Unidad 1. Manejo de ficheros

2º DAMS - Acceso a Datos

Documentación Java

Está disponible la documentación de Java con todos sus paquetes, todas sus clases, métodos, etc. Ante dudas de cómo funciona algo, consultar ahí primero.

Para la versión 8 de Java, la documentación está disponible en:
https://docs.oracle.com/javase/8/docs/api/overview-summary.html

Para la versión 20 está en:
https://docs.oracle.com/en/java/javase/20/docs/api/index.html

1. Clases asociadas a las operaciones de Gestión de ficheros

La clase **File** proporciona un conjunto de utilidades relacionadas con los ficheros que nos van a proporcionar **información** acerca de los mismos. También se puede utilizar para **crear** un nuevo directorio o una trayectoria de directorios completa si esta no existe.

Para crear un objeto File, se puede utilizar cualquiera de los tres constructores siguientes:

- File f = new File(String pathCompleto)
- File f = new File(String directorio, String fichero)
- File f = new File(File directorio, String Fichero)

File: Algunos métodos importantes

- **list**() devuelve un array de String con los ficheros y directorios.
- **listFiles**() devuelve un array de objetos File con los ficheros y directorios.
- **getName**() devuelve el nombre del fichero o directorio
- **getPath**() devuelve el camino relativo
- **getAsolutepath**() devuelve el camino absoluto del fichero/directorio
- **canRead**() / **canWrite**() true si el fichero se puede leer / escribir
- **length**() tamaño del fichero en bytes
- **createNewFile**() crea un fichero vacío siempre y cuando no exista ya
- **Mkdir**() crea el directorio pasado en el constructor
- … **delete**(), **exists**(), **getParent**(), **isDirectory**(), **isFile**(), **mkdir**(), **renameTo**()

Documentación File

Java 8

https://docs.oracle.com/javase/8/docs/api/java/io/File.html

Java 20

https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/io/File.html

Práctica 1

Realiza un programa Java que:

- Muestre los ficheros de un directorio. Primero con list y luego con listfile (indica por pantalla cuándo es cada caso, es decir que se vea “Mostrando ficheros con list:” y luego “Mostrando ficheros con listfile:”)
- El nombre del directorio se pasará desde la línea de comandos al ejecutarlo.
- Si el directorio no existe mostrará un mensaje por pantalla indicándolo.

Recordatorio para ejecutar un archivo java desde la consola de comandos:

- Navega hasta la carpeta con el archivo ejemplo.java
- Ejecuta el comando ‘javac ejemplo.java’ para lanzar el compilador javac.exe
- Tras haber creado el archivo ejemplo.class en bytecode, ejecuta el interprete mediante: ‘java ejemplo’

Práctica 2

Realiza un programa Java que cree tres niveles de directorios en el directorio base que se pase por línea de comandos. En el último nivel crea el fichero ‘MiFichero.txt’.

Ejemplo: Con el directorio base ‘Documentos’, el resultado será:

Documentos -> PrimerDirectorio -> SegundoDirectorio -> TercerDirectorio -> MiFichero.txt

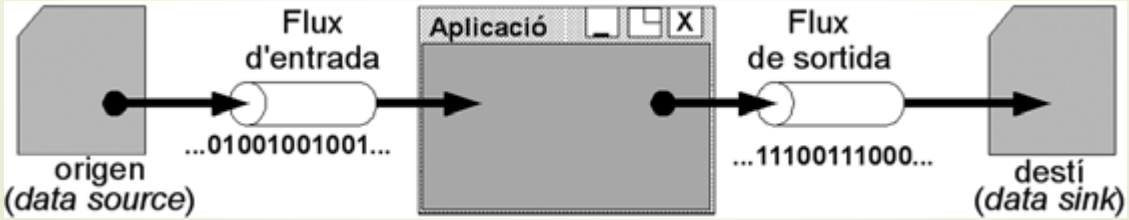
2. Flujos o streams

El sistema de entrada/salida en Java presenta una gran cantidad de clases que se implementan en el paquete **java.io**.

Usa la abstracción del flujo (**stream**) para tratar la comunicación de información entre una fuente y un destino.

Desde el punto de vista de la aplicación se pueden generar dos tipos de flujos:

- De entrada: Sirven para leer datos desde un origen para ser procesados
- De salida: Son los responsables de enviar los datos a un destino



Documentación Java.io

Java 8

https://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html

Java 20

https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/io/package-summary.html

2. Flujos o streams: Tipos

Cualquier programa que tenga que obtener información de cualquier fuente necesita **abrir un stream** y se escribirá la información **en serie**. Se definen dos tipos de flujos:

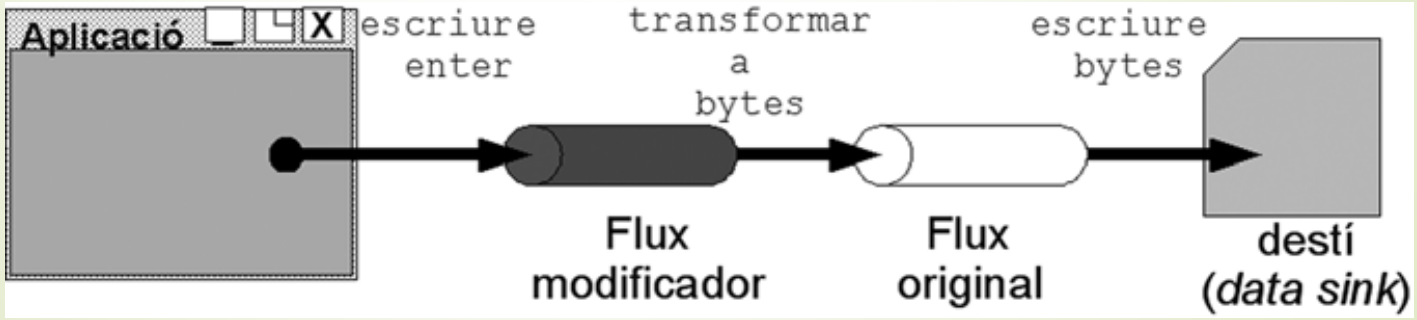
- **Flujos de datos o binario**: Los datos procesados se interpretan como bytes, orientado a la lectura/escritura de datos binarios. Opera con el tipo primitivo byte
- Clases **InputStream** / **OutputStream**
- **Flujos de caracteres** (16 bits): Los datos procesados se interpretan como texto (UNICODE funciona con 16 bits). Opera con el tipo primitivo char
- Clases **Reader** y **Writer**.

La excepción vinculada a errores de E/S definida en el paquete java.io es IOException

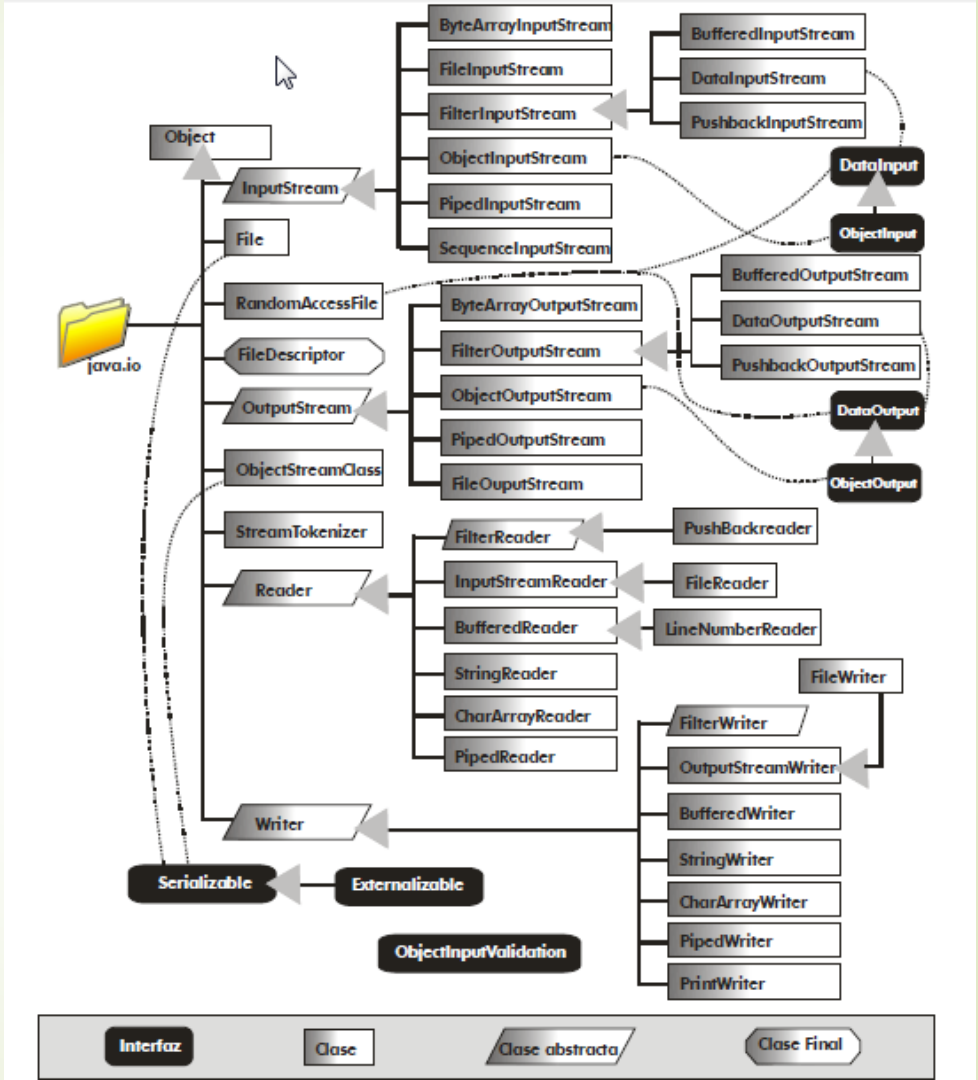
2. Flujos o streams: Filtros o modificadores de flujo

- Una clase modificadora de un flujo altera su funcionamiento por defecto y proporciona métodos adicionales que permiten el pre-procesado de datos complejos antes de escribirlas o leerlas del flujo

- Este pre-proceso se realiza de manera transparente al desarrollador



Clases de java.io



Algunas de las clases más relevantes de java.io

	bytes	Caracteres
lectura	InputStream FilterInputStream BufferedInputStream DataInputStream FileInputStream ObjectInputStream	Reader BufferedReader InputStreamReader FileReader
escritura	OutputStream FilteroutputStream BufferedOutputStream DataOutStream PrintStream FileOutputStream ObjectOutputStream	Writer BufferedWriter OutputStreamReader FileWriter

2.1 Flujos de Bytes

- **InputStream:** De esta clase heredan subclases que producen entradas de distintas fuentes. Todas tienen su correspondiente Output para la salida.

