14: Hilos múltiples

Los objetos proporcionan una forma de dividir un programa en secciones independientes. A menudo, también es necesario convertir un programa en subtareas separadas que se ejecuten independientemente.

Cada una de estas subtareas independientes recibe el nombre de *hilo*, y uno programa como si cada hilo se ejecutara por sí mismo y tuviera la UCP para él sólo. Algún mecanismo subyacente divide de hecho el tiempo de UCP entre ellos, pero generalmente el programador no tiene por qué pensar en ello, lo que hace de la programación de hilos múltiples una tarea mucho más sencilla.

Un *proceso* es un programa en ejecución autocontenido con su propio espacio de direcciones. Un sistema operativo *multitarea* es capaz de ejecutar más de un proceso (programa) a la vez, mientras hace que parezca como si cada uno fuera el único que se está ejecutando, proporcionándole ciclos de UCP periódicamente. Por consiguiente, un único proceso puede tener múltiples hilos ejecutándose concurrentemente.

Hay muchos usos posibles del multihilo, pero, en general, se tendrá parte del programa vinculado a un evento o recurso particular, no deseando que el resto del programa pueda verse afectado por esta vinculación. Por tanto, se crea un hilo asociado a ese evento o tarea y se deja que se ejecute independientemente del programa principal. Un buen ejemplo es un botón de "salir" —no hay por qué verse obligado a probar el botón de salir en todos los fragmentos de código que se escriban en el programa, aunque sí se desea que el botón de salir responda, como si se *estuviera comprobando* regularmente. De hecho, una de las razones más importantes para la existencia del multihilo es la existencia de interfaces de usuario que respondan rápidamente.

Interfaces de respuesta de usuario rápida

Como punto de partida, puede considerarse un programa que lleva a cabo alguna operación intensa de UCP y que acaba ignorando la entrada de usuario y por tanto no emite respuestas. Éste, un applet/aplicación, simplemente mostrará el resultado de un contador en ejecución:

```
//: c14:Contador1.java
// Una interfaz de usuario que no responde.
// <applet code=Contador1 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Contador1 extends JApplet {
```

```
  private int conteo = 0;
  private JButton
    empezar = new JButton("Empezar"),
    onOff = new JButton("Conmutar");
  private JTextField t = new JTextField(10);
  private boolean flagEjecutar = true;
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    empezar.addActionListener(new EmpezarL());
    cp.add(empezar);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public void comenzar() {
    while (true) {
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
      if (FlagEjecutar)
        t.setText(Integer.toString(count++));
    }
  }
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      comenzar();
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      flagEjecutar = !flagEjecutar;
    }
  }
  public static void main(String[] args) {
    Console.run(new Contador1(), 300, 100);
  }
} ///:~
```

En este punto, el código del *applet* y Swing deberían ser racionalmente familiares, al estar explicados en el Capítulo 13. En el método **comenzar( )** es donde permanece ocupado el programa: pone el valor actual de **conteo** en el **JTextField t**, y después incrementa **conteo**.

Parte del bucle infinito interno a **comenzar( )** llama a **sleep( )**. Éste debe estar asociado con un objeto **Thread**, y resulta que toda aplicación tiene *algún* hilo asociado a él. (De hecho, Java se basa en hilos y siempre hay alguna ejecución junto con la aplicación.) Por tanto, independientemente de si se usan o no hilos de forma explícita, se puede producir el hilo actual que usa el programa con **Hilos** y el método **static sleep( )**.

Nótese que **sleep( )** puede lanzar una **InterruptedException**, aunque lanzar esta excepción se considera una forma hostil de romper un hilo, por lo que no se recomienda. (Una vez más, las excepciones son para condiciones excepcionales, no el flujo normal de control.) La capacidad de interrumpir a un hilo durmiente se ha incluido para soportar una faceta futura del lenguaje.

Cuando se presiona el botón **Empezar**, se invoca a **comenzar( )**. Al examinar **comenzar( )**, se podría pensar estúpidamente (como hicimos) que debería permitir el multihilo porque se va a dormir. Es decir, mientras que el método está dormido, parece como si la UCP pudiera estar ocupada monitorizando otras presiones sobre el botón. Pero resulta que el problema real es que **comenzar( )** nunca devuelve nada, puesto que está en un bucle infinito, y esto significa que **actionPerformed( )** nunca devuelve nada. Puesto que uno está enclavado en **actionPerformed( )** debido a la primera vez que se presionó el botón, el programa no puede gestionar ningún otro evento. (Para salir, de alguna forma hay que matar el proceso; la forma más sencilla de hacerlo es presionar Control-C en la ventana de la consola, si es que se lanzó desde la consola. Si se empieza vía el navegador, hay que matar la ventana del navegador.)

El problema básico aquí es que **comenzar( )** necesita continuar llevando a cabo sus operaciones, y al mismo tiempo necesita devolver algo, de forma que **actionPerformed( )** pueda completar su operación y la interfaz de usuario continúe respondiendo al usuario. Pero en un método convencional como **comenzar( )** no puede continuar *y* al mismo tiempo devolver el control al resto del programa. Esto suena a imposible de lograr, como si la UCP debiera estar en dos lugares a la vez, pero ésta es precisamente la ilusión que proporciona el multihilo.

El modelo de hilos (y su soporte de programación en Java) es una conveniencia de programación para simplificar estos juegos malabares y operaciones que se dan simultáneamente en un único programa. Con los hilos, la UCP puede ir rotando y dar a cada hilo parte de su tiempo. Cada hilo tiene impresión de tener la UCP para sí mismo durante todo el tiempo. La excepción se da siempre que el programa se ejecute en múltiples UCP. Pero uno de los aspectos más importantes de los hilos es que uno se abstrae de esta capa, de forma que el código no tiene por qué saber si se está ejecutando en una o en varias UCP. Por consiguiente, los hilos son una forma de crear programas transparentemente escalables.

Los hilos pueden reducir algo la eficiencia de computación, pero la mejora en el diseño de programa, el balanceo de recursos y la conveniencia del usuario suelen ser muy valiosos. Por supuesto, si se tiene más de una UCP, el sistema operativo puede dedicar cada UCP a un conjunto de hilos o incluso a un único hilo, logrando que el programa, en su globalidad, se ejecute mucho más rápido. La multitarea y el multihilo tienden a ser las formas más razonables de usar sistemas multiprocesador.

Heredar de **Thread**

La forma más simple de crear un hilo es heredar de la clase **Thread**, que tiene todo lo necesario para crear y ejecutar hilos. El método más importante de **Thread** es **run( )**, que debe ser sobreescrito para hacer que el hilo haga lo que se le mande. Por consiguiente, **run( )** es el código que se ejecutará "simultáneamente" con los otros hilos del programa.

El ejemplo siguiente crea cualquier número de hilos de los que realiza un seguimiento asignando a cada uno con un único número, generado con una variable **static**. El método **run( )** de **Thread** se sobreescribe para que disminuya cada vez que pase por el bucle y acabe cuando valga cero (en el momento en que acabe **run( )**, se termina el hilo).

```
//: c14:HiloSimple.java
// Ejemplo muy simple de hilos.

public class HiloSimple extends Thread {
  private int cuentaAtras = 5;
  private static int conteoHilos = 0;
  private int numeroHilo = ++conteoHilos;
  public HiloSimple() {
    System.out.println("Creando " + numeroHilo);
  }
  public void run() {
    while(true) {
      System.out.println("Hilo " +
        numeroHilo + "(" + cuentaAtras + ")");
      if(--cuentaAtras == 0) return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new HiloSimple().start();
    System.out.println("Todos los hilos Arrancados");
  }
} ///:~
```

Un método **run( )** suele tener siempre algún tipo de bucle que continúa hasta que el hilo deja de ser necesario, por lo que hay que establecer la condición en la que romper el bucle y salir (o, en el caso de arriba, simplemente **return** de **run( )**). A menudo, se convierte **run( )** en forma de bucle infinito, lo que significa que a falta de algún factor externo que haga que **run( )** termine, continuará para siempre.

En el método **main( )** se puede ver el número de hilos que se están creando y ejecutando. El método **start( )** de la clase **Thread** lleva a cabo alguna inicialización especial para el hilo y después llama a **run( )**. Por tanto, los pasos son: se llama al constructor para que contruya el objeto, después

start( ) configura el hilo y llama a **run( )**. Si no se llama a **start( )** (lo que puede hacerse en el constructor si es apropiado) nunca se dará comienzo al hilo.

La salida de una ejecución de este programa (que será distinta cada vez) es:

```
Creando 1
Creando 2
Creando 3
Creando 4
Creando 5
Hilo 1(5)
Hilo 1(4)
Hilo 1(3)
Hilo 1(2)
Hilo 2(5)
Hilo 2(4)
Hilo 2(3)
Hilo 2(2)
Hilo 2(1)
Hilo 1(1)
Todos los hilos Arrancados
Hilo 3(5)
Hilo 4(5)
Hilo 4(4)
Hilo 4(3)
Hilo 4(2)
Hilo 4(1)
Hilo 5(5)
Hilo 5(4)
Hilo 5(3)
Hilo 5(2)
Hilo 5(1)
Hilo 3(4)
Hilo 3(3)
Hilo 3(2)
Hilo 3(1)
```

Se verá que en este ejemplo no se llama nunca a **sleep( )**, y la salida sigue indicando que cada hilo obtiene una porción del tiempo de UCP en el que ejecutarse. Esto muestra que **sleep( )**, aunque descansa en la existencia de un hilo para poder ejecutarse, no está involucrado en la habilitación o deshabilitación de hilos. Es simplemente otro método.

También puede verse que los hilos no se ejecutan en el orden en el que se crean. De hecho, el orden en que la UCP atiende a un conjunto de hilos existente es indeterminado, a menos que se cambien las prioridades haciendo uso del método **setPriority( )** de **Thread**.

Cuando **main( )** crea los objetos **Thread** no captura las referencias a ninguno de ellos. Un objeto ordinario debería ser un juego justo para la recolección de basura, pero no un **Thread**. Cada **Thread** "se registra" a sí mismo de forma que haya una referencia al mismo en algún lugar y el recolector de basura no pueda limpiarlo.

Hilos para una interfaz con respuesta rápida

Ahora es posible solucionar el problema de **Contador1.java** con un hilo. El truco es colocar la subtarea —es decir, el bucle de dentro de **comenzar( )**— dentro del método **run( )** de un hilo. Cuando el usuario presione el botón **empezar**, se arranca el hilo, pero después se completa la *creación* del hilo, por lo que aunque se esté ejecutando el hilo, puede continuar el trabajo principal del programa (estando pendiente y respondiendo a eventos de la interfaz de usuario). He aquí la solución:

```
//: c14:Contador2.java
// Una interfaz de usuario de respuesta rápida con hilos.
// <applet code=Contador2 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Contador2 extends JApplet {
  private class SubtareaSeparada extends Thread {
    private int conteo = 0;
    private boolean flagEjecutar = true;
    SubtareaSeparada() { start(); }
    void invertirflag() { flagEjecutar = !flagEjecutar; }
    public void run() {
      while (true) {
       try {
         sleep(100);
       } catch(InterruptedException e) {
         System.err.println("Interrumpido");
       }
       if(flagEjecutar)
         t.setText(Integer.toString(conteo++));
       }
    }
  }
  private SubtareaSeparada sp = null;
  private JTextField t = new JTextField(10);
  private JButton
    empezar = new JButton("Empezar"),
    onOff = new JButton("Conmutar");
```

```
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(sp == null)
        sp = new SubtareaSeparada();
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(sp != null)
        sp.invertirFlag();
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    empezar.addActionListener(new StartL());
    cp.add(empezar);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public static void main(String[] args) {
    Console.run(new Contador2 (), 300, 100);
  }
} ///:~
```

Contador2 es un programa directo, cuya única tarea es establecer y mantener la interfaz de usuario. Pero ahora, cuando el usuario presiona el botón **empezar**, el código de gestión de eventos no llama a un método. En su lugar, se crea un hilo de clase **SubTareaSeparada**, y continúa el bucle de eventos de **Counter2**.

La clase **SubTareaSeparada** es una simple extensión de **Thread** con un constructor que ejecuta el hilo invocando a **start( )**, y un método **run( )** que esencialmente contiene el código "**comenzar( )**" de **Contador1.java**.

Dado que **SubtareaSeparada** es una clase interna, puede acceder directamente al **JTextField t** de **Contador2**; se puede ver que esto ocurre dentro de **run( )**. El campo **t** de la clase externa es **private** puesto que **SubTareaSeparada** puede acceder a él sin ningún permiso especial —y siempre es bueno hacer campos "tan **private** como sea posible", de forma que puedan ser cambiados accidentalmente por fuerzas externas a la clase.

Cuando se presiona el botón **onOff** conmuta el **flagEjecutar** de dentro del objeto **SubTareaSeparada**. Ese hilo (cuando mira al *flag*) puede empezar y pararse por sí mismo. Presionar el botón **onOff** produce una respuesta aparentemente instantánea. Por supuesto, la respuesta no es verdaderamente instantánea, no como la de un sistema dirigido por interrupciones. El contador sólo se detiene cuando el hilo tiene la UCP y se da cuenta de que el *flag* ha cambiado.

Se puede ver que la clase interna **SubTareaSeparada** es **private**, lo que significa que sus campos y métodos pueden tener el acceso por defecto (excepto en el caso de **run( )**, que debe ser **public**, puesto que es **public** en la clase base). La clase **private** interna no está accesible más que a **Contador2**, y ambas clases están fuertemente acopladas. En cualquier momento en que dos clases parezcan estar fuertemente acopladas entre sí, hay que considerar las mejoras de codificación y mantenimiento que se obtendrían utilizando clases internas.

Combinar el hilo con la clase principal

En el ejemplo de arriba puede verse que la clase hilo está separada de la clase principal del programa. Esto tiene mucho sentido y es relativamente fácil de entender. Sin embargo, hay una forma alternativa que se verá a menudo que no está tan clara, pero que suele ser más concisa (y que es probablemente lo que la dota de popularidad). Esta forma combina la clase principal del programa con la clase hilo haciendo que la clase principal del programa sea un hilo. Puesto que para un programa IGU la clase principal del programa debe heredarse de **Frame** o de **Applet**, hay que usar una interfaz para añadirle funcionalidad adicional. A esta interfaz se le denomina **Runnable**, y contiene el mismo método básico que **Thread**. De hecho, **Thread** también implementa **Runnable**, lo que sólo especifica la existencia de un método **run( )**.

