

# UT9 Ficheros en Java

1. Conceptos Básicos de Ficheros

2. Operaciones sobre ficheros

3. Tipos de fichero

4. Conceptos Básicos de Entrada/Salida

5. Ficheros de Texto

6. Ficheros Binarios

7. Ficheros de Acceso Aleatorio

8. Serialización

## 1. Conceptos Básicos de Ficheros

Hasta ahora todos los datos que creábamos en nuestros programas solamente existían durante la ejecución de los mismos. Cuando salíamos del programa, todo lo que habíamos generado se perdía.

A veces nos interesa que la vida de los datos vaya más allá de la de los programas que los generaron. Es decir, que al salir de un programa los datos generados queden guardados en algún lugar que permita su recuperación desde el mismo u otros programas. Esto que quiere decir que queremos que los datos sean *persistentes*.

En este tema veremos el uso básico de **archivos/ficheros** en Java para conseguir persistencia de datos. Para ello, presentaremos conceptos básicos sobre archivos y algunas de las clases de la biblioteca estándar de Java para su creación y manipulación.

Además, el uso de esas bibliotecas nos obligará a introducir algunos conceptos “avanzados” de programación en Java para transformar nuestros datos a vectores de bytes.

Cuando se desea guardar información más allá del tiempo de ejecución de un programa, lo habitual es organizar esa información en uno o varios **ficheros** almacenados en algún soporte de almacenamiento persistente.

Otras posibilidades como el uso de **bases de datos**, también utilizan archivos como soporte para el almacenamiento de la información.

Desde el punto de vista de más bajo nivel, podemos definir un archivo (o fichero) como:

Un conjunto de bits almacenados en un dispositivo, y accesible a través de un camino de acceso (pathname) que lo identifica. Es decir, un conjunto de 0s y 1s que reside fuera de la memoria del ordenador, ya sea en el disco duro, un pendrive, un CD...

Esa versión de bajo nivel, si bien es completamente cierta, desde el punto de vista de la programación de aplicaciones, es demasiado simple. Por ello definiremos varios criterios para distinguir diversas subcategorías de archivos.

Estos tipos de archivos se diferenciarán desde el punto de vista de la programación: Cada uno de ellos proporcionará diferentes funcionalidades (métodos) para su manipulación.

### 1.1 Objeto file

*https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/File.html*

En el paquete java.io se encuentra la clase **File** pensada para poder realizar operaciones de información sobre archivos. No proporciona métodos de acceso a los archivos, sino operaciones **a nivel de sistema** de archivos (listado de archivos, crear carpetas, borrar ficheros, cambiar nombre,...).

Aclaro que quiero decir con “No proporciona métodos de acceso a los archivos”: no sirven para crear un fichero y poder acceder a él para escribir o leer su contenido, sino que sirven para ver información del fichero desde el punto de vista del sistema operativo.

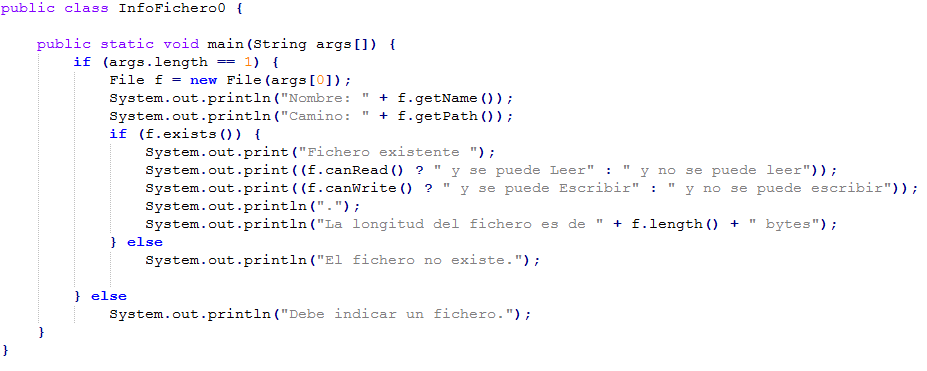
Vamos a realizar una serie de ejercicios sobre este concepto.

Un **objeto File** puederepresentar un archivo o un directorio y sirve para obtener información (permisos, tamaño,…) y también para navegar por la estructura de archivos.

Su *constructor* puede recibir un único parámetro: una cadena que representa una ruta (path) en el sistema de archivos. O también puede recibir, opcionalmente, un segundo parámetro con una ruta segunda que se define a partir de la posición de la primera.

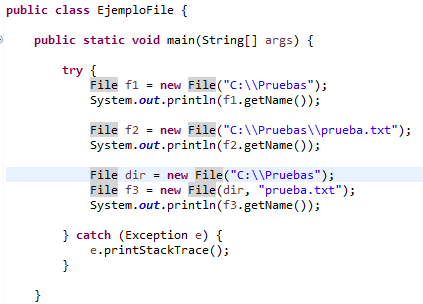
Si el archivo o carpeta que se intenta examinar no existe, la clase File **no devuelve** una excepción. Por lo que si necesitamos saberlo, hay que utilizar el método exists. Este método devuelve true si la carpeta o archivo existe. Vamos a ver un ejemplo.

|  |
| --- |
| Ojo, que **new File()** no crea el fichero físicamente, ¡es un mero descriptor que apunta a la información del fichero al que apunta o apunte en el futuro! |



Según la ayuda del api, los constructores son:

|  |  |
| --- | --- |
| **[File](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/File.html" \l "<init>(java.io.File,java.lang.String)) ([File](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/File.html" \t "class in java.io) padre, [String](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html" \t "class in java.lang) hijo)** | Crea una instancia de File a partir de un File padre y una ruta que se une a la ruta del hijo (String) |
| **[File](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/File.html" \l "<init>(java.lang.String)) ([String](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html" \t "class in java.lang) ruta**) | Crea una instancia de File usando la cadena recibida como ruta. |
| **[File](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/File.html" \l "<init>(java.lang.String,java.lang.String)) ([String](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html" \t "class in java.lang) padre, [String](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html" \t "class in java.lang) hijo)** | Crea una instancia de File usando las 2 cadenas recibidas y la ruta es la concatenación de la primera con la segunda |
| **[File](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/File.html" \l "<init>(java.net.URI)) ([URI](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/net/URI.html" \t "class in java.net) uri)** | Crea una instancia de File convirtiendo la URI en una ruta |



Arriba tienes ejemplo de los 3 primeros constructores, pruébalo tecleando el programa en tu Eclipse (**ejemplos029\_ficheros –> CreaFichero2.java**). Un ejemplo de la última, que quizás sea la que puedes ver menos clara, sería:

package IO.infoFile;

import java.io.File;

import java.net.URI;

import java.net.URISyntaxException;

public class EjemploURI {

public static void main(String[] args) {

File aFile;

try {

//aFile = new File(new URI("file:///c:/a.bat"));

aFile = new File("https://elpais.com");

System.out.println(aFile.getName());// false

} catch (Exception e) {

e.printStackTrace();

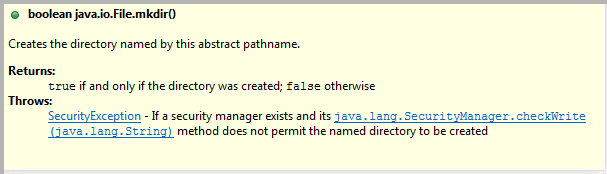
}

}

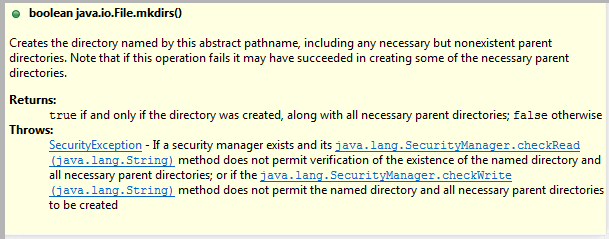
}

### 2.2 Creación de directorios/carpetas

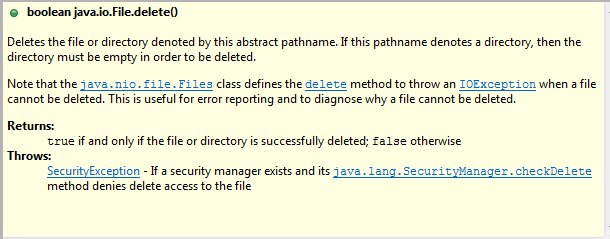
Esta necesita que exista la carpeta contenedora:



Esta crea las carpetas contenedoras si no existen:

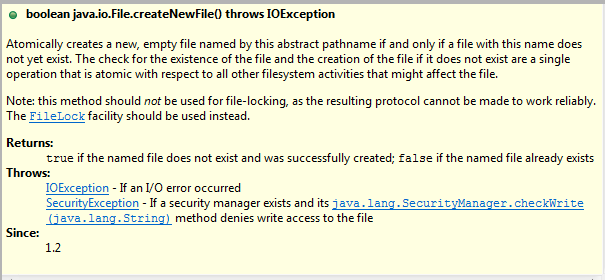


Borrado de ficheros/directorios:



Se borra el fichero si la ruta es de un fichero, o la carpeta si se refiere a la ruta de una carpeta. No borra el resto de carpetas: ¡Ojo, que solo se borra la carpeta si está vacía! Haz la prueba.

