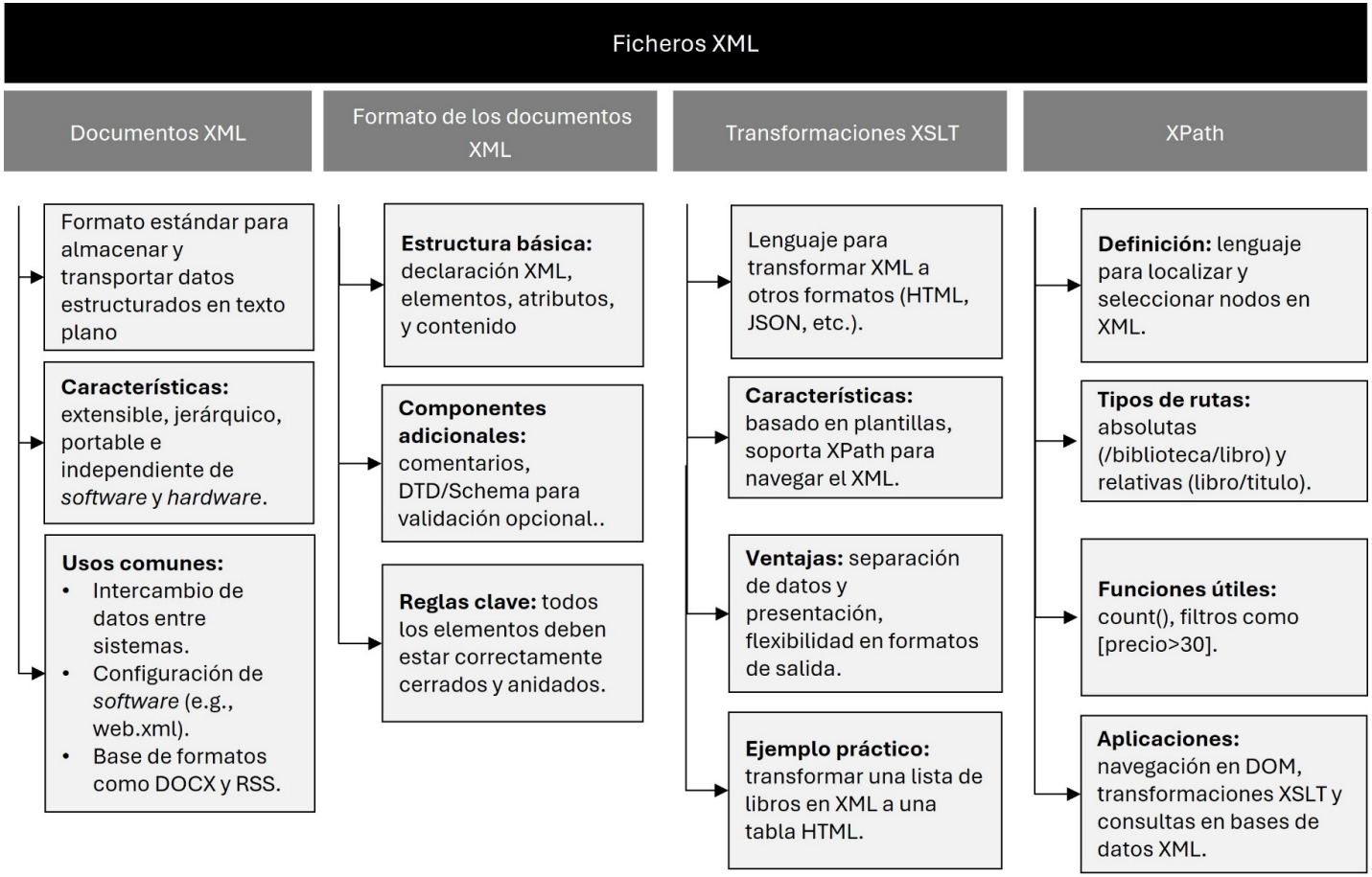
Acceso a Datos

Tema 2. Ficheros XML

Índice

Esquema


Ficheros XML
Documentos XML
Formato de los documentos XML
Transformaciones XSLT
XPath
Formato estándar para almacenar y transportar datos estructurados en texto plano
Características: extensible, jerárquico, portable e independiente de *software* y *hardware*.
Usos comunes:
• Intercambio de datos entre sistemas.
• Configuración de *software* (e.g., web.xml).
• Base de formatos como DOCX y RSS.
Estructura básica: declaración XML, elementos, atributos, y contenido
Componentes adicionales: comentarios, DTD/Schema para validación opcional..
Reglas clave: todos los elementos deben estar correctamente cerrados y anidados.
Lenguaje para transformar XML a otros formatos (HTML, JSON, etc.).
Características: basado en plantillas, soporta XPath para navegar el XML.
Ventajas: separación de datos y presentación, flexibilidad en formatos de salida.
Ejemplo práctico: transformar una lista de libros en XML a una tabla HTML.
Definición: lenguaje para localizar y seleccionar nodos en XML.
Tipos de rutas: absolutas (/biblioteca/libro) y relativas (libro/titulo).
Funciones útiles: count(), filtros como [precio>30].
Aplicaciones: navegación en DOM, transformaciones XSLT y consultas en bases de datos XML.

2.1. Introducción y objetivos

En el desarrollo de aplicaciones, los **datos** juegan un papel fundamental. La forma en que estos se estructuran, almacenan y transfieren afecta directamente a la eficiencia y escalabilidad de los sistemas. Uno de los formatos más extendidos y versátiles para la representación de datos estructurados es **XML** *(extensible markup language).* Desde su creación, XML ha sido adoptado como un estándar en múltiples áreas gracias a su capacidad para describir datos de manera jerárquica y legible, tanto para humanos como para máquinas.

Este tema se centra en explorar los **aspectos esenciales** de los documentos XML, incluyendo su estructura, formato y las herramientas que permiten manipularlos de forma eficaz. Además, se abordarán tecnologías relacionadas como **XSL,** para transformar documentos XML, y **XPath,** para navegar por ellos. Estas herramientas son clave para trabajar con XML en aplicaciones multiplataforma.

Dado que el uso de XML es común en entornos de desarrollo con **Java,** se estudiará la integración de XML con este lenguaje de programación. En particular, se revisarán técnicas como el uso del **parser DOM** para procesar documentos XML, lo que permitirá a los estudiantes entender cómo leer, modificar y generar archivos XML desde una aplicación.

Los objetivos que se pretenden conseguir en este tema son:

- **Comprender los fundamentos de XML:** analizar qué es XML, su propósito y las características principales que lo hacen un estándar para la estructuración y el intercambio de datos.
- **Aprender el formato y la estructura de documentos XML:** identificar los componentes esenciales de un archivo XML y las reglas que rigen su formato para garantizar la correcta validación y compatibilidad.

Material de estudio

- **Dominar las transformaciones de datos con XSL:** estudiar cómo transformar documentos XML a otros formatos como HTML o JSON utilizando XSL *(extensible stylesheet language).*
- **Explorar las posibilidades de XPath:** aprender a navegar y seleccionar partes específicas de un documento XML utilizando consultas XPath.
- **Manejar documentos XML con Java:** implementar aplicaciones en Java que permitan leer, modificar y generar documentos XML mediante herramientas como el parser DOM.
- **Aplicar los conocimientos adquiridos en casos prácticos:** desarrollar ejemplos funcionales que integren XML con aplicaciones multiplataforma.

2.2. Documentos XML

¿Qué son los ficheros XML?

XML o *extensible markup language* (lenguaje de marcado extensible) es un formato estándar creado para almacenar, estructurar y transportar datos de una manera que sea legible para los humanos y procesable por las máquinas. Fue desarrollado por el World Wide Web Consortium (W3C) y es ampliamente utilizado en aplicaciones web, bases de datos, intercambio de información entre sistemas y configuraciones de *software.*

Un fichero XML es esencialmente un documento de texto plano que utiliza etiquetas (similares a HTML) para organizar los datos en una estructura jerárquica.

