

UNIDAD 1

MANEJO DE FICHEROS

OBJETIVOS

El alumno al término de esta unidad debe ser capaz de:

- 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.

CONTENIDOS

- 1.1 INTRODUCCIÓN
- 1.2 CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS
- 1.3 FLUJOS O STREAMS. TIPOS
 - 1.3.1 Flujos de bytes (Byte streams)
 - 1.3.2 Flujos de caracteres (Character streams)
- 1.4 FORMAS DE ACCESO A UN FICHERO
- 1.5 OPERACIONES SOBRE FICHEROS
 - 1.5.1 Operaciones sobre ficheros secuenciales
 - 1.5.2 Operaciones sobre ficheros aleatorios
- 1.6 CLASES PARA GESTIÓN DE FLUJOS DE DATOS DESDE/HACIA FICHEROS
 - 1.6.1 Ficheros de texto
 - 1.6.2 Ficheros binarios
 - 1.6.3 Ficheros de acceso aleatorio
- 1.7 TRABAJO CON FICHEROS XML
 - 1.7.1 Acceso a ficheros XML con DOM
 - 1.7.2 Acceso a ficheros XML con SAX
 - 1.7.3 Serialización de objetos a XML
 - 1.7.4 Conversión de ficheros XML a otro formato
- 1.8 EXCEPCIONES: DETECCIÓN Y TRATAMIENTO
 - 1.8.1 Capturar excepciones.
 - 1.8.2 Especificar excepciones
- 1.9 EJERCICIOS

1.1 INTRODUCCIÓN

Un **fichero** o **archivo** es un conjunto de bits almacenado 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 archivo.

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 diseñe.

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

1.2 CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE 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:

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

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

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

El siguiente ejemplo muestra la lista de ficheros en el directorio actual. Se utiliza el método `list()` que devuelve un array de Strings 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 el parámetro ":":

```
import java.io.*;
public class VerDir {
    public static void main(String[] args) {
        System.out.println("Ficheros en el directorio actual:");
        File f = new File(".");
        String[] archivos = f.list();
        for (int i = 0; i < archivos.length; i++) {
            System.out.println(archivos[i]);
        }
    }
}
```

La siguiente declaración 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: Realiza un programa Java que muestre los ficheros de un directorio. El nombre del directorio se pasará al programa desde la línea de comandos al ejecutarlo.

Algunos métodos importantes del objeto `File` son:

- `getName()`: devuelve el nombre del fichero o directorio.
- `getPath()`: devuelve el camino relativo.
- `getAbsolutePath()`: devuelve el camino absoluto del fichero/directorio.
- `canRead()`: devuelve true si el fichero se puede leer.
- `canWrite()`: devuelve true si el fichero se puede escribir.
- `length()`: nos devuelve el tamaño del fichero en bytes.
- `createNewFile()`: crea un nuevo fichero, vacío, asociado a `File` si y solo si no existe un fichero con dicho nombre.
- `delete()`: borra el fichero o directorio asociado al `File`.
- `exists()`: devuelve true si el fichero/directorio existe.
- `getParent()`: devuelve el nombre del directorio padre, o `null` si no existe.
- `isDirectory()`: devuelve true si el objeto `File` corresponde a un directorio.

- *isFile()*: devuelve true si el objeto **File** corresponde a un fichero normal.
- *mkdir()*: crea un directorio con el nombre indicado en la creación del objeto **File**.
- *renameTo(File nuevonombre)*: renombra el fichero.

