



# Acceso a datos Ficheros

Ana Rivera Pérez  
IES Valle Inclán  
Curso 2021/2022



# Ficheros

- Cuando un programa termina, los datos que maneja se pierden pues se almacenan en estructuras de datos en memoria: Array, Listas, etc.
- Una solución para hacer que los datos **persistan** de una ejecución para otra es almacenarlos en un fichero en disco.
- Un **fichero** o **archivo** es una colección de información que almacenamos en un soporte para poder manipularla en cualquier momento.

# Registros

- Cuando se almacenan datos en los ficheros aparece el concepto de **registro**.
- Un registro es una agrupación de datos. Cada elemento de un registro se denomina **campo**.





# Operaciones sobre ficheros

- Operación de **apertura**, varios modos: Para que un programa pueda operar con un fichero, la primera operación que tiene que realizar es la apertura del mismo. Un fichero se puede abrir de **lectura, escritura o lectura/escritura**.
- Operación de **creación**: El fichero se crea en el disco con un nombre que después se debe utilizar para acceder a él.
- Operación de **lectura**: Consiste en transferir información del fichero a la memoria, a través de una variable o variables en las que se depositarán los datos extraídos del fichero.
- Operación de **escritura**: consiste en transferir información de la memoria al fichero.
- Operación de **inserción (altas)**: consiste en añadir un nuevo registro al fichero.



# Operaciones sobre ficheros

- **Operación de borrado (bajas)** : consiste en eliminar del fichero un registro ya existente. La eliminación puede ser lógica, cambiando el valor de algún campo del registro para controlar esa 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.
- Operación de **modificación**: Consiste en cambiar parte del contenido de un registro. Antes de realizar la modificación será necesario localizar el registro y una vez localizado se realizan los cambios y se reescribe el registro.
- Operación de **consultas**: consiste en buscar en el fichero un registro determinado.
- Operación de **cierre**: Una vez que se cierra el fichero se hacen efectivos todos los cambios realizados.



# Tipos de ficheros

- Según la organización de los registros puede ser:
  - **Organización secuencial:** Registros almacenados consecutivamente según el orden lógico en que se han ido insertando.
  - **Organización directa o aleatoria:** El orden físico de almacenamiento puede no coincidir con el orden en que han sido insertados.
  - **Organización indexada:** Dos ficheros:
    - Ficheros de **datos**: Información.
    - Fichero de **índice**: Contiene la posición de cada uno de los registros en el fichero de datos.



# Tipos de ficheros

Según el tipo de información almacenada:

- **Ficheros binarios (o de bytes)** : Almacenan secuencias de dígitos binarios (bytes). Los bytes que contienen representan otra información: imágenes, música, vídeo, etc. Estos ficheros solo pueden ser abiertos por aplicaciones concretas que entiendan cómo están dispuestos los bytes dentro del fichero, y así poder reconocer la información que contiene.
- **Ficheros de caracteres (o de texto)**: Almacenan caracteres alfanuméricos en un formato estándar (ASCII, UNICODE, etc). Pueden ser creados y visualizados utilizando cualquier editor de texto que ofrezca el sistema operativo (por ejemplo: Notepad, Vi, Edit, etc.).





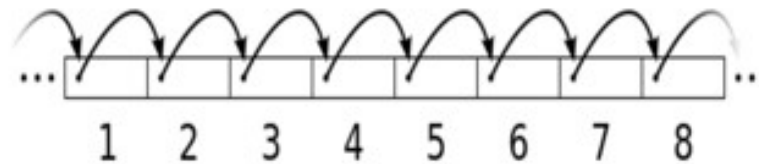
# Tipos de ficheros

- Según el acceso a la información almacenada:
  - **Ficheros de acceso secuencial:** En este tipo de ficheros la información es almacenada como una secuencia de bytes (o caracteres), de manera que para acceder al byte (o carácter)  $i$ -ésimo, es necesario pasar antes por todos los anteriores ( $i-1$ ). La escritura de datos se hará a partir del último dato escrito.
  - **Ficheros de acceso directo o aleatorio:** A diferencia de los anteriores, el acceso puede ser directamente a una posición concreta del fichero, sin necesidad de recorrer los datos anteriores. Los datos están almacenados en registros de tamaño conocido. Nos podemos mover de un registro a otro de forma aleatoria para leerlos o modificarlos.

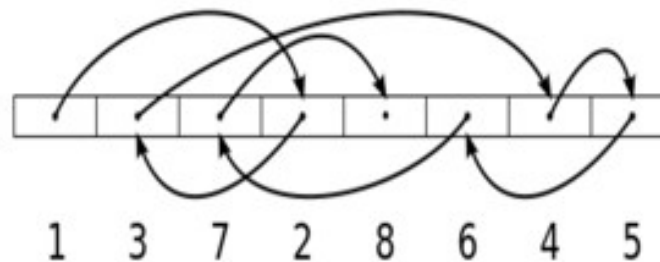


# Tipos de ficheros

Acceso secuencial



Acceso aleatorio





# Clase File

- La clase **File** **representa un archivo o un directorio** dentro de un sistema de ficheros, por lo tanto sus métodos permitirán interrogar al sistema sobre todas las características de ese fichero o directorio. La clase proporciona el constructor siguiente:

**public File(String directorioyfichero)**

- Este constructor crea un objeto **File** a partir de un nombre de fichero más su ruta de acceso (relativa o absoluta). Por ejemplo, el siguiente código crea un objeto **File** a partir de la ruta relativa *proyecto\texto.txt*. El separador de directorios viene especificado en Windows por la secuencia de escape '\\'. Este separador en Unix es '/'.

