



Unidad 10

Streams y Ficheros



Introducción

La información básica de ficheros, de la clase File y la entrada y salida en ficheros de texto podéis leerla en el siguiente enlace:

https://campusvirtual.ull.es/ocw/pluginfile.php/15444/mod_resource/content/1/Tema%205.%20Manejo%20de%20ficheros%20en%20Java.pdf

Donde también encontraréis más información sobre Java:

[Universidad La Laguna](#)

En este tema veremos más información de ficheros, pero las páginas con el fondo PUNTEADO son informativas únicamente



Introducción

- Las estructuras de datos estudiadas: Arrays, Listas, etc. son almacenadas en memoria (RAM)
 - Rápida
 - Volátil
 - Tamaño Limitado
- Para tratar **grandes volúmenes** de información y de forma **permanente**, hay que almacenarla en memoria secundaria (Disco Duro, CD-ROM, USB, etc.)
 - Los datos se agrupan en **archivos o ficheros**



Concepto de fichero

Un archivo o fichero no es más que una **serie de bytes de datos en un soporte físico permanente o volátil.**

- Un fichero se trata como una secuencia continua de datos, ya sean bytes o caracteres
- El tamaño de un fichero es variable puede crecer según se van insertando registros




Operaciones básicas

- Para trabajar con un fichero siempre habrá que realizar los tres pasos siguientes:
 - **Abrir el fichero**
 - Creando un objeto de una determinada clase de acceso a ficheros al que se le especifica el fichero que se quiere utilizar así como una serie de posibles opciones
 - **Realizar las operaciones de lectura, escritura, posicionamiento, etc.**
 - **Cerrar el fichero**
 - Si el fichero no se cierra puede que no se guarden definitivamente algunos de los datos



Clasificación de Ficheros

- Según el tipo de información almacenada
- Según la organización interna
- Según el acceso a la información



Tipos de ficheros el tipo de la información almacenada

- **Ficheros Binarios:** Almacenan secuencias de dígitos binarios (ej: ficheros que almacenan enteros, floats,...). Ilegibles para los humanos
- **Ficheros de Texto:** Almacenan caracteres alfanuméricos en un formato estándar (ASCII, Unicode, UTF8, UTF16, etc.). Pueden ser leídos y/o modificados por aplicaciones denominadas editores de texto (Ej: bloc de notas).



Tipos de fichero según la organización interna

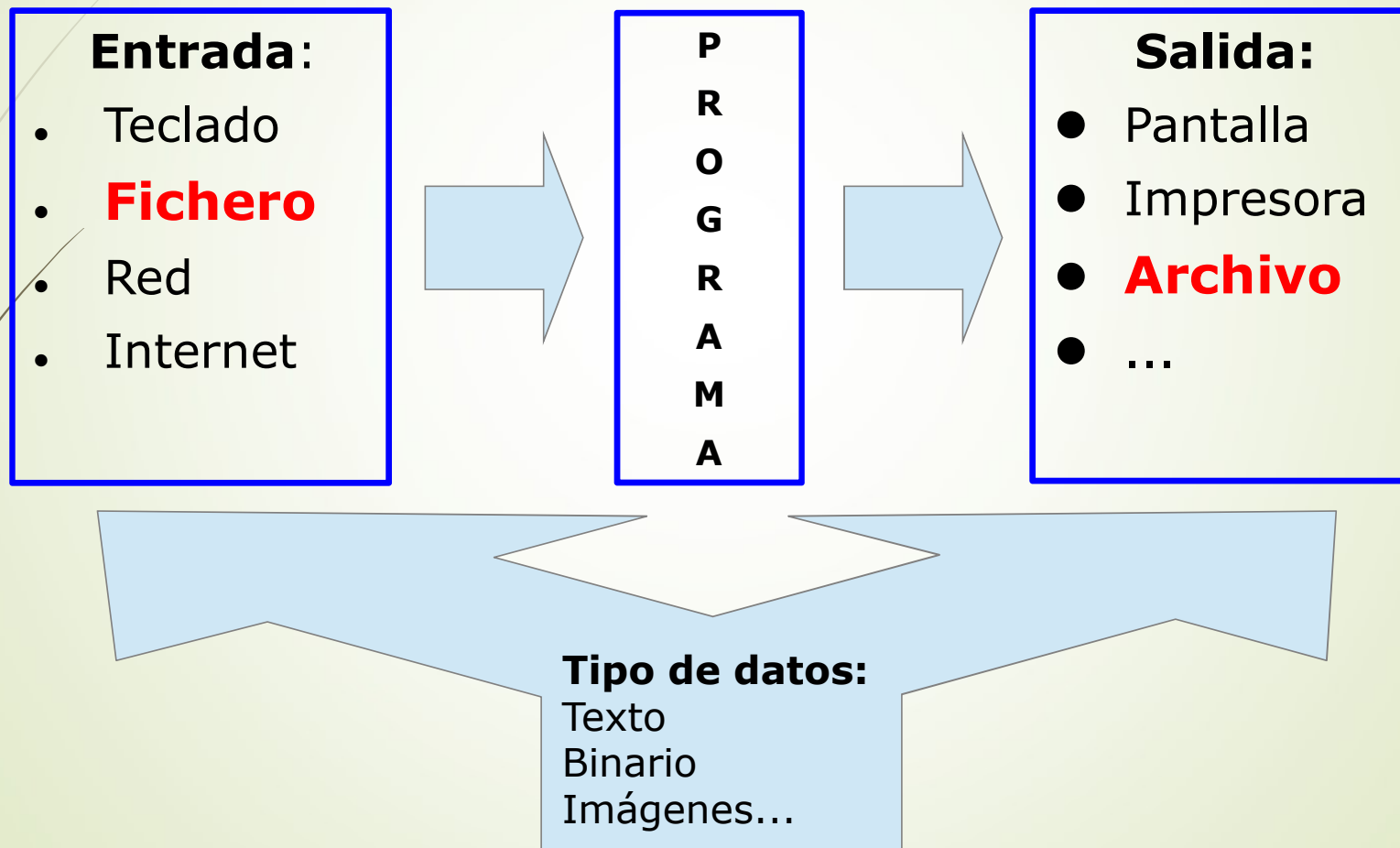
- **Secuenciales:** los registros se almacenan consecutivamente en memoria según el orden lógico en que se han ido insertando
- **Directos o Aleatorios:** El orden físico en memoria puede no coincidir con el orden de inserción
- **Indexados.** Se usan 2 ficheros:
 - Fichero de datos: Información
 - Fichero índice: Contiene la posición de cada uno de los registros en el fichero de datos



