2.1. INTRODUCCIÓN

En el capítulo anterior se estudió la programación concurrente y cómo se podían realizar programas concurrentes con el lenguaje Java. Se hizo una breve introducción al concepto de hilo y las diferencias entre estos y los procesos.

Recordemos que los hilos comparten el espacio de memoria del usuario, es decir, corren dentro del contexto de otro programa; y los procesos generalmente mantienen su propio espacio de direcciones y entorno de operaciones. Por ello a los hilos se les conoce a menudo como **procesos ligeros**.

En este capítulo usaremos los hilos en Java para realizar programas concurrentes.

2.2. QUÉ SON LOS HILOS

Un **hilo** (hebra, *thread* en inglés) es una secuencia de código en ejecución dentro del contexto de un proceso. Los hilos no pueden ejecutarse ellos solos, necesitan la supervisión de un proceso padre para ejecutarse. Dentro de cada proceso hay varios hilos ejecutándose. La Figura 2.1 muestra la relación entre hilos y procesos.

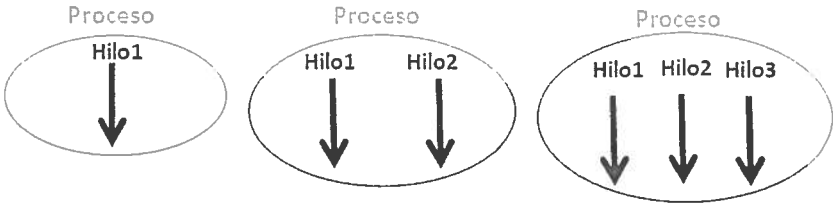

Figura 2.1. Relación entre hilos y procesos.

Podemos usar los hilos para diferentes aplicaciones: para realizar programas que tengan que realizar varias tareas simultáneamente, en los que la ejecución de una parte requiera tiempo y no deba detener el resto del programa. Por ejemplo, un programa que controla sensores en una fábrica, cada sensor puede ser un hilo independiente y recoge un tipo de información; y todos deben controlarse de forma simultánea. Un programa de impresión de documentos debe seguir funcionando, aunque se esté imprimiendo un documento, tarea que se puede llevar por medio de un hilo. Un programa procesador de textos puede tener un hilo comprobando la gramática del texto que estoy escribiendo y otro hilo guardando el texto en disco cada cierto tiempo. En un programa de bases de datos un hilo pinta la interfaz gráfica al usuario. En un servidor web, un hilo puede atender las peticiones entrantes y crear un hilo por cada cliente que tenga que servir.

2.3. CLASES PARA LA CREACIÓN DE HILOS

En Java existen dos formas para crear hilos: extendiendo la clase **Thread** o implementando la interfaz **Runnable**. Ambas son parte del paquete **java.lang**.

2.3.1. La clase THREAD

La forma más simple de añadir funcionalidad de hilo a una clase es extender la clase **Thread**. O lo que es lo mismo crear una subclase de la clase **Thread**. Esta subclase debe sobrescribir el método **run()** con las acciones que el hilo debe desarrollar. La clase **Thread** define también los métodos **start()** y **stop()** (actualmente en desuso) para iniciar y parar la ejecución del hilo. La forma general de declarar un hilo extendiendo **Thread** es la siguiente:

```
class NombreHilo extends Thread {
      //propiedades, constructores y métodos de la clase
      public void run() {
```

```
        //acciones que lleva a cabo el hilo
    }
}
```

Para crear un objeto hilo con el comportamiento de *NombreHilo* escribo:

```
NombreHilo h = new NombreHilo();
```

Y para iniciar su ejecución utilizamos el método **start()**:

```
h.start();
```

El siguiente ejemplo declara la clase *PrimerHilo* que extiende la clase **Thread**, desde el constructor se inicializa una variable numérica que se usará para pintar un número de veces un mensaje; en el método **run()** se escribe la funcionalidad del hilo:

```
public class PrimerHilo extends Thread {
   private int x;
   PrimerHilo (int x)
   {
     this.x=x;
   }

   public void run() {
     for (int i=0; i<x; i++)
       System.out.println("En el Hilo... "+i);
   }

}//PrimerHilo
```

A continuación, para crear un objeto hilo escribimos:

```
PrimerHilo p = new PrimerHilo (10);
```

Y para iniciar su ejecución:

```
p.start();
```

Dentro de la clase anterior podemos añadir el método **main()** para crear el hilo e iniciar su ejecución:

```
public static void main(String[] args) {
     PrimerHilo p = new PrimerHilo(10);
     p.start();
}// main
```