**File fichero = new File("proyecto\\texto.txt");** // En Windows

**File fichero = new File("proyecto/texto.txt");** // En Linux

- Con ruta absoluta sería:

**File fichero = new File("C:\\Ejercicios\\proyecto\\texto.txt");**

# Clase File

- El segundo constructor crea un objeto **File** a partir de una ruta (absoluta o relativa) y un nombre de fichero separado:

```
public File(String directorio, String  
            nombre_fichero)
```

- Por ejemplo, el objeto *fichero* anterior podría definirse también así:

```
File fichero = new File("proyecto", "texto.txt");
```

- Como el objeto File puede representar un fichero o directorio, en el siguiente constructor, el primer parámetro funcionará como directorio (ruta relativa) y el segundo parámetro como el nombre del fichero.

```
public File(File directorio, String nombre_fichero)
```

```
File fichero = new File(new File("dir_proyecto"), "texto.txt");
```



# Métodos de la clase File

<i>Método</i>	<i>Significado</i>
<b>getName</b>	Devuelve el nombre del fichero o directorio
<b>getParent</b>	Devuelve el nombre del directorio padre o null si no existe.
<b>getPath</b>	Devuelve la ruta relativa.
<b>getAbsolutePath</b>	Devuelve la ruta absoluta del fichero/directorio.
<b>exists</b>	Devuelve <b>true</b> si el fichero/directorio existe.
<b>canWriter</b>	Devuelve <b>true</b> si se puede escribir en el fichero o directorio especificado por el objeto <b>File</b> .
<b>canRead</b>	Devuelve <b>true</b> si se puede leer desde el fichero.
<b>isFile</b>	Devuelve <b>true</b> si se trata de un fichero válido.
<b>isDirectory</b>	Devuelve <b>true</b> si el objeto File corresponde a un directorio válido.
<b>isHidden</b>	Devuelve <b>true</b> si se trata de un fichero o directorio oculto.
<b>length</b>	Devuelve el tamaño en bytes del fichero (si es un directorio devuelve 0).
<b>createNewFile</b>	Crea un nuevo fichero vacío, asociado a File si y solo si no existe un fichero con ese nombre.



# Métodos de la clase File

Método	Significado
<b>list</b>	Devuelve una matriz de objetos <b>String</b> que almacena los nombres de los ficheros y directorios que hay en el directorio especificado por <b>File</b> .
<b>mkdir</b>	Crea un directorio con el nombre indicado en la creación del objeto <b>File</b> .
<b>makedirs</b>	Crea el directorio especificado por el objeto <b>File</b> incluyendo los directorios que no existan en la ruta especificada.
<b>delete</b>	Borra el fichero o directorio especificado asociado al objeto <b>File</b> . Cuando se trate de un directorio, éste debe estar vacío.
<b>deleteOnExit</b>	Igual que <b>delete</b> , cuando la máquina virtual termina.
<b>createTempFile</b>	Crea el fichero vacío especificado por los argumentos pasados en el directorio temporal del sistema
<b>renameTo(File nuevonombre)</b>	Renombra el fichero especificado por el objeto File que recibe este mensaje, con el nombre del objeto File pasado como argumento
<b>setReadOnly</b>	Marca el fichero o directorio especificado de sólo lectura
<b>toString</b>	Devuelve la ruta especificada cuando se creó el objeto <b>File</b> .

# Ejemplo

- 1. 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:

```
package File;
import java.io.File;
public class Directorios {

    public static void main(String[] args) {
        System.out.println("Ficheros en el directorio actual");
        File f = new File(".");
        String[] archivos = f.list();
        for (int i=0; i< archivos.length; i++) {
            System.out.println(archivos[i]);
        }
    }
}
```



# Ejemplo

- 2. El siguiente ejemplo muestra información del fichero texto.txt (debe existir en la carpeta del proyecto):

```
package File;
import java.io.File;
public class VerInf {
public static void main(String[] args) {
System.out.println("Información sobre el fichero");
File f = new File("texto.txt");
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 escribir :"+f.canWrite());
System.out.println("Se puede leer    :"+f.canRead());
System.out.println("Tamaño          :"+f.length());
System.out.println("Es un directorio :"+f.isDirectory());
System.out.println("Es un fichero    :"+f.isFile());
}
}
```



# Ejemplo

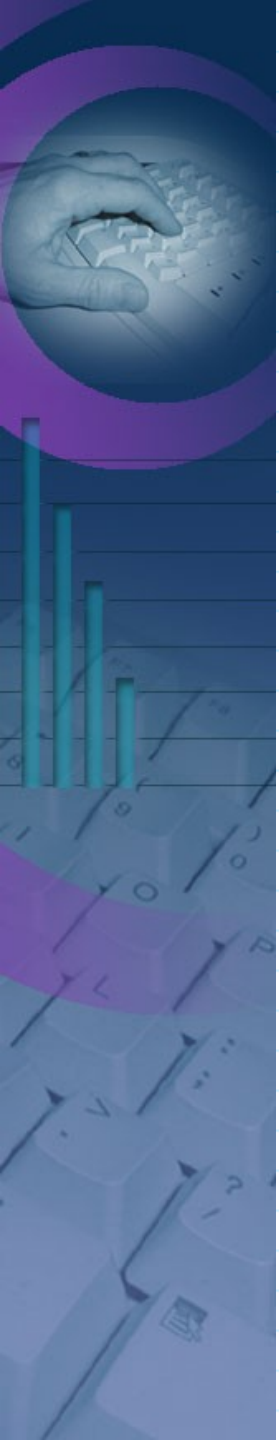
- 3. El siguiente ejemplo crea un directorio llamado NUEVODIR en el directorio actual, a continuación crea dos ficheros vacíos en dicho directorio y uno de ellos lo renombra:

```
import java.io.File;
import java.io.IOException;
public class CrearDir {
    public static void main(String[] args) {
        File d = new File ("NUEVODIR");
        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 e){
            e.printStackTrace();
        }
        f1.renameTo(new File(d,"FICHERONUEVO")); //Renombro FICHERO1
    }
}
```



