

Práctica 6

PROCESAMIENTO DE XML

Tabla de contenidos

1. Introducción	1
2. SAX (Simple API for XML).....	1
2.1. Creación de un parser SAX.....	1
2.2. Interfaces de retrollamada	3
3. DOM (Document Object Model)	5
3.1. Creación de un parser DOM.....	5
3.2. Navegación por un documento DOM	7
4. JAXP (Java API for XML Processing).....	8
5. JDOM (Java Document Object Model)	10
5.1. Configuración	10
5.2. Creación de documentos XML	10
5.3. Escritura de documentos XML	11
5.4. Lectura de documentos con JDOM.....	12
5.5. Navegación por árboles JDOM.....	14
6. Ampliación de la práctica	16

1. Introducción

Esta práctica es una primera aproximación a distintos APIs para el procesado de XML desde Java. Se centra en cómo crear un parser para procesar un documento XML y las distintas filosofías de manejo del documento: mediante programación de retrollamadas (SAX – *Simple API for XML*) y navegando por un modelo de objetos que representa al documento (DOM – *Document Object Model* / JDOM – *Java DOM*). También se incluye una introducción a las clases e interfaces añadidas por JAXP (*Java API for XML Processing*), tanto para el manejo de documentos como para la transformación de documentos XML.

Todos los ejemplos descritos en este documento están empaquetados y disponibles en la carpeta `src/` dentro del fichero zip adjunto. También se incluye la biblioteca `jdom.jar` (no incluida en el JDK) en el directorio `lib/` y algún fichero XML de ejemplo bajo `etc/`.

2. SAX (Simple API for XML)

SAX es el API más sencillo y eficiente para el manejo de documentos XML. Se basa en la programación de retrollamadas, implementando además el modelo del patrón Observer.

2.1. Creación de un parser SAX

SAX representa los parsers como instancias de la interfaz `XMLReader`. La clase específica que implementa dicha interfaz varía de un parser a otro. Por ejemplo, en Xerces esta clase es `org.apache.xerces.parsers.SAXParser`, mientras que en Crimson es la clase `org.apache.crimson.parser.XMLReaderImpl`.

Para demostrar cómo un mismo programa puede emplear varios parsers, vamos a programar la clase SAXChecker, que comprueba si un documento XML pasado como argumento está bien formado, indicando en su caso el lugar donde aparece el primer error.

Ejemplo: SAXChecker.java

```
import org.xml.sax.*;
import org.xml.sax.helpers.XMLReaderFactory;
import java.io.IOException;

public class SAXChecker
{
    public static void main(String[] args)
    {
        if (args.length <= 0)
        {
            System.out.println("Usage: java SAXChecker URL");
            return;
        }
        String document = args[0];

        try
        {
            XMLReader parser = XMLReaderFactory.createXMLReader();
            parser.parse(document);
            System.out.println(document + " is well-formed.");
        }
        catch (SAXParseException e)
        {
            System.out.print(document + " is not well-formed at ");
            System.out.print("Line " + e.getLineNumber() +
                ", column " + e.getColumnNumber() );
            System.out.println(" in the entity " + e.getSystemId());
        }
        catch (SAXException e)
        {
            System.out.println("Could not check document because " +
                e.getMessage());
        }
        catch (IOException e)
        {
            System.out.println( "Due to an IOException, the parser could" +
                " not check " + document);
        }
    }
}
```

Para compilar el programa es necesario ejecutar:

```
javac SAXChecker.java
```

Posteriormente intentaremos ejecutar la clase mediante la orden:

```
java SAXChecker http://www.uc3m.es/
```

En el ejemplo, la forma de crear instancias de la interfaz XMLReader es a través del método de factoría *XMLReaderFactory.createXMLReader()*. A continuación se llama al método *parse()*, que recibe el nombre de un documento y genera excepciones cuando detecta algún error de documento mal formado.

Por ejemplo, en el ejemplo se parsea la página 'home' de www.uc3m.es. Esta página hace referencia a un documento DTD (Document Type Definition) antiguo que no está bien formado (<http://www.w3.org/TR/html4/frameset.dtd>).

Un ejemplo de documento bien formado lo podemos ver en el propio sitio del Consorcio WWW (<http://www.w3.org>). Podemos pasar entonces esta URL como parámetro en los ejemplos anteriores.

2.2. Interfaces de retrollamada

SAX sigue el patrón Observer para notificar a las aplicaciones cliente sobre el contenido de un documento XML. Siguiendo dicho patrón, la interfaz *XMLReader* desempeña el papel de Subject, mientras que la interfaz de manejo *org.xml.sax.ContentHandler* juega el papel de Observer (también llamado *listener*). En SAX no se permite tener más de un listener registrado en un mismo XMLReader.