En el siguiente ejemplo se crea una clase que extiende **Thread**. Dentro de la clase se definen el constructor, el método **run()** con la funcionalidad que realizará el hilo y el método **main()** donde se crearán 3 hilos. La misión del hilo, descrita en el método **run()**, será visualizar un mensaje donde se muestre el nombre del hilo que se está ejecutando y el contenido de un contador. Se utiliza una variable para mostrar el nombre del hilo que se ejecuta, esta variable se pasa al constructor y éste se lo pasa al constructor de la clase base **Thread** mediante la palabra reservada **super**, para acceder a este nombre se usa el método **getName()**. Desde el método **main()** se crean los hilos y para iniciar cada hilo usamos el método **start()**:

```
public class HiloEjemplo1 extends Thread {
      //constructor
      public HiloEjemplo1(String nombre) {
            super(nombre);
            System.out.println("CREANDO HILO:" + getName());
      }
```

```
    // método run
    public void run() {
        for (int i=0; i<5; i++)
          System.out.println("Hilo:" + getName() + " C = " + i);
    }

    //
    public static void main(String[] args) {
          HiloEjemplo1 h1 = new HiloEjemplo1("Hilo 1");
          HiloEjemplo1 h2 = new HiloEjemplo1("Hilo 2");
          HiloEjemplo1 h3 = new HiloEjemplo1("Hilo 3");

          h1.start();
          h2.start();
          h3.start();

          System.out.println("3 HILOS INICIADOS...");
    }// main

}// HiloEjemplo1
```

Es muy típico ver dentro del método **run()** un bucle infinito de forma que el hilo no termina nunca (más adelante veremos cómo detener el hilo). La ejecución del ejemplo anterior no siempre muestra la misma salida, en este caso se puede observar que los hilos no se ejecutan en el orden en que se crean:

```
CREANDO HILO:Hilo 1
CREANDO HILO:Hilo 2
CREANDO HILO:Hilo 3
3 HILOS INICIADOS...
Hilo: Hilo 1 C = 0
Hilo: Hilo 1 C = 1
Hilo: Hilo 1 C = 2
Hilo: Hilo 1 C = 3
Hilo: Hilo 1 C = 4
Hilo: Hilo 3 C = 0
Hilo: Hilo 2 C = 0
Hilo: Hilo 3 C = 1
Hilo: Hilo 3 C = 2
Hilo: Hilo 3 C = 3
Hilo: Hilo 3 C = 4
Hilo: Hilo 2 C = 1
Hilo: Hilo 2 C = 2
Hilo: Hilo 2 C = 3
Hilo: Hilo 2 C = 4
```

En este ejemplo se ha incluido el método **main()** dentro de la clase hilo. Podemos definir por un lado la clase hilo y por otro la clase que usa el hilo, tendríamos dos clases, la que extiende **Thread**, *HiloEjemplo1_V2.java*:

```
public class HiloEjemplo1_V2 extends Thread {
      // constructor
      public  HiloEjemplo1_V2(String nombre) {
            super(nombre);
            System.out.println("CREANDO HILO:" + getName());
      }
      // método run
```

```
    public void run() {
      for (int i=0; i<5; i++)
          System.out.println("Hilo:" + getName() + " C = " + i);
    }
}// HiloEjemplo1_V2
```

Y la clase que usa el hilo *UsaHiloEjemplo1_V2.java*:

```
public class UsaHiloEjemplo1_V2  {
    public static void main(String[] args) {
      HiloEjemplo1 h1 = new HiloEjemplo1("Hilo 1");
      HiloEjemplo1 h2 = new HiloEjemplo1("Hilo 2");
      HiloEjemplo1 h3 = new HiloEjemplo1("Hilo 3");

      h1.start();
      h2.start();
      h3.start();

      System.out.println("3 HILOS INICIADOS...");
   }
}//UsaHiloEjemplo1_V2
```

Se compila primero la clase hilo y después la que usa el hilo, se ejecuta la clase que usa el hilo:

```
D:\CAPIT2>javac HiloEjemplo1_V2.java
D:\CAPIT2>javac UsaHiloEjemplo1_V2.java
D:\CAPIT2>java UsaHiloEjemplo1_V2
```

En la siguiente tabla se muestran algunos métodos útiles sobre los hilos, algunos ya se han usado:

MÉTODOS	MISIÓN
start()	Hace que el hilo comience la ejecución; la máquina virtual de Java llama al método **run()** de este hilo.
boolean isAlive()	Comprueba si el hilo está vivo
sleep(long mils)	Hace que el hilo actualmente en ejecución pase a dormir temporalmente durante el número de milisegundos especificado. Puede lanzar la excepción *InterruptedException*.
run()	Constituye el cuerpo del hilo. Es llamado por el método **start()** después de que el hilo apropiado del sistema se haya inicializado. Si el método **run()** devuelve el control, el hilo se detiene. Es el único método de la interfaz **Runnable**.
String toString()	Devuelve una representación en formato cadena de este hilo, incluyendo el nombre del hilo, la prioridad, y el grupo de hilos. Ejemplo: Thread[HILO1,2,main]
long getId()	Devuelve el identificador del hilo.
void yield()	Hace que el hilo actual de ejecución pare temporalmente y permita que otros hilos se ejecuten.
String getName()	Devuelve el nombre del hilo.
setName(String name)	Cambia el nombre de este hilo, asignándole el especificado como argumento.