Creación de ficheros, crea el fichero si no existe:



**Ejemplo** de creación de un fichero (**ejemplos029\_ficheros –> CreaFichero2.java)**:

import java.io.\*;

public class CreaFichero1 {

public static void main(String args[]){

// Crea un objeto File dada la ruta completa

File f1 = new File("C:\\Ficheros\\nuevo.txt");

try {

// A partir del objeto File creamos el

// fichero físicamente

if (f1.createNewFile())

System.out.println("El fichero se ha creado correctamente");

else

System.out.println("No ha podido ser creado

el fichero");

} catch (IOException ioe) {

ioe.printStackTrace();

}

// Crea un objeto File dada la ruta del directorio y el nombre

//del fichero por separado

File f2 = new File("C:\\Ficheros", "nuevo2.txt");

// Crea un objeto File dado el directorio y el nombre

//del fichero por separado

try {

// A partir del objeto File creamos el

// fichero físicamente

if (f2.createNewFile())

System.out.println("El fichero se ha creado correctamente");

else

System.out.println("No ha podido ser creado el fichero");

} catch (IOException ioe) {

ioe.printStackTrace();

}

File dir = new File("C:\\Ficheros");

File f3 = new File(dir, "nuevo3.txt");

try {

// A partir del objeto File creamos el

// fichero físicamente

if (f3.createNewFile())

System.out.println("El fichero se ha creado correctamente");

else

System.out.println("No ha podido ser creado

el fichero");

} catch (IOException ioe) {

ioe.printStackTrace();

}

}

}

Ejemplo de creación de un fichero para escritura:

package ejemplos029\_ficheros3\_readerWriter;

import java.io.\*;

public class CreaFichero2 {

public static void main(String []args){

String frase="Esto es un ejemplo de escritura de ficheros de texto";

try{

//Crear un objeto File se encarga de crear o abrir

// acceso a un archivo que se especifica en su constructor

File archivo = new File("texto.txt");

// Crear objeto FileWriter que será el que nos ayude a

// escribir sobre archivo

FileWriter escribir = new FileWriter(archivo);

// Crea el fichero en la carpeta del proyecto

// FileWriter escribir = new FileWriter(archivo,true);

// Para añadir

// Se escribe en el archivo con el metodo write

// true: si ya existe el fichero no lo machaco,

// continuo escribiendo donde lo deje escribir.write(frase);

escribir.close(); // Se cierra la conexión

}

catch(Exception e){ System.out.println("Error al escribir"); }

}

}

## 2. Operaciones sobre ficheros

Los tipos de operaciones son:

a) Operación de Creación

b) Operación de Apertura

Varios modos:

b.1) Solo lectura

b.2) Solo escritura

b.3) Lectura y escritura

c) Operaciones de lectura / escritura

d) Operaciones de inserción / borrado

e) Operaciones de renombrado / eliminación

f) Operación de desplazamiento dentro de un fichero

g) Operación de cierre

Las operaciones para el manejo habitual de un fichero:

1.- Crearlo (solo si no existía previamente)

2.- Abrirlo

3.- Operar sobre él (lectura/escritura, inserción, borrado, etc.)

4.- Cerrarlo

## 3. Tipos de ficheros

La clasificación de los ficheros según el **acceso** a la información almacenada es:

a) **Acceso secuencial**: Para acceder a los datos es necesario pasar por todos los anteriores. **Ej**: Cinta de Cassete.

b) **Acceso directo o aleatorio**: Se puede acceder a un dato sin pasar por todos los anteriores. **Ej**: Disco Duro, arrays en Java.

Clasificación de los ficheros según el **contenido**:

Sabemos que es diferente manipular números que Strings, aunque en el fondo ambos acaben siendo bits en la memoria del ordenador. Por eso, cuando manipulamos archivos, distinguiremos dos clases de archivos dependiendo del tipo de datos que contienen:

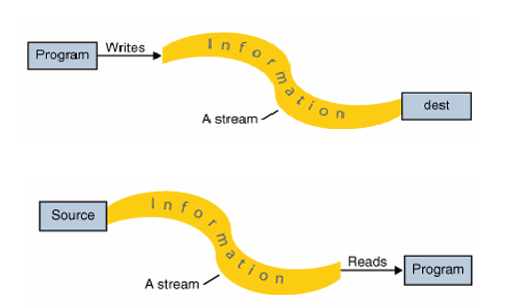
* Los archivos de **caracteres** (o **de texto**). Almacenan caracteres alfanuméricos en un formato estándar (ASCII, Unicode, UTF8, UTF16, etc.).
* Los archivos de **bytes** (o **binarios**). Almacenan secuencias de dígitos binarios (ej: ficheros que almacenan enteros, floats, imágenes…). *Más adecuado para copiarlos*

|  |
| --- |
| Un **fichero de texto** es aquel formado exclusivamente por caracteres y que, por tanto, puede crearse y visualizarse usando un editor (plano). Las operaciones de lectura y escritura trabajarán con caracteres. Por ejemplo, los ficheros con código java son ficheros de texto.  En cambio, un **fichero binario** ya no está formado por caracteres, sino que los bytes que contiene pueden representar otras cosas como números, imágenes, sonido, etc. |

## 4. Conceptos Básicos de Entrada/Salida

En Java se define la abstracción de **stream** (flujo) para tratar la comunicación de información entre el programa y el exterior. Entre una fuente y un destino fluye una secuencia de datos.

Los flujos actúan como **interfaz con el dispositivo o clase asociada** y proporcionan una **operación independiente del tipo de datos y del dispositivo**. Nos permiten usar diversidad de dispositivos (fichero, pantalla, teclado, red…) y diversidad de formas de comunicación.



**Flujos estándar:**

**Entrada estándar** - habitualmente el teclado

**Salida estándar** - habitualmente la consola

**Salida de error** - habitualmente la consola

En Java se accede a la E/S estándar a través de atributos estáticos de la clase java.lang.System:

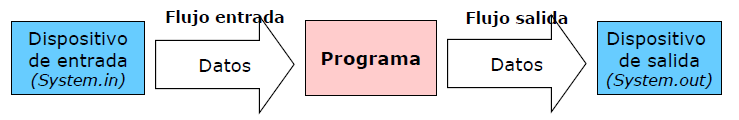
**System.in** implementa la entrada estándar *atributo estatico de la clase System*

**System.out** implementa la salida estándar

**System.err** implementa la salida de error

La entrada/salida estándar (normalmente el teclado y la pantalla, respectivamente) se define mediante dos objetos que puede usar el programador sin tener que crear flujos específicos.

La clase ***System*** tiene un miembro denominado ***in*** que es una instancia de la clase ***InputStream*** que representa al teclado o flujo de entrada estándar *https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStream.html* Sin embargo, el miembro ***out*** de la clase ***System*** es un objeto de la clase ***PrintStream***, que imprime texto en la pantalla (la salida estándar.



**Flujos estándar:**

***System.in***

Instancia de la clase *InputStream*: flujo de bytes de entrada

Métodos para la lectura de datos:

• *read()* permite leer un byte de la entrada como entero

• *skip(n)* ignora n bytes de la entrada

• *available()* número de bytes disponibles para leer en la entrada

***System.out***

Instancia de la clase *PrintStream*: flujo de bytes de salida

Métodos para impresión de datos:

• *print(), println()*

• *flush() vacía el buffer de salida escribiendo su contenido*

***System.err***

Funcionamiento similar a *System.out*

*Se utiliza para enviar mensajes de error*

*(por ejemplo a un fichero de log o a la consola)*

|  |
| --- |
| Para **leer un carácter** solamente tenemos que llamar a la función ***read*** desde ***System.in (***recuerda que System.in que es una instancia de la clase ***InputStream)***. *//pulse cualquier tecla para continuar*  try{  **System.in.read();** }catch (IOException ex) { } |

**Ejemplo**: package ejemplos029\_ficheros3\_leeLineaEntradaEstandar;

import java.io.\*;

// Ejemplo de lectura de líneas de la entrada estándar

// carácter a carácter

// Contamos los caracteres que hay en una línea

class LecturaDeLinea {

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

//el throws en el main no debe hacerse,

//no nos enteramos cuando el main la relanza

int c;

int contador = 0;

