Tema 2

Almacenamiento en ficheros XML.

2.1. Persistencia en XML.

Parser (o procesador XML): interpreta la estructura del archivo XML y la carga en memoria.

Los *parsers* más conocidos son:

- DOM (Document Object Model).
- SAX (Simple API for XML).
- StAX (Streaming API for XML):
  - Para Java.
  - Procesa elemento a elemento. Utiliza un cursor que marca dónde nos encontramos –> la aplicación debe mantener estado entre eventos.
  - Es la aplicación la que controla la entrega de datos.
- XStream:
  - librería open source (Licencia BSD) para serializar y deserializar objetos Java a XML.
  - No es necesario especificar cómo se mapea el objetos y a XML (aunque lo permite).
  - Serializa todos los campos (incluidos privados, clases, etc.).
  - No es necesario que la clase tenga constructor vacío.
  - Soporte para JSON.
- JAXB (Jakarta XML Binding[1]

[1] Anteriormente conocida como Java Architecture for XML Binding)

  - API más completa.
  - Parte de Eclipse Enterprise for Java (EE4J)[2].
  - Funcionalidades como crear clases desde XSD.
- Apache XMLBeans:
  - Similar a JAXB en funciones, rendimiento, etc.

2.2. XML con DOM.

Características:

- Independiente del lenguaje. Hay implementaciones para los más conocidos.
- Carga todo el documento en un grafo en memoria –> consume demasiada.
- Una vez cargado se recorre el grafo como se quiera.
- Alto consumo de memoria –> usarlo si no hay más opciones.

Nos interesan dos paquetes:

- `org.w3c.dom`
- `javax.xml.parsers`

Clases fundamentales:

- `DocumentBuilderFactory`
- `DocumentBuilder`

Interfaces importantes:

- `Document`: objeto que equivale a un ejemplar de un documento XML.
- `Node`: cualquier nodo del documento. Superinterfaz de todas las siguientes:
  - `Element`: cada elemento del documento XML. Define propiedades y métodos para manipular los elementos del documento y sus atributos.
  - `Attr`: atributos de un nodo.
  - `Text`: contenido textual de un nodo. El nodo puede además contener hijos si es de tipo mixto.
  - `CharacterData`: los datos carácter presentes en el documento. Proporciona atributos y métodos para manipular caracteres.

[2] Anteriormente Java EE.

    - `DocumentType`: información contenida en la etiqueta `<!DOCTYPE>`.
- `NodeList`: lista de nodos.

La interfaz `Node` tiene muchos componentes útiles, por ejemplo las constantes que permiten identificar el tipo de nodo que es. ¡Échales un vistazo!

Lo mismo pasa con los métodos, por lo que voy a listar algunos importantes. Esto no debería impedir que mires la documentación para conocer todos los que incluye:

- `appendChild(Node newChild)`
- `getAttributes()`
- `getChildNodes()`
- `getFirstChild()`
- `getLastChild()`
- `getNextSibling()`
- `getNodeName()`
- `getNodeType()`
- `getNodeValue()`
- `getParentNode()`
- `getPreviousSibling()`
- `getTextContent()`
- `hasAttributes()`
- `hasChildNodes()`
- `insertBefore(Node newChild, Node refChild)`
- `isEqualNode(Node arg)`
- `isSameNode(Node other)`
- `normalize()`
- `removeChild(Node oldChild)`
- `replaceChild(Node newChild, Node oldChild)`
- `setNodeValue(String nodeValue)`
- `setTextContent(String textContent)`

En `Element` pasa lo mismo:

- `getAttribute(String name)`

- `getAttributeNode(String name)`
- `getElementsByTagName(String name)`
- `getTagName()`
- `hasAttribute(String name)`
- `removeAttribute(String name)`
- `setAttribute(String name, String value)`
- `setAttributeNode(Attr newAttr)`
- `setIdAttribute(String name, boolean isId)`
- `setIdAttributeNode(Attr idAttr, boolean isId)`

2.2.1. Leer documentos.

Antes de cualquier operación de lectura debemos obtener el *parser* y *parsear* el documento:

2.2.1.1. Obtener un *parser*.

Necesario para procesar el documento:

```
// Creamos el parser.
DocumentBuilderFactory builderFactory =
DocumentBuilderFactory.newInstance();
DocumentBuilder builder = null;

try {
   builder = builderFactory.newDocumentBuilder();
} catch (ParserConfigurationException ex) {
   // no se ha podido crear el parser
   ex.printStackTrace();
}
```

