Indice Unidad III (Apunte)**– Programación Concurrente**

Autor: Ing. Tymoschuk, Jorge

Introducción

En una computadora secuencial
- sólo se ejecuta un programa en un momento determinado.
- La computadora únicamente tiene un CPU,
  (elemento de procesamiento (PE, Processing Element)
  que ejecuta un programa a la vez, (proceso) incluye :
  rutinas, datos, pila, código y estructuras del sistema operativo. a
  Máquina Virtual de Java corre su programa, ➔ proceso.

Una computadora secuencial puede realizar multiprogramación
- al reasignar rápidamente el PE entre diversos procesos,
- dando el aspecto de paralelismo,
- al ejecutarse concurrentemente.

  (verdadero paralelismo: computadora con varios PEs)

Dentro de **un proceso**, el control puede ser
- **un sólo hilo** de ejecución,
  (main() ➔ rutinas ➔ rutinas ➔ main())

- **varios hilos** de ejecución o multihilos (multithread).
  (**varias secuencias de ejecución concurrentes**.)

Cada **secuencias de ejecución** es un **hilo independiente:**
- **comparte**
  - espacio de direcciones
  - recursos del sistema operativo
  - todos los datos y procedimientos del proceso,
- **tiene su propio**
  - contador de programa
  - pila de llamado a procedimientos.
- es mucho más fácil de crear que un nuevo proceso.
  (se le conoce a veces como **proceso ligero)**.

Ventajas del procesamiento multihilos

Los programas de un **solo hilo**
- adecuados para procesos con **mínima interacción.**
  (Generalmente Batch Processing)
  ejemplo, actualización de un archivo de personal
  (altas, bajas, modificaciones, previamente grabadas)
  (Así se trabajaba en el pasado, muy poco hoy)

Los programas **multi hilos**
- adecuados para procesamientos interactivos,
  - controlados por eventos,
  - varias partes activas independientes
  - interactúan para alcanzar las metas.
- ejemplo, un programa de animación,
  (una bola que rebota en los márgenes de su frame)
  consta de clases independientes
  - controles de eventos,
  - generación de gráficos y movimiento,
  - conservación de resultados y estadísticas, etc.

  (Resulta muy difícil programar,en un solo hilo:
  - control de eventos
  - generación de gráficos

Desafíos en programación multihilos

- exclusión mutua,
- sincronización,
- calendarización
- punto muerto.

Exclusión mutua

- Los hilos de un programa deben trabajan **cooperativamente.**
- esto implica que diferentes hilos → mismos datos
- incluso simultáneamente,(→ resultados erróneos)
  Ejemplo: Un hilo puede tardar en terminar un calculo,
  si un segundo hilo pretende el resultado "antes de tiempo"
  problemas … entonces:
- imprescindible **acceso mutuamente exclusivo a datos compartidos.**
  (sólo un hilo puede accesar en un momento dado a esos datos)

- En POO podemos encapsular:
  - el recurso (datos) compartido
  - las operaciones necesarias a su manipuleo.
  - Esas operaciones críticas se programan obligando exclusión mutua:
    - Hilo 1 usa **objeto** anfitrion, lo **bloquea para si**
    - Hilos 2, 3 , 4 forzados a aguardar.
    - Hilo 1 termina con objeto, lo libera.
    - Alguno de los que aguardaban lo toma, bloqueandolo para si

Sincronización

- Normalmente los hilos corren independientemente.
- Quien termina (o hace una pausa) antes es impredecible
- No programar suponiendo tiempos de PE para acciones determinadas.
- Ni sabemos el orden en que la cpu le asignará PEs a los hilos.
- No sabemos si el hilo que entra está en condiciones
- es necesario **coordinar el orden** de las acciones. Como?

- Retardando la ejecución de su parte activa
  - Hay varias maneras de hacerlo.
    (un ciclo de espera por una condición)

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

(Esta forma de trabajar se llama "espera ocupada"
Y no es la mejor)

Calendarización de hilos

- Supongamos un proceso constituido por varios hilos
- Supongamos que está corriendo h1
- Cuando termina el PE de h1? Cual hilo corre luego?
- Dependerá de la **política de calendarización** del sistema operativo (sobre el cual corre la JVM)
- Además los hilos tienen un atributo de prioridad.