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("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.getAbsolutePath());
            System.out.println("Se puede escribir : "+f.canRead());
            System.out.println("Se puede leer : "+f.canWrite());
            System.out.println("Tamaño : "+f.length());
            System.out.println("Es un directorio : "+f.isDirectory());
            System.out.println("Es un fichero : "+f.isFile());
        }
    }
}
```

Visualiza la siguiente información del fichero:

```
INFORMACIÓN SOBRE EL FICHERO:
Nombre del fichero : VerInf.java
Ruta : VerInf.java
Ruta absoluta : D:\unil\VerInf.java
Se puede escribir : true
Se puede leer : true
Tamaño : 794
Es un directorio : false
Es un fichero : true
```

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 a partir del actual
        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 eliminarlos. Para borrar `FICHERO2` 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 |
|-------------------------|---|
| ByteArrayInputStream | Permite usar un espacio de almacenamiento intermedio de memoria |
| StringBufferInputStream | Convierte un String en un InputStream |
| FileInputStream | Para leer información de un fichero |
| PipedInputStream | Implementa el concepto de "tubería" |
| FilterInputStream | Proporciona funcionalidad útil a otras clases InputStream |
| SequenceInputStream | Convierte dos o más objetos InputStream en un InputStream único |

Los tipos de **OutputStream** incluyen las clases que deciden donde 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 | Para enviar información a un fichero |
| PipedOutputStream | Cualquier información que se desee escribir aquí acaba automáticamente como entrada del PipedInputStream asociado. Implementa el concepto de "tubería" |
| FilterOutputStream | Proporciona funcionalidad útil a otras clases OutputStream |

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

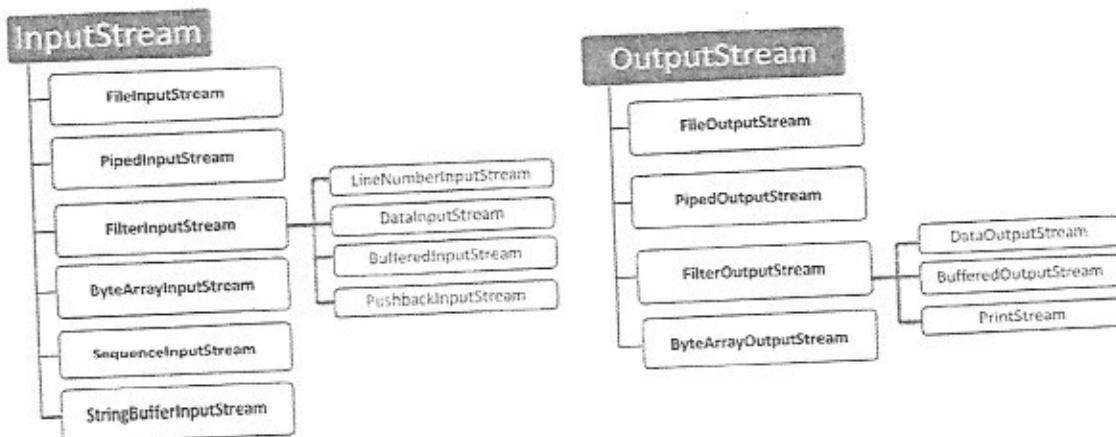


Figura 1.1. Jerarquía de clases para lectura y escritura de bytes.

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** (lee bytes y los convierte en caracteres) y **OutputStreamReader** que convierte un **OutputStream** en un **Writer**.

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 OutputStreamWriter |
| 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.

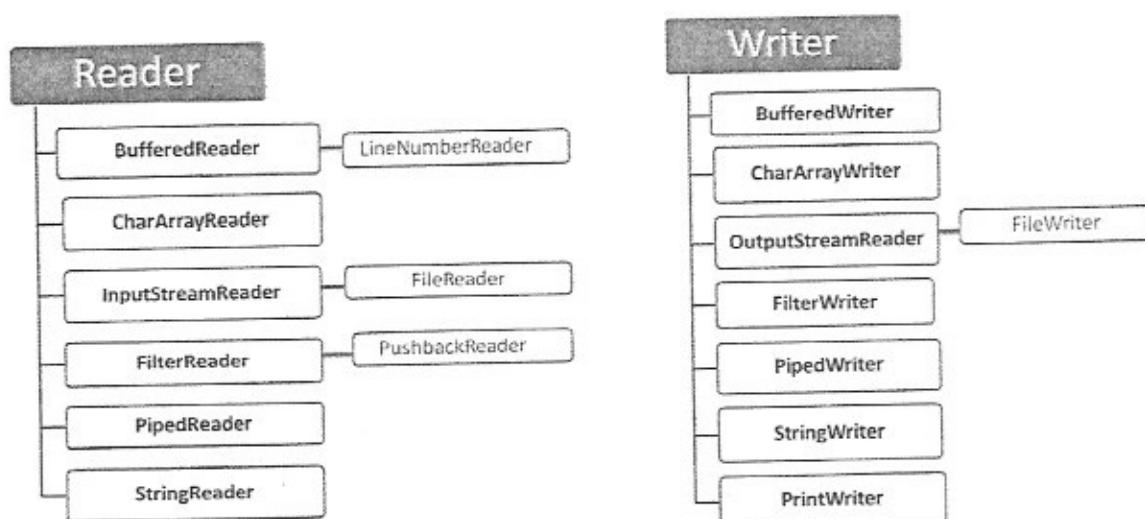


Figura 1.2. Jerarquía de clases lectura y escritura de flujos de caracteres.

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 **InputStream** y **OutputStream**; 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 como 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, es decir, 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 posi-

ción 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*,...).

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 existe 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:

- *int read()*: lee un carácter y lo devuelve.
- *int read(char[] buf)*: lee hasta *buf.length* caracteres de datos de una matriz de caracteres (*buf*). Los caracteres leídos del fichero se van almacenando en *buf*.
- *int read(char[] buf, int desplazamiento, int n)*: lee hasta *n* caracteres de datos de la matriz *buf* comenzando por *buf[desplazamiento]* y devuelve el número leído de caracteres.

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");
    FileReader fic = new FileReader(fichero); //crear el flujo de entrada
    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.

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 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 desde la línea de comandos.

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

- `void write(int c)`: escribe un carácter.
- `void write(char[] buf)`: escribe un array de caracteres.
- `void write(char[] buf, int desplazamiento, int n)`: escribe n caracteres de datos en la matriz `buf` y comenzando por `buf[desplazamiento]`.
- `void write(String str)`: escribe una cadena de caracteres.
- `append(char c)`: 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`:

```

import java.io.*;
public class EscribirFichTexto {
    public static void main(String[] args) throws IOException {
        File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichTexto.txt");
        FileWriter fic = new FileWriter(fichero); //crear el flujo de salida
        String cadena = "Esto es una prueba con FileWriter";
        char[] cad = cadena.toCharArray(); //convierte un String en array de caract
    }
}

```

```

for(int i=0; i<cad.length; i++)
    fic.write(cad[i]); //se va escribiendo un carácter

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

```

En vez de escribir los caracteres uno a uno, también podemos escribir todo el array: `fic.write(cad)`;

El siguiente ejemplo escribe cadenas de caracteres que se obtienen de un array de String, las cadenas se irán grabando 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) //lee una linea del fichero
                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 linea
                fichero.newLine(); //escribe un salto de 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 **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 imprimen en un fichero, el segundo método salta 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:

```

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:

- *int read():* lee un byte y lo devuelve.
- *int read(byte[] b):* lee hasta *b.length* bytes de datos de una matriz de bytes.
- *int read(byte[] b, int desplazamiento , int n):* lee hasta *n* bytes de la matriz *b* comenzando por *b[desplazamiento]* y devuelve el número leído de bytes.

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

- *void write(int b):* escribe un byte.
- *void write(byte[] b):* escribe *b.length* bytes.
- *void write(byte[] b, int desplazamiento , int n):* escribe *n* bytes a partir de la matriz de bytes de entrada y comenzando por *b[desplazamiento]*.