2.2.1.2. *Parsear* el documento.

```
// "parsear" el documento.
try {
   // deberíamos asegurarnos de que existe el fichero.
   Document document = builder.parse(new File("data\\carrera.xml"));

   // normalizamos el documento.
   document.getDocumentElement().normalize();
} catch (SAXException ex) {
   // si se produce algún error al parsear.
   ex.printStackTrace();
} catch(FileNotFoundException ex) {
   // Si el archivo no existe
   ex.printStackTrace();
} catch (IOException ex) {
   // Si se producen errores de E/S al parsear el archivo.
   ex.printStackTrace();
}
```

2.2.1.3. Normalizar el documento.

La sentencia

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

- Sirve para normalizar el nodo ≈procesar los espacios innecesarios.
- Lo hace para todo el árbol que cuelga de este nodo.
- Al hacerlo en el raíz –> todo el documento.

Si tenemos

```
<saludo>hola
a       to
dos</saludo>
```

Se representaría en un nodo no normalizado:


```
Element saludo
   Text node: "hola "
   Text node: "a "
   Text node: "to "
   Text node: "dos"
```

Al normalizarlo queda como:

```
Element saludo
   Text node: "hola a todos"
```

2.2.1.4. Ejemplos de operaciones de lectura.

Recorrer todos los hijos de un nodo (los hijos de este elemento raíz a su vez tienen hijos, etc. pero se deja el ejemplo sencillo):

```
//Recorrer los hijos de un nodo.
Element raiz = document.getDocumentElement();

NodeList hijos = raiz.getChildNodes();

for(int i=0; i<hijos.getLength(); i++) {
   Node hijo = hijos.item(i);

   if(hijo.getNodeType() == Node.ELEMENT_NODE) {
      Element elemHijo = (Element) hijo;
      System.out.println("Hijo"+i+": "+elemHijo.getTextContent());
   }
}
```

Sacar todos los nodos que se correspondan con una etiqueta y mostrar su contenido:


```
// leer por nombre de etiqueta.
NodeList vehiculos = document.getElementsByTagName("vehiculo");

for (int i = 0; i < vehiculos.getLength(); i++){
   Node vehiculo = vehiculos.item(i);

   if (vehiculo.getNodeType() == Node.ELEMENT_NODE) {
      Element elemento = (Element) vehiculo; // tipo de nodo
      System.out.println("Nombre: "+ getNodo("nombre", elemento));
      System.out.println("\tNúmero: "+ getNodo("numero", elemento));
      System.out.println("\tTiempo: "+ getNodo("tiempo", elemento));
   }
}
```

Usamos una función `getNodo()` que depende de que sean elementos, que el contenido sea texto, etc.

```
private static String getNodo(String etiqueta, Element elem) {
   NodeList lista =
      elem.getElementsByTagName(etiqueta).item(0).getChildNodes();

   Node valornodo = (Node) lista.item(0);
   return valornodo.getNodeValue();
}
```

Si necesitamos leer el valor de un atributo usaríamos:

```
String valorAtributo = element.getAttribute("nombreAtributo");
```

IMPORTANTE:

Vemos que el procesado de XML depende muchísimo de la estructura del documento. Por ello, si trabajamos con archivos que provienen de fuentes que no controlamos, sería muy conveniente validarlo con un esquema. No entraré en detalles, pero aquí puedes aprender.

2.2.1.5. Iterar sobre el archivo.

Podemos usar un iterador para recorrer el archivo.

El ejemplo solo muestra por pantalla el nombre del nodo pero puede usarse para obtener cualquier tipo de información.

```
// recorrer con iterador
DocumentTraversal trav = (DocumentTraversal) document;

Node elemRaiz = document.getDocumentElement();
NodeFilter filtrar = null; // no filtrar: coger todos los elmentos.
boolean expandirEntidades = true; // expandir entidades XML.

NodeIterator it = trav.createNodeIterator(elemRaiz,
   NodeFilter.SHOW_ELEMENT, filtrar, expandirEntidades);

for (Node node = it.nextNode(); node != null; node = it.nextNode()) {
   String name = node.getNodeName();

   System.out.println(name);
}
```

Ejercicios 2.1

1. Crea un programa que lea todos los nombres de los pilotos del archivo `carrera.xml`, les asigne un número de orden dentro de su vehículo y muestre por pantalla el número del vehículo, el número asignado y el nombre:

   P.ej: 6-2-Soldado Meekly

2. Crea un método estático que reciba la ruta de un fichero XML y devuelva el elemento `Document` normalizado.

   Modifica el ejercicio 2.1.1 para que utilice el método que acabas de crear.

