DAM. UNIT 1. ACCESS TO FILES PART 2. FILES. XML & XSL

DAM. Acceso a Datos (ADA) (a distancia en inglés)

Unit 1. ACCESS TO FILES

Part 2. Files. XML & XSL

Abelardo Martínez

Year 2023-2024

1. Introduction

The use of independent binary formats goes beyond the Java language barrier, allowing other languages to read files that follow the specified format. However, binary formats still present some incompatibility if the operating systems you are working on use different binary data representation systems. As you know, not all systems represent data in the same way. Some use BCD to represent numbers, others use 2's complement. When a data occupies more than one byte, it can be read starting from the smallest byte (Little Endian) or from the largest (Big Endian), etc.

In other words, **binary formats do not guarantee data compatibility between systems**. Therefore, when you want to store data that has to be read by applications running on multiple platforms, it is necessary to resort to more standardised formats, such as mark-up languages. These **mark-up languages**, such as **XML**, **are used to solve this problem**.

2. XML language

XML documents structure information by interspersing a series of tags called hashtags. In XML, hashtags have a certain similarity to an information container. Thus, a hashtag can contain other hashtags or textual information. In this way, we will be able to subdivide the information by structuring it so that it can be information by structuring it in a way that it can be easily interpreted.

As all information is textual, there is no problem of representing the data in different ways. Any data, whether numeric or Boolean, will have to be transcribed in text mode, so that whatever the data representation system is, it will be possible to read and interpret correctly the information contained in an XML file.

It is true that characters can also be written using different encoding systems, but XML offers several techniques to avoid this being a problem. For example, it is possible to include in the file header which encoding has been used during storage, or it is also possible to write ASCII code characters greater than 127, using character entities, a universal way of encoding any symbol.

XML is able to structure any kind of hierarchical information. Some similarity can be drawn between the way information is stored in the objects of an application and the way it would be stored in an XML document.

Information, in object-oriented applications, is structured, grouped and hierarchised in classes, and in XML documents it is structured, organised and hierarchised in hashtags contained within each other and attributes of the hashtags.

2.1. What is XML?

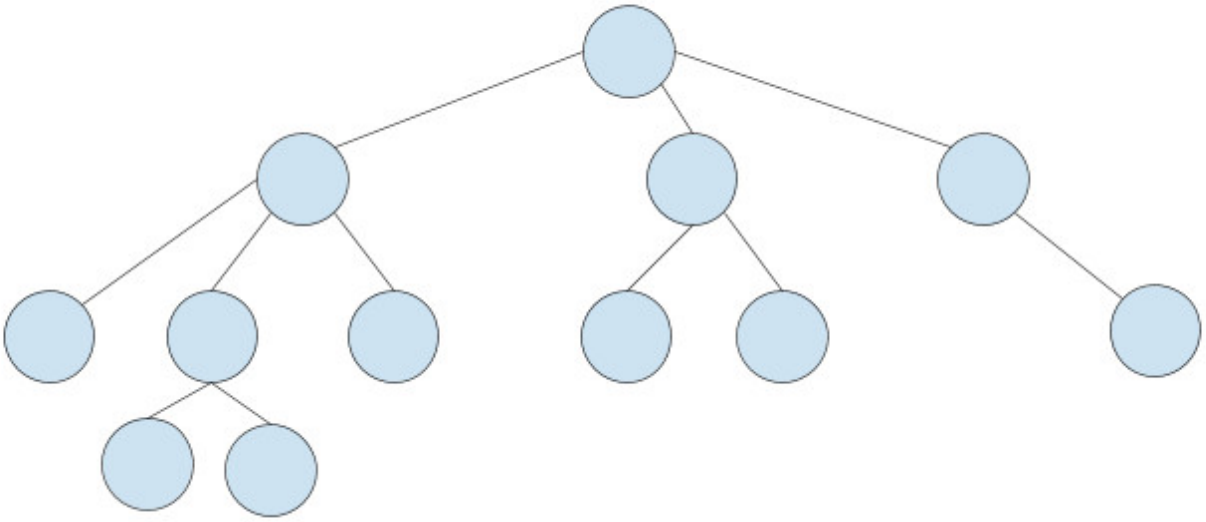
XML (E**x**tensible **M**arkup **L**anguage) is a markup language similar to HTML, but without predefined tags to use. Instead, you define your own tags designed specifically for your needs. This is a powerful way to store data in a format that can be stored, searched, and shared. Most importantly, since the fundamental format of XML is standardized, if you share or transmit XML across systems or platforms, either locally or over the internet, the recipient can still parse the data due to the standardized XML syntax.

