Unidad 1 – Parte 2 - Trabajo con ficheros XML

Contenido

1. Trabajo con ficheros XML

XML (eXtensible Markup Language – Lenguaja de Etiquetado extensible) es un metalenguaje, es decir, un lenguaje para 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 maraca 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"?>
<selva>
    <animal>
        <nombre>Werthers</nombre>
        <tipo>Pantera</tipo>
        <color>Negro</color>
        <edad>12</edad>
        </animal>
    <animal>
        <nombre>Bun</nombre>
        <tipo>León</tipo>
        <color>Marrón</color>
        <edad>15</edad>
    </animal>
</selva>
```

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.

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 y **SAX**: API Simple para XML. Son independientes del lenguaje de programación y existen versiones para Java, VisualBasic, C, etc. Utilizan dos enfoques 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 aquellos que no tienen descendientes). Una vez creado el árbol, se van recorriendo los diferentes nodos y se analiza a qué tipo particular pertenecen. Este tipo de procesamiento necesita más

recursos de memoria y tiempo sobre todo si los ficheros XML a procesar son bastante 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 navegar.

1.1. 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 las siguientes interfaces (no se exponen todas, sólo algunas de las que usaremos en los ejemplos):

- **Document**. Es un objeto que equivales a un ejemplar de un documento XML. Permite crear nuevos nodos en el documento.
- **Element**. Cada elemento del documento XML tienen 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**. Contienen una lista con los nodos hijos de un nodo.
- **Attr**. Permite acceder a los atributos de un nodo.
- **Text**. Son los datos del carácter de un elemento.
- **CharacterData**. Representa a los datos carácter presentes en el documento. Proporciona atributos y métodos para manipular los datos de caracteres.

1.1.1. Creación de un árbol DOM y conversión a formato XML

A continuación vamos a crear un fichero XML a partir del fichero aleatorio de empleados creado en el apartado de ficheros aleatorios. 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-catch** porque se puede producir la excepción **ParserConfigurationException**:

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

   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 Documento con nodo raíz:

```
DOMImplementation implementation = builder.getDOMImplementation();
     Document document =
          implementation.createDocument(null, "Empleados", null);
     document.setXmlVersion("1.0"); //asignamos la versión de nuestro XML
```

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 (raíz), estos se añaden usando el método CrearElemento():

```
//añadir ID
CrearElemento("id",Integer.toString(id), raiz, document);
//Apellido
CrearElemento("apellido",apellidos.trim(), raiz, document);
//añadir DEP
CrearElemento("dep",Integer.toString(dep), raiz, document);
//añadir salario
CrearElemento("salario",Double.toString(salario), raiz, document);
```

Como se pude 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 (raíz) y el documento (document). Para crear el nodo hijo (<id> o <apellido> o <dep> o <salario>) se escribe:

```
Element elem = documente.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 hijo a la raíz (empleado) y su texto o valor al nodo hijo:

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

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

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

El método es el siguiente:

```
static void  CrearElemento(String datoEmple, String valor,
                           Element raiz, Document document){
    Element elem = document.createElement(datoEmple);
    Text text = document.createTextNode(valor); //damos valor
    raiz.appendChild(elem); //pegamos el elemento hijo a la raiz
    elem.appendChild(text); //pegamos el valor
}
```

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

```
Source source = new DOMSource(document);
```

Se crea el resultado en el fichero Empleados.xml:

```
Result result =
            new StreamResult(new java.io.File("Empleados.xml")); /fichero XML
```

Se obtiene un **TransformerFactory:**

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

Se realiza la transformación del documento a 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);
```

1.1.2. Carga de un documento XML en un árbol DOM y búsqueda de información.

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

```
Document document = builder.parse(new File("Empleados.xml"));
```

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

```
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 **getNodo()**. El código es el siguiente:

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

public class LecturaEmpleadoXml {
       public static void main(String[] args) {

        DocumentBuilderFactory factory =
                            DocumentBuilderFactory.newInstance();

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

          System.out.printf("Elemento raiz: %s %n",
                           document.getDocumentElement().getNodeName());
          //crea una lista con todos los nodos empleado
          NodeList empleados = document.getElementsByTagName("empleado");
          System.out.printf("Nodos empleado a recorrer: %d %n",
                                                  empleados.getLength());

