

# UT12. Lectura y escritura de información

## Tabla de contenido

<b>1.</b>	<b>UTILIZACIÓN DE FLUJOS .....</b>	<b>2</b>
<b>1.1.</b>	<b>CLASES RELATIVAS A FLUJOS .....</b>	<b>2</b>
1.1.1.	CLASES PARA MANEJO DE FLUJOS DE BYTES .....	2
1.1.2.	CLASES PARA MANEJO DE FLUJOS DE CARACTERES .....	6
<b>2.</b>	<b>APLICACIONES DEL ALMACENAMIENTO DE INFORMACIÓN EN FICHEROS .....</b>	<b>11</b>
2.1.	TIPOS DE FICHEROS. REGISTRO.....	11
<b>2.2.</b>	<b>CREACIÓN Y ELIMINACIÓN DE FICHEROS Y DIRECTORIOS .....</b>	<b>11</b>
<b>2.3.</b>	<b>APERTURA Y CIERRE DE FICHEROS. MODOS DE ACCESO .....</b>	<b>13</b>
2.3.1.	ACCESO SECUENCIAL.....	13
2.3.2.	ACCESO DIRECTO.....	15
<b>2.4.</b>	<b>ESCRITURA Y LECTURA DE INFORMACIÓN EN FICHEROS .....</b>	<b>16</b>
2.4.1.	ACCESO SECUENCIAL.....	16
2.4.2.	ACCESO DIRECTO.....	17
<b>2.5.</b>	<b>UTILIZACIÓN DE LOS SISTEMA DE FICHEROS .....</b>	<b>19</b>

## 1. Utilización de flujos

Todos los flujos de información funcionan por igual y han de seguir los siguientes pasos:

- Crear el Stream para abrir el flujo a una fuente de datos.
- Leer-escribir mientras haya datos disponibles.
- Cerrar el Stream.

### 1.1. Clases relativas a flujos

Las clases para el trabajo con el flujo de datos se encuentran en el paquete “java.io” (io es abreviatura de input-output).

La clase para un flujo de entrada de “bytes” heredan de las clases abstractas “InputStream” y “OutputStream”. Implementan los siguientes métodos:

#### 1.1.1. Clases para manejo de flujos de bytes

Métodos InpuStream:

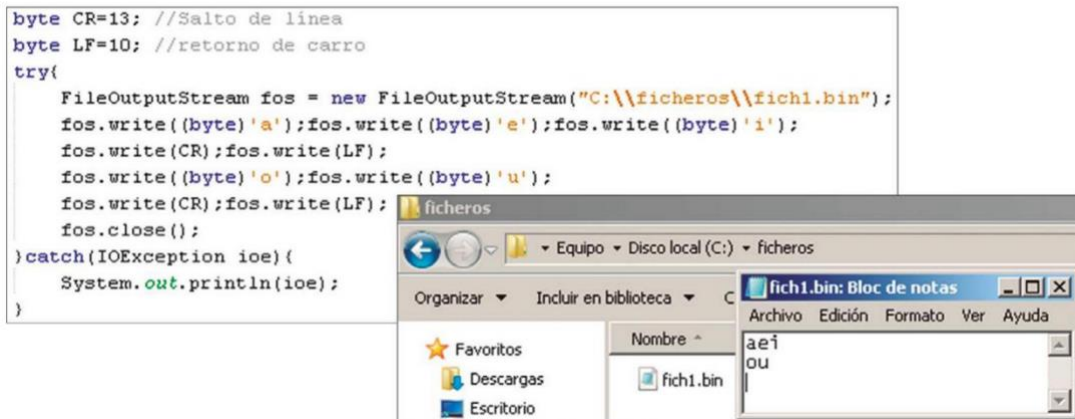
<code>int read()</code>	Lee el siguiente <i>byte</i> desde el <i>InputStream</i> . Si retorna -1, indicará que no se pueden leer más <i>bytes</i> .
<code>int available()</code>	Devuelve el número de <i>bytes</i> que se pueden leer del <i>InputStream</i> sin un bloqueo.
<code>void close()</code>	Cierra el <i>InputStream</i> .

Métodos OutputStream:

<code>void write(int b)</code>	Escribe un <i>byte</i> en el <i>OutputStream</i> .
<code>void write(byte[] a)</code>	Escribe todos los <i>bytes</i> del <i>array a</i> en el <i>OutputStream</i> .
<code>void flush()</code>	Vacía el flujo de salida actual del <i>OutputStream</i> .
<code>void close()</code>	Cierra el <i>OutputStream</i> y escribe los contenidos que haya en ese momento.

