

UNIDAD 1 MANEJO DE FICHEROS.

1.1. INTRODUCCIÓN	2
1.2. CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS	2
1.3. FLUJOS O STREAMS. TIPOS.	5
1.3.1. Flujos de bytes (Byte streams)	6
1.3.2. Flujos de caracteres (Character streams)	7
1.4. FORMAS DE ACCESO A UN FICHERO.	8
1.5. OPERACIONES SOBRE FICHEROS.	9
1.5.1. Operaciones sobre ficheros secuenciales.	10
1.5.2. Operaciones sobre ficheros aleatorios.	10

1.1. INTRODUCCIÓN

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

Los ficheros tienen un nombre y se ubican en directorios o carpetas, el nombre debe ser único en ese directorio.

Por convención cuentan con diferentes extensiones que por lo general suelen ser de 3 letras (PDF, DOC, GIF, ...) y nos permiten saber el tipo de fichero.

Un fichero está formado por un conjunto de registros o líneas y cada registro por un conjunto de campos relacionados, por ejemplo un fichero de empleados puede contener datos de los empleados de una empresa, un fichero de texto puede contener líneas de texto, correspondientes a líneas impresas en una hoja de papel.

1.2. CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS

Antes de ver las clases que leen y escriben datos en ficheros vamos a manejar la clase **File**.

Librería **java.io.***

Para crear un objeto **File**, se puede utilizar cualquiera de los tres constructores siguientes:

- **File(String directorioyfichero):** en Linux: *new File("/directorio/fichero.txt")*; en plataformas Microsoft Windows: *new File("C:\\directorio\\fichero.txt")*;
- **File(String directorio, String nombrefichero):** *new File("directorio", "fichero.txt")*;
- **File(File directorio, String fichero):** *new File(new File("directorio"), "fichero.txt")*;


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

```
//Windows
File fichero1 = new File( "C:\\EJERCICIOS\\UNI1\\ejemplo1.txt");
//Linux
File fichero1 = new File( "/home/ejercicios/un1/ejemplo1.txt");

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

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

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



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

Ejemplos:

- **Ver los ficheros y directorios de un directorio:**

Para indicar que estamos en el directorio actual creamos un objeto **File** y le pasamos la variable *dir* con el valor `"."`. Se define un segundo objeto **File** utilizando el tercer constructor para saber si el fichero obtenido es un fichero o un directorio:

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

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

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

La siguiente declaración mostraría la lista de ficheros del directorio *d:\db*:

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

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

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

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

Para poner la ubicación del fichero que se encuentra en la carpeta *src*, pondríamos, por ejemplo:

```
File f = new File("./src/principal.java");
```

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

Visualiza la siguiente información del fichero:

```
INFORMACIÓN SOBRE EL FICHERO:
Nombre del fichero    : VerInf.java
Ruta                 : D:\ADAT\UNI1\VerInf.java
Ruta absoluta        : D:\ADAT\UNI1\VerInf.java
Se puede leer        : true
Se puede escribir     : true
Tamaño               : 824
Es un directorio      : false
Es un fichero         : true
Nombre del directorio padre: D:\ADAT\UNI1
```

- El siguiente ejemplo crea un directorio (de nombre *NUEVODIR*) en el directorio actual, a continuación, crea dos ficheros vacíos en dicho directorio y uno de ellos lo renombra.

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

```

File f2 = new File(d, "FICHERO2.TXT");

d.mkdir(); //CREAR DIRECTORIO

try {
    if (f1.createNewFile())
        System.out.println("FICHERO1 creado correctamente...");
    else
        System.out.println("No se ha podido crear FICHERO1...");
    if (f2.createNewFile())
        System.out.println("FICHERO2 creado correctamente...");
    else
        System.out.println("No se ha podido crear FICHERO2...");
} catch (IOException ioe) {ioe.printStackTrace();}

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

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

```

Para borrar un fichero o un directorio usamos el método **delete()**, en el ejemplo anterior no podemos borrar el directorio creado porque contiene ficheros, antes habría que eliminarlos. Para borrar *FICHERO2* escribimos:

```

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

```

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

1.3. FLUJOS O STREAMS. TIPOS.

Paquete para el tratamiento de ficherosn **java.io**.

Se usan **flujos o 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 y/o enviar información de/a cualquier fuente necesita abrir un stream.

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)	<ul style="list-style-type: none"> - 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 descenden de las clases abstractas InputStream y OutputStream,
Flujos de caracteres (16 bits)	<ul style="list-style-type: none"> - Realizan operaciones de entradas y salidas de caracteres. (Un carácter en Java se representa con 16 bits) - El flujo de caracteres viene gobernado por las clases abstractas Reader y Writer que manejan flujo de caracteres UNICODE(16 bits).

1.3.1. Flujos de bytes (Byte streams)

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

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

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

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

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

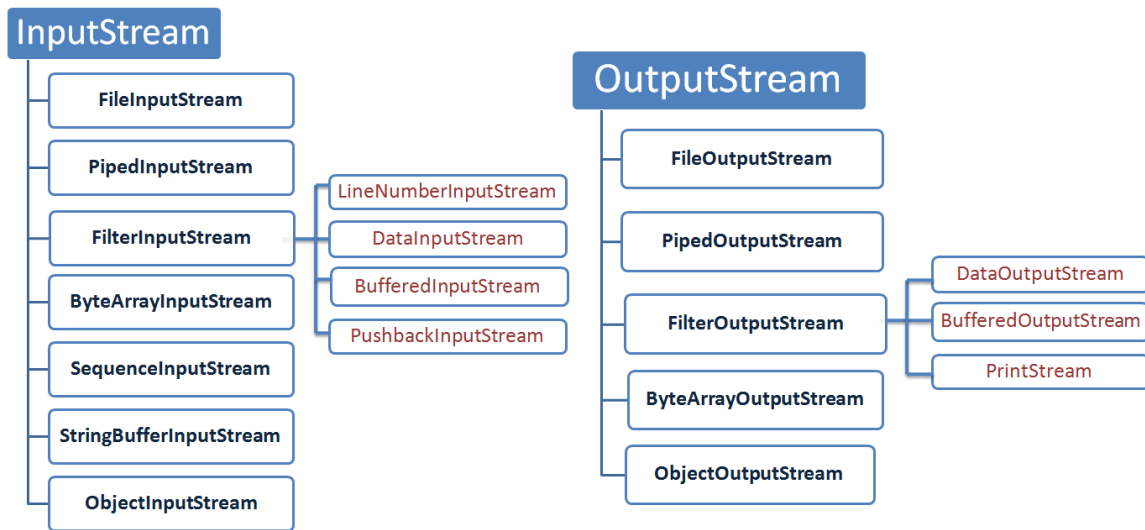


Figura 1.1. Jerarquía de clases para lectura y escritura de bytes.

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

1.3.2. Flujos de caracteres (Character streams)

Las clases **Reader** y **Writer** manejan flujos de caracteres Unicode.

Para lograr esto hay clases “puente” (es decir, convierte los streams de bytes a streams de caracteres):

InputStreamReader que convierte un **InputStream** en un **Reader** (convierte streams de bytes a streams de caracteres) y

OutputStreamWriter que convierte un **OutputStream** en un **Writer** (convierte streams de caracteres a streams de bytes).

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

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

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

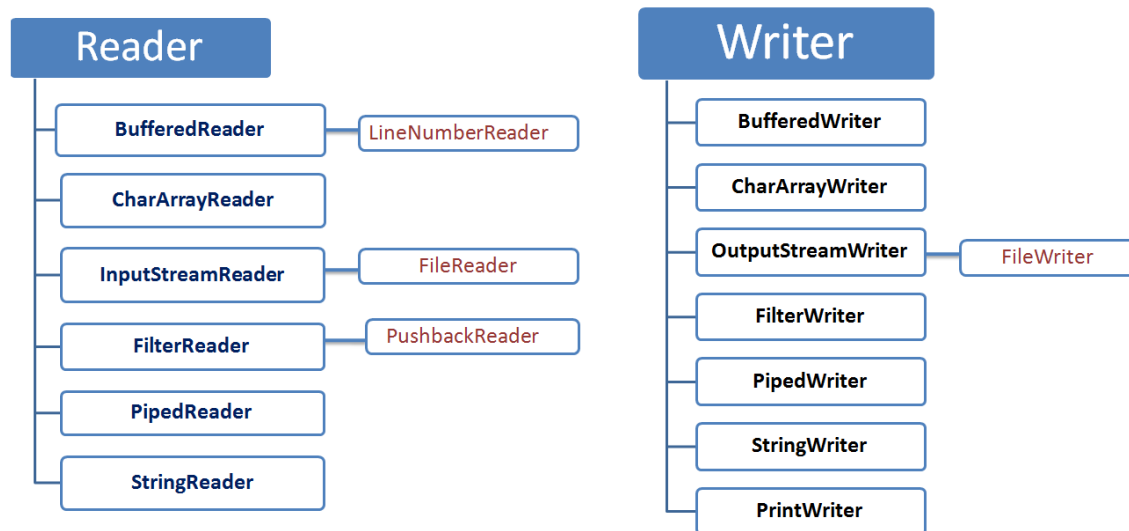


Figura 1.2. Jerarquía de clases para lectura y escritura de flujos de caracteres.

Las clases de flujos de caracteres más importantes son:

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

1.4. FORMAS DE ACCESO A UN FICHERO.

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

<p>Acceso secuencial</p>	<ul style="list-style-type: none"> - Los datos o registros se leen y se escriben en orden. - Para acceder a un dato, o a un reg. hay que leer los anteriores. - Se añaden datos o reg a partir del último. <p>Para acceder a un fichero de forma secuencial utilizaremos las clases:</p> <ul style="list-style-type: none"> - FileInputStream y FileOutputStream, para el acceso binario - FileReader y FileWriter, para el acceso a caracteres (texto). Utilizaremos estas clases para tratar los ficheros de texto.
<p>Acceso directo o aleatorio</p>	<ul style="list-style-type: none"> - Permite acceder directamente a un dato o registro sin necesidad de leer los anteriores. - Los datos se almacenan en reg. de tamaños conocidos. <p>Para el acceso aleatorio se utiliza la clase RandomAccessFile</p>

1.5. OPERACIONES SOBRE FICHEROS.