Sin embargo, a diferencia de HTML, XML no está diseñado para mostrar datos, sino para describirlos.

Características principales:

- **Estructura jerárquica**: los datos están organizados en una estructura de árbol con elementos padre e hijo.
- **Extensible**: puedes definir tus propias etiquetas, adaptando el documento a las necesidades específicas de cada aplicación.
- **Portabilidad**: al ser texto plano, los documentos XML pueden ser compartidos entre diferentes plataformas y sistemas.
- **Independencia de *software* y *hardware***: XML no está ligado a un lenguaje de programación ni a un sistema operativo concreto.

¿Para qué se usan los ficheros XML?

Los ficheros XML tienen múltiples aplicaciones en el desarrollo de *software* y la transferencia de datos. Algunas de las más comunes incluyen:

- **Intercambio de datos entre sistemas**: XML es ampliamente utilizado para enviar y recibir datos entre aplicaciones que pueden estar desarrolladas en diferentes lenguajes o plataformas. Por ejemplo, en las API (interfaces de programación de aplicaciones) web.
- **Almacenamiento de configuraciones**: muchos programas utilizan XML para almacenar configuraciones y preferencias de usuario, como los archivos web.xml en aplicaciones Java.
- **Documentos estructurados**: XML es la base de muchos formatos de documentos, como DOCX (Microsoft Word), XLSX (Excel) o SVG (gráficos vectoriales).
- **Formatos de datos específicos**: XML se utiliza como base para otros lenguajes derivados como RSS (para *feeds*), SOAP (para servicios web) y XHTML (para páginas web).
- **Bases de datos**: en bases de datos NoSQL, como MongoDB, XML se utiliza como formato de datos para almacenar información estructurada.
- **Transformaciones de datos**: con herramientas como XSLT, los documentos XML pueden transformarse en otros formatos, como HTML o JSON, lo que facilita su uso en diversas aplicaciones.

Ventajas del uso de XML:

- **Estandarización:** es un formato reconocido y ampliamente soportado.
- **Flexibilidad:** permite personalizar etiquetas y estructuras.
- **Compatibilidad:** funciona en múltiples plataformas y lenguajes.

Desventajas del uso de XML:

- **Verbosidad:** los documentos XML pueden volverse muy grandes y difíciles de leer cuando contienen grandes cantidades de datos.
- **Rendimiento:** procesar documentos XML puede ser más lento en comparación con otros formatos más ligeros, como JSON.

2.3. Formato de los documentos XML

El formato de un documento XML sigue una serie de **reglas estrictas** para garantizar su validez y compatibilidad. Veamos sus componentes principales:

- **Declaración XML**: aparece al inicio del documento e indica la versión de XML y la codificación de caracteres utilizada.

```
<?xml version="1.0" encoding="UTF-8"?>
```

- **Elementos**: son las etiquetas que definen los datos y su estructura. Cada elemento tiene una etiqueta de apertura y una de cierre.

```
<nombre>Juan Pérez</nombre>
```

- **Atributos**: proveen información adicional sobre los elementos. Se escriben dentro de la etiqueta de apertura.

```
<persona id="123">
    <nombre>Juan Pérez</nombre>
</persona>
```

- **Contenido de los elementos**: puede ser texto, otros elementos (*hijos*), o estar vacío.

```
<direccion>
    <calle>Gran Vía</calle>
    <ciudad>Madrid</ciudad>
</direccion>
```

- **Comentarios**: permiten incluir anotaciones en el documento que no serán procesadas.

```
<!-- Este es un comentario -->
```

- **Prolog y DTD/XSD (opcional)**: opcionalmente, un documento XML puede incluir un DTD (*document type definition*) o un esquema (*XML Schema*) para validar su estructura y contenido.

Ejemplo de un fichero XML válido}

```
<?xml version="1.0" encoding="UTF-8"?>

<libro>

<titulo>Introducción a XML</titulo>

<autor>Juan Pérez</autor>

<año>2024</año>

<precio moneda="EUR">29.99</precio>

</libro>
```

2.4 Transformaciones XSL

Transformaciones XSLT (extensible stylesheet language transformations)

XSLT es un lenguaje diseñado para transformar documentos XML en otros formatos. Estas transformaciones son útiles para adaptar el contenido de un archivo XML a diferentes contextos, como convertirlo a HTML para mostrarlo en un navegador, JSON para las API o incluso otro XML con una estructura diferente.

XSLT utiliza un conjunto de **reglas definidas en hojas de estilo XSL** *(extensible stylesheet language).* Estas reglas se aplican al documento XML original para generar el resultado transformado.

XSLT es una herramienta poderosa y versátil para trabajar con datos estructurados en XML. Aprender a dominarla te permitirá realizar transformaciones avanzadas y aprovechar al máximo la flexibilidad de XML en proyectos multiplataforma.

Características principales de XSLT

- **Basado en plantillas**: define plantillas que se aplican a elementos o conjuntos de elementos del XML de origen.
- **Declarativo**: no se especifican pasos imperativos, sino reglas sobre cómo transformar cada parte del XML.
- **Soporte para XPath**: utiliza XPath para seleccionar y navegar por los nodos del XML.
- **Transformaciones flexibles**: permite generar HTML, texto plano, JSON, otro XML, entre otros formatos.

Sintaxis básica de XSLT

Una transformación XSLT se define en un archivo con una estructura similar a XML. Las reglas principales son:

- **Declaración inicial**: la transformación comienza con una declaración estándar que indica que se trata de un archivo XSLT:
  - `version="1.0"` especifica la versión de XSLT.
  - `xmlns:xsl="..."` define el espacio de nombres para las etiquetas XSLT.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

- **Plantillas**: se definen con la etiqueta `<xsl:template>` y se aplican a elementos específicos del XML.
- **Salidas**: se especifica el formato del resultado usando `<xsl:output>` .

Ejemplo de transformación XML a HTML

XML de entrada:

```
<?xml version="1.0" encoding="UTF-8"?>

<libros>

    <libro>

        <titulo>Introducción a XML</titulo>

        <autor>Juan Pérez</autor>

        <precio>29.99</precio>

    </libro>
```

```
    <libro>
        <titulo>Transformaciones XSLT</titulo>
        <autor>María López</autor>
        <precio>39.99</precio>
    </libro>
</libros>
```

XSLT para generar HTML:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <!-- Configuración de salida como HTML -->
    <xsl:output method="html" indent="yes"/>
    <!-- Plantilla para el nodo raíz -->
    <xsl:template match="/">
        <html>
            <head>
                <title>Lista de Libros</title>
            </head>
            <body>
                <h1>Libros Disponibles</h1>
                <table border="1">
```

```
                    <tr>
                        <th>Título</th>
                        <th>Autor</th>
                        <th>Precio</th>
                    </tr>
                    <!-- Iterar sobre los libros -->
                    <xsl:for-each select="libros/libro">
                        <tr>
                            <td><xsl:value-of select="titulo"/></td>
                            <td><xsl:value-of select="autor"/></td>
                            <td><xsl:value-of select="precio"/></td>
                        </tr>
                    </xsl:for-each>
                </table>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

Salida HTML generada:

```
<html>
    <head>
        <title>Lista de Libros</title>
    </head>
    <body>
        <h1>Libros Disponibles</h1>
        <table border="1">
            <tr>
                <th>Título</th>
                <th>Autor</th>
                <th>Precio</th>
            </tr>
            <tr>
                <td>Introducción a XML</td>
                <td>Juan Pérez</td>
                <td>29.99</td>
            </tr>
            <tr>
                <td>Transformaciones XSLT</td>
```

```
                <td>María López</td>
                <td>39.99</td>
            </tr>
        </table>
    </body>
</html>
```

