# Programación de Servicios y Procesos

*Publicación 1.0*

**Oscar Gomez**

2014-2015

Índice general

CAPÍTULO 1

# Programación multiproceso

## 1.1 Ejecutables. Procesos. Servicios.

### 1.1.1 Ejecutables

Un ejecutable es un archivo con la estructura necesaria para que el sistema operativo pueda poner en marcha el programa que hay dentro. En Windows, los ejecutables suelen ser archivos con la extension .EXE.

Sin embargo, Java genera ficheros .JAR o .CLASS. Estos ficheros *no son ejecutables* sino que son archivos que el intérprete de JAVA (el archivo `java.exe`) leerá y ejecutará.

El intérprete toma el programa y lo traduce a instrucciones del microprocesador en el que estemos, que puede ser x86 o un x64 o lo que sea. Ese proceso se hace "al instante" o JIT (Just-In-Time).

Un EXE puede que no contenga las instrucciones de los microprocesadores más modernos. Como todos son compatibles no es un gran problema, sin embargo, puede que no aprovechemos al 100 % la capacidad de nuestro micro.

### 1.1.2 Procesos

Es un archivo que está en ejecución y bajo el control del sistema operativo. Un proceso puede atravesar diversas etapas en su "ciclo de vida". Los estados en los que puede estar son:

- En ejecución: está dentro del microprocesador.
- Pausado/detenido/en espera: el proceso tiene que seguir en ejecución pero en ese momento el S.O tomó la decisión de dejar paso a otro.
- Interrumpido: el proceso tiene que seguir en ejecución pero *el usuario* ha decidido interrumpir la ejecución.
- Existen otros estados pero ya son muy dependientes del sistema operativo concreto.

### 1.1.3 Servicios

Un servicio es un proceso que no muestra ninguna ventana ni gráfico en pantalla porque no está pensado para que el usuario lo maneje directamente.

Habitualmente, un servicio es un programa que atiende a otro programa.

## 1.2 Hilos.

Un hilo es un concepto más avanzado que un proceso: al hablar de procesos cada uno tiene su propio espacio en memoria. Si abrimos 20 procesos cada uno de ellos consume 20x de memoria RAM. Un hilo es un proceso mucho más ligero, en el que el código y los datos se comparten de una forma distinta.

Un proceso no tiene acceso a los datos de otro procesos. Sin embargo un hilo sí accede a los datos de otro hilo. Esto complicará algunas cuestiones a la hora de programar.

## 1.3 Programación concurrente.

La programación concurrente es la parte de la programación que se ocupa de crear programas que pueden tener varios procesos/hilos que colaboran para ejecutar un trabajo y aprovechar al máximo el rendimiento de sistemas multinúcleo.

## 1.4 Programación paralela y distribuida.

Dentro de la programación concurrente tenemos la paralela y la distribuida:

- En general se denomina "programación paralela" a la creación de software que se ejecuta siempre en un solo ordenador (con varios núcleos o no)
- Se denomina "programación distribuida" a la creación de software que se ejecuta en ordenadores distintos y que se comunican a través de una red.

## 1.5 Creación de procesos.

En Java es posible crear procesos utilizando algunas clases que el entorno ofrece para esta tarea. En este tema, veremos en profundidad la clase ProcessBuilder.

El ejemplo siguiente muestra como lanzar un proceso de Acrobat Reader:

```
public class LanzadorProcesos {
        public void ejecutar(String ruta){

                ProcessBuilder pb;
```

```
                try {
                        pb = new ProcessBuilder(ruta);
                        pb.start();
                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
        /**
         * @param args
         */
        public static void main(String[] args) {
                String ruta=
                        "C:\\Program Files (x86)\\Adobe\\Reader 11.0\\Reader\\Ac
                LanzadorProcesos lp=new LanzadorProcesos();
                lp.ejecutar(ruta);
                System.out.println("Finalizado");
        }

}
```

Supongamos que necesitamos crear un programa que aproveche al máximo el número de CPUs para realizar alguna tarea intensiva. Supongamos que dicha tarea consiste en sumar números.

Enunciado: crear una clase Java que sea capaz de sumar todos los números comprendidos entre dos valores incluyendo ambos valores.

Para resolverlo crearemos una clase `Sumador` que tenga un método que acepte dos números `n1` y `n2` y que devuelva la suma de todo el intervalor.

Además, incluiremos un método `main` que ejecute la operación de suma tomando los números de la línea de comandos (es decir, se pasan como argumentos al main).

El código de dicha clase podría ser algo así:

```
package com.ies;

public class Sumador {
        public int sumar(int n1, int n2){
                int resultado=0;
                for (int i=n1;i<=n2;i++){
                        resultado=resultado+i;
                }
                return resultado;
        }
        public static void main(String[] args){
                Sumador s=new Sumador();
                int n1=Integer.parseInt(args[0]);
                int n2=Integer.parseInt(args[1]);
                int resultado=s.sumar(n1, n2);
                System.out.println(resultado);
```

```
        }
}
```

Para ejecutar este programa desde dentro de Eclipse es necesario indicar que deseamos enviar *argumentos* al programa. Por ejemplo, si deseamos sumar los números del 2 al 10, deberemos ir a la venta "Run configuration" y en la pestaña "Arguments" indicar los argumentos (que en este caso son los dos números a indicar).