Tipos de ficheros según el acceso a la información

- **Acceso secuencial:** Para acceder a un registro es necesario pasar por todos los anteriores. Ej: Cinta de Casete
- **Acceso directo o aleatorio:** Se puede acceder a un registro sin pasar por todos los anteriores. Ej: Disco Duro

Comunicación del programa con los dispositivos I/O





Ficheros en Java



Clase File



La clase File

- La clase File no sirve para leer ni para escribir en un archivo sino que permite, entre otras operaciones:
 - Obtener el **tamaño** del archivo.
 - Obtener el **nombre** completo, incluida la ruta.
 - **Cambiar el nombre.**
 - **Eliminar** el fichero.
 - **Saber si** es un **directorio** o un **archivo**.
 - Si es un directorio, obtener **la lista de los archivos y directorios que contiene**.
 - **Crear un directorio.**

Clase File

java.io.File	
<code>boolean canRead()</code>	TRUE si el fichero se puede leer
<code>boolean canWrite()</code>	TRUE si el fichero se puede escribir
<code>boolean delete()</code>	elimina el fichero; devuelve FALSE si no puede eliminarlo
<code>boolean exists()</code>	TRUE si el fichero existe
<code>String getAbsolutePath()</code>	devuelve la ruta completa
<code>String getCanonicalPath()</code>	devuelve la ruta completa
<code>String getName()</code>	el nombre del fichero, sin ruta
<code>String getParent()</code>	la ruta del 'padre' o directorio en el que se encuentra
<code>File getParentFile()</code>	el 'padre' o directorio en el que se encuentra
<code>boolean isDirectory()</code>	TRUE si es un directorio
<code>boolean isFile()</code>	TRUE si no es un directorio
<code>long length()</code>	tamaño del fichero en bytes
<code>String[] list()</code>	si se trata de un directorio, un array con los nombres de los ficheros que contiene
<code>File[] listFiles()</code>	si se trata de un directorio, un array con los ficheros que contiene
<code>boolean mkdir()</code>	crea el directorio si no existe
<code>boolean mkdirs()</code>	crea todos los directorios que haga falta, si no existen
<code>boolean renameTo(File nuevo)</code>	cambio de nombre

Un ejemplo

```
String nombre;  
Scanner teclado = new Scanner(System.in);  
System.out.println("Nombre");  
nombre = teclado.nextLine();  
File f = new File(nombre);  
System.out.println("Nombre:" + f.getName());  
System.out.println("ruta = " + f.getPath());  
  
if (f.isFile()) {  
    System.out.println("tamaño: " + f.length());  
} else { //si es directorio mostramos el contenido  
    File[] archivos = f.listFiles();  
    for (File arc : archivos) {  
        System.out.println(arc.getName());  
    }  
}
```

Creando la instancia

Obteniendo el nombre

Obteniendo
La ruta

¿es fichero?

Obteniendo
La longitud

Obteniendo
La lista de
ficheros
y directorios

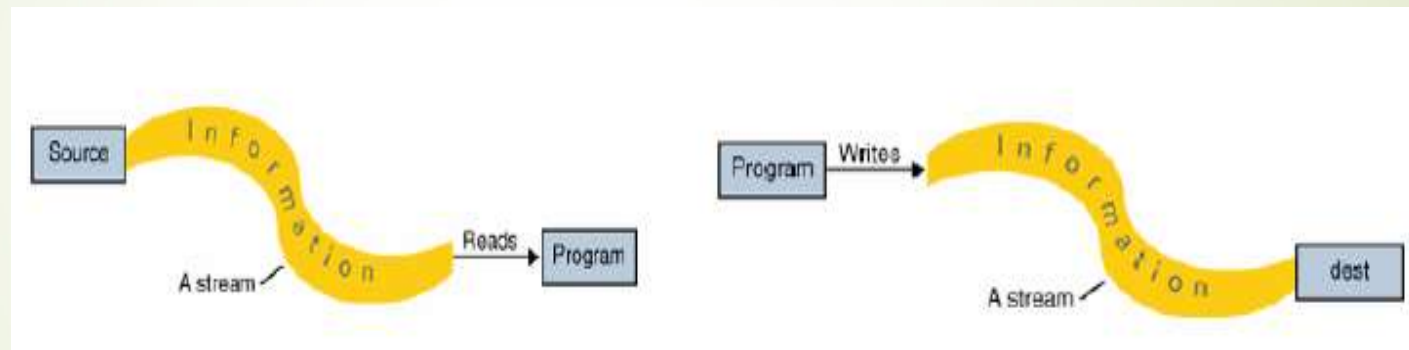


Ficheros en JAVA .Introducción

- En Java la comunicación con los dispositivos de E/S es especialmente conflictiva por el hecho de trabajar con una máquina virtual y no sobre la máquina real, ya que la comunicación con estos dispositivos depende del sistema.
- Esta comunicación se abstraerá utilizando los flujos de datos o Streams
- Los programas se comunican con los dispositivos externos a través de **flujos de datos (STREAMS)**
- En este tema nos centraremos en los **FICHEROS** como dispositivos de E/S

Ficheros en JAVA

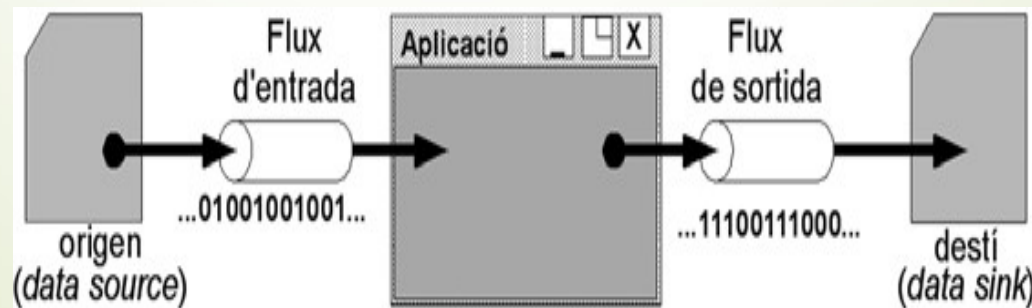
- Como hemos dicho antes, a los ficheros, en Java, se accede a través de **Streams**: Canales, flujos de datos o “tuberías”. **Entrada** (InputStream) o **Salida** (OutputStream).