Uso avanzado de XSLT

- **Aplicación condicional**: puedes usar etiquetas como <xsl:if> y <xsl:choose> para manejar lógica condicional. En este ejemplo, se muestran solo los libros con precio mayor a treinta.

```
<xsl:for-each select="libros/libro">
    <xsl:if test="precio &gt; 30">
        <tr>
            <td><xsl:value-of select="titulo"/></td>
            <td><xsl:value-of select="autor"/></td>
            <td><xsl:value-of select="precio"/></td>
        </tr>
    </xsl:if>
</xsl:for-each>
```

- **Transformaciones a otros formatos**: puedes generar JSON u otros XML.

```
<xsl:output method="text"/>
```

```
<xsl:template match="/">
    {
        "libros": [
            <xsl:for-each select="libros/libro">
                {
                    "titulo": "<xsl:value-of select='titulo'/>",
                    "autor": "<xsl:value-of select='autor'/>",
                    "precio": "<xsl:value-of select='precio'/>"
                }<xsl:if test="position() != last()">,</xsl:if>
            </xsl:for-each>
        ]
    }
</xsl:template>
```

- **Reutilización con plantillas nombradas:** puedes definir plantillas que se invocan por nombre para mejorar la modularidad:

```
<xsl:template name="mostrar-libro">
    <tr>
        <td><xsl:value-of select="titulo"/></td>
        <td><xsl:value-of select="autor"/></td>
        <td><xsl:value-of select="precio"/></td>
```

```
    </tr>
</xsl:template>
```

Invocación: <xsl:call-template name="mostrar-libro"/>

Ventajas de XSLT

- **Separación de contenido y presentación**: permite transformar datos sin modificar el XML original.
- **Flexibilidad**: puedes generar múltiples formatos de salida.
- **Reutilización**: las hojas de estilo XSLT pueden aplicarse a varios documentos XML similares.

Limitaciones de XSLT

- **Complejidad sintáctica**: su estructura puede resultar complicada para transformaciones complejas.
- **Rendimiento**: en grandes volúmenes de datos, el procesamiento puede ser más lento que otras técnicas como programación directa en lenguajes como Python o Java.

2.5. XPath

XPath, o *XML Path Language*, es un lenguaje diseñado para navegar y seleccionar nodos dentro de documentos XML. Permite localizar elementos, atributos y otros componentes de un XML de manera precisa, lo que facilita su manipulación y consulta.

Estructura de un documento XML

Antes de profundizar en XPath, es esencial recordar que un documento XML se representa como un **árbol de nodos,** donde cada elemento, atributo y texto constituye un nodo en esta estructura jerárquica.

Ejemplo de documento XML

```
<?xml version="1.0" encoding="UTF-8"?>

<biblioteca>

<libro id="1">

<titulo>Aprendiendo XML</titulo>

<autor>Juan Pérez</autor>

<precio>29.99</precio>

</libro>

<libro id="2">

<titulo>Dominando XPath</titulo>

<autor>María Gómez</autor>

<precio>39.99</precio>
```

```
</libro>

</biblioteca>
```

Expresiones XPath

XPath utiliza expresiones de ruta similares a las de los sistemas de archivos para identificar nodos específicos en un documento XML. Estas expresiones pueden ser absolutas o relativas:

- **Ruta absoluta**: comienza desde el nodo raíz.
  - /biblioteca/libro selecciona todos los elementos <libro> que son hijos directos de <biblioteca>.
- **Ruta relativa**: parte del nodo actual en el contexto.
  - libro/titulo selecciona todos los elementos <titulo> que son hijos de <libro> desde el nodo actual.

Selección de nodos

XPath permite seleccionar diferentes tipos de nodos:

- **Elementos**: /biblioteca/libro selecciona todos los elementos <libro> bajo <biblioteca> .
- **Atributos**: /biblioteca/libro/@id selecciona el atributo id de cada <libro> .
- **Texto**: /biblioteca/libro/titulo/text() selecciona el contenido textual de cada <titulo> .

Operadores y funciones en XPath

XPath ofrece operadores y funciones para realizar consultas más complejas:

- **Predicados**: filtran nodos basándose en condiciones.
  - /biblioteca/libro[precio>30] selecciona libros con un precio mayor a treinta.
- **Funciones**: realizan operaciones sobre nodos o valores.
  - count(/biblioteca/libro) devuelve el número de libros en la biblioteca.

Ejes en XPath

Los ejes determinan la relación entre nodos en la navegación:

- **child**: selecciona hijos directos.
  - child::libro es equivalente a libro.
- **descendant**: selecciona todos los descendientes.
  - descendant::titulo selecciona todos los <titulo> en todos los niveles.
- **ancestor**: Selecciona todos los ancestros.
  - ancestor::biblioteca selecciona el ancestro <biblioteca>.
- **following-sibling**: selecciona los hermanos siguientes.
  - following-sibling::libro selecciona los elementos <libro > que son hermanos siguientes del nodo actual.

Ejemplos prácticos

- **Seleccionar el título del primer libro:** /biblioteca/libro[1]/titulo. Esto selecciona el elemento <titulo> del primer <libro> en la <biblioteca>.
- **Seleccionar libros con precio inferior a 35:** /biblioteca/libro[precio<35] . Esto selecciona todos los elementos <libro> cuyo <precio> es menor que 35.
- **Contar el número de libros:** count(/biblioteca/libro) . Esto devuelve el número total de elementos <libro> en la <biblioteca>.

Aplicaciones de XPath

XPath se utiliza en diversas tecnologías y lenguajes de programación para manipular y consultar documentos XML:

- **XSLT**: para transformar documentos XML en otros formatos, utilizando XPath para seleccionar nodos específicos.
- **XQuery**: para consultas avanzadas en bases de datos XML. Para ello, aprovecha la sintaxis de XPath.
- **DOM**: en lenguajes como JavaScript, para navegar y manipular documentos XML o HTML.

Dominar **XPath** es esencial para trabajar eficazmente con XML, ya que proporciona una herramienta poderosa para acceder y manipular datos estructurados de manera precisa y eficiente.

2.6. Parser DOM

Un ***parser*** o **analizador sintáctico** es una herramienta *software* capaz de analizar el contenido de un documento XML y generar, a partir de él, un modelo de objetos Java. También puede realizar la operación inversa, es decir, construir a partir de un modelo de objetos Java un documento XML.

Un *parser* realiza **dos tipos de tareas**:

- Primero, a partir de un documento XML, lo examina para ver si su sintaxis es correcta. Después, una vez analizada la sintaxis, construye a partir de él un modelo de objetos que podrá ser manipulado por un programa.
- A partir de un modelo de objetos Java, lo examina con el fin de construir a partir de él un documento XML o editar uno existente.

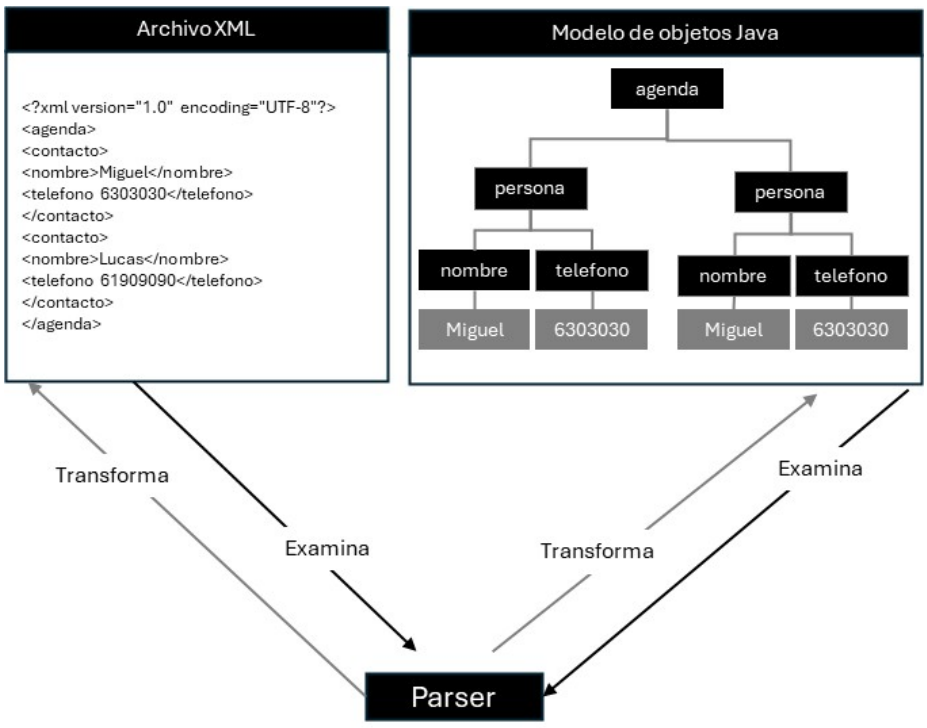


Figura 1. Tareas de un *parser*. Fuente: elaboración propia.

