UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
DEPARTAMENTO DE SISTEMAS DE INFORMACIÓN
CÁTEDRA:
PARADIGMAS DE PROGRAMACIÓN

APUNTE TEORICO-PRACTICO

UNIDAD 3

CONCURRENCIA Y TRABAJO EN RED

AUTORES
Jorge Tymoschuk
Analía Guzmán

AÑO 2004

TRABAJO EN RED

1. INTRODUCCION

En TOPICOS INTRODUCTORIOS,

QUE ES UN PARADIGMA?,

citamos del historiador Thomas Kuhn la frase:

"un conjunto de teorías, estándares y métodos que en conjunto representan una forma de organizar el conocimiento, es decir, una forma de ver la realidad".

Más adelante se afirma que un programa es la "Simulación computacional de una porción de la realidad". Y que el mejor paradigma es aquel cuyo modelo representa mas fielmente la realidad que logramos percibir, aquel cuyo "gap semántico" es menor.

En nuestra realidad hoy es omnipresente "La Red de Redes", Internet. Al respecto no es necesaria ninguna fundamentación. Los proyectos de sistemas de información que desarrollemos no pueden ignorar esta realidad. Algunos de ellos, muy locales o específicos, podrán abstraerse de ello, pero los que pretendan ser de uso general necesariamente deberán contemplarla.

Esa es la "idea fuerza" que orienta la distribución de contenidos de esta unidad.

En su primera parte, el tema de la concurrencia, procesamiento multi hilos. Hacemos hincapié en el esquema productor/consumidor. En su segunda parte, Trabajo en Red, este mismo esquema se generaliza a mas de una computadora, hablamos de cliente/ servidor y presentamos un par de tecnologías para su implementación.

2. CONCEPTO DE CONCURRENCIA

Supongamos una máquina dotada de un único procesador, encargado de administrar todas las operaciones. Alguien desea leer un documento y es este procesador el que interactuando con los controladores del disco debe controlar los movimientos del cabezal, encontrar y leer los distintos sectores, colocar la información en el bus de datos para luego recibirla y procesarla.

Naturalmente no es así como funcionan las máquinas de hoy. En el disco duro existe un procesador de menor inteligencia que el procesador central , que se encarga de mover el cabezal, rastrear sectores y manejar el bus de datos.

En conclusión llegamos a que es imposible que un procesador maneje todas las operaciones que se realizan en unos cuantos milisegundos dentro de una computadora brindando un tiempo de respuesta aceptable.

Los procesadores más pequeños que se encuentran en los dispositivos o en las placas adaptadoras se llaman controladores y no son un invento nuevo. En los años 50 se desarrollaron equipos que tenían, además de un procesador central, otros capaces de manejar dispositivos de entrada salida y bus de datos, para lograr un mejor rendimiento del procesador.

Con esta configuración un equipo es capaz de correr dos o más procesos al mismo tiempo, mientras el procesador central esta ocupado con cierta tarea, el controlador del disco esta moviendo el cabezal, el cañón del monitor está dibujando la imagen, el coprocesador matemático está haciendo algunos cálculos, etc. y de esa forma se ahorra mucho tiempo, porque los procesos corren en el mismo momento, y no es necesario terminar de imprimir para leer el disco o calcular una suma.

Con el mismo concepto, se han creado grandes equipos, que además de sus controladores, tienen más de un procesador central, de esta manera distintos programas pueden ejecutarse al mismo tiempo, uno en cada procesador y además hacer uso de todos los recursos del equipo por medio de sus controladores.

Esto último se denomina PARALELISMO EN HARDWARE, más de un procesador permite ejecutar más de un proceso al mismo tiempo.

Se dice que dos o más procesos son concurrentes si están construidos de manera tal que pueden ejecutarse al mismo tiempo, compartiendo recursos y datos.

En todos los paradigmas analizados hasta el momento, el seguimiento del programa ha sido siempre secuencial, es decir que se ejecuta una instrucción debajo de la otra. Si un proceso llama a una función, este se interrumpe, comienzan a ejecutarse secuencialmente una a una las instrucciones de la función hasta que esta retorna un valor que es tomado por el proceso llamador para continuar con la instrucción siguiente.

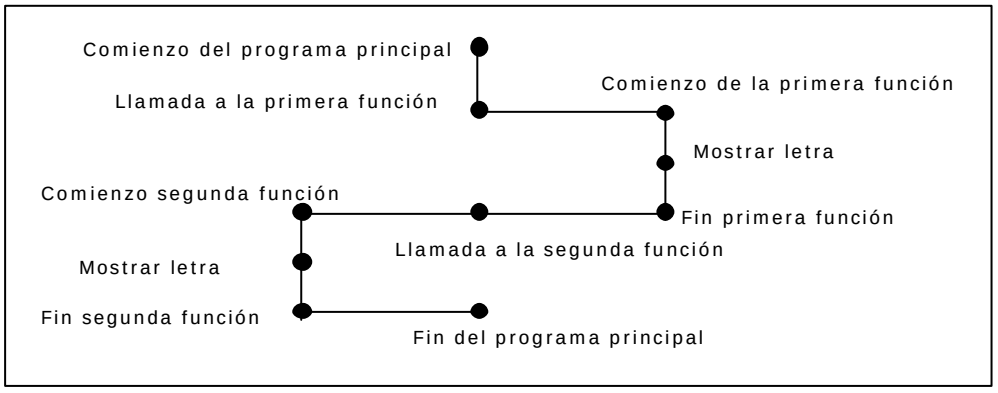
Lenguajes como ADA, JAVA,etc permiten que dos procesos o más se ejecuten simultáneamente, siempre que estén corriendo sobre un hardware paralelo, de esta manera, los procesos pueden interactuar compartiendo datos y recursos, como funciones y procedimientos comunes, pero no necesariamente debe interrumpirse uno para que otro comience hasta finalizar de devolver el control.

Es muy importante que se entienda que pese a que paralelismo y concurrencia son dos conceptos muy relacionados puede existir uno sin el otro. Puede haber paralelismo en el hardware sin concurrencia, por ejemplo, cuando se corren dos o más procesos en paralelo, en diferentes procesadores, pero no están preparados para compartir datos, no interactuan, son aplicaciones distintas que se diseñaron para correr independientemente una de la otra, aunque un equipo con más de un procesador pueda soportarlas al mismo tiempo. Por otro lado puede haber concurrencia sin paralelismo, si una aplicación está compuesta por procesos que no necesariamente deben ejecutarse en forma secuencial, comparten datos, interactuan entre si y pueden hacerlo simultáneamente, pero al ser soportadas por un único procesador se ejecutan en forma secuencial.

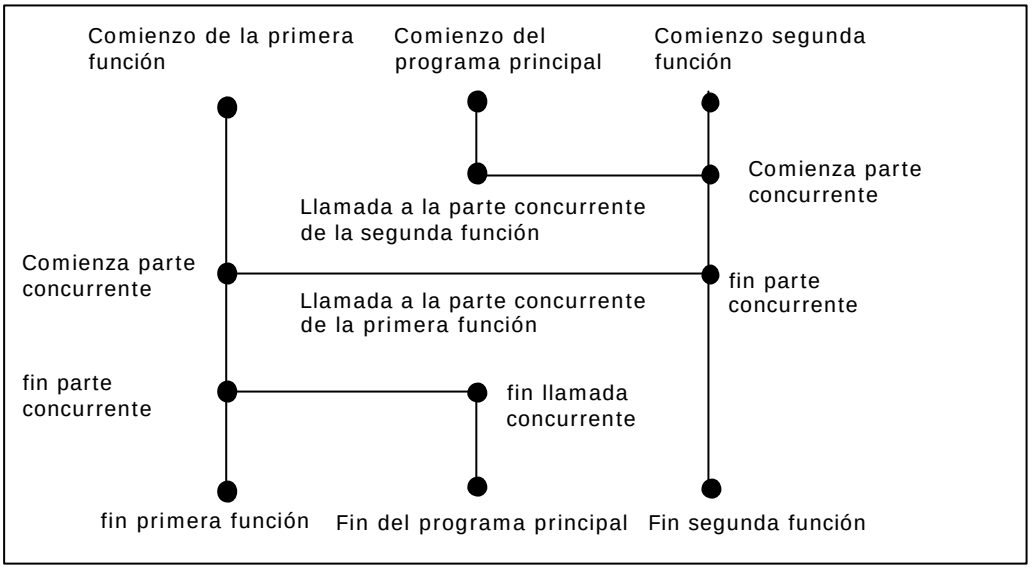
3. UN EJEMPLO TEORICO

El objetivo de este punto es clarificar el concepto de concurrencia mediante el uso de un ejemplo muy sencillo.

Supongamos un programa que debe mostrar dos letras en la pantalla y para ello llama una función que muestra la primera y a otra que hace lo mismo con la segunda. En un programa secuencial, gráficamente esto es:

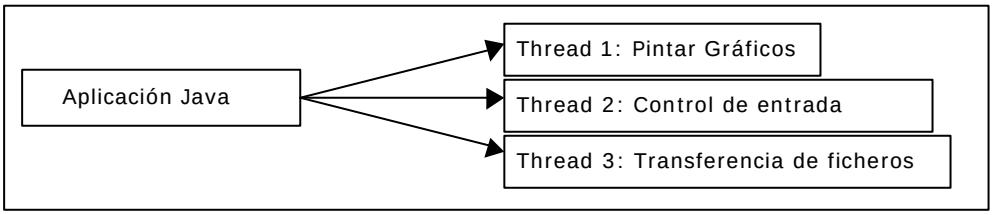


En un programa concurrente se lanzan los tres procesos, el principal y las dos funciones. Si el hardware lo permite, las tres tareas pueden comenzar a ejecutarse al mismo tiempo, pero si en un dado momento requieren simultaneamente un recurso único (Impresora, por ejemplo), concurren por él, la ejecución pasa a ser concurrente. El ejemplo anterior podría ser:



Java nos trae un concepto muy importante

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



Hay que distinguir multihilo (multithread) de multiproceso. El multiproceso se refiere a dos programas que se ejecutan "aparentemente" a la vez, bajo el control del Sistema Operativo. Los programas no necesitan tener relación unos con otros, simplemente el hecho de que el usuario desee que se ejecuten a la vez.

Multihilo se refiere a que dos o más tareas se ejecutan "aparentemente" a la vez, dentro de un mismo programa.

Se usa "aparentemente" en ambos casos, porque normalmente las plataformas tienen una sola CPU, con lo cual, los procesos se ejecutan en realidad "concurrentemente", dado que comparten la CPU. En plataformas con varias CPU, sí es posible que los procesos se ejecuten realmente a la vez.

Tanto en el multiproceso como en el multihilo (multitarea), el Sistema Operativo se encarga de que se genere la ilusión de que todo se ejecuta a la vez. Sin embargo, la multitarea puede producir programas que realicen más trabajo en la misma cantidad de tiempo que el multiproceso, debido a que la CPU está compartida entre tareas de un mismo proceso. Además, como el multiproceso está implementado a nivel de sistema operativo, el programador no puede intervenir en el planteamiento de su ejecución; mientras que en el caso del multihilo, como el

programa debe ser diseñado expresamente para que pueda soportar esta característica, es imprescindible que el autor tenga que planificar adecuadamente la ejecución de cada hilo, o tarea.

Actualmente hay diferencias en la especificación del intérprete de Java, porque el intérprete de Windows '95 conmuta los hilos de igual prioridad mediante un algoritmo circular (round-robin), mientras que el de Solaris 2.X deja que un hilo ocupe la CPU indefinidamente, lo que implica la inanición de los demás. ////// **Actualizar esto** /////

4. PROGRAMAS DE FLUJO UNICO

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

```
public class Paradigmas {
        static public void main( String args[] ) {
          System.out.println( "Me encanta analizar los paradigmas de la
                programación" ); }}
```

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

Debido a que la mayor parte de los entornos operativos no solían ofrecer un soporte razonable para múltiples hilos de control, los lenguajes de programación tradicionales, tales como C++, no incorporaron mecanismos para describir de manera elegante situaciones de este tipo. La sincronización entre las múltiples partes de un programa se llevaba a cabo mediante un bucle de suceso único. Estos entornos son de tipo síncrono, gestionados por sucesos. Entornos tales como el de Macintosh de Apple, Windows de Microsoft y X11/Motif fueron diseñados en torno al modelo de bucle de suceso.

5. PROGRAMAS DE FLUJO MULTIPLE

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

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

Los navegadores utilizan diferentes hilos ejecutándose en paralelo para realizar varias tareas, "aparentemente" concurrentemente. Por ejemplo, en muchas páginas web, se puede desplazar la página e ir leyendo el texto antes de que todas las imágenes estén presentes en la pantalla. En este caso, el navegador está trayéndose las imágenes en un hilo de ejecución y soportando el desplazamiento de la página en otro hilo diferente.

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

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

6. PLANIFICACION DE HILOS

Los hilos realizan distintas tareas dentro de nuestras aplicaciones, y estas tareas pueden tener asignadas diferentes niveles de prioridad. Para ello cada hilo tiene asociada una *prioridad* utilizada por el sistema en tiempo de ejecución como ayuda para determinar qué hilo debe ejecutarse en un instante determinado. Los programas se pueden ejecutar en máquinas

monoprocesadoras y multiprocesadoras, y podemos realizar la ejecución utilizando un solo hilo o múltiples hilos, de forma que las garantías de la planificación de hilos son muy generales. En un sistema con *N* procesadores disponibles, veremos ejecutándose generalmente los *N* hilos de mayor prioridad que sean ejecutables. Los hilos de menos prioridad sólo se ejecutarán generalmente cuando los hilos de mayor prioridad estén bloqueados (no sean ejecutables). Pero de hecho, los hilos de menor prioridad podrían ejecutarse en otras ocasiones (*crecimiento de la prioridad)* para evitar la inanición.

Un hilo en ejecución continúa ejecutándose hasta que realiza una operación de bloqueo (como wait, sleep o la realización de algunos tipos de *E/S),* o hasta que es desalojado. Un hilo puede ser desalojado por otro hilo de mayor prioridad que se hace ejecutable, o porque el planificador de hilos decide que es el momento de que otro hilo tome algunos ciclos de CPU. Esta *división temporal* limita la cantidad de tiempo que un hilo puede estar ejecutándose antes de ser desalojado.

Cuándo ocurre exactamente el desalojo depende de la máquina virtual que tengamos. No hay garantía, sino sólo la esperanza general de que la preferencia se dará generalmente a los hilos de mayor prioridad en ejecución. Es conveniente utilizar la prioridad sólo para afectar a la política de planificación con el objetivo de mejorar la eficiencia. No es conveniente basar el diseño de un algoritmo en la prioridad de los hilos. Para escribir código con múltiples hilos que se ejecute correctamente en diversas plataformas debemos asumir que **un hilo puede ser desalojado en cualquier momento**, entonces es necesario proteger siempre el acceso a los recursos compartidos. Si deseamos que el desalojo ocurra en un momento determinado, debemos utilizar los mecanismos explícitos de comunicación entre hilos, como **wait y notify**. Tampoco podemos suponer un orden en:

- La concesión de bloqueos a los hilos.
- La recepción de notificaciones por parte de los hilos en espera.

La prioridad de un hilo es inicialmente la prioridad del hilo que lo creó. Esta prioridad se puede cambiar utilizando **setPriority** con un valor entre las constantes de Thread denominadas **MIN_PRIORITY y MAX_PRIORITY**. La prioridad por defecto es **NORM_PRIORITY**. La prioridad de un hilo en ejecución se puede cambiar en cualquier momento. Si cambiamos la prioridad de un hilo por otra menor, el sistema puede permitir que otro hilo comience a ejecutarse, ya que el hilo original podría no estar entre los de prioridad más alta. El método **getPriority** devuelve la prioridad de un hilo.

