TEMA 9. ANÁLISIS Y ESTUDIO DE LOS FLUJOS DE ENTRADA-SALIDA. FICHEROS

INDICE

FLUJOS DE DATOS (STREAMS). FICHEROS

Hasta ahora cada vez que ejecutamos una aplicación debemos introducir los datos necesarios para su correcta ejecución ya que al finalizar la ejecución se borran de la memoria.

Cuando estamos probando una aplicación y necesitamos introducir datos muchas veces este proceso puede ser muy tedioso.

Además, en algunas ocasiones necesitamos guardar datos para una siguiente ejecución. Un ejemplo de esto son las cookies de los navegadores web.

En estos casos podemos almacenar esa información en un fichero y cargarla cuando nos haga falta.

Antes de trabajar con ficheros es muy importante saber cómo vamos a recuperar los datos (lectura) ya que el modo en que hagamos la recuperación condiciona la escritura de los mismos.

En Java un fichero es un tipo de flujo de datos (stream). Para facilitar el trabajo con flujos de datos (streams) Java dispone de la biblioteca **java.io**. Si queremos usar las clases y métodos de java.io los debemos importar escribiendo

```
import java.io.*;
```

FICHEROS DE TEXTO

Antes de poder trabajar con un fichero de texto tenemos que abrirlo del modo adecuado.

Por ejemplo, si queremos abrir un fichero de texto de nombre **prueba.txt** para leer su contenido debemos usar un objeto **FileReader** al que asignamos un objeto de tipo **File** asociado con el fichero vamos a usar para la lectura. Además, para facilitar el proceso de lectura debemos crear un buffer de memoria asociado al fichero para lo que creamos un objeto **BufferedReader** y lo asociamos al objeto FileReader. Para realizar todo lo anterior escribimos

```
File archivo = null;
FileReader fr = null;
BufferedReader br = null;
archivo = new File ("prueba.txt");
fr = new FileReader (archivo);
br = new BufferedReader(fr);
```

Una vez abierto el fichero de texto en modo lectura leemos su contenido línea a línea desde el buffer usando el método **readLine** escribiendo

```
String linea;
while((linea=br.readLine())!=null){
 System.out.println(linea);
}
```

Cuando ya no vaya a trabajar más con el fichero de texto lo cierro usando el método **close** escribiendo

```
fr.close();
```

El proceso a realizar si queremos abrir un fichero de texto para escribir en él es muy parecido.

Por ejemplo, si queremos abrir un fichero de texto de nombre **prueba.txt** para escribir algo en él debemos usar un objeto **FileWriter** al que asignamos un objeto de tipo **File** asociado con el fichero vamos a usar para la escritura. Además, para facilitar el proceso de escritura debemos crear un buffer de memoria asociado al fichero para lo que creamos un objeto **PrintWriter** y lo asociamos al objeto FileWriter. Para realizar todo lo anterior escribimos

```
FileWriter fichero = null;
PrintWriter pw = null;
fichero = new FileWriter("prueba.txt");
pw = new PrintWriter(fichero);
```

Una vez abierto el fichero de texto en modo escritura podemos escribir en él usando cualquiera de los métodos que hemos usado para escribir algo por pantalla (print, println, printf). Por ejemplo, para escribir una línea usando el método **println** (al igual que hacemos en System.out.println) escribimos

```
pw.println("Hola Mundo.");
```

Cuando ya no vaya a trabajar más con el fichero de texto lo cierro al igual que en el modo lectura usando el método **close** escribiendo

```
fichero.close();
```

FICHEROS BINARIOS

El modo de trabajar con ficheros binarios en Java es similar al modo de trabajar con ficheros de texto. La única diferencia está en los objetos usados a la hora de trabajar con ficheros binarios.

Por ejemplo, si queremos abrir un fichero binario de nombre **prueba.bin** para leer su contenido debemos usar un objeto **FileInputStream** al que pasamos como parámetro un String con la **ruta del fichero** que queremos abrir. Además, para facilitar el proceso de lectura debemos crear un buffer de memoria asociado al fichero para lo que creamos un objeto **BufferedInputStream** y lo asociamos al objeto FileInputStream. Para realizar todo lo anterior escribimos

```
FileInputStream fis = new FileInputStream("prueba.bin");
BufferedInputStream bis = new BufferedInputStream(fis);
```

Una vez abierto el fichero binario en modo lectura leemos su contenido línea a línea usando el método **read** que devuelve el número de bytes que se han leído (devuelve **0 si no ha podido leer nada** y **-1** si ha llegado al **final del fichero**). Para mejorar el proceso de lectura creamos un **array** de tipo **byte** y de nombre **datos** con el tamaño de los bytes que queramos leer en cada lectura. Por ejemplo, si queremos leer de 8 en 8 bytes escribimos

```
byte [] datos = new byte[8];
bis.read(datos);
```

Cuando ya no vaya a trabajar más con el fichero lo cierro usando el método **close** escribiendo

```
bis.close();
```

El proceso a realizar si queremos abrir un fichero binario para escribir en él es muy parecido.

Por ejemplo, si queremos abrir un fichero binario de nombre **prueba.bin** para escribir en él debemos usar un objeto **FileOutputStream** al que pasamos como parámetro un String con la **ruta del fichero** que queremos abrir. Además, para facilitar el proceso de lectura debemos crear un buffer de memoria asociado al fichero para lo que creamos un objeto **BufferedOutputStream** y lo asociamos al objeto FileOutputStream. Para realizar todo lo anterior escribimos

```
FileOutputStream fos = new FileOutputStream("prueba.bin");
BufferedOutputStream bos = new BufferedOutputStream(fos);
```

Una vez abierto el fichero de texto en modo escritura escribimos en él contenido línea a línea usando el método **write** que recibe los **datos** a escribir, el **desplazamiento en bytes** desde la posición actual (0 para escribir a continuación), y el **número de bytes** a escribir. Para mejorar el proceso de escritura creamos un array de tipo **byte** y de nombre **datos** con el tamaño de los bytes que queramos escribir en cada escritura. Por ejemplo, si queremos escribir de 8 en 8 bytes escribimos

```
byte [] datos = new byte[8];
bos.write(datos,0,8);
```

Cuando ya no vaya a trabajar más con el fichero de texto lo cierro usando el método **close** escribiendo

```
bos.close();
```

BUFFER EN ENTRADA Y SALIDA DE DATOS

Los objetos de las clases FileInputStream, FileOuputStream, FileReader o FileWriter, realizan las operaciones físicamente en el disco duro. Esto disminuye notablemente el rendimiento del sistema ya que las operaciones sobre dispositivos físicos consumen mucho tiempo.

Para mejorar el rendimiento de las operaciones de entrada y salida de datos de usan buffers (memorias intermedias) que actúan como intermediarios entre el dispositivo físico y las aplicaciones.

Cuando una aplicación tiene que leer desde un dispositivo físico hace una petición a su buffer de lectura que se encarga de leer los datos de manera más eficiente (aprovechando los momentos ociosos del procesador). Además de almacenar los datos solicitados, el buffer almacena los datos que se encuentran a continuación por si se van a necesitar. Esto provoca que si el buffer acierta a la hora de cargar los siguientes datos no haga falta volver a cargar los datos desde el dispositivo físico lo que aumenta considerablemente el rendimiento.

Cuando una aplicación tiene que escribir en un dispositivo físico manda los datos a su buffer de escritura que se encarga de escribir los datos de manera más eficiente (aprovechando los momentos ociosos del procesador). El buffer almacena los datos de varias operaciones de escritura antes de escribir los datos lo que disminuye el número de accesos al dispositivo físico. Esto aumenta considerablemente el rendimiento.

En Java las clases BufferedReader, BufferedInputStream, BufferedWriter y BufferedOutputStream permiten crear un buffer intermedio a la hora de leer o escribir en un dispositivo físico.

Para mejorar el rendimiento de las operaciones de entrada y salida usaremos siempre buffers.

ACCESO ALEATORIO EN FICHEROS

Al **abrir un fichero** el sistema guarda la **posición actual** del mismo **en un apuntador**. Esta posición actual es de la que lee los datos o en la que escribe los datos. Cuando se leen unos datos el apuntador pasa a apuntar a la posición siguiente al último dato leído y al escribir unos datos el apuntador pasa a apuntar a la posición siguiente al último dato escrito.

Cuando trabajamos con ficheros normalmente lo hacemos de forma secuencial, primero leemos un dato, luego el siguiente, y así hasta el final del fichero.

En algunas ocasiones podemos necesitar modificar el valor de la posición para saltarnos ese acceso secuencial. A ese tipo de acceso en el que **modificamos la posición del apuntador** según nuestras necesidades se le denomina **acceso aleatorio**.

Para facilitar el acceso aleatorio en ficheros Java proporciona la clase **RandomAccessFile** que contiene los métodos **getFilePointer** que devuelve la posición actual del apuntador, **length** que devuelve el tamaño en bytes del fichero, y **seek** que recibe el valor de la posición a la que queremos mover el apuntador y lo mueve.