There are many languages based on XML, including XHTML, MathML, SVG, RSS, and RDF. You can also define your own.

2.2. Structure of an XML document

XML is a metalanguage (language for the definition of markup languages) that allows us to hierarchise and describe the contents within the document itself. XML files are files where the information is organised sequentially and in hierarchical order.

In an XML document, **information is organised in a hierarchical way**, so that elements are related to each other through relationships of parent, child, sibling, parentage, descent, etc. **XML documents form a tree structure** that starts at "the root" and branches to "the leaves".



2.2.1. Correct design rules

For an XML document to be correct, the following conditions must be fulfilled:

- Document must be well-formed.
- Document must conform to all XML syntax rules.
- Document must conform to semantic rules, which are usually set in an XML schema or a DTD (Document Type Definition).

2.2.2. XML syntax

An XML document is a string of characters. Every legal Unicode character (except Null) may appear in an (1.1) XML document (while some are discouraged).

a) Markup and content

The characters making up an XML document are divided into markup and content, which may be distinguished by the application of simple syntactic rules. Generally, strings that constitute markup either begin with the character **<** and end with a **>**, or they begin with the character **&** and end with a **;**. Strings of characters that are not markup are content. However, in a CDATA section, the delimiters **<![CDATA[** and **]]>** are classified as markup, while the text between them is classified as content. In addition, whitespace before and after the outermost element is classified as markup.

b) Tag

A tag is a markup construct that **begins with < and ends with >**. There are three types of tag:

- ***start-tag***, such as <section>
- ***end-tag***, such as </section>
- ***empty-element tag***, such as <line-break />

c) Element

An element is a logical document component that either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag. The characters between the start-tag and end-tag, if any, are the element's content, and may contain markup, including other elements, which are called child elements.

Examples:

```
<greeting>Hello, world!</greeting>

<line-break />
```

d) Attribute

An attribute is a markup construct consisting of a name–value pair that exists within a start-tag or empty-element tag.

Examples:

```
<img src="valentia.jpg" alt="Valentia" /> <!-- where the names of the attribut

<step number="3">Connect A to B.</step> <!-- where the name of the attribute i
```

An XML attribute can only have a single value and each attribute can appear at most once on each element. In the common situation where a list of multiple values is desired, this must be done by encoding the list into a well-formed XML attribute with some format beyond what XML defines itself. Usually this is either a comma or semi-colon delimited list or, if the individual values are known not to contain spaces, a space-delimited list can be used.

e) XML declaration

XML documents may begin with an XML declaration that describes some information about themselves. XML declaration is not a tag. It is used for the transmission of the meta-data of a document.

Example:

```
<?xml version="1.0" encoding="UTF-8"?> <!-- where version means used version X
```

f) Comments

Logically, XML allows comments to be inserted within the document structure. Comments may appear anywhere in a document outside other markup. Comments cannot appear before the XML declaration.

Example:

```
<!-- Comment -->
```

2.2.3. Entities

Like HTML, XML offers methods (called entities) for referring to some special reserved characters (such as a greater than sign which is used for tags). There are five of these characters that you should know:

Entity	Character	Description
<	<	Less than sign
>	>	Greater than sign
&	&	Ampersand
"	"	One double-quotation mark
'	'	One apostrophe (or single-quotation mark)

Even though there are only 5 declared entities, more can be added using the document's Document Type Definition. You can also use numeric character references to specify special characters; for example, © is the "©" symbol.

