MEDAC
Instituto Oficial de Formación Profesional

PROGRAMACIÓN

Operaciones con ficheros

7

ÍNDICE

OBJETIVOS

Almacenar información en ficheros de texto.

Almacenar información en ficheros binarios.

Leer información de ficheros de texto.

Leer información de ficheros binarios.

Crear y eliminar directorios.

Almacenar objetos en ficheros.

Leer objetos de ficheros.

/ 1. Introducción y contextualización práctica

En esta unidad, vamos a profundizar sobre el concepto de fichero.

Vamos a ver cómo podemos escribir y leer información de ficheros de texto, además de cómo podemos tratarlos de forma correcta.

Al igual que con los ficheros de texto, vamos a estudiar cómo podemos escribir y leer información de ficheros binarios.

También, vamos a aprender a almacenar y obtener objetos directamente de ficheros, es decir, lo que se conoce como la serialización de objetos.

Por último, nos detendremos en el tratamiento de directorios desde Java.

Escucha el siguiente audio en el que planteamos el caso práctico que iremos resolviendo a lo largo de esta unidad.

Fig. 1. Sistema de ficheros.

Audio intro. "Utilidad de ficheros"

https://bit.ly/3jRcLKN

/ 2. Excepciones y cierre

Ya conocemos de unidades anteriores cómo podemos tratar las excepciones que se lancen en nuestros programas con el bloque *try-catch*.

Cuando tratamos con ficheros, las excepciones van a estar muy presentes en nuestros desarrollos. Recordando lo estudiando en la unidad anterior, comentábamos que podremos tener las siguientes:

- ***FileNotFoundException:*** Esta excepción se lanzará siempre que la ruta del fichero al que queramos acceder no sea correcta o dicho fichero no exista.
- ***IOException:*** Esta excepción se lanzará siempre que:
  - El fichero al que queremos acceder no tenga permisos de lectura en el caso de querer leer información de él.
  - No tenga permisos de escritura en el caso de querer escribir información en él.
  - El fichero esté corrupto por cualquier motivo.
  - El usuario que estemos usando no tenga los permisos de lectura o escritura sobre ficheros.

Una de las acciones que vamos a tener que hacer con los ficheros es cerrarlos. Recordemos que el tratamiento de ficheros se basa en cuatro pasos:

1. Abrir.
2. Comprobar.
3. Operar con él.
4. Cerrar.

La parte de cierre es un paso obligatorio que tendremos que realizar se hayan producido excepciones con el tratamiento del fichero o no, es decir, en un principio, habría que hacerlo tanto en el bloque try como en el catch. No obstante, esa no es la forma más correcta de realizarlo, ya que puede fallar, por lo que vamos a introducir un nuevo bloque, el bloque *finally*.

Este bloque irá con los *try-catch* y se ejecutará después de que se haya ejecutado el *try*, en caso de que no haya errores, o el catch, en caso de que sí haya errores, es decir, se ejecutará siempre. Es aquí donde tendremos que cerrar el fichero.

Solo podremos tener un bloque *finally* en los bloques *try-catch*.

```
try
{

}
catch(FileNotFoundException | IOException error)
{
    System.out.println("Error: " + error.toString());
}
finally
{

}
```

Fig. 2. Bloque try-catch-finally.

/ 3. Escritura en ficheros de texto

Los ficheros de texto son los ficheros cuya codificación es en caracteres, esto quiere decir que podremos editarlos con cualquier editor de texto, como bloc de notas, por ejemplo.

Para escribir en un fichero de texto, necesitamos las siguientes clases:

- ***FileWriter:*** Con esta clase, podremos abrir un flujo de escritura carácter a carácter a un fichero de texto.
- ***PrintWriter*:** Con esta, podremos abrir un flujo de escritura línea a línea a un fichero de texto.