En los sistemas operativos existen dos **políticas de calendarización**

- **multitarea apropiativa** (preemptive multitasking)
  - Se interrumpe el hilo sin ningún tipo de consulta; (se le acabaron sus milisegundos, afuera).
  - Es la más adecuada.
  - Así trabajan los sistemas operativos UNIX/Linux, Windows NT, Windows 95/98/XP (32 bits)
- **multitarea cooperativa** (cooperative multitasking)
  - Se interrumpe el hilo cuando otros desean obtener el control (y el que está corriendo está de acuerdo):
  - Muy dependiente de la codificación. (un programa mal codificado puede bloquear el rsesto del mundo).

Punto muerto

- Si varios hilos son interdependientes de muchas maneras y
- comparten recursos bajo exclusión mutua y
- subtareas bajo sincronización, podemos llegar a
- un **punto muerto:** hilos aguardan eventos que nunca sucederán.

Hilos en Java

- Un programa Java empieza la ejecución en un flujo principal
  - el main de control de la aplicación
  - el applet de Java
  - Ambos pueden producir otros hilos (que se ejecutan concurrentemente)
  - Estos hilos son **objetos de una clase** que:
    - extienden la clase **Thread**
    - implantan la interfaz **Runnable.**

- El punto de entrada es siempre su método **public void run();**
  - este método puede invocar otros.
  - El hilo termina cuando en el método run finaliza.
  - El run() de la clase Thread no hace nada.
  - Siempre debemos redefinirlo para que ejecute lo necesario.

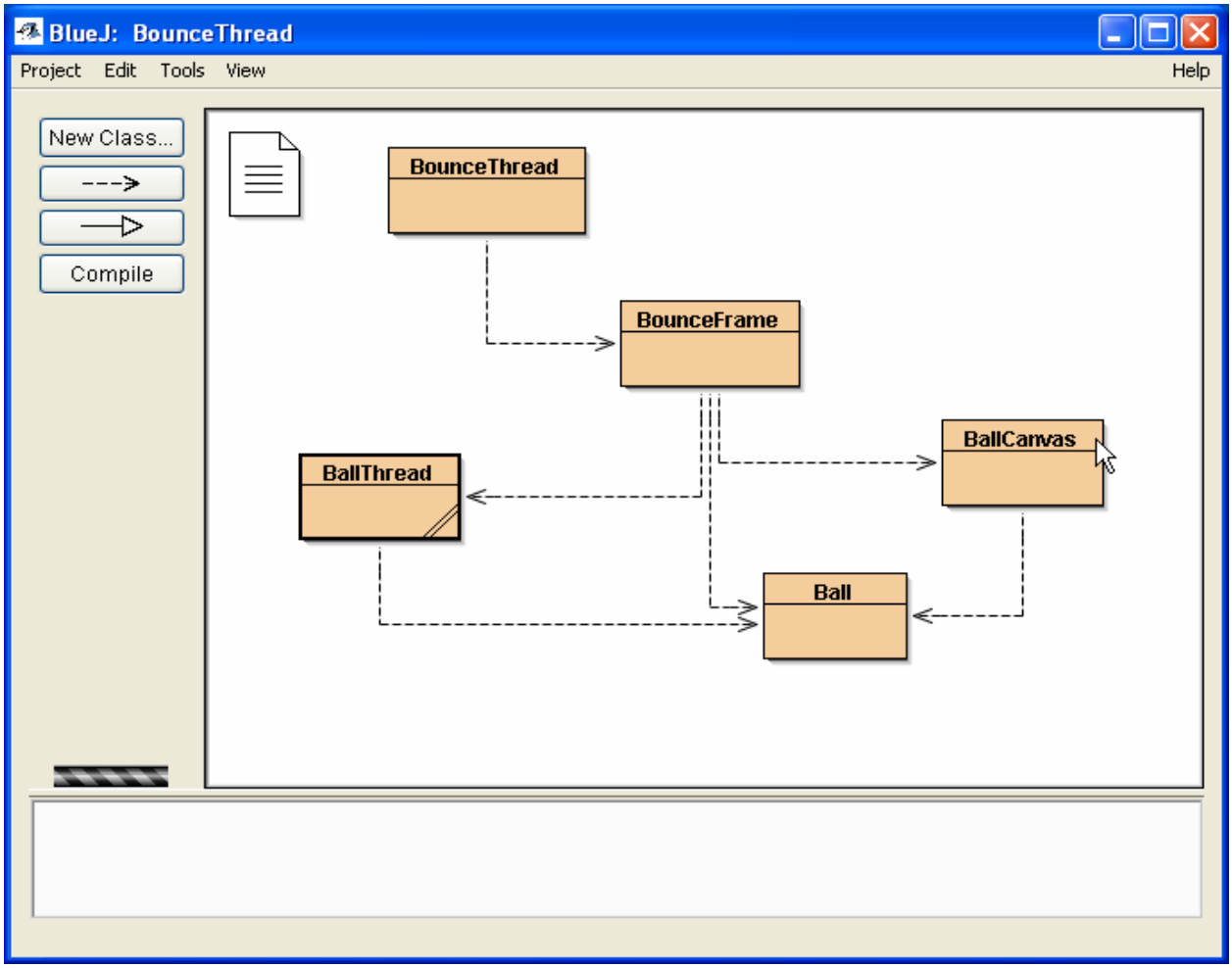
```
package javaapplication13;
public class Main {
    public static void main(String[] args) throws InterruptedException{
        System.out.println("Arrancamos main");
        ParImpar thr01 = new ParImpar(1, 20);
        ParImpar thr02 = new ParImpar(0, 30);
        System.out.println("Objetos instanciados, 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!
```