2.2.4. Displaying XML

XML is usually used for descriptive purposes, but there are ways to display XML data. If you don't define a specific way for the XML to be rendered, the raw XML is displayed in the browser.

One way to style XML output is to specify CSS to apply to the document using the **xml-stylesheet** processing instruction.

Example:

```
<?xml-stylesheet type="text/css" href="stylesheet.css"?>
```

There is also another more powerful way to display XML: the **Extensible Stylesheet Language Transformations** (XSLT) which can be used to transform XML into other languages such as HTML. This makes XML incredibly versatile.

Example:

```
<?xml-stylesheet type="text/xsl" href="transform.xsl"?>
```

2.2.5. Examples

Example 1. Incorrect XML (bad-formed)

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
<warning>
Hello World
<!--missing </warning> -->
</message>
```

Example 2. Valid XML (well-formed)

Now let's look at a corrected version of that same document:

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
<warning>
Hello World
</warning>
</message>
```

A document that contains an undefined tag is invalid. For example, if we never defined the <warning> tag, the document above wouldn't be valid.

Most browsers offer a debugger that can identify poorly-formed XML documents.

3. Working with XML files

XML files can be used to:

- Provide data to a DB.
- Store information in special databases.
- Configuration files.
- Exchange of information in web environments (SOAP).
- Execution of commands on remote servers.
- Etc.

To carry out any of these operations, a programming language that provides XML with functionality is necessary (XML is not a complete Turing language).

3.1. XML processor or parser

The processor analyzes the markup and passes structured information to an application. The specification places requirements on what an XML processor must do and not do, but the application is outside its scope. The processor (as the specification calls it) is often referred to colloquially as an XML parser.

An XML Parser is a class that aims to parse and classify the content of an XML file by extracting the information contained in each of the hashtags and relating it according to its position within the hierarchy.

3.1.1. Sequential parsers

Sequential parsers, which allow content to be extracted as the opening and closing hashtags are discovered, are called **syntactic parsers**. They are very fast parsers, but have the problem that every time you need to access a piece of content you have to reread the whole document from top to bottom.

In Java, the most popular parser is called SAX, which stands for **S**imple **A**PI for **X**ML. It is a parser widely used in a number of XML data processing libraries, but not often used in end-applications.

- **Advantages**
  - They are very fast.
- **Disadvantages**
  - They have to read the whole document at each query.

3.1.2. Hierarchical parsers

Generally, **end applications** that have to work with XML data **tend to use hierarchical parsers**, because in addition to performing a sequential parsing that allows them to classify the content, they are stored in RAM following the hierarchical structure detected in the document. This makes it much easier to perform queries.

This greatly facilitates queries that need to be repeated several times, since the hierarchical structures in RAM have a very efficient partial data access performance.

The format of the structure where the information is stored in RAM has been specified by the international body W3C (World Wide Web Consortium) and is commonly known as **DOM** (**D**ocument **O**bject **M**odel). It is a structure that HTML and JavaScript have made very popular and it is a specification that Java materialises in the form of interfaces. The main one is called Document and represents a whole XML document. Since it is an interface, it can be be implemented by several classes.

- **Advantages**
  - They are ideal for applications that require continuous querying of data.
- **Disadvantages**
  - They store all XML document data in memory (RAM) within a hierarchical structure.

4. SAX parser

The XML-DEV mailing group developed a **S**imple **A**PI for **X**ML also called the **SAX**, which is an event-driven online algorithm for parsing XML documents. SAX (Simple API for XML) is a set of classes and interfaces that **provide a tool for processing XML documents**. The SAX API is fully included within the JRE.

SAX is a way of reading data from an XML document that is an alternative to the Document Object Model's mechanism (DOM). Whereas the DOM works on the document as a whole, creating the whole abstract syntax tree of an XML document for the user's convenience, SAX parsers work on each element of the XML document sequentially, issuing parsing events while passing through the input stream in a single pass. Unlike DOM, SAX does not have a formal specification.