Clases principales para el manejo de flujos:

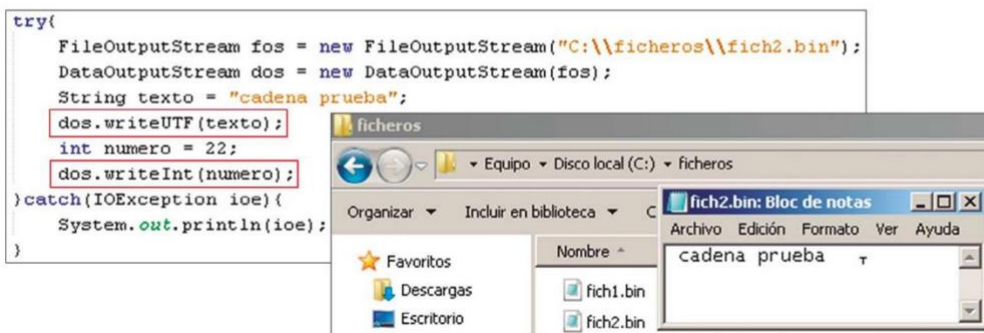
1. *FileOutputStream*: permite escribir *bytes* hacia un fichero binario.



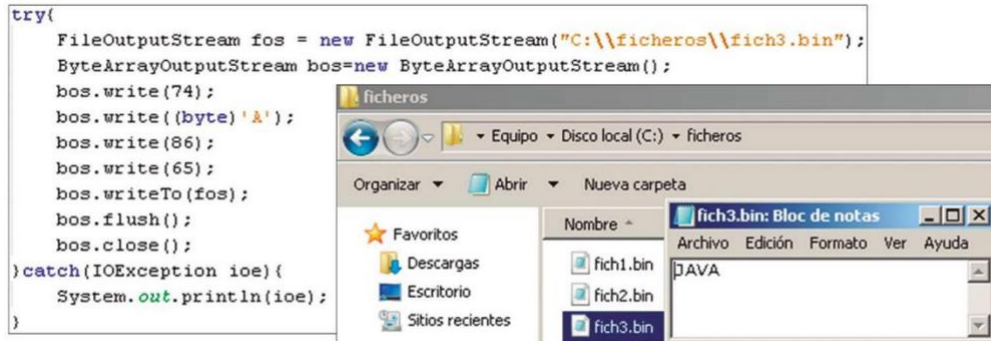
2. *BufferedOutputStream*: posibilita escribir información de otro *OutputStream* utilizando un *buffer* interno que mejora el rendimiento.

```
byte CR=13; //Salto de linea
byte LF=10; //retorno de carro
try{
    FileOutputStream fos = new FileOutputStream("C:\\\\ficheros\\\\fich1.bin");
    BufferedOutputStream bos=new BufferedOutputStream(fos);
    bos.write((byte)'a');bos.write((byte)'e');bos.write((byte)'i');
    bos.write(CR);bos.write(LF);
    bos.write((byte)'o');bos.write((byte)'u');
    bos.write(CR);bos.write(LF);
    bos.close();
    fos.close();
}catch(IOException ioe){
    System.out.println(ioe);
}
```

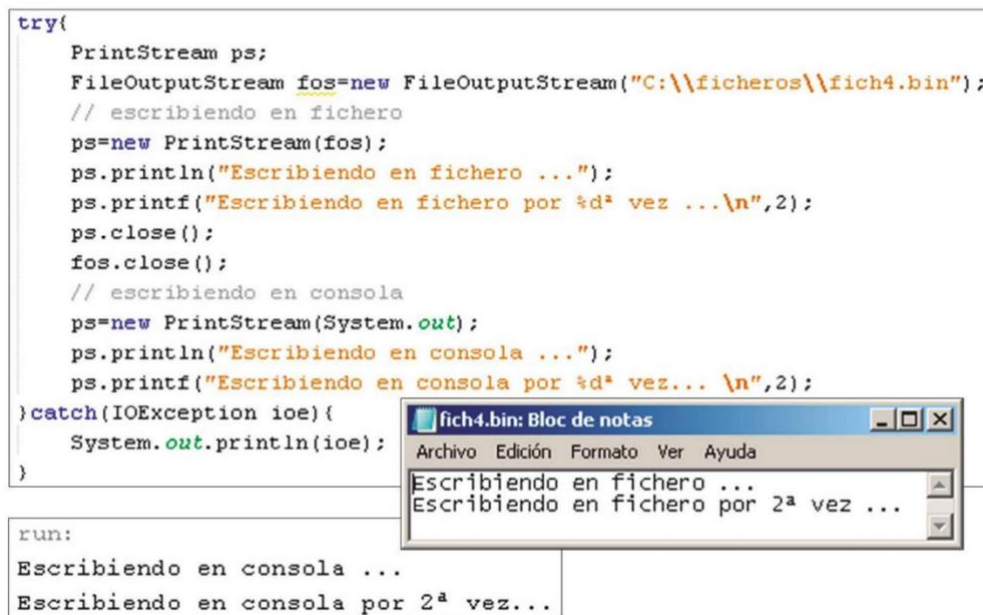
3. *DataOutputStream*: define los métodos *writeBoolean*, *writeByte*, *writeInt*..., que permiten escribir datos de tipo primitivo desde un *OutputStream*. Como puede verse en el ejemplo, no todo el contenido del fichero es legible ya que es un fichero binario.



4. *ByteArrayOutputStream*: posibilita usar un *array* de *bytes* como un *OutputStream*. El método *writeTo* permite indicar el fichero al que se transferirá el contenido del *array*. Una vez se ha terminado de escribir en el *array* los *bytes* a transferir, se ejecutará el método *flush()* que forzará a que sean transferidos.



