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 permanece en el dispositivo aun cuando apagamos 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 tres letras (PDF, DOC, GIF,…) y permite saber el tipo de fichero.

Un fichero está formado por un conjunto de registros o líneas. Un registro está formado, a su vez, 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 las líneas impresas en una hoja de papel.

Por tanto, cada dato almacenado en un fichero se denomina registro. Un **registro** está formado, a su vez, por uno o varios datos que pueden ser de diferentes tipos. Cada dato que contiene el registro se le conoce con el nombre de **campo**.

Clasificación de ficheros según el tipo de contenido:

- **Ficheros de caracteres (o de texto):** son aquellos creados exclusivamente con caracteres (código ASCII), por lo que pueden ser creados y visualizados por cualquier editor de texto.
- **Ficheros binarios (o de bytes):** son aquellos que no contienen caracteres reconocibles, sino que los bytes que contienen representan otra información como imágenes, música o vídeo. Estos ficheros solo pueden ser abiertos por aplicaciones concretas que conozcan cómo están organizados los bytes dentro del fichero.

Clasificación de ficheros según el modo de acceso:

- **Ficheros secuenciales:** la información se almacena como una secuencia de bytes o caracteres, de manera que para acceder al byte i-ésimo, es necesario pasar antes por todos los anteriores.
- **Ficheros aleatorios:** se puede acceder directamente a una posición concreta del fichero, sin necesidad de recorrer los bytes anteriores.

2. OPERACIONES DE GESTIÓN DE 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 usar para acceder a él. Este proceso solo se hace una vez.
- **Apertura del fichero.** Para que un programa pueda operar con un fichero, primero debe realizar su apertura. Para ello, el programa utilizará algún método para identificar el fichero, como usar un descriptor del fichero.
- **Cierre del fichero.** El fichero se debe cerrar cuando el programa no lo vaya a utilizar. Suele ser la última instrucción del programa.
- **Lectura de datos del fichero.** Este proceso consiste en transferir información del fichero a la memoria, a través de variables del programa.
- **Escritura de datos en el fichero**. Este proceso consiste en transferir información de la memoria, por medio de las variables del programa, al fichero.

Una vez abierto, las operaciones típicas que se realizan sobre un fichero son:

- **Altas**: consiste en añadir un nuevo registro al fichero.
- **Bajas**: consiste en eliminar del fichero un registro ya existente.
- **Modificaciones**: consiste en cambiar parte del contenido de un registro. Antes de la modificación, será necesario localizar el registro a modificar dentro del fichero.
- **Consultas**: consiste en buscar en el fichero un registro determinado.

2.1. CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS

El paquete **java.io** contiene las clases para manejar la entada/salida en Java y lo necesitaremos importar dicho paquete para trabajar con ficheros. En Java, la clase **File** proporciona un conjunto de utilidades relacionadas con ficheros que proporcionan información acerca de los mismos. 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 ésta no existe.

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

Constructor	Explicación
`File(String directorioYFichero)`	Recibe en la instanciación del objeto el camino completo donde está el fichero junto con el nombre. Por defecto, si no se indica, lo busca en la carpeta del proyecto. directorioYFichero puede ser también el camino a un directorio, sin indicar al final el nombre de ningún fichero.
`File(String directorio, String nombreFichero)`	Recibe en la instanciación del objeto el camino completo donde está el fichero, como primer parámetro, y el nombre del fichero como segundo parámetro.
`File(File directorio, String fichero)`	Recibe en la instanciación del objeto un objeto de tipo File, que hace referencia a un directorio, como primer parámetro y el nombre del fichero como segundo parámetro.

- En Linux se utiliza como prefijo de una ruta absoluta "/". En 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 ficehro3 = new file ("direc, "ejemplo3.txt");
```

La clase File tiene los siguientes métodos que sirven para ficheros y directorios:

Método	Explicación
boolean canRead()	Informa si se puede leer la información que contiene.
boolean canWrite()	Informa si se puede guardar información.
boolean exists()	Informa si el fichero o el directorio existe.
boolean isFile()	Devuelve true si el objeto File corresponde a un fichero normal.
boolean isDirectory()	Devuelve true si el objeto File corresponde a un directorio.
long lastModified()	Retorna la fecha de la última modificación.
String getName()	Devuelve el nombre del fichero o directorio.
String getPath()	Devuelve la ruta relativa.
String getAbsolutePath()	Devuelve la ruta absoluta del fichero/directorio.
String getParent()	Devuelve el nombre del directorio padre o null si no existe.
delete()	Borra el fichero.
long length()	Retorna el tamaño del archivo en bytes.
boolean renameTo(File nuevoNombre)	Cambia el nombre del fichero asignándoel nuevoNombre.
boolean mkdir()	Crea el directorio. Solo lo creará si no existe.
String[] list()	Devuelve un array de String con los nombres de ficheros y directorios asociados al objeto File.
File[] listFiles()	Devuelve un array de objetos File conteniendo los ficehros que estén dentro del directorio representado por el objeto File.
boolean delete()	Borra el fichero o directorio asociado al objeto File.
Boolean createNewFile()	Crea un nuevo fichero, vacío, asociado a File si y solo si no existe un fichero con dicho nombre.

Hay que tener en cuenta que se puede:

- indicar el nombre de un fichero **sin la ruta**, se buscará el fichero en el directorio actual.
- indicar el nombre de un fichero con la **ruta relativa**.
- indicar el nombre de un fichero con la **ruta absoluta**.

El siguiente programa Java muestra los nombres de los archivos y directorios que se encuentren en el directorio que se pasa como argumento (args[0]):

```
import java.io.File;


public class _2x2x01
{
    public static void metodo(String[] args)
    {
        File file = new File(args[0]);

        if( file.isDirectory() )
        {
            File[] ficheros = file.listFiles();

            for( File f : ficheros )
                System.out.println(f.getName());
        }
    }


    public static void main(String[] args)
    {
        String[] argumentos = { "C:\\" };
        metodo(argumentos);
    }
}
```

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 o 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 directorio:

```
public class VerDir {
  public static void main(String[] args) {
    String dir = "."; //directorio 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 mostrará 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
```

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 dos 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"); //directorio 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();}
 }
}
```

