







3. Almacenamiento en ficheros

 Date	Empty
 Status	Not started
 Type	Empty
Units	 8. Flujos de datos

Almacenamiento de información en ficheros

La clase File

Lectura y escritura de caracteres en ficheros

Lectura

Escritura

Lectura y escritura de bytes en ficheros

Lectura

Escritura

Lectura y escritura con flujo de datos

DataInput y DataOutput

DataInput

DataOutput

Trabajando con objetos: `ObjectStream`

`ObjectInput`

`ObjectOutput`

Ficheros aleatorios

`RandomAccessFile`

Almacenamiento de información en ficheros

La clase `File`

La clase `File` hace referencia a una dirección expresada en `String` donde se apunta al fichero – directorio con el que se quiere trabajar. Es importante tener en cuenta que esta ruta puede apuntar a una ruta absoluta o ruta relativa, siempre siendo recomendable utilizar la ruta relativa

```
File f = new File ("src/documentos/ejemplo.txt");
```

Otra cosa a tener en cuenta de un objeto de tipo `file` es que puede ser tanto un fichero final como un directorio.

```
public class EntradaIntro { public static void main(String[]  
args) { File f = new File("src/github/flujodatos/documento  
s"); System.out.println(f.isDirectory()); } }
```

Algunas de las acciones que permite esta clase son:

```
// obtiene la ruta del padre f.getParent() // evalúa si es un  
directorio f.isDirectory() // evalúa si es n fichero f.isFile  
(); // crea el directorio correspondiente f.mkdir(); // crea  
todos los directorios si no están creados hasta llegar al ind  
icado f.mkdirs(); // comprueba si existe f.exists(); // crea e  
l fichero indicado f.create(); // borra el fichero indicado  
f.delete(); // lista los archivos del directorio indicado f.l  
ist(); // lista los archivos del directorio indicado (en form  
ato file) f.listFiles(); // indica si se puede leer f.canRead  
(); // indica si se puede escribir f.canWrite(); // renombre  
el fichero indicado a un nombre dado f.renameTo():
```

```
... fichero asociado a un nombre dado (ejemplo: //)
```

Ejemplos a realizar (realizarlo con las comprobaciones de existencia):

1. Comprobar si no existe el directorio de un determinado fichero
2. Comprobar si un determinado fichero no existe
3. Borrar un fichero
4. Renombrar un fichero
5. Crear un fichero en un directorio conocido.
6. Comprobar cuales son todos los discos conectados al sistema

Modifica todos los ejemplos para pedir los nombres por consola

Lectura y escritura de caracteres en ficheros

Lectura

Para poder leer de un fichero lo primero que se necesita es un objeto de tipo `FileReader`, a la cual se le asigna un objeto de tipo `file`

```
File f = new File("ejemplo.txt"); FileReader fileReader = new  
FileReader(f); }
```

Antes de explicar el resto de elementos de este objeto es necesario saber que lo que obligatoriamente hay que hacer al terminar de leer un fichero es cerrar el flujo de datos.

```
fileReader.close();
```

El objeto `FileReader` por si solo no permite una lectura "comprensible" sino que lo hace en formato de bytes. Para poder trabajar de forma comprensible se utiliza un objeto de tipo `BufferedReader` a partir de un `FileReader`

Una vez creado el objeto `bufferedReader` se puede leer el contenido del fichero en formato carácter con el método `readLine()`

```
FileReader fileReader = new FileReader(f); BufferedReader buf  
feredReader= new BufferedReader(fileReader); bufferedReader.r
```

```
readLine();
```

Para la lectura completa de todas las líneas de un fichero:

```
String cadena; try { BufferedReader bufferedReader = new BufferedReader(new FileReader(new File("src/ejemplo.txt"))); while ((cadena = bufferedReader.readLine()) != null) { System.out.println(cadena); } bufferedReader.close(); } catch (FileNotFoundException e) { e.printStackTrace(); } catch (IOException e) { e.printStackTrace(); }
```

En el bloque while lo que se hace es leer cada una de las líneas siempre y cuando la lectura devuelva un valor diferente de null. El fichero parará de leerse cuando se encuentre con un EOF

