Ficheros

1 Clase File

La clase `File` del paquete `java.io` nos ofrece una abstracción para gestionar ficheros o directorios al margen de la plataforma en la cual se ejecute el programa (Windows, Linux…). Esta clase nos ofrece métodos para:

- Conocer las propiedades básicas de ficheros o directorios.
- Eliminar o renombrar ficheros o directorios, y crear directorios.

La creación de un objeto `File` se realiza, como es habitual, a través de su constructor:

```
File fd = new File("file.txt");  // nombre de fichero relativo al directorio actual
```

Esta sentencia crea un objeto `File` representando al fichero "`file.txt`" del directorio actual. Esto no implica necesariamente que el objeto exista realmente. De hecho, uno de los métodos de la clase es el llamado `exists` que devuelve `true/false` según el objeto exista o no. Podemos, lógicamente, crear objetos `File` con nombres de ficheros/directorios absolutos o relativos:

```
File fd = new File("/etc/file.txt");      // nombre  absoluto
File fd = new File("/etc", "file.txt"); // equivalente al anterior
File fd = new File("classes/file.txt"); // nombre relativo al directorio actual
```

La clase `File` intenta, en lo posible, abstraerse de las diferencias entre los distintos sistemas operativos, ofreciendo una sintaxis válida universalmente. Por ejemplo, la última de las anteriores líneas funcionará tanto en Linux como en Windows, a pesar de que en esta última plataforma el separador de directorios en una ruta es '\'. Por ello, resulta recomendable no usar nombres absolutos, sino siempre relativos al directorio actual.

Crear un objeto `File` no implica crear el fichero en caso de que no exista. Podemos tener un objeto `File` que represente a un fichero que aún no existe, y que no se creará hasta que escribamos en él.

Tras crear un objeto `File` asociado a un fichero o directorio, algunos de los métodos más útiles que podemos emplear son (en MOOVI hay un enlace a la documentación de la clase):

- `exists()` → nos dice si el fichero/directorio existe en realidad
- `isFile()` → devuelve `true` si existe y se trata de un fichero
- `isDirectory()` → devuelve `true` si existe y se trata de un directorio
- `canRead () / canWrite()` → devuelve `true` si existe y nos permite leer / escribir
- `length()` → nos devuelve su tamaño
- `lastModified()` → nos devuelve cuándo fue modificado por última vez (el número de milisegundos transcurridos desde el 1/1/1970).
- `delete() / renameTo()` → nos permite eliminarlo / renombrarlo

Como vemos, un objeto de la clase `File` nos permite algunas operaciones (crear un directorio, eliminar un fichero…) y conocer las propiedades de los objetos existentes, pero no nos ofrece ningún mecanismo para leer el contenido de un fichero existente o escribir en un fichero (nuevo o no).

1.1 Ficheros de texto o binarios

Los ficheros de texto son aquellos que almacenan en su interior caracteres representados en algún sistema de codificación (ASCII, ISO Latin-1, UTF-8…). Son legibles y tratables con un editor de texto.

Los ficheros binarios almacenan tipos de datos del lenguaje, ya sea básicos (`int`, `double`…) o más complejos (por ejemplo, un objeto de una determinada clase).

En esta práctica trabajaremos únicamente con ficheros de texto.

2 Escritura en un fichero

Para crear y escribir en un fichero de texto emplearemos la clase `PrintWriter`. De hecho, esa es la clase del objeto `out` que hemos usado hasta ahora cuando imprimimos en la consola. Nos basta crear un objeto `PrintWriter` asociado a un fichero en lugar de a la consola, y todo lo que hemos usado hasta ahora para imprimir en la consola (los métodos `print/println`) es perfectamente válido para escribir en un fichero.

Podemos crear un objeto `PrintWriter` a partir de un objeto `File` (la variable `fd`) asociado a un fichero:

```
PrintWriter pw = new PrintWriter(fd);
```

O incluso podemos crear el objeto `PrintWriter` utilizando directamente el nombre del fichero:

```
PrintWriter pw = new PrintWriter("file.txt");
```

En cualquiera de los dos casos, es en el momento de crear el objeto `PrintWriter` cuando se crea realmente el fichero. A partir de ese momento, podremos verlo en el sistema de ficheros. En caso de que el fichero ya exista, su contenido previo se pierde. Si queremos añadir contenido a un fichero ya existente, debemos emplear un método alternativo, que veremos más adelante.

