ACCESO A DATOS

**TÉCNICO EN DESARROLLO DE APLICACIONES**

**MULTIPLATAFORMA**

# Introducción al manejo de ficheros

## Definición y tipos de ficheros

Definimos un fichero como la sucesión de bits que finalmente son almacenados en un determinado dispositivo. Está compuesto por nombre (identifica al fichero) y extensión (indica la tipología del archivo). Podemos organizar los ficheros en 2 grandes familias:

- **De texto (ASCII):** en su interior encontraremos líneas de texto organizadas en código ASCII.
- **Binarios:** aquellos ficheros cuya información está representada en código binario.

## La clase java.io.File

Nos será de gran ayuda para **crear, modificar y eliminar ficheros**, además de que nos proporcionará información adicional de los mismos. Puedes observar como se **instancia un objeto de clase *File***:

```
File fichero = new File (pathname: "/carpeta/archivo");
```

Una vez instanciado el objeto podremos también **crear dicho fichero**, para ello usaremos la siguiente línea:

```
fichero.createNewFile();
```

Por último, otra acción básica que deberemos de aprender para el uso y gestión de ficheros en Java es la **eliminación de ficheros** y directorios, usaremos la siguiente línea:

```
fichero.delete();
```

## Otros métodos de utilidad

Una vez realizado el aprendizaje de la instanciación, creación y borrado de un fichero, veamos en código sobre cómo **mover un fichero**:

```
package accesoADatosMoverFichero;

import java.io.File;

public class AccesoADatosMoverFichero {

	public static void main(String[] args) {

		try{

			File fileOrigen = new File("C:\\temp\\pruebas1.text");
			File fileDestino = new File("C:\\temp\\pruebas\\
                                        pruebas2.text");

			if(fileOrigen.renameTo(fileDestino)){
				System.out.println("El fichero se movió correctamente!");
			}else{
				System.out.println("El fichero no pudo moverse");
			}
		}catch(Exception e){
			e.printStackTrace();
		}
	}
}
```

Para poder mover un fichero, lo primero de lo que hay que asegurarse, es que éste debe de existir. Por otra parte, al cargar la ruta del fichero origen “fileOrigen” y ejecutar el método renameTo(), debemos indicar al método, el fichero destino “fileDestino”. Hay que tener en cuenta que la **ruta destino** debe de ser una **ruta operativa y existente**.

**Métodos más usados de la clase File**:

| Método | Descripción |
|---|---|
| createNewFile() | Genera el fichero indicado |
| delete() | Borra el fichero |
| mkdirs() | Crea directorio indicado |
| getName() | Devuelve String con nombre del fichero |
| getPath(), getAbsolutePath() | Devuelve la ruta relativa y absoluta |
| getParent() | Devuelve el directorio superior |
| renameTo() | Acepta como parámetro un nuevo File, el cual será la nueva ruta del fichero |
| exists() | Comprueba si existe el fichero |
| canWrite(), canRead() | Comprueba si puede ser escrito o leído |
| listFiles() | Devuelve un array con los ficheros del directorio indicado |
| lastModified() | Devuelve últimas modificaciones |

## Formas de acceder a un fichero

Cuando nos disponemos a acceder al contenido de un fichero es importante tener claro el criterio de acceso que vamos a usar. Disponemos de dos:

- Acceso **secuencial**.
- Acceso **aleatorio o directo**.

Previamente, tendremos que realizar un estudio del aplicativo a desarrollar para saber qué modo de acceso nos conviene. En el **modo secuencial**, como su nombre indica, la información de nuestro fichero es una secuencia de caracteres o bytes, de forma que, para acceder a un determinado byte o carácter del fichero, deberíamos de haber pasado previamente por todos los anteriores.

Si lo que deseamos es **acceder a un registro o posición determinada** de nuestro fichero, entonces nuestro método a escoger será el **modo aleatorio.**

La diferencia más notable entre estos dos tipos de acceso es la siguiente: mientras los ficheros de **acceso secuencial** deberán ser recorridos byte a byte o carácter a carácter con el tiempo, procesado

de la información, y uso de recursos que esto conllevaría, mientras que, en los **ficheros de acceso aleatorio o directo**, se establecerá un **puntero en bytes**, el cual indicará la posición exacta donde vamos a realizar la lectura y/o escritura, y al que se podrá **acceder directamente**.