  - **ByteArrayInputStream**: Permite usar un espacio de almacenamiento intermedio de memoria.

  - **StringBufferInputStream**: Convierte un String en un InputStream.

  - **FileImputStream**: Flujo de entrada hacia fichero, lo usaremos para leer información de un fichero.

  - **PipedInputStream**: Implementa el concepto de "tubería".

  - **FilterInputStream**: Clase abstracta. Proporciona funcionalidad útil a otras clases InputStream.

    - **BufferInputStream**: permite obtener un buffer de datos en memoria sobre el que volcar los datos de un stream, y viceversa.

    - **DataInputStream**: permite convertir un stream de bytes en uno de los tipos primitivos de java, viceversa.

  - **ObjectInputStream**: convierte un stream de bytes en un objeto (clase), y permite guardar sus atributos.

  - **SequenceInputStream**: Convierte dos o más objetos InputStream en un InputStream único.

2.1 Flujos de Bytes

Un ejemplo clásico de uso de **InputStream** y **OutputStream** es la copia de un fichero en otro:

```
void copia (String origen, String destino) throws IOException {
    InputStream in= new FileInputStream(origen);
    OutputStream out= new FileOutputStream(destino);
    byte[] buffer= new byte[256]; //buffer 256 bytes, aunque se leerá por bytes
    while (true) {
      int n= in.read(buffer);     //lee un byte del origen
      if (n < 0)
        break;
      out.write(buffer, 0, n);
    }
    in.close();
    out.close();
  }
```

2.2 Flujos de caracteres

Las clases **Reader** y **Writer** manejan flujos de caracteres Unicode.

Hay ocasiones en las que hay que usar las clases que manejan bytes en combinación con las clases que manejan caracteres y que convierten stream a caracteres:

- Clases puente -> InputStreamReader / OutputStreamReader)

FileInputReader -> FileReader

FileOutputReader -> OutputReader

2.2 Flujos de caracteres

- **Reader:** De esta clase heredan subclases que producen entradas de caracteres. Todas tienen su correspondiente Writer para la salida. Las más importantes por su utilidad están subrayadas.

  - **BufferedReader**: Crea un buffer a partir de un flujo de caracteres de entrada.

  - **CharArrayReader**: Crea un stream a partir de un array de caracteres

  - **InputStreamReader**: Crea un stream de caracters a partir de uno de bytes.

- **FileReader**: para acceso a ficheros y lectura de caracteres.

  - **FilterReader**: Clase abstracta. Proporciona funcionalidad útil a otras clases Reader.

  - **PipedReader**: Implementa el concepto de “tubería” de caracteres.

  - **StringReader**: Crea un stream de caracteres a partir de un string.

3. Formas de acceso a un fichero

Hay dos formas de acceso a la información:

-**Acceso secuencial**: los datos o registros se leen y se escriben en orden. Se utilizan en soportes como cintas. Si se quiere acceder a un dato que está hacia la mitad del registro es necesario leer antes todos los anteriores.
En java se suelen utilizar las clases **FileInputStream** y **FileReader**, y las correspondientes para la salida.

-**Acceso directo o aleatorio**: permite acceder directamente a un dato o registro. Se utilizan en soportes como discos o memorias de semiconductores. Los datos están almacenados en registros de tamaño conocido. En java, la clase **RandomAccessFile** proporciona acceso aleatorio.

4. Operaciones sobre ficheros

- Creación
- Apertura
- Cierre
- Lectura de los datos
- Escritura de los datos

Las operaciones típicas una vez abierto el fichero son: Altas (añadir registros), Bajas (eliminar registros), Modificaciones y Consultas

5. Clases para gestión de flujos de datos desde/hasta ficheros

Dos tipos de ficheros (Java):

- **Texto**: almacenan caracteres legibles en formato estándar (ASCII, UNICODE, UTF8, etc).

- **Binarios**: almacenan bits, y por tanto cualquier tipo de dato. (int, float, boolean...), no son legibles, pero ocupan menos espacio en disco.

Ficheros de texto

- **FileReader**: clase para leer archivos de caracteres.
  - Con los siguientes constructores:
    - FileReader(File file)
    - FileReader(FileDescriptor fd)
    - FileReader(String fileName)
  - Tiene los siguientes métodos para la lectura de caracteres.:
  - int read( ): Lee un carácter y lo devuelve.
  - int read(char[ ] buf): Lee hasta buf.length caracteres de datos de una matriz de caracteres(buf). Los caracteres leídos del fichero se van almacenando en buf.
  - Int read(char[ ] buf, int desplazamiento, int n): Lee hasta n caracteres de datos de la matriz buf comenzando por buf[desplazamiento] y devuelve el número leído de caracteres.

Ficheros de texto

- **FileWriter**: clase para leer archivos de caracteres.
  - Con los siguientes constructores:
    - FileWriter(File file)
    - FileWriter(File file, boolean append)
    - FileWriter(FileDescriptor fd)
    - FileWriter(String fileName)
    - FileWriter(String fileName, boolean append)
  - Tiene los siguientes métodos para la escritura de caracteres.:
    - void write (int c): Escribe un carácter.
    - void write (char [ ] buf): Escribe un array de caracteres.
    - void write (char [ ] buf, int desplazamiento, int n): Escribe n caracteres de datos en la matriz buf comenzando por buf[desplazamiento].
    - void write (String str): Escribe una cadena de caracteres.
    - void append (char c): Añade un carácter a un fichero.

