MEDAC
Instituto Oficial de Formación Profesional

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMAS

Trabajo con ficheros XML

03

ÍNDICE

OBJETIVOS

Afianzar los distintos tipos de acceso DOM y SAX

Aprender cómo realizar el tratamiento de excepciones con ficheros XML.

Realizar baterías de test en relación a estas clases.

/ 1. Introducción y contextualización práctica

Durante el siguiente tema profundizaremos sobre el análisis de ficheros XML. Veremos diferentes formas de capturar la información desde ficheros totalmente esquematizados a transformarlos en objetos y clases Java o DTOs (Data transfer Object), objetos cuyo propósito se centra en capturar la información y los atributos de una entidad proveniente de un servidor o un servicio externo.

Se nombrarán los distintos métodos de acceso a datos DOM y SAX, se realizará un breve resumen sobre la conversión de ficheros XML y el procesamiento de los mismo con XPATH.

En el siguiente tema, sobre todo, profundizaremos en el tratamiento de excepciones que rodean las operaciones citadas anteriormente. Analizaremos los tipos de excepciones que pueden ser capturadas en función a las operaciones estemos realizando.

Aplicaremos todos estos conceptos adentrándonos también en el mundo del Testing y realizaremos algunas pruebas de tipo unitario relacionados con los temas anteriores.

Fig. 1 Trabajo con ficheros XML

Escucha el siguiente audio en el que planteamos el caso práctico que iremos resolviendo a lo largo de esta unidad.

Audio intro. Manejo de excepciones.
https://bit.ly/2ZrnkLX

/ 2. Acceso a datos con DOM Y SAX

Tanto DOM como SAX son estándares, herramientas que nos ofrecen la posibilidad de lectura de ficheros XML. Estas herramientas básicamente se dedican a verificar si sintácticamente son ficheros válidos. Son los llamados "parsers" o analizadores. A continuación, vamos a ir nombrando algunas de las características y diferencias entre DOM y SAX:

- La ventaja que tenemos con el sistema DOM es que una vez introducimos el fichero HTML o XML, obtenemos el árbol ya formado de los nodos y demás objetos, preparado para trabajar. Sin embargo, es algo más lento y menos versátil. Por el contrario, SAX es más rápida, pero menos potente que DOM. Con SAX necesitamos introducir líneas de programación para obtener partes determinadas de los ficheros. Nos ofrece mayor nivel de funcionalidad y versatilidad.

- La operativa o la forma de funcionar de DOM y SAX es muy diferente: mientras DOM carga el fichero completo, SAX tiene en memoria sólo la parte del nodo o el evento actual. Por lo tanto, es cierto que en DOM tenemos todo el árbol del fichero disponible, pero ocupa mucha más memoria que SAX.

- Resumiendo, es aconsejable usar SAX para recorrer secuencialmente los elementos del fichero XML y realizar ciertas operaciones, mientras que se debería usar DOM cuando tengamos el objetivo claro sobre el que queremos trabajar, a partir de un árbol creado en memoria.

Características de ambos sistemas:

SAX	DOM
Basado en eventos	Carga el fichero en memoria
Va analizando nodo por nodo	Búsqueda de tags hacia delante y hacia detrás
En principio sin muchas restricciones de memoria ya que no carga la totalidad del fichero	Estructuras de árbol
Rapidez en tiempo de ejecución	Más lento en tiempo de ejecución
Es de sólo lectura	Se pueden insertar o eliminar nodos.
Es de sólo lectura	DataInputStream, DataOutputStream, PrintStream

Tabla 1. Características DOM y SAX

/ 3. Conversión de ficheros XML

Actualmente existen muchas librerías que se utilizan como parsers de ficheros XML. En este caso, veremos en rasgos generales qué podemos encontrar en la paquetería: "javax.xml.parsers".

A continuación, veremos un ejemplo de un "parser" de tipo DOM, que como hemos comentado anteriormente, la estructura se cargará en memoria y tendremos disponible el fichero completo.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

factory.setValidating(true);

factory.setIgnoringElementContentWhitespace(true);

DocumentBuilder builder = factory.newDocumentBuilder();

File file = new File("C:\\temp\\pruebas\\prueba.xml");

