4. CLASES ASOCIADAS A LAS OPERACIONES DE GESTION DE FICHEROS.

4.1.File (http://docs.oracle.com/javase/7/docs/api/java/io/File.html)

En Java, para gestionar el sistema de archivos se utiliza la clase **'File'**. Es una clase que debe entenderse como una referencia a la ruta o localización de archivos/directorios del sistema.

NO representa el contenido de ningún fichero, sino la ruta del sistema donde se localizan. Como se trata de una ruta, la clase puede representar tanto archivos como carpetas o directorios.

La clase File encapsula prácticamente toda la funcionalidad necesaria para gestionar un sistema de archivos organizado en árbol de directorios. Es una gestión completa que incluye:

1. Funciones de manipulación y consulta de la propia estructura jerárquica (creación, eliminación, obtención de la ubicación, etc. de archivos o carpetas)
2. Funciones de manipulación y consulta de las características particulares de los elementos (nombres, tamaño o capacidad, etc.)
3. Funciones de manipulación y consulta de atributos específicos de cada sistema operativo y que, por tanto, sólo será funcional si el sistema operativo anfitrión soporta también la funcionalidad. Nos referimos, por ejemplo, los permisos de escritura, de ejecución, atributos de ocultación, etc.

Constructores y Descripción
File(**File** parent, **String** child) Crea una nueva instancia **File** a partir de una ruta de acceso abstracta principal y una cadena de ruta de acceso secundaria.
File(**String** pathname) Crea una nueva instancia **File** al convertir la cadena de nombre de ruta dada en un nombre de ruta abstracto.
File(**String** parent, **String** child) Crea una nueva instancia **File** a partir de una cadena de nombre de ruta principal y una cadena de nombre de ruta secundaria.
File(**URI** uri) Crea una nueva instancia **File** a partir de una URI

Métodos más utilizados	
	Método y descripción
boolean	**canExecute**() Comprueba si la aplicación puede ejecutar el archivo indicado por este nombre de ruta abstracto.
boolean	**canRead**() Comprueba si la aplicación puede leer el archivo indicado por este nombre de ruta abstracto.
boolean	**canWrite**() Comprueba si la aplicación puede modificar el archivo indicado por este nombre de ruta abstracto.
Int	**compareTo**(**File** pathname) Compara dos rutas abstractas lexicográficamente.
boolean	**createNewFile**() Crea un nuevo archivo vacío si y solo si un archivo con este nombre aún no existe.

boolean	**delete**() Elimina el archivo o directorio indicado por este nombre de ruta abstracto.
boolean	**exists**() Comprueba si el archivo o directorio indicado por este nombre de ruta abstracto existe.
String	**getAbsolutePath**() Devuelve la cadena de nombre de ruta absoluta de este nombre de ruta abstracto.
String	**getName**() Devuelve el nombre del archivo o directorio indicado por este nombre de ruta abstracto.
String	**getParent**() Devuelve la cadena de nombre de ruta del padre de este nombre de ruta abstracto, o null si este nombre de ruta no nombra un directorio padre.
File	**getParentFile**() Devuelve el nombre de ruta abstracto del padre de este nombre de ruta abstracto, o null si este nombre de ruta no nombra un directorio padre.
String	**getPath**() Convierte este nombre de ruta abstracto en una cadena de nombre de ruta.
boolean	**isAbsolute**() Comprueba si este nombre de ruta abstracto es absoluto.
boolean	**isDirectory**() Comprueba si el archivo indicado por este nombre de ruta abstracto es un directorio.
boolean	**isFile**() Comprueba si el archivo indicado por este nombre de ruta abstracto es un archivo normal.
long	**lastModified**() Devuelve la hora en que se modificó por última vez el archivo indicado por este nombre de ruta abstracto.
long	**length**() Devuelve la longitud del archivo denotado por este nombre de ruta abstracto.
String[]	**list**() Devuelve un array de Strings que nombran los archivos y directorios en el directorio indicado.
File[]	**listFiles**() Devuelve un array de Files que denotan los archivos en el directorio indicado.
boolean	**mkdir**() Crea el directorio nombrado por este nombre de ruta abstracto.
boolean	**renameTo**(**File** dest) Renombra el archivo indicado por este nombre de ruta abstracto.
URI	**toURI**() Construye un archivo URI que representa este nombre de ruta abstracto.

Ejercicios 01, 02 y 03.

5. FLUJOS O STREAMS. TIPOS.

http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html

El concepto de flujo (*stream*) no es exclusivo de los ficheros, sino que se trata de una abstracción relacionada con cualquier proceso de transmisión de información entre un contenedor de datos (en nuestro caso un fichero, aunque también puede ser una zona de memoria, otra aplicación, etc..) y una zona de la memoria controlada por la aplicación.

Llamaremos **flujo de entrada** aquellos procesos de transmisión de información que trasladen datos desde un contenedor cualquiera a la zona de memoria controlada por la aplicación. Es decir, que envíen datos a fin de ser procesadas durante la ejecución.

Llamaremos **flujo de salida** aquellos procesos de transmisión de información que trasladen datos desde la zona de memoria controlada por la aplicación hacia cualquier otro contenedor de datos.

La transmisión de datos por medio de flujos entiende siempre de manera secuencial, es decir, el orden en que salen los datos del emisor es el mismo en que llegan al receptor.

Dada una secuencia de bits, no hay manera, a priori, de saber la composición de sus datos. Sólo si conocemos el tamaño y el orden en que se compactaron, podremos recuperar los valores originales del flujo.