Ficheros binarios

- **FileInputStream**: clase para leer archivos binarios.
  - Con los siguientes constructores entre otros:
  - InputStreamReader(InputStream in): con juego de caracteres por defecto.
  - InputStreamReader(InputStream in, Charset cs): con juego de caracteres específico.
  - Tiene los siguientes métodos para la lectura de datos.:
  - int read (): Lee un byte.
  - int read (byte [ ] b): Lee hasta b.length bytes de datos de una matriz de bytes.
  - int read (byte [ ] b, int desplazamiento, int n): Lee hasta n bytes de la matriz b comenzando por b[desplazamiento] y devuelve el número leído de bytes.

Ficheros binarios

- **FileOutputStream**: clase para leer archivos binarios.
  - Con los siguientes constructores:
    - FileOutputStream(File file)
    - FileOutputStream(File file, boolean append)
    - FileOutputStream(FileDescriptor fdObj)
    - FileOutputStream(String name)
    - FileOutputStream(String name, boolean append)
  - Tiene los siguientes métodos para la escritura de datos.:
    - void write (int b): Escribe un byte
    - void write (byte [ ] b): Escribe b.length bytes
    - void write (byte [ ] b, int desplazamiento, int n): Escribe n bytes a partir de la matriz de bytes de entrada comenzando por b[desplazamiento]

Ficheros binarios

- Para poder trabajar con tipos primitivos se utilizan las clases DataInputStream y DataOutputStream, con los métodos readXXX() y writeXXX(), donde XXX será Booean,Byte,Long,Char, etc.

Pasos para trabajar con ficheros

Para trabajar con un fichero siempre habrá que realizar los **tres pasos** siguientes:

1. Abrir el fichero Creando un objeto de una determinada clase de acceso a ficheros al que se le especifica el fichero que se quiere utilizar así como una serie de posibles opciones

2. Realizar las operaciones de lectura, escritura, posicionamiento, etc.

3. Cerrar el fichero Si el fichero no se cierra puede que no se guarden definitivamente algunos de los datos

Práctica 3

- Implementa un programa Java que copie el contenido revertido de un fichero a otro. Procura que el archivo a copiar tenga varias líneas.
- En un segundo paso se abrirá el archivo copiado, para añadir texto al final del mismo (tu nombre y apellidos).
- En la tercera parte, se abrirá el archivo copiado con el nuevo texto y se mostrará por pantalla línea a línea (utiliza para ello la clase BufferedReader)

Práctica 4

- Implementa un programa Java que guarde en un archivo diferentes los siguientes tipos de datos con la clase DataOutputStream:
  - Boolean
  - Int
  - Float
  - Double
  - Char
  - String
  - Byte
- Comprueba que en el archivo creado sólo es legible los que se ha escrito de los tipos char y String.
- Lee el contenido del fichero con DataInputStream y sácalo por pantalla.

6. Objetos serializables 1

Java nos permite guardar objetos en ficheros binarios. El objeto tiene que implementar la interfaz **Serializable** que permite guardar y leer objetos en ficheros binarios. Esto permite guardar un objeto y todos (o parte) de sus atributos (por ejemplo un array de empleados cada uno con nombre, dirección, teléfono, etc.)

La serialización permite tomar cualquier objeto que implemente Serializable y convertirlo en una secuencia de bits, con los métodos:

- Object readObject(): se utiliza para leer un objeto del ObjectInputStream. Puede lanzar las excepciones IOException y ClassNotFoundException.
- void writeObject(Object obj): se utiliza para escribir el objeto especificado en el ObjectOutputStream. Puede lanzar la excepción IOException.

ObjectInputStream y ObjectOutputStream son las clases con las que deberemos trabajar.

6. Objetos serializables 2

A tener en cuenta:

- Su en la clase que implementa Serializable se utiliza el modificador **transient** en un atributo, éste no se almacenará.
- El atributo **serialVersionUID** permite conocer la versión de un objeto serializado. Si no se fija de forma manual, java le pone un identificador de forma automática al serializar.
- Al leer un objeto serializado hay que tener en cuenta las excepciones que puede dar: **IOException**, **ClassNotFoundException**.

6. Objetos serializables 3

Un ejemplo: la clase persona la hacemos serializable:

```
public class Persona implements Serializable{
  private String nombre;
  private int edad;
  public Persona(){this.nombre=null;this.edad=null;}
  public Persona(String nombre, int
edad){this.nombre=nombre;this.edad=edad;}
}
```

En otra clase, que utilizase la clase persona, esta se guardaría a un fichero abriendo el flujo de salida:

```
 File fichero = new File("archivoPersona.dat");
 ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(fichero));
```

6. Objetos serializables 4

Y escribiendo el dato (podrían ser tantas personas u otras clases como queramos):

```
 oos.writeObject(persona);
```

También podríamos leerlo posteriormente con los correspondientes de lectura:

```
 ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(fichero));
 Persona = (Persona) ois.readObject();
```

Tan solo hay que tener en cuenta, que se debe hacer una conversión de tipos, y que si metemos diferentes clases, deberemos leerlas y hacer la conversión en el orden en que se almacenaron. Por último, cuando se guarda un objeto en un fichero se le añade una cabecera.
Si se abre un fichero para añadir más registros, se añade otra cabecera que al leerla dará una excepción. Solución: crear heredar la clase ObjectOutputStream y sobreescribir el método que añade la cabecera llamado writeStreamHeader para que no haga nada.

Práctica 5

Implementa un programa Java que defina un objeto Persona (nombre, apellido, edad, dni).

Dni no deberá serializarse.

En su método de entrada (main) deberá crear 2 instancias del objeto persona y serializarlas en un fichero.

