

## CAPÍTULO 2

# PROGRAMACIÓN MULTITHREAD

### Contenidos

- Hilos. Estados de un hilo.
- Gestión de hilos.
- Creación de hilos en Java.
- Compartir información entre hilos.
- Sincronización de hilos.
- Gestión de prioridades.

### Objetivos

- Conocer las características de los hilos en Java.
- Crear y gestionar hilos.
- Crear programas para compartir información entre hilos.
- Crear programas formados por varios hilos sincronizados.

### RESUMEN DEL CAPÍTULO

En este capítulo estudiaremos los hilos. Aprenderemos a crear y gestionar hilos en programas Java.

## 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.

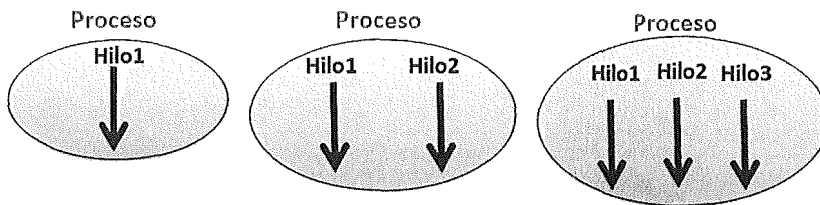


Figura 2.1. Relación entre hilos y procesos.

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
<b>start()</b>	Hace que el hilo comience la ejecución; la máquina virtual de Java llama al método <b>run()</b> de este hilo.
<b>boolean isAlive()</b>	Comprueba si el hilo está vivo
<b>sleep(long mils)</b>	Hace que el hilo actualmente en ejecución pase a dormir temporalmente durante el número de milisegundos especificado. Puede lanzar la excepción <i>InterruptedException</i> .
<b>run()</b>	Constituye el cuerpo del hilo. Es llamado por el método <b>start()</b> después de que el hilo apropiado del sistema se haya inicializado. Si el método <b>run()</b> devuelve el control, el hilo se detiene. Es el único método de la interfaz <b>Runnable</b> .
<b>String toString()</b>	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]
<b>long getId()</b>	Devuelve el identificador del hilo.
<b>void yield()</b>	Hace que el hilo actual de ejecución pare temporalmente y permita que otros hilos se ejecuten.
<b>String getName()</b>	Devuelve el nombre del hilo.
<b>setName(String name)</b>	Cambia el nombre de este hilo, asignándole el especificado como argumento.

MÉTODOS	MISIÓN
<b>int getPriority()</b>	Devuelve la prioridad del hilo.
<b>setPriority(int p)</b>	Cambia la prioridad del hilo al valor entero p.
<b>void interrupt()</b>	Interrumpe la ejecución del hilo
<b>boolean interrupted()</b>	Comprueba si el hilo actual ha sido interrumpido.
<b>Thread currentThread()</b>	Devuelve una referencia al objeto hilo que se está ejecutando actualmente.
<b>boolean isDaemon()</b>	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 ( <i>garbage collector</i> ).
<b>setDaemon(boolean on)</b>	Establece este hilo como hilo Daemon, asignando el valor <i>true</i> , o como hilo de usuario, pasando el valor <i>false</i> .
<b>void stop()</b>	Detiene el hilo. Este método está en desuso.
<b>Thread currentThread()</b>	Devuelve una referencia al objeto hilo actualmente en ejecución.
<b>int activeCount()</b>	Este método devuelve el número de hilos activos en el grupo de hilos del hilo actual.
<b>Thread.State getState()</b>	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.

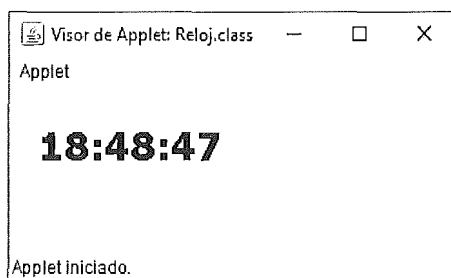


Figura 2.2. Applet Reloj.java.

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.

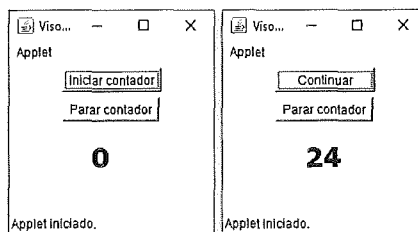


Figura 2.3. Applet ContadorApplet.java.

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.

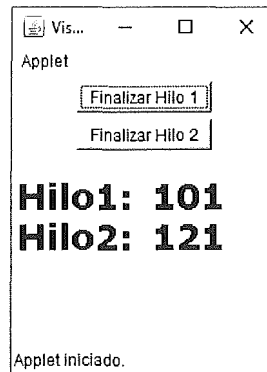


Figura 2.4. Applet Actividad 2.3.

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

    }
}

```

- **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.



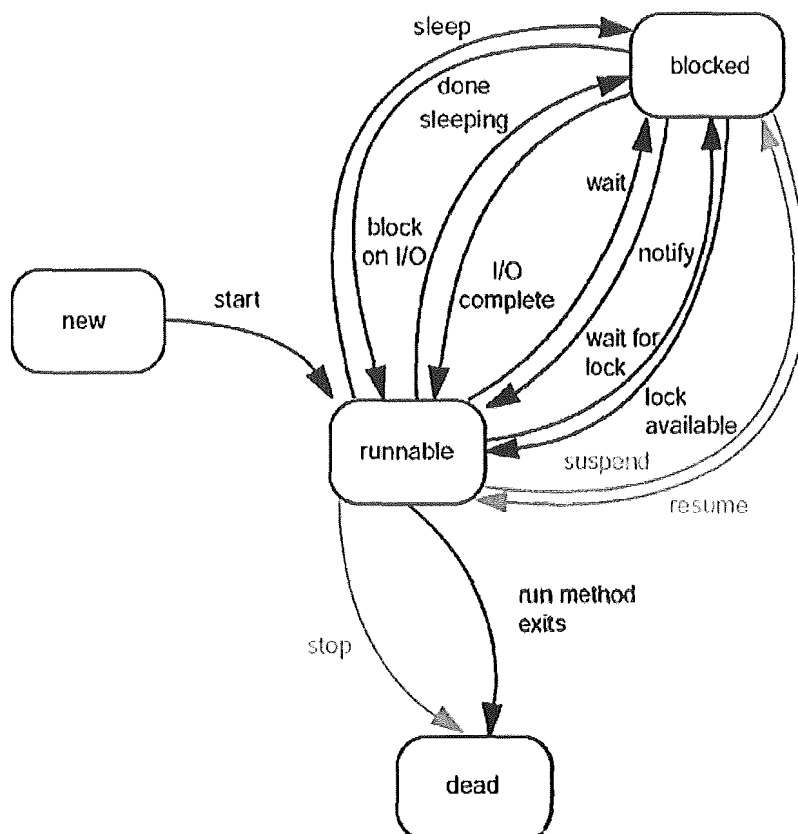


Figura 2.5. Estados de un hilo<sup>1</sup>.

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.

<sup>1</sup> Figura obtenida del libro Core Java™ 2: Volume II—Advanced Features. Cay S. Horstmann, Gary Cornell.

### 2.5.1. CREAR Y ARRANCAR HILOS

Para crear un hilo extendemos la clase **Thread** o implementamos la interfaz **Runnable**. La siguiente línea de código crea un hilo donde *MiHilo* es una subclase de **Thread** (o una clase que implementa la interfaz **Runnable**), se le pasan dos argumentos que se deben definir en el constructor de la clase y se utilizan, por ejemplo, para iniciar variables del hilo:

```
MiHilo h = new MiHilo("Hilo 1", 200);
```

Si todo va bien en la creación del hilo tendremos en *h* el objeto hilo. Para arrancar el hilo usamos el método **start()** de esta manera si extiende **Thread**:

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

Y si implementa **Runnable** lo arrancamos así:

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

Lo que hace este método es llamar al método **run()** del hilo que es donde se colocan las acciones que queremos que haga el hilo, cuando finalice el método finalizará también el hilo.

### 2.5.2. SUSPENSIÓN DE UN HILO

En ejemplos anteriores usamos el método **sleep()** para detener un hilo un número de milisegundos. Realmente el hilo no se detiene, sino que se queda “dormido” el número de milisegundos que indiquemos. Lo utilizábamos en los ejercicios de los contadores para que nos diese tiempo a ver cómo se van incrementando de 1 en 1.

El método **suspend()** permite detener la actividad del hilo durante un intervalo de tiempo indeterminado. Este método es útil cuando se realizan applets con animaciones y en algún momento se decide parar la animación para luego continuar cuando lo decida el usuario. Para volver a activar el hilo se necesita invocar al método **resume()**.

El método **suspend()** es un método obsoleto y tiende a no utilizarse porque puede producir situaciones de interbloqueos. Por ejemplo, si un hilo está bloqueando un recurso y este hilo se suspende puede dar lugar a que otros hilos que esperaban el recurso queden “congelados” ya que el hilo suspendido mantiene los recursos bloqueados. Igualmente el método **resume()** también está en desuso.

Para suspender de forma segura el hilo se debe introducir en el hilo una variable, por ejemplo *suspender* y comprobar su valor dentro del método **run()**, es lo que se hace en la llamada al método *suspender.esperandoParaReanudar()* del ejemplo siguiente. El método *Suspende()* del hilo da valor *true* a la variable para suspender el hilo. El método *Reanuda()* da el valor *false* para que detenga la suspensión y continúe ejecutándose el hilo:

```
class MyHilo extends Thread {
    private SolicitaSuspender suspender = new SolicitaSuspender();
                                     //petición de SUSPENDER HILO
    public void Suspende() { suspender.set(true); }
                                     //petición de CONTINUAR
    public void Reanuda()  { suspender.set(false); }

    public void run() {
        try {
            while(haya trabajo por hacer) {
                . . . . .
                suspender.esperandoParaReanudar(); //comprobar
                . . . . .
            }
        }
    }
}
```

```

        } catch (InterruptedException exception) { }
    }
}

```

Para mayor claridad, se envuelve la variable (a la que se hacía alusión anteriormente) en la clase *SolicitaSuspende*, en esta clase se define el método *set()* que da el valor *true* o *false* a la variable y llama al método *notifyAll()*, este notifica a todos los hilos que esperan (han ejecutado un *wait()*) un cambio de estado sobre el objeto. En el método *esperandoParaReanudar()* se hace un *wait()* cuando el valor de la variable es *true*, el método *wait()* hace que el hilo espere hasta que le llegue un *notify()* o un *notifyAll()*;

```

public class SolicitaSuspende {
    private boolean suspender;

    public synchronized void set(boolean b) {
        suspender = b; //cambio de estado sobre el objeto
        notifyAll();
    }

    public synchronized void esperandoParaReanudar()
        throws InterruptedException {
        while (suspender)
            wait(); //SUSPENDER HILO HASTA RECIBIR notify() o notifyAll()
    }
}

```

El método *wait()* sólo puede ser llamado desde dentro de un método sincronizado (*synchronized*). Estos tres métodos: *wait()*, *notify()* y *notifyAll()*, se usan en sincronización de hilos. Forman parte de la clase **Object**, y no parte de **Thread** como es el caso de *sleep()*, *suspend()* y *resume()*. Se tratarán más adelante.

---

### ACTIVIDAD 2.4

Partimos de las clases anteriores *MyHilo* y *SolicitaSuspende*. Vamos a modificar la clase *MyHilo*. Se define una variable contador y se inicia con valor 0. En el método *run()* y dentro de un bucle que controle el fin del hilo mediante una variable, se incrementa en 1 el valor del contador y se visualiza su valor, se incluye también un *sleep()* para que podamos ver los números. Haz una llamada al método *esperandoParaReanudar()* para suspender el hilo, el *sleep()* lo podemos hacer antes o después. Crea en la clase un método que devuelva el valor del contador. Al finalizar el bucle visualiza un mensaje.

Para probar las clases crea un método *main()*, en el que introducirás una cadena por teclado en un proceso repetitivo hasta introducir un \*. Si la cadena introducida es S se suspenderá el hilo, si la cadena es R se reanudará el hilo. El hilo se lanzará solo una vez después de introducir la primera cadena. Al finalizar el proceso repetitivo visualizar el valor del contador y finalizar el hilo. Comprueba que todos los mensajes se visualicen correctamente.

Realiza el Ejercicio 8.

---

### 2.5.3. PARADA DE UN HILO

El método *stop()* detiene la ejecución de un hilo de forma permanente y ésta no se puede reanudar con el método *start()*:

```
h.stop();
```

Este método al igual que **suspend()**, **resume()** y **destroy()** han sido abolidos en Java 2 para reducir la posibilidad de interbloqueo. El método **run()** no libera los bloqueos que haya adquirido el hilo, y si los objetos están en un estado inconsistente los demás hilos podrán verlos y modificarlos en ese estado. En lugar de usar este método se puede usar una variable como se vio en el estado **Dead** del hilo.

El método **isAlive()** devuelve *true* si el hilo está vivo, es decir ha llamado a su método **run()** y aún no ha terminado su ejecución o no ha sido detenido con **stop()**; en caso contrario devuelve *false*. En ejemplos anteriores vimos cómo se usaba este método.

El método **interrupt()** envía una petición de interrupción a un hilo. Si el hilo se encuentra bloqueado por una llamada a **sleep()** o **wait()** se lanza una excepción *InterruptedException*. El método **isInterrupted()** devuelve *true* si el hilo ha sido interrumpido, en caso contrario devuelve *false*. El siguiente ejemplo usa interrupciones para detener el hilo. En el método **run()** se comprueba en el bucle while si el hilo está interrumpido, si no lo está se ejecuta el código. El método **interrumpir()** ejecuta el método **interrupt()** que lanza una interrupción que es recogida por el manejador (**catch**):

```
public class HiloEjemploInterrup extends Thread {
    public void run() {
        try {
            while (!isInterrupted()) {
                System.out.println("En el Hilo");
                Thread.sleep(10);
            }
        } catch (InterruptedException e) {
            System.out.println("HA OCURRIDO UNA EXCEPCIÓN");
        }
        System.out.println("FIN HILO");
    }

    public void interrumpir() {
        interrupt();
    }

    public static void main(String[] args) {
        HiloEjemploInterrup h = new HiloEjemploInterrup();
        h.start();
        for(int i=0; i<1000000000; i++) ;//no hago nada
        h.interrumpir();
    }
}
```

Un ejemplo de ejecución muestra la siguiente información:

```
En el Hilo
En el Hilo
HA OCURRIDO UNA EXCEPCIÓN
FIN HILO
```

Si en el código anterior quitamos la línea *Thread.sleep(10);* también hay que quitar el bloque **try-catch**, la interrupción será recogida por el método **isInterrupted()**, que será *true* con lo que la ejecución del hilo terminará ya que finaliza el método **run()**.

El método **join()** provoca que el hilo que hace la llamada espere la finalización de otros hilos. Por ejemplo si en el hilo actual escribo *h1.join()*, el hilo se queda en espera hasta que muera el

hilo sobre el que se realiza el **join()**, en este caso *h1*. En el siguiente ejemplo el método **run()** de la clase *HiloJoin* visualiza en un bucle *for* un contador que empieza en 1 hasta un valor *n* que recibe el constructor del hilo:

```
class HiloJoin extends Thread {
    private int n;
    public HiloJoin(String nom, int n) {
        super(nom);
        this.n=n;
    }
    public void run() {
        for(int i=1; i<= n; i++) {
            System.out.println(getName() + ": " + i);
            try {
                sleep(1000);
            } catch (InterruptedException ignore) {}
        }
        System.out.println("Fin Bucle "+getName());
    }
}

public class EjemploJoin {
    public static void main(String[] args) {
        HiloJoin h1 = new HiloJoin("Hilo1",2);
        HiloJoin h2 = new HiloJoin("Hilo2",5);
        HiloJoin h3 = new HiloJoin("Hilo3",7);
        h1.start();
        h2.start();
        h3.start();
        try {
            h1.join(); h2.join(); h3.join();
        } catch (InterruptedException e) {}
        System.out.println("FINAL DE PROGRAMA");
    }
}
```

En el método *main()* se crean 3 hilos, cada uno da un valor diferente a la *n*, el primero el valor más pequeño y el tercero el valor más grande, parece lógico que por los valores del contador el primer hilo debe terminar el primero y el tercer hilo el último. Llamando a **join()** podemos hacer que *main()* espere a la finalización de los hilos y cada hilo finalice en el orden marcado según la llamada a **join()**, cuando salgan del bloque **try-catch** los tres hilos habrán finalizado y el texto FINAL DE PROGRAMA se visualizará al final.

La ejecución muestra la siguiente salida:

```
Hilo1: 1
Hilo3: 1
Hilo2: 1
Hilo2: 2
Hilo3: 2
Hilo1: 2
Hilo2: 3
Hilo3: 3
Fin Bucle Hilo1
Hilo2: 4
Hilo3: 4
Hilo2: 5
Hilo3: 5
```