Cuando se lee una palabra se puede aplicar multitud de métodos sobre esta todos ellos pertenecientes a la clase String que ya vimos en los primeros temas. Alguno de los ejemplos son:

```
// retorna un String[] con todas las palabras una vez se quita el carácter indicado
linea.split(","); // retorna la posición del carácter indicado
linea.indexOf("a"); // retorna el carácter de la posición indicada
linea.charAt(0); // retorna la última posición encontrada del carácter indicado
linea.lastIndexOf('a'); // retorna la palabra ubicada entre las posiciones indicadas
linea.substring(0,4); // elimina los espacios vacíos al principio y final de la línea
linea.trim();
```

En el caso de utilizar el método read() se obtiene un int, el cual es el número asociado al carácter (tabla ASCII), pudiendo convertirlo de la siguiente forma

```
public void conversionASCIIletra(File file){ try { FileReader fileReader = new FileReader(file); BufferedReader bufferedReader = new BufferedReader(fileReader); int linea; while ((linea = bufferedReader.read()) > -1){ // este es el código ASCII
System.out.println(linea); // Esta es la conversión a letra
System.out.println(Character.toChars(linea)); } } catch (FileNotFoundException e) { e.printStackTrace(); } catch (IOException e) { e.printStackTrace(); } }
```

Escritura

Para trabajar con la escritura el proceso es tremendamente similar, con la diferencia del cambio de clases.

```
File f = new File ("src/documentos/ejemplo.txt"); FileWriter f
r = new FileWriter(f); BufferedWriter bw = new BufferedWriter
(fr);
```

Los métodos más utilizados son

```
// escribe los datos en el fichero bw.write(); // escribe un
salto de linea en el fichero bw.newLine();
```

Del mismo modo que pasaba con la lectura, es obligatorio cerrar el flujo al terminar cualquier operación

Ejemplos:

1. Crea un programa que pida por consola los siguientes datos: nombre, apellido, edad. Tras pedir el último dato lo escribirá en un fichero llamado usuarios.txt (se creará si no existe en la raíz del proyecto). El programa seguirá pidiendo datos hasta que no se pare la ejecución

Lectura y escritura de bytes en ficheros

Para la lectura y escritura de bytes el proceso es muy parecido. Una de las primeras diferencias que encontramos es la necesidad de tener un objeto de tipo `FileInputStream`, que depende de un objeto de tipo `File`

```
File f = new File ("src/documentos/ejemplo.txt"); FileInputStr
eam fis = new FileInputStream(f);
```

Este flujo deberá ser cerrado cada vez que se termine la acción que se lleva a cabo

```
fis.close();
```

```
    fis.close(),
```

Lectura

Para la lectura se utiliza el objeto de tipo `FileInputStream` y el método `read`

```
FileInputStream fis = new FileInputStream(f); int caracteres;  
byte[] caracterByte = new byte[1]; while ((caracteres = fis.  
read(caracterByte)) != -1) { //devolución byte convertido a co  
digo ASCII System.out.println((char) caracteres); System.ou  
t.println(new String(caracterByte)); } fis.close();
```

Hay que tener en cuenta que este método me devuelve el valor numérico correspondiente, el cual hay que pasar a `byte` y a `String` consecutivamente para poder leer el contenido de forma comprensible

Escritura

Para la escritura se utiliza el objeto de tipo `FileOutputStream` y el método `write`. Para poder utilizar este método hay que pasarle una cadena de bytes, por lo que habrá que convertir el objeto a guardar en bytes

```
FileOutputStream fos = new FileOutputStream(f); String texto  
= "ejemplo de texto a guardar codificado en bytes"; byte[] ca  
racterByte = texto.getBytes(); System.out.println(caracterByt  
e); fos.write(caracterByte); fos.close();
```

Al trabajar todo el rato en bytes y tener que convertir de forma manual los datos, este proceso resulta bastante tedioso. Para poder mejorar el flujo de datos se utiliza la clase `DataStream`, la cual escribe bytes y lee el byte traduciéndolo a su tipo correspondiente

