

# MANEJO DE FICHEROS



Versión 1 – Agosto 2020 – © Alberto Carrera Martín

Acceso a datos - Ciclo DAM - Curso 2022-23

1. Introducción
2. Clases asociadas a las operaciones de gestión de ficheros y directorios
3. Flujos (streams)
4. Formas de acceso a los ficheros
5. Operaciones sobre ficheros
  - 5.1. Operaciones sobre ficheros secuenciales
  - 5.2. Operaciones sobre ficheros aleatorios
6. Clases para gestión de flujos de datos desde/hacia ficheros
  - 6.1. Ficheros de texto
  - 6.2. Ficheros binarios
  - 6.3. Ficheros de acceso aleatorio
7. Trabajo con ficheros XML
8. Trabajo con ficheros JSON

Actividades

Actividades voluntarias

---

## 1. Introducción

Los ficheros o archivos representan la forma de almacenamiento de los datos más elemental. En los años 80 fueron desplazados por la aparición de los gestores de bases de datos relacionales como modo de soporte para el tratamiento de la información. Aún así hay que resaltar que las bases de datos, no solo las relacionales sino de cualquier otro tipo como las orientadas a objetos, guardan toda su información (tablas, datos, índices, objetos...) dentro de ficheros.

El fichero es un conjunto de bits almacenado en un dispositivo (disco duro, pendrive...). Son el mecanismo para garantizar la persistencia de los datos, es decir, que se sigan conservando éstos tras la finalización de los programas.

Un fichero está formado por un conjunto de registros o líneas. Los registros están compuestos de un conjunto de campos relacionados, p.ej. título, autor, ISBN... para el caso de los libros de una biblioteca. Si se trata de líneas los ficheros contendrían conjuntos de caracteres (legibles o no).

Hoy en día los ficheros se utilizan como medio de intercambio de datos entre aplicaciones (ficheros XML o JSON p.ej.) o incluso para guardar la configuración de programas o servidores entre otros usos.

## 2. Clases asociadas a las operaciones de gestión de ficheros y directorios

Existen diferentes clases para gestionar directorios y ficheros, una de ellas es la clase *File* dentro del paquete *java.io*:

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

Se trata de una representación abstracta de nombres de rutas de archivos y directorios.

Para crear un objeto que represente un archivo o directorio tenemos 4 posibilidades (ver la tabla “Constructors” del enlace anterior). La que más hemos utilizado en anteriores ocasiones y que usamos en la línea 8 del ejemplo de la ilustración 1 de la siguiente página es la segunda:

```
File f = new File("C:\\WINDOWS");
```

En este caso f representará a la carpeta Windows de nuestro disco duro C:.

Hay una lista de métodos de la clase *File* en el enlace anterior (ver tabla “Method Summary”). Pasaremos a comentar algunos de ellos en el siguiente ejemplo (ilustración 1):

Imprimimos por la ventana de consola la ruta de f con el método *getPath()* (línea 9). El método *listFiles()* (línea 10) obtiene un array de objetos *File* que son los ficheros y directorios de f (la carpeta Windows del disco C). En la línea 11 comenzamos a recorrer uno a uno cada uno de esos objetos; en el caso de que sean ficheros (método *isFile()*— línea 12) imprimimos su nombre, longitud, la fecha que fue modificado, su ruta, si se puede leer y si se puede escribir (líneas 13 a 18). En caso de que se trate de un directorio, no haríamos ninguna de las acciones anteriores.

```

3 import java.io.*;
4 import java.util.*;
5
6 public class EjemploClaseFile01 {
7     public static void main (String arg[]){
8         File f = new File("C:\\WINDOWS");
9         System.out.println("Ruta: " + f.getPath());
10        File[] archivos = f.listFiles();
11        for (File archivo : archivos) {
12            if(archivo.isFile()){
13                System.out.println("Nombre " + archivo.getName());
14                System.out.println("Longitud en caracteres " + archivo.length());
15                System.out.println("Modificado " + new Date(archivo.lastModified()));
16                System.out.println("Ruta " + archivo.getPath());
17                System.out.println("Se puede escribir " + archivo.canRead());
18                System.out.println("Se puede leer " + archivo.canWrite());
19            }
20            System.out.println();
21        }
22    }
23 }

```

Ilustración 1. EjemploClaseFile01

Resultado de la ejecución de la clase anterior:

```

Ruta: C:\WINDOWS

Nombre AsChkDev.txt
Longitud en caracteres 29726
Modificado Tue Dec 30 03:53:11 CET 2014
Ruta C:\WINDOWS\AsChkDev.txt
Se puede escribir true
Se puede leer true

...

Nombre write.exe
Longitud en caracteres 11264
Modificado Tue Mar 19 05:45:54 CET 2019
Ruta C:\WINDOWS\write.exe
Se puede escribir true
Se puede leer true

```

Para obtener la misma información de los subdirectorios no tienes más que sustituir la línea 12 por esta otra:

```
if(archivo.isDirectory()){
```

y obtendrás como resultado por la consola:

```

Ruta: C:\WINDOWS
Nombre addins
Longitud en caracteres 0
Modificado Tue Mar 19 05:52:52 CET 2019
Ruta C:\WINDOWS\addins
Se puede escribir true
Se puede leer true

...

Nombre WinSxS
Longitud en caracteres 11796480
Modificado Mon Apr 20 20:07:35 CEST 2020
Ruta C:\WINDOWS\WinSxS
Se puede escribir true
Se puede leer true

```



Ya puedes realizar las actividades UD2.1 y UD2.2 del final de estos materiales. Necesitarás para ello consultar además la tabla de métodos de la clase File del enlace anterior

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

Seguimos con más ejemplos de operaciones sobre directorios y ficheros, en este caso con métodos para crearlos. En la siguiente ilustración intentamos crear un directorio de nombre *DAM2* en el directorio raíz de C, para ello creamos un objeto que represente dicho directorio (línea 7) y utilizamos el método *mkdir()* para crearlo (línea 8). Este método devuelve *true* si consigue crearlo o *false* en caso contrario (si no lo puede crear posiblemente sea porque exista otro ya con ese mismo nombre). Dentro de ese directorio intentaremos crear un archivo (línea 16) con el método *createNewFile()* que devuelve *true* o *false* si consigue crearlo o no al igual que para el caso de los directorios. Se diferencia en que este último método puede provocar la excepción *IOException* (intento de crear el fichero en una unidad de disco protegida o inexistente p.ej.) por lo que debemos agrupar las instrucciones en un bloque try-catch (líneas 16 a 26) si queremos tratarla.

```
3 import java.io.*;
4 public class EjemploClaseFile02 {
5     public static void main (String arg[]){
6         boolean resultado;
7         File directorio = new File("C:\\\\DAM2");
8         resultado=directorio.mkdir();
9         if (resultado)
10             System.out.println ("Directorio Creado");
11         else {
12             System.out.println ("No se pudo crear el directorio");
13             //Posiblemente exista
14             System.exit(-1); // Terminamos
15         }
16         try {
17             File fichero = new File ("C:\\\\DAM2\\\\Alberto.txt");
18             resultado = fichero.createNewFile();
19             if (resultado)
20                 System.out.println("Archivo creado");
21             else
22                 System.out.println("No se pudo crear el archivo");
23             //Posiblemente exista
24         }catch (IOException e) {
25             System.out.println("Se produjo el error: "+ e.getMessage());
26         }
27     }
28 }
```

Ilustración 2. EjemploClaseFile02



Ya puedes realizar la actividad UD2.3.

Necesitarás para ello consultar además la tabla de métodos de la clase File del enlace

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

### 3. Flujos (streams)

Un flujo o stream representa en Java un flujo de información, una secuencia ordenada de datos:

- Procedente de una fuente (teclado, fichero, memoria, red, etc.) o
- Dirigida a un destino (pantalla, fichero, etc.)

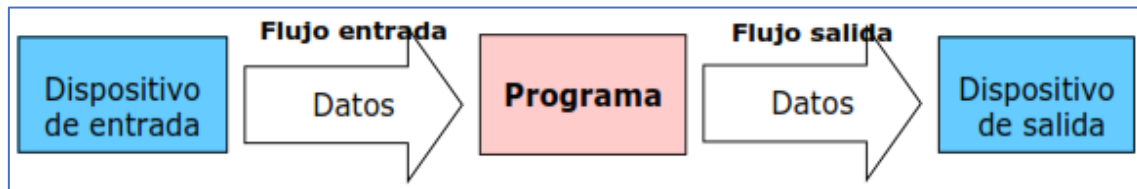


Ilustración 3. Flujos de entrada y salida

Los streams comparten una misma interfaz que hace abstracción de los detalles específicos de cada dispositivo de E/S, lo que facilita la tarea de programación al ser independiente del dispositivo. La vinculación de un stream al dispositivo físico concreto la hará el sistema de entrada/salida de Java.

Todas las clases de streams están en el paquete *java.io*

#### TIPOS DE FLUJOS

Desde el punto de vista de la representación de la información existen dos tipos de flujos:

- **Flujos de bytes:** De 8 bits, para leer y escribir datos binarios como los que se encuentran en archivos de imágenes, vídeos, programas... Las clases que trabajan con estos flujos heredan de las clases abstractas *InputStream* y *OutputStream* (y estas dos últimas a su vez de la clase *Object*).

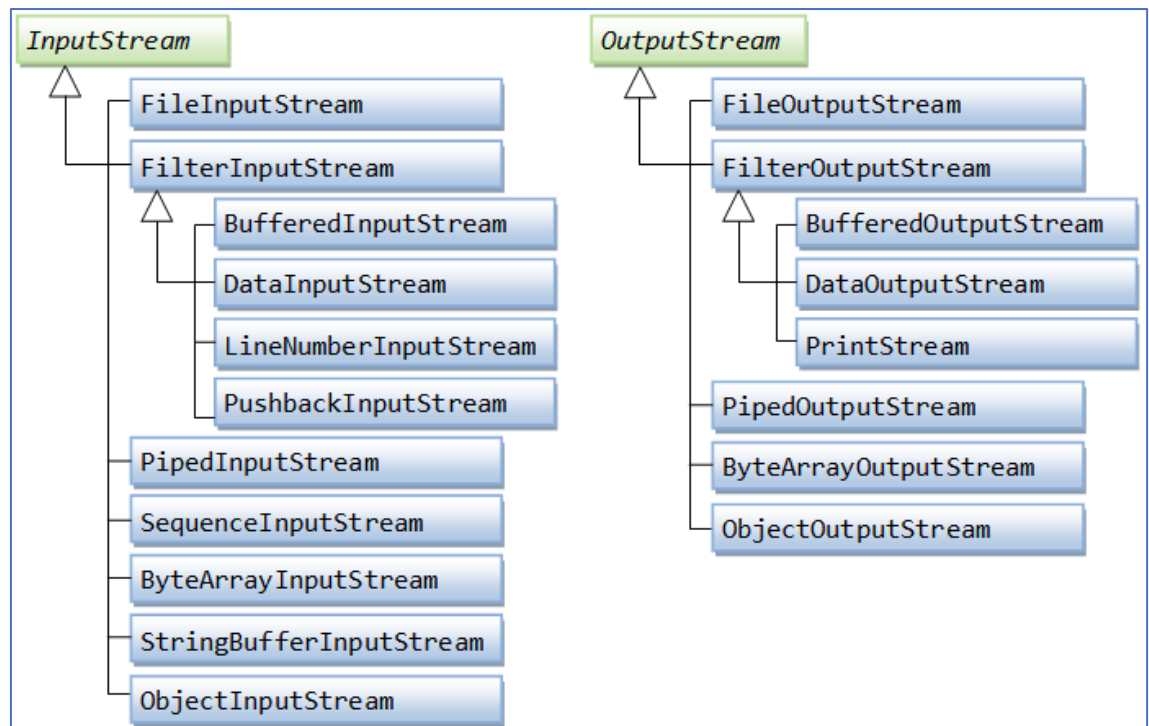


Ilustración 4. Jerarquía de clases para lectura y escritura de bytes

- **Flujos de caracteres:** De 16 bits, para leer y escribir caracteres de texto, legibles, como los que se encuentran en ficheros de texto creados con el bloc de notas de Windows. Las clases que trabajan con estos flujos heredan de las clases abstractas *Reader* y *Writer*. (y estas dos últimas a su vez de la clase *Object*)

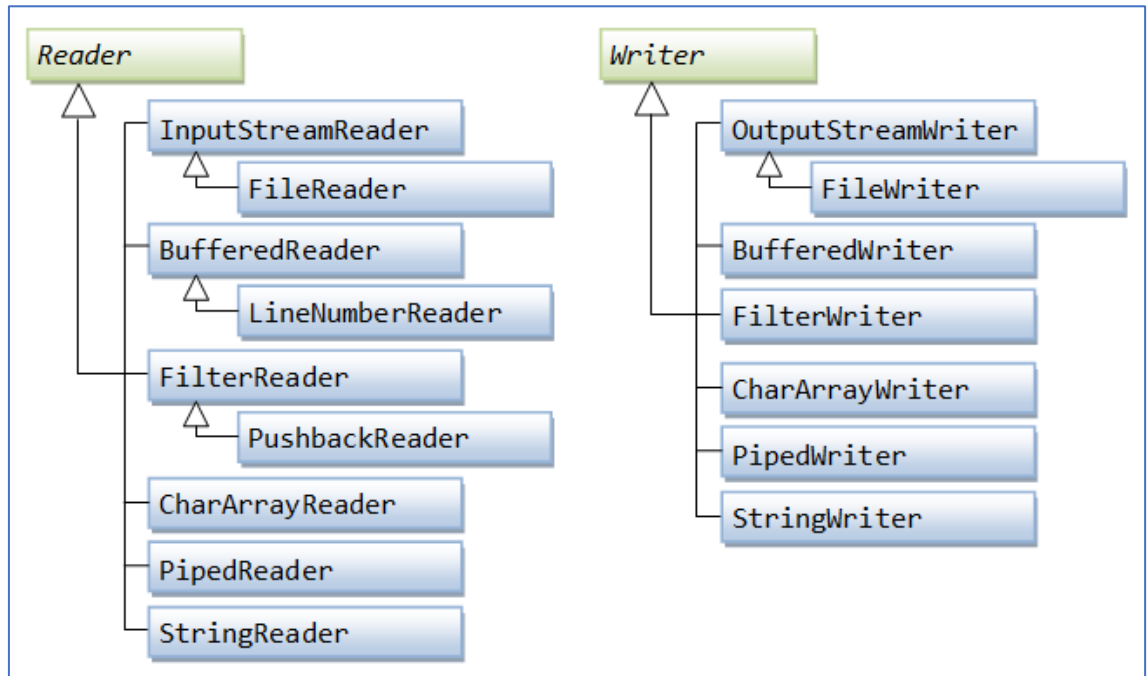


Ilustración 5. Jerarquía de clases para lectura y escritura de caracteres

Para la entrada/salida estándar (teclado y pantalla generalmente), el programador dispone de objetos sin tener que crear flujos específicos. Son atributos estáticos de la clase *java.lang.System*:

- *System.in*: Flujo de bytes de entrada. (Instancia de la clase *InputStream*).
- *System.out*: Flujo de bytes de salida. (Instancia de la clase *PrintStream*).
- *System.err*: Similar a *System.out*, utilizado para enviar mensajes de error a la consola (color rojo en lugar del negro) o a un fichero log.