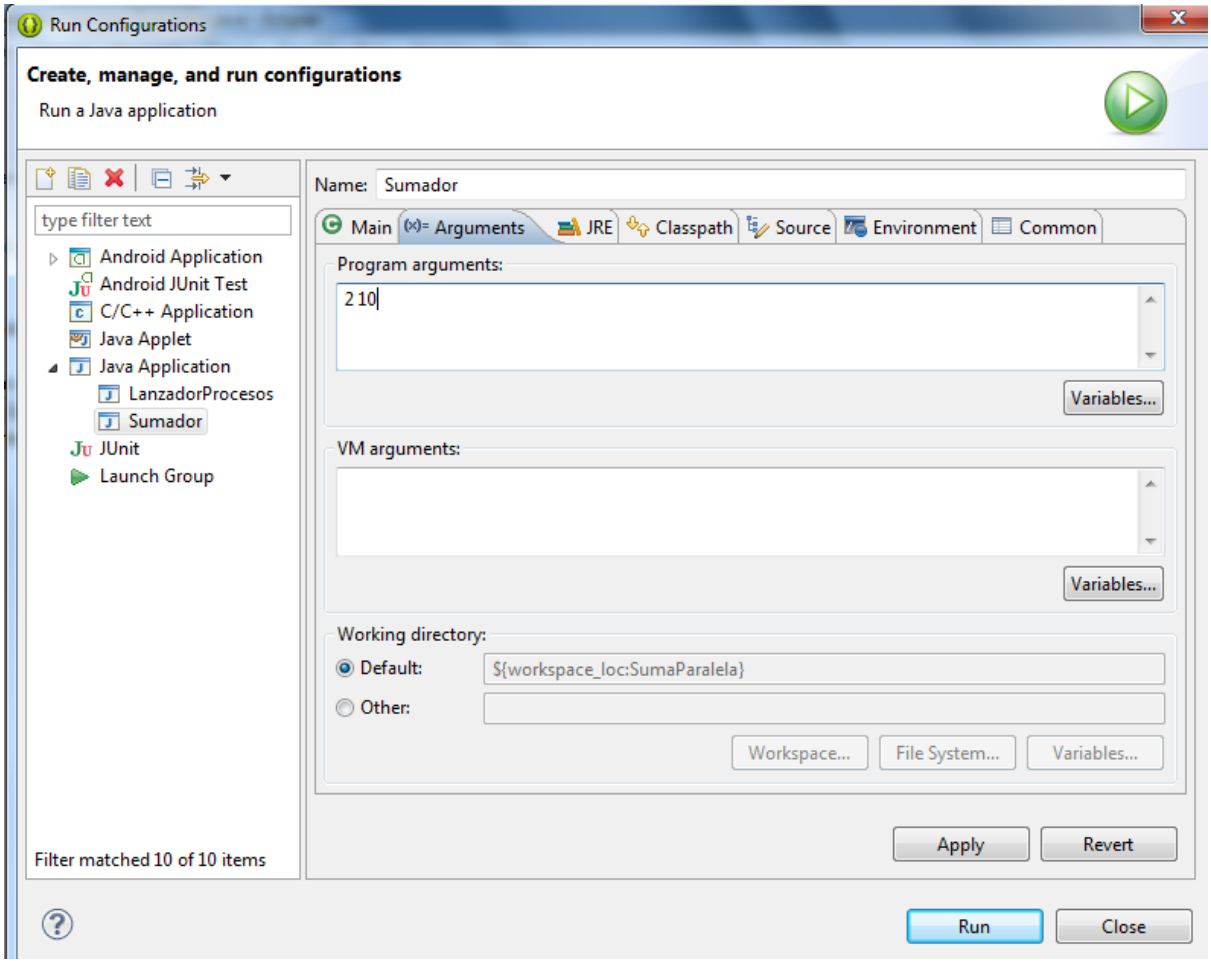


Figura 1.1: Modificando los argumentos del programa

Una vez hecha la prueba de la clase sumador, le quitamos el main, y crearemos una clase que sea capaz de lanzar varios procesos. La clase `Sumador` se quedará así:

```
public class Sumador {
        public int sumar(int n1, int n2){
                int resultado=0;
                for (int i=n1;i<=n2;i++){
                        resultado=resultado+i;
                }
                return resultado;
        }
}
```

Y ahora tendremos una clase que lanza procesos de esta forma:

```
package com.ies;

public class Lanzador {
        public void lanzarSumador(Integer n1,
                        Integer n2){
                String clase="com.ies.Sumador";
                ProcessBuilder pb;
                try {
                        pb = new ProcessBuilder(
                                        "java",clase,
                                        n1.toString(),
                                        n2.toString());
                        pb.start();
                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
        public static void main(String[] args){
                Lanzador l=new Lanzador();
                l.lanzarSumador(1, 51);
                l.lanzarSumador(51, 100);
                System.out.println("Ok");
        }
}
```

## 1.6 Comunicación entre procesos.

Las operaciones multiproceso pueden implicar que sea necesario comunicar información entre muchos procesos, lo que obliga a la necesidad de utilizar mecanismos específicos de comunicación que ofrecerá Java o a diseñar alguno separado que evite los problemas que puedan aparecer.

En el ejemplo, el segundo proceso suele sobreescribir el resultado del primero, así que modificaremos el código del lanzador para que cada proceso use su propio fichero de resultados.

```
public class Lanzador {
        public void lanzarSumador(Integer n1,
                        Integer n2, String fichResultado){
                String clase="com.ies.Sumador";
                ProcessBuilder pb;
                try {
                        pb = new ProcessBuilder(
                                        "java",clase,
                                        n1.toString(),
                                        n2.toString());

                        pb.redirectError(new File("errores.txt"));
                        pb.redirectOutput(new File(fichResultado));
```

```
                pb.start();
            } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        public static void main(String[] args){
            Lanzador l=new Lanzador();
            l.lanzarSumador(1, 5, "result1.txt");
            l.lanzarSumador(6,10, "result2.txt");
            System.out.println("Ok");
        }
}
```

Cuando se lanza un programa desde Eclipse no ocurre lo mismo que cuando se lanza desde Windows. Eclipse trabaja con unos directorios predefinidos y puede ser necesario indicar a nuestro programa cual es la ruta donde hay que buscar algo.

Usando el método `.directory(new File("c:\\dir\\))` se puede indicar a Java donde está el archivo que se desea ejecutar.

## 1.7 Gestión de procesos.

La gestión de procesos se realiza de dos formas **muy distintas** en función de los dos grandes sistemas operativos: Windows y Linux.

- En Windows toda la gestión de procesos se realiza desde el "Administrador de tareas" al cual se accede con Ctrl+Alt+Supr. Existen otros programas algo más sofisticados que proporcionan algo más de información sobre los procesos, como Processviewer.

## 1.8 Comandos para la gestión de procesos en sistemas libres y propietarios.

En sistemas Windows, no existen apenas comandos para gestionar procesos. Puede obligarse al sistema operativo a arrancar la aplicación asociada a un archivo con el comando `START`. Es decir, si se ejecuta lo siguiente:

```
START documento.pdf
```

se abrirá el visor de archivos PDF el cual cargará automáticamente el fichero `documento.pdf`

En GNU/Linux se puede utilizar un terminal de consola para la gestión de procesos, lo que implica que no solo se pueden arrancar procesos si no tambien detenerlos, reanudarlos, terminarlos y modificar su prioridad de ejecución.

- Para arrancar un proceso, simplemente tenemos que escribir el nombre del comando correspondiente. Desde GNU/Linux se pueden controlar los servicios que se ejecutan

con un comando llamado `service`. Por ejemplo, se puede usar `sudo service apache2 stop` para parar el servidor web y `sudo service apache2 start` para volver a ponerlo en marcha. También se puede reiniciar un servicio (tal vez para que relea un fichero de configuración que hemos cambiado) con `sudo service apache2 restart`.

- Se puede detener y/o terminar un proceso con el comando `kill`. Se puede usar este comando para **terminar un proceso** sin guardar nada usando `kill -SIGKILL <numproceso>` o `kill -9 <numproceso>`. Se puede pausar un proceso con `kill -SIGPAUSE <numproceso>` y rearrancarlo con `kill -SIGCONT`
- Se puede enviar un proceso a segundo plano con comandos como `bg` o al arrancar el proceso escribir el nombre del comando terminado en `&`.
- Se puede devolver un proceso a primer plano con el comando `fg`.