3. Crea una clase `Participante` que permita cargar los datos de un participante del archivo `carrera.xml`.

   Sobrescribe el método `toString` para que se muestren los datos adecuadamente. Por ejemplo:


```
Participante:
   Código: 7-FFD-AMG
   Conductores:
      Sargento Blast
      Soldado Meekly
   Vehiculo:
      Nombre: El Súper Chatarra Special
      Número: 6
      Tiempo: 4H6M11S
```

En caso de que solo haya un conductor, debe poner *Conductor* en vez de *Conductores*.

Recuerda que puedes usar \n y \t para crear saltos de línea y tabulaciones en el `String`.

Modifica el ejercicio 2.1.2 para que utilice la clase que acabas de crear y muestre todos los datos de cada participante en lugar de solo lo que se pide en el ejercicio 2.1.1.

4. Crea un método que permita buscar en el archivo `carrera.xml` elementos por el nombre de uno de pilotos lo devuelva en un objeto de la clase `Participante`.

   Pruébalo desde el método `main` de tu proyecto.

2.2.2. Crear un fichero.

2.2.2.1. Preparación del documento.

Lo primero es crear un *builder* de forma similar a la lectura.

También necesitamos una implementación de DOM.

```
// creamos el builder que nos permiterá crear documentos, etc.
DocumentBuilderFactory builderFactory =
   DocumentBuilderFactory.newInstance();
DocumentBuilder builder = builderFactory.newDocumentBuilder();

// obtenemos una implementación del DOM
DOMImplementation implementation = builder.getDOMImplementation();
```


A continuación podemos crear el documento en sí. Aprovechamos para guardar el elemento raíz porque lo usaremos bastante.

```
// Creamos el documento
Document documento = implementation.createDocument(null,
   "jugadoresNFL", null);
// establecemos la version de XML
documento.setXmlVersion("1.0");

// Obtenemos el elemento raíz
Element raiz = documento.getDocumentElement();
```

2.2.2.2. Añadir elementos.

Aquí vamos a usar un ejemplo en el que añadimos jugadores de NFL que tienen los siguientes campos:

```
private String nombre;
private short numero;
private boolean lesionado;
private final GregorianCalendar fechaContrato;
```

Suponemos que están en una lista por lo que solo los vamos transformando a XML y añadiendo como hijos al nodo raíz.

Se podrían crear más niveles añadiendo a otros nodos en lugar de al nodo raíz.

```
// creamos nodos de jugadores y los añadimos al elemento raíz.
for(JugadorNFL jugador : jugadores) {
   Element jugXML = crearJugadorNFL(jugador, documento);
   raiz.appendChild(jugXML);
}
```

Hemos usado un método auxiliar que convierte el objeto a XML:


```java
/**
* Crea un elemento XML con los datos de un {@link JugadorNFL JugadorNFL}
*
* @param jugador El jugador del que obtendremos los datos.
* @param doc El documento XML para el que se crea el elemento.
* @return el elemento creado.
*/
public static Element crearJugadorNFL(JugadorNFL jugador,
      Document doc) {
   Element juga = doc.createElement("jugador");

   juga.setAttribute("lesionado", jugador.isLesionado()?"sí":"no");
   juga.appendChild(crearElemento("nombre", jugador.getNombre(), doc));
   //usamos un truquillo para convertir a String.
   juga.appendChild(crearElemento("numero", ""+jugador.getNumero(),
      doc));
   juga.appendChild(crearElemento("fechaContrato",
      jugador.fechaContrato2String(), doc));

   return juga;
}
```

Este método está usando a su vez otro que crea nodos de texto (etiqueta con contenido de tipo texto):


```java
/**
* Crea un elemento XML.
*
* @param nombre El nombre de la etiqueta del elemento a crear.
* @param valor El valor que contendrá.
* @param doc  El documento XML para el que se crea el elemento.
* @return el elemento creado.
*/
public static Element crearElemento(String nombre, String valor,
      Document doc) {
   Element elemento = doc.createElement(nombre);
   Text texto = doc.createTextNode(valor);

   elemento.appendChild(texto);

   return elemento;
}
```

2.2.2.3. Escribir el fichero.

El DOM de Java utiliza un `Transformer` para generar el archivo XML.

Se llama transformador porque puede usarse también para transformar el documento XML mediante el lenguaje XSLT, aunque nosotros lo usaremos solo para escribir el fichero XML.