Una animación. Una bola rebotando en los límites de su campo de acción.



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

- **Public class BounceThread**: Contiene el main() que instancia JFrame = new **BounceFrame**(), y ejecuta frame.show().
- **class BounceFrame extends JFrame**, hilo principal, atiende los eventos de la interfaz de usuario.
- **class BallThread extends Thread**, ciclo del demo
- **Class BallCanvas extends JPanel**, añade la bola al lienzo (JPanel)
- **Class Ball**, movimiento de la bola, sus rebotes, …

A continuación el programa completo.

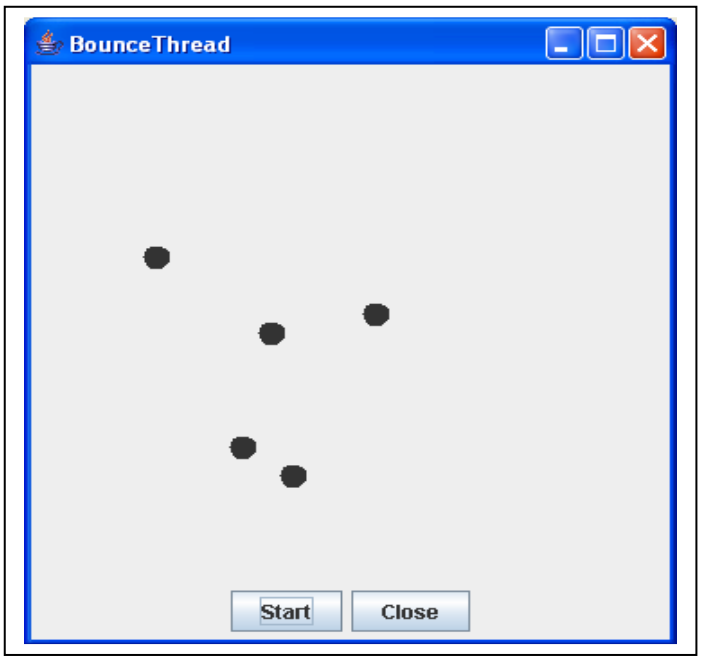
```
import javax.swing.*;
public class BounceThread{  //  "Rebote hilado..."
   public static void main(String[] args){
      JFrame frame = new BounceFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.show(); //  Mostrando la estructura grafica
   }
}

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class BounceFrame extends JFrame{ //   canvas y botones
   private BallCanvas canvas;     //   Referencia a BallCanvas
   public static final int WIDTH = 450;
   public static final int HEIGHT = 350;
   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 vt){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); //    Usamos add() de Container
      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 colección ArrayList (Visto en U I)
   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, pos. 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

- necesitamos comportamiento de hilo en clase que extiende otra.

```
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)
```

Interrupción de threads

- Un thread finaliza cuando su método **run** regresa.
- No existe una forma fácil de forzar a que un thread finalice.
- puede usar **interrupt**() para solicitar dicha finalización.
- **El run de ese thread deberá comprobar periodicamente si debe salir.**

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

- Y porque no hay forma facil?
- un thread **no debe** estar continuamente trabajando,
- debe "siestear" de vez en cuando para que otros trabajen.
- Problema: si "duerme", no comprueba si debe finalizar.
- Solución: **InterruptedException.**
  - Cuando en thrd01 ejecutamos el método **thrd02.interrupt()** y
  - thrd02 está durmiendo (bloqueado), dicho bloqueo (producido por sleep() o wait())
  - es finalizado en una InterruptedException.

