



CFGS APLICACIONES MULTIPLATAFORMA - PROGRAMACIÓN



UNIDAD 8

GESTIÓN DE FICHEROS



DIEGO VALERO ARÉVALO

BASADO EN APUNTES Y EJERCICIOS PROPORCIONADOS POR
M^{re} CARMEN DÍAZ GONZÁLEZ - IES VIRGEN DE LA PALOMA

U8 - GESTIÓN DE FICHEROS

ÍNDICE

8.1 - INTRODUCCIÓN A LA GESTIÓN DE FICHEROS	1
>> 8.1.1- CLASE FILE Y LIBRERÍA JAVA.IO	1
8.1.1.1 - Información relativa a las rutas	1
>> 8.1.2- RUTAS ABSOLUTAS Y RELATIVAS	2
· Ruta absoluta y ruta relativa.	
>> 8.1.3- TRABAJANDO CON LA CLASE FILE	2
>> 8.1.4- COMPROBACIÓN DEL ESTADO DE LOS FICHEROS	4
>> 8.1.5- PROPIEDADES DE FICHEROS	4
>> 8.1.6- MANIPULACIÓN DE FICHEROS	4
>> 8.1.7- LISTADO DEL CONTENIDO DE UN FICHERO	5
8.2 - LECTURA Y ESCRITURA DE FICHEROS	5
>> 8.2.1- FICHEROS DE TEXTO	6
8.2.1.1 - Lectura de ficheros de texto	6
8.2.1.2 - Escritura de ficheros de texto	8
8.2.1.3 - Sobreescritura de ficheros de texto	9
>> 8.2.2- FICHEROS BINARIOS	11
8.2.2.1 - Cosas importantes sobre un fichero binario	11
8.2.2.2 - Lectura de ficheros binarios	11
8.2.2.3 - Escritura de ficheros binarios	13
8.2.2.4 - Sobreescritura de ficheros binarios	14
8.2.2.5 - Escritura de objetos	17
U8 - BATERÍA DE EJERCICIOS	18
Ejercicios con ficheros de texto	18
Ejercicios con ficheros binarios	19

8.1 - INTRODUCCIÓN A LA GESTIÓN DE FICHEROS

La mayoría de los sistemas operativos traen consigo una manera de **poder gestionar archivos**, ya sea de **manera gráfica e intuitiva** o a través de **líneas de comandos de texto**, pero todos se basan en la misma organización: **a través de una jerarquía de carpetas y ficheros**.

Pero esto **no es exclusivo de los S.O.**. Muchos lenguajes de programación tienen **librerías y métodos** que permiten **manejar, acceder y manipular** los mecanismos internos del sistema del explorador de archivos e incluso el contenido de los ficheros, y JAVA es uno de ellos.

>> 8.1.1- CLASE FILE Y LIBRERÍA JAVA . IO

La clase `File`, perteneciente a la librería `java.io`, permite **alojar y operar** con **archivos y directorios**. Antes de trabajar con ella, debemos importarla:

```
import java.io.File;

File nombreFichero = new File("ruta");
```

<code>File</code>	Nuestro objeto será de tipo <code>File</code> .
<code>nombreFichero</code>	Asignamos un nombre al mismo.
<code>new</code>	El operador <code>new</code> asigna durante la ejecución del programa un espacio de memoria al objeto , y almacena esta referencia en una variable.
<code>File("ruta");</code>	La clase en la que se va a basar el objeto, que será <code>File</code> . Como parámetros pasaremos la ruta (relativa) (la más recomendada) o absoluta del fichero. Podemos escribirla directamente o pasar una variable <code>String</code> .

Aunque pensemos que `File` se refiere a un archivo en concreto, esto es engañoso. `File` es un **objeto** que representa **la ruta dentro del sistema de archivos**, y nos permite trabajar con un **fichero** o un **directorio** según esta, independientemente de que **ya exista o no**. Es importante entender que sólo representa **una única ruta**. Para operar con diferentes rutas a la vez, habrá que crear y manipular **varios objetos**.

EJEMPLO

.java

```
1 File carpetaFotos = new File("C:/Fotos");
2 File foto1 = new File("C:/Fotos/Foto1.png");
3 File foto2 = new File("C:/Fotos/Foto2.png");
```

La ruta que maneja `File` no afecta a su comportamiento hasta que se utiliza, por eso hasta que no se llamen a los métodos o leamos o escribamos información no podremos saber si es una ruta correcta o no, ya que es en ese momento cuando se accede realmente al sistema de archivos.

· 8.1.1.1- INFORMACIÓN RELATIVA A LAS RUTAS

Debemos tener en cuenta que el formato de las rutas cambia dependiendo del **sistema operativo**. Afectará tanto el inicio de la ruta como los separadores de ubicación. Veamos dos ejemplos:

WINDOWS	UNIX
C:\Windows\System32	/usr/bin
Windows inicia sus rutas en una unidad .	Unix no tiene inicio, comienza con un separador .
Se usan los separadores \.	Se usan los separadores /.

Si nuestro programa va dirigido a diferentes sistemas operativos, podemos usar la constante `File.separator`, pero no la veremos de momento.

Lo bueno de JAVA es que nos permite usar el **separador** / para identificar rutas en Windows, para simplificar las cosas.

>> 8.1.2- RUTAS ABSOLUTAS Y RELATIVAS

Recuerda que para asignar la ruta a un `File` podemos usar una ruta **absoluta** o **relativa**. ¿En qué se diferencian estas dos?

RUTA ABSOLUTA	RUTA RELATIVA				
Aquella que especifica claramente dónde se encuentra el fichero.	Aquella que especifica sólo parte del directorio, sin incluir la raíz .				
C:/Fotos/Foto1.png	../Fotos/Foto1.png				
Al usar una ruta absoluta siempre hay que usar la representación correcta según el sistema en el que se ejecuta el programa. El problema de usar rutas absolutas en proyectos es que si otra persona abre un archivo <code>.java</code> en su ordenador, la referencia de los <code>File</code> debe ser exactamente la misma que en el primero, lo cual no es lo común y probablemente haya que redireccionar los ficheros a mano.	Ya que no se necesita localizar exactamente la ruta del sistema, cada uno lo debería leer igual, por lo que la notación usando / ó \ es indiferente . Al usar una ruta relativa , la búsqueda suele partir desde la ubicación donde está el programa que la llama , lo que facilita la portabilidad, ya que los ficheros pueden incluirse en el directorio del proyecto. Para especificar dónde comienza una ruta relativa, se deja en blanco o se usa <code>.</code> ó <code>..</code> . <table> <tr> <td>.</td><td>..</td></tr> <tr> <td>Indica la localización actual del programa.</td><td>Se usa para acceder a la carpeta superior.</td></tr> </table>	.	..	Indica la localización actual del programa.	Se usa para acceder a la carpeta superior .
.	..				
Indica la localización actual del programa.	Se usa para acceder a la carpeta superior .				