E l **parser DOM** *(document object model)* es una herramienta que permite a las aplicaciones leer y manipular documentos XML o HTML. Esto lo logra convirtiéndolos en una estructura de árbol en memoria. Esta representación jerárquica facilita el acceso y la modificación de los elementos y atributos del documento de manera programática.

¿Qué es el modelo de objetos del documento (DOM)?

E l **DOM** es una interfaz de programación que representa la estructura de un documento como un árbol de nodos, donde cada nodo corresponde a una parte del documento, como elementos, atributos o texto. Esta estructura permite a los lenguajes de programación interactuar con el contenido y la estructura del documento de forma dinámica.

Funcionamiento de un parser DOM

Un ***parser* DOM** analiza un documento XML o HTML y construye una representación en memoria en forma de árbol. Cada elemento del documento se convierte en un **nodo dentro de este árbol,** lo que permite a los desarrolladores navegar, buscar y modificar el contenido del documento, utilizando métodos y propiedades proporcionados por el DOM.

Ventajas y consideraciones del parser DOM

Las **ventajas** del *paser* DOM son:

- **Acceso aleatorio**: permite acceder y modificar cualquier parte del documento en cualquier momento.
- **Facilidad de uso**: proporciona una interfaz intuitiva para navegar y manipular la estructura del documento.

Las **consideraciones** del *paser* DOM son:

- **Consumo de memoria**: al cargar todo el documento en memoria, puede ser ineficiente para archivos muy grandes.
- **Rendimiento**: la construcción y manipulación del árbol DOM puede ser más lenta en comparación con otros métodos de análisis, especialmente con documentos extensos.

Para **documentos de gran tamaño** o cuando se requiere un procesamiento más eficiente, se pueden considerar alternativas como los *parsers* basados en eventos, como SAX *(simple API for XML)*, que no requieren cargar todo el documento en memoria.

Ejemplo práctico en Java

En Java, la **API de procesamiento XML (JAXP)** proporciona herramientas para trabajar con el DOM. A continuación, se presenta un ejemplo de cómo cargar y manipular un documento XML utilizando el *parser* DOM en Java:

```
import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;

import org.w3c.dom.Element;

import org.w3c.dom.NodeList;

public class DOMParserExample {

    public static void main(String[] args) {

        try {

            // Crear una instancia de DocumentBuilderFactory
```

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



            // Crear un DocumentBuilder

            DocumentBuilder builder = factory.newDocumentBuilder();



            // Parsear el documento XML y obtener el árbol DOM

            Document document = builder.parse("ruta/al/archivo.xml");



            // Obtener el elemento raíz

            Element root = document.getDocumentElement();

            System.out.println("Elemento raíz: " + root.getNodeName());



            // Obtener una lista de nodos con el nombre "libro"

            NodeList libros = document.getElementsByTagName("libro");



            // Recorrer la lista de libros y mostrar información

            for (int i = 0; i < libros.getLength(); i++) {

                Element libro = (Element) libros.item(i);

                String titulo =
libro.getElementsByTagName("titulo").item(0).getTextContent();
```

```
                String autor =
libro.getElementsByTagName("autor").item(0).getTextContent();

                System.out.println("Título: " + titulo);

                System.out.println("Autor: " + autor);

            }

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}
```

En este ejemplo, el programa realiza las siguientes acciones:

- **Inicializa** una fábrica de constructores de documentos (DocumentBuilderFactory) .
- **Crea** un constructor de documentos (DocumentBuilder) a partir de la fábrica.
- **Analiza** el archivo XML especificado y construye el árbol DOM correspondiente.
- **Obtiene** el elemento raíz del documento y lo imprime.
- **Recupera** todos los elementos <libro> y, para cada uno, extrae y muestra el título y el autor.

2.7. XML con Java

En este apartado utilizaremos el *parser* DOM para leer un documento cruceros.xml y construir, a partir de él, un árbol jerárquico de objetos Java denominado árbol DOM *(document object model)*.

Para lograr el objetivo necesitaremos **dos librerías** distintas:

- javax.xml.parsers: provee clases que permiten el procesamiento de documentos XML.
- org.w3c.dom: proporciona las interfaces para la representación del DOM *(document object model).*

Comenzaremos por un **ejemplo simple** que muestra todo el contenido de texto de la etiqueta raíz (cruceros), es decir, sin incluir las etiquetas, sólo los textos. Para ponerlo en práctica, puedes abrir el proyecto de la lección anterior denominado **ProyectoXML** y añadir la clase Java siguiente:

```
import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;

import org.w3c.dom.Node;

public class LeerCruceros {

    public static void main(String[] args) {

        DocumentBuilderFactory fabrica =
DocumentBuilderFactory.newInstance();

        DocumentBuilder analizador;

        Document doc;
```

```
        Node raiz;
        try {
            analizador = fabrica.newDocumentBuilder();
            doc = analizador.parse("cruceros.xml");
            raiz = doc.getDocumentElement();
            System.out.println(raiz.getTextContent());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Vamos a analizar detenidamente el ejemplo:

- DocumentBuilderFactory fabrica = DocumentBuilderFactory.newInstance();
  - La clase DocumentBuilderFactory , situada en el paquete javax.xml.parsers , nos permite obtener el objeto DocumentBuilder a partir de su método newInstante(). El objeto DocumentBuilder es imprescindible para analizar un documento XML y construir a partir de él un árbol DOM.
- DocumentBuilder analizador = fabrica.newDocumentBuilder();
  - La clase DocumentBuilder , situada en el paquete javax.xml.parsers , representa un analizador o *parser* cuyos objetos nos permiten construir el árbol DOM a partir del documento XML por medio de su método parse() .

- Document doc = analizador.parse("cruceros.xml");

  - La clase Document , situada en el paquete org.w3c.dom.Document , representa un modelo de objetos como réplica de un documento XML. Es tarea del objeto DocumentBuilder analizar el contenido del documento XML y devolver el objeto Document con el árbol DOM. En nuestro ejemplo, el objeto que hemos denominado doc contiene toda la estructura del documento XML.

- Node raiz = doc.getDocumentElement();

  - Un documento XML está formado por nodos o elementos que pueden, a su vez, contener otros nodos. El método getDocumentElement() de la clase Document devuelve el objeto Node, que representa el nodo raíz, que para nuestro ejemplo es el nodo *cruceros*.

- System.out.println(raiz.getTextContent());

  - Nuestra variable *raiz* es una referencia al objeto Node que representa el nodo *cruceros*. El método getTextContent() muestra todo el contenido de texto sin incluir las etiquetas.