Para crear un nuevo objeto de la clase RandomAccessFile necesitamos proporcionar un objeto de tipo **File** o un **String** con la **ruta** del fichero y un **String** indicando el **modo de apertura.** Los modos de apertura son:
"**r**". Lectura. Si el fichero no existe da un error.
"w". Escritura. Si el fichero existe se sobreescribe.
"**rw**". Lectura y escritura. Si el fichero no existe se crea. Si el fichero existe NO se sobreescribe.

Por ejemplo, para crear un nuevo objeto de la clase RandomAccessFile que abra el fichero logo.png en modo lectura para obtener su tamaño en bytes escribimos

```
RandomAccessFile raf = new RandomAccessFile("logo.png", "r");
long fileSize = raf.length();
```

El acceso aleatorio solo modifica la posición del apuntador. El resto de operaciones siguen siendo las mismas que se realizan sobre los ficheros con acceso secuencial.

EJERCICIOS DE ENTRADA / SALIDA DE DATOS

1. Crea la clase FicheroNombres que solicita nombres que va guardando en un fichero de texto de nombre **nombres.txt** hasta que se introduzca una cadena en blanco ("") que no se guarda. Después muestra el contenido del fichero por pantalla. Se deben controlar las posibles excepciones.

2. Crea la clase FicheroCopia que copia el fichero origen de nombre **logo.png** en el fichero destino de nombre **logocopia.png**. Para ello usa un array de tipo byte con 512 posiciones. Una vez copiado, abre el fichero de imagen logocopia.png para comprobar que la copia se ha realizado correctamente. Se deben controlar las posibles excepciones.

3. Crea la clase FicheroAccesoAleatorioSize que muestra la longitud en KB del fichero logo.png.

4. Crea la clase FicheroNombresAppend que abre el fichero de texto de nombre **nombres.txt** y añade el nombre "--------" al final del fichero. Después muestra el contenido del fichero por pantalla. Se deben controlar las posibles excepciones.

FLUJOS PREDETERMINADOS

Java proporciona unos flujos predeterminados para facilitar las operaciones de entrada / salida de datos.

El flujo predeterminado de escritura de datos es **System.out**. La función predeterminada de este flujo es facilitar las operaciones que muestren datos por pantalla. Dentro de la clase System.out se incluyen métodos que permiten escribir datos por pantalla como por ejemplo **System.out.print()**.

El flujo predeterminado de lectura de datos es **System.in**. La función predeterminada de este flujo es facilitar las operaciones de lectura de datos por teclado. Dentro de la clase System.in se incluyen métodos que permiten leer datos por teclado como por ejemplo **System.in.read()**.

Como las operaciones de entrada de datos suelen provocar muchos problemas se ha creado otra clase Java de nombre **Scanner** que permite trabajar con System.in de manera más sencilla.

Al realizar una operación de lectura de un dato de tipo numérico mediante la clase Scanner se coge el valor numérico del buffer de entrada y se asigna el valor a la variable correspondiente dejando en el buffer el carácter de fin de línea '\n'.

Si después de leer un dato de tipo numérico se lee un dato de tipo String o char ese carácter '\n' que se ha quedado en el buffer se asigna a la variable que se quiere leer dando la sensación que se ha saltado la lectura.

Para evitar ese problema si después de leer un dato numérico tenemos que leer un dato de tipo String o char debemos poner antes de la sentencia de lectura real de los datos una operación de lectura que elimine el carácter '\n' del buffer.

Por ejemplo, si queremos leer la variable de tipo String cadena después de leer la variable de tipo int opción debemos escribir

```
// leo opcion
System.out.print("Opción: ");
opcion = teclado.nextInt();

// limpio el buffer de entrada
teclado.nextLine();

// leo cadena
System.out.print("Cadena: ");
cadena = teclado.nextLine();
```

EJERCICIOS FLUJOS PREDETERMINADOS

5. Realiza la clase Java LeerEnteroString que lee un número entero y un String y después muestra sus valores por pantalla.

FICHEROS DE DATOS. CONCEPTO DE REGISTRO

En los lenguajes estructurados (por ejemplo en lenguaje C) los datos que comparten características comunes se guardan en unas estructuras de datos que reciben el nombre de registros. Agrupar los datos con características comunes en registros facilita su manipulación.

En los lenguajes orientados a objetos como Java para trabajar con datos que comparten características comunes se crean clases.

Los ficheros de registros y los ficheros de objetos son dos tipos diferentes de almacenamiento de información. Nosotros nos vamos a centrar en el almacenamiento de objetos.

OPERACIONES CON FICHEROS

Las operaciones que se realizan sobre los ficheros son

- Abrir el fichero en el modo que corresponda. Se puede abrir un fichero para leer, escribir, escribir al final o una combinación de las anteriores.
- Recorrer el fichero. Se pueden leer los elementos de un fichero uno a uno hasta llegar al final del fichero. Este proceso se puede usar para buscar si un valor está en el fichero o no.
- Posicionarse en el fichero (acceso aleatorio). Podemos posicionarnos en la posición del fichero que queramos para realizar una determinada operación.

Siempre que sea posible, es conveniente minimizar el trabajo directo con ficheros. Lo ideal es cargar el contenido del fichero en memoria (por ejemplo en un ArrayList) manipularlo y, si hay que actualizarlo, grabar los datos que están en memoria en el fichero.

EJERCICIOS DE FICHEROS

6. Realiza la clase Java ArrayListCadenasMenuFicheros que modifica ArrayListCadenasMenu para que al comienzo de la ejecución del programa se cargue el contenido del fichero cadenas.txt en el ArrayList y al finalizar la ejecución, si se han modificado los datos, se graban los datos del ArrayList en el fichero cadenas.txt. Las excepciones se tienen que controlar. Si se produce algún error al manipular el fichero se mostrará un mensaje de error indicando el porqué de ese error.

ALMACENAMIENTO DE OBJETOS EN FICHEROS

En los lenguajes estructurados (por ejemplo en lenguaje C) los datos que comparten características comunes se guardan en unas estructuras de datos que reciben el nombre de registros. Agrupar los datos con características comunes en registros facilita su manipulación.

En los lenguajes orientados a objetos como Java para trabajar con datos que comparten características comunes se crean clases.

Los ficheros de registros y los ficheros de objetos son dos tipos diferentes de almacenamiento de información. Nosotros nos vamos a centrar en el almacenamiento de objetos.

A la hora de almacenar los datos de un objeto en un fichero podemos almacenar uno a uno los atributos que queramos guardar en el fichero y después recuperar desde el fichero los datos de los atributos respetando ese orden o podemos guardar el objeto completo como un conjunto de bytes y después recuperar desde el archivo el objeto completo.

El proceso de grabar un objeto completo en un fichero como un conjunto de bytes recibe el nombre de **serialización**.

SERIALIZACIÓN. Serializable

La **serialización** es un mecanismo que permite tratar los objetos como un conjunto de bytes independientemente del contenido que tengan. Esto simplifica mucho operaciones como grabar y recuperar los datos en ficheros o mandar los datos por la red desde un equipo y recuperarlos en otro.

Para poder utilizar la serialización de objetos en una clase, dicha clase debe implementar el interfaz **Serializable**. Por ejemplo, si queremos que la clase Complejo pueda usar la serialización de objetos debemos modificar su declaración de la siguiente manera

```
public class Complejo implements Serializable{…}
```

Al leer o escribir un objeto Serializable en un fichero, las clases ObjectInputStream (mediante el método **readObject**) y ObjectOutputStream (mediante el método **writeObject**) se encargan de convertir los datos de

ese objeto a un array de tipo byte para realizar la lectura o la escritura del objeto Serializable como un grupo de bits.

Por ejemplo, para escribir un objeto de la clase Complejo serializada de nombre c en el fichero **complejos.dat** debemos escribir

```
// grabo los datos en complejos.dat
FileOutputStream fos=new FileOutputStream("complejos.dat");
ObjectOutputStream oos = new ObjectOutputStream (fos);

// lo grabo
oos.writeObject(c);

// cierro el fichero
oos.close();
```

Si queremos leer un objeto de la clase Complejo serializada desde el fichero **complejos.dat** debemos escribir

```
FileInputStream fis=new FileInputStream("complejos.dat");
ObjectInputStream ois = new ObjectInputStream(fis);
c = (Complejo) ois.readObject(); // convierte los bytes leídos en un objeto de la clase Complejo
ois.close();
```

Con lo anterior es suficiente para grabar y leer objetos serializados pero si por algún motivo queremos realizar nosotros la conversión del objeto Serializable a un array de tipo byte debemos escribir

```
ByteArrayOutputStream baos= new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream (baos);
oos.writeObject(miObjetoSerializable);
oos.close();
byte[] bytes =  baos.toByteArray(); // guarda los bytes en el array de tipo byte y de nombre bytes
```

Si queremos realizar el proceso inverso, es decir, la conversión de un array de tipo byte a un objeto Serializable debemos escribir

```
ByteArrayInputStream bais= new ByteArrayInputStream(bytes);
ObjectInputStream ois = new ObjectInputStream(bais);
ClaseSerializable miObjetoSerializable = (ClaseSerializable)ois.readObject(); // convierte los bytes leídos en un objeto de la ClaseSerializable
ois.close();
```

Si la información de los objetos Serializados va a viajar a través de la red de un equipo a otro nos podemos encontrar con que la versión de la clase del objeto serializado sea distinta en el quipo origen y en el equipo destino. Para evitar eso, en los objetos Serializable se incluye una constante de nombre **serialVersionUID** que contiene la versión de la clase de tal modo que se pueda saber si la versión de la clase origen y de la clase destino son distintas. Naturalmente versiones distintas de la clase deberán tener valores distintos.

