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

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

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

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

```
//Windows
File fichero1 = new File( "C:\\EJERCICIOS\\UNI1\\ejemplo1.txt");
//Linux
File fichero1 = new File( "/home/ejercicios/uni1/ejemplo1.txt");

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

File direc = new File(directorio);
```

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

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

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

```
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 aplicada al ejemplo anterior mostraría la lista de ficheros del directorio *d:\db*:

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

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

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

ACTIVIDAD 1.1

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.

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

```
import java.io.*;
public class VerInf {
public static void main(String[] args) {
  System.out.println("INFORMACIÓN SOBRE EL FICHERO:");
  File f = new File("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. En este caso para crear los ficheros se definen 2 parámetros en el objeto **File**: *File(File directorio, String nombrefich)*, en el primero indicamos el directorio donde se creará el fichero y en el segundo indicamos el nombre del fichero:

```
import java.io.*;
public class CrearDir {
 public static void main(String[] args) {
   File d = new File("NUEVODIR"); //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 un fichero en el disco duro, en la memoria, en algún lugar de la red, e incluso en otro programa. Cualquier programa que tenga que obtener información de cualquier fuente necesita abrir un stream, igualmente si necesita enviar información abrirá un stream y se escribirá la información en serie. La vinculación de este stream al dispositivo físico la hace el sistema de entrada y salida de Java. Se definen dos tipos de flujos:

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

1.3.1. Flujos de bytes (Byte streams)

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

CLASE	Función
`ByteArrayInputStream`	Permite usar un espacio de almacenamiento intermedio de memoria
`StringBufferInputStream`	Convierte un String en un **InputStream**
`FileInputStream`	Flujo de entrada hacia fichero, lo usaremos para leer información de un fichero
`PipedInputStream`	Implementa el concepto de "tubería"
`FilterInputStream`	Proporciona funcionalidad útil a otras clases **InputStream**
`SequenceInputStream`	Convierte dos o más objetos **InputStream** en un **InputStream** único

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

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

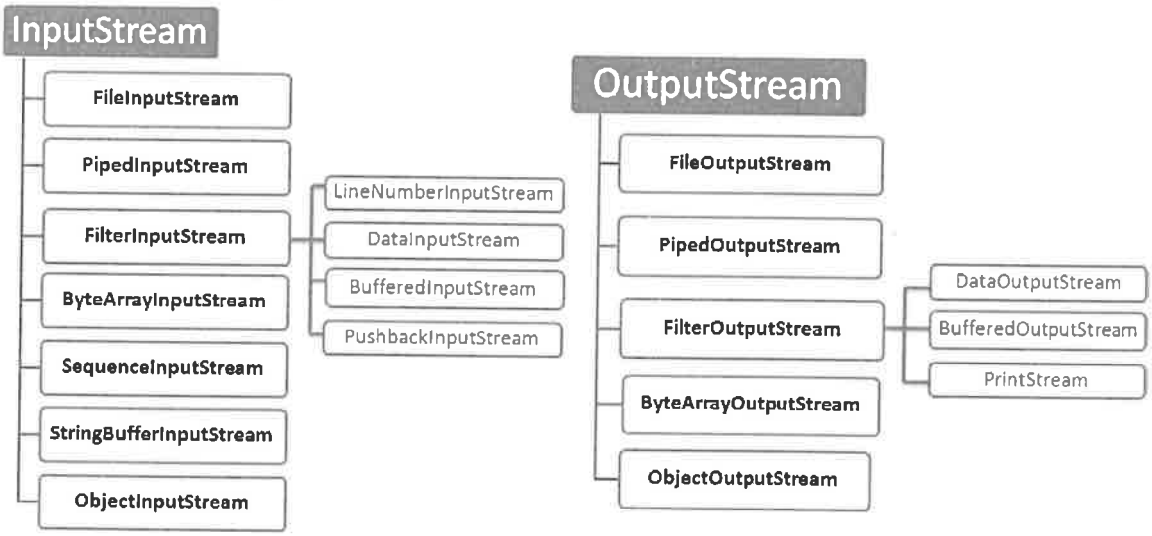

Figura 1.1. Jerarquía de clases para lectura y escritura de bytes.

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

1.3.2. Flujos de caracteres (Character streams)

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

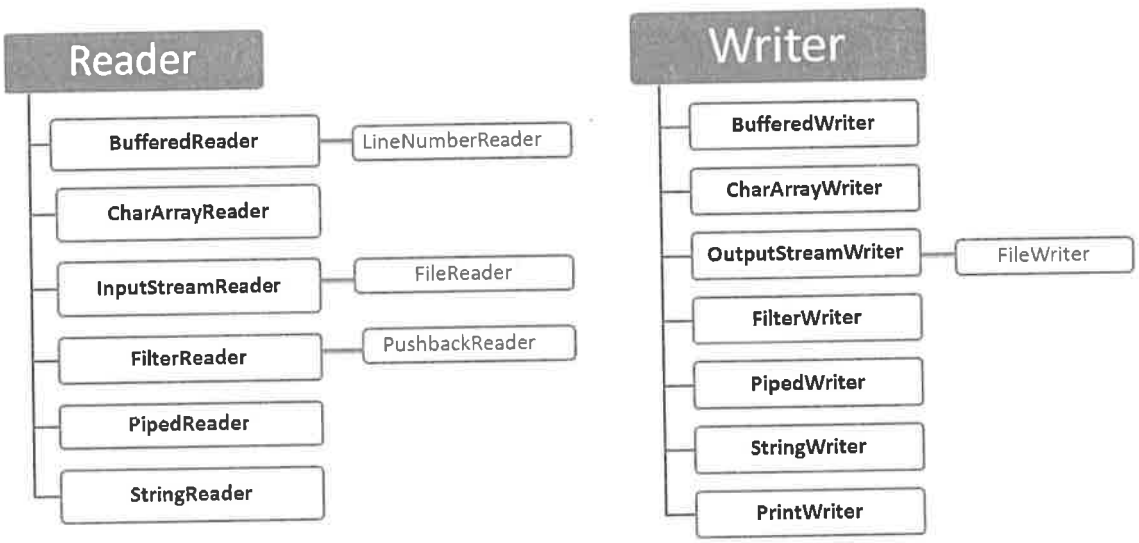

Figura 1.2. Jerarquía de clases para lectura y escritura de flujos de caracteres.

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:

- **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:

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

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