SAX is a programming interface for processing XML files based on events. The DOM's counterpart, SAX, has a very different way of reading XML code. The Java implementation of SAX is regarded as the de-facto standard. SAX processes documents state-independently, in contrast to DOM which is used for state-dependent processing of XML documents.

In a nutshell, SAX is more complex to program than DOM, however it offers lower memory consumption, making it more **suitable for processing large XML files**.

Advantages

- It parses the XML file as a stream rather than allocating RAM for the complete file.
- Since, it uses less memory and is faster than the DOM Parser because the complete file is not stored in memory.
- Therefore, it is considered to be useful in parsing large XML files.

Drawbacks

- There are no update methods in the SAX Parser. Since the complete file isn't kept in memory, it is possible to access items only in a sequential manner and the elements cannot be accessed randomly.

4.1. Read XML files with Java

The SAX stands for the “Simple API for XML”. Unlike a DOM parser which loads document in the memory, SAX s an event-based parse. It works on the events when an event occur it calls some callback methods. The parse() method of the SAXParser will starts the XML processing.

SAX parser works on following events:

- startDocument
- startElement
- characters
- comments
- processing instructions
- endElement
- endDocument

Reading an XML with SAX produces events that cause different methods to be called. Commonly used methods of SAX XML Parser:

Method	Description
startDocument()	It is called at the beginning of the xml document.
endDocument()	It is called at the end of the xml document.
startElement(String uri, String localName, String qName, Attributes atts)	It is called at the beginning of an element.
endElement(String uri, String localName,String qName)	It is called at the end of an element.
characters(char[] ch, int start, int length)	It is called when text data is encountered between start and end tags of an element.

All this methods are defined in the **DefaultHandler**. The following are the **steps to read a file using SAX**:

1. Create a class to extend org.xml.sax.helpers.DefaultHandler, and override the startElement, endElement and characters methods to print all the XML elements, attributes, comments and texts.

2. Create a new class that will be in charge of the main execution of our application (MAIN):
    1. Create a SAXParserFactory instance, as determined by the setting of the javax.xml.parsers.SAXParserFactory system property. The factory to be created is set up to support XML namespaces by setting setNamespaceAware to true, and then a SAXParser instance is obtained from the factory by invoking its newSAXParser() method.
    2. Create an object of the DefaultHander class we have just created.
    3. Use the parse method of our SAXParserFactory to read the file. We pass as parameters the name of the XML file and the object of the DefaultHander class.

4.2. Example of reading

Let's see a small example in Java.

1) The first thing we will do is to import all the necessary classes and interfaces.

```
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.SAXException;
```

2) Then we create a class that extends DefaultHandler in order to handle read events. This class provides the default implementation for all its methods, the programmer must define those methods to be used by the application. This is the class we will extend in order to create our XML Parser.

```
/*
 * ----------------------------
 * SAX Handler for Students
 * ----------------------------
 */
public class PrintAllHandlerSax extends DefaultHandler{

    //This is the list which shall be populated while parsing the XML
    private StringBuilder sbCurrentValue = new StringBuilder();

    @Override
    public void startDocument() {
        System.out.println("Start Document");
    }

    @Override
    public void endDocument() throws SAXException {
```

```
        System.out.println("End Document");
    }

    @Override
    public void startElement(String uri, String localName, String qName, Attri
        //reset the tag value
        sbCurrentValue.setLength(0);
        System.out.printf("  Start Element: %s%n", qName);
    }

    @Override
    public void endElement(String uri, String localName, String qName) throws
        //handle the value based on to which element it belongs
        if (qName.equalsIgnoreCase("studentID")) {
            System.out.printf("    ID : %s%n", sbCurrentValue.toString());
        }
        System.out.printf("  End Element: %s%n", qName);
    }

    /*
     * This will be called everytime parser encounter a value node
     */
    @Override
    public void characters(char[] ch, int start, int length) throws SAXExcepti
        // The characters() method can be called multiple times for a single t
        // Some values may missing if assign to a new string
        sbCurrentValue.append(ch, start, length);
    }

}
```