Afortunadamente, los tipos básicos de datos tienen un tamaño prefijado y el almacenamiento se hace considerando todos los bits. Las clases de Java orientadas a flujos transfieren los datos de forma transparente al programador. No hay que indicar la cantidad de bits que hay que transferir, sino que se deduce a partir del tipo de dato que la variable representa.

Existen dos grandes clases de flujos:

a. Flujos de bytes (8 bits): realizan operaciones I/O de bytes y su uso está orientado a la lectura/escritura de bytes. Todas las clases que tratan este flujo de datos descienden de las clases **InputStream** y **OutputStream**, cada una de las cuales tienen diferentes subclases dependiendo del medio o contenedor de la información.

b. Flujos de caracteres (16 bits): realizan operaciones I/O de caracteres. Todas las clases que tratan este flujo de datos descienden de las clases **Reader** y **Writer**, cada una de las cuales tienen diferentes subclases dependiendo del medio o contenedor de la información.

5.1. FLUJOS DE BYTES (8 bits). Orientados a la lectura/escritura de datos binarios.

5.1.1. Clase Genérica InputStream

```
public abstract class InputStream extends Object implements Closeable
```

Esta clase abstracta es la superclase de todas las clases que representan un flujo de entrada de bytes.

Subclases directas conocidas:

AudioInputStream , ByteArrayInputStream , FileInputStream , FilterInputStream , InputStream , ObjectInputStream , PipedInputStream , SequenceInputStream , StringBufferInputStream

Métodos	
	Método y descripción
int	**available**() Devuelve una estimación del número de bytes que se pueden leer (o saltar) de esta secuencia de entrada sin bloquear mediante la próxima invocación de un método para esta secuencia de entrada.
void	**close**() Cierra esta secuencia de entrada y libera todos los recursos asociados con la secuencia.
void	**mark**(int readlimit) Marca la posición actual en esta secuencia de entrada.
boolean	**markSupported**() Comprueba si esta secuencia de entrada admite los métodos mark y reset.
abstract int	**read**() Lee el siguiente byte de datos de la secuencia de entrada.
int	**read**(byte[] b) Lee cierto número de bytes de la secuencia de entrada y los almacena en el array b.
int	**read**(byte[] b, int off, int len) Lee hasta **len** bytes de datos de la secuencia de entrada en una matriz de bytes.
void	**reset**() Reposiciona este flujo a la posición en el momento en mark que se llamó por última vez al método en este flujo de entrada.
long	**skip**(long n) Salta y descarta **n** bytes de datos de esta secuencia de entrada.

5.1.2. Clase Genérica OutputStream

```
public abstract class OutputStream
extends Object
implements Closeable, Flushable
```

Esta clase abstracta es la superclase de todas las clases que representan un flujo de bytes de salida. Una secuencia de salida acepta bytes de salida y los envía a algún contenedor o almacén. Las aplicaciones que necesitan definir una subclase de OutputStream siempre deben proporcionar al menos un método que escriba un byte de salida.

Subclases directas conocidas:

ByteArrayOutputStream , FileOutputStream , FilterOutputStream , ObjectOutputStream , OutputStream , PipedOutputStream

Métodos	
	Método y descripción
void	**close**() Cierra esta secuencia de salida y libera todos los recursos asociados con esta secuencia.
void	**flush**() Vacía esta secuencia de salida y obliga a escribir los bytes de salida almacenados en búfer.
void	**write**(byte[] b) Escribe b.length bytes del array de bytes especificada en esta secuencia de salida.
void	**write**(byte[] b, int off, int len) Escribe len bytes del array de bytes especificado comenzando en el desplazamiento off en esta secuencia de salida.
abstract void	**write**(int b) Escribe el byte especificado en esta secuencia de salida.

Ejercicio 04.

5.1.3. Objetos Serializables.

La serialización de objetos de Java permite tomar cualquier objeto que implemente la interfaz **Serializable** y convertirlo en una secuencia de bits, que puede ser posteriormente restaurada para regenerar el objeto original. Esta interfaz contiene los siguientes métodos:

- `private void writeObject(ObjectOutputStream stream) throws IOException;`
- `private void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException;`

Para leer y escribir objetos serializables a un stream se utilizan las clases java **ObjectInputStream** y **ObjectOutputStream**. De tal forma que para, por ejemplo, escribir un objeto en un fichero necesitamos crear un flujo de salida a disco con FileOutputStream y un flujo de salida de objetos ObjectOutputStream que es el que procesa los datos y vinculado al flujo FileOutputStream.

```
FileOutputStream fos = new FileOutputStream("t.tmp");
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

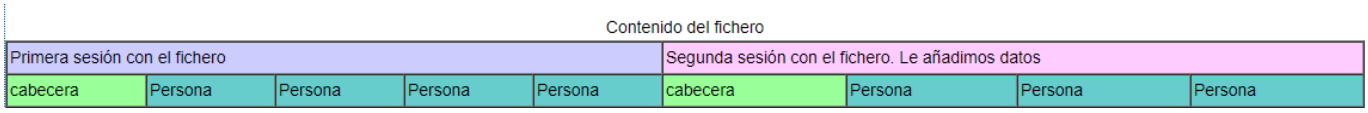
Métodos más utilizados ObjectOutputStream	
void	**close**() Cierra la corriente.
void	**flush**() Vacía la corriente.
void	**write**(int val) Escribe un byte.
void	**writeBoolean**(boolean val) Escribe un booleano.
void	**writeChar**(int val) Escribe un char de 16 bits.
void	**writeDouble**(double val) Escribe un doble de 64 bits.
void	**writeFloat**(float val) Escribe un flotante de 32 bits.
void	**writeInt**(int val) Escribe un int 32 bit.
void	**writeLong**(long val) Escribe un largo de 64 bits.
void	**writeObject**(**Object** obj) Escribe el objeto especificado en ObjectOutputStream.
void	**writeUTF**(**String** str) Escritura de datos primitivos de esta cadena en formato **UTF-8 modificado** .

Métodos más utilizados ObjectInputStream	
	Método y descripción
int	**available**() Devuelve el número de bytes que se pueden leer sin bloquear.
void	**close**() Cierra la secuencia de entrada.
int	**read**() Lee un byte de datos.
boolean	**readBoolean**() Lee en un booleano.
char	**readChar**() Lee un char de 16 bits.
double	**readDouble**() Lee un doble de 64 bits.
float	**readFloat**() Lee un flotador de 32 bits.
Object	**readObject**() Leer un objeto de ObjectInputStream.
String	**readUTF**() Lee una cadena en formato **UTF-8 modificado** .

Existe un problema con los ficheros de objetos. Al crear un fichero de objetos se crea una cabecera inicial con información, y a continuación se añaden los objetos. Si el fichero se utiliza de nuevo para añadir más registros, se crea una nueva cabecera y se añaden los objetos a partir de esta cabecera. El problema surge al leer el fichero cuando en su lectura se encuentra con la segunda cabecera y aparece una excepción.