Generalmente, la parte de nuestra aplicación que se ejecuta continuamente debería ejecutarse utilizando un hilo de menor prioridad que la parte que gestiona eventos. Cuando el usuario pulsa un botón "Cancelar", por ejemplo, espera que la aplicación cancele lo que está haciendo. Si la actualización de la pantalla y la entrada de usuario tienen la misma prioridad y la pantalla se está actualizando, puede pasar un tiempo considerable antes de que el hilo que gestiona la entrada de usuario responda al botón. Aun con prioridad mayor el hilo que responde al evento "Cancelar" puede demorar en desalojar al hilo que actualiza la pantalla. A un hilo que realiza actualizaciones continuas se le asigna la prioridad NORM_PRIORITY - 1. A un hilo de control de eventos se le asigna normalmente la prioridad NORM_PRIORITY+1.

7. LA CLASE THREAD

Es la clase que encapsula todo el control necesario sobre los hilos de ejecución (threads). Hay que distinguir claramente un objeto Thread de un hilo de ejecución o thread. Esta distinción resulta complicada, aunque se puede simplificar si se considera al objeto Thread como el panel de control de un hilo de ejecución (thread). La clase Thread es la única forma de controlar el comportamiento de los hilos y para ello se sirve de los métodos que se exponen en las secciones siguientes.

7.1 METODOS DE CLASE

Por convención, los métodos estáticos de la clase Thread siempre se aplican al hilo en ejecución en curso. Se utilizan para saber el estado del hilo o para replanificarlo, esto es, posibilitar que otros hilos puedan ejecutarse.

public static void sleep(long milis) throws InterruptedException

Hiberna el hilo actual en ejecución por lo menos el número de milisegundos especificado.

Si el hilo se interrumpe mientras está en hibernación, se lanza una InterruptedException.

Vamos a modificar el programa de flujo único transformandolo en multihilo. Generamos tres hilos de ejecución individuales, que imprimen cada uno de ellos su propio mensaje; La clase será MultiHola.java:

```
// Definimos unos sencillos hilos. Se detendrán un antes de imprimir sus
nombres y retardos. No se preocupe si no sabe exactamente que hace cada
sentencia.(Investíguelo...!)

class TestTh extends Thread {
     private String nombre;
     private int retardo;
     public TestTh( String s,int d ) { // Constructor para almacenar nuestro
          nombre = s;                  //  nombre y el retardo
          retardo = d;}
     // El metodo run() es similar al main(), pero para threads. Cuando
     //run() termina el thread muere
     public void run() {
           try {      // Retrasamos la ejecución el tiempo especificado
            sleep( retardo );
          }  catch( InterruptedException e ) {;}
          // Ahora imprimimos el nombre
          System.out.println( "Hola Java! "+nombre+" "+retardo ); } }

public class MultiHola {
     public static void main( String args[] ) {
          TestTh t1,t2,t3;  // Definimos/Creamos los threads
          t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
          t2 = new TestTh( "Thread 2",(int)(Math.random()*2000) );
          t3 = new TestTh( "Thread 3",(int)(Math.random()*2000) );
          t1.start(); // Arrancamos los threads
          t2.start();
          t3.start(); }
} //  MltiHola
```

Al ejecutar este programa se notará que si bien los hilos no corren siempre en el mismo orden, ello es explicado debido a que el retardo es aleatorizado. Un par de captures ...

```
Hola Java! Thread 1 419
Hola Java! Thread 2 995
Hola Java! Thread 3 1387
Process Exit...
```

```
Hola Java! Thread 2 1486
Hola Java! Thread 3 1754
Hola Java! Thread 1 1791
Process Exit...
```

Ahora bién, que ocurrirá si en el método run ejecutamos el println dentro de un ciclo que también tiene su propio retardo, y el ciclo es suficientemente durable como para mantener el hilo ejecutando cuando ya vencieron los tiempos de retardo de los otros hilos ? Investigue.

public static void yield()

Proporciona a la máquina virtual Java la información de que el hilo actual no necesita ejecutarse en el presente momento. La máquina puede seguir esta sugerencia o ignorarla, según le parezca adecuado.
El siguiente programa ilustra cómo yield puede afectar a la planificación de hilos. La aplicación es ejecutada acompañada de una lista de parámetros. El primero, de tipo lógico, informa si los hilos efectuarán yield tras cada impresión. El segundo, cantidad de veces que cada hilo debe repetir su palabra. A continuación, las palabras a repetir.

```
//  Nuestro programa :
class Contar extends Thread {
    static boolean yie;   //  Comparto CPU con otros hilos ?
    static int ciclo;         //  Cuántas veces imprimir
   private String palabra; //     Mi palabra
    Contar(String pal) { palabra = pal;}
    public void run()  {
      for (int i = 0; i < ciclo; i++) {
```

```
        System.out.print(palabra+" ");
        if (yie)
           yield();     // oportunidad a otro hilo
      }
   }

    public static void main(String[] args) {
      yie   = new Boolean(args[0]).booleanValue();
     ciclo = Integer.parseInt(args[1]);
     System.out.println("-------------");
     //      hilos para 3  caracteres
      for (int i = 2; i < args.length; i++)
        new Contar(args[i]).start();
      System.out.println("=============");
    }
}
```

Para analizar lo que ocurre escribimos los dos comandos:

```
java Contar false   3  '1' '2' '3'
java Contar true    3  '1' '2' '3'
```

en un archivo **Contar.bat** y lo grabamos en el directorio de java. Asimismo trasladamos Contar.class a ese mismo directorio.

Una primera ejecución de Contar.bat

```
F:\Java\jdk1.3\bin>java Contar false   3  '1' '2' '3'
-------------
=============
'2' '2' '2' '1' '1' '1' '3' '3' '3'
F:\Java\jdk1.3\bin>java Contar true    3  '1' '2' '3'
-------------
=============
'2' '3' '2' '3' '1' '3' '1' '2' '1'
F:\Java\jdk1.3\bin>
```

Y una segunda:

```
F:\Java\jdk1.3\bin>java Contar false   3  '1' '2' '3'
-------------
=============
'2' '2' '2' '1' '1' '1' '3' '3' '3'
F:\Java\jdk1.3\bin>java Contar true    3  '1' '2' '3'
-------------
=============
'2' '1' '2' '3' '2' '3' '1' '1' '3'
F:\Java\jdk1.3\bin>
```

Cuando los hilos no usan yield, cada uno de ellos utiliza grandes intervalos de tiempo, **probablemente** lo suficientemente grandes como para asegurar que todas las impresiones finalizan sin que otro hilo obtenga ciclos mientras tanto.

Si el alumno está ejecutando un programa multihilo por primera vez, quedará sorprendido por la salida. Es necesario ejecuarlo repetidas veces y comparar. Luego, en el .bat agregue parámetros (4,5,6), o bien incremente el límite del ciclo, ejecute nuevamente. Como verá, aún sin yield no hay garantías de terminar un hilo sin interrupciones.

currentThread()

Este método devuelve el objeto thread que representa al hilo de ejecución que se está ejecutando actualmente.

7.2 METODOS DE INSTANCIA

Se los llama así por estar directamente relaciodos con objetos. Los detallamos:

suspend()

El método suspend() es distinto de stop(). suspend() toma el hilo y provoca que se detenga su ejecución sin destruir el hilo de sistema subyacente, ni el estado del hilo

anteriormente en ejecución. Si la ejecución de un hilo se suspende, puede llamarse a resume() sobre el mismo hilo para lograr que vuelva a ejecutarse de nuevo.

Puede resultar útil suspender la ejecución de un hilo sin marcar un límite de tiempo. Si, por ejemplo, está construyendo un applet con un hilo de animación, seguramente se querrá permitir al usuario la opción de detener la animación hasta que quiera continuar. No se trata de terminar la animación, sino desactivarla. Para este tipo de control de los hilos de ejecución se puede utilizar el método suspend().

```
t1.suspend();
```

resume()

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

setName(String)

Este método permite identificar al hilo con un nombre mnemónico. De esta manera se facilita la depuración de programas multihilo. El nombre mnemónico aparecerá en todas las líneas de trazado que se muestran cada vez que el intérprete Java imprime excepciones no capturadas.

getName()

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

interrupt()

El último elemento de control que se necesita sobre los hilos de ejecución es su detención.

En programas simples es posible dejar que el hilo termine "naturalmente". En programas mas complejos se requiere gestionar esta finalización. Para ello se definió el método stop(). Debido a serios defectos, fué desestimado(Ya JDK 1.3 lo señala como error de compilación). En su lugar debe usarse interrupt(), que permite la cancelación cooperativa de un hilo.

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

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

t1.isAlive();

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

En el ejemplo no hay problemas de realizar una parada incondicional, al estar todos los hilos vivos.

7. 2.1 CREACION DE UN THREAD

```
Thread objeto = new Thread();
```

Tras crear un objeto Thread, podemos configurarlo y ejecutarlo. La configuración de un hilo requiere establecer su prioridad inicial, su nombre, etc. Cuando el hilo está listo para ejecutarse, se invoca al método start.

7. 2.2 ARRANQUE DE UN THREAD

```
objeto.start()
```

El método start lanza un nuevo hilo de control basado en los datos del objeto Thread, y después retorna. Ahora la máquina virtual invoca al nuevo **método run** del hilo, con lo que el hilo se activa. Sólo se puede invocar a start una vez para cada hilo. Invocarlo otra vez produciría una excepción IllegalThreadStateException.

Cuando se retorna de un método run de un hilo, el hilo ha finalizado. Podemos requerir que un hilo termine invocando al método interrupt(). Mientras un hilo se está ejecutando, se puede interactuar con él de otras formas, como veremos.

La implementación estándar de Thread.run no hace nada. Hay dos modos de conseguir hilos de ejecución (threads) en Java. Una es extender la clase thread, la otra es implementar la interfaz runnable. Un mismo ejemplo aplicado en ambas modalidades:

7.2.1.1 *Extendiendo Thread, 10 hilos contando*

```
 //////  Ejemplo 1 - CountThreadTest.java    Inicio  /////////////
 * <p>
 * Cuando comienza a correr, este nuevo Thread imprime en pantalla
 * una cuenta en números ascendentes.
 * @author Gus
 * @version 1.0
 */
class CountThreadTest extends Thread {
     int from; //  Límite inferior del conteo.
     int to;   //  Límite superior.
     public CountThreadTest(int from, int to) { //  Constructor
       this.from = from;
       this.to = to;
     }

     public void run() {
       for (int i=from; i<to; i++) {
        System.out.println(toString() + " i : " + i);

/* toString(): método del paquete java.lang, clase Thread, (Para este caso, pues
existen una enormidad de ellos) informa el número de hilo, prioridad y grupo
*/
       }   // for
      }     //  void run()

/* Punto de inicio del programa, donde se crean 10  <code>Threads</code>
cada una de las cuales se ejecuta en los momentos que la máquina virtual le
asigna.    */

     public static void main(String[] args) {
         for (int i=0; i<10; i++) {  // Creamos 10 hilos, los
             // inicializamos p/intervalos  0/9, 10/19, ..., 90/99
              CountThreadTest t = new CountThreadTest(i*10,(i+1)*10);
         t.start(); // start invoca a run de CountThreadTest
         }  // for
    }   // void main()
}       //  class CountThreadTest

/*  En este ejemplo, al comenzar a correr cada hilo no pierde el control hasta exhibir su
secuencia de 10 números. En cambio, el orden de ejecución no está garantizado, pese a ser
"start()ados"  dentro de un for ...                              */
```

//////// Ejemplo CountThreadTest.java Fin /////////////

Un capture de la ejecución:

```
Class Path = .;C:\Kawa4.01\kawac
Thread[Thread-0,5,main] i = 0
Thread[Thread-0,5,main] i = 1
Thread[Thread-0,5,main] i = 2
Thread[Thread-0,5,main] i = 3
Thread[Thread-0,5,main] i = 4
Thread[Thread-0,5,main] i = 5
Thread[Thread-0,5,main] i = 6
Thread[Thread-0,5,main] i = 7
Thread[Thread-0,5,main] i = 8
Thread[Thread-0,5,main] i = 9
Thread[Thread-1,5,main] i = 10
Thread[Thread-1,5,main] i = 11
Thread[Thread-1,5,main] i = 12
Thread[Thread-1,5,main] i = 13
Thread[Thread-1,5,main] i = 14
Thread[Thread-1,5,main] i = 15
Thread[Thread-1,5,main] i = 16
Thread[Thread-1,5,main] i = 17
Thread[Thread-1,5,main] i = 18
Thread[Thread-1,5,main] i = 19
Thread[Thread-3,5,main] i = 30
Thread[Thread-3,5,main] i = 31
Thread[Thread-3,5,main] i = 32
Thread[Thread-3,5,main] i = 33
Thread[Thread-3,5,main] i = 34
```

Hemos visto que Thread se puede extender para proporcionar los cálculos específicos del hilo, pero esta solución en muchos casos resulta engorrosa. En primer lugar, la extensión de una clase es herencia simple. Si extendemos una clase para hacerla ejecutable en un hilo, no podremos extender ninguna otra clase, aunque lo necesitemos. Además, si nuestra clase necesita sólo ser ejecutable, heredar toda la sobrecarga de Thread no es necesario.

La implementación de Runnable es más fácil en muchos casos. Podemos ejecutar un objeto Runnable en su propio hilo pasándoselo a un constructor de Thread. Si se construye un objeto Thread con un objeto Runnable, la implementación de Thread.run invocará al método run del objeto ejecutable.

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

Hay una cuantas diferencias entre interfaz y clase que aquí solamente se resumen. Primero, un interfaz solamente puede contener métodos abstractos y/o variables estáticas y finales (constantes). Las clases, por otro lado, pueden implementar métodos y contener variables que no sean constantes. Segundo, un interfaz no puede implementar cualquier método. Una clase que implemente un interfaz debe implementar todos los métodos definidos en ese interfaz. Un interfaz tiene la posibilidad de poder extenderse de otros interfaces y, al contrario que las clases, puede extenderse de múltiples interfaces.

En este caso necesitamos crear una instancia de Thread antes de que el sistema pueda ejecutar el proceso como un hilo. Además, el método abstracto run() está definido en el interfaz Runnable y tiene que ser implementado. La única diferencia entre los dos métodos es que este último es mucho más flexible. En el ejemplo anterior, todavía está la oportunidad de extender la clase MiThread, si fuese necesario. La mayoría de las clases creadas que necesiten ejecutarse como un hilo, implementarán el interfaz Runnable, ya que probablemente extenderán alguna de su funcionalidad a otras clases.