EJEMPLO	MiPrograma.java
<pre> v [miProyecto] v [fotos] - [foto.png] v [programa] > [MiPrograma.java] </pre>	<pre> 1 package miProyecto.programa; 2 3 import java.io.File; 4 5 public class MiPrograma{ 6 File foto = new File("../fotos/foto.png"); 7 } </pre>

En este ejemplo vemos que para llegar desde donde se ejecuta el programa hasta el archivo que necesitamos, la ruta relativa empieza con `..`, ya que este necesita **retroceder una carpeta** para poder encontrar el resto de la ruta.

Si pusiésemos `./fotos/foto.png`, el programa lo interpretaría como `programa/fotos/foto.png`. y tendríamos un error porque esa ruta **no existe**.

>> 8.1.3- TRABAJANDO CON LA CLASE FILE

Esta clase trae consigo múltiples métodos que podemos usar para trabajar con los ficheros que necesitemos.

String	<code>.getParent();</code>	Devuelve la ruta de la carpeta del elemento referido por esta ruta. En resumen, devuelve exactamente la ruta que le pasamos pero sin el último elemento .
String	<code>.getName();</code>	En este caso es al contrario que <code>getParent()</code> : este método nos devuelve sólo el nombre del último elemento , sea un directorio o archivo.
String	<code>.getAbsolutePath();</code>	Devuelve la ruta absoluta . Si el fichero se instanció con una ruta relativa, este método recopila el resto de la ruta del sistema de archivos.

EJEMPLO

Ejemplo.java

```
1 import java.io.File;
2
3 public class Ejemplo{
4     public static void main(String[] args){
5
6         File carpetaRAbsoluta = new File("F:/proyecto/fotos");
7         File archivoRAbsoluta = new File("F:/proyecto/fotos/foto.png");
8
9         File carpetaRRelativa = new File("fotos");
10        File archivoRRelativa = new File("fotos/foto.png");
11
12        informacionRutas(carpetaRAbsoluta);
13        informacionRutas(archivoRAbsoluta);
14        informacionRutas(carpetaRRelativa);
15        informacionRutas(archivoRRelativa);
16
17    }
18
19    public static void informacionRutas(File ruta){
20        System.out.println("    Ruta original: " + ruta);
21        System.out.println("        getParent(): " + ruta.getParent());
22        System.out.println("        getName(): " + ruta.getName());
23        System.out.println("getAbsolutePath(): " + ruta.getAbsolutePath());
24        System.out.println();
25    }
26 }
```

Console

```
Ruta original: F:\proyecto\fotos
getParent(): F:\proyecto
getName(): fotos
getAbsolutePath(): F:\proyecto\fotos

Ruta original: F:\proyecto\fotos\foto.png
getParent(): F:\proyecto\fotos
getName(): foto.png
getAbsolutePath(): F:\proyecto\fotos\foto.png

Ruta original: fotos
```

```

    getParent(): null
    getName(): fotos
    getAbsolutePath(): F:\programacion\proyecto\src\fotos

    Ruta original: fotos\foto.png
    getParent(): fotos
    getName(): foto.png
    getAbsolutePath(): F:\programacion\proyecto\src\fotos\foto.png

```

>> 8.1.4- COMPROBACIÓN DEL ESTADO DE LOS FICHEROS

boolean	<code>.exists();</code>	Comprueba si la ruta existe dentro del sistema de ficheros . Devolverá true si existe y false en caso contrario.
boolean	<code>.getName();</code>	Comprueba el sistema de ficheros en busca de la ruta y devuelve true si existe y es un fichero . Devolverá false si no existe , o si existe pero no es un fichero .
boolean	<code>.isDirectory();</code>	Funciona como el anterior, pero comprueba si es un directorio .

>> 8.1.5- PROPIEDADES DE FICHEROS

boolean	<code>.length();</code>	Devuelve el tamaño de un archivo en bytes . Este método sólo puede ser llamado sobre una ruta que represente un archivo , de lo contrario no se puede garantizar que el resultado sea válido.
boolean	<code>.lastModified();</code>	Devuelve la última fecha de edición del elemento representado por esta ruta. El resultado se codifica en un único número entero cuyo valor es el número de milisegundos que han pasado desde el 1 de junio de 1970 .



Si ejecutamos el programa, y posteriormente **modificamos el fichero** y **volvemos a ejecutar**, la fecha de última modificación **deberá haberse actualizado**, así como su **tamaño**.

>> 8.1.6- MANIPULACIÓN DE FICHEROS

boolean	<code>.mkdir();</code>	Crea la carpeta indicada en la ruta. La ruta debe indicar el nombre de una carpeta que no existe en el momento de invocar el método . Devuelve true si se ha creado correctamente, en caso contrario devuelve false (si la ruta es incorrecta , la carpeta ya existe o el usuario no tiene permisos de escritura).
boolean	<code>.delete();</code>	Borra el archivo o carpeta indicada en la ruta. La ruta debe indicar el nombre de un archivo o carpeta que sí existe en el momento de invocar el método . Se podrá borrar una carpeta sólo si está vacía (que no contenga ni carpetas ni archivos). Devuelve true o false según si la operación se ha podido llevar a cabo.
boolean	<code>.renameTo(File destino);</code>	El nombre de este método es algo engañoso, ya que su función no es “renombrar”, sino cambiar la ubicación completa . El método se invoca el objeto <code>File</code> con la ruta donde se encuentra el archivo o carpeta , y se le da como argumento otro objeto File con la ruta destino . Devuelve true o false según si la operación se ha podido llevar a cabo correctamente o no (la ruta origen y destino son correctos, no existe ya un archivo con este nombre en el destino,

etc.). Nótese que, en el caso de carpetas, es posible **moverlas aunque contengan archivos**.