### 1.8.1 Prioridades

En sistemas como GNU/Linux se puede modificar la prioridad con que se ejecuta un proceso. Esto implica dos posibilidades

- Si pensamos que un programa que necesitamos ejecutar es muy importante podemos darle más prioridad para que reciba "más turnos" del planificador.
- Y por el contrario, si pensamos que un programa no es muy necesario podemos quitarle prioridad y reservar "más turnos de planificador" para otros posibles procesos.

El comando `nice` permite indicar prioridades entre -20 y 19. El -20 implica que un proceso reciba la **máxima prioridad**, y el 19 supone asignar la **mínima prioridad**

## 1.9 Sincronización entre procesos.

Cuando se lanza más de un proceso de una misma sección de código no se sabe qué proceso ejecutará qué instrucción en un cierto momento, lo que es muy peligroso:

```
int i;
i=0;
if (i==0){
        i=i+1;
        j=j+1
}
System.out.println("Ok");
i=i*2;
j=j-1;
```

Si dos o más procesos avanzan por esta sección de código es perfectamente que unas veces nuestro programa multiproceso se ejecute bien y otras no.

En todo programa multiproceso pueden encontrarse estas zonas de código "peligrosas" que deben protegerse especialmente utilizando ciertos mecanismos. El nombre global para todos

los lenguajes es denominar a estos trozos "secciones críticas".

### 1.9.1 Mecanismos para controlar secciones críticas

Los mecanismos más típicos son los ofrecidos por UNIX/Windows:

- Semáforos.
- Colas de mensajes.
- Tuberías (pipes)
- Bloques de memoria compartida.

En realidad algunos de estos mecanismos se utilizan más para intercomunicar procesos, aunque para los programadores Java la forma de resolver el problema de la "sección crítica" es más simple.

En Java, si el programador piensa que un trozo de código es peligroso puede ponerle la palabra clave `synchronized` y la máquina virtual Java protege el código automáticamente.

```
/* La máquina virtual Java evitará que más de un proceso/hilo acceda a este méto
synchronized
        public void actualizarPension(int nuevoValor){
        /*..trozo de código largo omitido*/
        this.pension=nuevoValor
}


/* Otro ejemplo, ahora no hemos protegido un método entero, sino solo un pequeño
for (int i=0; i=i+1; i++){
        /* Código omitido*/
        synchronized {
                i=i*2;
                j=j+1;
        }
```

## 1.10 Documentación

Para hacer la documentación tradicionalmente hemos usado JavaDOC. Sin embargo, las versiones más modernas de Java incluyen las **anotaciones**.

Una anotación es un texto que pueden utilizar otras herramientas (no solo el Javadoc) para comprender mejor qué hace ese código o como documentarlo.

Cualquiera puede crear sus propias anotaciones simplemente definiéndolas como un interfaz Java. Sin embargo tendremos que programar nuestras propias para extraer la información que proporcionan dichas anotaciones.

## 1.11 Depuración.

¿Como se depura un programa multiproceso/multihilo? Por desgracia puede ser muy difícil:

1. No todos los depuradores son capaces.
2. A veces cuando un depurador interviene en un proceso puede ocurrir que el resto de procesos consigan ejecutarse en el orden correcto y dar lugar a que el programa parezca que funciona bien.
3. Un error muy típico es la `NullPointerException`

En general todos los fallos en un programa multiproceso vienen derivado de no usar `synchronized` de la forma correcta.

## 1.12 Examen

Se realizará el 24 de octubre.

CAPÍTULO 2

# Programación multihilo

## 2.1 Recursos compartidos por los hilos.

Cuando creamos varios objetos de una clase, puede ocurrir que varios hilos de ejecución accedan a un objeto. Es importante recordar que **todos los campos del objeto son compartidos entre todos los hilos**.

Supongamos una clase como esta:

```
public class Empleado(){
        int numHorasTrabajadas=0;
        public void int incrementarHoras(){
                numHorasTrabajadas++;
        }
}
```

Si varios hilos ejecutan sin querer el método `incrementar` ocurrirá que el número se incrementará tantas veces como procesos.

## 2.2 Estados de un hilo. Cambios de estado.

Aunque no lo vemos, un hilo cambia de estado: puede pasar de la nada a la ejecución. De la ejecución al estado "en espera". De ahí puede volver a estar en ejecución. De cualquier estado se puede pasar al estado "finalizado".

El programador no necesita controlar esto, lo hace el sistema operativo. Sin embargo un programa multihilo mal hecho puede dar lugar problemas como los siguientes:

- Interbloqueo. Se produce cuando las peticiones y las esperas se entrelazan de forma que ningún proceso puede avanzar.
- Inanición. Ningún proceso consigue hacer ninguna tarea útil y por lo tanto hay que esperar a que el administrador del sistema detecte el interbloqueo y mate procesos (o hasta que alguien reinicie el equipo).

## 2.3 Elementos relacionados con la programación de hilos. Librerías y clases.

Para crear programas multihilo en Java se pueden hacer dos cosas:

1. Heredar de la clase Thread.
2. Implementar la interfaz Runnable.

Los documentos de Java aconsejan el segundo. Lo único que hay que hacer es algo como esto.

```
class EjecutorTareaCompleja implements Runnable{
        private String nombre;
        int numEjecucion;
        public EjecutorTareaCompleja(String nombre){
                this.nombre=nombre;
        }
        @Override
        public void run() {
                String cad;
                while (numEjecucion<100){
                        for (double i=0; i<4999.99; i=i+0.04)
                        {
                                Math.sqrt(i);
                        }
                        cad="Soy el hilo "+this.nombre;
                        cad+=" y mi valor de i es "+numEjecucion;
                        System.out.println(cad);
                        numEjecucion++;
                }
        }

}
public class LanzaHilos {

        /**
         * @param args
         */
        public static void main(String[] args) {
                int NUM_HILOS=500;
                EjecutorTareaCompleja op;
                for (int i=0; i<NUM_HILOS; i++)
                {
                        op=new EjecutorTareaCompleja ("Operacion "+i);
                        Thread hilo=new Thread(op);
                        hilo.start();
                }
        }
}
```

**Advertencia:** Este código tiene un problema **muy grave** y es que no se controla el acceso a variables compartidas, es decir **HAY UNA SECCIÓN CRÍTICA QUE NO ESTÁ PROTEGIDA** por lo que el resultado de la ejecución no muestra ningún sentido aunque el programa esté bien.

## 2.4 Gestión de hilos.

Con los hilos se pueden efectuar diversas operaciones que sean de utilidad al programador (y al administrador de sistemas a veces).