De los anteriores el que usamos habitualmente para probar nuestros programas por la ventana de consola es el segundo. Un ejemplo de los otros dos aparece en la siguiente ilustración, donde se va leyendo (y contando) byte a byte del teclado hasta finalizar con un retorno de carro:

```

3 import java.io.IOException;
4
5 public class EjemploFlujosEstandar {
6     public static void main( String args[] ) throws IOException {
7         int cuentaBytes = 0;
8         while(System.in.read() != '\r' )
9             cuentaBytes++;
10        System.err.println( "Has tecleado " + cuentaBytes + " bytes");
11    }
12 }

```

<terminated> EjemploFlujosEstandar [Java Application] C:\Program Files\Java\jre1.8.0\_261\bin\javaw.exe (15-ago-2020 18:58:51 – 18:58:59)  
 ¡Viva San Lorenzo!  
 Has tecleado 18 bytes

Ilustración 6. Ejemplo de flujos estándar

## 4. Formas de acceso a los ficheros

Existen dos formas de acceder a los contenidos de los ficheros tanto para operaciones de lectura como de escritura:

- **Acceso secuencial:** Se accede a los datos comenzando desde el principio del fichero. Si se quiere leer un dato que está en la mitad del fichero es necesario recorrer antes todos los anteriores desde el principio del fichero. Para escribir un dato se hace a partir del último escrito, no es posible hacerlo en otras posiciones del fichero.
- **Acceso directo o aleatorio:** Se accede directamente a los datos. Si se quiere leer un dato que está en la mitad del fichero se accede directamente al mismo sin tener que recorrer antes todos los anteriores.

Hoy en día la totalidad de los dispositivos de almacenamiento permiten los dos tipos de acceso. Antiguamente las cintas magnéticas empleadas para copias de seguridad de datos, solo permitían el acceso secuencial.

## 5. Operaciones sobre ficheros

Independientemente de la forma de acceso, las operaciones serían:

- Creación del fichero en disco.
- Apertura del fichero. Para poder trabajar con el fichero primero hay que abrirlo, utilizando algún identificador del mismo como puede ser un objeto de una clase específica.
- Cierre del fichero. Al finalizar el programa de tratamiento y si no se va a utilizar más.
- Lectura de datos del fichero. Utilizando métodos de lectura para pasar los contenidos a variables en memoria para que puedan trabajar con ellos los programas.
- Escritura de datos en el fichero. Utilizando métodos de escritura para hacer el proceso inverso al anterior, pasar los datos de la memoria al fichero.

Las operaciones que se realizan sobre un fichero después de abrirlo son:

- Altas: Añadir nuevos registros al fichero.
- Bajas: Suprimir registros del fichero.
- Modificaciones: Cambiar algún dato de los registros.
- Consultas: Encontrar determinados registros atendiendo a unos criterios de búsqueda.

### 5.1. Operaciones sobre ficheros secuenciales

- Consultas: Para localizar un determinado registro es necesario empezar la búsqueda (lectura) desde el primer registro del fichero e ir leyendo uno a uno cada registro hasta encontrarlo (o hasta llegar al final del fichero sin haberlo localizado).
- Altas: Los registros se insertan en orden cronológico uno a continuación de otro. Solo se permiten añadir registros por el final del fichero.
- Bajas: Para dar de baja un registro primero se localiza mediante una consulta. Después de encontrarlo se pasan todos los registros anteriores y posteriores a él a un fichero auxiliar. Al final se renombra este fichero auxiliar dándole el nombre del fichero original.



- Modificaciones: Proceso muy parecido al de bajas pero escribiendo también el registro modificado en el fichero auxiliar.

Los ficheros secuenciales suelen usarse en aplicaciones que realizan copias de seguridad/restauración de datos, pues en ellas se procesan todos los registros que se encuentran en el fichero. Son útiles cuando se necesita recorrer todos los registros del fichero y porque aprovechan muy bien la utilización del espacio de almacenamiento. No son apropiados para operaciones de localización de registros o en aplicaciones que realizan con frecuencia mantenimiento (altas-bajas-modificaciones) de sus registros.

## 5.2. Operaciones sobre ficheros aleatorios

Utilizaremos la siguiente representación

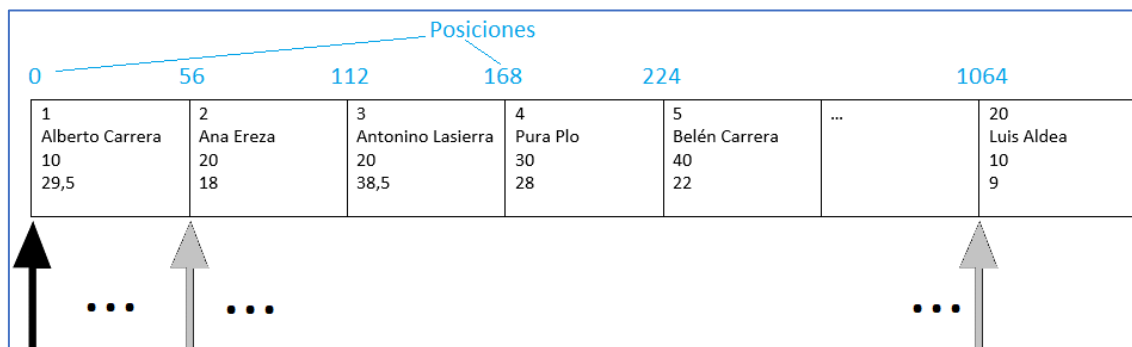


Ilustración 7. Ejemplo de fichero aleatorio

Donde cada uno de los registros tiene la siguiente estructura de campos:

Id (entero – 4 bytes)
Nombre (20 caracteres – 40 bytes)
Departamento (entero – 4 bytes)
Antigüedad (real – 8bytes)

En total cada registro ocupa 56 bytes. El primero de ellos corresponde a la persona de id 1, de nombre Alberto Carrera, departamento 10 y antigüedad en el cuerpo de profesores de 29,5 años. En total hay 6 registros en el fichero.

Al abrir el fichero el puntero asociado al mismo se sitúa en la primera posición, la 0.

Para posicionarnos en el último registro o en cualquier otro, lo haremos directamente sin tener que recorrer los anteriores como era el caso del acceso secuencial visto en el apartado anterior, situando el puntero directamente en la posición 1064 para el caso del último registro p.ej. ¿Cómo lo conseguimos? Simplemente aplicando una función de conversión (hash) al campo clave que es el id. Para ir directamente a un registro bastaría con aplicar la siguiente función de conversión que relaciona la clave (id) con el tamaño de cada registro:  $(id-1)*56$ .

De esta manera para localizar al registro de clave 3 situaríamos el puntero del fichero en la posición  $(3-1)*56 = 112$  y así para cualquier otro registro que quisiéramos localizar.

En muchas ocasiones la función de conversión no es tan sencilla como la anterior pues la clave está formada por caracteres distintos de los numéricos y puede ocurrir que para dos registros distintos nos asigne la misma posición (=¡colisión!). En estos casos suele haber una zona de

excedentes o “repetidos” (p.ej. en un segundo fichero, o en un tercero...) donde situar a estos registros de claves distintas pero con la misma posición asignada para su almacenamiento.

Operaciones sobre ficheros aleatorios:

- Consultas: Se ha comentado en los párrafos anteriores. Para encontrar un registro se aplica una función de conversión a la clave y se lee a partir de dicha posición. Si no estuviera allí entonces habría que encontrarlo en la zona de excedentes. Es importante que la función de conversión no genere muchas colisiones (repetidos) pues podría ralentizar las búsquedas.
- Altas: Aplicamos la función de conversión para obtener la dirección donde almacenarlo. Si esa posición estuviera ocupada entonces habría que guardar el registro en la zona de excedentes.
- Bajas: Se trataría de una baja no física sino “lógica”. Se localizaría la posición del registro a borrar con la función de conversión y en el campo correspondiente a la clave se pondría una marca de borrado, p.ej. un 0. De esta manera esa posición queda libre para ser ocupada por otro registro.
- Modificaciones: Una vez localizado el registro, se modifican los datos y se reescribe el registro en esa misma posición.

La gran ventaja de utilizar este acceso es la rapidez para localizar un registro para consultarlo o manipularlo. Si la función de conversión genera muchas colisiones puede ser un inconveniente pues ralentizaría todo el proceso de consulta o manipulación. Además se pierde mucho espacio de almacenamiento en disco (en el ejemplo anterior aparecen ocupadas solo 6 posiciones de 20 del fichero).

## 6. Clases para gestión de flujos de datos desde/hacia ficheros

El esquema de trabajo con cualquier stream o flujo siempre es el mismo:

Entrada de datos (leer datos)	Salida de datos (escribir datos)
1. Se crea un objeto flujo de datos de lectura	1. Se crea un objeto flujo de datos de escritura
2. Se leen datos de él con los métodos apropiados	2. Se escriben datos de él con los métodos apropiados
3. Se cierra el flujo de datos	3. Se cierra el flujo de datos
↓	
Entrada de datos (leer datos)	Salida de datos (escribir datos)
1. Abrir el flujo del archivo	1. Abrir el flujo del archivo
2. Mientras queden datos leer el siguiente dato	2. Mientras haya datos por escribir escribir en el archivo
3. Se cierra el flujo de datos	3. Se cierra el flujo de datos

*Ilustración 8. Esquema de trabajo con streams en Java*

### 6.1. Ficheros de texto

Son aquellos que están formados por caracteres legibles y que se pueden crear con cualquier editor de texto como el bloc de notas de Windows. Almacenan caracteres alfanuméricos en un determinado formato (ASCII, UTF8...).

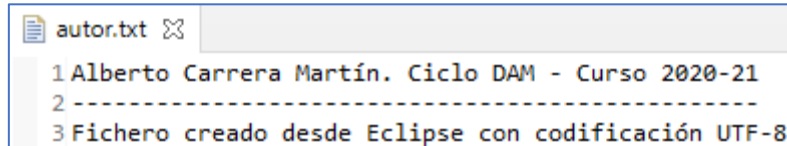
Las clases que gestionan estos flujos de caracteres son *FileReader* para leer caracteres y *FileWriter* para escribirlos (la jerarquía de clases aparecía en la ilustración 5 anterior). En la ayuda

de Eclipse y en los siguientes enlaces puedes encontrar más información de las mismas, así como de sus métodos y las posibles excepciones que generan, pues en este documento solo se explicarán los que se utilicen en los ejemplos:

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

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

Para este apartado utilizaremos el siguiente fichero como ejemplo:



```
autor.txt
1 Alberto Carrera Martín. Ciclo DAM - Curso 2020-21
2 -----
3 Fichero creado desde Eclipse con codificación UTF-8
```

Ilustración 9. Contenido del fichero autor.txt

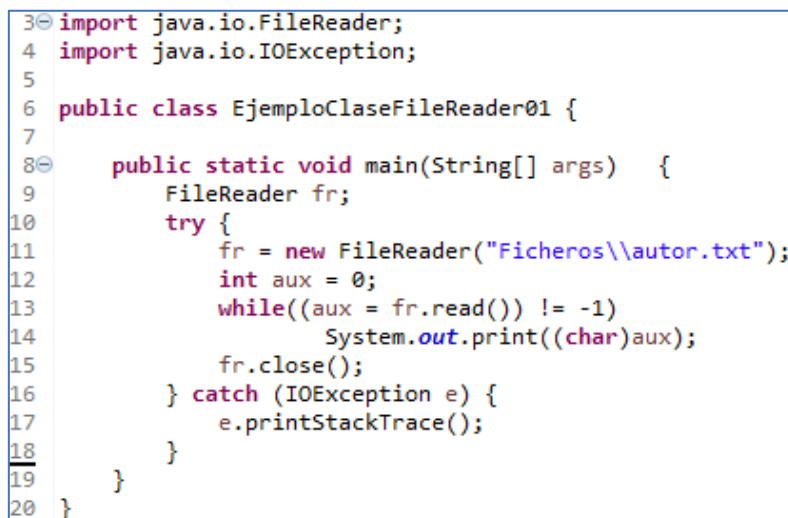
Se ha creado dentro del ide Eclipse con la opción de menú *File>New>Untitled Text File*. Una vez escrito y guardado, dentro de la pestaña *Package Explorer*, *botón derecho sobre él > Properties > Pestaña Resource > Sección Attributes > Opción Other - UTF8*. Los ficheros de las clases que trabajen con él deberán tener también la misma codificación para no visualizar caracteres extraños cuando tratemos con el carácter ñ o tildes.