Antes de realizar la transformación propiamente dicha, hay que preparar algunas cosas:


```
// Creamos el transformador.
TransformerFactory transformerFactory = TransformerFactory.newInstance();
Transformer transf = transformerFactory.newTransformer();

// establecemos codificación y pedimos que indente.
transf.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
transf.setOutputProperty(OutputKeys.INDENT, "yes");
transf.setOutputProperty("{http://xml.apache.org/xslt}indent-amount",
   "2");

// crea el árbol para una transformación.
DOMSource source = new DOMSource(documento);

// Creamos el archivo para guardar los datos.
File myFile = new File("jugadoresNFL.xml");
```

Luego creamos los flujos para mostrar el resultado y realizamos la transformación. En este caso vamos a usar dos: la salida estándar y el fichero:

```
// Mostrar los resultados por consola y escribirlos en el archivo.
StreamResult console = new StreamResult(System.out);
StreamResult file = new StreamResult(myFile);

transf.transform(source, console);
transf.transform(source, file);
```

Esto daría como resultado un fichero así:


```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<jugadoresNFL>
   <jugador lesionado="no">
      <nombre>Herbert</nombre>
      <numero>10</numero>
      <fechaContrato>2022-10-5</fechaContrato>
   </jugador>
   <jugador lesionado="sí">
      <nombre>Allen</nombre>
      <numero>13</numero>
      <fechaContrato>2022-10-5</fechaContrato>
   </jugador>
   <jugador lesionado="no">
      <nombre>James Jr.</nombre>
      <numero>3</numero>
      <fechaContrato>2022-10-5</fechaContrato>
   </jugador>
   <jugador lesionado="no">
      <nombre>Mahomes</nombre>
      <numero>15</numero>
      <fechaContrato>2022-10-5</fechaContrato>
   </jugador>
</jugadoresNFL>
```

2.2.2.4. El problema de lectura con Java 9.

Como se menciona aquí, en Java 9 hay un problema con los espacios, que al ser leídos, se interpretan como nodos de texto.

Esto no sucedía en versiones anteriores de Java así que es posible que se solucione en las próximas.

El árbol queda así:


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

   <participante>

      <codigo>23-FRT-FAM</codigo>

      <conductor>

         <nombre>Macana</nombre>

         <nombre>Piedro</nombre>

         <nombre>Roco</nombre>

      </conductor>
...
```

Y debería salir así:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<carrera>
   <participante>
      <codigo>23-FRT-FAM</codigo>
      <conductor>
         <nombre>Macana</nombre>
         <nombre>Piedro</nombre>
         <nombre>Roco</nombre>
      </conductor>
...
```

Hay muchos arreglos propuestos en Internet, pero el que creo más adecuado para nuestro caso se propone aquí de donde he cogido el archivo XSLT pero he adaptado un poco.

Aprovechamos que el `Transformer` creado tiene la capacidad de aplicar XSLT para hacerlo con esta:


```
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:strip-space elements="*"/>
   <xsl:output method="xml" encoding="UTF-8"/>

   <xsl:template match="@*|node()">
      <xsl:copy>
         <xsl:apply-templates select="@*|node()"/>
      </xsl:copy>
   </xsl:template>
</xsl:stylesheet>
```

Luego tenemos que crear el `Transformer` con él como argumento, cambiando la línea:

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

Por estas:

```
File xsltLimpiar = new File("limpiar.xslt");
StreamSource limpiarStream = new StreamSource(xsltLimpiar);
Transformer transf = transformerFactory.newTransformer(limpiarStream);
```

Así se aplicará la hoja de estilos al realizar la transformación.

Ejercicios 2.2

1. En la clase `Participante` debes crear un método que permita añadir más conductores.
2. Crea un método en la clase `Participante` para que se añada como hijo a un elemento de un documento XML. El elemento y el documento se recibirán como argumentos.
3. Usando el método del ejercicio 2.2.2 crea un `main` que añada varios participantes a un documento, lo escriba en un fichero y lo muestre por pantalla. Debe haber participantes con un único conductor y con varios conductores.
4. Crea un programa que repita el ejercicio 2.2.3 pero añadiendo los participantes a un árbol DOM que se haya leído de un archivo que ya contenía otros participantes. El resultado se debe escribir en el mismo fichero del que hemos leído, por ejemplo `carrera.xml`.

5. Ahora debes incluir un método que busque un participante por el código. Devolverá un objeto de tipo `Element` o `Node`.
6. Usa el método del ejercicio 2.2.5 para crear otro que elimine un participante sabiendo su código.
7. Crea una implementación de PokemonDAO que utilice un archivo XML como almacenamiento y lo maneje usando DOM.

