

Unidad didáctica 1: Manejo de Ficheros

7. Ficheros XML

XML (*eXtensible Markup Language- Lenguaje de Etiquetado Extensible*) es un metalenguaje, es decir, un lenguaje para la definición de lenguajes de marcado. Nos permite jerarquizar y estructurar la información y describir los contenidos dentro del propio documento. Los ficheros XML son ficheros de texto escritos en lenguaje XML, donde la información está organizada de forma secuencial y en orden jerárquico. Existen una serie de marcas especiales como son los símbolos menor que, < y mayor que, > que se usan para delimitar las marcas que dan la estructura al documento. Cada marca tiene un nombre y puede tener 0 o más atributos. Un fichero XML sencillo tiene la siguiente estructura:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Empleados>
  <empleado>
    <id>1</id>
    <apellido>FERNANDEZ</apellido>
    <dep>10</dep>
    <salario>1000.45</salario>
  </empleado>
  <empleado>
    <id>2</id>
    <apellido>GIL</apellido>
    <dep>20</dep>
    <salario>2400.6</salario>
  </empleado>
  <empleado>
    <id>3</id>
    <apellido>LOPEZ</apellido>
    <dep>10</dep>
    <salario>3000.0</salario>
  </empleado>
</Empleados>
```

Los ficheros XML se pueden utilizar para proporcionar datos a una base de datos, o para almacenar copias de partes del contenido de la base de datos. También se utilizan para escribir ficheros de configuración de programas o en el protocolo SOAP¹ (*Simple Object Access Protocol*), para ejecutar comandos en servidores remotos; la información enviada al servidor remoto y el resultado de la ejecución del comando se envían en ficheros XML.

Para leer los ficheros XML y acceder a su contenido y estructura, se utiliza un procesador de XML o parser. El procesador lee los documentos y proporciona acceso a su contenido y estructura. Algunos de los procesadores más empleados son: DOM: *Modelo de Objetos de Documento*² y SAX: *API Simple para XML*³. Son independientes del lenguaje de programación y existen versiones particulares para Java, VisualBasic, C, etc. Utilizan dos enfoques muy diferentes:

- **DOM:** un procesador XML que utilice este planteamiento almacena toda la estructura del documento en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales (que son

¹ SOAP define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML.

² DOM es esencialmente una interfaz de plataforma que proporciona un conjunto estándar de objetos para representar documentos HTML, XHTML y XML un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos. A través del DOM, los programas pueden acceder y modificar el contenido, estructura y estilo de los documentos HTML y XML, que es para lo que se diseñó principalmente.

³ SAX son las siglas de "Simple API for XML", originalmente una API únicamente para el lenguaje de programación Java que después se convirtió en la API estándar de facto para usar XML en JAVA. Existen versiones de SAX no sólo para JAVA, sino también para otros lenguajes de programación (como Python). (API= Interfaz de Programación de Aplicaciones)

aquellos que no tienen descendientes). Una vez creado el árbol, se van recorriendo los diferentes nodos y se analiza a qué tipo particular pertenecen. Tiene su origen en el W3C. Este tipo de procesamiento necesita más recursos de memoria y tiempo sobre todo si los ficheros XML a procesar son grandes y complejos.

- **SAX**: un procesador que utilice este planteamiento lee un fichero XML de forma secuencial y produce una secuencia de eventos (comienzo/fin del documento, comienzo/fin de una etiqueta, etc.) en función de los resultados de la lectura. Cada evento invoca a un método definido por el programador. Este tipo de procesamiento prácticamente no consume memoria, pero por otra parte, impide tener una visión global del documento por el que navega.

7.1 Acceso a ficheros XML con DOM

Para poder trabajar con DOM en Java necesitamos las clases e interfaces que componen el paquete **org.w3c.dom** (contenido en el JSDK) y el paquete **javax.xml.parsers** del API estándar de Java que proporciona un par de clases abstractas que toda implementación DOM para Java debe extender. Estas clases ofrecen métodos para cargar documentos desde una fuente de datos (fichero, **InputStream**, etc.) Contiene dos clases fundamentales: **DocumentBuilderFactory** y **DocumentBuilder**.

DOM no define ningún mecanismo para generar un fichero XML a partir de un árbol DOM. Para eso usaremos el paquete **javax.xml.transform** que permite especificar una fuente y un resultado. La fuente y el resultado pueden ser ficheros, flujos de datos o nodos DOM entre otros.

Los programas Java que utilicen DOM necesitan estas interfaces (no se exponen todas, solo algunas de las que usaremos en los ejemplos):

- **Document**. Es un objeto que equivale a un ejemplar de un documento XML. Permite crear nuevos nodos en el documento.
- **Element**. Cada elemento del documento XML tiene un equivalente en un objeto de este tipo. Expone propiedades y métodos para manipular los elementos del documento y sus atributos.
- **Node**. Representa a cualquier nodo del documento.
- **NodeList**. Contiene una lista con los nodos hijos de un nodo.
- **Attr**. Permite acceder a los atributos de un nodo.
- **Text**. Son los datos carácter de un elemento.
- **CharacterData**. Representa a los datos carácter presentes en el documento. Proporciona atributos y métodos para manipular los datos de caracteres.
- **DocumentType**. Proporciona información contenida en la etiqueta `<!DOCTYPE>`.

A continuación un **ejemplo**, vamos a **crear un fichero XML** a partir del fichero aleatorio de empleados creado en el apartado anterior (*EscribirFichAleatorio.java*). Lo primero que hemos de hacer es importar los paquetes necesarios:

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import java.io.*;
```

A continuación creamos una instancia de **DocumentBuilderFactory** para construir el parser, se debe encerrar entre **try-cath** porque se puede producir la excepción **ParserConfigurationException**:

```

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
// Creamos una instancia de DocumentBuilderFactory para crear el parser. Entre
try-catch
    try{
        DocumentBuilder builder = factory.newDocumentBuilder();

```

Creamos un documento vacío de nombre *document* con el nodo raíz de nombre *Empleados* y asignamos la versión del XML, la interfaz **DOMImplementation** permite crear objetos **Document** con nodo raíz:

```

DOMImplementation implementation = builder.getDOMImplementation();
Document document = implementation.createDocument(null, "Empleados", null);
document.setXmlVersion("1.0");

```

El siguiente paso sería recorrer el fichero con los datos de los empleados y por cada registro crear un nodo empleado con 4 hijos (*id*, *apellido*, *dep* y *salario*). Cada nodo hijo tendrá su valor (*por ejemplo: 1, FERNANDEZ, 10, 1000.45*). Para crear un elemento usamos el método **createElement(String)** llevando como parámetro el nombre que se pone entre las etiquetas menor que y mayor que. El siguiente código crea y añade el nodo `<empleado>` al documento:

```

//creamos el nodo empleado
Element raiz = document.createElement("empleado");
//lo pegamos a la raíz del documento
document.getDocumentElement().appendChild(raiz);

```

A continuación se añaden los hijos de ese nodo (raiz), estos se añaden en el método *CrearElemento()*:

```

CrearElemento("id", Integer.toString(id), raiz, document); //añadir ID

CrearElemento("apellido", apellidos.trim(), raiz, document); //añadir APELLIDO

CrearElemento("dep", Integer.toString(dep), raiz, document); //añadir DEP

CrearElemento("salario", Double.toString(salario), raiz, document); //añadir SALARIO

```

Como se puede ver el método recibe el nombre del nodo hijo (*id*, *apellido*, *dep* o *salario*) y sus textos o valores que tienen que estar en formato String (*1, FERNANDEZ, 10, 1000.45*), el nodo al que se va a añadir (*raiz*) y el documento (*document*). Para crear el nodo hijo (`<id>` o `<apellido>` o `<dep>` o `<salario>`) se escribe:

```

Element elem = document.createElement(datoEmple); //Creamos un hijo

```

Para añadir su valor o su texto se usa el método `createTextNode(String)`:

```

Text text = document.createTextNode(valor); //damos valor

```

A continuación se añade el nodo hijo a la raíz (empleado) y su texto o valor al nodo hijo:

```

raiz.appendChild(elem); //pegamos el elemento hijo a la raíz
elem.appendChild(text); //pegamos el valor al elemento

```

Al final se generaría algo similar a esto por cada empleado:

```

<empleado><id>1</id><apellido>FERNANDEZ</apellido><dep>10</dep><salario>1000.45</
salario></empleado>

```

En los últimos pasos se crea la fuente XML a partir del documento:

```

//Crear la fuente XML a partir del documento
Source source = new DOMSource(document);

```

Se crea el resultado en el fichero *Empleados.xml*:

```
//Crear el resultado en e fichero Empleados.xml
Result result = new StreamResult(new java.io.File("Empleados.xml"));
```

Se obtiene un **TransformerFactory**:

```
//Se obtiene un TransformerFactory
Transformer transformer = TransformerFactory.newInstance().newTransformer();
```

Se realiza la transformación del documento a fichero:

```
//Se realiza la transformacion del documento al fichero
transformer.transform(source, result);
```

Para mostrar el documento por pantalla podemos especificar como resultado el canal de salida *System.out*:

```
Result console = new StreamResult(System.out);
transformer.transform(source, console);
```

El código completo

```
CrearEmpleadoXML.java X
1 package ficheros_XMLconDOM;
2
3 import java.io.*;
4
5 import javax.xml.parsers.*;
6 import javax.xml.transform.Transformer;
7 import javax.xml.transform.TransformerFactory;
8 import javax.xml.transform.dom.DOMSource;
9 import javax.xml.transform.stream.StreamResult;
10
11 import org.w3c.dom.DOMImplementation;
12 import org.w3c.dom.Document;
13 import org.w3c.dom.Element;
14 import org.w3c.dom.Text;
15
16
17 public class CrearEmpleadoXML {
18     /*
19      * Crear fichero XML a partir del fichero binario Empleados,
20      * ejemplo utilizado para ficheros acceso aleatorio
21      */
22     */
23
24     public static void main(String args[]) throws IOException{
25         String ruta="C:\\Users\\Isabel\\eclipse-workspace\\adt_t1\\src\\FicherosPruebas\\FichBinarioAleato
26         File fichero = new File(ruta + "AleatorioEmple.dat");
27         RandomAccessFile file = new RandomAccessFile(fichero, "r");
28
29         int id, dep, posicion=0; //para situarnos al principio del fichero
30         Double salario;
31         char apellido[] = new char[10], aux;
32
33         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
34         // Creamos una instancia de DocumentBuilderFactory para crear el parser. Entre try-catch
35         try{
36             DocumentBuilder builder = factory.newDocumentBuilder();
37             //Creamos un documento vacio de nombre document, con el nodo raiz de nombre Empleados y asignamos
38             //la interfaz DOMImplementation permite crear objetos Document con nodo raiz
39             DOMImplementation implementation = builder.getDOMImplementation();
40             Document document = implementation.createDocument(null, "Empleados", null);
41             document.setXmlVersion("1.0");
42         }
```

```

42
43 for(;;) {
44     file.seek(posicion); //nos posicionamos
45     id=file.readInt(); // obtengo id de empleado
46     for (int i = 0; i < apellido.length; i++) {
47         aux = file.readChar();
48         apellido[i] = aux;
49     }
50     String apellidos = new String(apellido);
51     dep = file.readInt();
52     salario = file.readDouble();
53
54     if(id>0) { //id validos a partir de 1
55         Element raiz = document.createElement("empleado"); //creamos nodo empleado
56         document.getDocumentElement().appendChild(raiz); // lo pegamos a la raiz del documento
57
58         //aadir ID
59         CrearElemento("id",Integer.toString(id), raiz, document);
60         //Apellido
61         CrearElemento("apellido",apellidos.trim(), raiz, document);
62         //aadir DEP
63         CrearElemento("dep",Integer.toString(dep), raiz, document);
64         //aadir salario
65         CrearElemento("salario",Double.toString(salario), raiz, document);
66     }
67     posicion= posicion + 36; // me posiciono para el sig empleado
68     if (file.getFilePointer() == file.length()) break;
69
70 } //fin del for que recorre el fichero
71
72 //Crear la fuente XML a partir del documento
73 DOMSource source = new DOMSource(document);
74 //Crear el resultado en el fichero Empleados.xml
75 StreamResult result = new StreamResult(new java.io.File(ruta + "Empleados.xml"));
76 //Se obtiene un TransformerFactory
77 Transformer transformer = TransformerFactory.newInstance().newTransformer();
78 //Se realiza la transformacion del documento al fichero
79 transformer.transform(source, result);
80
81 }catch(Exception e){ System.err.println("Error: "+e); }
82
83 file.close(); //cerrar fichero
84 } //fin de main
85
86 //Insercion de los datos del empleado
87 static void CrearElemento(String datoEmple, String valor,
88                         Element raiz, Document document){
89     Element elem = document.createElement(datoEmple); //creamos hijo
90     Text text = document.createTextNode(valor); //damos valor
91     raiz.appendChild(elem); //pegamos el elemento hijo a la raiz
92     elem.appendChild(text); //pegamos el valor
93 }
94 } //fin de la clase
95