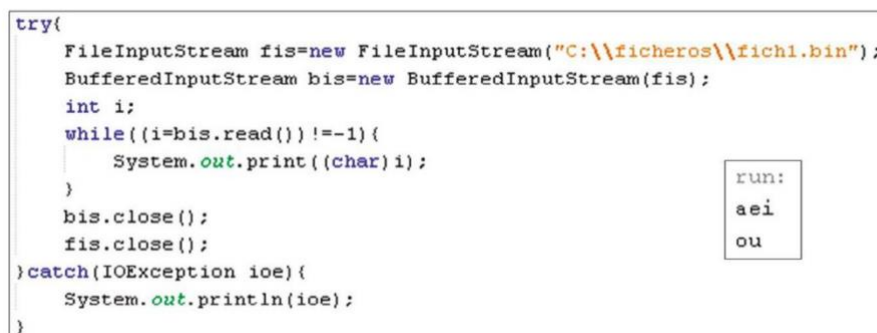
5. *PrintStream*: permite escribir los datos de un *Stream* en otro automáticamente y sin necesidad de invocar el método *flush()*. Este método no lanzará excepciones *IOException*. Los métodos que se pueden emplear ya fueron descritos en el apartado 5.2.2. (Entrada y salida por consola).



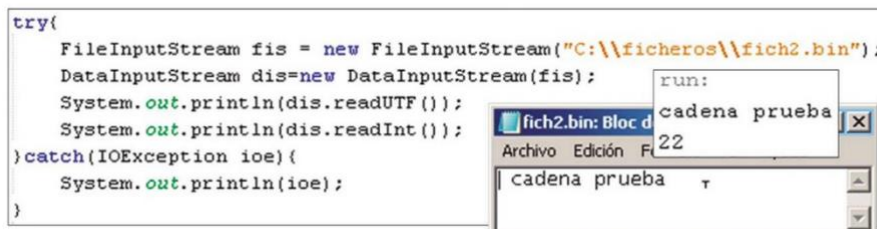
6. *FileInputStream*: posibilita leer *bytes* desde un fichero binario. Puesto que lo que se leen son *bytes*, se debe hacer un *cast* de los datos para poderlos manejar convenientemente.



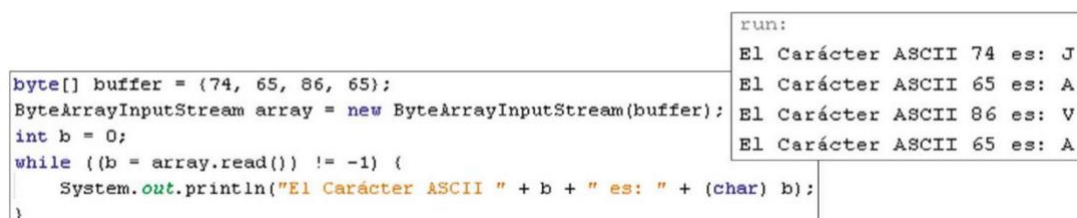
7. *BufferedInputStream*: permite leer información de otro *InputStream* utilizando un *buffer* interno que mejora el rendimiento. El resultado final es el mismo que utilizando únicamente *FileInputStream*.



8. *DataInputStream*: define los métodos *readBoolean*, *readByte*, *readUTF*..., que permiten leer datos de tipo primitivo desde un *InputStream*.



9. *ByteArrayInputStream*: permite usar un *array* de *bytes* como un *InputStream*.



10. *SequenceInputStream*: flujo de entrada que puede combinar varios *InputStream* en uno.

```
try{
    byte[] buffer = {65, 80, 82, 69, 78, 68, 73, 69, 78, 68, 79, 32};
    ByteArrayInputStream io1 = new ByteArrayInputStream(buffer);
    FileInputStream io2 = new FileInputStream("C:\\\\ficheros\\\\fich3.bin");
    SequenceInputStream inst=new SequenceInputStream(io1, io2);
    int j;
    while((j=inst.read())!=-1){
        System.out.print((char)j);
    }
    System.out.println(" !!!");
    inst.close();
    io1.close();
    io2.close();
}catch(IOException ioe){
    System.out.println(ioe);
}
```

run:  
APRENDIENDO JAVA !!!

11. *PipedOutputStream* y *PipedInputStream*: representan el concepto de *tubería* y son utilizados en programas multihilo. Permiten reproducir en un extremo (*PipedInputStream*) los datos que se están escribiendo en el extremo opuesto (*PipedOutputStream*).

### 1.1.2. Clases para manejo de flujos de caracteres

Métodos base de Reader:

<code>int read()</code>	Lee el siguiente carácter desde el <i>InputStream</i> . Si retorna -1, indicará que no se pueden leer más <i>bytes</i> .
<code>long skip(n)</code>	Hace que se omitan los <i>n</i> primeros caracteres.
<code>void close()</code>	Cierra el <i>Reader</i> .

