que lo hagamos) con la cláusula throws y lo tratamos en el método llamador de indUlt(), en este caso el run() de WorkStat.

Volviendo al ejemplo del servidor de impresión, podemos ver ahora que el hilo consumidor contiene un bucle, intentando continuamente extraer trabajos de la cola. El uso de wait() significa que dicho hilo está suspendido mientras no tenga trabajo que hacer. Por el contrario, si usáramos una cola que devolviera null cuando estuviera vacía, el hilo de impresión invocaría continuamente a exdPri(), utilizando la CPU todo el tiempo, situación conocida como espera con ocupación. En un sistema con múltiples hilos, no desearemos que suceda esto. Debemos suspender las tareas hasta que reciban la notificación que lo que esperaban ha sucedido. Ésta es la esencia de la comunicación entre hilos utilizando el mecanismo basado en wait() y notifyAll()/notify().

El paquete que sigue a continuación, que implementa lo ya descrito, usa simultáneamente dos mecánicas de bloqueo:
· **Métodos sincronizados a nivel instancia**, que cuidan de la no corrupción del objeto cola.

. **Métodos estáticos sincronizados a nivel clase**, responsable de la correcta implementación de la mecánica de impresión deseada: Diez textos por línea, un total de 40 textos. Terminada la tarea, se termina la sesión. Se usan reiteradamente los mismos diez hilos productores, en un orden variable dependiendo de su tiempo de sleep(),y dos consumidores, trabajando sobre un único objeto cola. La codificación es la siguiente:

```
package prodcons02;
public class Main{
    private Usuario[] usr;     //  Usuarios
    private PrintServ[] prs;    //  Impresoras
    public Main() {
        usr = new Usuario[10];   //  10 usuarios generando documentos
        prs = new PrintServ[2];   //  2 impresoras  imprimiendo
    }
    public static void main(String[] args){
        Main main = new Main();
        main.demo();
    }
    void demo(){
        Cola cola = new Cola();     //  Generamos una cola de impresión
        for(int j=0;j<2;j++){
            prs[j] = new PrintServ(cola);   //  Definimos  2 impresoras
            prs[j].start();             //  las arrancamos ...
        }
        String aux;
        for(int j=0;j<10;j++){
            aux="0"+j;
            usr[j] = new Usuario(cola, aux); //  Definimos 10 usuarios
            usr[j].start();             //  y los ponemos a trabajar
        }
    }   //  void demo()
}       //  class Main

class Cola{             //  Oficia de monitor
    private Celda primero, ultimo;  //  Punteros al primer y ultimo documento
    private static int tCon;        //  total impreso
    private static int pos;         //  Posicion en el texto
    private static String texto;    //  texto a tratar
    Cola(){    // constructor sin argumentos
        primero=ultimo=null;
        tCon = 0;
        pos = 0;
        texto = "Durante dos días sopló un viento cálido del Sur.";
        texto+= "La primavera azul se apoderó de la vasta estepa del Don...";
    }/* Los métodos estáticos sincronizados endFila() y contSes()
    implementan bloqueo a nivel clase. Este bloqueo en necesario para
    conseguir imprimir los cuarenta documentos que generan 10 usuarios
```

```
    y que deben ser impresos a razón de 10 p/sesion (o renglon)     */

    public static synchronized boolean endFila(){  // Fin de fila o renglón
        if(++tCon%10==0) return true;
        else return false;
    }

    public static synchronized boolean contSes(){  //  Continúa la sesión ?
        if(pos+4 < texto.length()) return true;
        else return false;
    }

    /* Los métodos de instancia sincronizados inclUlt() (Incluir último)
     y exclPri() (Excluir primero) implementan bloqueo a nivel instancia.
    El bloqueo es necesario para conseguir gestionar adecuadamente un único
    objeto Cola que recibe textos de 10 hilos productores (Clase Usuario)
    y los descarga en 2 hilos consumidores (clase printServ)        */

    public synchronized void inclUlt(String nome) throws NullPointerException{
        String doc = "("+nome+") ";
        doc+= texto.substring(pos,pos+4)+" ";
        if(primero==null) primero=ultimo=new Celda(doc,null);
        else ultimo=ultimo.siguiente=new Celda (doc,null);
        pos+=4;
        notifyAll();
        return;
    }

    public synchronized void exclPri() throws InterruptedException{
        String textImp = "***************";  //  texto a imprimir si error
        while (primero==null && contSes())   // No tenemos texto en cola, pero
            wait();         // la sesión no terminó, esperemos que entre algo
        if(primero != null){     // Ya entro texto en cola
            textImp = primero.elemento;
            if(primero.equals(ultimo)) // Sólo un texto en cola
                primero=ultimo=null;
            else primero=primero.siguiente;
            System.out.print(textImp);
            notifyAll();
        }
    }   //  exclPri()
}       // class

class Celda{
    Celda siguiente;
    String elemento;
    public Celda(String elemento){
        this.elemento = elemento;
    }
    public Celda(String elemento, Celda siguiente){
        this.elemento= elemento;
        this.siguiente=siguiente;
    }

    public String getElemento(String elemento){return elemento;}
    public void setElemento(String elemento){ this.elemento=elemento;}
    public Celda getSiguiente(){return siguiente;}
    public void setSiguiente(){this.siguiente=siguiente;}
}

// PRODUCTOR
class Usuario extends Thread{ // Estaciones generan spool
    private Cola cola;
    private String nome;
```

```
    public Usuario(Cola cola, String nome){ // Constructor
         this.cola = cola;
         this.nome = nome;
    }
    public void run (){
        while(Cola.contSes()){
            try{
                cola.inclUlt(nome); // método de instancia sincronizado
                sleep((int)(1000*Math.random()));
            }catch (NullPointerException e){
                System.err.println("InclUlt() :" + e.getMessage());
            }catch (InterruptedException e){
                System.err.println("Int. Except. " + e.getMessage());
            }
        }   //  while
    }       //  run()
}   //  class Usuario

// CONSUMIDOR
class PrintServ extends Thread{ // Servidores de impresión atienden spool
    private Cola cola;
    PrintServ(Cola cola){this.cola = cola;}

    public void run(){
        while(Cola.contSes()){
            try{
                cola.exclPri();
                if(Cola.endFila())System.out.println(); //  Avanzamos renglón
            }catch (InterruptedException e){
                System.err.println("exclPri(): " + e.getMessage());
            }
        }   //  while
    }   //  void run()
}      //  class printServ
```