- Al usar ficheros, debemos forzosamente controlar las excepciones que se puedan producir, como que no exista el fichero o cualquier otra excepción

Flujos de información o Streams

- Un Flujo o Stream es un objeto que hace de intermediario entre el programa y el origen o destino de la información
- El programa lee o escribe en el Stream sin importarle desde dónde viene la información o a dónde va
- Desde el punto de vista de la aplicación se pueden generar dos tipos de flujos:
 - **De entrada:** Sirven para leer datos desde un origen para ser procesados
 - **De salida:** Son los responsables de enviar los datos a un destino



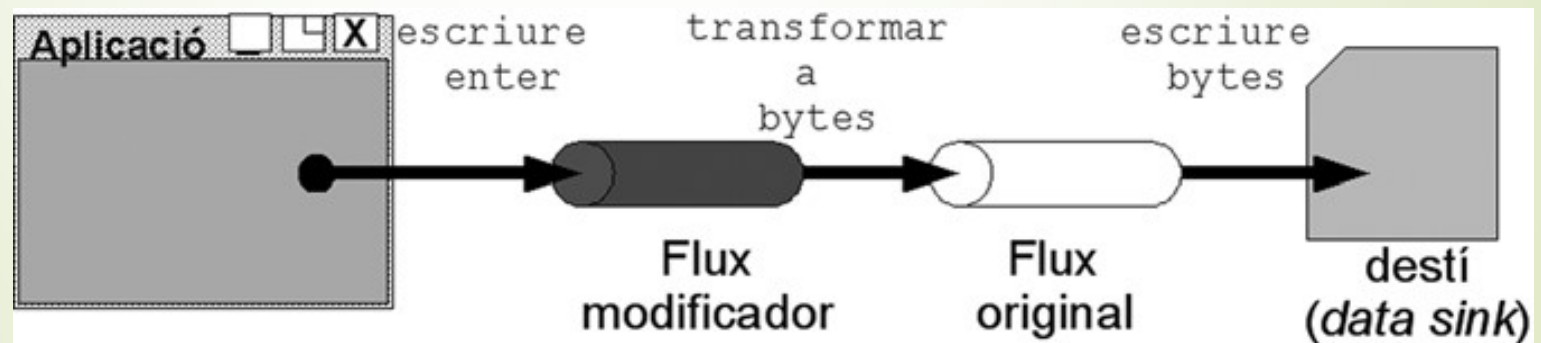


Flujos de información o Streams

- Los flujos los datos siempre se transmiten y procesan secuencialmente
- Java ofrece dos tipos de flujos:
 - De datos o binarios
 - Los datos procesados se interpretan como bytes
 - Opera con el tipo primitivo **byte**
 - De caracter
 - Los datos procesados se interpretan como texto
 - Opera con el tipo primitivo **char**

Filtros o modificadores de flujo

- Una clase modificadora de un flujo altera su funcionamiento por defecto y proporciona métodos adicionales que permiten el pre-procesado de datos complejos antes de escribirlas o leerlas del flujo
- Este pre-proceso se realiza de manera transparente al desarrollador





Clases de Java

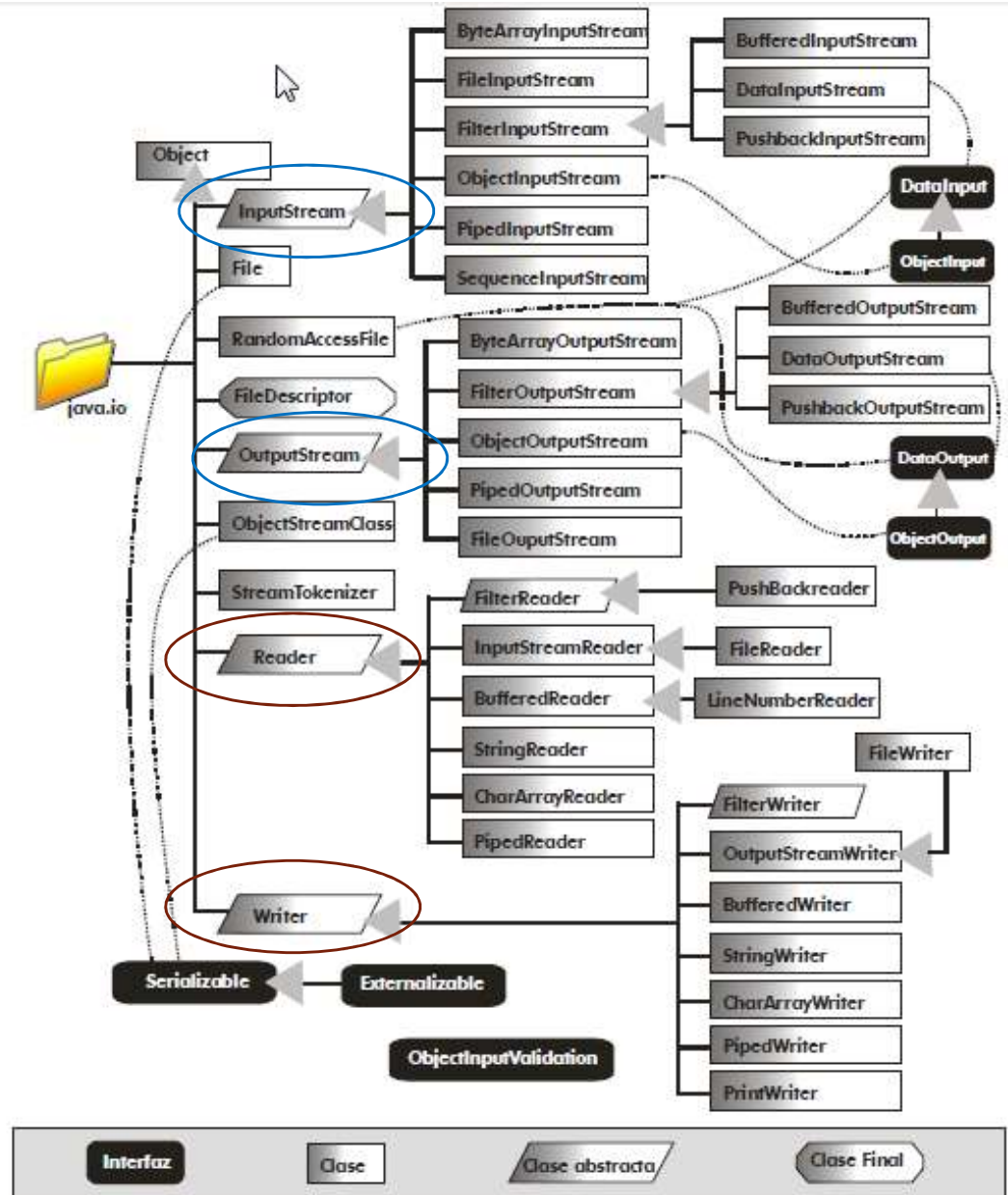
- ▶ Las clases vinculadas a la entrada/salida se encuentran definidas en el **paquete java.io**
- ▶ La excepción vinculada a errores de E/S definida en el mismo paquete es **IOException**