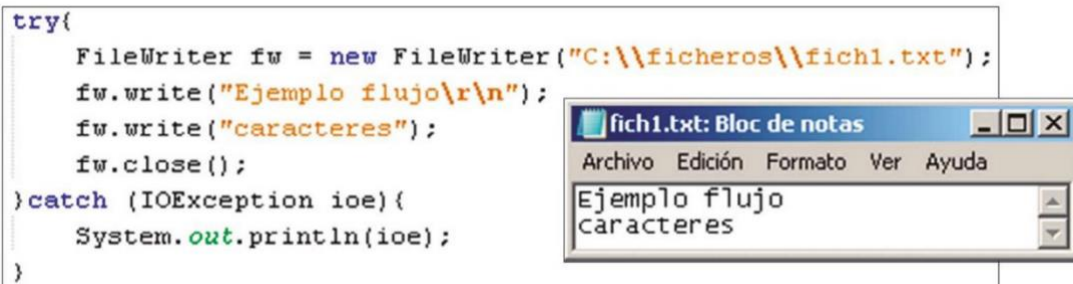
Métodos base de Writer:

<code>void write(int c)</code>	Escribe un carácter.
<code>void write(String cadena)</code>	Escribe la cadena.
<code>void write(char [] array)</code>	Escribe el <i>array</i> de caracteres.
<code>void flush()</code>	Vacía el flujo de salida actual del <i>Writer</i> .
<code>void close()</code>	Vacía el flujo de salida actual del <i>Writer</i> y lo cierra.

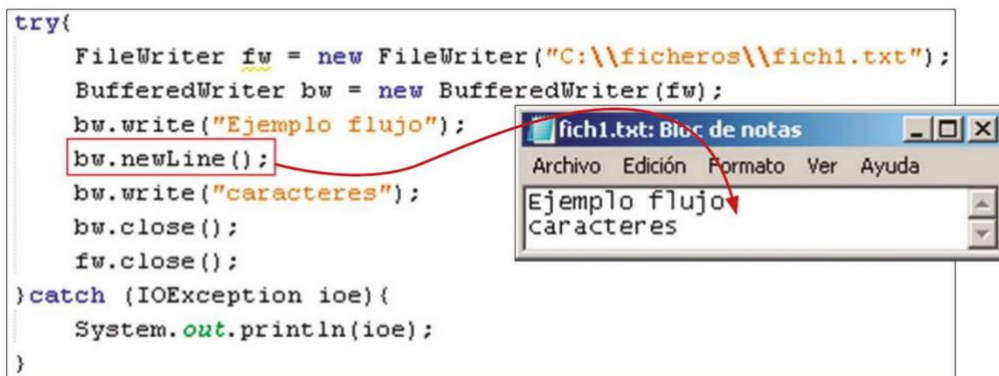


## Clases principales para el manejo de flujos:

1. *FileWriter*: permite escribir caracteres hacia un fichero de texto.



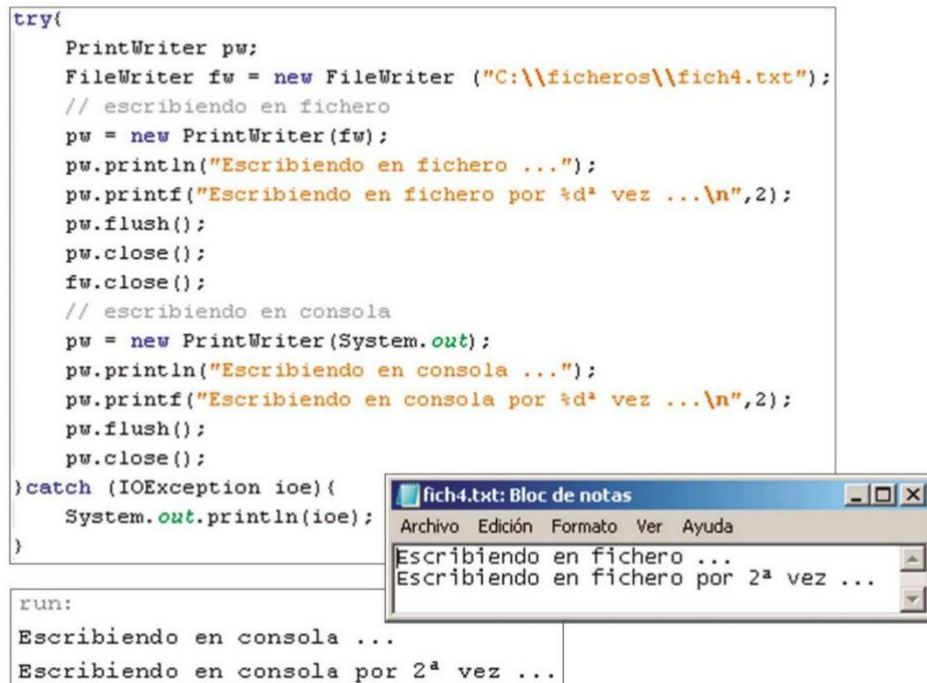
2. *BufferedWriter*: posibilita escribir información de otro *Writer* utilizando un *buffer* interno que mejora el rendimiento. El método *newLine()* permite añadir un salto de línea y retorno de carro en la cadena.