El *uso* de programa/hilo combinado no es tan obvio. Cuando empieza el programa se crea un objeto que es **Runnable**, pero no se arranca el hilo. Esto hay que hacerlo explícitamente. Esto se puede ver en el programa siguiente, que reproduce la funcionalidad de **Contador2**:

```
//: c14:Contador3.java
// Usando la interfaz Runnable para convertir la clase
// principal en un hilo.
// <applet code=Contador3 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Contador3
    extends JApplet implements Runnable {
  private int conteo = 0;
  private boolean flagEjecutar = true;
  private Thread hiloPropio = null;
  private JButton
    empezar = new JButton("Empezar"),
    onOff = new JButton("Conmutar");
  private JTextField t = new JTextField(10);
  public void run() {
    while (true) {
      try {
```

```
            hiloPropio.sleep(100);
          } catch(InterruptedException e) {
            System.err.println("Interrumpido");
          }
          if(flagEjecutar)
            t.setText(Integer.toString(conteo++));
        }
      }
      class EmpezarL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
          if(hiloPropio == null) {
            hiloPropio = new Thread(Contador3.this);
            hiloPropio.start();
          }
        }
      }
      class OnOffL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
          flagEjecutar = !flagEjecutar;
        }
      }
      public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        empezar.addActionListener(new EmpezarL());
        cp.add(empezar);
        onOff.addActionListener(new OnOffL());
        cp.add(onOff);
      }
      public static void main(String[] args) {
        Console.run(new Contador3(), 300, 100);
      }
    } ///:~
```

Ahora el **run( )** está dentro de la clase, pero sigue estando dormido tras completarse **init( )**. Cuando se presiona el botón **Empezar**, se crea el hilo (si no existe ya) en una expresión bastante oscura:

```
new Thread(Contador3.this)
```

Cuando algo tiene una interfaz **Runnable**, simplemente significa que tiene un método **run( )**, pero no hay nada especial en ello —no produce ninguna habilidad innata a los hilos, como las de una clase heredada de **Thread**. Por tanto, para producir un hilo a partir de un objeto **Runnable**, hay que crear un objeto **Thread** separado como se mostró arriba, pasándole el objeto **Runnable** al constructor **Thread** especial. Después se puede llamar al **start( )** de ese hilo:

```
hiloPropio.start();
```

Esta sentencia lleva a cabo la inicialización habitual y después llama a **run( )**.

El aspecto conveniente de la **interface Runnable** es que todo pertenece a la misma clase. Si es necesario acceder a algo, simplemente se hace sin recorrer un objeto separado. Sin embargo, como se vio en el capítulo anterior, este acceso es tan sencillo como usar una clase interna[1].

Construir muchos hilos

Considérese la creación de muchos hilos distintos. Esto no se puede hacer con el ejemplo de antes, por lo que hay que volver hacia atrás, cuando se tenían clases separadas heredadas de **Thread** para encapsular el método **run( )**. Pero ésta es una solución más general y más fácil de entender, por lo que mientras que el ejemplo anterior muestra un estilo de codificación muy abundante, no podemos recomendarlo para la mayoría de los casos porque es un poco más confuso y menos flexible.

El ejemplo siguiente repite la forma de los ejemplos de arriba con contadores y botones de conmutación. Pero ahora toda la información de un contador particular, incluyendo el botón y el campo de texto, están dentro de su propio objeto, heredado de **Thread**. Todos los campos de **Teletipo** son **private**, lo que significa que se puede cambiar la implementación de **Teletipo** cuando sea necesario, incluyendo la cantidad y tipo de componentes de datos a adquirir y la información a mostrar. Cuando se crea un objeto **Teletipo**, el constructor añade sus componentes visuales al panel contenedor del objeto externo:

```
//: c14:Contador4.java
// Manteniendo el hilo como una clase distinta,
// se puede tener tantos hilos como se desee.
// <applet code=Contador4 width=200 height=600>
// <param name=tamanio value="12"></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Contador4 extends JApplet {
  private JButton empezar = new JButton("Empezar");
  private boolean empezado = false;
  private Teletipo[] s;
  private boolean esApplet = true;
  private int tamanio = 12;
  class Teletipo extends Thread {
    private JButton b = new JButton("Conmutar");
```

[1] **Runnable** ya estaba en Java 1.0, mientras que las clases internas no se introdujeron hasta Java 1.1, que pueda deberse probablemente a la existencia de **Runnable**. También las arquitecturas multihilo tradicionales se centraron en que se ejecutara una función en vez de un objeto. Preferimos heredar de **Thread** siempre que se pueda; nos parece más claro y más flexible.

```
    private JTextField t = new JTextField(10);
    private int conteo = 0;
    private boolean flagEjecutar = true;
    public Teletipo() {
      b.addActionListener(new Conmutador());
      JPanel p = new JPanel();
      p.add(t);
      p.add(b);
      // Invoca JApplet.getContentPane().add():
      getContentPane().add(p);
    }
    class ConmutadorL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        flagEjecutar = !flagEjecutar;
      }
    }
    public void run() {
      while (true) {
        if (flagEjecutar)
          t.setText(Integer.toString(conteo++));
        try {
          sleep(100);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
    }
  }
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!empezado) {
        empezado = true;
        for (int i = 0; i < s.length; i++)
          s[i].start();
      }
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    // Capturar el parámetro "tamanio" de la página a tamanio Web:
    if (esApplet) {
      String tam = getParameter("tamanio");
      if(tam != null)
        tamanio = Integer.parseInt(tam);
```

```
      }
      s = new Teletipo[tamanio];
      for (int i = 0; i < s.length; i++)
        s[i] = new Teletipo();
      empezar.addActionListener(new EmpezarL());
      cp.add(empezar);
    }
    public static void main(String[] args) {
      Contador4 applet = new Contador4();
      // Esto no es un applet, por lo que se pone el flag a uno
      // y se producen los valores de parámetros para args:
      applet.esApplet = false;
      if(args.length != 0)
        applet.size = Integer.parseInt(args[0]);
      Console.run(applet, 200, applet.tamanio * 50);
    }
} ///:~
```

Teletipo no sólo contiene su equipamiento como hilo, sino que también incluye la forma de controlar y mostrar el hilo. Se pueden crear tantos hilos como se desee sin crear explícitamente los componentes de ventanas.

En **Contador4** hay un array de objetos **Teletipo** llamado **s**. Para maximizar la flexibilidad, se inicializa el tamaño de este array saliendo a la página web utilizando los parámetros del applet. Esto es lo que aparenta el parámetro tamanio en la página, insertado en la etiqueta applet:

```
<param name=tamanio value="20">
```

Las palabras clave **param**, **name** y **value** pertenecen a HTML. La palabra **name** es aquello a lo que se hará referencia en el programa, y **value** puede ser una cadena de caracteres, no sólo algo que desemboca en un número.

Se verá que la determinación del tamaño del array **s** se hace dentro de **init( )**, y no como parte de una definición de **s**. Es decir, *no se puede* decir como parte de la definición de clase (fuera de todo método):

```
int tamanio = Integer.parseInt(getParameter("Tamanio"));
Teletipo[] s = new Teletipo[tamanio];
```

Esto se puede compilar, pero se obtiene una "null-pointer exception" extraña en tiempo de ejecución. Funciona bien si se mueve la inicialización **getParameter( )** dentro de **init( )**. El marco de trabajo *applet* lleva a cabo la inicialización necesaria en los parámetros antes de **init( )**.

Además, este código puede ser tanto un *applet* como una aplicación. Cuando es una aplicación, se extrae el parámetro **tamanio** de la línea de comandos (o se utiliza un valor por defecto).

Una vez que se establece el tamaño del array, se crean nuevos objetos **Teletipo**; como parte del constructor **Teletipo** se añade al *applet* el botón y el campo texto de cada **Teletipo**.

Presionar el botón **empezar** implica recorrer todo el array de **Teletipos** y llamar al método **start( )** de cada uno. Recuérdese que **start( )** lleva a cabo la inicialización necesaria por cada hilo, invocando al método **run( )** del hilo.

El oyente **ConmutadorL** simplemente invierte el *flag* de **Teletipo**, de forma que cuando el hilo asociado tome nota, pueda reaccionar de forma acorde.

Uno de los aspectos más valiosos de este ejemplo es que permite crear fácilmente conjuntos grandes de subtareas independientes además de monitorizar su comportamiento. En este caso, se verá que a medida que crece el número de tareas, la máquina mostrará mayor divergencia en los números que muestra debido a la forma de servir esos hilos.

También se puede experimentar para descubrir la importancia de **sleep(100)** dentro de **teletipo.run( )**. Si se retira el **sleep( )**, todo funcionará correctamente hasta presionar un botón de conmutar. Después, ese hilo particular tendrá un **flagEjecutar** falso, y el **run( )** se verá envuelto en un bucle infinito y rígido, que parece difícil de romper, haciendo que el grado de respuesta y la velocidad del programa descienda drásticamente.

Hilos demonio

Un hilo "demonio" es aquél que supuestamente proporciona un servicio general en segundo plano mientras se está ejecutando el programa, no siendo parte de la esencia del programa. Por consiguiente, cuando todos los hilos no demonio acaban, se finaliza el programa. Consecuentemente, mientras se siga ejecutando algún hilo no demonio, el programa no acabará. (Por ejemplo, puede haber un hilo ejecutando el método **main( )**.)

Se puede averiguar si un hilo es un demonio llamando a **isDaemon( )**, y se puede activar o desactivar el funcionamiento como demonio de un hilo con **setDaemon( )**. Si un hilo es un demonio, todos los hilos que cree serán a su vez demonios.

El ejemplo siguiente, demuestra los hilos demonio:

```
//: c14:Demonios.java
// Comportamiento endemoniado.
import java.io.*;

class Demonio extends Thread {
  private static final int TAMANIO = 10;
  private Thread[] t = new Thread[TAMANIO];
  public Demonio() {
    setDaemon(true);
    start();
  }
  public void run() {
    for(int i = 0; i < TAMANIO; i++)
      t[i] = new EngendrarDemonio(i);
```

```
      for(int i = 0; i < TAMANIO; i++)
           t [i] = new EngendrarDemonio(i);
           for (int = 0; <TAMANIO; i++)
        System.out.println(
          "t[" + i + "].isDaemon() = "
          + t[i].isDaemon());
      while(true)
        yield();
    }
}

class EngendrarDemonio extends Thread {
  public EngendrarDemonio(int i) {
    System.out.println(
      "EngendrarDemonio " + i + " empezado");
    start();
  }
  public void run() {
    while(true)
      yield();
  }
}

public class Demonios {
  public static void main(String[] args)
  throws IOException {
    Thread d = new Demonio();
    System.out.println(
      "d.isDaemon() = " + d.isDaemon());
    // Permitir a los hilos demonio
    // acabar sus procesos de arranque:
    System.out.println("Presione cualquier tecla");
    System.in.read();
  }
} ///:~
```

El hilo **Demonio** pone su hilo a true y después engendra otros muchos hilos para mostrar que son también demonios. Después se introduce en un bucle infinito y llama a **yield( )** para ceder el control a los otros procesos. En una versión anterior de este programa, los bucles infinitos incrementarían contadores **int**, pero eso parecía bloquear todo el programa. Usar **yield( )** hace que el ejemplo sea bastante picante.

No hay nada para evitar que el programa termine una vez que acabe el método **main( )**, puesto que no hay nada más que hilos demonio en ejecución. Para poder ver los resultados de lanzar todos los hilos demonio se coloca **System.in** para leer, de forma que el programa espere una pulsación de tecla antes de terminar. Sin esto sólo se ve alguno de los resultados de la creación de los hilos demonio. (Puede probarse a reemplazar el código de **read( )** con llamadas a **sleep( )** de varias longitudes y observar el comportamiento.)

Compartir recursos limitados

Se puede pensar que un programa de un hilo es una entidad solitaria que recorre el espacio del problema haciendo sólo una cosa en cada momento. Dado que sólo hay una entidad, no hay que pensar nunca que pueda haber dos entidades intentando usar el mismo recurso a la vez, como si fueran dos conductores intentando aparcar en el mismo sitio, o atravesar la misma puerta simultáneamente, o incluso, hablar.

Con la capacidad multihilo, los elementos dejan de ser solitarios, y ahora existe la posibilidad de que dos o más hilos traten de usar el mismo recurso limitado a la vez. Hay que prevenir las colisiones por un recurso o, de lo contrario, se tendrán dos hilos intentando acceder a la misma cuenta bancaria a la vez, o imprimir en la misma impresora o variar la misma válvula, etc.

Acceder a los recursos de forma inadecuada

Considérese una variación de los contadores que se han venido usando hasta ahora en el capítulo. En el ejemplo siguiente, cada hilo contiene dos contadores que se incrementan y muestran dentro de **run( )**. Además, hay otro hilo de clase **Observador** que vigila los contadores para que siempre sean equivalentes. Ésta parece una actividad innecesaria, puesto que mirando al código parece obvio que los contadores siempre tendrán el mismo valor. Pero es justamente ahí donde aparece la sorpresa. He aquí la primera versión del programa:

```
//: c14:Compartiendo1.java
// Problemas con la compartición de recursos y los hilos.
// <applet code=Compartiendo1 width=350 height=500>
// <param name=tamanio value="12">
// <param name=observadores value="15">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Compartiendo1 extends JApplet {
  private static int recuentoAcceso = 0;
  private static JTextField conteoA =
    new JTextField("0", 7);
  public static void incrementarAcceso() {
    recuentoAcceso++;
    conteoA.setText(Integer.toString(recuentoAcceso));
  }
  private JButton
    empezar = new JButton("Empezar"),
    observador = new JButton("Vigilar");
  private boolean esApplet = true;
```

```
  private int numContadores = 12;
  private int numObservadores = 15;
  private DosContadores[] s;
  class DosContadores extends Thread {
    private boolean empezado = false;
    private JTextField
      t1 = new JTextField(5),
      t2 = new JTextField(5);
    private JLabel l =
      new JLabel("conteo1 == conteo2");
    private int conteo1 = 0, conteo2 = 0;
    // Añadir los componentes a mostrar como un panel:
    public DosContadores() {
      JPanel p = new JPanel();
      p.add(t1);
      p.add(t2);
      p.add(l);
      getContentPane().add(p);
    }
    public void start() {
      if(!empezado) {
        empezado = true;
        super.start();
      }
    }
    public void run() {
      while (true) {
        t1.setText(Integer.toString(conteo1++));
        t2.setText(Integer.toString(conteo2++));
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
    }
    public void pruebaSinc() {
      Compartiendo1.incrementarAcceso();
      if(conteo1 != conteo2)
        l.setText("Sin sincronizar");
    }
  }
  class Observador extends Thread {
    public Observador() { start(); }
    public void run() {
```

```
      while(true) {
        for(int i = 0; i < s.length; i++)
          s[i].pruebaSinc();
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
    }
  }
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < s.length; i++)
        s[i].start();
    }
  }
  class ObservadorL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < numObservadores; i++)
        new Observador();
    }
  }
  public void init() {
    if(esApplet) {
      String contadores = getParameter("tamanio");
      if(contadores != null)
        numContadores = Integer.parseInt(contadores);
      String observadores = getParameter("observadores");
      if(observadores != null)
        numObservadores = Integer.parseInt(observadores);
    }
    s = new DosContadores[numContadores];
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new DosContadores();
    JPanel p = new JPanel();
    empezar.addActionListener(new EmpezarL());
    p.add(empezar);
    observador.addActionListener(new ObservadorL());
    p.add(observador);
    p.add(new JLabel("Cuenta de accesos"));
    p.add(conteoA);
    cp.add(p);
```

```
  }
  public static void main(String[] args) {
    Compartiendo1 applet = new Compartiendo1();
    // Esto no es un applet, por lo que se pone el flag a uno
    // y se producen los valores de parámetros para args:
    applet.esApplet = false;
    applet.numContadores =
      (args.length == 0 ? 12 :
        Integer.parseInt(args[0]));
    applet.numObservadores =
      (args.length < 2 ? 15 :
        Integer.parseInt(args[1]));
    Console.run(applet, 350,
      applet.numObservadores * 50);
  }
} ///:~
```

Como antes, cada contador sólo contiene sus componentes propios a visualizar: dos campos de texto y una etiqueta que inicialmente indica que los contadores son equivalentes. Estos componentes se añaden al panel de contenidos del objeto de la clase externa en el constructor **DosContadores**.

