Leer y escribir ficheros de texto en Java

Contenido

La clase File 2
Ruta absoluta y relativa 2
Métodos básicos 3
Gestión de archivos 4
  Creación de archivos 4
  Creación de directorios 4
  Eliminación de archivos y directorios 5
  Cambio de nombre y de ubicación de archivos y directorios 6
Lectura de archivos 6
  Lectura de datos mediante Scanner 6
  Leer todo el texto de un archivo como una sola cadena 8
Escribir archivos 9
  La clase FileWriter 9
  Cierre de un filewriter 10
  La clase PrintWriter 10
Jerarquías de archivos 11
  Métodos para iterar a través de jerarquías de archivos 11
  Un ejemplo de jerarquía 11

A menudo se da el caso de que un programa necesita procesar y almacenar datos ubicados fuera del fichero en el que estamos programando: valores en ficheros de configuración, algún conjunto de datos para procesar, logs, etc.

La forma más sencilla de almacenar datos es utilizar archivos. Podemos considerar un **archivo como una colección de datos almacenados en un disco u otro dispositivo, y que se pueden manipular como una sola unidad mediante su nombre.** Los archivos se pueden organizar en directorios que actúan como carpetas para otros archivos y directorios.

NOTA: es lo mismo archivo que fichero, y también es lo mismo carpeta que directorio. Utilizaremos estos términos indistintamente.

La clase File

La clase File se encuentra en el paquete java.io. Un objeto de esta clase representa un archivo (que puede existir o no en el sistema de archivos) o un directorio. Esta clase se puede utilizar para manipular archivos y directorios: crear, eliminar, acceder a propiedades y mucho más.

La forma más sencilla de crear un objeto File es pasar la ruta de acceso en forma de cadena a su constructor. El formato válido del String depende del sistema operativo:

- **Windows utiliza barras diagonales inversas para rutas de acceso ('\'), mientras que**
- **Linux, OS X, Android y otros sistemas similares a UNIX utilizan la barra diagonal ('/').**

Si el sistema operativo es Windows, no olvides utilizar el carácter de escape '\'.

Para averiguar el carácter para separar la ruta de acceso a un archivo que utiliza tu equipo, puedes imprimir lo siguiente:

```
System.out.println(File.separator);  // ‘\’ para Windows,'/' para Linux
```

Vamos a crear dos objetos de la File clase para diferentes plataformas.

```
File ficheroEnUnix = new  File("/home/username/Documents");     // un directorio en un archivo de sistema similar a UNIX

File ficheroEnWin =  new File("D:\\Materials\\java-materials.pdf"); // un archivo en Windows
```

El código funcionará incluso si un archivo o un directorio no existe realmente en el sistema de archivos. No crea un nuevo archivo o directorio. Solo representa "un archivo o directorio virtual" que ya existe o se puede crear en el futuro.

Los objetos de la clase `File` son inmutables; es decir, una vez creado, el nombre de ruta abstracta representado por un objeto nunca cambiará.

Ruta absoluta y relativa

Una ruta de acceso es **absoluta** si comienza con el elemento raíz del sistema de archivos. Contiene la información completa sobre la ubicación del archivo. Se considera como una mala práctica localizar un archivo utilizando su ruta absoluta, ya que se pierde la capacidad de ejecutar este programa en diferentes plataformas**.**

Una **ruta de acceso relativa** es una ruta de acceso que no incluye el elemento raíz del sistema de archivos. Siempre comienza **desde el directorio de trabajo.** Este directorio está representado por un . (punto). El carácter de punto se puede omitir al representar una ruta relativa.

Con el fin de construir programas independientes de la plataforma, es una convención común utilizar la ruta relativa siempre que sea posible.

Métodos básicos

Una instancia de `File` proporciona una lista de métodos para gestionar archivos. Algunos de ellos:

- `String getPath()` devuelve la ruta de acceso de cadena a este archivo o directorio;
- `String getName()` devuelve el nombre de este archivo o directorio (solo el apellido de la ruta de acceso)
- `boolean isDirectory()` devuelve `true` si es un directorio y existe, de lo contrario, `false;`
- `boolean isFile()` devuelve `true` si es un archivo que existe (no un directorio), de lo contrario, `false;`
- `boolean exists()` devuelve `true` si este archivo o directorio existe realmente en el sistema de archivos, de lo contrario, `false;`
- `String getParent()` devuelve la ruta de acceso de cadena al directorio primario que contiene este archivo o directorio.