Por ejemplo, un hilo puede tener un nombre. Si queremos asignar un nombre a un hilo podemos usar el método `setName("Nombre que sea")`. También podemos obtener un objeto que represente el hilo de ejecución con `getCurrentThread` que nos devolverá un objeto de la clase `Thread`.

Otra operación de utilidad al gestionar hilos es indicar la prioridad que queremos darle a un hilo. En realidad esta prioridad es indicativa, el sistema operativo no está obligado a respetarla aunque por lo general lo hacen. Se puede indicar la prioridad con `setPriority(10)`. La máxima prioridad posible es `MAX_PRIORITY`, y la mínima es `MIN_PRIORITY`.

Cuando lanzamos una operación también podemos usar el método `Thread.sleep(numero)` y poner nuestro hilo "a dormir".

Cuando se trabaja con prioridades en hilos **no hay garantías de que un hilo termine cuando esperemos**.

Podemos terminar un hilo de ejecución llamando al método `join`. Este método devuelve el control al hilo principal que lanzó el hilo secundario con la posibilidad de elegir un tiempo de espera en milisegundos.

El siguiente programa ilustra el uso de estos métodos:

```
class Calculador implements Runnable{
        @Override
        public void run() {
                int num=0;
                while(num<5){
                        System.out.println("Calculando...");
                        try {
                                long tiempo=(long) (1000*Math.random()*10);
                                if (tiempo>8000){
                                        Thread hilo=Thread.currentThread();
                                        System.out.println(
                                                        "Terminando hilo:"+
                                                                        hilo.get

                                        );
                                        hilo.join();
                                }
                                Thread.sleep(tiempo);
                        } catch (InterruptedException e) {
```

```
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                        System.out.println("Calculado y reiniciando.");
                        num++;
                }
                Thread hilo=Thread.currentThread();
                String miNombre=hilo.getName();
                System.out.println("Hilo terminado:"+miNombre);
        }
}

public class LanzadorHilos {
        public static void main(String[] args) {
                Calculador vHilos[]=new Calculador[5];
                for (int i=0; i<5;i++){
                        vHilos[i]=new Calculador();
                        Thread hilo=new Thread(vHilos[i]);
                        hilo.setName("Hilo "+i);
                        if (i==0){
                                hilo.setPriority(Thread.MAX_PRIORITY);
                        }
                        hilo.start();
                }
        }
}
```

Ejercicio: crear un programa que lance 10 hilos de ejecución donde a cada hilo se le pasará la base y la altura de un triángulo, y cada hilo ejecutará el cálculo del área de dicho triángulo informando de qué base y qué altura recibió y cual es el área resultado.

Una posibilidad (quizá incorrecta) sería esta:

```
package com.ies;

import java.util.Random;

class CalculadorAreas implements Runnable{
        int base, altura;
        public CalculadorAreas(int base, int altura){
                this.base=base;
                this.altura=altura;
        }
        @Override
        public void run() {
                float area=this.base*this.altura/2;
                System.out.print("Base:"+this.base);
                System.out.print("Altura:"+this.altura);
                System.out.println("Area:"+area);
        }
```

```
}
public class AreasEnParalelo {

        public static void main(String[] args) {
                Random generador=new Random();
                int numHilos=10000;
                int baseMaxima=3;
                int alturaMaxima=5;
                for (int i=0; i<numHilos; i++){
                        //Sumamos 1 para evitar casos como base=0
                        int base=1+generador.nextInt(baseMaxima);
                        int altura=1+generador.nextInt(alturaMaxima);
                        CalculadorAreas ca=
                                        new CalculadorAreas(base, altura);
                        Thread hiloAsociado=new Thread(ca);
                        hiloAsociado.start();
                }
        }
}
```

Las secciones siguientes sirven como resumen de como crear una aplicación multihilo

## 2.5 Creación, inicio y finalización.

- Podemos heredar de Thread o implementar `Runnable`. Usaremos el segundo recordando implementar el método `public void run()`.
- Para crear un hilo asociado a un objeto usaremos algo como:

```
Thread hilo=new Thread(objetoDeClase)
```

Lo más habitual es guardar en un vector todos los hilos que hagan algo, y no en un objeto suelto.

- Cada objeto que tenga un hilo asociado debe iniciarse así:

```
hilo.start();
```

- Todo programa multihilo tiene un "hilo principal", el cual deberá esperar a que terminen los hilos asociados ejecutando el método `join()` .

## 2.6 Sincronización de hilos.

Cuando un método acceda a una variable miembro que esté compartida deberemos proteger dicha sección crítica, usando `synchronized`. Se puede poner todo el método `synchronized` o marcar un trozo de código más pequeño.

## 2.7 Información entre hilos.

Todos los hilos comparten todo, así que obtener información es tan sencillo como consultar un miembro. En realidad podemos comunicar los hilos con otro mecanismo llamado `sockets de red`, pero se ve en el tema siguiente.

## 2.8 Prioridades de los hilos.

Podemos asignar distintas prioridades a los hilos usando los campos estáticos `MAX_PRIORITY` y `MIN_PRIORITY`. Usando valores entre estas dos constantes podemos hacer que un hilo reciba más procesador que otro (se hace en contadas ocasiones).

Para ello se usa el método `setPriority(valor)`

## 2.9 Gestión de prioridades.

En realidad un sistema operativo no está obligado a respetar las prioridades, sino que se lo tomará como "recomendaciones". En general hasta ahora todos respetan hasta cierto punto las prioridades que pone el programador pero no debe tomarse como algo absoluto.

## 2.10 Programación de aplicaciones multihilo.

La estructura típica de un programa multihilo es esta:

```
class TareaCompleja implements Runnable{
        @Override
        public void run() {
                for (int i=0; i<100;i++){
                        int a=i*3;
                }
                Thread hiloActual=Thread.currentThread();
                String miNombre=hiloActual.getName();
                System.out.println(
                                "Finalizado el hilo"+miNombre);
        }
}
public class LanzadorHilos {
        public static void main(String[] args) {
                int NUM_HILOS=100;
                Thread[] hilosAsociados;

                hilosAsociados=new Thread[NUM_HILOS];
                for (int i=0;i<NUM_HILOS;i++){
                        TareaCompleja t=new TareaCompleja();
```

```
                        Thread hilo=new Thread(t);
                        hilo.setName("Hilo: "+i);
                        hilo.start();
                        hilosAsociados[i]=hilo;
                }

                /* Despues de crear todo, nos aseguramos
                 * de esperar que todos los hilos acaben. */

                for (int i=0; i<NUM_HILOS; i++){
                        Thread hilo=hilosAsociados[i];
                        try {
                                //Espera a que el hilo acabe
                                hilo.join();
                        } catch (InterruptedException e) {
                                System.out.print("Algun hilo acabó ");
                                System.out.println(" antes de tiempo!");
                        }
                }
                System.out.println("El principal ha terminado");
        }
}
```

Supongamos que la tarea es más compleja y que el bucle se ejecuta un número al azar de veces. Esto significaría que nuestro bucle es algo como esto:

```
Random generador= new Random();
int numAzar=(1+generador.nextInt(5))*100;
for (int i=0; i<numAzar;i++){
        int a=i*3;
}
```

¿Como podríamos modificar el programa para que podamos saber cuantas multiplicaciones se han hecho en total entre todos los hilos?