Dado que el hilo **DosContadores** empieza vía una pulsación de tecla por parte del usuario, es posible que se llame a **start( )** más de una vez. Es ilegal que se llame a **Thread.start( )** más de una vez para un mismo hilo (se lanza una excepción). Se puede ver la maquinaria que evita esto en el flag **empezado** y el método **start( )** superpuesto.

En **run( )**, se incrementan y muestran **conteo1** y **conteo2,** de forma que parecen ser idénticos. Después se llama a **sleep( )**; sin esta llamada el programa se detiene bruscamente porque la UCP tiene dificultad para conmutar las tareas.

El método **pruebaSinc( )** lleva a cabo la aparentemente inútil actividad de comprobar si **conteo1** es equivalente a **conteo2**; si no son equivalentes, pone la etiqueta a "Sin Sincronizar" para indicar esta circunstancia. Pero primero llama a un miembro estático de la clase **Compartiendo1,** que incrementa y muestra un contador de accesos para mostrar cuántas veces se ha dado esta comprobación con éxito. (La razón de esto se hará evidente en variaciones ulteriores de este ejemplo.)

La clase **Observador** es un hilo cuyo trabajo es invocar a **pruebaSinc( )** para todos los objetos de **DosContenedores** activos. Lo hace recorriendo el array mantenido en el objeto **Compartiendo1**. Se puede pensar que **Observador** está mirando constantemente por encima del hombro de los objetos **DosContadores**.

Compartiendo1 contiene un array de objetos **DosContenedores** que inicializa en **init( )** y comienza como hilos al presionar el botón "empezar". Más adelante, al presionar el botón "Vigilar", se crean uno o más vigilantes que se liberan sobre los hilos **DosContadores**.

Nótese que para ejecutar esto como un *applet* en un navegador, la etiqueta *applet* tendrá que contener las líneas:

```
<param name=tamanio value="20">
```

```
<param name=observadores value="1">
```

Se puede experimentar variando la anchura, altura y parámetros para satisfacer los gustos de cada uno. Cambiando el **tamaño** y los **observadores**, se puede variar el comportamiento del programa. Este programa está diseñado para ejecutarse como una aplicación independiente a la que se pasan los parámetros por la línea de comandos (o proporcionando valores por defecto).

He aquí la parte más sorprendente. En **DosContadores.run( )**, se va pasando repetidamente por el bucle infinito recorriendo las líneas siguientes:

```
t1.setText(Integer.toString(conteo1++));
t2.setText(Integer.toString(conteo2++));
```

(además de dormirse, pero eso ahora no importa). Al ejecutar el programa, sin embargo, se descubrirá que se observarán **conteo1** y **conteo2** (por parte de los **Observadores**) ¡para que a veces no sean iguales! Esto se debe a la naturaleza de los hilos —que pueden ser suspendidos en cualquier momento. Por ello, en ocasiones, se da la suspensión *justo* cuando se ha ejecutado la primera de estas líneas y no la segunda, y aparece el hilo **Observador** ejecutando la comprobación justo en ese momento, descubriendo, por consiguiente, que ambos hilos son distintos.

Este ejemplo muestra un problema fundamental del uso de los hilos. Nunca se sabe cuándo se podría ejecutar un hilo. Imagínese sentado en una mesa con un tenedor, justo a punto de engullir el último fragmento de comida del plato y justo cuando el tenedor va a alcanzarla, la comida simplemente se desvanece (porque se suspendió el hilo y apareció otro que robó la comida). Éste es el problema con el que se está tratando.

En ocasiones, no importa que un mismo recurso esté siendo accedido a la vez que se está intentado usar (la comida está en algún otro plato). Pero para que el multihilo funcione, es necesario disponer de alguna forma de evitar que dos hilos accedan al mismo recurso, al menos durante ciertos periodos críticos.

Evitar este tipo de colisión es simplemente un problema de poner un bloqueo en el recurso cuando lo esté usando un hilo. El primer hilo que accede al recurso lo bloquea, de forma que posteriormente los demás hilos no pueden acceder a este recurso hasta que éste quede desbloqueado, momento en el que es otro el hilo que lo bloquea y usa, etc. Si el asiento delantero de un coche es un recurso limitado, el primer niño que grite: "¡Para mí!", lo habrá bloqueado.

Cómo comparte Java los recursos

Java tiene soporte integrado para prevenir colisiones sobre cierto tipo de recurso: la memoria de un objeto. Puesto que generalmente se hacen los elementos de datos de clase **private** y se accede a esa memoria sólo a través de métodos, se pueden evitar las colisiones haciendo que un método particular sea **synchronized**. Sólo un hilo puede invocar a un método **synchronized** en cada instante para cada objeto (aunque ese hilo puede invocar a más de un método **synchronized** de varios objetos). He aquí métodos **synchronized** sencillos:

```
synchronized void f()  { /* ... */ }
```

```
synchronized void g() { /* ... */ }
```

Cada objeto contiene un único bloqueo (llamado también *monitor*) que forma parte del objeto automáticamente (no hay que escribir ningún código especial). Cuando se llama a cualquier método **synchronized**, se bloquea el objeto y no se puede invocar a ningún otro método **synchronized** del objeto hasta que el primero acabe y libere el bloqueo. En el ejemplo de arriba, si se invoca a **f( )** de un objeto, no se puede invocar a **g( )** de ese mismo objeto hasta que se complete **f( )** y libere el bloqueo. Por consiguiente, hay un único bloqueo que es compartido por todos los métodos **synchronized** de un objeto en particular, y este bloqueo evita que la memoria en común sea escrita por más de un método en cada instante (es decir, más de un hilo en cada momento).

También hay un único bloqueo por clase (como parte del objeto **Class** de la clase), de forma que los métodos **synchronized static** pueden bloquearse mutuamente por accesos simultáneos a datos **static** en el ámbito de una clase.

Nótese que si se desea proteger algún recurso de accesos simultáneos por parte de múltiples hilos, se puede hacer forzando el acceso a ese recurso mediante métodos **synchronized**.

Sincronizar los contadores

Armado con esta nueva palabra clave, parece que la solución está a mano: simplemente se usará la palabra **synchronized** para los métodos de **DosContadores**. El ejemplo siguiente es igual que el anterior, con la adición de la nueva palabra:

```
//: c14:Compartiendo2.java
// Usando la palabra synchronized para evitar
// el acceso múltiple a un recurso en particular.
// <applet code=Compartiendo2 width=350 height=500>
// <param name=tamanio value="12">
// <param name=observadores value="15">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Compartiendo2 extends JApplet {
  DosContadores[] s;
  private static int recuentoAcceso = 0;
  private static JTextField conteoA =
    new JTextField("0", 7);
  public static void incrementarAccesos() {
    recuentoAcceso++;
    conteoA.setText(Integer.toString(recuentoAcceso));
  }
  private JButton
    empezar = new JButton("Empezar"),
```

```
    observador = new JButton("Vigilar");
  private boolean esApplet = true;
  private int numContadores = 12;
  private int numObservadores = 15;

  class DosContadores extends Thread {
    private boolean empezado = false;
    private JTextField
      t1 = new JTextField(5),
      t2 = new JTextField(5);
    private JLabel l =
      new JLabel("conteo1 == conteo2");
    private int conteo1 = 0, conteo2 = 0;
    public DosContadores() {
      JPanel p = new JPanel();
      p.add(t1);
      p.add(t2);
      p.add(l);
      getContentPane().add(p);
    }
    public void start() {
      if(!empezado) {
        empezado = true;
        super.start();
      }
    }
    public synchronized void run() {
      while (true) {
        t1.setText(Integer.toString(conteo1++));
        t2.setText(Integer.toString(conteo2++));
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
    }
    public synchronized void pruebaSinc() {
      Compartiendo2.incrementarAcceso();
      if(conteo1 != conteo2)
        l.setText("Sin Sincronizar");
    }
  }

  class Observador extends Thread {
```

```
    public Observador() { start(); }
    public void run() {
      while(true) {
        for(int i = 0; i < s.length; i++)
          s[i].pruebaSinc();
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
    }
  }
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < s.length; i++)
        s[i].start();
    }
  }
  class ObservadorL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < numObservadores; i++)
        new Observador();
    }
  }
  public void init() {
    if(esApplet) {
      String contadores = getParameter("tamanio");
      if(contadores != null)
        numContadores = Integer.parseInt(contadores);
      String observadores = getParameter("observadores");
      if(observadores != null)
        numObservadores = Integer.parseInt(observadores);
    }
    s = new DosContadores[numContadores];
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new DosContadores();
    JPanel p = new JPanel();
    empezar.addActionListener(new EmpezarL());
    p.add(empezar);
    observador.addActionListener(new ObservadorL());
    p.add(observador);
    p.add(new Label("Cuenta de accesos"));
```

```
    p.add(ConteoA);
    cp.add(p);
  }
  public static void main(String[] args) {
    Compartiendo2 applet = new Compartiendo2();
    // Esto no es un applet, por lo que se pone el flag a uno
    // y se producen los valores de parámetros para args:
    applet.esApplet = false;
    applet.numContadores =
      (args.length == 0 ? 12 :
        Integer.parseInt(args[0]));
    applet.numObservadores =
      (args.length < 2 ? 15 :
        Integer.parseInt(args[1]));
    Console.run(applet, 350,
      applet.numContadores * 50);
  }
} ///:~
```

Se verá que, *tanto* **run( )** *como* **pruebaSinc( )** son **synchronized**. Si se sincroniza sólo uno de los métodos, el otro es libre de ignorar el bloqueo del objeto y accederlo con impunidad. Éste es un punto importante: todo método que acceda a recursos críticos compartidos debe ser **synchronized** o no funcionará correctamente.

Ahora aparece un nuevo aspecto. El **Observador** nunca puede saber qué está ocurriendo exactamente porque todo el método **run( )** está **synchronized**, y dado que **run( )** siempre se está ejecutando para cada objeto, el bloqueo siempre está activado y no se puede llamar nunca a **pruebaSinc ( )**. Esto se puede ver porque **RecuentoAcceso** nunca cambia.

Lo que nos gustaría de este ejemplo es alguna forma de aislar sólo *parte* del código dentro de **run( )**. La sección de código que se desea aislar así se denomina una *sección crítica* y la palabra clave **synchronized** se usa de forma distinta para establecer una sección crítica. Java soporta secciones críticas con el *bloque sincronizado*; esta vez, **synchronized** se usa para especificar el objeto cuyo bloqueo se usa para sincronizar el código adjunto:

```
synchronized(objetoSinc) {
  // A este código sólo puede
  // acceder un thread a la vez
}
```

Antes de poder entrar al bloque sincronizado, hay que adquirir el bloqueo en **objetoSinc**. Si algún otro hilo ya tiene este bloqueo, no se puede entrar en este bloque hasta que el bloqueo ceda.

El ejemplo **Compartiendo2** puede modificarse quitando la palabra clave **synchronized** de todo el método **run( )** y pendiendo en su lugar un bloque **synchronized** en torno a las dos líneas críticas. Pero ¿qué objeto debería usarse como bloqueo? El que ya está involucrado en **pruebaSinc( )**, que es el objeto actual ¡(**this**)! Por tanto, el método **run( )** modificado es así:

```
public void run() {
   while (true) {
      synchronized(this) {
         t1.setText(Integer.toString(conteo1++));
         t2.setText(Integer.toString(conteo2++));
      }
      try {
         sleep(500);
      } catch(InterruptedException e) {
         System.err.println("Interrumpido");
      }
   }
}
```

Éste es el único cambio que habría que hacer a **Compartiendo2.java**, y se verá que, mientras que los dos contadores nunca están fuera de sincronismo (de acuerdo al momento en que **Observador** puede consultar su valor), se sigue proporcionando un acceso adecuado a **Observador** durante la ejecución de **run( )**.

Por supuesto, toda la sincronización depende de la diligencia del programador: todo fragmento de código que pueda acceder a un recurso compartido deberá envolverse en un bloque sincronizado.

Eficiencia sincronizada

Dado que tener dos métodos que escriben al mismo fragmento de información no parece *nunca* ser una buena idea, podría parecer que tiene sentido que todos los métodos sean automáticamente **synchronized** y eliminar de golpe todas las palabras **synchronized**. (Por supuesto, el ejemplo con un **synchronized run( )** muestra que esto tampoco funcionaría.) Pero resulta que adquirir un bloqueo no es una operación barata —multiplica el coste de una llamada a un método (es decir, la entrada y salida del método, no la ejecución del método) al menos por cuatro, y podría ser mucho más dependiente de la implementación en sí. Por tanto, si se sabe que un método en particular no causará problemas de contención, es mejor no poner la palabra clave **synchronized**. Por otro lado, dejar de lado la palabra **synchronized** por considerarla un cuello de botella, esperando que no se den colisiones, es una invitación al desastre.

Revisar los JavaBeans

Ahora que se entiende la sincronización, se puede echar un nuevo vistazo a los JavaBeans. Cuando se cree un Bean, hay que asumir que se ejecutará en un entorno multihilo. Esto significa que:

1. Siempre que sea posible, todos los métodos **public** de un Bean deberían ser **synchronized**. Por supuesto, esto incurre en cierta sobrecarga en tiempo de ejecución. Si eso es un problema, se pueden dejar no **synchronized** los métodos que no causen problemas en secciones críticas, pero hay que tener en cuenta que esto no siempre es obvio. Los métodos encargados de calificar suelen ser pequeños (como es el caso de **getTamanioCirculo()** en el ejemplo siguiente) y/o "atómicos", es decir, la llamada al método se ejecuta en una cantidad de código

tan pequeña que el objeto no puede variar durante la ejecución. Hacer estos métodos no **synchronized** podría no tener un efecto significativo en la velocidad de ejecución de un programa. También se podrían hacer **public** todos los métodos de un Bean. También se podrían hacer **synchronized** todos los métodos **public** de un Bean, y eliminar la palabra clave **synchronized** sólo cuando se tiene la total seguridad de que es necesario hacerlo, y que su eliminación surtirá algún efecto.

2. Al disparar un evento multidifusión a un conjunto de oyentes interesados, hay que asumir que se podrían añadir o eliminar oyentes al recorrer la lista.