Lectura y escritura con flujo de datos

Del mismo modo que se ha utilizado antes las clase de `FileWriter` / `Reader` `BufferedWriter` / `Reader` para poder trabajar con un fichero leyendo y/o escribiendo caracteres en texto plano, también existe la posibilidad de escribir y/o leer datos en tipos primitivos concretos. En

los casos anteriores si se quería escribir el número 1 se escribía como un carácter, del mismo modo que si se recuperaba. En el segundo caso si se quería tratar como un número había que realizar un casteo al dato al que se quería pasar.

```
FileWriter writer = new FileWriter(f); BufferedWriter buffere  
dWriter = new BufferedWriter(writer); bufferedWriter.write(bo  
olean);
```

```
FileReader reader = new FileReader(f); BufferedReader buffere  
dReader = new BufferedReader(reader); int numero = Integer.pa  
rseInt(bufferedReader.readLine());
```

En muchas ocasiones esto no es muy práctico, ya que se necesita tanto guardad datos en un tipo concreto como recuperarlos en su tipo correspondiente. Para poder hacer esto se utiliza un flujo de streams que utiliza bytes tanto los fileinputstream (output) como los data input que se explican a continuación.

DataInput y DataOutput

Las clases DataInputStream y DataOutputStream permiten procesar ficheros binarios secuenciales de tipos básicos. Tanto la entrada como la salida tienen dependencias de los explicado en los puntos anteriores (FileStream y FileInput / FileOutput). Este tipo de flujos permiten guardad datos primitivos como tal, al mismo tiempo que recuperarlos de la misma forma.

DataInput

La clase DataInputStream permite leer registros, campo a campo, de ficheros binarios de tipos básicos.

```
DataInputStream fe = new DataInputStream(new FileInputStream  
(new File("src/ejemplo.bin")));
```

Los métodos que se pueden utilizar para la lectura son:

```
a = fe.readUTF(); a = fe.readChar(); a = fe.readBoolean(); a
= fe.readByte(); a = fe.readShort(); a = fe.readInt(); a = f
e.readLong(); a = fe.readFloat(); a = fe.readDouble(); a = f
e.readLine();
```

DataOutput

La clase `DataOutputStream` permite escribir registros de ficheros binarios de tipos básicos.

```
DataOutputStream fs = new FileOutputStream(new FileOutputStre
am(new File("src/ejemplo.bin")));
```

Los métodos más utilizados para la escritura son:

```
DataOutputStream dos = new FileOutputStream(new FileOutputStr
eam(f)); dos.writeChars("Ejemplo de escritura"); dos.writeCha
r('\n'); dos.writeInt(100);
```

Del mismo modo que se han escrito `char` o `int` se puede escribir cualquier tipo indicándolo en el método `writeXXX()`;

En todos los casos expuestos en el trabajo con ficheros, es necesario la captura de excepciones por la posibilidad de no existencia de los ficheros, errores de I/O, etc...

Trabajando con objetos: ObjectOutputStream

Las clases `ObjectInputStream` y `ObjectOutputStream` permiten procesar ficheros binarios secuenciales de objetos. La extensión del fichero deberá ser `.obj`

Antes de poder realizar la escritura y la lectura de elementos de tipo objeto, hay que tener en cuenta que los objetos leídos y/o escritos deben de ser de tipo `Serializable`. Para poder cumplir esto simplemente hay que implementar dicha interfaz, lo que permite utilizar el polimorfismo.

La implementación sería de la siguiente forma. Para un objeto de tipo `Usuario` que tiene varios atributos y que se quiere guardar en un fichero sería de la siguiente forma:

ficiero seria de la siguiente forma.

```
import java.io.Serializable; public class Usuario implements
Serializable { private String nombre, apellido; private int t
elefono; public Usuario(String nombre, String apellido, int t
elefono) { this.nombre = nombre; this.apellido = apellido; th
is.telefono = telefono; } public String getNombre() { return
nombre; } public void setNombre(String nombre) { this.nombre
= nombre; } public String getApellido() { return apellido; }
public void setApellido(String apellido) { this.apellido = ap
ellido; } public int getTelefono() { return telefono; } publi
c void setTelefono(int telefono) { this.telefono = telefono;
} }
```

ObjectInput

La clase `ObjectInputStream` permite leer registros de tipo un objeto. Hay que tener en cuenta que para poder guardar / leer un objeto este debe ser serializado (implementar la interfaz serializable) de forma que sea procesado por partes.

```
ObjectInputStream ois = new ObjectInputStream(new FileInputSt
ream(f)); Object o = ois.readObject(); for (Producto p: (Arra
yList<Producto>o ) { System.out.println(p.getNombre()); } oi
s.close();
```