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

1.5.2. Operaciones sobre ficheros aleatorios

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

Normalmente para posicionarnos en un registro es necesario aplicar una función de conversión, que usualmente tiene que ver con el tamaño del registro y con la clave del mismo (la clave es el campo o campos que identifica de forma 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 con identificador X necesitamos acceder a la posición *tamaño*(X-1)* para acceder a los datos de dicho empleado.

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

Veamos cómo se realizan las operaciones típicas:

- **Consultas**: para consultar un determinado registro necesitamos saber su clave, aplicar la función de conversión a la clave para obtener la dirección y leer el registro ubicado en esa posición. Habría que comprobar si el registro buscado está en esta posición, si no está, se buscaría en la zona de excedentes.
- **Altas**: para insertar un registro necesitamos saber su clave, aplicar la función de conversión a la clave para obtener la dirección y escribir el registro en la posición devuelta. Si la posición está ocupada por otro registro, en ese caso el registro se insertaría en la zona de excedentes.
- **Bajas**: las bajas suelen realizarse de forma lógica, es decir, se suele utilizar un campo del registro a modo de switch que tenga el valor 1 cuando el registro 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. CLASES PARA GESTIÓN DE FLUJOS DE DATOS DESDE/HACIA FICHEROS

En Java podemos utilizar dos tipos de ficheros: de texto o binarios; y el acceso a los mismos se puede realizar de forma secuencial o aleatoria. Los ficheros de texto están compuestos de caracteres legibles, mientras que los binarios pueden almacenar cualquier tipo de dato (*int, float, boolean*, etc.)

1.6.1. Ficheros de texto

Los ficheros de texto, los que normalmente se generan con un editor, almacenan caracteres alfanuméricos en un formato estándar (ASCII, UNICODE, UTF8, etc.) Para trabajar con ellos usaremos las clases **FileReader** para leer caracteres y **FileWriter** para escribir los caracteres en el fichero. Cuando trabajamos con ficheros, cada vez que leemos o escribimos en uno debemos hacerlo dentro de un manejador de excepciones **try-catch**. Al usar la clase **FileReader** se puede generar la excepción ***FileNotFoundException*** (porque el nombre del fichero no exista o no sea válido) y al usar la clase **FileWriter** la excepción ***IOException*** (el disco está lleno o protegido contra escritura).

Los métodos que proporciona la clase **FileReader** para lectura son los siguientes, estos métodos devuelven el número de caracteres leídos o -1 si se ha llegado al final del fichero:

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

En un programa Java para crear o abrir un fichero se invoca a la clase **File** y a continuación se crea el flujo de entrada hacia el fichero con la clase **FileReader**. Después se realizan las operaciones de lectura o escritura y cuando terminemos de usarlo lo cerraremos mediante el método ***close()***.

El siguiente ejemplo lee cada uno de los caracteres del fichero de texto de nombre *LeerFichTexto.java* (localizado en la carpeta *C:\EJERCICIOS\UNI1*) y los muestra en pantalla, los métodos ***read()*** pueden lanzar la excepción **IOException**, por ello en ***main()*** se ha añadido ***throws IOException*** ya que no se incluye el manejador **try-catch**:

```
import java.io.*;
public class LeerFichTexto {
  public static void main(String[] args) throws IOException {
    //declarar fichero
    File fichero =
         new File("C:\\EJERCICIOS\\UNI1\\LeerFichTexto.java");
    //crear el flujo de entrada hacia el fichero
    FileReader fic = new FileReader(fichero);
    int i;
    while ((i = fic.read()) != -1) //se va leyendo un carácter
      System.out.println((char) i);
    fic.close(); //cerrar fichero
  }
}
```

En el ejemplo, la expresión *((char) i)* convierte el valor entero recuperado por el método ***read()*** a carácter, es decir, hacemos un *cast* a *char*. Se llega al final del fichero cuando el método ***read()*** devuelve -1. También se puede declarar el fichero de la siguiente manera:

```
FileReader fic =
          new FileReader("C:\\EJERCICIOS\\UNI1\\LeerFichTexto.java");
```

Para ir leyendo de 20 en 20 caracteres escribimos:

```
char b[]= new char[20];
while ((i = fic.read(b)) != -1) System.out.println(b);
```

ACTIVIDAD 1.2

Crea un fichero de texto con algún editor de textos y después realiza un programa Java que visualice su contenido. Cambia el programa Java para que el nombre del fichero se acepte al ejecutar el programa desde la línea de comandos.

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

Método	Función
`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* comenzando por *buf[desplazamiento]*.
`void write(String str)`	Escribe una cadena de caracteres.
`void append(char c)`	Añade un carácter a un fichero.

Estos métodos también pueden lanzar la excepción ***IOException***. Igual que antes declaramos el fichero mediante la clase **File** y a continuación se crea el flujo de salida hacia el fichero con la clase **FileWriter**. El siguiente ejemplo escribe caracteres en un fichero de nombre *FichTexto.txt* (si no existe lo crea). Los caracteres se escriben uno a uno y se obtienen de un *String* que se convierte en array de caracteres:

```
import java.io.*;
public class EscribirFichTexto {
  public static void main(String[] args) throws IOException {
   File fichero = new
      File("C:\\EJERCICIOS\\UNI1\\FichTexto.txt");//declarar fichero
   //crear flujo de salida
   FileWriter fic = new FileWriter(fichero);

   String cadena ="Esto es una prueba con FileWriter";
   //convierte la cadena en array de caracteres para extraerlos 1 a 1
   char[] cad = cadena.toCharArray();
   for(int i=0; i<cad.length; i++)
         fic.write(cad[i]);  //se va escribiendo un carácter

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

En vez de escribir los caracteres uno a uno, también podemos escribir todo el array usando ***fic.write(cad)***. El siguiente ejemplo escribe cadenas de caracteres que se obtienen de un array de String, las cadenas se irán insertando en el fichero una a continuación de la otra sin saltos de línea:

```
String prov[] =
    {"Albacete","Avila","Badajoz","Cáceres","Huelva","Jaén",
     "Madrid","Segovia","Soria","Toledo","Valladolid","Zamora"};

for(int i=0; i<prov.length; i++) fic.write(prov[i]);
```

Hay que tener en cuenta que si el fichero existe cuando vayamos a escribir caracteres sobre él, todo lo que tenia almacenado anteriormente se borrará. Si queremos añadir caracteres al final, usaremos la clase **FileWriter** de la siguiente manera, colocando en el segundo parámetro del constructor el valor ***true***:

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

FileReader no contiene métodos que nos permitan leer líneas completas, pero **BufferedReader** sí; dispone del método ***readLine()*** que lee una línea del fichero y la devuelve, o devuelve *null* si no hay nada que leer o llegamos al final del fichero. También dispone del método ***read()*** para leer un carácter. Para construir un **BufferedReader** necesitamos la clase **FileReader**:

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

El siguiente ejemplo lee el fichero *LeerFichTexto.java* línea por línea y las va visualizando en pantalla, en este caso, las instrucciones se han agrupado dentro de un bloque **try-catch**:

```
import java.io.*;
public class LeerFichTextoBuf {
  public static void main(String[] args) {
  try{
      BufferedReader fichero = new BufferedReader(
                       new FileReader("LeerFichTexto.java"));
      String linea;
      while((linea = fichero.readLine())!=null)
        System.out.println(linea);

      fichero.close();
      }
      catch (FileNotFoundException fn ){
               System.out.println("No se encuentra el fichero");}
      catch (IOException io) {
               System.out.println("Error de E/S ");}
  }
}
```

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

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

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

```
import java.io.*;
public class EscribirFichTextoBuf {
  public static void main(String[] args) {
  try{
      BufferedWriter fichero = new BufferedWriter
```

```
                                (new FileWriter("FichTexto.txt"));
        for (int i=1; i<11; i++){
            fichero.write("Fila numero: "+i); //escribe una línea
            fichero.newLine(); //escribe un salto de línea
        }
        fichero.close();
        }
        catch (FileNotFoundException fn ){
                System.out.println("No se encuentra el fichero");}
        catch (IOException io) {
                System.out.println("Error de E/S ");}
    }
}
```