```
/* El true del final indica que se abre el fichero para añadir datos al final del fichero.*/

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fichero,true));
```



http://www.chuidiang.org/java/ficheros/ObjetosFichero.php

Ejercicio 05.

5.2.FLUJOS DE CARACTERES (16 bits). Orientados a la lectura/escritura de caracteres.

5.2.1. Clase Genérica Reader

```
public abstract class Reader extends Object implements Readable, Closeable
```

Clase abstracta para leer secuencias de caracteres.

Subclases:

BufferedReader, InputStreamReader, FileReader, FilterReader, PipedReader, StringReader

Métodos más utilizados	
abstract void	**close**() Cierra el stream y libera todos los recursos del sistema asociados a ella.
void	**mark**(int readAheadLimit) Marca la posición actual en la secuencia.
boolean	**markSupported**() Indica si el stream admite la operación mark ().
Int	**read**() Lee un solo caracter.
Int	**read**(char[] cbuf) Lee caracteres en un array.
Boolean	**ready**() Indica si esta transmisión está lista para ser leída.
Long	**skip**(long n) Salta n caracteres.

5.2.2. Clase Genérica Writer

```
public abstract class Writer extends Object implements Appendable, Closeable, Flushable
```

Clase abstracta para escribir en secuencias de caracteres.

WriterBufferedWriter, FilterWriter, OutputStreamWriter, FileWriter, PipedWriter, PrintWriter

Métodos más utilizados	
Writer	**append**(char c) - **append**(**CharSequence** csq) Agrega el carácter / secuencia caracteres a este stream.
abstract void	**close**() Cierra la corriente y la descarga primero.
void	**write**(char[] cbuf) Escribe una cadena de caracteres.
void	**write**(int c) Escribe un solo personaje.
void	**write**(**String** str) Escribe una cadena.

Los ficheros de texto, que son los que normalmente utilizamos, almacenan caracteres alfanuméricos en un formato estándar. Para trabajar con ellos usaremos la clase **FileReader** para leer caracteres y **FileWriter** para escribir caracteres en ellos. Además de controlar las excepciones que se puedan producir

(IOException, FileNotFoundException, etc.) mediante un try – catch – finally, hay que tener especial cuidado con dos detalles:

1. Es muy importante cerrar los ficheros antes de terminar la aplicación.

```
public void copiaFicheros ( File origen, File destino ) throws IOException {
    FileReader reader = null ;
    FileWriter writer = null ;
    try {
        reader = new FileReader ( origen ) ;
        writer = new FileWriter ( destino ) ;
        copia ( reader, writer ) ;
    } catch (FileNotFoundException fnf) {
          …
    } catch (IOException io) {
          …
    } finally {
        reader.close( );
        writer.close( );
    }
}
```

2. Hay que especificar, al crear el FileWriter si vamos a sobrescribir el contenido del fichero con nueva información o a añadir esta información al final de la ya existente.

```
FileWriter fw = new FileWriter ( file );
FileWriter fw = new FileWriter ( file , true);
```

Ejercicios 06, 07, 08 y 09.

c. FICHEROS DE ACCESO ALEATORIO o DIRECTO.