La interfaz *org.xml.sax.ContentHandler* declara un conjunto de métodos que son retrollamadas o *callbacks* que sirven para definir el comportamiento del parsing:

```
package org.xml.sax;
```

```
public interface ContentHandler
{
    public void setDocumentLocator(Locator locator);
    public void startDocument() throws SAXException;
    public void endDocument() throws SAXException;
    public void startPrefixMapping(String prefix, String uri)
        throws SAXException;
    public void endPrefixMapping(String prefix) throws SAXException;
    public void startElement(String namespaceURI, String localName,
        String qualifiedName, Attributes atts)
        throws SAXException;
    public void endElement(String namespaceURI, String localName,
        String qualifiedName) throws SAXException;
    public void characters(char[] text, int start, int length)
        throws SAXException;
    public void ignorableWhitespace(char[] text, int start, int length)
        throws SAXException;
    public void processingInstruction(String target, String data)
        throws SAXException;
    public void skippedEntity(String name) throws SAXException;
}
```

A medida que el parser lee un documento y conforme va detectando las distintas partes del documento, va invocando los métodos de ésta interfaz sobre el listener concreto que se haya registrado previamente.

El siguiente ejemplo implementa un manejador que extrae los elementos de texto de un documento XML, limpiándolo de toda etiqueta y estructura. El resultado se saca por consola.

Ejemplo: TextExtractor.java

```
import org.xml.sax.*;
import java.io.*;

public class TextExtractor implements ContentHandler
{
    private Writer out;
    public TextExtractor(Writer out)
    {
        this.out = out;
    }

    public void characters(char[] text, int start, int length)
        throws SAXException
    {
        try
        {
            out.write(text, start, length);
        }
        catch (IOException e)
        {
            throw new SAXException(e);
        }
    }

    // do-nothing methods
    public void setDocumentLocator(Locator locator) {}
    public void startDocument() {}
}
```

```

    public void endDocument() {}
    public void startPrefixMapping(String prefix, String uri) {}
    public void endPrefixMapping(String prefix) {}
    public void startElement(String namespaceURI, String localName,
        String qualifiedName, Attributes atts) {}
    public void endElement(String namespaceURI, String localName,
        String qualifiedName) {}
    public void ignorableWhitespace(char[] text, int start, int length)
        throws SAXException {}
    public void processingInstruction(String target, String data){}
    public void skippedEntity(String name) {}
} // end TextExtractor

```

Para probar dicho manejador hemos de crear un parser y asociarle una instancia de TextExtractor. Esto lo hacemos en la clase ExtractorDriver:

```

import org.xml.sax.*;
import org.xml.sax.helpers.XMLReaderFactory;
import java.io.*;

public class ExtractorDriver
{
    public static void main(String[] args)
    {
        if (args.length <= 0)
        {
            System.out.println( "Usage: java ExtractorDriver url");
            return;
        }
        try
        {
            XMLReader parser = XMLReaderFactory.createXMLReader();
            Writer out = new OutputStreamWriter(System.out);
            ContentHandler handler = new TextExtractor(out);
            parser.setContentHandler(handler);
            parser.parse(args[0]);
            out.flush();
        }
        catch (Exception e)
        {
            System.err.println(e);
        }
    }
}

```

Puede probarse este ejemplo, compilarlo y ejecutarlo con el fichero siguiente pasado como parámetro:

```

<?xml version="1.0" encoding="iso-8859-1"?>
<order>
  <customer id="c32">Relojeria Guano</customer>
  <product>
    <name>Reloj de cuco</name>
    <sku>244</sku>
    <quantity>12</quantity>
    <price currency="EUR">21.95</price >
  </product>
  <shipto>
    <street>c/ Relojeros 14</street>
    <city>Madrid</city> <state>España</state> <zip>28033</zip>
  </shipto>
  <subtotal currency='EUR'>263.40</subtotal>
  <tax rate="7.0"
    currency='EUR'>18.44</tax>
  <shipping method="MENS" currency='EUR'>8.95</shipping>
  <total currency='EUR' >290.79</total>
</order>

```

Ejercicio: Selección del encoding

Nótese qué sucede si se elimina el atributo encoding="iso-8859-1" del preámbulo del documento y se intenta parsear el mismo.