// Se lee hasta encontrar el fin de línea

while((c = System.in.read()) != '\n')

{

contador++;

System.out.print( (char) c );

}

System.out.println(); // Se escribe el fin de línea

System.err.println( "Contados "+ contador

+" bytes/caracteres en total." );

}

}

**Utilización de los flujos:**

|  |
| --- |
| **FLUJOS** DE ENTRADA Y SALIDA EN **JAVA**. ... El **flujo (stream)** es una secuencia ordenada de datos que tiene una fuente (**flujos** de entrada) o un destino (**flujos** de salida). Los streams soportan varios tipos de datos: bytes simples, tipos de datos primitivos, caracteres localizados, y objetos. Los flujos se implementan en las clases del paquete java.io.  Esencialmente todos funcionan igual, independientemente de la fuente de datos. |

**Lectura**:

**1.** Abrir un flujo a una fuente de datos (creación del objeto stream)

• Teclado (como hemos visto en el ejemplo ***leerUnaLineaEntrEstanda.java***)

• Fichero

• Socket remoto

**2.** Mientras existan datos disponibles:

• Leer datos

**3.** Cerrar el flujo (método close())

**Escritura**:

**1.** Abrir un flujo a una fuente de datos (creación del objeto stream)

• Pantalla (como hemos visto en el ejemplo ***leerUnaLineaEntrEstanda.java***)

• Fichero

• Socket local

**2.** Mientras existan datos disponibles:

• Escribir datos

**3.** Cerrar el flujo (método close)

***Nota****: para los flujos estándar ya se encarga el sistema de abrirlos y cerrarlos*

Un fallo en cualquier punto produce la excepción IOException, es obligatorio capturarla.

**Clasificación de flujos**

**Según la representación de la información:**

Flujos de bytes: clases *InputStream* y *OutputStream*

Flujos de caracteres: clases *Reader* y *Writer*

Recuerda que System.in es un *InputStream*

Se puede pasar de un flujo de bytes a uno de caracteres con las clases *InputStreamReader* y *OutputStreamWriter*

**Según el propósito:**

Entrada: *InputStream, Reader*

Salida: *OutputStream, Writer*

Lectura/Escritura: *RandomAccessFile (acceso directo, no aleatorio)*

Transformación de los datos:

Realizan algún tipo de procesamiento sobre los datos (p.e. *buffering*, conversiones, filtrados): *BufferedReader, BufferedWriter.*

**Según el tipo de acceso:**

Secuencial - (Para acceder a un elemento hay que pasar por todos los demás)

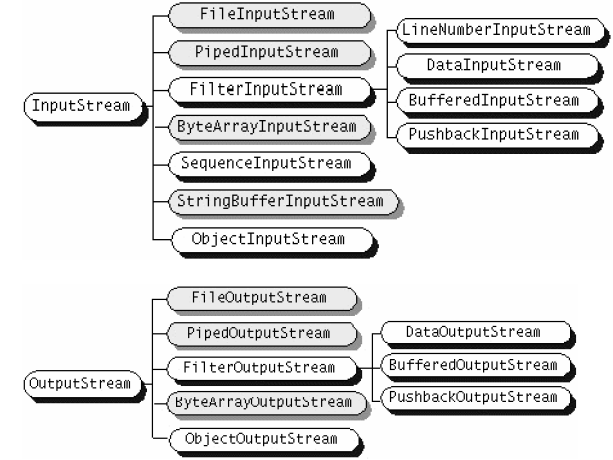
Aleatorio - (*RandomAccessFile*) (Se puede acceder directamente a un elemento)

Los flujos están agrupados en el paquete java.io y se dividen en dos jerarquías de clases independientes, una para lectura/escritura binaria (bytes) y otra para lectura/escritura de caracteres de texto (char).

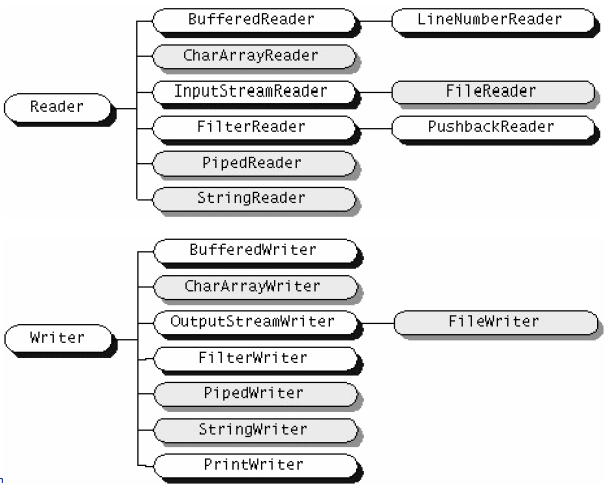
**Jerarquía de flujos de bytes (Streams)**

*Son para leer ficheros binarios*  
En el siguiente gráfico vemos la relación entre la clase InputStream y sus hijas. Existen para facilitarnos las distintas operaciones que tenemos que hacer. Veremos distintos ejemplos con las más importantes. Recuerda que tratamos con estas clases cuando leamos bytes o conjuntos de bytes directamente.

Con estas clases (que tratan con Streams) se podría leer cualquier tipo de fichero, aunque cuando se trate de ficheros de texto puro (de los que se editan en un editor de ficheros planos, como notepad) es más fácil tratarlos con las clases que se muestran en el segundo gráfico (Jerarquía de flujos de caracteres).



**Jerarquía de flujos de caracteres**



Puedes investigar sobre las principales clases de ambos tipos en la ayuda de Java:

https://docs.oracle.com/javase/8/docs/api/java/io/Writer.html

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/Writer.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/Reader.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/Reader.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html>

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStream.html

<https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html>

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/OutputStream.html

|  |
| --- |
| Vamos a ir viendo cómo se producen las distintas operaciones de lectura/escritura con las distintas clases. Se utilizarán unas u otras según las necesidades de nuestros programas. |

**Entrada de caracteres (ambas clases son descendientes de la clase Reader, flujo de caracteres)**

**InputStreamReader**

Lee bytes de un flujo InputStream y los convierte en caracteres Unicode.

Métodos de utilidad:

• *read()* lee un único carácter

• *ready()* indica cuando está listo el flujo para lectura

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStreamReader.html

**BufferedReader**

Entrada mediante búfer, mejora el rendimiento.

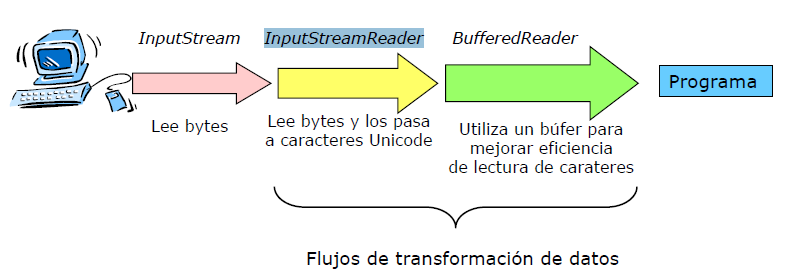
Método de utilidad:

• *readLine()* lectura de una línea como cadena

Ejemplo (*antes de tener Scanner para leer una línea había que hacer esto)*:

|  |
| --- |
| **InputStreamReader** entrada **= new InputStreamReader(System.in);** |
| **BufferedReader** teclado **= new BufferedReader (entrada);** |
| **String** cadena **= teclado.readLine();** |

Los flujos se pueden combinar para obtener la funcionalidad deseada:



**Ejemplo de combinación de flujos:**

ejemplos029\_ficheros3\_leeLineaEntradaEstandar;

import java.io.\*;

public class Eco {

public static void main (String[] args) {

BufferedReader entradaEstandar =   
new BufferedReader (new InputStreamReader(System.in));

String mensaje;

System.out.println ("Introducir una línea de texto:");

mensaje = entradaEstandar.readLine();

System.out.println("Introducido: " + mensaje);

}

}

**Ejemplo de entrada de texto desde un fichero:**

class LeeFicheroLineas3: ejemplos029\_ficheros3\_bufferedReader

*<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/BufferedReader.html>  
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/FileReader.html>*

try {

BufferedReader reader =

new BufferedReader(new FileReader(“*nombrefichero*"));

String linea = reader.readLine();

while(linea != null) {

*// procesar el texto de la línea*

linea = reader.readLine();

}

reader.close();

}

catch(FileNotFoundException e) {

*// No se encontró el fichero*

}

catch(IOException e) {

*// Algo fue mal al leer o cerrar el fichero*

}

**Ejemplo** (como la clase Scanner): ejemplos029\_ficheros3\_leeLineaEntradaEstandar;

import java.io.\*;

class Teclado {

/\*\* variable de clase asignada a la entrada estándar

\* del sistema

\*/

public static BufferedReader entrada =

new BufferedReader(new InputStreamReader(System.in));

/\*\* lee una cadena desde la entrada estándar

\* @return cadena de tipo String

\*/

public static String leerString() {

String cadena="";

try {

cadena = new String(entrada.readLine());

//antes la clase String no estaba en Java implicitamente

}catch (IOException e) {

System.out.println("Error de E/S");

}

return cadena;

}

/\*\* lee un numero entero desde la entrada estandar

\* @return numero entero de tipo int

\*/

public static int leerInt() {

int entero = 0;

boolean error = false;

do {

try {

error = false;

entero = Integer.valueOf(entrada.readLine());

}catch (NumberFormatException e1) {

error = true;

System.out.println("Error en el formato del número,

intentelo de nuevo.");

}catch (IOException e) {

System.out.println("Error de E/S");}

} while (error);

return entero;

}

}

## 5. Ficheros de Texto

Existen dos clases que manejan caracteres que son **FileWriter** y **FileReader**.