3) Once we have our own handler, we create a new class that will be in charge of the main execution of our application (MAIN)

- Create a SAXParserFactory instance, as determined by the setting of the javax.xml.parsers.SAXParserFactory system property. The factory to be created is

set up to support XML namespaces by setting setNamespaceAware to true, and then a SAXParser instance is obtained from the factory by invoking its newSAXParser() method.

```
SAXParser saxParser = saxFactory.newSAXParser();
```

- Create an object of the DefaultHander class we have just created.

```
PrintAllHandlerSax saxHandler = new PrintAllHandlerSax();
```

- We must tell our XML Parser to use an instance of it in order to read the file. Once we have the source created and the ContentHandler assigned we will proceed to read the file simply by using the Parse method of our XML processor.

```
saxParser.parse(XML_FILENAME, saxHandler);
```

Complete code of the main class:

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

import org.xml.sax.SAXException;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

/*
 * ----------------------------------------
 * Reading an XML file with SAX
```

```
 * ----------------------------------------
 */

public class StudentReaderSAX {

    /**
     * ----------------------------------------
     * GLOBAL VARIABLES AND CONSTANTS
     * ----------------------------------------
     */
    //File name constant. We assume that our file is located in the project fo
    static final String XML_FILENAME = "students.xml";

    /**
     * ----------------------------------------
     * MAIN PROGRAMME
     * ----------------------------------------
     */
    public static void main(String[] stArgs) {

        System.out.println("");
        SAXParserFactory saxFactory = SAXParserFactory.newInstance();
        try {
            SAXParser saxParser = saxFactory.newSAXParser();
            PrintAllHandlerSax saxHandler = new PrintAllHandlerSax();
            saxParser.parse(XML_FILENAME, saxHandler);
        /*
         * DO NOT FORGET TO SET THE PROPER EXCEPTION HANDLING HERE
         */
        } catch (ParserConfigurationException | SAXException saxe) {
            System.out.println("File not found");
        } catch (FileNotFoundException fne) {
            System.out.println("File not found");
        } catch (IOException ioe) {
            System.out.println("IO Error");
        }
```