```

Ejemplo Para leer un documento XML, creamos una instancia de **DocumentBuilderFactory** para construir el parser y cargamos el documento con el método *parse()*:

```

DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse(new File("Empleados.xml"));

```

Obtenemos la lista de nodos con nombre empleado de todo el documento con el método:

```

NodeList empleados = document.getElementsByTagName("empleado");

```

Se realiza un bucle para recorrer esta lista de nodos. Por cada nodo se obtienen sus etiquetas y sus valores llamando a la función *getNode()*.

El código completo es el siguiente:

```
LeerEmpleadoXML.java
2 import java.io.File;
3 import javax.xml.parsers.*;
4 import org.w3c.dom.*;
5
6 public class LeerEmpleadoXML {
7     public static void main(String[] args) {
8         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
9         try {
10             DocumentBuilder builder = factory.newDocumentBuilder();
11             Document document = builder.parse(new File("Empleados.xml"));
12             document.getDocumentElement().normalize();
13
14             System.out.printf("Elemento raiz: %s %n", document.getDocumentElement().getNodeName());
15             //crea una lista con todos los nodos empleado
16             NodeList empleados = document.getElementsByTagName("empleado");
17             System.out.printf("Nodos empleado a recorrer: %d %n", empleados.getLength());
18
19             //recorrer la lista
20             for (int i = 0; i < empleados.getLength(); i++) {
21                 Node emple = empleados.item(i); //obtener un nodo empleado
22                 if (emple.getNodeType() == Node.ELEMENT_NODE) { //tipo de nodo
23                     //obtener los elementos del nodo
24                     Element elemento = (Element) emple;
25                     System.out.printf("ID = %s %n", elemento.getElementsByTagName("id").item(0).getTextContent());
26                     System.out.printf(" * Apellido = %s %n", elemento.getElementsByTagName("apellido").item(0).getTextContent());
27                     System.out.printf(" * Departamento = %s %n", elemento.getElementsByTagName("dep").item(0).getTextContent());
28                     System.out.printf(" * Salario = %s %n", elemento.getElementsByTagName("salario").item(0).getTextContent());
29                 }
30             }
31         } catch (Exception e) {
32             {e.printStackTrace();}
33         } //fin de main
34     } //fin de la clase
}
```

Realizar:

Modificar Empleados.xml de manera que el id sea un atributo de empleado.

Crear una clase LeerEmpleadoXML_2

Crear una clase LeerEmpleadoXML_3, para el supuesto de que no conozcamos la estructura del archivo xml

Crear una clase CrearPersonaXML a partir del fichero de objetos generado con la clase EscribirFichObjeto.java y otra clase LeerPersonaXML para leer el contenido del archivo Personas.xml generado anteriormente.

Realizar la Actividad 1.5

7.2 Acceso a ficheros XML con SAX

SAX (*API Simple para XML*) es el primer punto de unión del mundo de XML con el mundo de la programación en general, y en particular con Java

Esta API consiste en un conjunto de interfaces y clases con sus correspondientes métodos que ofrecen una herramienta muy útil para el procesamiento de documentos XML desde un programa escrito en Java. Permite analizar los documentos de forma secuencial (es decir, no carga todo el fichero en memoria como hace DOM), esto implica poco consumo de memoria aunque los documentos sean de gran tamaño, en contraposición, impide tener una visión global del documento que se va a analizar. SAX es más complejo de programar que DOM, es un API totalmente escrita en Java e incluida dentro del JRE (*Java Runtime Environment*) que nos permite crear nuestro propio parser de XML.

Esta API SAX consta de una serie de clases, con sus correspondientes métodos, que nos permiten trabajar con un documento XML desde un programa escrito en Java, pudiendo acceder a los datos, comprobar si está bien formado, si es válido, etc.

Para poder trabajar con documentos XML mediante SAX necesitamos un analizador SAX. El analizador realiza el trabajo sucio (detectar cuándo empieza y termina un elemento, gestionar los espacios de nombres, comprobar que el documento está bien formado, etc.), de forma que podemos concentrarnos en los aspectos específicos de nuestra aplicación.

Existen muchos analizadores en el mercado, pero no todos se pueden utilizar desde Java.

Para utilizar SAX desde un programa escrito en Java necesitamos conseguir las clases que componen el analizador y asegurarnos de incluir estas clases en la ruta de clase

Realizar el ejemplo está bien formado (clases AnalizarFichXMLBienFormado.java y AnalizaDocHandler.java)

La lectura de un documento XML produce eventos que ocasiona la llamada a métodos, los eventos son encontrar la etiqueta de inicio y fin del documento (**startDocument()** y **endDocument()**), la etiqueta de inicio y fin de un elemento (**startElement()** y **endElement()**), los caracteres entre etiquetas (**characters()**), etc:

Documento XML (alumnos.xml)	Métodos asociados a eventos del documento
<pre><?xml version="1.0"?> <listadealumnos> <alumno> <nombre> Juan </nombre> <edad> 19 </edad> </alumno> <alumno> <nombre> Maria </nombre> <edad> 20 </edad> </alumno> </listadealumnos></pre>	<pre>startDocument() startElement() startElement() startElement() characters() endElement() startElement() characters() endElement() endElement() startElement() startElement() characters() endElement() startElement() characters() endElement() endElement() endElement() endDocument()</pre>

Vamos a construir un **ejemplo** sencillo en Java (EjemploSAXalumnos.java) que muestra los pasos básicos necesarios para hacer que se puedan tratar los eventos. **Modificado a continuación, la clase XMLReaderFactory está obsoleta desde la versión 9. Hay que crear una instancia de la implementación Se crea una instancia de la implementación predeterminada de SAXParserFactory**

En primer lugar se incluyen las clases e interfaces de SAX

```
import java.io.*;
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;
```

Se crea un objeto procesador de XML, es decir, un **XMLReader**, durante la creación de este objeto se puede producir una excepción (**SAXException**) que es necesario capturar (se incluye en el método main()):

```
XMLReader procesadorXML = XMLReaderFactory.createXMLReader();
```

```

        SAXParserFactory parserFactory = SAXParserFactory.newInstance();
// Se crea una instancia de SAXParser usando los parametros actuales del SAXParserFactory
        SAXParser parser = parserFactory.newSAXParser();
// Se crea el objeto procesador de XML
        XMLReader procesadorXML = parser.getXMLReader();

```

A continuación hay que indicar al **XMLReader** qué objetos poseen los métodos que tratarán los eventos. Estos objetos serán normalmente implementaciones de las siguientes interfaces:

- **ContentHandler**: recibe las notificaciones de los eventos que ocurren en el documento.
- **DTDHandler**: recoge eventos relacionados con la DTD.
- **ErrorHandler**: define métodos de tratamientos de errores.
- **EntityResolver**: sus métodos se llaman cada vez que se encuentra una referencia a una entidad.
- **DefaultHandler**: clase que provee una implementación por defecto para todos sus métodos, **el programador definirá los métodos que sean utilizados por el programa**. Esta clase es de la que extenderemos para poder **crear nuestro parser de XML**. En el ejemplo, la clase se llama *GestionContenido2* y se tratan solo los eventos básicos: inicio y fin de documento, inicio y fin de etiqueta encontrada, encuentra datos carácter (**startDocument()**, **endDocument()**, **startElement()**, **endElement()**, **characters()**):
 - **startDocument**: se produce al comenzar el procesamiento del documento XML.
 - **endDocument**: se produce al finalizar el procesamiento del documento XML.
 - **startElement**: se produce al comenzar el procesamiento de una etiqueta XML. Es aquí donde se leen los atributos de las etiquetas.
 - **endElement**: se produce al finalizar el procesamiento de una etiqueta XML.
 - **characters**: se produce al encontrar una cadena de texto.

Para indicar al procesador XML los objetos que realizarán el tratamiento se utiliza alguno de los siguientes métodos incluidos dentro de los objetos **XMLReader**: **setContentHandler()**, **setDTDHandler()**, **setEntityResolver()** y **setErrorHandler()**; cada uno trata un tipo de evento y está asociado con una interfaz determinada. En el ejemplo usaremos **setContentHandler()** para tratar los eventos que ocurren en el documento:

```

GestionContenido2 gestor= new GestionContenido2();
procesadorXML.setContentHandler(gestor);

```

A continuación se define el fichero XML que se va a leer mediante un objeto `InputStream`:

```

InputStream fileXML = new InputStream("alumnos.xml");

```

Por último, se procesa el documento XML mediante el método `parse()` del objeto `XMLReader`, le pasamos un objeto `InputStream`:

```

procesadorXML.parse(fileXML);