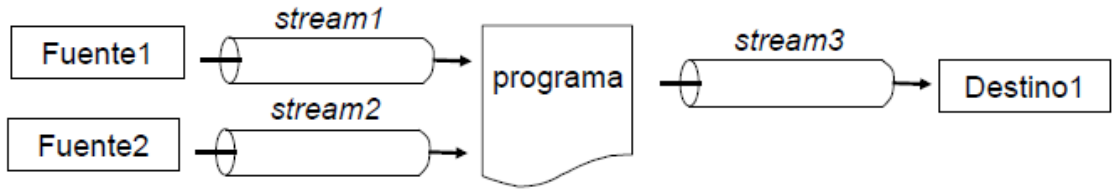
Para borrar un fichero o directorio usamos el método ***delete(),***en el ejemplo anterior no podemos borrar el directorio creado porque contiene ficheros, antes habría que eliminar estos ficheros. Para borrar el objeto f2 escribimos:

```
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 un bloque de **try-cath.**

3. FLUJOS O STREAMS. TIPOS

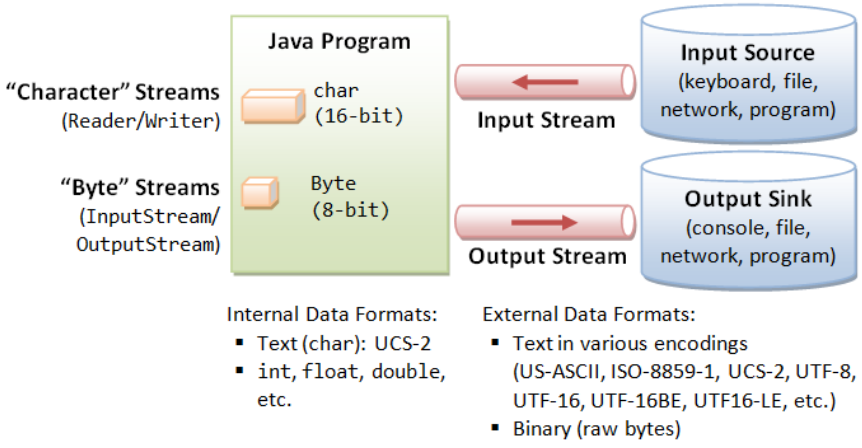
El sistema de entrada/salida en Java presenta una gran variedad de clases que se implementan en el paquete **java.io**. Utiliza la abstracción del flujo o **stream** para tratar la comunicación de información entre una fuente y un destino, y por tanto, las operaciones que un programa realiza sobre un flujo son independientes del dispositivo al que esté asociado. Cualquier programa que tenga que obtener información abrirá 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.



Así pues, un **archivo** es simplemente un flujo externo, es decir, una secuencia de bytes almacenados en un dispositivo externo. Los programas leen o escriben en el flujo, que puede estar conectado a un dispositivo o a otro.

Hay dos tipos de flujos de datos definidos:

- **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 soportaba 8 bits no manejando caracteres Unicode de 16 bits que se utiliza con fines de internacionalización.
- **Flujos de bytes (8 bits):** realizan operaciones de entradas y salidas de bytes y su uso está orientado a la lectura y 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 entreda/salida que se pueden utilizar.



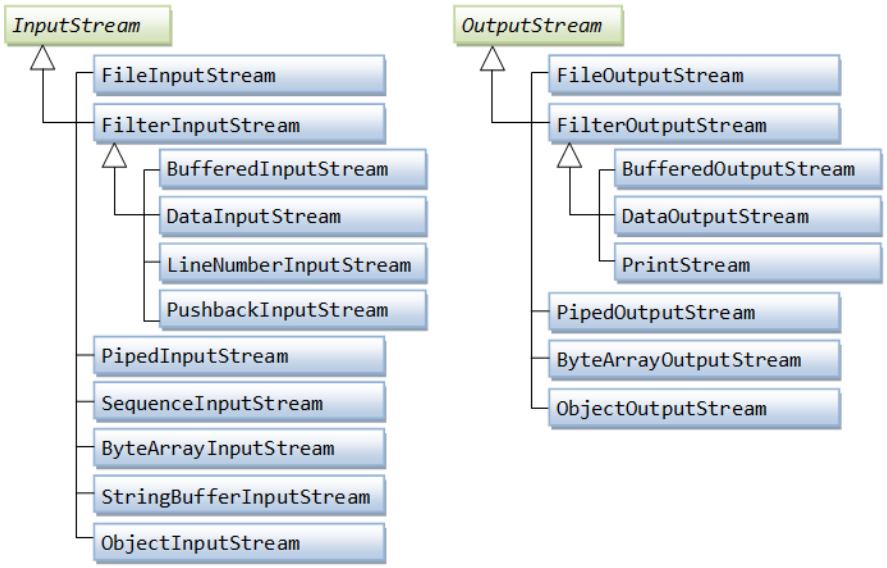
En Java, la entrada desde el teclado y la salida por la pantalla están gestionadas por la clase **System**. Esta clase pertenece al paquete **java.lang** y tienen tres atributos, los llamados flujos predefinidos: in, out y err, que son public y static:

- **Sytem.in**: hace referencia a la entrada estándar de datos (teclado).
- **System.out**: hace referencia a la salida estándar de datos (pantalla).
- **System.err**: hace referencia a la salida estándar de información de errores (pantalla).

3.1. FLUJOS DE BYTES

La clase **InputStream** representa las clases que producen entradas de distintas fuentes, como un array de bytes, un objeto String, un fichero, una tubería (se ponen los elementos en un extremo y salen por otro) ,etc. La clase abstracta **OutputStream** decide donde irá la salida, como a un array de bytes, un fichero o una tubería.

La siguiente figura muestra la jerarquía de clases para la lectura y la escritura de flujos de bytes.



Los **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	Flujo de entrada hacia fichero, lo usaremos 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 **OutputStream**se incluyen en 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 desee escribir aquí acaba automáticamente como entrada del **PipedInputStream** asociado. Implementa el concepto de "tubería"
FilterOutputStream	Proporciona funcionalidad útil a otras clases **OutputStream**

Dentro de los flujos de bytes están las clases **FileInputStream** y **FileOutputStream**que manipulan los flujos de bytes que provienen o se dirigen hacia ficheros en disco y que estudiaremos en siguientes apartados.

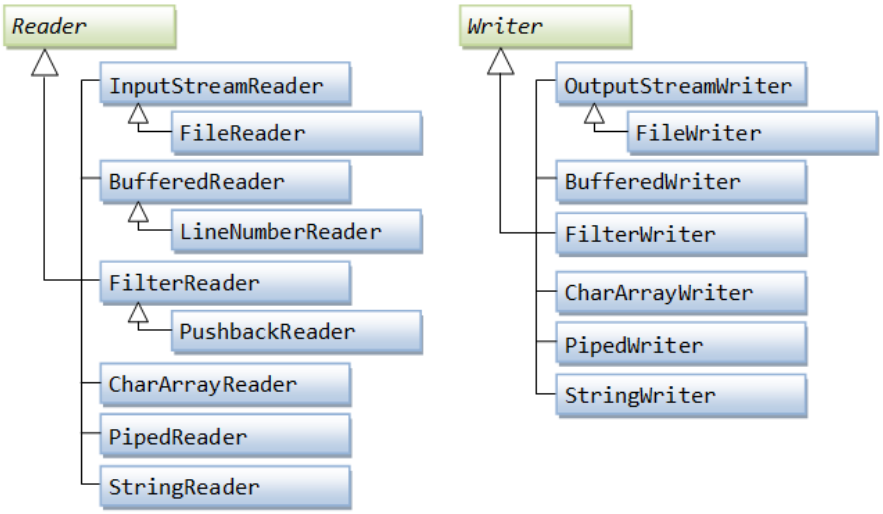
3.2. FLUJOS DE CARACTERES

Las clases **Reader** y **Writer** manejan flujos de caracteres Unicode. Hay ocasiones en las que hay que usar las clases que manejan bytes en combinación con las clases que manejan caracteres. Para lograr esto, existen varias clases "puente" (que convierten los flujos de bytes a flujos de caracteres): **InputStreamReader** convierte un InputStream en un Reader (lee bytes y los convierte a caracteres) y **OutputStreamReader** convierte un OutputStream en un Writer.

La tabla muestra la correspondencia entra 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 siguiente figura muestra la jerarquía de clases para la lectura y la 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 la buferización de datos, **BufferedReader** y **BufferedWriter**, las cuales se utilizan para evitar que cada lectura o escritura acceda directamente al fichero, ya que usan un buffer intermedio entre la memoria y el flujo de datos.

4. FORMAS DE ACCESO A UN FICHERO

Hay dos formas de acceso a la información almacenada en un fichero: acceso secuencial y acceso 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 en 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 haya 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 fichero puede ser binario o de caracteres. Para el acceso binario se usan las clases **FileInputStream y FileOutputStream;** para el acceso a caracteres (texto) se usan las clases **FileReader** y F**ileWriter**. En acceso aleatorio se utiliza la clase **RandomAccessFile**.

4.1. 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 realicen 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:** Frecuencia las altas se realizan al final el último registro insertado , es decir, sólo 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 1 y escribirlos en un fichero auxiliar, salvo el que deseamos dar de baja. Una vez es reescrito hemos de borrar el fichero inicial y renombrar el fichero auxiliar dándole nombre del fichero original
- **Modificaciones**: consiste en localizar el registro a modificar, efectuar la modificación y reescribir el fichero inicial y 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 adjudicaciones 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 aprovechar 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 el rey escribirlos totalmente. Para las aplicaciones interactivas que incluyen peticiones o actualizaciones de registros individuales, los ficheros secuenciales ofrecen un rendimiento pobre.

4.2. 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. Críticas de acceso aleatorio en disco manipulan direcciones son activas y 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 unívoca a un registro). Por ejemplo disponemos de un fichero de empleados con tres campos: identificador, apellido y salario. Usamos el identificado como campo clave del mismo, y le damos 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 este caso, Habría que usar una nueva posición libre en el fichero para ubicar dicho registro utilizar una zona de excedentes dentro del miso para ir ubicando estos registros.

Veamos cómo se realizan las operaciones típicas:

Consultas: Para consultar determinado registronecesitamos 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 es 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 de registro a modo de switch que tenga el valor 1 cuando el registro exista y le damos el valor 0 para darle de baja, físicamente el registro no desaparece el disco. Habría que localizar el registro a dar de baja a partir de un campo que clave y reescribir en el campo el valor 0.

Modificaciones: A modificar un registro al que localizarlo, necesitamos saber su clave para aplicar la función de conversión y así obtener la dirección, modificar los datos que nos interesan y reescribir el registro en esa posición.

Una de las principales **ventajas** 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 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 a fichero, ya que se pueden producir huecos (posiciones no ocupadas) entre un registro y otro.

5. CLASES PARA LA 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 lo binarios pueden almacenar cualquier tipo de dato (*int, float, boolean, etc.)*

5.1. Ficheros de texto: Clases FileReader y FileWriter

Los ficheros de texto, que normalmente se generan con un editor de textos, almacenan caracteres alfanuméricos en un formato estándar (ASCII, Unicode, UTF-8, etc.). Para trabajar con ellos en Java, se utilizan la clase **FileReader** para leer caracteres y la clase **FileWriter** para escribir los caracteres en el fichero. Cuando trabajamos con ficheros, las lecturas o escrituras realizadas sobre un fichero deben escribirse dentro de un manejador de excepciones **try-catch**.

Al instanciar un objeto con la clase **FileReader**, el programa abre el fichero que se envía como argumento para lectura y su información se puede leer de forma secuencial carácter a carácter. Si el fichero no existe o no es válido, se lanza la excepción FileNotFoundException. El en un

Constructores de la clase FileReader:

Método	Explicación
`FileReader(File fichero)`	Lanza la excepción FileNotFoundException si el fichero pasado no existe.
`FileReader(String fichero)`	Lanza la excepción FileNotFoundException si el fichero pasado no existe.

Principales métodos de la clase FileReader:

Método	Explicación
`int read()`	Lee un carácter y lo devuelve.
`int read(char[] buf)`	Lee hasta buf.length caracteres de datos de la matriz de caracteres (buf). Los caracteres leídos se van almacenando en buf pasado como parámetro.
`int read(char[] buf, int desplazamiento, int n)`	Lee hasta n caracteres de datos de una 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 y cuando terminemos de usarlo lo cerraremos mediante el método **close().**

El siguiente ejemplo lee cada uno de los caracteres del fichero de texto con el 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 {
    File fichero = new File("Fichero1.txt");  //declarar fichero
FileReader fic = new FileReader(fichero); //crear el flujo de entrada
//crear el flujo de entrada

    int i;
    while ((i = fic.read()) != -1) //se va leyendo un carácter
System.out.println( (char) i + "==>"+ 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\\LeerFicheroTexto.java");
```