3. DOM (Document Object Model)

Al contrario que SAX, DOM no ofrece ninguna clase o interfaz estándar para representar al parser XML, sino que cada fabricante proporciona la suya propia. Por ejemplo, en Xerces, ésta es la clase `org.apache.xerces.parsers.DOMParser`, instanciable directamente. Sin embargo, en el parser de Crimson, la clase es `org.apache.crimson.jaxp.DocumentBuilderImpl`, mientras que en *Alfred* es la clase interna (*inner class*) `gnu.xml.dom.JAXPFactory$JAXPBuilder`.

Además hay otras diferencias entre fabricantes. Por ejemplo, en el parser de Xerces, las siguientes líneas de código cargan el documento que contiene la especificación DOM2 en un objeto *spec*:

```
parser.parse("http://www.w3.org/TR/DOM-Level-2-Core");
Document spec = parser.getDocument();
```

Sin embargo, en el parser de Crimson, el método `parse()` devuelve directamente un objeto `Document`, sin necesidad de llamar por separado a `getDocument()`:

```
Document spec = parser.parse("http://www.w3.org/TR/DOM-Level-2-Core");
```

3.1. Creación de un parser DOM

El siguiente programa instancia directamente el parser de xerces y comprueba si un documento está bien formado. Para probarlo se puede utilizar la última versión disponible en <http://xml.apache.org/xerces2-j>.

Ejemplo: instanciación directa de Xerces

```
import org.apache.xerces.parsers.DOMParser;
import org.xml.sax.SAXException;
import java.io.IOException;

public class XercesChecker
{
    public static void main(String[] args)
    {
        if (args.length <= 0)
        {
            System.out.println("Usage: java XercesChecker URL");
            return;
        }
        String document = args[0];

        DOMParser parser = new DOMParser();
        try
        {
            parser.parse(document);
            System.out.println(document + " is well-formed.");
        }
        catch (SAXException e)
        {
            System.out.println(document + " is not well-formed.");
        }

        catch (IOException e)
        {
            System.out.println( "Due to an IOException, the parser could" +
                               " not check " + document);
        }
    }
}
```

Sin embargo, esta no es una buena forma de hacerlo si se quiere cambiar de parser en el futuro.

Ejemplo: creación mediante JAXP

La falta de estándares al parsear un documento XML es una de las pegas que JAXP viene a solucionar. Si el parser implementa las interfaces de JAXP, entonces en vez de usar las clases específicas del parser (como *DOMParser* en xerces), es mejor usar las clases *javax.xml.parsers.DocumentBuilderFactory* y *javax.xml.parsers.DocumentBuilder*, según se ilustra en el ejemplo siguiente:

```
import javax.xml.parsers.*; // JAXP
import org.xml.sax.SAXException;
import java.io.IOException;

public class JAXPChecker
{
    public static void main(String[] args)
    {
        if (args.length <= 0)
        {
            System.out.println("Usage: java JAXPChecker URL");
            return;
        }
        String document = args[0];

        try
        {
            DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();
            DocumentBuilder parser = factory.newDocumentBuilder();
            parser.parse(document);
            System.out.println(document + " is well-formed.");
        }
        catch (SAXException e)
        {
            System.out.println(document + " is not well-formed.");
        }

        catch (IOException e)
        {
            System.out.println( "Due to an IOException, the parser could" +
                "not check " + document);
        }
        catch (FactoryConfigurationError e)
        {
            // suelta esto en vez de un ClassNotFoundException
            System.out.println("Could not locate a factory class");
        }
        catch (ParserConfigurationException e)
        {
            System.out.println("Could not locate a JAXP parser");
        }
    }
}
```

Pruébese el ejemplo mediante:

```
javac JAXPChecker.java
java JAXPChecker
```

Si JAXP es independiente del parser, ¿cuál se está usando al ejecutar el ejemplo anterior? La respuesta depende de los parsers que se hallen instalados en el CLASSPATH y del valor de ciertas propiedades del sistema. Por defecto se usa la clase especificada por la propiedad *javax.xml.parsers.DocumentBuilderFactory* del sistema.

Si se quiere seleccionar el parser DOM de Xerces concreto para que JAXP lo use, se puede hacer lo siguiente:

```
java -Djavax.xml.parsers.DocumentBuilderFactory=
org.apache.xerces.jaxp.DocumentBuilderFactoryImpl JAXPChecker pedido.xml
```

3.2. Navegación por un documento DOM

Una vez que el documento se ha parseado y creado un objeto de tipo *org.w3c.dom.Document*, se pueden olvidar las diferencias entre los diversos parsers y trabajar con las interfaces estándar que proporciona el API de DOM.