En la clase de la siguiente ilustración 10 hacemos una lectura carácter a carácter del fichero anterior. Cada carácter que se va leyendo se imprime por la consola.

El esquema de trabajo es el indicado en ilustraciones anteriores:

- Se abre el flujo de datos procedente del archivo *autor.txt* (línea 11).
- Se lee el dato y se trabaja (imprime) con él (líneas 13-14) mientras no hayamos llegado al final.
- Se cierra el flujo cuando no haya más datos que leer (línea 15).

El método *read()* (línea 13) lee un carácter y devuelve su valor entero en la variable *aux*. El final del fichero viene representado por el carácter -1. Mientras no sea el final del fichero va leyendo carácter a carácter e imprimiéndolo por la consola (línea 14). Como el dato leído es realmente un entero, a la hora de imprimirlo como carácter hacemos un cast para convertirlo (*(char)aux*).

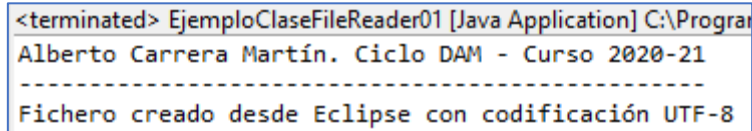


```
3 import java.io.FileReader;
4 import java.io.IOException;
5
6 public class EjemploClaseFileReader01 {
7
8     public static void main(String[] args) {
9         FileReader fr;
10        try {
11            fr = new FileReader("Ficheros\\autor.txt");
12            int aux = 0;
13            while((aux = fr.read()) != -1)
14                System.out.print((char)aux);
15            fr.close();
16        } catch (IOException e) {
17            e.printStackTrace();
18        }
19    }
20 }
```

Ilustración 10. EjemploClaseFileReader01

Se ha tratado la excepción *IOException* (líneas 16 a 18) pues los métodos *read()* y *close()* pueden provocarla. Si el fichero indicado en la línea 11 no existiera se produciría la excepción *FileNotFoundException* (“descendiente” de *IOException* y por tanto también sería tratada).

El resultado sería:



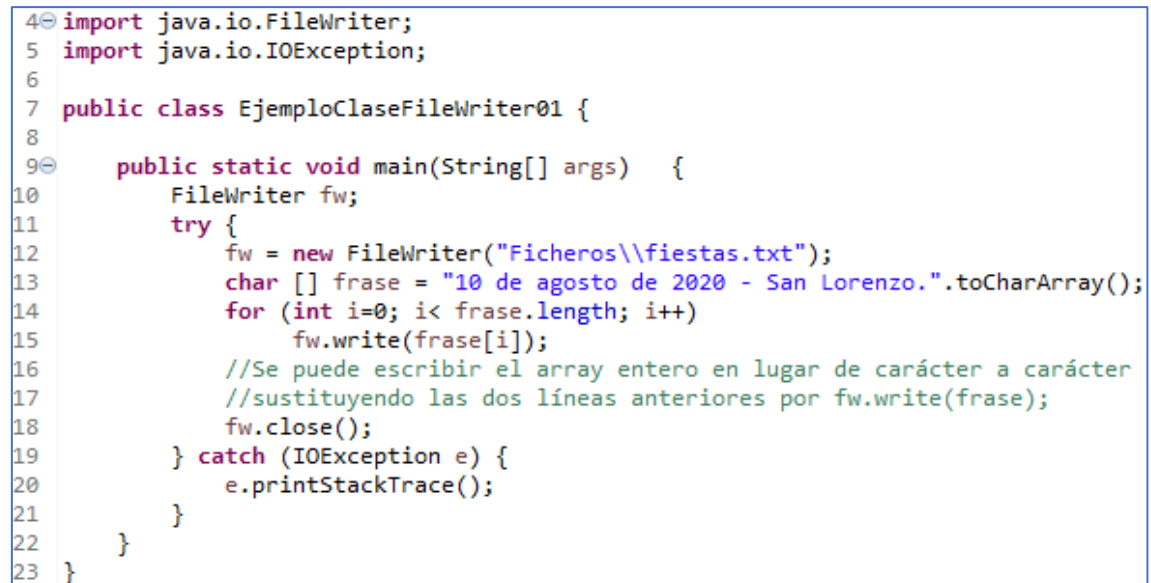
```
<terminated> EjemploClaseFileReader01 [Java Application] C:\Program Files\Java\jre7\bin\java.exe
Alberto Carrera Martín. Ciclo DAM - Curso 2020-21
-----
Fichero creado desde Eclipse con codificación UTF-8
```

Ilustración 11. Resultado de la ejecución de *EjemploClaseFileReader01*

En ejemplo de la siguiente ilustración 12 se escribe un array de caracteres de texto (construido a partir de un *String* – línea 13) a un fichero de texto (*fiestas.txt*). La escritura se realiza carácter a carácter (método *write()* - línea 15). El esquema de trabajo es el indicado en ilustraciones y párrafos anteriores:

- Se abre el flujo de datos hacia el archivo *fiestas.txt* (línea 12), si el fichero existe se sobrescribe y si no se crea nuevo.
- Se envía al fichero cada uno de los caracteres del array (líneas 14-15).
- Se cierra el flujo cuando no haya más datos que escribir (línea 18).

Se ha tratado la excepción *IOException* (líneas 19 a 21) pues la apertura del flujo, el método *write* y el cierre de flujo pueden provocarla en caso de encontrarse algún problema de escritura.



```
4 import java.io.FileWriter;
5 import java.io.IOException;
6
7 public class EjemploClaseFileWriter01 {
8
9     public static void main(String[] args) {
10         FileWriter fw;
11         try {
12             fw = new FileWriter("Ficheros\\fiestas.txt");
13             char [] frase = "10 de agosto de 2020 - San Lorenzo.".toCharArray();
14             for (int i=0; i< frase.length; i++)
15                 fw.write(frase[i]);
16             //Se puede escribir el array entero en lugar de carácter a carácter
17             //sustituyendo las dos líneas anteriores por fw.write(frase);
18             fw.close();
19         } catch (IOException e) {
20             e.printStackTrace();
21         }
22     }
23 }
```

Ilustración 12. *EjemploClaseFileWriter01*

Como se ha indicado anteriormente, si el fichero asociado al flujo existe se sobrescribe. Si queremos mantenerlo y añadir caracteres por el final solo tendremos que usar otro constructor a la hora de crear dicho flujo como aparece en la línea 12 de la siguiente ilustración 13, en el que el segundo argumento (*true*) así se lo especifica:

```
fw = new FileWriter("Ficheros\\fiestas.txt", true);
```

```

4 import java.io.FileWriter;
5 import java.io.IOException;
6
7 public class EjemploClaseFileWriter01a {
8
9     public static void main(String[] args) {
10         FileWriter fw;
11         try {
12             fw = new FileWriter("Ficheros\\fiestas.txt", true);
13             char [] frase = " ¡Viva San Lorenzo!".toCharArray();
14             fw.write(frase);
15             fw.close();
16         } catch (IOException e) {
17             e.printStackTrace();
18         }
19     }
20 }

```

Ilustración 13. EjemploClaseFileWriter01a

De esta manera el contenido del fichero fiestas.txt quedaría:

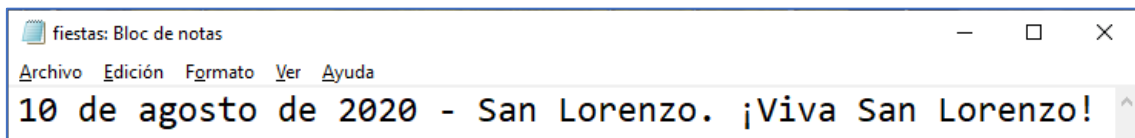


Ilustración 14. Fichero fiestas.txt después de añadir la última cadena

La clase *FileReader* solo nos permite leer caracteres. Si queremos realizar una **lectura eficiente** y leer **líneas enteras**, podemos **combinar flujos** como indica la siguiente ilustración:

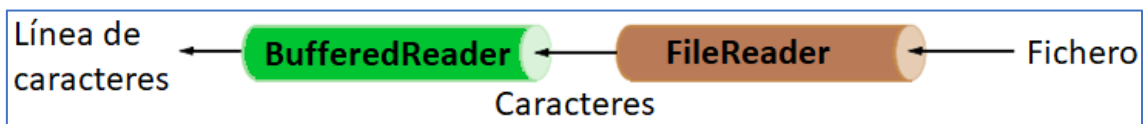


Ilustración 15. Combinando flujos

Los caracteres leídos por la clase *FileReader* sirven de entrada para la clase *BufferedReader* que es capaz de leerlos por líneas completas. La forma de declarar en Java estos flujos sería:

```
BufferedReader br = new BufferedReader(new FileReader("Ficheros\\fiestas.txt"));
```



Ilustración 16. Combinando flujos II

A continuación (ilustración 17) un ejemplo de esta clase *BufferedReader*. Utilizamos el fichero *autor.txt* (ilustración 9) pues está formado por varias líneas de texto. Después de ejecutarlo, se visualiza por ventana de consola el contenido de dicho fichero (línea 14). El método *readLine()* (línea 13) va leyendo línea a línea del fichero hasta encontrarse el final del mismo, momento en que devuelve el valor *null* para indicarlo.

```

2 import java.io.BufferedReader;
3 import java.io.FileReader;
4 import java.io.IOException;
5
6 public class EjemploClaseBufferedReader01 {
7
8     public static void main(String[] args) {
9         try {
10             BufferedReader br = new BufferedReader(
11                 new FileReader("Ficheros\\autor.txt"));
12             String linea;
13             while((linea = br.readLine())!=null)
14                 System.out.println(linea);
15             br.close();
16         } catch (IOException e) {
17             e.printStackTrace();
18         }
19     }
20 }

```

Ilustración 17. EjemploClaseBufferedReader01

En las líneas 16 a 18 se ha manejado la excepción *IOException* que, como ocurría con los flujos de caracteres, pueden provocar los métodos de apertura/cierre de flujo así como el de lectura.

Del mismo modo que utilizamos un buffer para leer líneas enteras, podemos utilizar otro para escribirlas haciendo uso de la clase *BufferedWriter*. El mecanismo de funcionamiento es análogo al anterior y su ejemplo aparece en la ilustración 20.

```
BufferedWriter bw = new BufferedWriter(new FileWriter("Ficheros\\fiestas2.txt"));
```

Ilustración 18. Combinando flujos III

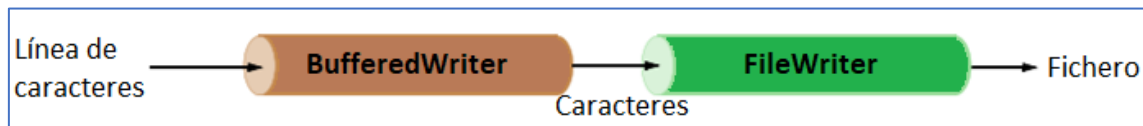


Ilustración 19. Combinando flujos IV

```

3 import java.io.BufferedWriter;
4 import java.io.FileWriter;
5 import java.io.IOException;
6
7 public class EjemploClaseBufferedWriter01 {
8
9     public static void main(String[] args) {
10         try {
11             BufferedWriter bw = new BufferedWriter(
12                 new FileWriter("Ficheros\\fiestas2.txt"));
13             bw.write("10 de agosto de 2020.");
14             bw.newLine();
15             bw.write(";Viva San Lorenzo!");
16             bw.close();
17         } catch (IOException e) {
18             e.printStackTrace();
19         }
20     }
21 }
22 }

```

Ilustración 20. EjemploClaseBufferedWriter01

Se abre el flujo de salida en las líneas 11 y 12: Si el fichero existe se sobrescribe y si no se crea nuevo. En este caso el método *write()* de la clase *BufferedWriter* (líneas 13 y 15) escribe cadenas. El método *newLine()* añade un separador de líneas ('\n'). El resultado es la creación del fichero *fiestas2.txt*.

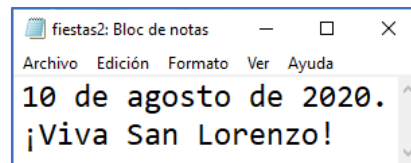


Ilustración 21. Contenido del fichero *fiestas2.txt*

En las líneas 17 a 19 se ha manejado la excepción *IOException* que pueden provocar los métodos de apertura/cierre de flujo así como el de escritura.

