


Open · Opened 2 weeks ago by  **Rubén Montero**

Ficheros

Resumen

- Hablaremos del árbol de directorios del sistema operativo
- Veremos cómo leer un fichero línea a línea en Java
- Hablaremos del carácter de escapado (`\`)
- Crearemos una clase `HomeCinemaPreferences` que tendrá un método para leer el fichero `assets\cinemaPrefs.txt` e imprimir su contenido por pantalla

Descripción

Hasta el momento, todo nuestro trabajo se limita a clases Java y código que manipula variables en la [memoria RAM](#) del ordenador.

Escribamos programas que interaccionen con el mundo real. Y por mundo real, nos referimos al disco duro.

¿El disco duro (HDD)?

También llamado [almacenamiento secundario](#) o **no volátil**. En él se almacenan archivos mediante un árbol de directorios.

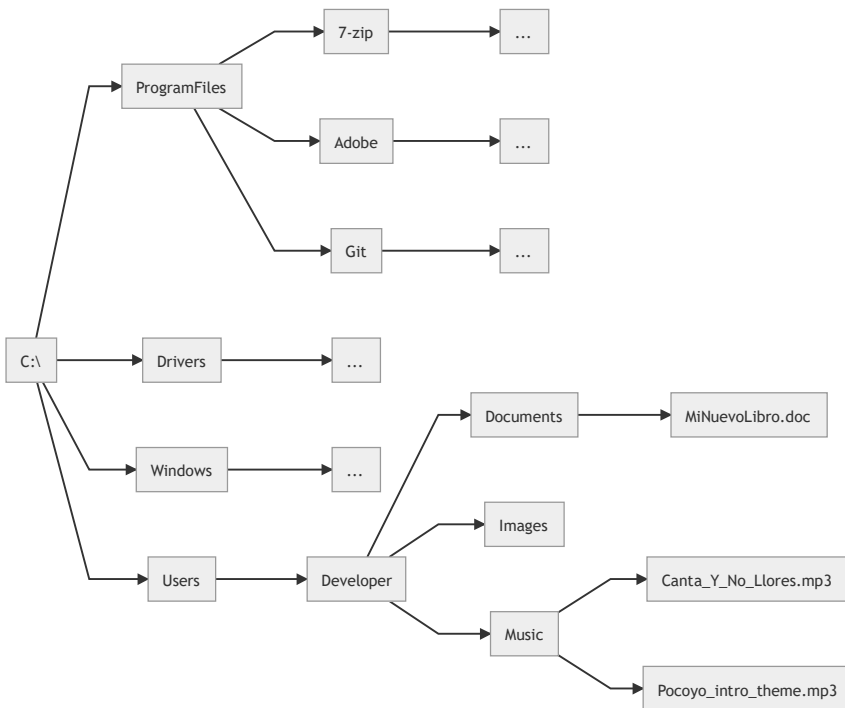
¿Árbol de directorios?

Los ficheros (MP3, vídeos, código Java,...) de nuestro ordenador no se hallan en el disco duro como juguetes en un baúl desordenado. Están muy bien **organizados** mediante carpetas, también llamadas directorios.

En sistemas Unix (Linux, MacOS), la **raíz** del árbol de directorios se señala con un *slash* (`/`).

En Windows, la **raíz** del árbol de directorios se señala con la letra asignada al disco duro (por ejemplo, `C:\` ó `D:\`). A partir de ahí, hay **carpetas** y **ficheros**. Cada **carpeta** puede contener otras **carpetas o ficheros**, de tal forma que queda organizado **jerárquicamente** como un **árbol**.

Por ejemplo:



Ruta

El concepto de **ruta** de un fichero ó directorio simboliza el *camino* a seguir para llegar a él, viajando por el árbol de directorios.

Absoluta

Una **ruta absoluta** comienza en el directorio **raíz**.

Por ejemplo:

```
C:\Users\Developer\Music\Canta_Y_No_Llores.mp3
```

Relativa

Una **ruta relativa** comienza en un directorio arbitrario.

Por ejemplo:

```
Desde el directorio HOME del usuario,  
que equivale a C:\Users\Developer  
  
Music\Canta_Y_No_Llores.mp3
```

The more you know...