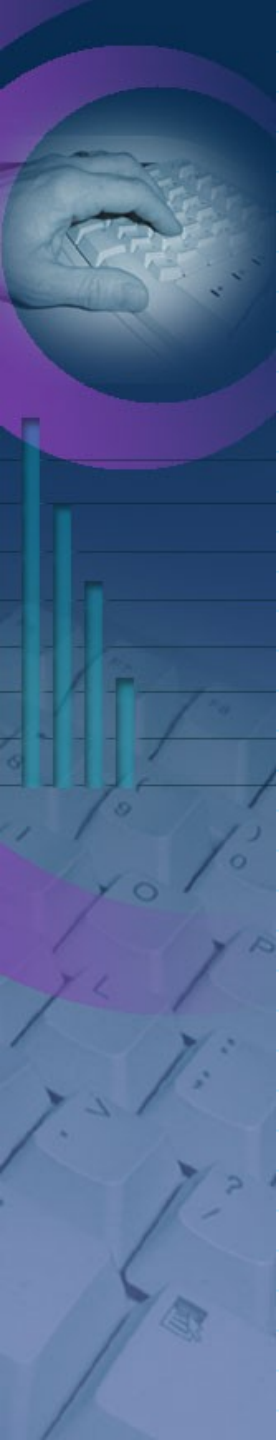
# Ficheros de acceso secuencial

- Es el tipo de acceso más simple a un fichero.
- Un fichero abierto para acceso secuencial puede almacenar registros de cualquier longitud, incluso de un sólo byte.
- Como la información se escribe registro a registro, éstos son colocados uno a continuación de otro, y cuando se lee, se empieza por el primer registro y se continúa al siguiente hasta alcanzar el final.
- Este tipo de acceso generalmente se utiliza con ficheros de texto en los que se escribe toda la información desde el principio hasta el final y se lee de la misma forma.
- No son apropiados para almacenar grandes series de números, porque cada número es almacenado como una secuencia de bytes.



# Operaciones sobre ficheros secuenciales

- **Altas:** Se realizan al final del último registro insertado, es decir, sólo se permite añadir datos al final.
- **Bajas:** Para dar de baja a un registro 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.
- **Consultas:** Para consultar un determinado registro es necesario empezar la lectura desde el primer registro y continuar leyendo secuencialmente hasta localizar el registro buscado.
- **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.



# Clases asociadas a ficheros de acceso secuencial

- Las clases **FileOutputStream** y **FileInputStream** permite escribir o leer datos de un fichero **byte a byte**.(ficheros binarios)
- Las clases **FileWriter** y **FileReader** permiten escribir o leer **caracteres** en un fichero. (ficheros de textos)
- Las clases **DataInputStream** y **DataOutputStream** permiten leer y escribir datos de cualquier **datos primitivo** (**int, long, float**, etc) y cadenas de caracteres en un fichero.(ficheros de datos primitivos)



# Ficheros de texto o caracteres

- Los ficheros de texto normalmente se generan con un editor, almacenan caracteres alfanuméricos en un formato estándar (ASCII, UNICODE, UTF8, etc). Para trabajar con ellos usaremos las clases **FileWriter** para escribir caracteres en el fichero y **FileReader** para leer caracteres de un 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 **FileReader** se puede generar la excepción **FileNotFoundException** ( el nombre del fichero no existe o no es válido) y al usar la clase **FileWriter** la excepción **IOException** (el disco está lleno o protegido contra escritura)



# FileReader

- Un flujo de la clase **FileReader** permite **leer caracteres desde un fichero de manera secuencial**. Además de los métodos que esta clase hereda de **Reader** (métodos **read**), la clase proporciona los constructores siguientes:

**FileReader(String nombre)**  
**FileReader(File fichero)**

- El primer constructor abre un flujo de entrada desde el fichero especificado por nombre y el segundo lo hace a partir de un objeto **File**.
- Al usar la clase **FileReader** se puede generar la excepción **FileNotFoundException** (porque el nombre del fichero no exista o no sea válido).
- **FileReader** no contiene métodos que permitan leer líneas completas, pero **BufferedReader** sí; dispone del método **readLine()** que lee una línea del fichero y la devuelve o null si es fin de fichero.

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



# FileReader

- El siguiente ejemplo lee cada uno de los caracteres de un fichero de texto y los muestra en pantalla:

```
package FicherosTexto;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
public class LeerFichTexto {

    public static void main(String[] args) throws
    IOException{
        File fichero = new File("LeerFichTexto.java");
        FileReader fic = new FileReader(fichero); //crea el flujo
de entrada
        int i;
        while ((i=fic.read()) != -1) // se va leyendo un caracter
hasta que sea -1 (fin fichero)
            System.out.print((char) i);
        fic.close();
    }
}
```



# BufferedReader

- El siguiente ejemplo lee el fichero FichTexto.java línea a línea y las va visualizando en pantalla:

```
package FicherosTexto;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
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) //lee una línea del
fichero
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 Entrada/Salida");}
}
}
```



# FileWriter

- Un flujo de la clase **FileWriter** permite **escribir caracteres en un fichero secuencial**. Además de los métodos que esta clase hereda de **Writer** (métodos **writer**), la clase proporciona los constructores siguientes:

**FileWriter(String nombre)**

**FileWriter(String nombre, boolean añadir)**

**FileWriter(File fichero)**

**FileWriter(File fichero, boolean añadir)**

- El primer constructor abre un flujo de salida hacia el fichero especificado por *nombre*, mientras que el segundo hace lo mismo, pero con la posibilidad de añadir datos a un fichero existente (añadir = true); el tercero lo hace a partir de un objeto File y el cuarto permite añadir datos a partir del objeto File.
- Genera la excepción **IOException** (El disco está lleno o protegido contra escritura).

# FileWriter

- El siguiente ejemplo escribe caracteres en un fichero de nombre Fichexto.txt (si no existe lo crea). Los caracteres se escriben uno a uno y se obtienen de un String que se convierte en un array de caracteres:

```
package FicherosTexto;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
public class EscribirFichTexto {

    public static void main(String[] args) throws IOException {
        File fichero = new File("FichTexto.txt");
        FileWriter fic = new FileWriter(fichero); //crear el flujo de salida
        String cadena = "Esto es una prueba con FileWriter";
        char [] cad = cadena.toCharArray(); //convierte un String en array de caracteres
        for(int i=0;i<cad.length; i++)
            fic.write(cad[i]); //se va escribiendo un caracter
            fic.append('*'); //añado al final un *
        fic.close();
    }
}
```

# BufferedWriter

- 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()**:

```
package FicherosTexto;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
public class EscribirFichTextoBuf {
public static void main(String[] args) {
try
{
BufferedWriter fichero=new BufferedWriter(new FileWriter("FichTexto.txt"));
for(int i=0; i<11;i++){
fichero.write("Fila número "+i); //escribe una línea
fichero.newLine();
}
fichero.close();
}
catch(FileNotFoundException fn){
System.out.println("No se encuentra el fichero");
}
catch(IOException io){
System.out.println("Error de Entrada/Salida");
}
}
}
```



# PrintWriter

- La clase **PrintWriter**, que también deriva de **Writer**, posee los métodos **print(String)** y **println(String)** (idénticos a los del **System.out**) para escribir en un fichero.
- Para construir un **PrintWriter** necesitamos la clase **FileWriter**:

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

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

```
PrintWriter fichero = new PrintWriter(new FileWriter("FichTexto.txt"));
```

```
for (int i=0; i<11;i++)
```

```
    fichero.println("Fila número: "+i);
```

```
fichero.close();
```



# Ficheros binarios

- Los ficheros binarios almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario.
- Tienen la ventaja de que ocupan menos espacio en disco.
- Los datos pueden ser escritos o leídos de un fichero binario byte a byte utilizando flujos de las clases **FileOutputStream** y **FileInputStream**.