Para trabajar con ficheros se habrá de importar las clases a utilizar
Y las operaciones sobre streams se harán siempre dentro de un **try-catch**

Clases de java.io

Tenemos unas clases abstractas básicas para los tipos de flujos que hemos visto:

- **De texto:**
 - Entrada: Reader
 - Salida: Writer
- **Binario:**
 - Entrada: InputStream
 - Salida: OutputStream



Algunas clases fundamentales de java.io

	bytes	Caracteres
lectura	<u>InputStream</u> FilterInputStream BufferedInputStream DataInputStream FileInputStream ObjectInputStream	<u>Reader</u> BufferedReader InputStreamReader FileReader
escritura	<u>OutputStream</u> FilterOutputStream BufferedOutputStream DataOutputStream PrintStream FileOutputStream ObjectOutputStream	<u>Writer</u> BufferedWriter OutputStreamWriter FileWriter



Métodos básicos de los flujos de entrada

- Métodos básicos de **Reader**:
 - **int read()**
 - int read(char cbuf[])
 - int read(char cbuf[], int offset, int length)
- Métodos básicos de **InputStream**:
 - **int read()**
 - int read(byte cbuf[])
 - int read(byte cbuf[], int offset, int length)



Métodos básicos de los flujos de salida

- Métodos básicos de **Writer**:
 - **int write(int c)**
 - int write(char cbuf[])
 - int write(char cbuf[], int offset, int length)
- Métodos básicos de **OutputStream**:
 - **int write(int c)**
 - int write(byte cbuf[])
 - int write(byte cbuf[], int offset, int length)
- Los Streams se abren automáticamente al crearlos, pero es necesario cerrarlos explícitamente llamando al método **close()** cuando se dejan de usar

Reader (abstracto) y FileReader

java.io.Reader	
<code>void close()</code>	cierra el fichero
<code>int read()</code>	lee un carácter devuelve el carácter leído pasado a entero devuelve -1 si el fichero se ha acabado
<code>int read(char[] chars)</code>	lee un número 'n' de caracteres igual o menor que la longitud del array 'chars' devuelve el número de caracteres leídos; los caracteres leídos están en las posiciones [0 .. n-1] del array 'chars' devuelve -1 si el fichero se ha acabado
<code>int read(char[] chars, int start, int n)</code>	lee un número 'n' de caracteres devuelve el número de caracteres leídos; los caracteres leídos están en las posiciones [start .. start+n-1] del array 'chars' devuelve -1 si el fichero se ha acabado

- La clase derivada más habitual

java.io.FileReader	
<code>FileReader(File file)</code>	constructor
<code>FileReader(String nombre)</code>	constructor

InputStream(abstrcto) y FileInputStream

java.io.InputStream	
<code>int available()</code>	una estimación del número de bytes que quedan por leer
<code>void close()</code>	cierra el fichero
<code>int read()</code>	lee un byte devuelve el byte leído pasado a entero devuelve -1 si el fichero se ha acabado
<code>int read(byte[] bytes)</code>	lee un número 'n' de bytes igual o menor que la longitud del array 'bytes' devuelve el número de bytes leídos; los bytes leídos están en las posiciones [0 .. n-1] del array 'bytes' devuelve -1 si el fichero se ha acabado
<code>int read(byte[] bytes, int start, int n)</code>	lee un número 'n' de bytes devuelve el número de bytes leídos; los bytes leídos están en las posiciones [start .. start+n-1] del array 'bytes' devuelve -1 si el fichero se ha acabado

- La clase derivada más habitual

java.io.FileInputStream	
<code>FileInputStream(File file)</code>	constructor
<code>FileInputStream(String nombre)</code>	constructor

Writer (abstracto) y FileWriter

java.io.Writer	
<code>Writer append(char c)</code>	añade un carácter al final del fichero
<code>void close()</code>	cierra el fichero, asegurando que todo queda bien escrito en el fichero en disco
<code>void flush()</code>	asegura que todos los caracteres quedan bien escritos en el disco, sin cerrar el fichero
<code>void write(char[] chars)</code>	escribe en el fichero el array de caracteres
<code>void write(char[] chars, int start, int n)</code>	escribe 'n' caracteres en el fichero, concretamente, los del array 'chars', empezando en la posición 'start'.
<code>void write(String s)</code>	escribe en el fichero la cadena 's'
<code>void write(String s, int start, int n)</code>	escribe 'n' caracteres en el fichero, concretamente, los de la cadena 's', empezando en la posición 'start'.

- La clase derivada más habitual

java.io.FileWriter	
<code>FileWriter(File file)</code>	constructor
<code>FileWriter (File file, boolean append)</code>	constructor: añade al final
<code>FileWriter (String nombre)</code>	constructor
<code>FileWriter (String nombre, boolean append)</code>	constructor: añade al final

OutputStream (abstracto) y FileOutputStream

java.io.OutputStream	
<code>void close()</code>	cierra el fichero, asegurando que todo queda bien escrito en el fichero en disco
<code>void write(byte[] b)</code>	escribe en el fichero el array de bytes
<code>void write(byte[] b, int start, int n)</code>	escribe 'n' bytes en el fichero, concretamente, los del array 'b', empezando en la posición 'start'.