```
public class RandomAccessFile extends Object implements DataOutput,DataInput,Closeable
```

Todos los flujos que acabamos de ver trabajan de forma secuencial y en una única dirección. Es decir, empezaba la lectura en el primer byte/carácter/objeto y seguíamos leyendo uno a uno hasta el final del fichero.

En Java existe la clase **RandomAccessFile** orientada a bytes que permite el acceso relativo a cualquier parte de su contenido. Permite el acceso no secuencial (llamado también relativo/aleatorio/directo) en un archivo. Es un flujo bidireccional, es decir, un flujo que permite tanto la escritura como la lectura, a pesar de que se puede configurar como flujo unidireccional. Se trata de una clase independiente de las vistas hasta ahora y no es jerarquía de las clases InputStream ni OutputStream.

RandomAccessFile implementa todos los métodos de DataInputStream y de DataOutputStream, por lo que no necesita la envoltura para trabajar con los tipos de datos primitivos.

Cuando se abre un fichero aleatorio, se crea automáticamente un puntero que señala la posición desde la cual se va a hacer una lectura o escritura y que inicialmente tiene el valor 0. Las sucesivas llamadas a los métodos read() o write() ajustan el puntero según la cantidad de bytes leídos o escritos.

Las operaciones de salida que escriben más allá del final actual del fichero hacen que dicho fichero se extienda automáticamente. El puntero del archivo puede leerse por el método getFilePointer y establecerse por el método seek.

En general, para todas las rutinas de lectura en esta clase, si se alcanza el final del archivo antes de que se haya leído el número deseado de bytes, se arroja un EOFException (que es una especie de IOException).

Constructor y Descripción	
RandomAccessFile(**File** file, **String** mode)	
RandomAccessFile(**String** name, **String** mode)	
mode	
r	Abierto solo para lectura. Invocar cualquiera de los métodos de escritura del objeto resultante hará que se arroje un IOException
rw	Abierto para leer y escribir. Si el archivo aún no existe, se intentará crearlo.

Métodos más utilizados	
void	**close**() Cierra este archivo de acceso aleatorio y libera todos los recursos asociados.
long	**getFilePointer**() Devuelve el la posición actual del puntero en este archivo.
long	**length**() Devuelve la longitud de este archivo.
int	**read**() Lee un byte de datos de este archivo.
int	**read**(byte[] b) Lee hasta b.length bytes de datos de este archivo en una array de bytes.
boolean	**readBoolean**() Lee un booleande este archivo.

char	**readChar**() Lee un carácter de este archivo.
double	**readDouble**() Lee un double de este archivo.
float	**readFloat**() Lee un float de este archivo.
int	**readInt**() Lee un entero de 32 bits con signo de este archivo.
long	**readLong**() Lee un entero de 64 bits con signo de este archivo.
String	**readUTF**() Lee en una cadena de este archivo.
void	**seek**(long pos) Establece el desplazamiento del puntero del archivo, medido desde el comienzo del fichero.
void	**write**(byte[] b) Escribe un array de bytes en este archivo, comenzando en el puntero del archivo actual.
void	**writeBoolean**(boolean v)
void	**writeChar**(int v)
void	**writeDouble**(double v)
void	**writeFloat**(float v)
void	**writeInt**(int v)
void	**writeLong**(long v)
void	**writeShort**(int v)
void	**writeUTF**(**String** str)

5.2.3. Notas

1. Cada registro que se almacena en un fichero aleatorio está formado por, al menos, dos campos:

   1.1. Un número entero que actúa como identificador del registro y que será utilizado, entre otras cosas, para acceder a este registro.

   ```
   raf.seek((nMatricula - 1) * tamanio_registro);
   ```

   1.2. Información que se guarda en el registro.

2. Un carácter ocupa 2 bytes, un int ocupa 4 bytes, un double ocupa 8 bytes, .. (mirar tabla de tipos datos Java).
3. Se puede almacenar un registro en cualquier posición (identificador de registro) del fichero sin tener que tener almacenado (obligatoriamente) un registro en una posición anterior. En este caso, todos los registros que no han sido generados por el usuario, son generados de forma automática con el valor 0 para el identificador.
4. Cuando se crea el fichero, el puntero asociado se coloca en la posición (long) 0
5. Para saber si hemos llegado al final del fichero se compara length() y getFilePointer().
6. El borrado de registros en los ficheros aleatorios es lógica y no física.

Ejercicios 10 y 11.

3. ACCESO A FICHEROS XML.

XML (Lenguaje de Marcado eXtensible) es un metalenguaje stándar para el intercambio de datos (bases de datos, datos configuración, etc...). Los documentos XML consiguen estructurar la información intercalando una serie de marcas denominadas etiquetas. En XML, las marcas o etiquetas tienen cierta similitud con un contenedor de información. Así, una etiqueta puede contener otras etiquetas o información textual. De este modo, conseguiremos subdividir la información estructurando de forma que pueda ser fácilmente interpretada.

Como toda la información es textual, no existe el problema de representar los datos de diferente manera. Cualquier dato, ya sea numérico, booleano o de tipo fecha, será convertido a modo texto, de modo que cualquiera que sea el sistema de representación de datos, será posible leer e interpretar correctamente la información contenida en un archivo XML.

Para poder leer (o posteriormente modificar) el contenido y estructura de un fichero XML es necesario el uso de un procesador XML o *parser*. Estos son independientes del lenguaje de programación que se use para acceder al fichero XML. Los dos más utilizados son:

- **SAX** procesa el documento XML de forma **secuencial** y conforme se va encontrado partes del XML (comienzo/fin del documento, comienzo/fin de una etiqueta, etc.) produce una serie de **eventos** en función de lo que ha leído.

- **DOM, JDOM** y **XOM** procesan de forma **jerárquica** el documento XML al completo y generan un **árbol de nodos (nodos padre, nodos hijos, nodos finales)** en memoria que representa el documento XML. Una vez generado el árbol, se van recorriendo los nodos y se analiza a qué tipo pertenecen. Este árbol puede ser leído, modificado y de nuevo serializado a cadena o fichero XML. Incluso es posible crear un árbol desde cero, es decir, no hay que partir de un fichero o documento XML ya creado.

  DOM, **JDOM** y **XOM** es más potente que **SAX,** pero el **consumo de memoria** de DOM, JDOM y XOM es mucho mayor que el de SAX, ya que mantienen todo el árbol de nodos en memoria.

3.1. Acceso a ficheros XML mediante DOM.

DOM es un procesador XML o parser que además permite modificar los documentos parseados, así como crear documentos XML.

DOM genera un árbol de **árbol de nodos** a partir de un documento XML.

- Lee el documento XML, lo procesa y genera una serie de objetos que representan cada una de los elementos del documento XML.
- Todos estos elementos están relacionados entre sí.
- Cada uno de estos objetos (nodos o no del documento XML) se tratan mediante interfaces de Java que se encuentran en el paquete org.w3c.dom
  - **Document**: representa al documento XML en sí. Es el objeto principal del árbol y sólo puede haber uno por documento XML. El resto de objetos del árbol deben pertenecer al *ámbito* de este objeto para poder estar en el árbol.
  - **Element:** representa una etiqueta XML. Almacena entre otras la lista de atributos que posee *(getAttributes)*, la lista de nodos hijos *(getChildNodes)*, contenido de tipo texto *(getTextContent).* Tiene métodos de acceso directo a su primer hijo *(getFirstChild)* y acceso directo a su próximo nodo hermano *(getNextSibling).*
  - **Node :** cualquier nodo del documento

- **NodeList.** Contiene una lista con los nodos hijo de un nodo.
- **Attr:** atributo dentro de un Element.
- **CharacterData**: texto dentro del XML. Puede ser del tipo *CDATASection*, *Text* o *Comment.* Estas 3 interfaces heredan de *CharacterData.*
- **DocumentType.** Proporciona información contenida en el etiqueta <!DOCTYPE>

Para poder trabajar con DOM en java necesitamos las clases e interfaces que componen el paquete **org.w3c.com** (contenido en el JSDK) y el paquete **javax.xml.parsers**.

El estándar W3C define la especificación de la clase **DocumentBuilder** con el propósito de poder instanciar estructuras DOM a partir de un XML. La clase DocumentBuilder es una clase abstracta, y para que se pueda adaptar a las diferentes plataformas, puede necesitar fuentes de datos o requerimientos diversos. Recuerde que las clases abstractas no se pueden instanciar de forma directa. Por este motivo, el consorcio W3 especifica también la clase **DocumentBuilderFactory**.

Las instrucciones necesarias para leer un archivo XML y crear un objeto ***Document*** (que representa todo un documento XML) serían las siguientes:

```
import java.io.File;
import javax.xml.parsers.*;
import org.w3c.dom.*;
…
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
try {
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document document = builder.parse(new File("fichero.xml"));
       …
```

DocumentBuilder también instancia objetos Document vacíos que podrán ser manipulados a posteriori.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
try {
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document document = builder.newDocument();
       …
```

La escritura de la información contenida en el DOM se puede secuenciar en forma de texto utilizando otra utilidad de Java llamada ***Transformer*** . Se trata de una utilidad que permite realizar fácilmente conversiones entre diferentes representaciones de información jerárquica. Es capaz, por ejemplo, de pasar la información contenida en un objeto Document a un archivo de texto en formato XML.

Transformer es también una clase abstracta y requiere de un builder para poder ser instanciada. La clase Transformer puede trabajar con multitud de contenedores de información porque en realidad trabaja con un par de tipos adaptadores (clases que hacen compatibles jerarquías diferentes) que se llaman **Source y Result**. Las clases que implementen estas interfaces se encargarán de hacer compatible un tipo de contenedor específico al requerimiento de la clase Transformer. Así, disponemos de las clases DOMSource, SAXSource o StreamSorce como adaptadores del contenedor de la fuente de información (DOM, SAX o Stream respectivamente). DOMResult, SAXResult o StreamResult son los adaptadores equivalentes del contenedor destino.

El código básico para realizar una transformación de un árbol DOM en memoria a un archivo de texto XML sería el siguiente:

```
private static Document doc;

private static Transformer xformer;
private static Source source;
private static Result result;
```

```
…

try {
      xformer = TransformerFactory.newInstance().newTransformer();
} catch (TransformerConfigurationException | TransformerFactoryConfigurationError e) {
      e.printStackTrace();
}

/** Propiedades del fichero XML de salida */

xformer.setOutputProperty(OutputKeys.METHOD, "xml");
xformer.setOutputProperty(OutputKeys.INDENT, "yes");

/** Definimos la Entrada y la Salida de la Transformacion */

source = new DOMSource(doc);
result = new StreamResult(new File("alumnos.xml"));

/** Realizamos la Transformación mediante el metodo transform() */

try {
   xformer.transform(source, result);
} catch (TransformerException e) {
   e.printStackTrace();
}
```

Interface Document

```
public interface Document extends Node
```

La interfaz *Document* representa todo el documento HTML o XML. Conceptualmente, es la raíz del árbol de documentos y proporciona el acceso principal a los datos del documento.

Dado que los elementos, nodos de texto, comentarios, instrucciones de procesamiento, etc. no pueden existir fuera del contexto de *Document*, la interfaz *Document* también contiene los métodos necesarios para crear estos objetos. Los objetos *Node* creados tienen un atributo *ownerDocument* que los asocia con el *Document* en cuyo contexto fueron creados.