# FileOutputStream

- Un flujo de la clase **FileOutputStream** permite **escribir bytes en un fichero de manera secuencial**. Además de los métodos que esta clase hereda de **OutputStream** (métodos **write**), la clase proporciona los siguientes constructores:

**FileOutputStream(String nombre)**

**FileOutputStream(String nombre, boolean añadir)**

**FileOutputStream(File fichero)**

- El primer constructor abre un flujo de salida hacia el fichero especificado por *nombre*, mientras que el segundo hace lo mismo, pero con la posibilidad de añadir datos a un fichero existente (*añadir*=true); el tercero lo hace a partir de un objeto File.





# FileInputStream

- Un flujo de la clase **FileInputStream** permite **leer bytes de un fichero de manera secuencial**. Además de los métodos que esta clase hereda de **InputStream**(métodos **read**), la clase proporciona los constructores siguientes:

**FileInputStream(String nombre)**

**FileInputStream(File fichero)**

El primer constructor abre un flujo de entrada desde el fichero especificado por *nombre*, mientras que el segundo lo hace a partir de un objeto **File**.

# Ejemplo:

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

```
package ficherosbinarios;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.file.FileSystemNotFoundException;
public class EscribirFichBytes {
public static void main(String[] args) {
try {
File fichero=new File("FichBytes.dat");
//crea flujo de salida hacia el fichero
FileOutputStream fileout = new FileOutputStream(fichero);
//crea flujo de entrada
FileInputStream filein = new FileInputStream(fichero);
for(int i=1; i< 100; i++) //escribe datos en el flujo de salida
fileout.write(i);
fileout.close();
//visualizar los datos del fichero
int i;
while ((i=filein.read()) != -1 )
System.out.println(i);
}
catch(FileSystemNotFoundException nf) {
System.out.println("No se encuentra el fichero");
}
catch(IOException io) {
System.out.println("Error de E/S"); }}}}
```



## Otras clases

- Las clases **BufferedInputStream** y **BufferedOutputStream** añaden un buffer intermedio, como lo hacían las clases **BufferedReader** y **BufferedWriter**, pero en este caso para almacenar bytes.



# Flujos de datos primitivos

- Para poder escribir en un **fichero datos de tipos primitivos** (**boolean, byte, double, float, long, int y short**) y recuperarlos como tal, el paquete **java.io** proporciona las clases **DataInputStream** y **DataOutputStream**, las cuales permiten leer y escribir, respectivamente, datos de cualquier tipo primitivo.
- Estas clases no pueden utilizarse con los dispositivos ASCII de E/S estándar.
- Un flujo **DataInputStream** sólo puede leer datos almacenados en un fichero a través de un flujo **DataOutputStream**.
- Los flujos de estas clases actúan como filtros: esto es, los datos obtenidos del origen o enviados al destino son transformados mediante alguna operación: en este caso, sufren una conversión a un formato portable (UTF-8: Unicode ligeramente modificado) cuando son almacenados y viceversa cuando son recuperados.



# DataOutputStream

- Un flujo de la clase **DataOutputStream**, derivada indirectamente de **OutputStream**, permite **escribir en un flujo de salida subordinado, datos de cualquier tipo primitivo**.
- Las siguientes líneas de código definen un filtro que permitirá escribir datos de tipos primitivos en un fichero *datos.dat*:

```
File fichero = new File("datos.dat");
```

```
FileOutputStream fos = new FileOutputStream(fichero);
```

```
DataOutputStream dos = new DataOutputStream(fos)
```

- Un programa que quiera almacenar datos en el fichero *datos.dat*, escribirá tales datos en el filtro *dos*, que a su vez está conectado al flujo *fos* abierto hacia ese fichero.



# DataOutputStream

- El siguiente fragmento de código muestra como utilizar el filtro anterior para almacenar los datos *nombre, direccion y telefono* en un fichero:

```
File fichero = new File("datos.dat");
```

```
FileOutputStream fos = new FileOutputStream(fichero);
```

```
DataOutputStream dos = new DataOutputStream(fos);
```

```
dos.writeUTF("pepe");
```

```
dos.writeUTF("Sevilla");
```

```
dos.writeLong(954675434);
```

```
dos.close();
```

```
fos.close();
```



# Métodos de la clase DataOutputStream

<i><b>Método</b></i>	<i><b>Descripción</b></i>
<b>writeBoolean</b>	Escribe un valor de tipo <b>boolean</b> .
<b>writeByte</b>	Escribe un valor de tipo <b>byte</b> .
<b>writeBytes</b>	Escribe un <b>String</b> con una secuencia de bytes.
<b>writeChar</b>	Escribe un valor de tipo <b>char</b> .
<b>writeChars</b>	Escribe un <b>String</b> con una secuencia de caracteres.
<b>writeShort</b>	Escribe un valor de tipo <b>short</b> .
<b>writeInt</b>	Escribe un valor de tipo <b>int</b> .
<b>writeLong</b>	Escribe un valor de tipo <b>long</b> .
<b>writeFloat</b>	Escribe un valor de tipo <b>float</b> .
<b>writeDouble</b>	Escribe un valor de tipo <b>double</b> .
<b>writeUTF</b>	Escribe una cadena de caracteres en formato UTF-8; los dos primeros bytes especifican el número de bytes de datos escritos a continuación.