Los pasos a seguir para escribir en un fichero de texto son los siguientes:

1. Crear un objeto de tipo *FileWriter* a partir de la ruta del fichero. Con esto, crearemos el flujo de escritura hacia el fichero, pero solo podremos escribir carácter a carácter.
2. Crear un objeto de tipo *PrintWriter* a partir del objeto *FileWriter* para poder escribir cadenas de texto completas en una operación.
3. Escribir las cadenas que necesitemos a partir del método println de la clase *PrintWriter*. Este método funciona exactamente igual que cuando lo usamos para mostrar información en pantalla, escribirá una línea en el fichero e inmediatamente después, escribirá un salto de línea.
4. Cerrar el fichero. Para ello, utilizamos el método close de las clases *FileWriter y PrintWriter*. Es importante cerrar primero el objeto de *PrintWriter* y después el de *FileWriter.*

Una vez hayamos ejecutado el código, tendremos el fichero creado con el texto introducido.

Si el fichero en el que queremos escribir no existe, se creará, y si existe, se sobrescribirá.

Para que no se sobrescriba el texto, tendremos que pasarle un parámetro extra al constructor del objeto de *FileWriter* añadiendo un parámetro como true, que indicará que se escriba al final del fichero el texto nuevo.

Tienes un ejemplo completo de escritura en un fichero de texto en el archivo EjemploEscrituraTexto.java del apartado de *Recursos del tema*.

Fig. 3. Escribiendo en ficheros.

/ 4. Caso práctico 1: "Cerrando ficheros"

Planteamiento: Pilar y José están realizando su primer programa con ficheros. Es un programa muy simple, simplemente tienen que pedir 10 números y almacenarlos en un fichero de texto. Una de las cosas que tienen claras es que el fichero hay que cerrarlo. Pilar le comenta a José que tiene muy claro los pasos que hay que realizar con los ficheros, el abrirlos, comprobar que se han abierto correctamente, operar con ellos y cerrarlos, pero que no entiende muy bien la función del bloque *finally*: "Si puedo cerrar tanto en el try como en el catch, ¿para qué necesitamos el *finally*? Pase lo que pase, se va a cerrar igualmente". José asiente pensativo y no sabe qué contestarle a su compañera.

Nudo: ¿Crees que Pilar tiene razón? Si se puede cerrar el fichero tanto en el *try* como en el *catch*, ¿es innecesario el bloque *finally* para hacerlo?

Fig. 4. Cerrado.

Desenlace: Como bien saben nuestros amigos, cerrar el fichero después de trabajar con él es una parte importantísima, ya que si no lo cerramos, podrían producirse errores, como que no se guarde lo que hemos desarrollado o almacenado.

Como ya sabemos de otras unidades, cuando se lanza una excepción, todo el código que hay por debajo de ella no se ejecutará, yéndose al bloque *catch* a ejecutar el código de tratamiento de errores. Bien, puede ocurrir que si ha saltado una excepción y estamos en el boque *catch*, ocurra de nuevo otra excepción y la orden de cerrar el fichero que había preparada no se ejecute, quedándose este abierto con todo lo que ello pudiera conllevar. Por lo cual, es obligatoria la implementación del cierre del fichero dentro del bloque *finally*, para asegurar su cierre.

/ 5. Lectura de ficheros de texto

Para leer los datos en un fichero de texto, ya adelantamos algunas clases en el tema anterior, en concreto, necesitamos las siguientes:

- ***File*:** Con esta clase, podremos abrir un fichero.
- ***FileReader*:** Con la que podremos generar un flujo de lectura carácter a carácter a un fichero de texto.
- ***BufferedReader*:** Podremos generar un flujo de lectura línea a línea a un fichero de texto.

Los pasos a seguir para leer datos de un fichero de texto son los siguientes:

1. Creamos un objeto de tipo *File*.
2. Creamos un objeto de tipo *FileReader* a partir del objeto de tipo *File*. Con esto podremos leer datos carácter a carácter.
3. Creamos un objeto de tipo *BufferedReader* a partir del objeto de tipo *FileReader*. Con esto podremos leer líneas completas del fichero.
4. Leemos los datos que necesitemos. Para leer una línea, utilizaremos el método *readLine* de la clase *BufferedReader*, el cual nos devolverá la línea de texto correspondiente: devolviendo *null* si se ha llegado al final del fichero. Si queremos leer todo el contenido, lo haremos mediante un bucle *while*.
5. Cerrar el fichero. Para ello, utilizamos el método close de las clases *BufferedReader* y *FileReader*. Es importante cerrar primero el objeto de *BufferedReader* y después el de *FileReader*.