A continuación, leerá el contenido del fichero y mostrará en consola los datos de las instancias de Persona leídas, incluida la versión del objeto serializado.

Controla las posibles excepciones y comprueba que dni está a null.

7. Trabajo con Ficheros XML

- XML (Extensible Markup Language) es un estándar publicado por el Worldwide Web Consortium (W3C), metalenguaje para la definición de lenguajes de marcado.

- Proporciona una gramática que define las reglas por las cuales se rigen los ficheros XML.

- Reglas:

1. Cada dato está marcado con una etiqueta. Ej: <alumno> </alumno>
2. No puede haber solapamiento de etiquetas. Ej: <alumno> <nombre> se debe cerrar en el orden </nombre> </alumno>
3. En las etiquetas se pueden incluir atributos. Los valores de los atributos deben estar entrecomillados.

7. Trabajo con Ficheros XML

- Un documento XML se caracteriza por:
  - Bien formado
  - Extensible: Permite ampliar el lenguaje con nuevas etiquetas
  - Fácil de leer
  - Autodescriptivo: La estructura de la información de alguna manera está definida dentro del mismo documento
  - Intercambiable: Portable entre distintas arquitecturas
  - Se necesita un procesador parser para leerlo e interpretarlo. Existen productos y versiones libres.

7. Trabajo con Ficheros XML

- Dos de los procesadores más utilizados son:
  - **DOM** (Document Object Model): Almacena toda la estructura del documento en memoria en forma de árbol con nodos padres, hijos y finales. Se puede utilizar tanto para XML, como para HTML. Creado por el W3C.
    - **Contras**: Consumo elevado de memoria y tiempo
    - **Pros**: Permite actualizaciones de los datos
  - **SAX** (Simple Api for XML): Lee un fichero XML de forma secuencial y produce una secuencia de eventos en función de la lectura. Cada evento invoca un método definido por el programador. API creada para Java que se convirtió en estándar de facto.
    - **Contras**: Se pierde visión global, no permite actualizaciones
    - **Pros**: No consume memoria

DOM vs SAX

- Es mejor DOM cuando:
  - Se requiere modificar la estructura del XML
  - Se comparte el documento en memoria con otras aplicaciones
  - El tamaño del documento no es muy grande
- Es mejor SAX cuando:
  - La tarea a realizar requiere mucha memoria y alto rendimiento
  - No es necesario recorrer la estructura completa del documento
  - Se requiere ir procesando los elementos del archivo a medida que van llegando

ESTRUCTURA DEL ARCHIVO

- Cualquier documento XML bien formado, puede ser representado mediante un árbol y tendrá cinco tipos de nodos:
  - Raiz/Documento: Es el nodo que alberga todo el documento en forma de nodos hijos.
  - Elemento: Es un nodo que posee un nombre que lo identifica
  - Texto: Representa el texto que irá precedido y seguido por una etiqueta
  - Comentario: <!-- texto del comentario -->
  - Instrucción: Este tipo de nodos serán interpretados por el cliente o servidor que lea el documento. No pueden tener ningún nodo hijo

XML con DOM

- Para poder trabajar con DOM en JAVA:
  - Paquete org.w3c.dom (contenido en el JDK)
  - Paquete javax.xml.parsers (API estándar de Java)
  - Paquete javax.xml.transform (Necesario para generar un fichero XML a partir de un árbol DOM)

XML con DOM

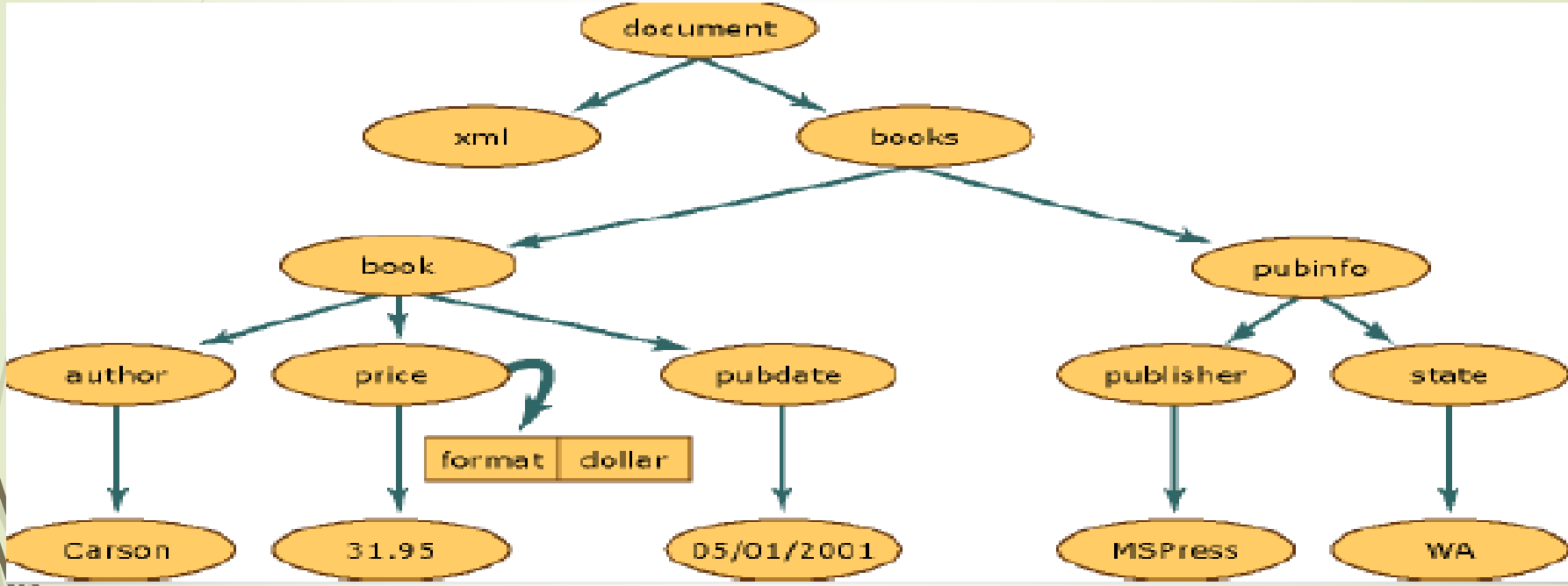
- Utilizaremos las siguientes interfaces (hay más):
  - Document: Representa el documento. Permite crear nuevos nodos
  - Element: Representa cualquier elemento del documento.
  - Node: Representa cualquier nodo del documento
  - NodeList: Contiene una lista con los nodos hijos de un nodo
  - Attr: Permite acceder a los atributos de un nodo
  - Text: Representa el texto almacenado
  - CharacterData: Proporciona atributos y métodos para manipular el texto
  - DocumentType: Proporciona información contenida en la etiqueta <!DOCTYPE>