Document doc = builder.parse(file);
```

Código1 DOM parser

Para comenzar se realizaría la instanciación de la clase **DocumentBuilderFactory.**

Posteriormente, se pondría el atributo de validación como "true" para asegurarnos que el fichero que se cargue esté bien validado. Se hace un set también, al atributo de "ignorar los elementos que contengan espacios en blanco", a "true".

Después de esto se crearía un objeto DocumentBuilder por medio de la factoría creada previamente.

A continuación, se instanciaría un nuevo fichero indicando la ruta del fichero a analizar.

Y por último, se cargaría el fichero completo con el método builder.parse(file), y se asignaría a un objeto de tipo Document. De esta forma quedará almacenado, y podremos realizar diferentes acciones con dicho objeto en las líneas siguientes.

A continuación, al igual que hemos visto cómo instanciar un parser de tipo DOM, ahora mostraremos cómo instanciar y trabajar con uno de tipo SAX. Sobre este modo de acceso, simplemente destacar que se instanciará, en el método parse, un Handler que será el responsable de ejecutar ciertas operaciones como iniciar elementos, operaciones con nodos, inicio/fin de documento, etc.

Es en la definición del Handler es donde debemos indicar las operaciones que realice nuestro analizador de código.

```
SAXParserFactory factory = SAXParserFactory.newInstance();

factory.setValidating(true);

SAXParser saxParser = factory.newSAXParser();

File file = new File("test.xml");

saxParser.parse(file, new DefaultHandler());
```

Código 2 SAX parser

Audio 1. Elige entre DOM y SAX
https://bit.ly/2NKSNDp

/ 4. Procesamiento de XML: XPath

Hablamos de XPATH como una forma de búsqueda de información a través de un documento XML. De hecho, XPATH es una recomendación oficial del consorcio del World Wide Web (W3C).

Se utiliza para recorrer elementos y atributos de un documento XML, y proporciona varios tipos de expresiones que pueden usarse para consultar información relevante.

A continuación, veremos algunas de las características principales de XPATH:

- **Definición de estructuras:** define las distintas partes de un documento XML como un elemento, atributos, textos, instrucciones de procesamiento, comentarios y nodos del documento.
- **Expresiones:** XPATH posee expresiones potentes para el manejo de ficheros, como por ejemplo seleccionar nodos o listas de nodos en ficheros XML.
- **Funciones estándar:** XPATH nos brinda una librería muy completa de funcionalidades estándar de manipulación de Strings, valores numéricos, fechas, comparaciones, secuencias, valores booleanos etc.

Una vez vistas las características de nuestra librería XPATH, a continuación indicaremos algunos puntos que nos servirán como guion cuando queramos hacer uso de esta librería.

Definiremos, por tanto, una **serie de pasos al usar la librería XPATH**:

- Importaremos los paquetes relacionados con dicha librería
- Crearemos un objeto de la clase DocumentBuilder
- Cargaremos un fichero o un flujo de datos.
- Crearemos un objeto XPATH y una expresión.
- Realizaremos una compilación de dicha expresión con el método Xpath.compile() y obtendremos una lista de los nodos evaluando la expresión previamente compilada usando Xpath.evaluate()
- Realizaremos una iteración por lo general de la lista de nodos.
- Examinaremos los atributos
- Examinaremos los sub elementos.

La instanciación del objeto Xpath la realizaremos de la siguiente forma:

```
XPath xPath =  XPathFactory.newInstance().newXPath();
```

Código 3. XPath instancia

Vídeo 1. " XPATH"

https://bit.ly/2YS7mv3

/ 5. Caso práctico 1: "DOM o SAX"

Planteamiento: el analista de tu equipo de desarrollo informa de que hay que usar un parser en una parte de una aplicación web para una tienda de muebles sobre la que estáis trabajando.

El objetivo es analizar los clientes de dicha tienda y disponéis de un fichero XML previo que será cargado.

Fig. 2 SAX parser.

Es importante tener en cuenta que:

- El fichero debe ser ligero en memoria.
- No se requiere la edición del fichero, solo su lectura.

Nudo: Tendríamos distintas opciones a la hora de poder elegir un analizador de código XML. Los más conocidos son acceso DOM y SAX. ¿Cuál usarías en la situación planteada?

Desenlace: dados los requerimientos nos decantaremos por el acceso al fichero por medio de las librerías SAX, ya que tienen un gasto notablemente inferior de memoria en el sistema. Es una herramienta ideal si no se necesitan manipular dichos ficheros en memoria, y es perfecta para recorrer de forma secuencial.

/ 6. Excepciones

En este punto del tema queremos mostrar una imagen global de las excepciones que rodean algunas de las implementaciones de parsers XML que hemos visto previamente, para ello, realizaremos una breve introducción a las mismas a modo genérico.

6.1. Excepciones en Java y tipos.

En primer lugar ¿Qué entendemos como excepción?

Una excepción no es más que un evento que ocurre durante la ejecución de un programa, y que interrumpe el flujo del mismo por algún motivo. Éstas pueden ocurrir por muchos y diferentes motivos: el usuario ha introducido datos erróneos, un fichero no es encontrado, cortes de conexión, problemas de acceso a medios, etc. Algunas de estas excepciones son ocasionadas por el usuario, otras por el desarrollador, y otras por el estado de los recursos físicos, es por ello, que las excepciones se dividen en 3 categorías:

- **Excepciones con chequeo (chequed exceptions):** son excepciones que son notificadas por el compilador en tiempo de compilación, no pueden ser ignoradas, y fuerzan al programador a manejarlas.
- **Excepciones sin chequeo (unchequed exceptions):** estas excepciones se originan en tiempo de ejecución, son llamadas también RuntimeExceptions. Se incluyen aquí también, errores de programación o cuando se ha usado mal una API de código, por ejemplo. Comentar también que este tipo de excepciones son ignoradas en tiempo de compilación.
- **Errores:** no son del todo excepciones, escapan del control del usuario o del propio desarrollador. Los errores generalmente se ignoran en el código porque rara vez se puede hacer algo al respecto. Un ejemplo de error es Stack overflow si hay un desbordamiento de pila, y difícilmente vamos a poder hacer algo para solucionar este problema. Este tipo de errores son ignorados en tiempo de compilación también.

6.2. Excepciones asociadas a clases XML I

En este apartado veremos algunas de las excepciones utilizadas en las clases anteriores, pero antes vamos a visualizar algunos de los métodos más importantes que debemos conocer si queremos hacer manejo de excepciones:

- **getMessage():** devuelve un mensaje detallado sobre la excepción que se acaba de lanzar. El mensaje es instanciado en el constructor de la clase Throwable.
- **getCause():** devuelve la causa representada en un objeto Throwable.
- **toString():** devuelve el nombre de la clase y se le concatena el resultado de getMessage().
- **printStackTrace():** imprime el resultado del método toString() junto con el error de sistema que devuelve la pila.
- **getStackTrace():** devuelve un array con cada uno de los elementos de la pila. El elemento 0 del array representa el elemento más alto de la pila.
- **fillInstackTrace():** rellena la pila del objeto Throwable con la pila actual. Le añade cualquier información previa en el seguimiento de la misma.

Para poder detectar y tratar una excepción necesitamos incluir un bloque de código try/catch. Este bloque se coloca alrededor del código que pudiera generar una excepción. El código incluido dentro de un bloque try/catch se conoce como código protegido.

Sintaxis de la sentencia:

```
try {
  // Código protegido
} catch (ExceptionName e) {
  // Operaciones tras capturar excepción
}
```

Código 4. Bloque try/catch

El código que es propenso a excepciones se coloca a continuación de la sentencia **try**. Cuando se lanza una excepción, es capturada por el bloque **catch** asociado a ella. Cada bloque **try** puede ir seguido de un bloque **catch** o de un bloque **finally.**

Un bloque catch como podemos observar implicará declarar un tipo de la excepción que queramos capturar

Un bloque finally lo encontraremos justo después del bloque **try** o después del catch. Es una parte de código que se ejecutará siempre, independientemente si pasa por éstos. Este tipo de bloques suelen usarse para labores de limpieza o liberación de recursos de memoria, por ejemplo.

6.3. Excepciones asociadas a clases XML II

Una vez desarrollado un aplicativo, habrá ciertas líneas que necesiten ser rodeadas de nuestras sentencias try/catch y otras no. Para aquellas sentencias que sea obligatorio, nuestro entorno de desarrollo nos lo notificará. Por ejemplo:

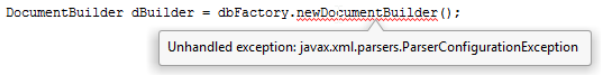

Fig. 3 Excepción no controlada.

Tal y como observamos, el IDE nos subraya una operación, y si situamos el cursor encima de ésta, nos avisa que hay una excepción que no está siendo controlada, además nos informa de que tipo es. Si nos fijamos en la parte izquierda de esa misma línea, en el margen de nuestro editor de texto observaremos una indicación. Si hacemos clic en ella tendremos varias opciones.

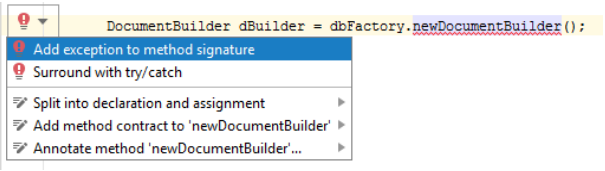

Fig. 4 Añadir excepciones

Concretamente, una vez hacemos clic en la pequeña bombilla roja, tendremos:

- **Añadir la excepción a la definición del método:** con esta opción lo que estaremos haciendo será lanzar la excepción a un nivel superior. De esta forma se irá lanzando la excepción de un nivel a otro hasta que alguno de ellos decida tratarla.

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

Código 5. Lanzamiento excepción

- **Rodear con sentencia try/catch:** tal y como hemos visto previamente, rodearíamos el código con la estructura básica try/catch. De esta forma el código quedará protegido y si se lanza dicha excepción, será capturada y tratada.