```

El código completo:


```

EjemploSAXalumnos.java X
7 import javax.xml.parsers.SAXParser;
8 import javax.xml.parsers.SAXParserFactory;
9
10 import org.xml.sax.Attributes;
11 import org.xml.sax.InputSource;
12 import org.xml.sax.SAXException;
13 import org.xml.sax.XMLReader;
14 import org.xml.sax.helpers.DefaultHandler;
15 //import org.xml.sax.helpers.XMLReaderFactory;
16
17 public class EjemploSAXalumnos {
18     public static void main(String[] args) throws FileNotFoundException, IOException, SAXException, ParserConfigurationException {
19         // Como la clase XMLReaderFactory esta obsoleta desde la version 9, utilizo la recomendada ahora.
20         // Se crea una instancia de la implementacion predeterminada de SAXParserFactory
21         // XMLReader procesadorXML = XMLReaderFactory.createXMLReader();
22         String ruta="C:\\Users\\Isabel\\eclipse-workspace\\adt_t1\\src\\FicherosPruebas\\";
23
24         SAXParserFactory parserFactory = SAXParserFactory.newInstance();
25         // Se especifica que el parser producido por este codigo tendra soporte para espacios de nombre XML
26         parserFactory.setNamespaceAware(true);
27         // Se crea una instancia de SAXParser usando los parametros actuales del SAXParserFactory
28         SAXParser parser = parserFactory.newSAXParser();
29         // Se crea el objeto procesador de XML
30         XMLReader procesadorXML = parser.getXMLReader();
31
32         GestionContenido2 gestor = new GestionContenido2();
33         procesadorXML.setContentHandler(gestor);
34         InputSource fileXML = new InputSource(ruta + "alumnos.xml");
35         procesadorXML.parse(fileXML);
36
37     }
38 } //fin EjemploSAX
39
40 /**
41  * Clase que hereda de DefaultHandler, tratando los eventos basicos
42  * inicio y fin de documento
43  * inicio y fin de etiqueta encontrada
44  * datos caracter encontrados
45  */
46 class GestionContenido2 extends DefaultHandler {
47
48     public GestionContenido2() {
49         super();
50     }
51
52     public void startDocument() {
53         System.out.println("Comienzo del Documento XML");
54     }
55
56     public void endDocument() {
57         System.out.println("Final del Documento XML");
58     }
59
60     public void startElement(String uri, String nombre, String nombreC, Attributes atts) {
61         System.out.printf("\t Principio Elemento: %s %n", nombre);
62     }
63
64     public void endElement(String uri, String nombre, String nombreC) {
65         System.out.printf("\t Fin Elemento: %s %n", nombre);
66     }
67
68     public void characters(char[] ch, int inicio, int longitud) throws SAXException {
69         String car = new String(ch, inicio, longitud);
70         car = car.replaceAll("[\t\n]", "");
71         System.out.printf("\t Caracteres: %s %n", car);
72     }
73 } // fin de GestionContenido

```

Realizar LeerEmpleadoXML: Visualiza utilizando SAX, el contenido del fichero Empleados.xml, creado en el ejercicio CrearEmpleadoXML.java (paquete ficheros_XMLconDOM)

Realizar LeerPersonaTfno.java (PersonaTfno es un XML con atributo teléfono, nombre y edad), en este ejemplo vamos a personalizar el Handler (PerosnaTfnoHandler.java)

```

import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

import org.xml.sax.SAXException;

public class LeerPersonaTfno {
    public static void main(String[] args) {
        String ruta= "C:\\Users\\Isabel\\eclipse-workspace\\adt_t1\\src\\FicherosPruebas\\";

        try {
            SAXParserFactory factory = SAXParserFactory.newInstance();

            SAXParser parser = factory.newSAXParser();

            PersonaTfnoHandler handler = new PersonaTfnoHandler();
            parser.parse(ruta + "PersonasTfno.xml", handler);

        } catch (ParserConfigurationException | SAXException | IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
}

public class PersonaTfnoHandler extends DefaultHandler {

    //StringBuilder para ir almacenando el contenido de la etiqueta.
    private StringBuilder value;

    public PersonaTfnoHandler() {
        this.value = new StringBuilder();
    }

    //sobrescribir: startElement, characters y endElement.
    //StringBuilder longitud a 0, ya que lo rellenaremos en el método characters
    //y lo mostraremos en el método endElement.

    //los atributos, lo cogemos desde el parámetro attributes.
    //El parámetro qName es el nombre de la etiqueta. Cuando la etiqueta es persona, cogemos el atributo
    @Override
    public void startDocument() {
        System.out.println("Comienzo de PersonasTfno.XML");
    }

    @Override
    public void endDocument() {
        System.out.println("Final PersonasTfno.XML");
    }

    @Override
    public void startElement(String uri, String localName,
        String qName, Attributes attributes)
        throws SAXException {
        this.value.setLength(0);
        if (qName.equals("persona")) {
            String tfno = attributes.getValue("telefono");
            System.out.println("Atributo teléfono: " + tfno);
        }
    }
}

```

```

41 //En el método characters, añadimos el contenido de la etiqueta al StringBuilder.
42 @Override
43 public void characters(char ch[], int start, int length)
44     throws SAXException {
45
46     this.value.append(ch, start, length);
47 }
48 //En el método endElement, salta cuando terminamos con esa etiqueta.
49 //mostrar el contenido de nuestro StringBuilder.
50 //Según el parámetro qName, mostraremos un mensaje u otro.
51 @Override
52 public void endElement(String uri, String localName, String qName)
53     throws SAXException {
54
55     switch (qName) {
56         case "persona":
57             System.out.println("");
58             break;
59         case "nombre":
60             System.out.println("Nombre: " + this.value.toString());
61             break;
62         case "edad":
63             System.out.println("Edad: " + this.value.toString());
64             break;
65     }
66 }
67 }

```

7.3 Serialización de objetos XML

A continuación, vamos a ver cómo se pueden serializar objetos Java a XML y viceversa; utilizaremos para ello la librería **XStream**. Para poder utilizarla hemos de descargarlos los JAR desde el sitio Web: <http://x-stream.github.io/download.html> (ir a versiones anteriores). Para el ejemplo se ha descargado el fichero **xstream-distribution-1.4.8-bin.zip**⁴ que hemos de descomprimir y buscar el JAR **xstream-1.4.8.jar** que está en la carpeta *lib* que es el que usaremos para el ejemplo. También necesitamos el fichero **kxml2-2.3.0.jar** que se localiza en la carpeta *lib\xstream*. Una vez que tenemos los dos ficheros los añadimos a nuestro proyecto Eclipse o NetBeans o los definimos en el CLASSPATH, por ejemplo, supongamos que tenemos los JAR en la carpeta D:\unil\xstream, el CLASSPATH nos quedaría:

```

SET CLASSPATH =
.;D:\unil\xstream\kxml2-2.3.0.jar;D:\unil\xstream\xstream-1.4.8.jar

```

(esto con un SO Windows)

Descomprimir

Crear una carpeta de nombre lib en el proyecto (botón derecho –nuevo-folder)

Desde el explorador de windows copiar los archivos jar en la carpeta lib

Ir a Eclipse y hacer Refresh sobre el proyecto (F5)

Expandir la carpeta lib, **seleccionar los .jar**

Botón derecho sobre la selección y BuildPath-Add tu Build Path.

Mas formas de agregar librerías: [https://es.wikihow.com/a%C3%B1adir-un-jar-a-un-proyecto-en-eclipse-\(java\)](https://es.wikihow.com/a%C3%B1adir-un-jar-a-un-proyecto-en-eclipse-(java))

Partimos del fichero *FichPersona.dat* que utilizamos en epígrafes anteriores y contiene objetos *Persona*. Crearemos una lista de objetos *Persona* y la convertiremos en un fichero de datos XML. Necesitaremos la clase *Persona* (ya definida) y la clase *ListaPersonas* en la que se define una lista de objetos *Persona* que pasaremos al fichero XML.

⁴ <https://repo1.maven.org/maven2/com/thoughtworks/xstream/xstream-distribution/1.4.8/>

```

1 package UD1;
2 import java.util.ArrayList;
3 import java.util.List;
4
5 public class ListaPersonas {
6
7     private List<Persona> lista = new ArrayList<Persona>();
8
9     public ListaPersonas(){
10    }
11
12    public void add(Persona per) {
13        lista.add(per);
14    }
15
16    public List<Persona> getListaPersonas() {
17        return lista;
18    }
19 }

```

El proceso consistirá en recorrer el fichero *FichPersona.dat* para crear una lista de personas que después se insertarán en el fichero *Personas.xml*, el código Java es el siguiente:

```

1 import java.io.*;
2 import com.thoughtworks.xstream.XStream;
3 public class EscribirPersonas {
4
5     public static void main(String[] args) throws IOException, ClassNotFoundException {
6         File fic = new File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\FichPersona.dat");
7         FileInputStream filein = new FileInputStream(fic); //flujo de entrada
8         //conecta el flujo de bytes al flujo de datos
9         ObjectInputStream objectIS = new ObjectInputStream(filein);
10        System.out.println("Comienza el proceso de creación del fichero a XML ...");
11        //Creamos un objeto Lista de Personas
12        ListaPersonas listaper = new ListaPersonas();
13        try {
14            while (true) { //lectura del fichero
15                //leer una Persona
16                Persona persona = (Persona) objectIS.readObject();
17                listaper.add(persona); //añadir persona a la lista
18            }
19        } catch (EOFException eo) {}
20        objectIS.close(); //cerrar stream de entrada
21        try {
22            XStream xstream = new XStream();
23            //cambiar de nombre a las etiquetas XML
24            xstream.alias("ListaPersonasMunicipio", ListaPersonas.class);
25            xstream.alias("DatosPersona", Persona.class);
26            //quitar etiqueta lista (atributo de la clase ListaPersonas)
27            xstream.addImplicitCollection(ListaPersonas.class, "lista");
28            //Insertar los objetos en el XML
29            xstream.toXML(listaper, new FileOutputStream("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\Personas.xml"));
30            System.out.println("Creado fichero XML....");
31        } catch (Exception e) {
32            e.printStackTrace();
33        }
34    } // fin main
35 } //fin EscribirPersonas

```

Analizando los que hemos realizado: En primer lugar para utilizar **XStream**, simplemente creamos una instancia de la clase XStream:

```
XStream xstream = new XStream();
```

En general las etiquetas XML se corresponden con el nombre de los atributos de la clase, pero se pueden cambiar usando el método *alias(String alias, Class clase)*. En el ejemplo se ha dado un alias a la clase *ListaPersonas*, en el XML aparecerá con el nombre *ListaPersonasMunicipio*:

```
xstream.alias("ListaPersonasMunicipio", ListaPersonas.class);
```

También se ha dado un alias a la clase *Persona*, en el XML aparecerá con el nombre *DatosPersona*:

```
xstream.alias("DatosPersona", Persona.class);
```

El método **aliasField(String alias, Class clase, String nombrecampo)**, permite crear un alias para un nombre de campo. Por ejemplo, si queremos cambiar el nombre de los campos *nombre* y *edad* (de la clase *Persona*) crearíamos los siguientes alias:

```
xstream.aliasField( "Nombre alumno", Persona.class, "nombre");
xstream.aliasField( "Edad alumno", Persona.class, "edad");
```

Entonces en el fichero XML se crearían con las etiquetas *<Nombre alumno>* en lugar de *<nombre>* y *<Edad alumno>* en lugar de *<edad>*.