```

Fin Bucle Hilo2
Hilo3: 6
Hilo3: 7
Fin Bucle Hilo3
FINAL DE PROGRAMA

```

Si en el ejemplo anterior quitamos los `join()` veremos que el texto FINAL DE PROGRAMA no se mostrará al final. El método `join()` puede lanzar la excepción *InterruptedException*, por ello se incluye en un bloque **try-catch**.

### ACTIVIDAD 2.5

Modifica el applet de la Actividad 2.3 de manera que la finalización de los hilos no se realice con el método `stop()` sino que se realice de alguna de las formas seguras vistas anteriormente (usando interrupciones o usando una variable para controlar el fin del hilo).

Realiza el ejercicio 9.

## 2.6. GESTIÓN DE PRIORIDADES

En el lenguaje de programación Java, cada hilo tiene una prioridad. Por defecto, un hilo hereda la prioridad del hilo padre que le crea, esta se puede aumentar o disminuir mediante el método `setPriority()`. El método `getPriority()` devuelve la prioridad del hilo.

La prioridad no es más que un valor entero entre 1 y 10, siendo el valor 1 la mínima prioridad, **MIN\_PRIORITY**; y el valor 10 la máxima, **MAX\_PRIORITY**. **NORM\_PRIORITY** se define como 5. El planificador elige el hilo que debe ejecutarse en función de la prioridad asignada; se ejecutará primero el hilo de mayor prioridad. Si dos o más hilos están listos para ejecutarse y tienen la misma prioridad, la máquina virtual va cediendo control de forma cíclica (*round-robin*).

El planificador es la parte de la máquina virtual de Java que decide qué hilo ejecutar en cada momento. Da más ventaja a hilos con mayor prioridad; hilos de igual prioridad en algún momento se ejecutarán.

El hilo de mayor prioridad sigue funcionando hasta que:

- Cede el control llamando al método `yield()` para que otros hilos de igual prioridad se ejecuten.
- Deja de ser ejecutable (ya sea por muerte o por entrar en el estado de bloqueo).
- Un hilo de mayor prioridad se convierte en ejecutable (porque se encontraba dormido o su operación de E/S ha finalizado o alguien lo desbloquea llamando a los métodos `notifyAll()` / `notify()`).

El uso del método `yield()` devuelve automáticamente el control al planificador, el hilo pasa a un estado de listo para ejecutar. Sin éste método el mecanismo de multihilos sigue funcionando aunque algo más lentamente.

En el siguiente ejemplo se crea una clase que extiende **Thread**, se define una variable contador que será incrementada en el método `run()`, se define un método para obtener el valor de la variable y otro método para finalizar el hilo, en el constructor se le da nombre al hilo:

```

class HiloPrioridad1 extends Thread {
    private int c = 0;
    private boolean stopHilo = false;

```

```

public HiloPrioridad1(String nombre) {
    super(nombre);
}
public int getContador() {
    return c;
}
public void pararHilo() {
    stopHilo = true;
}
public void run() {
    while (!stopHilo) {
        try {
            Thread.sleep(2);
        } catch (Exception e) { }
        c++;
    }
    System.out.println("Fin hilo  "+this.getName());
}
}
}

```

A continuación se crea la clase *EjemploHiloPrioridad1* con el método *main()* en el que se definen 3 objetos de la clase *HiloPrioridad1*, a cada uno se le asigna una prioridad. El contador del hilo al que se le ha asignado mayor prioridad contará más deprisa que el de menos prioridad. Al finalizar cada hilo se muestran los valores del contador invocando a cada método *getContador()* del hilo:

```

public class EjemploHiloPrioridad1 {
    public static void main(String args[]) {
        HiloPrioridad1 h1 = new HiloPrioridad1("Hilo1");
        HiloPrioridad1 h2 = new HiloPrioridad1("Hilo2");
        HiloPrioridad1 h3 = new HiloPrioridad1("Hilo3");

        h1.setPriority(Thread.NORM_PRIORITY);
        h2.setPriority(Thread.MAX_PRIORITY);
        h3.setPriority(Thread.MIN_PRIORITY);

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

        try {
            Thread.sleep(10000);
        } catch (Exception e) { }

        h1.pararHilo();
        h2.pararHilo();
        h3.pararHilo();

        System.out.println("h2 (Prioridad Maxima): " + h2.getContador());
        System.out.println("h1 (Prioridad Normal): " + h1.getContador());
        System.out.println("h3 (Prioridad Minima): " + h3.getContador());
    }
}

```

Varias ejecuciones del programa muestran las siguientes salidas, se puede observar que el máximo valor del contador lo obtiene el objeto hilo con prioridad máxima, y el mínimo el de prioridad mínima:

**h2 (Prioridad Maxima): 4856**

```
Fin hilo Hilo3
Fin hilo Hilo1
Fin hilo Hilo2
h1 (Prioridad Normal): 4855
h3 (Prioridad Minima): 4854
```

**h2 (Prioridad Maxima): 4767**

```
h1 (Prioridad Normal): 4684
h3 (Prioridad Minima): 4668
Fin hilo Hilo2
Fin hilo Hilo1
Fin hilo Hilo3
```

**h2 (Prioridad Maxima): 4565**

```
h1 (Prioridad Normal): 4545
h3 (Prioridad Minima): 4523
Fin hilo Hilo2
Fin hilo Hilo1
Fin hilo Hilo3
```

Pero no siempre ocurre esto. Podemos encontrar la siguiente salida en la que se observa que los valores de los contadores no dependen de la prioridad asignada al hilo:

**h2 (Prioridad Maxima): 4822**

```
Fin hilo Hilo3
Fin hilo Hilo2
Fin hilo Hilo1
h1 (Prioridad Normal): 4823
h3 (Prioridad Minima): 4823
```

**h2 (Prioridad Maxima): 4518**

```
Fin hilo Hilo2
h1 (Prioridad Normal): 4518
h3 (Prioridad Minima): 4517
Fin hilo Hilo1
Fin hilo Hilo3
```

En el siguiente ejemplo se asignan diferentes prioridades a cada uno de los hilos de la clase *EjemploHiloPrioridad2* que se crean. En la ejecución se puede observar que no siempre el hilo con más prioridad es el que antes se ejecuta:

```
public class EjemploHiloPrioridad2 extends Thread {
    EjemploHiloPrioridad2(String nom) {
        this.setName(nom);
    }

    public void run() {
        System.out.println("Ejecutando [" + getName() + "]");
        for (int i = 1; i <= 5; i++)
            System.out.println("\t(" + getName() + ": " + i + ")");
    }
}
```



```

public static void main(String[] args) {
    EjemploHiloPrioridad2 h1 = new EjemploHiloPrioridad2("Uno");
    EjemploHiloPrioridad2 h2 = new EjemploHiloPrioridad2("Dos");
    EjemploHiloPrioridad2 h3 = new EjemploHiloPrioridad2("Tres");
    EjemploHiloPrioridad2 h4 = new EjemploHiloPrioridad2("Cuatro");
    EjemploHiloPrioridad2 h5 = new EjemploHiloPrioridad2("Cinco");

    //asignamos prioridad
    h1.setPriority(Thread.MIN_PRIORITY);
    h2.setPriority(3);
    h3.setPriority(Thread.NORM_PRIORITY);
    h4.setPriority(7);
    h5.setPriority(Thread.MAX_PRIORITY);

    //se ejecutan los hilos
    h1.start();
    h2.start();
    h3.start();
    h4.start();
    h5.start();
}
}

```

Un ejemplo de ejecución muestra la siguiente salida:

```

Ejecutando [Cuatro]
    (Cuatro: 1)
    (Cuatro: 2)
    (Cuatro: 3)
    (Cuatro: 4)
    (Cuatro: 5)
Ejecutando [Tres]
    (Tres: 1)
    (Tres: 2)
    (Tres: 3)
    (Tres: 4)
    (Tres: 5)
Ejecutando [Cinco]
    (Cinco: 1)
    (Cinco: 2)
    (Cinco: 3)
    (Cinco: 4)
    (Cinco: 5)
Ejecutando [Dos]
    (Dos: 1)
    (Dos: 2)
    (Dos: 3)
    (Dos: 4)
    (Dos: 5)
Ejecutando [Uno]
    (Uno: 1)
    (Uno: 2)
    (Uno: 3)
    (Uno: 4)
    (Uno: 5)

```

A la hora de programar hilos con prioridades hemos de tener en cuenta que el comportamiento no está garantizado y dependerá de la plataforma en la que se ejecuten los programas y de las aplicaciones que se ejecuten al mismo tiempo. En la práctica casi nunca hay que establecer a mano las prioridades.

Cuando un hilo entra en ejecución y no cede voluntariamente el control para que puedan ejecutarse otros hilos, se dice que es un “hilo egoísta”. Algunos sistemas operativos, como Windows, combaten estas situaciones con una estrategia de planificación por división de tiempos (*time-slicing* o tiempo compartido), que opera con hilos de igual prioridad que compiten por la CPU. En estas condiciones el sistema operativo divide el tiempo de proceso de la CPU en espacios de tiempo y asigna el tiempo de proceso a los hilos dependiendo de su prioridad. Así se impide que uno de ellos se apropie del sistema durante un intervalo de tiempo prolongado.

---

### ACTIVIDAD 2.6

Prueba los ejemplos anteriores variando la prioridad y el orden de ejecución de cada hilo. Comprueba los resultados para el primer ejemplo y para el segundo.

Realiza el ejercicio 6

---

## 2.7. COMUNICACIÓN Y SINCRONIZACIÓN DE HILOS

A menudo los hilos necesitan comunicarse unos con otros, la forma de comunicarse consiste usualmente en compartir un objeto. En el siguiente ejemplo dos hilos comparten un objeto de la clase *Contador*. Esta clase define un atributo contador y tres métodos, uno de ellos incrementa una unidad su valor, el otro lo decremента y el tercero devuelve su valor; el constructor asigna un valor inicial al contador:

```
class Contador {
    private int c = 0; //atributo contador
    Contador(int c) { this.c = c; }
    public void incrementa() { c = c + 1; }
    public void decremента() { c = c - 1; }
    public int valor() { return c; }
} // CONTADOR
```

Para probar el objeto compartido se definen dos clases que extienden **Thread**. En la clase *HiloA* se usa el método del objeto contador que incrementa en uno su valor. En la clase *HiloB* se usa el método que decremента su valor. Se añade un **sleep()** intencionadamente para probar que un hilo se duerma y mientras el otro haga otra operación con el contador, así la CPU no realiza de una sola vez todo un hilo y después otro y podemos observar mejor el efecto:

```
class HiloA extends Thread {
    private Contador contador;
    public HiloA(String n, Contador c) {
        setName(n);
        contador = c;
    }
    public void run() {
        for (int j = 0; j < 300; j++) {
            contador.incrementa(); //incrementa el contador
            try {
                sleep(100);
            } catch (InterruptedException e) {}
        }
    }
}
```

```

        System.out.println(getName() + " contador vale " +
                           contador.valor());
    }
} // FIN HILOA

class HiloB extends Thread {
    private Contador contador;
    public HiloB(String n, Contador c) {
        setName(n);
        contador = c;
    }
    public void run() {
        for (int j = 0; j < 300; j++) {
            contador.decrementa(); //decrementa el contador
            try {
                sleep(100);
            } catch (InterruptedException e) {}
        }
        System.out.println(getName() + " contador vale " +
                           contador.valor());
    }
} // FIN HILOB

```

A continuación se crea el método *main()*, donde primero se define un objeto de la clase *Contador* y se le asigna el valor inicial de 100. A continuación, se crean los dos hilos pasándoles dos parámetros: un nombre y el objeto *Contador*. Seguidamente se inicia la ejecución de los hilos:

```

public class CompartirInf1 {
    public static void main(String[] args) {
        Contador cont = new Contador(100);
        HiloA a = new HiloA("HiloA", cont);
        HiloB b = new HiloB("HiloB", cont);
        a.start();
        b.start();
    }
}

```

Nos puede dar la impresión que al ejecutar los hilos el valor del contador en el hilo A debería ser 400, ya que empieza en 100 y le suma 300; y en B, 100 ya que se resta 300; pero no es así. Al ejecutar el programa los valores de salida pueden no ser los esperados y variará de una ejecución a otra:

```

HiloB contador vale 100
HiloA contador vale 100

```

Al probarlo sin el método *sleep()* da la sensación de que la salida es la esperada, pero no siempre nos va a dar dicha salida:

```

HiloA contador vale 400
HiloB contador vale 100

```

```

HiloB contador vale 100
HiloA contador vale 100

```

```

HiloA contador vale 43
HiloB contador vale 43

```

```
HiloB contador vale 100
HiloA contador vale 400
```

```
HiloB contador vale -200
HiloA contador vale 100
```

### 2.7.1. BLOQUES SINCRONIZADOS

Una forma de evitar que esto suceda es hacer que las operaciones de incremento y decremento del objeto contador se hagan de forma atómica, es decir, si estamos realizando la suma nos aseguramos que nadie realice la resta hasta que no terminemos la suma. Esto se puede lograr añadiendo la palabra **synchronized** a la parte de código que queramos que se ejecute de forma atómica. Java utiliza los **bloques synchronized** para implementar las **regiones críticas** (que se tratarán en el Capítulo 1). El formato es el siguiente:

```
synchronized (object) {
    //sentencias críticas
}
```

Los métodos **run()** de las clases *HiloA* e *HiloB* se pueden sustituir por los siguientes; para el *HiloB*:

```
public void run() {
    synchronized (contador) {
        for (int j = 0; j < 300; j++) {
            contador.decrementa();
        }
        System.out.println(getName() + " contador vale "
            + contador.valor());
    }
}
```

Para el *HiloA*:

```
public void run() {
    synchronized (contador) {
        for (int j = 0; j < 300; j++) {
            contador.incrementa();
        }
        System.out.println(getName() + " contador vale "
            + contador.valor());
    }
}
```

El bloque **synchronized** o **región crítica** (que aparece sombreado) lleva entre paréntesis la referencia al objeto compartido *Contador*. Cada vez que un hilo intenta acceder a un bloque sincronizado le pregunta a ese objeto si no hay algún otro hilo que ya le tenga bloqueado. Es decir, le pregunta si no hay otro hilo ejecutando algún bloque sincronizado con ese objeto. Si está tomado por otro hilo, entonces el hilo actual se suspende y se pone en espera hasta que se libere el bloqueo. Si está libre, el hilo actual bloquea el objeto y ejecuta el bloque; el siguiente hilo que intente ejecutar un bloque sincronizado con ese objeto, será puesto en espera. El bloqueo del objeto se libera cuando el hilo que lo tiene tomado sale del bloque porque termina la ejecución, ejecuta un **return** o lanza una excepción.

La compilación y ejecución muestra la siguiente salida (se ha guardado el ejemplo anterior con los cambios en *CompartirInf2.java*):

```
HiloA contador vale 400  
HiloB contador vale 100
```

Si se cambia el orden de la ejecución de los hilos, primero el HiloB y luego el HiloA, la salida será:

```
HiloB contador vale -200  
HiloA contador vale 100
```

---

### ACTIVIDAD 2.7

Crea un programa Java que lance cinco hilos, cada uno incrementará una variable contador de tipo entero, compartida por todos, 5000 veces y luego saldrá. Comprobar el resultado final de la variable. ¿Se obtiene el resultado correcto? Ahora sincroniza el acceso a dicha variable. Lanza los hilos primero mediante la clase **Thread** y luego mediante el interfaz **Runnable**. Comprueba el resultado.

---

### 2.7.2. MÉTODOS SINCRONIZADOS

Se debe evitar la sincronización de bloques de código y sustituirlas siempre que sea posible por la sincronización de métodos, **exclusión mutua** de los procesos respecto a la variable compartida. Imaginemos la situación que dos personas comparten una cuenta y pueden sacar dinero de ella en cualquier momento; antes de retirar dinero se comprueba siempre si existe saldo. La cuenta tiene 50€, una de las personas quiere retirar 40 y la otra 30. La primera llega al cajero, revisa el saldo, comprueba que hay dinero y se prepara para retirar el dinero, pero antes de retirarlo llega la otra persona a otro cajero, comprueba el saldo que todavía muestra los 50€ y también se dispone a retirar el dinero. Las dos personas retiran el dinero, pero entonces el saldo actual será ahora de -20.

Para sincronizar un método, simplemente añadimos la palabra clave **synchronized** a su declaración. Por ejemplo, la clase *Contador* con métodos sincronizados sería así:

```
public class ContadorSincronizado {  
    private int c = 0;  
  
    public synchronized void incrementa() {  
        c++;  
    }  
  
    public synchronized void decrementa() {  
        c--;  
    }  
  
    public synchronized int valor() {  
        return c;  
    }  
}
```

El uso de métodos sincronizados implica que no es posible invocar dos métodos sincronizados del mismo objeto a la vez. Cuando un hilo está ejecutando un método sincronizado de un objeto, los demás hilos que invoquen a métodos sincronizados para el mismo objeto se bloquean hasta que el primer hilo termine con la ejecución del método.

Veamos mediante clases como sería el ejemplo de compartir una cuenta sin usar métodos sincronizados. Se define la clase *Cuenta*, define un atributo saldo y tres métodos, uno devuelve el

valor del saldo, otro, resta al saldo una cantidad y el tercero realiza las comprobaciones para hacer la retirada de dinero, es decir que el saldo actual sea  $\geq$  que la cantidad que se quiere retirar; el constructor inicia el saldo actual. También se añade un `sleep()` intencionadamente para probar que un hilo se duerma y mientras el otro haga las operaciones:

```
class Cuenta {
    private int saldo ;

    Cuenta (int s) { saldo = s; }    //inicializa saldo actual
    int getSaldo() { return saldo; } //devuelve el saldo
    void restar(int cantidad) {      //se resta la cantidad al saldo
        saldo = saldo - cantidad;
    }

    void RetirarDinero(int cant, String nom) {
        if (getSaldo() >= cant) {
            System.out.println(nom + ": SE VA A RETIRAR SALDO (ACTUAL ES: " +
                               getSaldo() + ")");

            try {
                Thread.sleep(500);
            } catch (InterruptedException ex) { }

            restar(cant); //resta la cantidad del saldo

            System.out.println("\t" + nom +
                               " retira => " + cant + " ACTUAL(" + getSaldo() + ")");
        } else {
            System.out.println(nom +
                               " No puede retirar dinero, NO HAY SALDO (" + getSaldo() + ")");
        }
        if (getSaldo() < 0) {
            System.out.println("SALDO NEGATIVO => " + getSaldo());
        }
    } //RetirarDinero
} //Cuenta
```

A continuación, se crea la clase *SacarDinero* que extiende **Thread** y usa la clase *Cuenta* para retirar el dinero. El constructor recibe una cadena, para dar nombre al hilo; y la cuenta que será compartida por varias personas. En el método `run()` se realiza un bucle donde se invoca al método *RetirarDinero()* de la clase *Cuenta* varias veces con la cantidad a retirar, en este caso siempre es 10, y el nombre del hilo:


```
class SacarDinero extends Thread {
    private Cuenta c;
    String nom;
    public SacarDinero(String n, Cuenta c) {
        super(n);
        this.c = c;
    }
    public void run() {
        for (int x = 1; x <= 4; x++) {
            c.RetirarDinero(10, getName());
        }
    } // run
}
```

Por último se crea el método *main()*, donde primero se define un objeto de la clase *Cuenta* y se le asigna un saldo inicial de 40. A continuación se crean dos objetos de la clase *SacarDinero*, imaginemos que son las dos personas que comparten la cuenta, y se inician los hilos:

```
public class CompartirInf3 {
    public static void main(String[] args) {
        Cuenta c = new Cuenta(40);
        SacarDinero h1 = new SacarDinero("Ana", c);
        SacarDinero h2 = new SacarDinero("Juan", c);
        h1.start();
        h2.start();
    }
}
```

Al ejecutar puede ocurrir que se permita retirar saldo cuando este es 0:

```
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
      Ana retira =>10 ACTUAL(30)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 30)
      Juan retira =>10 ACTUAL(20)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 20)
      Ana retira =>10 ACTUAL(10)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 10)
      Juan retira =>10 ACTUAL(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
      Ana retira =>10 ACTUAL(-10)
SALDO NEGATIVO => -10
Ana No puede retirar dinero, NO HAY SALDO(-10)
SALDO NEGATIVO => -10
```



Para evitar esta situación la operación de retirar dinero, método *RetirarDinero()* de la clase *Cuenta*, debería ser atómica e indivisible, es decir si una persona está retirando dinero, la otra debería ser incapaz de retirarlo hasta que la primera haya realizado la operación. Para ello declaramos el método como **synchronized**. Sincronizar métodos permite prevenir inconsistencias cuando un objeto es accesible desde distintos hilos: si un objeto es visible para más de un hilo, todas las lecturas o escrituras de las variables de ese objeto se realizan a través de métodos sincronizados.

Cuando un hilo invoca un método **synchronized**, trata de tomar el bloqueo del objeto a que pertenezca. Si está libre, lo toma y se ejecuta. Si el bloqueo está tomado por otro hilo se suspende el que invoca hasta que aquel finalice y libere el bloqueo. La forma de declararlo es la siguiente:

```
synchronized public void metodo() {
    //instrucciones atómicas...
}
```

O bien:

```
public synchronized void metodo() {
    //instrucciones atómicas...
}
```

El método *RetirarDinero()* en nuestro ejemplo quedaría así:

```
synchronized void RetirarDinero(int cant, String nom) {
    //las mismas instrucciones que antes
}
```

La ejecución mostraría la siguiente salida, recuerda que de una ejecución a otra puede variar, pero en este caso el saldo no será negativo:

```
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
    Ana retira =>10 ACTUAL(30)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 30)
    Juan retira =>10 ACTUAL(20)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 20)
    Ana retira =>10 ACTUAL(10)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 10)
    Ana retira =>10 ACTUAL(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Ana No puede retirar dinero, NO HAY SALDO(0)
```

Se debe tener en cuenta que la sincronización disminuye el rendimiento de una aplicación, por tanto, debe emplearse solamente donde sea estrictamente necesario.

## ACTIVIDAD 2.8

Crea una clase de nombre *Saldo*, con un atributo que nos indica el saldo, el constructor que da un valor inicial al saldo. Crea varios métodos uno para obtener el saldo y otro para dar valor al saldo, en estos dos métodos añade un `sleep()` aleatorio. Y otro método que reciba una cantidad y se la añada al saldo, este método debe informar de quién añade cantidad al saldo, la cantidad que añade, el estado inicial del saldo (antes de añadir la cantidad) y el estado final del saldo después de añadir la cantidad. Define los parámetros necesarios que debe de recibir este método y defínelo como **synchronized**.

Crea una clase que extienda **Thread**, desde el método `run()` hemos de usar el método de la clase *Saldo* que añade la cantidad al saldo. Averigua los parámetros que se necesita en el constructor. No debe visualizar nada en pantalla.

Crea en el método `main()` un objeto *Saldo* asignándole un valor inicial. Visualiza el saldo inicial. Crea varios hilos que compartan ese objeto *Saldo*. A cada hilo le damos un nombre y le asignamos una cantidad. Lanzamos los hilos y esperamos a que finalicen para visualizar el saldo final del objeto *Saldo*. Comprueba los resultados quitando **synchronized** del método de la clase *Saldo* que reciba la cantidad y se la añada al saldo.

Realiza el Ejercicio 11.

## 2.7.3. BLOQUEO DE HILOS

En el siguiente ejemplo creamos una clase que define un método que recibe un String y lo pinta:

```
class ObjetoCompartido {
    public void PintaCadena (String s) {
        System.out.print(s);
    }
}
// ObjetoCompartido
```



Para usarla definimos un método *main()* en el que se crea un objeto de esa clase que además será compartido por dos hilos del tipo *HiloCadena*. Los hilos usarán el método del objeto compartido para pintar una cadena, esta cadena es enviada al crear el hilo (*new HiloCadena (objeto compartido, cadena)*):

```
public class BloqueoHilos {
    public static void main(String[] args) {
        ObjetoCompartido com = new ObjetoCompartido();
        HiloCadena a = new HiloCadena (com, " A ");
        HiloCadena b = new HiloCadena (com, " B ");
        a.start();
        b.start();
    }
} //BloqueoHilos
```

La clase *HiloCadena* extiende **Thread**; en su método **run()** invoca al método *PintaCadena()* del objeto compartido dentro de un bucle for:

```
class HiloCadena extends Thread {
    private ObjetoCompartido objeto;
    String cad;

    public HiloCadena (ObjetoCompartido c, String s) {
        this.objeto = c;
        this.cad=s;
    }

    public void run() {
        for (int j = 0; j < 10; j++)
            objeto.PintaCadena (cad) ;
    } //run
} //HiloCadena
```

Se pretende mostrar de forma alternativa los String que inicializa cada hilo y que la salida generada al ejecutar la función *main()* sea la siguiente: "A B A B A B ....". Parece que una primera aproximación para solucionarlo sería sincronizar el trozo de código que hace uso del objeto compartido (dentro del método **run()**):