En caso de que el fichero que queramos leer no exista, se producirá un error. Los datos se leerán en el mismo orden en el que se escribieron.

Tienes un ejemplo completo de lectura en un fichero de texto en el archivo *EjemploLecturaTexto*.java del apartado de Recursos del tema.

Fig. 5. Leyendo texto.

Audio 1. "Leyendo datos"
https://bit.ly/353FqrN

/ 6. Escritura en ficheros binarios

Los ficheros de binarios **son los ficheros cuya codificación es en *bytes***, esto quiere decir que no podremos editarlos con cualquier editor de texto, como el bloc de notas. Si bien es verdad que cuando los tengamos preparados, podremos abrirlos con editores, pero esto se debe a que estos contienen ya soporte para la codificación binaria.

Para escribir en un fichero binario, necesitamos las siguientes clases:

- ***FileOutputStream*:** Con esta clase, podremos generar un flujo de escritura carácter a carácter a un fichero binario.
- ***BufferedOutputStream*:** Con esta clase, podremos generar un flujo de escritura línea a línea a un fichero binario.

En este tipo de ficheros, cuando escribamos una línea no se agregará el salto de línea al final, sino que tendremos que agregarlo nosotros manualmente.

Los pasos a seguir para escribir en un fichero binario son los siguientes:

1. **Crear un objeto de tipo *FileOutputStream*** a partir de la ruta del fichero. Con esto crearemos el flujo de escritura hacia el fichero, pero solo podremos escribir carácter a carácter.
2. **Crear un objeto de tipo *BufferedOutputStream*** a partir del objeto *FileOutputStream* para poder escribir cadenas de texto completas en una operación.
3. **Escribir las cadenas** que necesitemos a partir del método **write** de la clase *BufferedOutputStream*. Este método escribirá la línea sin un salto de línea, teniendo que escribirlo inmediatamente después para que no se escriba todo en una misma línea. Es importante no olvidar que tenemos que indicar el tipo de codificación que queremos usar, en este caso será la codificación UTF8, mediante la clase *StandardCharsets*. Como estos ficheros están codificados en *bytes*, tendremos que obtener los bytes de la cadena mediante el método *getBytes* de la clase *String*.
4. **Cerrar el fichero.** Para ello, utilizamos el método close de las clases *FileOutputStream* y *BufferedOutputStream*. Es importante cerrar primero el objeto de *BufferedOutputStream* y después el de *FileOutputStream*.

Al igual que con los ficheros de texto, con los ficheros binarios podremos indicar que se empiece a escribir la información en el final del fichero, no sobrescribiendo la información que ya hubiese.

Vídeo 1. "Escritura en un fichero binario"
https://bit.ly/2Z6tVMg

/ 7. Lectura de ficheros binarios

Para leer los datos de un fichero binario, necesitamos las siguientes clases:

- ***FileInputStream:*** Con esta clase, podremos generar un flujo de lectura carácter a carácter a un fichero binario.
- ***BufferedInputStream:*** Con esta, podremos generar un flujo de lectura línea a línea a un fichero binario.
- ***BufferedReader:*** Y con esta, generar un flujo de lectura línea a línea a un fichero binario.

Los pasos a seguir para leer datos de un fichero binario son los siguientes:

1. **Creamos un objeto de tipo *FileInputStream*.**

2. **Creamos un objeto de tipo *BufferedInputStream*** a partir del **objeto de tipo *FileInputStream*.** Con esto podremos leer datos carácter a carácter.

3. **Creamos un objeto de tipo *BufferedReader*** a partir del **objeto de tipo *BufferedInputStream*.** Con esto podremos leer líneas completas del fichero.