ObjectOutput

Para la escritura se utiliza un objeto de tipo `ObjectOutputStream` con el método `writeXXX` y el tipo concreto. En este caso existe la posibilidad de ejecutar el método `writeObject()`, teniendo en cuenta que el objeto que se quiera escribir haya implementado la interfaz serializable

```
ArrayList<Producto> listaProductos = new ArrayList<Producto>
(); listaProductos.add(new Producto("Nombre1", "Descripcion
1", 12)); listaProductos.add(new Producto("Nombre2", "Descri
pcion2", 23)); listaProductos.add(new Producto("Nombre3", "Des
cripcion3", 43)); listaProductos.add(new Producto("Nombre4",
"Descripcion4", 62)); listaProductos.add(new Producto("Nombre
5", "Descripcion5", 145)); try { ObjectOutputStream oos = new
ObjectOutputStream(new FileOutputStream(f)); oos.writeObject
(listaProductos); } catch (IOException e) { e.printStackTrace(); }
```

```
(listaProductos); oos.close(); } catch (IOException e) { e.printStackTrace(); }
```

Cuando se intentan escribir objetos de tipo lista que apunta a otros objetos, estos también necesitan implementar la interfaz serializable: Por ejemplo si se tiene una clase que representa un Usuario:

```
import java.io.Serializable; public class Usuario implements
Serializable { private String nombre, apellido, dni; private
int telefono; public Usuario(String nombre, String apellido,
String dni, int telefono) { this.nombre = nombre; this.apelli
do = apellido; this.dni = dni; this.telefono = telefono; } pu
blic String getDni() { return dni; } public void setDni(Strin
g dni) { this.dni = dni; } public String getNombre() { return
nombre; } public void setNombre(String nombre) { this.nombre
= nombre; } public String getApellido() { return apellido; }
public void setApellido(String apellido) { this.apellido = ap
ellido; } public int getTelefono() { return telefono; } publi
c void setTelefono(int telefono) { this.telefono = telefono;
} public String mostrarDatos(){ return String.format("nombre:
%s, apellidos: %s, teléfono: %d %n", getNombre(), getApellido
(), getTelefono()); } }
```

Y un fichero que representa el conjunto de todos los usuarios, guardados en un ArrayList

```
package github.flujodatos.utils; import java.io.Serializable;
public class Usuario implements Serializable { private String
nombre, apellido, dni; private int telefono; public Usuario(S
tring nombre, String apellido, String dni, int telefono) { th
is.nombre = nombre; this.apellido = apellido; this.dni = dni;
this.telefono = telefono; } public String getDni() { return d
ni; } public void setDni(String dni) { this.dni = dni; } publ
ic String getNombre() { return nombre; } public void setNombr
e(String nombre) { this.nombre = nombre; } public String getA
pellido() { return apellido; } public void setApellido(String
apellido) { this.apellido = apellido; } public int getTelefon
o() { return telefono; } public void setTelefono(int telefon
o) { this.telefono = telefono; } public String mostrarDatos
(){ return String.format("nombre: %s, apellidos: %s, teléfon
o: %d %n", getNombre(), getApellido(), getTelefono()); } }
```

