

## MANEJO DE FICHEROS

### OBJETIVOS

Utilizar clases para la gestión de ficheros y directorios.

Valorar las ventajas y los inconvenientes de las distintas formas de acceso.

Utilizar las operaciones básicas para acceder a ficheros de acceso secuencial y aleatorio.

Utilizar clases para almacenar y recuperar información almacenada en un fichero XML.

Utilizar clases para convertir a otro formato información contenida en un fichero XML.

Gestionar excepciones.

Serializar objetos Java a representaciones XML.

### CONTENIDOS

Clases asociadas a las operaciones de gestión de ficheros. Flujos o stream. Tipos.

Formas de acceso a un fichero.

Clases para gestión de flujos de datos desde/hacia ficheros.

Operaciones básicas sobre ficheros de acceso secuencial.

Operaciones básicas sobre ficheros de acceso aleatorio.

Ficheros XML. Librerías para conversión de documentos XML a otros formatos.

Excepciones: detección y tratamiento.

Introducción a JAXB.

### RESUMEN

*En este capítulo aprenderemos a leer y escribir datos en ficheros secuenciales y directos en Java. Utilizaremos diferentes clases Java para el acceso a ficheros. Utilizaremos distintas librerías para conversión de ficheros XML a otros formatos. Aprenderemos a utilizar y gestionar excepciones. Utilizaremos la tecnología JAXB para serializar objetos Java a representaciones XML.*

## ✓ 1.1. INTRODUCCIÓN

Un **fichero** o **archivo** es un conjunto de bits almacenados en un dispositivo, como por ejemplo, un disco duro. La ventaja de utilizar ficheros es que los datos que guardamos permanecen en el dispositivo aun cuando apaguemos el ordenador, es decir, no son volátiles. Los ficheros tienen un nombre y se ubican en directorios o carpetas, el nombre debe ser único en ese directorio; es decir, no puede haber dos ficheros con el mismo nombre en el mismo directorio. Por convención cuentan con diferentes extensiones que por lo general suelen ser de 3 letras (PDF, DOC, GIF, ...) y nos permiten saber el tipo de fichero.

Un fichero está formado por un conjunto de registros o líneas y cada registro por un conjunto de campos relacionados, por ejemplo, un fichero de empleados puede contener datos de los empleados de una empresa, un fichero de texto puede contener líneas de texto correspondientes a líneas impresas en una hoja de papel. La manera en que se agrupan los datos en el fichero depende completamente de la persona que lo diseña.

En este tema aprenderemos a utilizar los ficheros con el lenguaje Java.

## ✓ 1.2. CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS

El paquete **java.io** contiene las clases para manejar la entrada/salida en Java, por tanto, necesitaremos importar dicho paquete cuando trabajemos con ficheros. Antes de ver las clases que leen y escriben datos en ficheros vamos a manejar la clase **File**. Esta clase proporciona un conjunto de utilidades relacionadas con los ficheros que nos van a proporcionar información acerca de los mismos, su nombre, sus atributos, los directorios, etc. Puede representar el nombre de un fichero particular o los nombres de un conjunto de ficheros de un directorio, también se puede usar 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(String directorioyfichero):** en Linux: `new File("/directorio/fichero.txt")`; en plataformas Microsoft Windows: `new File("C:\\directorio\\fichero.txt")`.
- **File(String directorio, String nombrefichero):** `new File("directorio", "fichero.txt")`.
- **File(File directorio, String fichero):** `new File(new File("directorio"), "fichero.txt")`.

En Linux se utiliza como prefijo de una ruta absoluta "/". En Microsoft Windows, el prefijo de un nombre de ruta consiste en la letra de la unidad seguida de ":" y, posiblemente, seguida por "\\\" si la ruta es absoluta.

Ejemplos de uso de la clase **File** donde se muestran diversas formas para declarar un fichero:

```
//Windows
File fichero1 = new File( "C:\\EJERCICIOS\\UNI1\\ejemplo1.txt");
//Linux
File fichero1 = new File( "/home/ejercicios/uni1/ejemplo1.txt");

String directorio= "C:/EJERCICIOS/UNI1";
File fichero2 = new File(directorio, "ejemplo2.txt");

File direc = new File(directorio);
```

```
File fichero3 = new File(direc, "ejemplo3.txt");
```

Algunos de los métodos más importantes de la clase **File** son los siguientes:

Método	Función
<code>String[] list()</code>	Devuelve un array de String con los nombres de ficheros y directorios asociados al objeto <b>File</b>
<code>File[] listFiles()</code>	Devuelve un array de objetos <b>File</b> contenido los ficheros que estén dentro del directorio representado por el objeto <b>File</b>
<code>String getName()</code>	Devuelve el nombre del fichero o directorio
<code>String getPath()</code>	Devuelve el camino relativo
<code>String getAbsolutePath()</code>	Devuelve el camino absoluto del fichero/directorio
<code>boolean exists()</code>	Devuelve <i>true</i> si el fichero/directorio existe
<code>boolean canWrite()</code>	Devuelve <i>true</i> si el fichero se puede escribir
<code>boolean canRead()</code>	Devuelve <i>true</i> si el fichero se puede leer
<code>boolean isFile()</code>	Devuelve <i>true</i> si el objeto <b>File</b> corresponde a un fichero normal
<code>boolean isDirectory()</code>	Devuelve <i>true</i> si el objeto <b>File</b> corresponde a un directorio
<code>long length()</code>	Devuelve el tamaño del fichero en bytes
<code>boolean mkdir()</code>	Crea un directorio con el nombre indicado en la creación del objeto <b>File</b> . Solo se creará si no existe
<code>boolean renameTo(File nuevonombre);</code>	Renombra el fichero representado por el objeto <b>File</b> asignándole <i>nuevonombre</i>
<code>boolean delete()</code>	Borra el fichero o directorio asociado al objeto <b>File</b>
<code>boolean createNewFile()</code>	Crea un nuevo fichero, vacío, asociado a <b>File</b> si y solo si no existe un fichero con dicho nombre
<code>String getParent()</code>	Devuelve el nombre del directorio padre, o <i>null</i> si no existe

El siguiente ejemplo muestra la lista de ficheros en el directorio actual. Se utiliza el método *list()* que devuelve un array de String con los nombres de los ficheros y directorios contenidos en el directorio asociado al objeto **File**. Para indicar que estamos en el directorio actual creamos un objeto **File** y le pasamos la variable *dir* con el valor “.”. Se define un segundo objeto **File** utilizando el tercer constructor para saber si el fichero obtenido es un fichero o un directorio:

```
import java.io.*;
public class VerDir {
    public static void main(String[] args) {
        String dir = ".."; //directorío actual
        File f = new File(dir);
        String[] archivos = f.list();
        System.out.printf("Ficheros en el directorio actual: %d %n",
                          archivos.length);
        for (int i = 0; i < archivos.length; i++) {
            File f2 = new File(f, archivos[i]);
            System.out.printf("Nombre: %s, es fichero?: %b, es directorio?: %b %n",
                              archivos[i], f2.isFile(), f2.isDirectory());
        }
    }
}
```

Un ejemplo de ejecución de este programa mostraría la siguiente salida:

```
Ficheros en el directorio actual: 3
Nombre: VerDir.class, es fichero?: true, es directorio?: false
Nombre: VerDir.java, es fichero?: true, es directorio?: false
Nombre: VerInf.java, es fichero?: true, es directorio?: false
```

La siguiente declaración aplicada al ejemplo anterior mostraría la lista de ficheros del directorio *d:\db*:

```
File f = new File("d:\\db");
```

Con la siguiente declaración se mostraría la lista de ficheros del directorio introducido desde la línea de comandos al ejecutar el programa:

```
String dir=args[0];
System.out.println("Archivos en el directorio " +dir);
File f = new File(dir);
```

---

### ACTIVIDAD 1.1

Realiza un programa Java que utilice el método **listFiles()** para mostrar la lista de ficheros en un directorio cualquiera, o en el directorio actual.

Realiza un programa Java que muestre los ficheros de un directorio. El nombre del directorio se pasará al programa desde los argumentos de *main()*. Si el directorio no existe se debe mostrar un mensaje indicándolo.

---

El siguiente ejemplo muestra información del fichero *VerInf.java*:

```
import java.io.*;
public class VerInf {
    public static void main(String[] args) {
        System.out.println("INFORMACIÓN SOBRE EL FICHERO:");
        File f = new File("D:\\ADAT\\UNI1\\VerInf.java");
        if(f.exists()){
            System.out.println("Nombre del fichero : "+f.getName());
            System.out.println("Ruta : "+f.getPath());
            System.out.println("Ruta absoluta : "+f.getAbsoluteFilePath());
            System.out.println("Se puede leer : "+f.canRead());
            System.out.println("Se puede escribir : "+f.canWrite());
            System.out.println("Tamaño : "+f.length());
            System.out.println("Es un directorio : "+f.isDirectory());
            System.out.println("Es un fichero : "+f.isFile());
            System.out.println("Nombre del directorio padre: "+f.getParent());
        }
    }
}
```

Visualiza la siguiente información del fichero:

```
INFORMACIÓN SOBRE EL FICHERO:
Nombre del fichero : VerInf.java
Ruta : D:\\ADAT\\UNI1\\VerInf.java
Ruta absoluta : D:\\ADAT\\UNI1\\VerInf.java
Se puede leer : true
Se puede escribir : true
```

```
Tamaño : 824
Es un directorio : false
Es un fichero : true
Nombre del directorio padre: D:\ADAT\UNI1
```

El siguiente ejemplo crea un directorio (de nombre *NUEVODIR*) en el directorio actual, a continuación crea dos ficheros vacíos en dicho directorio y uno de ellos lo renombra. En este caso para crear los ficheros se definen 2 parámetros en el objeto **File**: *File(File directorio, String nombrefich)*, en el primero indicamos el directorio donde se creará el fichero y en el segundo indicamos el nombre del fichero:

```
import java.io.*;
public class CrearDir {
    public static void main(String[] args) {
        File d = new File("NUEVODIR"); //directorío que creo
        File f1 = new File(d, "FICHERO1.TXT");
        File f2 = new File(d, "FICHERO2.TXT");

        d.mkdir(); //CREAR DIRECTORIO

        try {
            if (f1.createNewFile())
                System.out.println("FICHERO1 creado correctamente...");
            else
                System.out.println("No se ha podido crear FICHERO1...");

            if (f2.createNewFile())
                System.out.println("FICHERO2 creado correctamente...");
            else
                System.out.println("No se ha podido crear FICHERO2...");
        } catch (IOException ioe) {ioe.printStackTrace();}

        f1.renameTo(new File(d, "FICHERO1NUEVO")); //renombro FICHERO1

        try {
            File f3 = new File("NUEVODIR/FICHERO3.TXT");
            f3.createNewFile(); //crea FICHERO3 en NUEVODIR
        } catch (IOException ioe) {ioe.printStackTrace();
    }
}
```

Para borrar un fichero o un directorio usamos el método *delete()*, en el ejemplo anterior no podemos borrar el directorio creado porque contiene ficheros, antes habría que eliminar estos ficheros. Para borrar el objeto *f2* escribimos:

```
if(f2.delete())
    System.out.println("Fichero borrado...");
else
    System.out.println("No se ha podido borrar el fichero...");
```

El método *createNewFile()* puede lanzar la excepción **IOException**, por ello se utiliza el bloque **try-catch**.

## \* 1.3. FLUJOS O STREAMS. TIPOS

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; dicha información puede estar en un fichero en el disco duro, en la memoria, en algún lugar de la red, e incluso en otro programa. Cualquier programa que tenga que obtener información de cualquier fuente necesita abrir un stream, igualmente si necesita enviar información abrirá un stream y se escribirá la información en serie. La vinculación de este stream al dispositivo físico la hace el sistema de entrada y salida de Java. Se definen dos tipos de flujos:

- **Flujos de bytes (8 bits):** realizan operaciones de entradas y salidas de bytes y su uso está orientado a la lectura/escritura de datos binarios. Todas las clases de flujos de bytes descienden de las clases **InputStream** y **OutputStream**, cada una de estas clases tienen varias subclases que controlan las diferencias entre los distintos dispositivos de entrada/salida que se pueden utilizar.
- **Flujos de caracteres (16 bits):** realizan operaciones de entradas y salidas de caracteres. El flujo de caracteres viene gobernado por las clases **Reader** y **Writer**. La razón de ser de estas clases es la internacionalización; la antigua jerarquía de flujos de E/S solo soporta flujos de 8 bits no manejando caracteres Unicode de 16 bits que se utilizaba con fines de internacionalización.

### 1.3.1. Flujos de bytes (Byte streams)

La clase **InputStream** representa las clases que producen entradas de distintas fuentes, estas fuentes pueden ser: un array de bytes, un objeto String, un fichero, una “tubería” (se ponen los elementos en un extremo y salen por el otro), una secuencia de otros flujos, otras fuentes como una conexión a Internet, etc. Los tipos de **InputStream** se resumen en la siguiente tabla:

CLASE	Función
<b>ByteArrayInputStream</b>	Permite usar un espacio de almacenamiento intermedio de memoria
<b>StringBufferInputStream</b>	Convierte un String en un <b>InputStream</b>
<b>FileInputStream</b>	Flujo de entrada hacia fichero, lo usaremos para leer información de un fichero
<b>PipedInputStream</b>	Implementa el concepto de “tubería”
<b>FilterInputStream</b>	Proporciona funcionalidad útil a otras clases <b>InputStream</b>
<b>SequenceInputStream</b>	Convierte dos o más objetos <b>InputStream</b> en un <b>InputStream</b> único

Los tipos de **OutputStream** incluyen las clases que deciden dónde irá la salida: a un array de bytes, un fichero o una “tubería”. Se resumen en la siguiente tabla:

CLASE	Función
ByteArrayOutputStream	Crea un espacio de almacenamiento intermedio en memoria. Todos los datos que se envían al flujo se ubican en este espacio
FileOutputStream	Flujo de salida hacia fichero, lo usaremos para enviar información a un fichero
PipedOutputStream	Cualquier información que se deseé escribir aquí acaba automáticamente como entrada del <b>PipedInputStream</b> asociado. Implementa el concepto de “tubería”
FilterOutputStream	Proporciona funcionalidad útil a otras clases <b>OutputStream</b>

La Figura 1.1 muestra la jerarquía de clases para lectura y escritura de flujos de bytes.

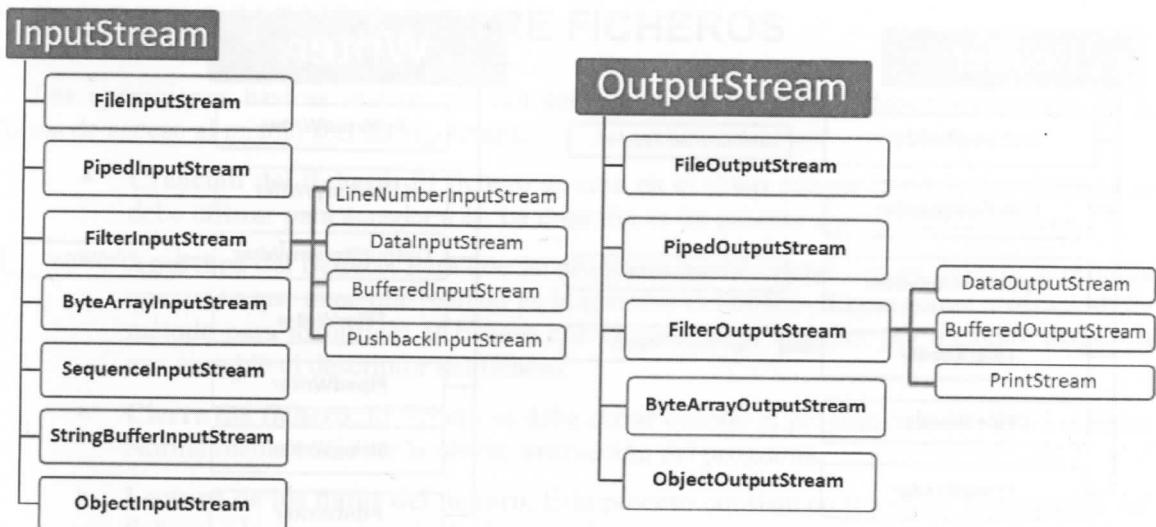


Figura 1.1. Jerarquía de clases para lectura y escritura de bytes.

Dentro de los flujos de bytes están las clases **FileInputStream** y **FileOutputStream** que manipulan los flujos de bytes provenientes o dirigidos hacia ficheros en disco y se estudiarán en los siguientes apartados.

### 1.3.2. Flujos de caracteres (Character streams)

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. Para lograr esto hay clases “puente” (es decir, convierte los streams de bytes a streams de caracteres): **InputStreamReader** que convierte un **InputStream** en un **Reader** y **OutputStreamWriter** que convierte un **OutputStream** en un **Writer** (convierte streams de caracteres a streams de bytes).

La siguiente tabla muestra la correspondencia entre las clases de flujos de bytes y de caracteres:

CLASES DE FLUJOS DE BYTES	CLASE CORRESPONDIENTE DE FLUJO DE CARACTERES
InputStream	Reader, convertidor InputStreamReader
OutputStream	Writer, convertidor OutputStreamReader
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
(sin clase correspondiente)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

La Figura 1.2 muestra la jerarquía de clases para lectura y escritura de flujos de caracteres.