Aquí entra el problema de la sincronización. Supongamos una clase contador muy simple como esta:

```
class Contador{
        int cuenta;
        public Contador(){
                cuenta=0;
        }
        public void incrementar(){
                cuenta=cuenta+1;
        }
        public int getCuenta(){
                return cuenta;
        }
}
```

De esta forma podríamos construir un objeto contador y pasárselo a todos los hilos para que en

ese único objeto se almacene el recuento final. El problema es que en la programación multihilo **SI EL OBJETO CONTADOR SE COMPARTE ENTRE VARIOS HILOS LA CUENTA FINAL RESULTANTE ES MUY POSIBLE QUE ESTÉ MAL**

Esta clase debería tener protegidas sus secciones críticas

```
class Contador{
        int cuenta;
        public Contador(){
                cuenta=0;
        }
        public synchronized void incrementar(){
                cuenta=cuenta+1;
        }
        public synchronized int getCuenta(){
                return cuenta;
        }
}
```

Se puede aprovechar todavía más rendimiento si en un método marcamos como sección crítica (o sincronizada) exclusivamente el código peligroso:

```
public  void incrementar(){
        System.out.println("Otras cosas");
        synchronized(this){
                cuenta=cuenta+1;
        }
        System.out.println("Mas cosas...");
        synchronized(this){
                if (cuenta>300){
                        System.out.println("Este hilo trabaja mucho");
                }
        }
}
```

## 2.11 Problema

En una mesa hay procesos que simulan el comportamiento de unos filósofos que intentan comer de un plato. Cada filósofo tiene un cubierto a su izquierda y uno a su derecha y para poder comer tiene que conseguir los dos. Si lo consigue, mostrará un mensaje en pantalla que indique "Filosofo 2 comiendo".

Despues de comer, soltará los cubiertos y esperará al azar un tiempo entre 1000 y 5000 milisegundos, indicando por pantalla "El filósofo 2 está pensando".

En general todos los objetos de la clase Filósofo está en un bucle infinito dedicándose a comer y a pensar.

Simular este problema en un programa Java que muestre el progreso de todos sin caer en problemas de sincronización ni de inanición.

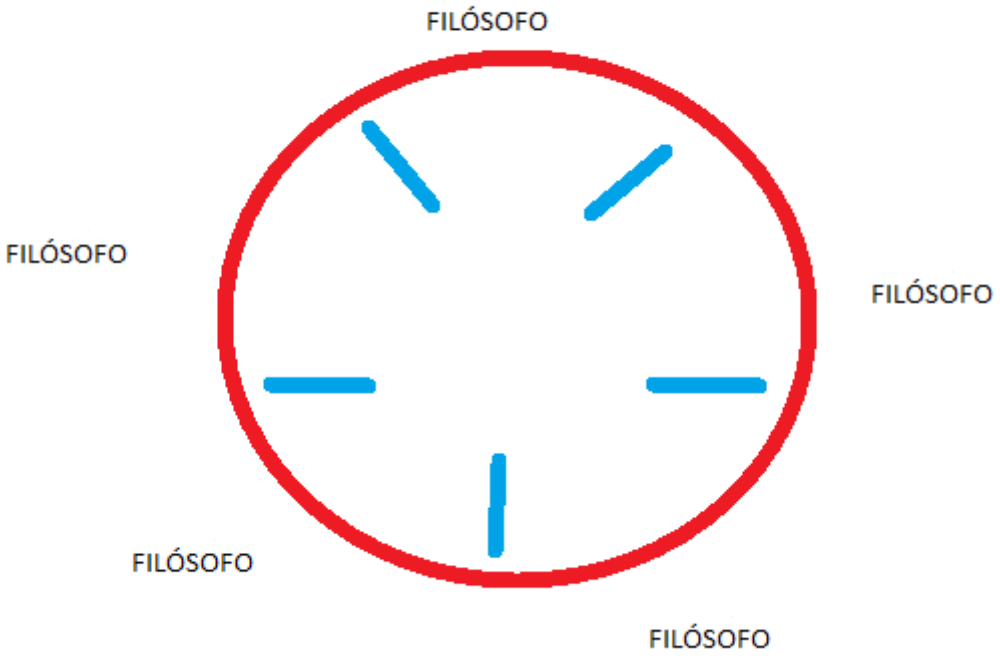


Figura 2.1: Esquema de los filósofos

### 2.11.1 Boceto de solución

```
import java.util.Random;


public class Filosofo  implements Runnable{
        public void run(){
                String miNombre=Thread.currentThread().getName();
                Random generador=new Random();
                while (true){
                        /* Comer*/
                        /* Intentar coger palillos*/
                        /* Si los coge:*/
                        System.out.println(miNombre+" comiendo...");
                        int milisegs=(1+generador.nextInt(5))*1000;
                        esperarTiempoAzar(miNombre, milisegs);
                        /* Pensando...*/
                        //Recordemos soltar los palillos
                        System.out.println(miNombre+"  pensando...");
                        esperarTiempoAzar(miNombre, milisegs);
                }
        }

private void esperarTiempoAzar(String miNombre, int milisegs) {
                try {
                        Thread.sleep(milisegs);
```

```
                } catch (InterruptedException e) {
                        System.out.println(
                                miNombre+" interrumpido!!. Saliendo...");
                        return ;
                }
        }
}
```

## 2.12 Solución completa al problema de los filósofos

### 2.12.1 Gestor de recursos compartidos (palillos)

```
public class GestorPalillos {
        /* False significa que no están cogidos*/
        private boolean[] palillos;
        public GestorPalillos(int num_filosofos){
                palillos=new boolean[num_filosofos];
                for (int i=0;i<palillos.length;i++){
                        palillos[i]=false;
                }
        }
        public synchronized boolean
                sePuedenCogerAmbosPalillos(
                        int num1,int num2){
                if ( (palillos[num1]==false) &&
                        (palillos[num2]==false) ) {
                        palillos[num1]=true;
                        palillos[num2]=true;
                        System.out.println(
                                "Alguien consiguio los palillos "
                                +num1+" y "+num2);
                        return true;
                }
                return false;
        }
        public synchronized void soltarPalillos(int num1, int num2){
                palillos[num1]=false;
                palillos[num2]=false;
                System.out.println(
                        "Alguien liberó los palillos "+
                        num1+" y "+num2);
        }
}
```