Para definir la constante serialVersionUID escribimos

```
private static final long serialVersionUID = -1664923408223878238L;
```

Algunos entornos de desarrollo, como **eclipse**, muestran un warning si una clase que implementa Serializable (o hereda de una clase que a su vez implementa Serializable) no tiene definido este campo. Además, Eclipse permite generar automáticamente ese número al definir una clase como Serializable.

Cuando leemos desde un fichero debemos de controlar si hemos llegado al final del fichero. Para controlar si hemos llegado al final de un objeto de tipo **FileInputStream** podemos usar el método **available**.

Por ejemplo para controlar si hemos llegado al final del fichero alcomplejos.dat debemos escribir

```
FileInputStream fis=new FileInputStream("alcomplejos.dat");
ObjectInputStream ois = new ObjectInputStream(fis);

// mientras que no sea el final del fichero leo los datos
while(fis.available() > 0){
 c = (Complejo)ois.readObject();
 …
}
```

EJERCICIOS DE ALMACENAMIENTO DE OBJETOS EN FICHEROS

7. Modifica la clase Complejo para que permita la serialización de objetos.

8. Crea la clase Java ComplejoMainSerializable que crea un objeto de la clase Complejo y lo guarda en el fichero complejos.dat. Después lee los datos del complejo desde complejos.dat y muestra el valor del complejo leído por pantalla.

9. Crea la clase Java ComplejoMainSerializableArrayList que crea cinco objetos de la clase Complejo y los almacena en un ArrayList. Después recorre el ArrayList guardando los datos de los objetos de tipo Complejo en el fichero alcomplejos.dat. Al finalizar lee los datos desde alcomplejos.dat y los almacena en un nuevo ArrayList. Para terminar recorre el ArrayList mostrando el valor de los objetos de tipo Complejo almacenados en él.

10. Modifica la clase Java ContadorGenerico para que permita la serialización de objetos.

11. Crea la clase Java ArrayListContadorIntegerMenuSerializable que modifica la clase ArrayListContadorIntegerMenu para que al inicio de la ejecución se carguen los datos desde el fichero contadorinteger.dat y para que al final de la ejecución, si los datos han sido modificados, los grabe de nuevo en contadorinteger.dat.

12. Crea la clase Java ArrayListContadorStringMenuSerializable que modifica la clase ArrayListContadorStringMenu para que al inicio de la ejecución se carguen los datos desde el fichero contadorstring.dat y para que al final de la ejecución, si los datos han sido modificados, los grabe de nuevo en contadorstring.dat.

SOCKETS. COMUNICACIÓN ENTRE DISTINTOS PROCESOS

Los **sockets** permiten intercomunicar procesos. En realidad un socket es un flujo que permite comunicar información entre dos procesos que, normalmente, están en sistemas informáticos distintos.

A efectos prácticos podemos considerar el trabajo con sockets idéntico al trabajo con ficheros con la única diferencia que en lugar de leer desde un fichero o escribir en un fichero leeremos desde un extremo del socket y escribiremos en otro extremo del socket.

Normalmente se utilizan sockets en aplicaciones **cliente** / **servidor**.

Normalmente, un **servidor** se ejecuta en una máquina específica y tiene un socket asociado a un **número de puerto** específico. El servidor simplemente espera a la escucha en el socket a que un cliente se conecte con una petición. El cliente conoce el nombre (o la dirección IP) de la máquina sobre la que está ejecutándose el servidor y el número de puerto al que está conectado. Solicitar una conexión consiste en intentar establecer una cita con el servidor en el puerto de la máquina servidora.

Si todo va bien, el servidor acepta la conexión. Pero antes, **el servidor crea un nuevo socket en un puerto diferente**. Es necesario crear un nuevo socket (y consecuentemente un número de puerto diferente) de forma que en el socket original se continúe a la escucha de las peticiones de nuevos clientes mientras se atiende a las necesidades del cliente conectado. En el cliente, si se acepta la conexión, el socket se crea satisfactoriamente y se puede utilizar para comunicarse con el servidor.

Debemos tener en cuenta que un socket es el extremo final de un enlace punto-a-punto que comunica dos procesos que se están ejecutando, normalmente, en distintos equipos.

Los **sockets** siempre están **asociados a un número de puerto** que es utilizado por TCP para identificar la aplicación a la que está destinada la solicitud y poder redirigírsela.

LA CLASE SOCKET

La clase **Socket** del paquete **java.net** es fácil de usar comparada con la que proporcionan otros lenguajes. Java oculta las complejidades derivadas del establecimiento de la conexión de red y del envío de datos a través de ella. En esencia, el paquete java.net proporciona la misma interfaz de programación que se utiliza cuando se trabaja con archivos.

Para crear un socket hay que indicar la dirección en la que se encuentra el equipo con el que nos queremos comunicar y el puerto en el que nos vamos a conectar. La dirección debe de ser un objeto de la clase InetAddress. Podemos crear una dirección del tipo InetAddress a partir de un String usando el método InetAddress.**getByName**(ipServer);

Por ejemplo, para crear un socket que se conecte al equipo que se encuentra en la **dirección** IP **127.0.0.1** en el **puerto 8888** escribimos:

```
int puerto = 8888;
String ipServidor="127.0.0.1";
InetAddress inetAddress = InetAddress.getByName(ipServidor);
Socket socket = null;
socket = new Socket(inetAddress, puerto);
```

Las operaciones con sockets provocan un montón de excepciones que debemos capturar si queremos que la aplicación funcione correctamente.

LA CLASE SERVERSOCKET

La clase **ServerSocket** es la que se utiliza a la hora de crear sockets en los procesos que actúan como servidores, al igual que la clase Socket se utiliza para crear sockets en los procesos que actúan como clientes.

Para crear un ServerSocket en el servidor basta con indicar el puerto en el que vamos a permitir las conexiones.

Cuando llega una petición a ese puerto se crea un nuevo socket para aceptar la conexión y permitir la comunicación entre el cliente y el servidor. Para ello se usa el método **accept** de ServerSocket.

Por ejemplo, para crear un ServerSocket que permita conexiones en el **puerto 8888** escribimos:

```
int puerto = 8888;
ServerSocket serverSocket;
Socket socket = null;
serverSocket = new ServerSocket(puerto);

socket = serverSocket.accept();
```

THREADS. MÚLTIPLES HILOS DE EJECUCIÓN

Un hilo o thread es un flujo de control dentro de un programa. Creando varios hilos podremos realizar varias tareas simultáneamente. Cada hilo tendrá sólo un contexto de ejecución (contador de programa, pila de ejecución). Es decir, a diferencia de los procesos UNIX, no tienen su propio espacio de memoria sino que acceden todos al mismo espacio de memoria común, por lo que será importante su **sincronización** cuando tengamos varios hilos accediendo a los mismos objetos.

En Java los hilos están encapsulados en la clase Thread. Para crear un hilo tenemos dos posibilidades:

- **Heredar de Thread** redefiniendo el método run().
- Crear una clase que **implemente** la interfaz **Runnable** que nos obliga a definir el método run().

En ambos casos debemos definir un método **run()** que será el que contenga el código del hilo. Desde dentro de este método podremos llamar a cualquier otro método de cualquier objeto, pero este método run() será el método que se invoque cuando iniciemos la ejecución de un hilo. El hilo terminará su ejecución cuando termine de ejecutarse este método run().

Para crear nuestro hilo mediante herencia haremos lo siguiente:

```
public class EjemploHilo extends Thread
{
   public void run() {
      // Código del hilo
   }
}
```

Una vez definida la clase de nuestro hilo deberemos instanciarlo y ejecutarlo de la siguiente forma

```
Thread t = new EjemploHilo();
t.start();
```

Al llamar al método start del hilo, comenzará ejecutarse su método run. Crear un hilo heredando de Thread tiene el problema de que al no haber herencia múltiple en Java, si heredamos de Thread no podremos heredar de ninguna otra clase, y por lo tanto un hilo no podría heredar de ninguna otra clase.