>> 8.1.7- LISTADO DEL CONTENIDO DE UN FICHERO

Array

`.listFiles()`;

Devuelve un `File[]` con **todos los elementos contenidos en la carpeta** (representados por objetos `File`, uno por elemento). Para que se ejecute correctamente, **la ruta debe indicar una carpeta**.

El tamaño del vector será **igual al número de elementos que contiene la carpeta**. Si el tamaño es **0**, el valor devuelto será `null` y **toda operación posterior sobre el vector será errónea**

El orden de los elementos es **aleatorio** (al contrario que en el explorador de archivos del sistema operativo, no se ordena automáticamente por tipo ni alfabéticamente. Si queremos ese resultado deberemos **manipularlo** posteriormente).

En el siguiente ejemplo puedes ver cómo se usan varios de los métodos que acabamos de ver:

EJEMPLO

Ejemplo.java

```
1  import java.io.File;
2
3  public class Ejemplo{
4      public static void main(String[] args){
5
6          File miDirectorio = new File("F:/proyecto/fotos");
7          File[] lista = miDirectorio.listFiles();
8          File archivoAMostrar;
9
10         System.out.println("Contenido de " + miDirectorio.getAbsolutePath() + ":");
11
12         for(int i = 0; i < lista.length; i++){
13             archivoAMostrar = lista[i];
14             if(archivoAMostrar.isDirectory()) {
15                 System.out.println("[DIRECT.] > " + archivoAMostrar.getName());
16             }
17             else {
18                 System.out.println("[ARCHIVO] > " + archivoAMostrar.getName());
19             }
20         }
21     }
22 }
```

8.2 - LECTURA Y ESCRITURA DE FICHEROS

Al crear programas que tengan que ver con ficheros, su utilidad principal no es manejarlos a través del sistema de estos del ordenador. Lo que se suele hacer es **manejar los datos** de los mismos.

Para saber cómo se tratan los datos de un fichero en un programa, hay que tener muy claro **cómo se estructuran**. Dentro de un archivo se pueden almacenar **todo tipo de valores de cualquier tipo** de

datos. La parte más importante es que estos valores se almacenan en forma de **secuencia**, uno tras otro. Por lo tanto, la forma más habitual de tratar ficheros es **secuencialmente**.

También tenemos que tener en cuenta qué **tipo de fichero** vamos a manejar, el cual puede ser de dos tipos:

DE TEXTO	BINARIO
Los datos se representan como una secuencia de cadenas de texto , donde cada valor se diferencia del otro usando un delimitador .	Los datos se representan directamente de acuerdo a su formato en binario , sin ninguna separación .


>> 8.2.1- FICHEROS DE TEXTO

Un **fichero de texto** es aquel que está **orientado a carácter**, como el que podría generar con cualquier editor de texto simple. Los valores están almacenados según su **representación en cadena**, que se distinguen al estar separados entre ellos con un **delimitador**, que por defecto es **cualquier conjunto de espacios en blanco o salto de línea**. Aunque estos valores se puedan distribuir en líneas de texto diferentes, conceptualmente, se puede considerar que están organizados **uno tras otro, secuencialmente**, como las palabras en la página de un libro.

· 8.2.1.1- LECTURA DE FICHEROS DE TEXTO

Para poder leer un fichero de texto vamos a utilizar dos librerías de JAVA combinadas: **BufferedReader** y **FileReader**, que deberemos importar antes de poder hacer cualquier operación.

BufferedReader es un **manejador de fichero** que transforma la información dada en bytes a caracteres de lenguaje humano, el cual se declara de la siguiente manera:

 Todas las operaciones con ficheros deben incluirse en bloques **try-catch**. Esto nos permitirá mostrar **mensajes de error** y **terminar el programa de una forma ordenada** en caso de que se produzca algún **fallo**.

```
import java.io.BufferedReader;
import java.io.FileReader;

try{
    BufferedReader nombreBR = new BufferedReader(new FileReader("ruta"));
    ...
}catch(tipoError nombreError){
    ...
}
```

BufferedReader	Nuestro objeto será de tipo BufferedReader .
nombreBR	Asignamos un nombre al mismo.
new	El operador new asigna durante la ejecución del programa un espacio de memoria al objeto , y almacena esta referencia en una variable.
BufferedReader(...);	La clase en la que se va a basar el objeto, que será BufferedReader .
new FileReader("ruta")	Como parámetros del BufferedReader pasaremos un FileReader que llevará como propio parámetro la ruta del

	archivo que vamos a leer. Podemos escribirla directamente o pasar una variable <code>String</code> .
<pre>catch(tipoError nombreError) { ... }</pre>	El/Los bloque/s <code>catch</code> tendrá/n excepciones relacionadas con las operaciones al archivo.

Para poder recopilar la información interna de un fichero de texto utilizamos el siguiente método en nuestro `BufferedReader`:

<code>String</code>	<code>.readLine();</code>	Lee cada cadena de caracteres hasta encontrar un salto de línea y devuelve un <code>String</code> .
-	<code>.close();</code>	Cierra el <code>BufferedReader</code> .



Es **importante** que siempre que **dejemos de operar con la lectura** de un fichero coloquemos un método `.close()` después. Si no lo hiciéramos, el `BufferedReader` mantendría el fichero abierto durante todo el programa y podría **provocar errores o corromper el mismo**.

Podemos implementar el método `.readLine()` a través de un `while` como en el siguiente ejemplo para recopilar todo el contenido de un fichero de texto. (¡No olvides el `.close()`!):

EJEMPLO
<pre>Ejemplo.java 1 import java.io.BufferedReader; 2 import java.io.FileReader; 3 import java.io.IOException; 4 5 public class Ejemplo{ 6 public static void main(String[] args){ 7 8 String linea = null; 9 ArrayList<String> contenidoFichero = new ArrayList<String>(); 10 11 try{ 12 13 System.out.println("Leyendo archivo..."); 14 15 BufferedReader br = new BufferedReader(new FileReader("miFichero.txt")); 16 17 while((linea = br.readLine()) != null){ 18 contenidoFichero.add(linea); 19 } 20 21 br.close(); 22 System.out.println("¡Se ha terminado de leer el archivo!"); 23 } 24 catch(FileNotFoundException e){ 25 System.out.println("No se ha encontrado el archivo."); 26 } 27 catch(IOException e){ 28 System.out.println("No se ha podido leer el archivo."); 29 } 30 31 System.out.println("Mostrando el contenido..."); 32 for(String v : contenidoFichero){ 33 System.out.println(v); 34 } 35 } 36 }</pre>

El lector de archivos va leyendo automáticamente todas las líneas, separadas por el carácter interno **salto de línea**. Cuando pasa a la siguiente y no hay ninguna clase de contenido, sale del bucle y continúa con el resto del programa.