### 2.12.2 Simulación de un filósofo

```
import java.util.Random;
public class Filosofo  implements Runnable{
        int num_palillo_izq;
        int num_palillo_der;
        GestorPalillos gestorPalillos;
        public Filosofo(GestorPalillos gp,
                        int p_izq, int p_der){
                this.gestorPalillos=gp;
                this.num_palillo_der=p_der;
                this.num_palillo_izq=p_izq;
        }
        public void run(){
                String miNombre=Thread.currentThread().getName();
                Random generador=new Random();
                while (true){
                        /* Comer*/
                        /* Intentar coger palillos*/
                        while(!gestorPalillos.sePuedenCogerAmbosPalillos
                                (
                                                num_palillo_izq,
                                                num_palillo_der
                                ))
                        {

                        }
                        /* Si los coge:*/

                        int milisegs=(1+generador.nextInt(5))*1000;
                        esperarTiempoAzar(miNombre, milisegs);
                        /* Pensando...*/
                        //Recordemos soltar los palillos
                        gestorPalillos.soltarPalillos(
                                num_palillo_izq,
                                num_palillo_der);

                        milisegs=(1+generador.nextInt(5))*1000;
                        esperarTiempoAzar(miNombre, milisegs);
                }
        }

private void esperarTiempoAzar(String miNombre, int milisegs) {
                try {
                        Thread.sleep(milisegs);
                } catch (InterruptedException e) {
                        System.out.println(
                                miNombre+
                                " interrumpido!!. Saliendo...");
                        return ;
                }
```

```
        }
}
```

### 2.12.3 Lanzador de hilos

```
public class LanzadorFilosofos {
        public static void main(String[] args) {
                int MAX_FILOSOFOS=5;
                Filosofo[] filosofos=new Filosofo[MAX_FILOSOFOS];
                Thread[] hilosAsociados=new Thread[MAX_FILOSOFOS];
                GestorPalillos gestorCompartido=
                                new GestorPalillos(MAX_FILOSOFOS);
                for (int i=0; i<MAX_FILOSOFOS; i++){
                        if (i==0){
                                filosofos[i]=
                                new Filosofo(
                                                gestorCompartido,
                                                i,MAX_FILOSOFOS-1);
                        }
                        else {
                                filosofos[i]=new Filosofo(
                                                gestorCompartido, i, i-1);
                        }
                        Thread hilo=new Thread(filosofos[i]);
                        hilo.setName("Filosofo "+i);
                        hilosAsociados[i]=hilo;
                        hilo.start();
                }
                /* Un poco inútil*/
                for (int i=0; i<MAX_FILOSOFOS;i++){
                        try {
                                hilosAsociados[i].join();
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }
        }
}
```

## 2.13 Problema

En una peluquería hay barberos y sillas para los clientes (siempre hay más sillas que clientes). Sin embargo, en esta peluquería no siempre hay trabajo por lo que los barberos duermen cuando no hay clientes a los que afeitar. Un cliente puede llegar a la barbería y encontrar alguna silla libre, en cuyo caso, el cliente se sienta y esperará que algún barbero le afeite. Puede ocurrir que

el cliente llegue y no haya sillas libres, en cuyo caso se marcha. Simular el comportamiento de la barbería mediante un programa Java.

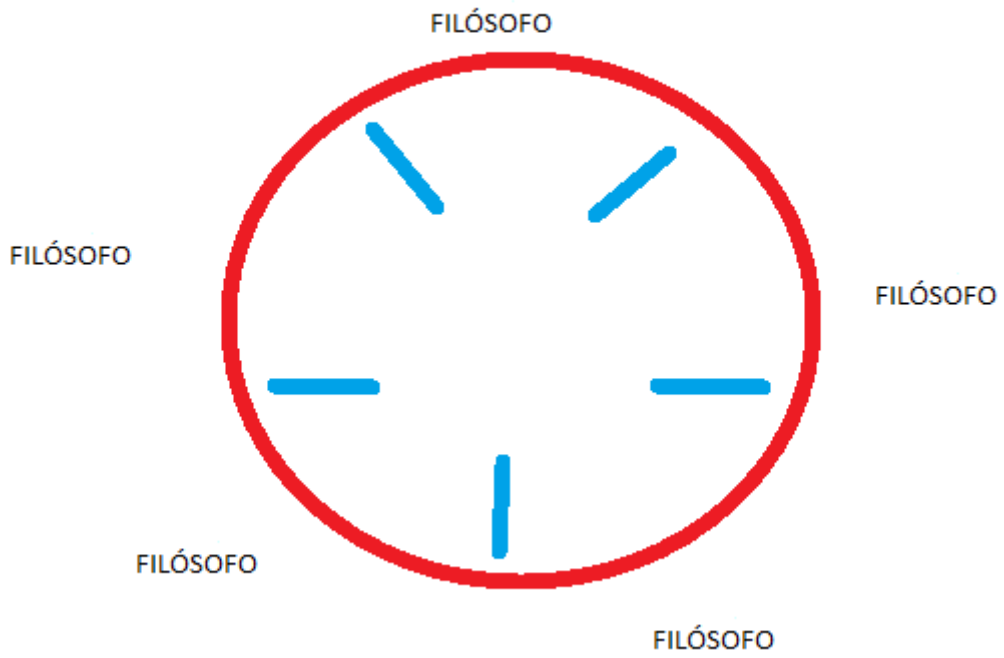


Figura 2.2: Los barberos dormilones

## 2.14 Una (mala) solución al problema de los barberos

Prueba la siguiente solución:

### 2.14.1 Clase Barbero

```
public class Barbero implements Runnable {
        private String                          nombre;
        private GestorConcurrencia      gc;
        private Random                          generador;
        private int                                     MAX_ESPERA_SEGS=5;
        public Barbero(GestorConcurrencia gc,String nombre){
                this.nombre             =nombre;
                this.gc                 =gc;
                this.generador  =new Random();
        }

        public void esperarTiempoAzar(int max){
                /* Se calculan unos milisegundos al azar*/
```

```
                int msgs=(1+generador.nextInt(max))*1000;
                try {
                        Thread.currentThread().sleep(msgs);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
        public void run(){
                while (true){
                        int num_silla=gc.atenderAlgunCliente();
                        while (num_silla==-1){
                                /* Mientras no haya nadie a quien
                                 * atender, dormimos
                                 */
                                esperarTiempoAzar(MAX_ESPERA_SEGS);
                                num_silla=gc.atenderAlgunCliente();
                        }
                        /* Si llegamos aqui es que había algún cliente
                         * Simulamos un tiempo de afeitado
                         */
                        esperarTiempoAzar(MAX_ESPERA_SEGS);
                        /* Tras ese tiempo de afeitado se
                         * libera la silla
                         */
                        gc.liberarSilla(num_silla);
                        /* Y vuelta a empezar*/
                }
        }
}
```