Para ir leyendo de 20 en 20 caracteres escribimos:

```
char b[]= new char[5];
    while ((i = fic.read(b)) != -1) System.out.println(b);
```

Al instanciar un objeto con la clase **FileWriter**, el programa abre el fichero especificado con el fin de guardar información, pero carácter a carácter. Si el fichero no existe, el programa lo crea de forma automática. Si el disco está lleno o protegido contra escritura, se lanza la excepción IOException.

Constructores de la clase FileWriter:

Constructor	Explicación
FileWriter(String nombreFich)	Recibe como parámetro el nombre del fichero a abrir y borra el contenido previo del fichero comenzando a escribir desde el principio.
FileWriter(File fichero)	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabaja y borra el contenido previo del fichero comenzando a escribir desde el principio.
FileWriter(String nombreDeFich, boolean append)	Recibe como parámetro el nombre del fichero a abrir y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.
FileWriter(File fichero, boolean append)	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabaja y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.

Principales métodos de la clase FileWriter:

Método	Explicación
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 de una matriz buf 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 y **FileWriter.**

El siguiente ejemplo escribe caracteres en un fichero de nombre *FichTexto.txt* (si no existe lo crea). Los caracteres escriben uno a uno y se obtiene de un String que se convierte en la 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 aray usando fic.write(cad). El siguiente ejemplo escribe cadenas de caracteres que se obtienen de un array de String, las cadenas se irán insertado 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 FileReader de la siguiente manera, colocando en el segundo parámetro del constructor el valor true:

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

Otro ejemplo

```
public static void main(String args[]) throws IOEXception{
   String cadTecFich, cadFichPant = "";
   int car;
   char cara;

  FileWriter fichEsc = new FileWriter("nuevo.txt");
  System.out.print("\nIntroduce una frase: ");
   cadTecFich = Leer.pedirCadena();
   fichEsc.write(cadTecFich);
  fichEsc.close();

   FileReader fichLect = new FileReader("nuevo.txt");
   car = fichLect.read();
  while(car!=-1){
   cara = (char) car;
     cadFichPant = cadFichPant + cara;
   car = fichLect.read();
  }
  fichLect.close();
  System.out.println("\nLa frase leída del fichero es: \"" + cadFichPant + "\"");
}
```

5.1.1. Las Clases BufferedReader y BufferedWriter

Las clases **BufferedReader** y **BufferedWriter** se pueden utilizar sobre las clases FileReader y FileWriter u otros flujos de caracteres para realizar operaciones de entrada/salida con un buffer, en lugar de carácter a carácter.

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

FileReader no contiene métodos que nos permitan leer líneas completas, pero **BufferReader** 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 disponen del método **read()** para leer un carácter. Para construir un **BufferedReader** necesitaremos la clase **FileReader:**

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

Por lo tanto, con la clase**BufferedReader** dispone del método readLine(), que lee una línea del fichero y la devuelve, o devuelve null si no hay nada que leer o se llega al final del fichero. Para construir un objeto BufferedReader, se necesita la clase FileReader:

```
        FileReader fileR = new FileReader(nombreFichero);
BufferedReaderbufferedR = new BufferedReader(fileR);
```

El siguiente ejemplo le el fichero *LeerFichTexto.java* línea por línea y la 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 ");}
  }
}
```

EJEMPLO

```
public static void main(String[] args) throws IOException{
   String cadena;
    File fich = new File("fichero.txt");
   BufferedReader flujo = new BufferedReader(new FileReader(fich));
   if (fich.exists()){
       System.out.println("\nEsta es la información que contiene el fichero: ");
      cadena = flujo.readLine();
      while(cadena!=null){
          System.out.println(cadena);
         cadena = flujo.readLine();
      }
   }
}
```

La clase **BufferedWriter** dispone del método write(), que escribe una línea en el fichero, y del método newLine(), que escribe un salto de línea en el fichero. Para construir un objeto BufferedWriter, se necesita la clase FileWriter:

```
        FileWriter fileW = new FileWriter(nombreFichero);
        BufferedWriterbufferedW = new BufferedWriter(fileW);
```

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

El siguiente ejemplo escribe 10 filas de caracteres en un fichero de texto y después de escribir

```
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 ");}
}
```

cada fila salta una linea con el método newline():

La clase **PrintWriter**, 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 asi:

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

5.2. Ficheros binarios: las Clases FileInputStream y FileOutputStream

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 permiten trabajar con ficheros son **FileInputStream** (para entrada) y **FileOutputStream** (para salida), que operan con flujos de bytes y crean un enlace entre el flujo de bytes y el fichero.

Cuando se instancia un objeto con la clase**FileInputStream**, el programa abre el fichero (que se debe enviar como argumento del constructor) en modo lectura. Una vez abierto, se podrá leer la información que contiene de forma secuencial byte a byte.

Constructores de la clase FileInputStream:

Constructor	Explicación	Excepción que lanza
`FileInputStream(String nombreDeFichero)`	Recibe como parámetro el nombre del fichero a abrir.	`FileNotFoundException` si el fichero no existe.
`FileInputStream(File fichero)`	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar.	`FileNotFoundException` si el fichero no existe.

Los principales métodos que proporciona la clase FileInputStream para la 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	Explicación
`int read()`	Lee un byte y lo devuelve
`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
`void close()`	Cierra el fichero.

EJEMPLO