```
synchronized (objeto) {
    for (int j = 0; j < 10; j++)
        objeto.PintaCadena (cad) ;
}
```

Pero al ejecutarlo, la salida no es la esperada ya que la sincronización evita que dos llamadas a métodos o bloques sincronizados del mismo objeto se mezclen; pero no garantiza el orden de las llamadas; y en este caso nos interesa que las llamadas al método *PintaCadena()* se realicen de forma alternativa. Se necesita por tanto mantener una cierta coordinación entre los dos hilos, para ello se usan los métodos **wait()**, **notify()** y **notifyAll()**:

- **Objeto.wait()**: un hilo que llama al método **wait()** de un cierto objeto queda suspendido hasta que otro hilo llame al método **notify()** o **notifyAll()** del mismo objeto.

- **Objeto.notify():** despierta sólo a uno de los hilos que realizó una llamada a **wait()** sobre el mismo objeto notificándole de que ha habido un cambio de estado sobre el objeto. Si varios hilos están esperando el objeto, solo uno de ellos es elegido para ser despertado, la elección es arbitraria.
- **Objeto.notifyAll():** despierta todos los hilos que están esperando el objeto.

En el ejemplo, dentro del bloque sincronizado y después de pintar la cadena se invocará al método **notify()** del objeto compartido para despertar al hilo que esté esperando el objeto (**notifyAll()** cuando varios hilos esperan el objeto). Inmediatamente después se llama al método **wait()** del objeto para que el hilo quede suspendido y el que estaba en espera tome el objeto para pintar la cadena; el hilo permanecerá suspendido hasta que se produzca un **notify()** sobre el objeto. El último **notify()** es necesario para que los hilos finalicen correctamente y ninguno quede bloqueado:

```
public void run() {
    synchronized (objeto) {

        for (int j = 0; j < 10; j++) {
            objeto.PintaCadena(cad);
            objeto.notify(); //aviso que ya he usado el objeto
            try {
                objeto.wait(); //esperar a que llegue un notify
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

objeto.notify(); //despertar a todos los wait sobre el objeto

} //fin bloque synchronized

System.out.print("\n"+cad + " finalizado");
} //run
```

La ejecución del ejemplo muestra la siguiente salida:

```
A B A B A B A B A B A B A B A B A B
B finalizado
A finalizado
```

Los métodos **notify()** y **wait()** pueden ser invocados sólo desde dentro de un método sincronizado o dentro de un bloque o una sentencia sincronizada.

## 2.7.4. EL MODELO PRODUCTOR-CONSUMIDOR

Un problema típico de sincronización es el que representa el modelo **Productor-Consumidor**. Se produce cuando uno o más hilos producen datos a procesar y otros hilos los consumen. El problema surge cuando el productor produce datos más rápido que el consumidor los consuma, dando lugar a que el consumidor se salte algún dato. Igualmente, el consumidor puede consumir más rápido que el productor produce, entonces el consumidor puede recoger varias veces el mismo dato o puede no tener datos que recoger o puede detenerse, etc.

Por ejemplo, imaginemos una aplicación donde un hilo (el productor) escribe datos en un fichero mientras que un segundo hilo (el consumidor) lee los datos del mismo fichero; en este caso los hilos comparten un mismo recurso (el fichero) y deben sincronizarse para realizar su tarea correctamente.

## EJEMPLO PRODUCTOR-CONSUMIDOR

Se definen 3 clases, la clase *Cola* que será el objeto compartido entre el productor y el consumidor; y las clases *Productor* y *Consumidor*. En el ejemplo el productor produce números y los coloca en una cola, estos serán consumidos por el consumidor. El recurso a compartir es la cola con los números.

El productor genera números de 0 a 5 en un bucle *for*, y los pone en el objeto *Cola* mediante el método *put()*; después se visualiza y se hace un pausa con *sleep()*, durante este tiempo el hilo esta en el estado *Not Runnable* (no ejecutable):

```
public class Productor extends Thread {
    private Cola cola;
    private int n;

    public Productor(Cola c, int n) {
        cola = c;
        this.n = n;
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            cola.put(i); //pone el número
            System.out.println(i + "=>Productor : " + n
                               + ", produce: " + i);

            try {
                sleep(100);
            } catch (InterruptedException e) { }
        }
    }
}
```

La clase *Consumidor* es muy similar a la clase *Productor*, solo que en lugar de poner un número en el objeto *Cola* lo recoge llamando al método *get()*. En este caso no se ha puesto pausa, con esto hacemos que el consumidor sea más rápido que el productor:

```
public class Consumidor extends Thread {
    private Cola cola;
    private int n;

    public Consumidor(Cola c, int n) {
        cola = c;
        this.n = n;
    }

    public void run() {
        int valor = 0;
        for (int i = 0; i < 5; i++) {
            valor = cola.get(); //recoge el número
            System.out.println(i + "=>Consumidor: " + n
                               + ", consume: " + valor);
        }
    }
}
```

```

    }
}

```

La clase *Cola* define 2 atributos y dos métodos. En el atributo *numero* se guarda el número entero y el atributo *disponible* se utiliza para indicar si hay disponible o no un número en la cola. El método *put()* guarda un entero en el atributo *numero* y hace que este esté disponible en la cola para que pueda ser consumido poniendo el valor *true* en *disponible* (cola llena). El método *get()* devuelve el entero de la cola si está disponible (*disponible=true*) y antes pone la variable a *false* indicando cola vacía; si el número no está en la cola (*disponible=false*) devuelve -1;

```

public class Cola {
    private int numero;
    private boolean disponible = false; //inicialmente cola vacia

    public int get() {
        if(disponible) {           //hay número en la cola
            disponible = false; //se pone cola vacía
            return numero;        //se devuelve
        }
        return -1;                //no hay número disponible, cola vacía
    }

    public void put(int valor) {
        numero = valor;           //coloca valor en la cola
        disponible = true;        //disponible para consumir, cola llena
    }
}

```

En el método *main()* que usa las clases anteriores creamos 3 objetos, un objeto de la clase *Cola*, un objeto de la clase *Productor* y otro objeto de la clase *Consumidor*. Al constructor de las clases *Productor* y *Consumidor* le pasamos el objeto compartido de la clase *Cola* y un número entero que lo identifique:

```

public class Produc_Consum {
    public static void main(String[] args) {
        Cola cola = new Cola();
        Productor p = new Productor(cola, 1);
        Consumidor c = new Consumidor(cola, 1);
        p.start();
        c.start();
    }
}

```

Al ejecutar se produce la siguiente salida, en la que se puede observar que el consumidor va más rápido que el productor (al que se le puso un *sleep()*) y no consume todos los números cuando se producen; el numerito de la izquierda de cada fila representa la iteración:

```

0=>Productor1 : produce: 0
0=>Consumidor1: consume: 0
1=>Consumidor1: consume: -1
2=>Consumidor1: consume: -1
3=>Consumidor1: consume: -1
4=>Consumidor1: consume: -1
1=>Productor1 : produce: 1
2=>Productor1 : produce: 2
3=>Productor1 : produce: 3
4=>Productor1 : produce: 4

```

En la iteración 0, el productor produce un 0 e inmediatamente el consumidor lo consume, la cola se queda vacía. En la iteración 1 el consumidor consume -1 que indica que la cola está vacía porque la iteración 1 del productor no se ha producido. En la iteración 2 pasa lo mismo el consumidor toma -1 porque el productor aún no ha dejado valor en la cola. Y así sucesivamente. La salida deseada es la siguiente: en cada iteración el productor produce un número (llena la cola) e inmediatamente el consumidor lo consume (la vacía):

```
0=>Productor1 : produce: 0
0=>Consumidor1: consume: 0
1=>Productor1 : produce: 1
1=>Consumidor1: consume: 1
2=>Productor1 : produce: 2
2=>Consumidor1: consume: 2
3=>Productor1 : produce: 3
3=>Consumidor1: consume: 3
4=>Productor1 : produce: 4
4=>Consumidor1: consume: 4
```

---

### ACTIVIDAD 2.9

Prueba las clases *Productor* y *Consumidor* quitando el método *sleep()* del productor o añadiendo uno al consumidor para hacer que uno sea más rápido que otro. ¿Se obtiene la salida deseada?.

---

Para obtener la salida anterior es necesario que los hilos estén sincronizados. Primero hemos de declarar los métodos *get()* y *put()* de la clase Cola como **synchronized**, de esta manera el productor y consumidor no podrán acceder simultáneamente al objeto Cola compartido; es decir el productor no puede cambiar el valor de la cola cuando el consumidor esté recogiendo su valor; y el consumidor no puede recoger el valor cuando el productor lo esté cambiando:

```
public synchronized int get() {
    //instrucciones
}
public synchronized void put(int valor) {
    //instrucciones
}
```

En segundo lugar, es necesario mantener una coordinación entre el productor y el consumidor de forma que cuando el productor ponga un número en la cola avise al consumidor de que la cola está disponible para recoger su valor; y al revés, cuando el consumidor recoja el valor de la cola debe avisar al productor de que la cola ha quedado vacía. A su vez, el consumidor deberá esperar hasta que la cola se llene y el productor esperará hasta que la cola esté nuevamente vacía para poner otro número.

Para mantener esta coordinación usamos los métodos **wait()**, **notify()** y **notifyAll()** vistos anteriormente. Estos sólo pueden ser invocados desde dentro de un método sincronizado o dentro de un bloque o una sentencia sincronizada.

El método *get()* tiene que esperar a que la cola se llene (Figura 2.6) , esto se realiza en el bucle while: mientras la cola esté vacía, es decir *disponible* es *false* (*while (!disponible)*), espero (*wait*). Se sale del bucle cuando llega un valor, en este caso se vuelve a poner *disponible* a *false* (porque se va a devolver quedando la cola vacía de nuevo), se notifica a todos los hilos que comparten el objeto este hecho y se devuelve el valor (Figura 2.7):

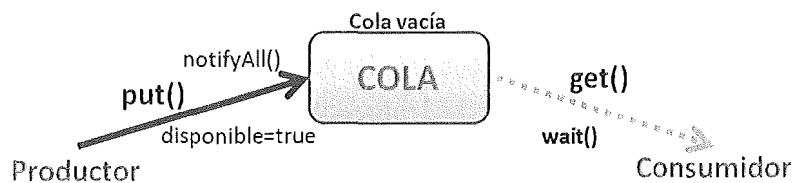


Figura 2.6. Método `get()` espera.

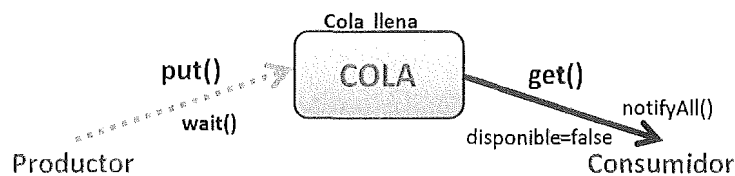


Figura 2.7. Método `get()` devuelve valor, `put()` espera.

Método <code>get()</code> sincronizado	Método <code>get()</code> anterior
<pre>public synchronized int get() {     while (!disponible) {         try {             wait();         } catch (InterruptedException e) { }     }     //visualizar número     disponible = false;     notify();     return numero; }</pre>	<pre>public int get() {     if(disponible) {         disponible=false;         return numero     }     return -1; }</pre>

El método `put()` tiene que esperar a que la cola se vacíe para poner el valor, entonces espera (`wait`) mientras haya valor en la cola (`while (disponible)`). Cuando la cola se vacía, `disponible` es `false`, entonces se sale del bucle, se asigna el valor a la cola, se vuelve a poner disponible a `true` (porque la cola esta llena) y se notifica a todos los hilos que comparten el objeto este hecho:

Método <code>put()</code> sincronizado	Método <code>put()</code> anterior
<pre>public synchronized void put(int valor) {     while (disponible){         try {             wait();         } catch (InterruptedException e) { }     }     numero = valor;     disponible = true;     //visualizar número     notify(); }</pre>	<pre>public void put(int valor) {     numero = valor;     disponible=true; }</pre>



## COMPRUEBA TU APRENDIZAJE

1º) Crea una clase que extienda **Thread** cuya única funcionalidad sea visualizar el mensaje "Hola mundo". Crea un programa Java que visualice el mensaje anterior 5 veces creando para ello 5 hilos diferentes usando la clase creada anteriormente. Modifica el mensaje "Hola mundo" en el hilo para incluir el identificador del hilo. Prueba de nuevo el programa Java creado anteriormente.

2º) Crea una clase que implemente la interfaz **Runnable** cuya única funcionalidad sea visualizar el mensaje "Hola mundo" seguido de una cadena que se recibirá en el constructor (es decir al crear un objeto de este tipo se enviará una cadena) y seguido del identificador del hilo. Crea un programa Java que visualice el mensaje anterior 5 veces creando para ello 5 hilos diferentes usando la clase creada anteriormente. Luego haz que antes de visualizar el mensaje el hilo espere un tiempo proporcional a su identificador; usa para ello el método **sleep()**. ¿Qué diferencias observas al ejecutar el programa usando o no el método **sleep()**?

3º) Implemente un programa que reciba a través de sus argumentos una lista de ficheros de texto y cuente el número de caracteres que hay en cada fichero. Modifica el programa para que se cree un hilo por cada fichero a contar. Muestra lo que se tarda en contar cada fichero en la primera tarea secuencial y usando hilos. Para calcular el tiempo que tarda en ejecutarse un proceso podemos usar el método **System.currentTimeMillis()** de la siguiente manera:

```
long t_comienzo, t_fin;
t_comienzo = System.currentTimeMillis();
Proceso(); // llamamos al proceso
t_fin = System.currentTimeMillis();
long tiempototal = t_fin - t_comienzo;
System.out.println("El proceso ha tardado: " + tiempototal + " miliseg");
```

4º) Haz un programa Java que reciba a través de sus argumentos una lista de ficheros de texto y cuente el número de palabras que hay en cada fichero. Se debe crear un hilo por cada fichero a contar. Muestra el número de palabras de cada fichero y lo que tarda en contar las palabras.

5º) Realiza un applet en el que una letra se mueva horizontalmente por la pantalla y rebote, Figura 2.8. La letra debe empezar en las posiciones  $x = 1$  e  $y = 100$  y se moverá de izquierda a derecha avanzando la  $x$ . Cuando la  $x$  sea mayor que el ancho del applet (**getSize().width**), hay que hacer que rebote y se mueva de derecha a izquierda. Habrá que comprobar si la letra llega a la posición 1 para hacerla rebotar de nuevo y se mueva de izquierda a derecha. El botón *Finalizar Hilo*, detiene el movimiento de la letra y muestra *Reanudar Hilo*, al pulsarlo de nuevo la letra continua su movimiento y el botón muestra el texto *Finalizar Hilo*.



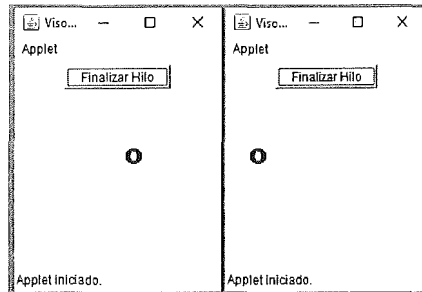


Figura 2.8. Ejercicio 5.

6º) Realiza una pantalla gráfica para simular la carrera de 3 hilos. Inicialmente se debe mostrar la pantalla de la Figura 2.9, donde se muestran 3 barras de progreso (componente *JProgressBar*), 3 controles deslizantes (componente *JSlider*) y 3 campos en los que se irá mostrando el contador de cada hilo. En las barras de progreso se va mostrando como va la carrera; inicialmente la prioridad asignada a cada hilo es de 5, se puede variar usando el control deslizante. El botón *Comenzar Proceso* crea los hilos y lanza su ejecución teniendo en cuenta la prioridad seleccionada, que se debe mostrar en la pantalla. El botón se desactivará volviendo a activarse cuando la carrera finalice, pudiendo empezar de nuevo otra carrera. Cuando la carrera finaliza se debe mostrar el hilo ganador. En el hilo se usará `sleep()`, el constructor debe recibir un valor numérico que indicará el tiempo que el hilo se queda dormido. Prueba el ejercicio con el mismo valor para todos los hilos y con un valor aleatorio, por ejemplo: `(long) Math.random() * 1000 + 1`. Ejecuta varias veces el ejercicio y comprueba si el hilo de máxima prioridad es el que gana.

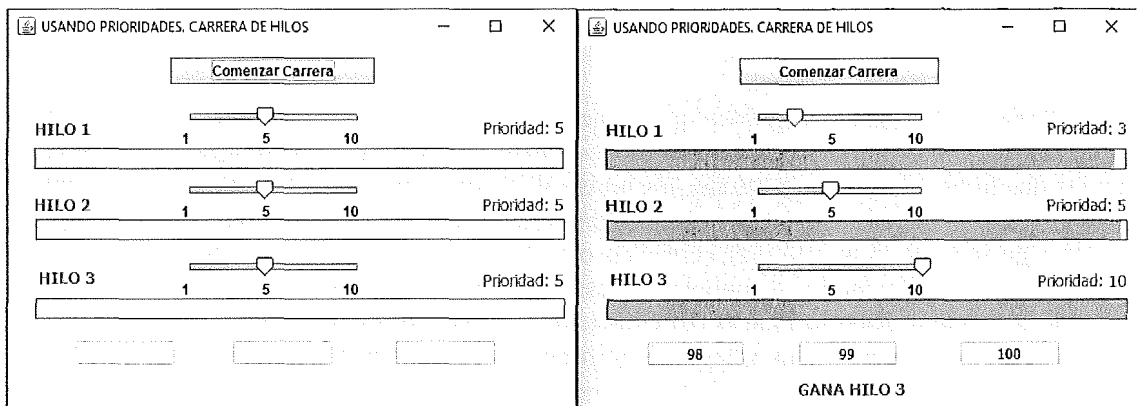


Figura 2.9. Ejercicio 6.

7º) Crea una clase para gestionar el saldo de una *Cuenta*. Debe tener métodos para obtener el saldo actual, hacer un ingreso (se incrementa al saldo), hacer un reintegro (se le resta al saldo), controlar si hay algún error por ejemplo si se hace un reintegro y no hay saldo; o si se hace un ingreso y el saldo supera el máximo; mostrar mensajes con los movimientos que se realicen. La cuenta recibe en su constructor el saldo actual y el valor máximo que puede tener. Los métodos de ingreso y reintegro deben definirse como **synchronized**.

Crea después la clase *Persona* que extienda **Thread** y que realice en su método `run()` 2 ingresos y 2 reintegros alternándolos y con algún `sleep()` en medio. Para crear los movimientos de dinero generar números aleatorios entre 1 y 500 con la función `random()`: `int aleatorio = ((int) (Math.random()*500+1))`. El constructor de la clase debe llevar el nombre de la persona.

Crea en el método *main()* un objeto *Cuenta* compartido por varios objetos *Persona* e inicia el proceso de realizar movimientos en la cuenta.

8º) Partiendo de la *Actividad 2\_4*, realiza la misma operación para suspender y reanudar el hilo, pero ahora se usará una pantalla gráfica con varios botones y se lanzarán dos hilos, véase Figura 2.10. El botón *Comenzar Proceso* crea los dos hilos y lanza su ejecución, los hilos sólo se crean una vez, el botón se desactivará al pulsarle. Cada hilo tendrá su botón para suspenderlo o reanudarlo. Se debe mostrar un mensaje en pantalla que indique el estado de cada hilo, corriendo o suspendido. El botón *Finalizar Proceso* detiene los dos hilos y muestra en consola el valor final de cada contador. El cierre de la ventana hace lo mismo. El constructor del hilo recibe dos parámetros, uno con el nombre del hilo y el segundo la cantidad de milisegundos que permanece el hilo dormido.

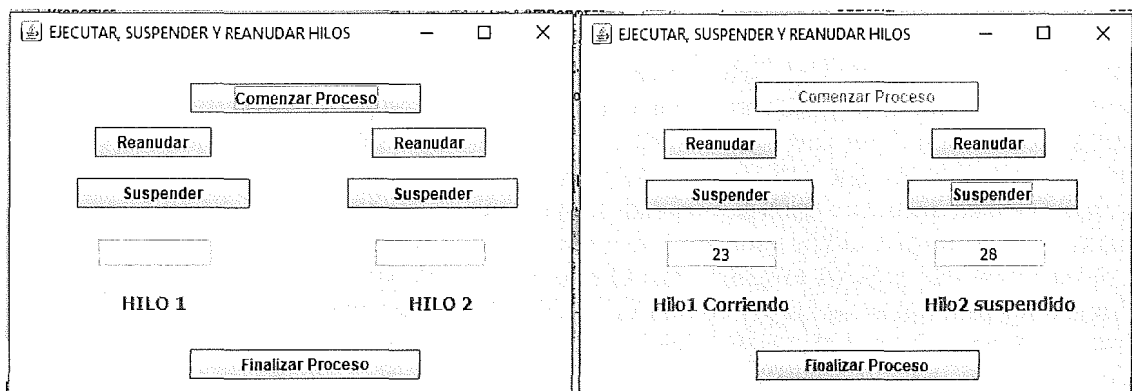


Figura 2.10. Ejercicio 8.

9º) Realiza una pantalla gráfica para iniciar dos hilos y finalizar su ejecución usando interrupciones. Se deben mostrar varios botones, véase Figura 2.11. El botón *Comenzar Proceso* crea los dos hilos y lanza su ejecución, los hilos sólo se crean una vez, el botón se desactivará al pulsarle. Cada hilo tendrá su botón para interrumpir su ejecución. Se debe mostrar un mensaje en pantalla que indique si el hilo está corriendo o se ha sido interrumpida su ejecución. El botón *Finalizar Proceso* detiene los dos hilos y muestra en consola el valor final de cada contador. El cierre de la ventana hace lo mismo. El constructor del hilo recibe dos parámetros, uno con el nombre del hilo y el segundo la cantidad de milisegundos que permanece el hilo dormido.

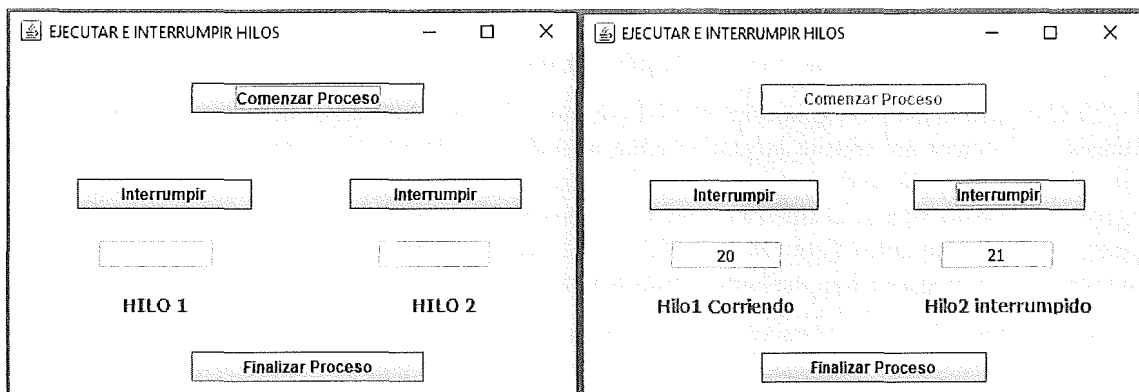


Figura 2.19. Ejercicio 9.

10º) Usando el modelo productor-consumidor, crea un productor que lea caracteres de un fichero de texto cuyo nombre se pasará en el constructor. Y un consumidor que obtenga los datos que produce el productor y los visualice en pantalla. Muestra al final del proceso del productor y del consumidor un mensaje indicando que el proceso ha finalizado. Prueba el programa con varios consumidores, ¿finalizan el proceso todos los consumidores?; utiliza el método `getState()` para comprobar el estado de los consumidores cuando el productor finaliza. Intenta que todos los consumidores finalicen correctamente.

11º) Se trata de simular el juego para adivinar un número. Se crearán varios hilos, los hilos son los jugadores que tienen que adivinar el número. Habrá un árbitro que generará el número a adivinar, comprobará la jugada del jugador y averiguará a qué jugador le toca jugar. El número tiene que estar comprendido entre 1 y 10, usa la siguiente fórmula para generar el número:  $1 + (int) (10 * Math.random())$ ;

Se definen 3 clases:

- *Árbitro*: Contiene el número a adivinar, el turno y muestra el resultado. Se definen los siguientes atributos: el número total de jugadores, el turno, el número a adivinar y si el juego acabó o no. En el constructor se recibe el número de jugadores que participan y se inicializan el número a adivinar y el turno. Tiene varios métodos: uno que devuelve el turno, otro que indica si el juego se acabó o no y el tercer método que comprueba la jugada del jugador y averigua a quien le toca a continuación, este método recibirá el identificador de jugador y el número que ha jugado; deberá definirse como **synchronized**, así cuando un jugador está haciendo la jugada, ningún otro podrá interferir. En este método se indicará cual es el siguiente turno y si el juego ha finalizado porque algún jugador ha acertado el número.
- *Jugador*: Extiende **Thread**. Su constructor recibe un identificador de jugador y el árbitro, todos los hilos comparten el árbitro. El jugador dentro del método `run` comprobará si es su turno, en ese caso generará un número aleatorio entre 1 y 10 y creará la jugada usando el método correspondiente del árbitro. Este proceso se repetirá hasta que el juego se acabe.
- *Main*: Esta clase inicializa el árbitro indicándole el número de jugadores y lanza los hilos de los jugadores, asignando un identificador a cada hilo y enviándoles el objeto árbitro que tienen que compartir.

Ejemplo de salida al ejecutar el programa:

```
NÚMERO A ADIVINAR: 3
Jugador1 dice: 9
    Le toca a Jug2
Jugador2 dice: 9
    Le toca a Jug3
Jugador3 dice: 10
    Le toca a Jug1
Jugador1 dice: 4
    Le toca a Jug2
Jugador2 dice: 7
    Le toca a Jug3
Jugador3 dice: 7
    Le toca a Jug1
Jugador1 dice: 6
    Le toca a Jug2
Jugador2 dice: 3
    Jugador 2 gana, adivinó el número!!!
```