Un par de corridas:

```
run:
(00) Dura (01) nte  (02) dos  (03) días (04)  sop (05) ló u (06) n vi (07) ento (08)  cál (09) ido
(01) del  (04) Sur. (01) La p (03) rima (04) vera (03)  azu (06) l se (01)  apo (05) deró (00)  de
(09) la v (08) asta (07)  est (02) epa  (04) del  (09) Don.
BUILD SUCCESSFUL (total time: 1 second)
```

```
run:
(00) Dura (01) nte  (02) dos  (03) días (04)  sop (05) ló u (06) n vi (07) ento (08)  cál (09) ido
(09) del  (08) Sur. (09) La p (00) rima (01) vera (02)  azu (06) l se (04)  apo (07) deró (05)  de
(03) la v (02) asta (04)  est (08) epa  (00) del  (09) Don.
BUILD SUCCESSFUL (total time: 2 seconds)
```

Usando Swing como interfaz de usuario

Barras de progreso

Una barra de progreso es un componente sencillo: un rectángulo que se va rellenando con un color para indicar el progreso de una operación.

Se puede construir una barra de progreso casi de la misma forma a como se define una barra de desplazamiento, indicando un valor mínimo y máximo y una orientación opcional:

```
progressBar = new JProgressBar(0, 1000);
progressBar = new JProgressBar(SwingConstants.VERTICAL, 0, 1000);
```

También es posible establecer los valores máximo y mínimo a través de los métodos setMinimum y setMaximum.

A diferencia de las barras de deslizamiento (o sliders) las barras de progreso no pueden ser ajustadas por el usuario. Para actualizada, su programa debe llamar a setValue.

Llamando a progressBar.setStringPainted(true);

la barra de progreso procesa el porcentaje de tarea completada y muestra una cadena con la forma "n%". Si desea cambiar el aspecto de esta cadena, puede sustituida con el método setString:

```
if (progressBar.getValue() > 900) progressBar.setString("Almost Done");
```

El programa SimulatedActivity muestra una barra de progreso que monitoriza una actividad simulada de consumo de tiempo.

