

INDICE

<u>1.1</u>	<u>INTRODUCCIÓN</u>	<u>1</u>
<u>1.2</u>	<u>CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS</u>	<u>1</u>
<u>1.3</u>	<u>FLUJOS O STREAMS. TIPOS</u>	<u>6</u>
<u>1.3.1</u>	<u>FLUJOS DE BYTES (BYTE STREAMS)</u>	<u>6</u>
<u>1.3.2</u>	<u>FLUJOS DE CARACTERES (CHARACTER STREAMS)</u>	<u>7</u>
<u>1.4</u>	<u>FORMAS DE ACCESO A UN FICHERO</u>	<u>8</u>
<u>1.5</u>	<u>OPERACIONES SOBRE FICHEROS</u>	<u>9</u>
<u>1.5.1</u>	<u>OPERACIONES SOBRE FICHEROS SECUENCIALES</u>	<u>10</u>
<u>1.5.2</u>	<u>OPERACIONES SOBRE FICHEROS ALEATORIOS</u>	<u>10</u>
<u>1.6</u>	<u>ACTIVIDADES</u>	<u>11</u>

1.1 INTRODUCCIÓN

Un fichero o archivo es un conjunto de bits almacenados en un dispositivo, como por ejemplo, un disco duro. La ventaja de utilizar ficheros es que los datos que guardamos permanecen en el dispositivo aun cuando apaguemos el ordenador, es decir, no son volátiles.

Los ficheros tienen un nombre y se ubican en directorios o carpetas, el nombre debe ser único en ese directorio; es decir, no puede haber dos ficheros con el mismo nombre en el mismo directorio. Por convención cuentan con diferentes extensiones que por lo general suelen ser de 3 letras (PDF, DOC, GIF,...) y nos permiten saber el tipo de fichero.

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

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

1.2 CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS

El paquete **java.io** contiene las clases para manejar la entrada/salida en Java, por tanto, necesitaremos importar dicho paquete cuando trabajemos con ficheros. Antes de ver las clases que leen y escriben datos en ficheros vamos a manejar la clase **File**. Esta clase proporciona un conjunto de utilidades relacionadas con los ficheros que nos van a proporcionar in-

formación acerca de los mismos, su nombre, sus atributos, los directorios, etc. Puede representar el nombre de un fichero particular o los nombres de un conjunto de ficheros de un directorio, también se puede usar para crear un nuevo directorio o una trayectoria de directorios completa si esta no existe. Para crear un objeto **File**, se puede utilizar cualquiera de los tres constructores siguientes:

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

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

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

```

/*
 * Formas para declarar un fichero
 */
package ejemplo1;

import java.io.File;

public class Ejemplo1 {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        //WINDOWS
        File fichero1 = new File ("C:\\EJERCICIOS\\UN1\\ejemplo1.txt");

        //LINUX
        //File fichero1 = new File ("/home/ejercicios/unil/ejemplo1.txt");
        String directorio = "C:/EJERCICIOS/UN1";
        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 <code>File</code>
<code>File[] listFiles()</code>	Devuelve un array de objetos <code>File</code> conteniendo los ficheros que estén dentro del directorio representado por el objeto <code>File</code>
<code>String getName()</code>	Devuelve el nombre del fichero o directorio
<code>String getPath()</code>	Devuelve el camino relativo
<code>String getAbsolutePath()</code>	Devuelve el camino absoluto del fichero/directorio
<code>boolean exists()</code>	Devuelve <i>true</i> si el fichero/directorio existe
<code>boolean canWrite()</code>	Devuelve <i>true</i> si el fichero se puede escribir
<code>boolean canRead()</code>	Devuelve <i>true</i> si el fichero se puede leer
<code>boolean isFile()</code>	Devuelve <i>true</i> si el objeto <code>File</code> corresponde a un fichero normal
<code>boolean isDirectory()</code>	Devuelve <i>true</i> si el objeto <code>File</code> 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 <code>File</code> . Solo se creará si no existe
<code>boolean renameTo(File nuevoNombre);</code>	Renombra el fichero representado por el objeto <code>File</code> asignándole <i>nuevoNombre</i>
<code>boolean delete()</code>	Borra el fichero o directorio asociado al objeto <code>File</code>
<code>boolean createNewFile()</code>	Crea un nuevo fichero, vacío, asociado a <code>File</code> si y solo si no existe un fichero con dicho nombre
<code>String getParent()</code>	Devuelve el nombre del directorio padre, o <i>null</i> si no existe

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

