

Unidad didáctica 1: Manejo de Ficheros

Contenidos:

1. Clases asociadas a las operaciones de gestión de ficheros. Flujos o stream. Tipos.
2. Formas de acceso a un fichero.
3. Clases para gestión de flujos de datos desde/hacia ficheros.
4. Operaciones básicas sobre ficheros de acceso secuencial.
5. Operaciones básicas sobre ficheros de acceso aleatorio.
6. Ficheros XML. Librerías para conversión de documentos XML a otros formatos.
7. Excepciones: detección y tratamiento.
8. Introducción a JAXB

1. Introducción

Un **fichero o archivo** es un conjunto de bits almacenados en un dispositivo (un disco duro, USB..). La ventaja de utilizar ficheros es que los datos guardados 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. 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.

Vamos a ver los ficheros en lenguaje Java.

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/unil/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 <i>true</i> si el fichero/directorio existe
boolean canWrite()	Devuelve <i>true</i> si el fichero se puede escribir
boolean canRead()	Devuelve <i>true</i> si el fichero se puede leer
boolean isFile()	Devuelve <i>true</i> si el objeto File corresponde a un fichero normal
boolean isDirectory()	Devuelve <i>true</i> 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 Fíle 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 <i>null</i> si no existe

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

```

1 import java.io.*;
2
3 public class VerDirActual {
4     public static void main (String[] args) {
5         String dir = "."; //directorio actual
6         File f = new File (dir);
7         String[] archivos = f.list();
8         System.out.printf("Ficheros en el directorio actual: %d %n", archivos.length);
9         for (int i = 0; i < archivos.length; i++) {
10             File f2 = new File(f, archivos[i]);
11             System.out.printf("Nombre: %s, es fichero?: %b, es directorio?: %b %n", archivos[i], f2.isFile(), f2.isDirectory());
12         }
13     }
14 }
15 }

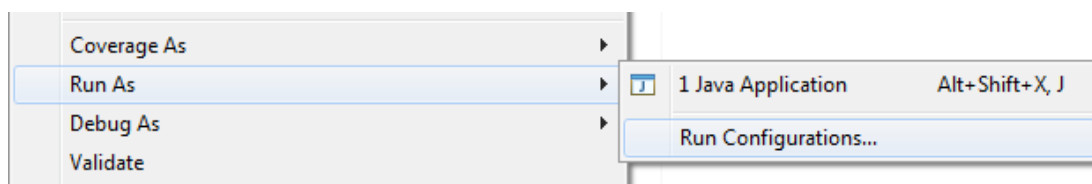
```

Propuesta Ejercicio:

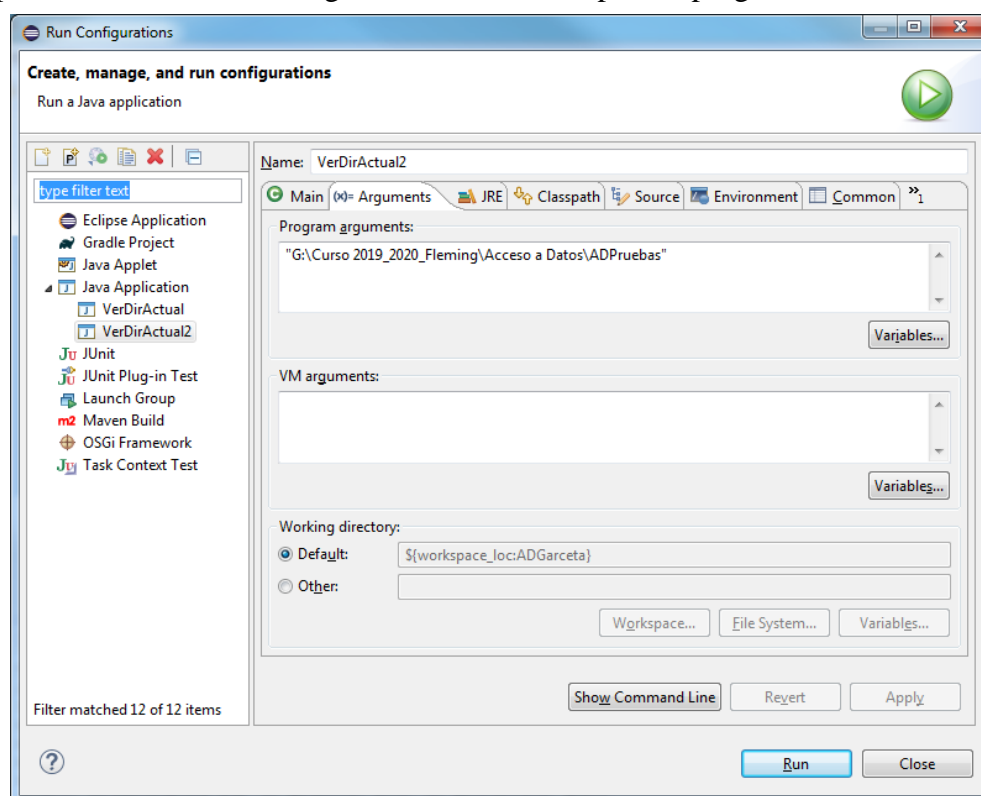
Cambia la declaración para que muestre la lista de ficheros de una carpeta de la unidad C: de tu ordenador

