Programación concurrente

Java y Servicios Web I
Master en Ingeniería Matemática

Manuel Montenegro
Dpto. Sistemas Informáticos y Computación

Desp. 467 (Mat)
montenegro@fdi.ucm.es

Programación concurrente

- Consiste en la ejecución simultánea de varias tareas, que pueden interactuar entre sí.
  - Paso de mensajes.
  - Acceso a memoria compartida.
- Un hilo (*thread*) es una tarea (conjunto de instrucciones) que puede ejecutarse en paralelo con las demás.
- Estos hilos pueden ser todos ejecutados por un sólo procesador, o distribuidos entre los distintos procesadores.
  - La asignación de hilos a procesadores es realizada por el sistema operativo.

Contenidos

- Operaciones con hilos
- Acceso a memoria compartida
  - Condiciones de carrera
  - Secciones críticas
- Multitarea con *Swing*

Creación de hilos

- Para crear un hilo, se deberá crear una clase que implemente la interfaz `Runnable`.

```
public interface Runnable {
  void run();
}
```

- La clase ha de tener un método `run()`, que realizará las acciones correspondientes.

Creación de hilos

- Para definir las acciones que se realizan en un hilo, se deberá crear una clase que implemente la interfaz `Runnable`.

```
public interface Runnable {
  void run();
}
```

- La clase ha de tener un método `run()`, que realizará las acciones correspondientes.

Ejemplo

```
public class Enanito implements Runnable {
    private String nombre;

    public Enanito(String nombre) {
        this.nombre = nombre;
    }

    public void run() {
        for (int contador = 1; contador <= 10; contador++) {
            System.out.printf("%s cuenta hasta %d\n", nombre, contador);
        }
        System.out.printf("%s terminó\n", nombre);
    }
}
```