```
try {
  dBuilder = dbFactory.newDocumentBuilder();
} catch (ParserConfigurationException e) {
  e.printStackTrace();
}
```

Código 6. uso de try/catch

6.4. Excepciones asociadas a clases XML III

Ya estudiado el procedimiento de cómo capturar una excepción desde un bloque de código try/catch, ahora veremos algunos otros ejemplos de diferentes excepciones, que lanzan las operaciones que ejecutamos con nuestros parsers:

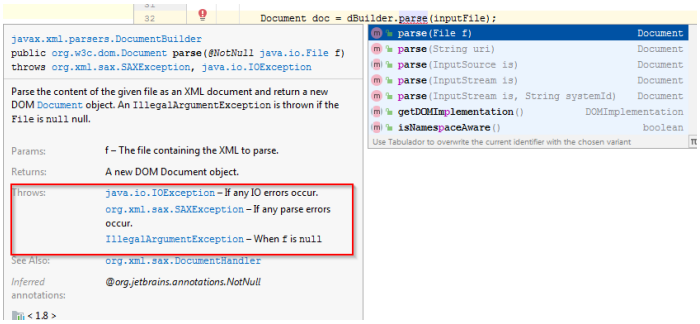

Fig. 5 Documentación excepciones

Como podemos observar en la imagen superior, disponemos de una línea de código en la que: un objeto de la clase DocumentBuilder está ejecutando el método parse() y le está pasando como parámetro un fichero. Bien, el IDE nos indica que hay un error, algo falta, una excepción que no está siendo manejada.

Todos los IDEs tienen una combinación de teclas para mostrar la documentación de Java sobre el método en el que se está, en este caso, parse(). En esta documentación, en el apartado de Throws, tal y como se marca en la imagen, se puede ver qué excepciones podría lanzar el método que se está ejecutando. De esta forma, es una muy buena práctica pensar y cubrir las distintas opciones con bloques catch, capturando cada una de estos posibles lanzamientos de excepciones.

Por último, comentar, que cuando estemos realizando bastantes operaciones del estilo, en las cuales, se vean envueltas un número considerable de excepciones, es una buena práctica poner un bloque **try,** añadir las líneas necesarias, y a continuación los bloques catch anidados unos con otros, para así evitar tener que ir escribiendo contínuamente bloques **try/catch**. Ejemplo:

```
try {
//Distintas operaciones parse

} catch (ParserConfigurationException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
} catch (XPathExpressionException e) {
  e.printStackTrace();
} catch (SAXException e) {
  e.printStackTrace();
}
```

Código 6: LineNumberReader

Como podemos observar englobamos todos los bloques catch de una serie de operaciones de análisis o “parsing”, teniéndolas mejor distribuidas y manejadas de esta forma.

/ 7. Pruebas y documentación. JUnit

Un test es una pieza de Software que ejecuta otra pieza de Software. Valida si los resultados de un código están en el estado que se espera, o ejecuta la secuencia esperada de operaciones o eventos. Ayudan al programador a verificar que un fragmento de código es correcto.

Junit es un framework que usa anotaciones para identificar diferentes tests. Una prueba unitaria Junit es realmente un método que está en el interior de una clase llamada Test class. Para definir que un método forma parte de un test se tendrá que añadir la anotación @Test sobre la cabecera del método.

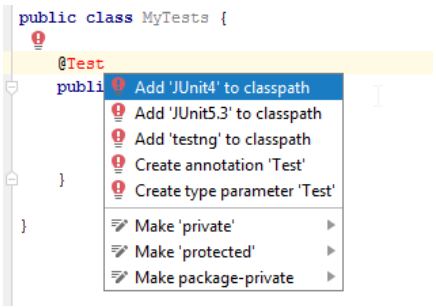

Fig. 6 Añadiendo Junit

Tal y como podemos observar en la imagen anterior, si queremos disponer de las librerías necesarias para poder realizar nuestras pruebas unitarias, una de las opciones es la que se muestra. Podríamos escribir la anotación "@ Test" ya que es una palabra clave, y nuestro IDE entenderá que queremos introducir librerías de Junit por ser el framework más extendido de Unit Testing.

Nos dará a elegir entre las versiones estables del momento y elegiremos la que más nos convenga.

Otra opción en otro tipo de proyectos (proyectos web por ejemplo) sería añadir una nueva dependencia maven con la librería correspondiente Junit y su versión.

Existen algunas anotaciones y algunos métodos muy interesantes que debemos tener en cuenta para la escritura de test unitarios, como por ejemplo:

- **Anotación @Before:** al inicio de la clase se definirá un método. Su utilidad será instanciar la mayor parte de las variables que vamos a necesitar para los test. Siempre que ejecutemos un test unitario, antes, se ejecutará el código de este método con anotación @Before.

- **Anotación @After:** Con esta anotación, se definirá un método cuyo código se ejecutará, siempre después de finalizar cualquier test dentro de nuestra clase.

Video 2. "Estructura JUnit"

https://bit.ly/30HPjsv

/ 8. Caso práctico 2: "Uso before"

Planteamiento: Miguel pertenece a un equipo de desarrollo de un proyecto de tal envergadura que requiere de que el equipo se segmente para crear equipos de trabajo más pequeños dentro del equipo principal, cuyas funciones quedarán mejor delimitadas.

En una de sus comprobaciones, necesita que, en cada test unitario, se realicen pruebas matemáticas con 2 números. Para ello, al ejecutar cada test, Miguel deberá cargar automáticamente en memoria una lista con 2 números antes de empezar el código de cada prueba unitaria.

Fig. 7 Los test son imprescindibles en desarrollo.

Nudo: Miguel ha estado investigando y quiere poner en práctica la anotación @Before a un método. ¿Cómo lo haría?

Desenlace: tal y como hemos visto previamente, la anotación @Before realizará un set up de los datos o una serie de configuraciones básicas que necesitemos para abordar los diferentes test. Básicamente, Miguel tendrá que crear un método del siguiente estilo:

```
@Before

public void cargaNumeros() {

   numeros.add(2);

   numeros.add(5);

                                        }
```

Código 8. Uso de @Before.

De esta forma, antes de comenzar la ejecución de cualquier test de la clase, tendrá cargada la lista «numeros» con datos previamente cargados en el método con anotación @Before.

/ 9. Resumen y resolución del caso práctico de la unidad

En esta unidad, hemos repasado diversos conceptos de análisis de ficheros XML. Hemos aprendido que existen diferentes sistemas y diversas librerías o parsers de código para realizar estas funciones (DOM, SAX).

Hemos profundizado en el mundo de las excepciones Java haciendo un breve resumen del concepto de excepción y de su jerarquía en las librerías de Java. Más tarde hemos aprendido su utilidad, los tipos de excepciones de existen, y una aplicación más práctica al tema que nos concierne sobre el tratamiento de ficheros XML .

Por último, hemos hecho una breve introducción a los sistemas de casos de prueba unitarios. Hemos explicado cómo podríamos usar el framework JUnit y cómo lo podemos incorporar a nuestro aplicativo o aplicación web Java.

Resolución del caso práctico de la unidad.

Tal y como se le ha planteado a José, dispone de 2 tipos de excepciones (IOException y SAXException). La primera debe de ser lanzada a la capa superior. ¿Cómo podría José realizar esta operación? De la siguiente forma:

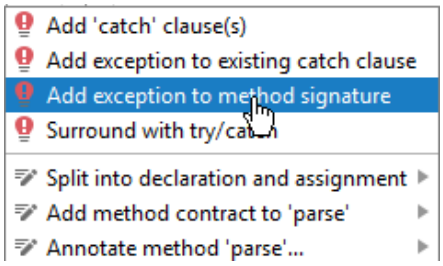

Fig 8. Añadir excepción al método.

De esta forma, se añadirá la excepción a la cabecera del método y ascenderá a la capa superior, por lo que tendrá la posibilidad de capturarla en dicha capa o de seguir lanzándola a capas superiores.

La segunda, de tipo SAXEception, sería el caso base. Bastaría con acotar la línea problemática entre un bloque try/catch; de esta forma, dicha excepción quedará capturada y en la parte del catch se podrán realizar las operaciones oportunas.

```
} catch (SAXException e) {

  e.printStackTrace();

}
```

Código 9. SAException.

/ 10. Webgrafía

Oracle web oficial:
https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html
https://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html