Es bastante fácil operar con el primer punto, pero el segundo requiere pensar un poco más. Considérese el ejemplo **BeanExplosion.java** del capítulo anterior. Éste eludía el problema del multihilo ignorando la palabra clave **synchronized** (que no se había presentado aún) y haciendo el evento unidifusión. He aquí un ejemplo modificado para que funcione en un evento multihilo y use la "multidifusión" para los eventos:

```
//: c14:BeanExplosion2.java
// Habría que escribir los Beans así para que puedan
// ejecutarse en un entorno multihilo.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class BeanExplosion2 extends JPanel
    implements Serializable {
  private int xm, ym;
  private int tamanioC = 20; // Tamaño del círculo
  private String texto = "¡Bang!";
  private int tamanioFuente = 48;
  private Color colorT = Color.red;
  private ArrayList OyentesAccion =
    new ArrayList();
  public BeanExplosion2() {
    addMouseListener(new ML());
    addMouseMotionListener(new MM());
  }
  public synchronized int getTamanioCirculo () {
    return tamanioC;
  }
  public synchronized void
  setTamanioCírculo(int nuevoTamanio) {
    tamanioC = nuevoTamanio;
  }
```

```
  public synchronized String getTextoExplosion() {
    return texto;
  }
  public synchronized void
  setTextoExplosion(String nuevoTexto) {
    texto = nuevoTexto;
  }
  public synchronized int getTamanioFuente() {
    return tamanioFuente;
  }
  public synchronized void
  setTamanioFuente(int nuevoTamanio) {
    tamanioFuente = nevoTamanio;
  }
  public synchronized Color getColorTexto() {
    return colorT;
  }
  public synchronized void
  setColorTexto(Color nuevoColor) {
    colorT = nuevoColor;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.black);
    g.drawOval(xm - tamanioC/2, ym - tamanioC/2,
      tamanioC, tamanioC);
  }
  // Éste es el oyente multidifusión, que suele usarse
  // más a menudo que la aproximación
  // unidifusion realizada en BeanExplosion.java:
  public synchronized void
    addActionListener(ActionListener l) {
    oyentesAccion.add(l);
  }
  public synchronized void
    removeActionListener(ActionListener l) {
    oyentesAccion.remove(l);
  }
  // Nótese que esto no es sincronizado:
  public void notifyListeners() {
    ActionEvent a =
      new ActionEvent(BeanExplosion2.this,
        ActionEvent.ACTION_PERFORMED, null);
    ArrayList lv = null;
    // Hacer una copia superficial de la Lista en el caso
```

```
    // de que alguien añada un oyente mientras estamos
    // invocando a oyentes:
    synchronized(this) {
      lv = (ArrayList)oyentesAccion.clone();
    }
    // Llamar a los métodos del oyente:
    for(int i = 0; i < lv.size(); i++)
      ((ActionListener)lv.get(i))
        .actionPerformed(a);
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      Graphics g = getGraphics();
      g.setColor(ColorT);
      g.setFont(
        new Font(
          "TimesRoman", Font.BOLD, tamanioFuente));
      int ancho =
        g.getFontMetrics().stringWidth(texto);
      g.drawString(texto,
        (getSize().width - ancho) /2,
        getSize().height/2);
      g.dispose();
      notifyListeners();
    }
  }
  class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
      xm = e.getX();
      ym = e.getY();
      repaint();
    }
  }
  public static void main(String[] args) {
    BeanExplosion2 bb = new BeanExplosion2();
    bb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        System.out.println("ActionEvent" + e);
      }
    });
    bb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        System.out.println("Accion BeanExplosion2 ");
      }
    });
```

```
    bb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        System.out.println("Mas accion");
      }
    });
    Console.run(bb, 300, 300);
  }
} ///:~
```

Añadir **synchronized** a los métodos es un cambio sencillo. Sin embargo, hay que darse cuenta de que en **addActionListener( )** y **removeActionListener( )** se añaden y eliminan **ActionListeners** de un **ArrayList**, por lo que se puede tener tantos como se quiera.

Se puede ver que el método **notifyListeners( )** *no* es **synchronized**. Puede ser invocado desde más de un hilo simultáneamente. También es posible que **addActionListener( )** o **removeActionListener( )** sean invocados en el medio de una llamada a **notifyListeners( )**, lo que supone un problema puesto que recorre el **ArrayList oyentesAccion**. Para aliviar el problema, se clona el **ArrayList** dentro de una cláusula **synchronized** y se recorre el clon (véase Apéndice A para obtener más detalles sobre la clonación). De esta forma se puede manipular el **ArrayList** original sin que esto suponga ningún impacto sobre **notifyListeners( )**.

El método **paintComponent( )** tampoco es **synchronized**. Decidir si sincronizar o no métodos superpuestos no está tan claro como al añadir métodos propios. En este ejemplo, resulta que **paint( )** parece funcionar bien esté o no **synchronized**. Pero hay que considerar aspectos como:

1. ¿Modifica el método el estado de variables "críticas" dentro del objeto? Para descubrir si las variables son o no "críticas" hay que determinar si serán leídas o modificadas por otros hilos del programa. (En este caso, la lectura y modificación son casi siempre llevadas a cabo por métodos **synchronized**, con lo que basta con examinar éstos.) En el caso de **paint( )**, no se hace ninguna modificación.

2. ¿Depende el método del estado de estas variables "críticas"? Si un método **synchronized** modifica una variable que usa tu método, entonces es más que deseable hacer que ese método también sea **synchronized**. Basándonos en esto, podría observarse que **tamanioC** se modifica en métodos **synchronized**, y que por consiguiente, **paint( )** debería ser **synchronized**. Sin embargo, aquí se puede preguntar: ¿qué es lo peor que puede ocurrir si se cambiase **tamanioC** durante un **paint( )**? Cuando se vea que no ocurre nada demasiado malo, se puede decidir dejar **paint( )** como no **synchronized** para evitar la sobrecarga extra intrínseca a llamadas a este tipo de métodos.

3. Una tercera pista es darse cuenta de si la versión base de **paint( )** es **synchronized**, que no lo es. Éste no es un argumento sólido, sólo una pista. En este caso, por ejemplo, se ha mezclado un campo que *se modifica* vía métodos **synchronized** (como **tamanioC**) en la fórmula **paint( )** y podría haber cambiado la situación. Nótese, sin embargo, que el ser **synchronized** no se hereda —es decir, si un método es **synchronized** en la clase base, *no* es automáticamente **synchronized** en la versión superpuesta de la clase derivada.

Se ha modificado el código de prueba de **BeanExplosion2** con respecto al del capítulo anterior para demostrar la habilidad multidifusión de **BeanExplosion2** añadiendo oyentes extra.

Bloqueo

Un hilo puede estar en uno de estos cuatro estados:

1. *Nuevo*: se ha creado el objeto hilo pero todavía no se ha arrancado, por lo que no se puede ejecutar.
2. *Ejecutable*: Significa que el hilo *puede* ponerse en ejecución cuando el mecanismo de reparto de tiempos de UCP tenga ciclos disponibles para el hilo. Por consiguiente, el hilo podría estar o no en ejecución, pero no hay nada para evitar que sea ejecutado si el planificador así lo dispone; no está ni muerto ni bloqueado.
3. *Muerto*: la forma normal de morir de un hilo es que finalice su método **run( )**. También se puede llamar a **stop( )**, pero esto lanza una excepción que es una subclase de **Error** (lo que significa que no hay obligación de poner la llamada en un bloque **try**). Recuérdese que el lanzamiento de una excepción debería ser un evento especial y no parte de la ejecución normal de un programa; por consiguiente, en Java 2 se ha abolido el uso de **stop( )**. También hay un método **destroy( )** (que jamás se implementó) al que nunca habría que llamar si puede evitarse, puesto que es drástico y no libera bloqueos sobre los objetos.
4. *Bloqueado:* podría ejecutarse el hilo, pero hay algo que lo evita. Mientras un hilo esté en estado bloqueado, el planificador simplemente se lo salta y no le cede ningún tipo de UCP. Hasta que el hilo no vuelva al estado ejecutable no hará ninguna operación.

Bloqueándose

El estado bloqueado es el más interesante, y merece la pena examinarlo más en detalle. Un hilo puede bloquearse por cinco motivos:

1. Se ha puesto el hilo a dormir llamando a **sleep(milisegundos)**, en cuyo caso no se ejecutará durante el tiempo especificado.
2. Se ha suspendido la ejecución del hilo con **suspend( )**. No se volverá ejecutable de nuevo hasta que el hilo reciba el mensaje **resume( )**. (Éstos están en desuso en Java 2, y se examinarán más adelante.)
3. Se ha suspendido la ejecución del hilo con **wait( )**. No se volverá ejecutable de nuevo hasta que el hilo reciba los mensajes **notify( )** o **notifyAll( )**. (Sí, esto parece idéntico al caso 2, pero hay una diferencia que luego revelaremos.)
4. El hilo está esperando a que se complete alguna E/S.
5. El hilo está intentando llamar a un método **synchronized** de otro objeto y el bloqueo del objeto no está disponible.

También se puede llamar a **yield( )** (un método de la clase **Thread**) para ceder voluntariamente la UCP de forma que se puedan ejecutar otros hilos. Sin embargo, si el planificador decide que un hilo

ya ha dispuesto de suficiente tiempo ocurre lo mismo, saltándose al siguiente hilo. Es decir, nada evita que el planificador mueva el hilo y le dé tiempo a otro hilo. Cuando se bloquea un hilo, hay alguna razón por la cual no puede continuar ejecutándose.

El ejemplo siguiente muestra las cinco maneras de bloquearse. Todo está en un único archivo denominado **Bloqueo.java**, pero se examinará en fragmentos discretos. (Se verán las etiquetas "Continuará" y "Continuación" que permiten a la herramienta de extracción de código componerlo todo junto.)

Dado que este ejemplo demuestra algunos métodos en desuso, se *obtendrán* mensajes de en "desuso" durante la compilación.

Primero, el marco de trabajo básico:

```
//: c14:Bloqueo.java
// Demuestra las distintas formas en que puede
// bloquearse un hilo.
// <applet code=Bloqueo width=350 height=550>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import com.bruceeckel.swing.*;

////////////// El marco de trabajo básico ///////////
class Bloqueable extends Thread {
  private Elector elector;
  protected JTextField estado = new JTextField(30);
  protected int i;
  public Bloqueable(Container c) {
    c.add(estado);
    elector = new Elector(this, c);
  }
  public synchronized int leer() { return i; }
  protected synchronized void actualizar() {
    estado.setText(getClass().getName()
      + " estado: i = " + i);
  }
  public void pararElector() {
    // elector.stop(); En desuso desde in Java 1.2
    elector.terminar(); // El enfoque preferido
  }
}

class Elector extends Thread {
```

```
  private Bloqueable b;
  private int sesion;
  private JTextField estado = new JTextField(30);
  private boolean parar = false;
  public Elector(Bloqueable b, Container c) {
    c.add(estado);
    this.b = b;
    start();
  }
  public void terminar() { parar = true; }
  public void run() {
    while (!parar) {
      estado.setText(b.getClass().getName()
        + " Elector " + (++sesion)
        + "; valor = " + b.read());
       try {
        sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
    }
  }
} ///:Continuará
```

La clase **Bloqueable** pretende ser la clase base de todas las clases en el ejemplo que muestra el bloqueo. Un objeto **Bloqueable** contiene un **JTextField** de nombre **estado** que se usa para mostrar información sobre el objeto. El método que muestra esta información es **actualizar( )**. Se puede ver que usa **getClass( ).getName( )** para producir el nombre de la clase, en vez de simplemente imprimirlo; esto se debe a que **actualizar( )** no puede conocer el nombre exacto de la clase a la que se llama, pues será una clase derivada de **Bloqueable**.

El indicador de cambio de **Bloqueable** es un **int i**, que será incrementado por el método **run( )** de la clase derivada.

Por cada objeto **Bloqueable** se arranca un hilo de clase **Elector**, cuyo trabajo es vigilar a su objeto **Bloqueable** asociado para ver los cambios en **i** llamando a **leer( )** e informando de ellos en su **JTextField estado**. Esto es importante: nótese que **leer( )** y **actualizar( )** son ambos **synchronized**, lo que significa que precisan que el bloqueo del objeto esté libre.

Dormir

La primera prueba del programa se hace con **sleep( )**:

```
///:Continuación
////////////// Bloqueando vía sleep() ///////////
class Durmiente1 extends Bloqueable {
  public Durmiente1(Container c) { super(c); }
```

```
  public synchronized void run() {
    while(true) {
      i++;
      actualizar();
       try {
        sleep(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
    }
  }
}

class Durmiente2 extends Bloqueable {
  public Durmiente2(Container c) { super(c); }
  public void run() {
    while(true) {
      cambiar();
       try {
        sleep(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
    }
  }
  public synchronized void cambiar() {
       i++;
       actualizar();
  }
} ///:Continuará
```

En **Durmiente1** todo el método **run( )** es **synchronized**. Se verá que el **Elector** asociado con este objeto se ejecutará alegremente *hasta* que el hilo comience, y después el **Elector** se detiene en seco. Ésta es una forma de bloquear: dado que **Durmiente1.run( )** es **synchronized**, y una vez que el objeto empieza, siempre está dentro de **run( )**, el método nunca cede el bloqueo del objeto, quedando **Elector** bloqueado.

Durmiente2 proporciona una solución haciendo **run( )** no **synchronized**. Sólo es **synchronized** el método **cambiar( )**, lo que significa que mientras **run( )** esté en **sleep( )**, el **Elector** puede acceder al método **synchronized** que necesite, en concreto a **leer( )**. Aquí se verá que el **Elector** continúa ejecutándose al empezar el hilo **Durmiente2**.

Suspender y continuar

La siguiente parte del ejemplo presenta el concepto de la suspensión. La clase **Thread** tiene un método **suspend( )** para detener temporalmente el hilo y **resume( )** lo continúa en el punto en el que

se detuvo. Hay que llamar a **resume( )** desde otro hilo fuera del suspendido, y en este caso hay una clase separada denominada **Resumidor** que hace exactamente eso. Cada una de las clases que demuestra suspender/continuar tiene un **Resumidor** asociado:

```
///:Continuación
////////////// Bloqueando vía suspend() ////////////
class SuspenderResumir extends Bloqueable {
  public SuspenderResumir(Container c) {
    super(c);
    new Resumidor(this);
  }
}

class SuspenderResumir1 extends SuspenderResumir {
  public SuspenderResumir1(Container c) { super(c);}
  public synchronized void run() {
    while(true) {
      i++;
      actualizar();
      suspend(); // En desuso desde Java 1.2
    }
  }
}

class SuspenderResumir2 extends SuspenderResumir {
  public SuspenderResumir2(Container c) { super(c);}
  public void run() {
    while(true) {
      cambiar();
      suspend(); // En desuso desde Java 1.2
    }
  }
  public synchronized void cambiar() {
      i++;
      actualizar();
  }
}

class Resumidor extends Thread {
  private SuspenderResumir sr;
  public Resumidor(SuspenderResumir sr) {
    this.sr = sr;
    start();
  }
  public void run() {
```

```
      while(true) {
          try {
            sleep(1000);
          } catch(InterruptedException e) {
            System.err.println("Interrumpido");
          }
          sr.resume(); // En desuso desde Java 1.2
      }
    }
} ///:Continuará
```

SuspenderResumir1 también tiene un método **syncronized run( )**. De nuevo, al arrancar este hilo se verá que su **Elector** asociado se bloquea esperando a que el bloqueo quede disponible, lo que no ocurre nunca. Esto se fija como antes en **SuspenderResumir2**, en el que no todo el **run( )** es **synchronized,** sino que usa un método **synchronized cambiar( )** diferente.