2.3. XML con SAX.

Nos encontramos en la versión 2 en la que se han desaconsejado[3] algunas funciones de la versión 1.

Características:

- Independiente del lenguaje.
- Solo lectura.
- Lee elemento a elemento.
- Funcionamiento por eventos (inicio/fin del documento, inicio/fin de etiqueta) por lo que es el *parser* el que dirige la entrega de datos.
- Cada evento invoca un método que determina el programador.
- Procesamiento por elementos impide tener una visión global del documento –> no procesa bien los documentos en los que para procesar un elemento tienes que conocer detalles de otro (p.Ej: validación DTD o XSD o procesamiento XSLT, etc.) –> solución: almacenarlo en variables u objetos en el programa. Aún así seguiría siendo más fácil con DOM.
- En general se prefiere a DOM para XML en aplicaciones con gran cantidad de datos por eficiencia.

2.3.1. Eventos.

Por eventos de los que los más importantes son:

- startDocument
- startElement
- characters

[3] deprecated.

- comments
- endElement
- endDocument

A la izquierda se muestra un documento xml de ejemplo (alumnos.xml) y a la derecha el método que invocaría el evento que se produce en cada caso. La línea extra de la derecha es porque el documento termina después de cerrarse el elemento raíz:

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

```
startDocument()
   startElement()
      startElement()
         startElement()
            characters()
         endElement()
         startElement()
            characters()
         endElement()
      endElement()
      startElement()
         startElement()
            characters()
         endElement()
         startElement()
            characters()
         endElement()
      endElement()
   endElement()
endDocument()
```

2.3.2. Manejadores (*Handlers*).

Para procesar todos estos métodos existe una interfaz que se llama `ContentHandler`. Las clases que creemos para procesar el contenido de los ficheros XML la deben implementar.

Así mismo existen otras tres interfaces básicas:

- `DTDHandler`: para procesar eventos relacionados con la validación mediante DTD.
- `ErrorHandler`: para procesar eventos relacionados errores al *parsear*.

- `EntityResolver`: para intervenir en cómo se resuelven las entidades XML.

Afortunadamente existe la clase `DefaultHandler` que implementa todos estos métodos por lo que podemos basarnos en ella.

> **Nota:**
>
> Aunque estamos usando el `DefaultHandler` básico, existe otro (`DefaultHandler2`) que incorpora alguna funcionalidad de SAX2. Hereda de este y lo que incorpora no nos hace falta por lo que no se ha incluido en la documentación.

2.3.3. Crear un *Parser Handler* heredando de `DefaultHandler`.

Para determinar qué hacer en estos eventos, crearemos una clase que herede de `DefaultHandler` y sobrescriba los métodos correspondientes a los eventos que queramos tener en cuenta.

A modo de ejemplo muy sencillo vamos a procesar el siguiente documento:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<jugadoresNFL>
   <jugador lesionado="no">
      <nombre>Herbert</nombre>
      <numero>10</numero>
      <fechaContrato>2022-10-14</fechaContrato>
   </jugador>
   <jugador lesionado="sí">
      <nombre>Allen</nombre>
      <numero>13</numero>
      <fechaContrato>2022-10-14</fechaContrato>
   </jugador>
</jugadoresNFL>
```

Para entender cómo funciona hacemos un manejador que nos muestre dónde estamos. Hemos dejado sin hacer nada el método `characters` para ver únicamente el recorrido:

```
public class JugadorNFLHandlerRecorrer extends DefaultHandler {

   @Override
   public void startDocument() throws SAXException {
      System.out.println("Comienzo del documento XML");
   }

   @Override
   public void endDocument() throws SAXException {
      System.out.println("Final del documento XML");
   }

   @Override
   public void startElement(String uri, String localName, String qName,
         Attributes attributes) throws SAXException {
      System.out.println("\tPrincipio Elemento:" + qName);
   }

   @Override
   public void endElement(String uri, String localName, String qName)
         throws SAXException {
      System.out.println("\tFin Elemento:" + qName);
   }
}
```

Y lo usamos para procesar el fichero:


```
public static void main(String[] args) {
   SAXParserFactory saxParserFactory = SAXParserFactory.newInstance();
   try {
      SAXParser saxParser = saxParserFactory.newSAXParser();

      //Creamos una instacia del manejador que hemos hecho.
      JugadorNFLHandlerRecorrer handler =
         new JugadorNFLHandlerRecorrer();

      /* le decimos al parser que procese un archivo determinado con
         dicho manejador. */
      saxParser.parse(new File("jugadoresNFLReco.xml"), handler);
   } catch (ParserConfigurationException | SAXException |
         IOException e) {
      e.printStackTrace();
   }
}
```

Esto produciría la siguiente salida. Ten en cuenta que el método que procesa los caracteres, se ha hecho para que **no se muestren los nodos que solo contienen espacios**. Esto facilita la comprensión de lo que está pasando. Más adelante vemos el recorrido completo.