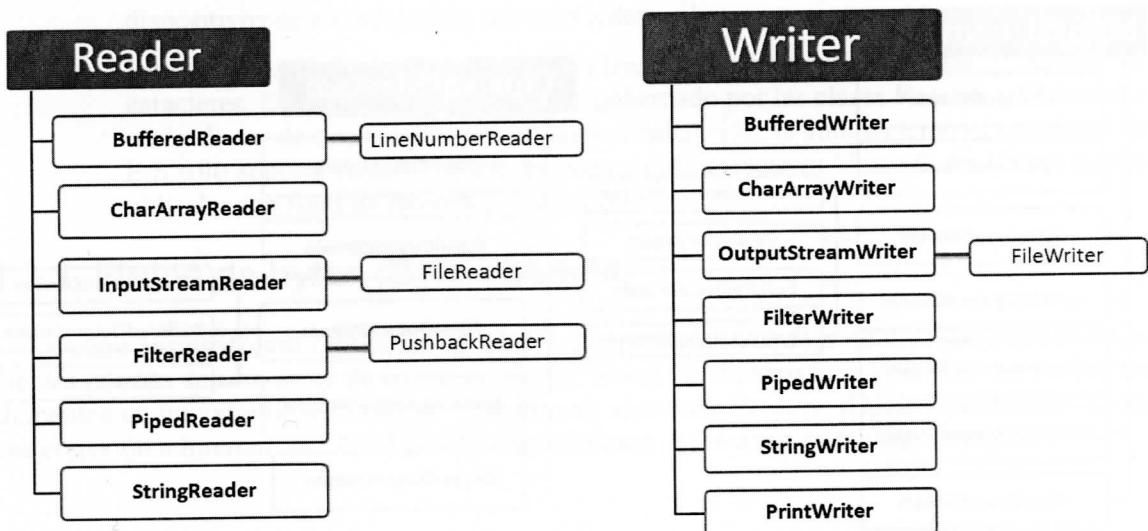


Figura 1.2. Jerarquía de clases para lectura y escritura de flujos de caracteres.

Las clases de flujos de caracteres más importantes son:

- Para acceso a ficheros, lectura y escritura de caracteres en ficheros: **FileReader** y **FileWriter**.
- Para acceso a caracteres, leen y escriben un flujo de caracteres en un array de caracteres: **CharArrayReader** y **CharArrayWriter**.
- Para buferización de datos: **BufferedReader** y **BufferedWriter**, se utilizan para evitar que cada lectura o escritura acceda directamente al fichero, ya que utilizan un buffer intermedio entre la memoria y el stream.

## 1.4. FORMAS DE ACCESO A UN FICHERO

Hay dos formas de acceso a la información almacenada en un fichero: acceso secuencial y acceso directo o aleatorio:

- **Acceso secuencial:** los datos o registros se leen y se escriben en orden, del mismo modo que se hace en una antigua cinta de vídeo. Si se quiere acceder a un dato o un registro que está hacia la mitad del fichero es necesario leer antes todos los anteriores. La escritura de datos se hará a partir del último dato escrito, no es posible hacer inserciones entre los datos que ya hay escritos.
- **Acceso directo o aleatorio:** permite acceder directamente a un dato o registro sin necesidad de leer los anteriores y se puede acceder a la información en cualquier orden. Los datos están almacenados en registros de tamaño conocido, nos podemos mover de un registro a otro de forma aleatoria para leerlos o modificarlos.

En Java el acceso secuencial más común en ficheros puede ser binario o a caracteres. Para el acceso binario: se usan las clases **FileInputStream** y **FileOutputStream**; para el acceso a caracteres (texto) se usan las clases **FileReader** y **FileWriter**. En el acceso aleatorio se utiliza la clase **RandomAccessFile**.

## 1.5. OPERACIONES SOBRE FICHEROS

Las operaciones básicas que se realizan sobre cualquier fichero independientemente de la forma de acceso al mismo son las siguientes:

- **Creación del fichero.** El fichero se crea en el disco con un nombre que después se debe utilizar para acceder a él. La creación es un proceso que se realiza una vez.
- **Apertura del fichero.** Para que un programa pueda operar con un fichero, la primera operación que tiene que realizar es la apertura del mismo. El programa utilizará algún método para identificar el fichero con el que quiere trabajar, por ejemplo, asignar a una variable el descriptor del fichero.
- **Cierre del fichero.** El fichero se debe cerrar cuando el programa no lo vaya a utilizar. Normalmente suele ser la última instrucción del programa.
- **Lectura de los datos del fichero.** Este proceso consiste en transferir información del fichero a la memoria principal, normalmente a través de alguna variable o variables de nuestro programa en las que se depositarán los datos extraídos del fichero.
- **Escritura de datos en el fichero.** En este caso el proceso consiste en transferir información de la memoria (por medio de las variables del programa) al fichero.

Normalmente las operaciones típicas que se realizan sobre un fichero una vez abierto son las siguientes:

- **Altas:** consiste en añadir un nuevo registro al fichero.
- **Bajas:** consiste en eliminar del fichero un registro ya existente. La eliminación puede ser lógica, cambiando el valor de algún campo del registro que usemos para controlar dicha situación; o física, eliminando físicamente el registro del fichero. El borrado físico consiste muchas veces en reescribir de nuevo el fichero en otro fichero sin los datos que se desean eliminar y luego renombrarlo al fichero original.
- **Modificaciones:** consiste en cambiar parte del contenido de un registro. Antes de realizar la modificación será necesario localizar el registro a modificar dentro del fichero; y una vez localizado se realizan los cambios y se reescribe el registro.
- **Consultas:** consiste en buscar en el fichero un registro determinado.

### 1.5.1. Operaciones sobre ficheros secuenciales

En los ficheros secuenciales los registros se insertan en orden cronológico, es decir, un registro se inserta a continuación del último insertado. Si hay que añadir nuevos registros estos se añaden a partir del final del fichero.

Veamos cómo se realizan las operaciones típicas:

- **Consultas:** para consultar un determinado registro es necesario empezar la lectura desde el primer registro, y continuar leyendo secuencialmente hasta localizar el registro buscado. Por ejemplo, si el registro a buscar es el 90 dentro del fichero, será necesario leer secuencialmente los 89 que le preceden.
- **Altas:** en un fichero secuencial las altas se realizan al final del último registro insertado, es decir, solo se permite añadir datos al final del fichero.
- **Bajas:** para dar de baja un registro de un fichero es necesario leer todos los registros uno a uno y escribirlos en un fichero auxiliar, salvo el que deseamos dar de baja. Una vez reescritos hemos de borrar el fichero inicial y renombrar el fichero auxiliar dándole el nombre del fichero original.
- **Modificaciones:** consiste en localizar el registro a modificar, efectuar la modificación y reescribir el fichero inicial en otro fichero auxiliar que incluya el registro modificado. El proceso es similar a las bajas.

Los ficheros secuenciales se usan típicamente en aplicaciones de proceso por lotes como, por ejemplo, en el respaldo de los datos o backup, y son óptimos en dichas aplicaciones si se procesan todos los registros. La **ventaja** de estos ficheros es la rápida capacidad de acceso al siguiente registro (son rápidos cuando se accede a los registros de forma secuencial) y que aprovechan mejor la utilización del espacio. También son sencillos de usar y aplicar.

La **desventaja** es que no se puede acceder directamente a un registro determinado, hay que leer antes todos los anteriores; es decir, no soporta acceso aleatorio. Otra desventaja es el proceso de actualización, la mayoría de los ficheros secuenciales no pueden ser actualizados, habrá que reescribirlos totalmente. Para las aplicaciones interactivas que incluyen peticiones o actualizaciones de registros individuales, los ficheros secuenciales ofrecen un rendimiento pobre.

### 1.5.2. Operaciones sobre ficheros aleatorios

Las operaciones en ficheros aleatorios son las vistas anteriormente, pero teniendo en cuenta que para acceder a un registro hay que localizar la posición o dirección donde se encuentra. Los ficheros de acceso aleatorio en disco manipulan direcciones relativas en lugar de direcciones absolutas (número de pista y número de sector en el disco), lo que hace al programa independiente de la dirección absoluta del fichero en el disco.

Normalmente para posicionarnos en un registro es necesario aplicar una función de conversión, que usualmente tiene que ver con el tamaño del registro y con la clave del mismo (la clave es el campo o campos que identifica de forma única a un registro). Por ejemplo, disponemos de un fichero de empleados con tres campos: identificador, apellido y salario. Usamos el identificador como campo clave del mismo, y le damos el valor 1 para el primer empleado, 2 para el segundo empleado y así sucesivamente; entonces, para localizar al empleado con identificador X necesitamos acceder a la posición  $tamaño*(X-1)$  para acceder a los datos de dicho empleado.

Puede ocurrir que al aplicar la función al campo clave nos devuelva una posición ocupada por otro registro, en ese caso, habría que buscar una nueva posición libre en el fichero para ubicar dicho registro o utilizar una **zona de excedentes** dentro del mismo para ir ubicando estos registros.

Veamos cómo se realizan las operaciones típicas:

- **Consultas:** para consultar un determinado registro necesitamos saber su clave, aplicar la función de conversión a la clave para obtener la dirección y leer el registro ubicado en esa posición. Habría que comprobar si el registro buscado está en esta posición, si no está, se buscaría en la zona de excedentes.
- **Altas:** para insertar un registro necesitamos saber su clave, aplicar la función de conversión a la clave para obtener la dirección y escribir el registro en la posición devuelta. Si la posición está ocupada por otro registro, en ese caso el registro se insertaría en la zona de excedentes.
- **Bajas:** las bajas suelen realizarse de forma lógica, es decir, se suele utilizar un campo del registro a modo de switch que tenga el valor 1 cuando el registro existe y le damos el valor 0 para darle de baja, físicamente el registro no desaparece del disco. Habría que localizar el registro a dar de baja a partir de su campo clave y reescribir en este campo el valor 0.
- **Modificaciones:** para modificar un registro hay que localizarlo, necesitamos saber su clave para aplicar la función de conversión y así obtener la dirección, modificar los datos que nos interesen y reescribir el registro en esa posición.

Una de las principales **ventajas** de los ficheros aleatorios es el rápido acceso a una posición determinada para leer o escribir un registro. El gran **inconveniente** es establecer la relación entre la posición que ocupa el registro y su contenido; ya que a veces al aplicar la función de conversión para obtener la posición se obtienen posiciones ocupadas y hay que recurrir a la zona de excedentes. Otro inconveniente es que se puede desaprovechar parte del espacio destinado al fichero, ya que se pueden producir huecos (posiciones no ocupadas) entre un registro y otro.

## 1.6. CLASES PARA GESTIÓN DE FLUJOS DE DATOS DESDE/HACIA FICHEROS

En Java podemos utilizar dos tipos de ficheros: de texto o binarios; y el acceso a los mismos se puede realizar de forma secuencial o aleatoria. Los ficheros de texto están compuestos de caracteres legibles, mientras que los binarios pueden almacenar cualquier tipo de dato (*int*, *float*, *boolean*, etc.)

### 1.6.1. Ficheros de texto

Los ficheros de texto, los que normalmente se generan con un editor, almacenan caracteres alfanuméricos en un formato estándar (ASCII, UNICODE, UTF8, etc.) Para trabajar con ellos usaremos las clases **FileReader** para leer caracteres y **FileWriter** para escribir los caracteres en el fichero. Cuando trabajamos con ficheros, cada vez que leemos o escribimos en uno debemos hacerlo dentro de un manejador de excepciones **try-catch**. Al usar la clase **FileReader** se puede generar la excepción **FileNotFoundException** (porque el nombre del fichero no exista o no sea válido) y al usar la clase **FileWriter** la excepción **IOException** (el disco está lleno o protegido contra escritura).

Los métodos que proporciona la clase **FileReader** para lectura son los siguientes, estos métodos devuelven el número de caracteres leídos o -1 si se ha llegado al final del fichero:

Método	Función
<code>int read()</code>	Lee un carácter y lo devuelve
<code>int read(char[] buf)</code>	Lee hasta <code>buf.length</code> caracteres de datos de una matriz de caracteres ( <code>buf</code> ). Los caracteres leídos del fichero se van almacenando en <code>buf</code>
<code>int read(char[] buf, int desplazamiento, int n)</code>	Lee hasta <code>n</code> caracteres de datos de la matriz <code>buf</code> comenzando por <code>buf[desplazamiento]</code> y devuelve el número leído de caracteres

En un programa Java para crear o abrir un fichero se invoca a la clase **File** y a continuación se crea el flujo de entrada hacia el fichero con la clase **FileReader**. Después se realizan las operaciones de lectura o escritura y cuando terminemos de usarlo lo cerraremos mediante el método `close()`.

El siguiente ejemplo lee cada uno de los caracteres del fichero de texto de nombre *LeerFichTexto.java* (localizado en la carpeta *C:\EJERCICIOS\UNI1*) y los muestra en pantalla, los métodos `read()` pueden lanzar la excepción **IOException**, por ello en `main()` se ha añadido **throws IOException** ya que no se incluye el manejador **try-catch**:

```
import java.io.*;
public class LeerFichTexto {
    public static void main(String[] args) throws IOException {
        //declarar fichero
        File fichero =
            new File("C:\\EJERCICIOS\\UNI1\\LeerFichTexto.java");
        //crear el flujo de entrada hacia el fichero
        FileReader fic = new FileReader(fichero);
        int i;
        while ((i = fic.read()) != -1) //se va leyendo un carácter
            System.out.println((char) i);
        fic.close(); //cerrar fichero
    }
}
```

En el ejemplo, la expresión `((char) i)` convierte el valor entero recuperado por el método `read()` a carácter, es decir, hacemos un *cast a char*. Se llega al final del fichero cuando el método `read()` devuelve -1. También se puede declarar el fichero de la siguiente manera:

```
FileReader fic =
    new FileReader("C:\\EJERCICIOS\\UNI1\\LeerFichTexto.java");
```

Para ir leyendo de 20 en 20 caracteres escribimos:

```
char b[] = new char[20];
while ((i = fic.read(b)) != -1) System.out.println(b);
```

## ACTIVIDAD 1.2

Crea un fichero de texto con algún editor de textos y después realiza un programa Java que visualice su contenido. Cambia el programa Java para que el nombre del fichero se acepte al ejecutar el programa desde la línea de comandos.

Los métodos que proporciona la clase **FileWriter** para escritura son:

Método	Función
<code>void write(int c)</code>	Escribe un carácter.
<code>void write(char[] buf)</code>	Escribe un array de caracteres.
<code>void write(char[] buf, int desplazamiento , int n)</code>	Escribe n caracteres de datos en la matriz <i>buf</i> comenzando por <i>buf[desplazamiento]</i> .
<code>void write(String str)</code>	Escribe una cadena de caracteres.
<code>void append(char c)</code>	Añade un carácter a un fichero.

Estos métodos también pueden lanzar la excepción **IOException**. Igual que antes declaramos el fichero mediante la clase **File** y a continuación se crea el flujo de salida hacia el fichero con la clase **FileWriter**. El siguiente ejemplo escribe caracteres en un fichero de nombre *FichTexto.txt* (si no existe lo crea). Los caracteres se escriben uno a uno y se obtienen de un *String* que se convierte en array de caracteres:

```
import java.io.*;
public class EscribirFichTexto {
    public static void main(String[] args) throws IOException {
        File fichero = new
            File("C:\\\\EJERCICIOS\\\\UNI1\\\\FichTexto.txt"); //declarar fichero
        //crear flujo de salida
        FileWriter fic = new FileWriter(fichero);

        String cadena = "Esto es una prueba con FileWriter";
        //convierte la cadena en array de caracteres para extraerlos 1 a 1
        char[] cad = cadena.toCharArray();
        for(int i=0; i<cad.length; i++)
            fic.write(cad[i]); //se va escribiendo un carácter

        fic.append('*'); //se añade al final un *
        fic.close(); //cerrar fichero
    }
}
```

En vez de escribir los caracteres uno a uno, también podemos escribir todo el array usando **fic.write(cad)**. El siguiente ejemplo escribe cadenas de caracteres que se obtienen de un array de *String*, las cadenas se irán insertando en el fichero una a continuación de la otra sin saltos de línea:

```
String prov[] =
    {"Albacete", "Avila", "Badajoz", "Cáceres", "Huelva", "Jaén",
     "Madrid", "Segovia", "Soria", "Toledo", "Valladolid", "Zamora"};

for(int i=0; i<prov.length; i++) fic.write(prov[i]);
```

Hay que tener en cuenta que si el fichero existe cuando vayamos a escribir caracteres sobre él, todo lo que tenía almacenado anteriormente se borrará. Si queremos añadir caracteres al final, usaremos la clase **FileWriter** de la siguiente manera, colocando en el segundo parámetro del constructor el valor **true**:

```
FileWriter fic = new FileWriter(fichero,true);
```

**FileReader** no contiene métodos que nos permitan leer líneas completas, pero **BufferedReader** sí; dispone del método **readLine()** que lee una línea del fichero y la devuelve, o devuelve **null** si no hay nada que leer o llegamos al final del fichero. También dispone del método **read()** para leer un carácter. Para construir un **BufferedReader** necesitamos la clase **FileReader**:

```
BufferedReader fichero = new  
    BufferedReader (new FileReader(NombreFichero));
```

El siguiente ejemplo lee el fichero *LeerFichTexto.java* línea por línea y las va visualizando en pantalla, en este caso, las instrucciones se han agrupado dentro de un bloque **try-catch**:

```
import java.io.*;  
public class LeerFichTextoBuf {  
    public static void main(String[] args) {  
        try{  
            BufferedReader fichero = new BufferedReader(  
                new FileReader("LeerFichTexto.java"));  
            String linea;  
            while((linea = fichero.readLine())!=null)  
                System.out.println(linea);  
  
            fichero.close();  
        }  
        catch (FileNotFoundException fn ){  
            System.out.println("No se encuentra el fichero");}  
        catch (IOException io) {  
            System.out.println("Error de E/S ");}  
    }  
}
```

La clase **BufferedWriter** también deriva de la clase **Writer**. Esta clase añade un buffer para realizar una escritura eficiente de caracteres. Para construir un **BufferedWriter** necesitamos la clase **FileWriter**:

```
BufferedWriter fichero = new  
    BufferedWriter(new FileWriter(NombreFichero));
```

El siguiente ejemplo escribe 10 filas de caracteres en un fichero de texto y después de escribir cada fila salta una línea con el método **newLine()**:

```
import java.io.*;  
public class EscribirFichTextoBuf {  
    public static void main(String[] args) {  
        try{  
            BufferedWriter fichero = new BufferedWriter
```

```

        (new FileWriter("FichTexto.txt"));
for (int i=1; i<11; i++) {
    fichero.write("Fila numero: "+i); //escribe una línea
    fichero.newLine(); //escribe un salto de línea
}
fichero.close();
}
catch (FileNotFoundException fn ) {
    System.out.println("No se encuentra el fichero");
}
catch (IOException io) {
    System.out.println("Error de E/S ");
}
}
}

```

La clase **PrintWriter**, que también deriva de **Writer**, posee los métodos ***print(String)*** y ***println(String)*** (idénticos a los de *System.out*) para escribir en un fichero. Ambos reciben un *String* y lo escriben en un fichero, el segundo método, además, produce un salto de línea. Para construir un **PrintWriter** necesitamos la clase **FileWriter**:

```
PrintWriter fichero = new
    PrintWriter(new FileWriter(NombreFichero));
```

El ejemplo anterior usando la clase **PrintWriter** y el método ***println()*** quedaría así:

```
PrintWriter fichero = new PrintWriter
    (new FileWriter("FichTexto.txt"));
for(int i=1; i<11; i++)
    fichero.println("Fila numero: "+i);
fichero.close();
```

## 1.6.2. Ficheros binarios

Los ficheros binarios almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario como ocurría con los ficheros de texto. Tienen la ventaja de que ocupan menos espacio en disco. En Java, las dos clases que nos permiten trabajar con ficheros son **FileInputStream** (para entrada) y **FileOutputStream** (para salida), estas trabajan con flujos de bytes y crean un enlace entre el flujo de bytes y el fichero.

