Apuntes de Servicios y Procesos

Publicación 1.3

Oscar Gomez

dic. 19, 2016

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.

Se pueden utilizar "desensambladores" para averiguar la secuencia de instrucciones que hay en un EXE. Incluso existen desensambladores en línea como http://onlinedisassembler.com

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 archivo .CLASS puede desensamblarse utilizando el comando `javap -c <archivo.class>` . Cuando se hace así, se obtiene un listado de "instrucciones" que no se corresponden con las instrucciones del microprocesador, sino con "instrucciones virtuales de Java". El intérprte Java (el archivo `java.exe`) traducirá en el momento del arranque dichas instrucciones virtuales Java a instrucciones reales del microprocesador.

Este último aspecto es el esgrimido por Java para defender que su ejecución puede ser más rápida que la de un EXE, ya que Java puede averiguar en qué microprocesador se está ejecutando y así generar el código más óptimo posible.

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\\AcroRd32.exe";
                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).

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;
```

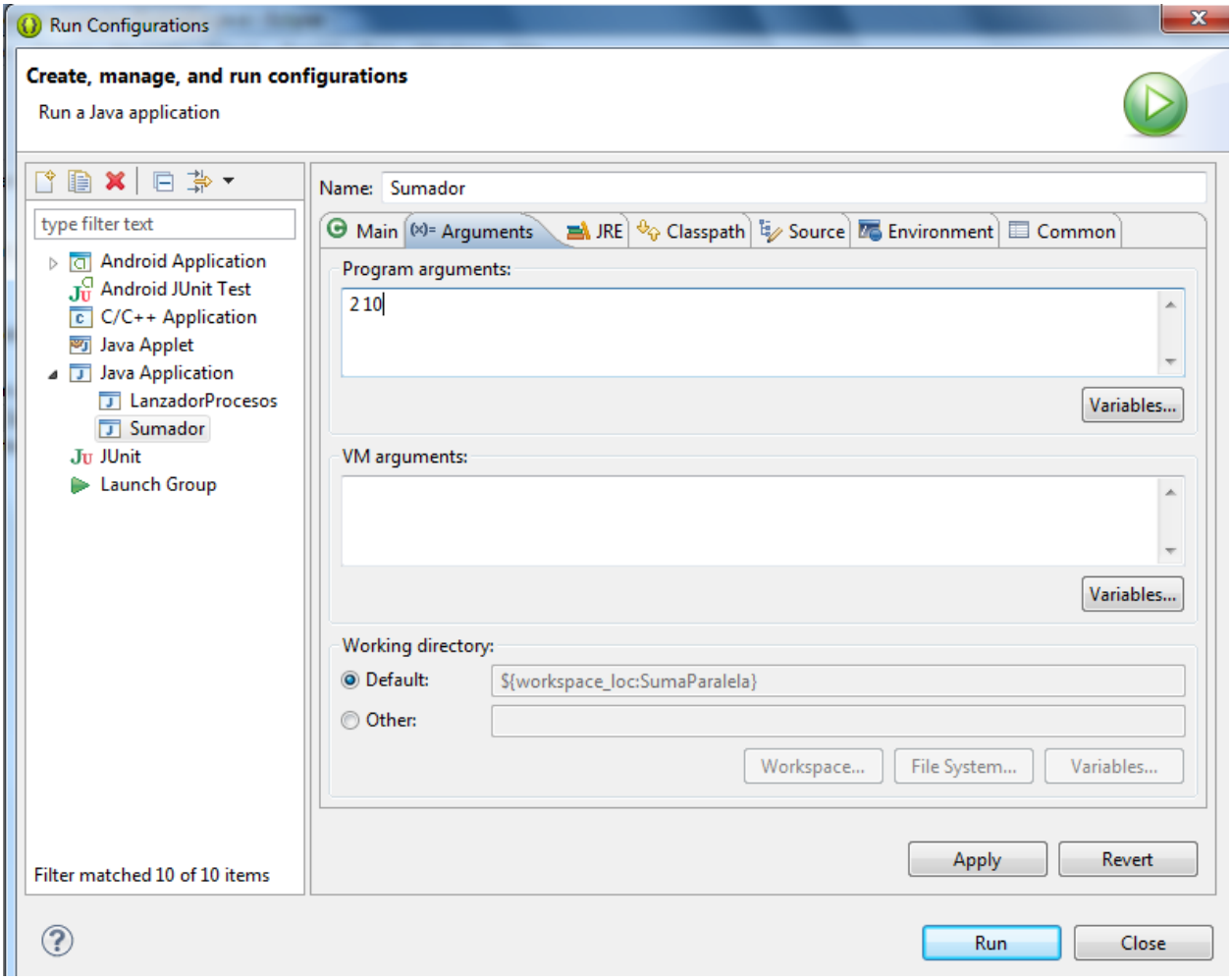

Figura 1.1: Modificando los argumentos del programa

```
                }
                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 Ejercicio

Crear un programa que permita parametrizar el lanzamiento de sumadores, que vuelque el contenido de las sumas en ficheros y que permita al programa principal recuperar las sumas de los ficheros parciales.

En el listado siguiente se muestra la clase Sumador

```
package es.ies.multiproceso;

public class Sumador {
        /** Suma todos los valores incluidos
         * entre dos valores
         * @param n1 Limite 1
         * @param n2 Limite 2
         * @return La suma de dichos valores
         */
        public static int sumar(int n1, int n2){
                int suma=0;
                if (n1>n2){
                        int aux=n1;
                        n1=n2;
                        n2=aux;
                }
                for (int i=n1; i<=n2; i++){
                        suma=suma+i;
                }
                return suma;
        }

        public static void main(String[] args){
                int n1=Integer.parseInt(args[0]);
                int n2=Integer.parseInt(args[1]);
                int suma=sumar(n1, n2);
                System.out.println(suma);
                System.out.flush();
```

```
        }
}
```

En el listado siguiente se muestra la clase Main

```
public class Main {

        static final int NUM_PROCESOS=4;
        static final String PREFIJO_FICHEROS="fich";


        public static void lanzarSumador(
                        int n1, int n2,String fichResultados) throws IOException{
                String comando;
                comando="es.ies.multiproceso.Sumador";

                File directorioSumador;
                directorioSumador=new File("C:\\Users\\"+
                "ogomez\\workspace\\"+
                "MultiProceso1\\bin\\");
                File fichResultado=new File(fichResultados);
                ProcessBuilder pb;
                pb=new ProcessBuilder("java",
                                comando,
                                String.valueOf(n1),
                                String.valueOf(n2) );
                pb.directory(directorioSumador);
                pb.redirectOutput(fichResultado);
                pb.start();
        }

        public static int getResultadoFichero(
                        String nombreFichero){

                int suma=0;
                try {
                        FileInputStream fichero=
                                        new FileInputStream(
                                                        nombreFichero);
                        InputStreamReader fir=
                                        new InputStreamReader(
                                                        fichero);
                        BufferedReader br=new BufferedReader(fir);
                        String linea=br.readLine();
                        suma= new Integer(linea);
                        return suma;
                } catch (FileNotFoundException e) {
                        System.out.println(
                                "No se pudo abrir "+nombreFichero);

                } catch (IOException e) {
                        System.out.println(
                                        "No hay nada en "+nombreFichero);
                }
                return suma;
        }
```

```
        public static int getSumaTotal(int numFicheros){
                int sumaTotal=0;
                for (int i=1; i<=NUM_PROCESOS;i++){
                        sumaTotal+=getResultadoFichero(
                                PREFIJO_FICHEROS+String.valueOf(i) );
                }
                return sumaTotal;
        }

        /* Recibe dos parámetros y hará
         * la suma de los valores comprendidos
         * entre ambos parametros
         */
        public static void main(String[] args) throws IOException,␣
↪InterruptedException{
                int n1=Integer.parseInt(args[0]);
                int n2=Integer.parseInt(args[1]);
                int salto=( n2 / NUM_PROCESOS ) - 1;
                for (int i=1;i<=NUM_PROCESOS; i++){
                        System.out.println("n1:"+n1);
                        int resultadoSumaConSalto=n1+salto;
                        System.out.println("n2:"+resultadoSumaConSalto);
                        lanzarSumador(n1, n1+salto ,
                                        PREFIJO_FICHEROS+String.valueOf(i));
                        n1=n1 + salto + 1;
                        System.out.println("Suma lanzada...");
                }
                Thread.sleep(5000);
                int sumaTotal=getSumaTotal(NUM_PROCESOS);
                System.out.println("La suma total es:"+
                                                sumaTotal);
        }
}
```

1.8 Ejercicio resuelto

Crear un programa que sea capaz de contar cuantas vocales hay en un fichero. El programa padre debe lanzar cinco procesos hijo, donde cada uno de ellos se ocupará de contar una vocal concreta (que puede ser minúscula o mayúscula). Cada subproceso que cuenta vocales deberá dejar el resultado en un fichero. El programa padre se ocupará de recuperar los resultados de los ficheros, sumar todos los subtotales y mostrar el resultado final en pantalla.

1.8.1 Operaciones con ficheros

Tanto en este ejercicio como en muchos otros vamos a necesitar realizar ciertas operaciones con ficheros. Para aumentar nuestra productividad utilizaremos una clase llamada `UtilidadesFicheros` que nos permita realizar ciertas tareas como las siguientes:

- Obtener un objeto `BufferedReader` a partir de un nombre de fichero de tipo `String`. Llamaremos a este método `getBufferedReader` y lo utilizaremos para poder manejar un fichero mediante una clase de muy alto nivel que nos facilita la lectura de ficheros. En el listado adjunto se puede consultar dicho método.

```
public static BufferedReader getBufferedReader(
    String nombreFichero) throws
FileNotFoundException {
```

```
    FileReader lector;
    lector = new FileReader(nombreFichero);
    BufferedReader bufferedReader;
    bufferedReader = new BufferedReader(lector);
    return bufferedReader;
}
```

- De la misma forma crearemos un método `getPrintWriter` que nos devuelva un objeto de la clase `PrintWriter` para poder realizar fácilmente operaciones de escritura en ficheros. En el listado adjunto se muestra el código.

```
public static PrintWriter getPrintWriter(
    String nombreFichero) throws IOException {
    PrintWriter printWriter;
    FileWriter fileWriter;
    fileWriter = new FileWriter(nombreFichero);
    printWriter = new PrintWriter(fileWriter);
    return printWriter;
//Fin de getPrintWriter
}
```

- Aunque nos estamos anticipando, puede ser útil tener un método que dado un nombre de fichero nos devuelva un `ArrayList<String>` con todas las líneas que hay en el fichero. **En este ejercicio no lo usaremos**. Este método podría implementarse así:

```
public static ArrayList<String> getLineasFichero(
    String nombreFichero) throws IOException {
    ArrayList<String> lineas = new ArrayList<String>();
    BufferedReader bfr = getBufferedReader(nombreFichero);
    //Leemos líneas del fichero...
    String linea = bfr.readLine();
    while (linea != null) {
        //Y las añadimos al array
        lineas.add(linea);
        linea = bfr.readLine();
    }
    //Fin del bucle que lee líneas
    return lineas;
}
```

1.8.2 Procesado de ficheros

Necesitaremos una clase `ProcesadorFichero` que nos permita procesar los ficheros de la forma pedida. Antes de crear un programa multiproceso empezaremos por crear una clase simple que nos resuelva este problema.

