

Unidad didáctica 1 Manejo de Ficheros

Ficheros.....	3
CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS. LA CLASE FILE.....	3
Constructores:.....	3
Métodos.....	4
Constantes:	6
FLUJOS O STREAMS. TIPOS	8
Flujos de Bytes (Byte Streams).....	8
Flujos de Caracteres (Character Streams).....	9
FORMAS DE ACCESO A UN FICHERO.....	10
Operaciones Sobre Ficheros	11
Operaciones sobre Ficheros Secuenciales	12
Operaciones sobre Ficheros Aleatorios	13
CLASES PARA GESTIÓN DE FLUJOS DE DATOS DESDE/HACIA FICHEROS.....	14
Ficheros de texto.....	14
Escritura en Ficheros de Texto	14
Lectura de un Fichero de Texto	19
Ficheros Binarios	21
Escritura de Datos Primitivos Java en un Fichero.....	21
Lectura de Tipos Primitivos en un Fichero	24
OBJETOS SERIALIZABLES	28
Escritura de Objetos en un Fichero	28
Lectura de Objetos de un Fichero	30
Ficheros de Acceso Aleatorio.....	37
TRABAJO CON FICHEROS XML	48
Acceso a Ficheros XML con DOM.....	49
Acceso a Ficheros XML con SAX.....	54
Serialización de Objetos XML.....	58
Conversión de Ficheros XML a otro formato	61
EXCEPCIONES: DETECCIÓN Y TRATAMIENTO	62
Captura de Excepciones	62
Especificar Excepciones.....	65

OBJETIVOS

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

Utilizar clases para la gestión de ficheros y directorios.

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

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

Utilizar clases para almacenar y recuperar informaciones almacenadas en ficheros XML.

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

Gestionar excepciones.

Unidad didáctica 1 MANEJO DE FICHEROS

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

Ficheros

Dispositivos de soporte magnético que se utilizan para guardar información de forma persistente de los programas java.

La información dentro de un fichero se guarda:

- Por **campos**: Son cada una de las variables miembro de un objeto.
- Por **registros**: Son cada uno de los objetos. Los registros son una agrupación de campos.

Por ejemplo, un fichero de empleados puede contener datos de los empleados de una empresa, un fichero de texto puede contener líneas de texto que se correspondan con las líneas de un papel.

La forma en que se agrupan los datos en el fichero depende completamente de la persona que los diseñe.

CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS. LA CLASE FILE

La clase `File` descende directamente de la clase `Object`. La clase `File` no se utiliza para transferir datos entre la aplicación y el disco, sino para obtener información sobre los ficheros y directorios de éste e incluso para la creación y eliminación de los mismos. La clase `File` representa un fichero (o directorio) del sistema de archivos, tiene múltiples métodos que nos van a permitir realizar todo tipo de operaciones con los ficheros.

Constructores:

`File(String ruta, String nombre)`: Crea un objeto de la clase que es un fichero. Se encuentra en la ruta que se indica en el primer argumento y cuyo nombre se pasa como segundo argumento.

- a) `File f = new File("directorio", "XXXXX.txt");`
- b) `String directorio = "C:\\AccesoDatos\\Ejercicios\\UD01";`
`File f = new File(directorio, "ejercicio01.txt");`

`File(File ruta, String nombre)`: Crea un objeto de la clase que es un fichero. El `path` se indica en el objeto `File` que se pasa en el primer argumento y cuyo nombre se pasa como cadena en el segundo argumento. En este caso el directorio indicado en `ruta` deberá existir.

- a) `File f = new File(new File ("directorio"), "XXXXX.txt");`
- b) `File dire = new File("directorio");`
`File f = new File(dire, "ejercicio01.txt");`

File(String rutaAbsoluta): Crea un objeto de la clase que es un fichero. La ruta o nombre se pasa como argumento.

```
File f = new File("C:\\directorio\\XXXXX.txt");
```

La creación de un objeto `File` no implica que exista el fichero o el directorio indicado en la ruta. Si éste no existe no se provoca una excepción, aunque tampoco será creado de forma implícita.

```
package UD01;
import java.io.File;
/*
 * Ejemplo que crea un objeto File con una ruta y un fichero inexistentes
 */
public class EjemploFile01 {
    public static void main(final String[] args) {
        File f = new File("XXXXX.txt");
    }
}
```

El siguiente ejemplo muestra la lista de ficheros en el directorio actual. Se utiliza el método **list()** que devuelve un array de Strings con los nombres de los ficheros y directorios contenidos en el directorio asociado al objeto **File**. Para indicar que estamos en el directorio actual creamos un objeto **File** y le pasamos el parámetro **“.”**;

```
package UD01;
import java.io.File;
/*
 * Ejemplo que muestra la lista de ficheros del directorio actual
 */
public class EjemploFile02 {
    public static void main(final String[] args) {
        System.out.println("Lista de ficheros del directorio actual:");
        File f = new File(".");
        String[] archivos = f.list();
        for(int i=0; i< archivos.length; i++){
            System.out.println(archivos[i]);
        }
    }
}
```

La siguiente declaración mostraría la lista de ficheros del directorio `c:\db`:

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

La siguiente declaración mostrará la lista de ficheros del directorio introducido desde la línea de comandos al ejecutar el programa.

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

Métodos

- Para crear físicamente el fichero o directorio indicado en el constructor de `File` utilizaremos los siguientes métodos

boolean createNewFile(). Crea el fichero cuyo nombre se indica en el constructor. Devuelve **true** si se ha podido crear el fichero, mientras que el resultado será **false** si ya existe y por tanto no ha sido creado.

Por ejemplo, si el fichero anterior XXXXX.txt no existe se podría ejecutar la siguiente instrucción para crearlo:

```
f.createNewFile();
```

La llamada a este método puede provocar una excepción **IOException** que habrá que capturar.

Si un objeto `File` hace referencia a un fichero inexistente y no se crea de forma explícita con `createNewFile()`, el fichero se creará de forma implícita cuando se vaya a utilizar el objeto `Writer` o `OutputStream` para realizar una operación de escritura sobre el fichero.

boolean mkdir(). Crea el directorio cuyo nombre se indica en el constructor. Devuelve **true** si se ha podido crear el directorio, mientras que el resultado será **false** si ya existe y por tanto no ha sido creado.

```
f.mkdir();
```

EJEMPLOS:

```
- File carpeta= new File("C:\\Ejemplos")
  carpeta.mkdir();
```

Crea una carpeta ejemplos dentro de la unidad C.

```
- File archivo= new File(carpeta, "Ejemplo1.txt")
  archivo.createNewFile();
```

Crea un fichero llamado ejemplo1 dentro de c:\Ejemplos.

```
- File archivo2 = new File("C:\\Ejemplos", "Ejemplo2.txt")
  archivo2.createNewFile();
```

Crea un fichero llamado ejemplo2 dentro de c:\Ejemplos.

```
- File archivo3= new File("C:\\Ejemplos\\Ejemplo2.txt")
  archivo3.createNewFile();
```

Crea un fichero llamado ejemplo2 dentro de c:\Ejemplos.

```
- File carpeta= new File("Ejemplos")
```

Al no poner ruta crea la carpeta donde se está ejecutando la aplicación.

boolean mkdirs(): Crear una carpeta. Devuelve **true** si se puede crear. En el caso de que la carpeta este dentro de un path absoluto que no exista en el sistema, se crea de manera completa.

Ejemplo: Con `mkdir` `C:\Nueva\Ejemplo` solo crearía Nueva y con `mkdirs` nueva y ejemplo.

- Para obtener información del objeto creado podemos utilizar los métodos de la clase `File`:

boolean canRead(): **True** si se puede hacer una operación de lectura en un fichero o carpeta.

boolean canWrite(): **True** si se puede hacer una operación de escritura en un fichero o carpeta.

canNewFile(): Permite crear en el sistema un fichero o carpeta que inicialmente está vacío.

canRead(): Devuelve **true** si el fichero se puede leer.

canWrite() : Devuelve **true** si el fichero se puede escribir.

boolean delete() : Se utiliza para eliminar un fichero o carpeta. Devuelve **true** si se puede eliminar.

deleteOnExit() : Permite eliminar un fichero o carpeta al finalizar la aplicación.

boolean equals() : **True** si son iguales los ficheros que se comparan.

boolean exists() : **True** si el fichero o carpeta existe.

getAbsolutePath() : Devuelve una cadena indicando la ruta completa donde se encuentra el archivo o carpeta.

getPath() : Devuelve una cadena indicando la ruta relativa donde se encuentra el archivo o carpeta.

getName() : Indica el nombre del fichero o carpeta.

getParent() : Devuelve el nombre del directorio padre, o **null** si no existe

isDirectory() : Devuelve **true** cuando el objeto `File` es una carpeta.

isFile() : Devuelve **true** cuando el objeto `File` es un fichero.

isHidden() : Devuelve **true** cuando el objeto `File` está oculto.

length() : Devuelve la longitud del fichero en bytes.

String []list() : Devuelve una array de tipo **String** con el nombre de todos los archivos y carpetas que haya dentro de otra carpeta.

String []list(filtro) : Devuelve una array de tipo **String** con el nombre de todos los archivos y carpetas con un determinado filtro (por ejemplo `"*.java"`).

renameTo() : Permite cambiar el nombre de un fichero o carpeta por el del objeto que se pasa como argumento. Devuelve **true** si lo ha cambiado.

setReadOnly() : Para establecer un fichero o carpeta de solo lectura.

Constantes:

pathSeparator: Devuelve un **String** indicando el carácter que se utiliza dentro del sistema para separar cada una de las cadenas de la variable `path`. En Windows es `;`.

pathSeparatorChar: Idem del anterior pero lo devuelve como carácter.

separator: Devuelve un **String** indicando el carácter que se utiliza dentro del sistema para establecer la ruta a los recursos. En Windows es `"\"`.

separatorChar: Idem del anterior pero lo devuelve como carácter.

```
package UD01;
import java.io.File;
/*
 * En el siguiente ejemplo mostramos todos los ficheros pdf que tenemos en la
 * carpeta CSDesarrolloAplicacionesMultiplataforma en Mis documentos.
 */
public class EjemploFile03 {
    public static void main(final String[] args) {
```

```

//separador almacena el tipo de separador utilizado en la plataforma, en
windows \
String separador= File.separator;

//En carpeta almaceno el path de la carpeta que quiero mirar sus subelementos
File carpeta= new File("c:" +separador+ "Users" +separador +"Usuario"
    +separador +"Mis documentos"
    +separador +"CSDAMultiplataforma");

System.out.println("Carpeta: " +carpeta);
//En elementos almaceno la matriz con todos los nombres de los archivos
// y carpetas dentro de la carpeta que le indiqué
String[] elementos = carpeta.list();

//Mostramos el número de carpetas dentro de la que le dije
System.out.println("Los ficheros .pdf son:\n");

for (int i=0; i<elementos.length; i++){
    if (elementos[i].endsWith(".pdf") || elementos[i].endsWith("PDF")){
        System.out.println("\t" +carpeta.getAbsolutePath() +separador
+elementos[i]);
    } // fin del if
} // fin de for
} // fin del main
} // fin de la clase

```

Ejemplo que muestra información del fichero **prueba.pdf**

```

package UD01;
import java.io.File;

// Ejemplo que muestra información del fichero EjemploFile01.java
public class EjemploFile04 {

    public static void main(final String[] args) {
        System.out.println("Información sobre el fichero\n");
        String separador= File.separator;

        //En carpeta almaceno el path de la carpeta que quiero mirar sus subelementos
        File f= new File("c:" +separador+ "Users" +separador +"Usuario"
            +separador +"Mis documentos" +separador +"CSDAMultiplataforma"
+separador
            +"BBDD" +separador +"EjModeloRelacional" +"prueba.pdf");

        System.out.println("Nombre del fichero: " +f.getName());
        System.out.println("Ruta          : " +f.getPath());
        System.out.println("Ruta absoluta   : " +f.getAbsolutePath());
        System.out.println("Ruta anterior  : " +f.getParent());
        System.out.println("Se puede escribir : " +f.canWrite());
        System.out.println("Se puede leer   : " +f.canRead());
        System.out.println("Tamaño         : " +f.length());
        System.out.println("Es un directorio : " +f.isDirectory());
        System.out.println("Se puede fichero : " +f.isFile());
    }
}

```

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

Para borrar un fichero o un directorio usamos el método `delete()`, para borrar un directorio deberá estar vacío.

```
if(f2.delete()){
    System.out.println("Fichero2 borrado correctamente");
}
else
    System.out.println("Fichero2 no se ha podido borrar");
```

FLUJOS O STREAMS. TIPOS

El sistema de entrada/salida en Java tiene varias clases que se implementan en el paquete **java.io**. Usa la abstracción del flujo (**stream**) para tratar la comunicación de información entre una fuente y un destino; dicha información puede estar en un fichero en el disco duro, en la memoria, en algún lugar de la red, e incluso en otro programa. Cualquier programa que tenga que obtener información de cualquier fuente necesita abrir un stream, de la misma forma si necesita enviar información abrirá un stream y se escribirá la información en serie. La vinculación de este stream al dispositivo físico la hace el sistema de entrada y salida de Java.

Se definen dos tipos de flujos:

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

Ejemplos de flujos de entrada serían cuando utilizamos la clase `BufferedReader`.