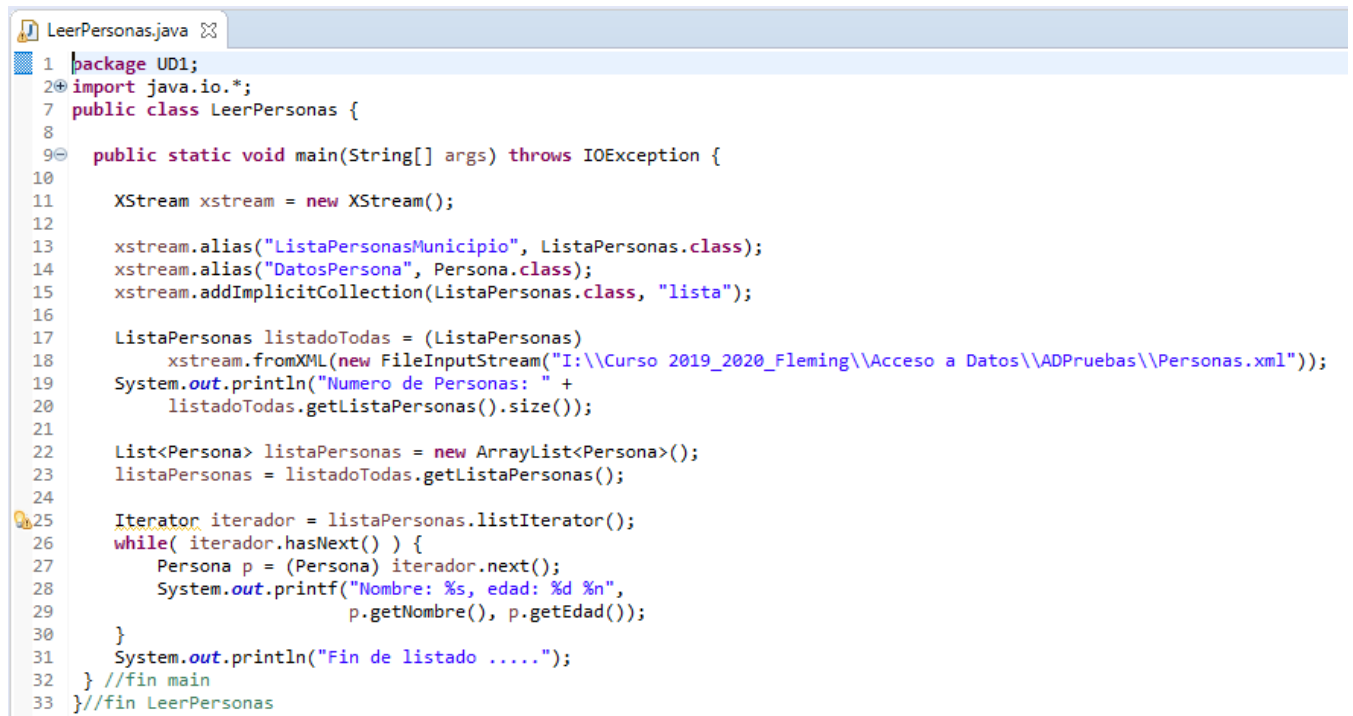
Para que no aparezca el atributo *lista* de la clase *ListaPersonas* en el XML generado se ha utilizado el método **addImplicitCollection(Class clase, String nombredecampo)**

```
xstream.addImplicitCollection(ListaPersonas.class, "lista");
```

Por último, para generar el fichero *Personas.xml* a partir de la lista de objetos se utiliza el método **toXML(Object objeto, OutputStream out)**:

```
xstream.toXML(listaper, new FileOutputStream("Personas.xml"));
```

El proceso para realizar la lectura del fichero XML generado es el siguiente:



```
LeerPersonas.java
1 package UD1;
2 import java.io.*;
7 public class LeerPersonas {
8
9     public static void main(String[] args) throws IOException {
10
11         XStream xstream = new XStream();
12
13         xstream.alias("ListaPersonasMunicipio", ListaPersonas.class);
14         xstream.alias("DatosPersona", Persona.class);
15         xstream.addImplicitCollection(ListaPersonas.class, "lista");
16
17         ListaPersonas listadoTodas = (ListaPersonas)
18             xstream.fromXML(new FileInputStream("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\Personas.xml"));
19         System.out.println("Numero de Personas: " +
20             listadoTodas.getListaPersonas().size());
21
22         List<Persona> listaPersonas = new ArrayList<Persona>();
23         listaPersonas = listadoTodas.getListaPersonas();
24
25         Iterator iterador = listaPersonas.listIterator();
26         while( iterador.hasNext() ) {
27             Persona p = (Persona) iterador.next();
28             System.out.printf("Nombre: %s, edad: %d %n",
29                 p.getNombre(), p.getEdad());
30         }
31         System.out.println("Fin de listado .....");
32     } //fin main
33 } //fin LeerPersonas
```

Se deben utilizar los métodos *alias()* y *addImplicitCollection()* para leer el XML ya que se usaron para hacer la escritura del mismo. Para obtener el objeto con la lista de personas o lo que es lo mismo para deserializar el objeto a partir del fichero, utilizarnos el método *fromXML (InputStream input)* que devuelve un tipo **Object**:

```
ListaPersonas listadoTodas = (ListaPersonas)
    xstream.fromXML(new FileInputStream("Personas.xml"));
```

7.4 Conversión de ficheros XML a otro formato

XSL (Extensible Stylesheet Language) es toda una familia de recomendaciones del World Wide Web Consortium (<http://www.w3.org/Style/XSL/>) para expresar hojas de estilo en lenguaje XML. Una hoja de estilo XSL describe el proceso de presentación a través de un pequeño conjunto de elementos XML. Esta hoja, puede contener elementos de reglas que representan a las reglas de construcción y elementos de reglas de estilo que representan a las reglas de mezcla de estilos. En el siguiente ejemplo vamos a ver cómo a partir de un fichero XML que contiene datos y otro XSL que contiene la presentación de esos datos, se puede generar un fichero HTML usando el lenguaje Java. Los ficheros son los siguientes:

FICHERO alumnos.xml

```
<?xml version="1.0"?>
<listadealumnos>
  <alumno>
    <nombre>Juan</nombre>
    <edad>19</edad>
  </alumno>
  <alumno>
    <nombre>Maria</nombre>
    <edad>20</edad>
  </alumno>
</listadealumnos>
```

FICHERO alumnosPlantilla.xsl

```
<?xml version="1.0" encoding='ISO-8859-1'?>
<xsl:stylesheet version="1.0" xmlns:xsl="
http://www.w3.org/1999/XSL/Transform">
  <xsl:template match='/'>
    <html><xsl:apply-templates /></html>
  </xsl:template>
  <xsl:template match='listadealumnos'>
    <head><title>LISTADO DE ALUMNOS</title></head>
    <body>
      <h1>LISTA DE ALUMNOS</h1>
      <table border='1'>
        <tr><th>Nombre</th><th>Edad</th></tr>
        <xsl:apply-templates select='alumno' />
      </table>
    </body>
  </xsl:template>
  <xsl:template match='alumno'>
    <tr><xsl:apply-templates /></tr>
  </xsl:template>
  <xsl:template match='nombre|edad'>
    <td><xsl:apply-templates /></td>
  </xsl:template>
</xsl:stylesheet>
```

Para realizar la transformación se necesita obtener un objeto **Transformer** que se obtiene creando una instancia de **TransformerFactory** y aplicando el método *newTransformer(Source source)* a la fuente XSL que vamos a utilizar para aplicar la transformación del fichero de datos XML, o lo que es lo mismo, para aplicar la hoja de estilos XSL al fichero XML:

```
Transformer transformer = TransformerFactory.newInstance().newTransformer(estilos);
```

La transformación se consigue llamando al método *transform(Source fuenteXml, Result resultado)*, pasándole los datos (el fichero XML) y el stream de salida (el fichero HTML):

```
transformer.transform(datos, result);
```



```

convertidor.java
2 import javax.xml.transform.*;
3 import javax.xml.transform.stream.*;
4 import java.io.*;
5
6 public class convertidor {
7     public static void main(String argv[]) throws IOException{
8         String hojaEstilo = "PlantillaConver.xsl";
9         String datosAlumnos = "alumnosConver.xml";
10        File pagHTML = new File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\mipaginaConver.html");
11        FileOutputStream fileout = new FileOutputStream(pagHTML); //crear fichero HTML
12
13        Source estilos =new StreamSource(hojaEstilo); //fuente XSL
14        Source datos =new StreamSource(datosAlumnos); //fuente XML
15        Result result = new StreamResult(fileout); //resultado de la transformación
16
17        try{
18            Transformer transformer = TransformerFactory.newInstance().newTransformer(estilos);
19            transformer.transform(datos, result); //obtiene el HTML
20        }
21        catch(Exception e){
22            System.err.println("Error: "+e);
23        }
24
25        fileout.close(); //cerrar fichero
26    } //de main
27 } //de la clase

```

8. Excepciones: Detección y Tratamiento.

Una excepción es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias. Cuando no es capturada por el programa, es capturada por el gestor de excepciones por defecto que retoma un mensaje y detiene el programa. La ejecución del siguiente programa (típica división entre 0) produce una excepción y visualiza un mensaje indicando el error:

```

ejemploExcepcion.java
1 package UD1;
2
3 public class ejemploExcepcion {
4     public static void main(String[] args) {
5         int nume=10, denom=0, cociente;
6         cociente=nume/denom;
7         System.out.println("Resultado:" +cociente);
8     }
9 }

```

Problems @ Javadoc Declaration Console

<terminated> ejemploExcepcion [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\jav
 Exception in thread "main" java.lang.ArithmeticException: / by zero
 at UD1.ejemploExcepcion.main(ejemploExcepcion.java:6)

Cuando dicho error ocurre dentro de un método Java, el método crea un objeto **Exception** y lo maneja fuera, en el sistema de ejecución. El manejo de excepciones en Java está diseñado pensando en situaciones en las que el método que detecta un error no es capaz de manejarlo, **un método así lanzará una excepción**.

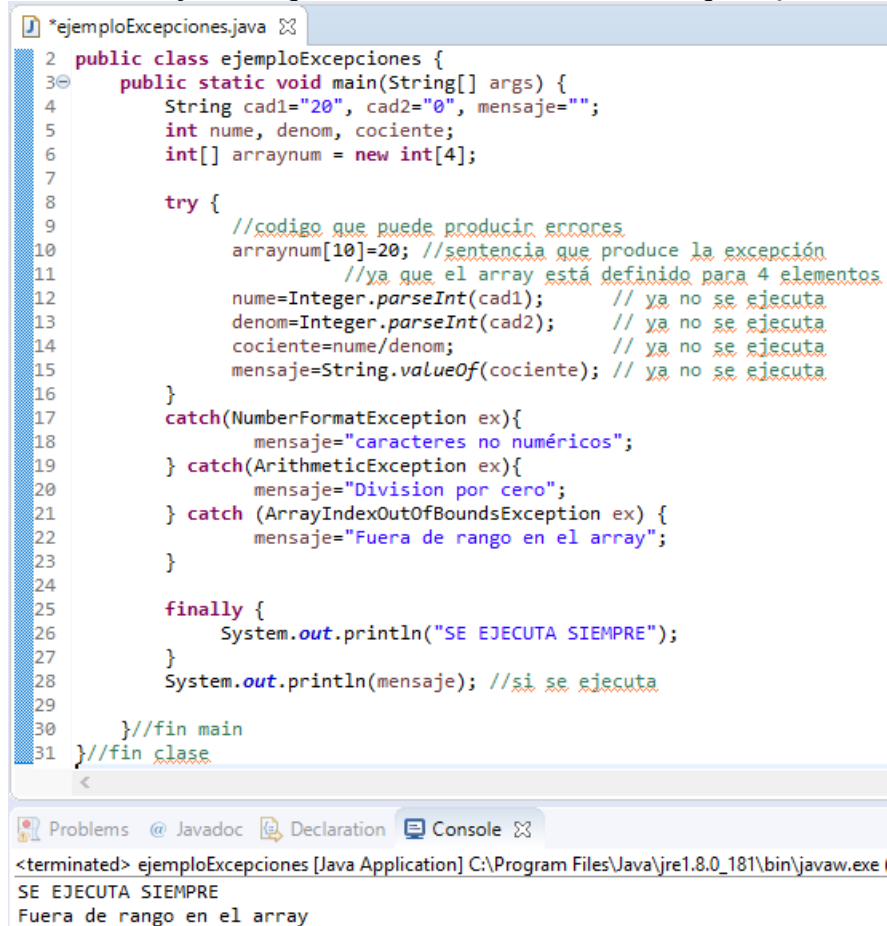
Las excepciones en Java son objetos de clases derivadas de la clase base **Exception** que a su vez es una clase derivada de la clase base **Throwable**.