MÉTODOS	MISIÓN
int getPriority()	Devuelve la prioridad del hilo.
setPriority(int p)	Cambia la prioridad del hilo al valor entero p.
void interrupt()	Interrumpe la ejecución del hilo
boolean interrupted()	Comprueba si el hilo actual ha sido interrumpido.
Thread currentThread()	Devuelve una referencia al objeto hilo que se está ejecutando actualmente.
boolean isDaemon()	Comprueba si el hilo es un hilo Daemon. Los hilos daemon o demonio son hilos con prioridad baja que normalmente se ejecutan en segundo plano. Un ejemplo de hilo demonio que está ejecutándose continuamente es el recolector de basura (*garbage collector*).
setDaemon(boolean on)	Establece este hilo como hilo Daemon, asignando el valor *true*, o como hilo de usuario, pasando el valor *false*.
void stop()	Detiene el hilo. Este método está en desuso.
Thread currentThread()	Devuelve una referencia al objeto hilo actualmente en ejecución.
int activeCount()	Este método devuelve el número de hilos activos en el grupo de hilos del hilo actual.
Thread.State getState()	Devuelve el estado del hilo: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED

En la URL *https://docs.oracle.com/javase/10/docs/api/java/lang/Thread.html* se puede consultar más información sobre todos estos métodos.

El siguiente ejemplo muestra el uso de algunos de los métodos anteriores:

```
public class HiloEjemplo2 extends Thread {
  public void run() {
     System.out.println(
       "Dentro del Hilo  : " + Thread.currentThread().getName() +
       "\n\tPrioridad    : " + Thread.currentThread().getPriority() +
       "\n\tID           : " + Thread.currentThread().getId() +
       "\n\tHilos activos: " + Thread.currentThread().activeCount());
  }
  //
  public static void main(String[] args) {
      Thread.currentThread().setName("Principal");//nombre a main
      System.out.println(Thread.currentThread().getName());
      System.out.println(Thread.currentThread().toString());

      HiloEjemplo2 h = null;

      for (int i = 0; i < 3; i++) {
        h = new HiloEjemplo2(); //crear hilo
        h.setName("HILO"+i);    //damos nombre al hilo
        h.setPriority(i+1);     //damos prioridad
        h.start();              //iniciar hilo
        System.out.println(
            "Información del " + h.getName() + ": " + h.toString());
      }
```

```
        System.out.println("3 HILOS CREADOS...");
        System.out.println("Hilos activos: " + Thread.activeCount());

    }//

}// HiloEjemplo2
```

La ejecución muestra la siguiente salida (que puede variar de una ejecución a otra), en la que podemos observar que el método **toString()** devuelve un string que representa al hilo: *Thread[nombre del hilo, la prioridad, grupo de hilos]*, el método **currentThread()** que devuelve una referencia al objeto hilo actualmente en ejecución y **activeCount()** que devuelve el número de hilos activos actualmente dentro del grupo:

```
Principal
Thread[Principal,5,main]
Informacion del HILO0: Thread[HILO0,1,main]
Informacion del HILO1: Thread[HILO1,2,main]
Dentro del Hilo  : HILO0
      Prioridad    : 1
      ID           : 10
      Hilos activos: 3
Informacion del HILO2: Thread[HILO2,3,main]
3 HILOS CREADOS...
Hilos activos: 4
Dentro del Hilo  : HILO2
      Prioridad    : 3
      ID           : 12
      Hilos activos: 3
Dentro del Hilo  : HILO1
      Prioridad    : 2
      ID           : 11
      Hilos activos: 2
```

Todo hilo de ejecución en Java debe formar parte de un grupo. Por defecto, si no se especifica ningún grupo en el constructor, los hilos serán miembros del grupo **main**, que es creado por el sistema cuando arranca la aplicación Java.

La clase **ThreadGroup** se utiliza para manejar grupos de hilos en las aplicaciones Java. La clase **Thread** proporciona constructores en los que se puede especificar el grupo del hilo que se está creando en el mismo momento de instanciarlo. El siguiente ejemplo crea un grupo de hilos de nombre *Grupo de hilos*. A continuación, crea tres hilos usando el siguiente constructor de la clase **Thread**:

```
Thread (grupo ThreadGroup, destino Runnable, nombre String)
```