### 2.14.2 Clase Cliente

```
public class Cliente implements Runnable{
        GestorConcurrencia      gc;
        public Cliente(GestorConcurrencia gc){
                this.gc         =gc;
        }
        public void run(){
                /* Los clientes no esperan que haya
                 * sillas libres, no hay bucle infinito.
                 * Si no hay sillas libres se van...
                 */
                gc.getSillaLibre();
        }
}
```

### 2.14.3 Clase GestorConcurrencia

```
public class GestorConcurrencia {
        /* Vector que indica cuantas sillas hay y
         * si están libres o no
         */
        boolean[] sillasLibres;
        /* Indica si el cliente sentado en esa
         * silla está atendido por un barbero o no
         */
        boolean[] clienteEstaAtendido;

        public GestorConcurrencia(int numSillas){
                /*Construimos los vectores...*/
                sillasLibres            =new boolean[numSillas];
                clienteEstaAtendido     =new boolean[numSillas];
                /* ... los inicializamos*/
                for (int i=0; i<numSillas;i++){
                        sillasLibres[i]                 =true;
                        clienteEstaAtendido[i]  =false;
                }
        }

        /**
         * Permite obtener una silla libre, usado por la
         * clase Cliente para saber si puede sentarse
         * en algún sitio o irse
         * @return Devuelve el número de la primera silla
         * que está libre o -1 si no hay ninguna
         */
        public synchronized int getSillaLibre(){
                for (int i=0; i<sillasLibres.length; i++){
                        /* Si está libre la silla ...*/
                        if (sillasLibres[i]) {
                                /* ...se marca como ocupada*/
                                sillasLibres[i]=false;
                                System.out.println(
                                        "Cliente sentado en silla "+i
                                                );
                                /*.. y devolvemos i...*/
                                return i;
                        }
                }
                /* Si llegamos aquí es que no había nada libre*/
                return -1;
        }

        /**
         * Nos dice qué silla tiene algun cliente
         * que no está atendido
         * @return un número de silla o -1 si no
```

```
         * hay clientes sin atender
         */
        public synchronized int atenderAlgunCliente(){
                for (int i=0; i<sillasLibres.length; i++){
                        /* Si una silla está ocupada (no libre, false)
                         * y está marcado como "sin atender" (false)
                         * entonces la marcamos como atendida
                         */
                        if (clienteEstaAtendido[i]==false){
                                clienteEstaAtendido[i]=true;
                                System.out.println(
                                                "Afeitando cliente en silla "+i)
                                return i;
                        }
                }
                return -1;
        }

        /* El cliente de esa silla, se marcha, por lo
         * que se marca esa silla como "libre"
         * y como "sin atender"
         */
        public synchronized void liberarSilla(int i){
                sillasLibres[i]                 =true;
                clienteEstaAtendido[i]  =false;
                System.out.println(
                                "Se marcha el cliente de la silla "+i);
        }
}
```

### 2.14.4 Clase Lanzador

```
public class Lanzador {

        public static void main(String[] args) {

                int MAX_BARBEROS        =2;
                int MAX_SILLAS          =MAX_BARBEROS+1;
                int MAX_CLIENTES        =MAX_BARBEROS*10;
                int MAX_ESPERA_SEGS     = 3;
                GestorConcurrencia gc;
                gc=new GestorConcurrencia(MAX_SILLAS);

                Thread[] vhBarberos     =new Thread[MAX_BARBEROS];
                for (int i=0; i<MAX_BARBEROS;i++){
                        Barbero b=new Barbero(gc, "Barbero "+i);
                        Thread hilo=new Thread(b);
                        vhBarberos[i]=hilo;
                        hilo.start();
```

```
                }

                /* Generamos unos cuantos clientes
                 * a intervalos aleatorios
                 */
                Random generador=new Random();
                for (int i=0; i<MAX_CLIENTES; i++){
                        Cliente c                       =new Cliente(gc);
                        Thread hiloCliente      =new Thread(c);
                        hiloCliente.start();

                        int msegs=generador.nextInt(3)*1000;
                        try {
                                Thread.sleep(msegs);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                } /* Fin del for*/
        }
}
```

### 2.14.5 Críticas a la solución anterior

¿Cual es el problema?

El problema está en que la forma que tiene el gestor de concurrencia de decirle a un barbero qué silla tiene un cliente sin afeitar es incorrecta: como siempre se empieza a buscar por el principio del vector, los clientes sentados al final **nunca son atendidos**. Hay que corregir esa asignación para *evitar que los procesos sufrán de inanición*.

### 2.14.6 Método corregido

```
public synchronized int atenderAlgunCliente(){
        for (int pasos=0;
                        pasos<clienteEstaAtendido.length;
                        pasos++)
        {
                if (
                                clienteEstaAtendido
                                [numUltimaSillaExaminada]
                                                == false
                                                )
                {
                        /*Atendemos a ese cliente*/
                        clienteEstaAtendido
                                [numUltimaSillaExaminada]=true;
                        System.out.println(
```

```
                                        "Afeitando cliente en silla "+
                                                numUltimaSillaExaminada);
                        return numUltimaSillaExaminada;
                } else {
                        numUltimaSillaExaminada=
                                        (numUltimaSillaExaminada+1)%
                                        clienteEstaAtendido.length;
                } //Fin del else
        } //Fin del for
        /* Si llegamos aquí hemos dado toda
         * una vuelta al vector y no había nadie sin
         * atender devolver -1
         */
        return -1;
} //Fin del método
```

## 2.15 Problema: productores y consumidores.

En un cierto programa se tienen procesos que producen números y procesos que leen esos números. Todos los números se introducen en una cola (o vector) limitada.

Todo el mundo lee y escribe de/en esa cola. Cuando un productor quiere poner un número tendrá que comprobar si la cola está llena. Si está llena, espera un tiempo al azar. Si no está llena pone su número en la última posición libre.

Cuando un lector quiere leer, examina si la cola está vacía. Si lo está espera un tiempo al azar, y sino coge el número que haya al principio de la cola y ese número *ya no está disponible para el siguiente*.

Crear un programa que simule el comportamiento de estos procesos evitando problemas de entrelazado e inanición.