- La clase derivada más habitual

java.io.FileOutputStream	
<code>FileOutputStream(File file)</code>	constructor
<code>FileOutputStream(File file, boolean append)</code>	constructor: añade al final
<code>FileOutputStream(String nombre)</code>	constructor
<code>FileOutputStream(String nombre, boolean append)</code>	constructor: añade al final



Excepciones

- Recordad que al trabajar con ficheros pueden darse varias excepciones:
 - Que no exista el fichero
 - Que esté ocupado
 - Que hayamos llegado al final
- Es imprescindible tratarlas en **un try{}catch{} al que le añadiremos un finally para que SIEMPRE se cierre el fichero al acabar**
- Fijaos en el siguiente ejemplo, en el bloque try-catch

Ej:Escribir en un fichero de texto, carácter a carácter

Se ha añadido el bloque **finally** para cerrar el fichero si ha habido algún error que lo haya dejado abierto

```
public class EjemploEscribirLetras {
    public static void main(String[] args) {
        String nombre = "fichero letrasPW.txt";
        FileWriter fw=null;
        try {
            fw = new FileWriter(nombre) ;//Si no existe , lo crea
            for(char c='A';c<='Z';c++)
                fw.write(c); //Escribe cada caracter
        } catch (Exception e) {
            System.out.println(e.getMessage());
        } finally{
            try{
                if(fw!=null)
                    fw.close();
            } catch (Exception e){
                System.out.println(e.getMessage());
            }
        }
    } //fin del main
}
```

Ej: Leer en un fichero de texto, carácter a carácter

```
public class EjemploLeerLetras {  
    public static void main(String[] args) {  
        String nombre = "fichero letras.txt";  
        int car;  
        //Declara una variable FileReader  
        FileReader fr = null;  
        try {  
            fr = new FileReader(nombre); //Abre el fichero  
            car = fr.read(); //Lee el primer caracter  
            while (car != -1) { //Si existe, entra en el bucle  
                System.out.print((char) car + "-"); //Escrib el carácter  
                car = fr.read(); //Lee otro caracter }  
        } catch (FileNotFoundException e) {  
            System.out.println("No existe el fichero");  
        } catch (Exception e) {  
            System.out.println(e.getMessage());  
        } finally {  
            try {  
                if (fr != null) {  
                    fr.close(); //En cualquier caso cierra el fichero  
                }  
            } catch (Exception e) {  
                System.out.println(e.getMessage());  
            }  
        }  
    }  
}
```




Filtros

- Son las clases que representan un flujo de datos (de lectura o escritura) cuyo origen o destino de los datos es otro flujo.
- Los filtros se conectan a otros flujos que ya existen **para transformar los datos**, proporcionando métodos de lectura o escritura más apropiados al programador.

Filtros: BufferedReader - BufferedWriter

- Las clases **FileReader**, **FileInputStream**, **FileWriter** o **FileOutputStream** hacen las operaciones de lectura o escritura directamente sobre el dispositivo de almacenamiento
 - si la aplicación hace estas operaciones de forma muy repetida se ralentiza la ejecución de la aplicación
- Para mejorar esta situación se pueden utilizar junto con las clases anteriores las clases **Buffered***
- La palabra buffered hace referencia a la capacidad de almacenamiento temporal en la lectura y escritura:
 - En las operaciones de lectura se almacenan más datos de los que realmente se necesitan en un momento determinado, de forma que en una siguiente operación de lectura es posible que esos datos ya estén en memoria y no sea necesario acceder de nuevo al dispositivo de almacenamiento.
 - En las operaciones de escritura los datos se van guardando en memoria y no se vuelcan al dispositivo hasta que hay una cantidad suficiente de datos para mejorar el rendimiento



Filtros: BufferedReader - BufferedWriter

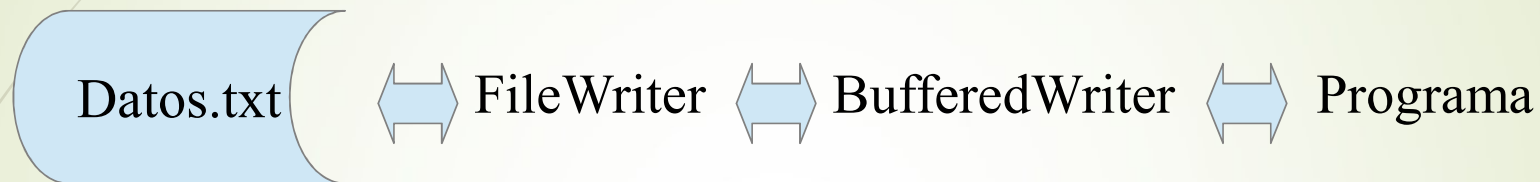
➤ La clase **BufferedReader**

- Se construye a partir de un objeto de la clase Reader(o FileReader)
 - **public BufferedReader(Reader in)**
- **readLine()**: Permite leer caracteres hasta la presencia de null o del salto de línea

➤ La clase **BufferedWriter**

- Se construye a partir de un objeto de la clase Writer(o FileWriter)
 - **public BufferedWriter(Writer out)**
- **writeLine()**: métodos para la escritura eficiente de caracteres
- Aporta el método **newLine()**. No todas las plataformas admiten '\n'

Escribir líneas completas



- El filtro de la clase `BufferedWriter` se conecta a flujo de la clase `FileWriter`. La clase `BufferedWriter` dispone del método **`writeLine()`** para escribir una línea completa de una sola vez.
- Cada vez que se llama al método `writeLine()` de la clase `BufferedWriter` este método, por la forma en que están enlazados, lo que hace es llamar las veces que precise al método `write()` de la clase `FileWriter` hasta escribir una línea completa.

```

FileWriter archivo;
BufferedWriter filtro;
String linea;
Scanner teclado = new Scanner(System.in);
try {
    archivo = new FileWriter(nombre);
    filtro = new BufferedWriter(archivo);
    System.out.println("Introduce texto");
    linea = teclado.nextLine();
    while (linea.length() != 0) {
        linea+="\n";
        filtro.write(linea);
        linea = teclado.nextLine();
    }
    filtro.close();
} catch (IOException e) {
    System.out.println("problemas fichero");
}

```

Sería equivalente:
 filtro.write(linea);
 Filtro.newLine();
 En lugar de añadir
 \n