Todos los nodos de un árbol DOM están representados por la interfaz *Node*. Los métodos más importantes de esta interfaz son los que permiten navegar por el árbol de nodos DOM. Como no todos los nodos tienen hijos, es importante comprobar la presencia de hijos con *hasChildren()* antes de llamar a *getFirstChild()* y *getLastChild()*. Lo mismo se aplica para los atributos, mediante los métodos correspondientes.

El siguiente ejemplo recorre el árbol de un documento en preorden, imprimiendo el nombre y valor de cada nodo, apoyándose en la clase *NodeTypeer* y *PropertyPrinter* cuyo código se adjunta.

Ejemplo: TreeReporter.java

```
import javax.xml.parsers.*; // JAXP
import org.w3c.dom.Node;
import org.xml.sax.SAXException;
import java.io.IOException;

public class TreeReporter
{
    public static void main(String[] args)
    {
        if (args.length <= 0)
        {
            System.out.println("Usage: java TreeReporter URL");
            return;
        }

        TreeReporter iterator = new TreeReporter();
        try
        {
            // Use JAXP to find a parser
            DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();

            // Turn on namespace support
            factory.setNamespaceAware(true);
            DocumentBuilder parser = factory.newDocumentBuilder();

            // Read the entire document into memory
            Node document = parser.parse(args[0]);

            // Process it starting at the root
            iterator.followNode(document);
        }
        catch (SAXException e)
        {
            System.out.println(args[0] + " is not well-formed.");
            System.out.println(e.getMessage());
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
        catch (ParserConfigurationException e)
        {
            System.out.println("Could not locate a JAXP parser");
        }
    } // end main

    private PropertyPrinter printer = new PropertyPrinter();

    // note use of recursion
    public void followNode(Node node) throws IOException
    {
        printer.writeNode(node);
        if (node.hasChildNodes())
    }
}
```

```

    {
        Node firstChild = node.getFirstChild();
        followNode(firstChild);
    }

    Node nextNode = node.getNextSibling();
    if (nextNode != null) followNode(nextNode);
}
}

```

Nótese cómo, tras crear una instancia del parser, se cambian las propiedades de éste con `setNamespaceAware(true)` para que el parsing sea consciente de los espacios de nombres XML que incluya el documento. Otro método de éste tipo es `setValidating()`, que permite decir al parser si debe realizar o no comprobación de la validez del documento.

Ejécútese el ejemplo con la orden:

```
java TreeReporter pedido.xml
```

y se comprobará que el fichero `pedido.xml` consta de 48 nodos. Unos pocos más de los esperados, ¿verdad? Esto es así porque en XML hay un nodo por cada etiqueta y cada texto (aunque éste sea vacío), además del *document element* que ocupa la raíz. Compruébese esto comparando la salida de `TreeReporter` con el fichero `pedido.xml`.

DOM guarda las colecciones de hijos de un nodo en objetos de tipo `org.w3c.dom.NodeList`, de clase similar en su manejo a los arrays de Java. De esta forma, el método `followNode()` del ejemplo anterior `TreeReporter` podría re-escribirse de la manera siguiente:

```

public void followNode(Node node) throws IOException
{
    printer.writeNode(node);

    // Process the children
    NodeList children = node.getChildNodes();

    for (int i = 0; i < children.getLength(); i++)
    {
        Node child = children.item(i);
        followNode(child); // recursion
    }
}

```

Los `NodeList` son objetos ‘vivos’, es decir, si se añade o elimina un nodo de la lista, dicho cambio se ve reflejado en el documento y viceversa. Esto puede dificultar el recorrido de una `NodeList` puesto que su longitud puede variar mientras se está recorriendo, y los nodos pueden moverse de un lugar a otro de la lista. Los `NodeList` no son objetos seguros para la concurrencia, como casi todo lo demás en DOM, por lo que hay que tener cuidado al leer y escribir simultáneamente en un `NodeList`.

4. JAXP (Java API for XML Processing)

En el apartado anterior ya se ha tratado una de las utilidades de JAXP, como conjunto de interfaces abstractas que uniformizan el manejo de parsers SAX y, especialmente, DOM, haciéndolo independiente del fabricante del parser.

Otra de las capacidades de JAXP es la adición a la biblioteca de Java de capacidades estándar de transformación de documentos XML. En el siguiente ejemplo se verá cómo emplear el API TrAX (*Transformations for XML*) de JAXP para serializar un documento XML, leyéndolo y escribiéndolo a través de objetos *Transformer*. Los *Transformer* son el soporte de TrAX para realizar transformaciones XSL sobre documentos XML.