8.1 Capturar excepciones

Para capturar una excepción se utiliza el bloque **try-catch**. Se encierra en un bloque **try** el código que puede generar una excepción, este bloque va seguido por uno o más bloques **catch**. Cada bloque **catch** especifica el tipo de excepción que puede atrapar y contiene un manejador de excepciones. Después del último bloque **catch** puede aparecer un bloque **finally** (opcional) que siempre se ejecuta haya ocurrido o no la excepción; se utiliza el bloque **finally** para cerrar ficheros o liberar otros recursos del sistema después de que ocurra una excepción:

```
try {  
    //Código que puede generar excepciones  
} catch(excepcion1 e1) {  
    //manejo de la excepcion1  
  
} catch(excepcion2 e2) {  
    //manejo de la excepcion2  
  
}  
//etc  
finally {  
    // Se ejecuta después de try o catch  
}
```

El siguiente **ejemplo** muestra la captura de 3 tipos de excepciones que se pueden producir. Cuando se encuentra el primer error se produce un salto al bloque **catch** que maneja dicho error; en este caso al encontrar la sentencia de asignación `arraynum[10] = 20;` se lanza la excepción *ArrayIndexOutOfBoundsException* (ya que el array está definido para 4 elementos y se da un valor al elemento de la posición 10) donde se ejecutan las instrucciones indicadas en el bloque, las sentencias situadas debajo de la que causó el error dentro del bloque **try** no se ejecutarán:



```
*ejemploExcepciones.java  X  
2 public class ejemploExcepciones {  
3     public static void main(String[] args) {  
4         String cad1="20", cad2="0", mensaje="";  
5         int nume, denom, cociente;  
6         int[] arraynum = new int[4];  
7  
8         try {  
9             //codigo que puede producir errores  
10            arraynum[10]=20; //sentencia que produce la excepción  
11                //ya que el array está definido para 4 elementos  
12            nume=Integer.parseInt(cad1); // ya no se ejecuta  
13            denom=Integer.parseInt(cad2); // ya no se ejecuta  
14            cociente=nume/denom; // ya no se ejecuta  
15            mensaje=String.valueOf(cociente); // ya no se ejecuta  
16        }  
17        catch(NumberFormatException ex){  
18            mensaje="caracteres no numéricos";  
19        } catch(ArithmeticException ex){  
20            mensaje="Division por cero";  
21        } catch (ArrayIndexOutOfBoundsException ex) {  
22            mensaje="Fuera de rango en el array";  
23        }  
24  
25        finally {  
26            System.out.println("SE EJECUTA SIEMPRE");  
27        }  
28        System.out.println(mensaje); //si se ejecuta  
29    } //fin main  
30 } //fin clase
```

Problems @ Javadoc Declaration Console X

<terminated> ejemploExcepciones [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe
SE EJECUTA SIEMPRE
Fuera de rango en el array

Para capturar cualquier excepción utilizamos la clase base **Exception**. Si se usa habrá que ponerla al **final de la lista de manejadores** para evitar que los manejadores que vienen después queden ignorados. Por ejemplo, el siguiente código maneja varias excepciones, si se produce alguna para la que no se ha definido manejador será capturada por **Exception**:

```
try{// código que puede producir error
} catch (NumberFormatException ex) {
    //tratamiento excepción
} catch (ArithmeticException ex) {
    //tratamiento excepción
} catch (ArrayIndexOutOfBoundsException ex) {
    //tratamiento excepción
} catch (Exception ex)
    //tratamiento si se produce cualquier otra excepción
} finally {
    //Se ejecuta haya o no excepción
}
```

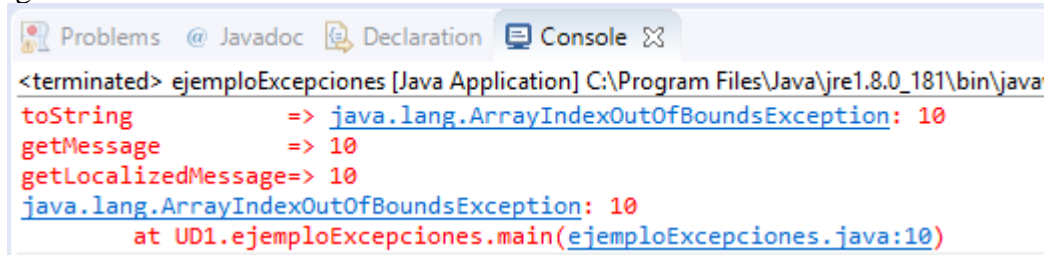
Para obtener más información sobre la excepción se puede llamar a los métodos de la clase base **Throwable**, algunos son:

Método	Función
String getMessage()	Devuelve la cadena de error del objeto
String getLocalizedMessage()	Crea una descripción local de este objeto
String toString()	Devuelve una breve descripción del objeto
void printStackTrace(), printStackTrace(PrintStream) o printStackTrace(PrintWriter)	Visualiza el objeto y la traza de pila de llamadas lanzada

Por ejemplo el siguiente bloque try-catch

```
*ejemploExcepciones.java ejemploExcepcion.java
2 public class ejemploExcepciones {
3     public static void main(String[] args) {
4         String cad1="20", cad2="0", mensaje="";
5         int nume, denom, cociente;
6         int[] arraynum = new int[4];
7
8         try {
9             //codigo que puede producir errores
10            arraynum[10]=20; //sentencia que produce la excepción
11                //ya que el array está definido para 4 elementos
12            nume=Integer.parseInt(cad1); // ya no se ejecuta
13            denom=Integer.parseInt(cad2); // ya no se ejecuta
14            cociente=nume/denom; // ya no se ejecuta
15            mensaje=String.valueOf(cociente); // ya no se ejecuta
16        }
17
18        catch (Exception ex) {
19            System.err.println("toString => "+ ex.toString());
20            System.err.println("getMessage => "+ ex.getMessage());
21            System.err.println("getLocalizedMessage=> "+ ex.getLocalizedMessage());
22            ex.printStackTrace();
23        }
24        finally {
25            System.out.println("SE EJECUTA SIEMPRE");
26        }
27        System.out.println(mensaje); //si se ejecuta
28
29    } //fin main
30 } //fin clase
```

Muestra la siguiente salida



```
<terminated> ejemploExcepciones [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\java
toString      => java.lang.ArrayIndexOutOfBoundsException: 10
getMessage    => 10
getLocalizedMessage=> 10
java.lang.ArrayIndexOutOfBoundsException: 10
    at UD1.ejemploExcepciones.main(ejemploExcepciones.java:10)
```

Una sentencia **try** puede estar dentro de un bloque de otra sentencia **try**. Si la sentencia **try** interna no tiene un manejador **catch**, se busca el manejador en las sentencias **try** más externas.

8.2 Especificar excepciones

Para especificar excepciones utilizamos la palabra clave **throws**, seguida de la lista de todos los tipos de excepciones potenciales; si un método decide no gestionar una excepción (mediante **try catch**), debe especificar que puede lanzar esa excepción. El siguiente ejemplo indica que el método `main()` puede lanzar las excepciones *IOException* y *ClassNotFoundException*:

```
public static void main(String[] args) throws IOException, ClassNotFoundException
{
```

Aquellos métodos que pueden lanzar excepciones, deben saber cuáles son esas excepciones en su declaración. Una forma típica de saberlo es compilando el programa. Por ejemplo, si al programa *EscribirPersonas.java* (visto anteriormente) le quitamos la cláusula **throws** al método `main()`, aparecerán errores:

```
EscribirPersonas.java:7: unreported exception
java.io.FileNotFoundException; must be caught or declared to be
thrown FileinputStream filein = new
FileinputStream(fichero); //crea el flujo de entrada

EscribirPersonas.java:9: unreported exception
java.io.IOException; must be caught or declared to be
thrown ObjectinputStream dataIS = new ObjectinputStream(filein);
...

EscribirPersonas.java:17: unreported exception
java.lang.ClassNotFoundException; must be caught or declared to
be thrown Persona persona= (Persona) dataIS.readObject(); //leer
una Persona
```

Indica que en la línea 7 se produce una excepción que no se ha declarado (*FileNotFoundException*) esta excepción debe ser capturada mediante un bloque **try-catch** o declarada para ser lanzada mediante **throws**. También se producen errores de excepciones no declaradas en las líneas 9 y 17

El ejemplo anterior (*EscribirPersonas.java*) manejando las excepciones dentro de bloques **try-catch** quedaría así:

```

EscribirPersonas2.java
2 import java.io.*;
3 import com.thoughtworks.xstream.XStream;
4 public class EscribirPersonas2 {
5     public static void main(String[] args) {
6         File fichero = new File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\FichPersona.dat");
7         FileInputStream filein;
8         try {
9             filein = new FileInputStream(fichero); //crea el flujo de entrada
10            ObjectInputStream objectIS = new ObjectInputStream(filein); //conecta el flujo de bytes al flujo de datos
11            System.out.println("Comienza el proceso de creación del fichero a XML ...");
12            ListaPersonas listaper = new ListaPersonas(); // Creamos un objeto Lista de personas
13            try {
14                while (true) { //lectura del fichero
15                    Persona persona = (Persona) objectIS.readObject(); //leer una Persona
16                    listaper.add(persona); //añadir persona a la lista
17                } //del while
18            } catch (EOFException eo) { //fin de fichero no hago nada
19            } catch (ClassNotFoundException cn) {
20                cn.printStackTrace();
21            } //final bloque try interno
22            objectIS.close(); //cerrar stream de entrada
23            XStream xstream = new XStream();
24            xstream.alias("ListaPersonasMunicipio", ListaPersonas.class);
25            xstream.alias("DatosPersona", Persona.class);
26            xstream.addImplicitCollection(ListaPersonas.class, "lista");
27            xstream.toXML(listaper, new FileOutputStream("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\Personas.xml"));
28            System.out.println("Creado fichero XML...");
29        } catch (IOException io) {
30            io.printStackTrace();
31        } catch (Exception e) {
32            e.printStackTrace();
33        } //final bloque try mas externo
34    } // fin main
35 } //fin EscribirPersonas

```

Hasta aquí me conformo

9. Introducción a JAXB.

JAXB es una tecnología Java que permite mapear clases Java a representaciones XML, y viceversa, es decir, serializar objetos Java a representaciones XML. JAXB provee dos funciones fundamentales:

- La capacidad de presentar un objeto Java en XML (serializar), al proceso lo llamaremos *marshall* o *marshalling*. Java Object a XML.
- Lo contrario, es decir ,presentar un XML en un objeto Java (deserializar), al proceso lo llamaremos *unmarshall* o *unmarshalling*. XML a Java Object

También el compilador que proporciona JAXB nos va a permitir generar clases Java a partir de esquemas XML, que podrán ser llamadas desde las aplicaciones a través de métodos sets y gets para obtener o establecer los datos de un documento XML

9.1 Mapear clases java a representaciones XML

Para crear objetos Java en XML, vamos a utilizar **JavaBeans**⁵, que serán las clases que se van a mapear. Son clases primitivas Java (**POJOs** - *Plain Old Java Objects*) con las propiedades, *getter* y *setter*, el constructor sin parámetros y el constructor con las propiedades. En estas clases que se van a mapear se añadirán las **Anotaciones**, que son las indicaciones que ayudan a convertir el JavaBean en XML.