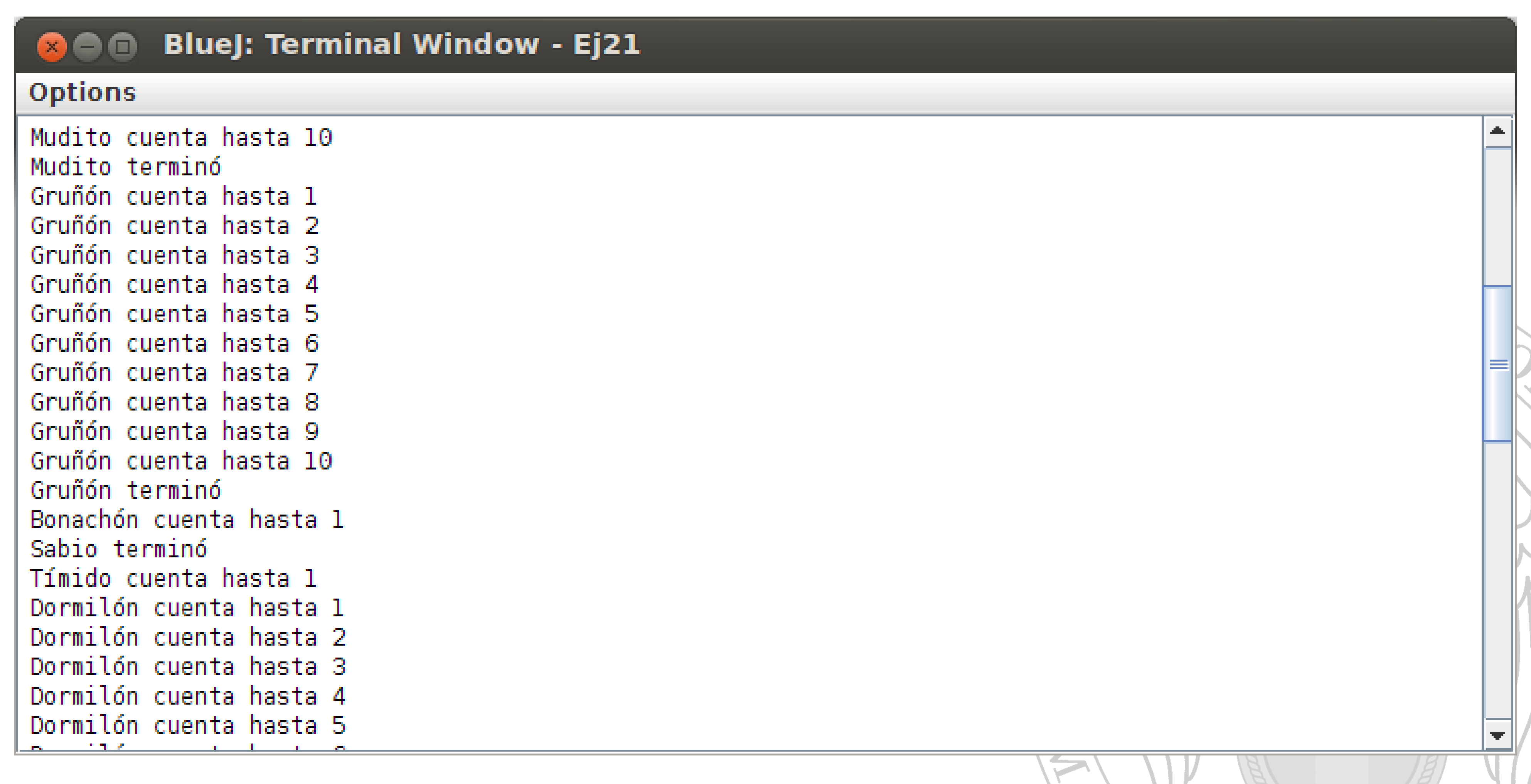
Creación de hilos

- El método tradicional de creación de hilos consiste en crear una instancia de la clase `Thread`, y pasarle el objeto `Runnable` correspondiente mediante su constructor.
- A continuación se debe llamar al método `start()` del objeto thread correspondiente para iniciar su ejecución.

Ejemplo

```
public class Test1 {
    public static void main(String[] args) {
        String[] nombres = {"Sabio", "Gruñón", "Mudito",
                            "Dormilón", "Tímido", "Tontín", "Bonachón"};
        for (String nomb : nombres) {
            Enanito e = new Enanito(nomb);
            Thread t = new Thread(e);
            t.start();
        }
    }
}
```

Ejemplo



Creación de hilos

- Desde la versión 5 de Java, no es necesario crear objetos de la clase `Thread` directamente.
- Existe un objeto `ExecutorService` (paquete `java.util.concurrent`) que recibe las tareas a ejecutar (objetos que implementan Runnable) y las distribuye en distintos hilos.
- Existen varios tipos de `ExecutorService`
  - Implementan distintas políticas de distribución de tareas en hilos.
- Un `ExecutorService` no tiene constructor. Se crea mediante métodos estáticos contenidos en la clase `Executors`.

Ejemplo

```
public class Test2 {
    public static void main(String[] args) {
        String[] nombres = {"Sabio", "Gruñón", "Mudito",
                            "Dormilón", "Tímido", "Tontín", "Bonachón"};
        ExecutorService exec = Executors.newCachedThreadPool();
        for (String n : nombres) {
            Enanito e = new Enanito(n);
            exec.execute(e);
        }
        exec.shutdown();
    }
}
```

No admitir más tareas en el Executor

Tipos de ExecutorService

- `CachedThreadPool`
  - Genera tantos hilos como considere necesarios.
  - Procura reciclar los hilos cuya tarea ha terminado.
- `FixedThreadPool`
  - Crea, al inicio, una cantidad de hilos fija.
- `SingleThreadExecutor`
  - Como un `FixedThreadPool`, pero con un único hilo.
  - Si se envían varias tareas a un `SingleThreadExecutor`, se pondrán en cola.

Dormir un hilo

```
public class Enanito implements Runnable {
    …

    public void run() {
        for (int contador = 1; contador <= 10; contador++) {
            System.out.printf("%s cuenta hasta %d\n", nombre, contador);
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                System.out.println("Se ha interrumpido");
            }
        }
        System.out.printf("%s terminó\n", nombre);
    }
}
```

Detener hilo 1 seg.

Prioridad de hilos

```
public class Enanito implements Runnable {
    …

    public void run() {
        if (nombre.equals("Acaparadorcito")) {
            Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
        } else {
            Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
        }
        for (int contador = 1; contador <= 10; contador++) {
            System.out.printf("%s cuenta hasta %d\n", nombre, contador);
        }
        System.out.printf("%s terminó\n", nombre);
    }
}
```

Prioridad de hilos

```
public class Test2 {
    public static void main(String[] args) {
        String[] nombres = {"Sabio", “Acaparadorcito”, "Gruñón", "Mudito",
                            "Dormilón", "Tímido", "Tontín", "Bonachón"};
        ExecutorService exec = Executors.newCachedThreadPool();
        for (String n : nombres) {
            Enanito e = new Enanito(n);
            exec.execute(e);
        }
        exec.shutdown();
    }
}
```

Esperar la finalización de un hilo

- El método `awaitTermination` de `ExecutorService` permite detener el hilo que ejecuta dicho método hasta que los hilos del servicio ejecutor terminen.
- Recibe como parámetro una cantidad de tiempo máximo a esperar, cuya unidad de medida es el segundo parámetro.
- Lanza una excepción `InterruptedException` si el la espera se interrumpe externamente.