En el que se especifica el grupo de hilos, el objeto hilo y el nombre del hilo. El código es el siguiente:

```
public class HiloEjemplo2Grupos extends Thread {
  public void run() {
      System.out.println("Informacion del hilo: " +
                         Thread.currentThread().toString());
      for (int i = 0; i < 1000; i++) i++;
      System.out.println(Thread.currentThread().getName() +
                " Finalizando la ejecución.");
  }
  //
```

```
  public static void main(String[] args) {
      Thread.currentThread().setName("Principal");
      System.out.println(Thread.currentThread().getName());
      System.out.println(Thread.currentThread().toString());

      ThreadGroup grupo = new ThreadGroup("Grupo de hilos");
      HiloEjemplo2Grupos h = new HiloEjemplo2Grupos();

      Thread h1 = new Thread(grupo, h, "Hilo 1");
      Thread h2 = new Thread(grupo, h, "Hilo 2");
      Thread h3 = new Thread(grupo, h, "Hilo 3");

      h1.start();
      h2.start();
      h3.start();

      System.out.println("3 HILOS CREADOS...");
      System.out.println("Hilos activos: " + Thread.activeCount());
  }
}// HiloEjemplo2Grupos
```

La ejecución muestra la siguiente salida:

```
Principal
Thread[Principal,5,main]
3 HILOS CREADOS...
Hilos activos: 4
Informacion del hilo: Thread[Hilo 1,5,Grupo de hilos]
Informacion del hilo: Thread[Hilo 2,5,Grupo de hilos]
Hilo 1 Finalizando la ejecución.
Hilo 2 Finalizando la ejecución.
Informacion del hilo: Thread[Hilo 3,5,Grupo de hilos]
Hilo 3 Finalizando la ejecución.
```

ACTIVIDAD 2.1

Crea dos clases (hilos) Java que extiendan la clase **Thread**. Uno de los hilos debe visualizar en pantalla en un bucle infinito la palabra TIC y el otro hilo la palabra TAC. Dentro del bucle utiliza el método **sleep()** para que nos de tiempo a ver las palabras que se visualizan cuando lo ejecutemos, tendrás que añadir un bloque **try-catch** (para capturar la excepción *InterruptedException*). Crea después la función *main()* que haga uso de los hilos anteriores. ¿Se visualizan los textos TIC y TAC de forma ordenada (es decir TIC TAC TIC TAC ...)?

Realiza el Ejercicio 1.

2.3.2. La interfaz RUNNABLE

Para añadir la funcionalidad de hilo a una clase que deriva de otra clase (por ejemplo, un applet), siendo esta distinta de **Thread**, se utiliza la interfaz **Runnable**. Esta interfaz añade la funcionalidad de hilo a una clase con solo implementarla. Por ejemplo, para añadir la funcionalidad de hilo a un applet definimos la clase como:

```
public class Reloj extends Applet implements Runnable {}
```

La interfaz **Runnable** proporciona un único método, el método **run()**. Este es ejecutado por el objeto hilo asociado. La forma general de declarar un hilo implementando la interfaz **Runnable** es la siguiente:

```
class NombreHilo implements Runnable {
      //propiedades, constructores y métodos de la clase
      public void run() {
         //acciones que lleva a cabo el hilo
      }
}//
```

Para crear un objeto hilo con el comportamiento de *NombreHilo* escribo lo siguiente:

```
NombreHilo h = new NombreHilo();
```

Y para iniciar su ejecución utilizamos el método **start()**:

```
new Thread(h).start();
```

O bien para lanzar el hilo escribimos lo anterior en dos pasos:

```
Thread h1 = new Thread(h).

h1.start();
```

O en un paso todo:

```
new Thread(new NombreHilo()).start();
```

El siguiente ejemplo declara la clase *PrimerHiloR* que implementa la interfaz **Runnable**, en el método **run()** se indica la funcionalidad del hilo, en este caso es pintar un mensaje y visualizar el identificador del hilo actualmente en ejecución:

```
public class PrimerHiloR implements Runnable {
   public void run() {
      System.out.println("Hola desde el Hilo! " +
                         Thread.currentThread().getId());
   }
}//PrimerHiloR
```

A continuación, se muestra la clase *UsaPrimerHiloR.java* donde se lanzan varios hilos del tipo anterior de distintas formas:

```
public class UsaPrimerHiloR {
   public static void main(String[] args) {
      //Primer hilo
      PrimerHiloR hilo1 = new PrimerHiloR();
      new Thread(hilo1).start();

      //Segundo hilo
      PrimerHiloR hilo2 = new PrimerHiloR();
      Thread hilo = new Thread(hilo2);
      hilo.start();

      //Tercer Hilo
      new Thread(new PrimerHiloR()).start();
   }
}//UsaPrimerHiloR
```

ACTIVIDAD 2.2

Transforma el Ejercicio 1 usando la interfaz **Runnable** para declarar el hilo. Después realiza el programa Java que pide el enunciado del ejercicio.

Realiza el Ejercicio 2.