Los métodos que proporciona la clase **FileInputStream** para lectura son similares a los vistos para la clase **FileReader**, estos métodos devuelven el número de bytes leídos o -1 si se ha llegado al final del fichero:

Método	Función
<code>int read()</code>	Lee un byte y lo devuelve
<code>int read(byte[] b)</code>	Lee hasta <i>b.length</i> bytes de datos de una matriz de bytes
<code>int read(byte[] b, int desplazamiento, int n)</code>	Lee hasta <i>n</i> bytes de la matriz <i>b</i> comenzando por <i>b[desplazamiento]</i> y devuelve el número leído de bytes

Los métodos que proporciona la clase **FileOutputStream** para escritura son:

Método	Función
void write(int b)	Escribe un byte
void write(byte[] b)	Escribe <i>b.length</i> bytes
void write(byte[] b, int desplazamiento, int n)	Escribe <i>n</i> bytes a partir de la matriz de bytes de entrada comenzando por <i>b[desplazamiento]</i>

El siguiente ejemplo escribe bytes en un fichero y después los visualiza:

```
import java.io.*;
public class EscribirFichBytes {
    public static void main(String[] args) throws IOException {
        File fichero = new
            File("C:\\\\EJERCICIOS\\\\UNI1\\\\FichBytes.dat"); //declara fichero
        //crea flujo de salida hacia el fichero
        FileOutputStream fileout = new FileOutputStream(fichero);

        //crea flujo de entrada
        FileInputStream filein = new FileInputStream(fichero);
        int i;

        for (i=1; i<100; i++)
            fileout.write(i); //escribe datos en el flujo de salida

        fileout.close(); //cerrar stream de salida

        //visualizar los datos del fichero
        while ((i = filein.read()) != -1) //lee datos del flujo de entrada
            System.out.println(i);
        filein.close(); //cerrar stream de entrada
    }
}
```

Para añadir bytes al final del fichero usaremos **FileOutputStream** de la siguiente manera, colocando en el segundo parámetro del constructor el valor **true**:

```
FileOutputStream fileout = new FileOutputStream(fichero,true);
```

Para leer y escribir datos de tipos primitivos: *int*, *float*, *long*, etc usaremos las clases **DataInputStream** y **DataOutputStream**. Estas clases definen diversos métodos *readXXX* y *writeXXX* que son variaciones de los métodos *read()* y *write()* de la clase base para leer y escribir datos de tipo primitivo. Algunos de los métodos se muestran en la siguiente tabla:

MÉTODOS PARA LECTURA	MÉTODOS PARA ESCRITURA
boolean readBoolean();	void writeBoolean(boolean v);
byte readByte();	void writeByte(int v);
int readUnsignedByte();	void writeBytes(String s);
int readUnsignedShort();	void writeShort(int v);
short readShort();	void writeChars(String s);
char readChar();	void writeChar(int v);
int readInt();	void writeInt(int v);
long readLong();	void writeLong(long v);
float readFloat();	void writeFloat(float v);
double readDouble();	void writeDouble(double v);
String readUTF();	void writeUTF(String str);

Para abrir un objeto **DataInputStream**, se utilizan los mismos métodos que para **FileInputStream**, ejemplo:

```
File fichero = new File("FichData.dat");
FileInputStream filein = new FileInputStream(fichero);
DataInputStream dataIS = new DataInputStream(filein);
```

O bien

```
File fichero = new File("FichData.dat");
DataInputStream dataIS = new
    DataInputStream(new FileInputStream(fichero));
```

Para abrir un objeto **DataOutputStream**, se utilizan los mismos métodos que para **FileOutputStream**, ejemplo:

```
File fichero = new File("FichData.dat");
FileOutputStream fileout = new FileOutputStream(fichero);
DataOutputStream dataOS = new DataOutputStream(fileout);
```

O bien

```
File fichero = new File("FichData.dat");
DataOutputStream dataOS = new
    DataOutputStream(new FileOutputStream(fichero));
```

El siguiente ejemplo inserta datos en el fichero *FichData.dat*, los datos los toma de dos arrays, uno contiene los nombres de una serie de personas y el otro sus edades, recorremos los arrays y vamos escribiendo en el fichero el nombre (mediante el método **writeUTF(String)**) y la edad (mediante el método **writeInt(int)**):

```
import java.io.*;
public class EscribirFichData {
    public static void main(String[] args) throws IOException {

        File fichero = new File("FichData.dat");
        FileOutputStream fileout = new FileOutputStream(fichero);
        DataOutputStream dataOS = new DataOutputStream(fileout);

        String nombres[] =
            {"Ana", "Luis Miguel", "Alicia", "Pedro", "Manuel",
             "Andrés", "Julio", "Antonio", "María Jesús"};

        int edades[] = {14, 15, 13, 15, 16, 12, 16, 14, 13};

        for (int i=0; i<edades.length; i++) {
            dataOS.writeUTF(nombres[i]); //escribe nombre
            dataOS.writeInt(edades[i]); //escribe edad
        }
        dataOS.close(); //cerrar stream
    }
}
```

El siguiente ejemplo visualiza los datos grabados anteriormente en el fichero, se deben recuperar en el mismo orden en el que se escribieron, es decir, primero obtenemos el nombre y luego la edad:

```
import java.io.*;
public class LeerFichData {
    public static void main(String[] args) throws IOException {
        File fichero = new File("FichData.dat");
        FileInputStream filein = new FileInputStream(fichero);
        DataInputStream dataIS = new DataInputStream(filein);
        String n;
        int e;

        try {
            while (true) {
                n = dataIS.readUTF(); //recupera el nombre
                e = dataIS.readInt(); //recupera la edad
                System.out.println("Nombre: " + n + ", edad: " + e);
            }
        } catch (EOFException eo) {}

        dataIS.close(); //cerrar stream
    }
}
```

Se obtiene la siguiente salida al ejecutar el programa:

```
Nombre: Ana, edad: 14
Nombre: Luis Miguel, edad: 15
Nombre: Alicia, edad: 13
Nombre: Pedro, edad: 15
Nombre: Manuel, edad: 16
Nombre: Andrés, edad: 12
Nombre: Julio, edad: 16
Nombre: Antonio, edad: 14
Nombre: María Jesús, edad: 13
```

### 1.6.3. Objetos en ficheros binarios

Hemos visto cómo se guardan los tipos de datos primitivos en un fichero, pero, por ejemplo, si tenemos un objeto de tipo empleado con varios atributos (el nombre, la dirección, el salario, el departamento, el oficio, etc.) y queremos guardarlos en un fichero, tendríamos que guardar cada atributo que forma parte del objeto por separado, esto se vuelve engoroso si tenemos gran cantidad de objetos. Por ello Java nos permite guardar objetos en ficheros binarios; para poder hacerlo, el objeto tiene que implementar la interfaz **Serializable** que dispone de una serie de métodos con los que podremos guardar y leer objetos en ficheros binarios. Los más importantes a utilizar son:

- **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**.

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.

Para leer y escribir objetos serializables a un stream se utilizan las clases Java **ObjectInputStream** y **ObjectOutputStream** respectivamente. A continuación se muestra la clase *Persona* que implementa la interfaz **Serializable** y que utilizaremos para escribir y leer objetos en un fichero binario. La clase tiene dos atributos: el nombre y la edad y los métodos *get* para obtener el valor del atributo y *set* para darle valor:

```
import java.io.Serializable;
public class Persona implements Serializable{
    private String nombre;
    private int edad;

    public Persona(String nombre,int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public Persona() {
        this.nombre = null;
    }
    public void setNombre(String nombre){this.nombre = nombre;}
    public void setEdad(int edad){this.edad = edad;}

    public String getNombre(){return this.nombre;}//devuelve nombre
    public int getEdad(){return this.edad;}           //devuelve edad
}
//fin Persona
```

El siguiente ejemplo escribe objetos *Persona* en un fichero. Necesitamos crear un flujo de salida a disco con **FileOutputStream** y a continuación se crea el flujo de salida **ObjectOutputStream** que es el que procesa los datos y se ha de vincular al fichero de **FileOutputStream**:

```
File fichero = new File("FichPersona.dat");
FileOutputStream fileout = new FileOutputStream(fichero);
ObjectOutputStream dataOS = new ObjectOutputStream(fileout);
```

O bien:

```
File fichero = new File("FichPersona.dat");
ObjectOutputStream dataOS = new
    ObjectOutputStream(new FileOutputStream(fichero));
```

El método **writeObject()** escribe los objetos al flujo de salida y los guarda en un fichero en disco: **dataOS.writeObject(persona)**. El código es el siguiente:

```
import java.io.*;
public class EscribirFichObject {
    public static void main(String[] args) throws IOException {
        Persona persona;//defino variable persona
        //declara el fichero
        File fichero = new File("FichPersona.dat");
```

```

//crea el flujo de salida
FileOutputStream fileout = new FileOutputStream(fichero);
//conecta el flujo de bytes al flujo de datos
ObjectOutputStream dataOS = new ObjectOutputStream(fileout);

String nombres[] = {"Ana", "Luis Miguel", "Alicia", "Pedro",
    "Manuel", "Andrés", "Julio", "Antonio", "María Jesús"};

int edades[] = {14,15,13,15,16,12,16,14,13};

for (int i=0;i<edades.length; i++){ //recorro los arrays
    persona= new Persona(nombres[i],edades[i]);
    dataOS.writeObject(persona); //escribo la persona en el fichero
}
dataOS.close(); //cerrar stream de salida
}
}
}

```

Para leer objetos *Persona* del fichero necesitamos el flujo de entrada a disco **FileInputStream** y a continuación crear el flujo de entrada **ObjectInputStream** que es el que procesa los datos y se ha de vincular al fichero de **FileInputStream**:

```

File fichero = new File("FichPersona.dat");
FileInputStream filein = new FileInputStream(fichero);
ObjectInputStream dataIS = new ObjectInputStream(filein);

```

O bien:

```

File fichero = new File("FichPersona.dat");
ObjectInputStream dataIS = new
    ObjectInputStream(new FileInputStream(fichero));

```

El método *readObject()* lee los objetos del flujo de entrada, puede lanzar la excepción **ClassNotFoundException** e **IOException**, por lo que será necesario controlarlas. El proceso de lectura se hace en un bucle *while(true)*, este se encierra en un bloque **try-catch** ya que la lectura finalizará cuando se llegue al final de fichero, entonces, se lanzará la excepción **EOFException**. El código es el siguiente:

```

import java.io.*;

public class LeerFichObject {
    public static void main(String[] args) throws
        IOException, ClassNotFoundException{
        Persona persona; //defino la variable persona
        File fichero = new File("FichPersona.dat");
        //crea el flujo de entrada
        FileInputStream filein = new FileInputStream(fichero);
        //conecta el flujo de bytes al flujo de datos
        ObjectInputStream dataIS = new ObjectInputStream(filein);

        try {
            while (true) { //lectura del fichero
                persona= (Persona) dataIS.readObject(); //leer una Persona
                System.out.printf("Nombre: %s, edad: %d %n",

```

```

        persona.getNombre(), persona.getEdad());
    }
} catch (EOFException eo) {
    System.out.println("FIN DE LECTURA.");
}

dataIS.close(); //cerrar stream de entrada
}
}

```

### Problema con los ficheros de objetos:

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 esa cabecera. El problema surge al leer el fichero cuando en la lectura se encuentra con la segunda cabecera, y aparece la excepción **StreamCorruptedException** y no podremos leer más objetos.

La cabecera se crea cada vez que se pone **new ObjectOutputStream(fichero)**. Para que no se añadan estas cabeceras lo que se hace es *redefinir la clase ObjectOutputStream creando una nueva clase que la herede (extends)*. Y dentro de esa clase se redefine el método **writeStreamHeader()** que es el que escribe las cabeceras, y hacemos que ese método no haga nada. De manera que si el fichero ya se ha creado se llamará a ese método de la clase redefinida.

La clase redefinida quedará así:

```

public class MiObjectOutputStream extends ObjectOutputStream
{
    public MiObjectOutputStream(OutputStream out) throws IOException
    {   super(out);   }
    protected MiObjectOutputStream()
        throws IOException, SecurityException
    {   super();   }
    // Redefinición del método de escribir la cabecera
    // para que no haga nada.
    protected void writeStreamHeader() throws IOException
    {   }
}

```

Y dentro de nuestro programa a la hora de abrir el fichero para añadir nuevos objetos se pregunta si ya existe, si existe, se crea el objeto con la clase redefinida, y si no existe, el fichero se crea con la clase **ObjectOutputStream**:

```

File fichero = new File(nombrefichero);
ObjectOutputStream dataOS;
if (!fichero.exists())
{ //Si el fichero no existe crea un ObjectOutputStream, la primera vez
    FileOutputStream fileout;
    fileout = new FileOutputStream(fichero);
    dataOS = new ObjectOutputStream(fileout);
}
else
{ // Si ya existe el fichero creará un ObjectOutputStream
}

```

```
// con el método writeStreamHeader redefinido (sin hacer nada)
dataOS = new MiObjectOutputStream
        (new FileOutputStream(fichero,true));
} //fin if
```

### 1.6.4. Ficheros de acceso aleatorio

Hasta ahora, todas las operaciones que hemos realizado sobre los ficheros se realizaban de forma secuencial. Se empezaba la lectura en el primer byte o el primer carácter o el primer objeto, y seguidamente se leían los siguientes uno a continuación de otro hasta llegar al fin del fichero. Igualmente cuando escribíamos los datos en el fichero se iban escribiendo a continuación de la última información escrita. Java dispone de la clase **RandomAccessFile** que dispone de métodos para acceder al contenido de un fichero binario de forma aleatoria (no secuencial) y para posicionarnos en una posición concreta del mismo. Esta clase no es parte de la jerarquía **InputStream/OutputStream**, ya que su comportamiento es totalmente distinto puesto que se puede avanzar y retroceder dentro de un fichero.

Disponemos de dos constructores para crear el fichero de acceso aleatorio, estos pueden lanzar la excepción **FileNotFoundException**:

- **RandomAccessFile(String nombrefichero, String modoAcceso):** escribiendo el nombre del fichero incluido el path.
- **RandomAccessFile(File objetoFile, String modoAcceso):** con un objeto **File** asociado a un fichero.

El argumento *modoAcceso* puede tener dos valores:

Modo de acceso	Significado
r	Abre el fichero en modo de solo lectura. El fichero debe existir. Una operación de escritura en este fichero lanzará la excepción <b>IOException</b>
rw	Abre el fichero en modo lectura y escritura. Si el fichero no existe se crea

Una vez abierto el fichero pueden usarse los métodos *readXXX* y *writeXXX* de las clases **DataInputStream** y **DataOutputStream** (vistos anteriormente). La clase **RandomAccessFile** maneja un puntero que indica la posición actual en el fichero. Cuando el fichero se crea el puntero al fichero se coloca en 0, apuntando al principio del mismo. Las sucesivas llamadas a los métodos *read()* y *write()* ajustan el puntero según la cantidad de bytes leídos o escritos.