Dicha clase tendrá un método `hacerRecuento` que resuelva el problema de contar el número de apariciones en un fichero dejando el total de apariciones en un fichero de salida distintos. En el código adjunto podemos ver como podría implementarse dicho método.

```
public static ArrayList<String> getLineasFichero(
    String nombreFichero) throws IOException {
    ArrayList<String> lineas = new ArrayList<String>();
    BufferedReader bfr = getBufferedReader(nombreFichero);
    //Leemos líneas del fichero...
    String linea = bfr.readLine();
    while (linea != null) {
```

```
        //Y las añadimos al array
        lineas.add(linea);
        linea = bfr.readLine();
    }
    //Fin del bucle que lee líneas
    return lineas;
}
```

La clase `ProcesadorFichero` será lanzada desde otra clase que llamaremos `Lanzador`. Nuestra `ProcesadorFichero` recogerá en su método `main` los siguientes parámetros:

1. El nombre del fichero a procesar. Lo llamaremos `nombreFicheroEntrada` y estará en la posición 0 de los argumentos.
2. La letra de la que hay que hacer el recuento de apariciones. La llamaremos `letra` y estará en la posición 1.
3. El nombre del fichero donde se dejarán los resultados. Lo llamaremos `nombreFicheroResultado` y estará en la posición 1 de los argumentos.

El código del main se muestra a continuación.

```
/**
         * Dado un fichero pasado como argumento, contará cuantas
         * apariciones hay de una cierta vocal (pasada como argumento)
         * y dejará la cantidad en otro fichero (también pasado como
         * argumento)
         * @throws IOException
         * @throws FileNotFoundException  */
public static void main(String[] args) throws
FileNotFoundException, IOException {
    String nombreFicheroEntrada = args[0];
    String letra = args[1];
    String nombreFicheroResultado = args[2];
    hacerRecuento(nombreFicheroEntrada, letra, nombreFicheroResultado);
//Fin del main
}
```

1.9 Lanzamiento de los procesos

La clase `Lanzador` tendrá un método `main` que se encargará de varias cosas:

1. Recoger el primer parámetro (`args[0]`), que contendrá el fichero a procesar.
2. Recoger el segundo parámetro, que contendrá el directorio de `CLASSPATH` donde habrá que buscar la clase `UtilidadesFicheros`.
3. Recoger el tercer parámetro, que contendrá el directorio donde habrá que buscar la clase `ProcesadorFicheros`.
4. Una vez recogidos los parámetros, se lanzarán los procesos utilizando la clase `ProcessBuilder`.
5. Los procesos se ejecutarán y después recogeremos los resultados de los ficheros.

En el código siguiente puede verse el método `main` de esta clase `Lanzador`. La recogida de resultados se deja como ejercicio:

```
public static void main(String[] args) throws
IOException, InterruptedException {
```

```
    String ficheroEntrada;
    ficheroEntrada = args[0];
    String classpathUtilidades;
    classpathUtilidades = args[1];
    String classpathProcesadorFichero;
    classpathProcesadorFichero = args[2];
    String[] vocales = { "A", "E", "I", "O", "U" };
    String classPath;
    classPath = classpathProcesadorFichero + ":" + classpathUtilidades;
    System.out.println("Usando classpath:" + classPath);
    /* Se lanzan los procesos*/
    for (int i = 0; i < vocales.length; i++) {
        String fichErrores = "Errores_" + vocales[i] +
                             ".txt";
        ProcessBuilder pb;
        pb = new ProcessBuilder("java", "-cp", classPath,
                                "com.ies.ProcesadorFichero", ficheroEntrada,
                                vocales[i], vocales[i] + ".txt");
        //Si hay algún error, almacenarlo en un fichero
        pb.redirectError(new File(fichErrores));
        pb.start();
        //Fin del for
    }
    /* Esperamos un poco*/
    Thread.sleep(3000);
    /* La recogida de resultados se deja como
                    * ejercicio al lector. ;) */
}
```

1.10 Ejercicio propuesto (I)

Se desea crear un programa que procese ficheros aprovechando el paralelismo de la máquina. Se tienen cinco ficheros con los siguientes nombres:

- `informatica.txt`
- `gerencia.txt`
- `contabilidad.txt`
- `comercio.txt`
- `rrhh.txt`

En cada fichero hay una lista de cantidades enteras que representa las contabilidades de distintos departamentos. Hay una cantidad en cada línea. Se desea que el programa creado sume la cantidad total que suman todas las cantidades de los cinco ficheros haciendo uso del paralelismo.

1.10.1 Solución al ejercicio propuesto (I)

Este ejercicio es bastante parecido al anterior, con la salvedad de que ahora necesitaremos sumar cantidades en lugar de buscar elementos. Por lo demás, también generaremos un fichero de resultados por cada fichero de datos y los llamaremos igual pero añadiendo la extensión ".res". Es decir, la suma de las cantidades de `informatica.txt` se dejará en `informatica.txt.res`.

Una vez generados todos los fichero `Resultado` tendremos que sumar todas las cantidades de estos ficheros. Su estructura es muy simple, todos contienen una única línea con una cantidad. Dado que esta operación de sumar cantidades almacenadas en distintos ficheros es una operación muy habitual, añadiremos un método a `UtilidadesFicheros` que se encargue de hacer esta operación. Obsérvese que usaremos tipos de datos `long` y no `int` dado que no sabemos como de grandes serán las cantidades a procesar.

A continuación se muestra el código de este método.

```
//Fin de getLineasFichero
public static long getSuma(String[]
                           listaNombresFichero) {
    long suma = 0;
    ArrayList<String> lineas;
    String lineaCantidad;
    long cantidad;
    for (String nombreFichero : listaNombresFichero) {
        try {
            //Recuperamos todas las lineas
            lineas = getLineasFichero(nombreFichero);
            //Pero solo nos interesa la primera
            lineaCantidad = lineas.get(0);
            //Convertimos la linea a número
            cantidad = Long.parseLong(lineaCantidad);
            //Y se incrementa la suma total
            suma = suma + cantidad;
        } catch (IOException e) {
            System.err.println("Fallo al procesar el fichero "
                               + nombreFichero);
            //Fin del catch
        }
        //Fin del for que recorre los nombres de fichero
    }
    return suma;
}
```

Una vez creado este método que usaremos al final, pasemos a crear una clase `ProcesadorContabilidad` que leerá un fichero, sumará las cantidades y las dejará en el fichero de resultados. Esta clase solo tendrá un método `main` que realizará esta tarea.

Como nuestro programa está formado por varias clases que están dispersas por distintos directorios, nuestro `main` tomará el `CLASSPATH` de los argumentos.

```
public static void main(String[] args) throws
IOException {
    String nombreFichero = args[0];
    String nombreFicheroResultado = args[1];
    ArrayList<String> cantidades;
    long total = 0;
    try {
        //Extraemos las cantidades
        cantidades = UtilidadesFicheros.getLineasFichero(nombreFichero);
        //Y las sumamos una por una
        for (String lineaCantidad : cantidades) {
            long cantidad = Long.parseLong(lineaCantidad);
            total = total + cantidad;
            //Fin del for que recorre las cantidades
        }
        //Almacenamos el total en un fichero
        PrintWriter pw;
```

```
        pw = UtilidadesFicheros.getPrintWriter(nombreFicheroResultado);
        pw.println(total);
        pw.close();
    }//Fin del try
    catch (IOException e) {
        System.err.println("No se pudo procesar el fichero "
        + nombreFichero);
        e.printStackTrace();
    }
//Fin del main
}
```

Por último, solo queda crear la clase `Lanzador` que recogerá los nombres de fichero que hay que sumar y lanzará un proceso de la clase `ProcesadorContabilidad` para cada uno de ellos. Por último, sumará todos los resultados y los dejará en un fichero que llamaremos (por ejemplo) `Resultado_global.txt`

```
public static void main(String[] args) throws
IOException {
    String classpath = args[0];
    String[] ficheros = { "informatica.txt", "gerencia.txt", "contabilidad.txt",
↪"comercio.txt", "rrhh.txt" };
    //Los nombres de los ficheros de resultados
    //se generarán y luego se almacenarán aquí
    String[] ficherosResultado;
    ficherosResultado = new String[ficheros.length];
    /* Lanzamos los procesos*/
    ProcessBuilder[] constructores;
    constructores = new ProcessBuilder[ficheros.length];
    for (int i = 0; i < ficheros.length; i++) {
        String fichResultado, fichErrores;
        fichResultado = ficheros[i] + SUFIJO_RESULTADO;
        fichErrores = ficheros[i] + SUFIJO_ERRORES;
        ficherosResultado[i] = fichResultado;
        constructores[i] = new ProcessBuilder();
        constructores[i].command("java", "-cp", classpath,
                                 "com.ies.ProcesadorContabilidad", ficheros[i],
                                 fichResultado);
        //El fichero de errores se generará, aunque
        //puede que vacío
        constructores[i].redirectError(new File(
                                           fichErrores));
        constructores[i].start();
        //Fin del for que recorre los ficheros
    }
    //Calculamos las sumas de cantidades
    long total = UtilidadesFicheros.getSuma(ficherosResultado);
    //Y las almacenamos
    PrintWriter pw = UtilidadesFicheros.getPrintWriter(RESULTADOS_GLOBALES);
    pw.println(total);
    pw.close();
//Fin del main
}
```

1.11 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 más sofisticados que proporcionan algo más de información sobre los procesos, como Process Explorer (antes conocido con el nombre de ProcessViewer).

1.12 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 -SIGSTOP <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`.

1.12.1 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**

1.13 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,j;
i=0;
if (i>=2){
        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 posible 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".

1.13.1 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␣
↪método*/
    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 trozo de código.*/
    for (int i=0; i=i+1; i++){
            /* Código omitido*/
            synchronized {
                    i=i*2;
                    j=j+1;
            }
```

1.14 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 clases para extraer la información que proporcionan dichas anotaciones.

1.15 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`, que en muchos casos se deben a la utilización de referencias Java no inicializadas o incluso a la devolución de valores NULL que luego no se comprueban en alguna parte del código.
4. Se puede usar el método `redirectError` pasándole un objeto de tipo `File` para que los mensajes de error vayan a un fichero.
5. Se debe recordar que la "visión" que tiene Eclipse del sistema puede ser **muy diferente** de la visión que tiene el proceso lanzado. Un problema muy común es que el proceso lanzado no encuentre clases, lo que obligará a indicar el `CLASSPATH`.
6. Un buen método para determinar errores consiste en utilizar el entorno de consola para lanzar comandos para ver "como es el sistema" que ve un proceso fuera de Eclipse (o de cualquier otro entorno).

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

1.16 Examen

Fecha por determinar

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` en teoría debería ocurrir que el número se incrementase tantas veces como procesos. Sin embargo, **es muy probable que eso no ocurra**

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 `currentThread` 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.getName()
                                        );
                                        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:

Para implementar la clase `CalculadorAreas`

```
class CalculadorAreas implements Runnable {

    float base, altura, area;

    public int contador = 0;

    public CalculadorAreas(float b, float a) {
        this.base = b;
        this.altura = a;
    }

    public synchronized void incrementarContador() {
        contador = contador + 1;
    }

    @Override
    public void run() {
        Random generador;
        generador = new Random();
        area = base * altura / 2;
        try {
            Thread.sleep(generador.nextInt(650));
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        this.incrementarContador();
    }
}
```

Para implementar la clase `AreasEnParalelo` que lanza los hilos para hacer muchos cálculos de áreas en paralelo usaremos esto:

```

```

```
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.