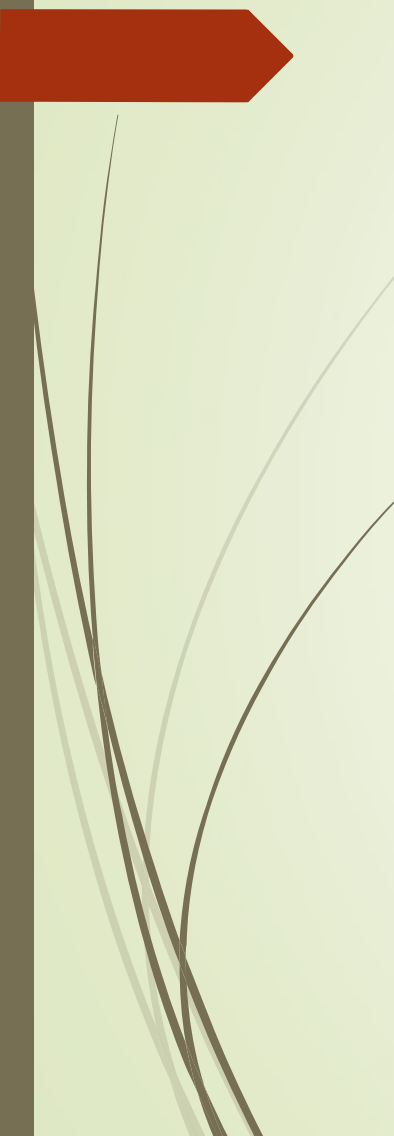
Falta el finally

Leer líneas completas

Datos.txt



- El filtro de la clase `BufferedReader` se conecta a flujo de la clase `FileReader`. La clase `BufferedReader` dispone del método **`readLine()`** para leer una línea completa de una sola vez.
- Cada vez que se llama al método `readLine()` de la clase `BufferedReader` este método, por la forma en que están enlazados, lo que hace es llamar las veces que precise al método `read()` de la clase `FileReader` hasta conseguir una línea completa.



```
FileReader archivo;
BufferedReader filtro;
String linea;
Scanner teclado = new Scanner(System.in);
try {
    archivo = new FileReader(nombre);
    filtro = new BufferedReader(archivo);
    linea = filtro.readLine();
    while (linea!=null) {
        System.out.println(linea);
        linea = filtro.readLine();
    }
    filtro.close();
} catch (IOException e) {
    System.out.println("problemas fichero");
}
```



Modo de apertura de ficheros

- En Java por defecto las clases que manejan ficheros si el fichero NO EXISTE lo crean y escriben y si YA EXISTE lo machacan.
- Para abrir un fichero para añadir datos, se añade **true** como 2º parámetro. Por ejemplo:

```
escribir = new FileWriter(nombre,true);
```

```
archivo = new FileWriter(nombre,true);  
filtro = new BufferedWriter(archivo);51
```



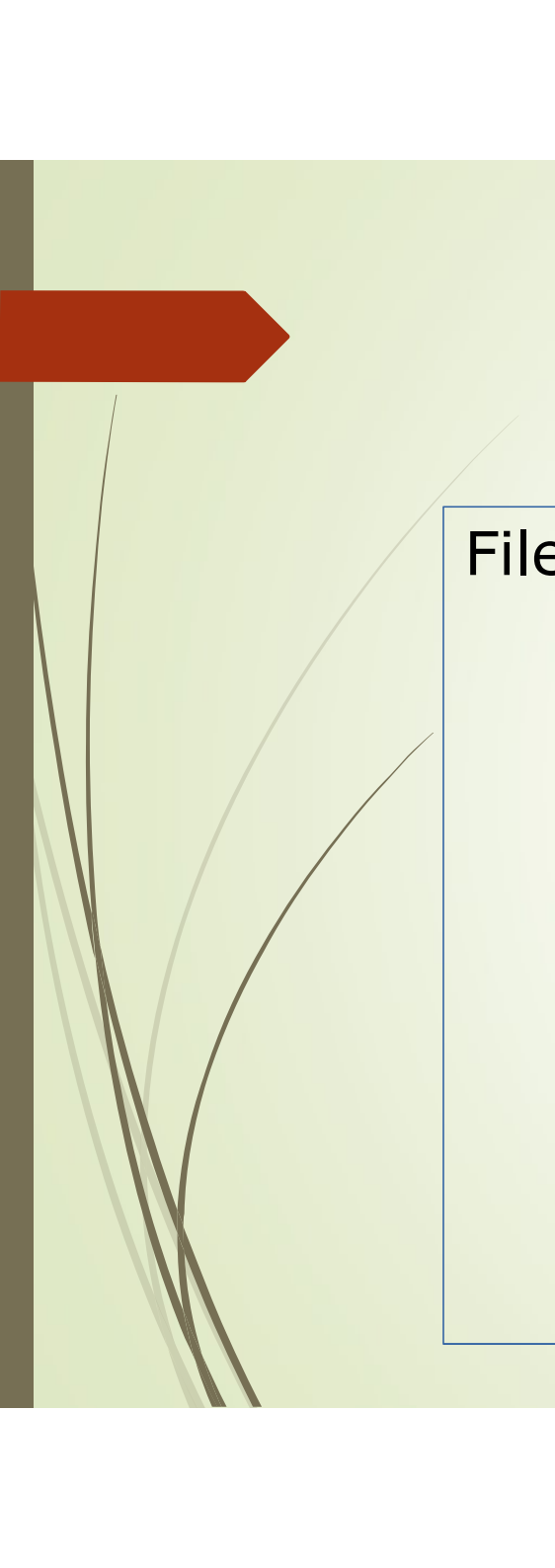

Ficheros Binarios



Ficheros Binarios.

Escribiendo datos primitivos

- Veamos como guardar y leer datos en un fichero binario utilizando como filtro de `FileOutputStream` y `FileInputStream` las clases **`DataOutputStream`** y **`DataInputStream`**
- Estos FILTROS, proporcionan métodos para la lectura de tipos primitivos independientemente de la máquina



FileInputStream

read

DataInputStream

readByte Lee tipo byte.

readShort Lee tipo short.

readChar Lee tipo char.

readInt Lee tipo int.

readLong Lee tipo long ■

readFloat Lee tipo float.

readDouble Lee tipo double.

ReadUTF lee string



FileOutputStream

write

DataOutputStream

writeByte	Escribe tipo byte.
writeShort	Escribe tipo short.
writeChar	Escribe tipo char.
writeInt	Escribe tipo int.
writeLong	Escribe tipo long.
writeFloat	Escribe tipo float.
writeDouble	Escribe tipo double.
WriteUTF	Escribe string



Ej.:

- Veamos por ejemplo cómo escribir una lista de la compra, escribiendo y leyendo cada dato con su tipo correspondiente.
- Para crear una lista de la compra, registramos para cada elemento los siguientes datos:
 - Nomprod
 - Cant
 - Precio
- Guardaremos los datos como String, int y float respectivamente

```

void escribir() throws FileNotFoundException, IOException {
    String nombrep;
    int cant;
    float precio;
    Scanner teclado = new Scanner(System.in);
    FileOutputStream salida;
    DataOutputStream dos;
    salida = new FileOutputStream(nombre);
    dos = new DataOutputStream(salida);
    System.out.println("Producto:");
    nombrep = teclado.nextLine();
    System.out.println("Cantidad:");
    cant = teclado.nextInt();
    System.out.println("precio:");
    precio = teclado.nextFloat();
    dos.writeUTF(nombrep);
    dos.writeInt(cant);
    dos.writeFloat(precio);
    dos.close();
}