Y un fichero entrada donde se escriben los datos

```
package github.flujodatos.utils; import java.io.*; import java.lang.reflect.Array; import java.util.ArrayList; public class Agenda implements Serializable { private ArrayList<Usuario> usuarios; public Agenda() { usuarios = new ArrayList(); } public ArrayList<Usuario> getUsuarios() { return usuarios; } public void setUsuarios(ArrayList<Usuario> usuarios) { this.usuarios = usuarios; } public void agregarUsuario(Usuario usuario) { Object[] existe = existeUsuario(usuario.getDni()); if ((boolean) existe[0]) { System.out.println("el usuario ya existe"); } else { usuarios.add(usuario); } } public void borrarUsuario(String dni) { Object[] existe = existeUsuario(dni); if ((boolean) existe[0]) { System.out.println("el usuario ya existe"); usuarios.remove((int) existe[1]); } else { System.out.println("este usuario no existe"); } } public void listarUsuarios() { for (Usuario usuario : usuarios) { System.out.print(usuario.mostrarDatos()); } } private Object[] existeUsuario(String dni) { int i = 0; for (Usuario usuario : usuarios) { if (usuario.getDni().equals(dni)) { return new Object[]{true, i}; } i++; } return new Object[]{false, i}; } public void exportarAgenda(File f) { ObjectOutputStream oos; try { oos = new ObjectOutputStream(new FileOutputStream(f)); oos.writeObject(usuarios); } catch (IOException e) { e.printStackTrace(); } } public void importarAgenda(File f) { ObjectInputStream ois; try { ois = new ObjectInputStream(new FileInputStream(f)); usuarios = (ArrayList<Usuario>) ois.readObject(); } catch (ClassNotFoundException e) { e.printStackTrace(); } catch (IOException e) { e.printStackTrace(); } } }
```

Este fichero representa todos los mecanismos para poder agregar/eliminar elementos al arraylist así como cualquier manipulación de los datos, como por ejemplo escribirlos/leerlos a un fichero

Por último el fichero que representa la entrada. En este ejemplo se comprueba si el fichero de los datos existe, y en caso de ser positivo los importa al sistema para que se pueda funcionar con datos desde el inicio. Del mismo modo, cuando se termina la ejecución del programa se exportan todos los datos del sistema

```
import java.io.*; public class Entrada { public static void main(String[] args) { File f = new File("src/github/flujodatos/documentos/agenda.obj"); Agenda a = new Agenda(); if (f.exists()) {
```



```
ts()){ a.importarAgenda(f); } if (a.getUsuarios().size()>0){
a.listarUsuarios(); }else{ System.out.println("la agenda está
vacía"); } a.agregarUsuario(new Usuario("Jose", "Martin Pere
z", "000000A", 111111)); a.agregarUsuario(new Usuario("Pedr
o", "Lopez Merino", "000000B", 222222)); a.agregarUsuario(new U
suario("Luis", "Herrera Gomez", "000000B", 222222)); a.borrarUs
uario("000000C"); a.borrarUsuario("000000B"); a.listarUsuario
s(); a.exportarAgenda(f); } }
```

Ficheros aleatorios

Hasta este punto se ha trabajado con ficheros secuenciales (aquellos que deben ser leídos en orden). Java permite la creación de ficheros de acceso aleatorio que permite el acceso para escritura o lectura en un punto determinado.

RandomAccessFile

La clase RandomAccessFile permite procesar ficheros binarios con acceso directo de tipos básicos u objetos. La extensión del fichero deberá ser .raf

```
RandomAccessFile f = new RandomAccessFile(new File(src/ejempl
o.raf), "r");
```

La r del constructor hace referencia al tipo de apertura (lectura). Las posibilidades son rw

Los métodos más utilizados son:

```
// Cerrar un fichero abierto en modo lectura o escritura: f.c
lose(); // Obtener la posición actual del fichero: pos = f.ge
tFilePointer(); //pos es de tipo long // Obtener la longitud
del fichero: lon = f.length(); //lon es de tipo long // Posic
ionarse al principio del fichero: f.seek(0); // Posicionarse
al final del fichero: f.seek(f.length()); // Posicionarse a “
pos” bytes del principio: f.seek(pos); //Leer el siguiente ca
mpo del registro (“a” sería de tipo “String”, //“char”, “bool
ean”, “byte”, “short”, “int”, “long”, “float”, //“double” y d
e nuevo un “String”): a = f.readUTF().trim(); a = f.readChar
(); a = f.readBoolean(); a = f.readByte(); a = f.readShort();
a = f.readInt(); a = f.readLong(); a = f.readFloat(); a = f.r
```

```
readDouble()); a = f.readLine(); // Grabar el siguiente campo d
el registro ("a" sería de tipo // "String", "char", "boolean",
"byte", "short", "int", "long", // "float" y "double"): f.writ
eUTF(a); f.writeChar(a); f.writeBoolean(a); f.writeByte(a);
f.writeShort(a); f.writeInt(a); f.writeLong(a); f.writeFloat
(a); f.writeDouble(a);
```