Flujos de Bytes (Byte Streams)

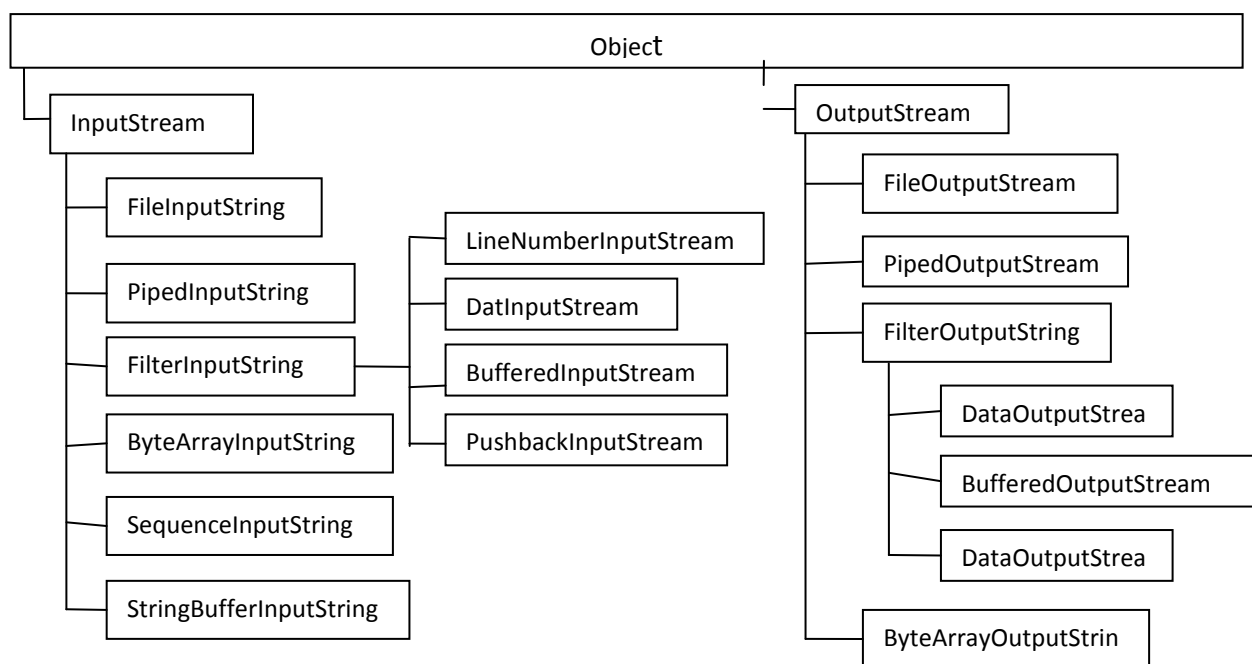
La clase **InputStream** representa las clases que producen entradas de distintas fuentes, estas fuentes pueden ser: un array de bytes, un objeto `String`, un fichero, una “tubería” (se ponen los elementos en un extremo y salen por el otro), una secuencia de otros flujos, otras fuentes como una conexión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

CLASE	FUNCIÓN
<code>ByteArrayInputStream</code>	Permite usar un espacio de almacenamiento intermedio de memoria.
<code>StringBufferInputStream</code>	Convierte un <code>String</code> en un <code>InputStream</code> .
<code>FileInputStream</code>	Para leer información de un fichero.
<code>PipedInputStream</code>	Implementa el concepto de “tubería”.
<code>FilterInputStream</code>	Proporciona funcionalidad útil a otras clases <code>InputStream</code> .
<code>SequenceInputStream</code>	Convierte dos o más objetos <code>InputStream</code> en un <code>InputStream</code> único.

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

CLASE	FUNCIÓN
ByteArrayOutputStream	Crea un espacio de almacenamiento intermedio en memoria. Todos los datos que se envían al flujo se ubican en este espacio.
FileOutputStream	Para enviar información a un fichero.
PipedOutputStream	Cualquier información que se desee escribir aquí acaba automáticamente como entrada del PipedInputStream asociado. Implementa el concepto de “tubería”.
FilterOutputStream	Proporciona funcionalidad útil a otras clases OutputStream.

Jerarquía de clases para lectura y escritura de flujos de datos: InputStream y OutputStream



Las clases **FileInputStream** y **FileOutputStream** manipulan los flujos de bytes provenientes o dirigidos hacia los ficheros en disco.

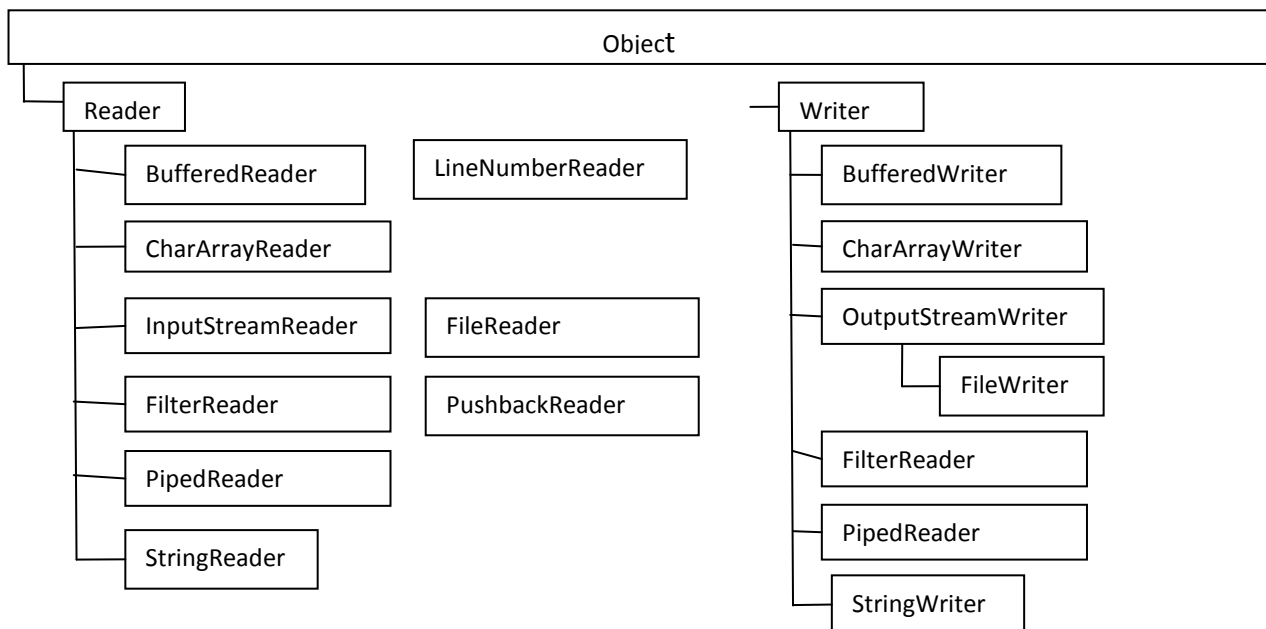
Flujos de Caracteres (Character Streams)

Las clases **Reader** y **Writer** manejan flujos de caracteres Unicode. En ocasiones hay que usar las clases que manejan bytes en combinación con las clases que manejan caracteres. Para lograr esto hay clases “puente” (es decir, convierte los streams de bytes a streams de caracteres): **InputStreamReader** convierte un **InputStream** en un **Reader** (lee bytes y los convierte en caracteres) y **OutputStreamWriter** que convierte **OutputStream** en un **Writer**

La siguiente tabla muestra la correspondencia entre las clases de flujo 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

Jerarquía de las clases para lectura y escritura: Reader, Writer



Las clases de flujos de caracteres más importantes son:

Para acceso a ficheros, lectura y escritura en ficheros: FileReader y FileWriter.

Para acceso a caracteres, leen y escribe un flujo de caracteres en un array de caracteres: CharArrayReader y CharArrarWriter.

Para buferización de datos: BufferedReader y BufferedWriter, se utilizan para evitar que cada lectura o escritura acceda directamente al fichero, ya que utilizan un buffer intermedio entre la memoria y el stream.

FORMAS DE ACCESO A UN FICHERO

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

- **Acceso secuencial:** los datos o registros se leen y se escriben en orden. Ejemplo cintas de vídeo, cintas de magnetofón. Si se quiere acceder a un dato o a un registro hay que leer todos los anteriores. La escritura de datos se hace a partir del último registro, no es posible hacer inserciones entre los datos que ya están 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 un tamaño conocido, nos podemos mover de un registro a otra de forma aleatoria para leerlos o modificarlos.

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

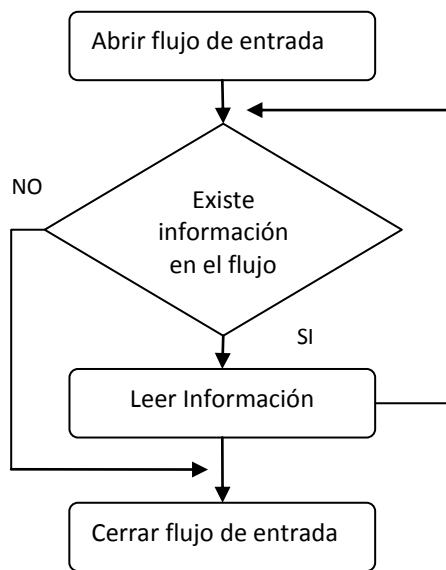
Operaciones Sobre Ficheros

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

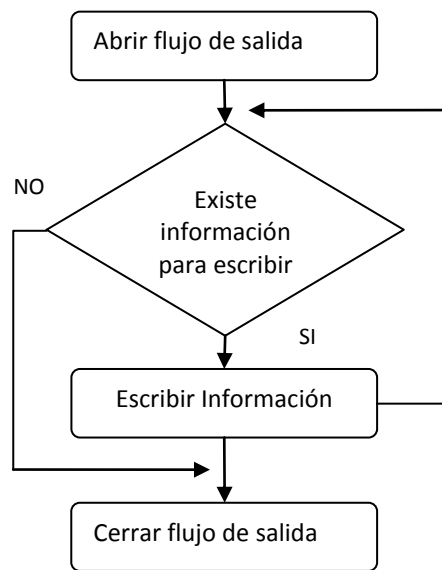
- **Creación del fichero.** El fichero se crea en el disco con un nombre que después de debe utilizar para acceder a él. La creación es un proceso que se realiza una vez.
- **Apertura del fichero.** Para que un programa pueda operar con un fichero, la primera operación que hay que realizar es la apertura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

Los programas cuando quieren leer datos de un fichero, lo primero que hacen es abrir un flujo de entrada y leen la información que contiene el fichero mediante el flujo de datos de entrada. Para grabar datos la operación es similar se abre un flujo de salida y el programa va escribiendo los datos en el flujo de salida y de esta forma se almacenan los datos.

ALGORITMO DE LECTURA DE FICHERO



ALGORITMO DE ESCRITURA DE FICHERO



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

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

Operaciones sobre Ficheros Secuenciales

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

Veamos como se realizan las operaciones típicas:

- **Consultas:** para consultar un determinado registro es necesario empezar la lectura desde el primer registro, y continuar leyendo secuencialmente hasta localizar el registro buscado. Por ejemplo, si el registro a buscar es el 90 dentro del fichero, será necesario leer secuencialmente los 89 anteriores.
- **Altas:** en un fichero secuencial las altas se realizan al final del último registro insertado, es decir, solo se permite añadir datos al final del fichero.
- **Bajas:** para dar de baja un registro de un fichero secuencial es necesario leer todos los registros uno a uno y escribirlos en un fichero auxiliar, salvo el que deseamos dar de baja.

Una vez reescritos hay que borrar el fichero inicial y renombrar el fichero auxiliar dándole el nombre del fichero original.

- **Modificaciones:** consiste en localizar el registro a modificar, efectuar la modificación y reescribir el fichero inicial en otro fichero auxiliar que incluya el registro modificado. El proceso es similar a las bajas.

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

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

Operaciones sobre Ficheros Aleatorios

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

Normalmente para posicionarse 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 identifican de forma unívoca a un registro). Por ejemplo, disponemos de un fichero de empleados con 3 campos: identificador, apellido y salario. Usamos el identificador como campo clave y le damos el valor 1 para el primer empleado, 2 para el segundo y así sucesivamente; para localizar al empleado con identificador X necesitamos acceder a la posición tamaño * (X-1) para acceder a los datos de dicho empleado.

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

Veamos como se realizan las operaciones típicas:

- **Consultas:** para consultar un determinado registro necesitamos saber su clave, aplicar la función de conversión a la clave para obtener la dirección y leer el registro ubicado en esa posición. Habría que comprobar si el registro buscado está en esta posición, si no está se buscará 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 se insertará en la zona de excedentes.
- **Bajas:** las bajas suelen realizarse de forma lógica, es decir, se suele utilizar un campo del registro a modo switch que tenga el valor 1 cuando el registro exista y le damos el valor 0

para darlo de baja, físicamente el registro no desaparece del disco. Habría que localizar el registro a dar de baja a partir del campo clave y reescribir en este campo el valor 0.

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

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

CLASES PARA GESTIÓN DE FLUJOS DE DATOS DESDE/HACIA FICHEROS

En java podemos utilizar dos tipos de ficheros:

- **Modo Texto:** en este caso los datos son almacenados usando código ASCII y por tanto son plenamente visibles usando cualquier editor.
- **Modo Binario:** en este caso los datos son almacenados en notación hexadecimal y por tanto se ocupa un editor binario para reconocerlos sin embargo un archivo binario es más compacto que un archivo texto.

En ambos casos el acceso a los mismos se puede realizar de forma secuencial o aleatoria.

Ficheros de texto

Los ficheros de texto normalmente se generan con un editor, almacenan caracteres alfanuméricos en un formato estándar (ASCII, UNICODE, UTF8, etc.). Utilizan las clases **FileReader** para leer caracteres y **FileWriter** para escribir los caracteres en el fichero.

Cuando trabajamos con ficheros, cada vez que leemos o escribimos en uno debemos hacerlo dentro de un manejador de excepciones **try-catch**. Al usar la clase **FileReader** se puede generar la excepción **FileNotFoundException** (porque el nombre del fichero no existe o no es válido) y al usar la clase **FileWriter** la excepción **IOException** (el disco está lleno o protegido contra escritura).

Escritura en Ficheros de Texto

Para escribir cadenas de caracteres en un fichero el paquete java.io proporciona las clases **FileWriter**, **BufferedWriter** y **PrintWriter**.

La clase **FileWriter** proporciona los siguientes métodos para escritura:

Método	Acción
void write(int c)	Escribe un carácter
void write(char[] buf)	Escribe un array de caracteres
void write(char[] buf, int desplazamiento, int n)	Escribe n caracteres de datos en la matriz buf y comenzando por buf [desplazamiento]
void write(String str)	Escribe una cadena de caracteres
append(char c)	Añade un carácter a un fichero

El primer paso para poder escribir en un fichero de texto, es crear un objeto **FileWriter** que nos permita tener acceso al fichero en modo escritura. Para ello utilizaremos unos de los siguientes **constructores**:

```
FileWriter(String path);  
    FileWriter fw = new FileWriter("Nombres.txt");  
FileWriter(File fichero);  
    File f = new File("Nombres.txt");  
    FileWriter fw = new FileWriter(f);  
FileWriter(String path, boolean append);  
    FileWriter fw = new FileWriter("Nombres.txt", true);  
FileWriter(File fichero, boolean append);  
    File f = new File("Nombres.txt");  
    FileWriter fw = new FileWriter(f, true);
```

Se puede construir un objeto **FileWriter** proporcionando la ruta del fichero directamente o a partir de un objeto **File** existente

El parámetro `append` permite indicar si los datos que se van a escribir se añadirán a los existentes (`true`) o sobrescribirán a éstos (`false`). Si se utiliza uno de los dos primeros constructores, los datos escritos en el fichero sustituirán a los existentes, es equivalente a `append false`.