Veamos su jerarquía:

|  |  |
| --- | --- |
| * **[java.lang.Object](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html" \t "class in java.lang)** * **[java.io.Writer](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/Writer.html" \t "class in java.io)** * **[java.io.OutputStreamWriter](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/OutputStreamWriter.html" \t "class in java.io)** * **java.io.FileWriter** | * **[java.lang.Object](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html" \t "class in java.lang)** * **[java.io.Reader](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/Reader.html" \t "class in java.io)** * **[java.io.InputStreamReader](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStreamReader.html" \t "class in java.io)** * **java.io.FileReader** |

La construcción de objetos del tipo **FileReader** se hace con un parámetro que puede ser un objeto **File** o un String que representarán a un determinado archivo.

La construcción de objetos **FileWriter** se hace igual, aunque también se le puede añadir un **segundo parámetro** booleano, que tomará valor true si se abre el archivo para añadir datos o false para crearlo por primera vez; si tuviese datos estos se borrarían en este segundo caso.

Para escribir se utiliza el método **write** que recibe como parámetro lo que se desea escribir en formato int o String y para leer se utiliza el método **read**, que devuelve un int, el valor en código ASCII del carácter leído. En el momento en que se haya alcanzado el **final del fichero** (no haya más datos que leer) el método read retorna el valor -1.

Métodos básicos de **Reader**:

int read()

int read(char cbuf[])

int read(char cbuf[], int offset, int length)

Métodos básicos de **Writer**:

int write(int c)

int write(char cbuf[])

int write(char cbuf[], int offset, int length)

Ejemplo de lectura de un fichero (ya existente) carácter a carácter:

package ejemplos029\_ficheros3\_readerWriter;

//Ejemplo de lectura de un fichero carácter a carácter.

import java.io.\*;

public class LeeFicheroCaracter{

public static void main(String arg[]){

// Se define un int que va a contener los

// caracteres del archivo

int c;

try{

// Se crea un objeto FileReader que

// obtiene lo que tenga el archivo

FileReader lector=new FileReader("texto.txt");

c=lector.read();

while(c!=-1){ //c=-1 indica el fin de fichero

System.out.println((char) c);

c=lector.read();

}

lector.close();

}

catch(Exception e){

System.out.println("Error al leer");

}

}

}

Las clases **BufferedReader** y **BufferedWriter** leen y escriben un texto desde un stream de salida o de entrada. Proporcionan un buffer de almacenamiento temporal.

La **escritura** se realiza con el método **write** que permite grabar caracteres, Strings y arrays de caracteres en el fichero. Permite utilizar el método newLine() que escribe un salto de línea en el archivo.

|  |  |
| --- | --- |
| * **[java.lang.Object](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html" \t "class in java.lang)** * **[java.io.Writer](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/Writer.html" \t "class in java.io)** * **java.io.BufferedWriter** | * **[java.lang.Object](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html" \t "class in java.lang)** * **[java.io.Reader](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/Reader.html" \t "class in java.io)** * **java.io.BufferedReader** |

Para **leer**, se tiene el método **readLine**() que lee una línea de texto entera cada vez. En este caso, cuando se haya alcanzado el final del fichero el método retorna el valor null.

No es posible usarlas por sí mismas, se deben envolver en un Reader o en un Writer dentro de un BufferedReader o de un BufferedWriter respectivamente. Esto es así porque sus constructores necesitan un parámetro de tipo Reader y Writer respectivamente. Busca estas clases en la ayuda del jdk y compruébalo.

public class LeeLineas{

public static void main(String arg[]){

// Creamos un String que va a ir conteniendo

// todo el texto del archivo

String texto="";

try{

// Se crea un objeto FileReader que

// obtiene lo que tenga el archivo

FileReader lector=new FileReader("texto.txt");

// El contenido del objeto se envuelve

// en un BufferedReader

BufferedReader contenido=new BufferedReader(lector);

texto= contenido.readLine();

while(texto!=null){

System.out.println(texto);

texto= contenido.readLine();

}

contenido.close(); // Se puede omitir

lector.close();

}catch(Exception e){

System.out.println("Error al leer");

}

}

}

Otra opción de escritura es usar la clase PrintWriter:

salida =new PrintWriter(new FileWriter(nombre))

FileWriter (nombre, añadir)   
// así escribimos al final del fichero

Con los métodos:

println()

print()

close():IOException

**Ejemplo**:

package ejemplos029\_ficheros5\_ficheroTextoBuffered;

import java.io.\*;

public class PruebaEscritura {

public static void main(String[] args) {

try {

FileWriter connection =

new FileWriter("C:\\test.txt", true);

PrintWriter file = new PrintWriter(connection);

file.println("Hola");

file.println("Hola");

file.close();

writeAgain();

}

catch (IOException e) {

System.out.println("IOException");

}

}

public static void writeAgain() throws IOException {

FileWriter connection =

new FileWriter("C:\\test2.txt", true);

BufferedWriter file = new BufferedWriter(connection);

file.write("Adios");

file.write("Adios");

file.close();

}

}

**BufferedWriter vs PrintWriter:**

**PrintWriter**: es el objeto que utilizamos para escribir directamente sobre el archivo de texto.

**BufferedWriter**: objeto que reserva un espacio en memoria donde se guarda la informacion antes de ser escrita en un archivo.

**Comparado** con las clases FileWriter, los BufferedWriters escriben relativamente grandes cantidades de informacion a un archivo, lo cual minimiza el numero de veces que las operaciones de escritura de archivos se llevan a cabo

*Es más optimo el bufferWriter porque hace menos accesos a disco*

* **FileWriter:** es un objeto que tiene como función escribir datos en un archivo.
* **BufferedWriter**: objeto que reserva un espacio en memoria (buffer) donde se guarda la información antes de ser escrita en un archivo.
* **PrintWriter**: Es el objeto que utilizamos para escribir directamente sobre el archivo de texto.

## 6. Ficheros binarios

**Definición y uso**

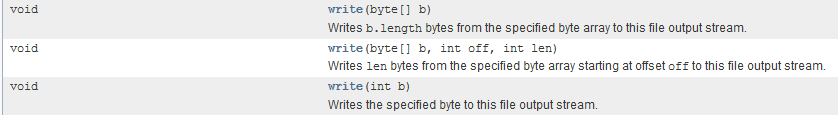
Un **fichero binario** es un fichero que almacena una secuencia de datos codificados en binario (secuencias de datos short, int, double, boolean, char, etc.)

Para trabajar con este tipo de ficheros, es decir, para escribir o leer en ellos la información binaria que almacenan se necesitan herramientas apropiadas y conocer el patrón que guía su disposición en el fichero. En un fichero binario no hay líneas. Vamos a ver dos formas de usarlos:

**FileOutputStream y FileInputStream:**

|  |  |
| --- | --- |
| * **[java.lang.Object](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html" \t "class in java.lang)** * **java.io.OutputStream** * **java.io. FileOutputStream** | * **[java.lang.Object](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html" \t "class in java.lang)** * **java.io.InputputStream** * **java.io. FileInputStream** |

Es una pareja de clases que nos permite leer grupos de bytes de un tamaño definido, por ejemplo nos serviría para copiar un fichero de imagen.



Pero como puedes ver en los ejemplos, también valdría para hacer una copia de un fichero de texto:

EjFileInputOutputStream1.java y EjFileInputOutputStream2.java

**DataOutputStream y DataInputStream:**

|  |  |
| --- | --- |
| * **[java.lang.Object](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html" \t "class in java.lang)** * **java.io.OutputStream** * **java.io.FilterOutputStream** * **java.io. DataOutputStream** | * **[java.lang.Object](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html" \t "class in java.lang)** * **java.io.InputputStream** * **java.io.FilterInputputStream** * **java.io. DataInputStream** |

La clase **DataOutputStream** permite crear un objeto que se asocia a un objeto **FileOutputStream** y facilita métodos para escribir o almacenar secuencialmente información codificada en binario en el fichero asociado a dicho objeto. La escritura de un nuevo dato se produce de forma secuencial.