Los métodos más importantes son:

Método	Función
<code>long getFilePointer()</code>	Devuelve la posición actual del puntero del fichero
<code>void seek(long posicion)</code>	Coloca el puntero del fichero en una posición determinada desde el comienzo del mismo
<code>long length()</code>	Devuelve el tamaño del fichero en bytes. La posición <i>length()</i> marca el final del fichero
<code>int skipBytes(int desplazamiento)</code>	Desplaza el puntero desde la posición actual el número de bytes indicados en <i>desplazamiento</i>

El ejemplo que se muestra a continuación inserta datos de empleados en un fichero aleatorio. Los datos a insertar: apellido, departamento y salario, se obtienen de varios arrays que se llenan en el programa, los datos se van introduciendo de forma secuencial por lo que no va a ser necesario usar el método *seek()*. Por cada empleado también se insertará un identificador (mayor que 0) que coincidirá con el índice +1 con el que se recorren los arrays. La longitud del registro de cada empleado es la misma (36 bytes) y los tipos que se insertan y su tamaño en bytes es el siguiente:

- Se inserta en primer lugar un entero, que es el identificador, ocupa 4 bytes.
- A continuación una cadena de 10 caracteres, es el apellido. Como Java utiliza caracteres UNICODE, cada carácter de una cadena de caracteres ocupa 16 bits (2 bytes), por tanto, el apellido ocupa 20 bytes.
- Un tipo entero que es el departamento, ocupa 4 bytes.
- Un tipo Double que es el salario, ocupa 8 bytes.

Tamaño de otros tipos: short (2 bytes), byte (1 byte), long (8 bytes), boolean (1bit), float (4 bytes), etc.

El fichero se abre en modo “*rw*” para lectura y escritura. El código es el siguiente:

```
import java.io.*;
public class EscribirFichAleatorio {
    public static void main(String[] args) throws IOException {
        File fichero = new File("AleatorioEmple.dat");
        //declara el fichero de acceso aleatorio
        RandomAccessFile file = new RandomAccessFile(fichero, "rw");

        //arrays con los datos
        String apellido[] = {"FERNANDEZ", "GIL", "LOPEZ", "RAMOS",
                             "SEVILLA", "CASILLA", "REY"}; //apellidos
        int dep[] = {10, 20, 10, 10, 30, 30, 20}; //departamentos
        Double salario[]={1000.45, 2400.60, 3000.0, 1500.56,
                          2200.0, 1435.87, 2000.0}; //salarios

        StringBuffer buffer = null; //buffer para almacenar apellido
        int n = apellido.length;//número de elementos del array

        for (int i = 0; i<n; i++){ //recorro los arrays
            file.writeInt(i+1); //uso i+1 para identificar empleado

            buffer = new StringBuffer( apellido[i] );
            buffer.setLength(10); //10 caracteres para el apellido
            file.writeChars(buffer.toString());//insertar apellido

            file.writeInt(dep[i]); //insertar departamento
            file.writeDouble(salario[i]); //insertar salario
        }
        file.close(); //cerrar fichero
    }
}
```

El siguiente ejemplo toma el fichero anterior y visualiza todos los registros. El posicionamiento para empezar a recorrer los registros empieza en 0, para recuperar los siguientes registros hay que sumar 36 (tamaño del registro) a la variable utilizada para el posicionamiento:

```
import java.io.*;
public class LeerFichAleatorio {
    public static void main(String[] args) throws IOException {
        File fichero = new File("AleatorioEmple.dat");
        //declara el fichero de acceso aleatorio
        RandomAccessFile file = new RandomAccessFile(fichero, "r");
        //
        int id, dep, posicion;
        Double salario;
        char apellido[] = new char[10], aux;

        posicion = 0; //para situarnos al principio

        for(;;){ //recorro el fichero
            file.seek(posicion); //nos posicionamos en posicion
            id = file.readInt(); // obtengo id de empleado

            //recorro uno a uno los caracteres del apellido
            for (int i = 0; i < apellido.length; i++) {
                aux = file.readChar();
                apellido[i] = aux; //los voy guardando en el array
            }

            //convierto a String el array
            String apellidos = new String(apellido);
            dep = file.readInt(); //obtengo dep
            salario = file.readDouble(); //obtengo salario

            if(id > 0)
                System.out.printf("ID: %s, Apellido: %s, Departamento: %d,
                                  Salario: %.2f %n",
                                  id, apellidos.trim(), dep, salario);

            //me posiciono para el sig empleado, cada empleado ocupa 36 bytes
            posicion= posicion + 36;

            //Si he recorrido todos los bytes salgo del for
            if (file.getFilePointer() == file.length())break;
        }
    }
}
```

La ejecución muestra la siguiente salida:

```
ID: 1, Apellido: FERNANDEZ, Departamento: 10, Salario: 1000.45
ID: 2, Apellido: GIL, Departamento: 20, Salario: 2400.6
ID: 3, Apellido: LOPEZ, Departamento: 10, Salario: 3000.0
ID: 4, Apellido: RAMOS, Departamento: 10, Salario: 1500.56
ID: 5, Apellido: SEVILLA, Departamento: 30, Salario: 2200.0
ID: 6, Apellido: CASILLA, Departamento: 30, Salario: 1435.87
ID: 7, Apellido: REY, Departamento: 20, Salario: 2000.0
```

Para consultar un empleado determinado no es necesario recorrer todos los registros del fichero, conociendo su identificador podemos acceder a la posición que ocupa dentro del mismo y obtener sus datos. Por ejemplo, supongamos que se desean obtener los datos del empleado con identificador 5, para calcular la posición hemos de tener en cuenta los bytes que ocupa cada registro (en este ejemplo son 36 bytes):

```
int identificador = 5;
//calcula donde empieza el registro
posicion = (identificador - 1) * 36;
if(posicion >= file.length())
    System.out.printf("ID: %d, NO EXISTE EMPLEADO...", identificador);
else{
    file.seek(posicion); //nos posicionamos
    id = file.readInt(); //obtengo id de empleado
    //obtener resto de los datos, como en el ejemplo anterior
}
```

Para añadir registros a partir del último insertado hemos de posicionar el puntero del fichero al final del mismo:

```
long posicion= file.length() ;
file.seek(posicion) ;
```

Para insertar un nuevo registro aplicamos la función al identificador para calcular la posición. El siguiente ejemplo inserta un empleado con identificador 20, se ha de calcular la posición donde irá el registro dentro del fichero (*identificador -1*) \* 36 bytes:

```
StringBuffer buffer = null; //buffer para almacenar apellido
String apellido = "GONZALEZ"; //apellido a insertar
Double salario = 1230.87; //salario
int id = 20; //id del empleado
int dep = 10; //dep del empleado

long posicion = (id -1 ) * 36; //calculamos la posición

file.seek(posicion); //nos posicionamos
file.writeInt(id); //se escribe id
buffer = new StringBuffer( apellido);
buffer.setLength(10); //10 caracteres para el apellido
file.writeChars(buffer.toString()); //insertar apellido
file.writeInt(dep); //insertar departamento
file.writeDouble(salario); //insertar salario

file.close(); //cerrar fichero
```

### ACTIVIDAD 1.3

**Consulta.** Crea un programa Java que consulte los datos de un empleado del fichero aleatorio. El programa se ejecutará desde la línea de comandos y debe recibir un identificador de empleado. Si el empleado existe se visualizarán sus datos, si no existe se visualizará un mensaje indicándolo.

**Inserción.** Crea un programa Java que inserte datos en el fichero aleatorio. El programa se ejecutará desde la línea de comandos y debe recibir 4 parámetros: identificador de empleado, apellido, departamento y salario. Antes de insertar se comprobará si el identificador existe, en ese caso se debe visualizar un mensaje indicándolo; si no existe se deberá insertar.

---

Para modificar un registro determinado, accedemos a su posición y efectuamos las modificaciones. El fichero debe abrirse en modo “**rw**”. Por ejemplo, para cambiar el departamento y salario del empleado con identificador 4 escribo lo siguiente:

```
int registro = 4;                                //id a modificar
long posicion = (registro - 1) * 36;              //calculo la posición
posición = posición + 4 + 20;                      //sumo el tamaño de ID + apellido
file.seek(posicion);                            //nos posicionamos
file.writeInt(40);                               //modifico departamento
file.writeDouble(4000.87);                         //modifico salario
```

---

### ACTIVIDAD 1.4

**Modificación.** Crea un programa Java que reciba desde la línea de comandos un identificador de empleado y un importe. Se debe realizar la modificación del salario. La modificación consistirá en sumar al salario del empleado el importe introducido. El programa debe visualizar el apellido, el salario antiguo y el nuevo. Si el identificador no existe se visualizará mensaje indicándolo.

**Borrado.** Crea un programa Java que al ejecutarlo desde la línea de comandos reciba un identificador de empleado y lo borre. Se hará un borrado lógico marcando el registro con la siguiente información: el identificador será igual a -1, el apellido será igual al identificador que se borra, y el departamento y salario serán 0.

A continuación haz otro programa Java (o crea un método dentro del anterior programa) que muestre los identificadores de los empleados borrados.

---

## 1.7. TRABAJO CON FICHEROS XML

XML (*eXtensible Markup Language- Lenguaje de Etiquetado Extensible*) es un metalenguaje, es decir, un lenguaje para la definición de lenguajes de marcado. Nos permite jerarquizar y estructurar la información y describir los contenidos dentro del propio documento. Los ficheros XML son ficheros de texto escritos en lenguaje XML, donde la información está organizada de forma secuencial y en orden jerárquico. Existen una serie de marcas especiales como son los símbolos menor que, < y mayor que , > que se usan para delimitar las marcas que dan la estructura al documento. Cada marca tiene un nombre y puede tener 0 o más atributos. Un fichero XML sencillo tiene la siguiente estructura:

```
<?xml version="1.0"?>
<Empleados>
    <empleado>
        <id>1</id>
        <apellido>FERNANDEZ</apellido>
```

```

<dep>10</dep>
<salario>1000.45</salario>
</empleado>
<empleado>
  <id>2</id>
  <apellido>GIL</apellido>
  <dep>20</dep>
  <salario>2400.6</salario>
</empleado>
<empleado>
  <id>3</id>
  <apellido>LOPEZ</apellido>
  <dep>10</dep>
  <salario>3000.0</salario>
</empleado>
</Empleados>

```

Los ficheros XML se pueden utilizar para proporcionar datos a una base de datos, o para almacenar copias de partes del contenido de la base de datos. También se utilizan para escribir ficheros de configuración de programas o en el protocolo SOAP (*Simple Object Access Protocol*), para ejecutar comandos en servidores remotos; la información enviada al servidor remoto y el resultado de la ejecución del comando se envían en ficheros XML.

Para leer los ficheros XML y acceder a su contenido y estructura, se utiliza un procesador de XML o parser. El procesador lee los documentos y proporciona acceso a su contenido y estructura. Algunos de los procesadores más empleados son: **DOM**: *Modelo de Objetos de Documento* y **SAX**: *API Simple para XML*. Son independientes del lenguaje de programación y existen versiones particulares para Java, VisualBasic, C, etc. Utilizan dos enfoques muy diferentes:

- **DOM**: un procesador XML que utilice este planteamiento almacena toda la estructura del documento en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales (que son aquellos que no tienen descendientes). Una vez creado el árbol, se van recorriendo los diferentes nodos y se analiza a qué tipo particular pertenecen. Tiene su origen en el W3C. Este tipo de procesamiento necesita más recursos de memoria y tiempo sobre todo si los ficheros XML a procesar son bastante grandes y complejos.
- **SAX**: un procesador que utilice este planteamiento lee un fichero XML de forma secuencial y produce una secuencia de eventos (comienzo/fín del documento, comienzo/fín de una etiqueta, etc.) en función de los resultados de la lectura. Cada evento invoca a un método definido por el programador. Este tipo de procesamiento prácticamente no consume memoria, pero por otra parte, impide tener una visión global del documento por el que navegar.

### 1.7.1. Acceso a ficheros XML con DOM

Para poder trabajar con DOM en Java necesitamos las clases e interfaces que componen el paquete **org.w3c.dom** (contenido en el JSDK) y el paquete **javax.xml.parsers** del API estándar de Java que proporciona un par de clases abstractas que toda implementación DOM para Java debe extender. Estas clases ofrecen métodos para cargar documentos desde una fuente de datos

(fichero, **InputStream**, etc.) Contiene dos clases fundamentales: **DocumentBuilderFactory** y **DocumentBuilder**.

DOM no define ningún mecanismo para generar un fichero XML a partir de un árbol DOM. Para eso usaremos el paquete **javax.xml.transform** que permite especificar una fuente y un resultado. La fuente y el resultado pueden ser ficheros, flujos de datos o nodos DOM entre otros.

Los programas Java que utilicen DOM necesitan estas interfaces (no se exponen todas, solo algunas de las que usaremos en los ejemplos):

- **Document**. Es un objeto que equivale a un ejemplar de un documento XML. Permite crear nuevos nodos en el documento.
- **Element**. Cada elemento del documento XML tiene un equivalente en un objeto de este tipo. Expone propiedades y métodos para manipular los elementos del documento y sus atributos.
- **Node**. Representa a 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**. Son los datos carácter de un elemento.
- **CharacterData**. Representa a los datos carácter presentes en el documento. Proporciona atributos y métodos para manipular los datos de caracteres.
- **DocumentType**. Proporciona información contenida en la etiqueta <!DOCTYPE>.

A continuación vamos a crear un fichero XML a partir del fichero aleatorio de empleados creado en el epígrafe anterior. Lo primero que hemos de hacer es importar los paquetes necesarios:

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

A continuación creamos una instancia de **DocumentBuilderFactory** para construir el parser, se debe encerrar entre **try-catch** porque se puede producir la excepción **ParserConfigurationException**:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();  
try{  
    DocumentBuilder builder = factory.newDocumentBuilder();  
    . . . . .
```

Creamos un documento vacío de nombre *document* con el nodo raíz de nombre *Empleados* y asignamos la versión del XML, la interfaz **DOMImplementation** permite crear objetos **Document** con nodo raíz:

```
DOMImplementation implementation = builder.getDOMImplementation();  
Document document = implementation.createDocument  
        (null, "Empleados", null);
```

```
document.setXmlVersion("1.0"); // asignamos la version de nuestro XML
```

El siguiente paso sería recorrer el fichero con los datos de los empleados y por cada registro crear un nodo empleado con 4 hijos (*id*, *apellido*, *dep* y *salario*). Cada nodo hijo tendrá su valor (por ejemplo: *1*, *FERNANDEZ*, *10*, *1000.45*). Para crear un elemento usamos el método ***createElement(String)*** llevando como parámetro el nombre que se pone entre las etiquetas menor que y mayor que. El siguiente código crea y añade el nodo <empleado> al documento:

```
//Creamos el nodo empleado
Element raiz = document.createElement("empleado");
//Lo pegamos a la raiz del documento
document.getDocumentElement().appendChild(raiz);
```

A continuación se añaden los hijos de ese nodo (*raiz*), estos se añaden en el método ***CrearElemento()***:

```
//Añadir ID
CrearElemento("id", Integer.toString(id), raiz, document);
//Añadir APELLIDO
CrearElemento("apellido", apellidoS.trim(), raiz, document);
//Añadir DEP
CrearElemento("dep", Integer.toString(dep), raiz, document);
//Añadir SALARIO
CrearElemento("salario", Double.toString(salario), raiz, document);
```

Como se puede ver el método recibe el nombre del nodo hijo (*id*, *apellido*, *dep* o *salario*) y sus textos o valores que tienen que estar en formato String (*1*, *FERNANDEZ*, *10*, *1000.45*), el nodo al que se va a añadir (*raiz*) y el documento (*document*). Para crear el nodo hijo (<*id*> o <*apellido*> o <*dep*> o <*salario*>) se escribe:

```
Element elem = document.createElement(datoEmple); //Creamos un hijo
```

Para añadir su valor o su texto se usa el método ***createTextNode(String)***:

```
Text text = document.createTextNode(valor); //damos valor
```

A continuación se añade el nodo hijo a la raíz (empleado) y su texto o valor al nodo hijo:

```
raiz.appendChild(elem); //Pegamos el elemento hijo a la raiz
elem.appendChild(text); //Pegamos el valor al elemento
```

Al final se generaría algo similar a esto por cada empleado:

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

El método es el siguiente:

```
static void CrearElemento(String datoEmple, String valor,
                           Element raiz, Document document){
    Element elem = document.createElement(datoEmple); //Creamos hijo
    Text text = document.createTextNode(valor); //damos valor
    raiz.appendChild(elem); //Pegamos el elemento hijo a la raiz
    elem.appendChild(text); //Pegamos el valor
}
```

En los últimos pasos se crea la fuente XML a partir del documento:

```
Source source = new DOMSource(document);
```

Se crea el resultado en el fichero *Empleados.xml*:

```
Result result = new StreamResult  
    (new java.io.File("Empleados.xml")); //fichero XML
```

Se obtiene un **TransformerFactory**:

```
Transformer transformer =  
    TransformerFactory.newInstance().newTransformer();
```

Se realiza la transformación del documento a fichero:

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

Para mostrar el documento por pantalla podemos especificar como resultado el canal de salida *System.out*:

```
Result console = new StreamResult(System.out);  
transformer.transform(source, console);
```

El código completo es el siguiente:

```
import org.w3c.dom.*;  
import javax.xml.parsers.*;  
import javax.xml.transform.*;  
import javax.xml.transform.dom.*;  
import javax.xml.transform.stream.*;  
import java.io.*;  
  
public class CrearEmpleadoXml {  
    public static void main(String args[]) throws IOException{  
        File fichero = new File("AleatorioEmple.dat");  
        RandomAccessFile file = new RandomAccessFile(fichero, "r");  
  
        int id, dep, posicion=0; //para situarnos al principio del fichero  
        Double salario;  
        char apellido[] = new char[10], aux;  
  