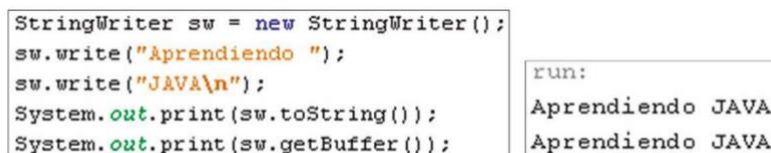
3. *CharArrayWriter*: permite usar un *array* de caracteres como *Writer*. El método *writeTo* permite indicar el *Writer* al que se transferirá el contenido del *array*.



4. *PrintWriter*: es la implementación de la clase *Writer*. Permite escribir una cadena de caracteres en cualquier *OutputStream*. Añade métodos *print*, *printf* y *println* para imprimir contenidos de forma similar a como hace *System.out*, pero hasta que no se ejecuta el método *flush()* o *close()* los contenidos no son escritos.



5. *StringWriter*: flujo de caracteres que recoge su salida en un *buffer* y que puede utilizarse para construir un *String*. El método *getBuffer()* devuelve el *buffer* contenedor de datos.

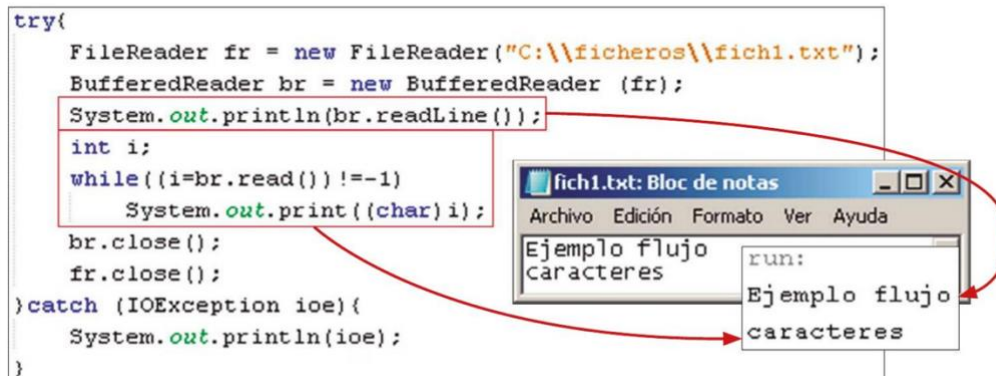


6. *FileReader*: flujo de caracteres destinado a la lectura de ficheros de texto.

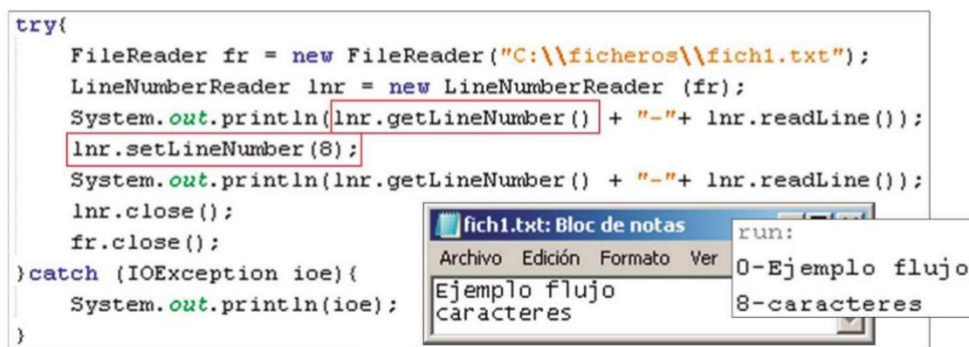




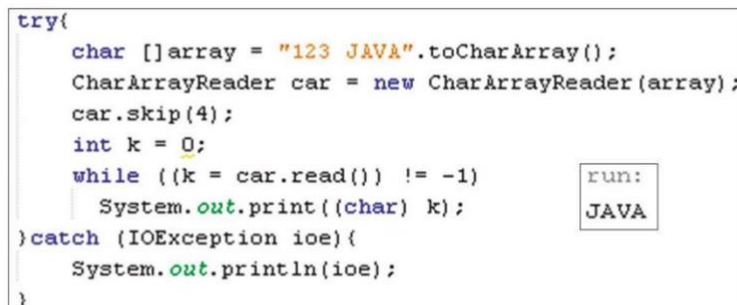
7. *BufferedReader*: permite leer información de otro *Reader* utilizando un *buffer* interno que mejora el rendimiento. Además del método *read()*, también dispone del método *readLine()*, que recupera una línea completa del *buffer*.



8. *LineNumberReader*: se trata de un tipo especial de *BufferedReader* que lleva un conteo del número de línea que está leyendo en cada momento. El método *getLineNumber()* devuelve el número de línea inicial (por defecto comienza en 0), y el método *setLineNumber(n)* permite establecer a *n* el número de la línea actual.



9. *CharArrayReader*: flujo de caracteres que recoge en un *buffer* los datos provenientes de un array de caracteres.