· 8.2.1.3- ESCRITURA DE FICHEROS DE TEXTO

Escribir en ficheros de texto es incluso **más fácil** que leerlos. Se pueden incluir saltos de línea, tabuladores y espacios igual que al mostrar un mensaje por pantalla. En vez de usar un **BufferedReader** y **FileReader** usaremos **BufferedWriter** y **FileWriter**. Sus declaraciones son exactamente iguales que los **Reader**, sólo debemos cambiar el tipo, por lo que aquí nos ahorramos explicar cada parte.



Todas las operaciones con ficheros deben incluirse en bloques **try-catch**. Esto nos permitirá mostrar **mensajes de error** y **terminar el programa de una forma ordenada** en caso de que se produzca algún **fallo**.

```
import java.io.BufferedWriter;
import java.io.FileWriter;

try{
    BufferedWriter nombreBW = new BufferedWriter(new FileWriter("ruta"));
    ...
}catch(tipoError nombreError){
    ...
}
```

```
.write(caracter);
```

Para **escribir** caracteres en el fichero.

```
.close();
```

Cierra el **BufferedWriter**.



Es **importante** que siempre que **dejemos de operar con la escritura** de un fichero coloquemos un método **.close()** después. Si no lo hiciéramos, el **BufferedWriter** mantendría el fichero abierto durante todo el programa y podría provocar **errores** o **corromper el mismo**.

¡IMPORTANTE!

Antes de **escribir cualquier cosa** en un fichero, debemos asegurarnos de que **la información sale como esperamos**. Si por error generamos un **bucle de escritura infinito**, es muy probable que el sistema **colapse**, ya que se estaría generando un **archivo gigante de contenido basura** que acapararía toda la memoria, pudiendo provocar que el sistema operativo no pueda ni arrancar de nuevo por falta de espacio.

Para evitar esto, es muy recomendable que **antes de escribir, saquemos la información por pantalla** para confirmar que se imprime **sólo lo que queremos** y que no hay ningún error.

EJEMPLO

Ejemplo.java

```
1 import java.io.BufferedWriter;
2 import java.io.FileWriter;
3 import java.io.IOException;
4
5 public class Ejemplo{
6     public static void main(String[] args){
7
8         try{
```

```

9
10     BufferedWriter bw = new BufferedWriter(new FileWriter("frutas.txt"));
11
12     bw.write("manzana\n");
13     bw.write("pera\n");
14     bw.write("plátano\n");
15
16     br.close();
17     System.out.println("¡Se ha terminado de escribir el archivo!");
18 }
19 catch(FileNotFoundException e){
20     System.out.println("No se ha encontrado el archivo.");
21 }
22 catch(IOException e){
23     System.out.println("No se ha podido escribir en el archivo.");
24 }
25
26 try{
27
28     System.out.println("Leyendo archivo...");
29
30     BufferedReader br = new BufferedReader(new FileReader("frutas.txt"));
31
32     while((linea = br.readLine()) != null){
33         System.out.println("> " + linea);
34     }
35
36     br.close();
37     System.out.println("¡Se ha terminado de leer el archivo!");
38 }
39 catch(FileNotFoundException e){
40     System.out.println("No se ha encontrado el archivo.");
41 }
42 catch(IOException e){
43     System.out.println("No se ha podido leer el archivo.");
44 }
45 }
46 }

```

Console

```

¡Se ha terminado de escribir el archivo!
> manzana
> pera
> plátano

```

· 8.2.1.3- SOBRESCRITURA DE FICHEROS DE TEXTO

¿Qué pasaría si quisiéramos **añadir nueva información** al fichero del ejemplo anterior? Podríamos simplemente hacer otro `.write()` nuevo y ya está, ¿no? Bueno, sí y no. Si un programa escribe en un fichero que ya existe, por defecto **sustituirá todo el contenido por el nuevo**, ya que su función es sobrescribir.

Para poder **añadir nuevas líneas al mismo**, debemos ejecutar el `FileWriter` en modo ***append***, el cual se activa **añadiendo `true` como parámetro**. Veamos las dos situaciones:

EJEMPLO

Ejemplo.java

```

1 import java.io.BufferedWriter;
2 import java.io.FileWriter;
3 import java.io.IOException;
4
5 public class Ejemplo{

```

```

6     public static void main(String[] args){
7
8         String ruta = "frutas.txt";
9
10        //Escribimos en modo append
11        try{
12            BufferedWriter bw = new BufferedWriter(new FileWriter(ruta, true));
13
14            bw.write("kiwi\n");
15
16            bw.close();
17            System.out.println("¡Se ha terminado de escribir el archivo!");
18        }
19        catch(FileNotFoundException e){
20            System.out.println("No se ha encontrado el archivo.");
21        }
22        catch(IOException e){
23            System.out.println("No se ha podido escribir en el archivo.");
24        }
25
26        leerFichero(ruta);
27
28        //Escribimos sin modo append
29        try{
30            BufferedWriter bw = new BufferedWriter(new FileWriter(ruta));
31
32            bw.write("pomelo\n");
33
34            bw.close();
35            System.out.println("¡Se ha terminado de escribir el archivo!");
36        }
37        catch(FileNotFoundException e){
38            System.out.println("No se ha encontrado el archivo.");
39        }
40        catch(IOException e){
41            System.out.println("No se ha podido escribir en el archivo.");
42        }
43
44        leerFichero(ruta);
45    }
46
47    public static void leerFichero(String ruta){
48
49        String linea;
50
51        try{
52            System.out.println("Leyendo archivo...");
53
54            BufferedReader br = new BufferedReader(new FileReader(ruta));
55
56            while((linea = br.readLine()) != null){
57                System.out.println("> " + linea);
58            }
59
60            br.close();
61            System.out.println("¡Se ha terminado de leer el archivo!\n");
62        }
63        catch(FileNotFoundException e){
64            System.out.println("No se ha encontrado el archivo.");
65        }
66        catch(IOException e){
67            System.out.println("No se ha podido leer el archivo.");
68        }
69    }
70 }

```

Console

¡Se ha terminado de escribir el archivo!

```

Leyendo archivo...
> manzana
> pera
> plátano
> kiwi
¡Se ha terminado de leer el archivo!

¡Se ha terminado de escribir el archivo!
Leyendo archivo...
> pomelo
¡Se ha terminado de leer el archivo!