        DocumentBuilderFactory factory =  
            DocumentBuilderFactory.newInstance();  
  
        try{  
            DocumentBuilder builder = factory.newDocumentBuilder();  
            DOMImplementation implementation = builder.getDOMImplementation();  
            Document document =  
                implementation.createDocument(null, "Empleados", null);  
            document.setXmlVersion("1.0");  
  
            for(;;){  
                file.seek(posicion); //nos posicionamos  
                id=file.readInt(); // obtengo id de empleado
```

```

for (int i = 0; i < apellido.length; i++) {
    aux = file.readChar();
    apellido[i] = aux;
}
String apellidos = new String(apellido);
dep = file.readInt();
salario = file.readDouble();

if(id>0) { //id validos a partir de 1
    Element raiz =
        document.createElement("empleado"); //nodo empleado
    document.getDocumentElement().appendChild(raiz);

    //añadir ID
    CrearElemento("id",Integer.toString(id), raiz, document);
    //Apellido
    CrearElemento("apellido",apellidos.trim(), raiz, document);
    //añadir DEP
    CrearElemento("dep",Integer.toString(dep), raiz, document);
    //añadir salario
    CrearElemento("salario",Double.toString(salario), raiz,
                  document);
}
posicion= posicion + 36; // me posiciono para el sig empleado
if (file.getFilePointer() == file.length()) break;

}//fin del for que recorre el fichero

Source source = new DOMSource(document);
Result result =
    new StreamResult(new java.io.File("Empleados.xml"));
Transformer transformer =
    TransformerFactory.newInstance().newTransformer();
transformer.transform(source, result);

}catch(Exception e){ System.err.println("Error: "+e); }

file.close(); //cerrar fichero
}//fin de main

//Inserción de los datos del empleado
static void CrearElemento(String datoEmple, String valor,
                           Element raiz, Document document){
    Element elem = document.createElement(datoEmple);
    Text text = document.createTextNode(valor); //damos valor
    raiz.appendChild(elem); //pegamos el elemento hijo a la raiz
    elem.appendChild(text); //pegamos el valor
}
}//fin de la clase

```

Para leer un documento XML, creamos una instancia de **DocumentBuilderFactory** para construir el parser y cargamos el documento con el método **parse()**:

```
Document document = builder.parse(new File("Empleados.xml"));
```

Obtenemos la lista de nodos con nombre *empleado* de todo el documento:

```
NodeList empleados = document.getElementsByTagName("empleado");
```

Se realiza un bucle para recorrer esta lista de nodos. Por cada nodo se obtienen sus etiquetas y sus valores llamando a la función *getNodo()*. El código es el siguiente:

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

public class LecturaEmpleadoXml {
    public static void main(String[] args) {

        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();

        try {
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document document = builder.parse(new File("Empleados.xml"));
            document.getDocumentElement().normalize();

            System.out.printf("Elemento raiz: %s %n",
                document.getDocumentElement().getNodeName());
            //crea una lista con todos los nodos empleado
            NodeList empleados = document.getElementsByTagName("empleado");
            System.out.printf("Nodos empleado a recorrer: %d %n",
                empleados.getLength());

            //recorrer la lista
            for (int i = 0; i < empleados.getLength(); i++) {
                Node emple = empleados.item(i); //obtener un nodo empleado
                if (emple.getNodeType() == Node.ELEMENT_NODE) {//tipo de nodo
                    //obtener los elementos del nodo
                    Element elemento = (Element) emple;
                    System.out.printf("ID = %s %n",
                        elemento.getElementsByTagName("id").
                            item(0).getTextContent());
                    System.out.printf(" * Apellido = %s %n",
                        elemento.getElementsByTagName("apellido").
                            item(0).getTextContent());
                    System.out.printf(" * Departamento = %s %n",
                        elemento.getElementsByTagName("dep").
                            item(0).getTextContent());
                    System.out.printf(" * Salario = %s %n",
                        elemento.getElementsByTagName("salario").
                            item(0).getTextContent());
                }
            }
        } catch (Exception e)
        {e.printStackTrace();}
    }
} //fin de main
//fin de la clase
```

**¡ ¡INTERESANTE !!**

API DOM: <http://docs.oracle.com/javase/8/docs/api/org/w3c/dom/package-tree.html>.

**ACTIVIDAD 1.5**

A partir del fichero de objetos Persona utilizado anteriormente crea un documento XML usando DOM.

**1.7.2. Acceso a ficheros 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 (es decir, no carga todo el fichero en memoria como hace DOM), esto implica poco consumo de memoria aunque los documentos sean de gran tamaño, en contraposición, impide tener una visión global del documento que se va a analizar. SAX es más complejo de programar que DOM, es un API totalmente escrita en Java e incluida dentro del JRE que nos permite crear nuestro propio parser de XML.

La lectura de un documento XML produce eventos que ocasiona la llamada a métodos, los eventos son encontrar la etiqueta de inicio y fin del documento (*startDocument()* y *endDocument()*), la etiqueta de inicio y fin de un elemento (*startElement()* y *endElement()*), los caracteres entre etiquetas (*characters()*), etc:

Documento XML (alumnos.xml)	Métodos asociados a eventos del documento
<pre>&lt;?xml version="1.0"?&gt; &lt;listadealumnos&gt;   &lt;alumno&gt;     &lt;nombre&gt;       Juan     &lt;/nombre&gt;     &lt;edad&gt;       19     &lt;/edad&gt;   &lt;/alumno&gt;   &lt;alumno&gt;     &lt;nombre&gt;       Maria     &lt;/nombre&gt;     &lt;edad&gt;       20     &lt;/edad&gt;   &lt;/alumno&gt; &lt;/listadealumnos&gt;</pre>	<pre>startDocument() startElement() startElement()   startElement()     characters()   endElement()   startElement()     characters()   endElement() endElement() startElement()   startElement()     characters()   endElement()   startElement()     characters()   endElement() endElement() endElement() endDocument()</pre>

Vamos a construir un ejemplo sencillo en Java que muestra los pasos básicos necesarios para hacer que se puedan tratar los eventos. En primer lugar se incluyen las clases e interfaces de SAX:

```

import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;

```

Se crea un objeto procesador de XML, es decir, un **XMLReader**, durante la creación de este objeto se puede producir una excepción (**SAXException**) que es necesario capturar (se incluye en el método *main()*):

```
XMLReader procesadorXML = XMLReaderFactory.createXMLReader();
```

A continuación hay que indicar al **XMLReader** qué objetos poseen los métodos que tratarán los eventos. Estos objetos serán normalmente implementaciones de las siguientes interfaces:

- **ContentHandler**: recibe las notificaciones de los eventos que ocurren en el documento.
- **DTDHandler**: recoge eventos relacionados con la DTD.
- **ErrorHandler**: define métodos de tratamientos 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, el programador definirá los métodos que sean utilizados por el programa. Esta clase es de la que extenderemos para poder crear nuestro parser de XML. En el ejemplo, la clase se llama *GestionContenido* y se tratan solo los eventos básicos: inicio y fin de documento, inicio y fin de etiqueta encontrada, encuentra datos carácter (*startDocument()*, *endDocument()*, *startElement()*, *endElement()*, *characters()*):
  - ***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.

Para indicar al procesador XML los objetos que realizarán el tratamiento se utiliza alguno de los siguientes métodos incluidos dentro de los objetos **XMLReader**: *setContentHandler()*, *setDTDHandler()*, *setEntityResolver()* y  *setErrorHandler()*; cada uno trata un tipo de evento y está asociado con una interfaz determinada. En el ejemplo usaremos *setContentHandler()* para tratar los eventos que ocurren en el documento:

```
GestionContenido gestor = new GestionContenido();
procesadorXML.setContentHandler(gestor);
```

A continuación se define el fichero XML que se va a leer mediante un objeto **InputSource**:

```
InputSource fileXML = new InputSource("alumnos.xml");
```

Por último, se procesa el documento XML mediante el método *parse()* del objeto **XMLReader**, le pasamos un objeto **InputSource**:

```
procesadorXML.parse(fileXML);
```

El ejemplo completo se muestra a continuación:

```
import java.io.*;
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;

public class PruebaSax1 {
    public static void main(String[] args)
        throws FileNotFoundException, IOException, SAXException{

        XMLReader procesadorXML = XMLReaderFactory.createXMLReader();
        GestionContenido gestor = new GestionContenido();
        procesadorXML.setContentHandler(gestor);
        InputSource fileXML = new InputSource("alumnos.xml");
        procesadorXML.parse(fileXML);
    }
}//fin PruebaSax1

class GestionContenido extends DefaultHandler {
    public GestionContenido() {
        super();
    }
    public void startDocument() {
        System.out.println("Comienzo del Documento XML");
    }
    public void endDocument() {
        System.out.println("Final del Documento XML");
    }
    public void startElement(String uri, String nombre,
                           String nombreC, Attributes atts) {
        System.out.printf("\tPrincipio Elemento: %s %n", nombre);
    }
    public void endElement(String uri, String nombre,
                           String nombreC) {
        System.out.printf("\tFin Elemento: %s %n", nombre);
    }
    public void characters(char[] ch, int inicio, int longitud)
        throws SAXException {
        String car = new String(ch, inicio, longitud);
        //quitar saltos de línea
        car = car.replaceAll("[\t\n]", "");
        System.out.printf ("\tCaracteres: %s %n", car);
    }
}//fin GestionContenido
```

En el resultado de ejecutar el programa con el fichero *alumnos.xml* se puede observar cómo el orden de ocurrencia de los eventos está relacionado con la estructura del documento:

Comienzo del Documento XML Principio Elemento: listadealumnos Caracteres: Principio Elemento: alumno Caracteres: Principio Elemento: nombre Caracteres: Maria Fin Elemento: nombre Caracteres: Principio Elemento: edad Caracteres: 20 Fin Elemento: edad Caracteres: Fin Elemento: alumno	Caracteres: Principio Elemento: alumno Caracteres: Principio Elemento: nombre Caracteres: Maria Fin Elemento: nombre Caracteres: Principio Elemento: edad Caracteres: 20 Fin Elemento: edad Caracteres: Fin Elemento: alumno Caracteres: Fin Elemento: listadealumnos Final del Documento XML
---	---

---

### ¡ ¡INTERESANTE ! !

API SAX: <http://www.saxproject.org/apidoc/org/xml/sax/package-tree.html>.

---

---

### ACTIVIDAD 1.6

Utiliza SAX para visualizar el contenido del fichero *Empleados.xml* creado anteriormente.

---

### 1.7.3. Serialización de objetos a XML

A continuación vamos a ver cómo se pueden serializar de forma sencilla objetos Java a XML y viceversa; utilizaremos para ello la librería **XStream**. Para poder utilizarla hemos de descargarlos los JAR desde el sitio Web: <http://x-stream.github.io/download.html>. Para el ejemplo se ha descargado el fichero **xstream-distribution-1.4.8-bin.zip** que hemos de descomprimir y buscar el JAR **xstream-1.4.8.jar** que está en la carpeta *lib* que es el que usaremos para el ejemplo. También necesitamos el fichero **kxml2-2.3.0.jar** que se localiza en la carpeta *lib\xstream*. Una vez que tenemos los dos ficheros los añadimos a nuestro proyecto Eclipse o NetBeans o los definimos en el CLASSPATH, por ejemplo, supongamos que tenemos los JAR en la carpeta *D:\uni1\xstream*, el CLASSPATH nos quedaría:

```
SET CLASSPATH =  
. ;D:\uni1\xstream\kxml2-2.3.0.jar;D:\uni1\xstream\xstream-1.4.8.jar
```

Partimos del fichero *FichPersona.dat* que utilizamos en epígrafes anteriores y contiene objetos *Persona*. Crearemos una lista de objetos *Persona* y la convertiremos en un fichero de datos XML. Necesitaremos la clase *Persona* (ya definida) y la clase *ListaPersonas* en la que se define una lista de objetos *Persona* que pasaremos al fichero XML:

```

import java.util.ArrayList;
import java.util.List;
public class ListaPersonas {
    private List<Persona> lista = new ArrayList<Persona>();
    public ListaPersonas(){ }
    public void add(Persona per) {
        lista.add(per);
    }
    public List<Persona> getListaPersonas() {
        return lista;
    }
}

```

El proceso consistirá en recorrer el fichero *FichPersona.dat* para crear una lista de personas que después se insertarán en el fichero *Personas.xml*, el código Java es el siguiente (fichero *EscribirPersonas.java*):

```

import java.io.*;
import com.thoughtworks.xstream.XStream;

public class EscribirPersonas {
    public static void main(String[] args) throws IOException,
                                                ClassNotFoundException {
        File fichero = new File("FichPersona.dat");
        FileInputStream filein = new FileInputStream(fichero);
        ObjectInputStream dataIS = new ObjectInputStream(filein);

        System.out.println("Comienza el proceso...");

        //Creamos un objeto Lista de Personas
        ListaPersonas listaper = new ListaPersonas();

        try {
            while (true) { //lectura del fichero
                Persona persona= (Persona) dataIS.readObject();
                listaper.add(persona); //añadir persona a la lista
            }
        } catch (EOFException eo) {}
        dataIS.close(); //cerrar stream de entrada

        try {
            XStream xstream = new XStream();
            //cambiar de nombre a las etiquetas XML
            xstream.alias("ListaPersonasMunicipio", ListaPersonas.class);
            xstream.alias("DatosPersona", Persona.class);
            //quitar etiqueta lista (atributo de la clase ListaPersonas)
            xstream.addImplicitCollection(ListaPersonas.class, "lista");
            //Insertar los objetos en el XML
            xstream.toXML(listaper,new FileOutputStream("Personas.xml"));
            System.out.println("Creado fichero XML....");

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

```

```

    } // fin main
} //fin EscribirPersonas

```

El fichero generado tiene el siguiente aspecto:

```

<ListaPersonasMunicipio>
  <DatosPersona>
    <nombre>Ana</nombre>
    <edad>14</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Luis Miguel</nombre>
    <edad>15</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Alicia</nombre>
    <edad>13</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Pedro</nombre>
    <edad>15</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Manuel</nombre>
    <edad>16</edad>
  </DatosPersona>
</ListaPersonasMunicipio>
  <DatosPersona>
    <nombre>Andrés</nombre>
    <edad>12</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Julio</nombre>
    <edad>16</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Antonio</nombre>
    <edad>14</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>María Jesús</nombre>
    <edad>13</edad>
  </DatosPersona>
</ListaPersonasMunicipio>

```

En primer lugar para utilizar **XStream**, simplemente creamos una instancia de la clase **XStream**:

```
xstream = new XStream();
```

En general las etiquetas XML se corresponden con el nombre de los atributos de la clase, pero se pueden cambiar usando el método **alias(String alias, Class clase)**. En el ejemplo se ha dado un alias a la clase *ListaPersonas*, en el XML aparecerá con el nombre *ListaPersonasMunicipio*:

```
xstream.alias("ListaPersonasMunicipio", ListaPersonas.class);
```

También se ha dado un alias a la clase *Persona*, en el XML aparecerá con el nombre *DatosPersona*:

```
xstream.alias("DatosPersona", Persona.class);
```

El método **aliasField(String alias, Class clase, String nombrecampo)**, permite crear un alias para un nombre de campo. Por ejemplo, si queremos cambiar el nombre de los campos *nombre* y *edad* (de la clase *Persona*) creariamos los siguientes alias:

```
xstream.aliasField("Nombre alumno", Persona.class, "nombre");
xstream.aliasField("Edad alumno", Persona.class, "edad");
```

Entonces en el fichero XML se crearán con las etiquetas *<Nombre alumno>* en lugar de *<nombre>* y *<Edad alumno>* en lugar de *<edad>*.

Para que no aparezca el atributo *lista* de la clase *ListaPersonas* en el XML generado se ha utilizado el método ***addImplicitCollection(Class clase, String nombredelcampo)***

```
xstream.addImplicitCollection(ListaPersonas.class, "lista");
```

Por último, para generar el fichero *Personas.xml* a partir de la lista de objetos se utiliza el método ***toXML(Object objeto, OutputStream out)***:

```
xstream.toXML(listaper, new FileOutputStream("Personas.xml"));
```

El proceso para realizar la lectura del fichero XML generado es el siguiente:

```
import java.io.*;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import com.thoughtworks.xstream.XStream;

public class LeerPersonas {
    public static void main(String[] args) throws IOException {

        XStream xstream = new XStream();

        xstream.alias("ListaPersonasMunicipio", ListaPersonas.class);
        xstream.alias("DatosPersona", Persona.class);
        xstream.addImplicitCollection(ListaPersonas.class, "lista");

        ListaPersonas listadoTodas = (ListaPersonas)
            xstream.fromXML(new FileInputStream("Personas.xml"));

        System.out.println("Numero de Personas: " +
                           listadoTodas.getListaPersonas().size());

        List<Persona> listaPersonas = new ArrayList<Persona>();
        listaPersonas = listadoTodas.getListaPersonas();

        Iterator iterador = listaPersonas.listIterator();
        while( iterador.hasNext() ) {
            Persona p = (Persona) iterador.next();
            System.out.printf("Nombre: %s, edad: %d %n",
                              p.getNombre(), p.getEdad());
        }
        System.out.println("Fin de listado .....");
    } //fin main
} //fin LeerPersonas
```

Se deben utilizar los métodos ***alias()*** y ***addImplicitCollection()*** para leer el XML ya que se usaron para hacer la escritura del mismo. Para obtener el objeto con la lista de personas o lo que es lo mismo para deserializar el objeto a partir del fichero, utilizamos el método ***fromXML(InputStream input)*** que devuelve un tipo **Object**:

```
ListaPersonas listadoTodas = (ListaPersonas)
    xstream.fromXML(new FileInputStream("Personas.xml"));
```

**¡¡INTERESANTE !!**

API XStream: <http://x-stream.github.io/javadoc/index.html>.