VNIVERSITAS COMPLVTENSIS MATRITENSIS
LIBERTAS PERFVNDET OMNIA LVCE

Esperar la finalización de un hilo

```
public class Test2 {
    public static void main(String[] args) {
        String[] nombres = {"Sabio", "Gruñón", "Mudito",
                            "Dormilón", "Tímido", "Tontín", "Bonachón"};
        ExecutorService exec = Executors.newCachedThreadPool();
        for (String n : nombres) {
            Enanito e = new Enanito(n);
            exec.execute(e);
        }
        exec.shutdown();

        try {
            exec.awaitTermination(100, TimeUnit.SECONDS);
            System.out.println("Los siete enanitos han terminado");
        } catch(InterruptedException e) {
            System.out.println("Se interrumpió");
        }
    }
}
```

Contenidos

- Operaciones con hilos
- Acceso a memoria compartida
  - Condiciones de carrera
  - Secciones críticas
- Multitarea con *Swing*

Acceso a memoria compartida

- Supongamos que ahora los hilos manipulan una zona de memoria común.

```
public class Contador {
    private int valor;

    public Contador() { valor = 0;  }

    public void incrementar() { valor++;  }

    public int getValor() { return valor;  }

}
```

Acceso a memoria compartida

```
public class Enanito implements Runnable {
    private String nombre;
    private Contador contador;

    public Enanito(String nombre, Contador contador) {
        this.nombre = nombre;
        this.contador = contador;
    }

    public void run() {
        for (int i = 0; i < 1000; i++)
            contador.incrementar();
    }
}
```