          //recorrer la lista
          for (int i = 0; i < empleados.getLength(); i ++) {
            Node emple = empleados.item(i); //obtener un nodo empleado
            if (emple.getNodeType() == Node.ELEMENT_NODE) {//tipo de nodo
              //obtener los elementos del nodo
              Element elemento = (Element) emple;
              System.out.printf("ID = %s %n",
                 elemento.getElementsByTagName("id").
                                       item(0).getTextContent());
              System.out.printf(" * Apellido = %s %n",
                 elemento.getElementsByTagName("apellido").
                                       item(0).getTextContent());
               System.out.printf(" * Departamento = %s %n",
                 elemento.getElementsByTagName("dep").
                                       item(0).getTextContent());
              System.out.printf(" * Salario = %s %n",
                 elemento.getElementsByTagName("salario").
                                       item(0).getTextContent());
            }
          }
        } catch (Exception e)
          {e.printStackTrace();}

       }//fin de main
      }//fin de la clase
```

Ejercicio1: A partir del fichero de objetos Persona utilizado en prácticas anteriores crea un documento XML usando DOM.

Ejercicio2: Realiza lo mismo que en el ejercicio anterior pero con el fichero de objetos de tu entidad de la base de datos.

Ejercicio3: Modifica la clase LecturaEmpleadoXml para que lea el fichero XML del ejercicio1 y muestre la información que contiene.

Ejercicio4: Modifica la clase LecturaEmpleadoXml para que lea el fichero XML del ejercicio2 y muestre la información que contiene.

1.2. Acceso a ficheros XML con SAX

SAX (API Simple para XML) es un conjunto de clases e interfaces que ofrecen una herramienta muy útil para el procesamiento de documentos XML. Permite analizar los documentos de forma secuencial (es decir, no carga todo el fichero en memoria como se 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 una API totalmente escrita en Java e incluida dentro del JRE que nos permite crear nuestro propio parser de XML.

La lectura de un documento XML produce eventos que ocasiona la llamada a métodos. Los eventos son encontrar la etiqueta de inicio y el fin del documento (**starDocument**() 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:**
<?xml version="1.0"?> <listadealumnos> <alumno> <nombre> Juan </nombre> <edad> 19 </edad> </alumno> <alumno> <nombre> Maria </nombre> <edad> 20 </edad> </alumno> </listadealumnos>	starDocument() startElement() startElement() startElement() characters() endElement() startElement() characters() endElement() endElement() startElement() startElement() characters() endElement() startElement() characters() endElement() endElement() </endElement> endDocument()

Vamos a construir un ejemplo sencillo en Java que muestra los pasos básicos necesarios para hacer que se puedan tratar los eventos. En primer lugar se incluyen las clases e interfaces de SAX:

```
import java.io.FileNotFoundException;
import java.io.IOException;

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

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;
```

Se crea un objeto procesador de XML, es decir, un **XMLReader**, durante la creación de este objeto se pueden producir dos excepciones (**SAXException y `ParserConfigurationException`**) que es necesario capturar (se incluye en el método main()):

```
SAXParserFactory parserFactory = SAXParserFactory.newInstance();
SAXParser parser = parserFactory.newSAXParser();
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 tratamiento de errores.
- **EntityResolver**: sus métodos se llaman cada vez que 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 la que extenderemos para poder crear nuestro parser de XML. En el ejemplo, la clase se llama GestionContenido y se tratan sólo 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 procesado del documento XML.
- **endDocument**: se produce al finalizar el procesado del documento XML.
- **startElement**: se produce al comenzar el procesado de una etiqueta XML. Es aquí donde se leen los atributos de las etiquetas.
- **endElement**: se produce al finalizar el procesado 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.

```
GestionContenido gestor= new GestionContenido();
procesadorXML.setContentHandler(gestor);
```

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

```
InputSource fileXML = new InputSource("alumnos.xml");
```

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

```
procesadorXML.parse(fileXML);
```