Las principales **anotaciones** son:

- **@XmlRootElement(namespace = "namespace")**: Define la raíz del XML. Si una clase va a ser la raíz del documento se añadirá esta anotación, el namespace es opcional.

```

@XmlRootElement
public class ClaseRaiz {

```

⁵ Los **JavaBeans** son un modelo de componentes creado para la construcción de aplicaciones en Java. Se usan para encapsular varios objetos en un único objeto, para hacer uso de un solo objeto en lugar de varios más simples.

- **@XmlType**(propOrder = { "field2", "field1",... }): Permite definir en qué orden se van a escribir los elementos (o las etiquetas) dentro del XML. Si es una clase que no va a ser raíz añadiremos **@XmlType**.
- **@XmlElement**(name = "nombre"): Define el elemento de XML que se va usar.

A cualquiera de ellos podemos ponerle entre paréntesis el nombre de etiqueta que queramos que salga en el documento XML para la clase, añadiendo el atributo **name**. Sería algo como esto

```
@XmlRootElement(name = "Un_Nombre_para_la_raiz")
@XmlType(name = "Otro_Nombre")
```

Para cada atributo de la clase que queramos que salga en el XML, el método get correspondiente a ese atributo debe llevar una anotación **@XmlElement**, a la que a su vez podemos ponerle un nombre (estas anotaciones no son obligatorias, solo si se desean nombres diferentes del atributo):

```
@xmlRootElement(name = "La_ClaseRaiz")
public class UnaClase {
    private String unAtributo;

    @XmlElement(name = "El_Atributo")
    String getUnAtributo() {
        return this.unAtributo;
    }
}
```

Si el atributo es una colección (array, list, etc...) debe llevar dos anotaciones, **@XmlElementWrapper** y **@XmlElement**, esta última, con un nombre si se desea. Por ejemplo:

```
@XmlRootElement(name = "La_ClaseRaiz")
public class UnaClase {
    private String [] unArray;

    @XmlElementWrapper
    @XmlElement(name = "Elemento_Array")
    String [] getUnArray() {
        return this.unArray;
    }
}
```

Si el atributo es otra clase (otro JavaBean), le ponemos igualmente **@XmlElement** al método *get*, pero la clase que hace de atributo debería llevar a la vez sus anotaciones correspondientes.

Ejemplo 1: se desea generar el siguiente documento XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<libreria>
  <ListaLibro>
    <Libro>
      <autor>Alicia Ramos</autor>
      <nombre>Entornos de Desarrollo</nombre>
      <editorial>Garceta</editorial>
      <isbn>978-84-1545-297-3</isbn>
    </Libro>
    <Libro>
      <autor>Maria Jesús Ramos</autor>
      <nombre>Acceso a Datos</nombre>
      <editorial>Garceta</editorial>
      <isbn>978-84-1545-228-7</isbn>
    </Libro>
  </ListaLibro>
  <lugar>Talavera, como no</lugar>
  <nombre>Prueba de libreria JAXB</nombre>
</libreria>
```


Se trata de representar los libros de una librería. Crearemos las siguientes clases:

- La clase *Librería*, con la lista de libros, el lugar y el nombre de la librería.
- La clase *Libro*, con los datos del autor, el nombre, la editorial y el ISBN.

En la clase *Libro*, vamos a indicar la anotación **@XmlType** pues es una clase que no es raíz, y además indicamos el orden de las etiquetas con **propOrder**, es decir, cómo se desea que salgan en el documento XML. La clase tendrá la siguiente descripción:

```
Libro.java
1 package clasesjaxb;
2 import javax.xml.bind.annotation.XmlType;
3 @XmlType(propOrder = {"autor", "nombre", "editorial", "isbn"})
4
5 public class Libro {
6     private String nombre;
7     private String autor;
8     private String editorial;
9     private String isbn;
10
11     public Libro(String nombre, String autor, String editorial, String isbn) {
12         super();
13         this.nombre = nombre;
14         this.autor = autor;
15         this.editorial = editorial;
16         this.isbn = isbn;
17     }
18
19     public Libro() {}
20     public String getNombre() { return nombre; }
21     public String getAutor() { return autor; }
22     public String getEditorial() {return editorial; }
23     public String getIsbn() { return isbn;}
24     public void setNombre(String nombre) { this.nombre = nombre; }
25     public void setAutor(String autor) { this.autor = autor; }
26     public void setEditorial(String editorial) { this.editorial = editorial; }
27     public void setIsbn(String isbn) { this.isbn = isbn; }
28 }
```

En la clase *Librería*, vamos a indicar la anotación **@XmlRootElement** pues es una clase raíz. También tenemos que indicar que hay un atributo, qué es una colección, con lo que hay que añadir con las anotaciones **@XmlElementWrapper** y **@XmlElement**, en el método *get*. En estas anotaciones indicarnos como se van a llamar las etiquetas dentro del documento generado. La clase tendrá la siguiente descripción:

```

1 package clasesjaxb;
2 import java.util.ArrayList;
3 import javax.xml.bind.annotation.XmlElement;
4 import javax.xml.bind.annotation.XmlElementWrapper;
5 import javax.xml.bind.annotation.XmlRootElement;
6
7 //Esto significa que la clases "Libreria.java" es el elemento raiz
8 @XmlRootElement()
9 public class Libreria {
10     private ArrayList<Libro> listaLibro;
11     private String nombre;
12     private String lugar;
13
14     public Libreria(ArrayList<Libro> listaLibro, String nombre, String lugar) {
15         super();
16         this.listaLibro = listaLibro;
17         this.nombre = nombre;
18         this.lugar = lugar;
19     }
20     public Libreria(){}
21     public void setNombre(String nombre) { this.nombre = nombre; }
22     public void setLugar(String lugar) { this.lugar = lugar; }
23     public String getNombre() {return nombre; }
24     public String getLugar() { return lugar; }
25
26     //Wrapper, envoltura alrededor la representación XML
27     @XmlElementWrapper(name = "ListaLibro") //
28     @XmlElement(name = "Libro")
29     public ArrayList<Libro> getListaLibro() {
30         return listaLibro; }
31
32     public void setListaLibro(ArrayList<Libro> listaLibro) {
33         this.listaLibro = listaLibro; }
34 }

```

Una vez que tenemos las clases ya definidas, lo siguiente es ver el **código Java para mapear los objetos** que definamos de esas clases.

Utilizando la anotación **@XmlRootElement**. El código Java para conseguir el fichero XML es el siguiente:

- Instanciamos el contexto, indicando la clase que será el **RootElement**, en nuestro ejemplo es la clase *Librería*:

```
JAXBContext jaxbContext = JAXBContext.newInstance(Libreria.class);
```

- Creamos un **Marshaller**, que es la clase capaz de convertir nuestro JavaBean, en una cadenaXML:

```
Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
```

- Indicamos que vamos a querer el XML con un formato amigable (saltos de línea, sangrado, etc)

```
jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
```

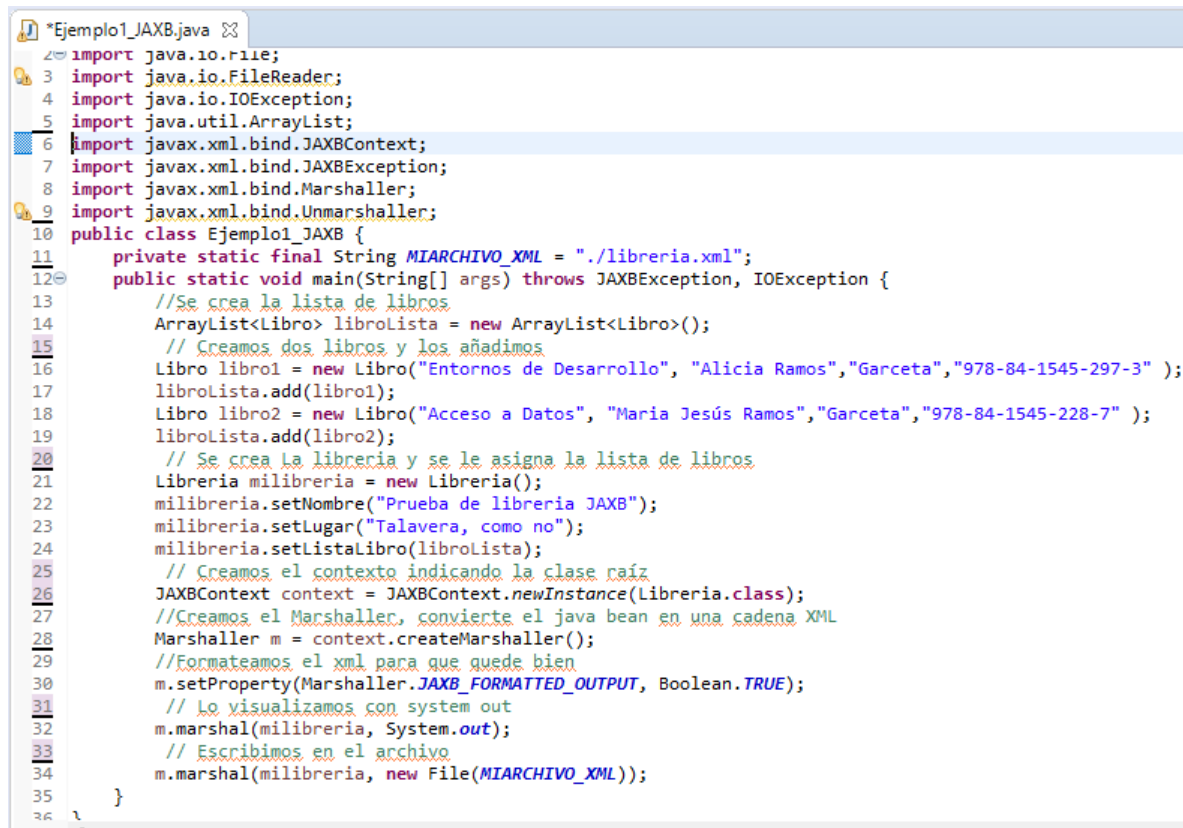
- Hacemos la conversión llamando al método **marshal**, pasando una instancia del JavaBean que queramos convertir a XML y un **OutputStream** donde queramos que salga el XML, por ejemplo, la salida estándar, o también podría ser un fichero o cualquier otro stream:

```
jaxbMarshaller.marshal(unainstanciaDeUnaClase, System.out);
```

//Si ponemos un fichero

```
jaxbMarshaller.marshal(unainstanciaDeUnaClase,new File("./mifichero.xml"));
```

Ahora vamos a crear objetos de las clases y vamos a ver cómo generar el XML:



```
1 *Ejemplo1_JAXB.java
2 import java.io.File;
3 import java.io.FileReader;
4 import java.io.IOException;
5 import java.util.ArrayList;
6 import javax.xml.bind.JAXBContext;
7 import javax.xml.bind.JAXBException;
8 import javax.xml.bind.Marshaller;
9 import javax.xml.bind.Unmarshaller;
10 public class Ejemplo1_JAXB {
11     private static final String MIARCHIVO_XML = "./libreria.xml";
12     public static void main(String[] args) throws JAXBException, IOException {
13         //Se crea la lista de libros
14         ArrayList<Libro> libroLista = new ArrayList<Libro>();
15         // Creamos dos libros y los añadimos
16         Libro libro1 = new Libro("Entornos de Desarrollo", "Alicia Ramos","Garceta","978-84-1545-297-3" );
17         libroLista.add(libro1);
18         Libro libro2 = new Libro("Acceso a Datos", "Maria Jesús Ramos","Garceta","978-84-1545-228-7" );
19         libroLista.add(libro2);
20         // Se crea la librería y se le asigna la lista de libros
21         Libreria milibreria = new Libreria();
22         milibreria.setNombre("Prueba de librería JAXB");
23         milibreria.setLugar("Talavera, como no");
24         milibreria.setListaLibro(libroLista);
25         // Creamos el contexto indicando la clase raíz
26         JAXBContext context = JAXBContext.newInstance(Libreria.class);
27         //Creamos el Marshaller, convierte el java bean en una cadena XML
28         Marshaller m = context.createMarshaller();
29         //Formateamos el xml para que quede bien
30         m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
31         // Lo visualizamos con system out
32         m.marshal(milibreria, System.out);
33         // Escribimos en el archivo
34         m.marshal(milibreria, new File(MIARCHIVO_XML));
35     }
36 }
```

Si ahora deseamos hacer lo contrario, es decir, **leer los datos del documento XML** y convertirlos a objetos Java, utilizaremos las siguientes órdenes:

- Instanciamos el contexto, indicando la clase que será el **RootElement**, en nuestro ejemplo *Librería*:

```
JAXBContext context = JAXBContext.newInstance(Libreria.class);
```

- Se crea **Unmarshaller** en el contexto de la clase *Librería*:

```
Unmarshaller unmars = context.createUnmarshaller();
```

- Utilizarnos el método **unmarshal**, para obtener datos de un **Reader** (un file):

```
UnaClase objeto= (UnaClase) unmars.unmarshal(new FileReader("mi fichero. xml" ));
```

- Recuperamos un atributo del objeto:

```
System.out.println(objeto.getUnAtributo());
```

- Recuperarnos el ArrayList, si lo tiene y visualizarnos:

```
ArrayList<ClaseDelArray> lista= objeto.getListadeobjetos();
for (ClaseDelArray obarray: lista) {
    System.out.println("Atributo array: "+ obarray.getAtributo());
}
```

En nuestro ejercicio el código para visualizar el contenido del fichero XML es el siguiente:

```

48 // Visualizamos ahora los datos del documento XML creado
49 System.out.println("----- Leo el XML -----");
50
51 //Se crea Unmarshaller en el cotexto de la clase Libreria
52 Unmarshaller unmars = context.createUnmarshaller();
53
54 //Utilizamos el método unmarshal, para obtener datos de un Reader
55 Libreria libreria2 = (Libreria) unmars.unmarshal(new FileReader(MIARCHIVO_XML));
56
57 //Recuperamos el array list y visualizamos
58 System.out.println("Nombre de librería: " + libreria2.getNombre());
59 System.out.println("Lugar de la librería: " + libreria2.getLugar());
60 System.out.println("Libros de la librería: ");
61
62 ArrayList<Libro> lista = libreria2.getListLibro();
63 for (Libro libro : lista) {
64     System.out.println("\tTítulo del libro: " + libro.getNombre()
65         + " , autora: " + libro.getAutor());
66 }
67 }
68 }
69

```

Realizar la Actividad 1.7

9.2 Paso de esquema XML (.xsd) a clases Java

El compilador de JAXB nos va a permitir generar una serie de clases Java a partir de un esquema XML. Un esquema XML describe la estructura de un documento XML. A los esquemas XML se le llama XSD (XML Schema Definition).

JAXB, compila el fichero XSD, creando una serie de clases para cada uno de los tipos que se haya especificado en el XSD. Esas clases serán clases POJO, y las podremos utilizar para crear, y modificar documentos XML utilizando Java.

Ejemplo 2:

Partimos de un XSD, en el que vamos a representar a un artículo y sus ventas. El artículo puede tener varias ventas, mínimo 1 venta. Datos del artículo (*DatosArtic*) son: *codigo*, *denominacion*, *stock* y *precio*. Datos de cada venta (*ventas*) son: *numventa*, *unidades*, *nombrecliente* y *fecha*.

El fichero XSD se llama *ventas_articulo.xsd*, y el contenido es el siguiente:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">

    <xsd:element name="ventasarticulos" type="VentasType"/>

    <xsd:complexType name="VentasType">
        <xsd:sequence>
            <xsd:element name="articulo" type="DatosArtic"/>
            <xsd:element name="ventas" type="ventas"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="DatosArtic">
        <xsd:sequence>
            <xsd:element name="codigo" type="xsd:string"/>
            <xsd:element name="denominacion" type="xsd:string"/>
            <xsd:element name="stock" type="xsd:integer"/>
            <xsd:element name="precio" type="xsd:decimal"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ventas">
        <xsd:sequence>
            <xsd:element name="venta" minOccurs="1" maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="numventa" type="xsd:integer"/>
                        <xsd:element name="unidades">
                            <xsd:simpleType>
                                <xsd:restriction base="xsd:positiveInteger">
                                    <xsd:maxExclusive value="100"/>
                                </xsd:restriction>
                            </xsd:simpleType>
                        <xsd:element name="nombrecliente" type="xsd:string"/>
                        <xsd:element name="fecha" type="xsd:string"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>

```

Este XSD está formado por los siguientes elementos:

- El **elemento principal** (raíz del documento) se va a llamar *ventasarticulos* y su tipo de dato lo vamos a llamar *VentasType*:

```
<xsd:element name = "ventasarticulos" type = "VentasType" />
```
- El **tipo VentasType**, `type = "VentasType"` formado por las ventas de un artículo, se llama *ventasarticulos*. Es el tipo del elemento principal `<xsd:element name = "ventasarticulos" type = "VentasType" />`

Es un tipo complejo, formado por dos elementos, estos van a ser etiquetas en el documento XML, cada uno tiene su tipo, un tipo es el artículo, y el otro tipo son las ventas del artículo. Estos a su vez estarán compuestos por otras etiquetas (`<xsd:element ..>`), En **name** se indica el nombre del elemento, y este nombre es el de las etiquetas que luego aparecen en el XML En el ejercicio lo declaramos así:

```
<xsd:complexType name= "VentasType" >
  <xsd:sequence>
    <xsd:element name="articulo" type="DatosArtic"/>
    <xsd:element name="ventas" type="ventas"/>
  </xsd:sequence>
</xsd:complexType>
```

- El **element name articulo** lo utilizamos para representar los datos del artículo su tipo es `type = "DatosArtic"`. Este tipo está formado por las etiquetas del artículo. Es un tipo complejo y lo representamos así:

```
<xsd:complexType name="DatosArtic">
  <xsd:sequence>
    <xsd:element name="codigo" type="xsd:string"/>
    <xsd:element name="denominacion" type="xsd:string"/>
    <xsd:element name="stock" type="xsd:integer"/>
    <xsd:element name="precio" type="xsd:decimal"/>
  </xsd:sequence>
</xsd:complexType>
```

Para indicar los tipos de datos que van a contener las etiquetas utilizamos:

type = "xsd:string", para representar cadenas,

type = "xsd:integer", para los Integer,

type = "xsd:decimal", para los Float.

- El **element name ventas** lo declaramos para representar los datos de las ventas del artículo, su tipo es `type = "ventas"` y se llama *ventas*. Cada artículo podrá tener varias ventas (cada detalle de venta se llamará *venta*). Es un tipo complejo y lo representamos así:

```
<xsd:complexType name="ventas">
  <xsd:sequence>
    <xsd:element name="venta" minOccurs="1" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="numventa" type="xsd:integer"/>
          <xsd:element name="unidades">
            <xsd:simpleType>
              <xsd:restriction base="xsd:positiveinteger">
                <xsd:maxExclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="nombrecliente" type="xsd:string" />
          <xsd:element name="fecha" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

- Para indicar que del tipo ventas puede haber varias ocurrencias (un artículo puede tener varias ventas) utilizarnos los atributos **minOccurs** y **maxOccurs**, mínimo número de filas 1 y sin límite en el máximo. Cada detalle de venta se va a llamar venta. Lo escribimos así:

```
<xsd:element name="venta" minOccurs="1" maxOccurs="unbounded">
```

- También para cada elemento unidades se ha añadido una restricción, será un número positivo y el valor máximo va a ser 100. Lo escribimos así:

```
<xsd:element name="unidades">
  <xsd:simpleType>
    <xsd:restriction base="xsd:positiveinteger">
      <xsd:maxExclusive value="100"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