Este problema desaparece si utilizamos la interfaz Runnable para crear el hilo, ya que una clase puede implementar varios interfaces. Definiremos la clase que contenga el hilo como se muestra a continuación

```
public class EjemploHilo implements Runnable
```

```
{
   public void run()  {
      // Código del hilo
   }
}
```

Para instanciar y ejecutar un hilo de este tipo deberemos hacer lo siguiente

```
Thread t = new Thread(new EjemploHilo());
t.start();
```

Esto es así debido a que en este caso EjemploHilo no deriva de una clase Thread, por lo que no se puede considerar un hilo, lo único que estamos haciendo implementando la interfaz es asegurar que vamos a tener definido el método run(). Con esto lo que haremos será proporcionar esta clase al constructor de la clase Thread, para que el objeto Thread que creemos llame al método run() de la clase que hemos definido al iniciarse la ejecución del hilo, ya que implementando la interfaz le aseguramos que esta función existe.

Un hilo pasará por varios estados durante su ciclo de vida.

Una vez se ha instanciado el objeto del hilo, diremos que está en estado de **Nuevo hilo**.

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

Cuando invoquemos su método start() el hilo pasará a ser un **hilo vivo**, comenzándose a ejecutar su método run(). Una vez haya salido de este método pasará a ser un **hilo muerto**.

```
t.start();
```

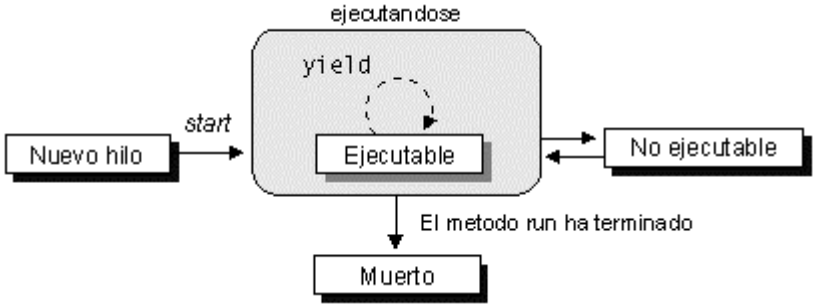
La única forma de parar un hilo es hacer que salga del método run() de forma natural. Podremos conseguir esto haciendo que se cumpla una condición de salida de run() (lógicamente, la condición que se nos ocurra dependerá del tipo de programa que estemos haciendo). Las funciones para parar, pausar y reanudar hilos están desaprobadas en las versiones actuales de Java.

Mientras el **hilo** esté **vivo**, podrá encontrarse en dos estados: **Ejecutable** y **No ejecutable**. El hilo pasará de Ejecutable a No ejecutable en los siguientes casos:

Cuando se encuentre **dormido** por haberse llamado al método **sleep**(), permanecerá No ejecutable hasta haber transcurrido el número de milisegundos especificados.

Cuando se encuentre **bloqueado** en una llamada al método **wait**() esperando que otro hilo lo desbloquee llamando a **notify**() o **notifyAll**(). Veremos cómo utilizar estos métodos más adelante.

Cuando se encuentre **bloqueado en** una petición de **E/S**, hasta que se complete la operación de E/S.



Podemos saber si un hilo se encuentra vivo o no, llamando a su método **isAlive**().

Una propiedad importante de los hilos será su **prioridad**. Mientras el hilo se encuentre vivo, el scheduler de la máquina virtual Java le asignará o lo sacará de la CPU, coordinando así el uso de la CPU por parte de todos los hilos activos basándose en su prioridad. Se puede **forzar la salida** de un hilo **de la CPU** llamando a su método **yield**(). También se sacará un hilo de la CPU cuando un hilo de mayor prioridad se haga Ejecutable, o cuando el tiempo que se le haya asignado expire.

Para **cambiar la prioridad** de un hilo se utiliza el método **setPriority**(), al que deberemos proporcionar un valor de prioridad entre MIN_PRIORITY y MAX_PRIORITY (tenéis constantes de prioridad disponibles dentro de la clase Thread, consultad el API de Java para ver qué valores de constantes hay).

Los objetos de clase Thread cuentan con un método **interrupt**() que permite al **hilo** ser **interrumpido**. En realidad la interrupción simplemente cambia un flag del hilo para marcar que ha de ser interrumpido, pero cada hilo debe estar programado para soportar su propia interrupción.

Si el hilo invoca un método que lance la excepción InterruptedException, tal como el método sleep(), entonces en este punto del código terminaría la ejecución del método run() del hilo. Podemos manejar esta circunstancia con:

```
for (int i = 0; i < maxI; i++) {
   // Pausar 4 segundos
   try {
      Thread.sleep(4000);
   } catch (InterruptedException e) {
      // El hilo ha sido interrumpido. Vamos a salir de run()
      return;
   }
   System.out.println(algo[i]);
}
```

De esta manera si queda algo que terminar se puede terminar, a pesar de que la ejecución del sleep() ha sido interrumpida.

Si nuestro hilo no llama a métodos que lancen InterruptedException, entonces debemos ocuparnos de comprobarla periódicamente

```
for (int i = 0; i < maxI; i++) {
   trabajoCon(i);
   if (Thread.interrupted()) {
      // El flag de interrupción ha sido activado
      return;
   }
```

```
}
```

THREADS. MÚLTIPLES HILOS DE EJECUCIÓN. SINCRONIZACIÓN

Muchas veces los hilos deberán trabajar de forma coordinada, por lo que es necesario un mecanismo de sincronización entre ellos.

Un primer mecanismo de sincronización es la variable cerrojo incluida en todo objeto Object, que permitirá evitar que más de un hilo entre en la sección crítica para un objeto determinado. Los **métodos** declarados como **synchronized** utilizan el cerrojo del objeto al que pertenecen evitando que más de un hilo entre en ellos al mismo tiempo.

```
public synchronized void metodo_seccion_critica()
{
   // Código sección crítica
}
```

Todos los métodos synchronized de un mismo objeto (no clase, sino objeto de esa clase), comparten el mismo cerrojo, y es distinto al cerrojo de otros objetos (de la misma clase, o de otras).

También podemos utilizar cualquier otro objeto para la sincronización dentro de nuestro método de la siguiente forma

```
synchronized(objeto_con_cerrojo)
{
  // Código sección crítica
}
```

De esta forma sincronizaríamos el código que escribiésemos dentro, con el código synchronized del objeto objeto_con_cerrojo.

Además podemos hacer que un **hilo** quede **bloqueado** a la espera de que otro hilo lo desbloquee cuando suceda un determinado evento. Para bloquear un hilo usaremos la función **wait**(), para lo cual el **hilo** que llama a esta función debe estar en **posesión del monitor**, cosa que ocurre dentro de un método synchronized, por lo que sólo podremos bloquear a un proceso dentro de estos métodos.

Para **desbloquear** a los **hilos** que haya bloqueados se utilizará **notifyAll**(), o bien **notify**() para desbloquear sólo uno de ellos aleatoriamente. Para invocar estos métodos ocurrirá lo mismo, el hilo deberá estar en posesión del monitor.

Cuando un hilo queda bloqueado liberará el cerrojo para que otro hilo pueda entrar en la sección crítica del objeto y desbloquearlo.

Por último, puede ser necesario **esperar a que un** determinado **hilo haya finalizado** su tarea para continuar. Esto lo podremos hacer llamando al método **join**() de dicho hilo, que nos bloqueará hasta que el hilo haya finalizado.

Se dice que en Java la **sincronización** es **reentrante** porque una sección crítica sincronizada puede contener dentro otra sección sincronizada sobre el mismo cerrojo y eso no causa un bloqueo. Por ejemplo el siguiente código funciona sin bloquearse

```
class Reentrante {
```

```
    public synchronized void a() {
        b();
        System.out.println(" estoy en a() ");
    }
    public synchronized void b() {
        System.out.println(" estoy en b() ");
    }
}
```

La salida sería

```
estoy en b()
n estoy en a()
```

THREADS. MÚLTIPLES HILOS DE EJECUCIÓN. SINCRONIZACIÓN. BLOQUES VIGILADOS

Mediante el uso de **guarded blocks** o **bloques de código vigilados** por determinada condición, hasta que la condición no se cumple, no se pasa a ejecutar dicho bloque de código.

En este caso lo importante es dejar libre el procesador durante el tiempo de espera. Así, el siguiente código sería ineficiente puesto que estaría ocupando la CPU durante la espera

```
public void bloqueVigilado() {
    // No hacerlo así!
    while(!condicion) {} // Se detiene aquí,
    //comprobando iterativamente la condición

    System.out.println("La condición se ha cumplido");
}
```

La **forma correcta** es invocar el método **wait**(). Así el hilo se bloqueará hasta que otro hilo le haga una llamada a notify(). Sin embargo hay que volver a hacer la comprobación de la condición, porque la notificación no significará necesariamente que se haya cumplido la condición. Por eso es necesario tener la llamada a wait() dentro de un bucle **while** que **comprueba la condición**. Si durante la espera se recibe la excepción InterrputedException, aún así lo que importa es comprobar, una vez más, la condición.

```
public synchronized bloqueVigilado() {
    while(!condicion) {
        try {
            wait(); // desocupa la CPU
        } catch (InterruptedException e) {}
    }
    System.out.println("La condición se ha cumplido");
}
```

El bloque es synchronized porque las llamadas a wait() y a notify() siempre deben hacerse desde un bloque de código sincronizado. Operan sobre la variable cerrojo del objeto (desde distintos hilos) y por tanto debe hacerse de forma sincronizada.