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");
        //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 además de los métodos **read()** y **write()** vistos anteriormente proporcionan métodos para la lectura y escritura de tipos primitivos de un modo independiente de la máquina. 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); |

(UTE, *Unicode Transmission Format*, se utiliza para transmitir los caracteres de un String de 16 bits codificada en 8 bits.)

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

```
File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichData.dat");
FileInputStream filein = new FileInputStream(fichero);
DataInputStream dataIS = new DataInputStream(filein);
```

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

```
File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichData.dat");
FileOutputStream fileout = new FileOutputStream(fichero);
DataOutputStream dataOS = new DataOutputStream(fileout);
```

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 (*dataOS.writeUTF(nombres[i]);*) y la edad (*dataOS.writeInt(edades[i]);*):

```
import java.io.*;
public class EscribirFichData {
    public static void main(String[] args) throws IOException {
        File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichData.dat");
        FileOutputStream fileout = new FileOutputStream(fichero);
```

MANEJO DE FICHEROS

```

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]); //inserta nombre
    dataOS.writeInt(edades[i]); //inserta 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 insertaron, 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("C:\\EJERCICIOS\\UNI1\\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:

```

D:\uni1>javac LeerFichData.java
D:\uni1>java LeerFichData
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

```

OBJETOS SERIALIZABLES

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 engorroso 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:

- void *readObject(java.io.ObjectInputStream stream)* throws IOException, ClassNotFoundException: para leer un objeto.
- void *writeObject(ObjectOutputStream stream)* throws IOException: para escribir un objeto.

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 nom){nombre=nom;}
    public void setEdad(int ed){edad=ed;}
    public String getNombre(){return nombre;} //devuelve nombre
    public int getEdad(){return 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**:

MANEJO DE FICHEROS

```
File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichPersona.dat");
FileOutputStream fileout = new FileOutputStream(fichero);
ObjectOutputStream dataOS = new ObjectOutputStream(fileout);
```

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("C:\\EJERCICIOS\\UNI1\\FichPersona.dat");
        FileOutputStream fileout =
            new FileOutputStream(fichero); //crea el flujo de salida
        //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]); //creo la persona
            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("C:\\EJERCICIOS\\UNI1\\FichPersona.dat");
FileInputStream filein = new FileInputStream(fichero);
ObjectInputStream dataIS = new ObjectInputStream(filein);
```

El método `readObject()` lee los objetos del flujo de entrada, puede lanzar la excepción `ClassNotFoundException`:

```
persona= (Persona) dataIS.readObject();
```

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("C:\\EJERCICIOS\\UNI1\\FichPersona.dat");
        FileInputStream filein = new FileInputStream(fichero); //crea el flujo de entrada
        //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.println("Nombre: " + persona.getNombre() +
                    ", edad: " + persona.getEdad());
            }
        }catch (EOFException eo) {}