```
public static void main(String args[]) throws IOException{
  int c;
  try{
    FileInputStream f = new FileInputStream("/pedidos.txt"); //Se puede poner / o  \\
    /*Al poner / va a buscar el fichero en la raíz del disco donde está el proyecto.
    Si no se ponela /, busca el fichero en la carpeta donde está ahora mismo*/

    while((c=f.read())!=-1)
      System.out.print((char) c);
    /*Si no se hace la conversión visualizaría el código ASCII
    de cada carácter que hayguardado en el fichero*/

    f.close();
  }
  catch(FileNotFoundException e){
    System.out.println("El fichero no existe.");
  }
}
```

Cuando se instancia un objeto con la clase**FileOutputStream**, lo que hace el programa es abrir el fichero para escritura. Una vez abierto, se podrá guardar información byte a byte. Si el fichero no existe, se crea en ese momento.

Constructores de la clase FileOutputStream:

Constructor	Explicación
`FileOutputStream(String nombreFich)`	Recibe como parámetro el nombre del fichero a abrir y borra el contenido previo del fichero comenzando a escribir desde el principio.
`FileOutputStream(File fichero)`	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabaja y borra el contenido previo del fichero comenzando a escribir desde el principio.
`FileOutputStream(String nombreDeFich, boolean append)`	Recibe como parámetro el nombre del fichero a abrir y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.
`FileOutputStream(File fichero, boolean append)`	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabaja y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.

Principales métodos de la clase FileOutputStream:

Método	Explicación
`int write(int byte)`	Escribe el byte que recibe como argumento en el fichero.
`int write(byte[] b)`	Escribe todos los bytes que contiene b, es decir escribe b.length bytes.
`write(byte[] b, int desaplazamiento, int n)`	Escribe*n* bytes a partir de la matriz de bytes de entrada *b* comenzando por b [*desplazamiento]*.
`void close()`	Cierra el fichero.

El siguiente ejemplo escribe en un fichero y después lo visualiza:

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

EJEMPLO

```
public static void main(String args[]) throws IOException{
  String cadena;
  int car;
  char cara;

  FileOutputStream f1 = new FileOutputStream("/hola.txt");
  System.out.print("\nIntroduce una frase: ");
  cadena=Leer.pedirCadena();
  for(int pos=0;pos<cadena.length();pos++){
    f1.write(cadena.charAt(pos));
  }
  f1.write('\n');//Para separar las frases
  f1.close();

  System.out.println("\nEl contenido del fichero es: ");

  FileInputStream f2 = new FileInputStream("/hola.txt");
   car=f2.read();
  while(car!=-1){
    cara=(char)car;
    System.out.print(cara);
      car=f2.read();
  }
  f2.close();
 }
```

5.2.1. Las Clases DataInputStream y DataOutputStream

Para leer y escribir datos de tipos primitivos (como boolean, int, long, float, double, char, etc.), se utilizan las clases **DataInputStream** y **DataOutputStream**. Además de los métodos ***read() y write()***, estas clases proporcionan métodos para la lectura y escritura de tipos primitivos de un modo independiente de la máquina. Algunos de los métodos disponibles son:

Métodos para lectura	Métodos para escritura
boolean readBoolean()	void writeBoolean(boolean b)
byte readByte()	void writeByte(int i)
int readUnsignedByte()	void writeBytes(String s)
int readUnsignedShort()	void writeShort(int i)
short readShort()	void writeChars(String s)
char readChar()	void writeChar(int i)
int readInt()	void writeInt(int i)
long readLong()	void writeLong(long l)
float readFloat()	void writeFloat(float f)
double readDouble()	void writeDouble(double d)
String readUTF()	void writeUTF(String s)

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

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

O bien

```
DataInputStream dataIS = new DataInputStream(new FileInputStream(fichero));
```

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

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

Hay que tener mucho cuidado con leer un fichero en el mismo formato que se ha escrito porque, de no ser así, daría errores en la ejecución al no corresponderse los tipos.

Para saber que se ha alcanzado el final del fichero, los métodos lanzan la excepción **EOFException**, así que hay que recogerla y tratarla correctamente.

El siguiente ejemplo inserta los 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, recorreremos 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");
      DataOutputStream dataOS = new
                        DataOutputStream(new FileOutputStream(fichero));


   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 deban 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");
    DataInputStream dataIS = new
                      DataInputStream(new FileInputStream(fichero));

   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
```

EJEMPLO

```
import utilidades.Leer;
import java.io.*;

public class EjemploData{
  public static void main(String args[]){
    int numInt;
    String cadena;
    float numFloat;

    FileOutputStream fichEscrit = new FileOutputStream("prueba.txt");
    DataOutputStream escritTipos = new DataOutputStream(fichEscrit);

    System.out.print("\nInserta un número entero: ");
    numInt=Leer.pedirEnteroValidar();
    escritTipos.writeInt(numInt);

    System.out.print("\nInserta un número con decimales: ");
    numFloat=Leer.pedirFloatValidar();
    escritTipos.writeFloat(numFloat);

    System.out.print("\nInserta una cadea: ");
    cadena=Leer.pedirCadena();
    escritTipos.writeChars(cadena);

    escritTipos.close();

    FileInputStream fichLect = new FileInputStream("prueba.txt");
    DataInputStream lectTipos = new DataInputStream(fichLect);

    numInt=lectTipos.readInt();
    numFloat=lectTipos.readFloat();
    cadena=lectTipos.readLine();

    System.out.println("\nLos datos leídos del fichero son: \n");
    System.out.println("\t"+numInt+"\t"+numFloat+"\t"+cadena);
  }
}
```

5.3. Objetos en ficheros binarios: Las Clases ObjectInputStream y ObjectOutputStream

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 guardarlo en un fichero, tendríamos que guarder cada atributo que forma parte del objeto por separado, que vuelven borrosos y si tenemos gran cantidad de objetos. Por ello, Java nos permite guardar objetos en ficheros binarios; para poder hacerlo el objeto tiene implementar la interfaz **Serializable**que dispone de métodos que permiten guardar y leer objetos en fichero binarios.Los más importantes son:

Object readObject(): si utiliza para leer un objeto del **ObjectInputStream**. Puede lanzar excepciones IOException y ClassNotFoundException. En sus

void writeObject(Object obj): escribir el objeto especificado en el **ObjectOutputStream.** Puede lanzar la excepción IOException.

Por tanto, se llama **persistencia** al proceso de almacenar toda la información que contiene un objeto, manteniendo su estructura, en un fichero binario. En Java, para poder guardar un objeto en un fichero binario, dicho objeto tiene que implementar la interfaz **Serializable**, que dispone de métodos que permiten escribir y leer objetos en ficheros binarios:

Método para escribir un objeto	**void writeObject(ObjectOutputStream stream) throws IOException**
Método para leer un objeto	**void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException**

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 **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;}
    public int getEdad(){return 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, crear el flujo de salida ObjectOutputStream, que procesa los datos y está vinculado al fichero**FileOutputStream**.