Seguidamente vamos a ver cómo usar un hilo en un applet para realizar una tarea repetitiva, en el ejemplo la tarea será mostrar la hora con los minutos y segundos: HH:MM:SS; véase Figura 2.2; normalmente la tarea repetitiva se encierra en un bucle infinito. Un applet es una aplicación Java que se puede insertar en una página web; cuando el navegador carga la página, el applet se carga y se ejecuta. Nuestro applet implementará la interfaz **Runnable**, por tanto debe incluir el método **run()** con la tarea repetitiva.

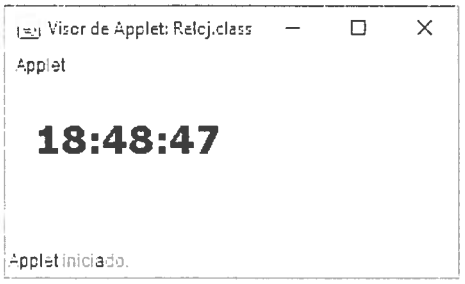

Figura 2.2. Applet Reloj.java.

Hemos de tener en cuenta que al utilizar applet en versiones de Java 10 y superiores se muestra una advertencia indicando que *la API de applet y AppletViewer están en desuso*.

En un applet se definen varios métodos:

- ***init()***: con instrucciones para inicializar el applet, este método es llamado una vez cuando se carga el applet.
- ***start()***: parecido a ***init()*** pero con la diferencia de que es llamado cuando se reinicia el applet.
- ***paint()***: que se encarga de mostrar el contenido del applet; se ejecuta cada vez que se tenga que redibujar.
- ***stop()***: es invocado al ocultar el applet, se utiliza para detener hilos.

El navegador web llama primero al método ***init()***, luego a ***paint()*** y a continuación al método start(). El hilo lo crearemos dentro del método ***start()*** usamos la siguiente expresión:

```
hilo = new Thread(this);
```

Al especificar *this* en la sentencia *new Thread()* se indica que el applet proporciona el cuerpo del hilo.

La estructura general de un applet que implementa **Runnable** es la siguiente:

```
import java.awt.*;
import java.applet.*;
public class AppletThread extends Applet implements Runnable {
      private Thread hilo = null;
      public void init() {
      }
      public void start() {
            if (hilo == null) {
                  // crea el hilo
                  hilo = new Thread(this);
                  hilo.start(); // lanza el hilo
            }
      }
      public void run() {
            Thread hiloActual = Thread.currentThread();
```

```
            while (hilo == hiloActual) {
             // tarea repetitiva
            }
      }
      public void stop() {
            hilo = null;
      }
      public void paint(Graphics g) {
      }
}
```

Cuando el applet necesita matar el hilo le asigna el valor *null*, esta acción se realiza en el método ***stop()*** del applet (se recomienda en los applets que implementan **Runnable**). Es una forma más suave de detener el hilo que utilizar el método **stop()** del hilo (*hilo.stop()*), ya que este puede resultar peligroso. El código del método **run()** es el siguiente:

```
public void run() {
            Thread hiloActual = Thread.currentThread();
            while (hilo == hiloActual) {
             // tarea repetitiva
            }
}
```

Donde se comprueba cual es el hilo actual con la expresión *Thread.currentThread()*; el proceso continúa o no dependiendo del valor de la variable del hilo; si la variable apunta al mismo hilo que está actualmente en ejecución el proceso continúa; si es *null* el proceso finaliza y si la variable hace referencia a otro hilo es que ha ocurrido una extraña situación, la tarea repetitiva no se ejecutará. Aplicamos la estructura anterior a nuestro applet, *Reloj.java*, que muestra la hora:

```
import java.applet.*;
import java.awt.*;
import java.text.SimpleDateFormat;
import java.util.*;

public class Reloj extends Applet implements Runnable {
      private Thread hilo = null; //hilo
      private Font fuente; //tipo de letra para la hora
      private String horaActual = "";

      public void init() {
            fuente = new Font("Verdana", Font.BOLD, 26);
            setBackground(Color.yellow); //color de fondo
            setFont(fuente); //fuente
      }

      public void start() {
            if (hilo == null) {
                  hilo = new Thread(this);
                  hilo.start();
            }
      }
      public void run() {
          Thread hiloActual = Thread.currentThread();
          while (hilo == hiloActual) {
            SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");
            Calendar cal = Calendar.getInstance();
            horaActual = sdf.format(cal.getTime());
            repaint(); //actualizar contenido del applet
```

```
            try {
                  Thread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }
    public void paint(Graphics g) {
          g.clearRect(1, 1, getSize().width, getSize().height);
          g.drawString(horaActual, 20, 50);//muestra la hora
    }
    public void stop() {
          hilo = null;
    }
}
```

El método ***repaint()*** se utiliza para actualizar el applet cuando cambian las imágenes contenidas en él. Los applets no tienen método *main()*, para ejecutarlos necesitamos crear un fichero HTML, por ejemplo creamos el fichero *Reloj.html* con el siguiente contenido:

```
<html>
    <applet code="Reloj.class" width="200" height="100">
    </applet>
</html>
```