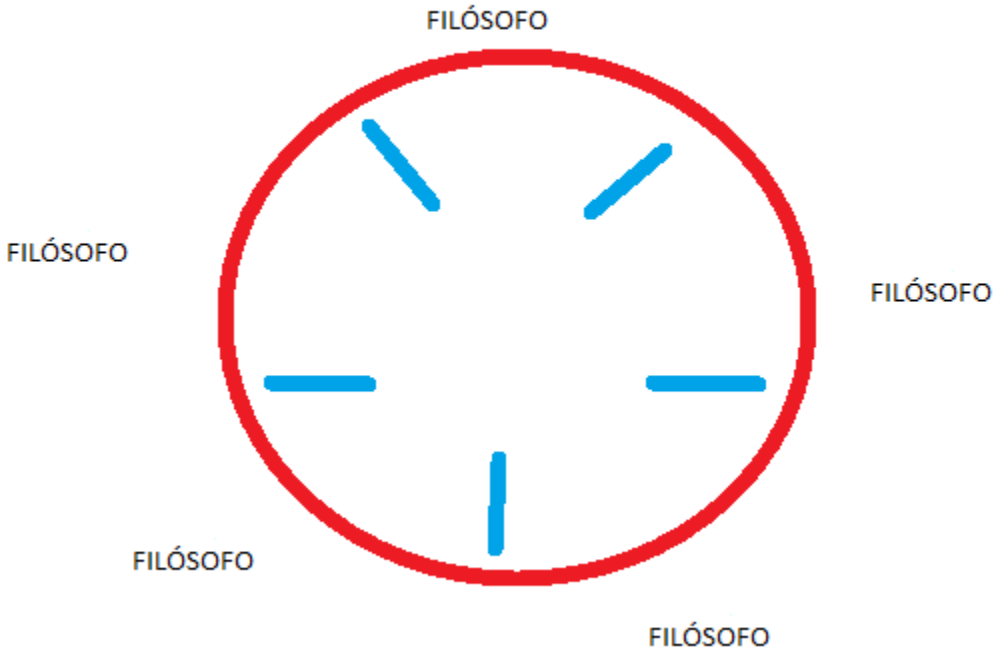

Figura 2.1: Esquema de los filósofos

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...");
↪           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 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: simulador de casino

Se desea simular los posibles beneficios de diversas estrategias de juego en un casino. La ruleta francesa es un juego en el que hay una ruleta con 37 números (del 0 al 36). Cada 3000 milisegundos el croupier saca un número al azar y los diversos hilos apuestan para ver si ganan. Todos los hilos empiezan con 1.000 euros y la banca (que controla la ruleta) con 50.000. Cuando los jugadores pierden dinero, la banca incrementa su saldo.

- Se puede jugar a un número concreto. Habrá 4 hilos que eligen números al azar del 1 al 36 (no el 0) y restarán 10 euros de su saldo para apostar a ese ese número. Si sale su número su saldo se incrementa en 360 euros (36 veces lo apostado).
- Se puede jugar a par/impar. Habrá 4 hilos que eligen al azar si apuestan a que saldrá un número par o un número impar. Siempre restan 10 euros para apostar y si ganan incrementan su saldo en 20 euros.
- Se puede jugar a la "martingala". Habrá 4 hilos que eligen números al azar. Elegirán un número y empezarán restando 10 euros de su saldo para apostar a ese número. Si ganan incrementan su saldo en 360 euros. Si pierden jugarán el doble de su apuesta anterior (es decir, 20, luego 40, luego 80, y así sucesivamente)
- La banca acepta todas las apuestas pero nunca paga más dinero del que tiene.
- Si sale el 0, todo el mundo pierde y la banca se queda con todo el dinero.

2.14 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.

2.15 Una (mala) solución al problema de los barberos

Prueba la siguiente solución:

2.15.1 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);
```

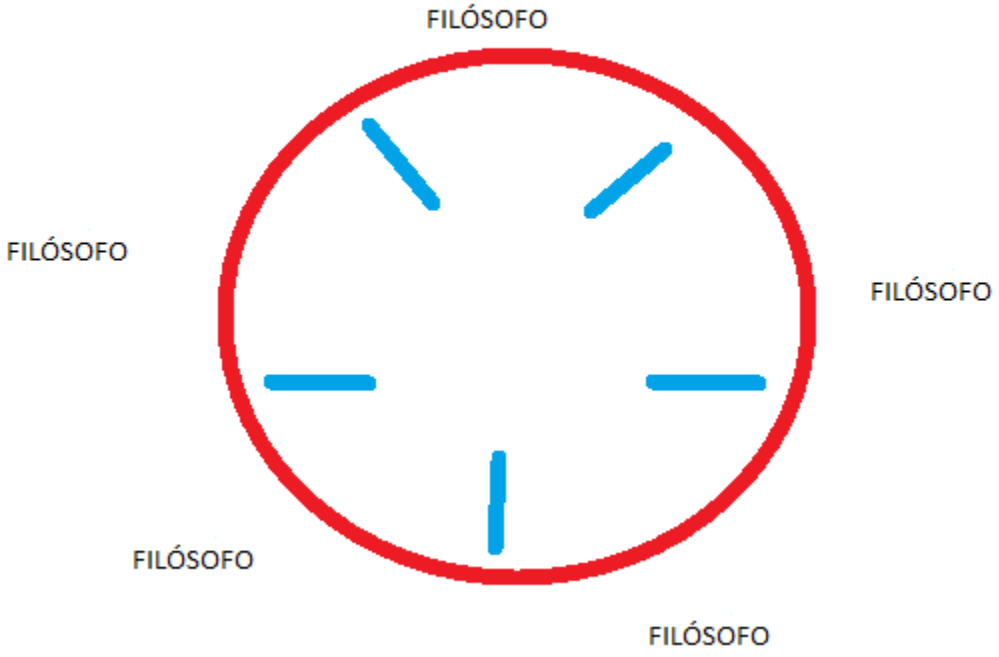

Figura 2.2: Los barberos dormilones

```
                } 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*/
                }
        }
}
```

2.15.2 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();
        }
}
```

2.15.3 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);
        }
}
```

2.15.4 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*/
        }
}
```

2.15.5 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*.

2.15.6 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
```

2.16 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.

2.17 Solución

2.17.1 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;
        }
}
```

2.17.2 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();
        }
}
```

2.17.3 La clase Productor

```
public class Productor implements Runnable{
        private Random                                                  generadorAzar;
        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;
                        }
                }
        }
}
```

2.17.4 La clase consumidor

```
public class Consumidor implements Runnable {
        private Random                                                  generadorAzar;
        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 ;
                        }
                }
        }
}
```

2.17.5 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();
                        }
                }
        }
}
```

2.18 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.

2.19 Documentación.

La estructura general de toda aplicación multihilo es algo similar a lo siguiente:

- Habrá tareas que se puedan paralelizar. Dichas tareas heredarán de `Runnable`.
- Habrá procesos que pueden ser accedidos por muchos hilos. Estos objetos **deberán usar synchronized**
- Habrá que crear los recursos compartidos y crear los hilos. Esta operación se debe hacer en este orden.

2.20 Depuración.

CAPÍTULO 3

Programación de comunicaciones en red

3.1 Comunicación entre aplicaciones.

En Java toda la comunicación vista en primer curso de DAM consiste en dos cosas

- Entrada/salida por consola: con las clases `System.in` o `System.out`.
- Lectura/escritura en ficheros: con las clases File y similares.

Se puede avanzar un paso más utilizando Java para enviar datos a través de Internet a otro programa Java remoto, que es lo que haremos en este capítulo.

3.2 Roles cliente y servidor.

Cuando se hacen programas Java que se comuniquen lo habitual es que uno o varios actúen de cliente y uno o varios actúen de servidores.

- Servidor: espera peticiones, recibe datos de entrada y devuelve respuestas.
- Cliente: genera peticiones, las envía a un servidor y espera respuestas.

Un factor fundamental en los servidores es que tienen que ser capaces de procesar varias peticiones a la vez: **deben ser multihilo**.

Su arquitectura típica es la siguiente:

```
while (true){
        peticion=esperarPeticion();
        hiloAsociado=new Hilo();
        hiloAsociado.atender(peticion);
}
```

3.3 Recordatorio de los flujos en Java

3.3.1 InputStreams y OutputStreams

Manejan bytes a secas. Por ejemplo, si queremos leer un fichero byte a byte usaremos `FileInputStream` y si queremos escribir usaremos `FileOutputStream`.

Son operaciones a muy bajo nivel que usaremos muy pocas veces (por ejemplo, solo si quisiéramos cambiar el primer byte de un archivo). En general usaremos otras clases más cómodas de usar.

3.3.2 Readers y Writers

En lugar de manejar *bytes* manejan *caracteres* (recordemos que hoy en día y con Unicode una letra como la *ñ* en realidad podría ocupar más de un byte).

Así, cuando queramos leer letras de un archivo usaremos clases como `FileReader` y `FileWriter`.

Las clases `Readers` y `Writers` en realidad se apoyan sobre las `InputStreams` y `OutputStreams`.

A veces nos interesará mezclar conceptos y por ejemplo poder tener una clase que use caracteres cuando a lo mejor Java nos ha dado una clase que usa bytes. Así, por ejemplo `InputStreamReader` puede coger un objeto que lea bytes y nos devolverá caracteres. De la misma forma `OutputStreamWriter` coge letras y devuelve los bytes que la componen.

3.3.3 `BufferedReaders` y `PrintWriters`

Cuando trabajamos con caracteres (que recordemos pueden tener varios bytes) normalmente no trabajamos de uno en uno. Es más frecuente usar **líneas** que se leen y escriben de una sola vez. Así por ejemplo, la clase `PrintWriter` tiene un método `print(ln)` que puede imprimir elementos complejos como `floats` o cadenas largas.

Además, Java ofrece clases que gestionan automáticamente los *buffers* por nosotros lo que nos da más comodidad y eficiencia. Por ello es muy habitual hacer cosas como esta:

```
lectorEficiente = new
        BufferedReader(new FileReader("fich1.txt"));
escritorEficiente = new
        BufferedWriter(new FileWriter("fich2.txt"));
```

En el primer caso creamos un objeto `FileReader` que es capaz de leer caracteres de `fich1.txt`. Como esto nos parece poco práctico creamos otro objeto a partir del primero de tipo `BufferedReader` que nos permitirá leer bloques enteros de texto.

De hecho, si se comprueba la ayuda de la clase `FileReader` se verá que solo hay un método `read` que devuelve un `int`, es decir el siguiente **carácter** disponible, lo que hace que el método sea muy incómodo. Sin embargo `BufferedReader` nos resuelve esta incomodidad permitiéndonos trabajar con líneas.

3.4 Elementos de programación de aplicaciones en red. Librerías.

En Java toda la infraestructura de clases para trabajar con redes está en el paquete `java.net`.

En muchos casos nuestros programas empezarán con la sentencia `import java.net.*` pero muchos entornos (como Eclipse) son capaces de importar automáticamente las clases necesarias.

3.4.1 La clase URL

La clase URL permite gestionar accesos a URLs del tipo `http://marca.com/fichero.html` y descargar cosas con bastante sencillez.

Al crear un objeto URL se debe capturar la excepción `MalformedURLException` que sucede cuando hay algún error en la URL, como por ejemplo escribir `htp://marca.com` en lugar de `http://marca.com` (obsérvese que el primero tiene un sola t en http en lugar de dos).

La clase URL nos ofrece un método `openStream` que nos devuelve un flujo básico de bytes. Podemos crear objetos más sofisticados para leer bloques como muestra el programa siguiente:

```
public class GestorDescargas {
        public void descargarArchivo(
                        String url_descargar,
                        String nombreArchivo){
                System.out.println("Descargando "
                                +url_descargar);
                try {
                        URL laUrl=new URL(url_descargar);
                        InputStream is=laUrl.openStream();
                        InputStreamReader reader=
                                        new InputStreamReader(is);
                        BufferedReader bReader=
                                        new BufferedReader(reader);
                        FileWriter escritorFichero=
                                new FileWriter(nombreArchivo);
                        String linea;
                        while ((linea=bReader.readLine())!=null){
                                escritorFichero.write(linea);
                        }
                        escritorFichero.close();
                        bReader.close();
                        reader.close();
                        is.close();
                } catch (MalformedURLException e) {
                        System.out.println("URL mal escrita!");
                        return ;
                } catch (IOException e) {
                        System.out.println(
                                        "Fallo en la lectura del fichero");
                        return ;
                }
        }
        public static void main (String[] argumentos){
                GestorDescargas gd=new GestorDescargas();
                String base=
                        "http://10.13.0.20:8000"+
                                        "/ServiciosProcesos/textos/";
                for (int i=1; i<=5; i++){
                        String url=base+"tema"+i+".rst";
                        gd.descargarArchivo(url);
                }
        }
}
```

3.5 Funciones y objetos de las librerías.

La clase URL proporciona un mecanismo muy sencillo pero por desgracia completamente atado al protocolo de las URL.