Ejemplo: `RestructureDriver.java`

```
import javax.xml.parsers.*; // JAXP
```

```

import javax.xml.transform.*; // JAXP
import javax.xml.transform.dom.DOMSource; // JAXP
import javax.xml.transform.stream.StreamResult; // JAXP
import org.xml.sax.SAXException;
import org.w3c.dom.Document;
import java.io.IOException;

public class RestructureDriver
{
    public static void main(String[] args)
    {
        if (args.length <= 0)
        {
            System.out.println("Usage: java RestructureDriver URL");
            return;
        }
        String url = args[0];

        try
        {
            // Find a parser
            DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();
            factory.setNamespaceAware(true);
            DocumentBuilder parser = factory.newDocumentBuilder();

            // Read the document
            Document document = parser.parse(url);

            // Write it out again
            TransformerFactory xformFactory =
                TransformerFactory.newInstance();
            Transformer idTransform = xformFactory.newTransformer();
            Source input = new DOMSource(document);
            Result output = new StreamResult(System.out);
            idTransform.transform(input, output);
        }
        catch (SAXException e)
        {
            System.out.println(url + " is not well-formed.");
        }
        catch (IOException e)
        {
            System.out.println("Due to an IOException, the parser could" +
                " not read " + url);
        }
        catch (FactoryConfigurationError e)
        {
            System.out.println("Could not locate a factory class");
        }
        catch (ParserConfigurationException e)
        {
            System.out.println("Could not locate a JAXP parser");
        }
        catch (TransformerConfigurationException e)
        {
            System.out.println("This DOM does not support transforms.");
        }
        catch (TransformerException e)
        {
            System.out.println("Transform failed.");
        }
    }
}

```

Realmente este ejemplo no hace ninguna transformación. Simplemente crea y usa un *Transformer* para volcar en la consola un documento XML tal cual lo lee de un origen de datos DOM (*DOMSource*). Si se indaga en el API de JAXP se verá cómo crear objetos *Transformer* que encapsulen las transformaciones especificadas a través de una hoja XSL.

5. JDOM (Java Document Object Model)

JDOM es un API que sirve para procesar documentos XML desde Java y que simplifica bastante dicha tarea en comparación a cómo se haría usando DOM.

JDOM representa un documento XML como un árbol de objetos en memoria, compuesto de elementos, atributos, comentarios, instrucciones de procesamiento, nodos de texto, secciones CDATA, etc. A diferencia de DOM, los distintos tipos de nodos de un árbol están representados con clases concretas en vez de con interfaces. En JDOM no existe una interfaz/clase *Node* o similar, suficientemente genérica y que todos los tipos de nodo implementen/extiendan.

Como JDOM está especialmente pensado para Java, sigue las convenciones de nombrado del lenguaje y utiliza las clases de biblioteca del JDK. Por ejemplo, todas las clases principales de JDOM implementan los métodos *equals()*, *toString()* y *hashCode()*, además de implementar las interfaces *Cloneable* y *Serializable*. Los hijos de un objeto de tipo *Document* se almacenan en una *java.util.List* en vez de en una *org.w3c.dom.NodeList* como hacía DOM.

Aunque el API de JDOM no incluye un parser XML por sí mismo, viene empaquetado junto al parser xerces. Sin embargo, es posible usar JDOM en conjunción con cualquier otro parser compatible (crimson, aelfred, oracle, etc.)

5.1. Configuración

En primer lugar hay que descargarse el jar de JDOM desde www.jdom.org y configurar el CLASSPATH para que encuentre las clases de JDOM.

Por ejemplo, si descargamos la última versión (*jdom-b10.zip*) y la descomprimos en la carpeta actual, encontraremos una versión operativa del jar que necesitamos en *./jdom-b10/build/jdom.jar*. Esto es todo lo que necesitamos añadir al classpath:

- En Windows: `set CLASSPATH=%CLASSPATH%;./jdom-b10/build/jdom.jar`
- En Linux: `CLASSPATH=$CLASSPATH:./jdom-b10/build/jdom.jar`

5.2. Creación de documentos XML

A continuación se muestra un ejemplo de la simplicidad de JDOM frente al equivalente DOM, para crear un documento XML con un único elemento:

```
<?xml version="1.0"?>
<saludo>Hola</saludo>
```

Versión JDOM

```
Element root = new Element("saludo");
root.setText("Hola");
Document doc = new Document(root);
```