La escritura física de datos en el fichero no se produce uno a uno, sino que se gestiona en bloques mediante un **buffer** o almacén intermedio. El programador debe gestionar la captura y tratamiento de IOException.

Los **métodos** más usados son:

void writeBoolean(boolean val)

void writeChar(int val)

void writeInt(int val)

void writeDouble(double val)

void writeChars(String str)   
//DEPRECATED -> obsolete (substitute: writeUTF)

void writeUTF(String str)

**Ej:** Crea un fichero binario compuesto por una secuencia de pares (Nmat, nota) solicitados por teclado.

import java.io.\*;

import java.util.\*;

public class Binarios1 {

public static void main(String args[]) {

int nm;

double nota;

Scanner teclado = new Scanner(System.in); // para leer

try {

FileOutputStream fos=new FileOutputStream("notas.dat");

DataOutputStream dos= new DataOutputStream(fos);

System.out.print("Introduzca un NMat (0 para acabar): ");

nm = teclado.nextInt();

while (nm != 0) {

dos.writeInt(nm);

System.out.print("Introduzca una nota: ");

nota = teclado.nextDouble();

dos.writeDouble(nota);

System.out.print("Introduzca un NMat (0 para acabar): ");

nm = teclado.nextInt();

}

dos.close();

fos.close();

}

catch (IOException ex) {

System.out.println("Error: " + ex.getMessage());

}

}

}

La clase **DataInputStream** permite crear un objeto que se asocia a un objeto **FileInputStream** y facilita métodos para leer de él secuencialmente información codificada en binario.

La lectura de un nuevo dato se produce de forma secuencial, se lee la secuencia, información almacenada en el fichero detrás de la última información leída.

El programador debe gestionar la captura y tratamiento de IOException. El intento de lectura de un dato cuando ya ha sido leído todo el fichero lanza la excepción EOFException. Su gestión es clave en la programación de operaciones de lectura de un fichero binario.

Los **métodos** más usados son:

boolean readBoolean()

byte readByte()

char readChar()

int readInt()

double readDouble()

String readUTF()

**Ej**: Muestra el fichero binario compuesto por una secuencia de pares (NMat, nota)

public class Binarios2 {

public static void main(String args[]) {

int nm;

double nota;

try {

FileInputStream fis = new FileInputStream("notas.dat");

DataInputStream dis = new DataInputStream(fis);

System.out.println(" NMAT Nota");

try {

while (true){

nm = dis.readInt();

nota = dis.readDouble();

System.out.printf("%6d %5.1f \n", nm, nota);

//nº matricula ocupa: 6, nota: 5 enteros y 1 decimal

}

}catch (EOFException ex) {

System.out.println("FINAL DE FICHERO " +

ex.getMessage());

}

dis.close();

fis.close();

}

catch (IOException ex) {

System.out.println("Error: " + ex.getMessage());

}

}

}

También podríamos haber usado para leer el fichero la condición: while (dis.available() > 0)

Evitando leer el fin de fichero

## 7. Ficheros de acceso aleatorio- acceso directo (binarios)

*<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/RandomAccessFile.html>*

*implementa autoclosable (se puede poner en un try con recursos)*

Hasta ahora los archivos se han leído o escrito secuencialmente, es decir, desde el inicio hasta el final; pero es posible, leer datos o escribir datos en una zona concreta del archivo.

La clase **RandomAccessFile** permite acceder al archivo en forma aleatoria. Es decir, se permite leer o escribir cualquier posición del archivo en cualquier momento. Esta clase implementa las **interfaces DataInput y DataOutput** que sirven para leer y escribir datos.

La construcción requiere de una cadena que contenga un archivo File y un segundo parámetro obligatorio denominado **modo**, que va a especificar el tipo de operación que se realizará sobre el fichero. El modo es una cadena que puede contener una **r** (lectura), **w** (escritura) o ambas, **rw**.

|  |
| --- |
| * **[java.lang.Object](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html" \t "class in java.lang)** * **java.io.**RandomAccessFile |

**Ej**:

File f=new File("prueba.txt");

RandomAccessFile archivo = new RandomAccessFile(f, "rw");

Como ocurría en las clases anteriores, hay que capturar la excepción. Los métodos fundamentales son:

**void seek(long pos).** Permite colocar el puntero en una posición concreta, contada en bytes, en el archivo. Lo que se coloca es la señal que marca la posición a leer o escribir.

**long getFilePointer().** Posición actual del puntero de acceso

**long length().** Devuelve el tamaño del archivo

**int skipBytes(int desplazamiento).** Desplaza el puntero desde la posición actual, el número de bytes indicado por desplazamiento

Los **métodos de lectura**, leen un dato del tipo indicado y son equivalentes a los disponibles en la clase DataInputStream.

**readBoolean, readByte, readInt, readDouble, readFloat, readUTF, readLine.**

Los **métodos de escritura**, reciben como parámetro el dato a escribir y escriben encima de lo hubiese ya escrito, por lo que para añadir hay que colocar el puntero de acceso al final del archivo.

**writeBoolean, writeByte, writeBytes, writeInt, writeDouble, writeFloat, writeUTF, writeLine.**

**Ej**: Crear un fichero binario, de acceso aleatorio, con los 100 primeros números y mostrarlo en pantalla.

import java.io.\*;

public class Aleatorio{

public static void main(String args[]) {

File f = new File("prueba.dat");

RandomAccessFile raf=null;

try {

raf = new RandomAccessFile(f,"rw");

for ( int i=1; i <= 100 ; i++ )

raf.writeInt( i );

// un entero ocupa 4 bytes; total 400 bytes

// total 400 bytes

try{

System.out.println( " El fichero ocupa " +

raf.length() + " bytes." );

raf.seek(0); // La primera posición empieza en 0

System.out.print(" En la posicion " +

raf.getFilePointer());

System.out.println(" está el número " +raf.readInt() );

raf.skipBytes( 9\*4 );

// Salto 9 => Elemento 10 más allá

// ya ha leído la 1era, asi que salto 10

System.out.print(" 10 elementos más allá, está el ");

System.out.println(raf.readInt());

raf.close();

}

catch(IOException e){

System.out.println();

}

}catch (FileNotFoundException e) {

System.out.println();

}

catch(IOException e){

System.out.println();

}

}

}

**Ej:** Crea un fichero con los 10 primeros números enteros y después modificarlo multiplicando por 2 las componentes pares (valores escritos en el fichero).

import java.io.\*;

public class Aleatorios2{

public static void main(String args[]) {

File f = new File("prueba.dat");

RandomAccessFile raf=null;

try {

raf = new RandomAccessFile(f,"rw");

for ( int i=1; i <= 10 ; i++ )

raf.writeInt( i );

try{

raf.seek(0); // La primera posición empieza en 0

while(true){

int valor=raf.readInt();

System.out.println(valor);

if(valor%2==0){

valor=valor\*2; System.out.println("Escribo "+valor+

" En la posicion " +

((raf.getFilePointer()-4)/4 + 1));

// restamos lo que ocupa un int: 4

raf.seek(raf.getFilePointer()-4);

//Me posiciono 4 bytes atrás

raf.writeInt(valor);

}

}

}catch(EOFException e){

}

}

catch(IOException e){

System.out.println(e);

} finally {

try {

if (raf != null) {

raf.close();

}

} catch (IOException e) {

System.out.println(e.getMessage());

}

}

}

}

## 8. Serialización

Un objeto **serializable** es un objeto que se puede convertir en una secuencia de bytes. Es una forma automática de guardar y cargar el estado de un objeto.

Se basa en la interfaz Serializable (en el paquete java.io) que es la que permite esta operación. Si una clase implementa esta interfaz, sus objetos pueden ser guardados y restaurados directamente en un archivo.

Cuando se desea utilizar un objeto para ser almacenado con esta técnica, la clase debe implementar la interfaz Serializable, incluyendo en la cabecera de la clase **implements Serializable.** Esta interfaz no posee métodos, pero es un requisito obligado para hacer que un objeto sea serializable.

Las clases ObjectInputStream y ObjectOutputStream se encargan de realizar los procesos de lectura o escritura del objeto de o a un fichero. Son herederas de InputStream y OutputStream y son casi iguales a DataInputStream/DataOutputStream solo que añaden los métodos **readObject** y **writeObject** que son los que permiten utilizar directamente los objetos.

|  |  |
| --- | --- |
| * **[java.lang.Object](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html" \t "class in java.lang)** * **java.io.OutputStream** * **java.io. ObjectOutputStream** | * **[java.lang.Object](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html" \t "class in java.lang)** * **java.io.InputputStream** * **java.io. ObjectInputStream** |

**Escribir objetos** al flujo de salida **ObjectOutputStream** es muy simple y requiere los siguientes **pasos**:

a) Se crea el objeto de la clase serializable.

b) Se crea un objeto de la clase FileOutputStream, con el nombre de fichero y opcionalmente, se le puede añadir el valor true si en el fichero se van a añadir datos.

c) Se vincula el objeto de la clase FileOutputStream al del flujo de salida de la clase ObjectOutputStream que es el que va a procesar los datos.