SINCRONIZACIÓN. EJEMPLO. PRODUCTOR/CONSUMIDOR

El ejemplo del Productor/Consumidor es un ejemplo clásico de sincronización de hilos. Es el ejemplo que aparece en la documentación de Java para explicar la sincronización de procesos. El enlace al ejemplo original es http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

En este problema hay dos hilos que producen y consumen simultáneamente datos de un mismo buffer o misma variable. El problema es que no se tienen que interbloquear, al tiempo que si el buffer está vacío, el consumidor se tiene que quedar en espera, y si el buffer está lleno, el productor se tiene que quedar en espera.

En este ejemplo el objeto que produce y consume es de clase Drop, declarada a continuación

```
public class Drop {
  // Message sent from producer
  // to consumer.
  private String message;
  // True if consumer should wait
  // for producer to send message,
  // false if producer should wait for
  // consumer to retrieve message.
  private boolean empty = true;

  public synchronized String take() {
    // Wait until message is
    // available.
    while (empty) {
      try {
        wait();
      } catch (InterruptedException e) {}
    }
    // Toggle status.
    empty = true;
    // Notify producer that
    // status has changed.
    notifyAll();
    return message;
  }

  public synchronized void put(String message) {
    // Wait until message has
    // been retrieved.
    while (!empty) {
      try {
        wait();
      } catch (InterruptedException e) {}
    }
    // Toggle status.
    empty = false;
    // Store message.
    this.message = message;
    // Notify consumer that status
    // has changed.
    notifyAll();
  }
```

```
}
```

El productor pone un String aleatorio cada cierto tiempo

```
import java.util.Random;

public class Producer implements Runnable {
   private Drop drop;

   public Producer(Drop drop) {
      this.drop = drop;
   }

   public void run() {
      String importantInfo[] = {
         "Mares eat oats",
         "Does eat oats",
         "Little lambs eat ivy",
         "A kid will eat ivy too"
      };
      Random random = new Random();

      for (int i = 0; i < importantInfo.length; i++) {
         drop.put(importantInfo[i]);
         try {
            Thread.sleep(random.nextInt(5000));
         } catch (InterruptedException e) {}
      }
      drop.put("DONE");
   }
}
```

El consumidor lee datos de tipo String hasta que recibe el String "DONE" y los muestra por pantalla.

```
import java.util.Random;

public class Consumer implements Runnable {
   private Drop drop;

   public Consumer(Drop drop) {
      this.drop = drop;
   }

   public void run() {
      Random random = new Random();
      for (String message = drop.take(); ! message.equals("DONE"); message = drop.take()) {
         System.out.format("MESSAGE RECEIVED: %s%n", message);
         try {
            Thread.sleep(random.nextInt(5000));
         } catch (InterruptedException e) {}
      }
   }
```

```
}
```

Para probar el funcionamiento tenemos que crear una clase de nombre ProducerConsumerExample cuyo contenido es

```
public class ProducerConsumerExample {
   public static void main(String[] args) {
      Drop drop = new Drop();
      (new Thread(new Producer(drop))).start();
      (new Thread(new Consumer(drop))).start();
   }
}
```

SINCRONIZACIÓN. TIPOS DE INTERBLOQUEOS

Los mecanismos de sincronización deben utilizarse de manera conveniente para evitar interbloqueos. Los interbloqueos se clasifican en:

- **Deadlock.** Un hilo A está a la espera de que un hilo B lo desbloquee, al tiempo que este hilo B está también bloqueado a la espera de que A lo desbloquee.
- **Livelock**. Similar al Deadlock, pero en esta situación A responde a una acción de B, y a causa de esta respuesta B responde con una acción a A, y así sucesivamente. Se ocupa toda la CPU produciendo un bloqueo de acciones continuas.
- **Stravation (Inanición)**. Un hilo necesita consumir recursos, o bien ocupar CPU, pero se lo impide la existencia de otros hilos "hambrientos" que operan más de la cuenta sobre dichos recursos. Se impide el funcionamiento fluido del hilo o incluso da la impresión de bloqueo.

Detectar los bloqueos es difícil, tanto a priori como a posteriori, y deben ser estudiados cuidadosamente a la hora de diseñar la sincronización entre hilos.

SINCRONIZACIÓN. MECANISMOS DE ALTO NIVEL. INTERFAZ LOCK

El "**lock**" o **cerrojo** reentrante de Java es fácil de usar pero tiene muchas limitaciones. Por eso el paquete **java.util.concurrent.locks** incluye una serie de utilidades relacionadas con lock. La interfaz más básica de éstas es Lock.

Los objetos cuyas clases implementan la interfaz Lock funcionan de manera muy similar a los locks implícitos que se utilizan en código sincronizado, de manera que sólo un hilo puede poseer el Lock al mismo tiempo.

La ventaja de los objetos Lock es que posibilitan rechazar un intento de adquirir el cerrojo. El método **tryLock**() rechaza darnos el lock si éste no está disponible inmediatamente, o bien tras un tiempo máximo de espera, si se especifica así. El método **lockInterruptibly** rechaza darnos el lock si otro hilo envía una interrupción antes de que el lock haya sido adquirido.

Un ejemplo de sincronización utilizando Lock en lugar de synchronized sería

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

//...
Lock l = new ReentrantLock();;
l.lock();
```

```
try {
   // acceder al recurso protegido por l
} finally {
   l.unlock();
}
```

Los objetos Lock también dan soporte a un mecanismo de wait/notify a través de objetos Condition.

```
class BufferLimitado {
 final Lock lock = new ReentrantLock();
 //Dos condiciones para notificar sólo a los hilos
 //que deban hacer put o take, respectivamente
 final Condition notFull  = lock.newCondition();
 final Condition notEmpty = lock.newCondition();

 final Object[] items = new Object[100];
 int putptr, takeptr, count;

 public void put(Object x) throws InterruptedException {
  lock.lock();
  try {
   while (count == items.length)
    notFull.await();
   items[putptr] = x;
   if (++putptr == items.length) putptr = 0;
   ++count;
   notEmpty.signal();
  } finally {
   lock.unlock();
  }
 }

 public Object take() throws InterruptedException {
  lock.lock();
  try {
   while (count == 0)
    notEmpty.await();
   Object x = items[takeptr];
   if (++takeptr == items.length) takeptr = 0;
   --count;
   notFull.signal();
   return x;
  } finally {
   lock.unlock();
  }
 }
}
```

SINCRONIZACIÓN. MECANISMOS DE ALTO NIVEL. COLECCIONES CONCURRENTES

Java proporciona algunas estructuras con métodos sincronizados, como por ejemplo Vector. Más allá de la simple sincronización, Java también proporciona una serie de clases de colecciones que facilitan la

concurrencia, y se encuentran en el paquete **java.util.concurrent**. Se pueden clasificar según las interfaces que implementan:

BlockingQueue define una estructura de datos FIFO que bloquea o establece un tiempo máximo de espera cuando se intenta añadir elementos a una cola llena o cuando se intenta obtener de una cola vacía.

ConcurrentMap es una subinterfaz de java.util.Map que define operaciones atómicas útiles: por ejemplo eliminar una clave-valor sólo si la clave está presente, o añadir una clave valor sólo si la clave no está presente. Al ser operaciones atómicas, no es necesario añadir otros mecanismos de sincronización. La implementación concreta es la clase ConcurrentHashMap.

La interfaz ConcurrentNavigableMap es para coincidencias aproximadas, con implementación concreta en la clase ConcurrentSkipListMap, que es el análogo concurrente de TreeMap.

SINCRONIZACIÓN. MECANISMOS DE ALTO NIVEL. VARIABLES ATÓMICAS

El paquete **java.util.concurrent.atomic** define clases que soportan operaciones atómicas sobre variables. Las operaciones atómicas son operaciones que no deben ser realizadas por dos hilos simultáneamente. Así, en el siguiente ejemplo de un objeto Contador, en lugar de tener que asegurar la consistencia manualmente