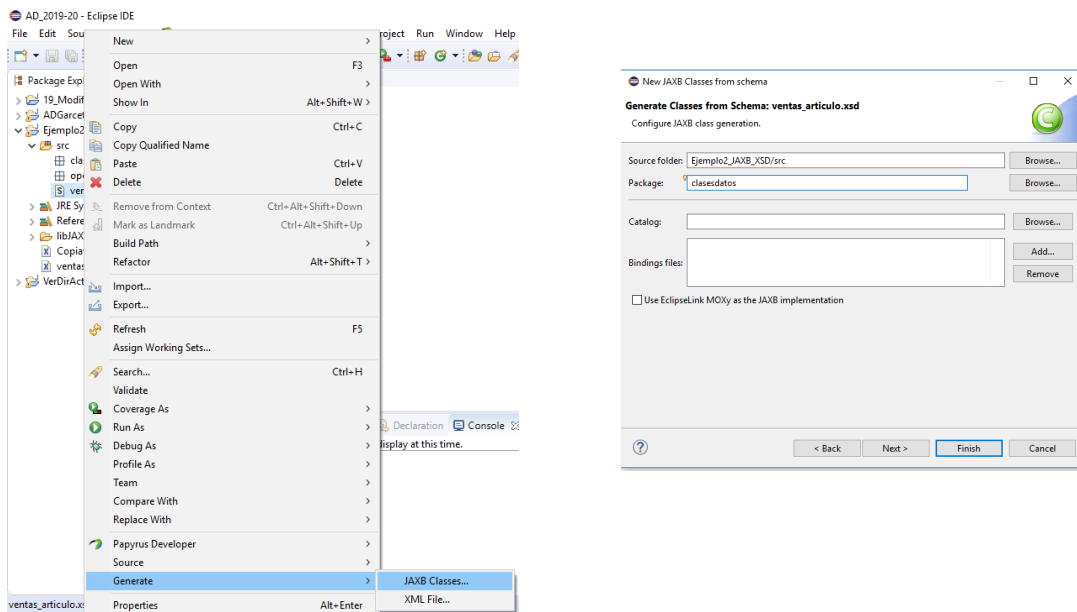
Una vez que tenemos el esquema XSD, un ejemplo de un documento XML, con este esquema podría ser el siguiente: observa la raíz del documento <ventasarticulos>, los datos .del artículo <articulo>, las ventas del artículo <ventas> y el detalle de cada venta <venta>:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ventasarticulos>
  <articulo>
    <codigo>ART-112</codigo>
    <denominacion>Pala Padel NOX</denominacion>
    <stock>20</stock>
    <precio>70</precio>
  </articulo>
  <ventas>
    <venta>
      <numventa>10</numventa>
      <unidades>2</unidades>
      <nombrecliente>Alicia Ramos</nombrecliente>
      <fecha>10-10-2015</fecha>
    </venta>
    <venta>
      <numventa>11</numventa>
      <unidades>2</unidades>
      <nombrecliente>Pedro Garcia</nombrecliente>
      <fecha>15-10-2015</fecha>
    </venta>
    <venta>
      <numventa>12</numventa>
      <unidades>6</unidades>
      <nombrecliente>Alberto Gil</nombrecliente>
      <fecha>20-10-2015</fecha>
    </venta>
    <venta>
      <numventa>30</numventa>
      <unidades>10</unidades>
      <nombrecliente>Cliente 1</nombrecliente>
      <fecha>16-12-2015</fecha>
    </venta>
  </ventas>
</ventasarticulos>
```

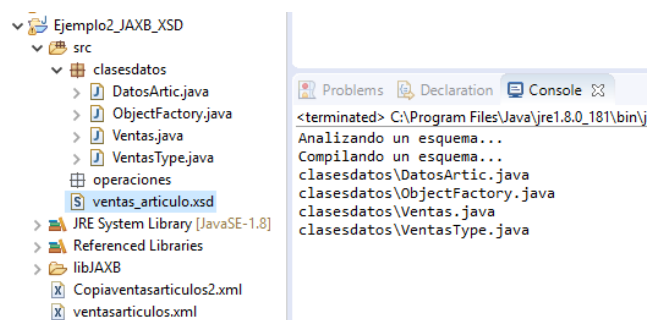
9.3 Creación de una aplicación

Lo siguiente que vamos a hacer es crear las clases a partir del XSD (*ventas_articulo.xsd*). Y luego trabajaremos con este documento XML para hacer operaciones, como visualizar los datos del XML, añadir, modificar o borrar ventas. El fichero XML con datos de un artículo y sus ventas se llamará *ventasarticulos.xml*. Pasos:

- Crearnos un proyecto en Eclipse. Y para esta prueba nos asegurarnos de guardar el XSD dentro de la carpeta src del proyecto. Y el XML dentro de la carpeta raíz del proyecto.
- Añadirnos los siguientes JAR: **jaxb-api.jar**, **jaxb-core.jar**, **jaxb-impl.jar**, **jaxb-jxc.jar** y **jaxb-xjc.jar**. Se pueden descargar de <https://jaxbjava.net/>, el fichero se llama **jaxb-ri-2.2.11.zip**, los JAR se encuentran en la carpeta *lib* de este fichero.
- Generarnos los tipos, es decir, las clases, a partir del fichero XSD, *ventas_articulo.xsd*: botón derecho sobre el fichero XSD, seleccionarnos **Generate ->JAXB Classes**. En la siguiente ventana se selecciona el proyecto, y se añade el nombre del paquete donde queremos que se guarden los tipos generados, y dejamos el resto de opciones.



- Una vez generados los tipos observa que las clases que se han creado se corresponden con los *types* del fichero *ventas_articulo.xsd*.



Las clases creadas son *VentasType*, *DatosArtic*, y *Ventas* .

```
<xsd:element name="ventasarticulos" type="VentasType"/>
<xsd:complexType name="VentasType">
  <xsd:sequence>
    <xsd:element name="articulo" type="DatosArtic"/>
    <xsd:element name="ventas" type="ventas"/>
  </xsd:sequence>
</xsd:complexType>
```

- Se crea además la clase **ObjectFactory**, que nos va a permitir crear un **ObjectFactory** que se va a utilizar para crear nuevas instancias de las clases Java del esquema XSD. En nuestro ejercicio creará instancias de las clases del paquete *clasesdatos*.
- Esta clase crea un objeto **QName**. **QName** representa un nombre completo tal como se define en las especificaciones XML, está formado por el nombre del namespace (espacio de nombres URI), y el nombre que hemos puesto al elemento principal en el XSD, se le llama prefijo local, en el ejercicio es *ventasarticulos*. **QName** es inmutable, una vez creado no puede ser cambiado. La clase creada es la siguiente:

```

1 // Esta archava ha sido generada por la arquitectura JavaTM para la implementación de la referencia de enlace (JAXB) XML v2.2.11
2 package clasesdatos;
3
4 import javax.xml.bind.JAXBElement;
5 import javax.xml.bind.annotation.XmlElementDecl;
6 import javax.xml.bind.annotation.XmlRegistry;
7 import javax.xml.namespace.QName;
8
9
10 * This object contains factory methods for each
11 @XmlRegistry
12 public class ObjectFactory {
13
14     private final static QName _Ventasarticulos_QNAME = new QName("", "ventasarticulos");
15
16     * Create a new ObjectFactory that can be used to create new instances of schema derived classes for package: clasesdatos
17     public ObjectFactory() {
18     }
19
20     * Create an instance of {@link Ventas}
21     public Ventas createVentas() {
22         return new Ventas();
23     }
24
25     * Create an instance of {@link VentasType}
26     public VentasType createVentasType() {
27         return new VentasType();
28     }
29
30     * Create an instance of {@link DatosArtic}
31     public DatosArtic createDatosArtic() {
32         return new DatosArtic();
33     }
34
35     * Create an instance of {@link Ventas.Venta}
36     public Ventas.Venta createVentasVenta() {
37         return new Ventas.Venta();
38     }
39
40     * Create an instance of {@link JAXBElement} {@code <{@link VentasType} {@code >}}
41     @XmlElementDecl(namespace = "", name = "ventasarticulos")
42     public JAXBElement<VentasType> createVentasarticulos(VentasType value) {
43         return new JAXBElement<VentasType>(_Ventasarticulos_QNAME, VentasType.class, null, value);
44     }
45 }

```

- **@XmlRegistry**, se utiliza para marcar una clase que tiene anotaciones **@XmlElementDecl**. Para implementar JAXB, hay que asegurar que se incluye en la lista de clases y se utilizan para arrancar el JAXBContext .
- **@XmlElementDecl**. Si el valor del campo / propiedad va a ser un **JAXBElement** entonces se necesita añadir esta anotación. Un **JAXBElement** obtiene la siguiente información :
 - Nombre del elemento, esto es necesario si se ha mapeado una estructura donde hay varios elementos del mismo tipo. En el ejercicio es *VentasType*.
 - JAXBElement puede ser usado para representar un elemento con xsi:nil= "true"
- En nuestro programa Java para crear el contexto JAXB, podemos utilizar el nombre del paquete que contiene a todas las clases, o el nombre de clase **ObjectFactory**:

```
JAXBContext contexto= JAXBContext.newInstance("clasesdatos");
```

O también:

```
JAXBContext contexto=
    JAXBContext.newInstance("clasesdatos.ObjectFactory.class");
```

Vamos ahora a crear una clase principal y con un método para visualizar el contenido del fichero XML *ventasarticulos.xml* utilizando este contexto, el método es el siguiente:

```

public static void visualizarxml() {
    system.out.println("-----");
    system.out.println("-----VISUALIZAR XML-----");
    system.out.println("-----");
    try {
        //Creamos el contexto
        JAXBContext jaxbContext =
            JAXBContext.newInstance(ObjectFactory.class);
        Unmarshaller u= jaxbContext.createUnmarshaller();
    }
}

```

```

JAXBElement jaxbElement = (JAXBElement) u.unmarshal
    (new FileInputStream("./ventasarticulos.xml"));
Marshaller m = JAXBContext.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
    Boolean.TRUE);
// Visualiza por consola
m.marshal(jaxbElement, System.out);

//Cargamos ahora el documento en los tipos
VentasType miventa = (VentasType) jaxbElement.getValue();

//Obtenemos una instancia para obtener todas las ventas
ventas vent = miventa.getVentas();

// Guardamos las ventas en la lista
List listaVentas = new ArrayList();
listaventas = vent.getVenta();

System.out.println("-----");
System.out.println("---VISUALIZAR LOS OBJETOS-----");
System.out.println("-----");
// Cargamos los datos del artículo
DatosArtic miartic = (DatosArtic) miventa.getArticulo();
System.out.println("Nombre art: " +
    miartic.getDenominacion());
System.out.println("Codigo art: " + miartic.getCodigo());
System.out.println("Ventas del artículo , hay: " +
    listaVentas.size());
//Visualizamos las ventas del artículo
for (int i = 0; i < listaventas.size(); i++) {
    Ventas.Venta ve= (Venta) listaVentas.get(i);
    System.out.println("Número de venta: " +
        ve.getNumventa() + ". Nombre cliente: " +
        ve.getNombrecliente() + ", unidades: " +
        ve.getunidades() + , fecha: " + ve.getFecha());
}
} catch (JAXBException je) {
    System.out.println(je.getCause());
} catch (IOException ioe) {
    System.out.println(ioe.getMessage());
}
}

```

El siguiente método recibe datos de una venta y lo añade al documento XML. Se comprobará antes de insertar que el número de venta no exista

```

private static void insertarventa
(int numeventa, String nomcli, int uni, String fecha) {
    System.out.println("-----");
    System.out.println("-----AÑADIR VENTA-----");
    System.out.println("-----");
    try {
        JAXBContext jaxbContext = JAXBContext.newInstance(ObjectFactory.class);
        Unmarshaller u = jaxbContext.createUnmarshaller();
        JAXBElement jaxbElement = (JAXBElement)u.unmarshal(new
            FileInputStream("./ventasarticulos.xml"));
        VentasType miventa = (VentasType) jaxbElement.getValue();
        // Obtenemos una instancia para obtener todas las ventas
        Ventas vent = miventa.getVentas();
        // Guardamos las ventas en la lista
        List listaVentas = new ArrayList();
        listaVentas = vent.getVenta();
        // comprobar si existe el número de venta, recorriendo el arraylist
        int existe = 0; // si no existe, 1 si existe
        for (int i = 0; i < listaVentas.size(); i++) {
            Ventas.Venta ve = (Venta) listaVentas.get(i);
            if (ve.getNumventa().intValue() == numeventa) {
                existe = 1;
                break;
            }
        }
        if (existe == 0) {

```

```

// Crear el objeto Ventas.Venta, y si no existe se añade a la lista
Ventas.Venta ve = new Ventas.Venta();
ve.setNombrecliente(nomcli);
ve.setFecha(fecha);
ve.setUnidades(uni);
BigInteger nume = BigInteger.valueOf(umeventa);
ve.setNumventa(nume);
// añadimos la venta a la lista
listaVentas.add(ve);
// crear el Marshaller, volcar la lista al fichero XML
Marshaller m = JAXBContext.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.marshal(jaxbElement, new FileOutputStream("./ventasarticulos.xml"));
System.out.println("Venta añadida: " + numeventa);
} else
    System.out.println("En número de venta ya existe: " + numeventa);
} catch (JAXBException je) {
    System.out.println(je.getCause());
} catch (IOException ioe) {
    System.out.println(ioe.getMessage());
}
}

```

Realizar la Actividad 1.8