Desde un entorno gráfico como Eclipse no sería necesario crear el HTML. Para compilarlo y ejecutarlo desde la línea de comandos usamos el comando *appletviewer*, se visualizará una ventanita similar a la de la Figura 2.2:

```
D:\CAPIT2>javac Reloj.java
D:\CAPIT2>appletviewer Reloj.html
```

Se usan las clases **Calendar** y **SimpleDateFormat** para obtener la hora y darle formato. En el método ***paint()*** del applet se han utilizado los siguientes métodos:

- ***clearRect (int x, int y, int ancho, int alto)***: borra el rectángulo especificado rellenándolo con el color de fondo de la superficie de dibujo actual.
- ***setBackground(Color c)***: establece el color de fondo.
- ***setFont(Font fuente)***: especifica la fuente.
- ***drawString(String texto, int x, int y)***: pinta el texto en las posiciones x e y.

En el siguiente ejemplo se crea un hilo que irá incrementando en 1 un contador, el contador se inicializa en 0. Se definen dos botones. El primero, *Iniciar contador*, crea el hilo, al pulsarle cambia el texto a *Continuar* y empieza a ejecutarse el hilo. El botón *Parar contador* hace que el hilo se detenga y deje de contar, finaliza el método **run()**. Al pulsar de nuevo en *Continuar* el contador continúa incrementándose a partir del último valor, se lanza un nuevo hilo. La Figura 2.3 muestra un momento de la ejecución.

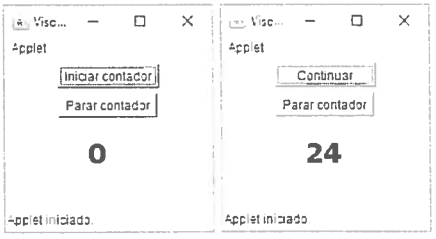

Figura 2.3. Applet ContadorApplet.java.

El applet implementa las interfaces **Runnable** y **ActionListener**. Esta última se usa para detectar y manejar eventos de acción, como por ejemplo hacer clic con el ratón en un botón. Posee un solo método ***actionPerformed(ActionEvent e)*** que es necesario implementar.

En el método ***init()*** del applet se añaden los botones con el método *add()*: *add (b1 = new Button("Iniciar contador"));* y con el método *addActionListener()* añadimos el listener para que detecte cuando se hace clic sobre el botón: *b1.addActionListener(this)*; se usa *this*, ya que es la clase la que implementa la interfaz.

Al pulsar uno de los botones se invocará al método ***actionPerformed(ActionEvent e)*** donde se analizará el evento que ocurre, la pulsación de un botón o del otro. Dentro del método se comprueba el botón que se ha pulsado. Si se ha pulsado el botón *b1* se cambia la etiqueta del botón y se comprueba si el hilo es distinto de nulo y está corriendo, en este caso no se hace nada; y si el hilo es nulo entonces se crea y se inicia su ejecución. Si se pulsa el segundo botón, *b2*, se utiliza la variable booleana *parar* asignándole valor *true* para controlar que no se incremente el contador:

```
public void actionPerformed(ActionEvent e) {
   if(e.getSource() == b1) //Pulso botón Iniciar contador o Continuar
   {
      b1.setLabel("Continuar");
      if(h != null && h.isAlive()) {}  //Si el hilo está corriendo
                                       //no hago nada.
      else {
          //creo hilo la primera vez y cuando finaliza el método run
          h = new Thread(this);
          h.start();
      }
   } else if(e.getSource() == b2) //Pulso Parar contador
           parar = true; //para que finalice el while en el método run

}//actionPerformed
```

En el método **run()** se escribe la funcionalidad del hilo. Se irá incrementando el contador siempre y cuando la variable *parar* sea false. El código completo es el siguiente:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ContadorApplet extends Applet
              implements Runnable, ActionListener {
      private Thread h;
      long CONTADOR = 0;
      private boolean parar;
      private Font fuente;
      private Button b1,b2; //botones del Applet

      public void start() {}

      public void init() {
        setBackground(Color.yellow); //color de fondo
        add(b1=new Button("Iniciar contador"));
        b1.addActionListener(this);
        add(b2=new Button("Parar contador"));
        b2.addActionListener(this);
        fuente = new Font("Verdana", Font.BOLD, 26);//tipo letra
```

```
    }

    public void run() {
      parar = false;
      Thread hiloActual = Thread.currentThread();
      while (h == hiloActual && !parar) {
          try {
                Thread.sleep(300);
          } catch (InterruptedException e) {
                e.printStackTrace();
          }
          repaint();
          CONTADOR++;
      }
    }

    public void paint(Graphics g) {
          g.clearRect(0, 0, 400, 400);
          g.setFont(fuente); //fuente
          g.drawString(Long.toString((long)CONTADOR),80,100);
    }