Hay que ser consciente de que Java 2 ha abolido el uso de **suspend( )** y **resume( )** porque **suspend( )** se guarda el bloqueo sobre el objeto y, por tanto, puede conducir fácilmente a interbloqueos. Es decir, se puede lograr fácilmente que varios objetos bloqueados esperen por culpa de otros que, a su vez, esperan por los primeros, y esto hará que el programa se detenga. Aunque se podrá ver que programas antiguos usan estos métodos, no habría que usarlos. Más adelante, dentro de este capítulo, se describe la solución.

Wait y notify

En los dos primeros ejemplos, es importante entender que, tanto **sleep( )** como **suspend( ),** *no* liberan el bloqueo cuando son invocados. Hay que ser consciente de este hecho si se trabaja con bloqueos. Por otro lado, el método **wait( )** *libera* el bloqueo cuando es invocado, lo que significa que se puede llamar a otros métodos **synchronized** del objeto hilo durante un **wait( )**. En los dos casos siguientes, se verá que el método **run( )** está totalmente **synchronized** en ambos casos, sin embargo, el **Elector** sigue teniendo acceso completo a los métodos **synchronized** durante un **wait( )**. Esto se debe a que **wait( )** libera el bloqueo sobre el objeto al suspender el método en el que es llamado.

También se verá que hay dos formas de **wait( )**. La primera toma como parámetro un número de milisegundos, con el mismo significado que en **sleep( )**: pausar durante ese periodo de tiempo. La diferencia es que en la **wait( )** se libera el bloqueo sobre el objeto *y* se puede salir de **wait( )** gracias a un **notify( ),** o a que una cuenta de reloj expira.

La segunda forma no toma parámetros, y quiere decir que continuará **wait( )** hasta que venga un **notify( ),** no acabando automáticamente tras ningún periodo de tiempo.

Un aspecto bastante único de **wait( )** y **notify( )** es que ambos métodos son parte de la clase base **Object** y no parte de **Thread**, como es el caso de **sleep( ), suspend( )** y **resume( )**. Aunque esto parece un poco extraño a primera vista —tener algo que es exclusivamente para los hilos como parte de la clase base universal— es esencial porque manipulan el bloqueo que también es parte de todo objeto. Como resultado, se puede poner un **wait( )** dentro de cualquier método **synchronized**,

independientemente de si hay algún tipo de hilo dentro de esa clase en particular. De hecho, el *único* lugar en el que se puede llamar a **wait( )** es dentro de un método o bloque **synchronized**. Si se llama a **wait( )** o **notify( )** dentro de un método que no es **synchronized**, el programa compilará, pero al ejecutarlo se obtendrá una **IllegalMonitorStateException**, con el mensaje no muy intuitivo de "hilo actual no propietario". Nótese que, desde métodos no **synchronized**, sí que se puede llamar a **sleep( )**, **suspend( )** y **resume( )** puesto que no manipulan el bloqueo.

Se puede llamar a **wait( )** o **notify( )** sólo para nuestro propio bloqueo. De nuevo, se puede compilar código que usa el bloque erróneo, pero producirá el mismo mensaje **IllegalMonitorStateException** que antes. No se puede jugar con el bloqueo de nadie más, pero se puede pedir a otro objeto que lleve a cabo una operación que manipule su propio bloqueo. Por tanto, un enfoque es crear un método **synchronized** que llame a **notify( )** para su propio objeto. Sin embargo, en **Notificador** se verá la llamada **notify( )** dentro de un bloque **synchronized**:

```
synchronized(en2) {
  en2.notify();
}
```

donde **en2** es el tipo de objeto **EsperarNotificar2**. Este método, que no es parte de **EsperarNotificar2**, adquiere el bloqueo sobre el objeto **en2**, instante en el que es legal que invoque al **notify( )** de **en2** sin obtener, por tanto, la **IllegalMonitorStateException**.

```
///Continuación
////////////// Blocking vía wait() ////////////
class EsperarNotificar1 extends Bloqueable {
  public EsperarNotificar1(Container c) { super(c); }
  public synchronized void run() {
    while(true) {
      i++;
      actualizar();
       try {
        wait(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrumpido");
      }
    }
  }
}

class EsperarNotificar2 extends Bloqueable {
  public EsperarNotificar2(Container c) {
    super(c);
    new Notificador(this);
  }
  public synchronized void run() {
    while(true) {
```

```
        i++;
        actualizar();
         try {
          wait();
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
   }
}

class Notificador extends Thread {
   private EsperarNotificar2 en2;
   public Notificador(EsperarNotificar2 en2) {
      this.en2 = en2;
      start();
   }
   public void run() {
      while(true) {
          try {
           sleep(2000);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
        synchronized(en2) {
          en2.notify();
        }
      }
   }
} ///: Continuará
```

wait( ) suele usarse cuando se ha llegado a un punto en el que se está esperando alguna otra condición, bajo el control de fuerzas externas al hilo y no se desea esperar ociosamente dentro del hilo. Por tanto, **wait( )** permite poner el hilo a dormir mientras espera a que el mundo cambie, y sólo cuando se da un **notify( )** o un **notifyAll( )** se despierta el método y comprueba posibles cambios. Por consiguiente, proporciona una forma de sincronización entre hilos.

Bloqueo en E/S

Si un flujo está esperando a alguna actividad de E/S, se bloqueará automáticamente. En la siguiente porción del ejemplo, ambas clases funcionan con objetos **Reader** y **Writer**, pero en el marco de trabajo de prueba, se establecerá un flujo entubado para permitir a ambos hilos pasarse datos mutuamente de forma segura (éste es el propósito de los flujos entubados).

El **Emisor** pone datos en el **Writer** y se duerme durante una cantidad de tiempo aleatoria. Sin embargo, **Receptor** no tiene **sleep( )**, **suspend( )** ni **wait( )**. Pero cuando hace un **read( )** se bloquea automáticamente si no hay más datos.

```
///:Continuación
class Emisor extends Bloqueable { // enviar
  private Writer salida;
  public Emisor(Container c, Writer salida) {
    super(c);
    this.salida = salida;
  }
  public void run() {
    while(true) {
      for(char c = 'A'; c <= 'z'; c++) {
        try {
          i++;
          salida.write(c);
          estado.setText("Emisor envio: "
            + (char)c);
          sleep((int)(3000 * Math.random()));
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        } catch(IOException e) {
          System.err.println("Problema de ES");
        }
      }
    }
  }
}
class Receptor extends Bloqueable {
  private Reader entrada;
  public Receptor(Container c, Reader entrada) {
    super(c);
    this.entrada = entrada;
  }
  public void run() {
    try {
      while(true) {
        i++; // Muestra que el elector está vivo
        // Bloquea hasta que los caracteres estén allí:
        estado.setText("Receptor lee: "
          + (char)entrada.read());
      }
    } catch(IOException e) {
      System.err.println("Problema de ES");
```

```
    }
  }
} ///:Continuará
```

Ambas clases también ponen información en sus campos **estado** y cambian **i,** de forma que el **Elector** pueda ver que el hilo se está ejecutando.

Probar

La clase *applet* principal es sorprendentemente simple, porque se ha puesto la mayoría de trabajo en el marco de trabajo **Bloqueable**. Básicamente, se crea un array de objetos **Bloqueable**, y puesto que cada uno es un hilo, llevan a cabo sus propias actividades al presionar el botón "empezar". También hay un botón y una cláusula **actionPerformed( )** para detener todos los objetos **Elector**, que **proporcionan** una demostración de la alternativa al método **stop( )** de Thread, en desuso (en Java 2).

Para establecer una conexión entre los objetos **Emisor** y **Receptor**, se crean un **PipedWriter** y un **PipedReader**. Nótese que el **PipedReader entrada** debe estar conectado al **PipedWriter salida** vía un parámetro del constructor. Después de eso, cualquier cosa que se coloque en **salida** podrá ser después extraída de **entrada**, como si pasara a través de una tubería (y de aquí viene el nombre). Los objetos **entrada** y **salida** se pasan a los constructores **Receptor** y **Emisor** respectivamente, que los tratan como objetos **Reader** y **Writer** de cualquier tipo (es decir, se les aplica un molde hacia arriba).

El array **b** de referencias a **Bloqueable** no se inicializa en este momento de la definición porque no se pueden establecer los flujos entubados antes de que se dé esa definición (lo evita la necesidad del bloque **try**).

```
///:Continuación
////////////// Probando todo //////////
public class Bloqueo extends JApplet {
  private JButton
    empezar = new JButton("Empezar"),
    pararElectores = new JButton("Detener los Electores");
  private boolean empezado = false;
  private Bloqueable[] b;
  private PipedWriter salida;
  private PipedReader entrada;
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!empezado) {
        empezado = true;
        for(int i = 0; i < b.length; i++)
          b[i].start();
      }
    }
  }
  class PararElectoresSL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
```

```
        // Demostración de la alternativa
        // preferida a Thread.stop():
        for(int i = 0; i < b.length; i++)
          b[i].pararElector();
      }
    }
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        salida = new PipedWriter();
      try {
        entrada = new PipedReader(salida);
      } catch(IOException e) {
        System.err.println("Problema de PipedReader");
      }
      b = new Bloqueable[] {
        new Durmiente1(cp),
        new Durmiente2(cp),
        new SuspenderResumir1(cp),
        new SuspenderResumir2(cp),
        new EsperarNotificar1(cp),
        new EsperarNotificar2(cp),
        new Emisor(cp, salida),
        new Receptor(cp, entrada)
      };
      empezar.addActionListener(new EmpezarL());
      cp.add(empezar);
      pararElectores.addActionListener(
        new PararElectoresL());
      cp.add(pararElectores);
    }
    public static void main(String[] args) {
      Console.run(new Bloqueo(), 350, 550);
    }
} ///:~
```

Nótese en **init( )** el bucle que recorre todo el array añadiendo los campos de texto **estado** y **elector.estado** a la página.

Cuando se crean inicialmente los hilos **Bloqueable**, cada uno crea y arranca automáticamente su propio **Elector**. Por tanto, se verán los **Electores** ejecutándose antes de que se arranquen los hilos **Bloqueable**. Esto es importante, puesto que algunos de los **Electores** se bloquearán y detendrán cuando arranquen los hilos **Bloqueable**, y es esencial ver esto para entender ese aspecto particular del bloqueo.

Interbloqueo

Dado que los hilos pueden bloquearse *y* dado que los objetos pueden tener métodos **synchronized** que evitan que los hilos accedan a ese objeto hasta liberar el bloqueo de sincronización, es posible que un hilo se quede parado esperando a otro, que, de hecho, espera a un tercero, etc. hasta que el último de la cadena resulte ser un hilo que espera por el primero. Se logra un ciclo continuo de hilos que esperan entre sí, de forma que ninguno puede avanzar. A esto se le llama *interbloqueo*. Es verdad que esto no ocurre a menudo, pero cuando le ocurre a uno es muy frustrante.

No hay ningún soporte de lenguaje en Java que ayude a prevenir el interbloqueo; cada uno debe evitarlo a través de un diseño cuidadoso. Estas palabras no serán suficiente para complacer al que esté tratando de depurar un programa con interbloqueos.

La abolición de **stop( )**, **suspend( )**, **resume( )** y **destroy( )** en Java 2

Uno de los cambios que se ha hecho en Java 2 para reducir la posibilidad de interbloqueo es abolir los métodos **stop( )**, **suspend( )**, **resume( )** y **destroy( )** de **Thread**.

La razón para abolir el método **stop( )** es que no libera los bloqueos que haya adquirido el hilo, y si los objetos están en un estado inconsistente ("dañados") los demás hilos podrán verlos y modificarlos en ese estado. Los problemas resultantes pueden ser grandes y además difíciles de detectar. En vez de usar **stop( )**, habría que seguir el ejemplo de **Bloqueo.java** y usar un indicador (*flag*) que indique al hilo cuando acabar saliendo de su método **run( )**.

Hay veces en que un hilo se bloquea —como cuando se está esperando una entrada— y no puede interrogar al indicador como ocurre en **Bloqueo.java**. En estos casos, se debería seguir sin usar el método **stop( )**, sino usar el método **interrupt( )** de **Thread** para salir del código bloqueado:

```
//: c14:Interrumpir.java
// El enfoque alternativo a usar
// stop() cuando se bloquea un hilo.
// <applet code=Interrumpir width=200 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Bloqueado extends Thread {
  public synchronized void run() {
    try {
      wait(); // Se bloquea
    } catch(InterruptedException e) {
      System.err.println("Interrumpido");
    }
    System.out.println("Saliendo de run()");
```

```
    }
}

public class Interrumpir extends JApplet {
  private JButton
    interrumpir = new JButton("Interrumpido");
  private Bloqueado bloqueado = new Bloqueado();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(interrumpir);
    interrumpir.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          System.out.println("Boton presionado");
          if(bloqueado == null) return;
          Thread eliminar = bloqueado;
          bloqueado = null; // para liberarlo
          eliminar.interrumpir();
        }
      });
    bloqueado.start();
  }
  public static void main(String[] args) {
    Console.run(new Interrumpir(), 200, 100);
  }
} ///:~
```

El **wait( )** de dentro de **Bloqueado.run( )** produce el hilo bloqueado. Al presionar el botón, se pone a **null** la referencia **bloqueado** de forma que será limpiada por el recolector de basura, y se invoca al método **interrupt( )** del objeto. La primera vez que se presione el botón se verá que el hilo acaba, pero una vez que no hay hilos que matar, simplemente hay que ver que se ha presionado el botón.

Los métodos **suspend( )** y **resume( )** resultan ser inherentemente causantes de interbloqueos. Cuando se llama a **suspend( )**, se detiene el hilo destino, pero sigue manteniendo los bloqueos que haya adquirido hasta ese momento. Por tanto, ningún otro hilo puede acceder a los recursos bloqueados, hasta que el hilo continúe. Cualquier hilo que desee continuar el hilo destino, y que también intente usar cualquiera de los recursos bloqueados, producirá interbloqueo. No se debería usar **suspend( )** y **resume( )**, sino que en su lugar se pone un indicador en la clase **Thread** para indicar si debería activarse o suspenderse el hilo. Si el *flag* indica que el hilo está suspendido, el hilo se mete en una espera usando **wait ( )**. Cuando el *flag* indica que debería continuarse el hilo, se reinicia éste con **notify( )**. Se puede producir un ejemplo modificando **Contador2.java**. Aunque el efecto es similar, se verá que la organización del código es bastante diferente —se usan clases internas anónimas para

todos los oyentes y el **Thread** es una clase interna, lo que hace la programación ligeramente más conveniente, puesto que elimina parte de la contabilidad necesaria en **Contador2.java**:

```
//: c14:Suspender.java
// El enfoque alternativo al uso de suspend()
// y resume(), abolidos en Java 2.
// <applet code=Suspender width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Suspender extends JApplet {
  private JTextField t = new JTextField(10);
  private JButton
    suspender = new JButton("Suspender"),
    resumir = new JButton("Continuar");
  private Suspendible ss = new Suspendible();
  class Suspendible extends Thread {
    private int conteo = 0;
    private boolean suspendido = false;
    public Suspendible() { start(); }
    public void fauxSuspender() {
      suspendido = true;
    }
    public synchronized void fauxResumir() {
      suspendido = false;
      notify();
    }
    public void run() {
      while (true) {
        try {
          sleep(100);
          synchronized(this) {
            while(suspendido)
              wait();
          }
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
        t.setText(Integer.toString(conteo++));
      }
    }
  }
```

```
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    suspender.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          ss.fauxSuspender();
        }
      });
    cp.add(suspender);
    resumir.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          ss.fauxResumir();
        }
      });
    cp.add(resumir);
  }
  public static void main(String[] args) {
    Console.run(new Suspender(), 300, 100);
  }
} ///:~
```

El indicador **suspendido** de **suspendido** se usa para activar y desactivar la suspensión. Para suspender, se pone el indicador a **true** llamando a **fauxSuspender( )**, y esto se detecta dentro de **run( )**. El método **wait( )**, como se describió anteriormente en este capítulo, debe ser **synchronized,** de forma que tenga el bloqueo al objeto. En **fauxResumir( )**, se pone el indicador **suspendedido** a **false** y se llama a **notifiy( )** —puesto que esto despierta a **wait( )** de una cláusula **synchronized**, el método **fauxResumir( )** también debe ser **syncronized( )** de forma que adquiera el bloqueo antes de llamar a **notify( )** (por consiguiente, el bloqueo queda disponible para **wait( )**...). Si se sigue el estilo mostrado en este programa, se puede evitar usar **suspend( )** y **resume( )**.