# DataInputStream

- Un flujo de la clase **DataStream**, derivada indirectamente de **InputStream**, permite **leer de un flujo de entrada subordinado, datos de cualquier tipo primitivo** escritos por un flujo de la clase **DataOutputStream**.
- Las siguientes líneas de código definen un filtro que permitirá leer datos de tipos primitivos desde un fichero *datos.dat*.

```
File fichero = new File("datos.dat");
```

```
FileInputStream fis = new FileInputStream(fichero);
```

```
DataInputStream dis = new DataInputStream(fis);
```

- Un programa que quiera leer datos desde el fichero *datos.dat*, leerá tales datos desde el filtro *dis*, que a su vez está conectado al flujo *fis* abierto desde ese fichero.



# DataInputStream

- El siguiente fragmento de código muestra como utilizar el filtro anterior para leer los datos *nombre*, *direccion* y *telefono* en un fichero:

```
File fichero = new File("datos.dat");
```

```
FileInputStream fis = new FileInputStream(fichero);
```

```
DataInputStream dis = new DataInputStream(fis);
```

```
nombre=dis.readUTF();
```

```
direccion=dis.readUTF()
```

```
telefono=dis.readLong();
```

```
dis.close();
```

```
fis.close();
```



# Métodos de la clase DataInputStream

<i><b>Método</b></i>	<i><b>Descripción</b></i>
<b>readBoolean</b>	Devuelve un valor de tipo <b>boolean</b> .
<b>readByte</b>	Devuelve un valor de tipo <b>byte</b> .
<b>readShort</b>	Devuelve un valor de tipo <b>short</b>
<b>readChar</b>	Devuelve un valor de tipo <b>char</b> .
<b>readInt</b>	Devuelve un valor de tipo <b>int</b> ,
<b>readLong</b>	Devuelve un valor de tipo <b>long</b>
<b>readFloat</b>	Devuelve un valor de tipo <b>float</b>
<b>readDouble</b>	Devuelve un valor de tipo <b>double</b>
<b>readUTF</b>	Devuelve una cadena de caracteres en formato UTF-8; los dos primeros bytes especifican el número de bytes de datos que serán leídos a continuación

# Ejemplo: Creación y lectura de un fichero con datos primitivos

```
package ficherosbinarios;
import java.io.*;
public class Test
{
    public static void main(String[] args)
    {
        // String nombreFichero = "datos.dat";
        String nombre = null, dirección = null;
        long teléfono = 0;

        // Escribir datos
        try
        {
            File fichero = new File("datos.dat");
            FileOutputStream fos = new FileOutputStream(fichero);
            DataOutputStream dos = new DataOutputStream(fos);
            // Almacenar el nombre la dirección y el teléfono en el fichero
            dos.writeUTF("Ana");
            dos.writeUTF("Sevilla");
            dos.writeLong(942334455);
            dos.close(); fos.close();
            // Leer datos
            FileInputStream fis = new FileInputStream(fichero);
            DataInputStream dis = new DataInputStream(fis);
            // Leer el nombre la dirección y el teléfono del fichero
            nombre = dis.readUTF();
            dirección = dis.readUTF();
            teléfono = dis.readLong();
            System.out.println(nombre);
            System.out.println(dirección);
            System.out.println(teléfono);
            dis.close(); fis.close();
        }
        catch (IOException e)
        {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```



# Ejercicio 1: Crear un fichero con datos primitivos

- Solicitar por teclado el nombre de un fichero. Si el fichero existe se preguntará si se desea sobrescribirlo o añadir nuevos registros a los ya existentes.
- El fichero tiene los siguientes datos de tipo primitivo:

Nombre	String
Dirección	String
Teléfono	String
Edad	Int

- Cada vez que se grabe un registro se preguntará si se desea grabar otro registro.
- Controlar todos los errores que se puedan producir.



## Ejercicio 2: Leer de un fichero con datos primitivos

- Visualizar los registros del fichero con datos primitivos creado anteriormente.
- Controlar que el fichero exista. Mostrar un mensaje de error en caso contrario.
- Cuando se termine de visualizar los datos se mostrará el mensaje “Fin de fichero”.





# Serialización de objetos

- La operación de enviar una serie de objetos a un fichero en disco para hacerlos persistentes recibe el nombre de **serialización**, y la operación de leer o recuperar su estado del fichero para reconstruirlos en memoria recibe el nombre de **deserialización**.
- Para realizar estas operaciones, el paquete **java.io** proporciona las clases **ObjectOutputStream** y **ObjectInputStream**.
- Ambas clases dan lugar a flujos que procesan sus datos; en este caso se trata de convertir el **estado de un objeto** (los atributos excepto las variables estáticas), incluyendo la clase del objeto y el prototipo de la misma, en una **secuencia de bytes** y viceversa.
- Para poder serializar los objetos de una clase, esta debe implementar la interfaz **Serializable**.



# Escribir objetos en un fichero

- Un flujo de la clase **ObjectOutputStream** permite enviar datos de tipo primitivo y objetos hacia un flujo **OutputStream** o derivado; concretamente, cuando se trate de almacenarlos en un fichero, utilizaremos un flujo **FileOutputStream**.
- Para escribir un objeto en un flujo **ObjectOutputStream** utilizaremos el método **writeObject(objeto)**. Este método lanzará la excepción **NotSerializableException** si se intenta escribir un objeto de una clase que no implementa la interfaz **Serializable**.
- Por ejemplo, el siguiente código construye un **ObjectOutputStream** sobre un **FileOutputStream** y lo utiliza para almacenar un **String** y un objeto *Cpersona* en un fichero llamado *datos*:
  - **File fichero = new File("datos");**  
**FileOutputStream fos = new FileOutputStream(fichero);**  
**ObjectOutputStream oos = new ObjectOutputStream(fos);**  
**oos.writeUTF("fichero datos");**  
**oos.writeObject(new Cpersona(nombre, direccion, telefono));**  
**oos.close();**



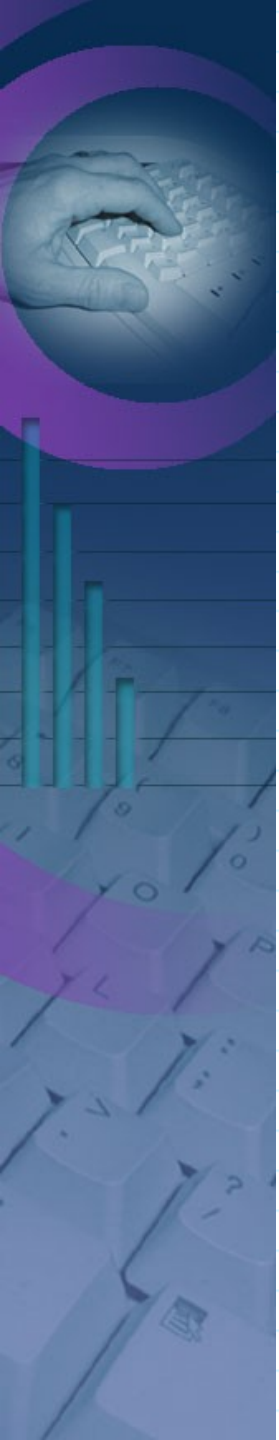
# Leer objetos desde un fichero

- Un flujo de la clase **ObjectInputStream** permite recuperar datos de tipos primitivos y objetos desde un flujo **InputStream** o derivado; cuando se trate de objetos almacenados en un fichero, utilizaremos un flujo **FileInputStream**.
- Para leer un objeto desde un flujo **ObjectInputStream** se utiliza el método **readObject()**. Puede lanzar **IOException** y **ClassNotFoundException**. Si se almacenan objetos y datos de tipos primitivos, deben ser recuperados en el mismo orden.
- El siguiente ejemplo construye un **ObjectInputStream** sobre un **FileInputStream** y lo utiliza para recuperar un **String** y un objeto *Cpersona* de un fichero denominado *datos*:

```
File fichero = new File("datos");  
FileInputStream fis = new FileInputStream(fichero);  
ObjectInputStream ois= new ObjectInputStream(fis);  
String str =(String)ois.readUTF();  
Cpersona persona =(Cpersona)ois.readObject();  
ois.close;
```

# Ejercicio 3: Fichero de personas

- Se quiere crear un fichero con información sobre personas. La primera vez que se ejecute el programa se creará el fichero sino existe y se mostrará un mensaje indicándolo. En llamadas posteriores se leerá del fichero y se cargará la información en una lista.
- Cada objeto persona será de la clase Cpersona y tendrá los siguientes atributos: Nombre, Dirección y Teléfono.
- El programa mostrará un menú con las siguientes opciones:
  - Buscar persona: Se solicita el nombre y se muestra los datos de la persona.
  - Buscar siguientes personas: Busca todas las personas con el mismo nombre que la búsqueda anterior.
  - Añadir persona: Se da de alta a una persona en la lista.
  - Eliminar persona: Dado un nombre de persona, realiza una búsqueda y si el usuario acepta lo elimina de la lista.
  - Mostrar todas las personas: Visualiza todas las personas de la lista.
  - Salir: Termina el programa y se vuelca la lista de personas al fichero.
- El mantenimiento anterior se realizará mediante un objeto de la clase ListaPersona que contendrá un ArrayList y todos los métodos anteriores.



# Serializar objetos que referencian a objetos

- Cuando en un fichero se escribe un objeto que hace referencia a otros, entonces todos los objetos accesibles desde el primero deben ser escritos en el mismo proceso para mantener así la relación existente entre todos ellos. Este proceso es llevado a cabo automáticamente por el método **writeObject**.
- Análogamente, si el objeto recuperado del flujo por el método **readObject** hace referencia a otros objetos, **readObject** recorrerá sus referencias a otros objetos recursivamente, para recuperar todos ellos manteniendo la relación que existía entre ellos cuando fueron escritos.





# Ficheros de acceso aleatorio

- Los ficheros de acceso aleatorio permiten acceder 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 y nos podemos mover de un registro a otro de forma aleatoria para leerlos o modificarlos.
- Para posicionarnos en un registro es necesario aplicar una función de conversión que normalmente tiene que ver con el tamaño del registro y con la clave del mismo.
- 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 registro, 2 para el segundo empleado y así sucesivamente;
- Para localizar el empleado X necesitamos acceder a la posición  $\text{tamaño} * (X-1)$ .





# Operaciones sobre ficheros aleatorios

- **Altas:** para insertar un registro necesitamos saber su clave, aplicar la función de conversión a la clave para obtener la dirección y escribir el registro en la posición devuelta. Si la posición está ocupada por otro registro, se insertaría en la zona de excendentes del mismo fichero.
- **Bajas:** Se realizan de forma lógica: 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.
- **Consultas:** Para consultar un registro necesitamos conocer su clave, aplicar la función de conversión a la clave para obtener su dirección y leer el registro ubicado en esa posición. Habrá que comprobar si el registro está en esa posición, si no está se buscaría en la zona de excendentes.
- **Modificaciones:** Para modificar un registro hay que localizarlo, necesitamos saber su clave para aplicar la función de conversión y así obtener su dirección, modificar los datos que nos interesen y reescribir el registro en esa posición.



# Ficheros de acceso aleatorio

- El paquete `java.io` contiene la clase **RandomAccessFile** que proporciona las capacidades que permiten un **acceso aleatorio a un fichero binario**.
- Un flujo de esta clase permite realizar tanto operaciones de lectura como de escritura sobre el fichero vinculado al mismo. Esta clase deriva directamente de **Object** e implementa **DataInput** y **DataOutput**.
- En estos ficheros, para acceder a los registros se utiliza un **puntero de lectura o escritura** el cual siempre indica la posición del siguiente registro que deberá ser leído o escrito. Dicho puntero se sitúa automáticamente al principio del fichero cuando éste se abre.



# Clase RandomAccessFile

- Un flujo de esta clase permite acceder directamente a cualquier posición dentro del fichero vinculado con él.
- Proporciona dos constructores:

```
RandomAccessFile(String nombre-fichero, String modo);  
RandomAccessFile(File fichero, String modo);
```

- El argumento **modo** puede ser:
  - **r**: read. Sólo permite operaciones de lectura. El fichero debe existir.
  - **rw**: read/write. Se puede realizar operaciones de lectura y escritura. Si el fichero no existe se crea.
- Por ejemplo, el siguiente fragmento de código construye un objeto **File** para verificar si el nombre del fichero existe. Si existe y no es un fichero se lanza una excepción; en caso contrario, se crea un flujo para leer y escribir a y desde ese fichero. Si no existe también se crea el flujo y el fichero:

```
File fichero = new File("listatfnos.dat");  
If (fichero.exists() && !fichero.isFile()) throw new  
IOException (fichero.getName() + "no es un fichero");  
RandomAccessFile listaTelefonos = new  
RandomAccessFile(fichero, "rw");
```



# Métodos de RandomAccessFile

- **getFilePointer**: Devuelve la posición actual en bytes del puntero de lectura/escritura:

**long getFilePointer() throw IOException**

- **length**: Este método devuelve el tamaño del fichero en bytes. La posición length() marca el final del fichero.

**long length() throw IOException**

- **seek**: Mueve el puntero de lectura/escritura a una nueva posición desplazada **pos** bytes desde el principio del fichero.

**long seek(long pos) throw IOException**

- **skipBytes**: desplaza el puntero desde la posición actual el número de bytes indicados en desplazamiento.

**int skipBytes(int desplazamiento)**

# Clase RandomAccessFile

- Según lo expuesto las dos siguientes líneas de código sitúan el puntero de L/E, la primera *desp* bytes antes del final del fichero y la segunda *desp* bytes después de la posición actual.

```
ListaTelefonos.seek(listaTelefonos.length() - desp);  
ListaTelefonos.seek(listaTelefonos.getFilePointer() + desp);
```

- Con esta clase **no se puede serializar objetos**. Los datos deben guardarse uno a uno.
- Pueden usarse los métodos **readXXX** y **writeXXX** de las clases **DataInputStream** y **DataOutputStream** vistos anteriormente. Por ejemplo, las siguientes líneas escriben en el fichero *datos* a partir de la posición *desp*, los atributos *nombre*, *direccion* y *telefono* relativos a un objeto *Cpersona*:

```
Cpersona objeto;  
RandomAccessFile fes new  
    RandomAccessFile("datos", "rw");  
fes.seek(desp);  
fes.writeUTF(objeto.obtenerNombre());  
fes.writeUTF(objeto.obtenerDireccion());  
fes.writeUTF(objeto.obtenerTelefono());
```

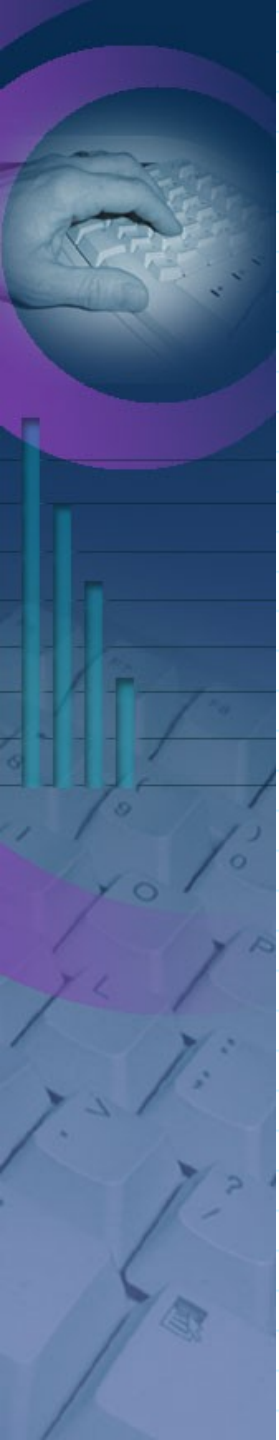




# Ejemplo: RandomAccessFile

- Insertar datos de empleados en un fichero aleatorio. Los datos a insertar: el apellido, departamento y salario se obtienen de varios arrays que se llenan en el programa. Los datos se van introduciendo de manera secuencial. Por cada empleado también se insertará un identificador 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 son:
  - Un entero, que es el identificador, ocupa 4 bytes.
  - Una cadena de 10 caracteres, el apellido, que ocupa 20 bytes. (cada carácter Unicode ocupa 2 bytes).
  - Un tipo entero, el departamento, ocupa 4 bytes.
  - Un tipo Double, el salario, ocupa 8 bytes.
- El tamaño de otros tipos: short (2 bytes), byte (1 byte), long (8 bytes), boolean (1 bit), float (4 bytes), etc.
- El fichero se abre en modo rw para lectura/escritura.



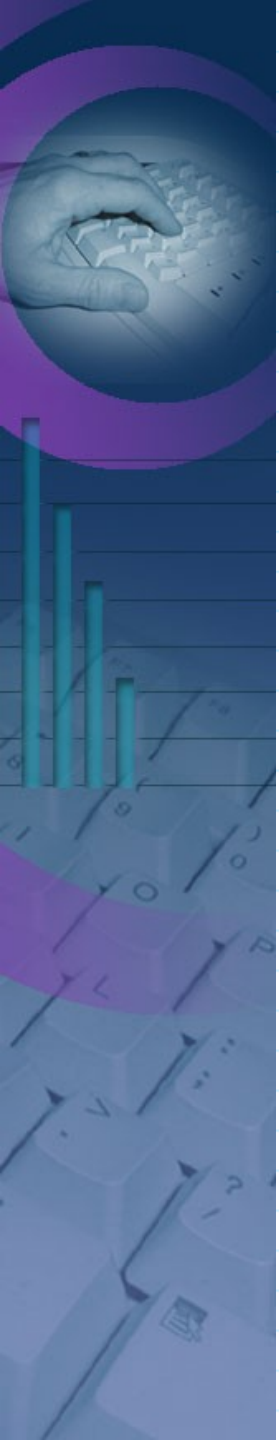
A hand is shown typing on a keyboard, with a bar chart overlaying the image. The bar chart has five bars of increasing height. The background is a dark blue gradient with a grid pattern.