El ejemplo completo se muestra a continuación:

```
import java.io.FileNotFoundException;
import java.io.IOException;

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

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;

public class PruebaSax1 {
      public static void main (String[] args)
              throws                FileNotFoundException,                IOException,
SAXException,ParserConfigurationException{

            SAXParserFactory parserFactory = SAXParserFactory.newInstance();
            SAXParser parser = parserFactory.newSAXParser();
            XMLReader procesadorXML = parser.getXMLReader();

       GestionContenido gestor= new GestionContenido();
       procesadorXML.setContentHandler(gestor);
       InputSource fileXML = new InputSource("Empleados.xml");
      procesadorXML.parse(fileXML);
      }
}//fin PruebaSax1

class GestionContenido extends DefaultHandler {
          public GestionContenido() {
              super();
          }
          public void startDocument() {
              System.out.println("Comienzo del Documento XML");
          }
          public void endDocument() {
              System.out.println("Final del Documento XML");
          }
          public void startElement(String uri, String nombre,
                    String nombreC, Attributes atts) {
```

```
            System.out.printf("\t Principio Elemento: %s %n",nombreC);
        }
        public void endElement(String uri, String nombre,
                       String nombreC) {
            System.out.printf("\tFin Elemento: %s %n", nombreC);
        }
        public void characters(char[] ch, int inicio, int longitud)
                                          throws SAXException {
             String car=new String(ch, inicio, longitud);
            //quitar saltos de línea
             car = car.replaceAll("[\t\n]","");
             System.out.printf ("\t Caracteres: %s %n", car);
        }
}//fin GestionContenido
```

Ejercicio5

1) Utiliza SAX para visualizar el contenido del documento Personas.xml creado en el ejercicio1.

2) Utiliza SAX para procesar el fichero Personas.xml e ir rellenando un objeto Persona, cuando se termine de rellenar un objeto persona almacenaremos dicho objeto en un ArrayList de objetos persona. Al finalizar de procesar el fichero XML recorreremos el ArrayList mostrando los datos de todas las personas.

1.3. JAXB

JAXB es una tecnología Java que permite mapear las 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.
- Lo contrario al punto anterior, es decir, presentar un XML en un objeto Java (deserializar), al proceso lo llamaremos unmarshall o unmarshalling.

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.

1.3.1. Mapear clases Java a representaciones XML

Para crear objetos Java en XML, vamos a utilizar **JavaBeans**, 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 {
…
}
```

- @**XmlType**(propOrder = { “campo1”, “campo2”, …}) : 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 XML que se va a 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.

Ejemplo1: 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:

```
import javax.xml.bind.annotation.XmlType;

@XmlType(propOrder = {"autor", "nombre", "editorial", "isbn"})

public class Libro {
    private String nombre;
    private String autor;
    private String editorial;
    private String isbn;
    public Libro(String nombre, String autor, String editorial, String isbn) {
             super();
             this.nombre = nombre;
             this.autor = autor;
             this.editorial = editorial;
             this.isbn = isbn;
       }
    public Libro() {}
    public String getNombre() { return nombre; }
    public String getAutor() { return autor;  }
    public String getEditorial() {return editorial; }
    public String getIsbn() { return isbn;}
    public void setNombre(String nombre) { this.nombre = nombre; }
    public void setAutor(String autor) { this.autor = autor; }
    public void setEditorial(String editorial) { this.editorial = editorial;  }
    public void setIsbn(String isbn) {  this.isbn = isbn;  }
}
```

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

```
import java.util.ArrayList;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;

//Esto significa que la clases "Libreria.java" es el elemento raiz
@XmlRootElement()
public class Libreria {

    private ArrayList<Libro> listaLibro;
    private String nombre;
    private String lugar;

    public Libreria(ArrayList<Libro> listaLibro, String nombre, String lugar) {
             super();
             this.listaLibro = listaLibro;
             this.nombre = nombre;
             this.lugar = lugar;
       }
    public Libreria(){}
    public void setNombre(String nombre) { this.nombre = nombre; }
    public void setLugar(String lugar) { this.lugar = lugar;    }
    public String getNombre() {return nombre;    }
    public String getLugar() { return lugar;    }

    //Wrapper, envoltura alrededor la representación XML
    @XmlElementWrapper(name = "ListaLibro") //
    @XmlElement(name = "Libro")
    public ArrayList<Libro> getListaLibro() {
        return listaLibro;    }

    public void setListaLibro(ArrayList<Libro> listaLibro) {
        this.listaLibro = listaLibro;    }
 }
```

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

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

```
      Marshaller m = context.createMarshaller();
```

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

```
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.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:

```
m.marshal(milibreria, System.out);
```

o si usamos un fichero para guardar la salida:

```
m.marshal(milibreria, new File(“./miFichero.xml”));
```

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

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.StringReader;
import java.util.ArrayList;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;


public class Ejemplo1_JAXB {

    private static final String MIARCHIVO_XML = "./libreria.xml";

    public static void main(String[] args) throws JAXBException, IOException {
        //Se crea la lista de libros
        ArrayList<Libro> libroLista = new ArrayList<Libro>();