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

```

- **ObjectOutputStream** puede darnos algunos problemas, por ejemplo, si escribimos datos en el fichero y lo cerramos. Después volvemos a abrirlo para añadir datos (*FileOutputStream(fichero,true)*), entonces se escribe una nueva cabecera al final de los objetos introducidos anteriormente y después se van añadiendo el resto de datos. Esto origina el problema que al leer el fichero se produzca la excepción *StreamCorruptedException*.

1.6.3 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 posibilidades para crear el fichero de acceso aleatorio:

- Escribiendo el nombre del fichero: *fichero = new RandomAccessFile(String nombre, String modoAcceso);*
- Con un objeto **File**: *fichero = new RandomAccessFile(File fich, String modoAcceso);*

El argumento *modoAcceso* puede ser: “r” para solo lectura o “rw” para lectura y escritura. Una vez abierto el fichero pueden usarse los métodos *read()* y *write()* de las clases **DataInputStream** y **DataOutputStream** (vistos anteriormente). La clase **RandomAccessFile** maneja un puntero que

indica la posición actual en el fichero. Cuando en el fichero se crea el *puntero* 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:

- *long getFilePointer()*: devuelve la posición actual del *puntero* del fichero.
- *void seek(long posicion)*: coloca el *puntero* del fichero en una posición determinada desde el comienzo del mismo.
- *long length()*: devuelve el tamaño del fichero en bytes. La posición *length()* marca el final del fichero.
- *int skipBytes(int desplazamiento)*: desplaza el *puntero* desde la posición actual el número de bytes indicados en desplazamiento.

El ejemplo que se muestra a continuación inserta datos de empleados en un fichero aleatorio. Los datos a insertar: el apellido, departamento y salario, se obtienen de varios arrays que se llenan en el programa, los datos se van introduciendo de forma secuencial (no se usará el método *seek()*). Por cada empleado también se insertará un identificador 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, cada carácter Unicode ocupa 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("C:\\EJERCICIOS\\UNI1\\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;//numero 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("C:\\EJERCICIOS\\UNIL\\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
            for (int i = 0; i < apellido.length; i++) {
                aux = file.readChar(); //recorro uno a uno los caracteres del apellido
                apellido[i] = aux; //los voy guardando en el array
            }
            String apellidoS= new String(apellido); //convierbo a String el array
            dep=file.readInt(); //obtengo dep
            salario=file.readDouble(); //obtengo salario

            System.out.println("ID: " + id + ", Apellido: " + apellidoS +
                               ", Departamento: "+dep + ", Salario: " + salario);
            posicion= posicion + 36; // me posiciono para el sig empleado
            //Cada empleado ocupa 36 bytes (4+20+4+8)
            //Si he recorrido todos los bytes salgo del for
            if (file.getFilePointer()==file.length())break;
            //fin bucle for
        }
    }
}

```

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 (36):

```

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

```

Actividad 3: Crea un programa Java que reciba un identificador de empleado desde la línea de comandos y visualice sus datos. Si el empleado no existe debe visualizar mensaje indicándolo.

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; //bufer 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

```

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:

```

int registro = 4; //id a modificar
long posicion = (registro - 1) * 36; //(4+20+4+8)
posicion+=4+20; //sumo el tamaño de ID+apellido
file.seek(posicion); //nos posicionamos
file.writeInt(40); //modif departamento
file.writeDouble(4000.87); //modif salario

```

Actividad 4: Crea un programa Java que reciba desde la línea de comandos un identificador de empleado y un importe. Se debe sumar al salario del empleado el importe tecleado. El programa debe visualizar el apellido, el salario antiguo y el nuevo. Si el identificador no existe se visualizará un mensaje indicándolo.

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 así como 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>

```

EXCEPCIONES DE FICHEROS

```

< 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/fin del documento, comienzo/fin de una etiqueta, etcétera) 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-cath** 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:

```
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:

```
Element raiz = document.createElement("empleado"); // creamos el nodo empleado
document.getDocumentElement().appendChild(raiz); // lo pegamos a la raiz del doc
```

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

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

Como se puede ver la función recibe el nombre del nodo hijo (*id*, *apellido*, *dep* o *salario*) y sus 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 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>
```

La función es la 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")); //fich 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 argv[]) throws IOException{
        File fichero = new File("C:\\EJERCICIOS\\UNI1\\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"); // asignamos la version de nuestro XML

            for(;;) {
                file.seek(posicion); //nos posicionamos
                id=file.readInt(); // obtengo id de empleado
                for (int i = 0; i < apellido.length; i++) {
                    aux = file.readChar(); //recorro uno a uno los caracteres del apellido
                    apellido[i] = aux; //los voy guardando en el array
                }
                String apellidoS= new String(apellido); //convierito a String el array
                dep=file.readInt(); //obtengo dep
                salario=file.readDouble(); //obtengo salario

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