4. **Leemos** los datos que necesitemos. Para leer una línea, utilizaremos **el método *readLine* de la clase *BufferedReader,*** el cual nos devolverá la línea de texto correspondiente, devolviendo *null* si se ha llegado al final del fichero. Si queremos leer todo el contenido, lo haremos mediante un bucle *while*.

5. **Cerrar el fichero.** Para ello, utilizamos el método *close* de las clases *BufferedInputStream* y *BufferedReader*. Es importante cerrar primero el objeto de *BufferedReader* y después el de *BufferedInputStream*.

Al igual que con los ficheros de texto, se leerán los datos en el orden en el que se escribieron, además, como pasaba también con los de texto, en caso de que el fichero que queramos leer no exista, se producirá un error.

Al ser un fichero binario, tendremos que indicar la codificación que utilizamos a la hora de crear el objeto de tipo *BufferedInputStream*.

Tienes un ejemplo completo de lectura de un fichero binario en el archivo *EjemploEscrituraBinario.java* del apartado de *Recursos del tema*.

Fig. 6. Leyendo.

/ 8. Serialización. Interfaz serializable

Mediante la serialización, vamos a poder **escribir objetos en los ficheros,** esto quiere decir que en lugar de escribir dato por dato todas las propiedades de un objeto, **podremos escribir el objeto directamente,** ahorrándonos muchos accesos a disco innecesarios.

Para que un objeto pueda ser serializado, tiene que implementar obligatoriamente la **interfaz *Serializable***, en caso contrario, cuando nos dispongamos a escribirlo en el fichero, nos provocará un error en nuestro programa.

Los ficheros en los que vamos a escribir los objetos son ficheros binarios, por lo que vamos a poder aprovechar lo estudiado en los puntos anteriores.

Si bien anteriormente cuando habíamos creado un fichero binario, lo podíamos abrir sin problema con un editor de texto, en este caso no pasará lo mismo, ya que debido a la configuración de la interfaz *Serializable*, el fichero, al abrirlo, será indescifrable y solo veremos una serie de números hexadecimales que no tendrán ningún significado para nosotros, pero que están codificando toda la información de los objetos.

Cuando se serializan los objetos, todos sus atributos serán escritos, pero podemos indicar que un atributo no se serialice, para lo que usaremos el modificador *transient*. Cuando declaramos una variable como *transient,* no guardará su valor al ser serializado un objeto de esa clase, teniendo que volver a asignarlo manualmente al recuperarlo.

Para poder utilizar la interfaz *Serializable*, tendremos que hacer el siguiente *import:*

```
import java.io.Serializable;
```

En el caso de que queramos serializar una clase que tenga como atributo un objeto, ese objeto tendrá que ser serializable obligatoriamente.

Un ejemplo de una clase que se podrá serializar es el siguiente:

```java
import java.io.Serializable;

/**
 * Esta clase representa una persona
 * @author Francisco Jesús Delgado Almirón
 */
public class Persona implements Serializable
{
    private String nif;
    private String nombre;
    // Este atributo no se serializará
    private transient int edad;

    public Persona(String nif, String nombre, int edad) {
        this.nif = nif;
        this.nombre = nombre;
        this.edad = edad;
    }
}
```

Fig. 7. Ejemplo de clase serializable.

8.1. Escritura de objetos

Para serializar objetos en un fichero binario, necesitamos las siguientes clases:

- ***FileOutputStream*:** Con esta clase, podremos generar un flujo de escritura carácter a carácter a un fichero binario.
- ***ObjectOutputStream*:** Y con esta, podremos generar un flujo de escritura de objetos a un fichero binario.