## 2.16 Solución

### 2.16.1 Una cola limitada en tamaño

```
public class ColaLimitada {
        int[] cola;
        int posParaEncolar;
        public ColaLimitada(int numElementos){
                cola=new int[numElementos];
                posParaEncolar=0;
        }
        public void ponerEnCola(int numero){
                if (posParaEncolar==cola.length){
                        System.out.println(
```

```
                                        "Cola llena, debe Vd. esperar");
                        //Cola llena.
                        return ;
                }
                //Aún queda sitio
                cola[posParaEncolar]=numero;
                posParaEncolar++;
        }
        public int sacarPrimero(){
                if (posParaEncolar==0){
                        System.out.println(
                                        "Warning:cola vacía, devolviendo 0"
                        );
                        return 0;
                }
                int elementoInicial=cola[0];
                /*Movemos los elementos hacia delante*/
                for (int pos=1; pos<cola.length; pos++){
                        cola[pos-1]=cola[pos];
                }
                /* Ahora la posParaEncolar ha disminuido*/
                posParaEncolar--;
                return elementoInicial;
        }
        public String toString(){
                String cadenaCola="";
                for (int pos=0; pos<posParaEncolar; pos++){
                        cadenaCola+=cola[pos]+"-";
                }
                cadenaCola+="FIN";
                return cadenaCola;
        }
}
```

### 2.16.2 Un gestor de concurrencia para la cola

```
public class GestorColasConcurrentes {
        private ColaLimitada colaProtegida;
        public GestorColasConcurrentes(int numElementos){
                colaProtegida=new ColaLimitada(numElementos);
        }
        public synchronized
                void ponerEnCola(int elemento){
                colaProtegida.ponerEnCola(elemento);
        }
        public synchronized int sacarDeCola(){
                return colaProtegida.sacarPrimero();
        }
}
```

### 2.16.3 La clase Productor

```
public class Productor implements Runnable{
        private Random                                                  generado
        private         GestorColasConcurrentes         gc;
        public Productor(GestorColasConcurrentes gc){
                this.gc=gc;
                this.generadorAzar=new Random();
        }
        public void run(){
                while (true){
                        int numero=generadorAzar.nextInt(20);
                        gc.ponerEnCola(numero);
                        int milisegs=generadorAzar.nextInt(2);
                        try {
                                Thread.currentThread().sleep(milisegs*1000);
                        } catch (InterruptedException e) {
                                System.out.println(
                                                "Productor interrumpido"
                                );
                                return;
                        }
                }
        }
}
```

### 2.16.4 La clase consumidor

```
public class Consumidor implements Runnable {
        private Random                                                  generado
        private GestorColasConcurrentes         gc;

        public Consumidor(GestorColasConcurrentes gc){
                this.gc=gc;
                this.generadorAzar=new Random();
        }
        public void run(){
                while (true){
                        int num=gc.sacarDeCola();
                        int milisegs=generadorAzar.nextInt(2);
                        try {
                                Thread.currentThread().sleep(milisegs*1000);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                System.out.println(
                                                "Consumidor interrumpido");
                                return ;
                        }
                }
```

```
        }
}
```

### 2.16.5 Un lanzador

```
public class Lanzador {
        public void test(){
                ColaLimitada c=new ColaLimitada(5);
                if (c.sacarPrimero()!=0){
                        System.out.println(
                          "Error, no se comprueba el caso cola vacía"
                        );
                }
                c.ponerEnCola(10);
                c.ponerEnCola(20);
                String cadenaCola=c.toString();
                if (!cadenaCola.equals("10-20-FIN")){
                        System.out.println("Fallos al encolar");
                }
        }
        public static void main(String[] argumentos){
                Lanzador l=new Lanzador();
                GestorColasConcurrentes gcl=
                                new GestorColasConcurrentes(10);

                int NUM_PRODUCTORES=5;
                Productor[]     productores;
                Thread[]                        hilosProductores;

                productores             =
                                new Productor[NUM_PRODUCTORES];
                hilosProductores        =
                                new Thread[NUM_PRODUCTORES];

                for (int i=0; i<NUM_PRODUCTORES; i++){
                        productores[i]=new Productor(gcl);
                        hilosProductores[i]=new Thread(
                                        productores[i]);
                        hilosProductores[i].start();
                }

                int NUM_CONSUMIDORES=10;
                Consumidor[]    consumidores;
                Thread[]                        hilosConsumidores;

                consumidores                    =
                                new Consumidor[NUM_CONSUMIDORES];
                hilosConsumidores       =
                                new Thread[NUM_CONSUMIDORES];
```

```
                for (int i=0; i<NUM_CONSUMIDORES; i++){
                        consumidores[i]=new Consumidor(gcl);
                        hilosConsumidores[i]=
                                        new Thread(consumidores[i]);
                        hilosConsumidores[i].start();
                }

                /* Se debería esperar a que todos terminen*/
                for (int i=0; i<NUM_PRODUCTORES; i++){
                        try {
                                hilosProductores[i].join();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
                for (int i=0; i<NUM_CONSUMIDORES; i++){
                        try {
                                hilosConsumidores[i].join();
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }
        }
}
```

## 2.17 Ejercicio

En unos grandes almacenes hay 300 clientes agolpados en la puerta para intentar conseguir un producto del cual solo hay 100 unidades.

Por la puerta solo cabe una persona, pero la paciencia de los clientes es limitada por lo que solo se harán un máximo de 10 intentos para entrar por la puerta. Si despues de 10 intentos la puerta no se ha encontrado libre ni una sola vez, el cliente desiste y se marcha.

Cuando se consigue entrar por la puerta el cliente puede encontrarse con dos situaciones:

1. Quedan productos: el cliente cogerá uno y se marchará.
2. No quedan productos: el cliente simplemente se marchará.

Realizar la simulación en Java de dicha situación.

## 2.18 Documentación.

## 2.19 Depuración.

CAPÍTULO 3

# Programación de comunicaciones en red

CAPÍTULO 4

# Generación de servicios en red

## 4.1 Protocolos estándar de comunicación en red a nivel de aplicación

### 4.1.1 Telnet

### 4.1.2 FTP

### 4.1.3 HTTP

### 4.1.4 POP3

### 4.1.5 SMTP

## 4.2 Librerías de clases y componentes.

## 4.3 Utilización de objetos predefinidos.

## 4.4 Propiedades de los objetos predefinidos.

## 4.5 Métodos y eventos de los objetos predefinidos.

## 4.6 Establecimiento y finalización de conexiones.

## 4.7 Transmisión de información.

## 4.8 Programación de aplicaciones cliente.

## 4.9 Programación de servidores.

## 4.10 Implementación de comunicaciones simultáneas

CAPÍTULO 5

# Utilización de técnicas de programación segura

5.1 Prácticas de programación segura.

5.2 Criptografía de clave pública y clave privada.

5.3 Principales aplicaciones de la criptografía.

5.4 Protocolos criptográficos.

5.5 Política de seguridad.

5.6 Programación de mecanismos de control de acceso.

5.7 Encriptación de información.

5.8 Protocolos seguros de comunicaciones.

5.9 Programación de aplicaciones con comunicaciones seguras.

5.10 Pruebas y depuración.