```

Como puedes ver, cuando escribimos la primera vez en el archivo usando el modo *append*, la nueva información **se añade al contenido anterior**, mientras que si no usamos *append*, **se sobrescribe el archivo entero**.

>> 8.2.2- FICHEROS BINARIOS

Un **fichero binario** es aquel que **su información no es compatible o no se reconoce como texto**, sino que se almacena en **formato binario** y no es reconocible por el lenguaje humano. Ejemplos de ficheros binarios son **archivos de texto con formato, imágenes, vídeos,...**

· 8.2.2.1- COSAS IMPORTANTES SOBRE UN FICHERO BINARIO

Para comprender un poco mejor cómo trabajar con ficheros binarios, debemos tener en cuenta varias cosas:

Estos ficheros tienen **caracteres especiales** que sólo pueden ser reconocidos por las máquinas, entre ellos la **cabecera de inicio y fin de lectura**, que indica a la máquina cuándo **se puede empezar a leer el contenido** y cuándo **finaliza**. Esto lo veremos al usar el modo *append* al sobrescribir un fichero binario.

El contenido de un fichero binario guarda también el **tipo del mismo**. Esto es importante a la hora de **leer** el contenido, ya que debemos saber este tipo para poder **leerlo** y **almacenarlo** (si es *int*, *long*, *object*,...). Si no sabemos el tipo, siempre podemos ir leyendo **byte a byte** o usar un `.parse()`.

Los objetos que veremos para leer ficheros **siempre lanzan** una excepción de tipo **EOFException (End Of File)** cuando llegan a la última línea del fichero, por lo que siempre será recomendable capturarlas para mostrar un mensaje que lo deje claro en vez del error de JAVA por defecto.

· 8.2.2.2- LECTURA DE FICHEROS BINARIOS

Para poder leer un fichero binario usaremos otras dos librerías de JAVA combinadas: **ObjectInputStream** y **FileInputStream**, que deberemos importar antes de poder hacer cualquier operación.



Todas las operaciones con ficheros deben incluirse en bloques **try-catch**. Esto nos permitirá mostrar **mensajes de error** y **terminar el programa de una forma ordenada** en caso de que se produzca algún **fallo**. Las clases **ObjectStream** pueden colocarse como **parámetros de un try**, el cual **abrirá y cerrará el objeto automáticamente**, sin necesidad de ningún método `.close()`.

```

import java.io.ObjectInputStream;
import java.io.FileInputStream;

try(ObjectInputStream nombreOIS = new ObjectInputStream(new FileInputStream("ruta"));){
    ...
}catch(tipoError nombreError){

```

```
}
```

Para poder leer la información interna de un fichero de texto utilizamos este método:

```
.readX();
```

Donde **x** es el **tipo de objeto que queremos leer**. Disponemos de muchos tipos de objetos que acepta el `ObjectInputStream`:

```
.readInt();
```

```
.readDouble();
```

```
.readObject();
```

```
.readLine();
```

```
.readBoolean();
```

```
.readByte();
```

```
.readChar();
```

```
.readFloat();
```

```
.readLong();
```

Y muchos otros tipos que puedes consultar en la **API** de JAVA.



Para poder guardar el objeto que leemos, la variable debe ser del **mismo tipo** de la lectura. No podemos leer un `long` y guardarlo en un `int`. (a menos que usemos algún tipo de `.parse`, claro).



Al leer contenido de tipo `Object`, será necesario hacer un **casting explícito para poder guardar el objeto aunque la variable sea del mismo tipo**, ya que `Object` está disponible para una gran cantidad de tipos de datos, así que debemos especificar cuál es.

Recuerda que `String` es un objeto, por lo que para leerlos debemos usar `.readObject()`.

EJEMPLO

```
1 String linea = null;
2 linea = (String) entrada.readObject();
```

La manera que usemos para leer el contenido de un fichero binario dependerá de lo que **pretendamos hacer**. Podemos simplemente leer un tipo y mostrarlo, o quizás queremos guardarlo todo en una colección. Una manera básica es guardar la información durante un bucle infinito. Aunque sea infinito, como al pasar de la última línea lanzará una `EOFException`, saldrá del bucle.

EJEMPLO

Ejemplo.java

```
1 import java.io.ObjectInputStream;
2 import java.io.FileInputStream;
3 import java.io.IOException;
4
5 public class Ejemplo{
6     public static void main(String[] args){
7
8         String linea = null;
9         ArrayList<String> contenidoFichero = new ArrayList<String>();
10
11         try(ObjectInputStream entrada = new ObjectInputStream(new
12             FileInputStream("frase.dat"))){
13
14             while(true){
15                 System.out.println(entrada.readObject());
16             }
17
18         }
19         catch(EOFException e){
20             System.out.println(";Se ha terminado de leer el archivo!");
21         }
22         catch(ClassNotFoundException e){
23             e.printStackTrace();
24         }
```

```

24     }
25     catch(IOException e){
26         System.out.println("No se ha podido leer el archivo.");
27     }
28 }
29 }

```

· 8.2.2.3- ESCRITURA DE FICHEROS BINARIOS

A la hora de escribir un fichero binario cambiaremos el **Input** por **Output**: **ObjectOutputStream** y **FileOutputStream**, que deberemos importar antes de poder hacer cualquier operación.