```
    }
}
```

5. DOM parser

The DOM defines a standard for accessing and manipulating documents: "The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

The XML DOM defines a standard way for accessing and manipulating XML documents. DOM stores all the data of the XML document in memory in a hierarchical tree structure.

Advantages:

- Simply put, a DOM parser works on the entire XML document.
- It’s ideal for applications requiring continuous querying of the data.

Disadvantages:

- It loads the whole document into memory. This type of processing requires more memory resources and time than SAX.
- Not suitable for large documents.

5.1. The DOM structure

The DOM structure takes the form of a tree, where each part of the XML is represented as a node. Depending on the position in the XML document, we will talk about different types of nodes. The main node representing the entire XML is called **document**, and the various hashtags, including the root tag, are known as **element** nodes. The textual content of a hashtag is instantiated as a node of type **TextElement** and the attributes as nodes of type **Attribute**.

Each specific node has methods to access its specific data (name, value, child nodes, peer node, etc.).

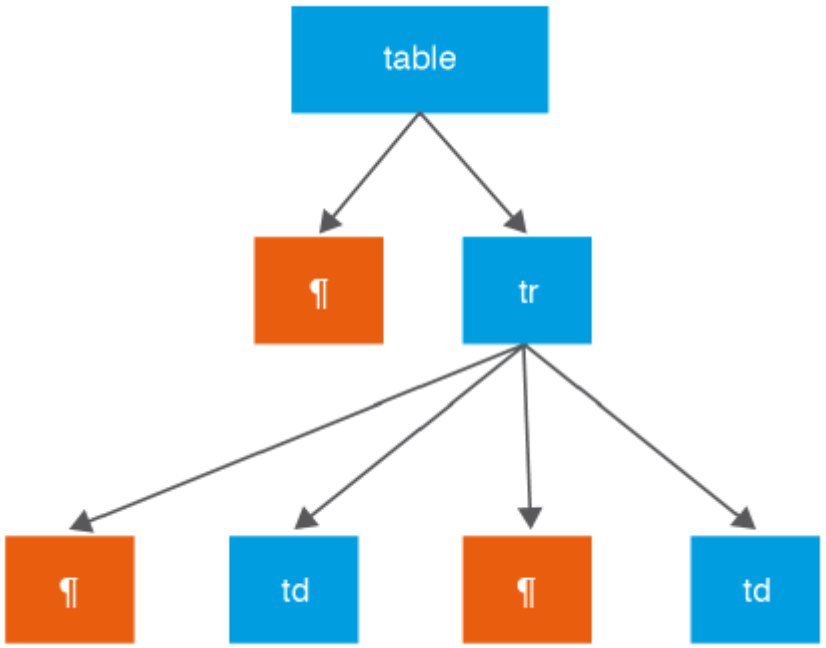
The resulting DOM obtained from an XML ends up being an exact copy of the file, but arranged in a different way. Both in the XML and in the DOM there will be non-visible information, such as carriage returns, which must be taken into account in order to know how to correctly process the content and avoid unintelligible surprises.

a) Representation of a DOM object with carriage returns

Let's imagine that we have an XML document with the following content:

```
<table>
<tr>
<td></td>
<td></td>
</tr>
</table>
```

The following figure shows the representation that the DOM object would have, once copied into memory. It should be noted that the table element will have two children. One will store the carriage return that places the hashtag tr on the next line; in the other we will find the hashtag tr. The same happens with the children of tr, before each td node we will find a carriage return.

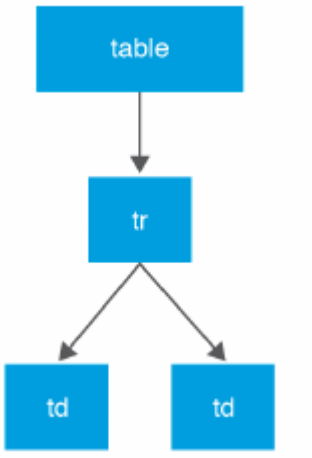


b) Representation of a DOM object without carriage returns

On the other hand, if we had started from an equivalent XML without carriage returns, the result would also have been different.

```
<table><tr><td></td><td></td></tr></table>
```

The above XML document, without carriage returns, would give the representation of the DOM object that you can see in the figure below.



The absence of carriage returns in the file also implies the absence of nodes containing the

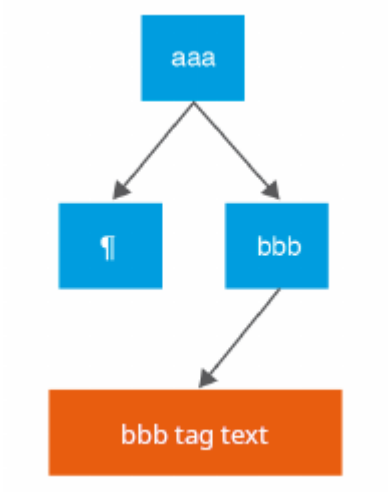
carriage returns in the DOM structure.

c) DOM representation of a hashtag with text

Another aspect to bear in mind about mapping XML files is that the textual content of hashtags is captured in the DOM as a child node of the hashtag container. In other words, to get the text of a hashtag you have to get the first child of the hashtag.