La clase **FileWriter** proporciona el método `write()` que permite escribir en el fichero la cadena de caracteres pasada como parámetro, aunque también se puede utilizar la clase estándar de escritura **PrintWriter** para realizar esta operación.

El segundo paso es crear un objeto **PrintWriter** que nos permite que la escritura en un fichero se realice de la misma forma que la escritura en pantalla. La diferencia está en que en el caso de la pantalla el objeto **PrintWriter** está asociado a **System.out**, mientras que para un fichero de texto habrá que construir el objeto **PrintWriter** a partir del objeto **FileWriter**, o bien utilizar un método de la clase **FileWriter**.

```
FileWriter fw = new FileWriter("datos.txt");  
PrintWriter salida = new PrintWriter(fw);  
Salida.println("Hola");
```

Desde la versión 5 de Java, también es posible crear un objeto **PrintWriter** a partir de un objeto **File** o incluso de la ruta del fichero, por lo que las instrucciones anteriores podrían reducirse a una.

```
PrintWriter salida = new PrintWriter("datos.txt");
```

Una vez creado el objeto **PrintWriter**, podemos utilizar los métodos `print()`, `println()` y `printf()` para escribir en el fichero.

Ejemplo: El programa almacenará unos nombres de un en un fichero en disco utilizando **PrintWriter**.

```
package ud01;  
  
import java.io.FileWriter;
```

```

import java.io.IOException;
/*
 * Ejemplo de utilización de las clase FileWriter y PrintWriter
 */
import java.io.PrintWriter;

public class Ejemplo01FileWriter {
    public static void main(String[] args) throws IOException {
        //array de nombres
        String [] nombres = {"María", "Ana", "Santiago", "Jorge", "Iciar",
"Isabel"};

        // File f = new File("Nombres.txt");
        // FileWriter fw = new FileWriter(f);
        FileWriter fw = new FileWriter("Nombres.txt");
        PrintWriter salida = new PrintWriter(fw);
        // las dos sentencias anteriores son equivalentes a la siguiente
        //PrintWriter salida = new PrintWriter("Nombres.txt");

        for (int i = 0; i < nombres.length; i++) {
            salida.println(nombres[i]);
        }
        salida.flush();
        salida.close();
    }
}

```

La llamada al método **flush()** garantiza que todos los datos enviados a través del buffer de salida han sido escritos en el fichero y el método **close()** cierra la conexión con el fichero y libera los recursos utilizados por ésta.

El siguiente ejemplo escribe caracteres en un fichero de nombre FichTexto01.txt (si no existe lo crea). Los caracteres se escriben uno a uno y se obtienen de un String. Utilizando el método **write()**.

```

package ud01;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

/*
 * escribe caracteres en un fichero de nombre FichTexto01.txt (si no existe lo crea).
 * Los caracteres se escriben uno a uno y se obtienen de un String
 */
public class Ejemplo02FileWriter {

    public static void main(final String[] args) throws IOException {
        //En carpeta almaceno el path de la carpeta que quiero mirar sus
subelementos
        File f= new File("c:\\Users\\Usuario\\Mis
documentos\\CSDAMultiplataforma\\AccesoDatosActual" +
"\\FicherosTexto\\FichTexto01.txt");
        FileWriter fw = new FileWriter(f); // crea el fichero de salida

        String cadena = "Esto es una prueba de FileWriter método write";
        char[] cad = cadena.toCharArray(); // convierte un String en un array de
caracteres
    }
}

```



```

        for(int i=0; i< cad.length; i++)
            fw.write(cad[i]); // se escribe un caracter

        fw.append('*'); // añade un * al final
        fw.close(); // cierra el fichero
    }
}

```

En lugar de escribir los caracteres uno a uno, también se pueden escribir todo el array: **fw.write(cad)**.

El siguiente ejemplo escribe cadenas de caracteres que se obtienen de un array de String, las cadenas se graban una a continuación de la otra sin saltos de línea.

```

package ud01;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

/*
 * Escribe cadenas de caracteres que se obtiene de un array de String, las cadenas se
graban una a continuación de la otra sin saltos de línea
 */
public class Ejemplo03FileWriter {

    public static void main(final String[] args) {
        String provincias[] = {"La Coruña", "Lugo", "Orense", "Pontevedra",
"Guipúzcoa", "Vizcaya", "Alava"};
        //En carpeta almaceno el path de la carpeta que quiero mirar sus
subelementos

        try{
            File f= new File("c:\\Users\\Usuario\\Mis
documentos\\CSDAMultiplataforma\\AccesoDatosActual" +
"\\FicherosTexto\\Provincias.txt");
            FileWriter fw = new FileWriter(f); // crea el fichero de salida

            for (int i=0; i< provincias.length; i++)
                fw.write(provincias[i]);
            fw.close();
        }catch(IOException ioe){
            System.out.println("Disco lleno o protegido");
        }
    }
}

```

Ejemplo de cómo utilizar el desplazamiento y el numero de caracteres en el método **write()**.

```

package ud01;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

/*
 * Escribe cadenas de caracteres que se obtiene de un array de String, las cadenas se
graban desde un * desplazamiento 10 caracteres
 */
public class Ejemplo04FileWriter {

```

```

    public static void main(final String[] args) {
        String cadena = "Probando el desplazamiento y el numero de caracteres a
escribir";
        //En carpeta almaceno el path de la carpeta que quiero mirar sus
subelementos

        try{
            File f= new File("c:\\Users\\Usuario\\Mis
documentos\\CSDAMultiplataforma\\AccesoDatosActual" +
                "\\FicherosTexto\\FichTexto02.txt");
            FileWriter fw = new FileWriter(f); // crea el fichero de salida

            fw.write(cadena, 3, 10);
            fw.close();
        }catch(IOException ioe){
            System.out.println("Disco lleno o protegido");
        }
    }
}

```

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

```

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

```

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

```

package ud01;

import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;

/*
 * El siguiente ejemplo escribe 10 filas de caracteres en un fichero de texto y después
de escribir
 * cada fila salta una línea con el método newLine().
 */
public class Ejemplo05FileWriter {

    public static void main(final String[] args) {
        try{
            BufferedWriter bw = new BufferedWriter(new FileWriter("Filas.txt"));
            for(int i=0; i<11; i++){
                bw.write("Fila número: " +i);
                bw.newLine();
            }
            bw.close();
        }catch(FileNotFoundException fne){
            System.out.println("No se encuentra el fichero");
        }catch(IOException ioe){
            System.out.println("Error de L/E");
        }
    }
}

```

Lectura de un Fichero de Texto

Para recuperar cadenas de caracteres de un fichero el paquete `java.io` proporciona las clases **`FileReader`** y **`BufferedReader`**.

El primer paso para recuperar información de un fichero de texto, es crear un objeto `FileReader` asociado al mismo. Un objeto `FileReader` representa un fichero de texto “abierto” para la lectura de datos. Este objeto es capaz de adaptar la información recuperada del fichero a las características de una aplicación Java, transformando los bytes almacenados en el mismo en caracteres Unicode.

Constructores

```
FileReader(String path); // construye un objeto
proporcionándole directamente la ruta

FileReader(File fichero); // construye un objeto a partir de un
objeto File existente
```

Ejemplo: crea un objeto `FileReader` haciendo referencia al fichero `datos.txt`

```
File f = new File("datos.txt");
FileReader fr = new FileReader(f);
```

La clase `FileReader` proporciona el método `read()` para la lectura de la información almacenada en un fichero. Se utiliza poco, porque recupera la información como byte y posteriormente hay que convertirla en `String`, por lo que se utiliza más la clase `BufferedReader` para leer, utilizando el objeto `FileReader` como puente para crear un objeto de este tipo.

El segundo paso consiste en crear un `BufferedReader`, para lo que se utiliza el constructor:

```
BufferedReader(Reader entrada);
```

Para leer información del disco el procedimiento es igual que cuando lo hacemos desde el teclado, sólo que en este caso el objeto `Reader` no es `System.in`, sino que será el objeto `FileReader` asociado al fichero.

```
BufferedReader br = new BufferedReader(fr);
```

Una vez creado el objeto se puede utilizar el método `readLine()` para leer líneas de texto del fichero de forma similar a como leemos del teclado. En el caso de los ficheros, como pueden estar formados por más de una línea, será necesario utilizar un bucle `while` para recuperar todas las líneas de texto del mismo de forma secuencial.

```
package ud01;
```

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
```

```
/*
 * Se va a mostrar por pantalla el contenido del fichero Nombres.txt
 */
```

```
public class EjemploFileReader01 {
```

```

public static void main(final String[] args) {
    try{
        File f = new File("Nombres.txt");
        if(f.exists()){
            FileReader fr = new FileReader(f);
            BufferedReader br = new BufferedReader(fr);
            String nombre; // variable donde se recupera la informacion

            while((nombre = br.readLine())!= null){
                System.out.println(nombre);
            }
        }
        catch(FileNotFoundException fn){
            System.out.println("No se encuentra el fichero");
        }
        catch(IOException ioe){
            System.out.println("Error de L/E");
        }
    }
}

```

El método `readLine()` apunta a la siguiente línea de texto después de recuperar la línea actual. Cuando no existen más líneas para leer, la llamada a `readLine()` devuelve `null`.

Si lo que se quiere es leer carácter a carácter en lugar de línea a línea se utiliza el método `read()` en lugar de `readLine()`. El método `read()` devuelve un entero que representa el código Unicode del carácter leído, siendo el resultado `-1` si no hay más caracteres para leer.

```

package ud01;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class EjemploFileReader02 {

    public static void main(final String[] args) throws IOException {
        File f = new File("Nombres.txt");
        if(f.exists()){
            FileReader fr = new FileReader(f);
            BufferedReader br = new BufferedReader(fr);
            int caracter; // variable qdon se recupera la informacion

            while((caracter = br.read())!= -1){
                System.out.println(caracter + "\t" +(char)caracter);
            }
        }
        else
            System.out.println("El fichero no existe");
    }
}

```

Los métodos que proporciona la clase **FileReader** devuelven el número de caracteres leídos o `-1` si se ha llegado al final del fichero. Son los siguientes:

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

Es posible realizar la lectura de los datos almacenados en un fichero de texto utilizando la clase `Scanner`, para ello se pasará el objeto `File` asociado al fichero al constructor de `Scanner`, y después se utilizarán los métodos de esta clase.

```
package ud01;

import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;

public class EjemploFileReader03 {

    public static void main(final String[] args) throws IOException {
        File f = new File("Nombres.txt");
        if(f.exists()){
            FileReader fr = new FileReader(f);
            Scanner sc = new Scanner(fr);

            while(sc.hasNext()){
                System.out.println(sc.next());
            }
            sc.close();
        }
        else
            System.out.println("El fichero no existe");
    }
}
```

Ficheros Binarios

Los ficheros binarios almacenan secuencias de dígitos binarios que nos son legibles directamente por el usuario como ocurría con los ficheros de texto. Tienen la ventaja de que ocupan menos espacio en disco. En Java, las dos clases que nos permiten trabajar con ficheros son `FileInputStream` (para entrada) y `FileOutputStream` (para salida), estas trabajan con flujos de bytes y crean un enlace entre el flujo de bytes y el fichero.

Escritura de Datos Primitivos Java en un Fichero

Para almacenar datos en un fichero en forma de tipos básicos Java, el paquete `java.io` proporciona las clases `FileOutputStream` y `DataOutputStream`.

El procedimiento es similar al empleado para la clase `FileWriter`, el primer paso es crear un objeto `FileOutputStream` que permita añadir información al fichero o sobrescribirla, para ello utilizamos los **constructores**:

```
FileOutputStream(File fichero, boolean append);
FileOutputStream(String path, boolean append);
```

La clase *FileOutputStream* es una subclase de *OutputStream*, que representa un stream o flujo de salida para la escritura de bytes.

El segundo paso es a partir del objeto `FileOutputStream` crear un objeto `DataOutputStream` para realizar la escritura. El **constructor** es:

```
DataOutputStream (Output Stream);
```

La clase `DataOutputStream` proporciona métodos para escribir datos en un fichero en cada uno de los ocho **tipos primitivos Java**. Estos métodos tienen el formato:

```
void writeXnn(xxx dato);
```

siendo xxx el nombre del tipo primitivo Java.

Métodos para Escritura
void writeBoolean(boolean v);
void writeByte(int v);
void writeBytes(String s);
void writeShort(int v);
void writeChars(String s);
void writeChar(int v);
void writeInt(int v);
void writeLong(long v);
void writeFloat(float v);
void writeDouble(double v);
void writeUTF(String str);

Ejemplo escritura en un fichero de un array de enteros.

