


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

Leyendo preferencias en XML

Resumen

- Veremos cómo se puede recuperar la información de un archivo XML
- Crearemos un nuevo constructor `HomeCinemaPreferences` que reciba un parámetro `boolean` :
 - Si es `false`, el método funcionará como hasta ahora, leyendo `cinemaPrefs.txt`
 - Si es `true`, leerá `cinemaPrefs.xml`
- Hablaremos de *deprecar* (marcar como obsoleto) código

Descripción

Ya sabemos cómo guardar información en XML. Pues bien, leerla es un proceso muy similar.

El primer paso será crear un nuevo `Document`, pero esta vez, en lugar de hacerlo *desde 0*:

```
Document xmlDocument = builder.newDocument();
```

...lo haremos *a partir* de un fichero. Así:

```
Document xmlDocument = builder.parse("assets\\cinemaPrefs.xml");
```

Vale, vamos allá

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

Comenzará así:

```
private void initializeFromXML() {  
    try {  
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();  
        DocumentBuilder builder = factory.newDocumentBuilder();  
        Document xmlDocument = builder.parse("assets\\cinemaPrefs.xml");  
  
        // ...  
    } catch (IOException | SAXException | ParserConfigurationException e) {  
        throw new RuntimeException(e);  
    }  
}
```

A continuación, emplearemos métodos de la librería `javax.xml` que empiezan por `get` para *recuperar* nodos del documento.

El primero será para *recuperar* el nodo raíz:

Preferences

```
Element rootElement = xmlDocument.getDocumentElement();
```

...y luego recuperar cada uno de sus *hijos*:

Username

PrefersDarkMode

```
NodeList childNodes = rootElement.getChildNodes();
```

Un método para *parsear* nodos

¿Recuerdas el método `parseLine` que escribimos para *parsear* cada una de las líneas del fichero `cinemaPrefs.txt` ?

Vamos a escribir un método análogo, pero en vez de trabajar con una línea de un fichero, en formato `String`, en esta ocasión trabajará con una

variable de tipo `Node` :

```
private void parseNode(Node node) {

}
```

Hay que invocarlo

Igual que cuando leíamos el fichero línea a línea, nosotros debemos ir nodo a nodo sobre nuestro `childNodes` esta vez.

El proceso es sencillo, con un bucle `for` así:

```
for (int i = 0; i < childNodes.getLength(); i++) {
    Node child = childNodes.item(i);
    parseNode(child);
}
```

Completemos `parseNode`

Volviendo a nuestro método *privado*, trabajemos con cada uno de los **nodos**.

Basta con emplear los métodos:

- `getNodeName`
- `getTextContent`

Por ejemplo, así:

```
private void parseNode(Node node) {
    if (node.getNodeName().equals("Username")) {
        this.username = node.getTextContent();
    }
    // ...
}
```

¿Encaja?

Para `prefersDarkMode`, podríamos escribir *algo como esto*:

```
private void parseNode(Node node) {
    if (node.getNodeName().equals("Username")) {
        this.username = node.getTextContent();
    }
+   if (node.getNodeName().equals("PrefersDarkMode")) {
+       this.darkModePreferred = node.getTextContent();
+   }
}
```

Pero el compilador **subraya en rojo** un problema, ya que intentamos asignar un `String` a un `boolean`.

Ya nos hemos enfrentado a un problema similar con anterioridad.

Consigue arreglarlo.

¿Y después?

Unos **sencillos pasos** para terminar la tarea.

Vamos a **escribir** un **nuevo método privado** `initializeFromTXT` :

```
private void initializeFromTXT() {

}
```

Y vamos a **cortar** (CTRL+X) y **pegar** (CTRL+V) el contenido actual del método constructor, que se encarga de *leer un archivo* `.txt` :

```
private void initializeFromTXT() {
+   String file ="assets\\cinemaPrefs.txt";
+   try {
```

```
+         FileReader fileReader = new FileReader(file);
+         BufferedReader bufferedReader = new BufferedReader(fileReader);
+         // ...
+
+     }
```

Para que funcione igual, en el método **constructor** que hemos dejado vacío, debemos **invocar** el método:

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

Muy bonito... ¿Para qué?

Ahora podemos **añadir** un segundo **método constructor** que reciba un parámetro **boolean** :

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

...sin **romper** el constructor anterior.

¿Romper?

Nos referimos a **borrarlo**.

Imagina que un compañero de trabajo emplea nuestro constructor usando `new HomeCinemaPreferences()`; . Si lo borramos, ¡**rompemos** su código!

Aún así...

Tenemos **código duplicado** en nuestra clase `HomeCinemaPreferences.java` . Eso **no** es deseable.

El código duplicado es el viejo constructor:

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

...ya que **invocar** el **nuevo constructor** y pasar **false** , es equivalente:

```
// Ahora mismo, esto...
HomeCinemaPreferences prefs = new HomeCinemaPreferences();
// ...y esto:
HomeCinemaPreferences prefs = new HomeCinemaPreferences(false);
// Es LO MISMO
```

Marcar como obsoleto

En situaciones como esta, es bueno marcar el viejo código como obsoleto o *deprecado*. Esto se consigue con la **anotación** `@Deprecated` .

Así:

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

Sirve para que *futuros* desarrolladores tengan en cuenta que **no** se debería usar ese código, ya que está ahí por razones *históricas* o *legacy*.

Por último

Verifica que el **test** funciona correctamente.

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



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



Ania Blanco @ania.blanco mentioned in commit [863e451f](#) 1 week ago