```
<aaa>
<bbb>
bbb tag text
</bbb>
</aaa>
```

The following figure illustrates the DOM representation of a hashtag containing text.



The **Document** interface provides a set of methods to select different parts of the tree from the hashtag name or an identifying attribute. The parts of the tree are returned as **Element** objects, which represent a node and all its children. In this way, we can explore parts of the tree without having to go through all of the need to go through all the nodes.

5.2. Manage XML files with Java

In order to work with DOM in Java we will make use of the packages

- org.w3c.dom (Contained in the JSDK)
- javax.xml.parsers (From the standard Java API)

They provide two abstract classes that we need to extend to work with DOM.

- **DocumentBuilderFactory**
- **DocumentBuilder**

DOM does not define any mechanism to generate an XML file from a DOM tree. For this we will use the javax.xml.transform package. This package allows to specify a source and a result:

- Files
- Data streams
- DOM nodes
- Etc.

Java programs using DOM need to make use of the following interfaces:

Interface	Description
Document	It is an object that represents the XML document. It allows the creation of new nodes.
Element	Represents each of the elements of the document.
Node	Represents any node in the document.
NodeList	Contains a list of child nodes of a given node.
Attr	Allows access to the attributes of a node.
Text	Represents the character data of a node.
CharacterData	Represents the character data present in the document.
DocumentType	Provides information contained in the <!DOCTYPE> tag.

5.3. Example of reading

To read an XML document using DOM we must **create an instance of DocumentBuilderFactory** to build the parser and, through it, load the document.

```
//https://howtodoinjava.com/java/xml/read-xml-dom-parser-example/
//We create the DocumentBuilder to be able to obtain the Document
DocumentBuilderFactory dbfEmployee= DocumentBuilderFactory.newInstance();
DocumentBuilder dbEmployee=dbfEmployee.newDocumentBuilder();
//We read the Document from the file
Document docEmployee= dbEmployee.parse(new File("Employees.xml"));
//We standardise the document to avoid reading errors
docEmployee.getDocumentElement().normalize();
```

We then create a list as all the elements used using the **class NodeList**.

```
//Display the name of the root element
System.out.println("The root element is "+
docEmployee.getDocumentElement().getNodeName());

//We create a list of all the nodes Employee
NodeList nlstEmployee= docEmployee.getElementsByTagName("employee");
//We show the number of Employee elements we have found
System.out.println("The following have been found "+nlstEmployee.getLength()+'
```

Finally, using a loop we **traverse the NodeList and display its contents**.

```
//We go through the list
for(int ii=0; ii<nlstEmployee.getLength();ii++) {
```

```
        //We get the first node in the list
        Node nodeEmployee= nlstEmployee.item(ii);
        //In case that node is an Element
        if(nodeEmployee.getNodeType()==Node.ELEMENT_NODE) {
            //We create the employee element and read its information
            Element objEmployee= (Element)nodeEmployee;
            System.out.print("ID: " + objEmployee.getElementsByTagName("id").
                    item(0).getTextContent() );
            System.out.print("\tName: " + objEmployee.getElementsByTagName("name")
                    item(0).getTextContent()  );
            System.out.println("\tSurname: " + objEmployee.getElementsByTagName("s
                    item(0).getTextContent()  );
        }
    }
```

5.4. Example of writing

Next we are going to create an XML file of employees.

The first thing to do is to **import the necessary packages**.

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

Then we will *create the Document* in which we are going to insert our employees.

```
//We create a DocumentBuilder using the DocumentBuilderFactory
DocumentBuilderFactory dbfEmployee= DocumentBuilderFactory.newInstance();
DocumentBuilder dbEmployee=dbfEmployee.newDocumentBuilder();

/* We create an empty document
 * Name --> EmployeeRecord
 * Root Node --> Employees
 */
DOMImplementation domImplement= dbEmployee.getDOMImplementation();
Document docEmployee = domImplement.createDocument(null, "employees", null);
//We assign the XML version
docEmployee.setXmlVersion("1.0");
```

And we will **create each of the employees with their data**.