La lista no está completa, pero por ahora, nos centraremos en estos.

Vamos a crear una instancia de un archivo existente e imprimir alguna información al respecto.

```
File = new  File("/home/username/Documents/javamaterials.pdf");
System.out.println("Existe: " + file.exists());
System.out.println("Nombre del archivo: " + file.getName());
System.out.println("Ruta del archivo: " + file.getPath());
System.out.println("Es archivo: " + file.isFile());
System.out.println("Es directorio: " + file.isDirectory());
System.out.println("Está en la carpeta:" + file.getParent());
```

Como era de esperar, el código imprime lo siguiente:

```
Nombre del archivo: javamaterials.pdf
Existe:  true
Ruta del archivo: /home/username/Documents/javamaterials.pdf
Es archivo: true
Es directorio:  false
Está en la carpeta: /home/username/Documents
```

Supongamos que ahora tenemos un objeto que representa un archivo inexistente e imprime la información al respecto:

```
Nombre del archivo: javamaterials1.pdf
Existe:  false
Ruta del archivo: /home/art/Documents/javamaterials1.pdf
Es archivo: false
Es directorio:  false
Está en la carpeta:/home/art/Documents
```

También hay un grupo de métodos `canRead()`, `canWrite()`, `canExecute()` para probar si la aplicación puede **leer / modificar / ejecutar** el archivo denotado por la ruta de acceso. Se recomienda utilizar estos métodos para asegurar que el usuario tiene suficientes permisos para realizar una operación con un archivo.

Gestión de archivos

Además de simplemente acceder a los ficheros, también podemos administrar archivos y directorios mediante su creación, eliminación y cambio de nombre. Hay varios métodos para hacerlo. En el ejemplo siguiente, trabajamos con un sistema operativo tipo UNIX.

Creación de archivos

Para crear un archivo en el sistema de archivos, tendremos que hacer lo siguiente:

1. Crear una instancia de `java.io.File` con la ruta especificada.
2. Invocar el método `createNewFile` de esta instancia. El método devuelve `true` si el archivo se creó correctamente y `false` si ya existe. No borra el contenido de un archivo existente.

```
File fichero = new  File("/home/username/Documents/file.txt");
try  {
    boolean creado  = fichero.createNewFile();
    if  (creado) {
        System.out.println("El archivo se creó correctamente." );
    }  else  {
        System.out.println("El archivo ya existe." );
    }
}  catch  (IOException e) {
    System.out.println("No se puede crear el archivo: " + file.getPath());
}
```

Trata de jugar con este código para entenderlo mejor.

Podríamos preguntarnos: "¿por qué devuelve el método false en lugar de producir una excepción cuando el archivo ya existe"? La respuesta es que a veces no importa para el programa si el archivo fue creado en este momento o ya existía.

Creación de directorios

Para crear un directorio también tenemos que empezar creando una instancia de `java.io.File`. Después de eso, debemos llamar a uno de los dos métodos de esta instancia:

- `boolean mkdir()`: crea el directorio; devuelve true sólo si se creó el directorio, de lo contrario devuelve false.
- `boolean mkdirs()`: crea el directorio que incluye todos los subdirectorios no existentes necesarios.

En el ejemplo siguiente se muestra el método `mkdir`.

```
File fichero = new file("/home/art/Documents/dir");
boolean directorioCreado = fichero.mkdir();
  if  (directorioCreado) {
    System.out.println("Se creó correctamente." );
}  else  {
    System.out.println("No se creó." );
}
```

Para entenderlo mejor, prueba este código en el equipo especificando diferentes rutas de acceso al archivo.

Por lo general, el código funciona de la siguiente manera: si el directorio no existe, este código lo crea. Si el directorio existe, el código no lo creará. Si hay un directorio no existente en la ruta de acceso, el directorio tampoco se creará. No se producen excepciones en ningún caso.

Aquí hay otro ejemplo, que muestra el método mkdirs. Crea el directorio de destino y todos los directorios intermedios si no existen.

```
File fichero = new  File("/home/art/Documents/dir/dir/dir");
boolean directorioNuevoCreado = fichero.mkdirs();
  if  (directorioNuevoCreado) {
    System.out.println("Se creó correctamente." );
}  else  {
    System.out.println("No se creó." );
}
```

La variable boolean es true si se crea el directorio de destino, independientemente de la existencia de los directorios intermedios de la ruta.