Ejemplo XML con DOM

```
<?xml version="1.0" encoding="ISO-8859-1"?>          <!--Instrucción-->
<books>                                              <!--Elemento hijo de Document-->
    <book category="COOKING">                        <!--Elemento hijo de books con atributo-->
    <title lang="en">Everyday Italian</title>        <!--Elemento hijo de book con atributo y texto-->
    <author>Giada De Laurentiis</author>             <!--Elemento hijo de book con texto-->
    <year>2005</year>                                <!--Elemento hijo de book con texto-->
    <price>30.00</price>                             <!--Elemento hijo de book con texto-->
    </book>                                          <!--Cierre de book-->

    <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
    </book>
```

Estructura de memoria – XML con DOM



LEER XML con DOM

- Crearemos en memoria la estructura DOM siguiendo estos pasos:
  - 1. Crear una factoria que permite utilizar un parser:
    - DocumentBuilderFactory factoria = DocumentBuilderFactory.newInstance();
  - 2. Crear un builder que permite crear docs DOM utilizando un parser:
    - DocumentBuilder builder = factoria.newDocumentBuilder();
  - 3. Crear el doc. DOM a partir de un fichero XML:
    - Document document = builder.parse(new File(“mixml.xml”));

LEER XML con DOM

- Teniendo la estructura DOM en memoria, podemos pasar a analizar (parser) su contenido.
- Dos clases que nos serán muy útiles para ello son:
  - NodeList: es una lista de objetos Node.
  - Node: Representa un elemento del documento XML, el cual puede tener otros elementos (nodos).
- Los tipos de nodo que podemos encontrar en un XML son:
  - Documento: Contenedor de todos los nodos del árbol. También se conoce como la raíz del documento, que no siempre coincide con el elemento raíz.
  - DocumentFragment: Contenedor temporal de uno o varios nodos sin estructura de árbol.
  - DocumentType: Representa el nodo <!DOCTYPE…>.
  - EntityReference: Representa el texto de referencias a entidades sin expandir.
  - Elemento: Representa un nodo de elemento.
  - Attr: Atributo de un elemento.
  - ProcessingInstruction: Nodo de instrucción de procesamiento.
  - Comentario: Nodo de comentario.
  - Text: Texto que pertenece a un elemento o atributo.
  - CDATASection: Representa CDATA.
  - Entity: Representa las declaraciones <!ENTITY…> de un documento XML, desde un subconjunto de definición de tipo de documento (DTD) interno o desde DTD externas y entidades de parámetros.
  - Notation: Representa una notación declarada en la DTD.

LEER XML con DOM

- Y aquí podemos ver los posibles nodos que pueden "colgar" de otro nodo según el tipo, y el resultado que da el método name(), y su nombre, valor y tipo. Los tipos pueden ser Node.ELEMENT_NODE, Node.TEXT_NODE, etc.

Tipos Node	Posibles nodos hijo
Document	Element (=1), ProcessingInstruction, Comment, DocumentType
DocumentFragment	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	--
EntityReference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	Text, EntityReference
ProcessingInstruction	--
Comment	--
Text	--
CDATASection	--
Entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	--

Tipos Node	name	value	nodeType
Document	#document	null	9
DocumentFragment	#document fragment	null	11
DocumentType	nombre del tipo de documento	null	10
EntityReference	nombre de la entidad referenciada	null	5
Element	nombre de la etiqueta	null	1
Attr	nombre del atributo	valor del atributo	2
ProcessingInstruction	destino (*target*)	contenido íntegro exceptuando el *target*	7
Comment	#comment	contenido del comentario	8
Text	#text	contenido de nodo *text*	3
CDATASection	#cdata-section	contenido de la sección CDATA	4
Entity	nombre de la entidad	null	6
Notation	nombre de la notación declarada en el DTD	null	12

LEER XML con DOM

- Por último vemos algunos métodos (entre otros) útiles para analizar el contenido de un archivo XML:
  - **getElementsbyTagName**(): Obtiene recursivamente todos los elementos de un nodo. Si se hace sobre un tipo Document, obtiene todos los elementos del archivo XML. Permite filtrado por nombre de elemento.
  - **getChildNodes**(): Obtiene todos los elementos hijos de un nodo.
  - **getAttributes**(): obtiene todos los atributos de un nodo.
  - **getNodeName**(): devuelve el nombre del nodo.
  - **getNodeValue**(): devuelve el valor del nodo.
  - **getNodeType**(): devuelve el tipo de nodo (es una constante)
  - **getTagName**(): devuelve la etiqueta del nodo.