La clase **PrintWriter**, que también deriva de **Writer**, posee los métodos ***print(String)*** y ***println(String)*** (idénticos a los de *System.out*) para escribir en un fichero. Ambos reciben un *String* y lo escriben en un fichero, el segundo método, además, produce un salto de línea. Para construir un **PrintWriter** necesitamos la clase **FileWriter**:

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

El ejemplo anterior usando la clase **PrintWriter** y el método ***println()*** quedaría así:

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

1.6.2. Ficheros binarios

Los ficheros binarios almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario como ocurría con los ficheros de texto. Tienen la ventaja de que ocupan menos espacio en disco. En Java, las dos clases que nos permiten trabajar con ficheros son **FileInputStream** (para entrada) y **FileOutputStream** (para salida), estas trabajan con flujos de bytes y crean un enlace entre el flujo de bytes y el fichero.

Los métodos que proporciona la clase **FileInputStream** para lectura son similares a los vistos para la clase **FileReader**, estos métodos devuelven el número de bytes leídos o -1 si se ha llegado al final del fichero:

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

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

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 comenzando por *b[desplazamiento]*

El siguiente ejemplo escribe bytes en un fichero y después los visualiza:

```
import java.io.*;
public class EscribirFichBytes {
  public static void main(String[] args) throws IOException {
   File fichero = new
        File("C:\\EJERCICIOS\\UNI1\\FichBytes.dat");//declara fichero
   //crea flujo de salida hacia el fichero
   FileOutputStream fileout = new FileOutputStream(fichero);

   //crea flujo de entrada
   FileInputStream filein = new FileInputStream(fichero);
   int i;

   for (i=1; i<100; i++)
       fileout.write(i); //escribe datos en el flujo de salida

   fileout.close(); //cerrar stream de salida

   //visualizar los datos del fichero
   while ((i = filein.read()) != -1) //lee datos del flujo de entrada
          System.out.println(i);
   filein.close();  //cerrar stream de entrada
  }
}
```

Para añadir bytes al final del fichero usaremos **FileOutputStream** de la siguiente manera, colocando en el segundo parámetro del constructor el valor ***true***:

```
FileOutputStream fileout = new FileOutputStream(fichero,true);
```

Para leer y escribir datos de tipos primitivos: *int, float, long*, etc usaremos las clases **DataInputStream** y **DataOutputStream**. Estas clases definen diversos métodos *readXXX* y *writeXXX* que son variaciones de los métodos ***read()*** y ***write()*** de la clase base para leer y escribir datos de tipo primitivo. Algunos de los métodos se muestran en la siguiente tabla:

MÉTODOS PARA LECTURA	MÉTODOS PARA ESCRITURA
`boolean readBoolean();`	`void writeBoolean(boolean v);`
`byte readByte();`	`void writeByte(int v);`
`int readUnsignedByte();`	`void writeBytes(String s);`
`int readUnsignedShort();`	`void writeShort(int v);`
`short readShort();`	`void writeChars(String s);`
`char readChar();`	`void writeChar(int v);`
`int readInt();`	`void writeInt(int v);`
`long readLong();`	`void writeLong(long v);`
`float readFloat();`	`void writeFloat(float v);`
`double readDouble();`	`void writeDouble(double v);`
`String readUTF();`	`void writeUTF(String str);`

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

```
File fichero = new File("FichData.dat");
FileInputStream filein = new FileInputStream(fichero);
DataInputStream dataIS = new DataInputStream(filein);
```

O bien

```
File fichero = new File("FichData.dat");
DataInputStream dataIS = new
                DataInputStream(new FileInputStream(fichero));
```

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

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

O bien

```
File fichero = new File("FichData.dat");
DataOutputStream dataOS = new
                DataOutputStream(new FileOutputStream(fichero));
```

El siguiente ejemplo inserta datos en el fichero *FichData.dat*, los datos los toma de dos arrays, uno contiene los nombres de una serie de personas y el otro sus edades, recorremos los arrays y vamos escribiendo en el fichero el nombre (mediante el método ***writeUTF(String)***) y la edad (mediante el método ***writeInt(int)***):

```
import java.io.*;
public class EscribirFichData {
  public static void main(String[] args) throws IOException {

   File fichero = new File("FichData.dat");
   FileOutputStream fileout = new FileOutputStream(fichero);
   DataOutputStream dataOS = new DataOutputStream(fileout);

   String nombres[] =
            {"Ana","Luis Miguel","Alicia","Pedro","Manuel",
             "Andrés", "Julio","Antonio","María Jesús"};

   int edades[] = {14,15,13,15,16,12,16,14,13};

   for (int i=0;i<edades.length; i++){
      dataOS.writeUTF(nombres[i]); //escribe nombre
      dataOS.writeInt(edades[i]);  //escribe edad
   }
   dataOS.close();  //cerrar stream
  }
}
```