Todas las operaciones con ficheros deben incluirse en bloques **try-catch**. Esto nos permitirá mostrar **mensajes de error** y **terminar el programa de una forma ordenada** en caso de que se produzca algún **fallo**. Las clases Stream pueden colocarse como **parámetros de un try**, el cual abrirá y cerrará el objeto automáticamente, sin necesidad de ningún método **.close()**.

```

import java.io.ObjectOutputStream;
import java.io.FileOutputStream;

try(ObjectOutputStream nombreOIS = new ObjectOutputStream(new FileOutputStream("ruta")); {
    ...
}catch(tipoError nombreError) {
    ...
}

```

Para poder escribir información en un fichero de texto utilizamos este método:

```
.writeX(dato);
```

Donde **x** es el **tipo de objeto que queremos escribir** y pasaremos por parámetro este **objeto**. Disponemos de muchos tipos que acepta el **ObjectOutputStream**:

<code>.writeInt(dato);</code>	<code>.writeDouble(dato);</code>	<code>.writeObject(dato);</code>
<code>.writeBoolean(dato);</code>	<code>.writeByte(dato);</code>	<code>.writeChar(dato);</code>
<code>.writeFloat(dato);</code>	<code>.writeLong(dato);</code>	<code>.writeShort(dato);</code>

Y muchos otros tipos que puedes consultar en la **API** de JAVA.

EJEMPLO

Ejemplo.java

```

1  import java.io.ObjectOutputStream;
2  import java.io.FileOutputStream;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5
6  public class Ejemplo{
7      public static void main(String[] args){
8
9          String linea = "Esta es mi frase";
10
11         try(ObjectOutputStream salida = new ObjectOutputStream(new
12             FileOutputStream("frase.dat"))){
13
14             salida.writeObject(linea);
15
16         }
17         catch(FileNotFoundException e){
18             System.out.println("No se ha encontrado el archivo.");
19         }
20     }
21 }

```

```

19     }
20     catch(IOException e){
21         System.out.println("No se ha podido leer el archivo.");
22     }
23 }
24 }

```



Al trabajar con ficheros, si tenemos **varios** podemos incluir como parámetros del `try` **todos los que necesitemos**, separados por `;`. Recuerda que son **declaraciones de objetos**.

· 8.2.1.4- SOBRESCRITURA DE FICHEROS BINARIOS

Al escribir ficheros binarios también podemos usar el modo **append** para añadir información sin sobrescribir la que ya existe, pero debemos tener una cosa en cuenta: ¿recuerdas que al inicio de este apartado hablábamos de que los ficheros binarios tienen **caracteres especiales**, y que uno de ellos es la **cabecera de inicio y fin de lectura**?

Pues bien, cada archivo sólo puede tener **uno de este conjunto** de cabeceras. Cualquier cabecera siguiente a las primeras **dará error de lectura a partir de ella**. El problema es que aunque usemos **append**, cada vez que añadimos nueva información, `ObjectOutputStream` genera una de estas cabeceras a continuación de la última, por lo que hay que evitar que se creen. ¿Cómo?

Dentro de la clase `ObjectOutputStream` existe un **método** que se encarga de este proceso, llamado `writeStreamHeader()`. Este método está automatizado y siempre se llama al **abrir y cerrar un archivo cuando se edita**. Para que no genere cabeceras al hacer **append**, debemos **crear nuestra propia clase que herede de `ObjectOutputStream` y sobrescribir este método para dejarlo vacío**. Heredar de `ObjectOutputStream` hace que tengamos que **implementar un constructor por defecto**, pero no tenemos que editarlo.

EJEMPLO

`MyObjectOutputStream.java`

```

1  import java.io.ObjectOutputStream;
2  import java.io.OutputStream;
3  import java.io.IOException;
4
5  public class MyObjectOutputStream extends ObjectOutputStream{
6
7      public MyObjectOutputStream (OutputStream out) throws IOException{
8          super(out);
9      }
10
11     @Override
12     protected void writeStreamHeader() throws IOException{
13     }
14 }

```

Una vez creada esta clase, deberemos usarla cuando queramos hacer **append** a un **archivo que ya existe** (`.exists()`). Si no, podemos usar la clase normal. Mira este ejemplo con la clase que acabamos de crear (`MyObjectOutputStream`) para añadir palabras a un fichero, con una **primera versión** donde no la usamos y su ejecución, y una **segunda versión** donde nos aseguramos de que sólo haya una cabecera:

EJEMPLO

`EjemploBinarioAppendSinClasePersonalizada.java`

```

1  import java.io.File;
2  import java.io.ObjectInputStream;

```



```

3 import java.io.FileInputStream;
4 import java.io.ObjectOutputStream;
5 import java.io.FileOutputStream;
6 import java.io.FileNotFoundException;
7 import java.io.IOException;
8
9 public class EjemploBinarioAppendSinClasePersonalizada{
10     public static void main(String[] args){
11
12         String linea = "Hola";
13         File archivo = new File("frase.dat");
14
15         try(ObjectOutputStream salida = new ObjectOutputStream(new FileOutputStream(archivo, true))){
16
17             salida.writeObject(linea);
18
19         }
20         catch(FileNotFoundException e){
21             System.out.println("No se ha encontrado el archivo.");
22         }
23         catch(IOException e){
24             System.out.println("No se ha podido leer el archivo.");
25         }
26
27         try(ObjectInputStream entrada = new ObjectInputStream(new FileInputStream(archivo))){
28
29             while(true){
30                 System.out.println(entrada.readObject());
31             }
32
33         }
34         catch(FileNotFoundException e){
35             System.out.println("No se ha encontrado el archivo.");
36         }
37         catch EOFException e){
38             System.out.println("¡Se ha terminado de leer el archivo!");
39         }
40         catch(ClassNotFoundException e){
41             e.printStackTrace();
42         }
43         catch(IOException e){
44             System.out.println("No se ha podido leer el archivo.");
45         }
46     }
47 }

```

Console
(primera ejecución)

Hola
¡Se ha terminado de leer el archivo!