```
package ud01;
import java.io.DataOutputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

/*
 * // Ejemplo escritura en un fichero de un array de enteros.
 */
public class EjemploFileOutputStream01 {

    public static void main(final String[] args) {
        FileOutputStream fEscritura;
        DataOutputStream ds = null;
        try{

            // array de enteros
```

```

    int [] array = {5, 18, 23, 12, 10, 1, 47};

    fEscritura = new FileOutputStream("Enteros.txt", false);
    ds = new DataOutputStream(fEscritura);
    // mientras el array tenga elementos los escribimos en el fichero
        for (int i : array) {
            ds.writeInt(i);
        }
    } catch (FileNotFoundException e){
        System.out.println("No se pudo abrir el fichero Enteros.txt");
    }

    catch (IOException e){
        System.out.println("No se pudo escribir en el fichero Enteros.txt");
    }

    finally{
        try{
            ds.close();
        }
        catch (IOException e){
            System.out.println("No se pudo cerrar el fichero
Enteros.txt");
        }
    }
}

```

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

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

```

package ud01;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class EjemploFileOutputStream02 {

    public static void main(final String[] args) {
        FileOutputStream fEscritura = null;

        try{
            // array de enteros
            int [] array = {55, 185, 237, 142, 150, 21, 487};

            fEscritura = new FileOutputStream("Enteros01.txt", false);

```

```

        // mientras el array tenga elementos los escribimos en el fichero
        for (int i : array) {
            fEscritura.write(i);
        }
    } catch (FileNotFoundException e) {
        System.out.println("No se pudo abrir el fichero Enteros.txt");
    }

    catch (IOException e) {
        System.out.println("No se pudo escribir en el fichero Enteros.txt");
    }

    finally {
        try {
            fEscritura.close();
        }
        catch (IOException e) {
            System.out.println("No se pudo cerrar el fichero
Enteros.txt");
        }
    }
}

```

Lectura de Tipos Primitivos en un Fichero

Para leer datos de un fichero en forma de tipos básicos Java, el paquete `java.io` proporciona las clases `FileInputStream` y `DataInputStream`.

El procedimiento es similar al empleado para la clase `FileReader`, el primer paso es crear un objeto `FileInputStream` que permita recuperar información del fichero, para ello utilizamos los **constructores**:

```

FileInputStream(File fichero);
FileInputStream (String path);

```

La clase `FileInputStream` es una subclase de `InputStream`, que representa u stream o flujo de entrada para la lectura de bytes.

El segundo paso es a partir del objeto `FileInputStream` crear un objeto `DataInputStream` para realizar la escritura. El **constructor** es:

```

DataInputStream(Input Stream);

```

La clase `DataInputStream` proporciona métodos para leer datos desde un fichero en cada uno de los ocho tipos primitivos Java. Estos métodos tienen el formato:

```

void readXnn(xxx dato);

```

siendo `xxx` el nombre del tipo primitivo Java.

Métodos para Lectura
boolean readBoolean();
byte readByte();
int readUnsignedByte();
int readUnsignedShort();
short readShort();
char readChar();
int readInt();
long readLong();
float readFloat();
double readDouble();
String readUTF();

Hay una excepción `EOFException` cuando intentamos leer de un fichero que ya no tiene información.

```
package ud01;

import java.io.DataInputStream;
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class EjemploFileInputStream01 {
    public static void main(final String[] args) {
        FileInputStream fLectura;
        DataInputStream ds;
        try{
            fLectura = new FileInputStream("Enteros.txt");
            ds = new DataInputStream(fLectura);
        }
        catch (FileNotFoundException e){
            System.out.println("ERROR GRAVE: El fichero
Enteros.txt no está disponible");
            return;
        }
        try{
            //bucle infinito
            while(true)
                System.out.print(ds.readInt() + " " );
        }
        catch (EOFException e){
        }catch(IOException ioe){
            System.out.print("Error no se ha podido leer de
Enteros.txt");
        }
        finally{
            try{
                ds.close();
            }
            catch(IOException e){
                System.out.print("Error no se ha podido cerrar
Enteros.txt");
            }
        }
    }
}
```

```

    }
}
}
}

```

Los métodos que proporciona la clase **FileInputStream** para lectura son:

Método	Función
void read(int b)	lee un byte y lo devuelve
void read(byte[] b)	Lee hasta b.length bytes de datos de una matriz de bytes
void read (byte[] b, int desplazamiento, int n)	Lee hasta n bytes a partir de la matriz b comenzando por b[desplazamiento] y devuelve el número de bytes leídos

Ejemplo:

```

package ud01;

import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.EOFException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
public class EjemploFileInputOutputStream01 {
    public static void main(final String[] args) {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        int opcion = 0;
        do{
            try{
                System.out.println("1.- Escribir en el fichero");
                System.out.println("2.- Leer del fichero");
                System.out.println("3.- Salir");
                System.out.println("Elegir opcion: ");
                opcion = Integer.parseInt(br.readLine());

                switch (opcion){
                    case 1:
                        EscrituraDisco();
                        break;
                    case 2:
                        LecturaDisco();
                        break;
                }
            }catch(NumberFormatException e){
                System.out.println("Error al introducir la opcion");
            }catch(IOException ioe){
                System.out.println("Error al introducir la opcion");
            }
        }
    }
}

```

```

        }while(opcion != 3);
    }
    static void EscrituraDisco(){
        File archivo = null;
        DataOutputStream flujoescritura = null;

        String respuesta="";
        System.out.println("Introduce el nombre del fichero de claves desde el
teclado") ;

        try{
            String nombrefich= new BufferedReader (new InputStreamReader
(System.in)).readLine().trim();
            archivo= new File (nombrefich) ;
            if (archivo.exists()){
                System.out.println ("El fichero ya existe ¿Desea
sobreescribirlo (SI/NO)?");
                respuesta= new BufferedReader (new InputStreamReader
(System.in)).readLine().trim();
            }
            //compareToIgnoreCase es que compare sin distinguir mayúsculas y
minúsculas
            if (respuesta.compareToIgnoreCase("si")==0){
                flujoescritura = new DataOutputStream ( new
FileOutputStream(nombrefich));
            }else{
                flujoescritura= new DataOutputStream (new
FileOutputStream(nombrefich,true));
            }

            System.out.println ("Introduce el nombre del usuario por teclado:
");
            String nombre= new BufferedReader (new InputStreamReader
(System.in)).readLine().trim();
            System.out.println ("Introduce el código del usuario por teclado:
");
            int codigo = Integer.parseInt(new BufferedReader (new
InputStreamReader (System.in)).readLine().trim());

            flujoescritura.writeInt(codigo);
            flujoescritura.writeUTF(nombre);
        }catch (IOException ioe){
            System.out.println("No se ha podido escribir la información en el
fichero " +archivo.getName());
        }finally{
            try{
                if (flujoescritura != null){
                    flujoescritura.close();
                }
            }catch (IOException ioe){
                System.out.println("No se ha podido cerrar correctamente el
flujo del fichero " + archivo.getName());
            }
        }
    }
    static void LecturaDisco(){
        File archivo = null;
        DataInputStream flujolectura =null;

```

```

        System.out.println("Introduce el nombre del fichero por teclado") ;

        try{
            archivo= new File (new BufferedReader (new InputStreamReader
(System.in)).readLine().trim());
            flujolectura= new DataInputStream (new FileInputStream(archivo));
            //Ojo el orden al leer tiene que ser el mismo que al escribir

            while (true){
                System.out.println ("Codigo de Usuario: "+
flujolectura.readInt());
                System.out.println ("Nombre de Usuario: "+
flujolectura.readUTF());
            }
        }catch (FileNotFoundException fnf){
            System.out.println("No se ha podido encontrar el fichero "
+archivo.getName());
        }catch (EOFException eof){
        }catch (IOException ioe){
            System.out.println("No se ha podido leer la información del fichero
" +archivo.getName());
        }finally{
            try{
                if (flujolectura!=null){
                    flujolectura.close();
                }
            }catch (IOException ioe){
                System.out.println("No se ha podido cerrar el flujo del
fichero " + archivo.getName());
            }
        } //fin finally
    } //fin main
} // fin clase

```

OBJETOS SERIALIZABLES

Hemos estudiado como se guardan los tipos primitivos en un fichero, pero por ejemplo, tenemos un objeto de tipo empleado con varios atributos (nombre, dirección, salario, departamento, etc.) y queremos guardarlo en un fichero tendríamos que guardar cada atributo que forma parte del objeto por separado, esto es engorroso si tenemos varios objetos. Por ello Java nos permite guardar objetos en ficheros binarios para poder hacerlo, el objeto tiene que implementar la interfaz `Serializable` que dispone de una serie de métodos con los que podemos guardar y leer objetos en ficheros binarios.

Escritura de Objetos en un Fichero

Para que un objeto pueda ser almacenado en disco, es necesario que la clase a la que pertenece sea **serializable**. Esta característica la poseen todas las clases que implementan la interfaz `java.io.Serializable`.

La **serialización** es el proceso por el cual un objeto o una colección de objetos se convierten en una serie de bytes que pueden ser almacenados en un fichero y recuperado posteriormente para su uso. Se dice que el objeto u objetos tienen **persistencia**. Cuando serializamos un objeto, almacenamos la estructura de la clase y todos los objetos referenciados por esta.

La interfaz **Serializable** no contiene ningún método, basta con que una clase la implemente para que sus objetos puedan ser serializados por la máquina virtual y por lo tanto almacenada en disco.

Ejemplo: Crea una clase Persona cuyos objetos encapsulan el nombre y la edad de una persona. Estos objetos pueden ser transferidos a disco, ya que Persona implementa la interfaz **Serializable**.

```
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() {
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public int getEdad() {
        return edad;
    }
    public void setEdad(int edad) {
        this.edad = edad;
    }
}
```

El primer paso es crear un objeto `FileOutputStream` que permita añadir información al fichero o sobrescribirla, para ello utilizamos los **constructores**:

```
FileOutputStream(File fichero, boolean append);
FileOutputStream(String path, boolean append);
```

Para escribir objetos en disco el segundo paso es crear un objeto de la clase `ObjectOutputStream`, cuyos constructores son:

```
ObjectOutputStream(); // Para crear un objeto que permite escribir objetos de
                        java sobre cualquier dispositivo.
ObjectOutputStream(OutputStream); // Crea un flujo que permite escribir
                                    objetos de java en el objeto OutputStream que se pasa
                                    como argumento.
```

Una vez creado el objeto, la clase dispone de los mismos métodos que `DataOutputStream` más:

```
void writeObject(Object) // Para escribir el objeto en el disco.
```

Propagan la excepción `IOException`.

```
package ud01EjemplosObjetos;
```

```
import java.io.FileNotFoundException;
```

```

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class EscrituraObjetos {
    public static void main(final String[] args) {
        FileOutputStream fs = null;
        ObjectOutputStream os = null;

        try{
            fs = new FileOutputStream("Personas.txt");
            os = new ObjectOutputStream(fs);

            Persona p = new Persona("Marta", 32);
            os.writeObject(p);

            os.writeObject(new Persona("Ana", 27));

            os.close();
        }catch(FileNotFoundException fne){
            System.out.println("Error en el fichero");
        }catch(IOException ioe){
            System.out.println("Error E/L");
        }
    }
}

```

Lectura de Objetos de un Fichero

Cuando se recupera un objeto del disco mediante la llamada a `readObject()`, se produce la **deserialización** del objeto que consiste en la reconstrucción de éste a partir de la información recuperada.

Durante este proceso, los datos miembro no serializables (los que han sido heredados de una clase no serializable) serán inicializados utilizando el constructor por defecto de su clase. Por el contrario, los datos miembro de la clase objeto serializado serán restaurados con los valores almacenados.

Para leer objetos de un fichero que han sido almacenados mediante `ObjectOutputStream`, hay que utilizar la clase `ObjectInputStream`.

El procedimiento es similar a los anteriores

El primer paso es crear un objeto `FileInputStream` que permita recuperar información del fichero, para ello utilizamos los **constructores**:

```

FileInputStream(File fichero);
FileInputStream (String path);

```

El segundo paso es a partir del objeto `FileInputStream` crear un objeto `ObjectInputStream` para realizar la escritura. El **constructor** es:

```

ObjectInputStream(); // Para crear un objeto que permite leer objetos de java
                    sobre cualquier dispositivo.

```

```
ObjectInputStream (InputStream); // Crea un flujo que permite leer objetos
de java en el objeto InputStream que se
pasa como argumento.
```

La clase `ObjectInputStream` proporciona métodos para leer datos desde un fichero:

Mismos métodos que `DataOutputStream` más:

```
Object readObject(); // Para leer un objeto asociado al flujo y devolverlo de tipo
Object. Hay que hacer obligatoriamente un cast a la clase
a la que lo vamos a convertir.
```

```
Persona obj=(Persona) flujolectura.readObject();
```

Todos los métodos propagan una excepción de la clase `IOException`.

El método `readObject()` propaga la excepción `ClassNotFoundException` (Cuando se intenta hacer un cast al objeto que se ha leído de una clase que no es), y la excepción `StreamCorruptedException` (cuando se intenta hacer más de una operación de lectura a través de la clase `ObjectInputStream`).

```
package ud01EjemplosObjetos;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;

public class LecturaObjetos {

    public static void main(final String[] args) {
        System.out.println("Nombre \t Edad" );
        try{
            FileInputStream fs = new FileInputStream("Personas.txt");

            ObjectInputStream os = new ObjectInputStream(fs);

            // solo recupera un objeto
            // os debe realizar un casting al tipo original
            Persona p = (Persona)os.readObject();
            System.out.println(p.getNombre() +"\t" +p.getEdad());

            os.close();
        }catch(ClassNotFoundException cnf){
            System.out.println("Error la clase");
        }catch(FileNotFoundException fnfe){
            System.out.println("Error en el fichero");
        }catch(IOException ioe){
            System.out.println("Error E/L");
        }
    }
}
```

El siguiente ejemplo lee todos los objetos guardados en el fichero.

```
package ud01EjemplosObjetos;

import java.io.FileInputStream;
```

```

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;

public class LecturaObjeto02 {

    public static void main(final String[] args) throws IOException {
        FileInputStream fs = null;
        ObjectInputStream os = null;

        System.out.println("Nombre \t Edad" );
        try{
            fs = new FileInputStream("Personas.txt");
            os = new ObjectInputStream(fs);
            while(true){ // lectura del fichero
                // os debe realizar un casting al tipo original
                Persona p = (Persona)os.readObject();
                System.out.println(p.getNombre() + "\t" + p.getEdad());
            }
        }catch(ClassNotFoundException cnf){
            System.out.println("Error la clase");
        }catch(FileNotFoundException fnfe){
            System.out.println("Error en el fichero");
        }catch(IOException ioe){
            //System.out.println("Error E/L");
        }
        os.close();
    }
}

```

Problemas con la clase ObjectOutputStream

1ª) Cuando escribimos un objeto, es como si guardara el array de bytes en el interior. Si cambiamos los valores de los atributos de ese objeto y volvemos a escribirlo el **ObjectOutputStream** lo escribe nuevamente, pero con los datos antiguos. Da la impresión de que no se entera del cambio y no recalcula los bytes que va a escribir en el fichero.

Ejemplo: Si escribimos así, con un solo new

Persona p = new Persona(); // un único new fuera del bucle

```

for(int i = 0; i < 5; i++){
    p.setPersona(i); /* cambiamos los datos de p, pero no creamos
un nuevo objeto con new */
    oos.writeObject(p);
}

oos.close(); // se cierra al terminar

package ud01EjemplosObjetos;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectOutputStream;

```



```

public class EscrituraObjetosError {

    public static void main(final String[] args) {
        FileOutputStream fs = null;
        ObjectOutputStream os = null;

        try{
            fs = new FileOutputStream("Personas01.txt");
            os = new ObjectOutputStream(fs);
            Persona p = new Persona();
            for (int i= 0; i<5; i++){
                //Persona p = new Persona();
                System.out.println("Introduce el nombre del empleado: " );
                p.setNombre(new BufferedReader (new InputStreamReader
(System.in))).readLine().trim());
                System.out.println("Introduce la edad del empleado: " );
                p.setEdad(Integer.parseInt(new BufferedReader (new InputStreamReader
(System.in))).readLine().trim()));
                os.writeObject(p);
            }
            os.close();
        }catch(FileNotFoundException fne){
            System.out.println("Error en el fichero");
        }catch(IOException ioe){
            System.out.println("Error E/L");
        }
    }
}

```

Cuando leamos, obtendremos cinco veces la primera persona.

Esto puede evitarse de tres formas:

- Haciendo un **new** de cada objeto que queramos escribir, sin reaprovechar instancias. Esto es lo que se ha hecho en el código anterior.
- Usar el método **writeUnshared()** en vez de **writeObject()**. Este método funcionará bien si cambiamos **TODO**s los atributos de la clase **Persona**. Si no modificamos uno de los atributos, obtendremos resultados extraños en ese atributo.
- Llamando al método **reset()** de **ObjectOutputStream** después de escribir cada objeto. Aunque funciona, no parece demasiado ortodoxo.