No pensar que el interfaz Runnable está haciendo alguna cosa cuando la tarea se está ejecutando. Solamente contiene métodos abstractos, con lo cual es una clase para dar idea sobre el diseño de la clase Thread. De hecho, si se observan los fuentes de Java, se puede comprobar que solamente contiene un método abstracto:

7.2.1.2 *Implementando Runnable, 10 hilos contando*

```
//////  Ejemplo 2 - CountThreadTest.java    Inicio  /////////////

class CountThreadTest{

   public static void main(String[] args) {
         for (int i=0; i<10; i++) {  // Creamos 10 hilos, los
             // inicializamos p/intervalos  0/9, 10/19, ..., 90/99
                    Thread t = new Thread(new masUnHilo(i*10,(i+1)*10));
             t.start(); // start invoca a run de CountThreadTest
         }  // for

   }   // void main()
}       //  class CountThreadTest

class masUnHilo implements Runnable {
       int from; //  Límite inferior del conteo.
       int to;   //  Límite superior.
       public masUnHilo(int from, int to) { //  Constructor
         this.from = from;
         this.to = to;
       }
       public void run() {
              for (int i=from; i<to; i++) {
             System.out.println(toString() + " i : " + i);
         }  //  for
    }        //  void run()
}

////////  Ejemplo CountThreadTest.java     Fin  /////////////
```

```
Class Path = .;C:\Kawa4.01\kawacla
Thread[Thread-0,5,main] i : 0
Thread[Thread-0,5,main] i : 1
Thread[Thread-0,5,main] i : 2
Thread[Thread-0,5,main] i : 3
Thread[Thread-0,5,main] i : 4
Thread[Thread-0,5,main] i : 5
Thread[Thread-0,5,main] i : 6
Thread[Thread-0,5,main] i : 7
Thread[Thread-0,5,main] i : 8
Thread[Thread-0,5,main] i : 9
Thread[Thread-2,5,main] i : 20
Thread[Thread-2,5,main] i : 21
Thread[Thread-2,5,main] i : 22
Thread[Thread-2,5,main] i : 23
Thread[Thread-2,5,main] i : 24
Thread[Thread-2,5,main] i : 25
```

8. GRUPOS DE HILOS

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

Los grupos de hilos permiten que sea posible recoger varios hilos de ejecución en un solo objeto y manipularlo como un grupo, en vez de individualmente. Por ejemplo, se pueden regenerar los hilos de un grupo mediante una sola sentencia.

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

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

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

9. SCHEDULING

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

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

10. HILOS DEMONIO

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

Los hilos demonio son útiles cuando un hilo debe ejecutarse en segundo plano durante largos períodos de tiempo.Un ejemplo de hilo demonio que está ejecutándose continuamente es el recolector de basura (garbage collector). Este hilo, proporcionado por la Máquina Virtual Java, comprueba las variables de los programas a las que no se accede nunca y libera estos recursos, devolviéndolos al sistema.

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

11. SINCRONIZACION

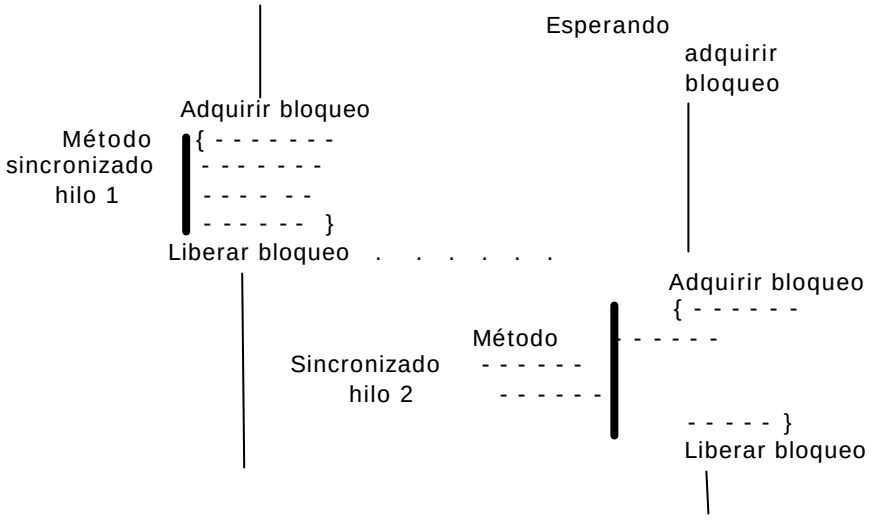
Cuando dos hilos necesitan utilizar el mismo archivo (objeto), aparece la posibilidad de operaciones entre-lazadas que pueden corromper los datos. Estas acciones potencialmente interferentes se denominan *secciones críticas* o *regiones críticas.* La *interferencia* se puede evitar *sincronizando* el acceso a esas regiones críticas. La acción equivalente cuando hay múltiples hilos es adquirir un *bloqueo* de un objeto. Los hilos cooperan acordando el protocolo de que antes de que ciertas acciones puedan ocurrir en un objeto se debe adquirir el bloqueo del objeto. Adquirir el bloqueo de un objeto impide que cualquier otro hilo adquiera ese bloqueo, hasta que el que posee el bloqueo lo libere. Si se hace correctamente, los múltiples hilos no realizarán acciones que puedan interferir entre si.

Todos los objetos tienen un bloqueo asociado, que puede ser adquirido y liberado mediante el uso de métodos y sentencias synchronized. El término código sincronizado describe a cualquier código que esté dentro de un método o sentencia synchronized.

11.1 METODOS SYNCHRONIZED

Una clase cuyos objetos se deben proteger de interferencias en un entorno con múltiples hilos declara generalmente sus métodos modificadores como synchronized. Si un hilo invoca a un método synchronized sobre un objeto, en primer lugar se adquiere el bloqueo de ese objeto, se

ejecuta el cuerpo del método y después se libera el bloqueo. Otro hilo que invoque un método synchronized sobre ese mismo objeto se bloqueará hasta que el bloqueo se libere:



La sincronización fuerza a que la ejecucion de los dos hilos sea *mutuamente exclusiva* en el tiempo. Los accesos no sincronizados no respetan ningún bloqueo, proceden independientemente de cualquier bloqueo

La posesión de los bloqueos es único ***por hilo***. Un método sincronizado **invocado desde el interior de un método sincronizado sobre el mismo objeto procederá sin bloqueo**. El bloqueo se liberará al concluir el cuerpo del método sincronizado externo. Este comportamiento de posesión por hilo impide que un hilo se bloquee por un bloqueo que ya posee, y permite invocaciones recursivas a métodos y además podemos invocar métodos heredados, que pueden estar sincronizados en su clase, sin bloqueo.

El bloqueo se libera tan pronto como termina el método sincronizado:

- normalmente porque se llega a una sentencia return, o
- se alcanza el final del cuerpo del método, o
- anormalmente lanzándose una excepción.

A diferencia de los sistemas donde los bloqueos/desbloqueos deben ser explícitamente declarados, este esquema de sincronización hace que sea imposible olvidarse de liberar un bloqueo.

La sincronización hace que el ejemplo de ejecución entrelazada funcione: si el código está en un método sincronizado, entonces cuando el segundo hilo intenta acceder al objeto mientras el primer hilo está utilizándolo, el segundo hilo se bloquea hasta que el primero finaliza.

Por ejemplo, una clase Cuenta funcionando en un entorno multihilo:

```
class Cuenta {
   private double saldo;
    public Cuenta(double apertura)  { saldo = apertura; }
    public synchronized double getsaldo() { return saldo;}
    public synchronized void deposito(double cantidad) { saldo += cantidad; }
}
```

Los constructores no necesitan ser synchonized porque se ejecutan sólo cuando se crea un objeto, y eso sólo sucede una vez para cada objeto. No pueden ser declarados synchronized.

El campo saldo se defiende de una modificación no sincronizada haciendo que sus métodos de acceso sean synchronized. Si el valor de un campo se puede modificar, su valor nunca se debe leer al mismo tiempo que otro hilo está modificándolo. Si un hilo leyera el valor mientras otro lo modifica, la lectura podría devolver un valor inválido. Incluso aunque el valor fuera válido, la secuencia de lectura, modificación y escritura requiere que el valor no cambie entre la

lectura y la escritura. En caso contrario el valor escrito no será válido. El acceso al campo debe ser sincronizado.

Utilizando la declaración synchronized tenemos la garantía de que dos o más hilos que se están ejecutando no interfieren entre sí. Si dos métodos se ejecutan con exclusión mutua, una vez la invocación a un método inicia su ejecución no puede iniciar la ejecución del segundo método hasta que la original se haya completado. Sin embargo, no hay garantía sobre el orden de las operaciones. Si se pide el saldo aproximadamente al mismo tiempo que se produce el depósito, uno de ellos se completará primero, pero no podemos decir cuál. Si deseamos que las acciones ocurran en un orden determinado, los hilos deben coordinar sus actividades de alguna forma que ya es específica de cada aplicación.

Cuando una clase extendida redefine a un método synchronized, el nuevo método puede ser synchronized o no. El método de la superclase será synchronized cuando se invoque. Si el método no sincronizado de la subclase utiliza **super** para invocar al método de la supercíase, el bloqueo del objeto se adquirirá en ese momento y se liberará cuando se vuelva del método de la superclase.

Los requerimientos de sincronización son parte de la implementación de una clase. Una clase extendida puede ser capaz de modificar una estructura de datos de forma que las invocaciones concurrentes de un método no interfieran y, por tanto, el método no necesitará ser sincronizado. Inversamente puede ocurrir que la clase extendida modifique el comportamiento de un método de forma que la interferencia sea posible; entonces debe ser sincronizado.

Incluimos un ejemplo. En él, creamos un arreglo de 1024 enteros generados aleatoriamente . Los hilos esclavos(4 o la cantidad parámetro de la línea de comando) procesan cada uno subrangos equitativos del array en forma simultánea, obteniendo los respectivos máximos parciales. Luego, usando **exclusión mutua** mediante métodos sincronizados a nivel instancia (**getmax(), setmax()**) obtenemos el máximo general.

```
class Maxguard {
       private int maxValue;

       public Maxguard (int e) {
         maxValue=e;
       }
       public synchronized int getMax() {
              return maxValue;
       }
       public synchronized void setMax(int e) {
              maxValue=Math.max(maxValue, e);
       }
}

class Encontrar extends Thread {
       private int l_max;
       private int low, high;
       private int incr=10;
       private int[] arr;
       private Maxguard xobj;

       public Encontrar(int[] ar, int l, int h, Maxguard m) {
         arr=ar;
         low=l+1;
         high=h;
         l_max=arr[1];
         xobj=m;
       }
       public void run() {
         try
         {
              while(low<high)
              {
                     int n= Math.min(incr,high-low);
                     for(int i=low;i<=low+n;i++)
                     {
                            l_max=Math.max(l_max,arr[i]);
```

```
                        }
                        sleep(5);
                        low+=n;
                    }
                } catch (InterruptedException e) {
                    return;
                }
                xobj.setMax(l_max);
            }
        }

        class Maximo {
         public static void main(String[] args) throws InterruptedException  {
              int[] a= new int[1024];
              for(int i=0;i<1024;i++)
              {
                    a[i]=(int)(100000*Math.random());
              }
              int nt=0;
              if(args.length==1)
              {
              nt=Integer.parseInt(args[0]);
              }
              if(nt<=0) nt=4;
              Maxguard ans=new Maxguard(a[0]);
              Encontrar[] slave=new Encontrar[nt];
              int range=1024/nt;
              int high,low=0;
              for(int i=0;i<nt-1;i++)
              {
                  high=low+range-1;
                  slave[i]=new Encontrar(a,low, high, ans);
                  low=high+1;
              }
              slave[nt-1]=new Encontrar(a,low,1023, ans);
              for(int i=0;i<nt;i++) slave[i].start();
              for(int i=0;i<nt;i++) slave[i].join();
              System.out.println("Maximo= "+ans.getMax());
          }
        }
```

Un primer capture

```
Class Path - .;C:\Kawa4.
Maximo= 99994
Process Exit...
```

Un segundo

```
Class Path - .;C:\Kawa.
Maximo= 99776
Process Exit...
```

11.2 METODOS ESTATICOS SINCRONIZADOS

Una segunda forma de obtener sincronización. Los métodos estáticos también se pueden sincronizar. Todos los objetos tienen asociado un objeto Class. Los métodos estáticos sincronizados adquieren el bloqueo del objeto Class de su clase. Dos hilos no pueden ejecutar metodos estáticos sincronizados de la misma clase al mismo tiempo, de la misma forma que dos hilos no pueden ejecutar métodos sincronizados del mismo objeto al mismo tiempo. Si se comparten datos estáticos entre hilos, su acceso debe ser protegido utilizando métodos estáticos sincronizados.

La adquisición del bloqueo del objeto Class en un método estático sincronizado no afecta a los objetos de esa clase. Se puede seguir invocando a métodos sincronizados de un objeto mientras otro hilo tiene el bloqueo del objeto Class en un método estático sincronizado. El bloqueo entre métodos de instncia y estático s es independiente.

11.3 LA SENTENCIA synchronized

Una tercera opción. La sentencia synchronized nos permite ejecutar código sincronizado que adquiere el bloqueo de cualquier objeto, no sólo del objeto actual. Puede usarse para duraciones menores que la invocación completa del método. La sentencia synchronized tiene dos partes: un objeto cuyo bloqueo se va a adquirir y una sentencia que se ejecuta cuando se adquiere el bloqueo. La forma general de la sentencia synchronized es:

```
synchronized (expr) {
      sentencias
      }
```

La expresión *expr* debe tener el valor de una referencia a objeto. Cuando se obtiene el bloqueo, se ejecutan las *sentencias* del bloque, y al final del bloque el bloqueo se libera. Si se produce una excepción no captu rada en el interior del bloque, el bloqueo también se libera. Un método synchronized es abreviatura de un método cuyo cuerpo está envuelto en una sentencia synchronized con una referencia a this.

Presentamos a continuación un método para sustituir los elementos de un array por sus valores absolutos, que se basa en la sentencia synchronized para controlar el acceso al array:

```
/* Lleva todos los elementos de un array a su valor absoluto */
public static void abs(int[] valores) {
        synchronized (valores) {
           for (int i = 0; i < valores.length; i++) {
               if (valores[iJ < 0)
                     valores[i] = - valores[i];
           }          //  for
       }              //  synchronized (valores)
}                     //    public  static
```

//////////// Ejemplo con dos hilos trabajando sobre el mismo array. Un wait por tiempo aleatorizado antes de correr el bloque de sentencias, de manera que al exhibir los resultados se sepa en que orden corrieron los hilos. También una bandera registrando el orden ///////////////

El array valores contiene los elementos que se van a modificar. Sincronizamos **valores** nombrándolo como el objeto de la sentencia synchronized. Ahora el bucle puede proceder, garantizándose que el array **valores** no es modificado durante su ejecución por otro código que esté sincronizado similarmente con dicho array. Esto es un ejemplo de lo que se conoce generalmente como sincronización *en la parte del cliente.* Todos los clientes que utilizan el objeto compartido (en este caso el array) están de acuerdo en sincro nizarse sobre ese objeto antes de manejarlo. En objetos como arrays, ésta es la única forma de protegerlos cuando se puede acceder a ellos directamente, ya que no tienen métodos que puedan ser sincronizados.

La sentencia synchronized tiene diversos usos y ventajas adicionales sobre la sincronización de métodos. En primer lugar, se puede definir una región de código sincronizada más pequeña que un método. La sincronización afecta a las prestaciones (cuando un hilo tiene un bloqueo otro hilo no puede adquirirlo), y una regla general de programación concurrente es que se deben mantener los bloqueos tan poco tiempo como sea posible. Utilizando una sentencia synchronized podemos mantener el bloqueo sólo cuando es absolutamente necesario. Por ejemplo, es frecuente que un método que realice un cálculo complejo y después asigne el resultado a un campo necesite proteger sólo la asignación del campo, no el proceso de cálculo.

En segundo lugar, las sentencias synchronized nos permiten sincronizarnos sobre objetos diferentes de this, con lo que se pueden implementar diversos diseños diferentes de sincronización. Por ejemplo, una situación podría ser que deseáramos aumentar el nivel de concurrencia de una clase utilizando una granularidad más fina de los bloqueos. Ruede ser que grupos diferentes de métodos dentro de una clase actúen sobre diferentes datos de esa clase y, aunque se necesite **exclusión mutua dentro de cada grupo de métodos**, ésta no sea necesaria entre grupos. En lugar de hacer que todos los métodos sean sincronizados, podemos definir objetos diferentes que se pueden utilizar como bloqueos de cada grupo y hacer que los métodos utilicen sentencias synchronized para otener el bloqueo apropiado. Por ejemplo:

```
class GruposSeparados {
      private double valA = 0.0;
```

```
      private double valB = 1.1;
      protected Object bloqueoA = new Obiect();
         protected Object bloqueoB = new Object();
         public double getA() {
          synchronized (bloqueoA) { return valA;}}
         publlc double setA(double val) {
             synchronlzed (bloqueoA) {valA = val; }}
         public double getB() {
          synchronlzed (bloqueoB) {return valB;}}
         publlc double setB(double val)  {
          synchronized (bloqueoB) {valB = val;}}
         publlc double inicializar   {
            synchronlzed (bloqueoA) {
              synchronized(bloqueoB) {valA = valB = 0.0;}}
}
```

Las dos referencias de bloqueo son protected de forma que una clase extendida pueda sincronizar correctamente sus propios métodos (como por ejemplo un método que ponga en valA y valB el mismo valor). Nótese también que inicializar adquiere ambos bloqueos antes de modificar los dos valores.