Ej:

FileOutputStream fos = new FileOutputStream("fichero.dat", true);

ObjectOutputStream oos = new ObjectOutputStream(fos);

d) Con el método writeObject se escriben los objetos al flujo de salida y los guarda en el archivo.

e) Finalmente hay que cerrar los flujos.

**Ejemplo:**

import java.io.\*;

public class Persona implements Serializable{

String nombre;

int edad;

public Persona(String n,int e) {

nombre=n;

edad=e;

}

void CambiarEdad(int e){

edad=e;

}

void Escribir(){

System.out.println("Nombre "+nombre+" Edad "+edad);

}

}

El **proceso de lectura** es similar al proceso de escritura y es muy simple. Los **pasos** a seguir son los siguientes:

a) Se crea un flujo de entrada a disco, pasándole el nombre del archivo.

b) El flujo de entrada ObjectInputStream es el que procesa los datos y se vincula a un objeto de la clase FileInputStream.

c) El método readObject lee los objetos del flujo de entrada, en el mismo orden en el que ha sido escritos. Es necesario hacer un cast al tipo de la clase. Cuando no haya más datos que leer saltará la excepción EOFException.

d) Finalmente, se cierra los flujos

Ejemplo: ejemplos030\_ficherosSerializable

import java.io.\*;

public class PruebaPersona {

public static void main(String[] args) {

Persona p1=new Persona("Pablo", 20);

Persona p2=new Persona("Rosa", 35);

try {

FileOutputStream fos =

new FileOutputStream("datos.obj");

ObjectOutputStream oos=new ObjectOutputStream(fos);

oos.writeObject(p1);

oos.writeObject(p2);

oos.close();

fos.close();

}

catch (IOException ex) {

System.out.println(ex);

}

try{

FileInputStream fis= new FileInputStream("datos.obj");

ObjectInputStream ois=new ObjectInputStream(fis);

try{

while(true){

p1=(Persona)ois.readObject();

System.out.println("Nombre "+p1.nombre+

" Edad "+p1.edad);

}

} catch(EOFException e){ }

ois.close();

fis.close();

} catch (IOException ex) {

System.out.println(ex);

} catch (ClassNotFoundException ex) {

//en el JDK buscar que excepciones provoca el metodo

System.out.println(ex);

}

}

}

Cuando un atributo de una clase contiene información que no se desea que se pueda guardar, hay técnicas para protegerla.

Cuando dicha información es privada, el atributo tiene el modificador private, pero una vez que se ha enviado al flujo de salida cualquiera puede leerla en el archivo en disco o interceptarla en la red. El modo más simple de proteger la información sensible, como una contraseña es la de poner el modificador **transient** delante del atributo que la guarda.

La *salida* del programa es:

Nombre: Pepe

Cuenta: No disponible

Lo que indica que la información que contiene el atributo Cuenta y que tiene el modificador transient no se ha guardado en el archivo. En la reconstrucción del objeto con la información guardada en el archivo el dato Cuenta toma el valor null.

Cuando se quieren añadir registros al fichero surge un problema, pues cada vez que se comienza a escribir se añade una cabecera, por lo que al leer posteriormente los registros salta un error cuando encuentra esas cabeceras. Para solucionar este problema, tenemos dos posibilidades:

**1.-** Crear un fichero auxiliar en el que se copien todos los registros que se tienen y a continuación los que se quieren añadir. Posteriormente, se borrará el primer fichero y el auxiliar se renombrará con el nombre del primer fichero.

**2.-** Crear una nueva clase que herede de la clase ObjectOuputStream y que el método writeStreamHeader() lo sobrescriba dejando en blanco.

**Ej:** Usando la 1ª manera:

Ejemplo: ejemplos030\_ficherosSerializable

import java.io.\*;

public class PruebaPersona2 {

public static void main(String[] args) {

Persona p1;

try{

//Se crea el fichero con dos registros

FileOutputStream fos= new FileOutputStream("datos.obj");

ObjectOutputStream oos=new ObjectOutputStream(fos);

p1=new Persona("Pepe",40);

oos.writeObject(p1);

p1=new Persona("Carmen",15);

oos.writeObject(p1);

oos.close();

fos.close();

}catch (IOException ex) {

System.out.println(ex);

}

try{

//Se añaden dos registros más usando el primer método.

File f=new File("auxiliar.obj");

FileOutputStream fos= new FileOutputStream(f);

ObjectOutputStream oos=new ObjectOutputStream(fos);

File f1=new File("datos.obj");

FileInputStream fis= new FileInputStream(f1);

ObjectInputStream ois=new ObjectInputStream(fis);

try{

while(true){

p1=(Persona)ois.readObject();

oos.writeObject(p1);

}

}

catch(EOFException e){ }

catch(ClassNotFoundException e){ }

// Hemos leído los objetos del fichero datos.obj

// y los hemos pasado

// al nuevo fichero temporal

//Añadimos dos nuevos objetos

p1=new Persona("Juan",50);

oos.writeObject(p1);

p1=new Persona("Maria",35);

oos.writeObject(p1);

ois.close();

fis.close();

oos.close();

fos.close();

f1.delete();

f.renameTo(f1);

// auxiliar pasa a ser el fichero datos.obj

}catch (IOException ex) {

System.out.println(ex);

}

//Ahora leemos el fichero con todos los datos

try{

FileInputStream fis= new FileInputStream("datos.obj");

ObjectInputStream ois=new ObjectInputStream(fis);

try{

while(true){

p1=(Persona)ois.readObject();

System.out.println("Nombre "+p1.nombre+

" Edad "+p1.edad);

}

}

catch(EOFException e){}

ois.close();

fis.close();

}

catch (IOException ex) {

System.out.println(ex);

}

catch (ClassNotFoundException ex) {

System.out.println(ex);

}

}

}

**Ej:** A ese fichero que ya tiene 4 registros se le van a añadir 2 más usando la 2ª manera.

Ejemplo: ejemplos030\_ficherosSerializable

import java.io.\*;

public class PruebaPersona3{

public static void main(String[] args){

Persona p1;

try{

// Se añaden dos registros más usando el segundo método.

// El fichero tiene que existir previamente para que esto

// funcione

File f=new File("datos.obj");

ClaseOutput co;

FileOutputStream fos= new FileOutputStream(f,true);

co=new ClaseOutput(fos);

*//clase que hereda de ObjectOutputStream*

p1=new Persona("Pepi",53);

co.writeObject(p1);

p1=new Persona("Julia",17);

co.writeObject(p1);

co.close();

fos.close();

} catch (IOException ex) {

System.out.println(ex);

}

try{

//Se lee el fichero con todos los datos

FileInputStream fis= new FileInputStream("datos.obj");

ObjectInputStream ois=new ObjectInputStream(fis);

try{

while(true){

p1=(Persona) ois.readObject();

if(p1 instanceof Persona){ *//aseguramos*

p1.Escribir();

}

}

}catch(EOFException e){}

ois.close();

fis.close();

}catch (IOException ex){

System.out.println(ex);

}catch (ClassNotFoundException ex){

System.out.println(ex);

}

}

}

Primero se crea la clase que hereda de ObjectOutputStream y que reescribirá el método writeStreamHeader() dejándolo en blanco. Será la clase que se use para escribir en el fichero.

¿Qué haríamos para que el segundo fichero funcione tanto con un fichero vacío como un fichero con datos ya introducidos?

***Recapitulando:***

**Flujos de bytes especiales**

**File streams:**

Para escribir y leer datos en ficheros

**Filter streams:**

Permiten filtrar datos mientras se escriben o leen (Se construyen sobre otro flujo)

Permiten manipular tipos de datos primitivos.

Implementan las interfaces DataInput y DataOutput y pueden heredar de las clases FilterInputStream yFilterOutputStream.

El mejor ejemplo son las clases DataInputStream y DataOutputStream para leer y escribir datos de tipos básicos

**Object streams:**

Para escribir y leer objetos

Implementa lo que se denomina ***serialización de objetos*** (guardar un objeto con una representación de bytes).