Los pasos a seguir para escribir en un fichero binario son los siguientes:

1. **Crear un objeto de tipo *FileOutputStream* a partir de la ruta del fichero.** Con esto crearemos el flujo de escritura hacia el fichero, pero solo podremos escribir carácter a carácter.
2. **Crear un objeto de tipo *ObjectOutputStream* a partir del objeto *FileOutputStream*** para poder serializar objetos en el fichero.
3. **Escribir todos los objetos de una clase que implemente la interfaz Serializable mediante el método writeObject de la clase *ObjectOutputStream*.** Los objetos se irán escribiendo uno detrás de otro en nuestro fichero binario.
4. **Cerrar el fichero.** Para ello, utilizamos el método close de las clases *FileOutputStream* y *ObjectOutputStream*. Es importante cerrar primero el objeto de *ObjectOutputStream* y después el de *FileOutputStream*.

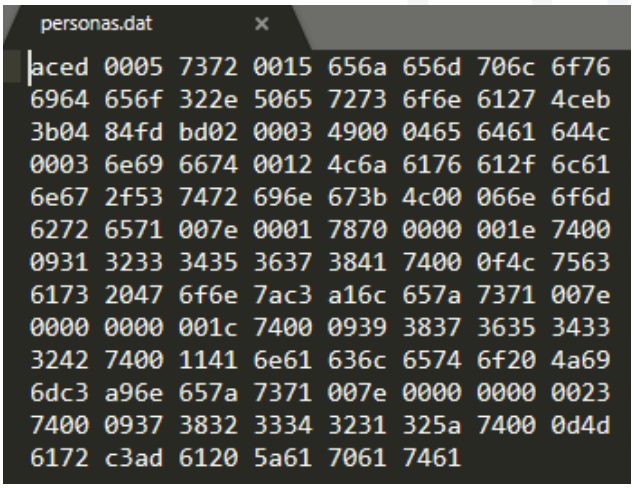

Fig. 8. Resultado de abrir un fichero serializado.

Al igual que con los demás tipos de ficheros, podremos indicar que se empiece a escribir la información en el final del fichero, no sobrescribiendo la información que ya hubiese.

Vídeo 2. "Serialización de objetos"
https://bit.ly/3bsNT94

8.2. Lectura de objetos

Para leer los objetos serializados en un fichero binario, necesitamos las siguientes clases:

- ***FileInputStream:*** Con esta clase, podremos generar un flujo de lectura carácter a carácter a un fichero binario.
- ***ObjectInputStream:*** Y con esta, podremos generar un flujo de lectura de objetos a un fichero binario.

Los pasos a seguir para leer datos de un fichero binario son los siguientes:

1. **Creamos un objeto de tipo FileInputStream.**
2. **Creamos un objeto de tipo *ObjectInputStream* a partir del objeto de tipo *FileInputStream*.** Con esto podremos leer objetos directamente.
3. **Leemos** los objetos que necesitemos. Para leer un objeto, utilizaremos el método readObject de la clase ObjectInputStream, el cual nos devolverá un dato de tipo Object, por lo que debemos hacer un casting al tipo de objeto que nosotros necesitemos.
4. **Cerrar el fichero.** Para ello, utilizamos el método *close* de las clases *FileInputStream y ObjectInputStream*. Es importante cerrar primero el objeto de *ObjectInputStream* y después el de.

Al igual que con los demás ficheros, se leerán los objetos en el orden en el que se escribieron. Al igual que pasaba con los demás ficheros, en caso de que el fichero que queramos leer no exista, se producirá un error.

```
public static void main(String[] args) {
    FileInputStream fis = null;
    ObjectInputStream entrada = null;
    Persona p;

    try {
        fis = new FileInputStream("personas.dat");
        entrada = new ObjectInputStream(fis);

        // Es necesario el casting
        p = (Persona) entrada.readObject();
        System.out.println(p.toString());
        p = (Persona) entrada.readObject();
        System.out.println(p.toString());
        p = (Persona) entrada.readObject();
        System.out.println(p.toString());
    } catch (IOException | ClassNotFoundException e) {
        System.out.println("Error 1: " + e.toString());
    } finally {
        try {
            entrada.close();
            fis.close();
        } catch (IOException e) {
            System.out.println("Error 2: " + e.toString());
        }
    }
}
```

Fig. 9. Ejemplo de lectura de objetos serializados.

