ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMAS

Ficheros XML

Acceso a datos con DOM Y SAX

Tanto **DOM** como **SAX** son estándares, herramientas que nos ofrecen la posibilidad de **lectura de ficheros XML**. V**erifican si sintácticamente son ficheros válidos**. Son los llamados "**parsers**" o **analizadores**. Algunas de 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**. Pero es más **lento** y **menos versátil**. **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**.

• El funcionamiento de DOM y SAX es muy diferente:

- DOM carga el fichero completo, tenemos todo el árbol del fichero disponible, pero ocupa mucha **más memoria** que SAX.
- **SAX tiene en memoria sólo** la parte del **nodo** o el **evento actual**.

En resumen:

- Usar **SAX** para **recorrer secuencialmente** los elementos del fichero XML y realizar ciertas operaciones.
- 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	Búsqueda de tags hacia delante y hacia detrás.
Va analizando nodo por nodo Secuencial	Estructuras de árbol Nodos en modo árbol Análisis del fichero completo
Sin muchas restricciones de memoria Menos memoria en uso no carga la totalidad del fichero solo si necesitamos fragmentos de documento	Carga el fichero en memoria Ocupa más memoria
Rapidez en tiempo de ejecución.	Más lento en tiempo de ejecución.
Menos potente	Más potente
Es solo de lectura	Se pueden insertar o eliminar nodos Buena opción para editar Múltiples procesos

Conversión de ficheros XML

Parsers (analizadores) de ficheros XML hay muchos. Pero ahora nos centraremos en la paquetería: "**javax.xml.parsers**".

Ejemplo de un "**parser" de tipo DOM.** La estructura se cargará en memoria y tendremos disponible el fichero completo.

```
package parserxml_dom;
import java.io.File;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;
public class ParserXML_DOM {
    static DocumentBuilder builder;
    public static void main(String[] args) {
        // Instanciamos la clase DocumentBuilderFactory
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        /*
        Establecemos el atributo de validación como "true" para
        asegurarnos que el fichero que se cargue esté bien validado.
        */
        factory.setValidating(true);
        /*
        Se hace un set también, al atributo de "ignorar los elementos que
        contengan espacios en blanco", a "true".
        */
        factory.setIgnoringElementContentWhitespace(true);
        try {
            /*
            Creamos un objeto DocumentBuilder por medio de la factoría creada
```

```
            previamente.
            */
            builder = factory.newDocumentBuilder();
        } catch (ParserConfigurationException ex) {
            Logger.getLogger(ParserXML_DOM.class.getName()).log(Level.SEVERE, null, ex);
        }
        // Instanciamos un nuevo fichero indicando la ruta del fichero a analizar.
        File file = new File("ejemplo.xml");
        try {
            /*
            Cargamos el fichero completo con el método builder.parse(file), y se
            asigna a un objeto de tipo Document. De esta forma quedará almacenado, y
            podremos realizar diferentes acciones con dicho objeto en las líneas
            siguientes.
            */
            Document doc = builder.parse(file);
        } catch (SAXException ex) {
            Logger.getLogger(ParserXML_DOM.class.getName()).log(Level.SEVERE, null, ex);
        } catch (IOException ex) {
            Logger.getLogger(ParserXML_DOM.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Ejemplo de cómo instanciar y trabajar con un parser (analizador) de XML, de tipo SAX.

```
package parserxml_sax;
import java.io.File;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
public class ParserXML_SAX {
    static File file;
    public static void main(String[] args) throws SAXException {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        factory.setValidating(true);
        SAXParser saxParser;
        try {
            saxParser = factory.newSAXParser();
            file= new File("ejemplo.xml");
            /*
            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 en donde debemos indicar las
            operaciones que realice nuestro analizador de código.
            */
            saxParser.parse(file, new DefaultHandler());
        } catch (ParserConfigurationException ex) {
            Logger.getLogger(ParserXML_SAX.class.getName()).log(Level.SEVERE, null, ex);
        } catch (IOException ex) {
```

```
            Logger.getLogger(ParserXML_SAX.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Procesamiento de XML
XPath

XPATH se usa para realizar **búsqueda de información** a través de un documento **XML**. 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.

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: posee **librería** muy completa de funcionalidades estándar de manipulación de Strings, valores numéricos, fechas, comparaciones, secuencias, valores booleanos...

Cómo hacer uso de esta librería XPath:

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**.