La clase SimulatedActivity implementa un thread que incrementa un valor current diez veces por segundo. Cuando éste alcanza el valor final, el thread finaliza. Si desea dado por concluido antes, debe interrumpido.

Al hacer clic en el botón Start, se inicia un nuevo thread SimulatedActivity. Para actualizar la barra de progreso, lo más sencillo para la simulación de la actividad del thread sería efectuar llamadas al método setValue. Pero esto haría que dicho thread no fuera seguro. (sólo se pueden llamar a los métodos de Swing desde el thread de expedición de eventos). En la práctica, esto es poco realista. Por lo general, un thread en ejecución no sabrá de la existencia de una barra de progreso. En cambio, el programa de ejemplo muestra cómo lanzar un temporizador que registre periódicamente el progreso de ese thread y actualice la barra.

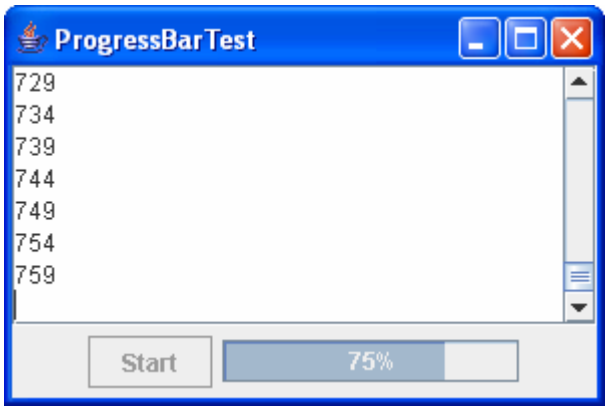
Recuerde que un temporizador Swing llama al método actionPerformed de sus "oyentes" (listeners), y que dichas llamadas se producen en el thread de expedición de eventos. Esto significa que resulta seguro actualizar componentes Swing en la retrollamada del temporizador. Aquí tiene dicha retrollamada. El valor actual de la simulación de actividad se muestra tanto en el área de texto como en la propia barra. Una vez que se alcanza el final de la simulación, el temporizador se detiene y se activa el botón Start.

NOTA: El SDK 1.4 incorpora soporte para barras de progreso indeterminadas, que muestran una animación para representar dicho progreso pero sin especificar el porcentaje del mismo. Éste es el tipo de barra de progreso que puede verse en los navegadores, e indica que dicho navegador está esperando la respuesta del servidor y no sabe cuanto tiempo deberá estar esperando. Para mostrar la animación "espera indeterminada", use al método setIndeterminate.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
```

```
import javax.swing.Timer;

/**
   Este programa muestra el uso de una barra de progreso (Qúmetro)
   para monitorear el progreso de una tarea en un thread
*/
public class ProgressBarTest{
   public static void main(String[] args){
      ProgressBarFrame frame = new ProgressBarFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show();
   }
}

/**
   Marco que contiene un botón para lanzar una barra de progreso con una
   actividad simulada y el area de texto para la salida de la actividad
*/
class ProgressBarFrame extends JFrame{
      private Timer activityMonitor;
      private JButton startButton;
      private JProgressBar progressBar;
      private JTextArea textArea;
      private SimulatedActivity activity;
      public static final int WIDTH = 300;
      public static final int HEIGHT = 200;
      public ProgressBarFrame(){
      setTitle("ProgressBarTest");
      setSize(WIDTH, HEIGHT);

      Container contentPane = getContentPane();

      // area de texto para la salida de la actividad
      textArea = new JTextArea();

      // panel con boton start y barra de progreso

      JPanel panel = new JPanel();
      startButton = new JButton("Start");
      progressBar = new JProgressBar();
      progressBar.setStringPainted(true);
      panel.add(startButton);
      panel.add(progressBar);
      contentPane.add(new JScrollPane(textArea),
         BorderLayout.CENTER);
      contentPane.add(panel, BorderLayout.SOUTH);

      // configurar la acción del boton

      startButton.addActionListener(new
         ActionListener(){
            public void actionPerformed(ActionEvent event){
               progressBar.setMaximum(1000);
               activity = new SimulatedActivity(1000);
               activity.start();
               activityMonitor.start();
               startButton.setEnabled(false);
            }
```