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

O bien,

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

Por último, el método writeObject() escribe los objetos al flujo de salida y los guarda en un fichero en el disco: dataOS.writeObjetct(persona).

```
dataOS.writeObject(persona);
```

```
import java.io.*;

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

   Persona persona;//defino variable persona

   File fichero = new File("FichPersona.dat");//declara el fichero
   FileOutputStream fileout = new FileOutputStream(fichero,true);  //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","Maria Jesús"};

   int edades[] = {14,15,13,15,16,12,16,14,13};
   System.out.println("GRABO LOS DATOS DE PERSONA.");
   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
      System.out.println("GRABO LOS DATOS DE PERSONA.");
   }
   dataOS.close();  //cerrar stream de salida
   }
}
```

Por ejemplo, para leer objetos *Persona*del fichero, se necesita crear el flujo de entrada a disco **FileInputStream**, y a continuación, crear el flujo de entrada **ObjectInputStream**, que procesa los datos y está vinculado al fichero**FileInputStream**..

```
File fichero = new File("C:\\EJERCICIOS\\FichPersona.dat");
FileInputStream filein = new FileInputStream(fichero);
ObjectInputStream dataIS = new ObjectInputStream(filein);


O bien

File fichero = new File("C:\\EJERCICIOS\\FichPersona.dat");
ObjectInputStream dataIS = new ObjectInputStream(new FileInputStream(fichero));
```

Por último, el método **readObject**() lee el objeto del flujo de entrada y 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 llegue al final del fichero, entonces, se lanzará la excepción ***IOException.***

```
persona = (Persona) dataIS.readObject();
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");
        ObjectInputStream dataIS = new ObjectInputStream(new FileInputStream(fichero));

        int i = 1;
        try {
            while (true) { // lectura del fichero
                persona = (Persona) dataIS.readObject(); // leer una Persona
                System.out.print(i + "=>");
                i++;
                System.out.printf("Nombre: %s, edad: %d %n",
                        persona.getNombre(),persona.getEdad());

            }
        } catch (EOFException eo) {
            System.out.println("FIN DE LECTURA.");
        } catch (StreamCorruptedException x) {
        }

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

Problemas con los ficheros de objetos:

Existe un problema con los ficheros de objetos. A crear un fichero de objetos se creará una cabecera inicial con información, y a continuación se añaden los objetos, pero al utilizarlo de nuevo para añadir más registros, y crea una nueva cabecera y añade de los objetos a partir de esa cabecera. El problema surge a leer el fichero cuando la lectura se encuentra con la segunda cabecera, y aparece la excepción StreamCorruptedException y no podemos leer más objetos.

La cabecera se crea cada vez que se pone new ObjectInputStream (fichero). Para que no se añadan estas cabeceras lo que se hace es redefinir la clase ObjectInputStream 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, hacemos es el método no haga nada. De manera que es el fichero ya se ha creado se llamara a ese método de la clase redefinida.

```
public class MiObjectOutputStream extends ObjectOutputStream
{
    /** Constructor que recibe OutputStream */
    public MiObjectOutputStream(OutputStream out) throws IOException
    {
        super(out);
    }

    /** Constructor sin parámetros */
    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 ya existe, se crea el objeto con la clase redefinida, y si no existe, el fichero se queda 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
```

6. 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 buen primer objeto, interiormente 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 permite posicionarnos en una posición concreta de un fichero y métodos acceder a su contenido de forma aleatoria o directa (no secuencial). Esta clase no es parte de la jerarquía Reader/Writer ni InputStream/OutputStream, ya 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 éstos pueden lanzar la excepción ***FileNotFoundException:***

Método	Explicación
RandomAccessFile(String nombrefichero, String modoAcceso)	nombrefichero es el nombre del fichero incluido el path y modoAcceso es el argumento que determina si el contenido del fichero se va a poder solo leer (**"r"**) o leer y escribir (**"rw"**).
RandomAccessFile(File objetoFile, String modoAcceso)	**objetoFile**es el objeto fichero y modoAcceso es el argumento que determina si el contenido del fichero se va a poder solo leer (**"r"**) o leer y escribir (**"rw"**).

Como se ha mencionado el argumento modoAccceso puede tener dos valores:

Modo de acceso	Significado
r	Abre el fichero el modo de solo lectura. El fichero debe existir. Operación de escritura en este fichero lanzaron la excepción **IOException**.
rw	Abre el fichero modo lectura y escritura. No existe se crea.

En un 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 o cursor que indica la posición actual en el fichero. Cuando el fichero se crea, el puntero se coloca en 0, es decir, apuntando al principio del mismo.

Una vez abierto un fichero, se pueden utilizar sus métodos *read()* y *write()* para cada tipo de dato y las sucesivas llamadas a estos métodos *read()* y *write()* ajustan el puntero según la cantidad de bytes leídos o escritos.

Principales métodos de la clase RandomAccessFile:

Método	Explicación
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 y la posición

	*length()*marca el final del fichero.
int skipBytes(int desplazamiento)	Desplaza el puntero del fichero desde la posición actual el número de bytes indicados en desplazamiento.

Cada tipo de dato tiene un tamaño concreto:

boolean (1 bit) **byte** (1 byte) **carácter Unicode** (2 bytes)
short (2 bytes) **int** (4 bytes) **long** (8 bytes)
float (4 bytes) **double** (8 bytes)

El siguiente programa Java abre un fichero para lectura y escritura, escribe los nombres que se leen por teclado en el fichero al final, y después los lee del fichero para mostrarlos en pantalla. A partir de los nombres leídos teclado y escritos en el fichero, el posicionamiento para empezar a recorrer los registros empieza en cero.

EJEMPLO

```
       public static void main(String[] args) throws IOException {
                 RandomAccessFile fiche;
                 String nomNuevo, nombre;
                 String cadena = "";
                 fiche = new RandomAccessFile("directo.txt", "rw");

                 System.out.print("\n Introduce un nombre (\"Ninguno\" para salir): ");
                 nomNuevo = Leer.pedirCadena(cadena);
                 while (!nomNuevo.equals("Ninguno")) {
                 fiche.seek(fiche.length());
                 fiche.writeBytes(nomNuevo);
                 fiche.write('\n');  //Para separar los nombres
                 System.out.print("Introduce otro nombre (\"Ninguno\" para salir):");
                 nomNuevo = Leer.pedirCadena(cadena);
                 }

                 fiche.seek(0);
                 nombre=fiche.readLine();
                 while(nombre != null) {
                 System.out.println("Nombre leído: " + nombre);
                 nombre = fiche.readLine();
                 }
                 fiche.close();
                }

    }
```

```
 Introduce un nombre ("Ninguno" para salir):
Juan
Introduce otro nombre ("Ninguno" para salir):
Pepe
Introduce otro nombre ("Ninguno" para salir):
Luis
Introduce otro nombre ("Ninguno" para salir):
ana
Introduce otro nombre ("Ninguno" para salir):
Laura
Introduce otro nombre ("Ninguno" para salir):
Ninguno
Nombre leído: Juan
Nombre leído: Pepe
Nombre leído: Luis
Nombre leído: ana
Nombre leído: Laura
```

El ejemplo que se muestra a continuación inserta datos de los empleados en un fichero aleatorio. Los datos a insertar: apellido departamento y salario, se obtiene de varios arrays que se llenan en el programa, los datos se van introduciendo de forma secuencial por lo que no va 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:

- En primer lugar un entero, que es el identificador, ocupa 4 bytes.
- A continuación una cadena de diez caracteres, es el apellido. Como java utiliza caracteres UNICODE, cada carácter de una cadena de caracteres ocupa 16 bits (2 bytes), por lo 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.

El fichero se abre el 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;//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 usa el fichero anterior y visualiza todos sus registros. El posicionamiento para empezar a recorrer los registros empieza en 0, y 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;
        }//fin bucle for
        file.close();  //cerrar fichero
    }
}
```

La ejecución muestra la siguiente salida:

```
ID: 1, Apellido: FERNANDEZ, Departamento: 10, Salario: 1000,45
ID: 2, Apellido: GIL, Departamento: 20, Salario: 2400,60
ID: 3, Apellido: LOPEZ, Departamento: 10, Salario: 3000,00
ID: 4, Apellido: RAMOS, Departamento: 10, Salario: 1500,56
ID: 5, Apellido: SEVILLA, Departamento: 30, Salario: 2200,00
ID: 6, Apellido: CASILLA, Departamento: 30, Salario: 1435,87
ID: 7, Apellido: REY, Departamento: 20, Salario: 2000,00
```