Algunos detalles extra:

- En Windows, las rutas y nombres de fichero son `caseInsensitive`. O sea que `C:\Users` equivale a `C:\users`, mientras que en Unix, son `caseSensitive`.
- En Windows, se usa el *slash* invertido (`\`) para separar los componentes de una ruta. En Unix, se usa el *slash* convencional (`/`)

Volvamos a Java

En nuestro proyecto existe una carpeta `assets`. Como puedes ver, contiene el archivo `cinemaPrefs.txt`:

```
username=John Doe  
prefersDarkMode=true
```

Vamos a suponer que este archivo contiene las **preferencias** para una nueva aplicación que se está desarrollando: `Home Cinema`. Será una aplicación que permita ver el contenido de Netflix, HBO, Amazon Prime y Disney+ de forma gratuita. (No te asustes por problemas legales; no llegaremos a desarrollarla nunca).

De momento, vamos a comenzar escribiendo una nueva clase.

HomeCinemaPreferences.java

Estará destinada a leer este fichero de preferencias e interpretarlas.

Comencemos escribiendo la clase y un método constructor sin parámetros:

```
public class HomeCinemaPreferences {  
    public HomeCinemaPreferences() {  
  
    }  
}
```

Ahora vamos a **leer el fichero** e **imprimir su contenido** por pantalla. Hay [muchas formas](#). Nosotros usaremos `BufferedReader` porque facilita leer *línea a línea*.

El primer paso será definir en un `String` la **ruta** del fichero. Es una ruta **relativa**, comenzando en el directorio del proyecto.

```
public class HomeCinemaPreferences {  
    public HomeCinemaPreferences() {  
        String file = "assets\\cinemaPrefs.txt";  
  
    }  
}
```

¡Vaya! IntelliJ IDEA lo subraya en rojo e indica un error.

El carácter de escapepo (`\`)

Hay cierto caracteres no imprimibles que se escriben de forma especial dentro de un String:

- `\n` equivale a un *salto de línea*
- `\t` equivale a una *tabulación*
- ...

Como puedes ver, son letras normales (`n` , `t` ,...) precedidas del carácter especial `\` . Este **carácter de escapado** se usa para eso: *Dar un significado especial* a otros caracteres, y así imprimir caracteres **no imprimibles**.

El problema es que, si emplear un *slash* invertido (`\`) dentro del String... ¿Cómo lo hacemos?

Escapando (`\`) el carácter de escapado (`\\`)

Problema solucionado:

```
- String file = "assets\cinemaPrefs.txt";
+ String file = "assets\\cinemaPrefs.txt";
```

Continuemos

El siguiente paso es instanciar la clase `FileReader` . Así:

```
public class HomeCinemaPreferences {
    public HomeCinemaPreferences() {
        String file = "assets\\cinemaPrefs.txt";
        FileReader reader = new FileReader(file);

    }
}
```

¡**Vaya!** IntelliJ IDEA lo subraya en rojo de nuevo e indica *Unhandled exception*.

Aún no hemos hablado de **excepciones**, pero, brevemente, son una *forma* usada en el código de los programas (en Java y otros lenguajes) para señalar errores fuera de lo habitual.

Por ejemplo, ¿qué pasa si el *fichero* `cinemaPrefs.txt` no existe?

Saltará una excepción `FileNotFoundException` .

Nosotros, de momento, nos conformaremos con *controlar* dicha excepción empleando el `try/catch` que se genera automáticamente si pasamos el ratón encima del error y seleccionamos **More actions > Surround with try/catch**:

```
public class HomeCinemaPreferences {
    public HomeCinemaPreferences() {
        String file = "assets\\cinemaPrefs.txt";
        try {
            FileReader reader = new FileReader(file);

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

BufferedReader

Se crea a partir de un `FileReader` , que es la única razón por la cual *instanciamos* la variable del paso anterior.

Construimos un `BufferedReader` así:

```
public class HomeCinemaPreferences {
    public HomeCinemaPreferences() {
        String file = "assets\\cinemaPrefs.txt";
        try {
            FileReader reader = new FileReader(file);
            BufferedReader bufferedReader = new BufferedReader(reader);

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

```

    }
}
}

```

`.readLine()`

...es el método que debemos *invocar* sobre `bufferedReader` para leer *una nueva línea* del fichero.

Cada vez¹ que lo *invocamos*, la siguiente línea **se devuelve** en formato String.