Eliminación de archivos y directorios

Ahora que sabemos cómo crear un directorio, averigüemos cómo deshacernos de uno.
Hay un método denominado delete() que devuelve true si el archivo o directorio se elimina correctamente, y false si no puede eliminarlo, bien porque no existe, bien porque no está vacío (este método, pues, no puede eliminar una jerarquía de ficheros y directorios, sólo un archivo determinado o un directorio vacío). También puede suceder que nuestro usuario no tenga permisos para eliminar el archivo o directorio.

```
File archivo = new  File("/home/art/Documents/dir/dir/dir");
  if  (archivo.delete()) {
    System.out.println("Se eliminó correctamente." );
}  else  {
    System.out.println("No se eliminó." );
}
```

Para eliminar un directorio que no está vacío, antes hay que eliminar todos los archivos y directorios anidados. El código siguiente recursivamente directorios con su contenido. Ten en cuenta que el método asume que el directorio que se pasa por parámetro (dir) existe. De lo contrario, se producirán excepciones (children == null y NullPointerException).

```
public void deleteDirRecursively(File dir) {
File[] children = dir.listFiles();
    for  (File child: children) {
        if (child.isDirectory()) {
              deleteDirRecursively(child);
        } else  {
            child.delete();
        }      }
    dir.delete();
}
```

Este método nunca produce una excepción IOException.
Otro método para eliminar archivos es deleteOnExit y elimina un archivo o un directorio cuando el programa se detiene. Ten en cuenta que una vez solicitada la eliminación, no hay forma de cancelarla.

Cambio de nombre y de ubicación de archivos y directorios

El método renameTo() cambia el nombre del archivo. Devuelve true si y sólo si el cambio de nombre se realizó correctamente; de lo contrario, false.

```
File fichero = new File("/home/art/Documents/dir/filename.txt");
boolean renombrado = fichero.renameTo(new
File("/home/art/Documents/dir/newname.txt"));
```

El mismo método se puede utilizar para mover el archivo o directorio de la ubicación actual a otra:

```
File fichero = new File("/home/art/Documents/dir/file.txt");
boolean movido = fichero.renameTo(new
File("/home/art/Documents/another/file.txt"));
```

Muchos aspectos del comportamiento de este método siguen siendo dependientes de la plataforma. Es posible que no se pueda mover un archivo de un sistema de archivos a otro y es posible que no se realice correctamente si ya existe un archivo con el mismo destino. El valor devuelto siempre debe comprobarse para asegurarse de que la operación se realizó correctamente.

```
File fichero = new  File("/home/art/Documents/dir/filename.txt");
File rebautizado = new File("/home/art/Documents/dir/newname.txt");
boolean renombrado = fichero.renameTo(rebautizado);
if (renombrado) {
    System.out.println("Se cambió correctamente el nombre.");
} else {
    System.out.println("No se cambió el nombre.");
}
```

El método renameTo() produce NullPointerException cuando el archivo de destino es null.

Lectura de archivos

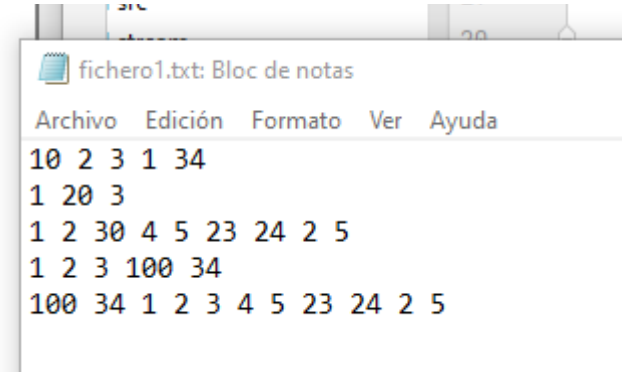
La biblioteca de clases estándar de Java proporciona varias formas de leer datos de archivos. Algunas de ellas son bastante antiguas, otras han aparecido recientemente. En este tema consideraremos sólo dos métodos básicos.

Lectura de datos mediante Scanner

Nuestra vieja conocida clase `java.util.Scanner` puede usarse también para leer datos de archivos. Esta clase es un enfoque de alto nivel para leer datos de entrada. Permite leer tipos o cadenas primitivas mediante expresiones regulares.
entrada estándar.
En primer lugar, necesitamos creear una instancia de `java.io.File` y, a continuación, una instancia de `Scanner` pasando el objeto `File` por parámetro.