El siguiente ejemplo visualiza los datos grabados anteriormente en el fichero, se deben recuperar en el mismo orden en el que se escribieron, es decir, primero obtenemos el nombre y luego la edad:

```
import java.io.*;
public class LeerFichData {
  public static void main(String[] args) throws IOException {
   File fichero = new File("FichData.dat");
   FileInputStream filein = new FileInputStream(fichero);
   DataInputStream dataIS = new DataInputStream(filein);
   String n;
   int e;

   try {
    while (true) {
        n = dataIS.readUTF(); //recupera el nombre
        e = dataIS.readInt(); //recupera la edad
        System.out.println("Nombre: " + n + ", edad: " + e);
    }
    }catch (EOFException eo) {}

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

Se obtiene la siguiente salida al ejecutar el programa:

```
Nombre: Ana, edad: 14
Nombre: Luis Miguel, edad: 15
Nombre: Alicia, edad: 13
Nombre: Pedro, edad: 15
Nombre: Manuel, edad: 16
Nombre: Andrés, edad: 12
Nombre: Julio, edad: 16
Nombre: Antonio, edad: 14
Nombre: María Jesús, edad: 13
```

1.6.3. Objetos en ficheros binarios

Hemos visto cómo se guardan los tipos de datos primitivos en un fichero, pero, por ejemplo, si tenemos un objeto de tipo empleado con varios atributos (el nombre, la dirección, el salario, el departamento, el oficio, etc.) y queremos guardarlo en un fichero, tendríamos que guardar cada atributo que forma parte del objeto por separado, esto se vuelve engorroso si tenemos gran cantidad de objetos. Por ello Java nos permite guardar objetos en ficheros binarios; para poder hacerlo, el objeto tiene que implementar la interfaz **Serializable** que dispone de una serie de métodos con los que podremos guardar y leer objetos en ficheros binarios. Los más importantes a utilizar son:

- **Object readObject():** se utiliza para leer un objeto del **ObjectInputStream**. Puede lanzar las excepciones ***IOException*** y ***ClassNotFoundException***.
- **void writeObject(Object obj):** se utiliza para escribir el objeto especificado en el **ObjectOutputStream.** Puede lanzar la excepción ***IOException***.

La serialización de objetos de Java permite tomar cualquier objeto que implemente la interfaz **Serializable** y convertirlo en una secuencia de bits que puede ser posteriormente restaurada para regenerar el objeto original.

Para leer y escribir objetos serializables a un stream se utilizan las clases Java **ObjectInputStream** y **ObjectOutputStream** respectivamente. A continuación se muestra la clase *Persona* que implementa la interfaz **Serializable** y que utilizaremos para escribir y leer objetos en un fichero binario. La clase tiene dos atributos: el nombre y la edad y los métodos *get* para obtener el valor del atributo y *set* para darle valor:

```
import java.io.Serializable;
public class Persona implements Serializable{
      private String nombre;
      private int edad;

      public Persona(String nombre,int edad)     {
        this.nombre = nombre;
        this.edad = edad;
       }
      public Persona() {
        this.nombre = null;
       }
      public void setNombre(String nombre){this.nombre = nombre;}
      public void setEdad(int edad){this.edad = edad;}

      public String getNombre(){return this.nombre;}//devuelve nombre
      public int getEdad(){return this.edad;}        //devuelve edad

}//fin Persona
```

El siguiente ejemplo escribe objetos *Persona* en un fichero. Necesitamos crear un flujo de salida a disco con **FileOutputStream** y a continuación se crea el flujo de salida **ObjectOutputStream** que es el que procesa los datos y se ha de vincular al fichero de **FileOutputStream**:

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

O bien:

```
File fichero = new File("FichPersona.dat");
ObjectOutputStream dataOS = new
                  ObjectOutputStream(new FileOutputStream(fichero));
```

El método **writeObject()** escribe los objetos al flujo de salida y los guarda en un fichero en disco: ***dataOS.writeObject(persona)***. El código es el siguiente:

```
import java.io.*;
public class EscribirFichObject {
  public static void main(String[] args) throws IOException {
   Persona persona;//defino variable persona
   //declara el fichero
   File fichero = new File("FichPersona.dat");
```

```
    //crea el flujo de salida
    FileOutputStream fileout = new FileOutputStream(fichero);
    //conecta el flujo de bytes al flujo de datos
    ObjectOutputStream dataOS = new ObjectOutputStream(fileout);

    String nombres[] = {"Ana","Luis Miguel", "Alicia",  "Pedro",
            "Manuel",  "Andrés",  "Julio",  "Antonio",  "María  Jesús"};

    int edades[] = {14,15,13,15,16,12,16,14,13};

    for (int i=0;i<edades.length; i++){ //recorro los arrays
        persona= new Persona(nombres[i],edades[i]);
        dataOS.writeObject(persona); //escribo la persona en el fichero
    }
    dataOS.close();  //cerrar stream de salida
    }
}
```

Para leer objetos *Persona* del fichero necesitamos el flujo de entrada a disco **FileInputStream** y a continuación crear el flujo de entrada **ObjectInputStream** que es el que procesa los datos y se ha de vincular al fichero de **FileInputStream**:

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

O bien:

```
File fichero = new File("FichPersona.dat");
ObjectInputStream dataIS = new
                  ObjectInputStream(new FileInputStream(fichero));
```

El método ***readObject()*** lee los objetos del flujo de entrada, puede lanzar la excepción ***ClassNotFoundException*** e ***IOException***, por lo que será necesario controlarlas. El proceso de lectura se hace en un bucle *while(true)*, este se encierra en un bloque **try-catch** ya que la lectura finalizará cuando se llegue al final de fichero, entonces, se lanzará la excepción ***EOFException.*** El código es el siguiente:

```
import java.io.*;

public class LeerFichObject {
  public static void main(String[] args) throws
                    IOException, ClassNotFoundException{
    Persona persona;   //defino la variable persona
    File fichero = new File("FichPersona.dat");
    //crea el flujo de entrada
    FileInputStream filein = new FileInputStream(fichero);
    //conecta el flujo de bytes al flujo de datos
    ObjectInputStream dataIS = new ObjectInputStream(filein);

    try {
      while (true) { //lectura del fichero
        persona= (Persona) dataIS.readObject(); //leer una Persona
        System.out.printf("Nombre: %s, edad: %d %n",
```

```
                    persona.getNombre(),persona.getEdad());
      }
    }catch (EOFException eo) {
       System.out.println("FIN DE LECTURA.");
    }

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

Problema con los ficheros de objetos:

Existe un problema con los ficheros de objetos. Al crear un fichero de objetos se crea una cabecera inicial con información, y a continuación se añaden los objetos. Si el fichero se utiliza de nuevo para añadir más registros, se crea una nueva cabecera y se añaden los objetos a partir de esa cabecera. El problema surge al leer el fichero cuando en la lectura se encuentra con la segunda cabecera, y aparece la excepción **StreamCorruptedException** y no podremos leer más objetos.

La cabecera se crea cada vez que se pone ***new ObjectOutputStream(fichero).*** Para que no se añadan estas cabeceras lo que se hace es ***reedefinir la clase ObjectOutputStream creando una nueva clase que la herede (extends).*** Y dentro de esa clase se redefine el método ***writeStreamHeader()*** que es el que escribe las cabeceras, y hacemos que ese método no haga nada. De manera que si el fichero ya se ha creado se llamará a ese método de la clase redefinida.

La clase redefinida quedará así:

```
public class MiObjectOutputStream extends ObjectOutputStream
{
    public MiObjectOutputStream(OutputStream out) throws IOException
    {    super(out);     }
    protected MiObjectOutputStream()
            throws IOException, SecurityException
    {   super();    }
    // Redefinición del método de escribir la cabecera
    // para que no haga nada.
    protected void writeStreamHeader() throws IOException
    {  }
}
```

Y dentro de nuestro programa a la hora de abrir el fichero para añadir nuevos objetos se pregunta si ya existe, si existe, se crea el objeto con la clase redefinida, y si no existe, el fichero se crea con la clase ***ObjectOutputStream***:

```
File fichero = new File(nombrefichero);
ObjectOutputStream dataOS;
if (!fichero.exists())
{  //Si el fichero no existe crea un ObjectOutputStream, la primera vez
    FileOutputStream fileout;
    fileout = new FileOutputStream(fichero);
    dataOS = new ObjectOutputStream(fileout);
}
else
{   // Si ya existe el fichero creará un ObjectOutputStream
```

```
    // con el método writeStreamHeader redefinido (sin hacer nada)
    dataOS = new MiObjectOutputStream
                (new FileOutputStream(fichero,true));
} //fin if
```

1.6.4. Ficheros de acceso aleatorio

Hasta ahora, todas las operaciones que hemos realizado sobre los ficheros se realizaban de forma secuencial. Se empezaba la lectura en el primer byte o el primer carácter o el primer objeto, y seguidamente se leían los siguientes uno a continuación de otro hasta llegar al fin del fichero. Igualmente cuando escribíamos los datos en el fichero se iban escribiendo a continuación de la última información escrita. Java dispone de la clase **RandomAccessFile** que dispone de métodos para acceder al contenido de un fichero binario de forma aleatoria (no secuencial) y para posicionarnos en una posición concreta del mismo. Esta clase no es parte de la jerarquía **InputStream/OutputStream**, ya que su comportamiento es totalmente distinto puesto que se puede avanzar y retroceder dentro de un fichero.