Cada enanito incrementa el contador 1000 veces

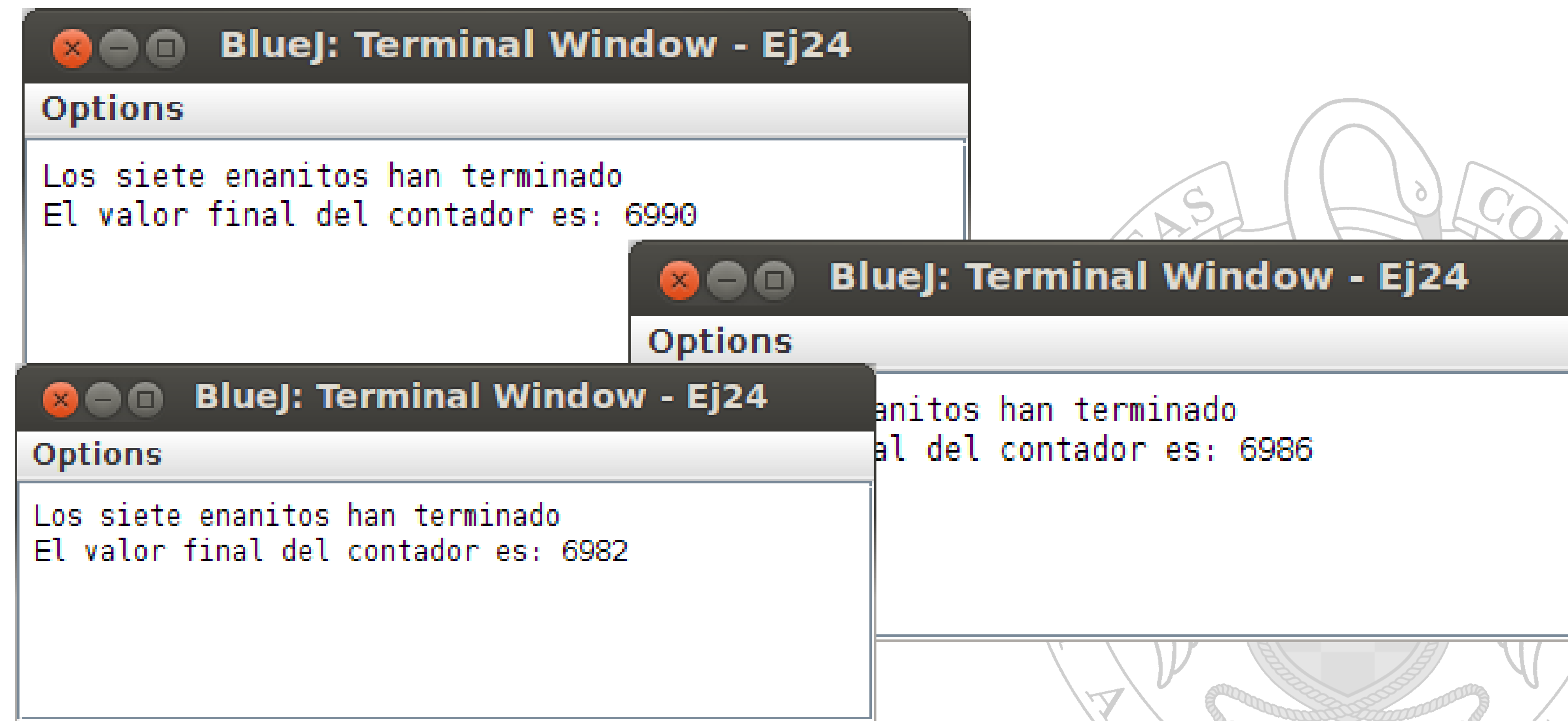
Acceso a memoria compartida

```
public class Test2 {
    public static void main(String[] args) {
        String[] nombres = {"Sabio", "Gruñón", "Mudito",
                            "Dormilón", "Tímido", "Tontín", "Bonachón"};
        Contador c = new Contador();
        ExecutorService exec = Executors.newCachedThreadPool();
        for (String n : nombres) {
            Enanito e = new Enanito(n, c);
            exec.execute(e);
        }
        exec.shutdown();
        try {
            exec.awaitTermination(100, TimeUnit.SECONDS);
            System.out.println("Los siete enanitos han terminado");
        } catch(InterruptedException e) {
            System.out.println("Se interrumpió");
        }
        System.out.printf("El valor final del contador es: %d\n", c.getValor());
    }
}
```

Todos los enanitos incrementan el mismo contador

Acceso a memoria compartida

- El valor final del contador debería ser 7000.
- Sin embargo:



Condiciones de carrera

contador = 0

...

...

contador++

...

...

Condiciones de carrera

contador = 0

...

...
LOAD contador
ADD 1
STORE contador

...

...

Condiciones de carrera

contador = 0

...

...

Registro = 0 ▶ T1 LOAD contador

ADD 1

STORE contador

...

...

Condiciones de carrera

contador = 0

...

...
LOAD contador
Registro = 1 ADD 1
STORE contador

...

...

Condiciones de carrera

contador = 1

T2 ...

...
LOAD contador
ADD 1
Registro = 1 ▶ T1 STORE contador

...

...

Condiciones de carrera

contador = 1

T2 ...

...
LOAD contador
ADD 1
STORE contador

Registro = 1 T1 ...

...

Condiciones de carrera

contador = 1

...

...

Registro = 1 ▶ T2 LOAD contador

ADD 1

STORE contador

Registro = 1 ‖ T1 ...

...

Condiciones de carrera

contador = 1

...

...
LOAD contador
Registro = 2

ADD 1
STORE contador

Registro = 1

...

...

Condiciones de carrera

contador = 2

...

...
LOAD contador
ADD 1
STORE contador

...

...

Registro = 2 T2

Registro = 1 T1

Condiciones de carrera

contador = 2 OK

...

...
LOAD contador
ADD 1
STORE contador

Registro = 1 T1 ...

Registro = 2 T2 ...

Condiciones de carrera

contador = 0

...

T1

...
LOAD contador
ADD 1
STORE contador

...

...

Condiciones de carrera

contador = 0

. . .

. . .

Registro = 0 ▶ T1 LOAD contador

ADD 1

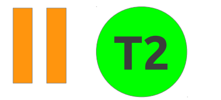
STORE contador

. . .

. . .

Condiciones de carrera

contador = 0



...

...

LOAD contador

Registro = 1 ▶ T1 ADD 1

STORE contador

...

...

Condiciones de carrera

contador = 0

. . .

. . .

LOAD contador

Registro = 1 T1 ADD 1

STORE contador

. . .