A partir de aquí ya podemos obtener el contenido del archivo de la misma manera que leemos de la entrada estándar (System.in).

Supongamos que tenemos un archivo que contiene una secuencia de números separados por espacios en una ruta que tenemos guardada bajo el nombre ruta:



Vamos a crear un objeto File y, a continuación, un scanner para leer datos del archivo.

```
File f = nuevo File(ruta);
Scanner lector = new Scanner(f); //  Puede saltar una FileNotFoundException
```

Al crear una instancia de `Scanner` pasando un archivo, hay que controlar la excepción `FileNotFoundException`.

Ahora, podemos usar métodos de `Scanner` para leer datos como cadenas, enteros, etc. La manera de utilizar los métodos aquí es similar a como hacíamos con los iteradores: vamos recorriendo el fichero mientras queden líneas por leer. En caso de que solicitemos una lectura cuando ya no queda nada por leer obtendremos la excepción `NoSuchElementException`.

Vamos a leer el fichero línea a línea:

```
while (lector.hasNext()) { System.out.print(scanner.nextLine());}
```

Este código lee cada línea del archivo y la envía a la salida estándar.

Como el fichero solamente contenía números, también podíamos haber leído número a número, así:

```
while (lector.hasNextInt()) {
    int n = lector.nextInt();
    System.out.println(n);
}
lector.close();
```

Después de usar un `scanner`, debemos cerrar el objeto. Se puede hacer con el método `close()` o bien con la instrucción try-with-resources, que sirve a la vez para cerrar el scanner (y liberar así la memoria) y controlar las excepciones. como se muestra a continuación.

```
File f = new File(ruta);
try (Scanner lector = new Scanner(f)) {
    while (lector.hasNext()) {
        System.out.print(lector.nextLine() + " ")
    }
```

```
} catch (FileNotFoundException e) {
    System.out.println("No se encuentra ningún archivo en: " + ruta);}
```

En lugar de mostrar los datos de lectura en la salida estándar, también podemos almacenarlos en un array o un String.

Leer todo el texto de un archivo como una sola cadena

Desde Java 1.7 hay un conjunto de clases y métodos para el manejo de archivos. Dentro de este tema, nos limitaremos a aprender a leer un archivo de texto completo. Ten en cuenta que este método solo debe utilizarse para archivos de texto pequeños. Aquí, pequeño significa que su tamaño sea menor que la RAM disponible para la JVM.

En primer lugar, realizamos las siguientes importaciones:

```
import java.nio.file.Files;
import  java.nio.file.Paths;
```

La clase `Files` consta de métodos que funcionan con archivos; la clase `Paths` contiene un conjunto de métodos que devuelven un objeto específico para representar la ruta de acceso a un archivo.

El método siguiente devuelve todo el texto de un archivo especificado:

```
public static String leerFileComoString(String fileName) throws  IOException
{
 return new String(Files.readAllBytes(Paths.get(fileName)));
}
```

Vamos a intentar usar el método `leerFileComoString` para leer el código fuente del archivo `HolaMundo.java` e imprimirlo en la salida estándar.

```
import java.io.IOException;
import  java.nio.file.Files;
import  java.nio.file.Paths;

public class LeerFichero {

   public static void main(String[] args) {
    String ruta = "/home/username/Projects/helloworld/HolaMundo.java";
    try  {
        System.out.println(readFileAsString(ruta));
    } catch  (IOException e) {
 System.out.println("No se puede leer el archivo: " + e.getMessage());
    }
  }

  public static String leerFileComoString(String fileName)  throws IOException {
      return new  String(Files.readAllBytes(Paths.get(fileName))));
  }
}
```

Escribir archivos

Ahora que hemos aprendido a crear y administrar archivos, vamos a ver cómo escribir texto dentro de un archivo. Java proporciona diferentes maneras de hacerlo, y aquí consideraremos dos de las más sencillas: usar las clases `java.io.FileWriter` y `java.io.PrintWriter`.

La clase FileWriter

La clase `FileWriter` tiene varios constructores para escribir caracteres y cadenas en un archivo especificado:

- `FileWriter(String fileName);`
- `FileWriter(String fileName, boolean append);`
- `FileWriter(File file);`
- `FileWriter(File file, boolean append);`

El parámetro append de dos de los constructores sirve para indicar lo que hay que hacer si ya hay texto escrito previamente en el fichero: si se debe anexar (es decir, añadir el texto nuevo a continuación del texto ya existente), ponemos `true;` si se debe sobrescribir, ponemos `false`.