Tabla orientativa sobre algunas de las **clases que podremos utilizar (métodos de acceso)**:

| Acceso basado en caracteres | | Acceso basado en Bytes | |
|---|---|---|---|
| Entrada | Salida | Entrada | Salida |
| FileReader | FileWriter | FileInputStream | FileOutputStream |
| | | RandomAccessFile | RandomAccessFile |

## Operaciones de gestión de ficheros. Clases asociadas

### Clases de uso de acceso secuencial

Dentro de la familia del acceso secuencial, debemos tener en cuenta la posibilidad de trabajar con bytes o caracteres según nuestra conveniencia. Para ello vamos a exponer algunas de sus diferencias.

### Acceso con bytes

Nos facilitan la lectura o escritura de un fichero como un **"stream" de bytes**. Para ello nos apoyaremos en la clase **FileInputStream (lectura** entrada de datos) y **FileOutputStream (escritura** salida de datos):

• **FileInputStream (lectura):** a continuación, se muestra un ejemplo de uso de esta clase:

```
InputStream entrada = new FileInputStream("prueba.txt");
```

Para empezar, instanciaremos el objeto de nuestra clase e indicaremos la ruta del fichero a cargar.

Con el método **read()** estaremos accediendo al **primer byte** del fichero. Éste devuelve un número entero, por lo tanto lo **asignamos a una variable entera**.

```
int data = entrada.read();
```

Una vez obtenido el primer byte, ya podremos trabajar con él. Con este método accedemos al primer Byte del fichero “en bruto” y nos servirá para **leer imágenes o archivos binarios**. Finalmente, cerramos el fichero y liberamos los recursos del sistema que estaban haciendo uso del “stream” de bytes.

A modo aclaración puntualizar que en estos ejemplos se ha suprimido el control de excepciones para mayor limpieza de código. Más adelante veremos cómo manejarlas.

**FileOutputStream (escritura)**

```
package uso_fileoutputstream;


import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;


public class CrearFicheroBytes {


	public static void main(String[] args) {
		String path = "salida.txt";
		String cadena = "Esto es una prueba de escritura";
		// pasamos a array de bytes la cadena de texto:
		byte[] arrayBytes = cadena.getBytes();

		try {
			// creamos el fichero en el path:
			FileOutputStream output = new FileOutputStream(path);
			// escribimos en el fichero:
			output.write(arrayBytes);
			output.close();
			System.out.println("Fichero escrito correctamente");
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			// TODO Auto-generated catch block
```

```
                e.printStackTrace();
            }
        }
}
```

## Acceso con caracteres

El modo de acceso secuencial con caracteres, hace posible la lectura y escritura de un fichero a través de un “stream” de caracteres. Mientras que las clases vistas previamente leen y escriben por medio de Bytes, las que veremos a continuación lo hacen **manipulando directamente caracteres**. Estas son: FileReader y FileWritter.

• **FileReader (lectura):** básicamente esta clase nos facilitará la lectura de los ficheros de caracteres. Pasamos a ver su utilización:

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;

public class LecturaStreamDeCaracteres {
      public static void main(String[] args) {
            try {
                  Reader lector = new FileReader("prueba2.txt");
                  int data = lector.read();
                  System.out.println((char)data);
                  lector.close();
            } catch (FileNotFoundException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      }
}
```

Como podemos observar, la lógica es muy parecida a la clase vista con anterioridad, FileInputStream. No obstante, ésta la usábamos para obtener los bytes “en bruto” directamente, mientras que la clase FileReader fue escrita con el propósito de leer **flujos** o “streams” **de caracteres**. Para ello, como podemos ver en el ejemplo, justo antes de mostrar el resultado por pantalla, realizamos un **casting (char)** para poder **visualizar** ese primer **carácter** de nuestro fichero “prueba2.txt”

• **FileWritter (escritura)**: con la siguiente clase seremos capaces de **escribir ficheros de caracteres**:

```
import java.io.Writer;
import java.io.FileWriter;
import java.io.IOException;

public class EscrituraStreamDeCaracteres {

	public static void main(String[] args) {
		try {
			// Se instancia clase de escritura y se indica ubicación de archivo
			// Si no existiera el archivo se crearía automáticamente
			// Si existe se sobrescribirá
			Writer escritorFicheros = new FileWriter("prueba.txt");
			escritorFicheros.write("Este es el contenido del fichero prueba.txt");
			escritorFicheros.close();

		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

En el código superior podemos observar como **instanciamos nuestra clase de escritura en la primera línea de código**, en donde, además, indicamos la ubicación. En el caso que **no existiera** el fichero, **se crearía** automáticamente.