Para consultar un empleado determinado no es necesario recorrer todos los registros del fichero. Conociendo su identificador, se puede acceder a la posición que ocupa dentro del fichero y obtener sus datos. Por ejemplo, desea 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;
//calculo donde empieza el registro
posicion = (identificador -1 ) * 36;
if(posicion >= file.length() )
   System.out.printf("ID: %d, NO EXISTE EMPLEADO...",registro);
 else{
    file.seek(posicion); //nos posicionamos
    id=file.readInt(); // obtengo id de empleado
    //obtener el resto de datos, como en el ejemplo anterior
```

Para añadir registros a partir del último insertado hemos de posicionar en 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
```

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
```

7. 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 por pantalla 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);
    }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / 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 el error no es capaz de manejarlo. En este caso, el método **lanzará una excepción**.

Ejemplos

```
try {
  resultado = dividendo / divisor;
  System.out.println(resultado);
} catch (ArithmeticException e) {
  System.out.println("División por cero");
}
...
try {
  resultado = m / v[i];
} catch (ArithmeticException e) {
  System.out.println("División por cero");
} catch (ArrayIndexOfBoundsException e) {
  System.out.println("Índice fuera de rango");
}
```

Si este código provoca una excepción, salta aquí

Pueden provocarse distintos clases de excepciones

Las excepciones en Java son objetos de clases derivadas de la clase base **Exception**donde define las excepciones que los programas deberían tratar (IOException, Arithmetic Exception…), que a su vez es una clase derivada de la superclase base **Throwable**que engloba a todas las excepciones.

7.1. CAPTURA DE EXCEPCIONES

¿Es obligatorio capturar todas las posibles excepciones? “No, además el código puede quedar atestado de sentencias try..catch disminuyendo la legibilidad del código…”

La clase RuntimeException define excepciones que songestionadas de forma automática por el intérprete Java. Elresto son de obligatorio tratamientoRuntimeException: ArithmeticException,IndexOfBoundsException, NullPointerException, ...No RuntimeException: IOExcepcion, ...

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 try va seguido de uno o más bloques catch. Cada bloque **catch** especifica un tipo de excepción que puede atrapar (capturar) y contiene un manejador de excepciones.

Después del último bloque catch puede aparecer un bloque **finally** (opcional, ejecutando en cualquier caso…). Las excepciones pueden hacer que un método acabe deforma prematura. En ocasiones es necesario realizar tareasespecíficas (liberar recursos, cerrar archivos,...) antes determinar, tanto si se produce una excepción como si no. Este bloque finally se utiliza 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 de programa Java 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 ejemploExcepciones1 {
    public static void main(String[] args) {
        String cad1 = "20", cad2 = "0", mensaje = "";
        int nume, denom, cociente;
        int[] arraynum = new int[4];
        try {
            //codigo 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 (ArithmeticException 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
```

Para capturar cualquier excepción se utiliza la clase base **Exception**. Si se usa, hay que ponerla al final de la lista de manejadores para evitar que los manejadores que van detrás queden ignorados.

Por ejemplo, el siguiente código maneja varias excepciones, si se produce alguna para la que nos ha definido manejador será capturada por **Exception:**

```
try {
       //código que puede producir errores
    } catch(NumberFormatException ex){
       //tratamiento excepción
    } catch(ArithmeticException 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 producida, se pueden invocar los métodos de la clase base **Throwable**. Algunos de estos métodos son:

- **String getMessage().** Devuelve la cadena de error del objeto.
- **String getLocalizedMessage().** Crea una descripción local de este objeto.
- **String toString().** Devuelve una breve descripción del objeto,
- **void printStackTrace()**, **printStackTrace(PrintStream)** o **printStackTrace(PrintWriter).**Visualiza el objeto y la traza de pila de llamadas lanzada.

Por ejemplo el siguiente bloque **try-cath:**

```
public class ejemploExcepciones2 {
    public static void main(String[] args) {
        String cad1 = "20", cad2 = "0", mensaje;
        int nume, denom, cociente;
        int[] arraynum = new int[4];
        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");
        }
    } // fin de main
} // fin de la clase
```

El anterior programa Java 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 buscará el manejador en las sentencias **try** más externas.

7.2. ESPECIFICACIÓN DE EXCEPCIONES

Para especificar una o varias excepciones se utiliza 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*:

```
publicstaticvoidmain(String[] args)
throws IOException, ClassNotFoundException {}
```

Si un método puede provocar una excepción que no es capaz de manejar, debe propagarla a niveles superiores. Enalguno de ellos debe ser capturada.

```
class ClaseX  {
  ...
  void metodoX(...) throws Exception {
    // algo que puede provocar una excepción
    ...
  }
}
...
try {
  ClaseX unObjetoX = new ClaseX(...);
  unObjetoX.metodoX(...)
} catch (Exception e) {
  //tratamiento de la excepción propagada desde metodoX
  ...
}
```

Aquellos métodos que pueden lanzar excepciones deben saber cuáles son esas excepciones que se pueden producir durante su ejecución e indicarlas en su declaración. Una forma típica de conocerlas es compilando el programa.

Por ejemplo, si al programa *EscribirPersona.java* 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 thrownObjectInputStream 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 las línea 7 (marcada en negrita) se produce una excepción que nos ha declarado (FileNotFoundException), esta excepción debe ser capturada mediante un bloque **try-catch** o declarada para ser lanzada mediante **throws.**

La siguiente figura muestra las excepciones que se pueden producir durante la ejecución de los diferentes métodos del paquete **java.io**:

Exception	Description
CharConversionException	Base class for character conversion exceptions.
EOFException	Signals that an end of file or end of stream has been reached unexpectedly during input.
FileNotFoundException	Signals that an attempt to open the file denoted by a specified pathname has failed.
InterruptedIOException	Signals that an I/O operation has been interrupted.
InvalidClassException	Thrown when the Serialization runtime detects one of the following problems with a Class.
InvalidObjectException	Indicates that one or more deserialized objects failed validation tests.
IOException	Signals that an I/O exception of some sort has occurred.
NotActiveException	Thrown when serialization or deserialization is not active.
NotSerializableException	Thrown when an instance is required to have a Serializable interface.
ObjectStreamException	Superclass of all exceptions specific to Object Stream classes.
OptionalDataException	Exception indicating the failure of an object read operation due to unread primitive data, or the end of data belonging to a serialized object in the stream.
StreamCorruptedException	Thrown when control information that was read from an object stream violates internal consistency checks.
SyncFailedException	Signals that a sync operation has failed.
UnsupportedEncodingException	The Character Encoding is not supported.
UTFDataFormatException	Signals that a malformed string in **modified UTF-8** format has been read in a data input stream or by any class that implements the data input interface.
WriteAbortedException	Signals that one of the ObjectStreamExceptions was thrown during a write operation.

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

Más información: https://docs.oracle.com/javase/tutorial/essential/exceptions

• Debemos usar las excepciones cuando...

- Puedan existir problemas en el estado interno del código
- Puedan existir riesgos de seguridad
- Pueda haber errores en un objeto o dato manipulado
- Parámetros erróneos

• Qué hacer al capturar una excepción...

- Escribir un mensaje de error y/o
- registrar el error en algún sitio y/o
- reintentar la operación con algún parámetro por defecto y/o
- restaurar el sistema en algún estado seguro y/o
- propagar la excepción si no podemos tratarla

8. FICHEROS XML

XML (eXtensible Markup Language) es un metalenguaje, es decir, un lenguaje para la definición de lenguajes de marcas. Permite jerarquizar y estructurar la información así como describir los contenidos dentro del propio documento, de forma independiente del lenguaje de programación y del sistema operativo empleados.

Los ficheros XML son ficheros de textos escritos en lenguaje XML donde la información está organizada de forma secuencial y en orden jerárquico. Existe una serie de marcas especiales, como los símbolos "<" y ">", 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.

El uso de ficheros XML en el desarrollo de aplicaciones (datos BBDD, copias partes BBDD, ficheros de configuración de programas…) hace que sean necesarias herramientas específicas (librerías) para acceder y manipular este tipo de archivos de forma potente, flexible y eficiente. Estas herramientas reducen los tiempos de desarrollo de aplicaciones y optimizan los propios accesos a XML, sin cargar innecesariamente el sistema.