Todos estos constructores pueden producir una `IOException` por varias razones:

- si el "archivo" con ese nombre existe, pero es un directorio;
- si un archivo no existe y no se puede crear;
- si existe un archivo pero no se puede abrir.

Nosotros a veces omitiremos el mecanismo de control de excepciones para simplificar nuestros ejemplos.

Consideremos el siguiente código:

```
File f1 = new  File("/home/username/path/to/your/file.txt");
FileWriter escritor1 = new  FileWriter(f1); // sobrescribe el archivo
escritor1.write("Hola");
escritor1.write(" caracola”);
escritor1.close();
```

Si el archivo especificado no existe, se creará después de ejecutar este código. Si el archivo ya existe, este código sobreescribe los datos.

El archivo contendrá el texto **Hola caracola**.

Si deseamos anexar algunos datos nuevos, hay que especificar el segundo argumento como true.

```
File f2 = new File("/home/username/path/to/your/file.txt");
FileWriter escritor2 =  new FileWriter(f2,  true); // añade texto al archivo
escritor2.write("\nMe escribo a continuación.");
escritor2.close();
```

Este código anexa una nueva línea al archivo. Ejecútalo varias veces para ver qué sucede.

Cierre de un filewriter

Es importante cerrar un `FileWriter` después de usarlo para evitar fugas de recursos. Esto se puede hacer invocando el método close, como hemos hecho en los ejemplos anteriores.

Desde Java 7, la forma conveniente de cerrar un objeto `FileWriter` es mediante la instrucción `try-with-resources`.

```
File file = new File("/home/username/path/to/your/file.txt");
try (FileWriter writer = new  FileWriter(file)) {
        writer.write("Hello, World");
} catch  (IOException e) {
    System.out.printf("Se produce una excepción %s", e.getMessage());
}
```

El objeto writer se cerrará automáticamente.

La clase PrintWriter

La clase `PrintWriter` permite escribir datos con formato en un archivo.

```
File file = new File("/home/art/Documents/file.txt");
try  (PrintWriter printWriter =  new  PrintWriter(file)) {
       printWriter.print("Hello"); // imprime una cadena
       printWriter.println("Java"); // imprime una cadena y añade un salto de línea
       printWriter.println(123); // imprime un número
       printWriter.printf("Tienes %d %s",  400,  "monedas de oro");
       // imprime un String con formato
}  catch  (IOException e) {
    System.out.printf("Se produce una excepción %s", e.getMessage()); }
```

En este ejemplo se crea primero una instancia de `File` y, en segundo lugar, una de PrintWriter en la instrucción **try-with-resources** (y así cuando terminemos se cerrará automáticamente). Escribe `"Hello"y "Java"` en la misma línea, y luego `123` en una nueva línea. En este ejemplo también se llama al método `printf` avanzado, que puede dar formato a un texto mediante `%d` (para indicar que ahí va un dato de tipo double), `%s` (para indicar que ahí va un dato de tipo String) y así sucesivamente.

El resultado contiene:

HelloJava

123

Tienes 400 monedas de oro

La clase tiene varios constructores. Algunos de ellos son similares a los constructores de FileWriter:

- `PrintWriter(String fileName);`
- `PrintWriter(File file);`

Otros permiten pasar `FileWriter` como una clase que extiende la clase abstracta Writer:

- `PrintWriter(Writer writer);`

Jerarquías de archivos

Una aplicación puede almacenar datos en archivos en una unidad de disco. Para estructurar los datos, los archivos se pueden organizar en directorios. Un directorio puede incluir otros directorios (subdirectorios). Así es como se crea una jerarquía de archivos. Por ejemplo, consideremos la jerarquía del sistema de archivos en Linux: tiene el directorio raíz `/` que incluye todos los demás archivos y directorios, incluso si se almacenan en diferentes dispositivos físicos.