. . .

Condiciones de carrera

contador = 0

...

...

Registro = 0 — T2 — LOAD contador

Registro = 1 — T1 — ADD 1

STORE contador

...

...

Condiciones de carrera

contador = 1

...

...

LOAD contador

Registro = 1 ▶ T2 ADD 1

STORE contador

...

...

Condiciones de carrera

contador = 1

...

...
LOAD contador
Registro = 1 T1 ADD 1
Registro = 1 T2 STORE contador

...

...

Condiciones de carrera

contador = 1

...

...
LOAD contador
ADD 1
STORE contador

Registro = 1 T1

...

Registro = 1 T2 ...

Condiciones de carrera

contador = 1

...

...
LOAD contador
Registro = 1 ▶ T1 ADD 1
STORE contador

...

Registro = 1 T2 ...

Condiciones de carrera

contador = 1

...

...
LOAD contador
ADD 1
Registro = 1 ▶ T1 STORE contador

...

Registro = 1 ‖ T2 ...

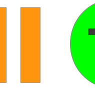
Condiciones de carrera

contador = 1

...

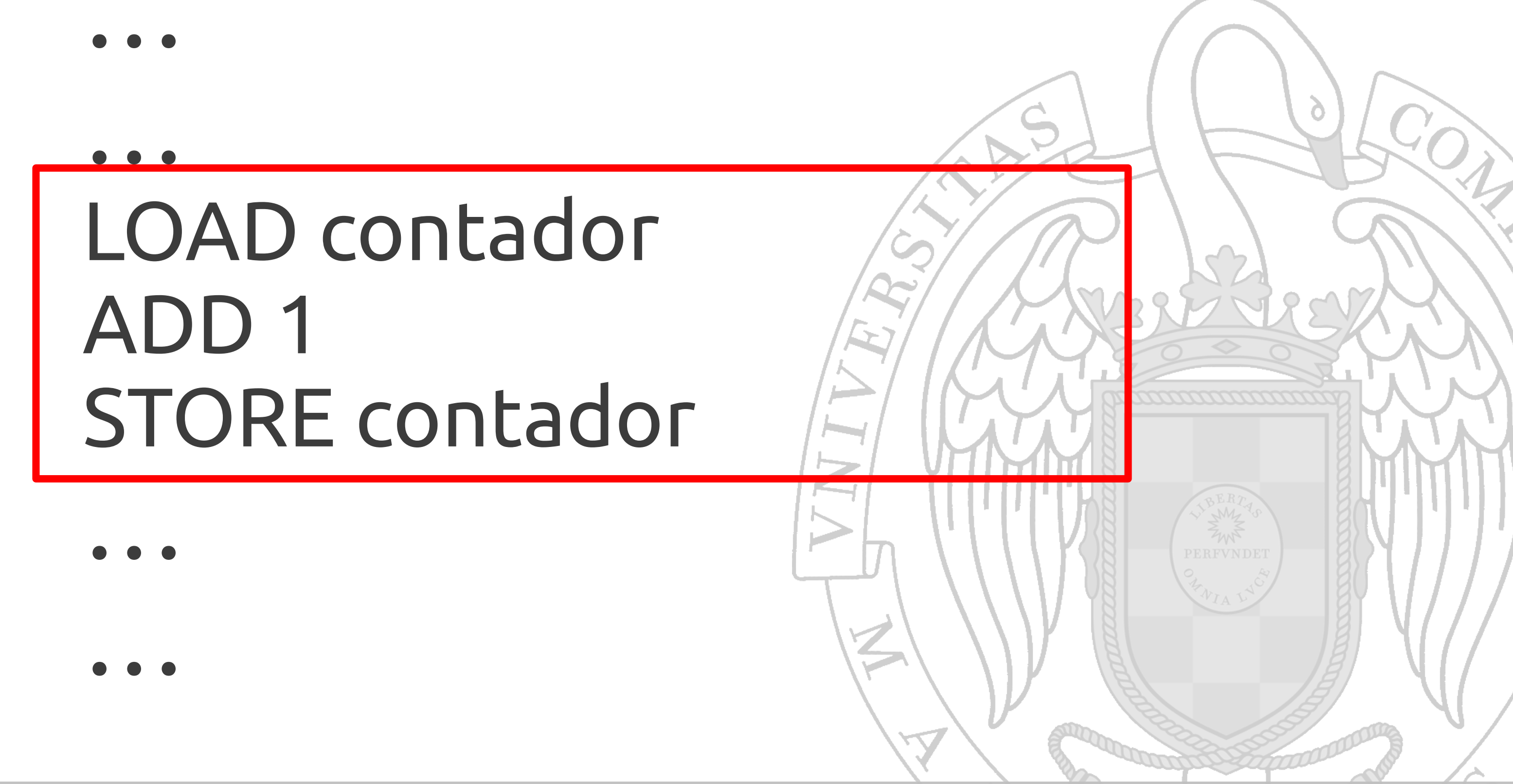
...
LOAD contador
ADD 1
STORE contador

Registro = 1 T1 ...

Registro = 1  T2 ...

Secciones críticas

- Idealmente, desearíamos que sólo un hilo pudiese ejecutar simultáneamente a determinadas regiones del programa → exclusión mutua.



```
...
...
LOAD contador
ADD 1
STORE contador
...
...
```

Secciones críticas

- Java proporciona un mecanismo de exclusión mutua integrado en el lenguaje.
- Cada objeto tiene asociado un candado (*lock*, o *monitor*).
- Se utiliza la palabra `synchronized` para especificar zonas de exclusión mutua dentro del código.

```
synchronized (objeto) {
    sentencias;
}
```

Secciones críticas

```
synchronized (objeto) {
    sentencias;
}
```

- Cuando la ejecución de un hilo llega a un bloque `synchronized`, adquiere el candado del objeto correspondiente, y lo libera cuando abandona dicho bloque.
- Ningún otro hilo podrá entrar en el bloque `synchronized` hasta que el hilo poseedor del candado lo libere.

Ejemplo

```
public class Enanito implements Runnable {
    …


    public void run() {
        for (int i = 0; i < 1000; i++) {
            synchronized(contador) {
                contador.incrementar();
            }
        }
    }
}
```

Ejemplo

```
public class Enanito implements Runnable {
    …


    public void run() {
        for (int i = 0; i < 1000; i++) {
            synchronized(contador) {
                contador.incrementar();
            }
        }
    }
}
```

BlueJ: Terminal Window - Ej24

Options

```
Los siete enanitos han terminado
El valor final del contador es: 7000
```

Solución alternativa

- Es aconsejable delimitar la zona de exclusión mutua en el código del propio contador, en lugar de en el código de los enanitos.

```
public class Contador {

    private int valor;
    public Contador() { valor = 0;  }

    public void incrementar() {
        synchronized(this) {
            valor++;
        }
    }

    public int getValor() { return valor;  }
}
```