Java ofrece otros objetos que permiten tener un mayor control sobre lo que se envía o recibe a través de la red. Por desgracia esto implica que en muchos casos tendremos solo flujos de bajo nivel (streams).

En concreto Java ofrece dos elementos fundamentales para crear programas que usen redes

- Sockets
- ServerSockets

3.5.1 Repaso de redes

En redes el protocolo IP es el responsable de dos cuestiones fundamentales:

- Establecer un sistema de direcciones universal (direcciones IP)
- Establecer los mecanismos de enrutado.

Como programadores el segundo no nos interesa, pero el primero será absolutamente fundamental para contactar con programas que estén en una ubicación remota.

Una ubicación remota *siempre* tendrá una dirección IP pero *solo a veces tendrá un nombre DNS*. Para nosotros no habrá diferencia ya que si es necesario el sistema operativo traducirá de nombre DNS a IP.

Otro elemento necesario en la comunicación en redes es el uso de un puerto de un cierto protocolo:

- TCP: ofrece fiabilidad a los programas.
- UDP: ofrece velocidad sacrificando la fiabilidad.

A partir de ahora cuando usemos un número de puerto habrá que comprobar si ese número ya está usado.

Por ejemplo, es mala idea que nuestros programas usen el puerto 80 TCP, probablemente ya esté en uso. Antes de usar un puerto en una aplicación comercial deberíamos consultar la lista de "IANA assigned ports".

En líneas generales se pueden usar los puertos desde 1024 TCP a 49151 TCP, pero deberíamos comprobar que el número que elegimos no sea un número usado por un puerto de alguna aplicación que haya en la empresa.

En las prácticas de clase usaremos el 9876 TCP. Si se desea conectar desde el instituto con algún programa ejecutado en casa se deberá "abrir el puerto 9876 TCP". Abrir un puerto consiste en configurar el router para que **SÍ ACEPTE TRÁFICO INICIADO DESDE EL EXTERIOR** cosa que no hace nunca por motivos de protección.

3.6 Sockets.

Un *socket* es un objeto Java que nos permite contactar con un programa o servidor remoto. Dicho objeto nos proporcionará flujos de entrada y/o salida y podremos comunicarnos con dicho programa.

Existe otro tipo de sockets, los *ServerSocket*. Se utilizan para crear programas que acepten conexiones o peticiones.

Todos los objetos mencionados en este tema están en el paquete `java.net`.

3.7 Creación de sockets.

En el siguiente código puede verse el proceso básico de creación de un socket. En los párrafos siguientes explicaremos el significado de los bloques de código.:

```
public class Conector {
        public static void main(String[] args) {
                String destino="www.google.com";
                int puertoDestino=80;
                Socket socket=new Socket();
                InetSocketAddress direccion=new InetSocketAddress(
                                destino, puertoDestino);
```

```
                try {
                        socket.connect(direccion);
                        //Si llegamos aquí es que la conexión
                        //sí se hizo.

                        InputStream is=socket.getInputStream();
                        OutputStream os=socket.getOutputStream();


                } catch (IOException e) {
                        System.out.println(
                                "No se pudo establecer la conexion "+
                                " o hubo un fallo al leer datos."
                        );
                }
        }
}
```

Para poder crear un socket primero necesitamos una dirección con la que contactar. Toda dirección está formada por dirección IP (o DNS) y un puerto. En nuestro caso intentaremos contactar con `www.google.com:80`.

```
String destino="www.google.com";
int puertoDestino=80;
Socket socket=new Socket();
InetSocketAddress direccion=new
        InetSocketAddress(
                        destino, puertoDestino);
```

3.8 Enlazado y establecimiento de conexiones.

El paso crítico para iniciar la comunicación es llamar al método `connect`. Este método puede disparar una excepción del tipo `IOException` que puede significar dos cosas:

- La conexión no se pudo establecer.
- Aunque la conexión se estableció no fue posible leer o escribir datos.

Así, la conexión debería realizarse así:

```
try {
        socket.connect(direccion);
        //Si llegamos aquí es que la conexión
        //sí se hizo.

        InputStream is=socket.getInputStream();
        OutputStream os=socket.getOutputStream();

}  //Fin del try
catch (IOException e) {
        System.out.println(
                "No se pudo establecer la conexion "+
                " o hubo un fallo al leer datos."
        );
} //Fin del catch IOException
```

3.9 Utilización de sockets para la transmisión y recepción de información.

La clase `Socket` tiene dos métodos llamados `getInputStream` y `getOutputSream` que nos permiten obtener *flujos orientados a bytes*. Recordemos que es posible crear nuestros propios flujos, con más métodos que ofrecen más comodidad.

3.9.1 El ejemplo completo

Podemos contactar con un programa cualquiera escrito en cualquier lenguaje y enviar las peticiones de acuerdo a un protocolo. Nuestro programa podrá leer las respuestas independientemente de como fuera el servidor.

```
public class Conector {
        public static void main(String[] args) {
                System.out.println("Iniciando...");
                String destino="10.8.0.253";
                int puertoDestino=80;
                Socket socket=new Socket();
                InetSocketAddress direccion=new InetSocketAddress(
                                destino, puertoDestino);
                try {
                        socket.connect(direccion);
                        //Si llegamos aquí es que la conexión
                        //sí se hizo.

                        InputStream is=socket.getInputStream();
                        OutputStream os=socket.getOutputStream();

                        //Flujos que manejan caracteres
                        InputStreamReader isr=
                                        new InputStreamReader(is);
                        OutputStreamWriter osw=
                                        new OutputStreamWriter(os);

                        //Flujos de líneas
                        BufferedReader bReader=
                                        new BufferedReader(isr);
                        PrintWriter pWriter=
                                        new PrintWriter(osw);


                        pWriter.println("GET /index.html");
                        pWriter.flush();
                        String linea;
                        FileWriter escritorArchivo=
                                        new FileWriter("resultado.txt");
                        while ((linea=bReader.readLine()) != null ){
                                escritorArchivo.write(linea);
                        }
                        escritorArchivo.close();
                        pWriter.close();
                        bReader.close();
                        isr.close();
                        osw.close();
                        is.close();
                        os.close();
```

```
                } catch (IOException e) {
                        System.out.println(
                                "No se pudo establecer la conexion "+
                                " o hubo un fallo al leer datos."
                        );
                } //Fin del catch
        } //Fin del main
} //Fin de la clase Conector
```

3.10 Programación de aplicaciones cliente y servidor.

Al crear aplicaciones cliente y servidor puede ocurrir que tengamos que implementar varias operaciones:

- Si tenemos que programar el servidor **deberemos definir un protocolo** de acceso a ese servidor.
- Si tenemos que programar solo el cliente **necesitaremos conocer el protocolo de acceso** a ese servidor.
- Si tenemos que programar los dos tendremos que empezar por **definir el protocolo de comunicación entre ambos**.

En el ejemplo siguiente puede verse un ejemplo para Python 3 que implementa un servidor de cálculo. El servidor tiene un protocolo muy rígido (demasiado) que consiste en lo siguiente:

1. El servidor espera que primero envíen la operación que puede ser + o −. La operación debe terminar con un fin de línea UNIX (\n)
2. Despues acepta un número de dos cifras (ni una ni tres) terminado en un fín de línea UNIX.
3. Despues acepta un segundo número de dos cifras terminado en un fin de línea UNIX.

```
import socketserver
TAM_MAXIMO_PARAMETROS=64
PUERTO=9876
class GestorConexion(
        socketserver.BaseRequestHandler):

        def leer_cadena(self, LONGITUD):
                cadena=self.request.recv(LONGITUD)
                return cadena.strip()

        def convertir_a_cadena(self, bytes):
                return bytes.decode("utf-8")

        def calcular_resultado(
                self, n1, op, n2):
                n1=int(n1)
                n2=int(n2)

                op=self.convertir_a_cadena(op)
                if (op=="+"):
                        return n1+n2
                if (op=="-"):
                        return n1-n2
                return 0
        """Controlador de evento 'NuevaConexion"""
        def handle(self):
                direccion=self.client_address[0]
                operacion    =    self.leer_cadena(2)
```

```
                num1        =   self.leer_cadena(3)
                num2        =   self.leer_cadena(3)
                print (direccion+" pregunta:"+str(num1)+" "+str(operacion)+"
↪"+str(num2))

                resultado=self.calcular_resultado(num1, operacion, num2)
                print ("Devolviendo a " + direccion+" el resultado "+str(resultado))
                bytes_resultado=bytearray(str(resultado), "utf-8");
                self.request.send(bytes_resultado)
servidor=socketserver.TCPServer(("10.13.0.20", 9876), GestorConexion)
print ("Servidor en marcha.")
servidor.serve_forever()
```

La comunicación Java con el servidor sería algo así:

```
byte[] bSuma="+\n".getBytes();
byte[] bOp1="42\n".getBytes();
byte[] bOp2="34\n".getBytes();

os.write(bSuma);
os.write(bOp1);
os.write(bOp2);
os.flush();

InputStreamReader isr=
        new InputStreamReader(is);
BufferedReader br=
        new BufferedReader(isr);
String cadenaRecibida=br.readLine();
System.out.println("Recibido:"+
                cadenaRecibida);

is.close();
os.close();
socket.close();
```

3.10.1 Ejemplo de servidor Java

Supongamos que se nos pide crear un servidor de operaciones de cálculo que sea menos estricto que el anterior:

- Cualquier parámetro que envíe el usuario debe ir terminado en un fin de línea UNIX (“n”).
- El usuario enviará primero un símbolo “+”, “-”, “*” o “/”.
- Despues se puede enviar un positivo de 1 a 8 cifras. El usuario podría equivocarse y enviar en vez de “3762” algo como “37a62”. En ese caso se asume que el parámetro es 0.
- Despues se envía un segundo positivo de 1 a 8 cifras igual que el anterior.
- Cuando se haya procesado todo el servidor contestará al cliente con un positivo de 1 a 12 cifras.