Console
(segunda ejecución)

Hola
java.io.StreamCorruptedException: invalid type code: AC
 at java.base/java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1764)
 at java.base/java.io.ObjectInputStream.readObject(ObjectInputStream.java:509)
 at java.base/java.io.ObjectInputStream.readObject(ObjectInputStream.java:467)
 at u9.ficherosBinarios.Ej00BinarioAppend.main(EjemploBinarioAppendSinClasePersonalizada.java:30)

Ahora vamos a cambiar el programa implementando la clase que hemos creado:

EJEMPLO

EjemploBinarioAppendConClasePersonalizada.java

```

1 import java.io.File;
2 import java.io.ObjectInputStream;

```

```

3 import java.io.FileInputStream;
4 import java.io.ObjectOutputStream;
5 import java.io.FileOutputStream;
6 import java.io.FileNotFoundException;
7 import java.io.EOFException;
8 import java.io.IOException;
9
10 public class EjemploBinarioAppendConClasePersonalizada{
11     public static void main(String[] args){
12
13         String linea = "Hola";
14         File archivo = new File("frase.dat");
15
16         if(archivo.exists()) {
17             try(MyObjectOutputStream salida = new MyObjectOutputStream(new FileOutputStream(archivo, true))){
18
19                 salida.writeObject(linea);
20
21             }
22             catch(FileNotFoundException e){
23                 System.out.println("No se ha encontrado el archivo.");
24             }
25             catch(IOException e){
26                 System.out.println("No se ha podido leer el archivo.");
27             }
28         }
29         else {
30             try(ObjectOutputStream salida = new ObjectOutputStream(new FileOutputStream(archivo, true))){
31
32                 salida.writeObject(linea);
33
34             }
35             catch(FileNotFoundException e){
36                 System.out.println("No se ha encontrado el archivo.");
37             }
38             catch(IOException e){
39                 System.out.println("No se ha podido leer el archivo.");
40             }
41         }
42
43         try(ObjectInputStream entrada = new ObjectInputStream(new FileInputStream(archivo))){
44
45             while(true){
46                 System.out.println(entrada.readObject());
47             }
48         }
49
50         catch(FileNotFoundException e){
51             System.out.println("No se ha encontrado el archivo.");
52         }
53         catch(EOFException e){
54             System.out.println("¡Se ha terminado de leer el archivo!");
55         }
56         catch(ClassNotFoundException e){
57             e.printStackTrace();
58         }
59         catch(IOException e){
60             System.out.println("No se ha podido leer el archivo.");
61         }
62     }
63 }
64 }

```

Console
(primera ejecución)

Hola
¡Se ha terminado de leer el archivo!

Console

(segunda ejecución)

```
Hola  
Hola  
¡Se ha terminado de leer el archivo!
```

Aquí puedes comprobar cómo hacemos que **si el archivo ya existe, use nuestra clase personalizada** donde el método `writeStreamHeader()` no hace nada. En cambio, **si no existe** utiliza la **clase normal** para escribir la cabecera que necesita el archivo binario.

· 8.2.2.5- ESCRITURA DE OBJETOS

Es importante que tengamos en cuenta que si vamos a meter **objetos** en un fichero binario, **la clase que los define debe tener implementada la interfaz `Serializable`**, para que al leerlo de vuelta en cualquier momento JAVA sepa que se trata de un objeto **que podemos serializar y editar sus atributos**. Si no implementamos `Serializable`, el programa lanzará un error al tratar de leer objetos de un fichero binario.

U8 - BATERÍA DE EJERCICIOS

>> EJERCICIOS CON FICHEROS DE TEXTO

Para probar estos ejercicios utilizar el archivo “Documentos.zip”. Descárgalo del repositorio y descomprímelo en la carpeta de cada proyecto que crees

1	<p>Implementa un programa que pida al usuario introducir por teclado una ruta del sistema de archivos (por ejemplo, “C:/Windows” o “Documentos”) y muestre información sobre dicha ruta (ver función más abajo). El proceso se repetirá una y otra vez hasta que el usuario introduzca una ruta vacía (tecla intro). Deberá manejar las posibles excepciones.</p> <p>Necesitarás crear la función void muestraInfoRuta(File ruta) que dada una ruta de tipo File haga lo siguiente:</p> <ul style="list-style-type: none">• Si es un archivo, mostrará por pantalla el nombre del archivo.• Si es un directorio, mostrará por pantalla la lista de directorios y archivos que contiene (sus nombres). Deberá mostrar primero los directorios y luego los archivos.• En cualquier caso, añade delante del nombre la etiqueta [*] o [A] para indicar si es un directorio o un archivo respectivamente.• Si el path no existe lanzará un FileNotFoundException.
2	<p>Partiendo de una copia del programa anterior, modifica la función muestraInfoRuta:</p> <ul style="list-style-type: none">• En el caso de un directorio, mostrará la lista de directorios y archivos en orden alfabético. Es decir, primero los directorios en orden alfabético y luego los archivos en orden alfabético. Te será útil Arrays.sort().• Añade un segundo argumento ‘boolean info’ que cuando sea ‘true’ mostrará, junto a la información de cada directorio o archivo, su tamaño en bytes y la fecha de la última modificación. Cuando ‘info’ sea ‘false’ mostrará la información como en el ejercicio anterior.
3	<p>Implementa un programa que haga lo siguiente:</p> <ul style="list-style-type: none">• Cambiar el nombre de la carpeta ‘Documentos’ a ‘DOCS’, el de la carpeta ‘Fotografías’ a ‘FOTOS’ y el de la carpeta ‘Libros’ a ‘LECTURAS’• Cambiar el nombre de todos los archivos de las carpetas FOTOS y LECTURAS quitándole la extensión. Por ejemplo, ‘astronauta.jpg’ pasará a llamarse ‘astronauta’.
4	<p>Implementa un programa que cree, dentro de ‘Documentos’, dos nuevas carpetas: ‘Mis Cosas’ y ‘Alfabeto’. Mueve las carpetas ‘Fotografías’ y ‘Libros’ dentro de ‘Mis Cosas’. Luego crea dentro de ‘Alfabeto’ una carpeta por cada letra ‘A’, ‘B’, ‘C’.</p>
5	<p>Escribe un programa que guarde en un fichero con nombre primos.txt los números primos que hay entre 1 y 500.</p>
6	<p>Realiza un programa que lea el fichero creado en el ejercicio anterior y que muestre los números por pantalla.</p>
7	<p>Implementa un programa que muestre por pantalla los valores máximos y mínimos del archivo ‘numeros.txt’.</p>
8	<p>Implementa un programa que pida al usuario un nombre de archivo A para lectura y otro nombre de archivo B para escritura. Leerá el contenido del archivo A (por ejemplo ‘usa_personas.txt’) y lo escribirá ordenado alfabéticamente en B (por ejemplo ‘usa_personas_sorted.txt’).</p>
9	<p>El archivo ‘alumnos_notas.txt’ contiene una lista de 10 alumnos y las notas que han obtenido en cada asignatura. El número de asignaturas de cada alumno es variable. Implementa un programa que muestre por pantalla la nota media de cada alumno junto a su nombre y apellido, ordenado por nombre de mayor a menor.</p>