    //para controlar que se pulsan los botones
    public void actionPerformed(ActionEvent e) {
      if(e.getSource() == b1) //Pulso Iniciar contador o Continuar
      {
       b1.setLabel("Continuar");
       if(h != null && h.isAlive()) {}  //Si el hilo está corriendo
                                        //no hago nada.
       else {
        //creo hilo la primera vez y cuando finaliza el método run
        h = new Thread(this);
        h.start();
       }
      } else if(e.getSource() == b2) //Pulso Parar contador
          parar=true;  //para que finalice el while en el método run

    }//actionPerformed

    public void stop() {
        h = null;
    }
}//fin ContadorApplet
```

ACTIVIDAD 2.3

Partiendo del ejemplo anterior separa el hilo en una clase aparte dentro del applet que extienda **Thread**. El applet ahora no implementará **Runnable**, debe quedar así:

```
public class actividad2_3 extends Applet implements ActionListener {
    class HiloContador extends Thread {
        //atributos y métodos
        . . .
    }//fin clase
  //atributos y métodos
  . . .
}//fin actividad2_3
```

Se debe crear un applet que lance dos hilos y muestre dos botones para finalizarlos, Figura 2.4.

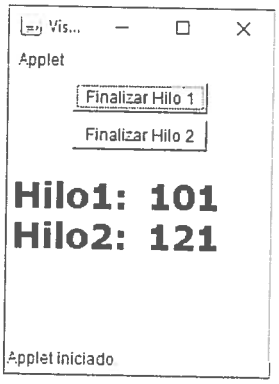

Figura 2.4. Applet Actividad 2.3.

Define en la clase *HiloContador* un constructor que reciba el valor inicial del contador a partir del cual empezará a contar; y el método *getContador()* que devuelva el valor actual del contador. El applet debe crear e iniciar 2 hilos de esta clase, cada uno debe empezar con un valor. Mostrará 2 botones, uno para detener el primer hilo y el otro el segundo, Figura 2.4. Para detener los hilos usa el método ***stop()***: *hilo.stop()* (veremos más adelante que este método está en desuso y no se debe usar). Cambia el texto de los botones cuando se pulsen, que muestre *Finalizado Hilo 1* o 2 dependiendo del botón pulsado.

En el método ***init()*** prepara la pantalla. En el método ***start()*** inicia los dos hilos. En el método ***paint()*** pinta la pantalla. En el método ***actionPerformed(ActionEvent e)*** controla los botones y en el método ***stop()*** finaliza los hilos asignándoles el valor *null*.

2.4. ESTADOS DE UN HILO

Un hilo puede estar en uno de estos estados:

- **New (Nuevo):** es el estado cuando se crea un objeto hilo con el operador *new*, por ejemplo *new Hilo()*, en este estado el hilo aún no se ejecuta; es decir, el programa no ha comenzado la ejecución del código del método **run()** del hilo.
- **Runnable (Ejecutable):** cuando se invoca al método **start()**, el hilo pasa a este estado. El sistema operativo tiene que asignar tiempo de CPU al hilo para que se ejecute; por tanto, en este estado el hilo puede estar o no en ejecución.
- **Dead (Muerto):** un hilo muere por varias razones: de muerte natural, porque el método **run()** finaliza con normalidad; y repentinamente debido a alguna excepción no capturada en el método **run()**. En particular, es posible matar a un hilo invocando su método **stop()**. Este método lanza una excepción *ThreadDeath* que mata al hilo. Sin embargo, el método **stop()** está en desuso y no se debe llamar ya que cuando un hilo se detiene, inmediatamente no libera los bloqueos de los objetos que ha bloqueado. Esto puede dejar a los objetos en un estado inconsistente. Para detener un hilo de manera segura, se puede usar una variable; en este ejemplo se usa la variable *stopHilo* que se inicializa con un valor *true* y se utiliza dentro de un bucle en el método **run()**. Para que termine el bucle del método **run()** se invoca al método *pararHilo()* que cambia el valor de la variable a *false*:

```
public class HiloEjemploDead extends Thread {
  private boolean stopHilo= false;

  public void pararHilo()  {
    stopHilo = true;
  }

  public void run() {
      while (!stopHilo) {
        System.out.println("En el Hilo");
      }
  }

  public static void main(String[] args) {
      HiloEjemploDead h = new HiloEjemploDead ();
      h.start();
      for(int i=0;i<1000000; i++) ;//no hago nada
      h.pararHilo(); //parar el hilo