```
Comienzo del documento XML
   Principio Elemento:jugadoresNFL
   Principio Elemento:jugador
   Principio Elemento:nombre
   Fin Elemento:nombre
   Principio Elemento:numero
   Fin Elemento:numero
   Principio Elemento:fechaContrato
   Fin Elemento:fechaContrato
   Fin Elemento:jugador
   Principio Elemento:jugador
   Principio Elemento:nombre
   Fin Elemento:nombre
   Principio Elemento:numero
   Fin Elemento:numero
   Principio Elemento:fechaContrato
   Fin Elemento:fechaContrato
   Fin Elemento:jugador
   Fin Elemento:jugadoresNFL
Final del documento XML
```

Si añadimos un método `characters` que se salte todos los que correspondan con nodos sin texto (vacíos, saltos de línea, tabuladores, etc.):

```java
@Override
public void characters(char[] ch, int start, int length)
throws SAXException {
   String car = new String(ch, start, length);
   car = car.replaceAll("[\t\n]",""); // quitar saltos de línea

   // saltamos los elementos que solo contienen espacios.
   if(!car.trim().isEmpty()) {
      System.out.println("\tCaracteres : "+car);
   }
}
```

La salida obtenida proporcionaría más información:


```
Comienzo del documento XML
   Principio Elemento:jugadoresNFL
   Principio Elemento:jugador
   Principio Elemento:nombre
   Caracteres : Herbert
   Fin Elemento:nombre
   Principio Elemento:numero
   Caracteres : 10
   Fin Elemento:numero
   Principio Elemento:fechaContrato
   Caracteres : 2022-10-14
   Fin Elemento:fechaContrato
   Fin Elemento:jugador
   Principio Elemento:jugador
   Principio Elemento:nombre
   Caracteres : Allen
   Fin Elemento:nombre
   Principio Elemento:numero
   Caracteres : 13
   Fin Elemento:numero
   Principio Elemento:fechaContrato
   Caracteres : 2022-10-14
   Fin Elemento:fechaContrato
   Fin Elemento:jugador
   Fin Elemento:jugadoresNFL
Final del documento XML
```

Si queremos ver realmente cómo se recorre el documento (debido a los saltos de línea, indentación, etc. que contiene para que lo veamos bien los humanos) dejaríamos el método de procesar caracteres así:

```java
@Override
public void characters(char[] ch, int start, int length)
      throws SAXException {
   String car = new String(ch, start, length);
   car = car.replaceAll("[\t\n]",""); // quitar saltos de línea
   System.out.println("\tCaracteres : "+car);
}
```

Observa que un conjunto de espacios, tabuladores, saltos de línea, etc. producen un

elemento vacío:

```
Comienzo del documento XML
   Principio Elemento:jugadoresNFL
   Caracteres :
   Principio Elemento:jugador
   Caracteres :
   Principio Elemento:nombre
   Caracteres : Herbert
   Fin Elemento:nombre
   Caracteres :
   Principio Elemento:numero
   Caracteres : 10
   Fin Elemento:numero
   Caracteres :
   Principio Elemento:fechaContrato
   Caracteres : 2022-10-14
   Fin Elemento:fechaContrato
   Caracteres :
   Fin Elemento:jugador
   Caracteres :
   Principio Elemento:jugador
   Caracteres :
   Principio Elemento:nombre
   Caracteres : Allen
   Fin Elemento:nombre
   Caracteres :
   Principio Elemento:numero
   Caracteres : 13
   Fin Elemento:numero
   Caracteres :
   Principio Elemento:fechaContrato
   Caracteres : 2022-10-14
   Fin Elemento:fechaContrato
   Caracteres :
   Fin Elemento:jugador
   Caracteres :
   Fin Elemento:jugadoresNFL
Final del documento XML
```


2.3.4. Un *Handler* que cargue datos en una clase.

Para hacer que los datos se carguen en clases crearemos un manejador un poco más complejo.

Como cada evento va a realizar una parte del trabajo, necesitaremos una lista para guardarlos a todos y un `jugadorNFL` para ir guardando el que estemos procesando en este momento:

```
// en esta lista metemos todos los jugadores del documento.
private List <JugadorNFL> listaJugadores = null;
// aquí iremos añadiendo los datos del jugador actual.
private JugadorNFL jugador = null;
```

Al comenzar el documento la inicializamos:

```
@Override
public void startDocument() throws SAXException {
   listaJugadores = new ArrayList <JugadorNFL> ();
}
```

Además, cada nodo es diferente, por lo que en el método `characters` necesitaremos saber a qué se corresponde el texto que recibamos. En `startElement` vamos a indicarlo poniendo a `true` unas variables según donde nos encontremos.

Las variables serán:

```
//variables para saber en qué elemento estamos al procesar caracteres.
boolean enNombre = false;
boolean enNumero = false;
boolean enFecha = false;
```

Y el método para procesar cada elemento:

```
@Override
public void startElement(String uri, String localName, String qName,
      Attributes attributes) throws SAXException {
   boolean lesionado = false;

   if (qName.equalsIgnoreCase("jugador")) {
      jugador = new JugadorNFL();

      if(attributes.getValue("lesionado").equalsIgnoreCase("sí")) {
         lesionado = true;
      }

      jugador.setLesionado(lesionado);
   }
   else if (qName.equalsIgnoreCase("nombre")) {
      enNombre = true;
   }
   else if (qName.equalsIgnoreCase("numero")) {
      enNumero = true;
   }
   else if (qName.equalsIgnoreCase("fechaContrato")) {
      enFecha = true;
   }
}
```

Solo necesitamos hacer algo al cerrarse `jugador` en el XML:

```
@Override
public void endElement(String uri, String localName, String qName)
throws SAXException {
   if (qName.equalsIgnoreCase("jugador")) {
      listaJugadores.add(jugador);
   }
}
```

Y lo principal que es cuando se encuentran caracteres (contenido textual) en un elemento:


```
@Override
public void characters(char[] ch, int start, int length)
      throws SAXException {
   if(enNombre) {
      jugador.setNombre(new String(ch, start, length));
      enNombre =false;
   }
   else if(enNumero) {
      jugador.setNumero(Short.parseShort(
         new String(ch, start, length)));
      enNumero = false;
   }
   else if(enFecha) {
      String fechaTexto = new String(ch, start, length);
      String[] camposFecha = fechaTexto.split("-");

      int anio = Integer.parseInt(camposFecha[0]);
      int mes = Integer.parseInt(camposFecha[1]);
      int dia = Integer.parseInt(camposFecha[2]);

      GregorianCalendar fechaCont =
         new GregorianCalendar(anio, mes, dia);

      jugador.setFechaContrato(fechaCont);

      enFecha = false;
   }
}
```

En este caso no necesitamos hacer nada al cerrar el documento.

Ahora, desde el `main` podré pedir la lista de jugadores y procesarla:


```
public static void main(String[] args) {
   List <JugadorNFL> listaJugadores = null;

   SAXParserFactory saxParserFactory = SAXParserFactory.newInstance();
   try {
      SAXParser saxParser = saxParserFactory.newSAXParser();

      //Creamos una instacia del manejador que hemos hecho.
      JugadorNFLHandler handler = new JugadorNFLHandler();

      /* le decimos al parser que procese un archivo determinado con
          dicho manejador.*/
      saxParser.parse(new File("jugadoresNFL.xml"), handler);

      // recuperamos la lista de jugadores que se hayan cargado.
      listaJugadores = handler.getListaJugadores();

      for(JugadorNFL jugador: listaJugadores) {
         System.out.println(jugador);
      }

   } catch (ParserConfigurationException | SAXException |
         IOException e) {
      e.printStackTrace();
   }
}
```

2.3.5. Un *Handler* para manejar errores.

El parser puede generar tres tipos de error:

- Error fatal: el *parser* no puede continuar por lo que si no se ha producido ninguna excepción, el *handler* por defecto de errores lanza una.
- Error: el *handler* por defecto no hace nada.
- Aviso (warning): el *handler* por defecto no hace nada.

Podemos crear nuestra propia gestión:

- Haciendo una clase específica que emplemente `ErrorHandler`. que define tres métodos.:

  - `error(SAXParseException exception)`

  - `fatalError(SAXParseException exception)`
  - `warning(SAXParseException exception)`

- Haciendo que la misma clase que implementa `ContentHandler`, implemente también `ErrorHandler`.
- `DefaultHandler` ya implementa esa interfaz así que podemos heredar y sobrescribir los métodos.

Vamos a hacerlo por separado para poder independizar la gestión de contenido y la de errores.