### 1.7.4. Conversión de ficheros XML a otro formato

**XSL (Extensible Stylesheet Language)** es toda una familia de recomendaciones del *World Wide Web Consortium* (<http://www.w3.org/Style/XSL/>) para expresar hojas de estilo en lenguaje XML. Una hoja de estilo XSL describe el proceso de presentación a través de un pequeño conjunto de elementos XML. Esta hoja, puede contener elementos de reglas que representan a las reglas de construcción y elementos de reglas de estilo que representan a las reglas de mezcla de estilos. En el siguiente ejemplo vamos a ver cómo a partir de un fichero XML que contiene datos y otro XSL que contiene la presentación de esos datos, se puede generar un fichero HTML usando el lenguaje Java. Los ficheros son los siguientes:

**FICHERO alumnos.xml:**

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

**FICHERO alumnosPlantilla.xsl:**

```
<?xml version="1.0" encoding='ISO-8859-1'?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match='/'>
        <html><xsl:apply-templates /></html>
    </xsl:template>
    <xsl:template match='listadealumnos'>
        <head><title>LISTADO DE ALUMNOS</title></head>
        <body>
            <h1>LISTA DE ALUMNOS</h1>
            <table border='1'>
                <tr><th>Nombre</th><th>Edad</th></tr>
                <xsl:apply-templates select='alumno' />
            </table>
        </body>
    </xsl:template>

    <xsl:template match='alumno'>
        <tr><xsl:apply-templates /></tr>
    </xsl:template>

    <xsl:template match='nombre|edad'>
        <td><xsl:apply-templates /></td>
    </xsl:template>
</xsl:stylesheet>
```

Para realizar la transformación se necesita obtener un objeto **Transformer** que se obtiene creando una instancia de **TransformerFactory** y aplicando el método ***newTransformer(Source source)*** a la fuente XSL que vamos a utilizar para aplicar la transformación del fichero de datos XML, o lo que es lo mismo, para aplicar la hoja de estilos XSL al fichero XML:

```
Transformer transformer =
    TransformerFactory.newInstance().newTransformer(estilos);
```

La transformación se consigue llamando al método ***transform(Source fuenteXml, Result resultado)***, pasándole los datos (el fichero XML) y el stream de salida (el fichero HTML):

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

El código es el siguiente:

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

public class convertidor {
    public static void main(String[] args) throws IOException{
        String hojaEstilo = "alumnosPlantilla.xsl";
        String datosAlumnos = "alumnos.xml";
        File pagHTML = new File("mipagina.html");

        //crear fichero HTML
        FileOutputStream os = new FileOutputStream(pagHTML);
        Source estilos = new StreamSource(hojaEstilo); //fuente XSL
        Source datos = new StreamSource(datosAlumnos); //fuente XML

        //resultado de la transformación
        Result result = new StreamResult(os);

        try{
            Transformer transformer =
                TransformerFactory.newInstance().newTransformer(estilos);
            transformer.transform(datos, result); //obtiene el HTML
        }
        catch(Exception e){System.err.println("Error: "+e);}

        os.close(); //cerrar fichero
    }//de main
}//de la clase
```

El fichero HTML generado se muestra en la Figura 1.3.



Figura 1.3. Fichero HTML generado.

**¡ ¡INTERESANTE !!**API de Java: <http://docs.oracle.com/javase/8/docs/api/index.html>.

## 1.8. EXCEPCIONES: DETECCIÓN Y TRATAMIENTO

Una **excepción** es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias. Cuando no es capturada por el programa, es capturada por el gestor de excepciones por defecto que retorna un mensaje y detiene el programa. La ejecución del siguiente programa produce una excepción y visualiza un mensaje indicando el error:

```
public class ejemploExcepcion {
    public static void main(String[] args) {
        int nume = 10, denom = 0, cociente;
        cociente = nume / denom;
        System.out.printf("Resultado: %d", cociente);
    }
} //
```

```
D:\unil>java ejemploExcepcion
Exception in thread "main" java.lang.ArithmetricException: / by zero
at ejemploExcepcion.main(ejemploExcepcion.java:4)
```

Cuando dicho error ocurre dentro de un método Java, el método crea un objeto **Exception** y lo maneja fuera, en el sistema de ejecución. El manejo de excepciones en Java está diseñado pensando en situaciones en las que el método que detecta un error no es capaz de manejarlo, un método así **lanzará una excepción**.

Las excepciones en Java son objetos de clases derivadas de la clase base **Exception** que a su vez es una clase derivada de la clase base **Throwable**.

### 1.8.1. Capturar excepciones

Para capturar una excepción se utiliza el bloque **try-catch**. Se encierra en un bloque **try** el código que puede generar una excepción, este bloque va seguido por uno o más bloques **catch**. Cada bloque **catch** especifica el tipo de excepción que puede atrapar y contiene un manejador de excepciones. Después del último bloque **catch** puede aparecer un bloque **finally** (opcional) que siempre se ejecuta haya ocurrido o no la excepción; se utiliza el bloque **finally** para cerrar ficheros o liberar otros recursos del sistema después de que ocurra una excepción:

```

try {
    //Código que puede generar excepciones
} catch(excepcion1 e1) {
    //manejo de la excepcion1
} catch(excepcion2 e2) {
    //manejo de la excepcion2
}
//etc .....
finally {
    // Se ejecuta después de try o catch
}

```

El siguiente ejemplo muestra la captura de 3 tipos de excepciones que se pueden producir. Cuando se encuentra el primer error se produce un salto al bloque **catch** que maneja dicho error; en este caso al encontrar la sentencia de asignación *arraynum[10] = 20;* se lanza la excepción **ArrayIndexOutOfBoundsException** (ya que el array está definido para 4 elementos y se da un valor al elemento de la posición 10) donde se ejecutan las instrucciones indicadas en el bloque, las sentencias situadas debajo de la que causó el error dentro del bloque **try** no se ejecutarán:

```

public class ejemploExcepciones {
public static void main(String[] args) {
    String cad1 = "20", cad2 = "0", mensaje;
    int nume, denom, cociente;
    int[] arraynum = new int[4];
    try {
        //código que puede producir errores
        arraynum[10] = 20;    //sentencia que produce la excepción
        nume = Integer.parseInt(cad1);      //no se ejecuta
        denom = Integer.parseInt(cad2);     //no se ejecuta
        cociente = nume/denom;            //no se ejecuta
        mensaje = String.valueOf(cociente); //no se ejecuta
    } catch(NumberFormatException ex) {
        mensaje = "Caracteres no numéricos";
    } catch(ArithmetricException ex) {
        mensaje = "Division por cero";
    } catch (ArrayIndexOutOfBoundsException ex) {
        mensaje = "Fuera de rango en el array";
    } finally {
        System.out.println("SE EJECUTA SIEMPRE");
    }
    System.out.println(mensaje); //sí se ejecuta
}//fin de main
}//fin de la clase

```

El programa muestra la siguiente salida:

```
D:\unil>java ejemploExcepciones
SE EJECUTA SIEMPRE
Fuera de rango en el array
```

Para capturar cualquier excepción utilizamos la clase base **Exception**. Si se usa habrá que ponerla al final de la lista de manejadores para evitar que los manejadores que vienen después queden ignorados. Por ejemplo, el siguiente código maneja varias excepciones, si se produce alguna para la que no se ha definido manejador será capturada por **Exception**:

```
try {
    //código que puede producir errores
} catch(NumberFormatException ex) {
    //tratamiento excepción
} catch(ArithmetricException ex) {
    //tratamiento excepción
} catch (ArrayIndexOutOfBoundsException ex) {
    //tratamiento excepción
} catch (Exception ex) {
    //tratamiento si se produce cualquier otra excepción
} finally {
    //se ejecuta haya o no excepción
}
```

Para obtener más información sobre la excepción se puede llamar a los métodos de la clase base **Throwable**, algunos son:

Método	Función
<code>String getMessage()</code>	Devuelve la cadena de error del objeto
<code>String getLocalizedMessage()</code>	Crea una descripción local de este objeto
<code>String toString()</code>	Devuelve una breve descripción del objeto
<code>void printStackTrace(), printStackTrace(PrintStream) o printStackTrace(PrintWriter)</code>	Visualiza el objeto y la traza de pila de llamadas lanzada

Por ejemplo, el siguiente bloque **try-catch**:

```
try {
    arraynum[10] = 20;
    nume=Integer.parseInt(cad1);
    denom=Integer.parseInt(cad2);
    cociente = nume / denom;
    mensaje = String.valueOf(cociente);
} catch(Exception ex){
    System.err.println("toString"                  => "+ ex.toString());
    System.err.println("getMessage"                => "+ ex.getMessage());
    System.err.println("getLocalizedMessage=> " +
                       ex.getLocalizedMessage());
    ex.printStackTrace();
} finally {
    System.out.println("SE EJECUTA SIEMPRE");
}
```

Muestra la siguiente salida al ejecutarse:

```
toString          => java.lang.ArrayIndexOutOfBoundsException: 10
getMessage        => 10
getLocalizedMessage=> 10
java.lang.ArrayIndexOutOfBoundsException: 10
    at ejemploExcepciones2.main(ejemploExcepciones2.java:8)
SE EJECUTA SIEMPRE
```

Una sentencia **try** puede estar dentro de un bloque de otra sentencia **try**. Si la sentencia **try** interna no tiene un manejador **catch**, se busca el manejador en las sentencias **try** más externas.

## 1.8.2. Especificar excepciones

Para especificar excepciones utilizamos la palabra clave **throws**, seguida de la lista de todos los tipos de excepciones potenciales; si un método decide no gestionar una excepción (mediante **try-catch**), debe especificar que puede lanzar esa excepción. El siguiente ejemplo indica que el método **main()** puede lanzar las excepciones **IOException** y **ClassNotFoundException**:

```
public static void main(String[] args) throws IOException,
                                              ClassNotFoundException {
```

Aquellos métodos que pueden lanzar excepciones, deben saber cuáles son esas excepciones en su declaración. Una forma típica de saberlo es compilando el programa. Por ejemplo, si al programa *EscribirPersonas.java* (visto en el epígrafe anterior) le quitamos la cláusula **throws** al método **main()**, al compilarlo aparecerán errores:

```
EscribirPersonas.java:7: unreported exception
java.io.FileNotFoundException; must be caught or declared to be thrown
FileInputStream filein = new FileInputStream(fichero); //crea el flujo
de entrada
EscribirPersonas.java:9: unreported exception java.io.IOException; must
be caught or declared to be thrown
ObjectInputStream dataIS = new
ObjectInputStream(filein);
.....
EscribirPersonas.java:17: unreported exception
java.lang.ClassNotFoundException; must be caught or declared to be
thrown
    Persona persona= (Persona) dataIS.readObject(); //leer una
Persona
```

Indica que en la línea 7 (marcada en negrita) se produce una excepción que no se ha declarado (**FileNotFoundException**) esta excepción debe ser capturada mediante un bloque **try-catch** o declarada para ser lanzada mediante **throws**. También se producen errores de excepciones no declaradas en las líneas 9 y 17 (marcadas en negrita).

El ejemplo anterior (*EscribirPersonas.java*) manejando las excepciones dentro de bloques **try-catch** quedaría así:

```
public class EscribirPersonas2 {
    public static void main(String[] args) {
        File fichero = new File("FichPersona.dat");
        FileInputStream filein;
        try {
```

```

filein = new FileInputStream(fichero);
ObjectInputStream dataIS = new ObjectInputStream(filein);
System.out.println("Comienza el proceso ...");

ListaPersonas listaper = new ListaPersonas();

try {
    while (true) { //lectura del fichero
        Persona persona= (Persona) dataIS.readObject();
        listaper.add(persona); //añadir persona a la lista
    }//del while
} catch (EOFException eo) {//fin de fichero bo hago nada
} catch (ClassNotFoundException cn) {
    cn.printStackTrace();
}//final bloque try interno

dataIS.close(); //cerrar stream de entrada

XStream xstream = new XStream();
xstream.alias("ListaPersonasMunicipio", ListaPersonas.class);

xstream.alias("DatosPersona", Persona.class);
xstream.addImplicitCollection(ListaPersonas.class, "lista");
xstream.toXML(listaper, new FileOutputStream("Personas.xml"));

System.out.println("Creado fichero XML....");

} catch (IOException io) {
    io.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
}//final bloque try mas externo
} // fin main
} //fin EscribirPersonas
}

```

Podemos consultar la API de Java <http://docs.oracle.com/javase/8/docs/api/> para ver las excepciones que se pueden producir en los diferentes paquetes.

## 1.9. INTRODUCCIÓN A JAXB

**JAXB** es una tecnología Java que permite mapear clases Java a representaciones XML, y viceversa, es decir, serializar objetos Java a representaciones XML. JAXB provee dos funciones fundamentales:

- La capacidad de presentar un objeto Java en XML (serializar), al proceso lo llamaremos *marshall* o *marshalling*. Java Object a XML.
- Lo contrario, es decir, presentar un XML en un objeto Java (deserializar), al proceso lo llamaremos *unmarshall* o *unmarshalling*. XML a Java Object

También el compilador que proporciona JAXB nos va a permitir generar clases Java a partir de esquemas XML, que podrán ser llamadas desde las aplicaciones a través de métodos *sets* y *gets* para obtener o establecer los datos de un documento XML.

### 1.9.1. Mapear clases Java a representaciones XML

Para crear objetos Java en XML, vamos a utilizar ***JavaBeans***, que serán las clases que se van a mapear. Son clases primitivas Java (**POJOs** - *Plain Old Java Objects*) con las propiedades, *getter* y *setter*, el constructor sin parámetros y el constructor con las propiedades. En estas clases que se van a mapear se añadirán las **Anotaciones**, que son las indicaciones que ayudan a convertir el JavaBean en XML.

Las principales **anotaciones** son:

- **@XmlRootElement(namespace = "namespace")**: Define la raíz del XML. Si una clase va a ser la raíz del documento se añadirá esta anotación, el *namespace* es opcional.

```
@XmlElement
public class ClaseRaiz {
    ...
}
```

- **@XmlType(propOrder = { "field2", "field1", ... })**: Permite definir en qué orden se van a escribir los elementos (o las etiquetas) dentro del XML. Si es una clase **que no va a ser raíz** añadiremos **@XmlType**.
- **@XmlElement(name = "nombre")**: Define el elemento de XML que se va usar.

A cualquiera de ellos podemos ponerle entre paréntesis el nombre de etiqueta que queramos que salga en el documento XML para la clase, añadiendo el atributo ***name***. Sería algo como esto

```
@XmlElement(name = "Un_Nombre_para_la_raiz")
@XmlType(name = "Otro_Nombre")
```

Para cada atributo de la clase que queramos que salga en el XML, el método *get* correspondiente a ese atributo debe llevar una anotación **@XmlElement**, a la que a su vez podemos ponerle un nombre (estas anotaciones no son obligatorias, solo si se desean nombres diferentes del atributo):

```
@XmlElement(name = "La_ClaseRaiz")
public class UnaClase {
    private String unAtributo;
    @XmlElement(name = "El_Atributo")
    String getUnAtributo() {
        return this.unAtributo;
    }
}
```

Si el atributo es una colección (array, list, etc...) debe llevar dos anotaciones, **@XmlElementWrapper** y **@XmlElement**, esta última, con un nombre si se desea. Por ejemplo:

```
@XmlElement(name = "La_ClaseRaiz")
public class UnaClase {
    private String [] unArray;

    @XmlElementWrapper
    @XmlElement(name = "Elemento_Array")
    String [] getUnArray() {
        return this.unArray;
    }
}
```

Si el atributo es otra clase (otro JavaBean), le ponemos igualmente `@XmlElement` al método `get`, pero la clase que hace de atributo debería llevar a la vez sus anotaciones correspondientes.

**Ejemplo1:** se desea generar el siguiente documento XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<libreria>
    <ListaLibro>
        <Libro>
            <autor>Alicia Ramos</autor>
            <nombre>Entornos de Desarrollo</nombre>
            <editorial>Garceta</editorial>
            <isbn>978-84-1545-297-3</isbn>
        </Libro>
        <Libro>
            <autor>María Jesús Ramos</autor>
            <nombre>Acceso a Datos</nombre>
            <editorial>Garceta</editorial>
            <isbn>978-84-1545-228-7</isbn>
        </Libro>
    </ListaLibro>
    <lugar>Talavera, como no</lugar>
    <nombre>Prueba de libreria JAXB</nombre>
</libreria>
```

Se trata de representar los libros de una librería. Crearemos las siguientes clases:

- La clase *Libreria*, con la lista de libros, el lugar y el nombre de la librería.
- La clase *Libro*, con los datos del autor, el nombre, la editorial y el ISBN.