```
package FicherosAleatorios;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;

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"};
        int[] dep={10,20,10,10,30,30,20};
        Double[] salario={1000.45,2400.60,3000.0, 1500.56,
        2200.0, 1435.87, 2000.0};

        StringBuffer buffer=null; //buffer para almacenar el 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 al 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 departament
            file.writeDouble(salario[i]); //insertar salario
        }
        file.close();
    }
}
```



```
package FicherosAleatorios;
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
public class LeerFichAleatorio {
    public static void main(String[] args) throws IOException {
        File fichero= new File("AleatorioEMple.dat");
        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);
            id=file.readInt(); //obtengo el id del empleado
            for(int i=0; i<apellido.length; i++){
                aux=file.readChar(); //recorro uno a uno los caracteres del array
                apellido[i]=aux; //los voy guardando en el array
            }
            String apellidoS=new String(apellido); //convierto a String el array
            dep=file.readInt(); //obtengo dep
            salario=file.readDouble(); //obtengo salario
            System.out.println("ID: "+id+ ", Apellido: "+ apellidoS +
                "Departamento: "+dep+ ", Salario: "+salario);
            posicion=posicion+36; //me posiciono para el siguiente empleado
            //Cada empleado ocupa 36 bytes
            //Si he recorrido todos los bytes salgo del for
            if(file.getFilePointer()==file.length()) break;
        } //fin bucle for
        file.close();
    }
}
```

- 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, para obtener los datos del empleado con identificador 5, para calcular la posición hemos de tener en cuenta los bytes que ocupa cada registro (36):

```
package FicherosAleatorios;
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;

public class ConsultarEmpleado {

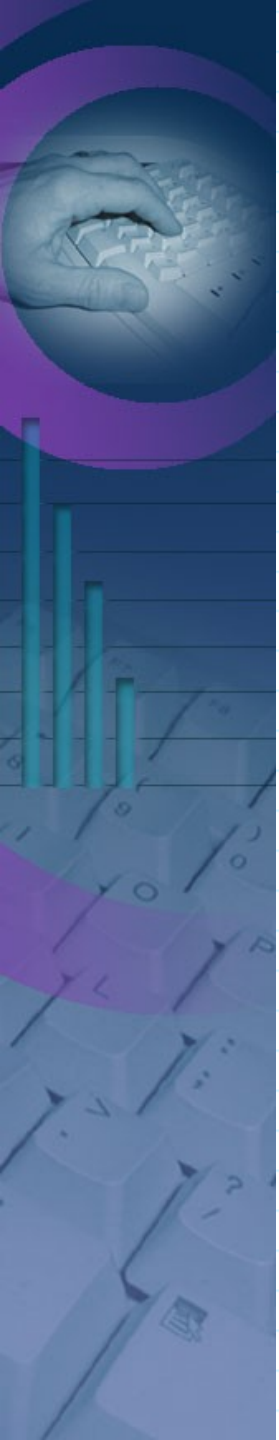
    public static void main(String[] args) throws IOException {
        File fichero= new File("AleatorioEMple.dat");
        RandomAccessFile file = new RandomAccessFile(fichero, "r");
        int id, dep, posicion;
        Double salario;
        char apellido[] = new char[10], aux;
        int identificador=5;
        posicion=(identificador-1)*36; //calculo donde empieza el registro
        if (posicion >= file.length())
            System.out.println("ID: "+ identificador+ ", No existe el empleado");
        else {
            file.seek(posicion); //nos posicionamos
            id=file.readInt(); //obtengo el id del empleado
            for(int i=0; i<apellido.length; i++){
                aux=file.readChar(); //recorro uno a uno los caracteres del array
                apellido[i]=aux; //los voy guardando en el array
            }
            String apellidoS=new String(apellido); //convierto a String el array
            dep=file.readInt(); //obtengo dep
            salario=file.readDouble(); //obtengo salario
            System.out.println("ID: "+id+ ", Apellido: "+ apellidoS +
                "Departamento: "+dep+ ", Salario: "+salario);
        }
        file.close();
    }
}
```

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

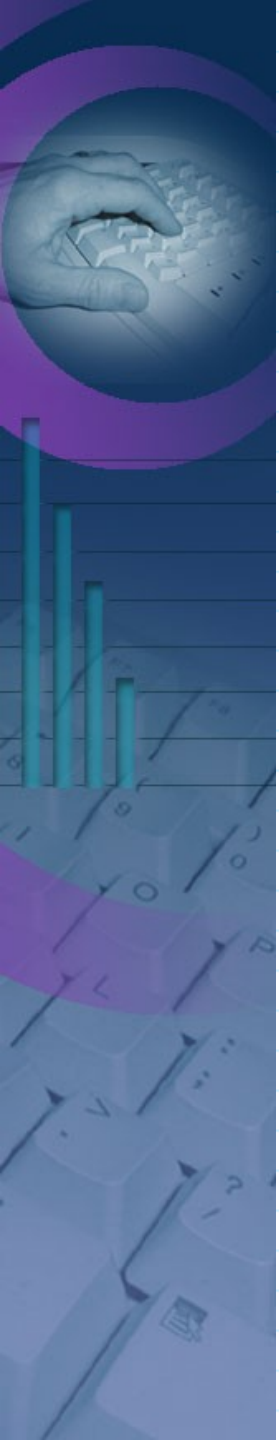
**long posicion=file.length();**  
**File.seek(posicion);**

- 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.

```
public class InsertarFichAleatorio {
    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");
        StringBuffer buffer=null; //buffer para almacenar el apellido
        String apellido = "GONZALEZ";
        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);
        buffer= new StringBuffer(apellido);
        buffer.setLength(10); //10 caracteres para el apellido
        file.writeChars(buffer.toString()); //inserta apellido
        file.writeInt(dep); //inserta departamento
        file.writeDouble(salario); //inserta salario
        file.close();
    }
}
```

- 
- Para modificar un registro determinado, accederemos a su posición y efectuaremos las modificaciones. El fichero debe abrirse en modo “rw”. Por ejemplo, para cambiar el departamento y salario del empleado con identificador 4:

```
int registro=4;  
long posicion = (registro-1) * 36; //4+20+4+8  
posicion=posicion+4+20; //sumo el tamaño del id+apellido  
file.seek(posicion); // nos posicionamos  
file.writeInt(40); //modifico departamento  
file.writeDouble(4000.87); // modifico salario
```



# Ejercicio 4: Mantenimiento de un fichero aleatorio

- Crea un programa que realice un mantenimiento (altas, bajas, consultas y modificaciones) sobre un fichero de empleados. Los campos de cada registro son:
  - Un entero, que es el identificador, ocupa 4 bytes.
  - Una cadena de 10 caracteres, el apellido que ocupa 20 bytes. (cada carácter Unicode ocupa 2 bytes).
  - Un tipo entero, el departamento, ocupa 4 bytes.
  - Un tipo Double, el salario, ocupa 8 bytes.
- La modificación se realizará sobre el campo salario.