Versión DOM

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
DOMImplementation impl = builder.getDOMImplementation();
Document doc = impl.createDocument(null, "saludo", null);
Text text = doc.createTextNode("Hola");
doc.appendChild(text);
```

A diferencia de DOM, donde todos los nodos que se crean deben hacerse asociados a un cierto documento, JDOM permite crear nodos "sueltos" que se pueden enganchar con posterioridad a un documento.

En la versión con JDOM del ejemplo, el elemento *root* está inicialmente suelto. Tras llamar al constructor de *Document*, quedará asociado como elemento raíz. Sin embargo, en JDOM un mismo nodo no puede formar parte de dos documentos a la vez. Antes de engancharlo a un segundo documento, debe ser desenganchado del primero llamando al método *detach()*.

Otras dos formas de creación del mismo documento del ejemplo anterior:

Forma 1ª

```
Element root = new Element("saludo");
Document doc = new Document(root);
root.setText("Hola");
```

Forma 2ª

```
Document doc = new Document();
Element root = new Element("saludo");
root.setText("Hola");
doc.setRootElement(root);
```

Como puede verse, se puede alterar el orden de los métodos e incluso usar un constructor sin parámetros para el Documento. En este último caso, el documento pasa por un estado transitorio durante el cual nada se podrá hacer con él excepto asignarle un elemento raíz o añadirle contenido. Cualquier otra operación provocará una excepción *IllegalStateException*.

5.3. Escritura de documentos XML

La clase `org.jdom.output.XMLOutputter` es la manera más común de serializar un documento XML en forma de fichero, un string, un socket, o cualquier otro *stream* de bytes. El siguiente fragmento de código muestra cómo se escribe un documento XML en la consola:

```
XMLOutputter outputter = new XMLOutputter();
try
{
    outputter.output(doc, System.out);
}
catch (IOException e)
{
    System.err.println(e);
}
```

No sólo se pueden escribir documentos completos, sino también partes de un documento. Por ejemplo, el siguiente código sólo escribe el elemento `<saludo>`:

```
XMLOutputter outputter = new XMLOutputter();
try
{
    Element element = new Element("saludo");
    outputter.output(element, System.out);
}
catch (IOException e)
{
    System.err.println(e);
}
```

Sin embargo, hay que tener cuidado con esto pues lo que se escribe puede que no sea un documento XML bien formado.

El siguiente ejemplo genera un documento XML que contiene números de Fibonacci, empleando el API de JDOM:

Ejemplo: `FibonacciJDOM.java`

```
import org.jdom.*;
import org.jdom.output.XMLOutputter;
import java.math.BigInteger;
import java.io.IOException;
```

```

public class FibonacciJDOM
{
    public static void main(String[] args)
    {
        Element root = new Element("numeros_fibonacci");
        BigInteger low = BigInteger.ONE;
        BigInteger high = BigInteger.ONE;

        for (int i = 1; i <= 5; i++)
        {
            Element fibonacci = new Element("fibonacci");
            fibonacci.setAttribute("indice", String.valueOf(i));
            fibonacci.setText(low.toString());
            root.addContent(fibonacci);
            BigInteger temp = high;
            high = high.add(low);
            low = temp;
        }

        Document doc = new Document(root);
        try
        {
            XMLOutputter serializer = new XMLOutputter();
            serializer.output(doc, System.out);
        }
        catch (IOException e)
        {
            System.err.println(e);
        }
    }
}

```

Si se escribe un fichero FibonacciJDOM.java con el programa anterior y se hace una llamada al mismo, se obtendrá la salida siguiente:

```

<?xml version="1.0" encoding="UTF-8"?>
<Fibonacci_Numbers><fibonacci indice="1">1</fibonacci><fibonacci
indice="2">1</fibonacci><fibonacci indice="3">2</fibonacci><fibonacci
indice="4">3</fibonacci><fibonacci indice
="5">5</fibonacci></Fibonacci_Numbers>

```

Para compilar y ejecutar el ejemplo anterior, es necesario incorporar la biblioteca *jdom.jar* al CLASSPATH si no se ha hecho aún.

Como puede verse, la salida generada es algo confusa, porque en ella no aparecen espacios en blanco, tabuladores y saltos de línea. Podemos solicitar al XMLOutputter que formatee el documento con un formato determinado pasándole al constructor una instancia de la clase *Format*.

Ejemplo: PrettyFormatFibonacci.java

```

import org.jdom.*;
import org.jdom.output.XMLOutputter;
import org.jdom.output.Format;
import java.math.BigInteger;
import java.io.IOException;

public class FibonacciJDOM
{
    public static void main(String[] args)
    {
        Element root = new Element("numeros_fibonacci");
        BigInteger low = BigInteger.ONE;
        BigInteger high = BigInteger.ONE;

        for (int i = 1; i <= 5; i++)
        {
            Element fibonacci = new Element("fibonacci");
            fibonacci.setAttribute("indice", String.valueOf(i));
            fibonacci.setText(low.toString());
            root.addContent(fibonacci);

```