Antes de empezar crear el código que permita procesar estos parámetros complejos.

```
public class ServidorCalculo {
        public int extraerNumero(String linea){
                /* 1. Comprobar si es un número
                 * 2. Ver si el número es correcto (32a75)
                 * 3. Ver si tiene de 1 a 8 cifras
```

```
                 */
                int numero;
                try{
                        numero=Integer.parseInt(linea);
                }
                catch (NumberFormatException e){
                        numero=0;
                }
                /* Si el número es mayor de 100 millones no
                 * es válido tampoco
                 */
                if (numero>=100000000){
                        numero=0;
                }
                return numero;

        }
        public void escuchar(){
                System.out.println("Arrancado el servidor");

                while (true){

                }
        }
}
```

Así, el código completo del servidor sería:

```
public class ServidorCalculo {
        public int extraerNumero(String linea){
                /* 1. Comprobar si es un número
                 * 2. Ver si el número es correcto (32a75)
                 * 3. Ver si tiene de 1 a 8 cifras
                 */
                int numero;
                try{
                        numero=Integer.parseInt(linea);
                }
                catch (NumberFormatException e){
                        numero=0;
                }
                /* Si el número es mayor de 100 millones no
                 * es válido tampoco
                 */
                if (numero>=100000000){
                        numero=0;
                }
                return numero;
        }

        public int calcular(String op, String n1, String n2){
                int resultado=0;
                char simbolo=op.charAt(0);
                int num1=this.extraerNumero(n1);
                int num2=this.extraerNumero(n2);
                if (simbolo=='+'){
                        resultado=num1+num2;
                }
```

```
                return resultado;
        }

        public void escuchar() throws IOException{
                System.out.println("Arrancado el servidor");
                ServerSocket socketEscucha=null;
                try {
                        socketEscucha=new ServerSocket(9876);
                } catch (IOException e) {
                        System.out.println(
                                        "No se pudo poner un socket "+
                                        "a escuchar en TCP 9876");
                        return;
                }
                while (true){
                        Socket conexion=socketEscucha.accept();
                        System.out.println("Conexion recibida!");
                        InputStream is=conexion.getInputStream();
                        InputStreamReader isr=
                                        new InputStreamReader(is);
                        BufferedReader bf=
                                        new BufferedReader(isr);
                        String linea=bf.readLine();
                        String num1=bf.readLine();
                        String num2=bf.readLine();
                        /* Calculamos el resultado*/
                        Integer result=this.calcular(linea, num1, num2);
                        OutputStream os=conexion.getOutputStream();
                        PrintWriter pw=new PrintWriter(os);
                        pw.write(result.toString()+"\n");
                        pw.flush();
                }
        }
}
```

Y el cliente sería:

```
public class ClienteCalculo {
        public static BufferedReader getFlujo(InputStream is){
                InputStreamReader isr=
                                new InputStreamReader(is);
                BufferedReader bfr=
                                new BufferedReader(isr);
                return bfr;
        }
        /**
         * @param args
         * @throws IOException
         */
        public static void main(String[] args) throws IOException {
                InetSocketAddress direccion=new
                                InetSocketAddress("10.13.0.20", 9876);
                Socket socket=new Socket();
                socket.connect(direccion);
                BufferedReader bfr=
                                ClienteCalculo.getFlujo(
                                                socket.getInputStream());
                PrintWriter pw=new
```

```
                                PrintWriter(socket.getOutputStream());
                pw.print("+\n");
                pw.print("42\n");
                pw.print("84\n");
                pw.flush();
                String resultado=bfr.readLine();
                System.out.println
                        ("El resultado fue:"+resultado);
        }
}
```

3.11 Utilización de hilos en la programación de aplicaciones en red.

En el caso de aplicaciones que necesiten aceptar varias conexiones **habrá que mover todo el código de gestión de peticiones a una clase que implemente Runnable**

Ahora el servidor será así:

```
while (true){
        Socket conexion=socketEscucha.accept();
        System.out.println("Conexion recibida");
        Peticion p=new Peticion(conexion);
        Thread hilo=new Thread(p);
        hilo.start();
}
```

Pero ahora tendremos una clase Petición como esta:

```
public class Peticion implements Runnable{
        BufferedReader bfr;
        PrintWriter pw;
        Socket socket;
        public Peticion(Socket socket){
                this.socket=socket;
        }
        public int extraerNumero(String linea){
                /* 1. Comprobar si es un número
                 * 2. Ver si el número es correcto (32a75)
                 * 3. Ver si tiene de 1 a 8 cifras
                 */
                int numero;
                try{
                        numero=Integer.parseInt(linea);
                }
                catch (NumberFormatException e){
                        numero=0;
                }
                /* Si el número es mayor de 100 millones no
                 * es válido tampoco
                 */
                if (numero>=100000000){
                        numero=0;
                }
                return numero;

        }
```

```
        public int calcular(String op, String n1, String n2){
                int resultado=0;
                char simbolo=op.charAt(0);
                int num1=this.extraerNumero(n1);
                int num2=this.extraerNumero(n2);
                if (simbolo=='+'){
                        resultado=num1+num2;
                }
                return resultado;
        }
        public void run(){
                try {
                        InputStream is=socket.getInputStream();
                        InputStreamReader isr=
                                        new InputStreamReader(is);
                        bfr=new BufferedReader(isr);
                        OutputStream os=socket.getOutputStream();
                        pw=new PrintWriter(os);
                        String linea;
                        while (true){
                                linea = bfr.readLine();
                                String num1=bfr.readLine();
                                String num2=bfr.readLine();
                                /* Calculamos el resultado*/
                                Integer result=this.calcular(linea, num1, num2);
                                System.out.println("El servidor dio resultado:
↪"+result);
                                pw.write(result.toString()+"\n");
                                pw.flush();
                        }
                } catch (IOException e) {
                }
        }
}
```

3.11.1 Ejercicio: servicio de ordenación

Crear una arquitectura cliente/servidor que permita a un cliente, enviar dos cadenas a un servidor para saber cual de ellas va antes que otra:

- Un cliente puede enviar las cadenas “hola”, “mundo”. El servidor comprobará que en el diccionario la primera va antes que la segunda, por lo cual contestará “hola”, “mundo”.
- Si el cliente enviase “mundo”, “hola” el servidor debe devolver la respuesta “hola”, “mundo”.

Debido a posibles mejoras futuras, se espera que el servidor sea capaz de saber qué versión del protocolo se maneja. Esto es debido a que en el futuro se espera lanzar una versión 2 del protocolo en la que se puedan enviar varias cadenas seguidas.

Crear el protocolo, el código Java del cliente y el código Java del servidor con capacidad para procesar muchas peticiones a la vez (multihilo).

Se debe aceptar que un cliente que ya tenga un socket abierto envíe todas las parejas de cadenas que desee.

3.11.2 Una clase Protocolo

Dado que los protocolos pueden ser variables puede ser útil encapsular el comportamiento del protocolo en una pequeña clase separada:

```
public class Protocolo {
        private final String terminador="\n";
        public String getMensajeVersion(int version){
                Integer i=version;
                return i.toString()+terminador;
        }
        public int getNumVersion(String mensaje){
                Integer num=Integer.parseInt(mensaje);
                return num;
        }
        public String getMensaje(String cadena){
                return cadena+terminador;
        }
}
```

3.11.3 Una clase con funciones de utilidad

Algunas operaciones son muy sencillas, pero muy engorrosas. Alargan el código innecesariamente y lo hacen más difícil de entender. Si además se realizan a menudo puede ser interesante empaquetar toda la funcionalidad en una clase.

```
public class Utilidades {
        /* Obtiene un flujo de escritura
        a partir de un socket*/
        public PrintWriter getFlujoEscritura
                (Socket s) throws IOException{
                OutputStream os=s.getOutputStream();
                PrintWriter pw=new PrintWriter(os);
                return pw;
        }
        /* Obtiene un flujo de lectura
        a partir de un socket*/
        public BufferedReader
                getFlujoLectura(Socket s)
                                throws IOException{
                InputStream is=s.getInputStream();
                InputStreamReader isr=
                                new InputStreamReader(is);
                BufferedReader bfr=new BufferedReader(isr);
                return bfr;
        }
}
```

3.11.4 La clase Petición

```
public class Peticion implements Runnable {
        Socket conexionParaAtender;

        public Peticion ( Socket s ){
```

```
                this.conexionParaAtender=s;
        }
        @Override
        public void run() {
                try{
                        PrintWriter flujoEscritura=
                                Utilidades.getFlujoEscritura(
                                                this.conexionParaAtender
                                                );
                        BufferedReader flujoLectura=
                                Utilidades.getFlujoLectura(
                                                conexionParaAtender);
                        String protocolo=
                                        flujoLectura.readLine();
                        int numVersion=
                                        Protocolo.getNumVersion(protocolo);
                        if (numVersion==1){
                                String linea1=
                                                flujoLectura.readLine();
                                String linea2=
                                                flujoLectura.readLine();
                                //Linea 1 va despues en el
                                if (linea1.compareTo(linea2)>0){
                                         dicc
                                        flujoEscritura.println(linea2);
                                        flujoEscritura.println(linea1);
                                        flujoEscritura.flush();
                                } else {
                                        flujoEscritura.println(linea1);
                                        flujoEscritura.println(linea2);
                                        flujoEscritura.flush();
                                }
                        }
                }
                catch (IOException e){
                        System.out.println(
                                        "No se pudo crear algún flujo");
                        return ;
                }
        }
}
```

3.11.5 La clase Servidor

```
public class ServidorOrdenacion {
        public void escuchar() throws IOException{
                ServerSocket socket;
                try{
                        socket=new ServerSocket(9876);
                }
                catch(Exception e){
                        System.out.println("No se pudo arrancar");
                        return ;
                }
                while (true){
                        System.out.println("Servidor esperando");
```

```
                        Socket conexionCliente=
                                        socket.accept();
                        System.out.println("Alguien conectó");
                        Peticion p=
                                        new Peticion(conexionCliente);
                        Thread hiloAsociado=
                                        new Thread(p);
                        hiloAsociado.start();
                }
        } // Fin del método escuchar
        public static void  main(String[] argumentos){
                ServidorOrdenacion s=
                                new ServidorOrdenacion();
                try {
                        s.escuchar();
                } catch (Exception e){
                        System.out.println("No se pudo arrancar");
                        System.out.println(" el cliente o el serv");
                }
        }
}
```

3.11.6 La clase Cliente

```
public class Cliente {
        public void ordenar(String s1, String s2) throws IOException{
                InetSocketAddress direccion=
                                new InetSocketAddress("10.13.0.20", 9876);
                Socket conexion=
                                new Socket();
                conexion.connect(direccion);
                System.out.println("Conexion establecida");
                /* Ahora hay que crear flujos de salida, enviar
                 * cadenas por allí y esperar los resultados.
                 */
                try{

                        BufferedReader flujoLectura=
                                Utilidades.getFlujoLectura(conexion);
                        PrintWriter flujoEscritura=
                                Utilidades.getFlujoEscritura(conexion);

                        flujoEscritura.println("1");
                        flujoEscritura.println(s1);
                        flujoEscritura.println(s2);
                        flujoEscritura.flush();
                        String linea1=flujoLectura.readLine();
                        String linea2=flujoLectura.readLine();
                        System.out.println("El servidor devolvió "+
                                        linea1 + " y "+linea2);
                } catch (IOException e){

                }
        }
        public static void main(String[] args) {
                Cliente c=new Cliente();
```

```
                try {
                        c.ordenar("aaa", "bbb");
                } catch (IOException e) {
                        System.out.println("Fallo la conexion o ");
                        System.out.println("los flujos");
                } //Fin del catch
        } //Fin del main
} //Fin de la clase
```

3.11.7 Ampliación

Finalmente la empresa va a necesitar una versión mejorada del servidor que permita a otros cliente enviar un número de palabras y luego las palabras. Se desea hacer todo sin romper la compatibilidad con los clientes viejos. Mostrar el código Java del servidor y del cliente.

En el servidor se añade este código extra a la hora de comprobar el protocolo:

```
if (numVersion==2){
        System.out.println("Llegó un v2");
        String lineaCantidadPalabras=
        flujoLectura.readLine();
        int numPalabras=
                Integer.parseInt
                (lineaCantidadPalabras);
        String[] palabras=
                                new String[numPalabras];
        for (int i=0;i<numPalabras;i++){
                palabras[i]=
                        flujoLectura.readLine();
        }
        palabras=this.ordenar(palabras);
        for (int i=0; i<palabras.length; i++){
                flujoEscritura.println(palabras[i]);
        }
        flujoEscritura.flush();
}
```

Y finalmente solo habría que implementar un método en la petición que reciba un vector de `String` (las palabras) y devuelva el mismo vector pero ordenado.

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.

4.11 Documentación.

4.12 Depuración.

CAPÍTULO 5

Utilización de técnicas de programación segura

5.1 Introducción

En general, cuando se envía algo a través de sockets se envía como "texto plano", es decir, no sabemos si hay alguien usando un sniffer en la red y por tanto no sabemos si alguien está capturando los datos.

En general, cualquier sistema que pretenda ser seguro necesitará usar cifrado.

5.2 Prácticas de programación segura.

Para enviar mensajes cifrados se necesita algún mecanismo o algoritmo para convertir un texto normal en uno más difícil de comprender.