```
         });


      // configurar la acción del temporizador

      activityMonitor = new Timer(500, new
         ActionListener(){
            public void actionPerformed(ActionEvent event){
               int current = activity.getCurrent();

               // mostrar progreso
               textArea.append(current + "\n");
               progressBar.setValue(current);

               // checkear si la tarea ha sido completada
               if (current == activity.getTarget()) {
                  activityMonitor.stop();
                  startButton.setEnabled(true);
               }
            }
         });
   }
}

class SimulatedActivity extends Thread{
   /**
      Construye el thread para simular actividad
      El thread incrementa un contador desde 0 hasta un valor informado
t
   */
   private int current;
   private int target;
   public SimulatedActivity(int t){
      current = 0;
      target = t;
   }

   public int getTarget(){
      return target;
   }

   public int getCurrent() {
      return current;
   }

   public void run(){
      try{
         while (current < target && !interrupted()){
            sleep(100);
            current++;
         }
      }
      catch(InterruptedException e){ }
   }
}
```

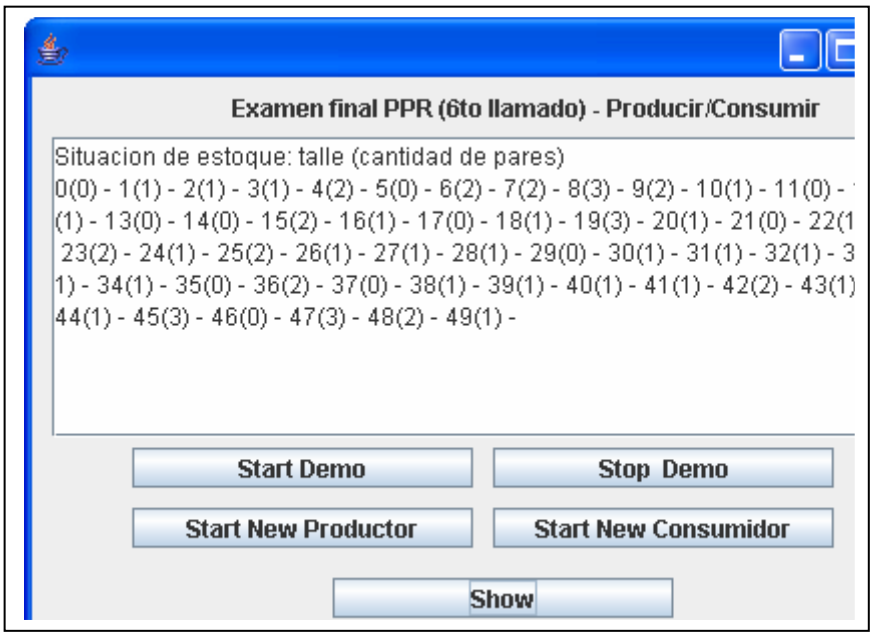


Ejemplos de evaluaciones Parciales o Finales.

Evaluacion Final 06-12-2006

```
// Examen final 6to llamado -  Tymoschuk, Jorge Enunciado
package javaapplication5;
import javax.swing.*; import java.awt.*; import java.awt.event.*;
import java.net.*;    import java.io.*;
public class Main {
    public static void main(String[] args) {
      Participante part = new Participante();
      part.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
class Participante extends JFrame{
    private JLabel titulo, miTarea;
    private JButton btnStartDemo, btnStopDemo, btnStartProd;
    private JButton btnStartCons,btnShow;
    private JTextArea sitStock;
    Productor  prod; Consumidor cons;  Stock stock;
    public Participante(){       //  Dimensiono pantalla
        stock = new Stock(); getContentPane().setLayout(null);
        setSize(450,320);
        //  Defino componentes
        titulo      = new JLabel("Examen final PPR (6to llamado) -
                      Producir/Consumir");
        btnStartDemo   = new JButton("Start Demo");
        btnStopDemo    = new JButton("Stop  Demo");
        btnStartProd   = new JButton("Start New Productor");
        btnStartCons   = new JButton("Start New Consumidor");
        btnShow        = new JButton("Show");
        sitStock       = new JTextArea();
        sitStock.setLineWrap(true);
        JScrollPane sCrollPane = new JScrollPane(sitStock);
        // Posiciono y dimensiono
        titulo.setBounds(100,05,400,20);
        sCrollPane.setBounds(10,30,415,150);
        btnStartDemo.setBounds(50,185,170,20);
        btnStopDemo.setBounds(230,185,170,20);
        btnStartProd.setBounds(50,215,170,20);
        btnStartCons.setBounds(230,215,170,20);
        btnShow.setBounds(150,250,170,20);
        // Incluyo componentes
        getContentPane().add(titulo);
        getContentPane().add(sCrollPane);
        getContentPane().add(btnStartDemo);
```