Cambia la declaración para que muestre la lista de ficheros del directorio introducido al ejecutar el programa. (VerDirActual2)

Nota: Para pasar parámetros a un programa en eclipse desde el menú ejecutar como "**Run As**" abrimos el menú de establecer configuraciones de ejecución "**Run configurations**".



En la ventana que nos aparece seleccionamos la pestaña "(x) = **Arguments**" y en la sección "**Program arguments**" podemos establecer los argumentos de entrada para el programa:



Realizar la Actividad 1.1

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

```
VerInf.java
1 package UD1;
2
3 import java.io.File;
4 public class VerInf {
5     public static void main( String args[] ) {
6         System.out.println( "INFORMACION SOBRE EL FICHERO: " );
7         //File f = new File( "C:\\Users\\Isabel\\Documents\\AD_2019-20\\ADGarceta\\src\\UD1\\VerInf.java" );
8         File f = new File ( "I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\VerInf.java");
9         if (f.exists()) {
10             System.out.println( "Nombre: "+f.getName() );
11             System.out.println( "Ruta: "+f.getPath() );
12             System.out.println( "Ruta: "+f.getAbsolutePath() );
13             System.out.println( "Se puede Leer: " + f.canRead());
14             System.out.println( "Se puede Escribir: " + f.canWrite());
15             System.out.println( "Tamaño: " + f.length());
16             System.out.println( "Es un directorio: " + f.isDirectory());
17             System.out.println( "Es un fichero: " + f.isFile());
18             System.out.println( "Nombre del directorio padre: " + f.getParent());
19         }
20     }
21 }
22 }
```

Propuesta Ejercicio (CrearDir)

Ha de crear un directorio NUEVODIR, (mkdir) y a continuación dos ficheros vacíos en dicho directorio y renombrar uno de ellos (Utilizar el tercer constructor File(File directorio, String nombrefichero)). En el primero indicamos directorio donde se creará el fichero y en el segundo el nombre del fichero.

```
CrearDir.java
1 package UD1;
2 import java.io.*;
3
4 public class CrearDir {
5     public static void main(String[] args) {
6         //File d = new File("NUEVODIR"); //directorio que se crea en el directorio actual
7         File d = new File ( "I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\NUEVODIR");
8         File f1 = new File(d,"FICHERO1.TXT");
9         File f2 = new File(d,"FICHERO2.TXT");
10        d.mkdir();//CREAR DIRECTORIO
11        try {
12            if (f1.createNewFile())
13                System.out.println("FICHERO1 creado correctamente...");
14            else
15                System.out.println("No se ha podido crear FICHERO1...");
16
17            if (f2.createNewFile())
18                System.out.println("FICHERO2 creado correctamente...");
19            else
20                System.out.println("No se ha podido crear FICHERO2...");
21        } catch (IOException ioe) {ioe.printStackTrace();}
22
23        f1.renameTo(new File(d,"FICHERO1NUEVO.ppp")); //renombrar FICHERO1
24        try {
25            File f3 = new File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\NUEVODIR\\FICHERO3.TXT");
26            f3.createNewFile();//crea FICHERO3 en NUEVODIR
27        } catch (IOException ioe) {ioe.printStackTrace();}
28    }
29 }
30 }
```

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**.