////////// Podría ser un ejemplo donde bloqueoA se aplique sobre métodos de incrementación y decrementación sobre valA, corriendo en hilos separados. Idem para valB. Trabajamos con 5 hilos, contando el de main() //////////

11.4 DISEÑOS DE SINCRONIZACION

El diseño del esquema de sincronización adecuado para una clase es complejo. Consideraremos algunos aspectos.

11.4.1 SINCRONIZACIÓN EN LA PARTE DEL CLIENTE

Requiere que **todos los clientes de un objeto compartido utilicen sentencias synchronized** para adquirir el bloqueo del objeto compartido antes de acceder a él. Este acuerdo o protocolo es frágil, ya que se basa en que **todos** los clientes se comportan de forma correcta.

11.4.2 SINCRONIZACIÓN EN LA PARTE DEL SERVIDOR

Generalmente, es mejor opción hacer que los objetos compartidos protejan su propio acceso marcando sus métodos como synchronized (o utilizando sentencias synchronized apropiadas dentro de dichos métodos). Esto hace imposible que un cliente utilice un objeto de una forma no sincronizada. Esta solución se denomina a veces sincronización *en la parte del servidor,* pero es sólo una extensión de la perspectiva de orientación a objetos en la que los objetos encapsulan su propio comportamiento, incluida la sincronización.

Algunas veces un diseñador no ha considerado un posible entorno con múltiples hilos al diseñar una clase, y no tiene ningún método sincronizado. Como utilizar una clase así en un entorno con múltiples hilos ? Tenemos varias opciones:

- Escoger entre utilizar sincronización en la parte del cliente, mediante sentencias synchronized.
- Crear una clase extendida que redefina los métodos apropiados, los declare synchronized y redirija las llamadas a métodos mediante la referencia super.
- Trabajar con una interfaz en vez de una clase. En ella proporcionamos métodos sincronizados que redirijan las llamadas que se efectuarían sobre los métodos no sincronizados de la clase. Esto funcionará con cualquier implementación de la interfaz, y es por tanto mejor solución que extender todas las clases para utilizar super en los métodos sincronizados. Esta flexibilidad es otro motivo para que diseñemos nuestros sistemas utilizando interfaces.

12 COMUNICACION ENTRE HILOS

Para establecer un diálogo, comunicarse, las personas no pueden hablar simultaneamente. Hacerlo así es pura interferencia. Es necesaria una alternancia de emisión/recepción, y de alguna manera esto debe lograrse. (Con algunas personas es difícil) Entre hilos, lo que torna posible la comunicación es la sincronización, al evitar interferencia. El mecanismo de bloqueo de synchronized es suficiente para evitar que los hilos se interfieran,

corrompan objetos, pero también necesitamos formas (métodos) de comunicación entre ellos. Es la forma de llegar a una tarea resuelta "en equipo"

12.1 wait, notifyAll y notífy

Con este fin, el método wait deja a un hilo esperando hasta que ocurre alguna condición, y los métodos de notificación notifyAll y notify indican a los hilos que esperan que **ha ocurrido algo** que podría satisfacer esa condición. Los métodos wait y los de notificación se definen en la clase Object y son heredados por todas las clases. Se aplican a objetos particulares, como en el caso de los bloqueos.

La espera de un hilo hasta que se produzca una condición debería ser algo como esto:

```
synchronized void hacerCuandoCondicion() {
                while (!condícion) { //   Aguardamos, la
                      . . .
                      wait();...}   // condición no se cumple ...
                Hacer lo que se debe hacer ya que la condición es verdadera
...
        }
```

Aquí están ocurriendo diversas cosas:

- Todo debe ejecutarse dentro de código sincronizado. Si no se hiciera así, el estado del objeto no sería estable. Por ejemplo, si el método no se declarara synchronized, quién nos garantiza que la condición de while no fuera modificada indebidamente por otro hilo ?

- Uno de los aspectos importantes de la definición de wait es que cuando detiene al hilo, **libera simultaneamente el bloqueo sobre el objeto**. Otro hilo puede trabajar sobre él. Cuando el hilo se reinicia, el bloqueo se vuelve a adquirir inmediatamente.

- La condición de prueba debería estar *siempre* en un bucle. Nunca debe asumirse que ser despertado significa que la condición se ha cumplido, ya que puede haber cambiado de nuevo desde que se cumplió. En otras palabras, no se debe cambiar el while por un if.

Los métodos de notificación son asimismo invocados por código sincronizado que cambia una o más condiciones por las que algún otro hilo puede estar esperando. Un código de notificación típico puede ser como éste:

```
synchronlzed void cambiarCondicion()  {
          ... cambia algún valor utilizado en un test de condicion...
          notifyAll(); // o notify()
}
```

Al utilizar notifyAll se despiertan todos los hilos en espera, y notify selecciona sólo un hilo para despertarlo. Puede haber **múltiples hilos esperando sobre el mismo objeto**, incluso por condiciones diferentes.

Si esperan por condiciones diferentes, debe utilizarse siempre **notifyAll** para despertar a todos los hilos en espera, en vez de usar notify. Usando notify corremos el riesgo de despertar un hilo que está esperando por una condición diferente a la satisfecha por el hilo que emite la notificación. Ese hilo descubrirá que su condición no ha sido satisfecha y volverá al estado de espera, mientras que algún otro hilo que estuviera esperando por la condición satisfecha **nunca despertará**. El uso de **notify es una optimización** que sólo se puede aplicar cuando

- Todos los hilos están esperando por la misma condición.
- Sólo un hilo como mucho se puede beneficiar de que la condición se cumpla.
    - Esto es válido para todas las posibles subclases.

A continaución un ejemplo clásico de **sincronización y comunicación de hilos** de ejecución es un modelo productor/consumidor. Un hilo produce algo, que otro hilo usa

(consume). 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 tambien 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 de una forma eficiente, sin interferencias.

```
import java.lang.String;
// PRODUCTOR
class Productor extends Thread {
    private Pila pila;
    private String alfabeto = "ABCDEFGHIJ#";

    public Productor( Pila letra){
       // copia del objeto compartido
        pila = letra;
    }
    public void run() {
        char letra = alfabeto.charAt(0); ;
        for( int i=1; letra!='#'; i++ ) //  Apila letras
        {
            pila.apilar( letra );
            System.out.println("  "+letra);
            letra = alfabeto.charAt(i);
                 try {             //  Da una chance al hilo consumidor
                    sleep( (int)(Math.random() * 200 ) );
            } catch( InterruptedException e ) {;}
        }                  //  for
             pila.FinPro();     //  Fin de la producción
    }
}

//  Notar que se crea una instancia de la clase Pila, y que
// se utiliza el método Pila.apilar() para ir produciendo.

// CONSUMIDOR
class Consumidor extends Thread {
    private Pila pila;
    public Consumidor( Pila t) {
    pila = t;
    }

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

//  En este caso, como en el del productor, se cuenta con un método
// en la clase Pila, pila.recoger(), para consumir. Observe Ud.
// que nuestro consumidor dejará la pila impecablemente vacia ...

// MONITOR
class Pila {
    private char buffer[] = new char[10];
```

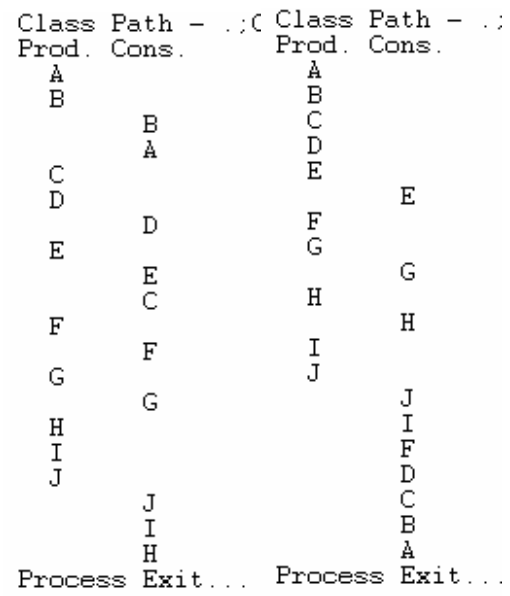
```
    private int siguiente = 0;
    // Sigen flags para saber el estado del buffer
    private boolean estaLlena = false;
    private boolean estaVacia = true;
    private boolean esperanza = true; // (De producción)
    public  static void encabe() {System.out.println("Prod. Cons.");}
    public synchronized char sacar() {  // Método para consumir
        while(estaVacia) //  Ciclo de espera por algo
        {                // para consumir; cuando haya
            try {        // saldremos  ...
                wait();
            } catch( InterruptedException e ) {;}
        }
        //  No está vacía, entonces
             siguiente--; // decrementamos, vamos consumir
        if( siguiente == 0 )  //  Fué la última, ya
            estaVacia = true; // no quedan mas letras
        estaLlena = false; // (Acabamos de consumir)
        notify();
        // Devuelve la letra al thread consumidor
        return( buffer[siguiente] );
    }

    public synchronized void apilar( char letra ) { // Método para producir
        while( estaLlena == true ) //  Ciclo de espera por sitio donde
        {                          // almacenar esta nueva producción.
            try {                  //  Lo habrá cuando recoger() cambie
                wait();            // esta bandera a false
            } catch( InterruptedException e ) {;}
       }
        buffer[siguiente++] = letra; // Incluimos en el buffer
        if( siguiente == 10 ) // Comprueba si el buffer está lleno
            estaLlena = true;
        estaVacia = false;
        notify();
    }
    public synchronized void FinPro() {     // Fin de la producción
             esperanza=false;
    }
    public synchronized boolean HayProd() { // Hay aún producción ?
       return esperanza;
    }
    public synchronized boolean HayStock() { // Pila con contenido ?
       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. Aquí se observa 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 sincronizados de acceso 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. (puede retirar otras). Está asegurada la integridad de cualquier objeto compartido. El método notify() al final de cada método de acceso avisa a cualquier proceso que esté esperando por el objeto, entonces el proceso que ha estado esperando intentará acceder de nuevo al objeto. En el método wait() se hace que el hilo se quede a la espera de que le llegue un notify(), ya sea enviado por el hilo de ejecución o por el sistema.

Ahora que ya se dispone de un productor, un consumidor y un objeto compartido, se necesita una aplicación que arranque los hilos y que consiga que todos traten con el mismo objeto que están compartiendo. Esto es lo que hace el siguiente trozo de código: */

```
class javaejemplo {
    public static void main( String args[] ) {
        Pila.encabe();   // Un método estático
        Pila pil = new Pila();
        Productor  pro = new Productor( pil );
        Consumidor con = new Consumidor( pil );
        pro.start();
        con.start();
    }
}
```

// Un primer capture , y luego otro ...



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 object como forma de independizarse de tipos de datos específicos.

Cuando se añade un elemento a la cola, (método **inclUlt)** 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 inclUlt(), en este caso el run() de WorkStat.

Volviendo al ejemplo del servidor de impresion, 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 **exclPri()**, utilizando la CPU todo el tiempo, situación conocida como ***espera con ocupación***. En un sistema con múltiples hilos, desearemos muy raramente 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 descripto, usa simultaneamente dos mecánicas de bloqueo:

- **Métodos sincronizados a nivel instancia**, que cuidan de la no corrupción del objeto cola.

- **Métodos estáticos sincronizados a nivel clase**, responsabls de la correcta implementación de la mecánica de impresión deseada: Diez textos por línea, un total de 40 textos. Completada esta impresión, termina la sesión. Se usan reiteradamente los mismos diez hilos productores, y dos consumidores, trabajando sobre un único objeto cola. La codificación es la siguiente:

```
class Cola                              //  oficia de monitor
{     Celda primero;                    //  punteros al primer
      Celda ultimo;                     //  y último elemento
      static int tCon=0;                //  total consumido

      Cola()                            //  constructor sin argumentos
      {     primero=ultimo=null;}

      /*    Los métodos estáticos sincronizados  endFila() y contSes()
            implementan bloqueo a nivel clase. Este bloqueo en necesario
            para conseguir imprimir los cuarenta textos que generan 10
            hilos productores y deben ser impresos a razón de 10 p/línea. */

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

      static synchronized boolean contSes()        //    Continúa la sesión ?
      {if(tCon<40) return true;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
            WorkStat) y los descarga en 2 hilos consumidores (clase PrintServ)
*/

      public synchronized void inclUlt(Object texto) throws NullPointerException
      {     if(primero==null)
                  primero=ultimo=new Celda(texto,null); // usamos constructor
con
            else                              //    argumentos de class Celda ...
                  ultimo=ultimo.siguiente=new Celda(texto,null);
            notifyAll();              //    Ola!!!, tengo trabajo ...
            return;
   }
      public synchronized void exclPri() throws InterruptedException
      {     Object textImp = "**************";     //    texto a imprimir
            while (primero==null & contSes())
                  //    No tenemos texto en cola, pero la sesión no terminó,
                  wait();                         //  esperemos que entre algo
...
      if (primero!=null)        //    Ya lo hizo !!!
            {     textImp = primero.elemento;
                  if(primero.equals(ultimo)) // Sólo un texto
                        primero=ultimo=null;
                  else primero=primero.siguiente;
                  System.out.print(textImp);
            }           //  if
      }                 //    exclPri
}                       //    class

class Celda
{     Celda siguiente;
      Object elemento;
      Celda(Object elemento) {    //  Constructor
            this.elemento = elemento;
      }
      public Celda(Object elemento, Celda siguiente) {
            this.elemento= elemento;
            this.siguiente=siguiente;
```

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

// PRODUCTOR
class WorkStat extends Thread { // Estaciones generan spool
	private Object elem;
	private Cola   cola;
	WorkStat(Cola cola, Object elem) {    //  Constructor
		this.elem = elem;
		this.cola = cola;
	 }
	public void run()
	{	try
		{	cola.inclUlt(elem); // método de instancia sincronizado
		}	catch (NullPointerException e)
			{	System.err.println("InclUlt():" + e.getMessage()); }
    }		//	run
}		//	class

// CONSUMIDOR
class PrintServ extends Thread { //    Servidores de impresión atienden spool
	private Cola   cola;
	PrintServ(Cola cola) {    //  Constructor
		this.cola = cola;
     }
     public void run()
       {	try
		{	do
			{
				cola.exclPri();
				if(Cola.endFila()) System.out.println();
			}
			while(Cola.contSes());
		}	catch (InterruptedException e)
			{	System.err.println("ExclPri():" + e.getMessage());
			}
	}
}     // class Consumidor

class javaejemplo
{
	private static int i;
	public static void main( String args[] )
	{	Cola col = new Cola();    //  Una sola vez construyo col
		for(i=1;i<=4;++i)
		{                                       //  usuarios generan
documentos ...
			WorkStat   wks0 = new WorkStat(col,"cero    ");
			WorkStat   wks1 = new WorkStat(col,"uno     ");
			WorkStat   wks2 = new WorkStat(col,"dos     ");
			WorkStat   wks3 = new WorkStat(col,"tres    ");
			WorkStat   wks4 = new WorkStat(col,"cuatro ");
			WorkStat   wks5 = new WorkStat(col,"cinco   ");
```

```
            //     primeros 6 usuarios quieren imprimir;
                   wks0.start();
                   wks1.start();
                   wks2.start();
                   wks3.start();
                   wks4.start();
                   wks5.start();

            // habilitamos 2 impresoras
                   PrintServ  prs1  = new PrintServ(col);
                   PrintServ  prs2  = new PrintServ(col);

            // las arrancamos
                   prs1.start();
                   prs2.start();

            // más 4 usuarios escribiendo
                   WorkStat   wks6 = new WorkStat(col,"seis   ");
                   WorkStat   wks7 = new WorkStat(col,"siete  ");
                   WorkStat   wks8 = new WorkStat(col,"ocho  ");
                   WorkStat   wks9 = new WorkStat(col,"nueve ");

            // más 4 quieren imprimir
                   wks6.start();
                   wks7.start();
                   wks8.start();
                   wks9.start();
            }            //    for
   }                     //    main
}                        //    class
```