Podemos obtener el mismo resultado (ilustración 22) utilizando también la clase *PrintWriter* (descendiente de la clase *Writer*) y utilizando sus métodos similares a la salida estándar (*System.out*):

```
4+ import java.io.FileWriter;
7
8 public class EjemploClasePrintWriter01 {
9
10- public static void main(String[] args) {
11     try {
12         PrintWriter pw = new PrintWriter(
13             new FileWriter("Ficheros\\fiestas2.txt"));
14         pw.println("10 de agosto de 2020.");
15         pw.print("¡Viva San Lorenzo!");
16         pw.close();
17     } catch (IOException e) {
18         e.printStackTrace();
19     }
20 }
21 }
```

Ilustración 22. *EjemploClasePrintWriter01*

Puedes encontrar más información de estas últimas clases en:

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

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

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



Ya puedes realizar las actividades UD2.4, UD2.5 y UD2.6.

## 6.2. Ficheros binarios

Almacenan secuencias de dígitos binarios, no legibles directamente por el usuario sino por los programas (vídeo, dibujo...) que los han creado. Ocupan menos espacio en disco que los de texto.

Las clases que gestionan estos flujos de caracteres son *FileInputStream* para leer bytes y *FileOutputStream* para escribirlos (la jerarquía de clases aparecía en la ilustración 4 anterior). En la ayuda de Eclipse y en los siguientes enlaces puedes encontrar más información de las mismas, así como de sus métodos y las posibles excepciones que generan, pues en este documento solo se explicarán los que se utilicen en los ejemplos:

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

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

La forma de trabajar es muy similar al caso de los flujos de caracteres. En el siguiente ejemplo hemos juntado tanto el flujo de lectura como el de escritura de bytes para copiar la imagen de Homer Simpson de la derecha (fichero Homer.png).



```
2 import java.io.*;
3 public class EjemploFlujosBytes01 {
4     public static void main(String args[]) {
5
6         try {
7             FileInputStream origen = new FileInputStream("Ficheros\\Homer.png");
8             FileOutputStream destino = new FileOutputStream("Ficheros\\HomerCopia.png", false);
9             int i = origen.read();
10            while (i != -1) {
11                destino.write(i);
12                i = origen.read();
13            }
14            origen.close(); destino.close();
15        } catch (IOException e) {
16            System.out.println("Error de ficheros");
17        }
18    }
19 }
```

Ilustración 23. EjemploFlujosBytes01

En la línea 7 de la ilustración anterior se abre un flujo de lectura de bytes sobre el fichero *Homer.png* que servirá de entrada o fuente. En la línea 8 se abre uno de escritura sobre el fichero destino *HomerCopia.png*; si este último fichero no existe se crea, en caso contrario se sobrescribe (el segundo argumento del constructor, *false*, nos indica que no se puede añadir por el final).

Se va leyendo byte a byte comenzando en la línea 9 con el método *read*, y si no se detecta el final del fichero con el byte leído (-1) que devuelve el método, se escribe dicho byte con el método *write* en el flujo de salida (línea 11). Así para todos los bytes que componen el fichero fuente para conseguir una copia exacta de dicho fichero. Antes de finalizar se cierran los dos flujos (línea 14). La creación de flujos puede provocar el error *FileNotFoundException* y los métodos de lectura/escritura de bytes así como los cierres de los flujos el error *IOException*, por eso aparecen las instrucciones entre un manejador de errores try-catch.

Las líneas 9 a 13 pueden compactarse y sustituirse por estas otras leyendo de otra forma la secuencia de bytes:



```
int i;
while ((i=origen.read()) != -1)
    destino.write(i);
```

De forma análoga a lo que hacíamos con los flujos de caracteres, podemos combinar flujos para aumentar la funcionalidad y del mismo modo que leíamos/escribíamos líneas completas de caracteres podemos leer/escribir datos de tipos primitivos (enteros, reales, cadenas...). Para ello se utilizan las clases *DataInputStream* (lectura de datos de tipos primitivos) y *DataOutputStream* (escritura de datos de tipos primitivos) que se encargan de realizar estas conversiones de datos primitivos a bytes. La declaración de un objeto de esta última clase sería:

```
DataOutputStream dos=new FileOutputStream(new File("Ficheros\\antigüedad.dat"));
```

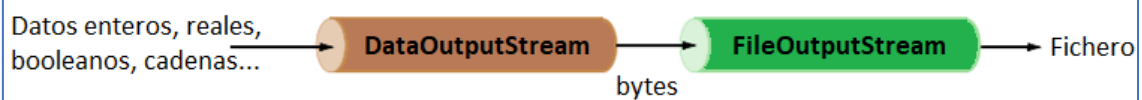


Ilustración 24. Combinando flujos V

Los datos de tipo primitivo (enteros, reales, booleanos...) son convertidos por la clase *DataOutputStream* a bytes y éstos enviados al fichero por la clase *FileOutputStream*. El ejemplo de la siguiente ilustración envía al fichero *antigüedad.dat* los siguientes pares de datos de tipo cadena y real:

Alberto Carrera 29.5 Ana Ereza 18 ... Luis Aldea 9

```
3 public class EjemploDataOutputStream01 {
4     public static void main (String[] args) throws IOException{
5         DataOutputStream dos=new DataOutputStream(new FileOutputStream("Ficheros\\antigüedad.dat"));
6         // Equivale a
7         //FileOutputStream fos = new FileOutputStream("Ficheros\\antigüedad.dat");
8         //DataOutputStream dos=new DataOutputStream(fos);
9
10        String nombres={"Alberto Carrera", "Ana Ereza",
11                        "Antonino Lasierra", "Pura Plo", "Belén Carrera", "Luis Aldea"};
12        double antigüedad[]={29.5, 18, 38.5, 28,22, 9};
13        for (int i=0; i<antigüedad.length;i++){
14            dos.writeUTF(nombres[i]);
15            dos.writeDouble(antigüedad[i]);
16        }
17        dos.close();
18    }
19 }
```

Ilustración 25. EjemploDataOutputStream01

El método *writeUTF* (línea 14) de la clase *DataOutputStream* convierte las cadenas de los nombres a bytes para ser enviadas (por la clase *FileOutputStream*) al fichero. El método *writeDouble* (línea 15) hace lo mismo pero con datos reales de los años de antigüedad. El resto de métodos de escritura los puedes encontrar en el siguiente enlace

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

En esta ocasión no se han manejado las excepciones que pueden provocar los métodos utilizados (línea 4).

Para leer los datos del fichero anterior, utilizaremos la clase *DataInputStream* y sus métodos de lectura. La declaración de un objeto de esta clase sería:

```
DataInputStream dis=new DataInputStream(new FileInputStream("Ficheros\\antigüedad.dat"));
```

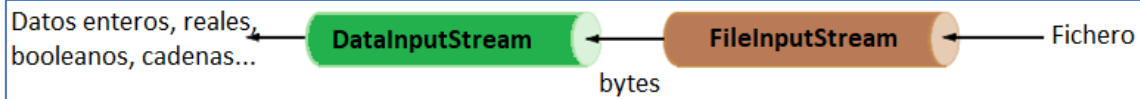


Ilustración 26. Combinando flujos VI

Los bytes leídos por la clase *FileInputStream* son convertidos por la clase *DataInputStream* al tipo de dato primitivo. En el siguiente ejemplo se leen los datos creados con la clase anterior. **Es muy importante indicar que si han sido grabados en el fichero de la forma cadena – real – cadena – real... hay que leerlos en ese mismo orden.** El método *readUTF()* (línea 11) es el encargado de leer las cadenas mientras que el método *readDouble()* (línea 12) lee los datos reales. Si se ha llegado al final del fichero y se intenta leer con alguno de los métodos anteriores (o con cualquiera) se produce la excepción *EOFException* que es tratada en el manejador de la línea 13 (realmente atrapa el error y no hace nada como puedes comprobar). Cualquier otra posible excepción no ha sido tratada (línea 4).

```

2 import java.io.*;
3 public class EjemploDataInputStream01 {
4     public static void main (String[] args) throws IOException{
5         DataInputStream dis=new DataInputStream(new FileInputStream("Ficheros\\antigüedad.dat"));
6         // Equivale a
7         // FileInputStream fis = new FileInputStream("Ficheros\\antigüedad.dat");
8         // DataInputStream dis=new DataInputStream(fis);
9         try{
10             while(true)
11                 System.out.println("Nombre: "+ dis.readUTF()
12                                     + ", antigüedad: " + dis.readDouble()+ " años");
13         }catch (EOFException eo) {}
14         dis.close();
15     }
16 }

```

Problems @ Javadoc Declaration Console

<terminated> EjemploDataInputStream01 [Java Application] C:\Program Files\Java\jre1.8.0\_261\bin\javaw.exe (17-ago-2020 19:00:23 – 1

Nombre: Alberto Carrera, antigüedad: 29.5 años  
 Nombre: Ana Ereza, antigüedad: 18.0 años  
 Nombre: Antonino Lasierra, antigüedad: 38.5 años  
 Nombre: Pura Plo, antigüedad: 28.0 años  
 Nombre: Belén Carrera, antigüedad: 22.0 años  
 Nombre: Luis Aldea, antigüedad: 9.0 años

Ilustración 27. EjemploDataInputStream01

Los métodos de lectura los podrás encontrar en el siguiente enlace

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



Ya puedes realizar la actividad UD2.7.

Hemos trabajado con datos binarios leyendo bytes directamente en primer lugar y datos de tipo primitivo usando una combinación de flujos. El siguiente escalón sería poder trabajar con objetos, es decir poder leer/escribir objetos en disco (o en otras fuentes/destinos). Para ello, para que un programa pueda convertir a bytes los objetos y enviarlos a disco, por la red... y también el proceso contrario, reconstruir los objetos a partir de los bytes recuperados, el objeto debe implementar la interfaz *Serializable*. Todos los objetos predeterminados de Java (*String*, *Date*...) ya la implementan y no hay que especificarlo.

Para leer y escribir objetos se utilizan las clases *ObjectInputStream* (lectura) y *ObjectOutputStream* (escritura).

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

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

Trabajaremos con objetos de la clase *Profesor* que tiene la siguiente estructura:

```
public class Profesor implements Serializable {
    private String nombre;
    private double antigüedad;

    ...
}
```

Para enviar objetos al fichero, declaramos un objeto de la clase *ObjectOutputStream*:

```
ObjectOutputStream oos=new ObjectOutputStream(new FileOutputStream("Ficheros\\antigüedad_obj.dat"));
```

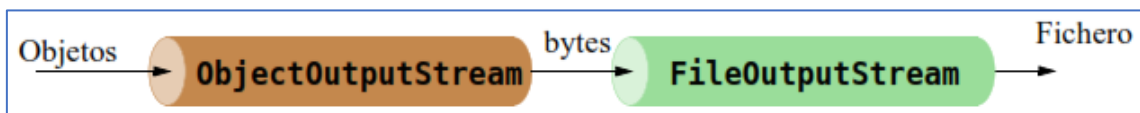


Ilustración 28. Combinando flujos VII

Los objetos son convertidos por la clase *ObjectOutputStream* a bytes y éstos enviados al fichero por la clase *FileOutputStream* como hemos visto en ejemplos anteriores. El ejemplo de la siguiente ilustración envía al fichero *antigüedad.dat\_obj.dat* 5 objetos de tipo persona.

```
2 import java.io.*;
3 public class EjemploObjectOutputStream01 {
4     public static void main (String[] args) throws IOException{
5         ObjectOutputStream oos=new ObjectOutputStream(new FileOutputStream("Ficheros\\antigüedad_obj.dat"));
6         // Equivale a
7         //FileOutputStream fos = new FileOutputStream("Ficheros\\antigüedad_obj.dat");
8         //ObjectOutputStream oos=new ObjectOutputStream(fos);
9         String nombres[]{"Alberto Carrera", "Ana Ereza",
10            "Antonino Lasierria", "Pura Plo", "Belén Carrera"};
11         double antigüedad[]={29.5, 18, 38.5, 28,22};
12         for (int i=0; i<nombres.length;i++)
13             oos.writeObject(new Profesor (nombres[i], antigüedad[i]));
14         oos.close();
15     }
16 }
17 }
```

Ilustración 29. EjemploObjectOutputStream01

En la línea 13 se crea cada uno de los 5 objetos personas, rellenando sus dos atributos a partir de arrays de cadenas y reales y después se escriben en el fichero utilizando el método *writeObject*. No se han manejado excepciones (línea 4).

Para leer objetos del fichero declaramos un objeto de la clase *ObjectInputStream*:

```
ObjectInputStream ois=new ObjectInputStream(new FileInputStream("Ficheros\\antiguedad_obj.dat"));
```

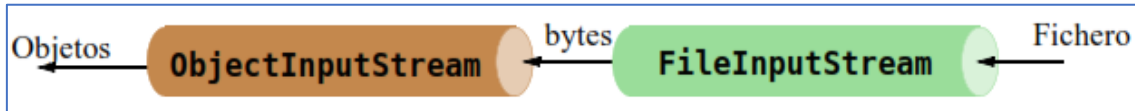


Ilustración 30. Combinando flujos VIII

Los bytes leídos por la clase *FileInputStream* son reconstruidos y pasados a objetos en la clase *ObjectOutputStream*.

```

2 import java.io.*;
3 public class EjemploObjectInputStream01 {
4     public static void main (String[] args) throws IOException, ClassNotFoundException{
5         ObjectInputStream ois=new ObjectInputStream(new FileInputStream("Ficheros\\antiguedad_obj.dat"));
6         // Equivale a
7         // FileInputStream fis = new FileInputStream("Ficheros\\antiguedad_obj.dat");
8         // ObjectInputStream ois=new ObjectInputStream(fis);
9         try {
10             while (true)
11                 System.out.println(ois.readObject());
12         } catch (EOFException e) {
13             System.out.println("- Final del fichero -");
14         }
15         ois.close();
16     }
17 }
  
```

Problems @ Javadoc Declaration Console

```

<terminated> EjemploObjectInputStream01 [Java Application] C:\Program Files\Java\jre1.8.0_261\bin\javaw.exe (17-ago-2020 22:32:50 - 22:32:52)
Profesor [nombre=Alberto Carrera, antiguedad=29.5]
Profesor [nombre=Ana Ereza, antiguedad=18.0]
Profesor [nombre=Antonino Lasierra, antiguedad=38.5]
Profesor [nombre=Pura Plo, antiguedad=28.0]
Profesor [nombre=Belén Carrera, antiguedad=22.0]
- Final del fichero -
  
```

Ilustración 31. EjemploObjectInputStream01

El método *readObject()* (línea 11 ilustración anterior) va leyendo objetos del fichero que a su vez son impresos por la consola. Cuando se lee la marca final de fichero se produce la excepción *EOFException* tratada en las líneas 12 a 14. El resto de excepciones no son tratadas (línea 4).