Escribir documentos XML

La **tecnología DOM** también nos permite construir archivos XML a partir de un modelo de objetos Java. Comienza por crear un proyecto Java y la siguiente clase principal, luego analizaremos el código detenidamente.

```
import java.io.File;

import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import javax.xml.parsers.ParserConfigurationException;

import javax.xml.transform.Transformer;
```

```
import javax.xml.transform.TransformerException;

import javax.xml.transform.TransformerFactory;

import javax.xml.transform.dom.DOMSource;

import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;

import org.w3c.dom.Element;

public class CrearAgenda {

    public static void main(String[] args) {

        try {

            // Crear una instancia de DocumentBuilderFactory

            DocumentBuilderFactory fabrica =
DocumentBuilderFactory.newInstance();



            // Crear un DocumentBuilder

            DocumentBuilder constructor = fabrica.newDocumentBuilder();



            // Crear un nuevo documento XML

            Document documento = constructor.newDocument();



            // Crear el elemento raíz "agenda"

            Element agenda = documento.createElement("agenda");
```

```
            documento.appendChild(agenda);

            // Crear un elemento "contacto" con sus hijos
            Element contacto = documento.createElement("contacto");
            agenda.appendChild(contacto);

            Element nombre = documento.createElement("nombre");
            nombre.appendChild(documento.createTextNode("Juan Pérez"));
            contacto.appendChild(nombre);

            Element telefono = documento.createElement("telefono");
            telefono.appendChild(documento.createTextNode("123456789"));
            contacto.appendChild(telefono);

            Element email = documento.createElement("email");
email.appendChild(documento.createTextNode("juan.perez@example.com"));
            contacto.appendChild(email);

            // Crear otro elemento "contacto" con sus hijos
            Element contacto2 = documento.createElement("contacto");
```

```
            agenda.appendChild(contacto2);


            Element nombre2 = documento.createElement("nombre");

            nombre2.appendChild(documento.createTextNode("María Gómez"));

            contacto2.appendChild(nombre2);


            Element telefono2 = documento.createElement("telefono");

            telefono2.appendChild(documento.createTextNode("987654321"));

            contacto2.appendChild(telefono2);


            Element email2 = documento.createElement("email");

email2.appendChild(documento.createTextNode("maria.gomez@example.com"));

            contacto2.appendChild(email2);


            // Crear una instancia de TransformerFactory

            TransformerFactory transformadorFactory =
TransformerFactory.newInstance();

            Transformer transformador =
transformadorFactory.newTransformer();


            // Definir la fuente y el resultado de la transformación
```

```
            DOMSource fuente = new DOMSource(documento);

            StreamResult resultado = new StreamResult(new
File("agenda.xml"));



            // Realizar la transformación de DOM a archivo físico

            transformador.transform(fuente, resultado);



            System.out.println("Archivo XML 'agenda.xml' creado con
éxito.");



        } catch (ParserConfigurationException | TransformerException e) {

            e.printStackTrace();

        }

    }

}
```

Descripción del código:

- **Creación del documento y elementos**: se crea un documento XML con un elemento raíz <agenda>. Dentro de este, se añaden elementos <contacto>, cada uno con sus respectivos hijos <nombre>, <telefono> y <email>.
- **Transformación y escritura del archivo**: se utiliza Transformer para convertir el documento DOM en un archivo físico llamado agenda.xml.

Consideraciones:

- **Manejo de excepciones**: se capturan posibles excepciones como ParserConfigurationException y TransformerException para manejar errores durante la creación y transformación del documento.
- **Codificación**: por defecto, el archivo se guarda con la codificación estándar. Si deseas especificar una codificación diferente, puedes configurar el Transformer adecuadamente.

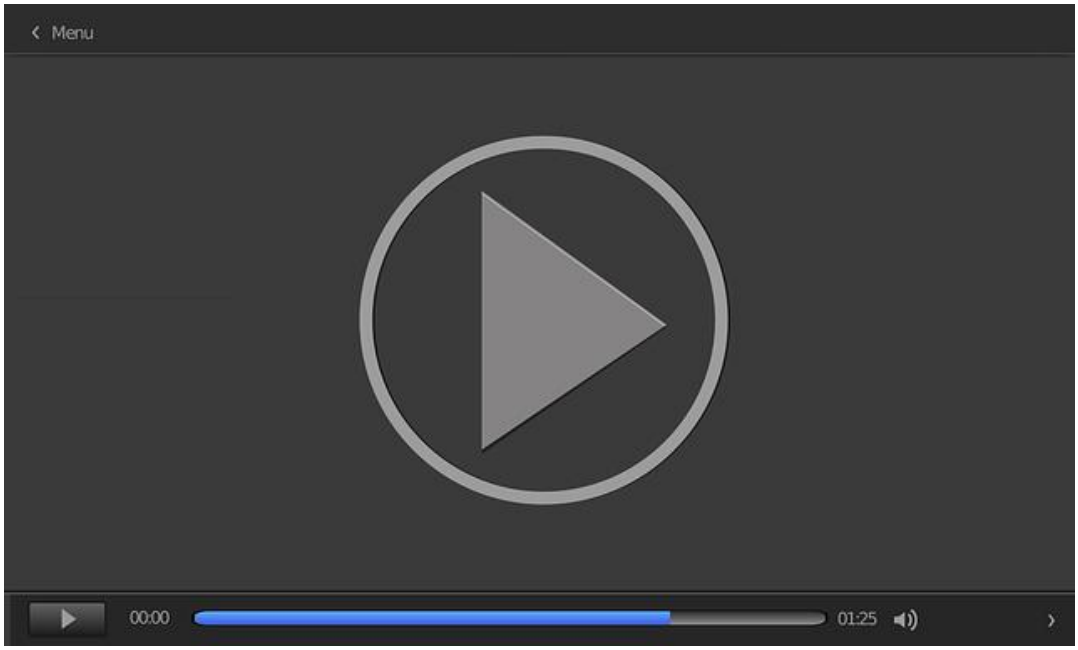
Este código genera un archivo agenda.xml con la siguiente estructura:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<agenda>