El método **destroy( )** de **Thread( )** nunca fue implementado; es como un **suspend( )** que no se puede continuar, por lo que tiene los mismos aspectos de interbloqueo que **suspend( )**. Sin embargo, éste no es un método abolido y puede que se implemente en una versión futura de Java (posterior a la 2) para situaciones especiales en las que el riesgo de interbloqueo sea aceptable.

Uno podría preguntarse por qué estos métodos, ahora abolidos, se incluyeron en Java en primer lugar. Parece admitir un error bastante importante para simplemente eliminarlas (e introduciendo otro agujero más en los argumentos que hablan del excepcional diseño de Java y de su infalibilidad, de los que tanto hablaban los encargados de marketing en Sun). Lo más alentador del cambio es que indica claramente que es el personal técnico y no el de marketing el que dirige el asunto —descubrieron el problema y lo están arreglando. Creemos que esto es mucho más prometedor y alen-

tador que dejar el problema ahí pues "corregirlo supondría admitir un error". Esto significa que Java continuará mejorando, incluso aunque esto conlleve alguna pequeña molestia para los programadores de Java. Preferimos, no obstante, afrontar estas molestias a ver cómo se estanca el lenguaje.

Prioridades

La *prioridad* de un hilo indica al planificador lo importante que es cada hilo. Si hay varios hilos bloqueados o en espera de ejecutarse, el planificador ejecutará el de mayor prioridad en primer lugar. Sin embargo, esto no significa que los hilos de menor prioridad no se ejecuten (es decir, no se llega a interbloqueo simplemente con la aplicación directa de estos principios). Los hilos de menor prioridad tienden a ejecutarse menos a menudo.

Aunque es interesante conocer las prioridades que se manejan, en la práctica casi nunca hay que establecer a mano las prioridades. Por tanto uno puede saltarse el resto de esta sección si no le interesan las prioridades.

Leer y establecer prioridades

Se puede leer la prioridad de un hilo con **getPriority( )** y cambiarla con **setPriority( )**. Se puede usar la forma de los ejemplos "contador" anteriores para mostrar el efecto de variar las prioridades. En este *applet* se verá que los contadores se ralentizan, dado que se han disminuido las prioridades de los hilos asociados:

```
//: c14:Contador5.java
// Ajustando las prioridades de los hilos.
// <applet code=Contador5 width=450 height=600>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Teletipo2 extends Thread {
  private JButton
    b = new JButton("Conmutar"),
   incPrioridad = new JButton("arriba"),
   decPrioridad = new JButton("abajo");
  private JTextField
    t = new JTextField(10),
    pr = new JTextField(3); // Mostrar la prioridad
  private int conteo = 0;
  private boolean flagEjecutar = true;
  public Teletipo2(Container c) {
    b.addActionListener(new ConmutadorL());
```

```
      incPrioridad.addActionListener(new ArribaL());
      decPrioridad.addActionListener(new AbajoL());
      JPanel p = new JPanel();
      p.add(t);
      p.add(pr);
      p.add(b);
      p.add(incPrioridad);
      p.add(decPrioridad);
      c.add(p);
    }
    class ConmutadorL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        flagEjecutar = !flagEjecutar;
      }
    }
    class ArribaL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        int nuevaPrioridad = getPriority() + 1;
        if(nuevaPrioridad > Thread.MAX_PRIORITY)
          nuevaPrioridad = Thread.MAX_PRIORITY;
        setPriority(nuevaPrioridad);
      }
    }
    class AbajoL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        int nuevaPrioridad = getPriority() - 1;
        if(nuevaPrioridad < Thread.MIN_PRIORITY)
          nuevaPrioridad = Thread.MIN_PRIORITY;
        setPriority(nuevaPrioridad);
      }
    }
    public void run() {
      while (true) {
        if(flagEjecutar) {
          t.setText(Integer.toString(conteo++));
          pr.setText(
            Integer.toString(getPriority()));
        }
        yield();
      }
    }
}

public class Contador5 extends JApplet {
  private JButton
```

```
    empezar = new JButton("Empezar"),
    arribaMax = new JButton("Inc Prioridad Max "),
    abajoMax = new JButton("Dec Prioridad Max ");
  private boolean empezado = false;
  private static final int TAMANIO = 10;
  private Teletipo2[] s = new Teletipo2[TAMANIO];
  private JTextField mp = new JTextField(3);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new Teletipo2(cp);
    cp.add(new JLabel(
      "MAX_PRIORITY = " + Thread.MAX_PRIORITY));
    cp.add(new JLabel("MIN_PRIORITY = "
      + Thread.MIN_PRIORITY));
    cp.add(new JLabel("Agrupar Prioridad Max = "));
    cp.add(mp);
    cp.add(empezar);
    cp.add(arribaMax);
    cp.add(abajoMax);
    empezar.addActionListener(new EmpezarL());
    arribaMax.addActionListener(new ArribaMaxL());
    abajoMax.addActionListener(new AbajoMaxL());
    mostrarMaxPrioridad();
    // Mostrar recursivamente los grupos de hilos padre:
    ThreadGroup padre =
      s[0].getThreadGroup().getParent();
    while(padre != null) {
      cp.add(new Label(
        "Max prioridad del grupo de hilos del padre = "
        + padre.getMaxPriority()));
      padre = padre.getParent();
    }
  }
  public void mostrarMaxPrioridad() {
    mp.setText(Integer.toString(
      s[0].getThreadGroup().getMaxPriority()));
  }
  class EmpezarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!empezado) {
        empezado = true;
        for(int i = 0; i < s.length; i++)
          s[i].start();
```

```
        }
      }
    }
    class ArribaMaxL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        int maxp =
          s[0].getThreadGroup().getMaxPriority();
        if(++maxp > Thread.MAX_PRIORITY)
          maxp = Thread.MAX_PRIORITY;
        s[0].getThreadGroup().setMaxPriority(maxp);
        mostrarMaxPrioridad();
      }
    }
    class AbajoMaxL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        int maxp =
          s[0].getThreadGroup().getMaxPriority();
        if(--maxp < Thread.MIN_PRIORITY)
          maxp = Thread.MIN_PRIORITY;
        s[0].getThreadGroup().setMaxPriority(maxp);
        mostrarMaxPrioridad();
      }
    }
    public static void main(String[] args) {
      Console.run(new Contador5(), 450, 600);
    }
} ///:~
```

Teletipo2 sigue la forma establecida previamente en este capítulo, pero hay un **JTextField** extra para mostrar la prioridad del hilo y dos botones más para incrementar y disminuir la prioridad.

Hay que tener en cuenta el uso de **yield( )**, que devuelve automáticamente el control al planificador. Sin éste, el mecanismo de multihilos sigue funcionando, pero se verá que funciona lentamente (puede intentarse eliminar la llamada a **yield( )** para comprobarlo). También se podría llamar a **sleep( )**, pero entonces el ratio de cuenta estaría controlado por la duración de **sleep( )** en vez de por la prioridad.

El **init( )** de **Contador5** crea un array de diez **Teletipos2**; sus botones y campos los ubica en el formulario el constructor de **Teletipo2**. **Contador5** añade botones para dar comienzo a todo además de incrementar y disminuir la prioridad máxima del grupo de hilos. Además, están las etiquetas que muestran las prioridades máxima y mínima posibles para un hilo, y un **JTextField** para mostrar la prioridad máxima del grupo de hilos. (La siguiente sección describirá los grupos de hilos). Finalmente, también se muestran como etiquetas las prioridades de los grupos de hilos padre.

Cuando se presiona un botón "arriba" o "abajo", se alcanza esa prioridad de **Teletipo2**, que es incrementada o disminuida de forma acorde.

Cuando se ejecute este programa, se apreciarán varias cosas. En primer lugar, la prioridad por defecto del grupo de hilos es cinco. Incluso si se disminuye la prioridad máxima por debajo de cinco antes de que los hilos empiecen (o antes de crearlos, lo que requiere cambiar el código), cada hilo tendrá su prioridad por defecto a cinco.

La prueba más sencilla es tomar un contador y disminuir su prioridad a uno, y observar que cuenta mucho más lentamente. Pero ahora puede intentarse incrementarla de nuevo. Se puede volver a la prioridad del grupo de hilos, pero no a ninguna superior. Ahora puede disminuirse un par de veces la prioridad de un grupo de hilos. Las prioridades de los hilos no varían, pero si se intenta modificar éstas hacia arriba o hacia abajo, se verá que sacan automáticamente la prioridad del grupo de hilos. Además, se seguirá dando a los hilos nuevos una prioridad por defecto, incluso aunque sea más alta que la prioridad del grupo. (Por consiguiente la prioridad del grupo no es una forma de evitar que los hilos nuevos tengan prioridades mayores a las de los existentes.)

Finalmente, puede intentarse incrementar la prioridad máxima del grupo. No puede hacerse. Las prioridades máximas de los grupos de hilos sólo pueden reducirse, nunca incrementarse.

Grupos de hilos

Todos los hilos pertenecen a un grupo de hilos. Éste puede ser, o bien el grupo de hilos por defecto, o un grupo explícitamente especificado al crear el hilo. En el momento de su creación, un hilo está vinculado a un grupo y no puede cambiar a otro distinto. Cada aplicación tiene, al menos, un hilo que pertenece al grupo de hilos del sistema. Si se crean más hilos sin especificar ningún grupo, éstos también pertenecerán al grupo de hilos del sistema.

Los grupos de hilos también deben pertenecer a otros grupos de hilos. El grupo de hilos al que pertenece uno nuevo debe especificarse en el constructor. Si se crea un grupo de hilos sin especificar el grupo de hilos al que pertenezca, se ubicará bajo el grupo de hilos del sistema. Por consiguiente, todos los grupos de hilos de la aplicación tendrán como último padre al grupo de hilos del sistema.

La razón de la existencia de grupos de hilos no es fácil de determinar a partir de la literatura, que tiende a ser confusa en este aspecto. Suelen citarse "razones de seguridad". De acuerdo con Arnold & Gosling[2], "Los hilos de un grupo de hilos pueden modificar a los otros hilos del grupo, incluyendo todos sus descendientes. Un hilo no puede modificar hilos de fuera de su grupo o grupos contenidos." Es difícil saber qué se supone que quiere decir en este caso el término "modificar". El ejemplo siguiente muestra un hilo en un subgrupo "hoja", modificando las prioridades de todos los hilos en su árbol de grupos de hilos, además de llamar a un método para todos los hilos de su árbol.

```
//: c14:PruebaAcceso.java
// Como los hilos pueden acceder a otros hilos de
// un grupo de hilos padre.
```

[2] *The Java Programming Language*, de Ken Arnold y James Gosling, Addison Wesley 1996 pag. 179.

```
public class PruebaAcceso {
  public static void main(String[] args) {
    ThreadGroup
      x = new ThreadGroup("x"),
      y = new ThreadGroup(x, "y"),
      z = new ThreadGroup(y, "z");
    Thread
      uno = new PruebaHilo1(x, "uno"),
      dos = new PruebaHilo2(z, "dos");
  }
}

class PruebaHilo1 extends Thread {
  private int i;
  PruebaHilo1(ThreadGroup g, String nombre) {
    super(g, nombre);
  }
  void f() {
    i++; // modificar este hilo
    System.out.println(getName() + " f()");
  }
}

class PruebaHilo2 extends PruebaHilo1 {
  PruebaHilo2(ThreadGroup g, String nombre) {
    super(g, nombre);
    start();
  }
  public void run() {
    ThreadGroup g =
      getThreadGroup().getParent().getParent();
    g.list();
    Thread[] gTodos = new Thread[g.activeCount()];
    g.enumerate(gAll);
    for(int i = 0; i < gTodos.length; i++) {
      gTodos[i].setPriority(Thread.MIN_PRIORITY);
      ((PruebaHilo1)gTodos[i]).f();
    }
    g.list();
  }
} ///:~
```

En el método **main( )** se crean varios **ThreadGroup**s, colgando unos de otros: **x** no tiene más parámetros que su nombre (un **String**), por lo que se ubica automáticamente en el grupo de hilos "del

sistema", mientras que **y** está bajo **x**, y **z** está bajo **y**. Nótese que la inicialización se da exactamente en el orden textual, por lo que este código es legal.

Se crean dos hilos y se ubican en distintos grupos de hilos. **PruebaHilo1** no tiene un método **run( )** pero tiene un **f( )** que modifica el hilo e imprime algo, de forma que pueda verse que fue invocado. **PruebaHilo2** es una subclase de **PruebaHilo1**, y su **run( )** está bastante elaborado. Primero toma el grupo de hilos del hilo actual, después asciende dos niveles por el árbol de herencia usando **getParent( )**. (Esto se hace así porque ubicamos el objeto **PruebaHilo2** dos niveles más abajo en la jerarquía a propósito.) En este momento, se crea un array de referencias a **Thread**s usando el método **activeCount( )** para preguntar cuántos hilos están en este grupo de hilos y en todos los grupos de hilos hijo. El método **enumerate( )** ubica referencias a todos estos hilos en el array **gTodos**, después simplemente recorremos todo el array invocando al método **f( )** de cada hilo, además de modificar la prioridad. Además, un hilo de un grupo de hilos "hoja" modifica los hilos en los grupos de hilos padre.

El método de depuración **list( )** imprime toda la información sobre un grupo de hilos en la salida estándar y es útil cuando se investiga el comportamiento de los grupos de hilos. He aquí la salida del programa:

```
java.lang.ThreadGroup[name=x,maxpri=10]
     Thread[uno,5,x]
     java.lang.ThreadGroup[name=y,maxpri=10]
          java.lang.ThreadGroup[name=z,maxpri=10]
               Thread[dos,5,x]
uno f()
dos f()
java.lang.ThreadGroup[name=x,maxpri=10]
     Thread[uno,1,x]
     java.lang.ThreadGroup[name=y,maxpri=10]
          java.lang.ThreadGroup[name=z,maxpri=10]
               Thread[dos,1,x]
```

El método **list( )** no sólo imprime el nombre de clase de **ThreadGroup** o **Thread**, sino que también imprime el nombre del grupo de hilos y su máxima prioridad. En el caso de los hilos, se imprime el nombre del hilo, seguido de la prioridad del hilo y el grupo al que pertenece. Nótese que **list( )** va indentando los hilos y grupos de hilos para indicar que son hijos del grupo no indentado.