Un capture

```
F:\Java\jdk1.3\bin\java.exe   javaejemplo
Working Directory - F:\Java\Introducción a Java\Hilos 2002\Cola\
Class Path -
F:\java\jdk1.3\demo\applets\SerieAri;.;F:\Java\kawaclasses.zip;F:\Java\jd
k1.3\lib\tools.jar;F:\Java\jdk1.3\jre\lib\rt.jar;F:\Java\jdk1.3\jre\lib\i
18n.jar

cero   dos    cuatro seis   siete  ocho   nueve  cero   uno    dos
tres   cuatro cinco  seis   siete  ocho   nueve  cero   uno    dos
tres   cuatro cinco  seis   siete  ocho   nueve  cero   uno    dos
tres   cuatro cinco  seis   siete  ocho   nueve  uno    tres   cinco
Process Exit...
```

Otro capture.

```
Class Path - .;C:\Kawa4.U1\kawaclasses.zip;t:\java\jdk1.3\lib\tools.jar;t
uno     tres    cinco   seis   siete  ocho  nueve cero    uno     dos
tres    cuatro cinco   seis   siete  ocho  nueve cero    uno     dos
tres    cuatro cinco   seis   siete  ocho  nueve cero    uno     dos
tres    cuatro cinco   seis   siete  ocho  nueve cero    dos     cuatro
Process Exit...
```

12.2 MAS SOBRE ESPERA Y NOTIFICACIÓN

Hay tres formas de realizar walt y dos formas de notificación. Todas ellas son métodos de la clase Object, y son todos final, de forma que su comportamiento no se puede cambiar:

public final void **wait(long tiempo> throws InterruptedException**

El hilo actual espera hasta que sucede una de estas cuatro cosas:

- se invoca a notify sobre este objeto y este hilo está seleccionado como ejecutable *(runnable);*

- se invoca a notifyAll sobre este objeto;
- el tiempo especificado expira;
- se invoca al método interrupt del hilo.

Tiempo se especifica en milisegundos. Si tiempo es cero, la espera se realizará indefinidamente hasta que se reciba una notificación. Durante la espera se libera el bloqueo del objeto, que se vuelve a adquirir automáticamente cuando walt finaliza (independientemente de cómo o por qué finaliza wait). Se lanza una InterruptedException si la espera finaliza porque el hilo es interrumpido.

public final void **wait(long tiempo, mt nanos)** throws InterruptedException

public final void wait() throws InterruptedException (Equivalente a wait (0))

public final void notifyAll()
Notifica a *todos* los hilos que están esperando a que cambie una condición. Los hilos volverán de la invocación de wait cuando puedan volver a adquirir el bloqueo del objeto.

public final void notify()

Notifica *como mucho* a un hilo que está esperando que cambió una condición. No podemos escoger a qué hilo se notifica, por lo que sólo es conveniente utilizar esta forma cuando estemos seguros de que sabemos qué hilos están esperando, qué están esperando y en qué momentos. Si no estamos seguros de todos esos factores, debemos utilizar notifyAll.

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 wait 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 sincronizado.

Cuando wait finaliza debido a que el tiempo especificado expira, no hay indicación de que ha ocurrido esto en vez de que el hilo haya sido notificado. Si un hilo necesita saber si el tiempo ha transcurrido o no, tendrá que llevar la cuenta del tiempo transcurrido. El uso del tiempo de espera es una medida de programación defensiva que nos permite recuperarnos de situaciones en las que una condición debería haberse cumplido, pero por alguna razón (probablemente un fallo en algún otro hilo), no se ha cumplido. Como el bloqueo del objeto se debe volver a adquirir, el uso del tiempo de espera no garantiza que wai retornará tras un periodo de tiempo finito.

//////// Escriba un programa que imprima el tiempo transcurrido desde el comienzo de su ejecución, con un hilo que imprima un mensaje cada quince segundos. Haga que el hilo de impresión del mensaje reciba notificaciones del hilo que imprime el tiempo. Añada otro hilo que imprima un mensaje diferente cada siete segundos sin modificar el hilo que imprime el tiempo.

13 FINALIZACIÓN DE LA EJECUCION DE HILOS

Un hilo que ha empezado a ejecutarse se convierte en un hilo *vivo* y el método **isAlive** aplicado a dicho hilo devolverá true. Un hilo continúa vivo hasta que termina, lo que puede ocurrir de tres maneras:

- El método run retoma normalmente.
- El método run finaliza bruscamente.
- El método destroy se invoca sobre ese hilo.
- Cuando el programa termina.

Hacer que run retorne es la forma normal de finalizar un hilo. Cada hilo realiza una tarea y cuando la tarea finaliza, el hilo debe irse. Si algo va mal, y ocurre una excepción que no se captura, el hilo también se terminará. En la sección xxx, "Hilos y excepciones", tratamos este punto. En el momento que un hilo termina no mantendrá más bloqueos, ya que debe haberse salido de todo el código sincronizado en el momento que run finaliza.

Invocar al método destroy sobre un hilo es algo drástico. El hilo se detiene en seco, independientemente de lo que estuviera haciendo y sin liberar los bloqueos que tuviera, por lo que utilizar destroy puede dejar a otros hilos bloqueados para siempre. El método destroy es un método que debe utilizarse como último recurso, cuando las técnicas de cancelación cooperativa que comentaremos a continuación no funcionan. En la práctica, muchos sistemas no han implementado nunca el método destroy, excepto para lanzar la excepción NoSuchMethodError (lo que puede finalizar el hilo que invoca, en vez del objetivo).

13.1 CANCELACIÓN DE UN HILO

A menudo necesitamos cancelar un trabajo antes de que se complete. El ejemplo más obvio 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 preferiría gestionar una acción de "cancelar" por parte del usuario tras 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 correctamente y 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; e 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 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 = 0; 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(10,1000);
            }
}           // prueCiclin
```

Un capture

```
Class Path = .;C:\Kawa4.01\kawaclasses.zip;f:\
   0  1  2  3  4  5  6  7  8  9Process 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 efectivamente logró interrumpirlo y luego el hilo interrumpido se ponga en marcha nuevamente. Usar interrupt, isInterrumped, interrumped. ////////////

13.2 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.

```
class  Factorial extends Thread {
int resultado,numero;
Factorial(int num)
```

```
{     numero=num;
}
public void run()
{
      try
      {
           resultado = factorial(numero);
      }    catch (InterruptedException e)
      {    System.out.println("problema método wait");
      }   //  bloque try
}

public  synchronized int getResultado() throws InterruptedException
{     // wait(1);
      return resultado; }
public synchronized int factorial(int lim) throws InterruptedException
{
      int aux=1;
      for(int i=1;i<=lim;++i)
           {
            aux=aux*i;
           }
      return aux;
}
}             //  class factorial


class DemoJoin
{     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 DemoJoin
F:\Java\jdk1.3\bin\java.exe   DemoJoin
Working Directory - F:\Java\Introducción a Java\Hilos 2002\
Class Path -
F:\java\jdk1.3\demo\applets\SerieAri;.;F:\Java\kawaclasses.zip;F:\Java\jdk1.3\lib
\tools.jar;F:\Java\jdk1.3\jre\lib\rt.jar;F:\Java\jdk1.3\jre\lib\i18n.jar
el factorial de 5 es 120
Process Exit...
```

Inquietudes para el alumno: Pruebe que ocurre comentarizando fact.join(). Luego "descomentarize" wait en el método getResultado(). Por que cree que este método y factorial fueron caracterizados synchronized ?
Las tres formas de join:

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

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

public final void **join()** throws InterruptedException

13.3 FINALIZACION DE LA EJECUCION DE APLICACIONES

Todas las aplicaciones comienzan ejecutando el hilo de. Si nuestra aplicación no crea otros hilos, finalizará cuando main retorne. Pero si en main creamos otros hilos, ¿qué sucede cuando

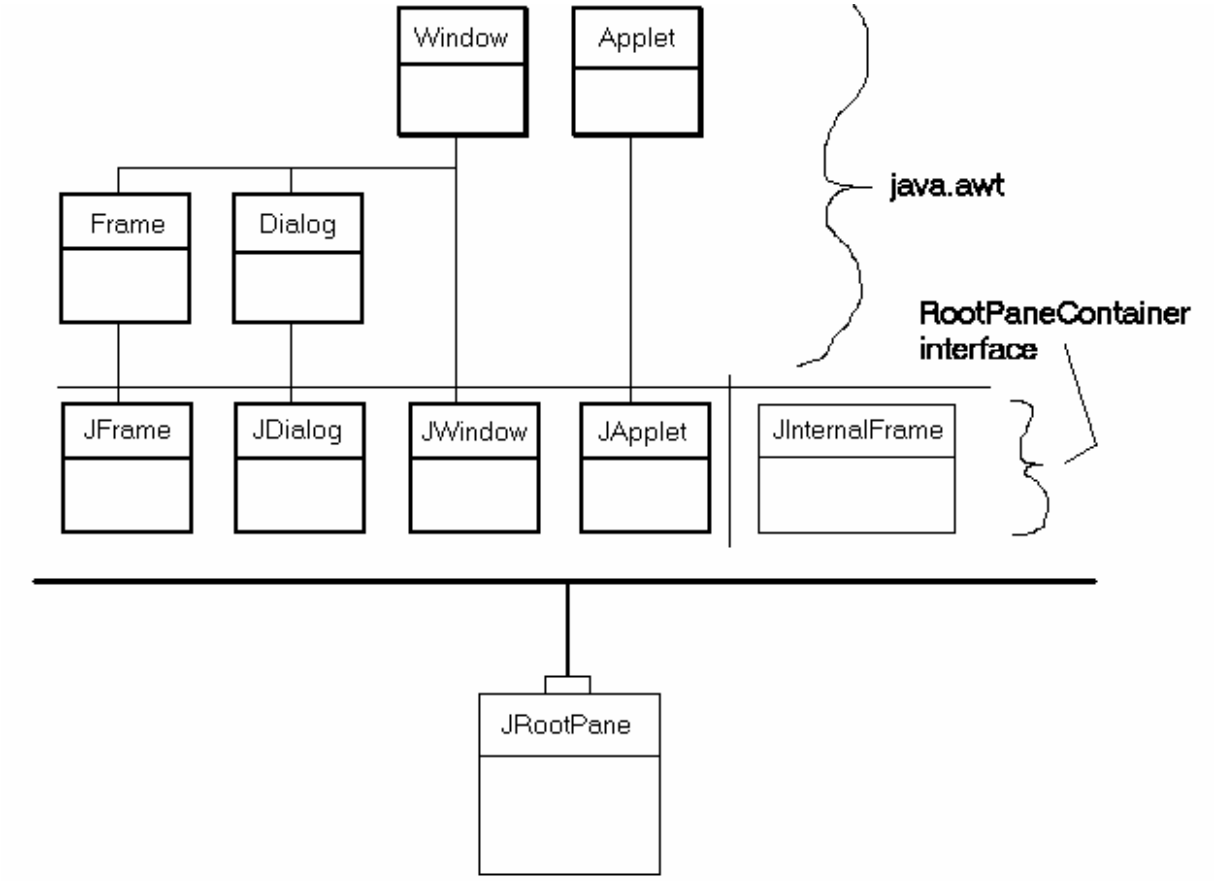
se retorna de main?

Hay dos clases de hilos: *usuario* y demonio (*daemon)*. La presencia de un hilo de usuario mantiene la aplicación ejecutándose, mientras que un hilo de *daemon* es prescindible. Cuando finaliza el último hilo de usuario, todos los hilos de *daemon* finalizan y la aplicación termina. Si nuestro método main produce un hilo, este hilo hereda la categoría de hilo de usuario que tenía el hilo original. Cuando main finalice, la aplicación continuará ejecutándose hasta que el otro hilo también finalice. Una aplicación se ejecutará **hasta que todos los hilos de usuario finalicen.**

14 - CASOS RESUELTOS DE APPLETS MULTIHILOS

14.1 - JApplet, JRootPane, contentPane – Como se arma el escenario...

La clase JRootPane es un contenedor liviano usado "detrás del escenario" ("behind the scenes") por sus contenedores JFrame, JDialog, JWindow, JApplet, y JinternalFrame. JApplet contiene a JRootPane como su único hijo (Child). Todos estos 5 contenedores implementan la interface **RootPaneContainer** y todos delegan sus operaciones en la clase JRootPane.



La clase JRootPane contiene un glassPane, un menuBar opcional y el contentPane, éstos dos últiimos gestionados por JLayeredLayout. El GlassPane se sitúa arriba de todo, donde está en posición de interceptar movimientos de ratón.