En la clase *Libro*, vamos a indicar la anotación `@XmlType` pues es una clase que no es raíz y además indicamos el orden de las etiquetas con `propOrder`, es decir, cómo se desea que salgan en el documento XML. La clase tendrá la siguiente descripción:

```
package clasesjaxb;

import javax.xml.bind.annotation.XmlType;

@XmlType(propOrder = {"autor", "nombre", "editorial", "isbn"})
public class Libro {
    private String nombre;
    private String autor;
    private String editorial;
    private String isbn;
    public Libro(String nombre, String autor, String editorial,
                String isbn) {
        super();
        this.nombre = nombre;
        this.autor = autor;
        this.editorial = editorial;
        this.isbn = isbn;
    }
    public Libro() {}
    public String getNombre() { return nombre; }
    public String getAutor() { return autor; }
    public String getEditorial() { return editorial; }
}
```

```

public String getIsbn() { return isbn; }
public void setNombre(String nombre) { this.nombre = nombre; }
public void setAutor(String autor) { this.autor = autor; }
public void setEditorial(String editorial)
    { this.editorial = editorial; }
public void setIsbn(String isbn) { this.isbn = isbn; }
}

```

En la clase *Libreria*, vamos a indicar la anotación **@XmlRootElement** pues es una clase raíz. También tenemos que indicar que hay un atributo, qué es una colección, con lo que hay que añadir con las anotaciones **@XmlElementWrapper** y **@XmlElement**, en el método *get*. En estas anotaciones indicamos como se van a llamar las etiquetas dentro del documento generado. La clase tendrá la siguiente descripción:

```

package clasesjaxb;
import java.util.ArrayList;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement()
public class Libreria {
    private ArrayList<Libro> listaLibro;
    private String nombre;
    private String lugar;

    public Libreria(ArrayList<Libro> listaLibro, String nombre,
        String lugar) {
        super();
        this.listaLibro = listaLibro;
        this.nombre = nombre;
        this.lugar = lugar; }

    public Libreria() {}

    public void setNombre(String nombre) { this.nombre = nombre; }
    public void setLugar(String lugar) { this.lugar = lugar; }
    public String getNombre() { return nombre; }
    public String getLugar() { return lugar; }

    //Wrapper, envoltura alrededor la representación XML
    @XmlElementWrapper(name = "ListaLibro") //
    @XmlElement(name = "Libro")
    public ArrayList<Libro> getListaLibro() {
        return listaLibro; }

    public void setListaLibro(ArrayList<Libro> listaLibro) {
        this.listaLibro = listaLibro; }
}

```

Una vez que tenemos las clases ya definidas, lo siguiente es ver el **código Java para mapear los objetos** que definimos de esas clases.

Utilizando la anotación **@XmlRootElement**. El código Java para conseguir el fichero XML es el siguiente:

- Instanciamos el contexto, indicando la clase que será el **RootElement**, en nuestro ejemplo es la clase *Libreria*:

```
JAXBContext jaxbContext = JAXBContext.newInstance(Libreria.class);
```

- Creamos un **Marshaller**, que es la clase capaz de convertir nuestro JavaBean, en una cadena XML:

```
Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
```

- Indicamos que vamos a querer el XML con un formato amigable (saltos de línea, sangrado, etc)

```
jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
```

- Hacemos la conversión llamando al método **marshal**, pasando una instancia del JavaBean que queramos convertir a XML y un **OutputStream** donde queramos que salga el XML, por ejemplo, la salida estándar, o también podría ser un fichero o cualquier otro stream:

```
jaxbMarshaller.marshal(unaInstanciaDeUnaClase, System.out);
//Si ponemos un fichero
jaxbMarshaller.marshal(unaInstanciaDeUnaClase,
                      new File("./mifichero.xml"));
```

Ahora vamos a crear objetos de las clases y vamos a ver cómo generar el XML:

```
package clasesjaxb;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;

public class Ejemplo1_JAXB {
    private static final String MIARCHIVO_XML = "./libreria.xml";
    public static void main(String[] args)
        throws JAXBException, IOException {
        //Se crea la lista de libros
        ArrayList<Libro> libroLista = new ArrayList<Libro>();
        // Creamos dos libros y los añadimos
        Libro libro1 = new Libro("Entornos de Desarrollo",
                               "Alicia Ramos", "Garceta", "978-84-1545-297-3" );
        libroLista.add(libro1);
        Libro libro2 = new Libro("Acceso a Datos", "Maria Jesús Ramos",
                               "Garceta", "978-84-1545-228-7" );
        libroLista.add(libro2);

        // Se crea La libreria y se le asigna la lista de libros
        Libreria milibreria = new Libreria();
        milibreria.setNombre("Prueba de libreria JAXB");
        milibreria.setLugar("Talavera, como no");
```

```

    milibreria.setListaLibro(libroLista);

    // Creamos el contexto indicando la clase raíz
    JAXBContext context = JAXBContext.newInstance(Libreria.class);
    //Creamos el Marshaller, convierte el java bean en una cadena XML
    Marshaller m = context.createMarshaller();
    //Formateamos el xml para que quede bien
    m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
    // Lo visualizamos con system out
    m.marshal(milibreria, System.out);
    // Escribimos en el archivo
    m.marshal(milibreria, new File(MIARCHIVO_XML));
}

}

```

Si ahora deseamos hacer lo contrario, es decir, **leer los datos del documento XML** y convertirlos a objetos Java, utilizaremos las siguientes órdenes:

- Instanciamos el contexto, indicando la clase que será el **RootElement**, en nuestro ejemplo *Libreria*:

```
JAXBContext context = JAXBContext.newInstance(Libreria.class);
```

- Se crea **Unmarshaler** en el contexto de la clase Libreria:

```
Unmarshaller unmars = context.createUnmarshaller();
```

- Utilizamos el método **unmarshal**, para obtener datos de un **Reader** (un file):

```
UnaClase objeto = (UnaClase) unmars.unmarshal(new  
FileReader("mifichero.xml"));
```

- Recuperamos un atributo del objeto:

```
System.out.println(objeto.getUnAtributo());
```

- Recuperamos el Arraylist, si lo tiene y visualizamos:

```
ArrayList<ClaseDelArray> lista = objeto.getListadeobjetos();
for (ClaseDelArray obarray : lista) {
    System.out.println("Atributo array: " + obarray.getAtributo());
```

En nuestro ejercicio el código para visualizar el contenido del fichero XML es el siguiente:

```

// Visualizamos ahora los datos del documento XML creado
System.out.println("----- Leo el XML -----");
//Se crea Unmarshaller en el contexto de la clase Libreria
Unmarshaller unmars = context.createUnmarshaller();

//Utilizamos el método unmarshal, para obtener datos de un Reader
Libreria libreria2 =(Libreria)
    unmars.unmarshal(new FileReader(MIARCHIVO_XML));

//Recuperamos los datos y visualizamos
System.out.println("Nombre de libreria: "+ libreria2.getNombre());
System.out.println("Lugar de la libreria: " +
                    libreria2.getLugar());
System.out.println("Libros de la librería: ");

```

```

ArrayList<Libro> lista = libreria2.getListaLibro();
for (Libro libro : lista) {
    System.out.println("\tTítulo del libro: "
        + libro.getNombre()
        + ", autora: " + libro.getAutor());
}

```

### ACTIVIDAD 1.7.

Realiza cambios en las clases anteriores y añade las clases que se necesitan, para generar un documento XML que agrupe a varias librerías con varios libros. Haz el programa Java que utilice esas clases, cree dos objetos librerías, una con dos libros, y otra con tres libros y genere un documento con nombre *Librerias.xml* con esta estructura:

```

<MISLIBRERIAS>
  <Libreria>
    <nombre>xxxxxx</nombre>
    <lugar>xxxxxx</lugar>
    <MiListaLibros>
      <Libro>
        <nombre>xxxxxx</nombre>
        <autor>xxxxxx</autor>
        <editorial>xxx</editorial>
        <isbn>xxxx</isbn>
      </Libro>
      <Libro>
        .....
        .....
      </Libro>
    </MiListaLibros>
  </Libreria>

```

```

<Libreria>
  <nombre>xxxxxx</nombre>
  <lugar>xxxxxx</lugar>
  <MiListaLibros>
    <Libro>
      <nombre>xxxxxx</nombre>
      <autor>xxxxxx</autor>
      <editorial>xxxx</editorial>
      <isbn>xxxx</isbn>
    </Libro>
    <Libro>
      .....
      .....
    </Libro>
  </MiListaLibros>
</Libreria>
</MISLIBRERIAS>

```

### 1.9.2. Paso de esquemas XML (.xsd) a clases Java

El compilador de JAXB nos va a permitir generar una serie de clases Java a partir de un esquema XML. Un esquema XML describe la estructura de un documento XML. A los esquemas XML se le llama XSD (*XML Schema Definition*).

JAXB, compila el fichero XSD, creando una serie de clases para cada uno de los tipos que se haya especificado en el XSD. Esas clases serán clases POJO, y las podremos utilizar para crear, y modificar documentos XML utilizando Java.

#### Ejemplo 2:

Partimos de un XSD, en el que vamos a representar a un artículo y sus ventas. El artículo puede tener varias ventas, mínimo 1 venta. Datos del artículo (*DatosArtic*) son: *codigo*, *denominacion*, *stock* y *precio*. Datos de cada venta (*ventas*) son: *numventa*, *unidades*, *nombrecliente* y *fecha*.

El fichero XSD se llama *ventas\_articulo.xsd*, y el contenido es el siguiente:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

```

```

<xsd:element name="ventasarticulos" type="VentasType"/>

<xsd:complexType name="VentasType">
  <xsd:sequence>
    <xsd:element name="articulo" type="DatosArtic"/>
    <xsd:element name="ventas" type="ventas"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="DatosArtic">
  <xsd:sequence>
    <xsd:element name="codigo" type="xsd:string"/>
    <xsd:element name="denominacion" type="xsd:string"/>
    <xsd:element name="stock" type="xsd:integer"/>
    <xsd:element name="precio" type="xsd:decimal"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ventas">
  <xsd:sequence>
    <xsd:element name="venta" minOccurs="1" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="numventa" type="xsd:integer"/>
          <xsd:element name="unidades">
            <xsd:simpleType>
              <xsd:restriction base="xsd:positiveInteger">
                <xsd:maxExclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="nombrecliente" type="xsd:string"/>
          <xsd:element name="fecha" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>

```

## ¡¡INTERESANTE !!

Para saber más sobre XSD puedes consultar estas URLs:

[http://www.w3schools.com/xml/schema\\_howto.asp](http://www.w3schools.com/xml/schema_howto.asp)  
[http://www.w3schools.com/xml/schema\\_schema.asp](http://www.w3schools.com/xml/schema_schema.asp)

Este XSD está formado por los siguientes elementos:

- El **elemento principal** (raíz del documento) se va a llamar *ventasarticulos* y su tipo de dato lo vamos a llamar *VentasType*:

```
<xsd:element name = "ventasarticulos" type = "VentasType"/>
```

- El tipo **VentasType**, `type = "VentasType"` formado por las ventas de un artículo, se llama `ventasarticulos`. Es el tipo del elemento principal `<xsd:element name = "ventasarticulos" type = "VentasType"/>`.

Es un tipo complejo, formado por dos elementos, estos van a ser etiquetas en el documento XML, cada uno tiene su tipo, un tipo es el *artículo*, y el otro tipo son las *ventas* del artículo. Estos a su vez estarán compuestos por otras etiquetas (`<xsd:element..>`). En `name` se indica el nombre del elemento, y este nombre es el de las etiquetas que luego aparecen en el XML. En el ejercicio lo declaramos así:

```
<xsd:complexType name="VentasType">
  <xsd:sequence>
    <xsd:element name="articulo" type="DatosArtic"/>
    <xsd:element name="ventas" type="ventas"/>
  </xsd:sequence>
</xsd:complexType>
```

- El **element name artículo** lo utilizamos para representar los datos del artículo su tipo es `type = "DatosArtic"`. Este tipo está formado por las etiquetas del artículo. Es un tipo complejo y lo representamos así:

```
<xsd:complexType name="DatosArtic">
  <xsd:sequence>
    <xsd:element name="codigo" type="xsd:string"/>
    <xsd:element name="denominacion" type="xsd:string"/>
    <xsd:element name="stock" type="xsd:integer"/>
    <xsd:element name="precio" type="xsd:decimal"/>
  </xsd:sequence>
</xsd:complexType>
```

Para indicar los tipos de datos que van a contener las etiquetas utilizamos:  
`type = "xsd:string"`, para representar cadenas,  
`type = "xsd:integer"`, para los *Integer*,  
`type = "xsd:decimal"`, para los *Float*.

## ¡ ¡INTERESANTE !!

Para saber más sobre tipos XSD podemos consultar las siguientes URLs:

[http://www.w3schools.com/xml/schema\\_dtypes\\_string.asp](http://www.w3schools.com/xml/schema_dtypes_string.asp)  
[http://www.w3schools.com/xml/schema\\_dtypes\\_date.asp](http://www.w3schools.com/xml/schema_dtypes_date.asp)  
[http://www.w3schools.com/xml/schema\\_dtypes\\_numeric.asp](http://www.w3schools.com/xml/schema_dtypes_numeric.asp)  
[http://www.w3schools.com/xml/schema\\_dtypes\\_misc.asp](http://www.w3schools.com/xml/schema_dtypes_misc.asp)  
[http://www.w3schools.com/xml/schema\\_example.asp](http://www.w3schools.com/xml/schema_example.asp)

- El **element name ventas** lo declaramos para representar los datos de las ventas del artículo, su tipo es `type = "ventas"` y se llama `ventas`. Cada artículo podrá tener varias ventas (cada detalle de venta se llamará `venta`). Es un tipo complejo y lo representamos así:

```
<xsd:complexType name="ventas">
  <xsd:sequence>
    <xsd:element name="venta" minOccurs="1" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
```

```

<xsd:element name="numventa" type="xsd:integer"/>
<xsd:element name="unidades">
    <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
            <xsd:maxExclusive value="100"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
<xsd:element name="nombreciente" type="xsd:string"/>
<xsd:element name="fecha" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>

```

- Para indicar que del tipo ventas puede haber varias ocurrencias (un artículo puede tener varias ventas) utilizamos los atributos **minOccurs** y **maxOccurs**, mínimo número de filas y sin límite en el máximo. Cada detalle de venta se va a llamar *venta*. Lo escribimos así:

```
<xsd:element name="venta" minOccurs="1" maxOccurs="unbounded">
```

- También para cada elemento *unidades* se ha añadido una restricción, será un número positivo y el valor máximo va a ser 100. Lo escribimos así:

```

<xsd:element name="unidades">
    <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
            <xsd:maxExclusive value="100"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>

```

## ¡¡INTERESANTE!!

Para saber más sobre restricciones en esquemas XSD podemos consultar las siguientes URLs:

[http://www.w3schools.com/xml/el\\_restriction.asp](http://www.w3schools.com/xml/el_restriction.asp)

[http://www.w3schools.com/xml/schema\\_facets.asp](http://www.w3schools.com/xml/schema_facets.asp)

---

Una vez que tenemos el esquema XSD, un ejemplo de un documento XML, con este esquema podría ser el siguiente: observa la raíz del documento *<ventasarticulos>*, los datos del artículo *<articulo>*, las ventas del artículo *<ventas>* y el detalle de cada venta *<venta>*:

```

<?xml version="1.0" encoding="UTF-8"?>
<ventasarticulos xmlns:xsi='http://www.w3.org/2001/XMLSchema-
instance' xsi:noNamespaceSchemaLocation='ventas_articulo.xsd'>

<articulo>
    <codigo>ART-112</codigo>
    <denominacion>Pala Padel NOX</denominacion>
    <stock>20</stock>
    <precio>70</precio>

```

```

</articulo>
<ventas>
    <venta>
        <numventa>10</numventa>
        <unidades>2</unidades>
        <nombreciente>Alicia Ramos</nombreciente>
        <fecha>10-10-2015</fecha>
    </venta>
    <venta>
        <numventa>11</numventa>
        <unidades>2</unidades>
        <nombreciente>Pedro García</nombreciente>
        <fecha>15-10-2015</fecha>
    </venta>
    <venta>
        <numventa>12</numventa>
        <unidades>6</unidades>
        <nombreciente>Alberto Gil</nombreciente>
        <fecha>20-10-2015</fecha>
    </venta>
</ventas>
</ventasarticulos>

```

### 1.9.3. Creación de una aplicación JAXB con Eclipse

Lo siguiente que vamos a hacer es crear las clases a partir del XSD (*ventas\_articulo.xsd*). Y luego trabajaremos con este documento XML para hacer operaciones, como visualizar los datos del XML, añadir, modificar o borrar ventas. El fichero XML con datos de un artículo y sus ventas se llamará *ventasarticulos.xml*. Pasos:

- Creamos un proyecto en Eclipse. Y para esta prueba nos aseguramos de guardar el XSD dentro de la carpeta *src* del proyecto. Y el XML dentro de la carpeta raíz del proyecto.
- Añadimos los siguientes JAR: **jAXB-api.jar**, **jAXB-core.jar**, **jAXB-impl.jar**, **jAXB-jxc.jar** y **jAXB-xjc.jar**. Se pueden descargar de <https://jaxb.java.net/>, el fichero se llama **jAXB-ri-2.2.11.zip**, los JAR se encuentran en la carpeta *lib* de este fichero.
- Generamos los tipos, es decir, las clases, a partir del fichero XSD, *ventas\_articulo.xsd*: botón derecho sobre el fichero XSD, seleccionamos **Generate -> JAXB Classes**. En la siguiente ventana se selecciona el proyecto, y se añade el nombre del paquete donde queremos que se guarden los tipos generados, y dejamos el resto de opciones. Véase Figura 1.4.
- Una vez generados los tipos observa que las clases que se han creado se corresponden con los *types* del fichero *ventas\_articulo.xsd*. Véase Figura 1.5

Las clases creadas son *VentasType*, *DatosArtic*, y *Ventas* (el nombre de clase va en mayúscula).