```
package ejemplo2verdir;

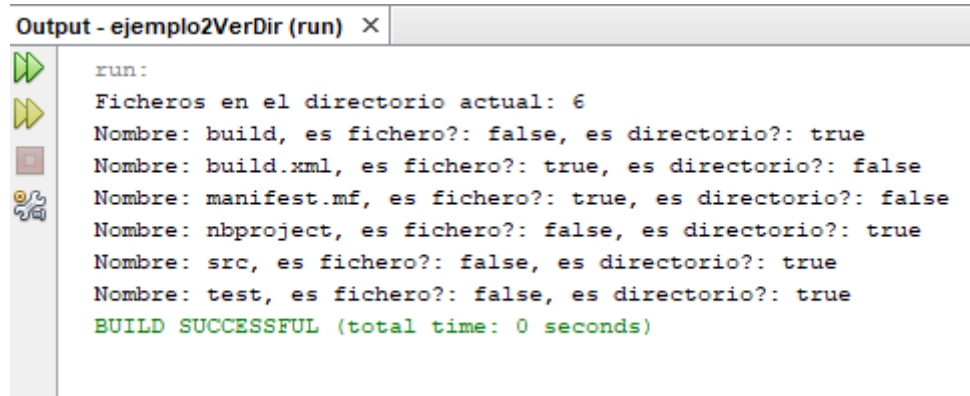
import java.io.File;

public class Ejemplo2VerDir {

    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 pantalla:



```

Output - ejemplo2VerDir (run) X
run:
Ficheros en el directorio actual: 6
Nombre: build, es fichero?: false, es directorio?: true
Nombre: build.xml, es fichero?: true, es directorio?: false
Nombre: manifest.mf, es fichero?: true, es directorio?: false
Nombre: nbproject, es fichero?: false, es directorio?: true
Nombre: src, es fichero?: false, es directorio?: true
Nombre: test, es fichero?: false, es directorio?: true
BUILD SUCCESSFUL (total time: 0 seconds)

```

La siguiente declaración aplicada al ejemplo anterior mostraría la lista de ficheros del directorio D:\IES CHAN DO MONTE

```
String dir = "D:\\IES CHAN DO MONTE";
```

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

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

```

public static void main(String[] args) {
    String dir=args[0];
    System.out.println("Archivos en el directorio " +dir);
    File f = new File(dir);
}

package ejemplo3verinf;

import java.io.File;

public class Ejemplo3VerInf {
    public static void main(String[] args) {
        System.out.println("INFORMACIÓN SOBRE EL FICHERO");
        File f = new File ("D:\\IES CHAN DO MONTE\\ADAT\\UNIl\\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 pued 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:

```
run:
INFORMACIÓN SOBRE EL FICHERO
Nombre del fichero      : VerInf.java
Ruta                   : D:\IES CHAN DO MONTE\ADAT\UNIl\VerInf.java
Ruta Absoluta          : D:\IES CHAN DO MONTE\ADAT\UNIl\VerInf.java
Se puede leer          : true
Se pued escribir       : true
Tamaño                 : 22
Es un directorio       : false
Es un fichero          : true
Nombre del directorio padre : D:\IES CHAN DO MONTE\ADAT\UNIl
BUILD SUCCESSFUL (total time: 0 seconds)
```

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

```
import java.io.File;
import java.io.IOException;

public class Ejemplo4CrearDir {

    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 eliminar estos ficheros. Para borrar el objeto *f2* escribimos:

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

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

1.3 FLUJOS O STREAMS. TIPOS

El sistema de entrada/salida en Java presenta una gran cantidad de clases que se implementan en el paquete `java.io`. Usa la abstracción del flujo (**stream**) para tratar la comunicación de información entre una fuente y un destino; dicha información puede estar en el fichero en el disco duro, en la memoria, en algún lugar de la red, e incluso en otro programa. Cualquier programa que tenga que obtener información de cualquier fuente necesita abrir un stream, igualmente si necesita enviar información abrirá un stream y se escribirá la información en serie. La vinculación de este stream al dispositivo físico lo 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 descenden de las clases `InputStream` y `OutputStream`, cada una de estas clases tienen varias subclases que controlan las diferencias entre los distintos dispositivos de entrada/salida que se pueden utilizar.
- **Flujos de caracteres (16 bits):** realizan operaciones de entradas y salidas de caracteres. El flujo de caracteres viene gobernado por las clases `Reader` y `Writer`. La razón de ser de estas clases es la internacionalización; la antigua jerarquía de flujos de E/S solo soportaba flujos de 8 bits no manejando caracteres Unicode de 16 bits que se utilizaba con fines de internacionalización.

1.3.1 Flujos de bytes (Byte streams)

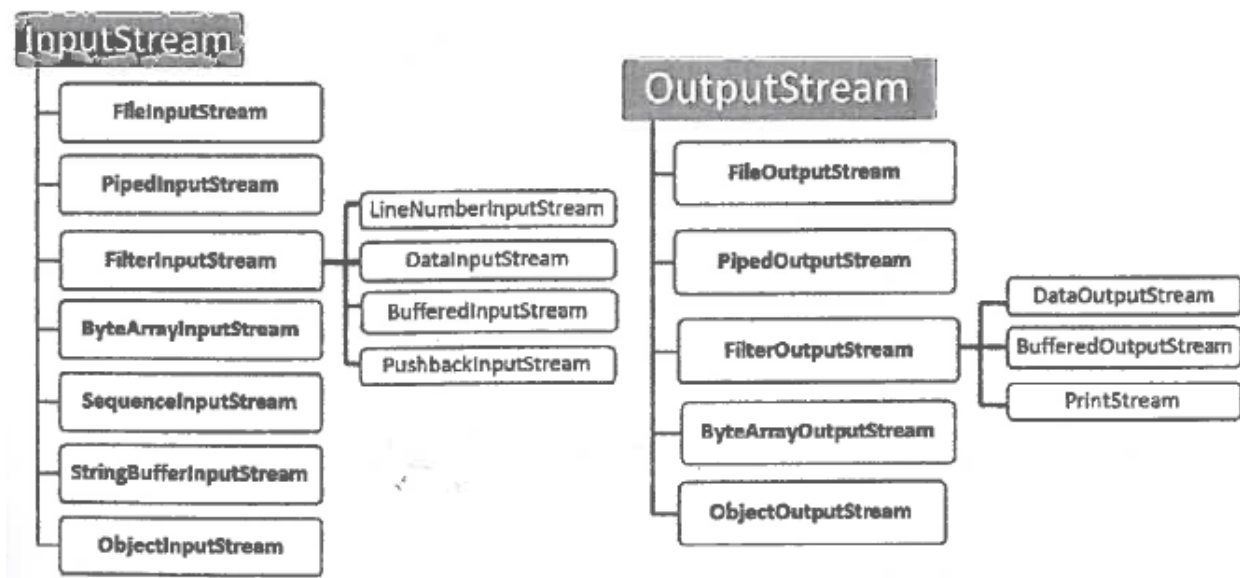
La clase `InputStream` representa las clases que producen entradas de distintas fuentes, estas fuentes pueden ser: un array de bytes, un objeto `String`, un fichero, una “tubería” (se ponen los elementos en un extremo y salen por el otro), una secuencia de otros flujos, otras fuentes como una conexión a Internet, etc. Los tipos `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>	Flujo de entrada hacia 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 <code>InputStream</code>
<code>SequenceInputStream</code>	Convierte dos o más objetos <code>InputStream</code> en un <code>InputStream</code> único

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

CLASE	Función
<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 <code>PipedInputStream</code> asociado. Implementa el concepto de “tubería”
<code>FilterOutputStream</code>	Proporciona funcionalidad útil a otras clases <code>OutputStream</code>

La imagen anterior muestra la **jerarquía de clases para lectura y escritura de flujos de bytes**.