```
        getContentPane().add(btnStopDemo);
        getContentPane().add(btnStartProd);
        getContentPane().add(btnStartCons);
        getContentPane().add(btnShow);
        //  Incluyo escuchas
        btnStartDemo.addActionListener(new ParticiparListener());
        btnStopDemo.addActionListener(new ParticiparListener());
        btnStartProd.addActionListener(new ParticiparListener());
        btnStartCons.addActionListener(new ParticiparListener());
        btnShow.addActionListener(new ParticiparListener());
        show();
    }// public Participante()

    class ParticiparListener implements ActionListener{
     public void actionPerformed(ActionEvent e){
           //   Codifica el alumno
        }       //  public void actionPerformed
    }           //  class ParticiparListener
}               //  class Participante
class Productor extends Thread {
    private Stock stock;
    public Productor(Stock stk){stock = stk;}
    public void run(){int talle;
        while(true){
            talle = ((int)(Math.random() * 50));
            stock.producir(talle);
            try {sleep( (int)(Math.random() * 200 ) );
            } catch( InterruptedException e ) {e.printStackTrace();}
        }   //  while
    }   //  void run()}
}
class Consumidor extends Thread {//  Codifica el alumno
}     // class Consumidor
class Stock {    // Monitor
    private int talles[] = new int[50];
    boolean estado = false;     //  Estado del demo
    public boolean demoDetenido(){return !estado;}
    public void setEstado(boolean est){estado = est;}
    public Stock(){
        for(int i = 0; i < talles.length;i++)talles[i] = 0;}
    public synchronized int consumir(int nro){ // Método para consumir
        int talle = -1;
        while(demoDetenido()){  //  Ciclo de espera
            try {  wait();} catch( InterruptedException e ) {;}
        }
        if(talles[nro]>0){talles[nro]--; talle = nro;}
        notify(); return talle;
    }
    public synchronized void producir(int nro){
        while(demoDetenido()){ //  Ciclo de espera
            try {wait();} catch( InterruptedException e ) {;}
       }
       talles[nro]++; notify();
    }
    public void inventario(JTextArea jTArea){
           //   Codifica alumno
    }
}
```



Enunciado: En la codificación que UD tiene en sus manos estamos implementando un modelo de producción/consumo de zapatos.
Debemos prever talles {0..49}
El **productor** incorpora su producción al array talles, el **consumidor** la retira.
Ambas clases definen que talle manipulan en cada ocasión al azar.

Su tarea: codificar las partes faltantes, indicadas de esta simulación.

Partes faltantes (La tarea del alumno, lo que se evalúa)

```
class Consumidor extends Thread {
    private Stock stock;
    public Consumidor(Stock stk){stock = stk;}
    public void run() {int talle;
       while(true){
           talle = ((int)(Math.random() * 50));
           stock.consumir(talle);
           try {sleep( (int)(Math.random() * 400 ) );
           } catch( InterruptedException e ) {;}
       } // while
    } // run()
}     // class Consumidor

    public void inventario(JTextArea jTArea){
        String aux = "Situacion de estoque: talle (cantidad de
                      pares)\n";
        for(int i=0;i<talles.length;i++)
            aux+=i+"("+talles[i]+") - ";
        jTArea.setText("");
        jTArea.append(aux);
    }

    public void actionPerformed(ActionEvent e){
        String label = e.getActionCommand();
        if(label=="Start Demo") stock.setEstado(true);
        if(label=="Stop  Demo") stock.setEstado(false);
        if(label=="Start New Productor") new Productor(stock).start();
        if(label=="Start New Consumidor") new Consumidor(stock).start();
        if(label=="Show") stock.inventario(sitStock);
        }       //  public void actionPerformed
    }           //  class ParticiparListener
```