El contentPane definido dentro de estos contenedores debe ser el ancestro de cualquier hijo de JRootPane. En vez de incluir directamente los hijos en JRootPane con **rootPane.add(child)** se debe usar **rootPane.getContentPane.add(child).**

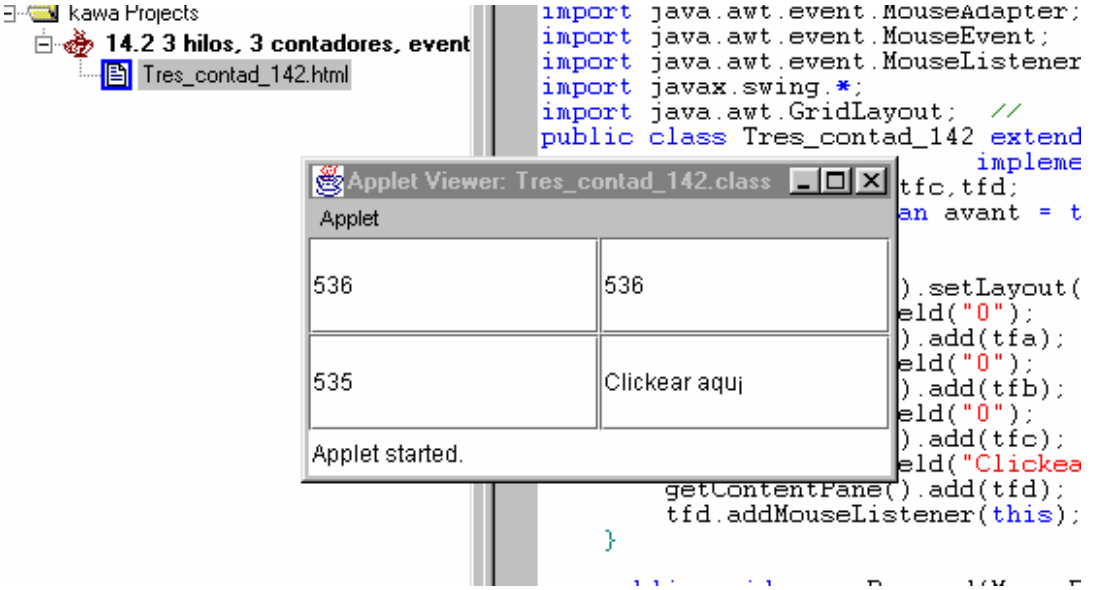
El layout default del manager de estos contenedores es BorderLayout. Como quiera que sea, JRootPane usa layout personalizado. Entonces, si Ud quiere cambiarlo para agregar componentes a JRootpane, use código como

RootPane.getContentPane.setLayout(newBorderlayout)

14.2 3 hilos, 3 contadores, eventos

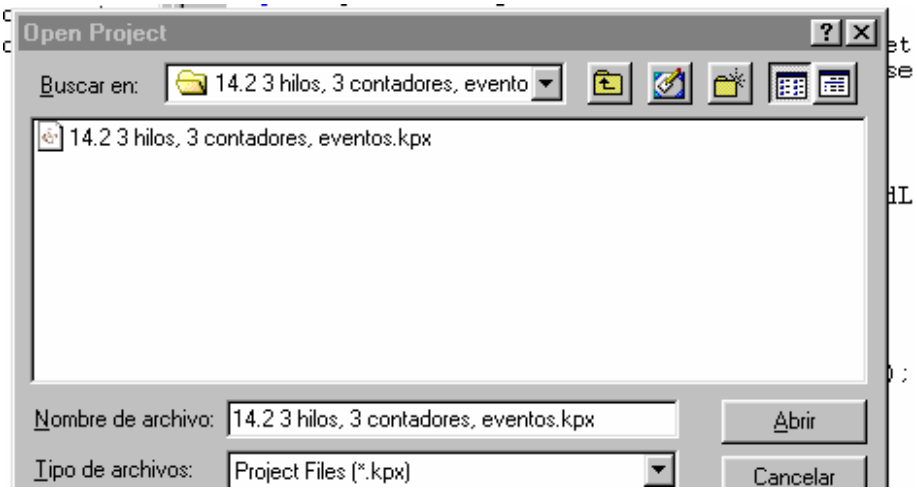
<HTML>
<h1>Tres contadores, tres hilos</h1>
<applet code="Tres_contad_142.class" width=300 height=100>
</APPLET>
<H2> Usando TextField </H2>
Area applet no sectorizada. Sin sincronización.
Eventos: class ClickHandler extends MouseAdapter
La escucha es a nivel applet. El evento es mousePressed.
Se la incorpora mediante: addMouseListener(new ClickHandler(this));
Los campos de texto se usan para exibir el valor de un contador.
El contador es incrementado/decrementado según : public static boolean avant = true/false;
avant es invertida cada clickeo del mouse.
</HTML>

Un capture de la ejecución, bajo Applet Viewer:

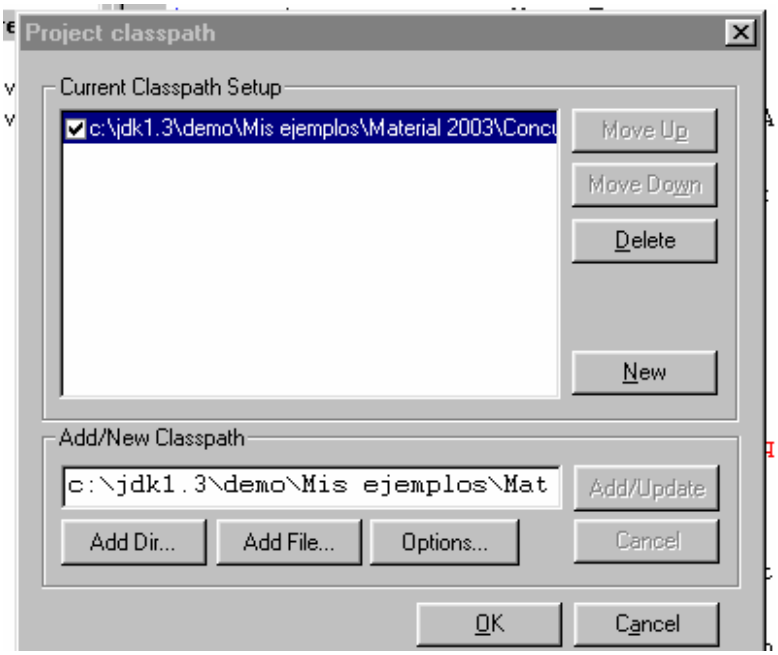


Nota: Para correr un JApplet desde el IDE (Entorno de desarrollo integrado), usando su applet viewer, previamente debemos:

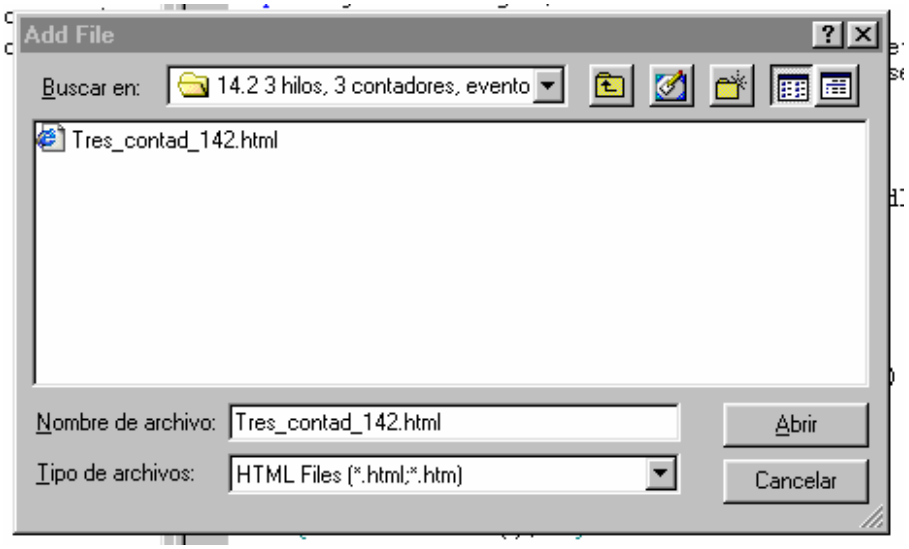
- Definir un proyecto: En el IDE Kawa, Proyect, Open, Nombre del archivo (Tipear o seleccionar el existente), abrir. La carpeta aparecerá subordinada a Kawa Proyects: **14.2 3 hilos, 3 contadores, eventos**.kpx).



- Setear adecuadamente su classpath: Carpeta donde se encuentra el .class correspondiente. Cursor sobre el proyecto, botón derecho, classpath. En el cuadro de diálogo abajo ejecute la acción que sea pertinente, <OK>.



- Incluir en el proyecto el archivo .html. En el cuadro de diálogo anterior, AddFile. En el nuevo cuadro, indicar tipo HTML y seleccionar, <Abrir>



- Caracterizar el .html como main(): Posicionado el cursor sobre el símbolo de .html, <botón derecho>, escoger main().

El resultado de toda esta manipulación debe ser algo como:

Y sin salir del IDE se lo prodrá ejecutar: build, run.

El código de este applet sigue:

```
//         Author Ing.Tymoschuk, Jorge
///////   Tres hilos, eventos comunes  ///////
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.*;
import java.awt.GridLayout;  //
public class Tres_contad_142 extends JApplet
                                                    implements MouseListener
{   JTextField tfa,tfb,tfc,tfd;
   public static boolean avant = true;

        public void init()
   {    getContentPane().setLayout(new GridLayout(2,2));
        tfa=new JTextField("0");
                getContentPane().add(tfa);
                tfb=new JTextField("0");
        getContentPane().add(tfb);
        tfc=new JTextField("0");
        getContentPane().add(tfc);
        tfd=new JTextField("Clickear aqu¡");
        getContentPane().add(tfd);
           tfd.addMouseListener(this);
   }

   public void mousePressed(MouseEvent e)
   {    invertir();  }

   public void mouseReleased(MouseEvent e) {
    }

    public void mouseEntered(MouseEvent e) {
    }

    public void mouseExited(MouseEvent e) {
    }

    public void mouseClicked(MouseEvent e) {
    }

        public void start()
        {       Thread A = new Thread (new Counter(tfa));
                A.start();
                Thread B = new Thread (new Counter(tfb));
                B.start();
                Thread C = new Thread (new Counter(tfc));
                C.start();
        }

        public void invertir()
   {   if(avant)
                        {       avant=false;
                                tfd.setText("Retrocedemos");
                        }
                else
                        {       avant=true;
                                tfd.setText("Avanzamos ..");
                        }
        }
}
```

```
class Counter extends Thread
{
        JTextField texto;
        String s;
        private int cont=0;
        public Counter(JTextField txtf)
        {       texto = txtf;
        }

        public void run() {
                try
                {   while(true)
                                {       if(Tres_contad_142.avant) cont++;
                                        else cont--;
                                        texto.setText(s.valueOf(cont));
                                        sleep(50);
                                        yield();
                                }       // while
                } catch (InterruptedException e){;}
        }       // Run
} // Counter
```

14.3 3 Hilos, 3 contadores, eventos

```
<HTML>
<h1>Tres contadores, tres hilos</h1>
<applet code="Tres_contad_143.class" width=300 height=100>
</APPLET>
<H2> Usando JTextField </H2>
Area applet no sectorizada. Sin sincronización.
Eventos: class ClickHandler extends MouseAdapter
La escucha es a nivel applet.
<p>
Se la incorpora mediante: addMouseListener(new ClickHandler(this));
El contador cont es variado dependiendo del valor de param: 0,1,-1;
param =  1, mouseEntered(MouseEvent e).
param =  0, mousePressed(MouseEvent e)
param = -1, mouseExited (MouseEvent e).
</HTML>
```

Un capture:

Entramos con el mouse, la reacción es: **Clickee, nos detendremos**... pero no podemos capturarlo, al salir para startar este nuevo capture el MouseListener detecta un nuevo evento ...

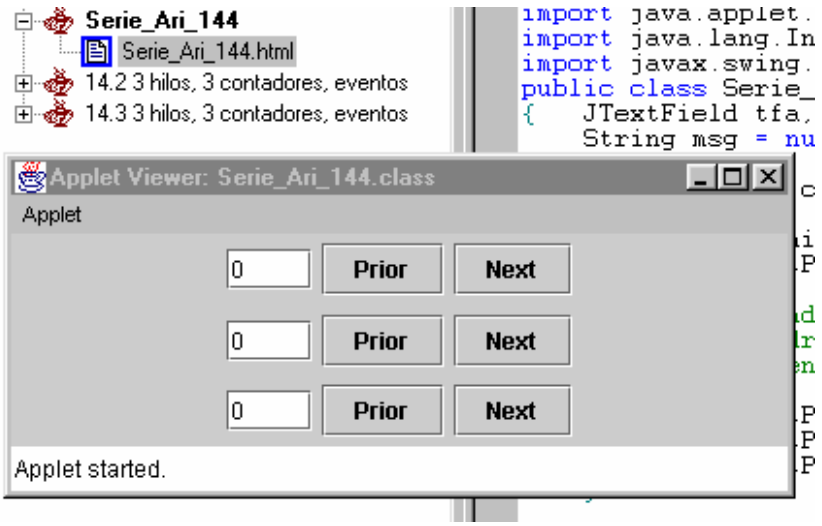
```
// Author Ing.Tymoschuk, Jorge
///////  Tres hilos, eventos comunes  ///////
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.*;
import java.awt.GridLayout;  //
public class Tres_contad_143 extends JApplet
                                                    implements MouseListener
{   JTextField tfa,tfb,tfc,tfd;
   public static int param = 0;

        public void init()
   {    getContentPane().setLayout(new GridLayout(2,2));
        tfa=new JTextField("0");
                getContentPane().add(tfa);
                tfb=new JTextField("0");
        getContentPane().add(tfb);
        tfc=new JTextField("0");
        getContentPane().add(tfc);
        tfd=new JTextField("Entre aquí, ascenderemos");
        getContentPane().add(tfd);
          tfd.addMouseListener(this);
   }

   public void mousePressed(MouseEvent e){
                param=0;
                tfd.setText("Salga, descenderemos");
        }

   public void mouseReleased(MouseEvent e) {
   }

   public void mouseEntered(MouseEvent e) {
                param=1;
                tfd.setText("Clikee, nos detendremos");
        }

    public void mouseExited(MouseEvent e) {
        param=-1;
        tfd.setText("Entre, ascenderemos");
    }

    public void mouseClicked(MouseEvent e) {
        param=0;
        tfd.setText("Salga, descenderemos");
    }

        public void start()
        {       Thread A = new Thread (new Counter(tfa));
                A.start();
                Thread B = new Thread (newCounter(tfb));
                B.start();
                Thread C = new Thread (new Counter(tfc));
                C.start();
        }
}
```

```
class Counter extends Thread
{
        JTextField texto;
        String s;
        private int cont=0;
        public Counter(JTextField txtf)
        {       texto = txtf;
        }

        public void run() {
                try
                {   while(true)
                                {       cont+=Tres_contad_143.param;
                                        texto.setText(s.valueOf(cont));
                                        sleep(50);
                                        yield();
                                }       // while
                } catch (InterruptedException e){;}
        }       // Run
}  //  Counter
```

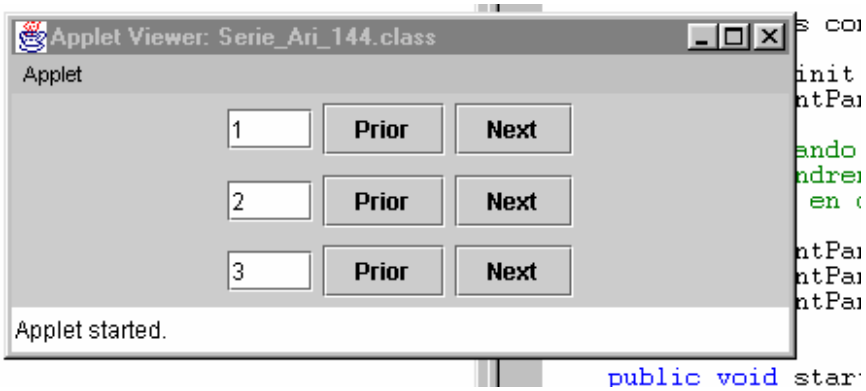
14.4 3 Hilos, Series Aritméticas

```
<HTML>
<H1>Serie Ari Multi Tread</H1>
<applet code="Serie_Ari_144.class" width=400 height=110>
</APPLET>
<H2>Escucha: ActionListener</H2>
Navegación de una serie aritmética.
Variación del ejemplo eventos 1.7 simplificando
un poco la entrada/salida de datos, pero procesando
en tres hilos simultaneamente.
Ahora usamos botones... para ello usamos las clases:
public class SerieAri_144 extends JApplet: provee la funcionalidad
del formato de java.awt.BorderLayout: cinco regiones: north, south,
east, west y center
class SerieControls extends Panel implements  Runnable, ActionListener.
En ella analizamos las acciones (actionPerformed()) y ejecutamos las
respuestas correspondientes.
Utilizanmos 3 hilos no sincronizados. Escucha de eventos a nivel hilo.
</HTML>
```

Un capture de la ejecución sigue:



Si clickeamos Next en cada uno de los botones ...