**Uso de filter streams**

Para **leer** tipos de **datos primitivos**

Se puede utilizar un *DataInputStream*

FileInputStream ficheroEntrada = new FileInputStream(“precios.cat");

DataInputStream entrada = new DataInputStream(ficheroEntrada);

double precio= entrada .readDouble();

entrada.close();

Para **escribir** tipos **de datos primitivos**

Se puede utilizar un *DataOutputStream*

FileOutputStream ficheroSalida = new FileOutputStream(“precios.cat");

DataOutputStream salida = new DataOutputStream(ficheroSalida);

salida.writeDouble(234.56);

salida.flush(); //Fuerza la escritura de datos

salida.close();

Métodos básicos de **InputStream**:

int read()

int read(byte cbuf[])

int read(byte 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.

**PrintStream / PrintWriter** se utilizan para escribir cadenas de texto.

**DataInputStream / DataOutputStream** se utilizan para escribir/leer tipos básicos (int, long, float,…).

Según el acceso a ficheros se utilizan unas clases u otras.

a) Acceso **Secuencial**: El más común. Puede ser:

a.1) Acceso **binario**: FileInputStream / FileOutputStream

a.2) Acceso a **caracteres** (texto): FileReader / FileWriter

b) Acceso **Aleatorio**: Se utiliza la clase RandomAccessFile

# UT9 Cómo convertir un Array en una Lista en Java

## 1. [Crear una lista vacía y añadir todos los elementos](https://www.delftstack.com/es/howto/java/how-to-convert-an-array-to-a-list-in-java/" \l "crear-una-lista-vac%25C3%25ADa-y-a%25C3%25B1adir-todos-los-elementos)

Es un método muy trivial y obvio. Podemos crear una lista vacía, hacer un bucle con todos los elementos del array y añadirlos a la lista.

El siguiente ejemplo ilustra esto:

import java.util.stream.\*;

import java.util.\*;

public class MyClass {

public static void main(String args[]) {

String[] myArray = new String[] { "1", "2", "3" };

List<String> myList = new ArrayList<>();

for (int i=0; i<myArray.length; i++){

myList.add(myArray[i]);

}

System.out.println(myList);

}

}

## 2. [Usa Arrays.asList() para convertir un array en una lista en Java](https://www.delftstack.com/es/howto/java/how-to-convert-an-array-to-a-list-in-java/" \l "usa-arrays.aslist-para-convertir-un-array-en-una-lista-en-java)

Podemos usar el método incorporado que proporciona la clase Arrays para convertir un array en una lista - Arrays.asList(arr). Importamos java.util.\* en el código.

El siguiente ejemplo lo ilustra.

import java.util.\*;

public class MyClass {

public static void main(String args[]) {

String[] myArray = new String[] { "1", "2", "3" };

List<String> myList = Arrays.asList(myArray);

System.out.println(myList);

}

}

Sin embargo, este método da como resultado una lista de tamaño fijo, y no podemos añadir más elementos a ella. Daría una excepción.

## 3. [Usa new ArrayList<>(Arrays.asList(arr)) para convertir un Array en una lista en Java](https://www.delftstack.com/es/howto/java/how-to-convert-an-array-to-a-list-in-java/" \l "usa-new-arraylistarrays.aslistarr-para-convertir-un-array-en-una-lista-en-java)

Este método permite convertir un array en una lista, y añadir también puede añadir elementos adicionales a la lista resultante. El siguiente ejemplo ilustra esto:

import java.util.\*;

public class MyClass {

public static void main(String args[]) {

String[] myArray = new String[] { "1", "2", "3" };

List<String> myList =

new ArrayList<>(Arrays.asList(myArray));

System.out.println("After conversion from array to list: "

+ myList);

myList.add("4");

System.out.println("After adding a new element: "

+ myList );

}

}

# UT9 Resumen Ficheros en Java

## 1. La clase File

File(String nombre)

File(String directorio, String nombre)

File(File directorio, String nombre)

File fichero=new File(nombre)

**Métodos:**

exists()

getName()

length()

lastModified()

list()

delete()

## 2. Ficheros secuenciales de texto

**BufferedReader y PrintWriter**

**Escritura**

**Sintaxis:**

PrintWriter salida;

salida =new PrintWriter(new FileWriter(nombre))

FileWriter (nombre, añadir)

**Excepciones que lanza el constructor FileWriter**: IOException

**Métodos:**

println()

print()

close():IOException

**Lectura**

**Sintaxis:**

BufferedReader entrada;

entrada= new BufferedReader(new FileReader(nombre));

**Excepciones que lanza el constructor FileReader:** FileNotFoundException

**Métodos:**

readLine(): null

read(): -1

BufferedReader entrada=new BufferedReader(

new FileReader(nombre);

char car=(char)(entrada.read());

**Excepciones que lanzan los métodos:** IOException

close():IOException

**Scanner y PrintWriter (a partir de la versión 5.0 de Java)**

**Escritura**

**Sintaxis:**

PrintWriter salida;

salida =new PrintWriter(new FileWriter(nombre))

FileWriter (nombre, añadir)

**Excepciones que lanza el constructor FileWriter**: IOException

**Métodos:**

println()

print()

printf()

close():IOException

**Lectura**

**Sintaxis:**

Scanner entrada;

entrada= new Scanner (new FileReader(nombre)); o

entrada=new Scanner (new File(nombre));

entrada=new Scanner (nombre);

**Excepciones que lanza el constructor:** FileNotFoundException

**Métodos:**

useLocale (Locale.US)

next (),nextLine()

nextInt(), nextDouble(), nextFloat(),…

hasNextInt (), hasNextDouble (), hasNextFloat (), ….

close()

**Escritura**

**BufferedWriter**

BufferedWriter bw;

bw= new BufferedWriter(new FileWriter("E:\\fichero1.txt"));

**Métodos:**

flush();

write();

newline();

close();

## 3. Ficheros secuenciales binarios

**Byte a byte**

FileOutputStream y FileInputStream

**Escritura**

**Sintaxis:**

FileOutputStream salida;

salida = new FileOutputStream(nombre);

FileOutputStream(File Objeto\_File) ;

FileOutputStream(String nombre\_fichero, boolean añadir);

**Excepciones que lanza el constructor:**

FileNotFoundException

**Métodos:**

write(int i): IOException

close(): IOException

**Lectura**

**Sintaxis:**

FileInputStream entrada;

entrada=new FileInputStream(nombre);

entrada=FileInputStream(objeto\_File);

**Excepciones que lanza el constructor:**

FileNotFoundException

**Métodos**:

read() : IOException

close(): IOException

**Datos pasados a byte**

DataOutputStream y DataInputStream

**Escritura**

**Sintaxis:**

DataOutputStream salida;

salida=new DataOutputStream(new FileOutputStream(nombre));

DataOutputStream salida=new DataOutputStream

(new BufferedOutputStream (new FileOutputStream(nombre)));

**Excepciones que lanza el constructor:** Las del FileOutputStream

**Métodos:**

writeInt(variable\_tipo\_entero)

writeUTF(objeto\_tipo\_cadena)

writeDouble(variable\_tipo\_doble)

writeFloat(variable\_tipo\_float)

writeChar(variable\_tipo\_carácter)

writeBoolean, writeByte, writeLong, writeShort, etc.

**Excepciones que lanzan:** IOException

close():IOException

**Lectura**

**Sintaxis:**

DataInputStream entrada;

entrada=new DataInputStream(new FileInputStream(nombre));

DataInputStream entrada=new DataInputStream(new

BufferedInputStream(new FileInputStream(nombre)));

**Excepciones que lanza el constructor:** Las del FileInputStream

**Métodos:**

readChar(), readDouble(), readInt(), readFloat(), readUTF()

readBoolean(), readByte(), readShort(), readLong(), available() etc.

**Excepciones que lanzan los métodos**: EOFException y IOException

close():IOException

**NOTA**: BufferedInputStream y BufferedOutputStream son similares a BufferedReader y BufferedReader

## 4. Ficheros de acceso directo

**Sintaxis**

RandomAccessFile(File objeto\_fichero, String modo)

RandomAccessFile(String nombre, String modo)

*modo*: r (*read*) y rw (*read-write*)

**Excepciones que lanza el constructor:** FileNotFoundException

**Métodos:**

void seek(long posición)

long getFilePointer()

int skipBytes(int desplazamiento)

long length()

**Excepciones que lanzan los métodos:** IOException

close(): IOException

**Escritura**

RandomAccessFile salida;

salida=new RandomAccessFile(nombre,”rw”);

**Métodos:**

writeInt(entero), writeDouble(doble), writeBytes(cadena)

writeUTF(String), etc

**Excepciones que lanzan los métodos:** IOException

close(): IOException

**Lectura**

RandomAccessFile entrada;

entrada=new RandomAccessFile(nombre, “r”);

**Métodos:**

readInt(), readDouble(), readUTF(), readFloat(),readShort(), etc.

