


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

Leyendo preferencias en JSON

Resumen

- Veremos cómo se puede recuperar la información de un archivo JSON
- Veremos qué es un `Enum` en Java
- Crearemos un nuevo constructor `HomeCinemaPreferences` que reciba un parámetro `Enum` :
 - Permitirá inicializar las preferencias desde `cinemaPrefs.txt`, `cinemaPrefs.xml` ó `cinemaPrefs.json` según su valor
- *Deprecaremos* los constructores anteriores

Descripción

El proceso de leer un fichero JSON es bastante sencillo. Basta con emplear un `FileReader` como el que ya sabemos *instanciar*:

```
FileReader reader = new FileReader("assets\\cinemaPrefs.json");
```

...y con ese objeto, crear un `JSONTokenizer`, que es la clase diseñada para *parsear* un fichero JSON en la librería `org.json:json20220320` :

```
FileReader reader = new FileReader("assets\\cinemaPrefs.json");
JSONTokenizer tokenizer = new JSONTokenizer(reader);
```

En Java, se permite escribir todo **en una sola línea** así:

```
JSONTokenizer tokenizer = new JSONTokenizer(new FileReader("assets\\cinemaPrefs.json"));
```

Vale, vamos allá

Crearemos un nuevo método *privado* en `HomeCinemaPreferences.java` que sirva para *inicializar* las preferencias desde un JSON.

Comenzará así:

```
private void initializeFromJSON() {
    try {
        JSONTokenizer tokenizer = new JSONTokenizer(new FileReader("assets\\cinemaPrefs.json"));

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

...y luego es necesario *instanciar* un `JSONObject`.

En la tarea anterior *instanciábamos* un `JSONObject` **vacío**, sin parámetros en el constructor.

Ahora, le pasamos el `tokenizer` para que **lea** el fichero asociado:

```
private void initializeFromJSON() {
    try {
        JSONTokenizer tokenizer = new JSONTokenizer(new FileReader("assets\\cinemaPrefs.json"));
+        JSONObject jsonObject = new JSONObject(tokenizer);

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

Los siguientes métodos se emplean para recuperar información de un objeto JSON (Si tecleas `jsonObject.` y esperas a que IntelliJ IDEA *autocomplete*, los verás):

- `.getString`
- `.getBoolean`
- `.getFloat`
- `.getInt`
- `.getJSONObject`
- `.getJSONArray`

Por lo tanto, debemos saber de **qué** tipo es el valor que estamos recuperando. En nuestro caso, `username` es un `String` y `prefersDarkMode` es un `boolean`, así que haríamos así:

```
private void initializeFromJSON() {
    try {
        FileReader reader = new FileReader("assets\\cinemaPrefs.json");
        JSONTokener tokenener = new JSONTokener(reader);
        JSONObject jsonObject = new JSONObject(tokenener);
+        String fileUsername = jsonObject.getString("username");
+        boolean fileDarkMode = jsonObject.getBoolean("prefersDarkMode");

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

¿Para qué queríamos estos valores?

¡Ah, sí! Estábamos **inicializando** las preferencias.

```
private void initializeFromJSON() {
    try {
        FileReader reader = new FileReader("assets\\cinemaPrefs.json");
        JSONTokener tokenener = new JSONTokener(reader);
        JSONObject jsonObject = new JSONObject(tokenener);
        String fileUsername = jsonObject.getString("username");
        boolean fileDarkMode = jsonObject.getBoolean("prefersDarkMode");
+        this.username = fileUsername;
+        this.darkModePreferred = fileDarkMode;
    } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
    }
}
```

Un nuevo constructor

Nuestro método `initializeFromJSON` funciona estupendamente. Estaría bien invocarlo desde el método constructor, pero **ya** hay dos métodos constructores:

```
public HomeCinemaPreferences(boolean readXML) {
    if (readXML) {
        initializeFromXML();
    } else {
        initializeFromTXT();
    }
}

@Deprecated
public HomeCinemaPreferences() {
    initializeFromTXT();
}
```

`boolean readXML` sólo admite **dos** valores, que significan *leer desde un `txt` ó leer desde un `xml`*. ¡Ahora hay un **tercer** caso!

Enum

Es un tipo de variable especial presente en muchos lenguajes de programación. Admite una serie de **valores prefijados** que ya se conocen en tiempo de **compilación**.

Es como un `booleano`, pero en vez de **2** valores, admite los que nosotros decidamos.

¿Y cómo los decidimos?

Haz **click derecho** en la carpeta `src/` y selecciona **New > Java Class**.

Selecciona `Enum`. Dale un nombre `HomeCinemaPreferencesMode`

Aparecerá un nuevo fichero de código fuente:

```
public enum HomeCinemaPreferencesMode {
}
```

Entre las llaves (`{ }`) añadiremos, separados por comas, los posibles **valores** que podrá tener este `Enum`. Suelen ir en mayúsculas:

```
public enum HomeCinemaPreferencesMode {
    MODE_TXT,
    MODE_XML,
    MODE_JSON
}
```

¡ `Enum` **listo!**

Resumen de `Enum`

- Es un tipo de dato definido por el desarrollador, como una clase
- La diferencia es que no tiene atributos
- Ni métodos
- ¡Consiste sólo en un *conjunto* de posibles valores!

¿Y cómo se usa?

Añade un nuevo método constructor a `HomeCinemaPreferences.java`:

```
public HomeCinemaPreferences(HomeCinemaPreferencesMode mode) {
}
```

Como ves, debe recibir un **parámetro** de **tipo** `HomeCinemaPreferencesMode`.

Ahora, en función del valor, invocaremos el método apropiado:

```
public HomeCinemaPreferences(HomeCinemaPreferencesMode mode) {
    if (mode == HomeCinemaPreferencesMode.MODE_TXT) {
        initializeFromTXT();
    } else if (mode == HomeCinemaPreferencesMode.MODE_XML) {
        initializeFromXML();
    } else if (mode == HomeCinemaPreferencesMode.MODE_JSON) {
        initializeFromJSON();
    }
}
```

Como ves, es como un `booleano` pero con **3** posibles valores.

Si lo prefieres, puede usar la sentencia `switch` **equivalente**¹ que es mucho más típica:

```
public HomeCinemaPreferences(HomeCinemaPreferencesMode mode) {
    switch (mode) {
        case MODE_TXT: initializeFromTXT(); break;
        case MODE_XML: initializeFromXML(); break;
        case MODE_JSON: initializeFromJSON(); break;
    }
}
```

Adaptarse o morir

Ahora podemos *deprecar* el método constructor de las tareas anteriores:

```
+  @Deprecated
public HomeCinemaPreferences(boolean readXML) {
    if (readXML) {
        initializeFromXML();
    } else {
        initializeFromTXT();
    }
}
```

...y comprobar el funcionamiento del nuevo constructor desde `Main.java`.


Por ejemplo:

```
// Leemos el XML, cambiamos el nombre y lo escribimos a JSON
HomeCinemaPreferences prefs1 =
    new HomeCinemaPreferences(HomeCinemaPreferencesMode.MODE_XML);
prefs1.setUsername("John Carter");
prefs1.saveAsJSON();
// Leemos las preferencias del JSON para verificar que son correctas
HomeCinemaPreferences prefs2 =
    new HomeCinemaPreferences(HomeCinemaPreferencesMode.MODE_JSON);
System.out.println("Username en .json es " + prefs2.getUsername());
```

Por último

Verifica que el test funciona correctamente.

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

1. ¿Te has preguntado qué es ese `break`? 



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



Ania Blanco @ania.blanco mentioned in commit [d13a2049](#) 1 week ago