```
public void run(){
     try{
          while (!interrupted() && 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
}
```

- operaciones de i/o **no son finalizadas** por interrupt()
- Al volver del i/o, if(**interrupted()**) // fue interrumpido.

- Podemos preguntar de dos formas:
  - **interrupted**: método estático que comprueba y reinicializa el thread
  - **islnterrupted**: método de instancia que solo comprueba nterrupción

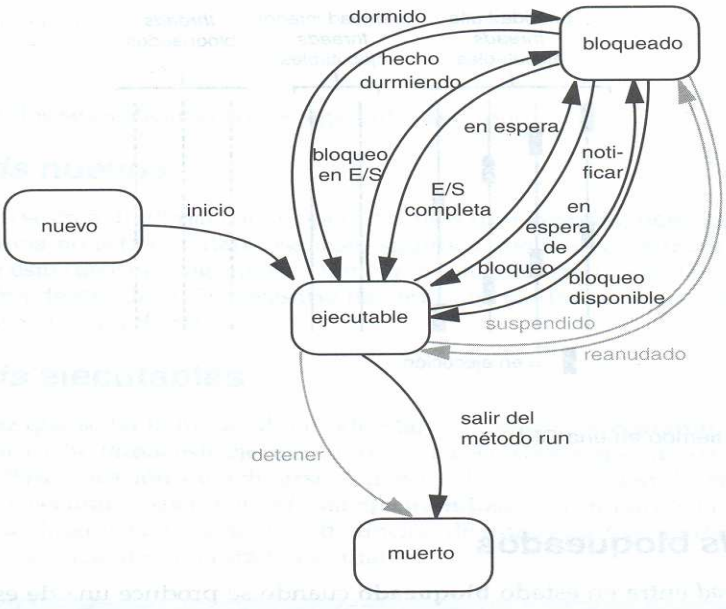
- En resumen, debemos usar ambas formas de tratar interrupciones:
  - comprobar el indicador de "interrumpido"
  - capturar una InterruptedException.

Estados de un thread (posibles cuatro)

Threads nuevos
- Recien creado por new()

Threads ejecutables
- Después de start(), el thread está en situación ejecutable
- el sistema operativo da a ese thread ocasión de activarse.
- Java no distingue entre ejecutable y en ejecución
- un thread en ejecución, lo estará mientras no desalojado.
- Este desalojo depende de los servicios del sistema operativo. (Ver calendarización de hilos).
  Ejemplo 1: green threads (Java 1.x, SO Solaris: El thread se ejecuta hasta que otro de una prioridad mayor "despierta" y toma el control.
  Ejemplo 2: Windows 95, NT, XT otorgan a cada thread ejecutable una porción de tiempo para realizar su tarea. (división de tiempo)

Threads bloqueados En esta situación cuando:
- llama al método sleep() del thread.
- llama a una operación de entrada/salida
- llama al método wait().
- intenta bloquear un objeto que ya bloqueado por otro thread.
- Otro thread lo suspende (censurado)



Cómo salir del estado bloqueado
siguiendo el camino inverso al recorrido para bloquearse:
- Sleep() - debe expirar el número de milisegundos especificado.
- Esperando I/0 - ésta operación debe finalizar.
- Wait() - otro thread ➔ notifyAll() o notify()
- El objeto pretendido esta bloqueado por otro thread – esperar
- suspendido, - resume (Métodos censurados)

Threads muertos (dos posibles razones)
- Naturalmente, debida a una salida normal del método run().
- Inesperadamente, una excepción no en el método run().

Es posible matar un thread invocando al método stop. (censurado)

Información sobre threads
- Método isAlive() devuelve:
  - ejecutable o bloqueado), trae
  - nuevo o muerto, false
- Limitaciones – No es posible diferenciar
  - 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
thrd.setDaemon(true); Ejms
envíar "señales de reloj" a otros threads.
recolector de basura…

Grupos de threads
- Es útil categorizar threads en función de su cometido.
- Podemos referirnos al grupo, mejor que individualmente.

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

Thread t = new Thread(grp, threadName); // incorporo hilos

if (grp.activeCount() == 0) // Pregunto si todos están detenidos

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

- Los grupos de threads pueden disponer de subgrupos hijo.

Prioridades de los threads
- cada thread tiene una prioridad,
- se hereda de su thread padre.
- Se puede aumentar o disminuir **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.

  (Lamentable: No todos los SO consideran la prioridad)
- El thread en ejecución sigue ejecutándose hasta que:
  - cede el control, invocando al método yield()
  - deje de ser ejecutable (muerto, bloqueado)
  - un thread de mayor prioridad pasa a ser ejecutable

Supongamos varios threads ejecutables con la misma máxima prioridad

- El planificador selecciona uno de ellos.
- no hay garantia de trato igualitario
- es posible (algunas plataformas) el planificador cualquiera.
- Windows NT dispone de menos niveles que Java
- En la JVM Sun para Linux, las prioridades no existen (¿)

programa ejemplo; si hace clisk en el boton
Start, se lanza bola negra, prioridad normal
Express, lanzará la bola roja, prioridad mayor.