```
package ud01EjemplosObjetos;
```

```

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectOutputStream;

public class EscrituraObjetosSinError {
    public static void main(final String[] args) {
        FileOutputStream fs = null;
        ObjectOutputStream os = null;

        try{
            fs = new FileOutputStream("Personas01.txt");
            os = new ObjectOutputStream(fs);

```

```

//      Persona p = new Persona();
      for (int i= 0; i<5; i++){
          Persona p = new Persona();
          System.out.println("Introduce el nombre del empleado: " );
          p.setNombre(new BufferedReader (new InputStreamReader
(System.in)).readLine().trim());
          System.out.println("Introduce la edad del empleado: " );
          p.setEdad(Integer.parseInt(new BufferedReader (new
InputStreamReader (System.in)).readLine().trim()));
          os.writeObject(p);
      }
      os.close();
  }catch(FileNotFoundException fne){
      System.out.println("Error en el fichero");
  }catch(IOException ioe){
      System.out.println("Error E/L");
  }
}
}

```

2º) Un segundo problema es que al instanciar el objeto **ObjectOutputStream**, escribe unos bytes de cabecera en el fichero, antes incluso de que escribamos nada. Como el **ObjectInputStream** lee correctamente estos bytes de cabecera, aparentemente no pasa nada y ni siquiera nos enteramos que existen.

El problema se presenta si escribimos unos datos en el fichero y lo cerramos. Luego volvemos a abrirlo para añadir datos, creando un nuevo **ObjectOutputStream** así

```

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

```

```

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

```

Esto escribe una nueva cabecera justo al final del fichero. Luego se irán añadiendo los objetos que vayamos escribiendo. El fichero contendrá lo del dibujo, con dos cabeceras.

Contenido del fichero							
Primera sesión con el fichero				Segunda sesión con el fichero			
cabecera	Persona	Persona	Persona	cabecera	Persona	Persona	Persona

Cuando leemos el fichero, lo primero será crear el **ObjectInputStream**, este leerá la cabecera del principio y luego se pone a leer objetos. Cuando llegamos a la segunda cabecera que se añadió al abrir por segunda vez el fichero para añadirle datos, obtendremos un error **StreamCorruptedException** y no podremos leer más objetos.

Una solución es evidente, no usar más que un solo **ObjectOutputStream** para escribir todo el fichero. Sin embargo, esto no es siempre posible.

Una solución es hacernos nuestro propio **ObjectOutputStream**, heredando del original y redefiniendo el método **writeStreamHeader()**, vacío, para que no haga nada.

```

/* Redefinición del método de escribir la cabecera para que no
haga nada. */

```

```

        protected void writeStreamHeader() throws IOException
        {
        }

    /* La clase MiObjectOutputStream quedaría. */

import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
/**
 * Redefinición de la clase ObjectOuputStream para que no escriba
una cabecera al inicio del Stream.
 *
 */
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
    {
    }
}

```

Ejemplo Lectura escritura de Objetos:

```

package ud01EjemplosObjetos;

import java.io.BufferedReader;
import java.io.EOFException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class EscrituraLecturaObjetos {

    public static void main(final String[] args) throws NumberFormatException,
IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int opcion;
        do{
            System.out.println("1.- Insertar Registros.");
            System.out.println("2.- Leer Registros.");

```

```

        System.out.println("3.- Salir.");
        System.out.println("Elegir opcion: ");
        opcion = Integer.parseInt(br.readLine());
        switch(opcion){
            case 1:
                Escritura();
                break;
            case 2:
                Lectura();
                break;
        }
    } // fin switch
} while (opcion != 3);

} // fin main

public static void Escritura(){
    ObjectOutputStream flujoescritura = null;
    File archivo, carpeta;
    carpeta = new File ("Clientes");
    if (!carpeta.exists()){
        carpeta.mkdir();
    }

    archivo = new File (carpeta, "Correos.txt");

    try{
        flujoescritura = new ObjectOutputStream ( new FileOutputStream
(archivo));

        flujoescritura.writeObject( new Persona ("Pepe Salas Pérez", 45));
        flujoescritura.writeObject( new Persona ("Maria Martín Sierra",
23));

        System.out.println("SE HA ESCRITO EN EL FICHERO "+ archivo);
    } catch (IOException e){
        System.out.println("NO SE HA PODIDO ESCRIBIR LA INFORMACION EN EL
FICHERO "+ archivo);
    } finally{
        try {
            if (flujoescritura!=null){
                flujoescritura.close();
            }
        } catch (IOException e){
            System.out.println("NO SE HA PODIDO CERRAR
CORRECTAMENTE EL FLUJO ASOCIADO AL FICHERO "+ archivo);
        }
    }
}

private static void Lectura() {
    ObjectInputStream flujolectura = null;
    File archivo;
    archivo = new File ("Clientes","Correos.txt");
    try{
        flujolectura = new ObjectInputStream (new FileInputStream
(archivo));

        System.out.println("NOMBRE\t EDAD");
        while(true){
            Persona obj = (Persona) flujolectura.readObject();
            System.out.println(obj.getNombre() + "\t" + obj.getEdad());
        }
    }
}

```

```

        }catch(ClassNotFoundException e){
            System.out.println("NO SE HA PODIDO CONVERTIR LA INFORMACION");
        }catch (EOFException eof){
            System.out.println("Ha finalizado la lectura del fichero");
        }catch(IOException e){
            System.out.println("NO SE HA PODIDO LEER LA INFORMACION DEL
FICHERO");
        }finally{
            try{
                if(flujolectura!=null){
                    flujolectura.close();
                }
            }catch(IOException e){
                System.out.println("NO SE HA PODIDO CERRAR CORRECTAMENTE EL
FLUJO ASOCIADO AL FICHERO");
            }
        }
    } // fin finally

    } // fin metodo lectura

} // fin clase

```

Ficheros de Acceso Aleatorio

Los archivos de acceso secuencial son útiles para la mayoría de las aplicaciones, pero a veces son necesarios archivos de acceso aleatorio que permiten acceder a su contenido de forma secuencial o aleatoria.

La clase **RandomAccessFile** del paquete `java.io` implementa un archivo de acceso aleatorio. Puede ser usada tanto para la lectura como para la escritura de bytes. Dicha clase dispone de métodos para acceder al contenido de un fichero binario de forma aleatoria y para posicionarnos en una posición determinada del mismo. Esta clase no es parte de la jerarquía **InputStream/OutputStream**, ya que su comportamiento es totalmente distinto puesto que se puede avanzar y retroceder dentro de un fichero.

Cuando queramos abrir un fichero de acceso aleatorio tendremos que crear un objeto de tipo **RandomAccessFile** y en el constructor indicaremos la ruta del fichero y el modo de apertura: sólo lectura "**r**", o lectura/escritura "**r/w**".

Hay dos posibilidades para abrir un fichero de acceso aleatorio:

```

RandomAccessFile(String path, String modo); // Con el nombre del
                                             fichero

RandomAccessFile(File fichero, String modo); // Con un objeto
                                             File

```

Ejemplos:

```

RandomAccessFile fichero = new RandomAccessFile("Datos.txt", "rw");
RandomAccessFile fichero = new RandomAccessFile(File archivo, "rw");

```

Todo objeto, instancia de **RandomAccessFile** soporta el concepto de puntero que indica la posición actual dentro del archivo. Cuando en el fichero se crea el puntero se coloca en 0, apuntando al principio del mismo. Las sucesivas llamadas a los métodos `read()` y `write()` ajustan el puntero según la cantidad de bytes leídos o escritos.

Desplazamiento: cualquier operación de lectura/escritura de datos se realiza a partir de la posición actual del “**puntero**” del archivo.

Métodos:	Función
<code>long getFilePointer()</code>	Devuelve la posición actual del puntero del archivo.
<code>void seek(long k)</code>	Coloca el puntero del archivo en la posición indicada por k (los archivos empieza en la posición 0).
<code>int skipBytes(int n)</code>	Intenta saltar n bytes desde la posición actual.
<code>long length()</code>	Devuelve la longitud del archivo.
<code>void setLength(long t)</code>	Establece a t el tamaño del archivo.

Métodos Escritura:	Función
<code>void write(byte b[], int ini, int len)</code>	Escribe len caracteres del vector b .
<code>void write(int i)</code>	Escribe la parte baja de i (un byte) en el flujo.
<code>void writeXxx(xxx)</code>	Escribe el tipo indicado en xxx .

Métodos Lectura:	Función
<code>xxx readXxx();</code>	Lee y devuelve el tipo leído
<code>void readFully(byte b[]);</code>	Lee bytes del archivo y los almacena en un vector de bytes.
<code>void readFully(byte b[], int ini, int len)</code>	Lee len bytes del archivo y los almacena en un vector de bytes.
<code>String readUTF()</code>	Lee una cadena codificada con el formato UTF-8.

```
/*
 * En el siguiente ejemplo crearemos una aplicación que nos permita guardar
 * datos de los alumnos de un curso en un fichero de texto con RandomAccessFile.
 * Crearemos un menú para poder seleccionar entre las opciones: Introducir nota,
 * Listado completo, Ver notas de un alumno, Modificar nota y Salir.
 */
```

```
package ud01;
```

```
import java.io.BufferedReader;
import java.io.IOException;
```

```

import java.io.InputStreamReader;
import java.io.RandomAccessFile;

public class EjemploFicherosAleatorios01 {

    //atributos
    private static final int TAMANHOREGISTRO=50; // tamaño del registro
    private static final int NOMBRELONGMAX=30; // longitud máxima del nombre
    private static final int MODULOLONGMAX=15; // // longitud máxima del modulo
    private static RandomAccessFile nota;

    public static void main(final String[] args) {
        int opcion;
        try{
            nota = new RandomAccessFile ("notas.txt","rw");
            fin:do{
                System.out.println("1.- Introducir Alumnos");
                System.out.println("2.- Listado Alumnos");
                System.out.println("3.- Buscar un Alumno");
                System.out.println("4.- Modificar notas un Alumno");
                System.out.println("5.- Salir");

                opcion = Integer.parseInt(Datos("Introduce una opcion: "));
                try{
                    switch (opcion){
                        case 1:
                            meterNotaEnFichero();
                            break;
                        case 2:
                            ListadoCompleto();
                            break;
                        case 3:
                            notasDe1Alumno();
                            break;
                        case 4:
                            modificarNota();
                            break;
                        case 5:
                            break fin;
                        default:
                            System.out.println("Opcion errónea");
                    }
                }catch (NumberFormatException e){
                    System.out.println("La opcion tiene que ser un
numero");
                }
            }while(opcion != 5);
        }catch (IOException ioe){
            System.out.print("\nError "+ioe.toString());
        }finally{
            try{
                if (nota!=null)
                    nota.close();
            }catch (IOException ioe){
                System.out.print("\nError "+ioe.toString());
            }
        }
    }
}
// fin del main
/*-----

```

```

    * Método para introducir datos desde el teclado
    */
    public static String Datos(final String s){
        try{
            System.out.println("-----
-----");
            System.out.print(s);
            return (new BufferedReader (new InputStreamReader
(System.in))).readLine();
        }catch (IOException ioe){
            System.out.println("\nError interno en sistema de
entrada/salida\n");
        }
        return "";
    }
    /*-----
    * Método que devuelve el número de registros del fichero
    */
    private static int NumRegistros(){
        try{
            return (int)Math.ceil ((double)nota.length()/
(double)TAMANHOREGISTRO);
        }catch (IOException e){
            System.out.println("ERROR GRAVE DE ENTRADA/SALIDA");
            return 0;
        }
    }
    /*-----
    * Escribe los datos en el fichero
    */
    public static void meterNotaEnFichero(){
        try{
            String aux="";
            nota.seek( NumRegistros()* TAMANHOREGISTRO);// se coloca al final
del fichero

            // si el nombre es demasiado largo vuelve a solicitarlo, en caso
contrario lo escribe en el fichero
            do{
                aux=Datos("Introduce nombre: ");
                if (aux.length()>=NOMBRELONGMAX){
                    System.out.println("\nNombre demasiado largo\n");
                    System.out.println("-----
-----");
                }
            }while(aux.length()>=NOMBRELONGMAX);
            nota.writeUTF(aux);

            // si el nombre del modelo es demasiado largo vuelve a solicitarlo,
en caso contrario
            // lo escribe en el fichero
            do{
                aux=Datos("Introduce modulo: ");
                if (aux.length()>=MODULOLONGMAX){
                    System.out.println("\nNombre de modulo demasiado
largo");
                    System.out.println("-----
-----");
                }
            }

```



```

        }while(aux.length()>=MODULO_LONGMAX);
        nota.writeUTF(aux);
        System.out.println("-----");

        boolean sw = false;
        do{
            sw = false;
            try{
                aux=Datos("Introduce nota: ");
                nota.writeInt(Integer.parseInt(aux));
            }catch (NumberFormatException e){
                System.out.println("Debes introducir un numero");
                sw = true;
            }
        }while(sw);
        }catch(IOException ioe){
            System.out.print("\nError "+ioe.toString());
        }
    }
    }// fin del método meterNotaEnFichero()