```

        BigInteger temp = high;
        high = high.add(low);
        low = temp;
    }

    Document doc = new Document(root);
    try
    {
        XMLOutputter serializer =
            new XMLOutputter(Format.getPrettyFormat());
        serializer.output(doc, System.out);
    }
    catch (IOException e)
    {
        System.err.println(e);
    }
}
}
}

```

5.4. Lectura de documentos con JDOM

Al contrario que SAX y DOM, no existen parsers específicos con capacidades para JDOM, sino que éste se apoya en los parsers SAX/DOM comunes como Xerces o Crimson. Nótese cómo el API de JDOM incluye adaptadores para los parsers más comunes en el paquete *org.jdom.adapters*.

JDOM puede leer documentos XML de una amplia variedad de fuentes, como ficheros (*java.io.File*), URLs (*java.net.URL*), strings, sockets, o cualquier otro objeto al que se le pueda asociar un *stream* (*java.io.InputStream*) o un *reader* (*java.io.Reader*) en Java. Además, también pueden emplearse árboles SAX y/o DOM como orígenes de datos, a través de las envolturas proporcionadas por *SAXBuilder* y *DOMBuilder*, respectivamente. También pueden emplearse orígenes de datos de SAX (*org.xml.sax.InputSource*).

El siguiente ejemplo muestra cómo se usa JDOM para chequear la corrección de un documento (que esté bien formado), apoyándose en un parsing SAX a través de *SAXBuilder*.

Ejemplo: JDOMChecker.java

```

import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import java.io.IOException;

public class JDOMChecker
{
    public static void main(String[] args)
    {
        if (args.length == 0)
        {
            System.out.println("Usage: java JDOMChecker URL");
            return;
        }

        SAXBuilder builder = new SAXBuilder();
        // command line should offer URIs or file names
        try
        {
            builder.build(args[0]);
            // If there are no well-formedness errors,
            // then no exception is thrown
            System.out.println(args[0] + " is well-formed.");
        }

        // indicates a well-formedness error
        catch (JDOMException e)
        {
            System.out.println(args[0] + " is not well-formed.");
            System.out.println(e.getMessage());
        }
        catch (IOException e)
        {
            System.out.println("Could not check " + args[0]);
        }
    }
}

```

```

        System.out.println(" because " + e.getMessage());
    }
}

```

Lo más común es usar JDOM en conjunción con un parser SAX, pues son más eficientes que DOM al ocupar menos memoria (Los parsers DOM suelen cargar el documento entero en memoria, en forma de un árbol de objetos DOM, para poder realizar el procesado).

La elección del parser SAX empleado por JDOM depende del entorno local. Por defecto, JDOM confía en JAXP para que elija la clase concreta que implementa el parser. Si esto falla, JDOM elige el parser de Xerces. Si se quiere especificar la clase concreta, puede pasarse el nombre de la clase que implementa el parser como argumento al constructor de SAXBuilder. Por ejemplo, para elegir Crimson:

```

SAXBuilder parser = new SAXBuilder
    ("org.apache.crimson.parser.XMLReaderImpl");

```

5.5. Navegación por árboles JDOM

La navegación por el árbol de objetos JDOM suele hacerse a través de objetos de tipo *Element*. La lista de hijos de un elemento determinado está disponible en forma de una *java.util.List* devuelta por los métodos *getChildren()* (devuelve sólo los hijos directos de tipo *Element*) y *getContent()* (devuelve todos los hijos de tipo *Element*, *Text*, *Comment*, etc.)

El ejemplo siguiente lee un documento y muestra los nombres de todos sus elementos, convenientemente sangrados según su nivel en el árbol.

Ejemplo: ElementLister.java

```

import org.jdom.*;
import org.jdom.input.SAXBuilder;
import java.io.IOException;
import java.util.*;

public class ElementLister
{
    public static void main(String[] args)
    {
        if (args.length == 0)
        {
            System.out.println("Usage: java ElementLister URL");
            return;
        }

        SAXBuilder builder = new SAXBuilder();
        try
        {
            Document doc = builder.build(args[0]);
            Element root = doc.getRootElement();
            listChildren(root, 0);
        }
        // indicates a well-formedness error
        catch (JDOMException e)
        {
            System.out.println(args[0] + " is not well-formed.");
            System.out.println(e.getMessage());
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }

    public static void listChildren(Element current, int depth)
    {
        printSpaces(depth);
        System.out.println(current.getName());
        List children = current.getChildren();
    }
}

```