```

<xsd:element name="ventasarticulos" type="VentasType"/>
<xsd:complexType name="VentasType">
    <xsd:sequence>
        <xsd:element name="articulo" type="DatosArtic"/>

```

```

<xsd:element name="ventas" type="ventas"/>
</xsd:sequence>
</xsd:complexType>

```

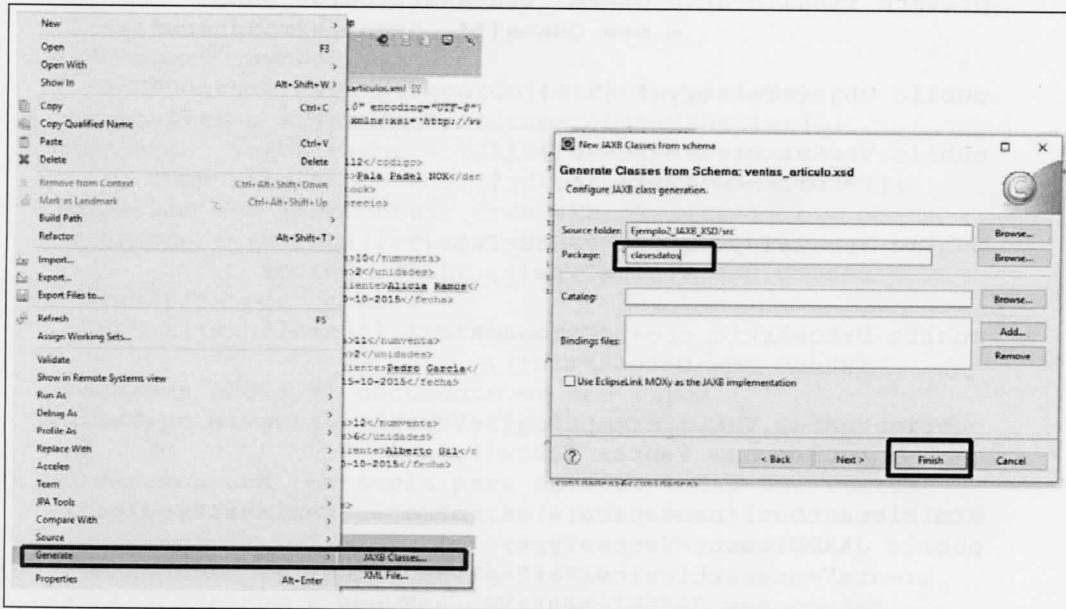


Figura 1.4. Creación de las clases JAXB.

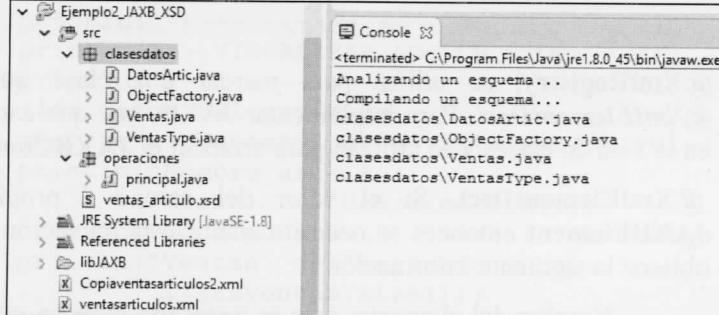


Figura 1.5. Creación de las clases JAXB.

- Se crea además la clase ***ObjectFactory***, que nos va a permitir crear un ***ObjectFactory*** que se va a utilizar para crear nuevas instancias de las clases Java del esquema XSD. En nuestro ejercicio creará instancias de las clases del paquete *clasesdatos*.

Esta clase crea un objeto ***QName***. ***QName*** representa un nombre completo tal como se define en las especificaciones XML, está formado por el nombre del namespace (espacio de nombres URI), y el nombre que hemos puesto al elemento principal en el XSD, se le llama prefijo local, en el ejercicio es *ventasarticulos*. ***QName*** es inmutable, una vez creado no puede ser cambiado. La clase creada es la siguiente:

```

package clasesdatos;

import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlElementDecl;
import javax.xml.bind.annotation.XmlRegistry;
import javax.xml.namespace.QName;

```

```

@XmlRegistry
public class ObjectFactory {

    private final static QName _Ventasarticulos_QNAME
        = new QName("", "ventasarticulos");

    public ObjectFactory() { }

    public Ventas createVentas() {
        return new Ventas();
    }

    public VentasType createVentasType() {
        return new VentasType();
    }

    public DatosArtic createDatosArtic() {
        return new DatosArtic();
    }

    public Ventas.Venta createVentasVenta() {
        return new Ventas.Venta();
    }

    @XmlElementDecl(namespace = "", name = "ventasarticulos")
    public JAXBElement<VentasType>
        createVentasarticulos(VentasType value) {
        return new JAXBElement<VentasType>
            (_Ventasarticulos_QNAME, VentasType.class, null, value);
    }
}

```

- **@XmlRegistry**, se utiliza para marcar una clase que tiene anotaciones **@XmlElementDecl**. Para implementar JAXB, hay que asegurar que se incluye en la lista de clases y se utilizan para arrancar el **JAXBContext**.
- **@XmlElementDecl**. Si el valor del campo / propiedad va a ser un **JAXBElement** entonces se necesita añadir esta anotación. Un **JAXBElement** obtiene la siguiente información :
  - Nombre del elemento, esto es necesario si se ha mapeado una estructura donde hay varios elementos del mismo tipo. En el ejercicio es *VentasType*.
  - **JAXBElement** puede ser usado para representar un elemento con *xsi:nil = "true"*.

- En nuestro programa Java para crear el contexto JAXB, podemos utilizar el nombre del paquete que contiene a todas las clases, o el nombre de clase **ObjectFactory**:

```
JAXBContext contexto = JAXBContext.newInstance("clasesdatos");
```

O también:

```
JAXBContext contexto =
JAXBContext.newInstance("clasesdatos.ObjectFactory.class");
```

Vamos ahora a crear una clase principal y con un método para visualizar el contenido del fichero XML *ventasarticulos.xml* utilizando este contexto, el método es el siguiente:

```

public static void visualizarxml() {
    System.out.println("-----");
    System.out.println("-----VISUALIZAR XML-----");
    System.out.println("-----");
    try {
        //Creamos el contexto
        JAXBContext jaxbContext =
            JAXBContext.newInstance(ObjectFactory.class);
        Unmarshaller u = jaxbContext.createUnmarshaller();
        JAXBELEMENT jaxbElement = (JAXBELEMENT) u.unmarshal(
            new FileInputStream("./ventasarticulos.xml"));
        Marshaller m = jaxbContext.createMarshaller();
        m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                      Boolean.TRUE);
        // Visualiza por consola
        m.marshal(jaxbElement, System.out);

        //Cargamos ahora el documento en los tipos
        VentasType miventa = (VentasType) jaxbElement.getValue();

        //Obtenemos una instancia para obtener todas las ventas
        Ventas vent = miventa.getVentas();

        // Guardamos las ventas en la lista
        List listaVentas = new ArrayList();
        listaVentas = vent.getVenta();

        System.out.println("-----");
        System.out.println("-----VISUALIZAR LOS OBJETOS---");
        System.out.println("-----");
        // Cargamos los datos del artículo
        DatosArtic miartic = (DatosArtic) miventa.getArticulo();
        System.out.println("Nombre art: " +
                           miartic.getDenominacion());
        System.out.println("Codigo art: " + miartic.getCodigo());
        System.out.println("Ventas del artículo , hay: " +
                           listaVentas.size());
        //Visualizamos las ventas del artículo
        for (int i = 0; i < listaVentas.size(); i++) {
            Ventas.Venta ve = (Venta) listaVentas.get(i);
            System.out.println("Número de venta: " +
                               ve.getNumventa() + ". Nombre cliente: " +
                               ve.getNombrecliente() + ", unidades: " +
                               ve.getUnidades() + ", fecha: " + ve.getFecha());
        }
    } catch (JAXBException je) {
        System.out.println(je.getCause());
    } catch (IOException ioe) {
        System.out.println(ioe.getMessage());
    }
}

```

El siguiente método recibe datos de una venta y lo añade al documento XML. Se comprobará antes de insertar que el número de venta no exista:

```

private static void insertarventa
    (int numevento, String nomcli, int uni, String fecha) {
System.out.println("----- ");
System.out.println("-----AÑADIR VENTA----- ");
System.out.println("----- ");
try {
    JAXBContext jaxbContext =
        JAXBContext.newInstance(ObjectFactory.class);
    Unmarshaller u = jaxbContext.createUnmarshaller();
    JAXBElement jaxbElement = (JAXBElement)
    u.unmarshal(new FileInputStream("./ventasarticulos.xml"));

    VentasType miventa = (VentasType) jaxbElement.getValue();

    Ventas vent = miventa.getVentas();

    List listaVentas = new ArrayList();
    listaVentas = vent.getVenta();

    // comprobar si existe el número de venta,
    // recorriendo el arraylist
    int existe = 0;
    for (int i = 0; i < listaVentas.size(); i++) {
        Ventas.Venta ve = (Venta) listaVentas.get(i);
        if (ve.getNumventa().intValue() == numevento) {
            existe = 1; break;
        }
    }
    if (existe == 0) {
        // Crear el objeto Ventas.Venta
        Ventas.Venta ve = new Ventas.Venta();
        ve.setNombrecliente(nomcli);
        ve.setFecha(fecha); ve.setUnidades(uni);
        BigInteger nume = BigInteger.valueOf(numevento);
        ve.setNumventa(nume);
        // Se añade la venta a la lista
        listaVentas.add(ve);
        //Se crea el Marshaller, volcar la lista al fichero XML
        Marshaller m = jaxbContext.createMarshaller();
        m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                      Boolean.TRUE);
        m.marshal(jaxbElement, new
                  FileOutputStream("./ventasarticulos.xml"));
        System.out.println("Venta añadida: " + numevento);
    } else
        System.out.println("En número de venta ya existe: " +
                           numevento);
} catch (JAXBException je) {
    System.out.println(je.getCause());
} catch (IOException ioe) {
    System.out.println(ioe.getMessage());}
}

```

---

**ACTIVIDAD 1.8**

Añade al proyecto anterior 3 métodos:

Método que reciba un número de venta y la borre, si existe, del documento XML. El método devolverá *true* si se ha borrado la venta y *false* si no se ha borrado o ha ocurrido algún error.

Método para modificar el stock del artículo. El método recibe una cantidad numérica y se debe sumar al stock del artículo. El método devuelve *true* si la operación se realiza correctamente y *false* si ocurre algún error.

Método para cambiar los datos de una venta, este método recibe el número de venta a modificar, las unidades y la fecha. Se desean modificar esos datos del número de venta. El método devuelve *true* si la operación se realiza correctamente y *false* si ocurre algún error.

Método para cambiar los datos de una venta, este método recibe el número de venta a modificar, las unidades y la fecha. Se desean modificar esos datos del número de venta. El método devuelve *true* si la operación se realiza correctamente y *false* si ocurre algún error.

---

**Ejemplo 3:**

En este ejemplo partimos del mismo XSD *ventas\_articulo.xsd*, y lo que vamos a hacer es crear un documento nuevo llamado *nuevo\_ventasarticulo.xml*, con datos de un artículo y dos ventas, que asignaremos manualmente. Los datos serán los siguientes:

Datos para el artículo:

Codigo	Denominacion	Stock	Precio
Arti 1	Bicicleta Plegable	10	200

Datos para las ventas:

Num venta	Unidades	Nombre de cliente	Fecha
1	2	Alicia Ramos	10-02-2016
2	1	Dori Martín	21-02-2016

El código Java es el siguiente:

```
public static void crearnuevoventasxml() {
    // Creo el objeto DatosArtic y asigno valores
    DatosArtic articulo = new DatosArtic();
    articulo.setCodigo("Arti 1");
    articulo.setDenominacion("Bicicleta Plegable");
    BigInteger stv = BigInteger.valueOf(10);
    BigDecimal pvv = BigDecimal.valueOf(200);
    articulo.setPrecio(pvv);
    articulo.setStock(stv);
```

```

// Creamos el objeto Ventas
Ventas ventas = new Ventas();

// Creo la primera venta y la añado a ventas
Ventas.Venta ven = new Ventas.Venta();
ven.setNombrecliente("Alicia Ramos");
ven.setNumventa(BigInteger.valueOf(1));
ven.setUnidades(2);
ven.setFecha("10-02-2016");
ventas.getVenta().add(ven);

// Creo la segunda venta y la añado a ventas
ven = new Ventas.Venta();
ven.setNombrecliente("Dori Martín");
ven.setNumventa(BigInteger.valueOf(2));
ven.setUnidades(1);
ven.setFecha("21-02-2016");
ventas.getVenta().add(ven);

// Creo un VentasType y asigno los datos del artículo y sus ventas
VentasType miventaarticulo = new VentasType();
miventaarticulo.setArticulo(articulo);
miventaarticulo.setVentas(ventas);

// Creo el ObjectFactory
ObjectFactory miarticulo = new ObjectFactory();

// Creo El JAXBELEMENT lo obtenemos del ObjectFactory y del VentasType
JAXBElement<VentasType> element =
    miarticulo.createVentasarticulos(miventaarticulo);
// Creo el contexto y el Marshaller
JAXBContext jaxbContext;

try {
    jaxbContext = JAXBContext.newInstance(ObjectFactory.class);
    Marshaller m = jaxbContext.createMarshaller();
    m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
        Boolean.TRUE);
    String fiche = "./nuevo_ventasarticulos.xml";

    m.marshal(element, new FileOutputStream(fiche));
    System.out.println("Venta creada. ");
    // Visualizamos por consola
    m.marshal(element, System.out);

} catch (JAXBException e) {e.printStackTrace();}

} catch (FileNotFoundException e){      e.printStackTrace();}

}

```

## COMPRUEBA TU APRENDIZAJE

1. Realiza un programa Java que al ejecutarlo desde la línea de comandos reciba un nombre de directorio. El programa deberá eliminar el directorio y los ficheros contenidos en él.
2. Realiza un programa Java que cree un fichero binario para guardar datos de departamentos, dale el nombre *Departamentos.dat*. Introduce varios departamentos. Los datos por cada departamento son: *Número de departamento: entero, Nombre: String y Localidad: String*.
3. Realiza un programa Java que te permita modificar los datos de un departamento. El programa recibe desde la línea de comandos el número de departamento a modificar, el nuevo nombre de departamento y la nueva localidad. Si el departamento no existe visualiza un mensaje indicándolo. Visualiza también los datos antiguos del departamento y los nuevos datos.
4. Realiza un programa Java que te permita eliminar un departamento. El programa recibe desde la línea de comandos el número de departamento a eliminar. Si el departamento no existe visualiza un mensaje indicándolo. Visualiza también el número total de departamentos que existen en el fichero.
5. Realiza un programa que copie dos ficheros. El nombre de los dos ficheros, origen y destino, se introducen desde la línea de comandos al ejecutar el programa.
6. Cuál de las siguientes afirmaciones sobre **SAX** y **DOM** es correcta:
  - a) Los procesadores **SAX** y **DOM** son independientes del lenguaje de programación y existen versiones particulares para Java, VisualBasic, C, etc.
  - b) Mediante **SAX** se carga todo el fichero XML en memoria y se lee de forma secuencial produciendo una secuencia de eventos.
  - c) Mediante **DOM** se almacena toda la estructura del documento XML en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales.
  - d) **SAX** es más complejo de programar que **DOM**.
  - e) El tipo de procesamiento de **SAX** necesita más recursos de memoria y tiempo, sobre todo si los ficheros XML a procesar son bastante grandes y complejos.
  - f) El tipo de procesamiento de **DOM** impide tener una visión global del documento por el que navegar.
7. A partir de los datos del fichero *Departamentos.dat* creado anteriormente crea un fichero XML usando **DOM**.
8. A partir de los datos del fichero *Departamentos.dat* creado anteriormente crea un fichero XML usando la librería **XStream**.
9. Crea una plantilla XSL para dar una presentación al fichero XML generado por el ejercicio anterior y realiza un programa Java para transformarlo en HTML.
10. Señala la respuesta correcta sobre las excepciones:
  - a) Una sentencia **try** no puede estar dentro de un bloque de otra sentencia **try**.
  - b) Un bloque **try** va seguido por un único bloque **catch**.

- g) Cada bloque catch especifica el tipo de excepción que puede atrapar y contiene un manejador de excepciones.
- h) El bloque **finally** se ejecuta si no ocurre la excepción.
11. Crea un esquema XSD, con nombre *centrosprofes.xsd* para representar los datos de un centro educativo y sus profesores. Y luego crea una aplicación JAXB para crear un fichero XML llamado *micentro.xml*, con los datos de un centro con tres profesores, y uno de ellos debe ser el director del centro.

La estructura del XML debe quedar de la siguiente manera:

```
<?xml version="1.0" encoding="UTF-8"?>
<datoscentro xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
               xsi:noNamespaceSchemaLocation = 'centrosprofes.xsd'>
    <centro>
        <codigocentro></codigo>
        <nombrecentro></nombre>
        <direccion></direccion>
        <director>
            <codigoprofesor></codigoprofesor>
            <nombrefprofe></nombrefprofe>
        </director>
    </centro>
    <profesores>
        <profe>
            <codigoprofesor></codigoprofesor>
            <nombrefprofe></nombrefprofe>
        </profe>
        .....
    </profesores>
</datoscentro>
```