Ejemplo de uso de librería Xpath:

```
package com.mycompany.usolibreriaxpath2;
import java.io.File;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
public class UsoLibreriaXPath2 {
    public static void main(String[] args) {
        File file = new File("ejemplo.xml");
        DocumentBuilderFactory dBFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder;
        try {
            dBuilder = dBFactory.newDocumentBuilder();
            /*
            Creamos un Document con el método parse (analiza) del DocumentBuilder
           con el que podremos analizar el documento .xml
             */
            Document doc = dBuilder.parse(file);
            doc.getDocumentElement().normalize();
```

```
            // La instanciación del objeto Xpath la realizaremos de la siguiente forma:
            XPath xPath = XPathFactory.newInstance().newXPath();
            String expresionXPath = "/pizzas/pizza";
            // creamos una lista de nodos para la clase /pizza
            NodeList nodeList = (NodeList) xPath.compile(
                    expresionXPath).evaluate(
                            doc, XPathConstants.NODESET);
            // recorremos los nodos para trabajar con el xml:
            for (int i = 0; i < nodeList.getLength(); i++) {
               Node nNode = nodeList.item(i);
               System.out.println("\nCurrent Element: " + nNode.getNodeName());
               if (nNode.getNodeType() == Node.ELEMENT_NODE) {
                  Element element = (Element) nNode;
                  System.out.println("Nombre de la pizza: "
                          + element.getAttribute("nombre")
                          + "\nPrecio: " + element.getAttribute("precio")
                     /*
                       podemos seguir iterando en nodos agregando nueva
                     expresion XPath...
                      */
                     );
               }
            }
            /* multicatch para en este caso ahorra líneas de código.
           Se aconseja tratar cada try-catch por separado...
            */
                        } catch (ParserConfigurationException | SAXException | IOException |
XPathExpressionException ex) {
            Logger.getLogger(UsoLibreriaXPath2.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Excepciones

Breve introducción a las mismas a modo genérico:

Excepciones en Java y tipos.

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.

Las **excepciones se dividen en 3 categorías**:

• Excepciones **con chequeo** (**chequed exceptions**): Notificadas por el compilador en tiempo de **compilación**, no pueden ser ignoradas, y **fuerzan al programador a manejarlas**.

• Excepciones **sin chequeo** (**unchequed exceptions**): En **tiempo de ejecución**, 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.

Métodos más importantes en el manejo de excepciones con ficheros XML

• **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.** Es **más completo** que los anteriores ya que envuelve a todos ellos.

• **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-catch:

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

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.

Para aquellas sentencias try-catch que sean **obligatorias**, nuestro entorno de desarrollo nos lo notificará.

El IDE nos subrayará una operación, y si situamos el cursor encima de ésta, nos avisará 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.

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 { ...
```

• 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();
        }
```

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**:

Dispondremos con el IDE 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**, podremos 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 {
// operaciones parse...
} catch (ParserConfigurationException e) {
        e.printStackTrace();
} catch (IOException e) {
        e.printStackTrace();
} catch (XPathExpressionException e) {
        e.printStackTrace();
} catch (SAXException e) {
        e.printStackTrace();
}
```

Englobamos todos los bloques catch de una **serie de operaciones de análisis o "parsing"**, teniéndolas **mejor distribuidas y manejadas** de esta forma.

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**.

Si queremos disponer de las **librerías necesarias** para poder realizar nuestras pruebas unitarias, 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**. Hará un set up de los datos de la clase a testear.

• Anotación **@After**: Con esta anotación, se definirá un método cuyo código **se ejecutará**, siempre después de **finalizar cualquier test** dentro de nuestra clase. El método podría llamarse tearDownClass, o demoler la clase, ya que se pretende eliminar lo que ya no es necesario, limpiar...

Ejemplo:

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
public class NewEmptyJUnitTest {
    static private int variable;
    public NewEmptyJUnitTest() {
    }
    // el código pasa antes por @BeforeAll:
    @BeforeAll
    public static void instanciarVariablesTests() {
```

```
        // se realiza un set up de datos, este es un ejemplo:
        variable = 2;
    }

    // realizamos nuestro test:
    @Test
    public void test1() {
        // Aquí implementamos nuestro código para testear

    }

    /*
     Se utiliza al final de cada test, sobre todo para liberar recursos,
    limpiar basura, ahorrar recursos...
     */
    @AfterAll
    public static void demolerClase() {
    }

    // otra manera de usar @Before
    @BeforeEach
    public void setUp() {
    }

    // otra manera de usar @After
    @AfterEach
    public void tearDown() {
    }
}
```