Métodos más utilizados	
Attr	**createAttribute**(**String** name)
CDATASection	**createCDATASection**(**String** data)
Comment	**createComment**(**String** data)
Element	**createElement**(**String** tagName)
Text	**createTextNode**(**String** data)
DocumentType	**getDoctype**()
Element	**getDocumentElement**() Nos devuelve el nodo raíz del documento.
Element	**getElementById**(**String** elementId) Devuelve el Element que tiene un atributo ID con el valor dado.
NodeList	**getElementsByTagName**(**String** tagname) Devuelve un orden NodeList de todos los Elements con un nombre de etiqueta dado.
NodeList	**getElementsByTagNameNS**(**String** namespaceURI, **String** localName) Devuelve una NodeListde todas las Elementscon un nombre local y un URI de espacio de nombres dados en orden de documento.
String	**getXmlVersion**() Un atributo que especifica, como parte de la **declaración XML** , el número de versión de este documento.
Void	**setXmlVersion**(**String** xmlVersion) Un atributo que especifica, como parte de la **declaración XML** , el número de versión de este documento.

```
// Obtenemos el nodo raiz del documento
Element nodoRaiz = doc.getDocumentElement();

// Obtenemos la lista de nodos que cuelgan del nodo raiz
NodeList hijosN1 = nodoRaiz.getElementsByTagName("CD");
```

Interface NodeList

```
public interface NodeList
```

La interfaz **NodeList** proporciona la abstracción de una colección ordenada de nodos, sin definir o restringir cómo se implementa esta colección. Se puede acceder a los elementos de un NodeList a través de un índice, comenzando desde 0.

Métodos	
Int	**getLength**() Devuelve el número de nodos en la lista.
Node	**item**(int index) Devuelve el elemento de la lista indicado por el índice. Si **index** es mayor o igual que el número de nodos en la lista, devuelve null.

Interface Node

```
public interface Node
```

La interfaz **Node** es el tipo de datos principal para todo el **DOM**. Representa un solo nodo en el árbol de documentos. Si bien todos los objetos que implementan la interfaz Node exponen métodos para tratar con nodos hijos, no todos los objetos que implementan la interfaz **Node** pueden tener hijos. Por ejemplo, los nodos **Text** pueden no tener hijos, y al agregar hijos a dichos nodos se genera un **DOMException**.

Los atributos **nodeName**, **nodeValue** y **attributes** se incluyen como un mecanismo para acceder a la información del nodo. En los casos en que no exista un mapeo obvio de estos atributos para un específico **nodeType** (por ejemplo, **nodeValue** para un **Element** or **attributes** para un **Comment**), se devuelve null.

<table>
<tr><th></th><th>Métodos más utilizados y descripción</th></tr>
<tr><td>Node</td><td>appendChild(Node newChild)
Agrega el nodo newChild al final de la lista de hijos de este nodo.</td></tr>
<tr><td>NamedNodeMap</td><td>getAttributes()
Devuelve un NamedNodeMap con los atributos de este nodo (si es un Element) o null en caso contrario.</td></tr>
<tr><td>NodeList</td><td>getChildNodes()
A NodeList que contiene todos los hijos de este nodo.</td></tr>
<tr><td>Node</td><td>getFirstChild()
El primer hijo de este nodo.</td></tr>
<tr><td>Node</td><td>getLastChild()
El último hijo de este nodo.</td></tr>
<tr><td>Node</td><td>getNextSibling()
El nodo que sigue inmediatamente a este nodo.</td></tr>
<tr><td>String</td><td>getNodeName()
El nombre de este nodo.</td></tr>
<tr><td>short</td><td>getNodeType()
Un código que representa el tipo del objeto.
<table>
<tr><td>ATTRIBUTE_NODE</td><td>The node is an Attr.</td></tr>
<tr><td>CDATA_SECTION_NODE</td><td>The node is a CDATASection.</td></tr>
<tr><td>COMMENT_NODE</td><td>The node is a Comment.</td></tr>
<tr><td>DOCUMENT_NODE</td><td>The node is a Document.</td></tr>
<tr><td>ELEMENT_NODE</td><td>The node is an Element.</td></tr>
<tr><td>ENTITY_NODE</td><td>The node is an Entity.</td></tr>
<tr><td>PROCESSING_INSTRUCTION_NODE</td><td>The node is a Processing Instruction.</td></tr>
<tr><td>TEXT_NODE</td><td>The node is a Text node.</td></tr>
</table></td></tr>
<tr><td>String</td><td>getNodeValue()
El valor de este nodo, dependiendo de su tipo.</td></tr>
<tr><td>Document</td><td>getOwnerDocument()
El Documentobjeto asociado a este nodo.</td></tr>
<tr><td>Node</td><td>getParentNode()
El padre de este nodo.</td></tr>
<tr><td>Node</td><td>getPreviousSibling()
El nodo que precede inmediatamente a este nodo.</td></tr>
<tr><td>String</td><td>getTextContent()
Este atributo devuelve el contenido de texto de este nodo y sus descendientes.</td></tr>
</table>

boolean	**hasAttributes**() Devuelve si este nodo (si es un elemento) tiene algún atributo.
boolean	**hasChildNodes**() Devuelve si este nodo tiene hijos.
boolean	**isEqualNode**(**Node** arg) Comprueba si dos nodos son iguales.
Node	**removeChild**(**Node** oldChild) Elimina el nodo hijo indicado por oldChild de la lista de hijos y lo devuelve.
void	**setNodeValue**(**String** nodeValue) Establece el valor de este nodo, dependiendo de su tipo; ver la tabla de arriba.
Object	**setUserData**(**String** key, **Object** data, **UserDataHandler** handler) Asociar un objeto a una clave en este nodo.

Interface Element

```
public interface Element extends Node
```

La interfaz **Node** es el tipo de datos principal para todo el **DOM**. Representa un solo nodo en el árbol de documentos. Si bien todos los objetos que implementan la interfaz Node exponen métodos para tratar con nodos hijos, no todos los objetos que

La interfaz **Element** representa un elemento en un documento HTML o XML. Los elementos pueden tener atributos asociados con ellos; dado que la interfaz **Element** hereda de Node, el atributo `attributes` puede usarse para recuperar el conjunto de todos los atributos para un Element. Existen métodos en la interfaz para recuperar cada uno de los objetos `Attr` por nombre o el valor de atributo por su nombre.