```
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 esto en máquinas Windows y Linux …
```

Cada ciclo del "Dispatcher" el planificador

- evalúa 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.
sólo con todos los express dormidos consigue turno un normal.

Threads egoístas: Si no cumple con llamar a sleep() o yield()
(para asegurar que no monopoliza CPU)

```
class Cualquiera extends Thread{
   private boolean flag;
   public void run(){
      try{
         for (int i = 1; i <= 1000; i++){
           b.move();
           if (flag){ // Activado por un evento en otro hilo
              long t = System.currentTimeMillis();
              while (System.currentTimeMillis() < t + 5000);
           }else sleep(10);
         }
      }
      catch (InterruptedException exception){;}
   }
}
```

Cuando se activa **flag**, 5 segs de monopolio.
Lo que ocurra dependerá de la calendarización (Ya vimos)

Sincronización

- En aplicaciones multithreaded, compartimos acceso a objetos.
- qué ocurriría si varios hilos modifican su estado?
- se pueden corromper los objetos.

Comunicación de threads sin sincronización -

- banco que dispone de 10 cuentas,
- transacciones aleatorias que muevan dinero entre ellas.
- Cada cuenta es gestionada por su propio thread
- Cada banco.transfer: cuenta 1 (dinero) ➔ cuenta 2
- No importa cuanto dinero hay en cada cuenta.
- al final del proceso **el monto total no debe haber cambiado**
- Cada 10.000 transacciones, **test** hace esta verificación.