    /*-----
    * Listado de todos los datos del fichero
    */
    public static void listadoCompleto(){
        if (NumRegistros()!=0){
            System.out.println("\n\t LISTADO COMPLETO\n-----");
            System.out.println("Nombre\tModulo\tNota");
            System.out.println("\n-----");

            try{
                for (int i=0; i<NumRegistros();i++){
                    nota.seek(i* TAMANHOREGISTRO);
                    System.out.println(nota.readUTF() +"\t"+ nota.readUTF()
+ "\t"+ nota.readInt());
                }
                System.out.println("\n-----");
            }catch (IOException ioe){
                System.out.println("\nError "+ioe.toString());
            }
        }else
            System.out.println("\nNo hay registros introducidos");
    }// fin método listadoCompleto()

    //-----
    public static void notasDe1Alumno(){
        String aux = "";
        boolean esta = false;
        if (NumRegistros()>0) {
            aux = Datos("Introduce el nombre de un alumno: ");
            try{
                for (int i=0; i<NumRegistros();i++){
                    nota.seek(i* TAMANHOREGISTRO);
                    if (aux.compareToIgnoreCase(nota.readUTF()) == 0){
                        nota.seek(i* TAMANHOREGISTRO);
                        System.out.println("\n-----");
                    }
                }
            }
        }
    }

```

```

        System.out.println("Nombre: "+ nota.readUTF());
        System.out.println("Modulo: "+ nota.readUTF());
        System.out.println("Nota: "+ nota.readInt());
        esta = true;
    }
}

if (esta==false)
    System.out.println("\nAlumno inexistente");

}catch (IOException ioe){
    System.out.println("\nError "+ioe.toString());
}
}else
    System.out.println("No hay datos en el fichero");
} // fin método notasDe1Alumno()

/*-----
 * Método que modifica la nota de un determinado alumno
 */
public static void modificarNota(){
    String aux="";
    String aux2="";
    boolean esta = false;

    if (NumRegistros()>0) {
        aux = Datos("Introduce el nombre de un alumno: ");
        aux2 = Datos("Introduce el modulo: ");
        String opcion="";
        try{
            for (int i = 0; i < NumRegistros(); i++){
                nota.seek(i* TAMANHOREGISTRO);
                if (aux.compareToIgnoreCase(nota.readUTF()) == 0){
                    if (aux2.compareToIgnoreCase(nota.readUTF())==0){
                        nota.seek(i* TAMANHOREGISTRO);
                        System.out.println("Nombre: "+
                            nota.readUTF());
                        System.out.println("Modulo: "+
                            nota.readUTF());
                        opcion = Datos("Introduce la nueva nota: ");
                        nota.writeInt(Integer.parseInt(opcion));
                        esta = true;
                    }
                }
            }
        } // fin for

        if (esta == false)
            System.out.println("Alumno inexistente");

    }catch (IOException ioe){
        System.out.println("\nError "+ioe.toString());
    }
}else
    System.out.println("No hay datos en el fichero");
} // fin método modificarNota
} // fin de la clase

```

En el siguiente ejemplo se crea un fichero aleatorio en el que la clave es la posición del registro dentro del fichero

```
/* crea un fichero de acceso directo
 * registro: clave (int), nombre (String 25), edad (int)
 * la clave es la posición del registro dentro del fichero
 */
package ud01;

import java.io.BufferedReader;
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.RandomAccessFile;

public class EjemploFicherosAleatorios02 {

    // en un fichero ramdow hay que establecer el tamaño del registro
    //recordar que un carácter son dos bytes en archivo
    private static final long tamanhoRegistro = 35;

    public static void main(final String[] args) {
        String opcion;
        do{
            System.out.println("1.- Introducir nuevo registro");
            System.out.println("2.- Listado completo");
            System.out.println("3.- Buscar registro");
            System.out.println("4.- Modificar registro");
            System.out.println("5.- Salir");

            opcion = Datos("Introduce una opcion: ");
            try{
                switch (Integer.parseInt(opcion)){
                    case 1:
                        insertarRegistro();
                        break;
                    case 2:
                        listadoCompleto();
                        break;
                    case 3:
                        buscarRegistro();
                        break;
                    case 4:
                        modificarRegistro();
                        break;
                    case 5:
                        System.exit(0);
                    default:
                        System.out.println("Opcion erronea");
                }
            }catch (NumberFormatException e){
                System.out.println("La opcion tiene que ser un
numero");
            }
        }while(!opcion.equals("5"));
    }

    // métodos publicos y static
}
```

```

/*-----
 * método para introducir los datos desde el teclado
 */
    public static String Datos(final String s){
        try{
            System.out.println("-----
-----");
            System.out.print(s);
            return (new BufferedReader (new InputStreamReader
(System.in))).readLine();
        }catch (IOException ioe){
            System.out.println("\nError interno en sistema de
entrada/salida\n");
        }
        return "";
    }
    }// fin metodo Datos()
/*-----
 * Metodo que inserta los registros en el fichero
 * el campo clave determina la posición del registro dentro del fichero
 */
public static void insertarRegistro(){

    RandomAccessFile nombreEdad = null;
    int clave=0;
    String nombre="";
    int edad=0;
    try{
        // abriendo archivo, capturando y grabando datos
        nombreEdad = new RandomAccessFile ("NombresEdades.txt","rw");

        int x=0;

        do{
            // teclea los datos
            clave = Integer.parseInt(Datos("Introduce la clave: "));

            //comprueba la longitud del nombre tecleado si es menor que 25 lo
rellena
            // si es mayor lo acorta
            nombre = Datos("Introduce el nombre: ");
            if (nombre.length() < 25) {
                for(int i=nombre.length(); i <25; i++)
                    nombre=nombre+" ";
            }
            else {
                nombre=nombre.substring(0,25);
            }

            edad = Integer.parseInt(Datos("Introduce la edad: "));
            // grabando al archivo
            if (nombreEdad.length()!= 0){
                nombreEdad.seek( nombreEdad.length() );
            }
            //colocamos el puntero según la clave
            nombreEdad.seek((clave-1) * tamañoRegistro);
            nombreEdad.writeInt(clave);
            nombreEdad.writeUTF(nombre);
            nombreEdad.writeInt(edad);

```

```

        x++;
    }while(x<3);
    }
    catch(NumberFormatException nfe){
        System.out.println("Error al introducir los datos");
    }catch(FileNotFoundException fnf){
        System.out.println("Fichero inexistente");
    }catch (IOException ioe) {
        System.out.println(" Error al escribir en el fichero");
    }finally{
        try{
            nombreEdad.close();
        }
        catch(IOException e){
            System.out.println(" Error al cerrar el fichero ");
        }
    }

}

//-----
public static void listadoCompleto(){
    int clave=0;
    String nombre="";
    int edad=0;
    RandomAccessFile nombreEdad = null;
    long contadorRegistros = 0;
    try {
        // abriendo archivo, capturando datos
        nombreEdad = new RandomAccessFile ("NombresEdades.txt","rw");

        //calculando el numero de registros
        contadorRegistros = nombreEdad.length()/tamanhoRegistro;
        System.out.println(contadorRegistros+" "
+nombreEdad.length()+ "\n\n");

        for (int r=0; r < contadorRegistros; r++) {
            nombreEdad.seek(r* tamanhoRegistro);
            clave=nombreEdad.readInt();
            nombre = nombreEdad.readUTF();
            edad=nombreEdad.readInt();

            System.out.println(clave+" "+nombre+" "+edad + "\n");
        }

    }catch EOFException eof){
        System.out.println("Final del fichero ");
    }catch(FileNotFoundException fnf){
        System.out.println("Fichero inexistente");
    }catch (IOException ioe) {
        System.out.println("Error al leer el fichero ");
    }finally{
        try{
            nombreEdad.close();
        }
        catch(IOException e){
            System.out.println(" Error al cerrar el fichero ");
        }
    }
}

```

```

    }

}
//-----
public static void buscarRegistro(){
    int clave=0;
    String nombre="";
    int edad=0;
    RandomAccessFile nombreEdad = null;
    try {
        // abriendo archivo, capturando datos
        nombreEdad = new RandomAccessFile ("NombresEdades.txt","rw");

        clave = Integer.parseInt(Datos("Introducir la clave. < 0 para
Finalizar>: "));
        while(clave != 0){
            //colocamos el puntero según la clave
            nombreEdad.seek((clave-1) * tamañoRegistro);

            //leemos los campos del registro
            clave=nombreEdad.readInt();
            nombre = nombreEdad.readUTF();
            edad=nombreEdad.readInt();

            //visualizamos los datos
            System.out.println(clave+" "+nombre+" "+edad);
            clave = Integer.parseInt(Datos("Introducir la clave. <
0 para Finalizar>: "));
        } // fin while

    } catch (NumberFormatException nfe){
        System.out.println("Error al introducir los datos");
    } catch (IOException ioe) {
        System.out.println("Error de posicionamiento o lectura");
        System.out.println(ioe.getMessage());
    } finally{
        try{
            nombreEdad.close();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}

}

/*-----
* Metodo que modificar datos de un registros en el fichero
* el campo clave determina la posición del registro dentro del fichero

* hay que posicionarse para leer el registro y hay que volver a
posicionarse
* antes de escribir
*/
public static void modificarRegistro(){
    int clave=0;
    String nombre="";
    String respuesta = "";
    int edad=0;

```

```

RandomAccessFile nombreEdad = null;
try {
    // abriendo archivo, capturando datos
    nombreEdad = new RandomAccessFile ("NombresEdades.txt","rw");

    clave = Integer.parseInt(Datos("Introducir la clave a
modificar. < 0 para Finalizar>: "));
    while(clave != 0){
        //colocamos el puntero según la clave
        nombreEdad.seek((clave-1) * tamanhoRegistro);

        //leemos los campos del registro
        clave=nombreEdad.readInt();
        nombre = nombreEdad.readUTF();
        edad=nombreEdad.readInt();

        //visualizamos los datos
        System.out.println(clave+" "+nombre+" "+edad);

        respuesta = Datos("Desea modificar el registro. S/N");
        if (respuesta.compareToIgnoreCase("s") == 0){
            //teclea los nuevos valores de los campos

            nombre = Datos("Introduce el nombre: ");
            if (nombre.length() < 25) {
                for(int i=nombre.length(); i <25; i++)
                    nombre=nombre+" ";
            }
            else {
                nombre=nombre.substring(0,25);
            }

            edad = Integer.parseInt(Datos("Introduce la edad: "));
            // grabando al archivo
            if (nombreEdad.length() != 0){
                nombreEdad.seek( nombreEdad.length() );
            }
            //colocamos el puntero según la clave
            nombreEdad.seek((clave-1) *
tamanhoRegistro);

            nombreEdad.writeInt(clave);
            nombreEdad.writeUTF(nombre);
            nombreEdad.writeInt(edad);
        }

        clave = Integer.parseInt(Datos("Introducir la clave. < 0 para
Finalizar>: "));
    } // fin while

} catch (NumberFormatException nfe){
    System.out.println("Error al introducir los datos");
} catch (IOException ioe) {
    System.out.println("Error de posicionamiento o lectura");
    System.out.println(ioe.getMessage());
} finally{
    try{
        nombreEdad.close();
    } catch (IOException e){
        e.printStackTrace();
    }
}

```

```

        }
    }

    } // fin metodo modificarEdad()
} // fin de la clase

```

TRABAJO CON FICHEROS XML

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

Un fichero XML sencillo tiene la siguiente estructura:

```

<?xml versión="1.0"?>
<Empleado>
    <empleado>
        <id>1</id>
        <apellido>GARCIA</apellido>
        <departamento>25</departamento>
        <salario>1826.25</salario>
    </empleado>
    <empleado>
        <id>2</id>
        <apellido>FERNANDEZ</apellido>
        <departamento>2</departamento>
        <salario>1256.25</salario>
    </empleado>
    <empleado>
        <id>3</id>
        <apellido>ALVAREZ</apellido>
        <departamento>10</departamento>
        <salario>2356.25</salario>
    </empleado>
</Empleado>

```

Los ficheros XML se pueden utilizar:

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

Para leer los ficheros XML y acceder a su contenido y estructura, se utiliza un procesador de XML o **parser**. El procesador lee los documentos y proporciona acceso a su contenido y estructura. Algunos de los procesadores más empleados son: **DOM**: *Modelo de Objetos de Documento* y **SAX**: *API Simple para XML*.

- **DOM**: un procesador XML que utilice este planteamiento almacena la estructura del documento en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales (que son aquellos que no tienen descendientes). Una vez creado el árbol, se van recorriendo los diferentes nodos (de arriba abajo y también se puede volver atrás) y se analiza a qué tipo particular pertenecen. Podemos modificar cualquier nodo del árbol. Tiene su origen en el W3C¹. Este tipo de procesamiento necesita más recursos de memoria y tiempo sobre todo si los ficheros XML a procesar son bastante grandes y complejos.

Con ficheros XML pequeños no tendremos problemas, pero si tuviéramos un árbol muy muy grande entonces tendríamos una falta de **heap space**².

- **SAX**: un procesador que utilice este planteamiento lee un fichero XML de forma secuencial y produce una secuencia de eventos (comienzo/fin del documento, comienzo/fin de una etiqueta, etc.) en función de los resultados de la lectura. Cada evento invoca a un método definido por el programador. Este tipo de procesamiento prácticamente no consume memoria, pero por otra parte, impide tener una visión global del documento por el que navegar.

Al contrario que con DOM, al procesar en SAX no vamos a tener la representación completa del árbol en memoria, pues SAX funciona con eventos. Esto implica:

Al no tener el árbol completo no puede volver atrás, pues va leyendo secuencialmente.

La modificación de un nodo y la inserción de nuevos nodos son mucho más complejas.

Como no tiene el árbol en memoria es mucho más **memory friendly**, de modo que es la única opción viable para casos de ficheros muy grandes, pero demasiado complejo para ficheros pequeños.

Al ser orientado a eventos, el procesado se vuelve bastante complejo.

Acceso a Ficheros XML con DOM

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

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

¹ El World Wide Web Consortium (W3C) es una comunidad internacional que desarrolla estándares que aseguran el crecimiento de la Web a largo plazo.

² Espacio de memoria que utilizan los objetos.

Los programas Java que utilicen **DOM** necesitan interfaces, algunas son:

Interface	Función
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 cualquier nodo del documento
NodeList	Contiene una lista con los nodos hijos de un nodo.
Att	Permite acceder a los atributos de un nodo
Text	Son los datos carácter de un elemento
CharacterData	Representa 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>

Ejemplo de creación de un fichero XML, en el ejemplo vamos a partir del fichero Personas.txt.

Lo primero que tenemos que hacer es importar los paquetes necesarios:

```
import java.io.*;

import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

import org.w3c.dom.*
```

Después crearemos una instancia de **DocumentBuilderFactory** para construir el parser (necesario para crear el árbol), se debe encerrar entre **try-catch** porque se puede producir la excepción **ParserConfigurationException**.

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
try{
    DocumentBuilder db = dbf.newDocumentBuilder();
    .....
}
```

Creamos un documento vacío de nombre **documento** con el nodo raíz de nombre Personas y asignamos la versión XML:

```
DOMImplementation implementacion = db.getDOMImplementation();
Document documento = implementacion.createDocument(null, "Personas", null);
documento.setXmlVersion("1.0"); // asignamos la versión de nuestro XML
```

El siguiente paso será recorrer el fichero con los datos de las personas y por cada registro crear un nodo persona con 3 hijos (clave, nombre, edad). Cada nodo hijo tendrá su valor (por ejemplo: 1. Isabel, 42). Para crear un elemento usamos el método **createElement(String)** llevando como parámetro el nombre que se pone entre las etiquetas menor que y mayor que. El siguiente código crea y añade el nodo <persona> al documento:

```
Element raiz = documento.createElement("persona"); //creamos el nodo
persona
documento.getDocumentElement().appendChild(raiz); // lo pegamos a la raíz
del documento
```

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

```
CrearElemento("clave", Integer.toString(clave),raiz,documento); // añadir
clave
CrearElemento("nombre", nombre.trim(),raiz,documento); // añadir nombre
CrearElemento("edad", Integer.toString(edad),raiz,documento); // añadir
edad
```

La función recibe el nombre del nodo hijo (clave, nombre, edad) y sus textos o valores que tienen que estar en formato String (1. Isabel, 42), el nodo al que se va a añadir (raíz) y el documento (documento). Para crear el nodo hijo (<clave> o <nombre> o <edad>) se escribe:

```
Element elemento = documento.createElement(datoPersona); // creamos hijo
```

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

```
Text texto = documento.createTextNode(valor); // damos valor
```

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

```
raiz.appendChild(elemento); // pegamos el elemento hijo a la raíz
elemento.appendChild(texto); // pegamos el valor
```

Al final se generaría algo similar a esto:

```
<?xml version="1.0" encoding="UTF-8"?>
<Personas>
  <persona>
    <clave>1</clave>
    <Nombre>isabel</Nombre>
    <edad>42</edad>
  </persona>
  <persona>
    <clave>2</clave>
    <Nombre>iciar</Nombre>
    <edad>16</edad>
  </persona>
  .....
</Personas>
```

La función es la siguiente:

```
// método de inserción de los datos de la persona
static void CrearElemento(final String datoPersona, final String valor, final Element
raiz, final Document documento){
    Element elemento = documento.createElement(datoPersona); // creamos hijo
    Text texto = documento.createTextNode(valor); // damos valor
    raiz.appendChild(elemento); // pegamos el elemento hijo a la raíz
    elemento.appendChild(texto); // pegamos el valor
} // fin del método
```

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

```
Source fuente = new DOMSource(documento);
```

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

```
Result resultado = new StreamResult(new java.io.File("Personas.xml"));
```

Se obtiene su **TransformerFactory**:

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

Se realiza la transformación del documento a fichero.

```
transformer.transform(fuente, resultado);
```

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

```
Result consola = new StreamResult(System.out);  
transformer.transform(fuente, consola);
```

El código completo es:

```
/*  
 * Ejemplo que crea un fichero XML con DOM a partir del fichero aleatorio Personas.txt  
 */  
package ud01;  
import java.io.File;  
import java.io.IOException;  
import java.io.RandomAccessFile;  
  
import javax.xml.parsers.DocumentBuilder;  
import javax.xml.parsers.DocumentBuilderFactory;  
import javax.xml.transform.Result;  
import javax.xml.transform.Source;  
import javax.xml.transform.Transformer;  
import javax.xml.transform.TransformerFactory;  
import javax.xml.transform.dom.DOMSource;  
import javax.xml.transform.stream.StreamResult;  
  
import org.w3c.dom.DOMImplementation;  
import org.w3c.dom.Document;  
import org.w3c.dom.Element;  
import org.w3c.dom.Text;  
  
public class EjemploFicheroXMLDOMEscritura {  
    private static final long tamanhoRegistro = 35;  
  
    public static void main(final String[] args) throws IOException {  
        File fichero = new File ("NombresEdades.txt");  
        RandomAccessFile raf = new RandomAccessFile(fichero, "r");  
  
        int clave, edad;  
        long posicion = 0; // para situarnos al principio del fichero  
        String nombre, aux;  
  
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();  
  
        try{  
            DocumentBuilder db = dbf.newDocumentBuilder();
```

```

DOMImplementation implementacion = db.getDOMImplementation();
Document documento = implementacion.createDocument(null, "Personas",
null); // crea el documento con el nodo raíz de nombre Personas
documento.setXmlVersion("1.0"); // asignamos la versión de nuestro
XML

        for(;;){
            raf.seek(posicion); // nos posicionamos al comienzo del
fichero

            clave=raf.readInt(); // leemos los datos del fichero
            nombre = raf.readUTF();
            edad=raf.readInt();

            if(clave > 0){ // clave valida
                Element raiz = documento.createElement("persona");
                //creamos el nodo persona
                documento.getDocumentElement().appendChild(raiz); // lo
                pegamos a la raíz del documento
                CrearElemento("clave",
                Integer.toString(clave),raiz,documento); // añadir clave
                CrearElemento("nombre", nombre.trim(),raiz,documento);
                // añadir nombre
                CrearElemento("edad",
                Integer.toString(edad),raiz,documento); // añadir edad
            } // fin if clave

            posicion = posicion+tamahoRegistro; // se posiciona para el
            siguiente registro

            if(raf.getFilePointer() == raf.length())
                break;
            } // fin del for que recorre el fichero
        // recorremos el fichero XML para ver su contenido
        Source fuente = new DOMSource(documento);
        Result resultado = new StreamResult(new
        java.io.File("Personas.xml"));
        Transformer transformer =
        TransformerFactory.newInstance().newTransformer();
        transformer.transform(fuente, resultado);

        // para mostrar el documento por pantalla, podemos especificar como
        resultado el canal de salida System.out
        Result consola = new StreamResult(System.out);
        transformer.transform(fuente, consola);

    } catch (Exception e){
        System.err.println("Error: " +e);
    }
    raf.close();
} // fin del main

// método de inserción de los datos de la persona
static void CrearElemento(final String datoPersona, final String valor, final
Element raiz, final Document documento){
    Element elemento = documento.createElement(datoPersona); // creamos hijo
    Text texto = documento.createTextNode(valor); // damos valor
    raiz.appendChild(elemento); // pegamos el elemento hijo a la raíz
    elemento.appendChild(texto); // pegamos el valor
} // fin del método
} // fin de la clase

```

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

```
DocumentBuilder db = dbf.newDocumentBuilder();
Document documento = db.parse(new File("Personas.xml"));
```

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

```
NodeList personas = documento.getElementsByTagName("persona");
```

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

```
/*
 * ejemplo de como leer un fichero XML DOM
 */
package ud01;
import java.io.File;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
public class EjemploFicheroXMLDOMLectura {

    public static void main(final String[] args) {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        try{
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document documento = db.parse(new File("Personas.xml"));
            documento.getDocumentElement().normalize();

            System.out.println("Elemento raiz: "
+documento.getDocumentElement().getNodeName());
            //crea una lista con todos los nodos de personas
            NodeList personas =
documento.getElementsByTagName("persona");
            // recorre la lista
            for(int i=0; i< personas.getLength(); i++){
                Node persona = personas.item(i); // obtener un nodo
                if(persona.getNodeType() == Node.ELEMENT_NODE){ // tipo
de nodo
                    Element elemento = (Element) persona; //obtener
los elementos del nodo

                    System.out.println("Clave: " +getNode("clave",
elemento));
                    System.out.println("Nombre: " +getNode("nombre",
elemento));
                    System.out.println("Edad: " +getNode("edad",
elemento));

                } // fin if
            } // fin for

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

```

    }
} // fin main

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

```

Acceso a Ficheros XML con SAX

SAX (API Simple para XML) es un conjunto de clases e interfaces que ofrecen una herramienta muy útil para el procesamiento de documentos XML. Permite analizar los documentos de forma secuencial (es decir, no carga todo el fichero en memoria como hace DOM), esto implica poco consumo de memoria aunque los documentos sean de gran tamaño, en contraposición, impide tener una visión global del documento que se va a analizar. SAX es más complejo de programar que DOM, es un API totalmente escrita en Java e incluida dentro de JRE que nos permite crear nuestro propio parser XML.

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

Documento XML (Personas.xml)	Métodos asociados a eventos del documento
<pre> <?xml version="1.0" encoding="UTF-8"?> <ListaPersonas> <persona> <clave> 1 </clave> <Nombre> Isabel </Nombre> <edad> 42 </edad> </persona> <persona> <clave> 2 </clave> <Nombre> Iciar </Nombre> <edad> 16 </pre>	<pre> startDocument() startElement() startElement() startElement() characters() endElement() startElement() characters() endElement() startElement() startElement() characters() endElement() startElement() startElement() characters() endElement() startElement() characters() </pre>

<pre> </edad> </persona> </ListaPersonas> </pre>	<pre> endElement() endElement() endElement() endDocument() </pre>
--	---

Ejemplo en Java en el que se muestran los pasos básicos necesarios para hacer que se puedan tratar los eventos.

En primer lugar se incluyen las clases e interfaces de SAX:

```

import java.io.IOException;

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

```

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

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

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

Interface	Función
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 se llaman cada vez que se encuentra una referencia a una entidad.
DefaultHandler	Clase que provee una implementación por defecto para todos sus métodos, el programador definirá los métodos que sean utilizados por el programa. Esta clase es de la que extenderemos para poder crear nuestro parser de XML.

DefaultHandler es una clase que va a procesar cada evento que lance el procesador SAX. Basta con heredar del manejador por defecto de **SAX DefaultHandler** y sobrescribir los métodos correspondientes a los eventos deseados. Los más comunes son:

- **startDocument**: se produce al comenzar el procesado del documento xml.
- **endDocument**: se produce al finalizar el procesado del documento xml.
- **startElement**: se produce al comenzar el procesado de una etiqueta xml. Es aquí donde se leen los atributos de las etiquetas.
- **endElement**: se produce al finalizar el procesado de una etiqueta xml.
- **characters**: se produce al encontrar una cadena de texto.

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

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

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

```
InputStream ficheroXML = new InputStream("Personas.xml");
```

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

```
procesadorXML.parse(ficheroXML);
```

El ejemplo completo sería:

```
/*
 * Ejemplo que crea un fichero XML con SAX a partir del fichero aleatorio Personas.txt
 */
package ud01;
import java.io.IOException;

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 EjemploFicheroXMLSAX01 {

    public static void main(final String[] args) throws SAXException, IOException {
        XMLReader procesadorXML = XMLReaderFactory.createXMLReader();
        GestionContenido gestor = new GestionContenido();
        procesadorXML.setContentHandler(gestor);
        InputSource ficheroXML = new InputSource("Personas.xml");
        procesadorXML.parse(ficheroXML);
    } // fin main
} // fin clase EjemploFicheroXMLSAX01

class GestionContenido extends DefaultHandler{
    public GestionContenido(){
        super();
    }

    @Override
    public void startDocument(){
        System.out.println("Comienzo del documento XML");
    }

    @Override
    public void endDocument(){
        System.out.println("Final del documento XML");
    }

    public void startElement(final String uri, final String nombre, final String
    nombreC){
        System.out.println("Principio Elemento: " + nombre);
    }
}
```

```

    }

    @Override
    public void endElement(final String uri, final String nombre, final String
nombreC){
        System.out.println("Fin Elemento: " +nombre);
    }

    @Override
    public void characters(final char[] ch, final int inicio, final int longitud){
        String car = new String(ch, inicio, longitud);
        car = car.replaceAll("[\t\n]", ""); //quitar saltos de linea
        System.out.println("\tCaracteres: " +car);
    }
}

```

Serialización de Objetos XML

Para serializar objetos Java a XML o viceversa necesitamos la librería **XStream**. Para poder utilizarla debemos descargar el fichero JAR desde la web: <http://xstream.codehaus.org/download.html>, para el ejemplo descargaremos el fichero **Binary distribution** (*xstream-distribution-1.4-2-bin.zip*) lo descomprimos y buscamos el fichero JAR **xstream-1.4.7.jar** que está en la carpeta **lib**. También necesitamos el fichero **kxml2-2.3.0.jar** que se puede descargar desde el apartado **Optional Dependencies**.

Partimos del fichero “*Personas.txt*” que contiene objetos *Persona*. El proceso es crear una lista de objetos *Persona* y la convertiremos en un fichero de datos XML. Necesitamos la clase *Persona* (que ya está definida) y la clase *ListaPersona* en la que definimos una lista de objetos *Persona* que pasaremos al fichero XML:

```

package ud01EjemplosObjetos;

import java.util.ArrayList;
import java.util.List;

public class ListaPersona {
    private final List<Persona> lista = new ArrayList<Persona>();

    public ListaPersona() {
        super();
    }
    public void add(final Persona per){
        lista.add(per);
    }
    public List<Persona> getListaPersonas(){
        return lista;
    }
}

```

El proceso consistirá en recorrer el fichero *Personas.txt* para crear una lista de persona que después se insertarán en el fichero *Personas.xml*.

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

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

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

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

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

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

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

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

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

```
xstream.toXML(listaper, new FileOutputStream("Personas01.xml"));
```

El código completo es el siguiente:

```
/*
 * Ejemplo que recorre el fichero Personas.txt para crear una lista de personas
 * que después se insertarán en el fichero Personas.xml
 */
package ud01EjemplosObjetos;
import java.io.EOFException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

import com.thoughtworks.xstream.XStream;

public class EscrituraObjetosFicheroXML {

    public static void main(final String[] args) throws IOException,
ClassNotFoundException {
        File fichero = new File("Personas.txt");
        FileInputStream lectura = new FileInputStream(fichero); // flujo de
entrada
        // conecta el flujo de bytes al flujo de datos
        ObjectInputStream datos = new ObjectInputStream(lectura);