Se puede ver que el método **run( )** de **PruebaHilo2** llama a **f( )**, por lo que es obvio que todos los hilos de un grupo sean vulnerables. Sin embargo, se puede acceder sólo a los hilos que se ramifican del propio árbol de grupos de hilos **sistema**, y quizás a esto hace referencia el término "seguridad". No se puede acceder a ningún otro árbol de grupos de hilos del sistema.

Controlar los grupos de hilos

Dejando de lado los aspectos de seguridad, algo para lo que los grupos de hilos parecen ser útiles es para controlar: se pueden llevar a cabo ciertas operaciones en todo un grupo de hilos con un único comando. El ejemplo siguiente lo demuestra, además de las restricciones de prioridades en los

grupos de hilos. Los números comentados entre paréntesis proporcionan una referencia para comparar la salida.

```
//: c14:GrupoHilos1.java
// Cómo los grupos de hilos controlan las prioridades
// de los hilos que contienen.

public class GrupoHilos1 {
  public static void main(String[] args) {
    // Lograr el hilo del sistema e imprimir su Info:
    ThreadGroup sis =
      Thread.currentThread().getThreadGroup();
    sis.list(); // (1)
    // Reducir la prioridad del grupo de hilos system:
    sis.setMaxPriority(Thread.MAX_PRIORITY - 1);
    // Incrementar la prioridad del hilo principal:
    Thread curr = Thread.currentThread();
    curr.setPriority(curr.getPriority() + 1);
    sis.list(); // (2)
    // Intentar poner un grupo a la prioridad max:
    ThreadGroup g1 = new ThreadGroup("g1");
    g1.setMaxPriority(Thread.MAX_PRIORITY);
    // Intentar poner un hilo nuevo a prioridad max:
    Thread t = new Thread(g1, "A");
    t.setPriority(Thread.MAX_PRIORITY);
    g1.list(); // (3)
    // Reducir la prioridad max de g1, después intentar
    // aumentarla:
    g1.setMaxPriority(Thread.MAX_PRIORITY - 2);
    g1.setMaxPriority(Thread.MAX_PRIORITY);
    g1.list(); // (4)
    // Intentar poner un hilo nuevo a prioridad max:
    t = new Thread(g1, "B");
    t.setPriority(Thread.MAX_PRIORITY);
    g1.list(); // (5)
    // Bajar la prioridad max por debajo de la prioridad
    // por defecto de los hilos:
    g1.setMaxPriority(Thread.MIN_PRIORITY + 2);
    // Mirar la prioridad de un hilo antes y
    // después de cambiarla:
    t = new Thread(g1, "C");
    g1.list(); // (6)
    t.setPriority(t.getPriority() -1);
    g1.list(); // (7)
    // Hacer que g2 sea un Threadgroup hijo de g1 e
```

```
    // intentar aumentar su prioridad:
    ThreadGroup g2 = new ThreadGroup(g1, "g2");
    g2.list(); // (8)
    g2.setMaxPriority(Thread.MAX_PRIORITY);
    g2.list(); // (9)
    // Añadir un conjunto de hilos nuevos a g2:
    for (int i = 0; i < 5; i++)
      new Thread(g2, Integer.toString(i));
    // Mostrar info sobre todos los grupos de hilos
    // e hilos:
    sis.list(); // (10)
    System.out.println("Comenzando todos los hilos:");
    Thread[] todos = new Thread[sis.activeCount()];
    sis.enumerate(todos);
    for(int i = 0; i < all.length; i++)
      if(!todos[i].isAlive())
        todos[i].start();
    // Suspende & Detiene todos los hilos de
    // este grupo y sus subgrupos:
    System.out.println("Todos los hilos arrancados");
    sis.suspend(); // Abolido en Java 2
    // Aquí nunca se llega...
    System.out.println("Todos los hilos suspendidos");
    sis.stop(); // Abolido en Java 2
    System.out.println("Todos los hilos detenidos");
  }
} ///:~
```

La salida que sigue se ha editado para permitir que entre en una página (se ha eliminado el **java.lang.**), y para añadir números que corresponden a los números en forma de comentario del listado de arriba.

```
(1)   ThreadGroup[name=system,maxpri=10]
        Thread[main,5,system]
(2)   ThreadGroup[name=system,maxpri=9]
        Thread[main,6,system]
(3)   ThreadGroup[name=g1,maxpri=9]
        Thread[A,9,g1]
(4)   ThreadGroup[name=g1,maxpri=8]
        Thread[A,9,g1]
(5)   ThreadGroup[name=g1,maxpri=8]
        Thread[A,9,g1]
        Thread[B,8,g1]
(6)   ThreadGroup[name=g1,maxpri=3]
        Thread[A,9,g1]
        Thread[B,8,g1]
```

```
        Thread[C,6,g1]
(7)   ThreadGroup[name=g1,maxpri=3]
        Thread[A,9,g1]
        Thread[B,8,g1]
        Thread[C,3,g1]
(8)   ThreadGroup[name=g2,maxpri=3]
(9)   ThreadGroup[name=g2,maxpri=3]
(10)  ThreadGroup[name=system,maxpri=9]
        Thread[main,6,system]
        ThreadGroup[name=g1,maxpri=3]
          Thread[A,9,g1]
          Thread[B,8,g1]
          Thread[C,3,g1]
          ThreadGroup[name=g2,maxpri=3]
            Thread[0,6,g2]
            Thread[1,6,g2]
            Thread[2,6,g2]
            Thread[3,6,g2]
            Thread[4,6,g2]
Comenzando todos los hilos
Todos los hilos arrancados
```

Todos los programas tienen al menos un hilo en ejecución, y lo primero que hace el método **main( )** es llamar al método **static** de **Thread** llamado **currentThread( )**. Desde este hilo, se produce el grupo de hilos y se llama a **list( )** para obtener el resultado. La salida es:

```
(1)   ThreadGroup[name=system,maxpri=10]
        Thread[main,5,system]
```

Puede verse que el nombre del grupo de hilos principal es **system**, y el nombre del hilo principal es **main**, y pertenece al grupo de hilos **system**.

El segundo ejercicio muestra que se puede reducir la prioridad máxima del grupo **system**, y que es posible incrementar la prioridad del hilo **main**:

```
(2)   ThreadGroup[name=system,maxpri=9]
        Thread[main,6,system]
```

El tercer ejercicio crea un grupo de hilos nuevo, **g1**, que pertenece automáticamente al grupo de hilos **system,** puesto que no se especifica nada más. Se ubica en **g1** un nuevo hilo **A**. Después de intentar establecer la prioridad máxima del grupo y la prioridad de **A** al nivel más alto el resultado es:

```
(3)   ThreadGroup[name=g1,maxpri=9]
        Thread[A,9,g1]
```

Por consiguiente, no es posible cambiar la prioridad máxima del grupo de hilos para que sea superior a la de su grupo de hilos padre.

El cuarto ejercicio reduce la prioridad máxima de **g1** por la mitad y después trata de incrementarla hasta **Thread.MAX_PRIORITY**. El resultado es:

```
(4)   ThreadGroup[name=g1,maxpri=8]
         Thread[A,9,g1]
```

Puede verse que no funcionó el incremento en la prioridad máxima. La prioridad máxima de un grupo de hilos sólo puede disminuirse, no incrementarse. También, nótese que la prioridad del hilo **A** no varió, y ahora es superior a la prioridad máxima del grupo de hilos. Cambiar la prioridad máxima de un grupo de hilos no afecta a los hilos existentes.

El quinto ejercicio intenta establecer como prioridad de un hilo la prioridad máxima:

```
(5)   ThreadGroup[name=g1,maxpri=8]
         Thread[A,9,g1]
         Thread[B,8,g1]
```

No se puede cambiar el nuevo hilo a nada superior a la prioridad máxima del grupo de hilos.

La prioridad por defecto para los hilos de este programa es seis; esa es la prioridad en la que se crearán los hilos nuevos y en la que éstos permanecerán mientras no se manipule su prioridad. El Ejercicio 6 disminuye la prioridad máxima del grupo de hilos por debajo de la prioridad por defecto para ver qué ocurre al crear un nuevo hilo bajo esta condición:

```
(6)   ThreadGroup[name=g1,maxpri=3]
         Thread[A,9,g1]
         Thread[B,8,g1]
         Thread[C,6,g1]
```

Incluso aunque la prioridad máxima del grupo de hilos es tres, el hilo nuevo se sigue creando usando la prioridad por defecto de seis. Por consiguiente, la prioridad máxima del grupo de hilos no afecta a la prioridad por defecto. (De hecho, parece no haber forma de establecer la prioridad por defecto de los hilos nuevos.)

Después de cambiar la prioridad, al intentar disminuirla en una unidad, el resultado es:

```
(7)   ThreadGroup[name=g1,maxpri=3]
         Thread[A,9,g1]
         Thread[B,8,g1]
         Thread[C,3,g1]
```

La prioridad máxima de los grupos de hilos sólo se ve reforzada al intentar cambiar la prioridad.

En (8) y (9) se hace un experimento semejante creando un nuevo grupo de hilos **g2** como hijo de **g1** y cambiando su prioridad máxima. Puede verse que es imposible que la prioridad máxima de **g2** sea superior a la de **g1**:

```
(8)   ThreadGroup[name=g2,maxpri=3]
(9)   ThreadGroup[name=g2,maxpri=3]
```

Nótese que la prioridad de **g2** se pone automáticamente en la prioridad máxima del grupo de hilos **g1** en el momento de su creación.

Después de todos estos experimentos, se imprime la totalidad del sistema de grupos de hilos e hilos:

```
(10) ThreadGroup[name=system,maxpri=9]
        Thread[main,6,system]
        ThreadGroup[name=g1,maxpri=3]
          Thread[A,9,g1]
          Thread[B,8,g1]
          Thread[C,3,g1]
          ThreadGroup[name=g2,maxpri=3]
            Thread[0,6,g2]
            Thread[1,6,g2]
            Thread[2,6,g2]
            Thread[3,6,g2]
            Thread[4,6,g2]
```

Por tanto, debido a las reglas de los grupos de hilos, un grupo hijo debe tener siempre una prioridad máxima inferior o igual a la prioridad máxima de su padre.

La última parte de este programa demuestra métodos para grupos completos de hilos. En primer lugar, el programa recorre todo el árbol de hilos y pone en marcha todos los que no hayan empezado. Después se suspende y finalmente se detiene el grupo **system**. (Aunque es interesante ver que **suspend( )** y **stop( )** afectan a todo el grupo de hilos, habría que recordar que estos métodos se han abolido en Java 2.) Pero cuando se suspende el grupo **system** también se suspende el hilo **main,** apagando todo el programa, por lo que nunca llega al punto en el que se detienen todos los hilos. De hecho, si no se detiene el hilo **main,** éste lanza una excepción **ThreadDeath**, lo cual no es lo más habitual. Puesto que **ThreadGroup** se hereda de **Object**, que contiene el método **wait( ),** también se puede elegir suspender el programa durante unos segundos invocando a **wait(segundos*1000)**. Por supuesto éste debe adquirir el bloqueo dentro de un bloqueo sincronizado.

La clase **ThreadGroup** también tiene métodos **suspend( )** y **resume( )** por lo que se puede parar y arrancar un grupo de hilos completo y todos sus hilos y subgrupos con un único comando. (De nuevo, **suspend( )** y **resume( )** están en desuso en Java 2.)

Los grupos de hilos pueden parecer algo misteriosos a primera vista, pero hay que tener en cuenta que probablemente no se usarán a menudo directamente.

Volver a visitar Runnable

Anteriormente en este capítulo, sugerimos que se pensara detenidamente antes de hacer un *applet* o un **Frame** principal como una implementación de **Runnable**. Por supuesto, si hay que heredar de una clase *y* se desea añadir comportamiento basado en hilos a la clase, la solución correcta es **Runnable**. El ejemplo final de este capítulo explota esto construyendo una clase **Runnable JPanel** que

pinta distintos colores por sí misma. Esta aplicación toma valores de la línea de comandos para determinar cuán grande es la rejilla de colores y cuán largo es el **sleep( )** entre los cambios de color. Jugando con estos valores se descubrirán algunas facetas interesantes y posiblemente inexplicables de los hilos:

```
//: c14:CajasColores.java
// Usando la interfaz Runnable.
// <applet code=CajasColores width=500 height=400>
// <param name=rejilla value="12">
// <param name=pausa value="50">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class CajaC extends JPanel implements Runnable {
  private Thread t;
  private int pausa;
  private static final Color[] colores = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color colorC = nuevoColor();
  private static final Color nuevoColor() {
    return colores[
      (int)(Math.random() * colores.length)
    ];
  }
  public void paintComponent(Graphics  g) {
    super.paintComponent(g);
    g.setColor(colorC);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  public CajaC(int pausa) {
    this.pausa = pausa;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    while(true) {
```

```
        colorC = nuevoColor();
        repaint();
        try {
          t.sleep(pause);
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
    }
  }

  public class CajasColores extends JApplet {
    private boolean esApplet = true;
    private int rejilla = 12;
    private int pausa = 50;
    public void init() {
      // Tomar los parámetros de la página Web:
      if (esApplet) {
        String tamanioR = getParameter("rejilla");
        if(tamanioR != null)
          rejilla = Integer.parseInt(tamanioR);
        String psa = getParameter("pausa");
        if(psa != null)
          pausa = Integer.parseInt(psa);
      }
      Container cp = getContentPane();
      cp.setLayout(new GridLayout(rejilla, rejilla));
      for (int i = 0; i < rejilla * rejilla; i++)
        cp.add(new CajaC(pausa));
    }
    public static void main(String[] args) {
      CajasColores applet = new CajasColores();
      applet.esApplet = false;
      if(args.length > 0)
        applet.rejilla = Integer.parseInt(args[0]);
      if(args.length > 1)
        applet.pausa = Integer.parseInt(args[1]);
      Console.run(applet, 500, 400);
    }
  } ///:~
```

CajasColores es la aplicación/*applet* habitual con un **init( )** que establece el IGU. Éste crea **GridLayout,** de forma que tenga celdas **rejilla** en cada dimensión. Después, añade el número apropiado de objetos **CajaC** para rellenar la rejilla, pasando el valor **pausa** a cada uno. En el método **main( )** pue-

de verse que **pausa** y **rejilla** tienen valores por defecto que pueden cambiarse si se pasan parámetros de línea de comandos, o usando parámetros del applet.

Todo el trabajo se da en **CajaC**. Ésta se hereda de **JPanel** e implementa la interfaz **Runnable** de forma que cada **JPanel** también puede ser un **Thread**. Recuérdese que al implementar **Runnable** no se hace un objeto **Thread,** sino simplemente una clase con un método **run( )**. Por consiguiente, un objeto **Thread** hay que crearlo explícitamente y pasarle el objeto **Runnable** al constructor, después llamar a **start( )** (esto ocurre en el constructor). En **CajaC** a este hilo se le denomina **t**.