Antes de cerrar este apartado hay que indicar un problema que existe a la hora de añadir objetos a un fichero existente ya creado. Cuando se crea un objeto de la clase *ObjectOutputStream*, antes de comenzar a guardar objetos, se añade al comienzo del fichero una información de cabecera de dicho fichero. Los objetos que creamos se escribirán a continuación de dicha cabecera. Si volvemos a abrir el fichero (y creamos por tanto un objeto de la clase *ObjectOutputStream*) para añadir objetos, se volverá a escribir otra vez la información de cabecera después del último objeto que se encontrara en el archivo y antes de añadir los nuevos objetos. Cuando posteriormente leamos los objetos profesor del fichero con un objeto de la clase *ObjectInputStream*, a partir de la segunda cabecera se lanzará una excepción *StreamCorruptedException* y terminará la ejecución del programa sin leer los restantes. Una solución posible sería un nuevo fichero que antes de incluir los nuevos objetos profesor se

rellenara con los antiguos existentes... y al terminar renombrar este segundo archivo con el nombre del archivo original. Otra solución sería crear una nueva clase que herede de la *ObjectOutputStream* y a continuación redefinirla, en especial el método *writeStreamHeader()* que es el encargado de escribir las cabeceras. Así se hace en la línea 13 de la siguiente ilustración, pasando dicho método a no hacer nada.

```
3 import java.io.IOException;
4 import java.io.ObjectOutputStream;
5 import java.io.OutputStream;
6
7 public class MiObjectOutputStream extends ObjectOutputStream {
8
9     public MiObjectOutputStream(OutputStream arg0) throws IOException {
10         super(arg0);
11     }
12     // Sobreescritura del método
13     protected void writeStreamHeader() throws IOException { }
14
15 }
```

Ilustración 32. Ejemplo *MiObjectOutputStream*

Teniendo en cuenta lo anterior, la clase para añadir objetos quedaría:

```
2 import java.io.*;
3 public class EjemploObjectOutputStream02 {
4     public static void main (String[] args) throws IOException{
5
6         File f = new File("Ficheros\\antiguedad_obj.dat");
7         if (!f.exists()) {
8             System.out.println("El fichero de profesores no ha sido todavía creado");
9             System.exit(-1);
10        }
11        FileOutputStream fos = new FileOutputStream(f, true);
12        MiObjectOutputStream moos = new MiObjectOutputStream(fos);
13        moos.writeObject(new Profesor ("Luis Aldea", 9));
14        moos.close();
15    }
16 }
```

Ilustración 33. *EjemploObjectOutputStream02*

Si el fichero existe, se crea un flujo de salida hacia él para añadir al profesor Luis Aldea por el final del mismo.



Ya puedes realizar la actividad UD2.8.

### 6.3. Ficheros de acceso aleatorio

La introducción a los mismos ya se ha visto en los apartados 4 y 5.2 anteriores (revisa si es necesario este último para poder entender lo que en éste se explica) .

En los ejemplos vistos hasta ahora hemos trabajado con los ficheros de manera secuencial. Se abría el flujo asociado al fichero e íbamos leyendo desde el principio cada dato (byte, dato primitivo carácter, línea, objeto...), uno a continuación de otro hasta alcanzar el final del mismo. Ídem si en lugar de leer lo abríamos para escribir. Con la organización de acceso aleatorio, podremos seguir realizando las operaciones anteriores y además acceder a cualquier dato directamente sin tener que recorrer los anteriores que se encuentran en el fichero.

Para ello utilizaremos la clase *RandomAccessFile*:

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

que tiene todas las propiedades de las clases *DataInputStream* y *DataOutputStream*; permite abrir un archivo como lectura o como lectura y escritura simultáneamente de cualquier tipo de datos primitivo. Si se utiliza para lectura (modo "r") dispone de métodos como: *readInt()*, *readLong()*, *readDouble()*, *readLine()*... Si se utiliza como lectura/escritura (modo "rw") dispone de métodos como *writeInt()*, *writeLong()*, *writeDouble()*, *writeBytes()*...

Constructores:

- *RandomAccessFile(File fichero, String modo)*
- *RandomAccessFile(String fichero, String modo)*

donde modo = "r" (lectura) o "rw" (lectura-escritura)

Métodos:

- *void seek(long posicion)*: Establece la posición actual del puntero en el byte indicado.
- *long getFilePointer()*: Devuelve la posición actual (en bytes).
- *int skipBytes(int desplazamiento)*: Mueve el puntero el nº de bytes del parámetro.
- *long length()*: Longitud del fichero en bytes.

En el siguiente ejemplo vamos a introducir los datos de los 5 primeros profesores que aparecen en la ilustración 7 del apartado 5.2 de estos materiales. Es muy importante para el resto de ejemplos de este apartado que tengas en cuenta las dimensiones de los campos con los que trabajamos:

Id (entero – 4 bytes)
Nombre (20 caracteres – 40 bytes)
Departamento (entero – 4 bytes)
Antigüedad (real – 8bytes)

```

3 import java.io.*;
4 public class EjemploClaseRandomAccessFile01 {
5     public static void main(String[] args) throws IOException {
6         File fichero = new File ("Ficheros\\ProfesFPSierraGuara.dat");
7         RandomAccessFile file = new RandomAccessFile(fichero, "rw");
8         String profesores[] = {"Alberto Carrera", "Ana Ezeza", "Antonino Lasierra", "Pura Plo", "Belén Carrera"};
9         int departamento[] = {10, 20, 20, 30, 40}; //Informática --> 10, Administrativo -->20...
10        Double antigüedad[]={29.5, 18.0, 38.5, 28.0, 22.0};//
11
12        StringBuffer sb = null;
13        int total = profesores.length;
14
15        for (int i=0;i<total; i++){
16            file.writeInt(i+1); //
17            sb = new StringBuffer(profesores[i]);
18            sb.setLength(20);
19            file.writeChars(sb.toString());
20            file.writeInt(departamento[i]);
21            file.writeDouble(antigüedad[i]);
22        }
23        file.close();
24    }
25 }

```

Ilustración 34. EjemploClaseRandomAccessFile01

En la línea 7 creamos el objeto que representará el fichero de acceso aleatorio, lo hemos construido a partir de un objeto *File* (línea 6) pero también se podía haber usado el segundo formato del constructor y haber empleado un *String*. Lo abrimos para lectura/escritura ("rw"). Realmente solo vamos a realizar operaciones de escritura, pero el segundo argumento del constructor no permite que sea "w". El primer dato que introducimos en el fichero es un entero (línea 16) que representa el identificador del profesor (1,2...5). El segundo dato corresponde a los 20 caracteres del nombre. Se preparan exactamente esos 20 caracteres en las líneas 17-18 y se escribe en el fichero en la línea 19. En tercer lugar se guarda el entero correspondiente al número de departamento (línea 20) y en último lugar la antigüedad en años de experiencia de cada profesor (línea 21). Antes de finalizar se cierra el objeto asociado al fichero aleatorio (línea 23). No se ha tratado la excepción *IOException* como se indica en el final de la línea 5 que es la que pueden provocar los métodos de escritura (*write()*) y cierre de flujo que aparecen en la clase anterior.

La clase que viene a continuación inserta al profesor Luis Aldea, de identificador 20, en su posición correspondiente:

```

3 import java.io.*;
4 public class EjemploClaseRandomAccessFile01a {
5     public static void main(String[] args) throws IOException {
6         File fichero = new File ("Ficheros\\ProfesFPSierraGuara.dat");
7         RandomAccessFile file = new RandomAccessFile(fichero, "rw");
8         file.seek((20-1)*56);
9         file.writeInt(20); //
10        StringBuffer sb = new StringBuffer("Luis Aldea");
11        sb.setLength(20);
12        file.writeChars(sb.toString());
13        file.writeInt(10);
14        file.writeDouble(9);
15        file.close();
16    }
17 }

```

Ilustración 35. EjemploClaseRandomAccessFile01a



El puntero del fichero se desplaza (línea 8) a la posición que le corresponde en el fichero. Situado en ella escribe su identificador (línea 9), su nombre (línea 12, su departamento (línea 13) y sus años de antigüedad (línea 14).

Seguidamente vamos a realizar un acceso secuencial leyendo todos los datos del fichero.

```

3 import java.io.*;
4 public class EjemploClaseRandomAccessFile02 {
5     public static void main(String[] args) throws IOException {
6         File fichero = new File ("Ficheros\\ProfesFPSierraGuara.dat");
7         RandomAccessFile file = new RandomAccessFile(fichero, "r");
8         int posicion = 0; //
9         int id, departamento;
10        Double antigüedad;
11        char profesores[] = new char[20];
12        System.out.println ("Id      Nombre      Departamento  Antigüedad");
13        System.out.println ("-----");
14        while (file.getFilePointer() != file.length()) {
15            file.seek(posicion); //nos posicionamos
16            id = file.readInt(); // obtengo id de empleado
17            for (int i = 0; i < profesores.length; i++)
18                profesores[i] = file.readChar();
19            String profesor = new String(profesores);
20            departamento = file.readInt();
21            antigüedad = file.readDouble();
22            System.out.println(id + " " + profesor + departamento + " " + antigüedad);
23            posicion= posicion + 56;
24        }
25        file.close();
26    }
27 }

```

Ilustración 36. EjemploClaseRandomAccessFile02

El fichero se ha abierto solo para lectura (línea 7). Comenzamos desde el principio (línea 8) e iremos leyendo y avanzando siempre y cuando no hayamos llegado al final del fichero (línea 14). A partir de la posición donde estemos situados (la 0 al principio) leemos el identificador del profesor (línea 16), después los 20 caracteres de su nombre (líneas 17 a 19), después el departamento al que pertenece (línea 20) para terminar leyendo su antigüedad (línea 21). Nos desplazamos al siguiente sumando 56 a la posición actual pues es el tamaño en bytes que ocupa los datos de cada profesor (línea 23).

El resultado por la consola sería:

Id	Nombre	Departamento	Antigüedad
1	Alberto Carrera	10	29.5
2	Ana Ereza	20	18.0
3	Antonino Lasierra	20	38.5
4	Pura Plo	30	28.0
5	Belén Carrera	40	22.0
0		0	0.0
0		0	0.0
...		...	...
20	Luis Aldea	10	9.0

Una solución muy parecida al anterior es



```

3 import java.io.*;
4 public class EjemploClaseRandomAccessFile02a {
5     public static void main(String[] args) throws IOException {
6         File fichero = new File ("Ficheros\\ProfesFPSierraGuara.dat");
7         RandomAccessFile file = new RandomAccessFile(fichero, "r");
8         int id, departamento;
9         Double antigüedad;
10        char profesores[] = new char[20];
11        System.out.println ("Id      Nombre      Departamento  Antigüedad");
12        System.out.println ("-----");
13        file.seek(0); //nos posicionamos en el primero
14        while (file.getFilePointer() != file.length()) {
15            id = file.readInt(); // obtengo id de empleado
16            for (int i = 0; i < profesores.length; i++)
17                profesores[i] = file.readChar();
18            String profesor = new String(profesores);
19            departamento = file.readInt();
20            antigüedad = file.readDouble();
21            System.out.println(id + " " + profesor + departamento + " " + antigüedad);
22        }
23        file.close();
24    }
25 }

```

Ilustración 37. EjemploClaseRandomAccessFile02a

¿Qué ha cambiado (suprimido) con respecto al anterior? Pista: Cuando leemos o escribimos, desplazamos el puntero de la posición ("hacia la derecha") tantos bytes como los que se utilizan para representar el dato que se lee o escribe.

En la siguiente clase intentamos acceder directamente a los datos de la profesora cuyo identificador es 5:

```

3 import java.io.*;
4 public class EjemploClaseRandomAccessFile03 {
5     public static void main(String[] args) throws IOException {
6         File fichero = new File ("Ficheros\\ProfesFPSierraGuara.dat");
7         RandomAccessFile file = new RandomAccessFile(fichero, "r");
8         int id=5;
9         int posicion= (id-1)*56;
10        if (posicion > file.length()-56) {
11            System.out.println ("No existe un empleado con ese identificador");
12            System.exit(-1);
13        }
14        int departamento;
15        Double antigüedad;
16        char profesores[] = new char[20];
17        System.out.println ("Id      Nombre      Departamento  Antigüedad");
18        System.out.println ("-----");
19        file.seek(posicion);
20        int id_pos = file.readInt();
21        for (int i = 0; i < profesores.length; i++)
22            profesores[i] = file.readChar();
23        String profesor = new String(profesores);
24        departamento = file.readInt();
25        antigüedad = file.readDouble();
26        System.out.println(id_pos + " " + profesor + departamento + " " + antigüedad);
27        file.close();
28    }
29 }

```

Ilustración 38. EjemploClaseRandomAccessFile03

Abrimos el fichero para solo lectura (línea 7) y calculamos la posición que le correspondería en las líneas 8 y 9. Si la posición queda fuera de las dimensiones del fichero el programa finaliza; en caso contrario se sitúa el puntero en la posición correspondiente (línea 19) y se pasan a leer sus datos.