```
▼<Libros xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="LibrosEsquema.xsd">
  ▼<Libro publicado_en="1840">
     <Titulo>El Capote</Titulo>
     <Autor>Nikolai Gogol</Autor>
   </Libro>
  ▼<Libro publicado_en="2008">
     <Titulo>El Sanador de Caballos</Titulo>
     <Autor>Gonzalo Giner</Autor>
   </Libro>
  ▼<Libro publicado_en="1981">
     <Titulo>El Nombre de la Rosa</Titulo>
     <Autor>Umberto Eco</Autor>
   </Libro>
 </Libros>
```

Para leer los ficheros XML y acceder a su contenido y estructura, se utiliza un procesador XML, analizador sintáctico o parser como herramienta para ello. Un parser XML es un módulo, biblioteca o programa encargado de leer un fichero XML y acceder a su contenido y estructura, mediante un modelo interno que optimiza su acceso.

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 tienen versiones o implementaciones particulares y diferentes para Java, C, VisualBasic y .NET.Utilizan dos enfoques muy distintos:

- **DOM (Document Object Model).**Un procesador que utilice este planteamiento, almacena toda la estructura del documento en memoria en forma de árbol, con nodos padre, nodos hijo y nodos finales (no tienen descendientes). Una vez creado el árbol, se recorren los diferentes nodos y se analiza a qué tipo particular pertenecen. Este procesamiento necesita más recursos de memoria y tiempo, sobre todo si los ficheros XML son grandes y complejos.
- **SAX (Simple API for XML).**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, etc.) en función de los resultados de lectura. Cada evento invoca a un método definido por el programador. Este procesamiento consume poca memoria, pero impide tener una visión global del fichero.

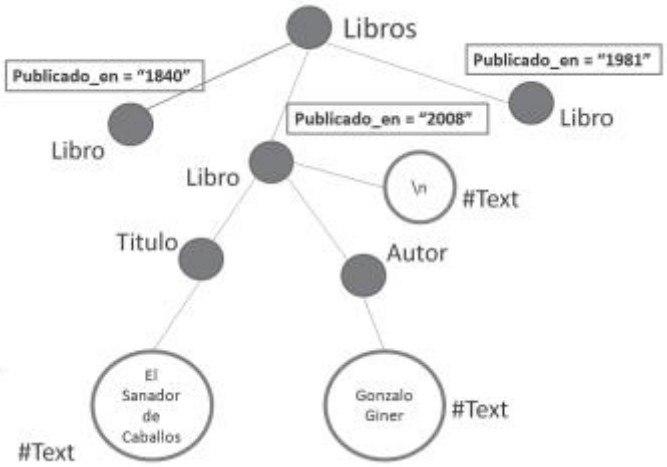
Estas herramientas DOM y SAX no validan los documentos XML. Es decir, solo comprueban que el documento está bien formado según la definición XML, pero no necesitan de un esquema asociado para comprobar si es válido con respecto a ese esquema.

8.1. ACCESO A DATOS CON DOM

DOM (Document Object Model) es una herramienta de programación que permite analizar y manipular dinámicamente y de manera global el contenido, el estilo y la estructura de un documento XML. Tiene su origen en el Consorcio World Wide Web (W3C).

A partir de un fichero XML, se almacena en memoria en forma de árbol un modelo del fichero, con nodos padre, nodos hijo y nodos finales (que son aquellos que no tienen descendientes). En este modelo, todas las estructuras de datos del fichero XML se transforman en algún tipo de nodo, y después, se organizan dichos nodos jerárquicamente en forma de árbol para representar la estructura descrita por el fichero XML.

Una vez creada en memoria esta estructura, los métodos de DOM permiten recorrer los distintos nodos del árbol y analizar a qué tipo particular pertenecen. En función del tipo de nodo, DOM ofrece una serie de funcionalidades para trabajar con la información contenida.



Para poder trabajar con DOM en Java, se necesitan las clases e interfaces que componen el paquete **org.w3c.dom**(contenido en el JSDK)y el paquete **javax.xml.parsers**dela API estándar de Java. 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 se usa 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.

Las principales interfaces que necesita un programa Java que utiliza DOM son:

- **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>.

El siguiente programa Java crea un fichero XML "empleados.xml" a partir de un fichero aleatorio de empleados y muestra el documento XML resultante en pantalla:

```
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);
                    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); //añadir salario
                }
                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
```

En este programa Java se realizan los siguientes pasos:

1) Lo primero que hemos de hacer es importar los paquetes necesarios.
2) En segundo lugar, se crea una instancia de *DocumentBuilderFactory* para construir el analizador sintáctico (parser). Se debe encerrar en un bloque try-catch porque se puede producir la excepción *ParserConfigurationException*.

3) Luego se crea un documento vacío de nombre *document* con el nodo raíz de nombre*Empleados* y se le asigna la versión de XML, la interfaz DOMImplementation permite crear objetos Document con nodo raíz.

4) El siguiente paso consiste en recorrer el fichero con los datos de los empleados y para cada registro se crea un nodo empleado con 4 hijos (id, apellido, dep y salario):
   - Para crear un elemento se utiliza el método *createElement(String),* que lleva como parámetro el nombre que se pone entre las etiquetas < y >.
   - A continuación se añaden los hijos de ese nodo (raíz) con la función definida *crearElemento(String, String, Element, Document)*. Esta funciónrecibe el nombre del nodo hijo, y sus textos o valores tiene que estar en formato String y crea el nodo hijo (<*id*>, <*apellido*>, <*dep*>o<*salario*>), lo añade a la raíz (<*empleado*>) y le asigna un valor (texto).
5) Por último, se crea la fuente XML a partir del documento y se crea el resultado en el fichero "empleados.xml". Se utiliza una instancia de *TransformerFactory* para realizar la transformación del documento a fichero.

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

El siguiente programa Java lee el fichero XML "empleados.xml" y lo visualiza en pantalla:

```
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());

            // crear una lista con todos los nodos empleado
            NodeList empleados = document.getElementsByTagName("empleado");
            System.out.println("Nodos empleado a recorrer: " + empleados.getLength());

            // recorrer la lista de empleados
            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.println("ID: " + getNodo("id", elemento));
                    System.out.println("Apellido: " + getNodo("apellido", elemento));
                    System.out.println("Departamento: " + getNodo("dep", elemento));
                    System.out.println("Salario: " + getNodo("salario", elemento));
                }
            }
        }
        catch (Exception e) { e.printStackTrace(); }
    } // fin de main