```
/**
 *  Este programa muestra posible 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);
      for (int 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];
      for (int 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 11 veces, Suma: 100000, Ctas sdo neg. 3
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)
```

Por que a veces que Suma < 100000?

- ocurre que al correr test(), algunas veces transfer ya debitó el monto, el dispacher le cortó el tiempo, y no conseguió creditar.
- En la próxima pasada esto puede compensarse

Es como que el gerente lleva plata a casa, luego la repone…
es probable que no desee depositar su dinero en un banco así.

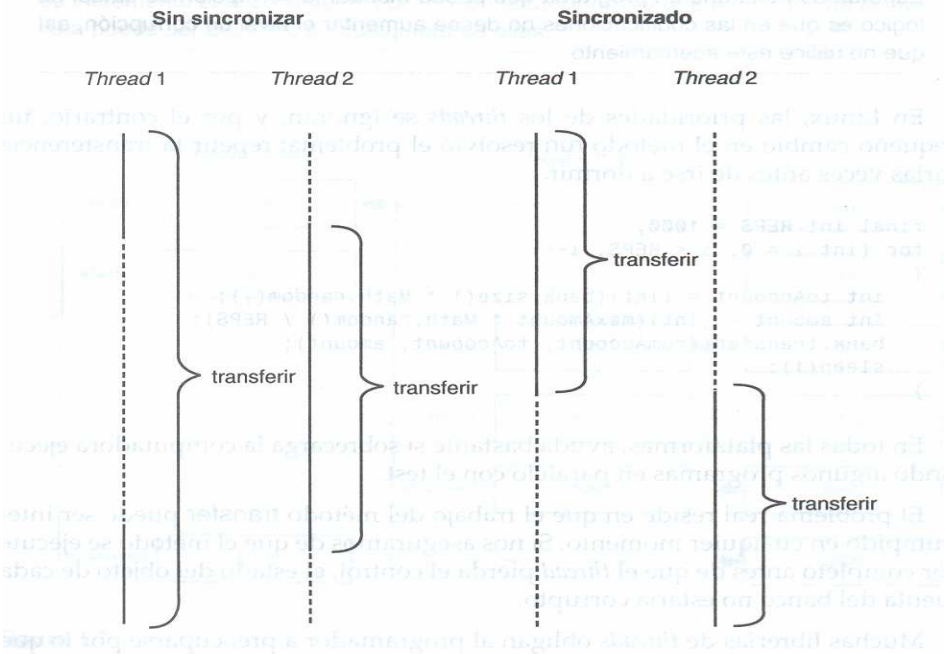
Esto ocurre porque **no hemos sincronizado** lo que hacen los hilos.
En realidad, "propiciamos" que esto ocurra,
demorando bastante en medio de la transfer().

Como sincronizamos?

- Java dispone de mecanismo inspirado por invencion de Tony Hoare.
- cláusula synchronized en los métodos críticos.

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

- thread llama a un método sincronizado,
- ese método **terminará** antes de que otro thread pueda ejecutar **cualquier otro método sincronizado del mismo objeto**.
- Entonces: un **thread llama a transfer**
- un segundo **intenta** hacer lo mismo,
- el segundo **no podrá** continuar,
- y será **desactivado** y
- puesto **en espera** hasta que el primero termine.



se etiquetarán como synchronized aquellos métodos que

- realicen múltiples operaciones de actualización
- recuperen valores desde dichas estructuras.

el mecanismo de sincronización **no es gratis**.

- Se ejecuta **código adicional** cada llamada a método sincronizado.
- no sincronizar cuando:
  - objetos no compartidos por varios threads
  - un método siempre devuelve el mismo
- puede resultar personalizar clases para threads.

Bloqueo de objetos

- thread → método sincronizado, su objeto se "bloquea". (Locked)
- otro thread intenta → método sincronizado que → mismo objeto, no tiene acceso, **espera**.
- Finalmente, primer thread termina, **libera** su objeto.
- Periódicamente, el "scheduler" revisa lista de threads en espera,
- De acuerdo a su calendarización activa un thread.
- Si su método es sincronizado, bloquea al objeto y ejecuta.
- Métodos no sincronizados no son afectados por el bloqueo.

- Una excepción en un Thread libera el bloqueo de su objeto.
- Sinronizado m1 → sinronizados m2, mismo objeto obj.
  - Se otorga inmediatamente el acceso a m2.
  - Supongemos m2 llama a m3, ok, acceso a m3.
  - El thread desbloquea obj al salir de m1.

cada objeto → contador de bloqueo

- cuantos métodos sincronizados han llamado al "bloqueador".
  - Nueva llamada sincr., contador++
  - finaliza un método sincronizado, contador
  - if(contador == 0) objeto desbloqueado

- mismo método sinronizado puede correr en varios hilos (<> objs).
- un thread puede bloquear de varios objetos a la vez.

Los métodos wait y notify

- Qué hacemos cuando no hay suficiente dinero en la cuenta?
- esperamos hasta que otro thread añada fondos.
- Problema: transfer es synchronized.
- Este thread tiene el acceso exclusivo al objeto banco, ningún thread podrá depositar. Que hacer?
  - Lllame al método wait() de la clase Object
    - **el thread actual se bloquea**
    - **y libera el bloqueo del objeto.**
    - (otro thread podrá aumentar el saldo de la cuenta).

- El método wait dispara InterruptedException si interrumpido mientras está esperando.
  - se puede activar el indicador de "interrumpido" o
  - **propagar la excepción**

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

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

- thrend → wait(), → **lista de espera** de ese objeto (bloqueado)
  - el scheduler ignora al thread, no tiene chance de ejecución
  - Para sacarlo de la **lista**, otro thread → notify/notifyAll
  - notifyAll excluye a todos de la **lista** de espera del objeto;
  - notify excluye un thread elegido arbitrariamente.
- thread excluido de la lista de espera,
  - vuelve a ser ejecutable,
  - y el planificador podrá activarlo de nuevo.
  - podrá intentar entrar de nuevo en el objeto.

- Es fundamental que los threads → notify/notifyAll periódicamente.
  - un thread llama a wait(), él solo no puede desbloquearse,
  - deposita sus esperanzas en los otros threads.
  - Si ninguno de ellos → notify/notifyAll, nunca más.
  - Pueden producirse bloqueos totales, colgarse el programa

Cuando usar **notify o notifyAll**?

- **notify** si **cualquier thread** de la lista de espera nos sirve
- **notifyAll** si sólo **alguno** de los hilos estará en condiciones

formatos para el método wait

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

- El **thread corriente** debe poseer su objeto monitor. *(tiene un)*
  - se bloquea
  - libera el bloqueo del objeto.
  - el scheduler lo ignorará
- yacerá durmiente hasta:
  - Algún otro thread invoca el método notify/notifyAll
  - Algún otro thread lo interrumpe
  - El tiempo especificado ha transcurrido.
- ocurrido alguno de estos tres puntos:
  - rehabilitado para la planificación de hilos.
  - pasa a competir por bloquear el objeto
- si bloquea el objeto
  - requerimientos de métodos sincronizados restaurados
  - método sincr. Retoma control

Las reglas de planificación

- métodos threads modifican un objeto, sincronizados.
- métodos sólo lectura **afectados**, sincronizados.
- Si se debe esperar a que el estado de un objeto cambie,
  - Use un método sincronizado
  - Dentro llame a wait.
- cambiado el estado de un objeto, llamar a notifyAll.

Volviendo al ejemplo como quedaría tranfer?. Le incorporamos:

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

