Manejo de Archivos

Leer y escribir archivos. Concepto de buffer.

Definición de Archivo

Desde el punto de vista de más bajo nivel, podemos definir un archivo (o fichero) como un conjunto de bits almacenados en un dispositivo, y accesible a través de un camino de acceso (pathname) que lo identifica.

Los archivos se usan para guardar los datos de nuestro programa para poder recuperarlos más adelante. Los archivos se usan cuando el volumen de datos no es relativamente muy elevado. Los archivos suelen organizarse en estructuras jerárquicas de directorios. Estos directorios son contenedores de ficheros (y de otros directorios) que permiten organizar los datos del disco. El nombre de un archivo está relacionado con su posición en el árbol de directorios que lo contiene, lo que permite no sólo identificar unívocamente cada fichero, sino encontrarlo en el disco a partir de su nombre.

Los archivos suelen tener una serie de metadatos asociados, como pueden ser su fecha de creación, la fecha de última modificación, su propietario o los permisos que tienen diferentes usuarios sobre ellos.

Archivos en Java

Para manejar archivos en Java se utiliza la clase **File**, representa un fichero y nos permite obtener información sobre él, como por ejemplo atributos, directorios, etc. Los archivos deben cerrarse si es que se realizo alguna operación con ellos. Posee tres constructores, siendo el primero el más utilizado:

- File(String path)
- File(String path, String name)
- File(File dir, String name)

El **path**, además de la ruta por la cual se llega al archivo, debe indicar el nombre de este último con su respectiva extensión, por ejemplo, “texto.txt”. La ruta es un string, cuya jerarquía empieza en la raíz del proyecto, si es que el archivo pertenece al mismo, empezando con “src/…” hasta llegar al fichero. **Crear un objeto de tipo File no significa que el archivo se abra o exista**, para ello hay distintos métodos que se van a ver a continuación.

Métodos importantes de File

- **String getName()** -> Retorna el nombre del archivo.
- **String getPath()** -> Retorna la ruta relativa del archivo.
- **String getAbsolutePath()** -> Retorna la ruta absoluta del archivo.
- **boolean exists()** -> Retorna true si el archivo existe.
- **boolean canWrite()** -> Retorna true si el archivo se puede escribir.
- **boolean canRead()** -> Retorna true si el archivo se puede leer.
- **boolean isFile()** -> Retorna true si el archivo es un archivo.
- **boolean isDirectory()** -> Retorna true si el archivo es un directorio.
- **long lastModified()** -> Retorna la fecha de la última modificación del archivo.
- **boolean renameTo(File dest)** -> Renombra el archivo.
- **boolean delete()** -> Elimina el archivo.

- **String[] list()** -> Retorna un array de String con los nombres de los archivos y directorios que contiene el directorio.
- **boolean mkdir()** -> Crea un directorio.
- **boolean mkdirs()** -> Crea un directorio y los directorios que no existan en la ruta.
- **boolean createNewFile()** -> Crea un archivo.
- **boolean isHidden()** -> Retorna true si el archivo está oculto.
- **long length()** -> Retorna el tamaño del archivo en bytes.

Hay que tener en cuenta la **diferencia entre ruta absoluta y relativa**. La primera comienza desde la raíz del sistema operativo y la segunda comienza desde el directorio donde se está ejecutando el programa, en nuestro caso, desde la carpeta src.

Otra diferencia importante es la de archivo y directorio, siendo este último un archivo que contiene uno o más archivos. Los directorios son conocidos como **carpetas.**

Concepto de Buffering

“Si no hubiera buffers, sería como comprar sin un carrito: debería llevar los productos uno a uno hasta la caja. Los buffers te dan un lugar en el que dejar temporalmente las cosas hasta que esté lleno. Por eso se realizan menos viajes cuando usas el carrito”.

Un buffer es un espacio de memoria temporal donde se almacenan datos. Sirve para optimizar el proceso de transferencia de datos entre un software y otro o entre un hardware y un software. No hay que confundirlo con el caché ya que el buffer solo guarda la información por el tiempo en que tarda el procesamiento de los datos.

Cualquier operación que implique acceder a memoria externa es muy costosa, por lo que es interesante reducir las operaciones de lectura/escritura que realizamos sobre los archivos. El buffer permite que la aplicación ignore los detalles concretos de eficiencia de hardware subyacente, encargándose de escribir en el disco siguiendo los ritmos más adecuados y eficientes.

Escribir en un archivo

Para escribir un archivo en Java se deben seguir tres pasos importantes:

1. Crear el objeto File que representa al archivo.
2. Crear un objeto de tipo **FileWriter**: El constructor de esta clase recibe como parámetro un objeto File y nos permite escribir en el archivo. Si el archivo no existe, lo crea y si existe, lo sobreescribe. Para que no pase esto último se debe pasar true (boolean) como segundo parámetro en el constructor. Para escribir el archivo se debe llamar al método append() del objeto FileWriter e indicar en sus parámetros el String a escribir. Este método se puede concatenar.
3. Cerrar el archivo con el método close() del objeto FileWriter. Si no se cierra el archivo no se escribe.

Los pasos 2 y 3 deben ir dentro de un bloque try-catch, ya que FileWriter puede arrojar una excepción, por ejemplo en el caso de escribir un archivo cuando no se poseen los permisos.

Escribir en un archivo con Buffering

Cada vez que se invoca el método append() de FileWriter se escribe en el disco. Si se tienen 1000 invocaciones de append, se escribirá 1000 veces en el disco, lo cual no es muy eficiente. Para ello se utiliza la clase **BufferedWriter,** que posee el mismo método pero no escribe en el disco cada vez que se llama Esta clase permite que las escrituras se vayan acumulando en el buffer hasta que se llene o se cierre el archivo. Una vez ocurre alguna de estas dos posibilidades, el buffer escribe en el disco. La eficiencia se nota cuando se tienen muchas escrituras en el archivo.

```
File file = new File(path);

try {

    BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(file, true));

    bufferedWriter.append("Hola mundo con BufferedWriter!" + "\n");

    bufferedWriter.close();

} catch (IOException e) {
    throw new RuntimeException(e);
}
```

Leer un archivo

Para leer un archivo se siguen los mismos pasos que al escribirlo, primero se crea el objeto File, luego se hacen las operaciones correspondientes y luego se cierra.

Para las operaciones de lectura se podría utilizar un objeto de tipo **FileReader** pero es conveniente utilizar **BufferedReader** por su eficiencia. Al instanciar un objeto de tipo BufferedReader se le debe pasar como parámetro otro de tipo FileReader. Para leer el archivo se debe utilizar el método readLine() que retorna un String. Se lee por líneas, si no hay líneas que leer, se retorna null, permitiendo leer el archivo a través de un while. **BufferedReader se encarga de recordar la última posición del fichero leído.** Tanto la instanciación de BufferedReader como sus operaciones deben estar dentro de un bloque try-catch, ya que si el archivo no existe se lanza una excepción.

Para que la lectura sea más cómoda se puede crear un objeto de tipo StringBuilder que vaya almacenando la información.