8.3. SerialVersionUID

Cuando trabajamos con serialización de objetos, tenemos que tener mucho cuidado con las versiones de nuestros programas.

Ya sabemos, de unidades anteriores, que cuando compilamos nuestro proyecto, se compilarán todos los ficheros Java que tengamos y se crearán ficheros *.class*, que son los que tienen el código intermedio que es capaz de ejecutar la máquina virtual.

Cada vez que compilamos, esos ficheros de código intermedio se van actualizando. Hasta aquí todo bien, pero ¿qué ocurre con la serialización?

Cuando de un fichero binario, recuperamos un objeto que fue serializado previamente, se comprobará que la clase de la instancia del objeto recuperado sea exactamente igual que la que había cuando se instanció el objeto antes de serializarlo.

Puede ocurrir que el proyecto haya cambiado y algunos de esos cambios se hayan efectuado en la clase del objeto serializado (podemos haberle cambiado algún atributo, haberlo eliminado, o haberle agregado algún atributo extra porque necesitase cubrir alguna necesidad).

Estas comprobaciones se hacen mediante un número de versión, que si nosotros no indicamos, lo hará automáticamente el compilador. No obstante, cuando haya cambiado la clase, el compilador no cambiará la versión correctamente, y al intentar recuperar un objeto que cuando se serializó su clase era diferente, tendremos un error.

Para evitar este problema, se aconseja que la clase del objeto a serializar tenga un atributo privado de esta forma:

```
private static final long serialVersionUID = 1L;
```

Así, el número que le asignemos será diferente en cada versión de la clase.

```
public class Persona implements Serializable {

    private static final long serialVersionUID = 1L;
    private String nif;
    private String nombre;
    private int edad;
```

Fig. 10. Ejemplo de serialVersionUID.

/ 9. Caso práctico 2: "¿Cómo es más conveniente almacenar los datos?"

Planteamiento: Pilar y José siguen desarrollando la aplicación de la barra del bar. Esta vez se les ha pedido que una vez que hayan introducido la información de un pedido, esta se guarde en un fichero para que no se pierda y poder recuperarla cuando el cliente quiera pagar la cuenta.

En este punto, Pilar y José están discutiendo sobre la forma de almacenar la información. Pilar insiste en que es mejor guardar la información dato por dato, ya que se entiende mejor cómo está en el fichero, mientras que José insiste en que es mucho mejor guardar un pedido directamente mediante la serialización.

Nudo: ¿Qué piensas al respecto? ¿Cómo crees que sería mejor guardar la información, dato por dato o directamente el objeto mediante la serialización?

Fig. 11. Datos e información.

Desenlace: Cuando vayamos a guardar información, hay que pensar bien cómo queremos hacerlo, ya que eso afectará al diseño de nuestro programa.

Una de las cosas que nos tenemos que preguntar en este punto es cuánta información queremos almacenar. En el caso de que sea poca información, unos cuantos números, alguna cadena, etc., no merece la pena serializarla, pero en el caso de que tengamos mucha información, y además estructurada en clases, la mejor opción es hacer que esas clases puedan serializarse y ser almacenadas en bloque, ya que la forma de recuperarlas es muchísimo más simple y rápida.

/ 10. Creación y eliminación de directorios

Al igual que podemos crear y eliminar ficheros, el lenguaje de programación Java también nos permite crear y eliminar **directorios,** también conocidos como **carpetas.**

En primer lugar, veamos cómo sería la creación.

Para poder **crear un directorio,** vamos a utilizar la clase File, que era con la que creamos los ficheros. En esta ocasión, la ruta que le indiquemos al objeto que vamos a crear debe ser la de un directorio, no la de un fichero:

```
File directorio = new File("carpeta");
```

En la ruta de nuestro nuevo directorio, si está contenida la barra (\ ) en el *String* que contiene la ruta, deberemos indicarla dos veces, ya que es un carácter especial.