LEER XML con DOM

```
public static void main(String[] args) {
    try {
    //Crear una factoria que permita usar un parser:
    DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
    //Crear un builder que permite crear documentos DOM usando un parser:
    Document documento = builder.parse(new File("DAM_AD_UD01_ Manejo de ficheros.xml"));
    //Los nodos de texto adyacentes los fusiona
    documento.getDocumentElement().normalize();
    System.out.println("Elemento raiz:"+documento.getDocumentElement().getNodeName());
    //Crear una lista con todos los nodos libro:
    NodeList libros = documento.getElementsByTagName("book");
    //Recorrer la lista:
    for (int i = 0; i < libros.getLength(); i++) {
        Node libro = libros.item(i);
        if(libro.getNodeType() == Node.ELEMENT_NODE){
        Element elemento = (Element) libro;//Obtenemos los elementos del nodo
        if(libro.getNodeType() == Node.ELEMENT_NODE){
        System.out.println(elemento.getElementsByTagName("title").item(0).getNodeName()+                    //Nombre del elemento
                    "-"+((Element)(elemento.getElementsByTagName("title").item(0))).getAttribute("lang")+   //Atributo del elemento
                    "-"+elemento.getElementsByTagName("title").item(0).getTextContent());                   //Texto del elemento
        System.out.println(elemento.getElementsByTagName("author").item(0).getNodeName()
                    +"-"+elemento.getElementsByTagName("author").item(0).getTextContent());
        System.out.println(elemento.getElementsByTagName("year").item(0).getNodeName()
                    +"-"+elemento.getElementsByTagName("year").item(0).getTextContent());
        System.out.println(elemento.getElementsByTagName("price").item(0).getNodeName()
                +"-"+elemento.getElementsByTagName("price").item(0).getTextContent());
                    }
                }
            }
    } catch (Exception ex) {
        System.err.println("Error: "+ex.getMessage());
        }
    }
```

Escribir XML con DOM

- Lo primero es crear el documento. Para ello:
  - 1. Crear una factoria que permite utilizar un parser:
    - DocumentBuilderFactory factoria = DocumentBuilderFactory.newInstance();
  - 2. Crear un builder que permite crear docs DOM utilizando un parser:
    - DocumentBuilder builder = factoria.newDocumentBuilder();
  - 3. Crear una implementación:
    - DOMImplementation implementation = builder.getDOMImplementation();
  - 4. Crear el doc. DOM y asignarle la etiqueta “mixml”
    - Document document = implementation.createDocument(num, “mixml, null”);

Escribir XML con DOM

- Una vez creado el documento DOM lo rellenamos. Supongamos que estamos rellenando un documento con información sobre empleados como esta:

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

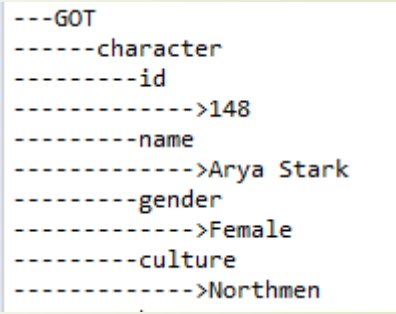
- Los pasos serían:
  - 1. Crear el nodo empleado:
    - `Element raiz = document.createElement("empleado");`
  - 2. Pegarlo a la raíz del documento:
    - `document.getDocumentElement().appendChild(raiz);`
  - 3. Añadir cada hijo al nodo raíz empleado, por ejemplo el id.
    - `Element elem = document.createElement("id"); //creamos un hijo`
    - `Text text = document.createTextNode("1"); //damos valor`
    - `raiz.appendChild(elem); //pegamos el elemento hijo a la raiz`
    - `elem.appendChild(text); //pegamos el valor`
  - 4. Esto se podría hacer con una función a la que se le pasara nombre, valor, raíz y documento.

Escribir XML con DOM

- - Una vez relleno, lo tenemos que guardar en el archivo XML siguiendo estos pasos:
  - 1. Crear una fuente DOM que rellenamos con la estructura DOM:
    - DOMSource fuente = new DOMSource(documento);
  - 2. Definir un destino:
    - Result resultado = new StreamResult(new java.io.File(“mixml.xml”));
  - 3.Crear un transformador
    - Transformer transformador = TransformerFactory.newInstance().newTransformer()
  - 4. Transformar
    - transformador.transform(fuente, resultado);

Práctica 6

- Utiliza el archivo XML con información sobre juego de tronos. Realiza un programa en Java que cargue los datos y los muestre por pantalla, incluido el nombre del elemento principal (GOT). Intenta sacar los datos por pantalla con cierta jerarquía, similar a esto:



- Intenta hacer la lectura del archivo utilizando los métodos:
  - getElementsbyTagName(). Al ser muy extenso sólo será necesario sacar de cada carácter: id, name y titles. (2 puntos)
  - getChildNodes() de forma iterativa hasta el nivel que necesites para sacar todo el documento, y sin getElementsbyTagName. (2 puntos)
  - getChildNodes() de forma recursiva y sin getElementsbyTagName, lo que no permitiría leer un documento XML sin conocer su estructura. (2 puntos)

- En la siguiente parte del código, añade a cada personaje el interprete en la serie con la etiqueta <playedBy>. (2 puntos obligatorio)
  - Los interpretes son:
  - Arya Stark: Alfie Allen
  - Brandon Stark: Isaac Hempstead-Wright
  - Rickon Stark: Art Parkinson
  - Robb Stark: Richard Madden
  - Sansa Stark: Sophie Turner

Práctica 06 (cont.)