10. *StringReader*: flujo de caracteres que recoge en un *buffer* los datos provenientes de un *String*.

```
try{
    StringReader sr = new StringReader("Aprendiendo JAVA\n");
    int c=0;
    while{(c=sr.read())!=-1)
        System.out.print((char)c);
}catch (IOException ioe){
    System.out.println(ioe);
}
```

run:  
Aprendiendo JAVA

11. *PipedWriter* y *PipedReader*: similares a *PipedInputStream* y *PipedOutputStream* pero con flujos de caracteres.

## 2. Aplicaciones del almacenamiento de información en ficheros

### 2.1. Tipos de ficheros. Registro

Hay que tener en cuenta el tipo de contenido que se almacenará en el fichero:

- Si es texto, el acceso se hace de forma “secuencial”.  
Si hemos de acceder al carácter en la posición “i”, se tiene que recorrer desde el primero hasta el i-1.
- Si el contenido es “binario”, el acceso puede ser de forma secuencial o mediante acceso directo.  
Se puede acceder al byte ubicado en la posición “i” de forma directa y sin tener que recorrer todos los anteriores (como si del acceso a un array se tratara).

Al conjunto de datos que permanecen juntos al ser leídos o escritos en un fichero se le conoce como “REGISTRO”.

### 2.2. Creación y eliminación de ficheros y directorios

Antes de acceder a la lectura y modificación de ficheros, se debe conocer los mecanismos de creación, eliminación y manejo de ficheros-directorios. La clase “File” proporciona la funcionalidad necesaria para esto.

Constructores de la clase File:

File (String pathname)	Constructor de objetos <i>File</i> a partir de un <i>path</i> completo.
File (String padre, String hijo)	Constructor de objetos <i>File</i> a partir de un <i>path</i> padre y el <i>path</i> hijo contenido en él.
File (File padre, String hijo)	Constructor de objetos <i>File</i> a partir de un objeto <i>File</i> existente y el <i>path</i> del hijo contenido.

Métodos de la clase File:

<code>boolean createNewFile ()</code>	Crea un fichero vacío a partir de un objeto <i>File</i> existente, cuyo nombre es el <i>path</i> completo (únicamente si no existe previamente).
<code>boolean mkdir ()</code>	Construye un directorio a partir de un objeto <i>File</i> existente, cuyo nombre es el <i>path</i> completo si no existe previamente.
<code>boolean mkdirs ()</code>	Similar a <i>mkdir</i> pero creando todos los directorios intermedios del <i>path</i> que no existan.
<code>boolean delete ()</code>	Elimina el fichero o directorio identificado por el <i>path</i> .
<code>void deleteOnExit ()</code>	Fuerza a ejecutar un borrado del fichero o directorio identificado por el <i>path</i> cuando la máquina virtual finalice en condiciones normales.

Ejemplo básico de uso de algunos de los métodos descritos:



En Java 1.7 se incluye la clase “File” en el paquete “java.nio”. Aquí los métodos son estáticos y tiene un repertorio más complejo para crear, eliminar y maneja ficheros. Además, tiene las clases “Path” que permite localizar un fichero en un sistema de ficheros.

## 2.3. Apertura y cierre de ficheros. Modos de acceso

Es necesario saber cómo abrir un fichero para acceder al contenido y cómo cerrarlo una vez se ha terminado de usar. Dependiendo de cómo se acceda, se usará unas clases u otras.

### 2.3.1. Acceso secuencial

Se pueden usar las clases “FileReader”, “FileWriter”, “FileInputStream” y “FileOutputStream”, así como las clases “Buffered” relacionadas con estas.

Constructores:

```
FileReader (String path)
```

```
FileReader (File file)
```

```
FileInputStream (String path)
```

```
FileInputStream (File file)
```

```
FileWriter (String path [, boolean append])
```

```
FileWriter (File file [, boolean append])
```

```
FileOutputStream (String path [, boolean append])
```

```
FileOutputStream (File file [, boolean append])
```

Si es abierto para escritura con “FileWriter” o “FileOutputStream”:

- Si el fichero NO EXISTE, se crea y posteriormente se abre para la escritura.
- Si el fichero EXISTE, se abre para escritura y:
  - o Si el argumento “append” es “true”, el contenido se añade al final del fichero.
  - o Si el argumento “append” es “false” o se omite, el fichero se SOBREScribe.

Para cerrar el fichero, tan solo llamamos al método “close()”.



## Files

La clase “Files” tiene métodos para obtener objetos como “BufferedReader” y “BufferedWriter” que permite el acceso “secuencial” a ficheros de texto plano, y objeto “InputStream” y “OutputStream” con los que se realiza acceso secuencial a ficheros binarios.

Se puede especificar el “modo” de acceso al fichero a través de opciones.

Métodos para el acceso a ficheros:

```
BufferedReader newBufferedReader (Path path [, Charset cs])
BufferedWriter newBufferedWriter (Path path [, Charset cs][, OpenOption opcion1 ..., OpenOption opcionN])
InputStream newInputStream (Path path [, OpenOption opcion1 ..., OpenOption opcionN])
OutputStream newOutputStream (Path path [, OpenOption opcion1 ..., OpenOption opcionN])
```

Path nos permite ubicar los ficheros en un sistema de archivos.

Cs referencia al conjunto de caracteres a emplear. Si se omite, por defecto se establece “StandardCharsets.UTF\_8”.