  }//

}//HiloEjemploDead
```

- **Blocked (Bloqueado):** en este estado podría ejecutarse el hilo, pero hay algo que lo evita. Un hilo entra en estado bloqueado cuando ocurre una de las siguientes acciones:
  1. Alguien llama al método **sleep()** del hilo, es decir, se ha puesto a dormir.
  2. El hilo está esperando a que se complete una operación de entrada/salida.
  3. El hilo llama al método **wait()**. El hilo no se volverá ejecutable hasta que reciba los mensajes **notify()** o **notifyAll()**.
  4. El hilo intenta bloquear un objeto que está actualmente bloqueado por otro hilo.
  5. Alguien llama al método **suspend()** del hilo. No se volverá ejecutable de nuevo hasta que reciba el mensaje **resume()**.

Del estado bloqueado se pasa a ejecutable cuando: expira el número de milisegundos de **sleep()**, se completa la operación de E/S, recibe los mensajes **notify()** o **notifyAll()**, el bloqueo del objeto finaliza, o se llama al método **resume()**. La Figura 2.5 muestra los estados que puede tener un hilo y las posibles transiciones de un estado a otro. Los métodos **resume(), suspend()** y **stop()** están en desuso.

Cuando un hilo está bloqueado (o, por supuesto, cuando muere), otro hilo está previsto para funcionar. Cuando un hilo bloqueado se reactiva (por ejemplo, porque finaliza el número de milisegundos que permanece dormido o porque la E/S que se esperaba se ha completado), el planificador comprueba si tiene una prioridad más alta que el hilo que se está ejecutando actualmente. Si es así, se antepone al actual hilo y selecciona el nuevo hilo para ejecutarlo. En una máquina con varios procesadores, cada procesador puede ejecutar un hilo, se pueden tener varios hilos en paralelo. En tal máquina, un hilo con máxima prioridad se adelantará a otro si no hay disponible procesador para ejecutarlo.

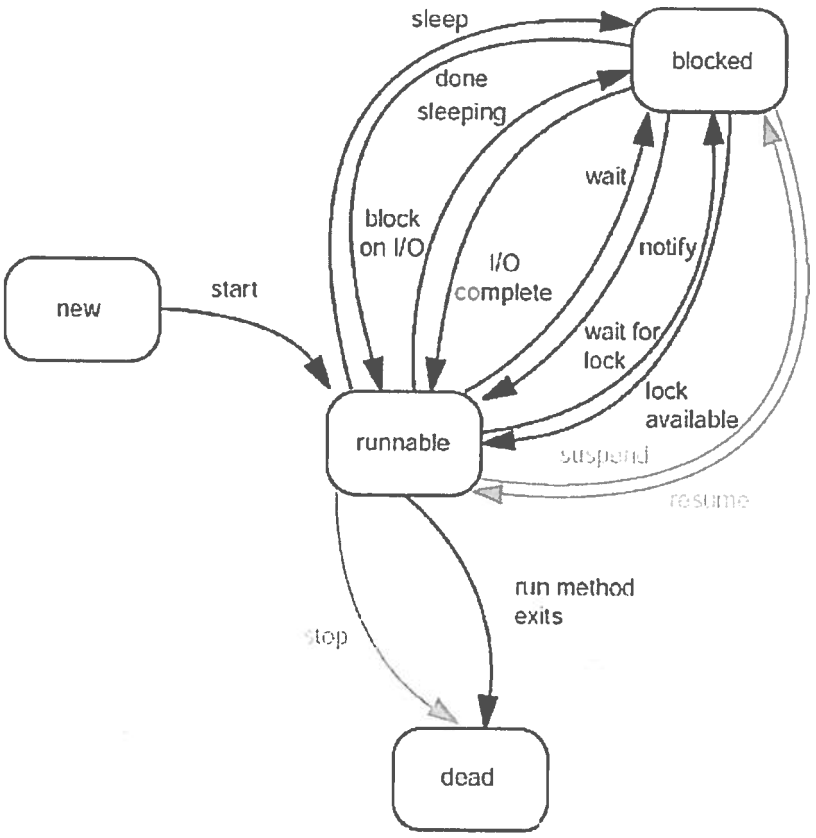

Figura 2.5. Estados de un hilo[1].

El método **getState()** devuelve una constante que indica el estado del hilo. Los valores son los siguientes:

- **NEW**: El hilo aún no se ha iniciado.
- **RUNNABLE**: El hilo se está ejecutando.
- **BLOCKED**: El hilo está bloqueado, esperando tomar el bloqueo de un objeto.
- **WAITING**: El hilo está esperando indefinidamente hasta que otro realice una acción. Por ejemplo un hilo que llama al método **wait()** de un objeto, está esperando hasta que otro hilo llame al método **notify()** del objeto.
- **TIMED_WAITING**: El hilo está esperando que otro hilo realice una acción un tiempo de espera especificado.
- **TERMINATED**: El hilo ha finalizado.

2.5. GESTIÓN DE HILOS

En ejemplos anteriores hemos visto como crear y utilizar los hilos, vamos a dedicar un apartado a los pasos vistos anteriormente.

---

[1] Figura obtenida del libro Core Java™ 2: Volume II–Advanced Features. Cay S. Horstmann, Gary Cornell.