Fijémonos en el array **colores** que es una enumeración de todos los colores de la clase **Color**. Se usa en **nuevoColor( )** para producir un color seleccionado al azar. El color de la celda actual es **celdaColor**.

El método **paintComponent( )** es bastante simple —simplemente pone el color a **color** y rellena todo el **JPanel** con ese color.

En **run( )** se ve el bucle infinito que establece el **Color** a un nuevo color al azar y después llama a **repaint( )** para mostrarlo. Después el hilo va a **sleep( )** durante la cantidad de tiempo especificada en la línea de comandos.

Precisamente porque este diseño es flexible y la capacidad de hilado está vinculada a cada elemento **JPanel**, se puede experimentar construyendo tantos hilos como se desee. (Realmente, hay una restricción impuesta por la cantidad de hilos que puede manejar cómodamente la JVM.)

Este programa también hace una medición interesante, puesto que puede mostrar diferencias de rendimiento drásticas entre una implementación de hilos de una JVM y otra.

Demasiados hilos

En algún momento, se verá que **CajasColores** se colapsa. En nuestra máquina esto ocurre en cualquier lugar tras una rejilla de 10 + 10. ¿Por qué ocurre esto?

Uno sospecha, naturalmente, que Swing debería estar relacionado con esto, por lo que hay un ejemplo que prueba esa premisa construyendo menos hilos. El siguiente código se ha reorganizado de forma que un **ArrayList implemente Runnable** y un **ArrayList** guarde un número de bloques de colores y elija al azar los que va a actualizar. Después, se crea un número de estos objetos **ArrayList**, dependiendo de la dimensión de la rejilla que se pueda elegir. Como resultado, se tienen bastantes menos hilos que bloques de color, por lo que si se produce un incremento de velocidad se sabrá que se debe a que hay menos hilos que en el ejemplo anterior:

```
//: c14:CajasColores2.java
// Balanceando el uso de hilos.
// <applet code=CajasColores2 width=600 height=500>
// <param name=rejilla value="12">
// <param name=pausa value="50">
// </applet>
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

class CajaC2 extends JPanel {
  private static final Color[] colores = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color colorC = nuevoColor();
  private static final Color nuevoColor() {
    return colores[
      (int)(Math.random() * colores.length)
    ];
  }
  void siguienteColor() {
    colorC = nuevoColor();
    repaint();
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(colorC);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
}

class ListaCajaC
  extends ArrayList implements Runnable {
  private Thread t;
  private int pausa;
  public ListaCajaC(int pausa) {
    this.pausa = pausa;
    t = new Thread(this);
  }
  public void comenzar() { t.start(); }
  public void run() {
    while(true) {
      int i = (int)(Math.random() * size());
      ((CajaC2)get(i)).nextColor();
      try {
        t.sleep(pausa);
```

```
        } catch(InterruptedException e) {
          System.err.println("Interrumpido");
        }
      }
    }
    public Object ultimo() { return get(size() - 1);}
}

public class CajasColores2 extends JApplet {
    private boolean esApplet = true;
    private int rejilla = 12;
    // Pausa por defecto más corta que CajasColores:
    private int pausa = 50;
    private ListaCajaC[] v;
    public void init() {
      // Tomar los parámetros de la página Web:
      if (esApplet) {
        String tamanioR = getParameter("rejilla");
        if(tamanioR != null)
          rejilla = Integer.parseInt(tamanioR);
        String psa = getParameter("pausa");
        if(psa != null)
          pausa = Integer.parseInt(psa);
      }
      Container cp = getContentPane();
      cp.setLayout(new GridLayout(rejilla, rejilla));
      v = new ListaCajaC[rejilla];
      for(int i = 0; i < rejilla; i++)
        v[i] = new ListaCajaC(pausa);
      for (int i = 0; i < rejilla * rejilla; i++) {
        v[i % rejilla].add(new CajaC2());
        cp.add((ultimo)v[i % rejilla].ultimo());
      }
      for(int i = 0; i < rejilla; i++)
        v[i].comenzar();
    }
    public static void main(String[] args) {
      CajaColores2 applet = new CajaColores2();
      applet.esApplet = false;
      if(args.length > 0)
        applet.rejilla = Integer.parseInt(args[0]);
      if(args.length > 1)
        applet.pausa = Integer.parseInt(args[1]);
      Console.run(applet, 500, 400);
    }
```

```
} ///:~
```

En **CajaColores2** se crea un array de **ListaCajaC** inicializándose para guardar la **rejilla ListasCajaC**, cada uno de los cuales sabe durante cuánto tiempo dormir. Posteriormente se añade un número igual de objetos **CajaC2** a cada **ListaCajaC**, y se dice a cada lista que **comenzar( )**, lo cual pone en marcha el hilo.

CajaC2 es semejante a **CajaC**: se pinta ListaCajaC a sí misma con un color elegido al azar. Pero esto es *todo* lo que hace un **CajaC2**. Toda la gestión de hilos está ahora en **ListaCajaC**.

ListaCajaC también podría haber heredado **Thread** y haber tenido un objeto miembro de tipo **ArrayList**. Ese diseño tiene la ventaja de que los métodos **add( )** y **get( )** podrían recibir posteriormente un argumento específico y devolver tipos de valores en vez de **Objects** genéricos. (También se podrían cambiar sus nombres para que sean más cortos.) Sin embargo, el diseño usado aquí parecía a primera vista requerir menos código. Además, retiene automáticamente todos los demás comportamientos de un **ArrayList**. Con todas las conversiones y paréntesis necesarios para **get( )**, éste podría no ser el caso a medida que crece el cuerpo del código.

Como antes, al implementar **Runnable** no se logra todo el equipamiento que viene con **Thread**, por lo que hay que crear un nuevo **Thread** y pasárselo explícitamente a su constructor para tener algo en **start( )**, como puede verse en el constructor **ListaCajaC** y en **comenzar( )**. El método **run( )** simplemente elige un número de elementos al azar dentro de la lista y llama al **siguienteColor( )** de ese elemento para que elija un nuevo color seleccionado al azar.

Al ejecutar este programa se ve que, de hecho, se ejecuta más rápido y responde más rápidamente (por ejemplo, cuando es interrumpido, se detiene más rápidamente) y no parece saturarse tanto en tamaños de rejilla grandes. Por consiguiente, se añade un nuevo factor a la ecuación de hilado: hay que vigilar para ver que no se tengan "demasiados hilos" (sea lo que sea lo que esto signifique para cada programa y plataforma en particular —aquí, la ralentización de **CajasColores** parece estar causada por el hecho de que sólo hay un hilo que es responsable de todo el pintado, y que se colapsa por demasiadas peticiones). Si se tienen demasiados hilos hay que intentar usar técnicas como la de arriba para "equilibrar" el número de hilos del programa. Si se aprecian problemas de rendimiento en un programa multihilo, hay ahora varios aspectos que examinar:

1. ¿Hay suficientes llamadas a **sleep( )**, **yield( )** y/o **wait( )**?
2. ¿Son las llamadas a **sleep( )** lo suficientemente rápidas?
3. ¿Se están ejecutando demasiados hilos?
4. ¿Has probado distintas plataformas y JVMs?

Aspectos como éste son la razón por la que a la programación multihilo se le suele considerar un arte.

Resumen

Es vital aprender cuándo hacer uso de capacidades multihilo y cuándo evitarlas. La razón principal de su uso es gestionar un número de tareas que al entremezclarse hagan un uso más eficiente del ordenador (incluyendo la habilidad de distribuir transparentemente las tareas a través de múltiples UCP), o ser más conveniente para el usuario. El ejemplo clásico de balanceo de recursos es usar la UCP durante las esperas de E/S. El ejemplo clásico de conveniencia del usuario es monitorizar un botón de "detención" durante descargas largas.

Las desventajas principales del multihilado son:

1. Ralentización durante la espera por recursos compartidos.
2. Sobrecarga adicional de la UCP necesaria para gestionar los hilos.
3. Complejidad sin recompensa, como la tonta idea de tener un hilo separado para actualizar cada elemento de un array.
4. Patologías que incluyen la inanición, la competición y el interbloqueo.

Una ventaja adicional de los hilos es que sustituyen a las conmutaciones de contexto de ejecución "ligera" (del orden de 100 instrucciones) por conmutaciones de contexto de ejecución "pesada" (del orden de miles de instrucciones). Puesto que todos los hilos de un determinado proceso comparten el mismo espacio de memoria, una conmutación de proceso ligera sólo cambia la ejecución del programa y las variables locales. Por otro lado, un cambio de proceso —la conmutación de contexto pesada— debe intercambiar todo el espacio de memoria.

El multihilado es como irrumpir paso a paso en un mundo completamente nuevo y aprender un nuevo lenguaje de programación o al menos un conjunto de conceptos de lenguaje nuevos. Con la apariencia de soporte a hilos, en la mayoría de sistemas operativos de microcomputador han ido apareciendo extensiones para hilos en lenguajes de programación y bibliotecas. En todos los casos, la programación de hilos (1) parece misteriosa y requiere un cambio en la forma de pensar al programar; y (2) parece similar al soporte de hilos en otros lenguajes, por lo que al entender los hilos se entiende una lengua común. Y aunque el soporte de hilos puede hacer que Java parezca un lenguaje más complicado, no hay que echar la culpa a Java. Los hilos son un truco.

Una de las mayores dificultades con los hilos se debe a que, dado que un recurso —como la memoria de un objeto— podría estar siendo compartido por más de un hilo, hay que asegurarse de que múltiples hilos no intenten leer y cambiar ese recurso simultáneamente. Esto requiere de un uso juicioso de la palabra clave **synchronized**, que es una herramienta útil pero que debe ser totalmente comprendida puesto que puede presentar silenciosamente situaciones de interbloqueos.

Además, hay determinado arte en la aplicación de los hilos. Java está diseñado para permitir la creación de tantos objetos como se necesite para solucionar un problema —al menos en teoría. (Crear millones de objetos para un análisis de elementos finitos de ingeniería, por ejemplo, podría no ser práctico en Java.) Sin embargo, parece que hay un límite superior al número de hilos a crear, puesto que en algún momento, un número de hilos más elevado da muestras de colapso. Este pun-

to crítico no se alcanza con varios miles, como en el caso de los objetos, sino en unos pocos cientos, o incluso a veces menos de 1.200. Como a menudo sólo se crea un puñado de hilos para solucionar un problema, este límite no suele ser tal, aunque puede parecer una limitación en diseños generales.

Un aspecto significativo y no intuitivo de los hilos es que, debido a la planificación de los hilos, se puede hacer que una aplicación se ejecute generalmente *más rápido* insertando llamadas a **sleep( )** dentro del bucle principal de **run( )**. Esto hace, sin duda, que su uso parezca un arte, especialmente cuando los retrasos más largos parecen incrementar el rendimiento. Por supuesto, la razón por la que ocurre esto es que retrasos más breves pueden hacer que la interrupción del planificador del final del **sleep( )** se dé antes de que el hilo en ejecución esté listo para ir a dormir, forzando al planificador a detenerlo y volver a arrancarlo más tarde para que pueda acabar con lo que estaba haciendo, para ir después a dormir. Hay que pensar bastante para darse cuenta en lo complicadas que pueden ser las cosas.

Algo que alguien podría echar de menos en este capítulo es un ejemplo de animación, que es una de las cosas más populares que se hacen con los *applets*. Sin embargo, con el Java JDK (disponible en *http://java.sun.com*) viene la solución completa (con sonido) a este problema, dentro de la sección de demostración. Además, se puede esperar que en las versiones futuras de Java se incluya un mejor soporte para animaciones, a la vez que están apareciendo distintas soluciones de animación para la Web, no Java, y que no son de programación, que pueden ser superiores a los enfoques tradicionales. Si se desean explicaciones de cómo funcionan las animaciones en Java, puede verse *Core Java 2*, de Horstmann & Cornell, Prentice-Hall, 1997. Para acceder a discusiones más avanzadas en el multihilado, puede verse *Concurrent Programming in Java*, de Doug Lea, Addison-Wesley, 1997, o *Java Threads* de Oaks & Wong, O'Reilly, 1997.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Heredar una clase de **Thread** y superponer el método **run( )**. Dentro de **run( )**, imprimir un mensaje y llamar después a **sleep( )**. Repetir esto tres veces, después de volver de **run( )** finalice. Poner un mensaje de empiece en el constructor y superponer **finalize( )** para imprimir un mensaje de apagado. Hacer una clase hilo separada que llame a **System.gc( )** y **System.runFinalization( )** dentro de **run( )**, imprimiendo un mensaje a medida que lo hace. Hacer varios objetos hilo de ambos tipos y ejecutarlos para ver qué ocurre.

2. Modificar **Compartiendo2.java** para añadir un bloque **synchronized** dentro del método **run ( )** de **DosContadores** en vez de sincronizar todo el método **run( )**.

3. Crear dos subclases **Thread**, una con un **run( )** que empiece, capture la referencia al segundo objeto **Thread** y llame después a **wait( )**. El método **run( )** de la otra clase debería llamar a **notifyAll( )** para el primer hilo, tras haber pasado cierto número de segundos, de forma que el primer hilo pueda imprimir un mensaje.

4. En **Contador5.java** dentro de **Teletipo2**, retirar el **yield( )** y explicar los resultados. Reemplazar el **yield( )** con un **sleep( )** y explicar los resultados.

5. En **grupoHilos.java**, reemplazar la llamada a **sis.suspend( )** con una llamada a **wait( )** para el grupo de hilos, haciendo que espere durante dos segundos. Para que esto funcione correctamente hay que adquirir el bloqueo de **sis** dentro de un bloque **synchronized**.

6. Cambiar **Demonios.java,** de forma que **main( )** tenga un **sleep( )** en vez de un **readLine ( )**. Experimentar con distintos tiempos en la **sleep( )** para ver qué ocurre.

7. En el Capítulo 8, localizar el ejemplo **ControlesInvernadero.java**, que consiste en tres archivos. En **Evento.java**, la clase **Evento** se basa en vigilar el tiempo. Cambiar **Evento** de forma que sea un **Thread**, y cambiar el resto del diseño de forma que funcione con este nuevo **Evento** basado en **Thread**.

8. Modificar el Ejercicio 7, de forma que se use la clase **java.util.Timer** del JDK 1.3 para ejecutar el sistema.

9. A partir de **OndaSeno.java** del Capítulo 13, crear un programa (un applet/aplicación usando la clase **Console**) que dibuje una onda seno animada que parezca desplazarse por la ventana como si fuera un osciloscopio, dirigiendo la animación con un **Thread**. La velocidad de la animación debería controlarse con un control **java.swing.JSlider**.

10. Modificar el Ejercicio 9, de forma que se creen varios paneles onda seno dentro de la aplicación. El número de paneles debería controlarse con etiquetas HTML o parámetros de línea de comando.

11. Modificar el Ejercicio 9, de forma que se use la clase **java.swing.Timer** para dirigir la animación. Nótese la diferencia entre éste y **java.util.Timer**.