```
//We create an Employee node
Element objEmployee= docEmployee.createElement("employee");
//We add him as a child of employees
docEmployee.getDocumentElement().appendChild(objEmployee);
//We create the node ID
Element eNodeID= docEmployee.createElement("id");
//Create the text node with the ID value
Text txtNode= docEmployee.createTextNode("01");
//We add the value to the ID node
eNodeID.appendChild(txtNode);
//We add the ID node to Employee
objEmployee.appendChild(eNodeID);
Element eNodeName= docEmployee.createElement("name");
txtNode= docEmployee.createTextNode("Pepe");
eNodeName.appendChild(txtNode);
objEmployee.appendChild(eNodeName);
Element eNodeSurname= docEmployee.createElement("surname");
txtNode= docEmployee.createTextNode("García");
eNodeSurname.appendChild(txtNode);
objEmployee.appendChild(eNodeSurname);
```

Finally, we must **write our Document to disk**.

```
/*
 * Finally, to save the document to disk we must:
 *
 * 1. Create the data source (Our Document)
 * 2. Create the result (The target file)
 * 3. Create a TransformerFactory
 * 4. Perform the transformation
 */
Source srcEmployee= new DOMSource(docEmployee);
Result resultFile= new StreamResult(new File ("Employees.xml"));
Transformer transfEmployee = TransformerFactory.newInstance().newTransformer()
```

```
transfEmployee.transform(srcEmployee, resultFile);
```

In the same way that we use the Transform class to send our Document to file, we can use it to display it via the **standard output**.

```
/*
 * We display the result by the standard output
 */
Result resultStdOutput = new StreamResult(System.out);
transfEmployee.transform(srcEmployee, resultStdOutput);
```

6. XSL language

XSL (which stands for e**X**tensible **S**tylesheet **L**anguage), is a family of recommendations for defining XML document transformation and presentation. That is, XSL is a styling language for XML.

It has been developed by the W3C. It consists of three parts:

- **XSL Transformations (XSLT)**. A language for transforming XML;
- **The XML Path Language (XPath)**. An expression language used by XSLT (and many other languages) to access or refer to parts of an XML document.
- **XSL Formatting Objects (XSL-FO)**. An XML vocabulary for specifying formatting semantics.

6.1. What is XSLT?

XSLT (e**X**tensible **S**tylesheet **L**anguage **T**ransformations) is an XML-based language used, in conjunction with specialized processing software, for the transformation of XML documents.

Although the process is referred to as "transformation," the original document is not changed; rather, a new XML document is created based on the content of an existing document. Then, the new document may be serialized (output) by the processor in standard XML syntax or in another format, such as HTML or plain text.

XSLT is most often used to convert data between different XML schemas or to convert XML data into web pages or PDF documents.

For further information, please consult: XSLT Element Reference.

7. Bibliography

Sources

- Wikipedia. XML. https://en.wikipedia.org/wiki/XML
- Developer Mozilla. XML introduction. https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction
- W3schools. Introduction to XML. https://www.w3schools.com/XML/xml_whatis.asp
- Josep Cañellas Bornas, Isidre Guixà Miranda. Accés a dades. Desenvolupament d'aplicacions multiplataforma. Creative Commons. Departament d'Ensenyament, Institut Obert de Catalunya. Dipòsit legal: B. 29430-2013. https://ioc.xtec.cat/educacio/recursos
- Alberto Oliva Molina. Acceso a datos. UD 1. Trabajo con ficheros XML. IES Tubalcaín. Tarazona (Zaragoza, España).
- Oracle Documentation. SAX. https://docs.oracle.com/javase/tutorial/jaxp/sax/index.html
- Geeksforgeeks. What is SAX in XML? https://www.geeksforgeeks.org/what-is-sax-in-xml/
- W3schools. SAX XML parser in Java. https://www.w3schools.blog/sax-xml-parser-in-java-tutorial-example
- Oracle Documentation. DOM. https://docs.oracle.com/javase/tutorial/jaxp/dom/index.html
- W3schools. XSLT Introduction. https://www.w3schools.com/xml/xsl_intro.asp