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

- **Creación del fichero.**
- **Apertura del fichero.**
- **Cierre del fichero.**
- **Lectura de los datos del fichero.**
- **Escritura de datos en el fichero.**

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

- **Altas.**
- **Bajas.**

- **Modificaciones.**
- **Consultas.**

1.5.1. Operaciones sobre ficheros secuenciales.

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

Veamos como se realizan las operaciones típicas:

- **Consultas:** la consulta es **secuencial**, para consultar un determinado registro es necesario empezar la lectura desde el primer registro, y leer secuencialmente hasta localizar el registro buscado.
- **Altas:** sólo se permite añadir datos al final del fichero.
- **Bajas:** para dar de baja un registro se leen todos los registros uno a uno y se escriben en un fichero auxiliar, salvo el que deseamos dar de baja. Luego se borra el fichero y se renombra el auxiliar.
- **Modificaciones:** El proceso es similar a las bajas, en este caso el registro se modifica.

VENTAJA: se utilizan en aplicaciones de proceso por lotes. Rápido acceso a los registros cuando se accede a ellos de forma secuencial. No se crean huecos, aprovechan mejor el espacio.

DESVENTAJA: NO se puede acceder directamente a un registro determinado, hay que leer antes todos los anteriores; es decir, no soporta acceso aleatorio. A la hora de actualizar se deben reescribir.

1.5.2. Operaciones sobre ficheros aleatorios.

Las operaciones en ficheros aleatorios son las vistas anteriormente, pero teniendo en cuenta que para acceder a un registro hay que localizar la posición o dirección donde se encuentra.

Normalmente para posicionarnos en un registro **es necesario aplicar una función de conversión que usualmente tiene que ver con el tamaño del registro y con la clave del mismo** (la clave es el campo o campos que identifica de forma unívoca a un registro).

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:** se necesita saber la clave y aplicar la función de conversión para obtener la dirección y leer el registro ubicado en esa posición.
- **Altas:** para insertar un registro necesitamos saber su clave, aplicar la función de conversión a la clave, obtener así la dirección y escribir el registro en la posición devuelta. Si la posición está ocupada por otro el registro se insertaría en la zona de **excedentes**.
- **Bajas:** las bajas suelen realizarse de forma lógica, es decir, se suele utilizar un campo del registro a modo de switch que tenga el valor 1 cuando el registro exista y le damos el valor 0 para darle de baja, físicamente el registro no desaparece del disco. Habría que localizar el registro a dar de baja a partir de su campo clave y reescribir en este campo el valor 0.
- **Modificaciones:** Se hace como en los anteriores casos, se localiza el registro con la clave y la función de conversión, se modifican los datos y se reescriben en esa posición.

VENTAJAS: el rápido acceso a una posición determinada para leer o escribir un registro.

INCONVENIENTES: Encontrar una función de conversión que no provoque duplicados. Se desaprovecha mucho espacio por los huecos que se generan.