Jerarquía de clases para lectura y escritura de bytes

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

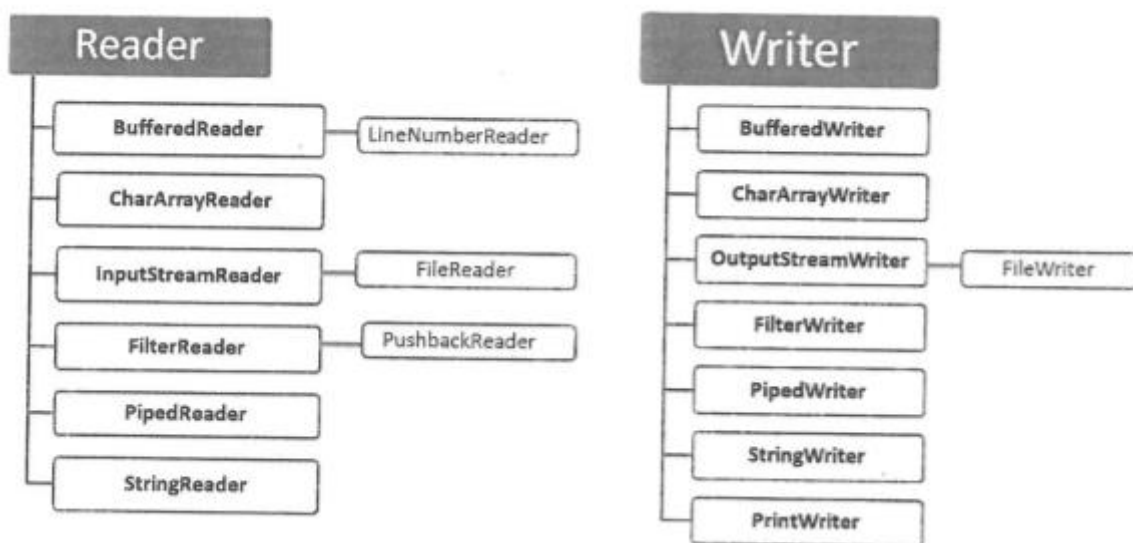
1.3.2 Flujos de caracteres (Character streams)

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

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

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

La imagen anterior muestra la **jerarquía de clases para lectura y escritura de flujos de caracteres**.



Jerarquía de clases para lectura y escritura de flujos de caracteres

Las clases de flujos de caracteres más importantes son:

- Para acceso a ficheros, lectura y escritura de caracteres en ficheros: **FileReader** y **FileWriter**:
- Para acceso a caracteres, leen y escriben un flujo de caracteres en un array de caracteres. **CharArrayReader** y **CharArrayWriter**.

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

1.4 FORMAS DE ACCESO A UN FICHERO

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

- **Acceso secuencial:** los datos o registros se leen y se escriben en orden, del mismo modo que se hace en una antigua cinta de vídeo. Si se quiere acceder a un dato o un registro que está hacia la mitad del fichero es necesario leer antes

todos los anteriores. La escritura de datos se hará a partir del último dato escrito, no es posible hacer inserciones entre los datos que ya hay escritos.

- **Acceso directo o aleatorio:** permite acceder directamente a un dato o registro sin necesidad de leer los anteriores y se puede acceder a la información en cualquier orden. Los datos están almacenados en registros de tamaño conocido, nos podemos mover de un registro a otro de forma aleatoria para leerlos o modificarlos.

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

1.5 OPERACIONES SOBRE FICHEROS

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

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

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

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

1.5.1 Operaciones sobre ficheros secuenciales

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

Veamos cómo se realizan las operaciones típicas:

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

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

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

1.5.2 Operaciones sobre ficheros aleatorios

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

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

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

Veamos cómo se realizan las operaciones típicas:

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

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

1.6 ACTIVIDADES

Actividad 1.1

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

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