Para la gestión de directorios, podremos utilizar los siguientes métodos:

Método	Descripción
mkdir()	Crea una referencia al directorio, pero necesita que este exista; si no, lanzará una excepción IOException.
mkdirs()	Crea una referencia al directorio, pero si este no existe, lo creará automáticamente.

Tabla 1. Métodos para la creación de directorios.

Para comprobar que un objeto de la clase File contiene un directorio, podemos utilizar el método *isDirectory()*, que devolverá verdadero en caso de que lo sea y falso en caso contrario.

10.1. Eliminación de directorios

Al igual que podemos crear directorios, también vamos a poder eliminarlos.

Para poder eliminar un directorio, tenemos que tener en cuenta que no puede contener ficheros ni otros directorios.

Por tanto, en primer lugar, lo que tendremos que hacer es **cerciorarnos de que el directorio está vacío.** Esto lo podemos hacer mediante el siguiente código:

```
File[] directorios = directorio.listFiles();
```

Una vez hayamos ejecutado la función *listFiles()*, nos devolverá un array con todos los ficheros y directorios que hay dentro del directorio que tenemos.

Ahora, lo siguiente que tenemos que comprobar es que cada uno de esos objetos sean un fichero o un directorio. En caso de ser un fichero, habrá que borrarlo mediante el método *delete()*, y en caso de que sean un directorio, habrá que volver a listar lo que tenga dentro y hacer todas las comprobaciones.

En la siguiente figura, podemos ver el código necesario:

Mediante el método *isFile()*, podremos saber si el objeto que tenemos es un fichero, y mediante el método *isDirectory()*, podremos saber si el objeto que tenemos es un directorio.

Una vez el directorio esté vacío, podremos borrarlo mediante el método *delete()*.

```
File directorio = new File("carpeta");
File[] directorios = directorio.listFiles();

for( File d : directorios )
{
    // Si tengo un fichero lo borro
    if( d.isFile() )
    {
        d.delete();
    }
    else
    {
        // Es un directorio
    }
}
```

Fig. 12. Borrando ficheros.

/ 11. Resumen y resolución del caso práctico de la unidad

A lo largo de esta unidad, hemos visto que podemos realizar operaciones de **escritura y lectura** con varios tipos de ficheros.

Los ficheros que hemos estudiado son los **ficheros de texto** y los binarios, que, aunque tengan una codificación diferente, podemos tanto escribir como leer información de ellos.

Hemos comprobado que en el bloque try-catch es necesario agregar un nuevo bloque, el bloque ***finally***, que se ejecutará cuando termine el bloque *try o catch*, indistintamente, y que nos servirá para cerrar los ficheros adecuadamente.

También hemos aprendido a escribir y leer objetos directamente de ficheros mediante la serialización, y que para que esto funcione, la clase de la que queremos serializar los objetos tendrá que implementar la interfaz Serializable.

Por último, hemos visto cómo podemos tratar **directorios** desde Java, pudiendo crearlos, listarlos y eliminarlos.

Fig. 13. Soporte eficiente de la información.

Resolución del caso práctico inicial

Es posible que cuando empezamos a estudiar la utilización de ficheros en programación, nos hagamos la misma pregunta que Pilar y José: "Si ya existen las bases de datos, ¿para qué necesitamos almacenar la información en un fichero?".

En cierto modo, tienen razón. Posiblemente, el almacenamiento en bases de datos sea más eficiente y robusto que en ficheros, pero tenemos que ponernos en situación.

Imaginemos que necesitamos guardar cierta información en nuestro programa, pero no es una cantidad de información grande, sino que se trata de unos datos que necesitaremos en la siguiente ejecución del programa, ¿vale la pena montar una base de datos, con todo lo que ello conllevará, para almacenar unos pocos números?

La respuesta es no, así que para almacenar una cierta información no muy extensa, siempre nos serán de gran utilidad los ficheros.

/ 12. Bibliografía

Cuervo, V. (2015, junio 2). *Crear un directorio con Java IO*. Recuperado 28 de mayo de 2020, de http://lineadecodigo.com/java/crear-un-directorio-con-java-io/