```


Abrir

en el FileOutputStream añadiríamos true para añadir

Pedimos datos

Los escribimos

Cerramos



```
void leer() throws FileNotFoundException, IOException {  
    FileInputStream entrada = new FileInputStream(nombre);  
    DataInputStream dis = new DataInputStream(entrada);  
    String nombrep;  
    int cant;  
    float precio;  
  
    while (dis.available() != 0) {  
        nombrep = dis.readUTF();  
        cant = dis.readInt();  
        precio = dis.readFloat();  
        System.out.println(nombrep + " " + cant + " " + precio);  
    }  
    dis.close();  
}
```

Abrir

Mientras hay
datos disponibles

Los leemos

Cerramos



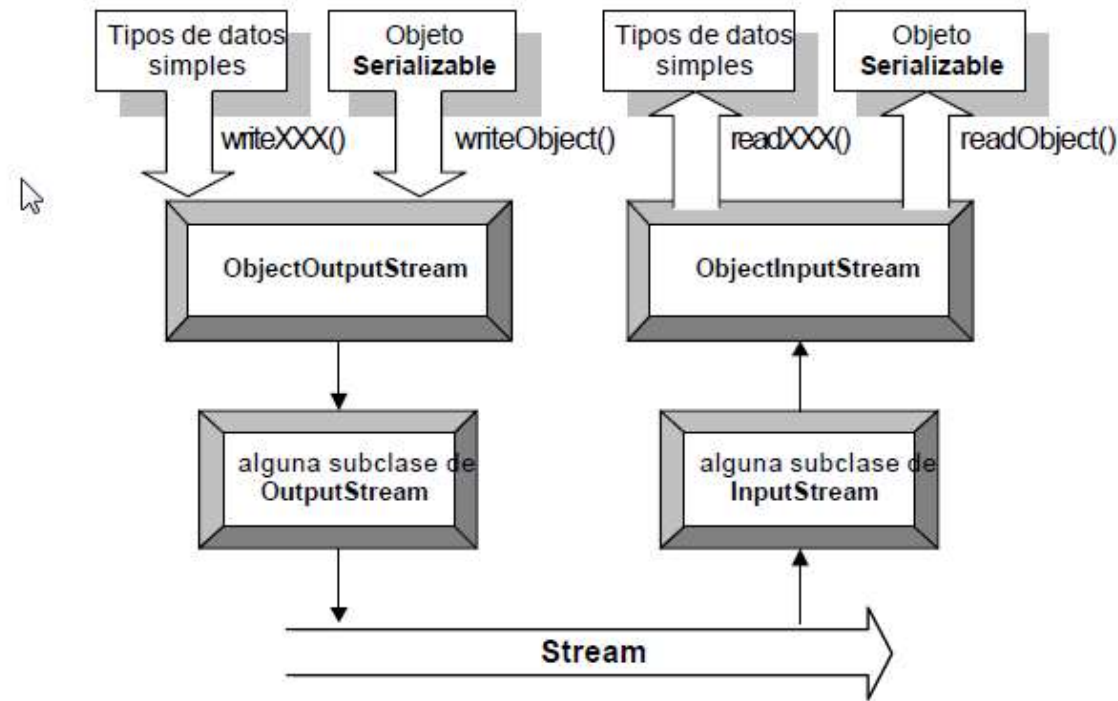
Objetos



Serialización de objetos

- Si deseamos guardar permanentemente el estado de un objeto podemos utilizar los streams estudiados para ir almacenando todos los valores de los atributos como valores char, int, String, etc.
 - Esto puede ser muy molesto y complicado
- La “**serialización**” de objetos es una forma más cómoda de enviar objetos a través de un stream como una secuencia de bytes para ser almacenados en disco y también para reconstruir objetos a partir de streams de entrada de bytes
- La serialización consiste en la transformación de un objeto Java en una secuencia de bytes para ser enviados a un stream

Esquema de funcionamiento de la serialización





Serialización de objetos

- Para enviar y recibir objetos serializados a través de un *stream se utilizan las clases*:
 - Para la salida: **ObjectOutputStream**
 - Pasando como parámetro en su constructor un objeto de la clase `FileOutputStream` los objetos serializados serán dirigidos a un fichero
 - Para la entrada **ObjectInputStream**
 - Pasando como parámetro en su constructor un objeto de la clase `FileInputStream` los objetos se deserializan después de obtener los bytes de un fichero
 - Sólo tiene sentido si los datos almacenados son objetos

Objetos serializables

- Sólo los objetos de clases que implementen la interface java.io.Serializable pueden ser serializados
 - La interface **Serializable** no posee ningún método
 - Sólo sirve para “marcar” las clases que pueden ser serializadas

```
import java.io.Serializable;

public class Persona implements Serializable {
    // Esta clase debe ser Serializable para
    // poder ser escrita en un stream de objetos
    private String nombre;
    private int edad;

    public Persona(String s, int i) {
        nombre = s;
        edad = i;
    }

    public String toString() {
        return nombre + ":" + edad;
    }
}
```

Simplemente
escribiremos:
**implements
Serializable** en la
cabecera de la
clase


Serialización. Límites

- Las variables marcadas como static, no pueden ser serializadas.
- Si un objeto contiene una referencia a una instancia de una clase que no es serializable. Esto produciría la excepción `NotSerializableException`. Para evitarlo, debemos marcar esa instancia como transient, y no se serializarían.