**Excepciones que lanzan los métodos:** EOFException y IOException

**Movimiento en un fichero:**

posicion=n\*l\_registro;

## 5. Ficheros y objetos

class Ejemplo implements Serializable {

--- Código para la clase Ejemplo ---

}

**Creación de un *stream* de objetos para salida:**

ObjectOutputStream salida;

salida=new ObjectOutputStream(new FileOutputStream(nombre));

**Creación de un *stream* de objetos para entrada:**

ObjectInputStream entrada;

entrada=new ObjectInputStream(new FileInputStream(nombre));

**Métodos:**

writeObject(Objeto)

readObject()

**Ejemplo:**

ObjectOutputStream salida;

salida=new ObjectOutputStream (new FileOutputStream(nombre));

salida.writeObject(obj1);

ObjectInputStream entrada;

entrada=new ObjectInputStream(new FileInputStream(nombre));

obj2=(Ejemplo)entrada.readObject();

# UT9 Ejercicios Ficheros de Texto

**1.-** Con un programa en java crea un fichero denominado parrafo.txt. Escribe en él las líneas que vaya introduciendo el usuario, terminando cuando escriba FIN.

**2.-** Con un programa en java lee el fichero de texto anterior parrafo.txt, y genera otro fichero que sea el inicial pero sin las vocales. Muestra por pantalla ambos ficheros.

**3.-** Crea manualmente dos ficheros f1.txt y f2.txt, que tendrán un solo carácter por línea y que estarán ordenados alfabéticamente, y escribe un programa en java que genere el fichero f3.txt que será la fusión de f1 y f2 manteniendo su ordenación. Escribir por pantalla el fichero resultante. (Supuestamente los 2 ficheros están ordenados)

Fichero1 Fichero2 Fichero3  
A B A  
F D B  
G D  
 F  
 G

**4.-** Haz programa en java que lea el fichero parrafo.txt y lo codifique sumándole a cada letra un valor que se pedirá por teclado, guardarlo en otro fichero denominado codificado.txt. Sacar por pantalla el nuevo fichero creado.

**5.-** Genera un fichero llamado nombres.txt en el que hay una lista de personas, con su nombre y apellidos (datos de cada persona en una línea diferente). Poner fin para terminar.

**6.-** Lee el fichero nombres.txt, generado en el ejercicio anterior, y calcula la longitud media de los nombres completos (nombre + apellido). Muestra el contenido de dicho fichero por pantalla de forma que para cada pareja de nombre y apellido se muestre su longitud y al final la media de las longitudes de los nombres+apellido.

**7.-** Como el 6, pero ahora calcula la longitud media de los nombres y apellidos por separado (se leen juntos en una línea y se separan usando el método split()).

# UT9 Ejercicios Ficheros Binarios

**1.-** Crea una aplicación que pida por teclado la cantidad de números aleatorios enteros positivos que se van a guardar en un fichero binario. El rango de los números aleatorios estará entre 0 y 100, incluyendo el 100.

Cada vez que se ejecute la aplicación se añadirán números al fichero sin borrar los anteriores.

Muestra en pantalla el fichero creado, después de realizar las operaciones de escritura sobre él.

**2.-** Crea una aplicación que almacene los datos básicos de un vehículo: matricula (String), marca (String), tamaño de depósito (double) y modelo (String) en un fichero binario.

Los datos se pedirán por teclado. Se irán añadiendo al fichero los datos de nuevos vehículos cada vez que ejecutemos la aplicación (no se sobrescriben los datos).

Cada vez que se modifica el fichero, se muestran por pantalla los datos de cada coche.

Utiliza la clase JOptionPane para leer los datos de entrada (showInputDialog) y mostrarlos (showMessageDialog).

**3.-** Realiza un programa que lea de teclado los siguientes datos y los almacene en un fichero binario llamado "datosbeca.bin".

* Nombre y apellido de un supuesto becario.
* Género (M-F)
* Edad (20-60)
* Numero de suspensos del curso anterior (0-4).
* Residencia familiar (SI o NO)
* Ingresos anuales de la familia.

**4.**- Realiza un programa que, partiendo del fichero binario “datosbeca.bin”, calcule la cuantía de la beca (en caso de que la haya). El total de la beca se calcula como sigue:

* Base fija de 1500€
* Si los ingresos anuales de la familia son menores o iguales a la media (12.000€), la beca se incrementa en 500€, en caso contrario no lleva complementos.
* Si la edad de la persona es inferior a 23 años, 200€ de gratificación, si es mayor no hay gratificación.
* Si no hay suspensos en el curso anterior, hay una gratificación de 500€, 1 suspenso 200€. Si hay 2 o más suspensos no hay beca.
* Si vive de alquiler (no residencia familiar), 1000€ más de gratificación.

Visualiza el nombre de cada uno de los becarios y su cuantía total (solo los que tienen beca).

# UT9 Ejercicios Ficheros Serializables

**1.-** Realiza un programa en JAVA en el que se le pida al usuario notas de asignaturas y se guarden en un fichero cuando ya se hayan introducido todas las notas.

Posteriormente se leerá el fichero y se calculará la nota media del curso.

Cada una de las asignaturas será un objeto que se encuentra en una lista, y cuyos atributos serán el nombre y la nota.

El usuario establecerá el nombre de la asignatura mediante un método que pedirá el nombre de la asignatura. El atributo nota, también será el usuario quien lo introduzca mediante un método que controle que la nota tenga un valor entre 0 y 10.

***Ejemplo de ejecución:***

Por favor, introduzca el nombre de la asignatura: Programación.

Introduzca la nota de Programación: 8,5

¿Desea introducir otra asignatura? SI

Por favor, introduzca el nombre de la asignatura: Lenguajes de Marcas.

Introduzca la nota de Lenguajes de Marcas: 9

¿Desea introducir otra asignatura? SI

Por favor, introduzca el nombre de la asignatura: Bases de Datos

Introduzca la nota de Bases de Datos: 8

¿Desea introducir otra asignatura? NO

\*\*\*\*\* Notas almacenadas en la lista \*\*\*\*\*

……………… Volcando la lista al fichero…………….

\*\*\*\*\* Volcado finalizado con éxito \*\*\*\*\*

………Leyendo el fichero (mostramos los datos) y calculando media…………….

Su nota media del curso es de: 8,5

**2.-** Se trata de hacer una aplicación en Java que gestione los clientes de una empresa. Los datos de los clientes, se almacenarán en un fichero serializado, denominado clientes.dat.

Los datos que se almacenarán sobre cada cliente son:

NIF.

Nombre.

Teléfono.

Dirección.

Deuda.

Mediante un menú se podrán realizar determinadas operaciones:

**Añadir cliente**. Esta opción pedirá los datos del cliente y añadirá el registro correspondiente en el fichero.

**Listar clientes**. Recorrerá el fichero mostrando los clientes almacenados en el mismo.

**Buscar clientes**. Pedirá al usuario el nif del cliente a buscar, y comprobará si existe en el fichero, mostrando sus datos.

**Borrar cliente.** Pedirá al usuario el nif del cliente a borrar, y si existe, lo borrará del fichero.

**Borrar fichero de clientes** completamente. Elimina del disco el fichero clientes.dat

**Salir de la aplicación**.

**3.** Escribe un programa que permita almacenar en un fichero serializable una **agenda telefónica**. Cada registro de la agenda deberá contener los siguientes datos: Nombre, Apellidos, Nº de Teléfono, E-mail.

El programa deberá permitir las siguientes operaciones, mediante un menú:

**a)** Leer los datos de un nuevo registro (dar de alta), almacenándolo en el fichero.

**b)** Buscar una persona de la agenda leyendo de teclado su nombre y apellidos. Se visualizarán los datos de la persona.

**c)** Modificar el teléfono y/o e\_mail de una persona que se pedirá por teclado (por su nombre y apellido).

**d)** Eliminar una persona de la agenda telefónica dando su nombre y apellidos. Todas las personas que se eliminen del fichero binario, se guardarán en un fichero de texto, llamado Eliminados.txt, en el que se guardará un registro por línea.

**e)** Ordenar el fichero alfabéticamente por nombre y apellido

**f)** Mostrar el contenido del fichero

**g)** Finalmente, cuando se salga del menú, se escribirá en consola el contenido del fichero de texto Eliminados.txt.

# UT9 Ejercicios Ficheros Aleatorios

**1.-** Utilizando las clases de acceso directo a ficheros, realiza un programa que le muestre al usuario un menú con las siguientes opciones:

1.‐ Añadir números de tipo double al principio del fichero.

2.‐ Añadir números de tipo double al final del fichero.

3.‐ Mostrar el fichero creado.

4.‐ Sustituir un número indicado por el usuario por otro número que también te indique el usuario.

5.‐ Ordenar los números en el fichero descendentemente (ayúdate de un ArrayList).

**Nota**: Un double en JAVA ocupa 8 bytes (en Java 8 existe un método para obtener el tamaño).