    <contacto>

        <nombre>Juan Pérez</nombre>

        <telefono>123456789</telefono>

        <email>juan.perez@example.com</email>

    </contacto>

    <contacto>

        <nombre>María Gómez</nombre>

        <telefono>987654321</telefono>

        <email>maria.gomez@example.com</email>

    </contacto>

</agenda>
```

Introducción a XSLT

nicosiored (2020, abril 17) *Introducción a XSLT - 1 - XSLT en español* [Vídeo]. YouTube. https://www.youtube.com/watch?v=bvWmayMoZxw

Este vídeo ofrece una introducción a los conceptos básicos de XSLT. Muestra cómo transformar documentos XML.



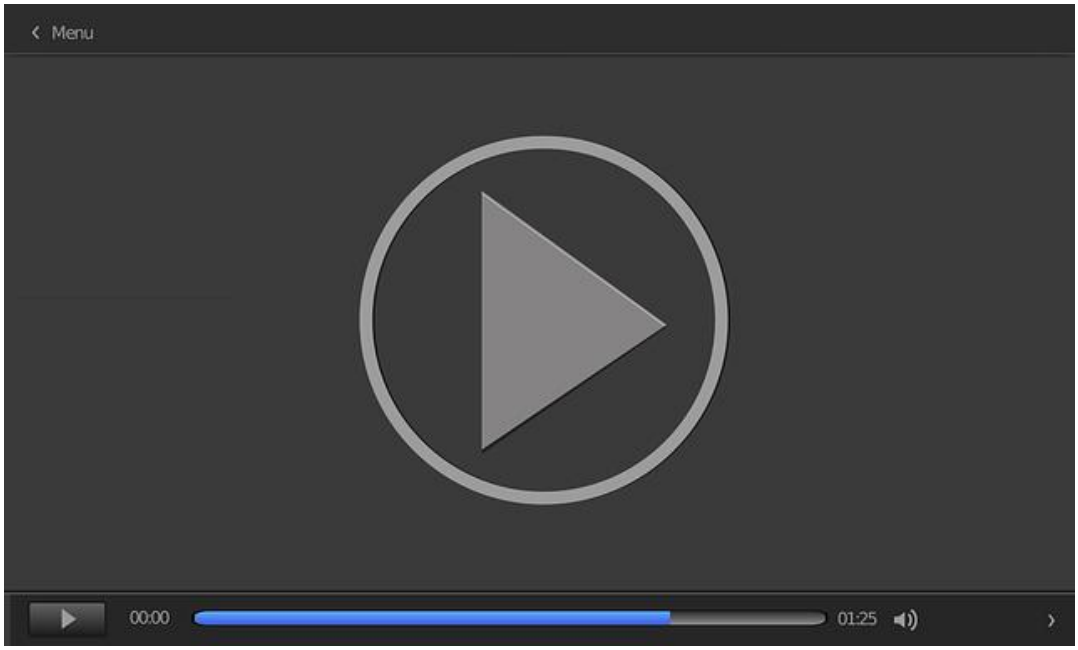
Accede al vídeo:

https://www.youtube.com/embed/bvWmayMoZxw

XPath, XML, Selenium y JavaScript

Damian Sire Desarrollo (2021, agosto 1). *XPath tutorial en español - XPath, XML, Selenium y JavaScript* [Video]. YouTube. https://www.youtube.com/watch?v=0VUegF7a0hg

Este tutorial en español aborda XPath, XML y su aplicación en Selenium y JavaScript.



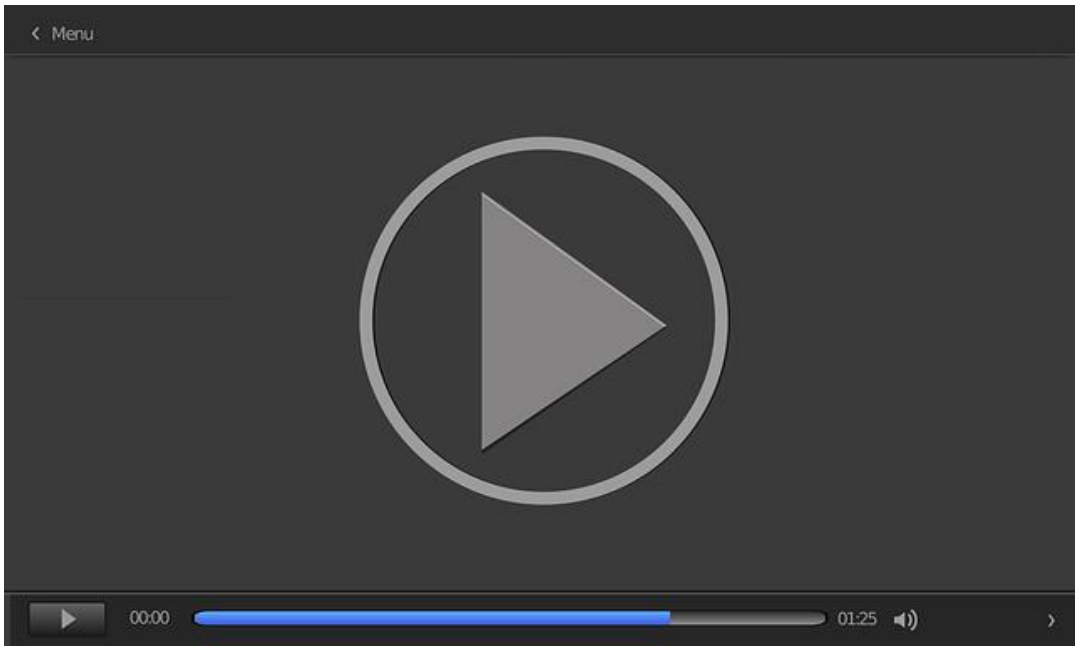
Accede al vídeo:

https://www.youtube.com/embed/0VUegF7a0hg

Tratamiento de XML en Java

Jey Code (2021, enero 10) *Tratamiento de XML en JAVA | Leer un XML con DOM* [Vídeo]. YouTube. https://www.youtube.com/watch?v=7mep1AtDFqo

Este vídeo muestra cómo leer documentos XML en Java utilizando el *parser* DOM del JDK.



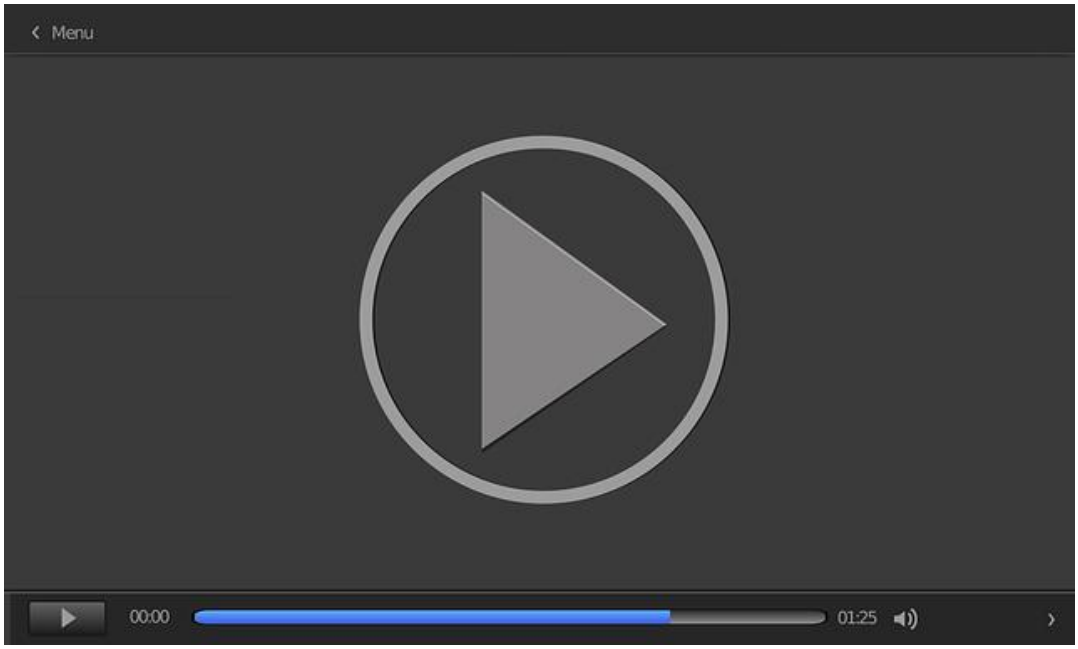
Accede al vídeo:

https://www.youtube.com/embed/7mep1AtDFqo

Introducción a XML

nicosiored (2018, febrero 5). *Introducción a XML - 1 - Tutorial XML básico en español* [Vídeo]. YouTube. https://www.youtube.com/watch?v=PxGICnkFZJU

Este vídeo introduce los conceptos fundamentales de XML y su utilización.



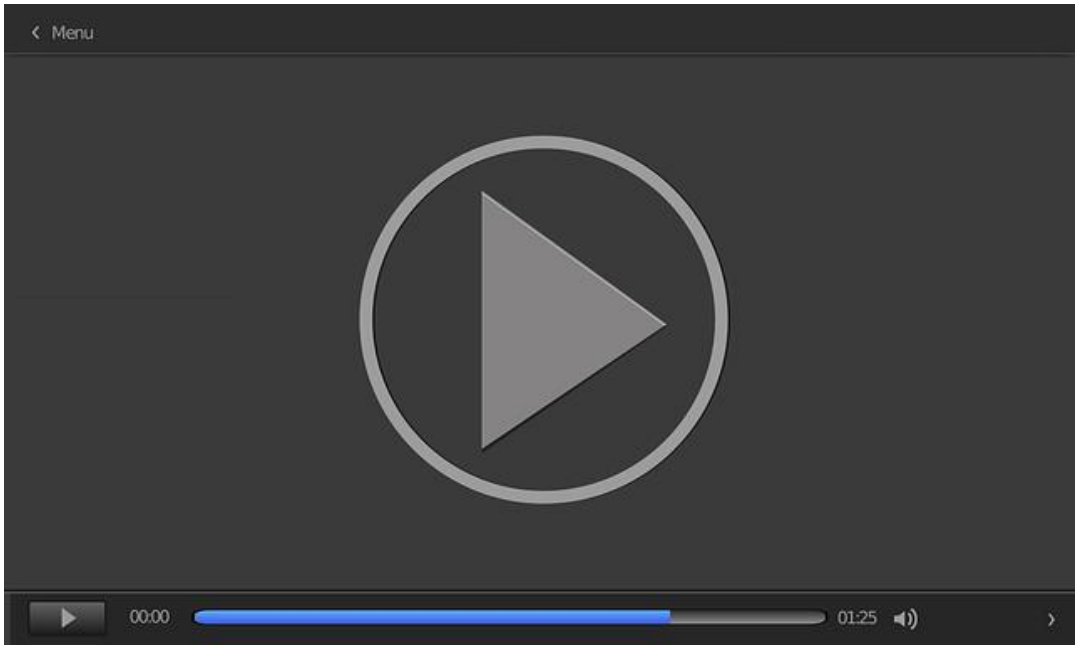
Accede al vídeo:

https://www.youtube.com/embed/PxGICnkFZJU

Leer y escribir archivos XML en JAVA con JAXB

Tec Gurus (2016, agosto 28) *Leer y escribir archivos XML en JAVA con JAXB* [Vídeo]. YouTube. https://www.youtube.com/watch?v=pTBVQlqw1A4

Este tutorial explica cómo leer y escribir archivos XML en Java utilizando JAXB.}



Accede al vídeo:

https://www.youtube.com/embed/pTBVQlqw1A4

Entrenamiento 1

Planteamiento del ejercicio

Crea un documento XML que represente una lista de estudiantes, donde cada estudiante tenga un nombre y una edad.

Desarrollo paso a paso

- **1. Declaración del encabezado XML:**
  - Indica que es un documento XML.
  - Define la **versión** (1.0) y la **codificación** de caracteres (UTF-8), que permite usar caracteres acentuados como en "García" o "López".