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

} // fin de la clase
```

En este programa Java se realizan los siguientes pasos:

1) Se crea una instancia de *DocumentBuilderFactoy* para construir el analizador sintáctico (parser) y se carga el documento XML con el método *parse().*

2) Se obtiene la lista de nodos con nombre *empleado* de todo el documento mediante el método *getElementsByTagName(String).*
3) Se realiza un bucle para recorrer esta lista de nodos *empleados*. Para cada nodo, se obtienen su etiqueta y su valor llamando a la función definida *getNodo(String, Element)*,y se muestran en pantalla.

8.2. ACCESO A DATOS CON SAX

SAX (Simple API for XML) es un conjunto de clases e interfaces que ofrecen una herramienta muy útil de programación para el procesamiento de documentos XML desde lenguajes de alto nivel. Dado que aborda el procesamiento de datos desde una óptica diferente a la de DOM, por lo general, SAX se usa cuando la información almacenada en un documento XML es clara, está bien estructurada y no se necesita hacer modificaciones.

Las principales características de SAX son:

- SAX ofrece una alternativa para leer documentos XML de manera secuencial. El documento solo se lee una vez: se recorre el secuencialmente el fichero hasta el final. A diferencia de DOM, el programador no se puede mover por el documento a su antojo.
- SAX no carga el documento en memoria, sino que lo lee directamente desde el fichero. Esto es especialmente útil cuando el fichero XML es muy grande implicando poco consumo de memoria pero no ofrece una visión global del documento.

SAX es más complejo de programar que DOM, es una API totalmente escrita en Java e incluida dentro del JRE que nos permite crear nuestro propio parser de XML.

Para trabajar con SAX en Java, se necesitan las clases e interfaces que componen el paquete **org.xml.sax**. Estas clases ofrecen métodos para leer secuencialmente documentos desde una fuente de datos (fichero, InputStream, etc.) y generar eventos relacionados. Contiene una clase fundamental: **InputSource**.

La lectura de un documento XML produce eventos que ocasiona la llamada a métodos, por lo que cuando SAX detecta un elemento propio de XML, entonces lanza un evento, que puede deberse a que:

- Se haya detectado el comienzo del documento XML. (***startDocument*()**)
- Se haya detectado el final del documento XML. ***(endDocument())***
- Se haya detectado una etiqueta de comienzo de un elemento. Por ejemplo <libro>. ***(startElement())***
- Se haya detectado una etiqueta de final de un elemento. Por ejemplo </libro>. ***(endElement())***
- Se haya detectado un atributo.
- Se haya detectado una cadena de caracteres que puede ser un texto. ***(characters())***
- Se haya detectado un error (en el documento, de Entrada/Salida, etc.).

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

Cuando SAX devuelve que ha detectado un evento, entonces dicho evento puede ser manejado con algún método de control (*callback*) de la clase **DefaultHandler**. Esta clase es la que se usa por defecto y contiene métodos que pueden ser sobrecargados por el programador. También se pueden implementar los *callbacks* de la interfaz **ContentHandler** para la gestión de dichos eventos.

Cuando SAX detecta un evento de error o un final de documento, entonces se termina el recorrido del fichero XML.

Las principales interfaces que necesita un programa Java que utiliza SAXson:

- **ContentHandler**. Recibe las notificaciones de los eventos que ocurren en el documento.
- **DTDHandler**. Recoge eventos relacionados con la DTD.
- **ErrorHandler**. Define métodos de tratamiento de errores.
- **EntityResolver**. Sus métodos llaman cada vez que se encuentra una referencia a una entidad.
- **DefaultHandler**. Clase que provee una implementación por defecto para todos sus métodos, en la que el programador definirá los métodos a usar en un programa. Esta es la clase que extenderemos para poder crear nuestro parser de XML (posteriormente en el ejemplo se llama GestionContenido). Por ejemplo:
  1) *startDocument*. Se invoca al comenzar el procesado del fichero XML.
  2) *endDocument*. Se invoca al finalizar el procesado del fichero XML.
  3) *startElement*. Se invoca al comenzar el procesado de una etiqueta XML. Aquí se leen los atributos de la etiqueta.
  4) *endElement*. Se invoca al finalizar el procesado de una etiqueta XML.
  5) *characters*. Se invoca cuando se encuentra una cadena de texto.

XMLReader.Realiza la lectura de un documento XML usando varios métodos (*setContentHandler(),setDTDHandler(),setEntityResolver()* y *setErrorHandler()*). Cada método trata un tipo de evento y está asociado con una interfaz determinada.

El siguiente programa Java lee el fichero XML “alumnos.xml” mediante la generación de eventos y su posterior tratamiento en los métodos definidos para ello.

```
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("\t Principio 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);
        car = car.replaceAll("[\t\n]","");  //quitar saltos de línea
        System.out.printf ("\t Caracteres: %s %n", car);
    }
}//fin GestionContenido
```

En este programa Java se realizan los siguientes pasos:

1) Se importan las clases e interfaces de SAX.
2) 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.
3) A continuación, hay que indicar al *XMLReader* qué objetos poseen los métodos que tratarán los eventos: setContentHandler(), setEntityResolver().... En nuestro caso, se ha definido la clase *GestionContenido*, en la que se tratan solo los eventos básicos: inicio y fin de documento, inicio y fin de etiqueta encontrada y cadena de caracteres encontrada. En este caso, se usa el método setContentHandler() para tratar los eventos que ocurren en el documento.
4) Luego se define el fichero XML "alumnos.xml" que se va a leer mediante un objeto *InputSource*.
5) Por último, se procesa dicho documento XML mediante el método *parse()* del objeto *XMLReader,* le pasamos un objeto InputSource.

El resultado de ejecutar el programa con el fichero *alumnos.xml* se puede observar cómo el orden de ocurrencias 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
```

8.3. CONVERSIÓN DE FICHEROS XML A OTRO FORMATO

XSL (Extensible Stylesheet Languaje) es toda una familia de recomendaciones del *World Wide Web Consortium* para expresar hojas de estilo del lenguaje xml. Una hoja de estilos XSL describe el proceso de presentación a través de un pequeño conjunto de elementos XML. Esta hoja, elementos de reglas que representan a las reglas de construcción y elementos de reglas de diseño que representan a las reglas de mezcla de estilos. Vamos a ver un ejemplo de cómo partir de un fichero XML que contiene datos y otro XSL que contiene la presentación de estos, se puede generar un fichero HTML usando el lenguaje Java.

Fichero de 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 de 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 tener 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.

Formación se considere llamando al método ***transform(Source fuenteXml, Result resultado)***, pasándole los datos (accediendo al fichero XML) y el stream de salida (el fichero HTML).

```
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
 }//de main
}//de la clase
```

El fichero HTML generado sería el siguiente:

LISTA DE ALUMNOS

Nombre	Edad
Juan	19
Maria	20

8.4. SERIALIZACION DE OBJETOS A XML

Ver cómo podemos se dializado de forma sencilla objeto java a xml y viceversa; para ello la librería XStream.

Para poder utilizarla y mousse de escapamos los JAR de ese sitio web: http://x-stream.github.io/download. A nuestro proyecto de eclipse debemos añadir los dos ficheros: xstream-1.4.10.jar y kxml2-2.3.0.jar.

Vamos a retomar el fichero *FichPersona.dat* que utilizamos anteriormente con objetos *Persona*. La convertiremos en un fichero de datos XML. Necesitaremos la clase *Persona* (ya definida) y *ListaPersonas* en la que se define una lista de objetos *Persona* que pasaremos al fichero XML.

La idea a continuación consiste en recorrer el fichero *FichPersona.dat* una lista de personas que después insertará en el fichero *Personas.xml*, veamos un ejemplo:

```
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
        //leer una Persona
        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);

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

        //quitar etiqueta lista (atributo de la claseListaPersonas)
        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
```

**Importaremos las librerías necesarias: `import java.io.*; import com.thoughtworks.xstream.XStream;`

En este programa Java se realizan los siguientes pasos:

1) Creamos una instacia de la clase Xstream para hacer uso de ella.
2) Las etiquetas XML se conrresponden con el nombre de los atributos de la clase, pero podemos cambiarlas usando el método ***alias(String alias, Class clase).*** En nuestro caso se ha dado un alias a las clase ListaPersonas, en XML será ListaPersonasMunicipios y de igual modo a Persona aparecerá como DatosPersona.

3) Con el método ***aliasField(String alias, Class clase, String nombrecampo),*** permite crear una alias para un nombre de campo y cambiaría la etiqueta correspondiente en el fichero XML. Por ejemplo el nombre o la edad de una Persona:

   xstream.aliasField("Nombre alumno", Persona.class, “nombre”);

   xstream.aliasField("Edad alumno", Persona.class, “edad”);

4) Utilizando el método ***addImplicitCollection(Class clase, String nombrecampo)*** hacemos que no aparezca el atributo lista de la clase *ListaPersonas* en el XML.
5) Finalmente, para generar Personas.xml a partir de la lista de objetos usamos ***toXML(Objeto objeto, OutputStream out)***

El fichero generado tendría es 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>
  <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>
```

Para realiza la lectura del fichero XML haremos:

```
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 se 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, utilizaremos el método **fromXML(InputStream input)** que devuelve un tipo Object.

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