En el siguiente ejemplo incrementamos en 1 año la antigüedad de la profesora de identificador 5. Una vez localizada como en el ejemplo anterior (línea 9), debemos situarnos al comienzo del campo antigüedad, que son los últimos 8 bytes del total de 56 bytes destinado a guardar la información de cada profesor (línea 19). Leemos el dato (línea 20) con lo que el puntero del fichero se desplaza 8 bytes ("a la derecha"), lo incrementamos en una unidad (línea 21), y debemos dejarlo en el mismo sitio donde estaba, por eso retrocedemos ("a la izquierda") los 8 bytes anteriores (línea 22) para escribirlo en su sitio correspondiente (línea 23) y de esta manera el puntero del fichero vuelve a desplazarse 8 bytes ("a la derecha") colocándose al final de los datos de la profesora de identificador 5; si queremos comprobar que todo está correcto tras la modificación retrocedemos ("a la izquierda") 56 bytes al comienzo del primer dato de la profesora (línea 24) y leemos e imprimimos sus datos.

```

3 import java.io.*;
4 public class EjemploClaseRandomAccessFile04 {
5     public static void main(String[] args) throws IOException {
6         File fichero = new File ("Ficheros\\ProfesFPSierraGuara.dat");
7         RandomAccessFile file = new RandomAccessFile(fichero, "rw");
8         int id=5;
9         int posicion= (id-1)*56;
10        if (posicion > file.length()-56) {
11            System.out.println ("No existe un empleado con ese identificador");
12            System.exit(-1);
13        }
14        int departamento;
15        Double antigüedad;
16        char profesores[] = new char[20];
17        System.out.println ("Id      Nombre      Departamento  Antigüedad");
18        System.out.println ("-----");
19        file.seek(posicion+48);
20        antigüedad=file.readDouble();
21        antigüedad++;
22        file.seek(file.getFilePointer()-8);
23        file.writeDouble(antigüedad);
24        file.seek(file.getFilePointer()-56);
25        id = file.readInt();
26        for (int i = 0; i < profesores.length; i++)
27            profesores[i] = file.readChar();
28        String profesor = new String(profesores);
29        departamento = file.readInt();
30        antigüedad = file.readDouble();
31        System.out.println(id + " " + profesor + departamento + " " + antigüedad);
32
33        file.close();
34    }
35 }

```

Ilustración 39. EjemploClaseRandomAccessFile04



Ya puedes realizar la actividad UD2.9.

## 7. Trabajo con ficheros XML

Los ficheros XML suelen utilizarse para intercambiar datos entre plataformas distintas e incompatibles como p.ej. importar datos a una base de datos procedentes de otra base de datos que utiliza otro gestor distinto y que los ha exportado en ese formato para facilitar la migración, en la configuración de programas, de servidores... entre otros usos.

Para poder trabajar con ficheros XML se utiliza un procesador o parser XML. Algunos de ellos ya los hemos visto en el curso anterior como *DOM* (recomendación oficial de la *World Wide Web Consortium - W3C*) o *SAX* (api originalmente pensada solo para Java); ambos son independientes del lenguaje de programación, existiendo versiones para muchos de ellos.

Utilizaremos *XStream* por tratarse de una solución sencilla para trabajar (escribir y leer) documentos XML desarrollada por un equipo particular de programadores. El sitio oficial se encuentra en:

<https://x-stream.github.io/>

y una forma rápida de conocerlo:

<https://x-stream.github.io/tutorial.html>

Las librerías que necesitaremos incorporar a nuestro proyecto las encontrarás en la subcarpeta *LibreriasXStream* dentro de la carpeta de *software*.

Para nuestros ejemplos utilizaremos la clase *Profesor* (nombre, departamento, antigüedad, lista de asignaturas que imparte):

```
public class Profesor {  
    private String nombre;  
    private int departamento;  
    private double antigüedad;  
    private String [] asignaturas;  
    ...  
    ...  
    ...  
}
```

Y la siguiente clase *ListaProfesores* para formar una colección de profesores:

```
3 import java.util.ArrayList;  
4 import java.util.List;  
5  
6 public class ListaProfesores {  
7     private List<Profesor> lista = new ArrayList<Profesor>();  
8     public ListaProfesores(){ }  
9     public void add(Profesor p){  
10         lista.add(p);  
11     }  
12     public List<Profesor> getListaProfesores() {  
13         return lista;  
14     }  
15  
16 }
```

Ilustración 40. Clase *ListaProfesores*

Para crear el fichero XML (ver siguiente ilustración) creamos un objeto de la clase *ListaProfesores* (línea 9) donde añadimos varios profesores junto con sus asignaturas (líneas 11 a 18).

Seguidamente instanciamos un objeto de la clase *XStream* (línea 21) preparado para trabajar con la codificación UTF-8 y no tener problemas con tildes y ñ's, y que utilizaremos junto con el método *toXML* para crear el fichero *ProfesFPSierraGuara.xml* a partir de la lista de profesores anterior (línea 25). El método *alias* (líneas 22 y 23) lo utilizamos para cambiar el nombre de las etiquetas de los nodos. Así la etiqueta del nodo raíz no se llamará *XML.ListaProfesores* sino *ProfesoresFPSierradeGuara* (línea 22), para cada profesor su etiqueta será *Profesor* en lugar de *XML.Profesor* (línea 23). El método *addImplicitCollection* (línea 24) lo utilizamos para suprimir la etiqueta *lista* (recordad que era el nombre del atributo de la lista de profesores a partir de la cual se construye el documento XML) y de esta manera todos los nodos de profesor cuelgan directamente del nodo raíz y no del nodo *lista*.

```

3 import java.io.*;
4 import com.thoughtworks.xstream.XStream;
5 import com.thoughtworks.xstream.io.xml.DomDriver;
6
7 public class EscribirProfesoresXML {
8     public static void main (String[] args) throws IOException, ClassNotFoundException{
9         ListaProfesores profesores = new ListaProfesores();
10
11         profesores.add (new Profesor("Alberto Carrera", 10, 29.5,
12             new String[]{"Acceso a datos", "Programación"}));
13         profesores.add( new Profesor("Antonino Lasierra", 20, 38.5,
14             null));
15         profesores.add(new Profesor ("Belén Carrera", 40, 22,
16             new String[]{"Formación y Orientación Laboral", "Empresa e iniciativa emprendedora"}));
17         profesores.add(new Profesor ("Luis Aldea", 10, 9,
18             new String[]{"Inglés I", "Inglés II, Informatica ESO"}));
19
20         try{
21             XStream xstream = new XStream(new DomDriver("UTF-8"));
22             xstream.alias("ProfesoresFPSierradeGuara", ListaProfesores.class);
23             xstream.alias("Profesor", Profesor.class);
24             xstream.addImplicitCollection(ListaProfesores.class, "lista");
25             xstream.toXML(profesores, new FileOutputStream ("Ficheros\\ProfesFPSierraGuara.xml"));
26             System.out.println("Creado fichero XML...");
27         }catch (Exception e) {e.printStackTrace(); }
28     }
29 }

```

Ilustración 41. Clase *EscribirProfesoresXML*

El resultado de la ejecución aparece a la derecha.

¿Cómo cambiarías el nombre de las etiquetas `<string>` por el de `<asignatura>`?

```

<?xml version="1.0"?>
- <ProfesoresFPSierradeGuara>
-   <Profesor>
      <nombre>Alberto Carrera</nombre>
      <departamento>10</departamento>
      <antiguedad>29.5</antiguedad>
      - <asignaturas>
            <string>Acceso a datos</string>
            <string>Programación</string>
          </asignaturas>
    </Profesor>
-   <Profesor>
      <nombre>Antonino Lasierra</nombre>
      <departamento>20</departamento>
      <antiguedad>38.5</antiguedad>
    </Profesor>
-   <Profesor>
      <nombre>Belén Carrera</nombre>
      <departamento>40</departamento>
      <antiguedad>22.0</antiguedad>
      - <asignaturas>
            <string>Formación y Orientación Laboral</string>
            <string>Empresa e iniciativa emprendedora</string>
          </asignaturas>
    </Profesor>
-   <Profesor>
      <nombre>Luis Aldea</nombre>
      <departamento>10</departamento>
      <antiguedad>9.0</antiguedad>
      - <asignaturas>
            <string>Inglés I</string>
            <string>Inglés II, Informatica ESO</string>
          </asignaturas>
    </Profesor>
  </ProfesoresFPSierradeGuara>

```

La lectura (y visualización por consola de los datos en este ejemplo) se haría como aparece en la siguiente ilustración:

```
3 import java.io.FileInputStream;
4 import java.io.IOException;
5 import com.thoughtworks.xstream.XStream;
6
7 public class LeerProfesoresXML {
8     public static void main(String[] args) throws IOException{
9         XStream xstream = new XStream();
10        xstream.alias("ProfesoresFPSierradeGuara", ListaProfesores.class);
11        xstream.alias("Profesor", Profesor.class);
12        xstream.addImplicitCollection(ListaProfesores.class, "lista");
13
14        ListaProfesores listadoTodos = (ListaProfesores)
15            xstream.fromXML
16            (new FileInputStream("Ficheros\\ProfesFPSierraGuara.xml"));
17        System.out.println("Número de Profesores: " +
18            listadoTodos.getListaProfesores().size());
19
20        for (Profesor p : listadoTodos.getListaProfesores())
21            System.out.println(p);
22
23        System.out.println("Fin del listado...");
24    }
25 }
```

Console

<terminated> LeerProfesoresXML [Java Application] C:\Program Files\Java\jre1.8.0\_261\bin\javaw.exe (19-ago-2020 22:31:36 – 22:31:38)

Número de Profesores: 4

Profesor [nombre=Alberto Carrera, departamento=10, antigüedad=29.5, asignaturas=[Acceso a datos, Programación]]

Profesor [nombre=Antonino Lasierra, departamento=20, antigüedad=38.5, asignaturas=null]

Profesor [nombre=Belén Carrera, departamento=40, antigüedad=22.0, asignaturas=[Formación y Orientación Laboral, E

Profesor [nombre=Luis Aldea, departamento=10, antigüedad=9.0, asignaturas=[Inglés I, Inglés II, Informatica ESO]]

Fin del listado....

Ilustración 42. Clase LeerProfesoresXML

El método *fromXML* (línea 15) del objeto *XStream* creado en la línea 9 permite recuperar la información de los nodos del documento XML sobre la lista de profesores *listadoTodos* (línea 14) que pasa a recorrerse y visualizarse por la consola en las últimas líneas de código.



Ya puedes realizar la actividad UD2.10.

## 8. Trabajo con ficheros JSON

JSON es un formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de JavaScript, aunque, debido a su amplia adopción como alternativa a XML, se considera un formato independiente del lenguaje (Wikipedia).

Utilizaremos el siguiente ejemplo para explicar sus elementos:

```
{ } *ProfesFPSierraGuara.json
1 [
2   { "nombre": "Alberto Carrera", "departamento": 10, "antigüedad": 29.5, "asignaturas": [ "Acceso a datos", "Programación" ] },
3   { "nombre": "Antonino Lasierra", "departamento": 20, "antigüedad": 38.5 },
4   { "nombre": "Belén Carrera", "departamento": 40, "antigüedad": 22.0, "asignaturas": [ "FOL", "Empresa e iniciativa emprendedora" ] },
5   { "nombre": "Luis Aldea", "departamento": 10, "antigüedad": 9.0, "asignaturas": [ "Inglés I", "Inglés II", "Informática ESO" ] }
6 ]
```

Ilustración 43. Ejemplo de fichero JSON

Un array es una colección de valores. Un array comienza con [ (corchete izquierdo) y termina con ] (corchete derecho). Los valores se separan por , (coma). En el ejemplo anterior, tenemos varias colecciones o arrays. La principal comienza en la línea 1 y termina en la 6, agrupando 4 profesores separados por comas (la coma del final de las líneas 2 a 4). A su vez cada profesor contiene un array con las asignaturas que imparte, exceptuando el profesor de la línea 3 que no tiene.

Un objeto es un conjunto desordenado de pares nombre/valor (o también conocidos como clave/valor). Comienza con { (llave izquierda) y termine con } (llave derecha). Cada nombre o clave es seguido por : y los pares nombre/valor están separados por , (coma). En el ejemplo anterior aparecen 4 objetos profesor, líneas 2 a 5, uno en cada línea. Para cada objeto como nombres o claves tenemos *nombre*, *departamento*, *antigüedad* y *asignaturas*. Sus valores van a continuación de los : del nombre o clave. Pueden existir objetos sin un par nombre/valor como es el caso del profesor de la línea 3 que no tiene asignaturas.