Si el fichero **existe**, **será sobrescrito**.

En la segunda línea directamente insertamos los **caracteres que deseamos incluir en nuestro fichero.** Los pasamos como parámetro del método "**write()**".

Por último, **cerramos el flujo aplicando el método "close()"**, de esta forma liberaremos los recursos del sistema que han sido ocupados previamente para dicha operación. Una vez **cerrado el flujo** tendremos la **información escrita** en nuestro **fichero (si no cerramos el flujo no se escribe en el fichero)**.

## Clases de uso de acceso aleatorio

### Instanciación de RandomAcessFile

Utilizaremos la clase **RandomAcessFile** para acceder a ficheros de forma aleatoria. A diferencia de las clases anteriores, permite abrir un fichero en modo lectura y además en **modo lectura-escritura**.

Esta clase permite acceder a un lugar determinado de un fichero y a su vez, leer información o escribirla. Veamos en detalle las distintas **funcionalidades** que nos ofrece dicha **clase**.

**Modos de acceso**:

• **"r":** modo lectura (read mode) obtendremos un IOException si utilizamos este modo en métodos de escritura.

• **"rw":** modo **lectura y escritura**.

• **"rwd":** modo de lectura y escritura de forma **síncrona**. Se escribirán todas las actualizaciones del contenido del fichero.

• **"rws":** modo de lectura y escritura de forma **síncrona**. Se escribirán todas las actualizaciones del contenido del fichero pero además, se escribirán los **metadatos**.

Veamos ahora cómo instanciar este tipo de objeto y algunos de sus métodos:

RandomAccessFile posee dos **constructores**, en ambos constructores el **segundo parámetro** es el mismo: el **modo de acceso**. Sin embargo, para el **primer parámetro** tenemos la versión del constructor en la que podemos introducir el **objeto File directamente** o, como en el caso de la imagen superior, introducir la **ruta del fichero** directamente como String.

```
RandomAccessFile file = new RandomAccessFile("salida.txt","r");
```

Con el método seek() básicamente nos posicionamos en el punto que indiquemos del fichero. Acepta como parámetro un objeto de tipo "long".

```
file.seek(0);
```

Si usamos el método **getFilePointer()**, como bien se define, obtendremos como respuesta un tipo long. Este número es exactamente la **posición del puntero en Bytes.**

```
long filePointer = file.getFilePointer();
```

*Método Close automático*

*Algo que puede ser realmente interesante es el **cierre automático del flujo** de escritura en la clase RandomAccessFile. Simplemente si instanciamos un nuevo objeto de tipo RandomAccessFile dentro de una sentencia TRY, automáticamente el flujo será cerrado **cuando terminemos las operaciones** que esta **sentencia TRY** englobe con sus llaves.*

## Lectura y escritura con RandomAccessFile

Continuamos con nuestra clase RandomAccessFile, a continuación, veremos la forma de **leer y escribir** un **Byte** con esta clase:

Con el método **read()**, podremos leer un byte directamente de nuestro fichero. Devolverá dicho byte a partir de la **posición actual del puntero**.

```
int unByte = file.read();
```

**Escritura**

Usaremos el método **write()** para escribir un Byte. Dicho Byte será escrito en la posición actual donde se encuentre el **puntero**. Acepta como **parámetro** un **entero** (el **byte a escribir**). En este caso como podemos observar en la imagen, insertaremos la letra “D” en el fichero que corresponde a 68 en la codificación de caracteres ASCII.

```
RandomAccessFile file = new RandomAccessFile("entrada.txt", "rw");
long posicionPuntero = 7;
file.seek(posicionPuntero);
file.write(68);
RandomAccessFile fileLectura = new RandomAccessFile("entrada.txt", "r");
long posicionPunteroLectura = 0;
while (posicionPunteroLectura <= file.length()) {
                        fileLectura.seek(posicionPunteroLectura);
                        System.out.print((char) fileLectura.read());
                        posicionPunteroLectura++;
                    }
```

Una vez **escrito el carácter** en el fichero la posición del **puntero avanzará** una posición.