```
/**
 *  Este programa resuelve la corrupcion de datos
 * que teníamos en UnSYnchBankTest
 * 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, problema de …
```

Puntos muertos (abrazo mortal, deadlock)

La sincronización no resuelve todos los problemas.
Supongamos que 1 y 2 son los únicos hilos que pueden correr

Thread 1: banco.transfer(1,2,3000); wait(); // saldo 2.000 dólares

Thread 2: banco.transfer(2,1,4000);wait(); // saldo 3.000 dólares

- UD es el responsable de **eludir estos puntos muertos.**
- debe diseñar los threads evitando estas situaciones

Que hacemos? modificamos los métodos transfer() y test():

- Transfer():
  - Transferirá si hay saldo suficiente;
  - O cuenta transac. no atendidas (**contTransNoAte++**).
- Test(). Informará transac. no atendidas
-

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

```
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 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 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)
```

Tener presente - **llamada a una operación de entrada/salida:**
se libera el objeto, y se **bloquea el thread**.

CANCELACIÓN DE UN HILO
necesitamos cancelar un trabajo antes de que se complete.
(Ej.: botón cancelar en una interfaz de usuario)

```
thread 1 // eventos usuario
thread2.interrupt();
```

```
thread 2 // interfaz gráfica
      while (!interrumpido()){
      // trabajando ...
}
```

métodos relacionanados:

- **interrupt()**, interrumpe **thread 2**;
  (Si thread 2 en sleep() o wait(), ➔ InterruptedException)
- **isInterrupted()**, comprueba interrupción;
- **interrupted()**, static, comprueba interrupcion y borra ese estado

Estado "interrumpido"

- sólo puede ser borrado por dicho hilo.
- Normalmente se necesita alguna limpieza; para asegurarse:
  - utilizar interrupted
  - hacer el trabajo de limpieza

ESPERA A QUE UN HILO FINALICE

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

Bloque o sentencia sincronizada.

- acceso exclusivo al objeto en algunas instrucciones
- Alternativa: invocar otro metodo sincronizado

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

Métodos estáticos sincronizados

Una **singleton** es una clase con un solo objeto.
(objeto global, a nivel clase)

- como colas de impresión,
- gestores de conexiones a bases de datos, etc.

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

Existe un único objeto de tipo Class que describe cada clase

- ese objeto es bloqueado por un método static synchronized.
- Bloqueos a nivel instanciar y clase son independientes.

Recapitulando, (métodos de JAVA.LANG.OBJECT)
wait(), notifyALL(), notify()

- sólo pueden ser llamados por un método sincronizado con objeto bloqueado
- disparan IIlegalMonitorStateException si el thread no tiene el bloqueo

¿Por qué censuraron los métodos stop() y suspend()? (Java 2)

La plataforma Java 1.0 definió:

- **stop**, que simplemente daba por finalizado un thread.
  - desbloquea todos sus objetos,
  - no chequea la coherencia de su estado
- **suspend**, que lo bloqueaba hasta que otro thread llamaba a resume.
  - produce puntos muertos con bastante frecuencia.

Si necesita detener un thread de una forma segura,

- compruebe periódicamente una variable al efecto.
- (consenso del thread a ser detenido, necesario)

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

Si necesita suspender un thread de una forma segura,

- compruebe periódicamente una variable al efecto, donde no bloqueemos objetos que otros necesiten)

```
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){…}
      }
}

class SuspendRequestor{
      private boolean suspendRequested = true;
      public synchronized void set(boolean b){
            suspendRequested = b;
            notifyAll();
      }
      public synchronized void waitForResume()
            throws InterruptedException{
            while (suspendRequested) wait () ; // wait bloquea objeto
      }
}
```

como funciona esto?

- run() solicita suspensión // *suspender.waitForResume()*
- Para reasumir, otro thread: suspender.set(false);

Nota documentativa.

En el release 5 de java, (jdk1.5.5) en todos sus updates (1.5.0_00 al 1.5.0_12, el último antes de pasar al release 6), la clase Thread tiene:

- 2 clases anidadas (nested)
- 3 campos estáticos (para la prioridad)
- Métodos
  - 8 constructores
  - 42 métodos propios
  - 10 heredados de java.lang.Object

  Total 60 métodos

El release 5 es el que estamos usando en este apunte (2007) y el que está actualmente implementado en los laboratorios de Sistemas.

El modelo productor/consumidor

- Hilos producen, otros consumen.

Ejemplo:

- un hilo produce caracteres (los inserta en una pila)
- otro hilo los consume (los extrae)
- un monitor controla sincronización de los hilos.