	Métodos más utilizados y descripción
String	**getAttribute**(**String** name) Recupera un valor de atributo por nombre.
Attr	**getAttributeNode**(**String** name) Recupera un nodo de atributo por nombre.
NodeList	**getElementsByTagName**(**String** name) Devuelve uno NodeList de todos los descendientes.
String	**getTagName**() El nombre del elemento.
boolean	**hasAttribute**(**String** name) Devuelve true cuando un atributo con un nombre dado se encuentra en este Element.
void	**removeAttribute**(**String** name) Elimina un atributo por nombre.
void	**setAttribute**(**String** name, **String** value) Agrega un nuevo atributo.

RECORRER NODOS DEL FICHERO XML MEDIANTE DOM.

```
import javax.xml.parsers.*;
import org.w3c.dom.*;

//Construir instancia del DocumentBuilder para construir el parser
DocumentBuilder documentBuilder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
// Procesamos el fichero XML y obtenemos nuestro objeto Document
Document doc = documentBuilder.parse(new File("/ruta_a_fichero/fichero.xml"));
// Obtenemos Nodo raiz
Element elementRaiz = doc.getDocumentElement();

// y ahora, o bien :  Obtenemos todos los nodos hijos del Nodo raiz
NodeList hijos = elementRaiz.getChildNodes();
// y recorremos sus hijos
for(int i=0; i<hijos.getLength(); i++){
   Node nodo = hijos.item(i);
   if (nodo instanceof Element){ System.out.println(nodo.getNodeName());  }
}

// o bien : Buscamos todas las etiquetas de un mismo nombre dentro del XML
       NodeList listaNodos = doc.getElementsByTagName("etiquetaHija");
// y recorremos sus elementos
```

RECORRER ATRIBUTOS DE UN NODO MEDIANTE DOM.

```
// Variable Node y NamedNodeMap donde recogemos los atributos ( similar a NodeList)
Node e;
NamedNodeMap attributes;

// Comprobamos que el Nodo tenga atributos y en caso afirmativo los obtenemos mediante el
// método getAttributes. A continuación recorremos el NamedNodeMap como un NodeList.

if (e.hasAttributes()) {
   attributes = e.getAttributes();
   System.out.println(nivel+"\tAtributos:");

   for (int i = 0; i < attributes.getLength(); i++) {
          System.out.println(nivel+"\t\t" +
          attributes.item(i).getNodeName() + ":" +
          attributes.item(i).getNodeValue() + "\t");
   }
}
```

MODIFICAR FICHERO XML MEDIANTE DOM.

Mediante DOM también se puede modificar el XML. Esto se hace modificando el árbol de nodos. Para añadir una etiqueta nueva hay que partir del *Document* donde la vamos a insertar y después añadirla como hija de otra etiqueta.

Esto es especialmente útil cuando a partir de unos datos de origen (fichero binario, fichero de caracteres, base de datos) se quiere generar un fichero XML.

```
// Añadimos una nueva etiqueta al documento
// Primero creamos la etiqueta (element)
Element nuevaEtiqueta = doc.createElement("nuevaEtiqueta");
// Añadimos atributos
nuevaEtiqueta.setAttribute("atributoNuevo", "Es un nuevo atributo");
// Añadimos contenido
nuevaEtiqueta.setTextContent("Contenido dentro de la nueva etiqueta");
// después se la añadimos como hija a una etiqueta ya existente
etiquetaHija.appendChild(nuevaEtiqueta);
```

Ejercicios 12 y 13.

Ejercicio 01.

Mostrar la lista de ficheros del directorio actual.

Ejercicio 02.

Mostrar la lista de ficheros del directorio que se pasa como parámetro desde la línea de comandos.

Ejercicio 03.

Mostrar la lista de ficheros del directorio que se pasa como parámetro desde la línea de comandos. Para cada fichero o directorio indicar al menos: su nombre, tamaño, ruta absoluta, si es un fichero o un directorio, …

Trabajo de Investigación.

En el **Ejercicio 03** teníamos que mostrar la lista de ficheros del directorio que se pasa como parámetro desde la línea de comandos. Para cada fichero o directorio había que indicar al menos: su nombre, tamaño, ruta absoluta, si es un fichero o un directorio.

Ahora haciendo uso de la interfaz FilenameFilter vamos a mostrar solo aquellos ficheros que cumplan un filtro.

Para esto hemos de crearnos una clase **Filtro** que implemente la interfaz **FilenameFilter** y modificaremos su método **accept** para indicar el filtro que deben cumplir el nombre de los ficheros.

Posteriormente en nuestra aplicación principal le pasaremos un objeto de la clase **Filtro** al método **list**() de nuestro objeto **File** para que solo muestre los ficheros que cumplan el filtro indicado.

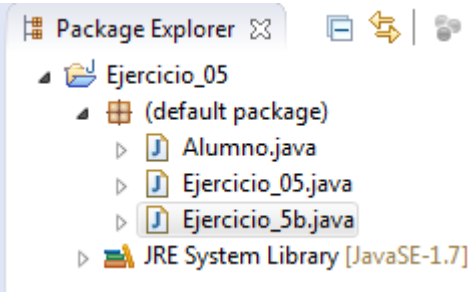
Ejercicio 04.

a. Crear un fichero binario llamado Datos.Dat en la carpeta C:\TEMP y escribe en él los números del 1 al 100.
b. Abre el fichero binario anterior y muestra su contenido.
c. Investigar como hacer para que cada vez que se ejecute el programa la información que contenía el fichero no se borre.