```
class ContadorSincronizado {
   private int c = 0;

   public synchronized void increment() {
      c++;
   }

   public synchronized void decrement() {
      c--;
   }

   public synchronized int value() {
      return c;
   }

}
```

Podríamos programarlo utilizado un entero atómico, **AtomicInteger**

```
import java.util.concurrent.atomic.AtomicInteger;

class ContadorAtomic {
   private AtomicInteger c = new AtomicInteger(0);

   public void increment() {
      c.incrementAndGet();
   }

   public void decrement() {
      c.decrementAndGet();
   }

   public int value() {
```

```
        return c.get();
    }

}
```

El paquete java.util.concurrent.atomic cuenta con clases para distintos tipos de variables:

AtomicBoolean
AtomicInteger
AtomicIntegerArray
AtomicIntegerFieldUpdater<T>
AtomicLong
AtomicLongArray
AtomicLongFieldUpdater<T>
AtomicMarkableReference<V>
AtomicReference<V>
AtomicReferenceArray<E>
AtomicReferenceFieldUpdater<T,V>
AtomicStampedReference<V>

SINCRONIZACIÓN. MECANISMOS DE ALTO NIVEL. EJECUTORES. THREAD POOLS

El manejo de la ejecución de hilos puede llevarse a cabo por el programador, o bien, en aplicaciones más complejas, la creación y manejo de los hilos se pueden separar en clases especializadas. Estas clases se conocen como **ejecutores**, o **Executors**.

La **interfaz Executor** nos obliga a implementar un único método, **execute**(). Si r es un objeto Runnable y e un objeto Executor, entonces en lugar de iniciar el hilo con (new Thread(r)).start(), lo iniciaremos con e.execute(r). De la segunda manera no sabemos si se creará un nuevo hilo para ejecutar el método run() del Runnable, o si se reutilizará un hilo "worker thread" que ejecuta distintas tareas. Es más probable lo segundo. El Executor está diseñado para ser utilizado a través de las siguientes subinterfaces (aunque también se puede utilizar sólo).

La **interfaz ExecutorService** es subinterfaz de la anterior y proporciona un método más versátil, **submit**() (significa enviar), que acepta objetos Runnable, pero también acepta objetos Callable, que permiten a una tarea devolver un valor. El método submit() devuelve un objeto Future a través del cual se obtiene el valor devuelto, y a través del cual se obtiene el estado de la tarea a ejecutar.

También se permite el envío de colecciones de objetos Callable. ExecutorService tiene métodos para para el ejecutor pero las tareas deben estar programadas manejar las interrupciones de manera adecuada (no capturarlas e ignorarlas).

La **interfaz ScheduledExecutorService** es a su vez subinterfaz de la última, y aporta el método **schedule**() que ejecuta un objeto Runnable o Callable después de un retardo determinado. Además define scheduleAtFixedRate y scheduleWithFixedDelay que ejecutan tareas de forma repetida a intervalos de tiempo determinados.

La mayoría de implementaciones de java.util.concurrent utilizan **pools de hilos** que consisten en "worker threads" que existen de manera separada de los Runnables y Callables. El uso de estos working thread minimiza la carga de CPU evitando creaciones de hilos nuevos. La carga consiste sobre todo en liberación y reseva de memoria, ya que los hilos utilizan mucha.

Un tipo de pool común es el "**fixed thread pool**" que tiene un número prefijado de hilos en ejecución. Si un hilo acaba mientras todavía está en uso, éste es automáticamente reemplazado por otro. Las tareas se envían al pool a través de una cola interna que mantiene las tareas extra que todavía no han podido entrar en un hilo de ejecución. De esta manera, si hubiera más tareas de lo normal, el número de hilos se mantendría fijo sin degradar el uso de CPU, aunque lógicamente, habrá tareas en espera y eso podrá repercutir, dependiendo de la aplicación.

Una manera sencilla de crear un ejecutor que utiliza un fixed thread pool es invocando el método estático **newFixedThreadPool**(int nThreads) de la clase java.util.concurrent.Executors. Esta misma clase también tiene los métodos newCachedThreadPool que crea un ejecutor con un pool de hilos ampliable, y el método newSingleThreadExecutor que crea un ejecutor que ejecuta una única tarea al mismo tiempo. Alternativamente se pueden crear instancias de java.util.concurrent.ThreadPoolExecutor o de java.util.concurrent.ScheduledThreadPoolExecutor que cuentan con más opciones.

Ejemplo de cómo crear un una instancia de java.util.concurrent.ThreadPoolExecutor

```
//Al principio del fichero:
//import java.util.concurrent.*;
//import java.util.*;

int poolSize = 2;
int maxPoolSize = 2;
long keepAliveTime = 10;
final ArrayBlockingQueue<Runnable> queue =  new ArrayBlockingQueue<Runnable>(5);
ThreadPoolExecutor threadPool = new ThreadPoolExecutor(poolSize, maxPoolSize, keepAliveTime, TimeUnit.SECONDS, queue);

Runnable myTasks[3] = ....; // y le asignamos tareas

//Poner a ejecutar dos tareas y una que quedará en cola:
for(int i=0; i<3; i++){
   threadPool.execute(task);
   System.out.println("Tareas:" + queue.size());
}

//Encolar otra tarea más que declaramos aquí mismo:
threadPool.execute( new Runnable() {
      public void run() {
         for (int i = 0; i < 10; i++) {
            try {
               System.out.println("i = " + i);
               Thread.sleep(1000);
            } catch (InterruptedException ie){  }
         }
      }
   });

//Ejecuta las tareas que queden pero ya no acepta nuevas:
threadPool.shutdown();
```

También es frecuente sobrecargar ThreadPoolExecutor para añadir algún comportamiento adicional. Por ejemplo, hacer que sea **pausable**

```
class PausableThreadPoolExecutor extends ThreadPoolExecutor {
   private boolean isPaused;
   private ReentrantLock pauseLock = new ReentrantLock();
   private Condition unpaused = pauseLock.newCondition();

   //Constructor: utilizamos el del padre
   public PausableThreadPoolExecutor(...) { super(...); }

   //Sobrecargamos el método:
   protected void beforeExecute(Thread t, Runnable r) {
    super.beforeExecute(t, r);
    pauseLock.lock(); //Sección sincronizada
    try {
     while (isPaused) unpaused.await(); //Bloquearlo
    } catch (InterruptedException ie) {
     t.interrupt();
    } finally {
     pauseLock.unlock();
    }
   }

   //Método nuevo:
   public void pause() {
    pauseLock.lock(); //Sección sincronizada
    try {
     isPaused = true;
    } finally {
     pauseLock.unlock();
    }
   }

   //Método nuevo:
   public void resume() {
    pauseLock.lock(); //Sección sincronizada
    try {
     isPaused = false;
     unpaused.signalAll(); //Desbloquear hilos bloqueados
    } finally {
     pauseLock.unlock();
    }
   }
}
```

Nótese que en el anterior código se pausa la ejecución de nuevas tareas pero no se pausan las que ya están ejecutándose. El problema aquí es que cada hilo debe comprobar por si mismo si debe seguir ejecutándose o no. Es decir, es responsabilidad del programador programar un mecanismo de pausado en sus hilos.

EJERCICIOS DE SINCRONIZACIÓN DE HILOS

13. Crea la clase Java SincronizarReloj que muestra la hora del sistema actualizándola segundo a segundo.

14. Crea la clase Java **Pizza** que contiene un String de nombre tipoPizza y los métodos sincronizados **repartirPizza** que espera hasta que haya pizzas disponibles y, cuando hay una pizza disponible, devuelve un String con el tipo de la pizza, y el método **hornearPizza** que recibe un String con un tipo de pizza, espera a que no queden pizzas disponibles y, cuando no hay pizzas disponibles, actualiza el valor del tipo de la pizza e informa de que ya hay pizzas disponibles. Crea también las clases **ProductorPizza**, **ConsumidorPizza**, y **ProductorConsumidorPizza** que se encargan de que la producción y el consumo de pizzas se haga de manera sincronizada.

15. Crea la clase Java **ThreadPoolMain** que crea un thread pool con 10 hilos de los que sólo pueden ejecutarse 2 cada vez. La clase encargada del código de los hilos es la clase Tarea que recibe el nombre del hilo, muestra un mensaje indicando que el hilo está en ejecución, espera un tiempo aleatorio entre 1 y 5 segundos, y muestra otro mensaje indicando la finalización del hilo. Es importante comprobar en la salida por pantalla que en cada momento solo hay dos hilos en ejecución.

SEMÁFOROS

Desde la versión 1.5 de Java disponemos de la clase **Semaphore** para facilitar la sincronicación de procesos. Para usar la clase Semaphore la debemos importar desde **java.util.concurrent.Semaphore**.

Los semáforos permiten al igual que los semáforos de la vida real acceder o no a un código determinado.

Se usan para controlar el acceso a recursos compartidos por varios hilos de ejecución.

Cuando se usan para permitir el acceso a un único hilo de ejecución cada vez se denominan semáforos binarios ya que permiten su inicialización es un 1 y sus únicos valores posibles son 1 y 0.

El número de procesos que pueden acceder al código protegido por el semáforo se indica en el momento de la creación del semáforo.
Por ejemplo, si queremos crear un semáforo que sólo permita el acceso simultáneo de un proceso a un determinado código escribimos