OpenOption se encuentra en la clase enumerada “StandardOpenOption”. La siguiente tabla refleja los posibles valores.

APPEND	Si el fichero es abierto para escritura, los <i>bytes</i> son añadidos al final del fichero.
CREATE	Crea un fichero nuevo en caso de que no exista.
CREATE_NEW	Crea un fichero nuevo en caso de que no exista y lanza una excepción en caso de que ya existiera.
DELETE_ON_CLOSE	Elimina el fichero al cerrarlo.
DSYNC	Fuerza que cada actualización del contenido del archivo se escriba de forma síncrona en el dispositivo de almacenamiento subyacente.
READ	Abre el fichero para lectura.
SPARSE	Cuando se usa con la opción CREATE_NEW, proporciona una sugerencia de que el nuevo archivo será un archivo <i>disperso</i> . La opción se ignora cuando el sistema de archivos no admite la creación de archivos dispersos.
SYNC	Fuerza que cada actualización del contenido del archivo o de sus metadatos se escriba de forma síncrona en el dispositivo de almacenamiento subyacente.
TRUNCATE_EXISTING	Si el fichero ya existe y es abierto para escritura, lo sobrescribe.
WRITE	Abre el fichero para escritura.

### 2.3.2. Acceso directo

Si el acceso es a un fichero “binario” lo hacemos a través de la clase “RandomAccessFile”.

Lista de constructores:

```
new RandomAccessFile(String path, String modo);  
new RandomAccessFile(File file, String modo);
```

Modos de acceso para RandomAccessFile:

r	Abrir para lectura. Al invocar cualquier método de escritura del objeto resultante se generará una excepción <i>IOException</i> .
rw	Abrir para leer y escribir. Si el archivo no existe, intentará crearlo.
rwd	Abrir para leer y escribir, como con “rw”, pero requiere que cada actualización del contenido del archivo se escriba de forma síncrona en el dispositivo de almacenamiento subyacente.
rws	Similar a “rwd”, pero también forzará a que se escriba de forma síncrona en el dispositivo de almacenamiento subyacente cualquier cambio producido en los metadatos del archivo.

## 2.4. Escritura y lectura de información en ficheros

### 2.4.1. Acceso secuencial

Para el acceso secuencial a ficheros de texto o binarios, podemos usar los métodos “read” o “write” de los flujos de entrada y salida.

#### Files

Si usamos Files para leer y escribir de forma secuencial los ficheros, podemos tener:

- Métodos de Files para la escritura y lectura de información:

<code>Path write (Path path, byte[] bytes [,OpenOption op1 ... [, OpenOption opN])</code>	Escribe los <i>bytes</i> en el fichero representado por <i>path</i> .
<code>byte[] readAllBytes (Path path)</code>	Lee todos los <i>bytes</i> del fichero representado por <i>path</i> y los devuelve en un <i>array</i> de <i>bytes</i> .
<code>List&lt;String&gt; readAllLines (Path path [, Charset cs])</code>	Devuelve una colección de <i>String</i> correspondiente a las líneas del fichero representado por <i>path</i> .

Ejemplo:

```
try{
    // Escritura
    Path fichero = Paths.get("C:\\ficheros\\Nuevo2.bin");
    byte [] texto1 = {65, 80, 82, 69, 78, 68, 73, 69, 78, 68, 79, 13, 10};
    byte [] texto2 = {74, 65, 86, 65, 13, 10};
    Files.write(fichero, texto1);
    Files.write(fichero, texto2, StandardOpenOption.APPEND);
    // Lectura en array de bytes
    byte [] contenido = Files.readAllBytes(fichero);
    for (int i=0; i<contenido.length; i++)
        System.out.print((char) contenido[i]);
    // Lectura en colección
    List<String> lineas = Files.readAllLines (fichero);
    System.out.print(lineas.toString());
}catch (IOException ioe){
    System.out.println(ioe);
}
```

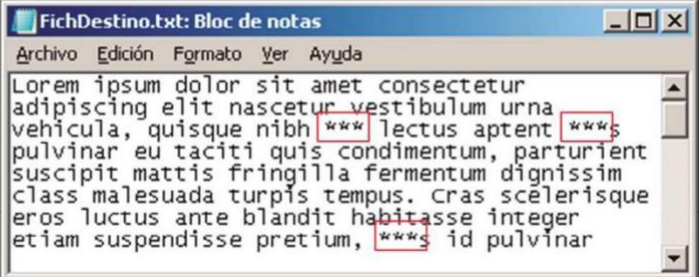
run:  
APRENDIENDO  
JAVA  
[APRENDIENDO, JAVA]

- Recuperamos el objeto “BufferedReader”, “BufferedWriter”, “InputStream” u “OutputStream” configurado correctamente con el modo de acceso.