```

document.getDocumentElement().appendChild(raiz);
CrearElemento("id", Integer.toString(id), raiz, document); //añadir ID
CrearElemento("apellido", apellidoS.trim(), raiz, document); //Apellido
CrearElemento("dep", Integer.toString(dep), raiz, document); //añadir DEP
CrearElemento("salario", Double.toString(salario), raiz, document); //SAL
}
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
}

Inserción de los datos del empleado
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 de la clase

```

Para leer un documento XML, creamos una instancia de *DocumentBuilderFactory* para construir *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.println("Elemento raíz: " +
        document.getDocumentElement().getnodeName());
    //crea una lista con todos los nodos empleado
    NodeList empleados = document.getElementsByTagName("empleado");
    //recorrer la lista
    for (int i = 0; i < empleados.getLength(); i++) {
        Node emple = empleados.item(i); //obtener un nodo
        if (emple.getNodeType() == Node.ELEMENT_NODE) {//tipo de nodo
            Element elemento = (Element) emple; //obtener los elementos del nodo
            System.out.println("ID: " + getNode("id", elemento));
            System.out.println("Apellido: " + getNode("apellido", elemento));
            System.out.println("Departamento: " + getNode("dep", elemento));
            System.out.println("Salario: " + getNode("salario", elemento));
        }
    }
} catch (Exception e) {e.printStackTrace();}
}//fin de main

//obtener la información de un nodo
private static String getNode(String etiqueta, Element elem)
{
    NodeList nodo= elem.getElementsByTagName(etiqueta).item(0).getChildNodes();
    Node valornodo = (Node) nodo.item(0);
    return valornodo.getNodeValue(); //devuelve el valor del nodo
}
}//fin de la clase

```

API DOM: <http://docs.oracle.com/javase/1.4.2/docs/api/org/w3c/dom/package-tree.html>

Actividad 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()*).

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);
```

```
que           InputSource fileXML = new InputSource("alumnos.xml");
res-         procesadorXML.parse(fileXML);
eti-         }
t(),       //fin PruebaSaxl

se         class GestionContenido extends DefaultHandler {
os          public GestionContenido() {
n-            super();
n-          }

on          public void startDocument() {
ie            System.out.println("Comienzo del Documento XML");
}

e          public void endDocument() {
            System.out.println("Final del Documento XML");
}

public void startElement(String uri, String nombre,
                         String nombreC, Attributes atts) {
            System.out.println("\tPrincipio Elemento: " + nombre);
}

public void endElement(String uri, String nombre, String nombreC) {
            System.out.println("\tFin Elemento: " + nombre);
}

public void characters(char[] ch, int inicio, int longitud)
                     throws SAXException {
            String car=new String(ch, inicio, longitud);
            car = car.replaceAll("[\t\n]","");
            System.out.println ("\tCaracteres: " + 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: Juan
    Fin Elemento: nombre
    Caracteres:
    Principio Elemento: edad
    Caracteres: 19
    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

```

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

Actividad 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

- ① viceversa; utilizaremos para ello la librería **XStream**. Para poder utilizarla hemos de descargar los JAR desde el sitio web: <http://xstream.codehaus.org/download.html>, para el ejemplo se ha descargado el fichero *Binary distribution (xstream-distribution-1.4.2-bin.zip)* que hemos de descomprimir
- ② y buscar el JAR *xstream-1.4.2.jar* que está en la carpeta *lib* que es el que usaremos para el ejemplo
- ③ También necesitamos el fichero *kxml2-2.3.0.jar* que se puede descargar desde el apartado *Optional Dependencies*. Una vez que tenemos los dos ficheros, los definimos en el CLASSPATH, por ejemplo supongamos que tenemos los JAR en la carpeta *D:\unil\xstream*:

```
SET CLASSPATH=.;D:\unil\xstream\kxml2-2.3.0.jar;D:\unil\xstream\xstream-1.4.2.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);
    }
}

```

Anaplixi