De la misma forma que hemos realizado una lectura y escritura de un byte, veremos cómo hacerlo de un **array de bytes** en los siguientes fragmentos:

Si en lugar de leer un byte, lo que nos interesa realmente es **leer** una **cantidad determinada de bytes**, podremos realizar una implementación del estilo a la imagen superior.

1) En primer lugar, crearemos un **array de bytes**, en concreto 1024.

```
byte[] arrayBytes = new byte[(int) pruebaFichero.length()];
```

2) A continuación, definimos la **posición del puntero** y el **tamaño de bytes** que queremos **extraer** de nuestro fichero.

```
int inicioPunteroMetodoRead = 0;
int size = 7;
long inicioPunteroLecturaSeek = 11;
                    file.seek(inicioPunteroLecturaSeek);
```

3) Por último, usamos la sentencia **read()**, que en este caso, acepta los siguientes parámetros:

- **el primer parámetro** el **array de bytes** dónde se va a **almacenar** la información,
- **el segundo parámetro**, el **inicio** o posición en la que comenzaremos a leer.
- **El último parámetro** indica la **cantidad de bytes** que deseamos **extraer**.

```
                    int bytesLeidos = file.read(arrayBytes,
inicioPunteroMetodoRead, size);
System.out.println("\n" + size + " bytes leídos, desde el puntero " +
inicioPunteroLecturaSeek + ":");
                    for (int n : arrayBytes) {
                        System.out.print((char) n);
                    }
```

Este método **devuelve** un número entero que representa la **cantidad de bytes** que han sido **leídos**.

## Operaciones con buffer

En este caso hablaremos de un conjunto de clases cuyo fin es leer y escribir información **mejorando el rendimiento del sistema** en comparación con nuestras clases aprendidas FileInputStream o FileOutputStream.

Concretamente, nos centraremos en las clases **BufferedInputStream, BufferedOutputStream, BufferedWriter y BufferedReader** que tienen una diferencia notable en relación con las estudiadas previamente.

Primero deberemos entender el concepto de **Buffer**.

Éste, básicamente es un **espacio determinado y temporal** que se aloja en **memoria** para realizar ciertas operaciones.

En las clases que vimos en apartados anteriores, realizábamos operaciones de lectura o escritura capturando Bytes, sin embargo, con las clases con sufijo Buffered el funcionamiento interno será ligeramente distinto ya que se **almacena** en memoria interna **bloques de bytes** completos (**buffer**), y como ya bien sabemos, el **acceso a memoria** es mucho **más rápido que a disco**.

Cada vez que **agotamos esa información** del buffer y se requiere más, se vuelve a **volcar otro bloque de bytes** a la memoria (buffer), de esta forma el rendimiento se ve notablemente mejorado.

A continuación, expondremos un ejemplo de clase **BufferedInputStream:**

```
BufferedInputStream flujoEntradaBuffer = null;
int bufferSize = 4 * 1024;
try {
                flujoEntradaBuffer = new BufferedInputStream(new
FileInputStream(archivoALeer), bufferSize);

                int datos;
while ((datos = flujoEntradaBuffer.read()) != -1) {
                        System.out.print((char) datos);
                }
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                try {
                        flujoEntradaBuffer.close();
                } catch (IOException e) {
```

```
                    e.printStackTrace();
                }
            }
        }
```

Definimos el **tamaño del buffer** que se usará posteriormente.

```
int bufferSize = 4 * 1024;
```

A continuación, se instancia nuestro **objeto**, el cual, necesita **dos parámetros**: El primero es un objeto de tipo **FileInputStream** con la **ruta** de nuestro **fichero (o el objeto File directamente)**, y el segundo parámetro es el **tamaño** que deseamos darle a nuestro **buffer**, definido en la primera línea.

```
flujoEntradaBuffer = new BufferedInputStream(
new FileInputStream(archivoALeer), bufferSize)
```

En el bucle "**while**" realizaremos la **lectura** de un bloque de datos siempre y cuando nuestra variable "info" no contenga -1, lo cual, si es así, indicará que el contenido del fichero ha finalizado y que nuestro buffer no tendrá información que proporcionar. Es dentro del bucle donde podremos manipular la información y realizar las operaciones que deseemos.

Por último, simplemente puntualizar la última línea como **cierre y liberación de recursos** de nuestro buffer.

```
flujoEntradaBuffer.close();
```

**Documentación API java:**

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