Solución alternativa

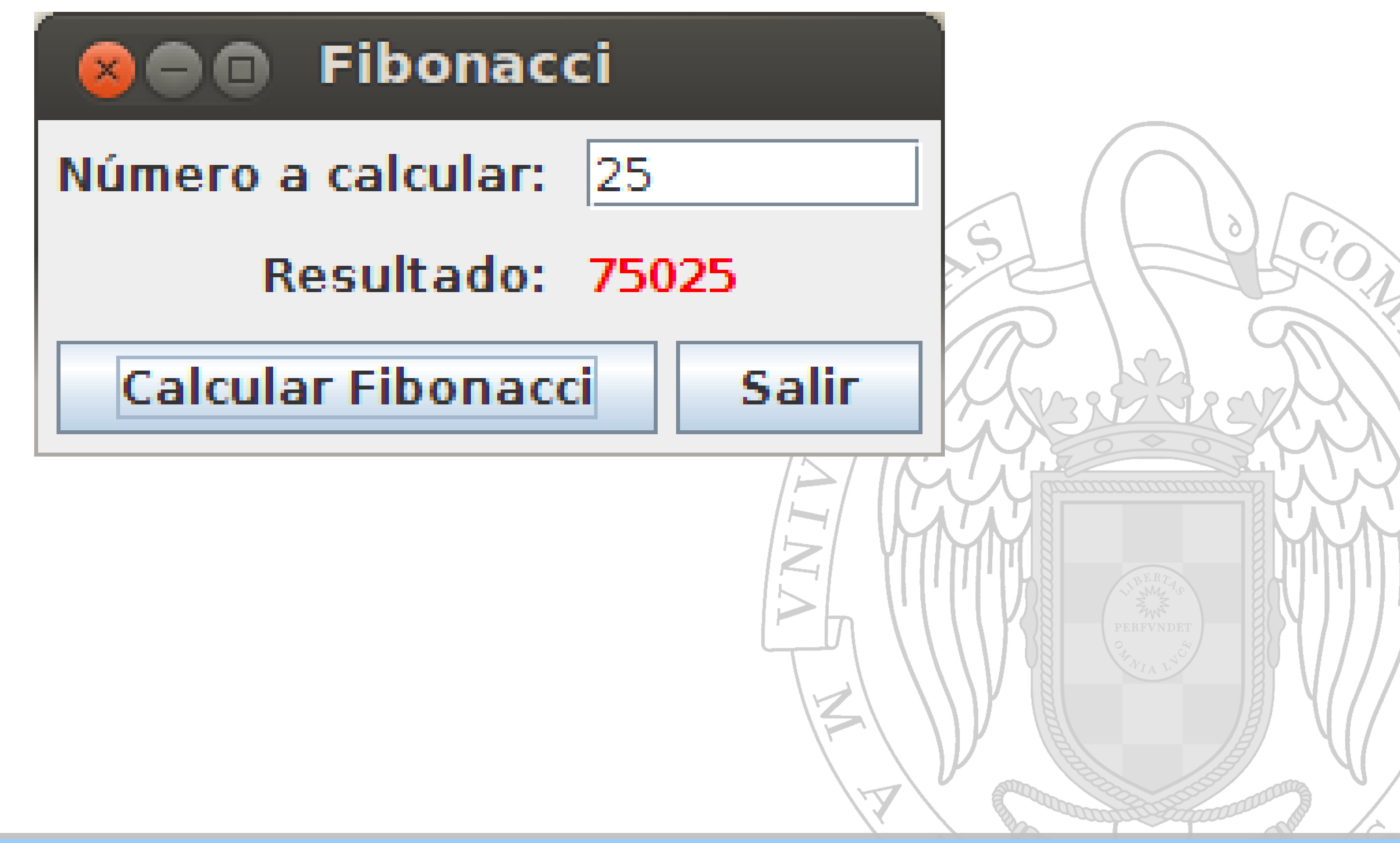
- Es aconsejable delimitar la zona de exclusión mutua en el código del propio contador, en lugar de en el código de los enanitos.

```
public class Contador {

    private int valor;
    public Contador() { valor = 0;  }

    public synchronized void incrementar() {
            valor++;
    }

    public int getValor() { return valor;  }
}
```

Contenidos

- Operaciones con hilos
- Acceso a memoria compartida
  - Condiciones de carrera
  - Secciones críticas
- Multitarea con *Swing*

Multitarea en *Swing*



Multitarea en *Swing*

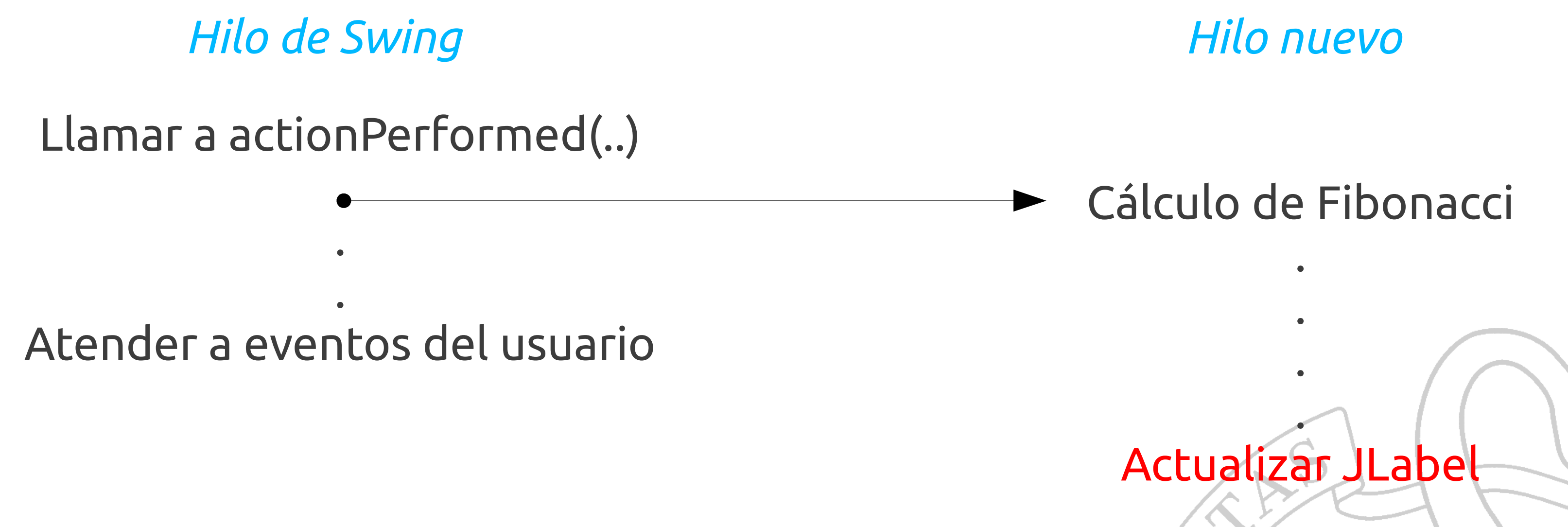
```
class GestorBotonOK implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String entrada = jTextField1.getText();
        if (entrada.isEmpty()) {
            JOptionPane.showMessageDialog(null, "No has introducido ningún número");
        } else {
            try {
                int n = Integer.parseInt(entrada);
                long result = fib(n);
                jLabel3.setText(String.valueOf(result));
            } catch (NumberFormatException exc) {
                JOptionPane.showMessageDialog(null, "Has de introducir un número");
            }
        }
    }

    public long fib(int n) {
      …
    }
}
```

