

```

        (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 <i>b.length</i> bytes de datos de una matriz de bytes
int read(byte[] b, int desplazamiento, int n)	Lee hasta <i>n</i> bytes de la matriz <i>b</i> comenzando por <i>b[desplazamiento]</i> 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 guardarlos en un fichero, tendríamos que guardar cada atributo que forma parte del objeto por separado, esto se vuelve engoroso 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 *redefinir 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 <i>length()</i> marca el final del fichero
int skipBytes(int desplazamiento)	Desplaza el puntero desde la posición actual el número de bytes indicados en <i>desplazamiento</i>

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
            }

            //convierbo 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;
//calcula 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>
```