Un valor puede ser una cadena de caracteres con comillas dobles, o un número, o true o false o null, o un objeto o un array. Estas estructuras pueden anidarse. En el ejemplo de la siguiente ilustración de un objeto profesor ({ }-líneas 3 y 8) aparecen 4 pares nombre/valor separados por : (dos puntos). Los nombres en color morado y los valores en azul (cadenas) o rojo (números).

```
3 {
4   "nombre": "Alberto Carrera",
5   "departamento": 10,
6   "antigüedad": 29.5,
7   "asignaturas": [ "Acceso a datos", "Programación" ]
8 }
```

Ilustración 44. Ejemplo de objeto JSON

Para nuestros ejemplos utilizaremos la clase *Profesor* (nombre, departamento, antigüedad, lista de asignaturas que imparte):

```
public class Profesor {
    private String nombre;
    private int departamento;
    private double antigüedad;
    private String [] asignaturas;
    ...
}
```

*Gson* es una librería java de *Google* que facilita la conversión de objetos java a cadenas JSON (serialización) y viceversa (deserialización), así como analizar cadenas JSON. La necesitaremos incorporar en nuestros proyectos cuando alguna de las clases trabajen con este tipo de archivos. Encontrarás la librería *gson-2.8.6.jar* en la subcarpeta *LibreriaGSON* dentro de la carpeta de software.

En la siguiente ilustración se ha creado el archivo JSON de la ilustración 43 anterior:

```

3 import java.io.FileWriter;
4 import java.util.Arrays;
5 import java.util.List;
6
7 import com.google.gson.Gson;
8
9 public class WriteJson01 {
10     public static void main(String[] args) {
11         try {
12
13             List<Profesor> profesores = Arrays.asList(
14                 new Profesor("Alberto Carrera", 10, 29.5,
15                     new String[]{"Acceso a datos", "Programación"}),
16                 new Profesor("Antonino Lasiera", 20, 38.5,
17                     null),
18                 new Profesor("Belén Carrera", 40, 22,
19                     new String[]{"Formación y Orientación Laboral", "Empresa e iniciativa emprendedora"}),
20                 new Profesor("Luis Aldea", 10, 9,
21                     new String[]{"Inglés I", "Inglés II, Informatica ESO"});
22
23             FileWriter fw = new FileWriter("Ficheros\\ProfesFPSierraGuara.json");
24             new Gson().toJson(profesores, fw);
25             fw.close();
26
27         } catch (Exception ex) {
28             ex.printStackTrace();
29         }
30     }
31 }

```

Ilustración 45. Clase WriteJson01

Se construye el fichero a partir de un objeto *Gson*, una lista de profesores, y un flujo de salida de caracteres (línea 25).

Una forma de recorrer el fichero anterior aparece a continuación:

```

3 import java.nio.file.Files;
4 import java.nio.file.Paths;
5 import java.util.Arrays;
6 import java.util.List;
7 import com.google.gson.Gson;
8
9 public class ReadJson01 {
10     public static void main(String[] args) {
11         try {
12             Gson gson = new Gson();
13             String sFichero = new String(Files.readAllBytes(Paths.get("Ficheros/ProfesFPSierraGuara.json")));
14             List<Profesor> profesores = Arrays.asList(gson.fromJson(sFichero, Profesor[].class));
15             for (Profesor p : profesores)
16                 System.out.println(p);
17             // Las dos líneas anteriores equivalen a --> profesores.forEach(System.out::println);
18         } catch (Exception ex) {
19             ex.printStackTrace();
20         }
21     }
22 }

```

Console [Java Application] C:\Program Files\Java\jre1.8.0\_261\bin\javaw.exe (20-ago-2020 17:47:36 – 17:47:38)

```

Profesor [nombre=Alberto Carrera, departamento=10, antigüedad=29.5, asignaturas=[Acceso a datos, Programación]]
Profesor [nombre=Antonino Lasiera, departamento=20, antigüedad=38.5, asignaturas=null]
Profesor [nombre=Belén Carrera, departamento=40, antigüedad=22.0, asignaturas=[Formación y Orientación Laboral, Em
Profesor [nombre=Luis Aldea, departamento=10, antigüedad=9.0, asignaturas=[Inglés I, Inglés II, Informatica ESO]]

```

Ilustración 46. Clase ReadJson01



El proceso inverso, pasar los datos del fichero JSON a una lista se realiza en la línea 14 a partir del método *fromJson* del objeto *gson* creado en la línea 12.

Lo que se ha hecho en primer lugar es leer todo el contenido del fichero (línea 13) y dejarlo en una cadena, *sFichero*. En lugar de utilizar un *String* o un objeto de la clase *File* para representar el fichero como hemos hecho en otras ocasiones, se utiliza la clase *Paths* (*java.nio.file.Paths* incluida en Java a partir de la versión 7) y su método estático *get*, pues ese tipo de argumento es el que hay que suministrarle al método *readAllBytes* de la clase *Files* (*import java.nio.file.Files* incluido en Java a partir de la versión 7) que lee todo el contenido del fichero.

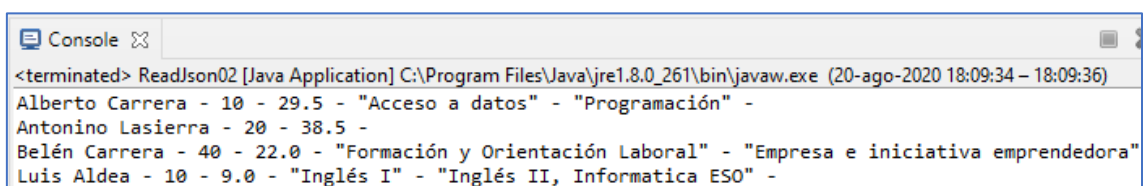
Otra forma de leer el documento, elemento a elemento, se muestra en la siguiente ilustración:

```

3 import java.nio.file.Files;
4 import java.nio.file.Paths;
5 import com.google.gson.JsonArray;
6 import com.google.gson.JsonElement;
7 import com.google.gson.JsonObject;
8 import com.google.gson.JsonParser;
9
10 public class ReadJson02 {
11     public static void main(String[] args) {
12         try {
13             String sFichero = new String(Files.readAllBytes(Paths.get("Ficheros/ProfesFPSierraGuara.json")));
14             // Utilizo métodos estáticos sin crear parser para eliminar "deprecated"
15             JsonArray gsonArr = JsonParser.parseString(sFichero).getAsJsonArray();
16             for (JsonElement obj : gsonArr) {
17                 JsonObject gsonObj = obj.getAsJsonObject();
18                 System.out.print(gsonObj.get("nombre").getAsString() + " - " +
19                     gsonObj.get("departamento").getAsInt() + " - " +
20                     gsonObj.get("antigüedad").getAsDouble() + " - ");
21                 if(gsonObj.has("asignaturas")) {
22                     JsonArray asignaturas = gsonObj.get("asignaturas").getAsJsonArray();
23                     for (JsonElement asignatura : asignaturas)
24                         System.out.print(asignatura + " - ");
25                 }
26                 System.out.println();
27             }
28         } catch (Exception ex) {
29             ex.printStackTrace();
30         }
31     }
32 }
33

```

Ilustración 47. Clase ReadJson02



```

<terminated> ReadJson02 [Java Application] C:\Program Files\Java\jre1.8.0_261\bin\javaw.exe (20-ago-2020 18:09:34 - 18:09:36)
Alberto Carrera - 10 - 29.5 - "Acceso a datos" - "Programación" -
Antonino Lasiera - 20 - 38.5 -
Belén Carrera - 40 - 22.0 - "Formación y Orientación Laboral" - "Empresa e iniciativa emprendedora"
Luis Aldea - 10 - 9.0 - "Inglés I" - "Inglés II, Informática ESO" -

```

Ilustración 48. Resultado de la ejecución de ReadJson02

En la línea 15 se deja en el objeto *gsonArr* el array de objetos profesor leídos del fichero. En la línea 16 se prepara para recorrer cada uno de esos objetos del array, imprimiendo sus valores a partir de sus nombres o claves. Como se conoce la estructura del documento y puede haber profesores que no impartan asignaturas, para evitar una excepción y finalización anormal del programa, se pregunta a cada objeto profesor si tiene asignaturas (línea 21). En ese caso se recuperan (línea 22) de manera similar a como se ha hecho antes con los profesores y se recorre dicho array de asignaturas para imprimirlo.

A continuación se comentan las clases utilizadas, algunas de ellas empleadas en el ejemplo anterior (líneas 5 a 7):

*JsonElement* representa cualquier elemento JSON que puede ser:



- *JsonObject*: Representa un objeto JSON.
- *JsonArray*: Representa un array JSON .
- *JsonPrimitive*: Representa un tipo de dato primitivo u objetos de datos simples (String p.ej.).
- *JsonNull*: Representa un objeto nulo.



Ya puedes realizar la actividad UD2.11.

## ACTIVIDADES

Los datos que utilizamos en los ejercicios se obtendrán principalmente bien solicitándolos desde la consola, o bien se pasarán como argumentos al método *main* desde el entorno de Eclipse. A continuación se hará un breve recordatorio de esta última forma. Utilizaremos la siguiente clase ejemplo y su array de argumentos (*String[] args*) del método *main* (línea 5):

```
3 public class EjemploPasoArgumentos {
4
5     public static void main(String[] args) {
6         if (args.length==0) {
7             System.out.println("No has enviado ningún argumento");
8             System.exit(-1);
9         }
10        for (int i =0; i<args.length;i++)
11            System.out.println("Argumento:" + (i+1) + "-> " + args[i]);
12    }
13 }
```

Ilustración 49. Clase EjemploPasoArgumentos

Una vez guardada, un instante antes de ejecutar la clase anterior, le enviaremos p.ej. 3 argumentos. Para introducirlos lo haremos desde la opción *Run > Run Configurations...* de Eclipse.

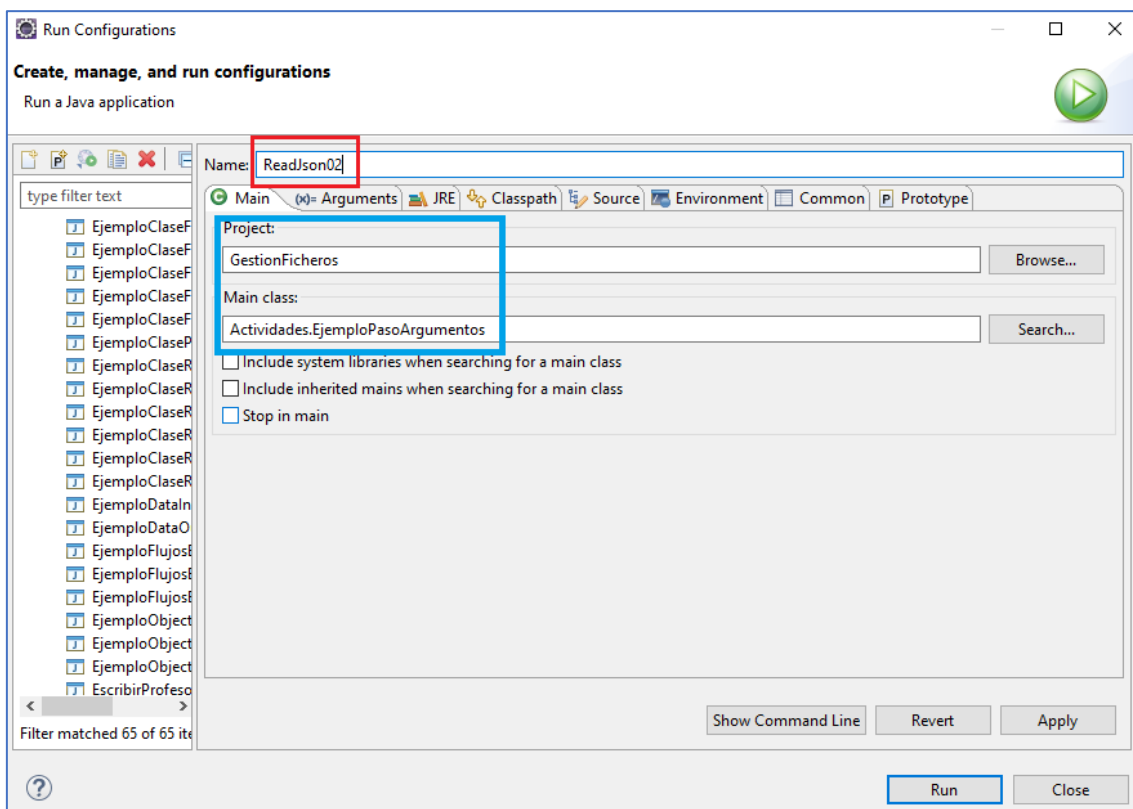


Ilustración 50. Run > Run Configurations... I

En la primera pestaña que aparece, *Main*, **es imprescindible elegir el proyecto y la clase** a cuyo método principal le pasamos los argumentos. En nuestro caso el proyecto es *GestionFicheros* y la clase *EjemploPasoArgumentos* que se encuentra dentro del paquete *Actividades* (**recuadro azul** de la ilustración anterior). Para localizar ambos, se utilizan los botones *Browse...* y *Search...*

Es importante prestar atención en este punto pues de lo contrario los argumentos que vayamos a introducir a continuación se pasarán al método *main* de la última clase que se ejecutó en Eclipse (llamada *ReadJson02* que aparece en recuadro rojo en la ilustración anterior) y nos llevará a una situación de confusión.

Una vez seleccionados el proyecto y la clase, se pincha en la segunda pestaña, *Arguments*, (ilustración siguiente) y se introducen los argumentos separados por un espacio en blanco en el campo de Texto *Program arguments* (los 3 argumentos introducidos son *Alberto Carrera 33*). Aunque los argumentos son de tipo *String*, no hace falta introducirlos entre comillas *""*. No te preocupes por el nombre que sigue apareciendo en el recuadro de texto *Name* (última clase que se ejecutó) pues en la pestaña *Main* previa ya se le han indicado los correctos.