```
Semaphore semaforo = new Semaphore (1);
```

Para esperar hasta que se libere el recurso por parte del semáforo y se pueda entrar en la zona protegida existe el método **adquire()**. Si el semáforo está cerrado el hilo de ejecución se queda en espera.

Para indicar que ya he terminado de ejecutar el código de la zona protegida y liberar el semáforo para que otro hilo de ejecución pueda acceder a él existe el método **release()**. Es muy importante que cuando un hilo finalice la ejecución del código de la zona protegida libere el semáforo ya que sino los otros hilos no pordrán continuar su ejecución y se producirá un interbloqueo.

EJERCICIOS DE SEMÁFOROS

16. Crea la clase Java SemaforosCaramelos que cuenta con una variable de tipo entero y de nombre caramelos que inicialmente vale 10 y tres objetos de la clase Hijo que hereda de Thread. Los hijos intentan coger caramelos mientras queden. Solo un hijo puede acceder cada vez al bote de caramelos, los demás deben esperar. El control de la sincronización se hará usando semáforos. La clase Hijo se creará en el mismo fichero que la clase principal.

17. Crea la clase Java SemaforosContadorMain que cuenta con una variable de tipo entero y de nombre contador que inicialmente vale 10 y tres objetos de la clase ContadorHilo que hereda de Thread. Los objetos de la clase ContadorHilo intentan acceder al contador mientras no sea nulo. Solo un hijo puede

acceder cada vez al contador, los demás deben esperar. El control de la sincronización se hará usando semáforos. La clase ContadorHilo se creará en el mismo fichero que la clase principal.

CYCLICBARRIER

Cuando trabajamos con hilos en Java, nos puede interesar que varios hilos comiencen su ejecución a la vez o nos puede interesar esperar a que terminen todos los hilos que hemos lanzado previamente. Esperar a un solo hilo no es problema, ya que cada hilo tiene el método **join()** que hace exactamente eso, esperar a que termine su ejecución.

Sin embargo, esperar por muchos hilos nos implicaría hacer muchas llamadas join(), una por cada hilo arrancado lo que no es práctico.

Para arrancar varios hilos a la vez o para esperar que todos ellos terminen, Java nos ofrece la clase **CyclicBarrier**. Esta clase se instancia pasándole en el constructor cuántos hilos debe sincronizar. Los hilos deben llamar al método **await()** de CyclicBarrier y se quedarán ahí detenidos. CyclicBarrier los liberará cuando tenga tantos hilos a la espera como se le haya indicado en el constructor.
Si queremos que varios hilos empiecen a la vez (por ejemplo, 100 hilos), tendremos que hacer lo siguiente

```
import java.util.concurrent.CyclicBarrier;

public class PruebaCyclicBarrier {

   public static void main(String[] args) {
      int numeroHilos = 100;
      final CyclicBarrier barreraInicio = new CyclicBarrier(numeroHilos + 1);
      final CyclicBarrier barreraFin = new CyclicBarrier(numeroHilos + 1);

      for (int i = 0; i < numeroHilos; i++) {
         Thread hilo = new Thread() {
            public void run() {
               try {
                  barreraInicio.await();
                  System.out.println("hilo ejecutandose");
                  barreraFin.await();
               } catch (Exception e) {
                  e.printStackTrace();
               }
            }
         };
         hilo.start();
      }

      try {
         System.out.println("levanto barrera");
         barreraInicio.await();
         barreraFin.await();
         System.out.println("todo acabado");
      } catch (Exception e) {
         e.printStackTrace();
      }

   }
```

```
}
```

El ejemplo anterior hace que el método main() lance 100 hilos que se quedan bloqueados hasta que les da la señal de arrancar (En realidad, hasta que se llama **barreraInicio.await()** por 101 vez, cualquier hilo puede ser el 101 en hacer la llamada, pero desde luego, todos esperarán por el último y empezarán a la vez). Una vez arrancados, espera a que termine la ejecución de todos ellos (**barreraFin.await()**).

EJERCICIOS DE CYCLICBARRIER

18. Crea la clase Java CyclicBarrierAutobus que simula el funcionamiento de un autobús de 56 plazas. Espera hasta que el autobús está lleno y realiza el viaje.

COMUNICACIÓN ENTRE PROCESOS. BLOCKINGQUEUE

Uno de los mayores problemas que nos encontramos cuando trabajamos con varios hilos de ejecución en Java consiste en controlar la comunicación de información entre los distintos hilos.

Java proporciona unas cuantas clases para ello. Una de ellas es la clase **BlockingQueue** que es una implementación de una cola que permite que un hilo deje datos para que otro lo coja asegurando que no se producen problemas de sincronismo.

Para definir un objeto de la clase BlockingQueue que va a almacenar datos de tipo Integer escribimos

```
BlockingQueue<Integer> fila;
```

La clase BlockingQueue dispone del método **put()** para añadir datos asegurando el sincronismo. Por ejemplo, si tenemos una clase de nombre ProductorBlockingQueue que genera hilos que hacen la función de productor y que se encarga de generar números y añadirlos a la cola escribimos.

```
class ProductorBlockingQueue implements Runnable{
	private final BlockingQueue<Integer> fila;

	public ProductorBlockingQueue(BlockingQueue<Integer> q) {
		fila = q;
	}

	@Override
	public void run() {
		try{
			int val=0;
			boolean flag=true;
			while(flag){
				Thread.sleep(1333);
				if(val>1000){
					flag=false;
				}
				fila.put(val);
				val++;
			}
		}catch(Exception ex){
			ex.printStackTrace();
```

```
                }
        }

}
```

La clase BlockingQueue también dispone del método **take()** que coge datos de la cola asegurando el sincronismo. Por ejemplo, si tenemos una clase de nombre ConsumidorBlockingQueue que genera hilos que hacen la función de consumidor y que se encarga de coger valores de la cola y mostrarlos por pantalla.

```
class ConsumidorBlockingQueue implements Runnable{

        private final BlockingQueue<Integer> fila;
        private final String hiloNombre;

        public ConsumidorBlockingQueue(BlockingQueue<Integer> q,String nom) {
                fila = q;
                hiloNombre = nom;
        }

        public void run(){
                try{
                        boolean flag=true;
                        int con=0;
                        while(flag){
                                consumir(fila.take());
                                if(con>400){
                                        flag=false;
                                }
                                con++;
                        }
                }catch(Exception ex){
                        ex.printStackTrace();
                }
        }

        void consumir(Object x) throws InterruptedException{
                System.out.println("Consumiendo "+hiloNombre+" Valor de la fila: "+x);
                Thread.sleep(1000);
        }
}
```

EJERCICIOS DE COMUNICACIÓN ENTRE PROCESOS. BLOCKINGQUEUE

19. Crea la clase Java BlockingQueueCadenaProduccion que simula el funcionamiento de una cadena de producción de tres fases. La cadena de producción cuenta con las clases Productor01 que genera la cadena "P01" y la introduce en la cola, la clase Productor02 que coge de la cola las piezas producidas por Productor01 las transforma (añade la cadena "P02" a la anterior con lo que queda "P01P02") y las coloca en otra cola, y clase Productor03 que coge de la cola las piezas producidas por Productor02 las transforma (añade la cadena "P03" a la anterior con lo que queda "P01P02P03") y las muestra por pantalla indicando el número de piezas producidas hasta el momento. El programa finalizará cuando se generen 100 piezas.

20. Crea la clase Java BlockingQueueCadenaProduccion2P1C que simula el funcionamiento de una cadena de producción de dos fases la primera de ellas con dos productores distintos. La cadena de producción cuenta con las clases ProductorP1 que genera la cadena "P01" y la introduce en la cola, la clase ProductorP2 que genera la cadena "P02" y la introduce en la cola, y clase ProductorP3 que coge de las colas las piezas producidas por Productor01 y Productor02 y las transforma (añade la cadena "P03" con lo que queda "P01P02P03") y las muestra por pantalla indicando el número de piezas producidas hasta el momento. El programa finalizará cuando se generen 100 piezas.

21. Crea la clase Java BlockingQueueChat1Cliente que simula el funcionamiento de un chat que tiene sólo 1 cliente. Para el correcto funcionamiento se usa una cola en la que escribirá el Cliente y leerá el Servidor. La ejecución finalizará cuando el cliente mande el mensaje "fin" al servidor. Nota: Para la simulación de chats de más de un cliente en modo local es mejor usar aplicaciones gráficas ya que en modo consola todos los clientes comparten la consola y eso produce muchos errores.

THREADS. MÚLTIPLES HILOS DE EJECUCIÓN EN APLICACIONES CLIENTE / SERVIDOR

En la mayoría de las aplicaciones cliente / servidor se trabaja con múltiples hilos de ejecución o threads.

Normalmente, para gestionar mejor cada uno de los clientes, cada vez que se establece una conexión en el proceso servidor se crea un nuevo hilo de ejecución o thread que se encarga de tratar con el cliente. De este modo si hay algún problema en un cliente se puede solucionar finalizando la ejecución del hilo de ejecución asociado a ese cliente sin afectar a la ejecución de los otros clientes.

Para crear un nuevo hilo de ejecución en Java existe la clase Thread.

Por ejemplo, si queremos que cada vez que se acepte una conexión desde un cliente se cree un nuevo thread escribimos dentro del código del servidor

```
while (true) {
   Socket clienteAceptado = s.accept();