Disponemos de dos constructores para crear el fichero de acceso aleatorio, estos pueden lanzar la excepción ***FileNotFoundException***:

- ***RandomAccessFile(String nombrefichero, String modoAcceso):*** escribiendo el nombre del fichero incluido el path.
- ***RandomAccessFile(File objetoFile, String modoAcceso):*** con un objeto **File** asociado a un fichero.

El argumento *modoAcceso* puede tener dos valores:

Modo de acceso	Significado
`r`	Abre el fichero en modo de solo lectura. El fichero debe existir. Una operación de escritura en este fichero lanzará la excepción *IOException*
`rw`	Abre el fichero en modo lectura y escritura. Si el fichero no existe se crea

Una vez abierto el fichero pueden usarse los métodos *readXXX* y *writeXXX* de las clases **DataInputStream** y **DataOutputStream** (vistos anteriormente). La clase **RandomAccessFile** maneja un puntero que indica la posición actual en el fichero. Cuando el fichero se crea el puntero al fichero se coloca en 0, apuntando al principio del mismo. Las sucesivas llamadas a los métodos ***read()*** y ***write()*** ajustan el puntero según la cantidad de bytes leídos o escritos.

Los métodos más importantes son:

Método	Función
`long getFilePointer()`	Devuelve la posición actual del puntero del fichero
`void seek(long posicion)`	Coloca el puntero del fichero en una posición determinada desde el comienzo del mismo
`long length()`	Devuelve el tamaño del fichero en bytes. La posición *length()* marca el final del fichero
`int skipBytes(int desplazamiento)`	Desplaza el puntero desde la posición actual el número de bytes indicados en *desplazamiento*

El ejemplo que se muestra a continuación inserta datos de empleados en un fichero aleatorio. Los datos a insertar: apellido, departamento y salario, se obtienen de varios arrays que se llenan en el programa, los datos se van introduciendo de forma secuencial por lo que no va a ser necesario usar el método ***seek()***. Por cada empleado también se insertará un identificador (mayor que 0) que coincidirá con el índice +1 con el que se recorren los arrays. La longitud del registro de cada empleado es la misma (36 bytes) y los tipos que se insertan y su tamaño en bytes es el siguiente:

- Se inserta en primer lugar un entero, que es el identificador, ocupa 4 bytes.
- A continuación una cadena de 10 caracteres, es el apellido. Como Java utiliza caracteres UNICODE, cada carácter de una cadena de caracteres ocupa 16 bits (2 bytes), por tanto, el apellido ocupa 20 bytes.
- Un tipo entero que es el departamento, ocupa 4 bytes.
- Un tipo Double que es el salario, ocupa 8 bytes.

Tamaño de otros tipos: short (2 bytes), byte (1 byte), long (8 bytes), boolean (1bit), float (4 bytes), etc.

El fichero se abre en modo "***rw***" para lectura y escritura. El código es el siguiente:

```
import java.io.*;
public class EscribirFichAleatorio {
  public static void main(String[] args) throws IOException {
   File fichero = new File("AleatorioEmple.dat");
   //declara el fichero de acceso aleatorio
   RandomAccessFile file = new RandomAccessFile(fichero, "rw");

   //arrays con los datos
   String apellido[] = {"FERNANDEZ","GIL","LOPEZ","RAMOS",
                        "SEVILLA","CASILLA", REY"};  //apellidos
   int dep[] = {10, 20, 10, 10, 30, 30, 20};         //departamentos
   Double salario[]={1000.45, 2400.60, 3000.0, 1500.56,
                     2200.0, 1435.87, 2000.0};       //salarios

   StringBuffer buffer = null; //buffer para almacenar apellido
   int n = apellido.length;//nÚmero de elementos del array

   for (int i = 0; i<n; i++){ //recorro los arrays
      file.writeInt(i+1); //uso i+1 para identificar empleado

      buffer = new StringBuffer( apellido[i] );
      buffer.setLength(10); //10 caracteres para el apellido
      file.writeChars(buffer.toString());//insertar apellido

      file.writeInt(dep[i]);       //insertar departamento
      file.writeDouble(salario[i]);//insertar salario
   }
   file.close();  //cerrar fichero
   }
}
```