El esquema general de todos los métodos es tener código como el siguiente:

```
public String cifrar (String texto, String clave)
{
}

public String descifrar(String texto,String clave)
{
}
```

5.2.1 Método César

Si el alfabeto es el siguiente:

ABCDEFGHIJKLMNÑOPQRSTUVWXYZ0123456789- BCDEFGHIJKLMNÑOPQRSTUVWXYZ0123456789-A

El mensaje HOLA MUNDO, con clave 1 sería así

HOLAMUNDO IPMBNVÑEP

El descifrado simplemente implicaría el método inverso. Si el desplazamiento es un valor distinto de 1, lo único que hay que hacer es construir el alfabeto rotado tantas veces como el desplazamiento

Esta clase implementa un sistema de rotado básico para poder efectuar

```
public class Cifrador {
        private String alfabeto=
                        "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ"+
```

```
                        "0123456789 ";
        private String alfabetoCifrado;
        /* Dada una cadena como
         * "ABC" y un número (p.ej 2)
         * devuelve la cadena rotada a la izq
         * tantas veces como indique el numero,
         * en este caso "CAB"
         */
        public String rotar(String cad,int numVeces){
                char[] resultado=new char[cad.length()];
                for (int i=0; i<cad.length();i++){
                        int posParaExtraer=(i+numVeces)%cad.length();
                        resultado[i]=cad.charAt(posParaExtraer);
                }
                String cadResultado=String.copyValueOf(resultado);
                return cadResultado;
        }
        public String cifrar
                (String mensaje, String clave){
                String mensajeCifrado="";

                return mensajeCifrado;
        }
        public String descifrar
                (String mensajeCifrado, String clave){
                String mensajeDescifrado="";

                return mensajeDescifrado;
        }
        public static void main(String[] args){
                Cifrador c=new Cifrador();
                String cad=c.rotar("ABCDEFG", 0);
                System.out.println(cad);
        }
}
```

5.3 Criptografía de clave pública y clave privada.

Los principales sistemas modernos de seguridad utilizan dos claves ,una para cifrar y otra para descifrar. Esto se puede usar de diversas formas.

5.4 Principales aplicaciones de la criptografía.

- Mensajería segura: todo el mundo da su clave de cifrado pero conserva la de descifrado. Si queremos enviar un mensaje a alguien cogemos su clave de cifrado y ciframos el mensaje que le enviamos. Solo él podrá descifrarlo.
- Firma digital: pilar del comercio electrónico. Permite verificar que un archivo no ha sido modificado.
- Mensajería segura: en este tipo de mensajería se intenta evitar que un atacante (quizá con un *sniffer*) consiga descifrar nuestros mensajes.
- Autenticación: los sistemas de autenticación intentan resolver una cuestión clave en la informática: **verificar que una máquina es quien dice ser**

5.5 Protocolos criptográficos.

En realidad protocolos criptográficos hay muchos, y suelen dividirse en sistemas simétricos o asimétricos.

- Los sistemas simétricos son aquellos basados en una función que convierte un mensaje en otro mensaje cifrado. Si se desea descifrar algo se aplica el proceso inverso con la misma clave que se usó.
- Los sistemas asimétricos utilizan una clave de cifrado y otra de descifrado. Aunque se tenga una clave es matemáticamente imposible averiguar la otra clave por lo que se puede dar a todo el mundo una de las claves (llamada habitualmente **clave pública**) y conservar la otra (llamada **clave privada**). Además, podemos usar las claves para lo que queramos y por ejemplo en unos casos cifraremos con la clave pública y en otros tal vez cifremos con la clave privada.

Hoy por hoy, las mayores garantías las ofrecen los asimétricos, de los cuales hay varios sistemas. El inconveniente que pueden tener los asimétricos es que son más lentos computacionalmente.

En este curso usaremos el cifrado asimétrico RSA.

5.6 Encriptación de información.

El siguiente código muestra como crear una clase que permita cifrar y descifrar textos.

```
public class GestorCifrado {
        KeyPair claves;
        KeyPairGenerator generadorClaves;
        Cipher cifrador;
        public GestorCifrado()
                        throws NoSuchAlgorithmException,
                        NoSuchPaddingException{
                generadorClaves=
                                KeyPairGenerator.getInstance("RSA");
                /*Usaremos una longitud de clave
                 * de 1024 bits */
                generadorClaves.initialize(1024);
                claves=generadorClaves.generateKeyPair();
                cifrador=Cipher.getInstance("RSA");
        }
        public PublicKey getPublica(){
                return claves.getPublic();
        }
        public PrivateKey getPrivada(){
                return claves.getPrivate();
        }

        public byte[] cifrar(byte[] paraCifrar,
                        Key claveCifrado
                        ) throws InvalidKeyException,
                        IllegalBlockSizeException,
                        BadPaddingException{
                byte[] resultado;
                /* Se pone el cifrador en modo cifrado*/
                cifrador.init(Cipher.ENCRYPT_MODE,
                                claveCifrado);
                resultado=cifrador.doFinal(paraCifrar);
                return resultado;
        }
```

```
    public byte[] descifrar(
                    byte[] paraDescifrar,
                    Key claveDescifrado)
                                    throws InvalidKeyException,
                                    IllegalBlockSizeException,
                                    BadPaddingException{
            byte[] resultado;
            /* Se pone el cifrador en modo descifrado*/
            cifrador.init(Cipher.DECRYPT_MODE,
                            claveDescifrado);
            resultado=cifrador.doFinal(paraDescifrar);
            return resultado;
    }



    public static void main(String[] args)
                    throws NoSuchAlgorithmException,
                    NoSuchPaddingException,
                    InvalidKeyException,
                    IllegalBlockSizeException,
                    BadPaddingException,
                    UnsupportedEncodingException
    {
            GestorCifrado gestorCifrado=
                            new GestorCifrado();
            String mensajeOriginal="Hola mundo";
            Key clavePublica=gestorCifrado.getPublica();

            byte[] mensajeCifrado=
                            gestorCifrado.cifrar(
                                            mensajeOriginal.getBytes(),
                                            clavePublica
            );
            String cadCifrada=
                            new String(mensajeCifrado, "UTF-8");

            System.out.println
                    ("Cadena original:"+mensajeOriginal);
            System.out.println
                    ("Cadena cifrada:"+cadCifrada);

            /* Cogemos la cadCifrada y la desciframos
             * con la otra clave
             */
            Key clavePrivada;
            clavePrivada=gestorCifrado.getPrivada();
            byte[] descifrada=
                            gestorCifrado.descifrar(
                                            mensajeCifrado,clavePrivada);
            String mensajeDescifrado;
            mensajeDescifrado=
                            new String(descifrada, "UTF-8");
            System.out.println(
                            "El mensaje descifrado es:"+
                                            mensajeDescifrado);
    }
```

```
}
```

Advertencia: Los objetos que cifran y descifran en Java utilizan estrictamente objetos `byte[]`, que son los que debemos manejar siempre. Las conversiones a `String` las hacemos nosotros para poder visualizar resultados.

5.7 Protocolos seguros de comunicaciones.

En general, ahora que ya conocemos sockets, el uso de servidores y clientes y el uso de la criptografía de clave asimétrica ya es posible crear aplicaciones que se comuniquen de forma muy segura.

En general, todo protocolo que queramos implementar dará estos pasos.

1. Todo cliente genera su pareja de claves.
2. Todo servidor genera su pareja de claves.
3. Cuando un cliente se conecte a un servidor, le envía su clave de cifrado y conserva la de descifrado.
4. Cuando un servidor recibe la conexión de un cliente recibe la clave de cifrado de dicho cliente.
5. El servidor envía su clave pública al cliente.
6. Ahora cliente y servidor pueden enviar mensajes al otro con la garantía de que solo servidor y cliente respectivamente pueden descifrar.

En realidad se puede asegurar más el proceso haciendo que en el paso 5 el servidor cifre su propia clave pública con la clave pública del cliente. De esta forma, incluso aunque alguien robara la clave privada del cliente tampoco tendría demasiado, ya que tendría que robar la clave privada del servidor.

5.8 Programación de aplicaciones con comunicaciones seguras.

Por fortuna Java dispone de clases ya prefabricadas que facilitan enormemente el que dos aplicaciones intercambios datos de forma segura a través de una red. Se deben considerar los siguientes puntos:

- El servidor debe tener su propio certificado. Si no lo tenemos, se puede generar primero una pareja de claves con la herramienta `keytool`, como se muestra en la figura adjunta. La herramienta guardará la pareja de claves en un almacén (el cual tiene su propia clave). Despues generaremos un certificado a partir de esa pareja con `keytool -export -file certificadoservidor.cer -keystore almacenclaves`.
- El código del servidor necesitará indicar el fichero donde se almacenan las claves y la clave para acceder a ese almacén.
- El cliente necesita indicar que confía en el certificado del servidor. Dicho certificado del servidor puede estar guardado (por ejemplo) en el almacén de claves del cliente.
- Aunque no suele hacerse también podría hacerse a la inversa y obligar al cliente a tener un certificado que el servidor pudiera importar, lo que aumentaría la seguridad.

Los pasos desglosados implican ejecutar estos comandos en el servidor:

```
# El servidor genera una pareja de claves que se almacena en un
#fichero llamado "clavesservidor". Dentro del fichero se indica
#un alias para poder referirnos a esa clave fácilmente
keytool -genkeypair -keyalg RSA
     -alias servidor -keystore clavesservidor
```

```
usuario@usuario-EQUIPO:~/repos/ServiciosProcesos$ keytool -genkeypair -keystore almacenclaves
Introduzca la contraseña del almacén de claves:
Volver a escribir la contraseña nueva:
¿Cuáles son su nombre y su apellido?
  [Unknown]:  Empresa ACME
¿Cuál es el nombre de su unidad de organización?
  [Unknown]:  IES Maestre de Calatrava
¿Cuál es el nombre de su organización?
  [Unknown]:  JCCM
¿Cuál es el nombre de su ciudad o localidad?
  [Unknown]:  Ciudad Real
¿Cuál es el nombre de su estado o provincia?
  [Unknown]:  Ciudad Real
¿Cuál es el código de país de dos letras de la unidad?
  [Unknown]:  ES
¿Es correcto CN=Empresa ACME, OU=IES Maestre de Calatrava, O=JCCM, L=Ciudad Real, ST=Ciudad Real, C=ES?
  [no]:  si

Introduzca la contraseña de clave para <mykey>
        (INTRO si es la misma contraseña que la del almacén de claves):
Volver a escribir la contraseña nueva:
```

Figura 5.1: Generando la pareja de claves del servidor.

```
#El servidor genera su "certificado", es decir un fichero que
#de alguna forma indica quien es él. El certificado se almacena
#en un fichero llamado clavesservidor y a partir de él queremos
#generar el certificado de un alias que tiene que haber llamado servidor
keytool --exportcert -alias servidor
     -file servidor.cer -keystore clavesservidor
```

En el cliente daremos estos pasos:

```
#Se genera una pareja de claves (en realidad no nos hace falta solo
#queremos tener un almacén de claves.
keytool -genkeypair -keyalg RSA -alias cliente -keystore clavescliente

#Se importa el certificado del servidor indicando que pertenece a
#la lista de certificados confiables.
keytool -importcert -trustcacerts -alias servidor -file servidor.cer -keystore␣
↪clavescliente
```

Una vez creados los ficheros iniciales se deben dar los siguientes pasos en Java (servidor y cliente van por separado):

1. El servidor debe cargar su almacén de claves (el fichero `clavesservidor`)
2. Ese almacén (cargado en un objeto Java llamado `KeyStore`), se usará para crear un gestor de claves (objeto `KeyManager`), el cual se obtiene a partir de una "fábrica" llamada `KeyManagerFactory`.
3. Se creará un contexto SSL (objeto `SSLContext`) a partir de la fábrica comentada.
4. El objeto `SSLContext` permitirá crear una fábrica de sockets que será la que finalmente nos permita tener un `SSLServerSocket`, es decir un socket de servidor que usará cifrado.