```

        Iterator iterator = children.iterator();

        while (iterator.hasNext())
        {
            Element child = (Element) iterator.next();
            listChildren(child, depth+1);
        }
    }

    private static void printSpaces(int n)
    {
        for (int i = 0; i < n; i++)
        {
            System.out.print(' ');
        }
    }
}

```

El método `getChildren()` sólo devuelve elementos, ignorando el resto de nodos, como los comentarios, las instrucciones de procesado, o los nodos de texto. Para llegar hasta estos nodos, hace falta emplear `getContent()`, que devuelve todos los tipos de nodo. Sin embargo, esto hace el procesado de las listas un poco más difícil, ya que no se puede suponer que todos sus elementos son de tipo `Element`, impidiendo el casting del ejemplo anterior, que ha de ser sustituido por una cadena de comprobaciones con *instanceof* para averiguar el tipo del nodo recuperado de la lista, como se muestra en el ejemplo siguiente.

Ejemplo: NodeLISTER.java

```

import org.jdom.*;
import org.jdom.input.SAXBuilder;
import java.io.IOException;
import java.util.*;

public class NodeLISTER
{
    public static void main(String[] args)
    {
        if (args.length == 0)
        {
            System.out.println("Usage: java NodeLISTER URL");
            return;
        }

        SAXBuilder builder = new SAXBuilder();
        try
        {
            Document doc = builder.build(args[0]);
            listNodes(doc, 0);
        }

        // indicates a well-formedness error
        catch (JDOMException e)
        {
            System.out.println(args[0] + " is not well-formed.");
            System.out.println(e.getMessage());
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }

    public static void listNodes(Object o, int depth)
    {
        printSpaces(depth);
        if (o instanceof Element)
        {
            Element element = (Element) o;
            System.out.println("Element: " + element.getName());
            List children = element.getContent();
            Iterator iterator = children.iterator();
            while (iterator.hasNext())
            {

```

```

        Object child = iterator.next();
        listNodes(child, depth+1);
    }
}
else if (o instanceof Document)
{
    System.out.println("Document");
    Document doc = (Document) o;
    List children = doc.getContent();
    Iterator iterator = children.iterator();
    while (iterator.hasNext())
    {
        Object child = iterator.next();
        listNodes(child, depth+1);
    }
}
else if (o instanceof Comment)
{
    System.out.println("Comment");
}
else if (o instanceof CDATA)
{
    System.out.println("CDATA section");
    // CDATA is a subclass of Text so this test must come
    // before the test for Text.
}
else if (o instanceof Text)
{
    System.out.println("Text");
}
else if (o instanceof EntityRef)
{
    System.out.println("Entity reference");
}
else if (o instanceof ProcessingInstruction)
{
    System.out.println("Processing Instruction");
}
else
{
    // This really shouldn't happen
    System.out.println("Unexpected type: " + o.getClass());
}
}

private static void printSpaces(int n)
{
    for (int i = 0; i < n; i++)
    {
        System.out.print(' ');
    }
}
}

```

El programa anterior podría arreglarse haciendo un buen diseño con herencia y polimorfismo de las clases que representan a los nodos del árbol.

Ejercicio

Re-factorizar el ejemplo anterior para suprimir el abuso de instanceof, mediante un diseño de clases con herencia que refleje los elementos y la definición de métodos polimórficos oportunos.

6. Ampliación de la práctica

En la aplicación de fútbol realizada en las prácticas anteriores con Servlets se incluía una base de datos relacional que guardaba datos sobre los jugadores de fútbol por los que los usuarios podían votar, así como un registro de los votos emitidos por los usuarios. Todas las consultas

y actualizaciones a la BD estaban encapsuladas en una clase creada para el acceso a los datos, cuyo código puede encontrarse en la práctica 5 (Servlets y Patrón MVC) implementado en la clase Jugadores.

Se pide:

1. Diseñar e implementar una clase JugadoresXML que funcione como adaptador (siguiendo el patrón *Adapter*) de Jugadores, ofreciendo una interfaz de consulta/actualización de la base de datos más acorde con la navegación por un documento XML.
2. Modificar la aplicación de la práctica 5 para que los JSP/servlets utilicen la clase JugadoresXML implementada en el punto anterior para averiguar los datos de futbolistas y estadísticas.
3. Implementar en Java la posibilidad de realizar distintas transformaciones XSL sobre las estadísticas de la aplicación, mostrándolas en diversos formatos: listados de texto, HTML. Modificar los servlets de la aplicación de la práctica 5 para que muestren el HTML a partir de dichas transformaciones.