Ejemplo:

Lee un fichero con contrnido “Lorem ipsum” y lo escribe en otro fichero donde reemplaza la secuencia de caracteres “dui” por “\*\*\*”.

```
try {
    Path input = Paths.get("C:\\ficheros\\FichOrigen.txt");
    Path output = Paths.get("C:\\ficheros\\FichDestino.txt");
    BufferedReader br = Files.newBufferedReader(input, Charset.defaultCharset());
    BufferedWriter bw = Files.newBufferedWriter(output, Charset.defaultCharset(),
        StandardOpenOption.WRITE, StandardOpenOption.CREATE,
        StandardOpenOption.TRUNCATE_EXISTING);
    String linea;
    while ((linea = br.readLine()) != null) {
        if (linea.contains("dui"))
            linea = linea.replaceAll("dui", "***");
        bw.write(linea, 0, linea.length());
        bw.newLine();
    }
    br.close();
    bw.close();
} catch (IOException ioe) {
    System.out.println(ioe);
}
```



## 2.4.2. Acceso directo

El acceso directo a un fichero binario se emplea la clase “RandomAccessFile”.

Lista de métodos:

<code>void close ()</code>	Cierra el fichero y libera los recursos asociados.
<code>int readInt ()</code>	Lee un entero con signo de 32 bits.
<code>String readUTF ()</code>	Lee una cadena de este archivo.
<code>void seek (long pos)</code>	Desplaza el puntero del archivo a la posición <i>pos</i> (desde el principio del archivo).
<code>int skipBytes(int n)</code>	Hace que se omitan los <i>n</i> primeros <i>bytes</i> .
<code>void writeDouble (double v)</code>	Convierte el argumento <i>double</i> a un <i>long</i> y lo escribe con ocho <i>bytes</i> .
<code>void writeFloat (float v)</code>	Convierte el argumento <i>float</i> a un <i>int</i> y lo escribe con cuatro <i>bytes</i> .
<code>void write (int b)</code>	Escribe el <i>byte</i> especificado en este archivo.
<code>int read ()</code>	Lee un <i>byte</i> del archivo.
<code>long length ()</code>	Devuelve la longitud del archivo.

Java hace uso de caracteres "Unicode" por esto cada carácter ocupa 2 bytes.

Ejemplo donde se escriben contenidos en un fichero binario y posteriormente se sobrescribe alguno con acceso directo a una posición.

```
File f = new File("C:\\ficheros\\Prueba.bin");
f.delete();
RandomAccessFile raf = new RandomAccessFile("C:\\ficheros\\Prueba.bin", "rw");
String s = "0123456789";
raf.writeChars(s);
raf.seek(0);
for (int i = 0; i < raf.length()/2; i++) {
    System.out.print(" " + raf.readChar());
}
System.out.println();
raf.seek(8);
raf.writeChars("");
raf.writeChars("");
raf.seek(0);
for (int i = 0; i < raf.length()/2; i++) {
    System.out.print(" " + raf.readChar());
}
System.out.println();
```

run:
0123456789
0123**6789

Se puede escribir datos de caracteres simples, como en el ejemplo, pero también se puede almacenar datos más complejos como un objeto. Se necesita conocer la longitud que ha de ser fija cuando se escriban, para así poder leerlos posteriormente. A través de la clase "StringBuffer" se puede establecer la longitud fija de una cadena.



## 2.5. Utilización de los sistema de ficheros

Lista de métodos ofrecidos por la clase `File`. Nos permiten operaciones sobre un sistema de ficheros:

<code>boolean exists()</code>	Devuelve <i>true</i> si el <i>path</i> existe, y <i>false</i> en caso contrario.
<code>long length()</code>	Devuelve la longitud del fichero.
<code>boolean renameTo (File dest)</code>	Renombra o mueve el fichero o directorio representado a <i>dest</i> .
<code>boolean isFile()</code>	Devuelve <i>true</i> si el objeto <i>File</i> desde el que se invoca representa a un fichero.
<code>boolean isDirectory()</code>	Devuelve <i>true</i> si el objeto <i>File</i> desde el que se invoca representa a un directorio.
<code>String[] list()</code>	Devuelve un listado de los nombres de ficheros contenidos en un directorio en un <i>array</i> de <i>Strings</i> .
<code>File[] listFiles()</code>	Obtiene un <i>array</i> de objetos <i>File</i> que representan a cada uno de los ficheros contenidos en un directorio.
<code>String getName()</code>	Devuelve el nombre del fichero o directorio.
<code>String getParent()</code>	Devuelve el nombre del directorio padre del fichero o directorio.
<code>File getParentFile()</code>	Devuelve el objeto <i>File</i> correspondiente al directorio padre del fichero o directorio.
<code>String getPath()</code>	Devuelve el <i>path</i> .
<code>boolean canRead()</code> <code>boolean canWrite()</code> <code>boolean canExecute()</code>	Devuelven <i>true</i> en caso de que el fichero pueda ser abierto para lectura, escritura o ejecución (respectivamente), y <i>false</i> en caso contrario.
<code>boolean setReadable</code> <code>(boolean ejecutable</code> <code>[,boolean propietario])</code> <code>boolean setWritable</code> <code>(boolean ejecutable</code> <code>[,boolean propietario])</code> <code>boolean setExecutable</code> <code>(boolean ejecutable</code> <code>[,boolean propietario])</code>	Establecen los permisos de lectura, escritura o ejecución (respectivamente) para todos los usuarios del sistema o, únicamente, para el usuario propietario.