Un ejemplo de programa completo que escribe en un fichero sería:

```
import java.io.File;
import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class WriteFile {

        public static void main (String[] args) throws FileNotFoundException {
                File fd = new File("file.txt");
                PrintWriter pw = new PrintWriter(fd);

                pw.println("Writing a text line to  a file");
                pw.println("16 34 56 32 78 32");
                pw.close();  // close the file
        }
}
```

Vemos que el programa importa la excepción `java.io.FileNotFoundException`, y que en el método `main` decimos que puede generar (`throws`) esa excepción. Esto se hace porque el constructor de la clase `PrintWriter` puede generar esa excepción (por ejemplo, si le pasamos un objeto `File` inválido). En esos casos, cuando en un método estamos utilizando una llamada que puede generar una excepción, estamos obligados a hacer una de las dos siguientes cosas:

- Capturar esa excepción con un `try/catch`.
- Declarar que nuestro método puede propagar esa excepción hacia arriba (mediante `throws`)

Recordemos que estamos trabajando con ficheros de texto. No importa lo que escribamos, en el fichero se escribirán caracteres en el sistema de codificación usado. Por ejemplo, cuando escribamos la cadena '16', lo que realmente se escribe al fichero son los códigos ASCII de los dígitos '1' y '6' (los números 49 y 54), equivalentes a su representación en UTF-8 (la codificación por defecto en Java).

Cuando llamamos a un `print` sobre un objeto `PrintWriter`, realmente no se escribe al fichero, sino a un búfer intermedio, que se vuelca al fichero cuando se llena[1]. Por eso, es importante cerrar (`close`) siempre un `PrintWriter` cuando ya no vamos a utilizarlo, pues eso nos asegura que el búfer es volcado al fichero. De lo contrario, si se produjera un error posterior en el programa, podría suceder que lo que hemos escrito no se traslade al fichero.

[1] Se hace así por eficiencia. Las operaciones de escritura se hacen por bloques, por lo que se tarda lo mismo en escribir un octeto que todo un bloque.

3 Lectura de un fichero

Hay varias posibilidades para leer de un fichero de texto, pero aquí nos vamos a centrar únicamente en una clase que ya conocemos, la clase `Scanner`. Para emplear esta clase para leer desde un fichero, basta crear el objeto `Scanner` asociándole el objeto `File` del fichero del que queremos leer, en lugar del `System.in` (el teclado) que empleamos en la práctica anterior.

```
File fd = new File("file.txt");
Scanner input = new Scanner(fd);
```

Tras esto, podríamos emplear el método que ya conocemos (`nextLine`) para leer línea a línea el contenido del fichero. Para proceder, lo único que nos queda por resolver es la forma de detectar el final del fichero. Y, para ello, podemos emplear un método de la clase `Scanner` (`hasNext`) que nos dice si quedan o no datos por leer.

Un ejemplo de programa completo que lee las líneas de un fichero sería:

```
import java.io.*;
import java.util.Scanner;

public class ReadFile {

        public static void main (String[] args) throws FileNotFoundException {

                Scanner input = new Scanner(new File("file.txt"));

                while (input.hasNext())
                        System.out.println(input.nextLine());

                input.close();
        }
}
```

Fíjate en varias cosas:

- En la primera línea importamos todas las clases de un determinado paquete de una vez, sin tener que ir una a una. Esto no supone ninguna sobrecarga, ya que las clases sólo se cargan si se utilizan.
- En la creación del `Scanner`, podemos crear el objeto `File` y el `Scanner` en una sola operación, sin necesidad de guardar el objeto `File`.
- En la lectura de las líneas, podemos leer del fichero y escribir en pantalla en una sola sentencia.

La combinación de dos sentencias en una sola que observamos en los dos anteriores puntos tiene una clara ventaja: reduce el número de líneas del programa. Pero también tiene dos inconvenientes:

- No conservamos el objeto `File` y la línea leída. El interés de esto dependerá de si necesitamos después el objeto `File` o la línea para algo más. Si fuera así, debemos separar ambas sentencias.
- Si se produce un error, resulta más difícil saber cuál es el origen (¿la construcción de `File` o de `Scanner`?). En estos primeros pasos en la programación en Java, donde los errores son frecuentes, resulta más recomendable no tratar de optimizar el programa, sino facilitar la depuración del mismo separando todas las sentencias para que el origen de los errores sea más fácil de identificar.

Cerrar el objeto `Scanner` (`close`) no es estrictamente necesario, pero hacerlo libera recursos (la memoria de las variables y las estructuras creadas para la tarea), y denota orden, algo muy conveniente para programar.

Este uso abstracto de la clase `Scanner`, para leer del teclado o de un fichero simplemente cambiando el objeto asociado que representa el origen de los datos, es directamente trasladable a otras fuentes. Por ejemplo, es posible leer datos de una cadena, simplemente empleándola como fuente al construir el objeto Scanner:

```
Scanner input = new Scanner("Juan 21 Pepe 213,5");
```