El siguiente ejemplo toma el fichero anterior y visualiza todos los registros. El posicionamiento para empezar a recorrer los registros empieza en 0, para recuperar los siguientes registros hay que sumar 36 (tamaño del registro) a la variable utilizada para el posicionamiento:

```
import java.io.*;
public class LeerFichAleatorio {
  public static void main(String[] args) throws IOException {
   File fichero = new File("AleatorioEmple.dat");
   //declara el fichero de acceso aleatorio
   RandomAccessFile file = new RandomAccessFile(fichero, "r");
   //
   int  id, dep, posicion;
   Double salario;
   char apellido[] = new char[10], aux;

   posicion = 0;  //para situarnos al principio

   for(;;){  //recorro el fichero
      file.seek(posicion);  //nos posicionamos en posicion
      id = file.readInt();  // obtengo id de empleado

      //recorro uno a uno los caracteres del apellido
      for (int i = 0; i < apellido.length; i++) {
         aux = file.readChar();
         apellido[i] = aux;     //los voy guardando en el array
      }

      //convierto a String el array
      String apellidos = new String(apellido);
      dep = file.readInt();         //obtengo dep
      salario = file.readDouble(); //obtengo salario

      if(id > 0)
        System.out.printf("ID: %s, Apellido: %s, Departamento: %d,
                          Salario: %.2f %n",
                          id,   apellidos.trim(), dep, salario);

      //me posiciono para el sig empleado, cada empleado ocupa 36 bytes
      posicion= posicion + 36;

      //Si he recorrido todos los bytes salgo del for
      if (file.getFilePointer() == file.length())break;

   }//fin bucle for
   file.close();  //cerrar fichero
   }
}
```

La ejecución muestra la siguiente salida:

```
ID: 1, Apellido: FERNANDEZ, Departamento: 10, Salario: 1000.45
ID: 2, Apellido: GIL, Departamento: 20, Salario: 2400.6
ID: 3, Apellido: LOPEZ, Departamento: 10, Salario: 3000.0
ID: 4, Apellido: RAMOS, Departamento: 10, Salario: 1500.56
ID: 5, Apellido: SEVILLA, Departamento: 30, Salario: 2200.0
ID: 6, Apellido: CASILLA, Departamento: 30, Salario: 1435.87
ID: 7, Apellido: REY, Departamento: 20, Salario: 2000.0
```

Para consultar un empleado determinado no es necesario recorrer todos los registros del fichero, conociendo su identificador podemos acceder a la posición que ocupa dentro del mismo y obtener sus datos. Por ejemplo, supongamos que se desean obtener los datos del empleado con identificador 5, para calcular la posición hemos de tener en cuenta los bytes que ocupa cada registro (en este ejemplo son 36 bytes):

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

Para añadir registros a partir del último insertado hemos de posicionar el puntero del fichero al final del mismo:

```
long posicion= file.length() ;
file.seek(posicion);
```

Para insertar un nuevo registro aplicamos la función al identificador para calcular la posición. El siguiente ejemplo inserta un empleado con identificador 20, se ha de calcular la posición donde irá el registro dentro del fichero *(identificador -1) * 36 bytes*:

```
StringBuffer buffer = null;    //buffer para almacenar apellido
String apellido = "GONZALEZ"; //apellido a insertar
Double salario = 1230.87;      //salario
int id = 20;                   //id del empleado
int dep = 10;                  //dep del empleado

long posicion = (id -1 ) * 36; //calculamos la posición

file.seek(posicion); //nos posicionamos
file.writeInt(id);   //se escribe id
buffer = new StringBuffer( apellido);
buffer.setLength(10); //10 caracteres para el apellido
file.writeChars(buffer.toString());//insertar apellido
file.writeInt(dep);                //insertar departamento
file.writeDouble(salario);         //insertar salario

file.close();  //cerrar fichero
```

ACTIVIDAD 1.3

Consulta. Crea un programa Java que consulte los datos de un empleado del fichero aleatorio. El programa se ejecutará desde la línea de comandos y debe recibir un identificador de empleado. Si el empleado existe se visualizarán sus datos, si no existe se visualizará un mensaje indicándolo.

Inserción. Crea un programa Java que inserte datos en el fichero aleatorio. El programa se ejecutará desde la línea de comandos y debe recibir 4 parámetros: identificador de empleado, apellido, departamento y salario. Antes de insertar se comprobará si el identificador existe, en ese caso se debe visualizar un mensaje indicándolo; si no existe se deberá insertar.

Para modificar un registro determinado, accedemos a su posición y efectuamos las modificaciones. El fichero debe abrirse en modo "***rw***". Por ejemplo, para cambiar el departamento y salario del empleado con identificador 4 escribo lo siguiente:

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

ACTIVIDAD 1.4

Modificación. Crea un programa Java que reciba desde la línea de comandos un identificador de empleado y un importe. Se debe realizar la modificación del salario. La modificación consistirá en sumar al salario del empleado el importe introducido. El programa debe visualizar el apellido, el salario antiguo y el nuevo. Si el identificador no existe se visualizará mensaje indicándolo.

Borrado. Crea un programa Java que al ejecutarlo desde la línea de comandos reciba un identificador de empleado y lo borre. Se hará un borrado lógico marcando el registro con la siguiente información: el identificador será igual a -1, el apellido será igual al identificador que se borra, y el departamento y salario serán 0.

A continuación haz otro programa Java (o crea un método dentro del anterior programa) que muestre los identificadores de los empleados borrados.

1.7. TRABAJO CON FICHEROS XML

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

```
<?xml version="1.0"?>
<Empleados>
    <empleado>
       <id>1</id>
       <apellido>FERNANDEZ</apellido>
```