La codificación:

```
// Author Ing.Tymoschuk, Jorge
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.lang.Integer;
import javax.swing.*;
public class Serie_Ari_144 extends JApplet
{       JTextField tfa,tfb,tfc; //  Referencias a objetos TextField
        String msg = null;      //  mensaje

   SerieControls contr01,contr02,contr03;   //  Controles para avanzar,retroceder

   public void init()
   {    getContentPane().setLayout(new BorderLayout());

        //  Seteando un formato de acuerdo a las posibilidades de java.awt.BorderLayout.
        // Dispondremos de un contenedor que organiza y redimensiona sus componentes para
        // caber en cinco regiones: north, south, east, west y center

                getContentPane().add("North" ,contr01 = new SerieControls(tfa,1));
                getContentPane().add("Center",contr02 = new SerieControls(tfb,2));
                getContentPane().add("South" ,contr03 = new SerieControls(tfc,3));
   }

   public void start()
        {       contr01.setEnabled(true);
        contr02.setEnabled(true);
                contr03.setEnabled(true);
                Thread A = new Thread (new SerieControls(tfa,1));
                Thread B = new Thread (new SerieControls(tfb,2));
                Thread C = new Thread (new SerieControls(tfc,3));
                /*  Estamos generando tres objetos Thread: A, B, C.
                        Cada uno de estos objetos recibe un objeto SerieControls,
                        previamente inicializado por su respectivo constructor con
                        el texto y la razón de la serie pasados como argumento.
                        La clase SerieControls implementa runnable y ActionListener */
                A.start();
                B.start();
                C.start();
        }

   public void stop()
        {       contr01.setEnabled(true);
```

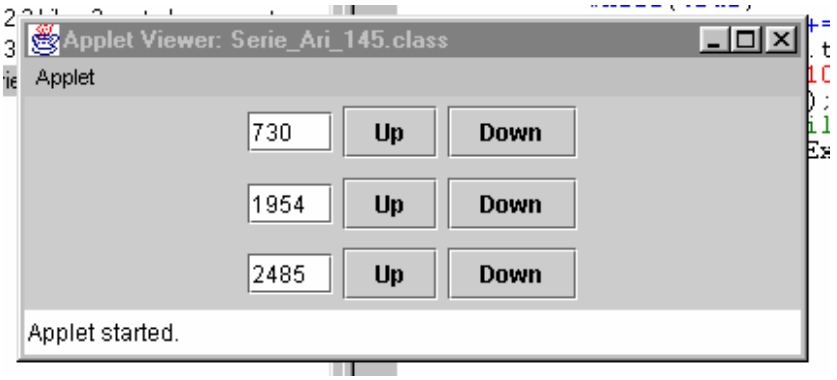
```
        contr02.setEnabled(true);
                contr03.setEnabled(true);
    }

    public void destroy()
        {
       remove(contr01);
       remove(contr02);
       remove(contr03);
    }

    public void processEvent(AWTEvent e)
        {       if (e.getID() == Event.WINDOW_DESTROY)
                        {System.exit(0);}
    }

    public void paint(Graphics g) // método invocado "implicitamente"
    {   g.drawRect(0, 0, getSize().width-1, // por start()
                getSize().height-1); //  Método de java.awt.Graphics
       g.drawString(msg, 5, 30);       // idem
    }

    public static void main(String args[])
        {       Frame f = new Frame("MultiThread Serie Test");
                Serie_Ari_144  serieAri = new Serie_Ari_144();

                serieAri.init();
                serieAri.start();

                f.add("Center", serieAri);
                f.setSize(400, 200);
                f.show();
    }

}               //  Serie_Ari_144

class SerieControls extends Panel
            implements Runnable, ActionListener
{       JTextField tfd;
        String s;
        int valAct=0;
    int razon;

        public SerieControls(JTextField txf, int raz)
        {       tfd  = txf;
                razon=raz;
                add(tfd = new JTextField("0", 4));
                JButton b = null;
                b = new JButton("Prior");
                b.addActionListener(this);
                add(b);
                b = new JButton("Next");
                b.addActionListener(this);
                add(b);
    }

        public void run()
        {       tfd.setText(s.valueOf(valAct));
        }

    public void actionPerformed(ActionEvent ev) {
        String label = ev.getActionCommand();
```

```
    if (label=="Prior")
                valAct-=razon;
        else
                valAct+=razon;
                tfd.setText(s.valueOf(valAct));
    }
}       //  SerieControls
```

14.5 - 3 hilos, 3 series aritméticas, eventos comandan ciclos

```
<HTML>
<H1>Serie Ari Multi Tread</H1>
<applet code="Serie_Ari_145.class" width=400 height=110>
</APPLET>
<H2>Escucha: ActionListener</H2>
Navegación de una serie aritmética.
Ahora usamos botones... para ello usamos las clases:
public class SerieAri_144 extends JApplet: provee la funcionalidad
del formato de java.awt.BorderLayout: cinco regiones: north, south,
east, west y center
class SerieControls extends Panel implements  Runnable, ActionListener.
En ella analizamos las acciones (actionPerformed()) y ejecutamos las
respuestas correspondientes.
Utilizanmos 3 hilos no sincronizados. Escucha de eventos a nivel hilo.
</HTML>
```

El correspondiente capture:



La codificación:

```
// Author Ing.Tymoschuk, Jorge
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.lang.Integer;
import javax.swing.*;
public class Serie_Ari_145  extends JApplet
{       JTextField tfa = new JTextField("Uno");
        JTextField tfb = new JTextField("Dos");
        JTextField tfc = new JTextField("Tre");

    SerieControl contr01,contr02,contr03;   //  Controles para avanzar,retroceder

    public void init()
    {       getContentPane().setLayout(new BorderLayout());

            //  Seteando un formato de acuerdo a las posibilidades de java.awt.BorderLayout.
```

```
        // Dispondremos de un contenedor que organiza y redimensiona sus componentes para
        // caber en cinco regiones: north, south, east, west y center

                getContentPane().add("North" ,contr01 = new SerieControl(tfa,1));
                getContentPane().add("Center",contr02 = new SerieControl(tfb,2));
                getContentPane().add("South" ,contr03 = new SerieControl(tfc,3));
    }

    public void start()
        {       contr01.setEnabled(true);
        contr01.setVisible(true);
        contr02.setEnabled(true);
                contr02.setVisible(true);
                contr03.setEnabled(true);
                contr03.setVisible(true);
                Thread A = new SerieAction(contr01);
                Thread B = new SerieAction(contr02);
                Thread C = new SerieAction(contr03);
                /*  Estamos generando tres objetos Thread: A, B, C.
                        Cada uno de estos objetos recibe un objeto SerieAction,
                        previamente inicializado por su constructor propio, (JTextField,
                        raz¢n y signo). */
                A.start();
                B.start();
                C.start();
        }

    public void stop()
        {       contr01.setEnabled(false);
        contr02.setEnabled(false);
                contr03.setEnabled(false);
    }

    public void destroy()
        {
       remove(contr01);
       remove(contr02);
       remove(contr03);
    }

    public void processEvent(AWTEvent e)
        {       if (e.getID() == Event.WINDOW_DESTROY)
                        {System.exit(0);}
    }

    public static void main(String args[])
        {       Serie_Ari_145  serieAri = new Serie_Ari_145();

                serieAri.init();
                serieAri.start();

    }

}               //  Serie_Ari_145

class SerieControl extends JPanel
                        implements ActionListener
{       JTextField tfd;
        int razon;
        int signo=1;

        public SerieControl(JTextField txf,int raz)
```

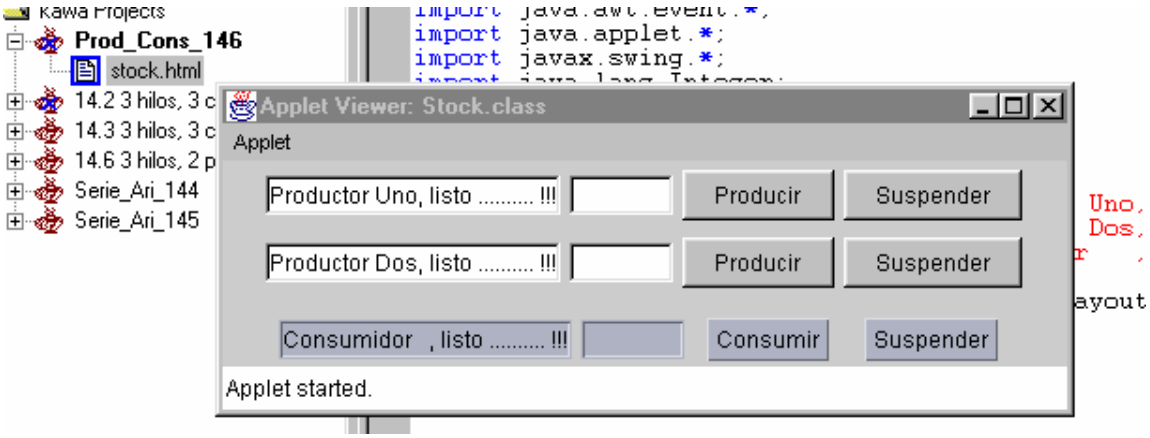
```
        {       tfd  = txf;
                razon=raz;
                add(tfd = new JTextField("0", 4));
                JButton b = null;
                b = new JButton("Up");
                b.addActionListener(this);
                add(b);
                b = new JButton("Down");
                b.addActionListener(this);
                add(b);
        }


   public void actionPerformed(ActionEvent ev)
   {
        String label = ev.getActionCommand();
        if (label=="Up")
                signo=1;
        else
                signo=-1;
   }
}       //  SerieControl

class SerieAction extends Thread
{       SerieControl serieC;
        int valAct=10;
        String s="string";
        public SerieAction(SerieControl sCont)
        {       serieC=sCont;
        }

        public void run()
        {
      Thread me = Thread.currentThread();
      me.setPriority(Thread.MIN_PRIORITY);

                try{
                        while(true)
                                {       valAct+=serieC.signo*serieC.razon;
                                        serieC.tfd.setText(s.valueOf(valAct));
                                        sleep(10);
                                        yield();
                                }       //  while
                }catch(InterruptedException e) {;}
        }       // run()
}
```

14.6 - 3 hilos, 2 productores, 1 consumidor, eventos, sincronización

```
<HTML>
<H1>Sincronización: Productor/Consumo/Monitor</H1>
<applet code="Stock.class" width=450 height=110>
</APPLET>
<H2>Escucha: ActionListener</H2>
<p>
La clase Stock, heredera de JApplet, incorpora setLayout(new gridLayout(3,1));
Esta declaración de layout define 3 filas con una columna cada una.
<p>
Las clases Producion y Consumo corren en hilos independientes, arrancados en Stock.
Producción corre en las dos primeras, Consumo en la ultima. En cada una de ellas
addActionListener(this) es la responsable de incorporar el escucha de acciones respectivo.
<p>
```

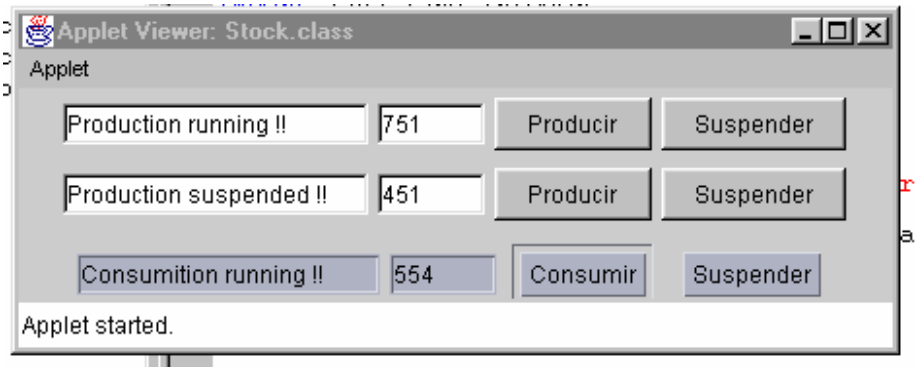
Las funciones de monitoreo estan definidas en dos clases adicionales, ControlsProd y ControlsCons, donde las acciones son detectadas por actionPerformed(ActionEvent ev) quien setea las variables de instancia producir y consumir, en las clases Produccion y Consumo,respectivamente.
<p>
El consumo está sincronizado con la producción. Para ello usamos una variable static stock definida en la clase Stock, actualizada por los métodos sincronizados sumStock() y hayStock() invocados por las clases Produccion y Consumo, respectivamente.
</HTML>