        System.out.println("Comienza el proceso de creación del fichero XML....");

        // Creamos un objeto Lista de Persona
        ListaPersona listaper = new ListaPersona();

        try{
            while(true){ // lectura del fichero
                Persona persona = (Persona)datos.readObject(); // leer una
persona
                listaper.add(persona);
            } // fin while
        } catch (EOFException eo){}
        datos.close();
    }
}
```

```

    try{
        XStream xstream = new XStream();
        //cambiar de nombre a las etiquetas XML
        xstream.alias("ListaPersonasMunicipio", ListaPersona.class);
        xstream.alias("DatosPersona", Persona.class);
        //quitar etiqueta lista (atributo de la clase ListaPersona
        xstream.addImplicitCollection(ListaPersona.class, "lista");
        //Insertar los objetos en el XML
        xstream.toXML(listaper, new FileOutputStream("Personas01.xml"));
        System.out.println("Creado el fichero xml");
    }catch(Exception e){
        e.printStackTrace();
    }
}
} // fin main
} // fin clase

```

El fichero generado tiene el siguiente aspecto:

```

<?xml version="1.0"?>
<ListaPersonasMunicipio>
  <DatosPersona>
    <nombre>
      Marta
    </nombre>
    <edad>
      32
    </edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>
      Ana
    </nombre>
    <edad>
      27
    </edad>
  </DatosPersona>
</ListaPersonasMunicipio>

```

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

```

/*
 * Lectura de un fichero XML a objetos
 */
package ud01EjemplosObjetos;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.thoughtworks.xstream.XStream;

public class LecturaObjetosFicheroXML {

    public static void main(final String[] args) throws FileNotFoundException {

```

```

//crear una instancia de la clase XStream
XStream xstream = new XStream();

//cambiar de nombre a las etiquetas XML
xstream.alias("ListaPersonasMunicipio", ListaPersona.class);
xstream.alias("DatosPersona", Persona.class);

//quitar etiqueta lista (atributo de la clase ListaPersona
xstream.addImplicitCollection(ListaPersona.class, "lista");

ListaPersona listadoTodas = (ListaPersona)
    xstream.fromXML(new FileInputStream("Personas01.xml"));

System.out.println("Número de personas: "
+listadoTodas.getListaPersonas());

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

Iterator iterator = listaPersonas.listIterator(); //recorrer los elementos
while(iterator.hasNext()){
    Persona p = (Persona) iterator.next(); //obtenemos el elemento
    System.out.println("Nombre: " +p.getNombre() +"\tEdad: "
+p.getEdad());
} // fin del while

System.out.println("\n\nFin del listado... ");
} // fin main
} //fin clase

```

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

```

ListaPersona listadoTodas = (ListaPersona)
    xstream.fromXML(new FileInputStream("Personas01.xml"));

```

API XStream:

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

Conversión de Ficheros XML a otro formato

XSL (*Extensible Stylesheet Language*) son recomendaciones del Word Wide Web Consortium (<http://www.w3.org/Style/XSL/>) para expresar hojas de estilo en lenguaje XML. Una hoja de estilo XSL describe el proceso de presentación a través de un pequeño conjunto de elementos XML. Esta hoja puede contener elementos de reglas que representan a las reglas de construcción y elementos de reglas de estilo que representan a las reglas de mezclas de estilos.

En el ejemplo vamos a ver como a partir de un fichero XML que contiene datos y otro XSL que contiene la presentación de esos datos se puede generar un fichero HTML usando el lenguaje Java.

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 el mensaje y detiene el programa.

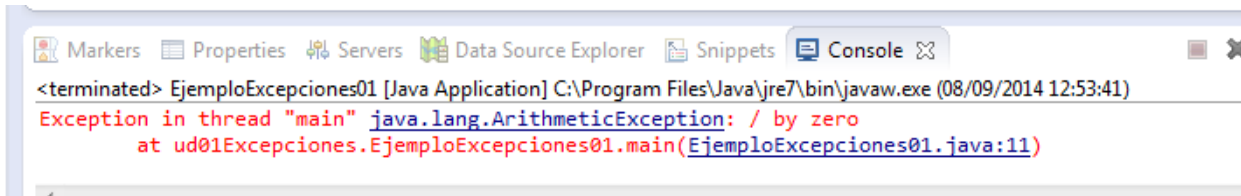
Ejemplo que produce una excepción y visualiza un mensaje indicando el error:

```
/*
 * Ejemplo que produce una excepción y visualiza un mensaje indicando el error:
 */
package ud01Excepciones;

public class EjemploExcepciones01 {

    public static void main(final String[] args) {
        int numerador = 10, denominador = 0, cociente;

        cociente = numerador/denominador;
        System.out.println("Resultado = "+cociente);
    }
}
```



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 Java está diseñado pensando en situaciones en las que el método que detecta un error no es capaz de manejarlo, un método así lanzará una *excepción*.

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

Captura de Excepciones

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

```
try{
    //código que genera la excepción
}
catch (Exception1 e1){
    //manejo de la excepción1
}
catch(Exception2 e2){
    //manejo de la excepción2
}
```

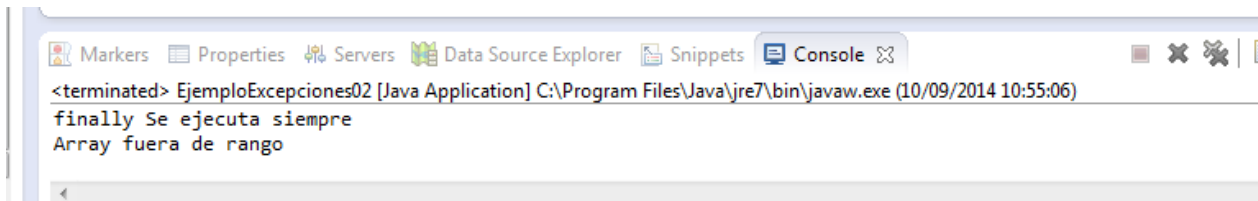
Etc.....

```
finally{  
    //se ejecuta después de try o catch  
}
```

El ejemplo captura 3 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 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:

```
/*  
 * El ejemplo captura 3 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 valor al elemento de la posición 10)  
 */  
package ud01Excepciones;  
  
public class EjemploExcepciones02 {  
  
    public static void main(final String[] args) {  
        String cad1 = "20", cad2 = "0", mensaje;  
  
        int numerador, denominador, cociente;  
        int[] arraynum = new int[4];  
  
        try{  
            //codigo que puede producir errores  
            arraynum[10] = 20; //sentencia que produce la excepcion  
            numerador = Integer.parseInt(cad1); //no se ejecuta  
            denominador = Integer.parseInt(cad2); //no se ejecuta  
            cociente = numerador/denominador; //no se ejecuta  
  
            mensaje = String.valueOf(cociente); //no se ejecuta  
        }catch(NumberFormatException nfe){  
            mensaje = "Caracteres no numéricos";  
        }catch (ArithmeticException ae) {  
            mensaje = "División por cero";  
        }catch (ArrayIndexOutOfBoundsException aio) {  
            mensaje = "Array fuera de rango";  
        }finally{  
            System.out.println("finally Se ejecuta siempre");  
        }  
        System.out.println(mensaje); //si se ejecuta  
    } // fin main  
} // fin clase
```

El programa muestra la siguiente pantalla:



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

```
try{
    //codigo que puede producir errores
}catch(NumberFormatException nfe){
    //tratamiento de la excepción
}catch (ArithmeticException ae) {
    //tratamiento de la excepción
}catch (ArrayIndexOutOfBoundsException aio) {
    //tratamiento de la excepción
}catch (Exception e) {
    //tratamiento si se produce cualquier otra excepción
}finally{
    //se ejecuta haya o no excepción
}
```

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

Método	Función
String getMessage()	Devuelve la cadena de error del código
String getLocalizedMessage()	Crea una descripción local de este objeto
String toString	Devuelve una breve descripción del objeto
void printStackTrace() void printStackTrace(PrintStream) void printStackTrace(PrintWriter)	Imprime el objeto y la traza de pila de llamadas lanzada.

El siguiente ejemplo utiliza dichos métodos:

```
/*
 * El ejemplo utiliza los métodos de la clase Throwable
 */
package ud01Excepciones;

public class EjemploExcepciones03 {

    public static void main(final String[] args) {
        String cad1 = "2a0", cad2 = "0", mensaje;

        int numerador, denominador, cociente;
        int[] arraynum = new int[4];

        try{
            //codigo que puede producir errores
```



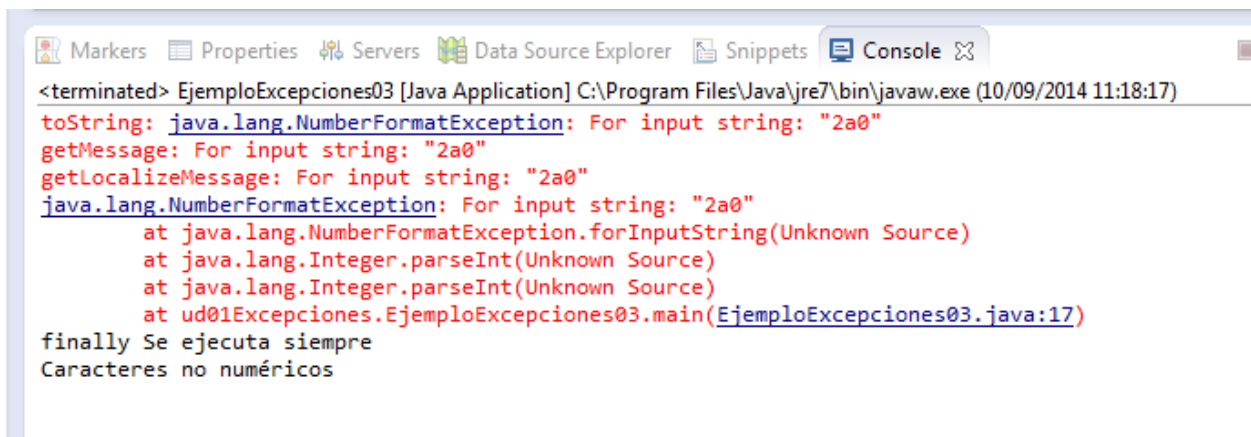
```

        arraynum[2] = 20; //sentencia que produce la excepcion
        numerador = Integer.parseInt(cad1); //no se ejecuta
        denominador = Integer.parseInt(cad2); //no se ejecuta
        cociente = numerador/denominador; //no se ejecuta

        mensaje = String.valueOf(cociente); //no se ejecuta
    }catch(NumberFormatException nfe){
        mensaje = "Caracteres no numéricos";
        System.err.println("toString: " +nfe.toString());
        System.err.println("getMessage: " +nfe.getMessage());
        System.err.println("getLocalizedMessage: "
+nfe.getLocalizedMessage());
        nfe.printStackTrace();
    }catch (ArithmeticException ae) {
        mensaje = "División por cero";
    }catch (ArrayIndexOutOfBoundsException aio) {
        mensaje = "Array fuera de rango";
    }finally{
        System.out.println("finally Se ejecuta siempre");
    }
    System.out.println(mensaje); //si se ejecuta
}
}

```

Muestra la siguiente salida al ejecutarse:



```

<terminated> EjemploExcepciones03 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (10/09/2014 11:18:17)
toString: java.lang.NumberFormatException: For input string: "2a0"
getMessage: For input string: "2a0"
getLocalizedMessage: For input string: "2a0"
java.lang.NumberFormatException: For input string: "2a0"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at ud01Excepciones.EjemploExcepciones03.main(EjemploExcepciones03.java:17)
finally Se ejecuta siempre
Caracteres no numéricos

```

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

Especificar Excepciones

Para especificar excepciones utilizamos la palabra **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 una excepción. El siguiente ejemplo indica que el método **main()** puede lanzar las excepciones **IOException** y **ClassNotFoundException**:

```

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

```

Los métodos que pueden lanzar excepciones, deben saber cuáles son esas excepciones en su declaración. Una forma típica de saberlo es compilando el programa. Por ejemplo, si al fichero **EjemploLecturaObjetos02.java** le quitamos la clausula **throws** del método **main()** y el bloque **try-catch** al compilar aparecerán errores:

```

17      System.out.println("Nombre \t Edad" );
18
19      fs = new FileInputStream("Personas01.txt");
20      os = new ObjectOutputStream(fs);
21      while(true){ // lectura del fichero
22          // os debe realizar un castingal tipo original
23          Persona p = (Persona)os.readObject();
24          System.out.println(p.getNombre() +"\t" +p.getEdad());
25      }
26
27      os.close();
28  }

```

Markers Properties Servers Data Source Explorer Snippets Console

<terminated> EjemploExcepciones04 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (10/09/2014 11:33:20)

Exception in thread "main" java.lang.Error: Unresolved compilation problems:

Unhandled exception type [FileNotFoundException](#)

Unhandled exception type [IOException](#)

Unhandled exception type [IOException](#)

Unhandled exception type [ClassNotFoundException](#)

Unreachable code

at ud01Excepciones.EjemploExcepciones04.main(EjemploExcepciones04.java:19)

Se producen errores de excepciones no declaradas. El ejemplo anterior manejando excepciones dentro del bloque try-catch quedaría:

```

package ud01Excepciones;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;

import ud01EjemplosObjetos.Persona;

public class EjemploExcepciones04 {
    public static void main(final String[] args) {
        FileInputStream fs = null;
        ObjectInputStream os = null;

        System.out.println("Nombre \t Edad" );
        try{
            fs = new FileInputStream("Personas01.txt");
            os = new ObjectOutputStream(fs);
            while(true){ // lectura del fichero
                // os debe realizar un castingal tipo original
                Persona p = (Persona)os.readObject();
                System.out.println(p.getNombre() +"\t" +p.getEdad());
            }
        }catch(ClassNotFoundException cnf){
            System.out.println("Error la clase");
        }catch(FileNotFoundException fnfe){
            System.out.println("Error en el fichero");
        }catch(IOException ioe){
            System.out.println("Error E/L");
        }finally{
            try{
                os.close();
            }catch(IOException ioe){
                System.out.println("Error E/L");
            }
        }
    }
}

```

```

    }
}
}

```

Se puede consultar la API de Java <http://docs.oracle.com/javase/7/docs/api/> para ver las excepciones que se pueden producir en los diferentes paquetes.