- Verás que falta uno de los hermanos Stark. Haz que el programa complete XML con el carácter que falta y con la siguiente información (se omite información obvia) (2 puntos obligatorio):
  - Id: 583
  - Nombre: Jon Snow
  - Nacido: In 283 AC, at Winterfell
  - Vivo: Falso
  - Titulos: Lord Commander of the Night's Watch, King in the North
  - AliasLord Snow, Ned Stark's Bastard, The Snow of Winterfell, The Crow-Come-Over, The 998th Lord Commander of the Night's Watch, The Bastard of Winterfell, The Black Bastard of the Wall, Lord Crow
  - Libros: todos
  - Temporadas: todos
  - Actor:Kit Harington

- Por último, escribe el contenido del documento XML, con toda la información añadida (actores, personaje nuevo) en un archivo .xml. (obligatorio)

XML con SAX

- SAX (API Simple para XML) es un conjunto de clases e interfaces que ofrecen una herramienta muy útil para el procesamiento de documentos XML.
- Permite analizar los documentos de forma secuencial (bajo consumo de memoria).
- Es un API escrito en Java, incluido dentro de JRE. Esto permite crear un parser de XML propio.
- SAX fue concebido para lectura de archivos XML, no para su escritura.
- La lectura de un documento XML produce eventos que ocasionan la llamada a métodos:
  - **startDocument**: se produce al comenzar el procesado del documento xml.
  - **endDocument**: se produce al finalizar el procesado del documento xml.
  - **startElement**: se produce al comenzar el procesado de una etiqueta xml. Es aquí donde se leen los atributos de las etiquetas.
  - **endElement**: se produce al finalizar el procesado de una etiqueta xml.
  - **characters**: se produce al encontrar una cadena de texto.

XML con SAX

- Para poder trabajar con DOM en JAVA:
  - Paquete org.xml.sax.Attributes.
  - Paquete org.xml.sax.InputSource.
  - Paquete org.xml.sax.SaxExceptions.
  - Paquete org.xml.sax.XMLReader.
  - Paquete org.xml.sax.DefaultHandler.
  - Paquete org.xml.sax.XMLReaderFactory.

LEER XML con SAX

- La lectura se realiza con un procesador XML
  - XMLReader parseadorXML = XMLReaderFactory.createXMLReader();
- A continuación hay que indicar al XMLReader qué objetos poseen los métodos que tratarán los eventos: Esos objetos serán implementaciones de las siguientes interfaces:
  - **ContentHandler**: recibe notificaciones de los eventos que ocurren en el documento.
  - **DTDHandler**. Recoge eventos relacionados con la DTD.
  - **ErrorHandler**: define métodos de tratamiento de errores.
  - **EntityResolver**: sus métodos se llaman cada vez que se encuentra una referencia a una entidad.
  - **DefaultHandler**: clase que provee una implementación por defecto para todos sus métodos. De esta clase se harán las extensiones para crear el parser de XML.
  - Por ejemplo:
    - GestorEventosXML gestor = new GestorEventosXML(); //Esta clase la crearemos luego, sobreescribiendo los métodos startDocument, etc
    - parseadorXML.setContentHandler(gestor);
- Le indicamos el fichero XML:
  - InputSource archivoXML = new InputSource(“archivo.xml”);
- Finalmente, con el XMLReader, utilizamos el método parse, con el fichero XML:
  - parseadorXML.parse(archivoXML);

LEER XML con SAX

- Ya sólo nos quedaría crear la clase GestorEventosXML que hereda de DefaultHandler, sobreescribiendo los métodos que tratan los diferentes eventos:

```
class GestorEventosXML extends DefaultHandler {
    public GestorEventosXML() {super();}
    public void startDocument(){ System.out.println("Leyendo documento XML"); }
    public void endDocument(){ …}
    …
    }
```

Nota: ten en cuenta al utilizar SAX tienes que controlar la excepción **SAXException.**

Práctica 7

```
Comienzo del documento XML
--->GOT
------>character
--------->id
------------>148
--------->name
------------>Arya Stark
--------->gender
------------>Female
--------->culture
------------>Northmen
```

- En esta página encontraréis un ejemplo completo:
  - https://www.tutorialspoint.com/java_xml/java_sax_parse_document.htm

- Puedes utilizarlo como base para poder leer nuestro documento XML de GOT y sacar su contenido por pantalla. También puedes intentar que tenga una jerarquía similar a la que hicimos en la práctica anterior. Por último es posible independizar tu código de las etiquetas, y hacerlo independiente del archivo de entrada:

- La puntuación del ejercicio será así:
  - Utilizando un código similar al ejemplo (utilizando etiquetas) optarás a 6 puntos.
  - Si no utilizas utilizas etiquetas (lo que permitiría sacar cualquier documento por pantalla con SAX) optarás a un 8.
  - Si consigues sacar una jerarquía similar a la que hicimos en la práctica anterior, optarás al 10.

SAX - Para saber más...

En el siguiente enlace tenéis disponible un tutorial en el que se explica como parsear un fichero XML utilizando SAX:

- https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html