Como queremos leer dos líneas, podemos hacer así:

```

public class HomeCinemaPreferences {
    public HomeCinemaPreferences() {
        String file = "assets\\cinemaPrefs.txt";
        try {
            FileReader reader = new FileReader(file);
            BufferedReader bufferedReader = new BufferedReader(reader);
            String firstLine = bufferedReader.readLine();
            String secondLine = bufferedReader.readLine();

        } catch (FileNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}

```

¡**Vaya!** IntelliJ IDEA lo subraya en rojo de nuevo e indica *Unhandled exception*. Ahora, controlar la excepción `FileNotFoundException` no es suficiente, porque pueden darse *otros errores*. Por ejemplo, si no disponemos de *privilegios* para leer el fichero.

Basta con sustituir `FileNotFoundException` por el error más general `IOException`:

```

public class HomeCinemaPreferences {
    public HomeCinemaPreferences() {
        String file = "assets\\cinemaPrefs.txt";
        try {
            FileReader reader = new FileReader(file);
            BufferedReader bufferedReader = new BufferedReader(reader);
            String firstLine = bufferedReader.readLine();
            String secondLine = bufferedReader.readLine();

-        } catch (FileNotFoundException e) {
+        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}

```

Lectura secuencial vs. aleatoria

Como puedes comprender, con `BufferedReader` leemos el fichero de forma **secuencial**. Aunque no trabajaremos con ella, existe otra clase `RandomAccessFile` que permite acceder a un fichero y [leer cualquier parte de su contenido de forma directa](#). Esto es, acceso **aleatorio**.

El acceso **secuencial** es más rápido pero menos flexible (estamos obligados a ir línea a línea).

`.readLine()` loop

Nuestra forma para leer el fichero es válida, porque esperamos **2** líneas. Pero es un poco mala.

¿Y si se añaden más líneas? Dejará de funcionar.

Es habitual emplear un bucle `do-while` para leer ficheros.

1. Se declara fuera del bucle la variable `String newLine`;
2. Dentro de `do { ... }` se asigna la nueva línea. **Si no hay más líneas**, `.readLine()` devolverá `null`
3. En la condición de salida del bucle `while(...)` especificamos que *el bucle debe repetirse mientras la línea (`newLine`) no sea `null`*

```

public class HomeCinemaPreferences {
    public HomeCinemaPreferences() {

```

```

        String file ="assets\\cinemaPrefs.txt";
        try {
            FileReader reader = new FileReader(file);
            BufferedReader bufferedReader = new BufferedReader(fileReader);
-            String firstLine = bufferedReader.readLine();
-            String secondLine = bufferedReader.readLine()
+            String newLine = null;
+            do {
+                newLine = bufferedReader.readLine();
+            } while (newLine != null);

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

```

¿Se entiende?

La prueba de fuego

Añadamos una sentencia dentro del bucle para imprimir por consola cada línea que leemos:

```

public class HomeCinemaPreferences {
    public HomeCinemaPreferences() {
        String file ="assets\\cinemaPrefs.txt";
        try {
            FileReader reader = new FileReader(file);
            BufferedReader bufferedReader = new BufferedReader(fileReader);
            String newLine = null;
            do {
                newLine = bufferedReader.readLine();
+                System.out.println(newLine);
            } while (newLine != null);
+            // Un detalle: Cerrar el fichero tras su uso
+            fileReader.close();

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

```

Si al instanciar esta clase desde `Main.java` :

```
HomeCinemaPreferences prefs = new HomeCinemaPreferences();
```

...vemos el siguiente contenido en la consola:

```

username=John Doe
prefersDarkMode=true
null

```


¡Funciona correctamente!

¡Enhorabuena! Has leído con éxito tu primer fichero de preferencias en Java.

Por último

Verifica que el *test* funciona correctamente.

Haz `commit` y `push` para subir los cambios al repositorio.

1. No hace falta que llevemos la cuenta de *qué* línea estamos leyendo. `BufferedReader` lo hace automáticamente. ¿Sabes qué significa eso?
¡Efectivamente! Almacena **internamente** el estado (en un atributo) de **qué** línea está leyendo 



Rubén Montero @ruben.montero changed milestone to [%Sprint 1](#) 2 weeks ago



Ania Blanco @ania.blanco mentioned in commit [cd8d5292](#) 2 weeks ago