```
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); //flujo de entrada
        //conecta el flujo de bytes al flujo de datos
        ObjectInputStream dataIS = new ObjectInputStream(filein);
        System.out.println("Comienza el proceso de creación del fichero a XML ...");

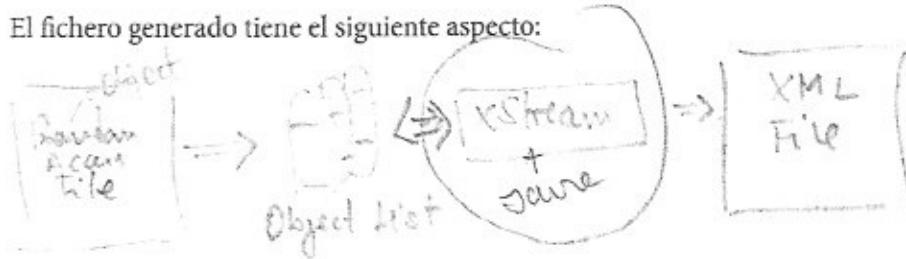
        //Creamos un objeto Lista de Personas
        ListaPersonas listaper = new ListaPersonas();

        try {
            while (true) { //lectura del fichero
                Persona persona= (Persona) dataIS.readObject(); //leer una Persona
                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:



| | |
|---|---|
| <pre> <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> </pre> | <pre> <edad>16</edad> </DatosPersona> <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> </pre> |
|---|---|

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 pueden cambiar usando el método *alias()*. 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);
```

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

```
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(objeto, OutputStream)*:

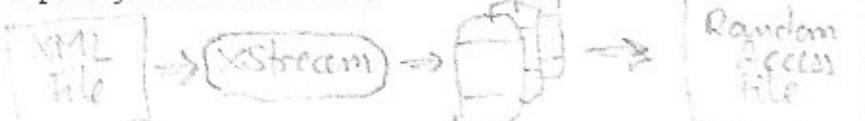
```
xstream.toXML(listaper, new FileOutputStream("Personas.xml"));
```

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("Número de Personas: " +
            listadoTodas.getListaPersonas().size());

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

        Iterator iterador = listaPersonas.listIterator(); //iterador los elementos
        while( iterador.hasNext() ) {
            Persona p = (Persona) iterador.next(); //Obtengo el elemento contenido
            System.out.println("Nombre: " + p.getNombre() +
                ", edad: " + p.getEdad());

        }
        System.out.println("Fin de listado .....");
    }
}
```

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, utilizamos el método *fromXML(InputStream)*:

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

API XStream: <http://xstream.codehaus.org/javadoc/com/thoughtworks/xstream/XStream.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

| ALUMNOS.XML | ALUMNOSPLANTILLA.XSL |
|--|--|
| <pre><?xml version="1.0"?> <listadealumnos> <alumno> <nOMBRE>Juan</nOMBRE> <edad>19</edad> </alumno> <alumno> <nOMBRE>Maria</nOMBRE> <edad>20</edad> </alumno> </listadealumnos></pre> | <pre><?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></pre> |

Para realizar la transformación se necesita obtener un objeto *Transformer* que se obtiene creando una instancia de *TransformerFactory* y aplicando el método *newTransformer* 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()*, 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 argv[]) throws IOException{
    String hojaEstilo = "alumnosPlantilla.xsl";
    String datosAlumnos = "alumnos.xml";
    File pagHTML = new File("mipagina.html");
    FileOutputStream os = new FileOutputStream(pagHTML); //crear fichero HTML
```

```

        source estilos = new StreamSource(hojaEstilo); //fuente XSL
        source datos = new StreamSource(datosAlumnos); //fuente XML
        result result = new StreamResult(os);           //resultado de la transformación

    }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
    }main
    } la clase
}

```

El fichero HTML generado se muestra en la Figura 1.3.



Figura 1.3. Fichero HTML generado.

API de Java: <http://docs.oracle.com/javase/6/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, se captura 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.println("Resultado:" +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;
```