```
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;}
}
```

```
run:
Prod. cons.
 A
 B
 C
       C
 D
 E
       E
       D
 F
 G
       G
 H
 I
 J
       J
       I
       H
       F
       B
       A
```

```
run:
Prod. cons.
 A
 B
 C
       C
 D
 E
 F
 G
       G
 H
 I
       I
 J
       J
       H
       F
       E
       D
       B
       A
```

```
run:
Prod. cons.
 A
 B
 C
 D
       D
       C
       B
 E
 F
 G
       G
 H
 I
       I
       H
 J
       J
       F
       E
       A
```

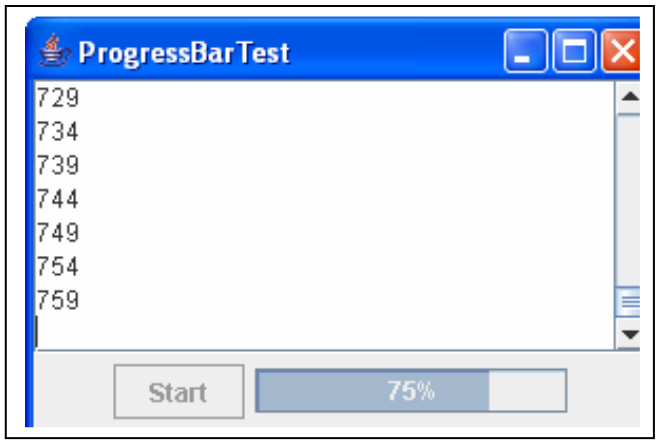
Usando Swing como interfaz de usuario

Barras de progreso

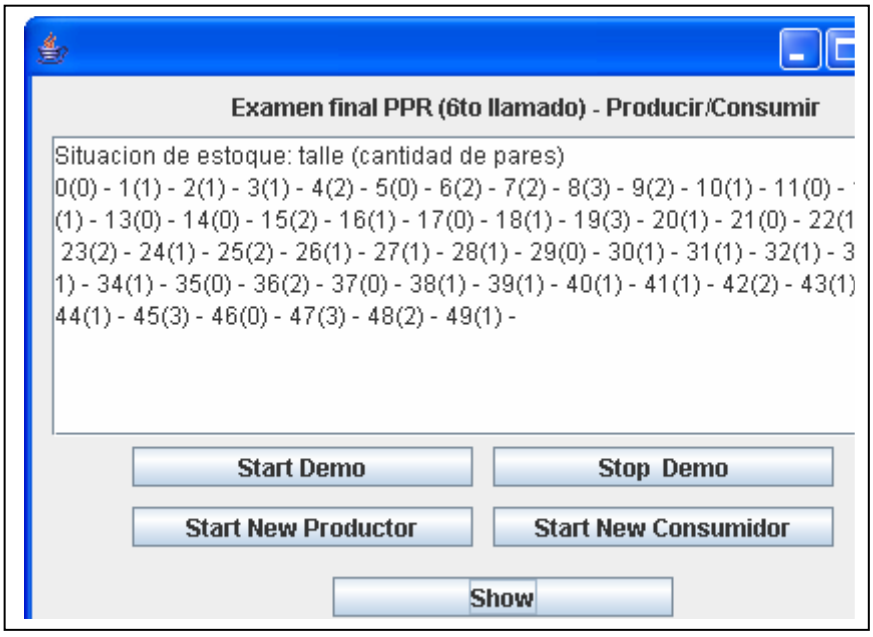
- un rectángulo que se llena indicando el progreso de una operación.

```
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.



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.

```
// Examen final 6to llamado - Tymoschuk, Jorge      (Material ➔ Alumno)
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 Stock {    // Monitor, clase con los métodos sincronizados
    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
    }
}
```

```
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 Participante extends JFrame{    // Entorno gráfico
    private JLabel titulo, miTarea;
    private JButton btnStartDemo, btnStopDemo, btnStartProd;
    private JButton btnStartCons,btnShow;
    private JTextArea sitStock;
    Productor  prod; Consumidor cons;  Stock stock;
    public Participante(){      //   Constructor
        stock = new Stock(); getContentPane().setLayout(null);
        setSize(450,320); //  Dimensiono pantalla
        //  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
```

Abajo, lo **faltante que se evalua** (Codificación del alumno)

```
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
```