El código Java del servidor sería algo así:

```
public OtroServidor (String rutaAlmacen, String claveAlmacen){
            this.rutaAlmacen=rutaAlmacen;
            this.claveAlmacen=claveAlmacen;
    }
```

```
    public SSLServerSocket getServerSocketSeguro()
                    throws KeyStoreException, NoSuchAlgorithmException,
                    CertificateException, IOException,
                    KeyManagementException, UnrecoverableKeyException
    {
            SSLServerSocket serverSocket=null;
            /* Paso 1, se carga el almacén de claves*/
            FileInputStream fichAlmacen=
                            new FileInputStream(this.rutaAlmacen);
            /* Paso 1.1, se crea un almacén del tipo por defecto
             * que es un JKS (Java Key Store), a día de hoy*/
            KeyStore almacen=KeyStore.getInstance(KeyStore.getDefaultType());
            almacen.load(fichAlmacen, claveAlmacen.toCharArray());
            /* Paso 2: obtener una fábrica de KeyManagers que ofrezcan
             * soporte al algoritmo por defecto*/
            KeyManagerFactory fabrica=
                            KeyManagerFactory.getInstance(
                                KeyManagerFactory.getDefaultAlgorithm());
            fabrica.init(almacen, claveAlmacen.toCharArray());
            /* Paso 3:Intentamos obtener un contexto SSL
             * que ofrezca soporte a TLS (el sistema más
             * seguro hoy día) */
            SSLContext contextoSSL=SSLContext.getInstance("TLS");
            contextoSSL.init(fabrica.getKeyManagers(), null, null);
            /* Paso 4: Se obtiene una fábrica de sockets que permita
             * obtener un SSLServerSocket */
            SSLServerSocketFactory fabricaSockets=
                            contextoSSL.getServerSocketFactory();
            serverSocket=
                    (SSLServerSocket)
                            fabricaSockets.createServerSocket(puerto);
            return serverSocket;
    }
    public void escuchar()
        throws KeyManagementException, UnrecoverableKeyException,
            KeyStoreException, NoSuchAlgorithmException,
            CertificateException, IOException
    {
            SSLServerSocket socketServidor=this.getServerSocketSeguro();
            BufferedReader entrada;
            PrintWriter salida;
            while (true){
                    Socket connRecibida=socketServidor.accept();
                    System.out.println("Conexion segura recibida");
                    entrada=
                        new BufferedReader(
                        new InputStreamReader(connRecibida.getInputStream()));
                    salida=
                        new PrintWriter(
                            new OutputStreamWriter(
                            connRecibida.getOutputStream()));
                    String linea=entrada.readLine();
                    salida.println(linea.length());
                    salida.flush();
            }
    }
```

En el cliente se tienen que dar algunos pasos parecidos:

1. En primer lugar se carga el almacén de claves del cliente (que contiene el certificado del servidor y que es la clave para poder "autenticar" el servidor)
2. El almacén del cliente se usará para crear un "gestor de confianza" (`TrustManager`) que Java usará para determinar si puede confiar o no en una conexión. Usaremos un `TrustManagerFactory` que usará el almacén del cliente para crear objetos que puedan gestionar la confianza.
3. Se creará un contexto SSL (`SSLContext`) que se basará en los `TrustManager` que pueda crear la fábrica.
4. A partir del contexto SSL el cliente ya puede crear un socket seguro (`SSLSocket`) que puede usar para conectar con el servidor de forma segura.

El código del cliente sería algo así:

```
public class OtroCliente {
    String almacen="/home/usuario/clavescliente";
    String clave="abcdabcd";
    SSLSocket conexion;
    public OtroCliente(String ip, int puerto)
                    throws UnknownHostException, IOException,
                    KeyManagementException, NoSuchAlgorithmException,
                    KeyStoreException, CertificateException{

            conexion=this.obtenerSocket(ip,puerto);
    }
    /* Envía un mensaje de prueba para verificar que la conexión
     * SSL es correcta */
    public void conectar() throws IOException{
            System.out.println("Iniciando..");
            BufferedReader entrada;
            PrintWriter salida;
            entrada=new BufferedReader(new InputStreamReader(conexion.
↪getInputStream()));
            salida=new PrintWriter(new OutputStreamWriter(conexion.
↪getOutputStream()));
            /* De esta linea se intenta averiguar la longitud*/
            salida.println("1234567890");
            salida.flush();
            /* Si todo va bien, el servidor nos contesta el numero*/
            String num=entrada.readLine();
            int longitud=Integer.parseInt(num);
            System.out.println("La longitud devuelta es:"+longitud);

    }
    public SSLSocket obtenerSocket(String ip, int puerto)
                    throws KeyStoreException, NoSuchAlgorithmException,
                    CertificateException, IOException, KeyManagementException
    {
            System.out.println("Obteniendo socket");
            SSLSocket socket=null;
            /* Paso 1: se carga al almacén de claves
             * (que recordemos debe contener el
             * certificado del servidor)*/
            KeyStore almacenCliente=KeyStore.getInstance(KeyStore.getDefaultType());
            FileInputStream ficheroAlmacenClaves=
                            new FileInputStream( this.almacen );
            almacenCliente.load(ficheroAlmacenClaves, clave.toCharArray());
            System.out.println("Almacen cargado");
            /* Paso 2, crearemos una fabrica de gestores de confianza
             * que use el almacén cargado antes (que contiene el
```

```
             * certificado del servidor)
             */
            TrustManagerFactory fabricaGestoresConfianza=
                            TrustManagerFactory.getInstance(
                                            TrustManagerFactory.
↪getDefaultAlgorithm());
            fabricaGestoresConfianza.init(almacenCliente);
            System.out.println("Fabrica Trust creada");
            /*Paso 3: se crea el contexto SSL, que ofrezca
             * soporte al algoritmo TLS*/
            SSLContext contexto=SSLContext.getInstance("TLS");
            contexto.init(
                            null, fabricaGestoresConfianza.getTrustManagers(), null);
            /* Paso 4: Se crea un socket que conecte con el servidor*/
            System.out.println("Contexto creado");
            SSLSocketFactory fabricaSockets=
                            contexto.getSocketFactory();
            socket=(SSLSocket) fabricaSockets.createSocket(ip, puerto);
            /* Y devolvemos el socket*/
            System.out.println("Socket creado");
            return socket;
    }
}
```

5.9 Firmado de aplicaciones

Utilizando la criptografía de clave pública es posible "firmar" aplicaciones. El firmado es un mecanismo que permite al usuario de una aplicación el verificar que la aplicación no ha sido alterada desde que el programador la creó (virus o programas malignos, personal descontento con la empresa, etc...).

Antes de efectuar el firmado se debe disponer de un par de claves generadas con la herramienta `keytool` mencionada anteriormente. Supongamos que el almacén de claves está creado y que en él hay uno o varios *alias* creados. El proceso de firmado es el siguiente:

1. Crear la aplicación, que puede estar formada por un conjunto de clases pero que en última instancia tendrá un `main`.
2. Empaquetar la aplicación con `jar cfe Aplicacion.jar com.ies.Aplicacion DirectorioPaquete`. Este comando crea un fichero (`f`) JAR en el cual el punto de entrada (`e`) es la clase `com.ies.Aplicacion` (que es la que tendrá el `main`).
3. Puede comprobarse que la aplicación dentro del JAR se ejecuta correctamente con `java -jar Aplicacion.jar`.
4. Ahora se puede ejecutar `jarsigner Aplicacion.jar <alias>`.

Con estos pasos se tiene un aplicación firmada que el usuario puede verificar si así lo desea. De hecho, si se extrae el contenido del JAR con `jar -xf Aplicacion.jar` se extraen los archivos `.class` y un fichero `META-INF/Manifest` que se puede abrir con un editor para ver que realmente está firmado.

Para que otras personas puedan comprobar que nuestra aplicacion es correcta los programadores deberemos exportar un certificado que los usuarios puedan importar para hacer el verificado. Recordemos que el comando es:

```
keytool -exportcert -keystore ..\Almacen.store -file Programador.cer -alias␣
↪Programador
```

5.10 Verificado de aplicaciones

Si ahora otro usuario desea ejecutar nuestra aplicación deberá importar nuestro certificado. El proceso de verificado es simple:

1. El usuario importa el certificado.
2. Ahora que tiene el certificado puede comprobar la aplicación con `jarsigner -verify Aplicacion.jar <alias_del_programador>`

El comando deberá responder con algo como `jar verified`. Sin embargo si no tenemos un certificado firmado por alguna autoridad de certificación (CA) la herramienta se quejará de que algunos criterios de seguridad no se cumplen.

5.11 Ejercicio

Intenta extraer el archivo JAR y reemplaza el `.class` por alguna otra clase. Vuelve a crear el archivo .JAR y vuelve a intentar verificarlo, ¿qué ocurre?

5.12 Recordatorio

Hemos hecho el proceso de firmado y verificado con **certificados autofirmados**, lo cual es útil para practicar pero **completamente inútil desde el punto de vista de la seguridad**. Para que un certificado sea seguro debemos hacer que previamente alguna autoridad de certificación nos lo firme primero (para lo cual suele ser habitual el tener que pagar).

5.13 Política de seguridad.

5.14 Programación de mecanismos de control de acceso.

5.15 Pruebas y depuración.

CAPÍTULO 6

Anexos

6.1 La clase `UtilidadesFicheros`

A continuación se muestra el código completo de la clase `UtilidadesFicheros`:

```
public class UtilidadesFicheros {

    public static BufferedReader getBufferedReader(
        String nombreFichero) throws
    FileNotFoundException {
        FileReader lector;
        lector = new FileReader(nombreFichero);
        BufferedReader bufferedReader;
        bufferedReader = new BufferedReader(lector);
        return bufferedReader;
    }

    public static PrintWriter getPrintWriter(
        String nombreFichero) throws IOException {
        PrintWriter printWriter;
        FileWriter fileWriter;
        fileWriter = new FileWriter(nombreFichero);
        printWriter = new PrintWriter(fileWriter);
        return printWriter;
        //Fin de getPrintWriter
    }

    public static ArrayList<String> getLineasFichero(
        String nombreFichero) throws IOException {
        ArrayList<String> lineas = new ArrayList<String>();
        BufferedReader bfr = getBufferedReader(nombreFichero);
        //Leemos líneas del fichero...
        String linea = bfr.readLine();
        while (linea != null) {
            //Y las añadimos al array
            lineas.add(linea);
            linea = bfr.readLine();
        }
        //Fin del bucle que lee líneas
        return lineas;
    }

    //Fin de getLineasFichero
```

```
    public static long getSuma(String[]
                               listaNombresFichero) {
        long suma = 0;
        ArrayList<String> lineas;
        String lineaCantidad;
        long cantidad;
        for (String nombreFichero : listaNombresFichero) {
            try {
                //Recuperamos todas las lineas
                lineas = getLineasFichero(nombreFichero);
                //Pero solo nos interesa la primera
                lineaCantidad = lineas.get(0);
                //Convertimos la linea a número
                cantidad = Long.parseLong(lineaCantidad);
                //Y se incrementa la suma total
                suma = suma + cantidad;
            } catch (IOException e) {
                System.err.println("Fallo al procesar el fichero "
                                   + nombreFichero);
                //Fin del catch
            }
            //Fin del for que recorre los nombres de fichero
        }
        return suma;
    }
}
```

6.2 Solución al problema del simulador de Casino

Se ha optado por un diseño muy simple y con un acoplamiento muy alto. En pocas palabras hay solo dos clases:

1. La clase `Banca`: ejecuta toda la simulación y acepta apuestas de jugadores. La banca crea los Jugadores (que serán hilos), y luego pasará a ejecutar la simulación, que puede pasar por diversos estados. Algunos métodos de la banca deben ser `synchronized`, ya que todos los hilos Jugador tienen una referencia a la clase `Banca` y por lo tanto **puede haber varios hilos intentando entrar en un cierto método**.
2. La clase `Jugador`: no necesita métodos `synchronized`, ya que cada `Jugador` accede solo a su propia información. *Sin embargo, si hubiese más de una banca (quizá porque un Jugador jugase a varias ruletas a la vez) la situación sería muy distinta.*
3. En realidad, la clase `Jugador` es solo una base para poder crear distintos tipos de Jugador que siguen distintas estrategias.