Cómputo largo

Multitarea en *Swing*

- El cálculo del número de Fibonacci indicado se realiza desde el mismo hilo que se encarga de atender los eventos de la interfaz.
- Mientras se realiza el cómputo del número de Fibonacci indicado, la interfaz no responde.
- Es necesario realizar dicho cómputo en un hilo, en paralelo con la gestión de eventos de la interfaz.

Multitarea en *Swing*



- Los componentes de *Swing* no ofrecen acceso concurrente. Sólo se pueden modificar los componentes de *Swing* desde el hilo de *Swing*.

Multitarea en *Swing*

- Extendiendo la clase `SwingWorker` podemos crear tareas que se ejecuten en paralelo, y puedan modificar los componentes de la interfaz de manera segura.

Multitarea en *Swing*

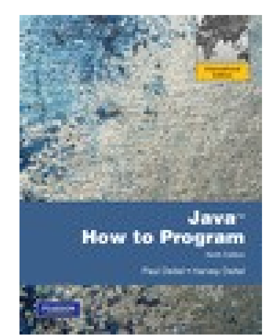
```
public class CalculadorFibonacci extends SwingWorker<Long, Object>
{
    private int n;
    private JLabel resultado;

    public CalculadorFibonacci(int n, JLabel resultado) {
        this.n = n; this.resultado = resultado;
    }

    public Long doInBackground() { return fib(n);  }

    protected void done() {
        try {
            Long res = get();
            resultado.setText(res.toString());
        } catch (InterruptedException e) {
            JOptionPane.showMessageDialog(null, "Interrumpido: " + e.getMessage());
        } catch (ExecutionException e) {
            JOptionPane.showMessageDialog(null, "Error de ejecución: " + e.getMessage());
        }
    }

    private long fib(int n) { … }
}
```

Multitarea en *Swing*

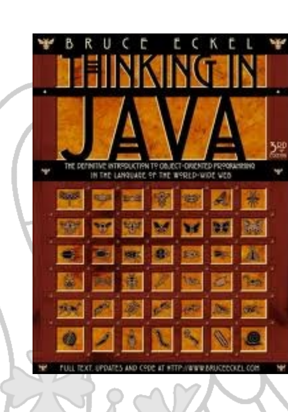
```
class GestorBotonOK implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String entrada = jTextField1.getText();
        if (entrada.isEmpty()) {
            JOptionPane.showMessageDialog(null, "No has introducido ningún número");
        } else {
            try {
                int n = Integer.parseInt(entrada);
                CalculadorFibonacci calc = new CalculadorFibonacci(n, jLabel3);
                calc.execute();
            } catch (NumberFormatException exc) {
                JOptionPane.showMessageDialog(null, "Has de introducir un número");
            }
        }
    }
}
```

Referencias

- P. Deitel, H. Deitel
  Java. How to Program (9th Edition)
  Caps. 26



- B. Eckel
  Thinking in Java (3rd Edition)
  Cap. 13



- Sincronización mediante paso de mensajes.
  http://mpj-express.org/