Seguidamente se pulsa el botón *Apply* y por último el botón *Run*.

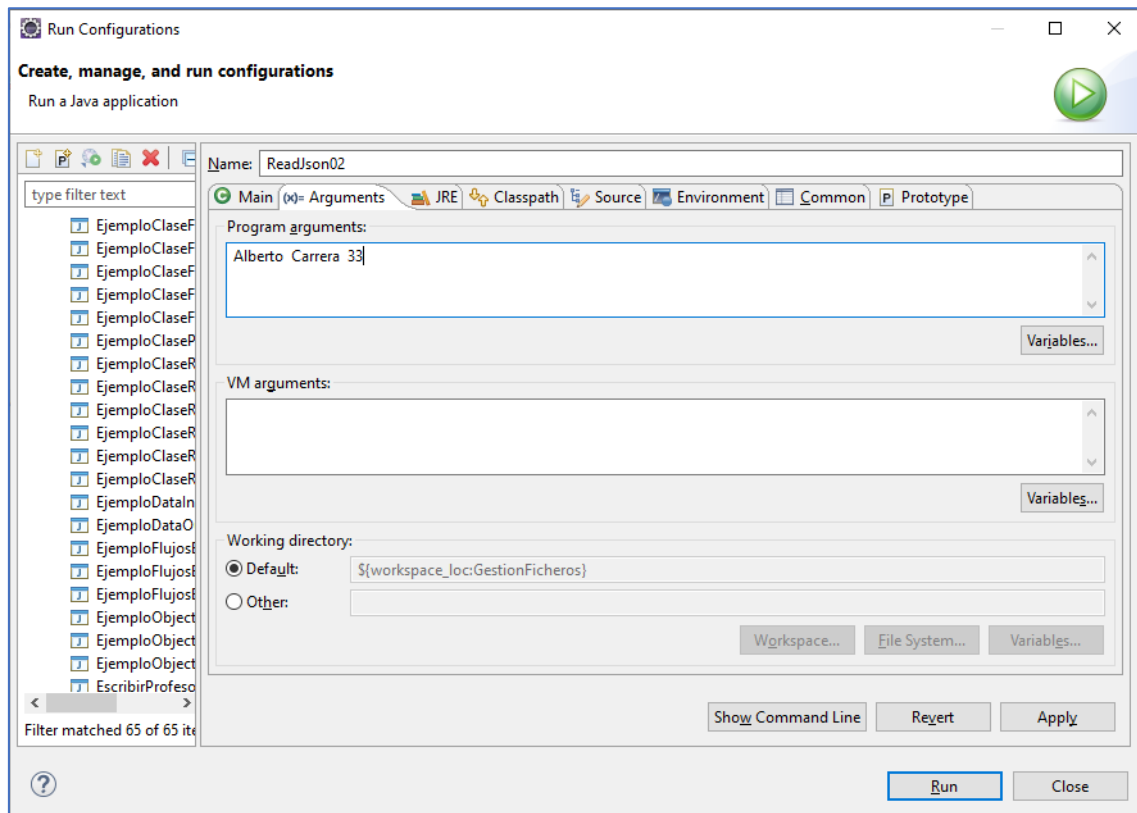


Ilustración 51. Run > Run Configurations...II

El resultado aparece en la siguiente ilustración:

```
1 package Actividades;
2
3 public class EjemploPasoArgumentos {
4
5     public static void main(String[] args) {
6         if (args.length==0) {
7             System.out.println("No has enviado ningún argumento");
8             System.exit(-1);
9         }
10        for (int i =0; i<args.length;i++)
11            System.out.println("Argumento:" + (i+1) + "-> " + args[i]);
12    }
13 }
```

Console

<terminated> ReadJson02 [Java Application] C:\Program Files\Java\jre1.8.0\_261\bin\javaw.exe (25-ago  
Argumento:1-> Alberto  
Argumento:2-> Carrera  
Argumento:3-> 33

Ilustración 52. Clase EjemploPasoArgumentos II

Los 3 argumentos se pasan dentro del array de cadenas *args* (línea 5), y en este caso se corresponden con:

args[0] → Alberto  
args[1] → Carrera  
args[2] → 33

Al principio del código podemos saber el número de argumentos pasados al método preguntando por el tamaño de dicho array (línea 6). Si dicho tamaño es 0 significa que no se le han pasado ningún argumento. En las líneas 10 y 11 se visualizan los argumentos enviados al método *main* y que serían los datos con los que trabajaría dicho método. Hay que indicar que los argumentos para dicha clase permanecerán siendo los mismos hasta que se borren o se sustituyan por otros. Los argumentos que correspondan a cantidades numéricas, deberán convertirse de cadena al tipo numérico correspondiente antes de operar con ellos.

**UD2.1** Realiza una clase UD2\_1 que muestre nombre, longitud, si se puede leer, si se puede escribir de todos los **archivos ocultos** de la carpeta Windows de tu disco; sólo de los que se encuentran en la carpeta principal de Windows, no en sus subcarpetas.

**UD2.2** Realiza una clase UD2\_2 cuyo método principal reciba como argumento una cadena con la trayectoria de un directorio o fichero e indique **si existe realmente o no** dicho directorio o fichero. Si el método principal no recibe ningún argumento se indicará por pantalla y finalizará su ejecución.

**UD2.3** Realiza una clase UD2\_3 que complete la clase *EjemploClaseFile02* de los materiales borrando el directorio y el fichero creados en ella (primero borra el fichero y después el directorio pues no se permite borrar directorios no vacíos).

**UD2.4** Realiza una clase UD2\_4 que guarde en un fichero con nombre *pares.txt* los números pares que hay entre 0 y 500, un número en cada línea del fichero. Seguidamente lee el fichero y muéstralo por la consola. Incluye también tratamiento de excepciones.

**UD2.5** Realiza una clase UD2\_5 que sea capaz de ordenar alfabéticamente las líneas contenidas en un fichero de texto (puedes crearlo con el bloc de notas). El nombre del fichero que contiene las líneas se debe pasar como argumento. El nombre del fichero resultado ya ordenado debe ser el mismo que el original añadiéndole la coletilla *\_sort* al nombre. Incluye también tratamiento de excepciones.

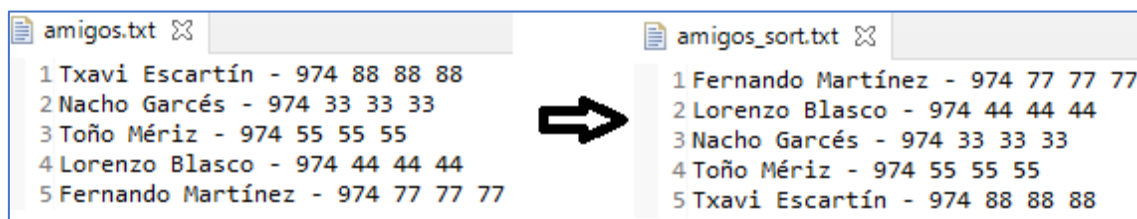


Ilustración 53. Ejemplo de ejecución

Pistas:

- Utiliza una colección (p.ej. *ArrayList*) donde guardar las líneas leídas del fichero y el método estático *Collections.sort* para ordenar dicha colección.
- Para formar el nuevo nombre de fichero puedes usar el método para cadenas *substring*.

**UD2.6** Realiza una clase UD2\_6 que indique cuántas veces aparece una palabra dentro de un fichero de texto (puedes crearlo con el bloc de notas). Tanto el nombre del fichero como la palabra se deben pasar como argumentos. No distinguir mayúsculas/minúsculas. Incluye también tratamiento de excepciones.

Pistas:

- Métodos para cadenas *substring* e *indexOf*.

**UD2.7** Realiza una clase UD2\_7 que guarde 20 números enteros aleatorios comprendidos entre 1 y 5 en el fichero *puntuación.dat*. Completa el código abriendo el fichero para visualizarlos todos por la consola indicando al final cuántas veces se repiten cada uno de ellos. Incluye también tratamiento de excepciones.

**UD2.8** Adaptación de los ejemplos vistos en los materiales con las clases *ObjectInputStream* y *ObjectOutputStream*. Realiza una clase UD2\_8 que pida al usuario datos de varios profesores

(nombre y la antigüedad) y los inserte en el fichero *antigüedad.dat\_obj.dat*. Si el fichero no existe se creará con los nuevos datos introducidos, en caso contrario se añadirán por el final. Antes de finalizar el código se recorrerá el fichero para visualizar su contenido. Prueba varias veces la ejecución de la clase.

**UD2.9** Adaptación de los ejemplos vistos en los materiales con la clase *RandomAccessFile*. Realiza una clase UD2\_9 que pida al usuario el identificador del profesor y lo borre del fichero *ProfesFPSierraGuara.dat*. Borrar un dato simplemente consiste en poner su campo id dentro del fichero a 0 para indicar que ese registro no existe y su posición está libre. Se deberá controlar que:

- El identificador del profesor esté dentro de los límites del fichero.
- El identificador del profesor debe existir. Si ha sido borrado previamente se advertirá de la situación.
- Antes de finalizar el código visualizar de manera secuencial todos los registros del fichero para comprobar la operación.

**UD2.10** Crea un paquete de nombre UD2\_10 e incluye en él todas las clases necesarias para construir el siguiente documento XML. Incluye otra clase que recorra el documento XML creado y visualice sus datos por la consola.

```
<?xml version="1.0"?>
- <películas>
  - <película>
    <id>1</id>
    <titulo>El Señor de los Anillos: la Comunidad del Anillo</titulo>
    <anyo>2001</anyo>
    <descripcion>Ambientada en la Tierra Media, cuenta la historia del Señor Oscuro Sauron, que está buscando el Anillo Único, el cual ha acabado en poder del hobbit Frodo Bolsón (Elijah Wood). El destino de la Tierra Media está en juego mientras Frodo y ocho compañeros que forman la Compañía del Anillo comienzan un largo y peligroso viaje hacia el Monte del Destino en la tierra de Mordor, que es el único lugar en el que el anillo puede ser destruido.</descripcion>
  </película>
  - <película>
    <id>2</id>
    <titulo>El Señor de los Anillos: las dos torres</titulo>
    <anyo>2002</anyo>
    <descripcion>La trama de la película comienza tras la disolución de la Compañía del Anillo. Boromir ha muerto a manos del jefe de los uruk-hai, Lurtz, en un intento de salvar a los hobbits Meriadoc Brandigamo y Peregrin Tuk, que acaban siendo capturados. Frodo Bolsón y Sam Gamyi parten solos hacia Mordor para destruir el Anillo Único en el Monte del Destino, mientras que Aragorn, Gimli y Legolas persiguen a los uruks con el fin de liberar a sus amigos capturados.</descripcion>
  </película>
  - <película>
    <id>3</id>
    <titulo>El Señor de los Anillos: el retorno del Rey</titulo>
    <anyo>2003</anyo>
    <descripcion>Trata sobre la última parte del viaje que emprendieron los nueve compañeros (de los cuales quedan solamente ocho) para salvar a la Tierra Media de la oscuridad impuesta por Sauron. En esta parte se decide el destino de todos los habitantes de estas tierras.</descripcion>
  </película>
</películas>
```

Ilustración 54. Documento XML del ejercicio UD2.10

**UD2.11** Crea un paquete de nombre UD2\_11 e incluye en él todas las clases necesarias para crear en disco un fichero JSON con la información que aparece en el ejercicio anterior. Incluye también las dos clases que recorren el documento JSON creado que se han explicado en los materiales.

## **ACTIVIDADES VOLUNTARIAS**

Para realizarlos debes **investigar por tu cuenta** la forma de resolverlos.

**UD2.12** Realiza una clase UD2\_12 con un método principal que reciba dos argumentos, un directorio y una extensión de fichero, e indique los datos (nombre, tamaño y fecha de creación) de los ficheros del directorio que tienen esa extensión. Pista: Interfaz *FilenameFilter*. Deberá comprobarse al principio del código que el directorio enviado como primer argumento existe, en caso contrario se indicará y finalizará su ejecución.

**UD2.13** Realiza una clase UD2\_13 que almacene como objetos en un fichero *ventas.dat* los datos básicos de los *clientes* como son el nombre completo (*String*), teléfono (*String*), dirección, (*String*), nif (*String*) y moroso (*String SI/NO*). Deberá codificarse para ellos 2 métodos:

- Introducir en el fichero anterior los datos de los clientes que se pedirán por teclado y se irán añadiendo al fichero. El atributo moroso no se incluirá en el fichero (aun así debe pedirse por teclado).
- Visualizar los datos del fichero.

Pista: Modificador *transient* a la hora de declarar el atributo moroso de la clase anterior.

**UD2.14** Realiza una clase UD2\_14 similar a la anterior pero utilizando la interfaz *Externalizable* y el fichero *ventas2.dat*.

**UD2.15** Crea un paquete de nombre UD2\_15 para realizar lo mismo que en el ejercicio UD2\_10 pero utilizando la *API DOM* de Java.

**UD2.16** Crea una clase de nombre UD2\_16 que lea el documento XML creado en el ejercicio UD2\_10 utilizando la *API SAX* de Java.

**UD2.17** A realizar en un paquete de nombre UD2\_17. Crea una plantilla *XSL* para dar una presentación (p.ej en forma de tabla) al fichero XML generado en el ejercicio UD2\_10 y realiza un programa Java para transformarlo en HTML.