```
<?xml version="1.0" encoding="UTF-8"?>
```

- **2. Elemento raíz:**
  - estudiantes es el **elemento raíz** que agrupa a todos los nodos individuales de tipo estudiante.
  - En un documento XML válido solo puede haber **un elemento raíz**.

```
<estudiantes>

    ...

</estudiantes>
```

- **3. Primer estudiante:**
  - Cada bloque estudiante representa a un estudiante.
  - Contiene dos elementos hijos. Por un lado, <nombre> que contiene el nombre del estudiante. Por otro, <edad> que contiene su edad como valor numérico.

```
<estudiante>
<nombre>Ana García</nombre>
 <edad>20</edad>
</estudiante>
```

- **4. Segundo estudiante:** igual que el anterior, con datos distintos.

```
<estudiante>
    <nombre>Carlos López</nombre>
    <edad>22</edad>
</estudiante>
```

Solución

```
<?xml version="1.0" encoding="UTF-8"?>
<estudiantes>
    <estudiante>
        <nombre>Ana García</nombre>
        <edad>20</edad>
    </estudiante>
```

```
    <estudiante>
        <nombre>Carlos López</nombre>
        <edad>22</edad>
    </estudiante>
</estudiantes>
```

Entrenamiento 2

Planteamiento del ejercicio

Utiliza XSLT para transformar el documento XML del Ejercicio 1 en una lista HTML.

Desarrollo paso a paso

- **1. Declaración del documento:** indica que es un archivo XML y que usará codificación UTF-8.

```
<?xml version="1.0" encoding="UTF-8"?>
```

- **2. Declaración del espacio de nombres XSLT:**
  - Define que el documento es un **estilo XSLT** versión 1.0.
  - Usa el espacio de nombres estándar de XSLT (xmlns:xsl=...) .

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

- **3. Plantilla principal:**
  - Aplica esta plantilla al **nodo raíz** (/) del documento XML.
  - Lo que se defina dentro generará la salida HTML.

```
<xsl:template match="/">
```

- **4. Estructura HTML de salida:•Esta estructura fija genera un documento HTML con:**
  - Un título <h2> .
  - Una lista <ul> donde se insertarán los estudiantes.

```
<html>
    <body>
        <h2>Lista de Estudiantes</h2>
        <ul>
            ...
        </ul>
    </body>
</html>
```

- **5. Bucle sobre los estudiantes:**

```
<xsl:for-each select="estudiantes/estudiante">
```

- **6. Generar cada ítem de la lista:** por cada estudiante, crea un ítem `<li>` con su nombre y edad.

```
<li>
    <xsl:value-of select="nombre"/> - Edad: <xsl:value-of select="edad"/>
</li>
```

- **7. Cierre de etiquetas:** todas las etiquetas `<xsl:...>` y HTML se cierran correctamente para que el documento sea válido.

Solución

Archivo XSLT (transformacion.xsl):

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="/">

        <html>

            <body>

                <h2>Lista de Estudiantes</h2>

                <ul>

                    <xsl:for-each select="estudiantes/estudiante">

                        <li>

                            <xsl:value-of select="nombre"/> - Edad:
<xsl:value-of select="edad"/>

                        </li>

                    </xsl:for-each>

                </ul>

            </body>

        </html>

    </xsl:template>

</xsl:stylesheet>
```

Al aplicar esta transformación al documento XML del Ejercicio 1, se obtiene una lista HTML con los nombres y edades de los estudiantes.

Entrenamiento 3

Planteamiento del ejercicio

Utiliza expresiones XPath para seleccionar todos los nombres de los estudiantes mayores de veintiún años en el documento XML del Ejercicio 1.

Desarrollo paso a paso

- **1. Contexto: el XML de partida.** El XML sobre el que se aplica esta consulta es como este:

```
<estudiantes>
    <estudiante>
        <nombre>Ana García</nombre>
        <edad>20</edad>
    </estudiante>
    <estudiante>
        <nombre>Carlos López</nombre>
        <edad>22</edad>
    </estudiante>
</estudiantes>
```

- **2. Expresión XPath utilizada:**

```
//estudiante[edad > 21]/nombre
```

- **3. Explicación de cada parte:**
  - //: selecciona **todos los nodos** en el documento, sin importar dónde se encuentren.
  - estudiante: selecciona todos los elementos <estudiante> encontrados.
  - [edad > 21]: **filtro** (predicado) que selecciona solo aquellos estudiantes cuyo elemento <edad> tiene un valor **mayor a veintiuno**.
  - /nombre: después de haber filtrado los estudiantes, selecciona el elemento <nombre> que está **dentro** de cada uno de ellos.
- **4. Resultado de aplicar la expresión:** con el XML de ejemplo, el único estudiante con edad mayor a veintiuno es **Carlos López**, así que el resultado de la expresión sería:

```
<nombre>Carlos López</nombre>
```

- **5. Uso práctico.** Esta expresión XPath se puede usar en:
  - Editores XML como Oxygen, XMLSpy o Notepad++ con plugin XML Tools.
  - Programas Java o Python que usen bibliotecas, como javax.xml.xpath en Java o lxml o ElementTree en Python.

Solución

Expresión XPath:

```
//estudiante[edad > 21]/nombre
```

Esta expresión selecciona los elementos <nombre> de los estudiantes cuya edad es mayor a veinticinco años.

Entrenamiento 4

Planteamiento del ejercicio

Escribe un programa en Java que lea el documento XML del ejercicio uno y muestre en consola el nombre y la edad de cada estudiante.

Desarrollo paso a paso

- **1. Importación de clases necesarias:** se importan las clases del API DOM *(document object model)* para **leer, analizar y manipular documentos XML**.

```
import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;

import org.w3c.dom.Element;

import org.w3c.dom.NodeList;
```

- **2. Inicializar y configurar el analizador XML:**
  - DocumentBuilderFactory crea una **fábrica de analizadores**.
  - DocumentBuilder se usa para **parsear el archivo XML**.