Un ejemplo: Fichero de objetos persona

- Vamos a crear una pequeña aplicación que almacene en un fichero objetos de tipo persona.
- Para ello primero crearemos la clase Persona y la serializaremos



```
public class Persona implements Serializable{
    String nombre;
    String apellidos;

    public String toString() {
        return nombre + " " + apellidos;
    }

    public Persona(String nombre, String apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
}
```

Guardando datos

Creando el filtro

```
Persona p;  
try {  
    fichero = new File("personas.dat");  
    archivo = new FileOutputStream(fichero, true);  
    oos = new ObjectOutputStream(archivo);  
    p = datosPersona();  
    oos.writeObject(p);  
    System.out.println("Guardando..." + p);  
} catch (IOException e) {  
    System.out.println("Problemas con el fichero");  
} catch (Exception e) {  
    System.out.println(e.getMessage());  
} finally {  
    try {  
        if (oos != null) {  
            oos.close(); //En cualquier caso cierra el fichero  
        }  
    } catch (Exception e) {  
        System.out.println(e.getMessage());  
    }  
}
```

Guardando el objeto

Cerrando el filtro

Leyendo los datos

```
Persona p;  
try {  
    archivo = new FileInputStream("personas.dat");  
    ois = new ObjectInputStream(archivo);  
    aux = ois.readObject();  
    while (aux != null) {  
        if (aux instanceof Persona) {  
            p = (Persona) aux;  
            System.out.println("Leyendo..." + p);  
        }  
        aux = ois.readObject();  
    }  
} catch (EOFException e) {  
    System.out.println("Fin del fichero");  
} catch (IOException e) {  
    System.out.println("Problemas cooooooon el fichero" + e.getMessage());  
} catch (Exception e) {  
    System.out.println(e.getMessage());  
} finally {  
    try {  
        if (ois != null) {  
            ois.close(); //En cualquier caso cierra el fichero  
        }  
    } catch (Exception e) {  
        System.out.println(e.getMessage());  
    }  
}
```

Creando el filtro

Leemos el objeto

Si hemos leído bien

Leemos el siguiente

Cerrando el filtro



Notas

- El método **readObject()** lee cualquier objeto del flujo de entrada, devuelve el objeto como tipo **Object**. Entonces, es necesario convertir Object al tipo del objeto que se espera leer. Por ejemplo, si el archivo es de objetos Racional:
 - `rac = (Racional) flujo.readObject();`
- La lectura de archivos con diversos tipos de objetos necesita una estructura de selección para conocer el tipo de objeto leído. El operador **instanceof** es útil para esta selección



Error al añadir datos a un fichero ya existente

- Al intentar añadir personas al fichero ya existente, le añade antes del nuevo registro una cabecera que luego provoca un error al leer.
- Para solucionarlo redefinimos la clase `ObjectOutputStream` (`MiObjectOutputStream`) modificando el método `writeStreamHeader`, para que no la escriba

Redefiniendo ObjectOutputStream

```
class MiObjectOutputStream extends ObjectOutputStream{
    public MiObjectOutputStream(OutputStream out) throws IOException
    {
        super(out);
    }

    /** Constructor sin parámetros */
    protected MiObjectOutputStream() throws IOException, SecurityException
    {
        super();
    }

    /** Redefinición del método de escribir la cabecera para que no haga nada. */
    protected void writeStreamHeader() throws IOException
    {
    }
}
```

Modificando el método guardar

```
File f = new File(fichero);
try {
    if (f.exists()) { //Si ya existe escribo sin cabecera con miobjectoutputstream
        MiObjectOutputStream oos = new MiObjectOutputStream(new FileOutputStream(fichero, true));

        Persona p = new Persona();
        oos.writeObject(p);
        System.out.println("Guardado.... " + p.toString());
        oos.close();
    } else { //sino le añadop ña cabecera delante con el objectoutputstream
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fichero, true));
        Persona p = new Persona();
        oos.writeObject(p);
        System.out.println("Guardado.... " + p.toString());
        oos.close();
    }
} catch (Exception e) {
    System.out.println("Error");
}
```




Ficheros de acceso aleatorio

- Permiten el acceso para leer y o escribir en cualquier parte de un fichero.

-



El acceso aleatorio en JAVA

- La clase `RandomAccessFile` permite abrir un archivo como de lectura, o de lectura y escritura simultáneamente.
 - Para lectura del archivo (modo "r"), dispone de métodos para leer elementos de cualquier tipo primitivo:
 - `readInt()`
 - `readLong()`
 - `readDouble()`
 - `readLine(), etc.`
 - Para lectura y escritura (modo "rw") dispone también de métodos de escritura para escribir los tipos de datos:
 - `writeInt()`
 - `writeDouble(),`
 - `writeLong()`
 - `write(int b)` ,Escribe en el fichero el byte indicado por parámetro
 - `writeBytes(String s)` , Escribe en el fichero la cadena de caracteres indicada por parámetro. No incluye salto de línea, por lo que hay que incluirlo en la cadena de caracteres si se desea añadir:
`writeBytes(cadena+"\n")`



Álgunos métodos de RandomAccessFile

- Hay un método read para cada tipo de dato básico: readChar, readInt, readDouble, readBoolean, etc...
- `public void write(int b)`
 - Escribe en el fichero el byte indicado por parámetro
- `public final void writeBytes(String s)`
 - Escribe en el fichero la cadena de caracteres indicada por parámetro
 - No incluye salto de línea, por lo que hay que incluirlo en la cadena de caracteres si se desea añadir: `writeBytes(cadena+"\n")`
- También existe un método write para cada tipo de dato básico: writeChar, writeInt, writeDouble, writeBoolean, etc..



Métodos de acceso aleatorio de la clase

- Los métodos específicos para el acceso aleatorio son los que permiten acceder a un lugar concreto dentro del archivo y conocer el punto del mismo en el que se va a realizar la operación de lectura y/o escritura:
 - `getPosition()`: Devuelve la posición actual donde se va a realizar la operación de lectura o escritura. Devuelve la posición, contando en bytes donde se encuentra actualmente el cursor del archivo.
 - `seek()`: Sitúa la posición de la próxima operación de lectura o escritura en el byte especificado.
 - `length()`: Devuelve el tamaño actual del archivo.
 - `void skipBytes (int t)`: Posicionamiento relativo saltando n bytes

Un ejemplo: convirtiendo las 'b' en 'B'

```
char c;  
boolean finArchivo = false;  
RandomAccessFile archivo = null;  
try {  
    archivo = new RandomAccessFile("prueba.txt", "rw");  
    System.out.println("El tamaño es: " + archivo.length());  
    do {  
        try {  
            c = (char) archivo.readByte();  
            if (c == 'b') {  
                archivo.seek(archivo.getFilePointer() - 1);  
                archivo.writeByte(Character.toUpperCase(c));  
            }  
        } catch (EOFException e) {  
            finArchivo = true;  
            archivo.close();  
            System.out.println("Convertidas las b a mayúsculas.");  
        }  
    } while (!finArchivo);  
} catch (FileNotFoundException e) {  
    System.out.println("No se encontró el archivo.");  
} catch (Exception ex) {  
    System.out.println("Problemas con el archivo.");  
}
```

Abrimos para
leer y escribir

Lee un caracter

Si en una 'b',
Se situa en
el anterior
Y lo escribe
en mayúsculas