10	Implementa un programa que lea un documento de texto y muestre por pantalla algunos datos estadísticos: nº de líneas, nº de palabras, nº de caracteres y cuál es la palabra o palabras (si hay varias que se repitan el mismo número de veces) más común (y cuántas veces aparecen). Prueba el programa con los archivos de la carpeta 'Libros'. (Se recomienda usar un HashMap para la cuenta de las palabras).
11	Crear una copia del archivo 'numeros.txt' e introducir en cualquier lugar del mismo varias líneas que no sean números. Repetir el Ejercicio 7, haciendo que funcione aunque el archivo contenga datos erróneos.
12	Escribe un programa que guarde en un fichero el contenido de otros dos ficheros, de tal forma que en el fichero resultante aparezcan las líneas de los primeros dos ficheros mezcladas, es decir, la primera línea será del primer fichero, la segunda será del segundo fichero, la tercera será la siguiente del primer fichero, etc. Hay que tener en cuenta que los ficheros de donde se van cogiendo las líneas pueden tener tamaños diferentes.
13	Escribe un programa capaz de quitar los comentarios de un programa de Java.
14	Escribe un programa que lea un archivo que contiene productos, cada producto tendrá nombre, cantidad y precio. Se deberá guardar en un objeto de tipo Producto, y mostrar todos los productos por pantalla, haciendo uso del método toString() de la clase. Si de algún producto no es posible recuperar alguna de sus propiedades no se mostrará, aparecerá un error, pero el programa debe continuar mostrando el resto de productos.
15	Escribe un programa que guarde productos en un fichero, las características de los mismos se solicitarán al usuario por consola. El programa nunca borrará productos que ya contenga el archivo, los añadirá a los existentes, y si el archivo no existe se creará. No está permitido leer el archivo en este programa, sólo escribirlo. El archivo generado tendrá un formato que permita el ejercicio 14 leerlo.
16	Crear un programa que dada una ruta del sistema, muestre todo el contenido de la misma, y de sus directorios, hasta que no haya más subdirectorios, se recomienda utilizar recursividad.

>> EJERCICIOS CON FICHEROS BINARIOS

Realizar gestión de errores y excepciones siempre que sea necesario en todos los ejercicios.

1	Pide un valor double por consola y guárdalo en un archivo binario.
2	Abre el fichero del ejercicio anterior, lee el valor double contenido en él y muéstralo por pantalla.
3	Escribe un programa que escriba en un fichero binario una tabla de números int y después lea el contenido del archivo y muestre el contenido de la tabla por consola.
4	Introduce por teclado una frase y guárdala en un archivo binario. A continuación, recupérala y muéstrala por pantalla.
5	Implementa un programa que lea números enteros desde el fichero numeros.dat (que deberás crear previamente) y los vaya guardando en los ficheros pares.dat e impares.dat, según su paridad.
6	Implementa una aplicación que gestione una lista de nombres ordenada por orden alfabético. Al arrancar se leerá de un fichero los nombres insertados anteriormente y se pedirán nombres nuevos hasta que se introduzca la cadena «fin». Cada nombre que se introduzca deberá añadirse a los que ya había, de forma que la lista permanezca ordenada. Al terminar, se guardará en el fichero la lista actualizada.
7	Escribe un texto, línea a línea, de forma que, cada vez que se pulse Intro, se guarde la línea en un archivo binario. El proceso se termina cuando introduzcamos una línea vacía. Después el programa lee el texto completo del archivo y lo muestra por pantalla.

8	<p>Por motivos puramente estadísticos se desea llevar constancia del número de llamadas recibidas cada día en una oficina. Para ello, al terminar cada jornada laboral se guarda dicho número al final de un archivo binario. Implementa una aplicación con un menú, que nos permita añadir el número correspondiente cada día y ver la lista completa en cualquier momento.</p>
9	<p>Implementa una aplicación que permita guardar y recuperar los datos de los clientes de una empresa. Para ello, define la clase cliente, que tendrá los atributos: id (identificador de cliente), nombre y teléfono. Los objetos cliente se insertarán en una tabla. Para realizar las distintas operaciones, la aplicación tendrá el siguiente menú</p> <ol style="list-style-type: none"> 1. Añadir nuevo cliente. 2. Modificar datos. 3. Dar de baja cliente. 4. Listar los clientes. <p>La información se guardará en un fichero binario, que se cargará en la memoria al iniciar la aplicación y se grabará en disco, actualizada, al terminar.</p>
10	<p>Repite el ejercicio anterior, pero insertando los objetos cliente en un objeto Lista para object.</p>
11	<p>Implementa una aplicación que gestione los empleados de un banco. Para ello se definirá la clase Empleado con los atributos dni, nombre y sueldo. Los empleados se guardarán en un objeto de la clase Lista para objetos de la clase object. La aplicación cargará en la memoria, al arrancar, la lista de empleados desde el archivo binario empleados.dat y mostrará un menú con las siguientes opciones:</p> <ol style="list-style-type: none"> 1. Alta empleado 2. Baja empleado 3. Mostrar datos empleado 4. Listar empleados 5. Salir. Al pulsar 5, se grabará en el disco la lista actualizada y terminará el programa.
12	<p>Implementa el método, Integer [] fusionar (String fichero1, String fichero2), al que se le pasan los nombres de dos ficheros binarios que contienen dos listas ordenadas de objetos Integer, y devuelve una tabla ordenada con todos los elementos de los dos ficheros fusionados.</p>
13	<p>En una tabla de cadenas se guardan los nombres de 4 ficheros binarios. En cada uno de ellos se guarda una tabla de números enteros ordenados en sentido creciente. Implementa una aplicación donde se introduce por teclado un número entero. El programa debe determinar si ese número se halla en alguno de los 4 ficheros y, en caso afirmativo, en cuál de ellos y en qué lugar de la tabla correspondiente.</p>