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

DocumentBuilder builder = factory.newDocumentBuilder();
```

- **3. Cargar y *parsear* el documento XML:** se lee el archivo estudiantes.xml desde el disco y se convierte en un árbol DOM (estructura jerárquica de nodos XML).

```
Document doc = builder.parse("estudiantes.xml");
```

- **4. Normalizar el documento:**
  - Elimina nodos vacíos y estandariza el contenido.
  - Es una buena práctica para evitar problemas al recorrer nodos.

```
doc.getDocumentElement().normalize();
```

- **5. Obtener la lista de elementos <estudiante>:** devuelve todos los elementos <estudiante> del documento en un NodeList .

```
NodeList listaEstudiantes = doc.getElementsByTagName("estudiante");
```

- **6. Recorrer la lista y extraer los datos:**
  - Se recorre cada nodo <estudiante> .
  - Se accede a los subelementos <nombre> y <edad> , y se obtiene su contenido textual.
  - Se imprime cada estudiante con su nombre y edad.

```
for (int i = 0; i < listaEstudiantes.getLength(); i++) {

    Element estudiante = (Element) listaEstudiantes.item(i);

    String nombre =
estudiante.getElementsByTagName("nombre").item(0).getTextContent();

    String edad =
estudiante.getElementsByTagName("edad").item(0).getTextContent();

    System.out.println("Nombre: " + nombre + ", Edad: " + edad);

}
```

- **7. Manejo de errores:** captura cualquier excepción (por ejemplo, si el archivo no existe o tiene errores de sintaxis).

```
} catch (Exception e) {
    e.printStackTrace();
}
```

- **Ejemplo de entrada (** estudiantes.xml **):**

```
<estudiantes>
    <estudiante>
        <nombre>Ana García</nombre>
        <edad>20</edad>
    </estudiante>
    <estudiante>
        <nombre>Carlos López</nombre>
        <edad>22</edad>
    </estudiante>
</estudiantes>
```

- Salida esperada en consola:
  - Nombre: Ana García, edad: 20.
  - Nombre: Carlos López, edad: 22.

Solución

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
```

```
import org.w3c.dom.Document;

import org.w3c.dom.Element;

import org.w3c.dom.NodeList;

public class LeerEstudiantes {

    public static void main(String[] args) {

        try {

            // Crear una instancia de DocumentBuilderFactory

            DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();

            // Crear un DocumentBuilder

            DocumentBuilder builder = factory.newDocumentBuilder();

            // Parsear el documento XML

            Document doc = builder.parse("estudiantes.xml");

            // Normalizar el documento

            doc.getDocumentElement().normalize();

            // Obtener la lista de elementos "estudiante"

            NodeList listaEstudiantes =
doc.getElementsByTagName("estudiante");

            // Recorrer la lista de estudiantes

            for (int i = 0; i < listaEstudiantes.getLength(); i++) {

                Element estudiante = (Element) listaEstudiantes.item(i);
```

```
                String nombre =
estudiante.getElementsByTagName("nombre").item(0).getTextContent();

                String edad =
estudiante.getElementsByTagName("edad").item(0).getTextContent();

                System.out.println("Nombre: " + nombre + ", Edad: " +
edad);

            }

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}
```

Este programa lee el archivo estudiantes.xml, extrae los nombres y edades de los estudiantes y los imprime en la consola.

Entrenamiento 5

Planteamiento del ejercicio

Escribe un programa en Java que añada un nuevo estudiante al documento XML del Ejercicio 1 y guarde los cambios en el archivo.

Desarrollo paso a paso

- **1. Importación de clases necesarias**: se usan clases para:
  - Leer y manipular el XML (DocumentBuilder , Element , etc.).
  - Transformar y guardar el DOM modificado (Transformer , StreamResult , etc.).

```
import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import javax.xml.transform.*;

import javax.xml.transform.dom.DOMSource;

import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;

import org.w3c.dom.Element;
```

- **2. Cargar el archivo XML:**
  - Se carga el archivo estudiantes.xml como un documento DOM.
  - Se normaliza el árbol para facilitar su manipulación.

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

```
DocumentBuilder builder = factory.newDocumentBuilder();

Document doc = builder.parse("estudiantes.xml");

doc.getDocumentElement().normalize();
```

- **3. Crear el nuevo nodo** <estudiante> **:** se crea el nuevo elemento raíz del estudiante.

```
Element nuevoEstudiante = doc.createElement("estudiante");
```

- **4. Crear y añadir el nombre:** se crea el subelemento <nombre>, se le asigna un texto, y se añade al nuevo estudiante.

```
Element nombre = doc.createElement("nombre");

nombre.appendChild(doc.createTextNode("Laura Martínez"));

nuevoEstudiante.appendChild(nombre);
```

- **5. Crear y añadir la edad:** igual que con el nombre, pero para el elemento <edad> .

```
Element edad = doc.createElement("edad");

edad.appendChild(doc.createTextNode("23"));

nuevoEstudiante.appendChild(edad);
```

- **6. Insertar el nuevo estudiante en el documento:** se añade el nuevo nodo <estudiante> al elemento raíz <estudiantes> .

```
doc.getDocumentElement().appendChild(nuevoEstudiante);
```

- **7. Guardar el documento actualizado:** se usa un transformador para sobrescribir el archivo estudiantes.xml con el nuevo contenido DOM.

```
TransformerFactory transformerFactory = TransformerFactory.newInstance();
```

```
Transformer transformer = transformerFactory.newTransformer();

DOMSource source = new DOMSource(doc);

StreamResult result = new StreamResult("estudiantes.xml");

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

- **8. Mensaje de éxito:** se informa al usuario de que la operación se realizó correctamente.

```
System.out.println("Nuevo estudiante añadido con éxito.");
```

- Resultado final esperado en estudiantes.xml:

Antes:

```
<estudiantes>

    <estudiante>

        <nombre>Ana García</nombre>

        <edad>20</edad>

    </estudiante>

</estudiantes>
```

Después:

```
<estudiantes>

    <estudiante>

        <nombre>Ana García</nombre>

        <edad>20</edad>
```

```
    </estudiante>
    <estudiante>
        <nombre>Laura Martínez</nombre>
        <edad>23</edad>
    </estudiante>
</estudiantes>
```

Solución

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
public class AnadirEstudiante {
    public static void main(String[] args) {
        try {
            // Crear una instancia de DocumentBuilderFactory
            DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
```

```
            // Crear un DocumentBuilder

            DocumentBuilder builder = factory.newDocumentBuilder();

            // Parsear el documento XML

            Document doc = builder.parse("estudiantes.xml");

            // Normalizar el documento

            doc.getDocumentElement().normalize();

            // Crear un nuevo elemento "estudiante"

            Element nuevoEstudiante = doc.createElement("estudiante");

            // Crear y añadir el elemento "nombre"

            Element nombre = doc.createElement("nombre");

            nombre.appendChild(doc.createTextNode("Laura Martínez"));

            nuevoEstudiante.appendChild(nombre);

            // Crear y añadir el elemento "edad"

            Element edad = doc.createElement("edad");

            edad.appendChild(doc.createTextNode("23"));

            nuevoEstudiante.appendChild(edad);

            // Añadir el nuevo estudiante al elemento raíz

            doc.getDocumentElement().appendChild(nuevoEstudiante);

            // Guardar los cambios en el archivo XML

            TransformerFactory transformerFactory =
TransformerFactory.newInstance();
```

```
            Transformer transformer = transformerFactory.newTransformer();
            DOMSource source = new DOMSource(doc);
            StreamResult result = new StreamResult("estudiantes.xml");
            transformer.transform(source, result);
            System.out.println("Nuevo estudiante añadido con éxito.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Este programa añade un nuevo estudiante al archivo estudiantes.xml y guarda los cambios, actualizando el documento XML con la nueva información.