Ejercicio 05.

a. Crear un programa que cree un fichero binario llamado Alumnos.DAT para guardar los datos de los alumnos. Los datos de cada alumno son:

   - Nombre : String
   - Edad : Entero
   - Nota : Double.



NOTA: Una vez que el fichero está abierto para escritura, se han de guardar todos los objetos antes de cerrarlo. Una vez cerrado, si se vuelve a abrir para guardar **más** datos, la lectura de sus datos puede dar problemas ya que cada vez que se abre para añadir información se guardan unos datos de cabecera antes de los datos que se van a escribir.

b. Crear un programa que lea el fichero anterior y muestre los datos por pantalla.

Ejercicio 06.

Crear un programa que visualice el contenido de un fichero de texto cuyo nombre se pasa como argumento en la línea de comandos. Hacerlo carácter a carácter.

Ejercicio 07.

Crear un programa que visualice el contenido de un fichero de texto cuyo nombre se pasa como argumento en la línea de comandos. Hacerlo línea a línea utilizando la clase BufferedReader y su método readLine() .

Ejercicio 08.

Crear un programa que guarde en un fichero de texto una serie de líneas con salto de línea. Hacerlo utilizando la clase BufferedWriter y su método newLine() .

Ejercicio 09.

Crear un programa que lea el contenido de un fichero de texto cuyo nombre se pasa como argumento en la línea de comandos. A partir de este debe crear otro con el mismo nombre pero con extensión COD con el mismo contenido que el fichero original pero cambiando las “a” por “e”, las “e” por “i”, las “i” por “o”, las “o” por “u” y las “u” por “a”.

Ejercicio 10.

Crear un programa que realice el mantenimiento de los registros de alumnos almacenados en un fichero llamado Alumnos.DAT. Los datos de cada alumno son:

- Nombre : String
- Edad : Entero
- Nota : Flotante

El acceso a dicho fichero es SECUENCIAL.

Ejercicio 11.

Crear un programa que realice el mantenimiento de los registros de alumnos almacenados en un fichero llamado Alumnos.DAT. Los datos de cada alumno son los mismos del ejercicio anterior.

El acceso a dicho fichero es ALEATORIO.

Ejercicio 12.

Crear un programa que muestre por pantalla el nombre y el contenido de los nodos finales del fichero XML “catalogo1.xml”

```
<CATALOG>
        <CD>Empire Burlesque</CD>
        <CD>Hide your heart</CD>
        <CD>Still got the blues</CD>
        <CD>One night only</CD>
        <CD>Stop</CD>
</CATALOG>
```

Ejercicio 13.

Crear un programa que muestre por pantalla el nombre y el contenido de los nodos finales del fichero XML “catalogo2.xml”

```
<CATALOG>
	<CD>
		<TITLE>Empire Burlesque</TITLE>
		<ARTIST>Bob Dylan</ARTIST>
		<COUNTRY>USA</COUNTRY>
		<COMPANY>Columbia</COMPANY>
		<PRICE>10.90</PRICE>
		<YEAR>1985</YEAR>
	</CD>
	<CD>
		<TITLE>Hide your heart</TITLE>
		<ARTIST>Bonnie Tyler</ARTIST>
		<COUNTRY>UK</COUNTRY>
		<COMPANY>CBS Records</COMPANY>
		<PRICE>9.90</PRICE>
		<YEAR>1988</YEAR>
	</CD>
	<CD>
		<TITLE>Still got the blues</TITLE>
		<ARTIST>Gary Moore</ARTIST>
		<COUNTRY>UK</COUNTRY>
		<COMPANY>Virgin records</COMPANY>
		<PRICE>10.20</PRICE>
		<YEAR>1990</YEAR>
	</CD>
	<CD>
		<TITLE>One night only</TITLE>
		<ARTIST>Bee Gees</ARTIST>
		<COUNTRY>UK</COUNTRY>
		<COMPANY>Polydor</COMPANY>
		<PRICE>10.90</PRICE>
		<YEAR>1998</YEAR>
	</CD>
	<CD>
		<TITLE>Stop</TITLE>
		<ARTIST>Sam Brown</ARTIST>
		<COUNTRY>UK</COUNTRY>
		<COMPANY>A and M</COMPANY>
		<PRICE>8.90</PRICE>
		<YEAR>1988</YEAR>
	</CD>
</CATALOG>
```

Trabajo de Investigación 1.

Pasar la información del fichero Alumnos.DAT del Ejercicio 11 mediante DOM a un fichero XML llamado Alumnos.XML.

Trabajo de Investigación 2.

Mediante JDOM leer el fichero Alumnos.XML y mostrarlo por pantalla

Trabajo de Investigación 3.

a. Serializar una lista de objetos en un fichero XML
b. Leer el fichero XML anterior mediante XStream y mostrar su contenido por pantalla.
   Consejo : Utilizar la clase Iterator para recorrer la lista de objetos.

Trabajo de Investigación 4.

Crear una función Java que pasándole dos parámetros de tipo File nos devuelva un booleano que sea true en caso de que los dos ficheros tienen el mismo tamaño y el mismo contenido y false en caso contrario.

REFERENCIAS.

Acceso a Datos. Ed. Garceta. Alicia/María Jesús Ramos Martin.

https://ioc.xtec.cat/educacio/

https://docs.oracle.com/javase