En las secciones siguientes se ilustra el código de las distintas clases.

6.2.1 La clase Banca

```
public class Banca {

    protected long saldo;

    protected boolean enBancarrota;

    protected Random generador;
```

```
    protected boolean sePuedenHacerApuestas;

    protected int numeroGanador;

    public enum Estado {

        INICIO, ACEPTANDO_APUESTAS, RULETA_GIRANDO, PAGANDO_APUESTAS, EN_BANCARROTA
    }

    ;

    private Estado estadoRuleta;

    private ArrayList<Jugador> apostadores;

    public Banca(long saldoInicial) {
        saldo = saldoInicial;
        enBancarrota = false;
        estadoRuleta = Estado.INICIO;
        generador = new Random();
        apostadores = new ArrayList<Jugador>();
    }

    public synchronized boolean enBancarrota() {
        return enBancarrota;
    }

    public synchronized void sumarSaldo(
        long cantidad) {
        saldo = saldo + cantidad;
    }

    public synchronized void restarSaldo(
        long cantidad) {
        if (saldo - cantidad <= 0) {
            saldo = 0;
            estadoRuleta = Estado.EN_BANCARROTA;
            return;
        }
        saldo = saldo - cantidad;
    }

    public synchronized void aceptarApuesta(
        Jugador jugador) {
        if (estadoRuleta == Estado.ACEPTANDO_APUESTAS) {
            apostadores.add(jugador);
        }
    }

    public synchronized boolean aceptaApuestas() {
        if (estadoRuleta == Estado.ACEPTANDO_APUESTAS) {
            return true;
        }
        return false;
    }

    public void comunicarNumeroGanador(int numero) {
        /* Al pasar el número a los jugadores, ellos nos
```

```
         * irán restando el saldo que les corresponda por haber ganado */
        int numApostadores = apostadores.size();
        for (Jugador apostador : apostadores) {
            apostador.comunicarNumero(numeroGanador);
        }
        /* Una vez comunicadas todas las apuestas, borramos
         * el vector. La partida va a volver a empezar */
        apostadores.clear();
    }

    public void girarRuleta() throws
    InterruptedException {
        int segundosAzar;
        System.out.println("¡Empieza el juego!");
        while (estadoRuleta != Estado.EN_BANCARROTA) {
            estadoRuleta = Estado.ACEPTANDO_APUESTAS;
            /* Se eligen unos milisegundos al azar para que los jugadores
             * elijan, aunque quizá no todos puedan llegar a apostar
             */
            segundosAzar = 1 + generador.nextInt(3);
            System.out.println("Hagan juego, tienen Vds " +
            segundosAzar + " segundos");
            Thread.sleep(1000 * segundosAzar);
            System.out.println("Ya no va más, señores. ¡Girando!");
            estadoRuleta = Estado.RULETA_GIRANDO;
            Thread.sleep(3000);
            numeroGanador = generador.nextInt(37);
            System.out.println("El número ganador es el :" +
            numeroGanador);
            estadoRuleta = Estado.PAGANDO_APUESTAS;
            this.comunicarNumeroGanador(numeroGanador);
            System.out.println("El saldo de la banca es ahora:"
            + saldo);
        }
    }

    public void simular(int jugadoresPar,
                        int jugadoresMartingala,
                        int jugadoresClasicos) throws
    InterruptedException {
        Thread[] hilosJugadoresPares = new Thread[jugadoresPar];
        for (int i = 0; i < jugadoresPar; i++) {
            JugadorParImpar jugador = new JugadorParImpar(
                1000, this);
            hilosJugadoresPares[i] = new Thread(jugador);
            hilosJugadoresPares[i].setName("Apostador par/impar "
            + i);
            hilosJugadoresPares[i].start();
        }
        Thread[] hilosJugadoresMartingala = new Thread[jugadoresMartingala];
        for (int i = 0; i < jugadoresMartingala; i++) {
            JugadorMartingala jugador = new JugadorMartingala(
                1000, this);
            hilosJugadoresMartingala[i] = new Thread(jugador);
            hilosJugadoresMartingala[i].setName("Apostador martingala "
            + i);
            hilosJugadoresMartingala[i].start();
        }
```

```
        Thread[] hilosJugadoresClasico = new Thread[jugadoresClasicos];
        for (int i = 0; i < jugadoresClasicos; i++) {
            JugadorClasico jugador = new JugadorClasico(1000,
            this);
            hilosJugadoresClasico[i] = new Thread(jugador);
            hilosJugadoresClasico[i].setName("Apostador clasico "
            + i);
            hilosJugadoresClasico[i].start();
        }
        this.girarRuleta();
    }

    public static void main(String[] args) throws
    InterruptedException {
        Banca b = new Banca(50000);
        b.simular(5, 5, 5);
    }
}
```

6.2.2 La clase Jugador

```
public abstract class Jugador implements
        Runnable {

    protected long saldo;

    protected boolean enBancarrota;

    protected long cantidadApostada;

    protected boolean apuestaRealizada;

    protected Banca banca;

    protected String nombreHilo;

    protected Random generador;

    public Jugador(long saldoInicial, Banca b) {
        saldo = saldoInicial;
        banca = b;
        apuestaRealizada = false;
        generador = new Random();
    }

    public void sumarSaldo(long cantidad) {
        saldo = saldo + cantidad;
    }

    public void restarSaldo(long cantidad) {
        if (saldo - cantidad <= 0) {
            saldo = 0;
            enBancarrota = true;
            return;
        }
        saldo = saldo - cantidad;
```

```
    }

    public boolean enBancarrota() {
        return enBancarrota;
    }

    /* Lo usa la banca para comunicarnos el número*/
    public abstract void comunicarNumero(int numero);

    public abstract void hacerApuesta();

    /* Todos los jugadores hacen lo mismo:
     * Mientras no estemos en bancarrota ni la
     * banca tampoco, hacemos apuestas. La banca
     * nos dirá el número que haya salido y en
     * ese momento (y si procede) incrementaremos
     * nuestro saldo
     */
    @Override
    public void run() {
        nombreHilo = Thread.currentThread().getName();
        while ((!enBancarrota)
                && (!banca.enBancarrota())) {
            int msAzar;
            /* Mientras la ruleta no acepte apuestas, dormimos un
             * periodo al azar */
            while (!banca.aceptaApuestas()) {
                msAzar = this.generador.nextInt(500);
                try {
                    //System.out.println(nombreHilo+":banca ocupada, durmiendo...");
                    Thread.sleep(msAzar);
                } catch (InterruptedException e) {
                    return;
                }
            }
            hacerApuesta();
        }
        String nombre = Thread.currentThread().getName();
        if (enBancarrota) {
            System.out.println(nombre +
                               ": ¡¡Me arruiné!!");
            return;
        }
        if (banca.enBancarrota()) {
            System.out.println(nombre +
                               " hizo saltar la banca");
        }
    }
}
```

6.2.3 La clase JugadorClasico

```
public class JugadorClasico extends Jugador {

    int numeroElegido;
```

```
    public JugadorClasico(long saldoInicial,
                          Banca b) {
        super(saldoInicial, b);
    }

    @Override
    public void comunicarNumero(int numero) {
        /* No hace falta comprobar si el numero es
         * 0, ya que este jugador no lo elige nunca */
        if (numero == numeroElegido) {
            System.out.println(nombreHilo +
                               ": ¡Gana 36 veces lo jugado, 360 euros");
            banca.restarSaldo(360);
            sumarSaldo(360);
        }
        System.out.println(nombreHilo +
                           " queda con un saldo de " + saldo);
        apuestaRealizada = false;
    }

    @Override
    public void hacerApuesta() {
        if (!banca.aceptaApuestas())
            return;
        if (apuestaRealizada)
            return;
        /* Elegimos del 1 al 36 (no se puede elegir el 0*/
        numeroElegido = 1 + generador.nextInt(36);
        banca.sumarSaldo(10);
        restarSaldo(10);
        apuestaRealizada = true;
        banca.aceptarApuesta(this);
    }
}
```

6.2.4 La clase JugadorParImpar

```
public class JugadorParImpar extends Jugador {

    public JugadorParImpar(long saldoInicial,
                           Banca b) {
        super(saldoInicial, b);
    }

    protected boolean jugamosAPares;

    @Override
    public void hacerApuesta() {
        if (!banca.aceptaApuestas())
            return;
        if (apuestaRealizada)
            return;
        /* Elegimos una apuesta...*/
        if (generador.nextBoolean() == true) {
            //System.out.println(nombreHilo+" elige apostar a par");
            jugamosAPares = true;
```

```
        } else {
            //System.out.println(nombreHilo+" elige apostar a impar");
            jugamosAPares = false;
        }
        banca.sumarSaldo(10);
        restarSaldo(10);
        apuestaRealizada = true;
        /* Y pedimos a la banca que nos la acepte*/
        banca.aceptarApuesta(this);
    }

    public boolean esGanador(int num) {
        /* Este jugador necesita comprobar si ha salido el 0,
         * aunque no lo elige nunca, ya que si hemos apostado
         * Par podríamos pensar que hemos ganado cuando no es así */
        if (num == 0) {
            return false;
        } else {
            if ((num % 2 == 0) && (jugamosAPares)) {
                return true;
            }
            if ((num % 2 != 0) && (!jugamosAPares)) {
                return true;
            }
        }
        //Fin del else externo
        return false;
        //Fin de esGanador
    }

    @Override
    public void comunicarNumero(int numero) {
        if (esGanador(numero)) {
            /*Ganamos y cogemos a la banca 20 euros*/
            System.out.println(nombreHilo +
                               " ganó 20 euros por acertar impar");
            banca.restarSaldo(20);
            this.sumarSaldo(20);
        }
        System.out.println(nombreHilo +
                           " se queda con un saldo de " + saldo);
        /* Sea como sea, al terminar indicamos que ya no tenemos
         * una apuesta realizada. Es decir, permitirmos al
         * jugador volver a apostar         */
        apuestaRealizada = false;
    }
}
```

6.2.5 La clase JugadorMartingala

```
public class JugadorMartingala extends Jugador {

    private int cantidadAApostar;

    private int numeroElegido;
```

```
    public JugadorMartingala(long saldoInicial,
                             Banca b) {
        super(saldoInicial, b);
        cantidadAApostar = 1;
    }

    @Override
    public void comunicarNumero(int numero) {
        if (numero == 0) {
            System.out.println(nombreHilo + " pierde " +
                               cantidadAApostar);
            cantidadAApostar = cantidadAApostar * 2;
            return;
        }
        if (numeroElegido == numero) {
            int beneficios = cantidadAApostar * 36;
            banca.restarSaldo(beneficios);
            sumarSaldo(beneficios);
            cantidadAApostar = 1;
        }
        if (numeroElegido != numero) {
            //System.out.println(nombreHilo + " pierde "+cantidadAApostar);
            cantidadAApostar = cantidadAApostar * 2;
        }
        System.out.println(nombreHilo +
                           " queda con un saldo de " + saldo);
        apuestaRealizada = false;
    }

    @Override
    public void hacerApuesta() {
        if (!banca.aceptaApuestas())
            return;
        if (apuestaRealizada)
            return;
        /* Elegimos del 1 al 36 (no se puede elegir el 0*/
        numeroElegido = 1 + generador.nextInt(36);
        banca.sumarSaldo(cantidadAApostar);
        restarSaldo(cantidadAApostar);
        apuestaRealizada = true;
        banca.aceptarApuesta(this);
    }
}
```