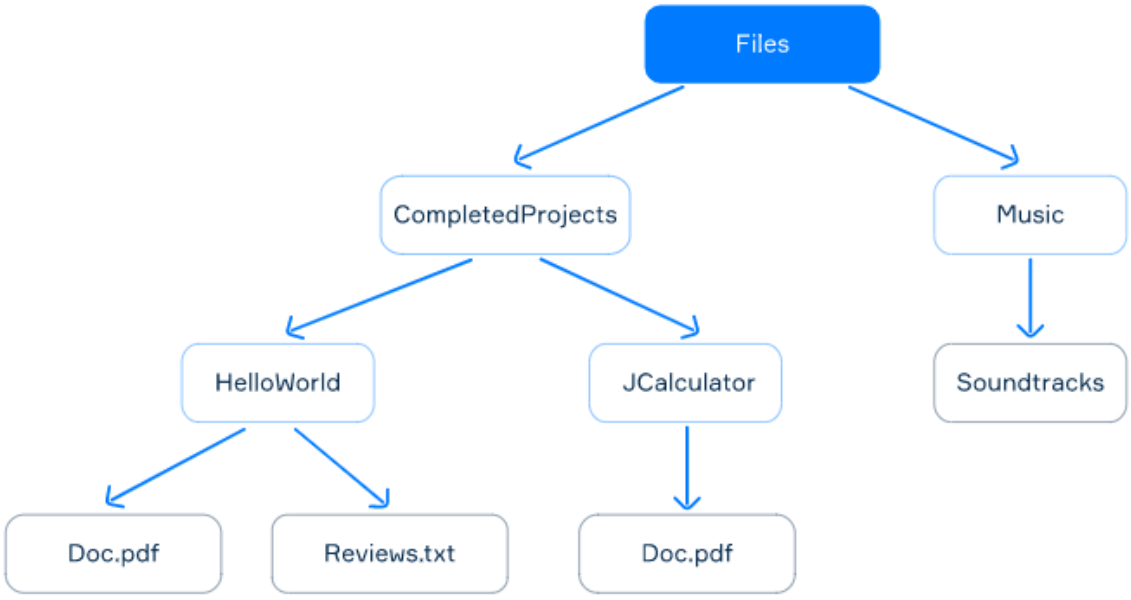
Métodos para iterar a través de jerarquías de archivos

Es posible recorrer una jerarquía de archivos en programas Java mediante la clase `java.io.File`. Consideremos cuatro métodos para hacerlo:

- `File` **`getParentFile()`** devuelve una instancia de `java.io.File` que representa el directorio padre de este archivo, o `null` si este archivo no tiene directorio padre (significaría que es la raíz);
- `String` **`getParent()`** devuelve una representación de cadena del directorio primario de este archivo, o `null` si este archivo no tiene un elemento primario;
- `File[]` **`listFiles()`** devuelve un array de archivos que se encuentran en este directorio, o `null` si esta instancia no es un directorio;
- `String[]` **`list()`** devuelve un array de cadenas de texto que nombran los archivos y directorios del directorio.

Un ejemplo de jerarquía

Consideremos una jerarquía de archivos con un directorio raíz denominado `Archivos (Files)`.



Puedes reproducir esta jerarquía en el sistema de archivos para intentar ejecutar los siguientes ejemplos.

Supongamos que tenemos una instancia de `java.io.File` denominada `completedProjsDirectory` que corresponde al directorio `CompletedProjects`. Ahora vamos a obtener sus dos subdirectorios, que contienen datos sobre proyectos.

```
File[] projects = completedProjsDirectory.listFiles(); // HelloWorld y JCalculator
```

El orden particular de los archivos en el array no está garantizado. Para encontrar el proyecto HelloWorld usaremos un método personalizado especial llamado `findFileByName` que devuelve el archivo encontrado, o `null` si no lo encuentra.

```
File helloWorldProject = findFileByName(projects, "HelloWorld");
```

Ahora vamos a obtener el archivo Reviews.txt mediante el mismo método.

```
File reviews = findFileByName(helloWorldProject.listFiles(), "Reviews.txt");
```

Ahora vamos a tratar de obtener una lista de archivos:

```
File[] files = reviews.listFiles(); // null
```

El array es `null` porque `reviews` no es un directorio y no puede incluir otros archivos o subdirectorios.

Ahora volvamos al directorio `CompletedProjects` para intentar obtener su elemento padre (la carpeta que lo contiene):

```
File filesDirectory = completedProjsDirectory.getParentFile();
```

En el código siguiente, obtenemos el directorio `Soundtracks:`

```
File music = findFileByName(filesDirectory.listFiles(), "Music");
File soundtracks = findFileByName(music.listFiles(), "Soundtracks");
```

Como el directorio Soundtracks está vacío, el método listFiles() devuelve un array vacío (que no null, lo cual sucedería en caso de que el objeto no fuera un directorio).

```
int length = soundtracks.listFiles().length; // 0, but not null
```