   Thread t = new ThreadServerHandler(clienteAceptado);
   t.start();
  }
```

El código que ejecuta cada Thread se suele incluir en una nueva clase que deriva de la clase Thread. Por ejemplo, si queremos crear la clase **ThreadServerHandler** que extiende a la clase Thread y contiene el bucle de comunicación entre el servidor y el cliente en su método **run()** escribimos

```
 class ThreadServerHandler extends Thread {
  ...
  public void run() {
   try {
    // Establecer los flujos de entrada/salida para el socket
    // Procesar las entradas y salidas según el protocolo
    // cerrar el socket
   }
   catch (Exception e) {
    // manipular las excepciones
   }
  }
```

```
}
```

SERVIDORES ITERATIVOS Y SERVIDORES CONCURRENTES

En las aplicaciones cliente / servidor podemos encontrarnos con servidores iterativos y servidores concurrentes.

Un **servidor iterativo** atiende las peticiones de los clientes una tras otra, es decir, **en serie**. No crea un nuevo hilo de ejecución por cada nuevo cliente.

Un **servidor concurrente** es capaz de tratar peticiones de varios clientes a la vez, es decir, atiende las peticiones **en paralelo**. Para ello crea un nuevo hilo de ejecución por cada nuevo cliente.

TRATAMIENTO DE EXCEPCIONES DE SOCKETS Y THREADS

Las aplicaciones cliente / servidor son muy propensas a fallos por lo que el control de excepciones es fundamental para el correcto funcionamiento de las aplicaciones.

Java proporciona unas clases que permite llevar un seguimiento de la ejecución de la aplicación para qué en caso de que se produzca un error saber que ha sido lo que lo ha provocado.

Estas clases se encuentran dentro de **java.util.logging**.

java.util.logging define algunos niveles predefinidos:

OFF. No se genera traza alguna.
SEVERE. Se usa para trazar errores catastróficos, que son aquellos de los que no hay recuperación posible, provocando la terminación del programa. La traza recoge el fallo causante de la detención.
WARNING. Se usa para trazar errores peligrosos, para los que hay previsto un mecanismo de supervivencia.
INFO. Trazas normales: para ir viendo lo que pasa.
CONFIG. Se usan típicamente al arrancar un programa para trazar la configuración inicial, que frecuentemente se lee de alguna parte.
FINE. Información de detalle, típicamente útil para localizar errores (depuración).
FINER. Información de más detalle, típicamente útil para localizar errores (depuración).
FINEST. Información de máximo detalle, típicamente útil para localizar errores (depuración).
ALL. Se traza todo, a cualquier nivel.

Algunos ejemplos de uso son

```
LOGGER.log(Logger.WARNING, "mensaje");
LOGGER.severe("mensaje");
LOGGER.warning("mensaje");
LOGGER.info("mensaje");
LOGGER.config("mensaje");
LOGGER.fine("mensaje");
LOGGER.finer("mensaje");
LOGGER.finest("mensaje");
```

Los mensajes de log se suelen utilizar para completar la información proporcionada por el tratamiento de excepciones.

Por ejemplo, para controlar los errores de entrada salida que se produzcan al trabajar con sockets en la clase HolaServidor escribimos

```
try {
 ...
} catch (IOException ex) {
 Logger.getLogger(HolaServidor.class.getName()).log(Level.SEVERE, null, ex);
} finally {
 if (socket != null) {
  try {
   socket.close();
  } catch (IOException ex) {
   Logger.getLogger(HolaServidor.class.getName()).log(Level.SEVERE, null, ex);
  }
 }
}
```

APLICACIONES CLIENTE / SERVIDOR MULTIHILO.

Hoy en día existen muchas aplicaciones cliente / servidor multihilo. Su funcionamiento es muy delicado ya que a los problemas de entrada / salida de datos se añaden los de comunicación entre distintos equipos y los de la gestión del funcionamiento de varios hilos de ejecución.

Los servidores que se encargan de gestionar las aplicaciones cliente / servidor multihilo son concurrentes ya que crean un nuevo hilo de ejecución por cada cliente conectado.

Como ejemplo de aplicación cliente / servidor multihilo vamos a crear un Chat. Es muy importante destacar que es más fácil la implementación del ejemplo usando una aplicación gráfica ya que al usar la consola para entrada y salida de datos se producen errores que no se producen en una aplicación gráfica. No obstante vamos a realizar el ejemplo usando la consola como medio de comunicación.

Para la aplicación vamos a necesitar varias clases.

Para comenzar vamos a crear un chat en el que los clientes sólo mandan un mensaje y finalizan su ejecución.

EJERCICIOS DE SOCKETS Y THREADS. SINCRONIZACIÓN DE HILOS

22. Crea la clase Java HolaServidor que permite la conexión de un cliente en el puerto 8888. Si se realiza correctamente la conexión enviará al cliente el mensaje "Hola Cliente". Si se produce un error durante el proceso de conexión mostrará los datos del error producido.

23. Crea la clase Java HolaCliente que se conecta al servidor que se encuentra en la dirección IP 127.0.0.1 en el puerto 8888. Si se realiza correctamente la conexión recibirá el mensaje "Hola Cliente" y lo mostrará. Si se produce un error durante el proceso de conexión mostrará los datos del error producido.

24. Crea la clase Java HolaServidorIterativo que permite la conexión de varios clientes en el puerto 8888. Si se realiza correctamente la conexión enviará al cliente el mensaje "Hola Cliente "+numeroCliente, dónde numeroCliente es una variable que controla el número de clientes que se han conectado. Si se produce un error durante el proceso de conexión mostrará los datos del error producido.

25. Crea la clase Java HolaClienteIterativo que se conecta al servidor que se encuentra en la dirección IP 127.0.0.1 en el puerto 8888. Si se realiza correctamente la conexión recibirá el mensaje "Hola Cliente "+numeroCliente, dónde numeroCliente es una variable que controla el número de clientes que se han conectado al servidor, y lo mostrará. Si se produce un error durante el proceso de conexión mostrará los datos del error producido.

26. Crea la clase Java HolaServidorConcurrente que permite la conexión de varios clientes en el puerto 8888. Si se realiza correctamente la conexión creará un nuevo hilo de ejecución usando la clase **ThreadHolaServidorConcurrente** que recibirá como parámetros el socket de la conexión y el número de cliente y enviará al cliente el mensaje "Hola Cliente "+numeroCliente. numeroCliente es una variable que controla el número de clientes que se han conectado.

27. Crea la clase Java HolaClienteConcurrente que se conecta al servidor que se encuentra en la dirección IP 127.0.0.1 en el puerto 8888. Si se realiza correctamente la conexión recibirá el mensaje "Hola Cliente "+numeroCliente, dónde numeroCliente es una variable que controla el número de clientes que se han conectado al servidor, y lo mostrará. Si se produce un error durante el proceso de conexión mostrará los datos del error producido.

28. Crea la clase Java **MensajesServidor** que se encarga de escuchar peticiones de clientes en el puerto 8888 y aceptar las conexiones. Cuando se acepta una conexión de un cliente se crea un nuevo hilo de ejecución de la clase MensajesServidorHilo. Almacena los sockets de los clientes en una lista. La finalización de la aplicación servidor no se produce nunca ya que siempre pueden llegar clientes nuevos.

29. Crea la clase Java **MensajesServidorHilo** que recibe el socket del cliente que queremos controlar y la lista de sockets cliente. Lee del socket los mensajes que envía el cliente y los reenvía a todos los clientes indicando los datos del cliente que lo envía, menos a sí mismo, que se lo envía sin modificar. Es importante no olvidar que hay que eliminar de la lista de sockets los sockets de los clientes que finalicen su ejecución.

30. Crea la clase Java **MensajesCliente** que se conecta al servidor y una vez conectado crea dos hilos de ejecución. El primero **MensajesClienteLeer** lee mensajes del socket y los muestra por pantalla. El segundo **MensajesClienteEscribir** pide mensajes por teclado y los escribe en el socket. Cuando finaliza la ejecución del hilo MensajesClienteEscribir, al recibir el mensaje "**fin**", usa el método **flush** de DataOutputStream en el socket para forzar a que el hilo MensajesClienteLeer lea el mensaje de fin. Además, para forzar la finalización completa del cliente (ya que aunque finalicen los hilos el cliente no termina pues los hilos se quedan en estado suspendido) uso el comando **System.exit(0)**. La finalización de un cliente se produce cuando manda el mensaje "**fin**". Es importante que al finalizar un cliente se liberen todos los recursos utilizados por él.

31. Crea la clase Java **MensajesClienteEscribir** que pide mensajes por teclado y los escribe en el socket. Cuando el cliente escribe el mensaje "**fin**" en el teclado introduce el valor "**fin**" en una cola sincronizada que comparte con el cliente para controlar cuando se produce el fin del hilo MensajesClienteEscribir. El cliente al recibir el mensaje de la cola finaliza su ejecución forzando a que el hilo MensajesClienteLeer lee todos los mensajes de la cola.

32. Crea la clase Java **MensajesClienteLeer** que lee mensajes del socket y los muestra por pantalla. Cuando lee el mensaje "**fin**" finaliza su ejecución.