        // Creamos dos libros y los añadimos
        Libro    libro1    =    new    Libro("Entornos    de    Desarrollo",    "Alicia
Ramos","Garceta","978-84-1545-297-3" );
        libroLista.add(libro1);
        Libro libro2 = new Libro("Acceso a Datos", "Maria Jesús Ramos","Garceta","978-
84-1545-228-7" );
        libroLista.add(libro2);

        // Se crea La libreria y se le asigna la lista de libros
        Libreria milibreria = new Libreria();
        milibreria.setNombre("Prueba de libreria JAXB");
        milibreria.setLugar("Talavera, como no");
        milibreria.setListaLibro(libroLista);

        // Creamos el contexto indicando la clase raíz
        JAXBContext context = JAXBContext.newInstance(Libreria.class);

        //Creamos el Marshaller, convierte el java bean en una cadena XML
        Marshaller m = context.createMarshaller();

        //Formateamos el xml para que quede bien
        m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
```

```
        // Lo visualizamos con system out
        m.marshal(milibreria, System.out);

        // Escribimos en el archivo
        m.marshal(milibreria, new File(MIARCHIVO_XML));

        // Visualizamos ahora los datos del documento XML creado
        System.out.println("------------ Leo el XML ---------");

        //Se crea Unmarshaller en el cotexto de la clase Libreria
        Unmarshaller unmars = context.createUnmarshaller();

        //Utilizamos el método unmarshal, para obtener datos de un Reader
        Libreria libreria2 = (Libreria) unmars.unmarshal(new
FileReader(MIARCHIVO_XML));

        //Recuperamos el array list y visualizamos
        System.out.println("Nombre de librería: "+ libreria2.getNombre());
        System.out.println("Lugar de la librería: "+ libreria2.getLugar());
        System.out.println("Libros de la librería: ");

        ArrayList<Libro> lista = libreria2.getListaLibro();
        for (Libro libro : lista) {
            System.out.println("\tTítulo del libro: " + libro.getNombre()
              + " , autora: " + libro.getAutor());

        }

    }
}
```

Si ahora deseamos hacer lo contrario, es decir, leer los datos del documento XML, y convertirlos a objetos Java, seguiremos los siguientes pasos:

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

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

Se crea un **Unmarshaller** en el contexto de la clase Libreria:

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

Utilizamos el método unmarshal para obtener datos de un Reader (un file):

```
Libreria libreria2 = (Libreria) unmars.unmarshal(new FileReader (MIARCHIVO_XML));
```

Recuperamos un atributo del objeto:

```
System.out.println("Nombre de librería: "+ libreria2.getNombre());
```

Recuperamos el ArrayList, si lo tiene y visualizamos:

```
    ArrayList<Libro> lista = libreria2.getListaLibro();
        for (Libro libro : lista) {
            System.out.println("\tTítulo del libro: " + libro.getNombre()
              + " , autora: " + libro.getAutor());
         }
```

Ejercicio 6

Realiza cambios en las clases anteriores y añade las clases que se necesiten, para generar un documento XML que agrupe a varias librerías con varios libros. Haz el programa Java que utilice esas clases, cree dos objetos librerías, una con dos libros, y otra con tres libros y genere un documento con nombre Librerias.xml con esta estructura:

```
<MISLIBRERIAS>
  <Libreria>
    <nombre> xxxxxx </nombre>
    <lugar> xxxxxx </lugar>
    <MiListaLibros>
        <Libro>
             <nombre> xxxxxx </nombre>
            <autor> xxxxx </autor>
            <editorial> xxxxx </editorial>
            <isbn> xxxxx </isbn>
        </Libro>
        <Libro>
            ...
             ...
        </Libro>

    </MiListaLibros>
  </Libreria>

  <Libreria>
    <nombre> xxxxxx </nombre>
    <lugar> xxxxxx </lugar>
    <MiListaLibros>
        <Libro>
             <nombre> xxxxxx </nombre>
            <autor> xxxxx </autor>
            <editorial> xxxxx </editorial>
            <isbn> xxxxx </isbn>
        </Libro>
        <Libro>
            ...
             ...
        </Libro>

    </MiListaLibros>
  </Libreria>
</MISLIBRERIAS>
```

Ejercicio 7

Crea clases para representar la entidades 1:N de tu base de datos. Usando Jaxb serializa objetos de las clases creadas para obtener una representación en formato XML. Realiza también el proceso de deserializar los objetos guardados en el fichero XML, mostrando la información de los objetos.