```
public class JNFLErrorHandler implements ErrorHandler {

   @Override
   public void warning(SAXParseException exception)
         throws SAXException {
      System.out.println("Warning: " +
         getParseExceptionInfo(exception));
   }

   @Override
   public void error(SAXParseException exception) throws SAXException {
      String mensaje = "Error: " + getParseExceptionInfo(exception);
      System.out.println();
      throw new SAXException(mensaje);
   }

   @Override
   public void fatalError(SAXParseException exception)
         throws SAXException {
      String mensaje = "Fatal Error: " +
         getParseExceptionInfo(exception);
      throw new SAXException(mensaje);
   }
}
```

Que se sirve de un método auxiliar para mostrar información de la excepción:


```
private String getParseExceptionInfo(SAXParseException exception) {
   String systemId = exception.getSystemId();

   if (systemId == null) {
      systemId = "null";
   }

   String info = "URI=" + systemId + " Line="
   + exception.getLineNumber() + ": " + exception.getMessage();

   return info;
}
```

Por lo que hay que gestionar todo de forma un poco diferente. Si hubiéramos creado una clase que implementara ambas interfaces podríamos usar el método anterior.


```
public static void main(String[] args) {
   List <JugadorNFL> listaJugadores = null;

   SAXParserFactory saxParserFactory = SAXParserFactory.newInstance();
   try {
      SAXParser saxParser = saxParserFactory.newSAXParser();

      //Creamos una instacia ambos manejadores.
      JugadorNFLHandler handlerContenido = new JugadorNFLHandler();
      JNFLErrorHandler handlerErrores = new JNFLErrorHandler();

      // Creamos un lector al que asociamos los manejadores.
      XMLReader xmlReader = saxParser.getXMLReader();
      xmlReader.setContentHandler(handlerContenido);
      xmlReader.setErrorHandler(handlerErrores);

      // Establecemos el archivo como un origen.
      //InputSource archivoInput = new InputSource("jugadoresNFL.xml");
      InputSource archivoInput =
         new InputSource("jugadoresNFLError.xml");

      //parseamos el documento.
      xmlReader.parse(archivoInput);

      // recuperamos la lista de jugadores que se hayan cargado.
      listaJugadores = handlerContenido.getListaJugadores();

      for(JugadorNFL jugador: listaJugadores) {
         System.out.println(jugador);
      }

   } catch (ParserConfigurationException | SAXException |
         IOException e) {
      e.printStackTrace();
   }
}
```


Validación XSD:

Aunque no lo haremos nosotros, esta gestión de errores se utiliza mucho en conjunto con la validación con esquemas XSD. Aquí puedes ver cómo se hace y aquí puedes profundizar.

Ejercicios 2.3

1. Crea una clase que herede de `DefaultHandler` para procesar el archivo `carrera.xml`. Ten en cuenta que ahora hay más niveles para procesar.

   En el *main* muestra el resultado de leer el archivo.

2. Añade un método `minConductores(int minimo)` que establezca cuántos conductores como mínimo debe tener cada participante que se vaya a leer. Si no se ha llamado al método, el mínimo será 0.

   Debes modificar el *handler* para que cargue solo los participantes que tengan al menos ese número de conductores.

3. Crea una clase que implemente `ErrorHandler` que escriba los errores en un archivo de texto plano. Funcionará como un *log* por lo que si ya había errores en el archivo, no se borrarán sino que se añadirán los nuevos al final.

   Los posibles errores deben ir en una línea cada uno y tendrán el siguiente formato:

   `Tipo: (Warning|Error|Fatal error) - Fecha y hora: (la fecha y hora cuando se produce) - Línea: (nLínea) - Mensaje: (mensaje de la excepción)`

   que escriba en un fichero (haciendo un log) y pongan el tipo de error (mirar qué produce un error, qué un warning, etc.), la fecha y hora en la que se produce y un mensaje explicativo

---

2.4. Bibliografía.

- Carlos Tessier Fernández (2021). Curso Acceso a Datos.
- Curso MEFP-IFCS02-AD. Ministerio de Educación y Formación Profesional.
- Santiago Faci y Fernando Valdeón (2022). Curso Acceso a datos. abrilCode.
- Documentación de la API de Java.
- Documentación de JAXB. Oracle.
- Java XML Tutorial. Jakob Jenkov.
- Normalization in DOM parsing with java - how does it work?. Stack Overflow.

- Java DOM. ZetCode.
- How to write XML file in Java – (DOM Parser). Mkyong.com
- Java SAX Tutorial . Java Guides.