Un primer capture, situación inicial

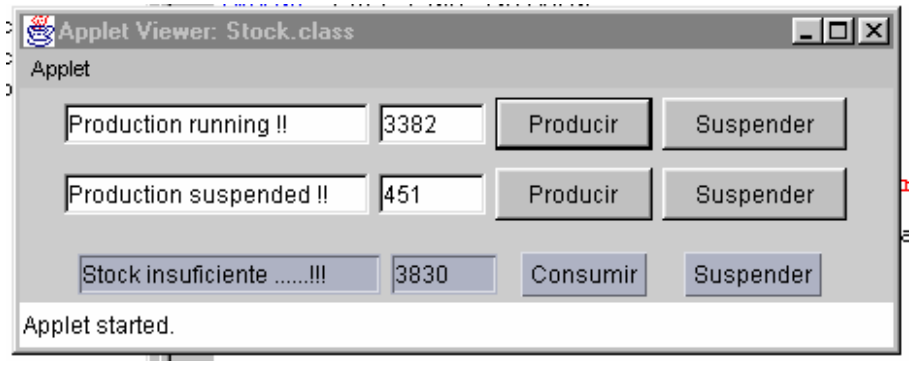


Ponemos los productores a trabajar

Suspendemos el segundo, arrancamos el consumo



El consumo es voraz, pronto llegamos a una situación de falta



La codificación :

```
// Author Ing.Tymoschuk, Jorge
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;
import java.lang.Integer;
import javax.swing.*;

public class Stock extends JApplet {
        static int stock=0;
        PanelProd cPro1=new PanelProd("Productor Uno, listo ..... !!!");
        PanelProd cPro2=new PanelProd("Productor Dos, listo ..... !!!");
        PanelCons cCons=new PanelCons("Consumidor   , listo ..... !!!");
   public void init() {
                getContentPane().setLayout(new GridLayout(3,1));
                getContentPane().add(cPro1);
                getContentPane().add(cPro2);
                getContentPane().add(cCons);
   }

   public void start() {
                cPro1.setEnabled(true);
                Thread A = new Thread (new Produccion(cPro1));
                A.start();
        cPro2.setEnabled(true);
        Thread B = new Thread (new Produccion(cPro2));
        B.start();
        cCons.setEnabled(true);
        Thread C = new Thread (new Consumo(cCons));
        C.start();
   }

   public void stop() {
                cPro1.setEnabled(false);
        cPro2.setEnabled(false);
        cCons.setEnabled(false);
   }

   public void destroy() {
      remove(cPro1);
        remove(cPro2);
        remove(cCons);
   }

   public void processEvent(AWTEvent e) {
      if (e.getID() == Event.WINDOW_DESTROY) {
```

```
            System.exit(0);
        }
    }

        static synchronized void sumStock()
        {       stock++;}

        static synchronized boolean hayStock()
        {       if(stock>2)
                {       stock-=2;
                        return true;
                }
                else return false;
        }
}

class Produccion extends Thread {
        PanelProd pProd;
        int produc=0;
        String s;
        public Produccion(PanelProd pprod){
                pProd=pprod;
        }
        public void run()
        {
                Thread me = Thread.currentThread();
                me.setPriority(Thread.MIN_PRIORITY);
                while (produc<10000)
                        {       if(pProd.getStatus()==1)
                                {
                                        pProd.prd.setText(s.valueOf(++produc));
                                        Stock.sumStock();
                                }
                                try
                                {
                                        sleep(50); //  Dormimos la ejecución 50 miliseg
                                        yield();
                                }catch( InterruptedException e ) {;}
                        }
        }
}

class PanelProd extends Panel
             implements ActionListener {
        JTextField msg,prd;
        String s;
        int status=0; // status: 0, listo;1, produciendo;2, suspendido

        public PanelProd(String hilera){
                try { // Los productores usan camiseta Windows
                        javax.swing.UIManager.setLookAndFeel(
                        "com.sun.ja va.swing.plaf.windows.WindowsLookAndFeel");
                } catch (Exception  e) {;}
                msg  = new JTextField(hilera);
                add(msg);
                prd = new JTextField(5);
                add(prd);
                JButton b = null;
                b = new JButton("Producir");
                b.addActionListener(this);
                add(b);
                b = new JButton("Suspender");
```

```
                b.addActionListener(this);
                add(b);
    }

    public void actionPerformed(ActionEvent ev) {
                String label = ev.getActionCommand();
                if (label=="Producir"){
                        status=1;
                        msg.setText(s.valueOf("Production running !!"));
                }
                else {
                        status=2;
                        msg.setText(s.valueOf("Production suspended !!"));
                }
    }
        int getStatus(){return status;}
}

class Consumo extends Thread {
        PanelCons pCons;
        String s;
        int consum=0;
        public Consumo(PanelCons pcons){
                pCons=pcons;
        }
        public void run()
        {
                Thread me = Thread.currentThread();
                me.setPriority(Thread.MIN_PRIORITY);

                while (consum<10000){
                        if(pCons.status==1)
                                if(Stock.hayStock())
                                        pCons.con.setText(s.valueOf(consum+=2));
                                else
                                        pCons.msg.setText("Stock insuficiente ......!!!");
                        try
                        {
                                sleep(50); //  Dormimos la ejecución 50 miliseg
                                yield();
                        }catch( InterruptedException e ) {;}
                }
        }
}

class PanelCons extends Panel
             implements ActionListener {
        JTextField msg,con;
        String s;
        int status=0; // status: 0, listo;1, produciendo;2, suspendido

        public PanelCons(String hilera) {
                try { // Los Consumidores prefieren camiseta Solaris ...
                        javax.swing.UIManager.setLookAndFeel(
                        "com.sun.ja va.swing.plaf.motif.MotifLookAndFeel");
                } catch (Exception  e) {;}
                msg  = new JTextField(hilera);
                add(msg);
                con = new JTextField(5);
                add(con);
                JButton b = null;
                b = new JButton("Consumir");
```

```
            b.addActionListener(this);
            add(b);
            b = new JButton("Suspender");
            b.addActionListener(this);
            add(b);
    }

    public void actionPerformed(ActionEvent ev) {
            String label = ev.getActionCommand();
            if (label=="Consumir"){
                    status=1;
                    msg.setText(s.valueOf("Consumition running !!"));
            }
            else {
                    status=2;
                    msg.setText(s.valueOf("Consumition suspended !!"));
            }
    }
}

// Vamos a preparar el almuerzo
```

15 - DEMO DE ORDENAMIENTO (Verlo en www.labsys.frc.utn.edu.ar)

16 - MAS EJERCICIOS RESUELTOS

16.1 REUNION DE AMIGOS

El siguiente ejemplo (Ejemplo19.java) usa threads para activar simultáneamente tres objetos de la misma clase, que comparten los recursos del procesador peleándose para escribir a la pantalla.

```
class Ejemplo19 {

        public static void main(String argv[]) throws InterruptedException {
                Thread Juan = new Thread (new Amigo("Juan"));
                Thread Luis = new Thread (new Amigo("Luis"));
                Thread Nora = new Thread (new Amigo("Nora"));
                Juan.start();
                Luis.start();
                Nora.start();
                Juan.join();
                Luis.join();
                Nora.join();
        }
}

class Amigo implements Runnable {
        String mensaje;
        public Amigo(String nombre)  {
                mensaje = "Hola, soy "+nombre+" y este es mi mensaje ";
        }

        public void run() {
                for (int i=1; i<6; i++) {
                        String msg = mensaje+i;
                        System.out.println(msg);
                }
        }
}
```

Compilarlo con *javac Ejemplo19.java* y ejecutarlo con *java Ejemplo19*.
la salida será más o menos así:

```
Hola, soy Juan y este es mi mensaje 1
Hola, soy Juan y este es mi mensaje 2
Hola, soy Luis y este es mi mensaje 1
Hola, soy Luis y este es mi mensaje 2
Hola, soy Nora y este es mi mensaje 1
```

```
Hola, soy Nora y este es mi mensaje 2
Hola, soy Nora y este es mi mensaje 3
Hola, soy Juan y este es mi mensaje 3
...........etc.
```

16.2 LA LIEBRE, LA TORTUGA Y EL GUEPARDO

Este programa simplemente crea tres animales (clase Animal), asigna un thread a cada uno y los ejecuta.

```
public class Ejemplo21 {
	static Animal tortuga;
	static Animal liebre;
	static Animal guepardo;

	public static void main(String argv[]) throws InterruptedException {
		tortuga = new Animal(2, "T");
		liebre = new Animal(3, "L");
		guepardo = new Animal(4, "G");
		tortuga.start();
		liebre.start();
		guepardo.start();
		tortuga.join();
		liebre.join();
		guepardo.join();
	}
}

class Animal extends Thread {
	String nombre;

	public Animal(int prioridad, String nombre) {
		this.nombre = nombre;
		setPriority(prioridad);
	}

	public void run() {
		for (int x = 0; x < 30; x++) {
			System.out.print( nombre );
			yield();
		}
		System.out.println("\nLlega "+nombre );
	}
}
```

La salida de este programa, ejecutado con *java Ejemplo2*1, es por ejemplo:

```
C:\java\curso>java Ejemplo21
GGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
Llega G
LTLLLLLLLLLLLLLLLLLLLLLLLLLL
Llega L
TTTTTTTTTTTTTTTTTTTTTTTTTT
Llega T
```

Como se ve, a pesar de haber arrancado antes la tortuga, casi todo el tiempo de CPU lo usa primero el Guepardo, luego la Liebre (aunque algo queda para la pobre tortuga, como se ve en la T marcada), y finalmente para la Tortuga. No todas las corridas ni todos los sistemas dan igual salida, ya que ésta depende de la carga del procesador y de la implementación de Java particular.

16.3 EL SUMADOR

```
// Archivo Ejemplo22.java, compilar con javac Ejemplo22.java, ejecutar
//con java Ejemplo22

public class Ejemplo22 {
	public static void main(String argv[]) {
		Sumador A = new Sumador(); // un único sumador
		Contador C1 = new Contador(A); // dos threads que lo usan...
		Contador C2 = new Contador(A); // ...para sumar
```

```
                C1.start();
                C2.start();
                try {
                        C1.join();
                        C2.join();
                } catch (Exception e) {
                        System.out.println(e);
                }
        }
}

class Contador extends Thread {
        Sumador s;

        Contador (Sumador sumador) {
                s = sumador; // le asigno un sumador a usar
        }

        public void run() {
                s.sumar(); // ejecuto la suma
        }
}

class Sumador {
        int a = 0;

        public void sumar() {
                for (int i=0; i<10000; i++ )
                {
                        if ( (i % 5000) == 0 )
                        {
                                // "%" da el resto de la división:
                                System.out.println(a); // imprimo cada 5000
                        }
                        a += 1;
                }
                System.out.println(a); // imprimo el final
        }
}
```

Ejecutando esto nos da más o menos así (cada corrida es diferente, dependiendo de cómo se "chocan" los threads y la carga de la CPU):

```
C:\java\curso>java Ejemplo22
0
87
8926
10434
14159
17855
```

16.4. CONTROLA LA LLEGADA DEL PERSONAL

```
public class Ejemplo23 {
        public static void main(String argv[]) {
                Saludo hola = new Saludo();
                Personal jefe = new Personal(hola, "JEFE", 3);
                Personal pablo = new Personal(hola, "Pablo");
                Personal luis = new Personal(hola, "Luis");
                Personal andrea = new Personal(hola, "Andrea");
                jefe.start();
                pablo.start();
                luis.start();
                andrea.start();
                try {
                        pablo.join();
                        luis.join();
```

```
                        andrea.join();
                        jefe.join();
                } catch (Exception e) {
                        System.out.println(e);
                }
        }
}

class Saludo {
        synchronized void esperarJefe(String empleado) {
                try {
                        wait();
                        System.out.println(empleado+"> Buenos dias jefe!");
                } catch (InterruptedException e) {
                        System.out.println(e.toString());
                }
        }

        synchronized void saludoJefe() {
                System.out.println("JEFE> Buenos dias!");
                notifyAll();
        }
}

class Personal extends Thread {
        String nombre;
        Saludo saludo;
        boolean esJefe;
        static int llegaron = 0;
        int numEmp;

        Personal (Saludo s, String n) {
                esJefe = false;
                nombre = n;
                saludo = s;
        }

        Personal (Saludo s, String n, int x) {
                esJefe = true;
                nombre = n;
                saludo = s;
                numEmp = x;
        }

        public void run() {
                System.out.println("("+nombre+" llega)");
                if (esJefe)
                {
                        while (llegaron < numEmp) {
                                System.out.println("(Esperando...)"); }
                                saludo.saludoJefe();
                        }
                else
                {
                        synchronized(this)
                        {
                                llegaron++;
                        }
                        saludo.esperarJefe(nombre);
                }
        }
}
```

Se usó notifyAll() en lugar de notify(), porque en el segundo caso sólo se notificaría al primer thread (el primer empleado en llegar) y no a los demás, que se

quedarían en el wait(). Como se ve en la salida, a pesar de que los empleados están en condiciones de saludar, no lo hacen hasta que no llega el jefe:

```
C:\java\curso>java Ejemplo23
(Pablo llega)
(Luis llega)
(Andrea llega)
(Pedro llega)
(JEFE llega)
JEFE> Buenos dias!
Luis> Buenos dias jefe!
Pedro> Buenos dias jefe!
Andrea> Buenos dias jefe!
Pablo> Buenos dias jefe!
```

16.5 SINCRONIZACION CON UN SEMAFORO

Archivo Data.java

```
/*** Clase que representa la memoria compartida entre el Productor y
 * el Consumidor para la transferencia de datos. */
class Data {
	//Valor que es creado por el Productor y utilizado por el Consumidor
	private int value = 0;

	//Constructor que define el valor inicial del dato
	public Data(int v) { value = v; }

	//Método que retorna el valor del dato
	public int getValue() { return value; }

	//Método que permite asignarle un valor al dato
	public void setValue(int v) { value = v; }
}
```

Archivo Semaforo.java

```
/*** Clase semáforo que posee un atributo con 'n' valores posibles que se
alternan secuencialmente, dependiendo de la cantidad de procesos que se
deseen sincronizar.  */
class Semaforo {
	//Atributo que guarda el valor del semáforo
	private int value = 0;

	//Atributo que define la cantidad de valores que puede asumir el
	//semáforo (uno por cada proceso sincronizado)
	private int maxValue;

	//Constructor que define la cantidad de procesos que deberá
	//sincronizar el semáforo
	public Semaforo(int n) {
		if(n > 1)    maxValue = n;
		else maxValue = 2;
	}

	//Retorna el estado (valor) del semáforo
	public int getValue() { return value; }

	//Setea el siguiente valor del semáforo
	synchronized public void nextValue() {
		// Incremeta el valor del semáforo
		value++;
		// Si está por encima del límite, se vuelve a 0 (cero)
		if(value >= maxValue) value = 0;
	}
}
```

Archivo Productor.java

```
/*** Clase que representa al Productor de datos que serán utilizados por el
Consumidor.  Este es un productor de números enteros en secuencia. */
class Productor extends Thread {
	//Referencia al objeto de tipo Data que se utiliza para comunicar al
	//Productor con el Consumidor
	private Data dato;

	//Referencia al semáforo que se utiliza para sincronizar al
Consumidor y
	//al Productor
	private Semaforo sem;

	//Atributo que guarda el valor que debe tener el semáforo para poder
	//acceder al dato desde el Consumidor
	private int semVal;

	//Constructor que setea el objeto Data y el Semáforo con los que
	//trabajará el Productor, así como el valor que deberá tener el
semáforo
	//para trabajar
	public Productor(Data d, Semaforo s, int v) {
		dato = d;
		sem = s;
		semVal = v;
	}

	//Método que genera el siguiente valor del dato compartido
	public void generar() {
		// Se espera mientras el semáforo no tenga el valor debido
		while(sem.getValue() != semVal);
		// Se toma el monitor del objeto Data
		synchronized(dato) {
			// Se obtiene el valor del dato
			int val = dato.getValue();
			// Se define el nuevo valor del dato
			val++;
			dato.setValue(val);
		}
		// Se cambia el valor del semáforo
		sem.nextValue();
	}

	//Método que se dispara al iniciarse el thread (hilo de control) del
	//Productor
	public void run() {
		// Se generan 10 números y se finaliza
		for(int i = 0; i < 10; i++)
			generar();
	}
}
```

Archivo Consumidor.java

```
/*** Clase que representa al Consumidor de datos que serán generados por el
Productor. Este consumidor sólo muestra por pantalla los valores obtenidos.
*/
class Consumidor extends Thread {
	//Referencia al objeto de tipo Data que se utiliza para comunicar al
	//Consumidor con el Productor
	private Data dato;

	//Referencia al semáforo que se utiliza para sincronizar al
Consumidor y
	//al Productor
	private Semaforo sem;

	//Atributo que guarda el valor que debe tener el semáforo para poder
```

```
        //acceder al dato desde el Consumidor
        private int semVal;

        //Constructor que setea el objeto Data y el Semáforo con los que
        //trabajará el Consumidor, así como el valor que deberá tener el
semáforo
        //para trabajar
        public Consumidor(Data d, Semaforo s, int v) {
                dato = d;
                sem = s;
                semVal = v;
        }

        //Método que obtiene el siguiente valor del dato compartido
        public void consumir() {
                // Se espera mientras el semáforo no tenga el valor debido
                while(sem.getValue() != semVal);
                // Se toma el monitor del objeto Data
                synchronized(dato) {
                        // Se obtiene el valor del dato
                        int val = dato.getValue();
                        // Se muestra el valor del dato obtenido
                        System.out.println("Valor: " + val);
                }
                // Se cambia el valor del semáforo
                sem.nextValue();
        }

        //Método que se dispara al iniciarse el thread (hilo de control) del
        //Consumidor
        public void run() {
                // Se leen 10 números y se finaliza
                for(int i = 0; i < 10; i++)
                        consumir();
        }
}
```

Archivo ProgPpal.java

```
/*** Clase que representa el Programa Principal */
class ProgPpal {
        // Método donde inicia la ejecución del programa
        public static void main(String[] args) {
                // Variables que manejan las referencias a los objetos
                Data d;
                Semaforo s;
                Productor p;
                Consumidor c;
                // Se crean los objetos del sistema Productor-Consumidor
                d = new Data(0);
                s = new Semaforo(2);
                p = new Productor(d, s, 0);
                c = new Consumidor(d, s, 1);
                // Se dispara el Productor
                p.start();
                // Se dispara el Consumidor
                c.start();
        }
}
```

Salida con sincronización

Valor: 1	*Valor: 6*
Valor: 2	*Valor: 7*
Valor: 3	*Valor: 8*
Valor: 4	*Valor: 9*
Valor: 5	*Valor: 10*

16.6 -TESTEA EVENTOS

```
class Even extends Thread
{
 public Even (int first,int interval)
 {
  io=first;
  delay=interval;
 }
 public void run()
 {
  try
   {
    for(int i=io;i<=100;i+=2)
     {
      System.out.println(i);
      sleep(delay);
     }
   }catch(InterruptedException e)
   {return;}
 }
 private int io, delay;
}

class Test
{
 public static void main (String[] arguments) throws InterruptedException
 {
  Even thread1= new Even(1,20);
  Even thread2= new Even(0,30);
  thread1.start();
  thread2.start();
  thread1.join();
  thread2.join();
  System.out.println("Proceso terminado");
 }
}
```