3. Flujos o stream. 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 descenden de las clases **InputStream** y **OutputStream**, cada una de estas clases tienen varias subclases que controlan las diferencias entre los distintos dispositivos de entrada/salida que se pueden utilizar.
- **Flujos de caracteres (16 bits):** realizan operaciones de entradas y salidas de caracteres. El flujo de caracteres viene gobernado por las clases **Reader** y **Writer**

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

3.2 Flujos de caracteres (Character streams)

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

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

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

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.

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**.

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

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.

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.

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 $\text{tamaño} \cdot (X-1)$ 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, 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.

6. Clases para gestionar el flujo de datos desde y 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.)

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 leernos 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
<code>int read()</code>	Lee un carácter y lo devuelve
<code>int read(char[] buf)</code>	Lee hasta <i>buf.length</i> caracteres de datos de una matriz de caracteres (<i>buf</i>). Los caracteres leídos del fichero se van almacenando en <i>buf</i>
<code>int read(char[] buf, int desplazamiento, int n)</code>	Lee hasta <i>n</i> caracteres de datos de la matriz <i>buf</i> comenzando por <i>buf[desplazamiento]</i> 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()**.

En el **ejemplo** siguiente, se leen cada uno de los caracteres del fichero de texto de nombre LeerFichTexto.java (localizado en una ruta concreta) 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`:

```
LeerFichTexto.java
2
3 import java.io.*;
4
5 public class LeerFichTexto {
6     public static void main(String[] args) throws IOException {
7         //declarar fichero
8         File fichero = new
9             File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\LeerFicheroTexto.txt");
10        //Crear el flujo de entrada hacia el fichero
11        FileReader fic = new FileReader(fichero);
12        int i;
13        while ((i = fic.read()) != -1) //se va leyendo un carácter
14            System.out.print( (char) i);
15        //cerrar fichero
16        fic.close();
17    }
18 }
```

La expresión ((char) i) convierte el valor entero recuperado por el método read() a carácter. 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( "rutanombrefichero");
```

Propuesta Ejercicio

Modificar para ir leyendo de 30 en 30 caracteres

```
15 // Para ir leyendo de 30 en 30 caracteres
16 char b[] = new char[30];
17 while ((i = fic.read(b)) != -1)
18     System.out.println(b);
19
20 //cerrar fichero
21 fic.close();
22 }
23 }
```

Realizar la Actividad 1.2

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 <i>buf</i> comenzando por <i>buf[desplazamiento]</i> .
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:

```
EscribirFichTexto.java
2
3 import java.io.*;
4 public class EscribirFichTexto {
5
6     public static void main(String[] args) throws IOException {
7         File fichero = new File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\FichTextoEscrito.txt"); //declara fichero
8         FileWriter fic = new FileWriter(fichero); //crear el flujo de salida
9         String cadena = "Esto es una prueba con FileWriter";
10        char[] cad = cadena.toCharArray(); //convierte un String en array de caracteres
11
12        for(int i=0; i<cad.length; i++)
13            fic.write(cad[i]); //se va escribiendo un carácter
14
15        fic.append('*'); //añado al final un *
16
17        fic.close(); //cerrar fichero
18    }
19 }
```

Propuesta Ejercicio

- Añade otra línea al fichero de modo que se escriba todo el array (utiliza fic.write(cad))
- Añade otra línea para que escriba un String. (igual que para el array
- Añade el contenido de un array de String (utiliza fic.write (nombarray[i]))

```

17 // Propuesta Ejercicio. En lugar de escribir los caracteres uno a uno, escribimos todo el array
18 fic.write("\n");
19 fic.write(cad);
20 String c="\n*esto es lo ultimo*";
21 fic.write(c);//escribir un String
22
23 //Escribe cadenas de caracteres que se extraen de un array de String.
24 String prov[] = {"Albacete","Avila","Badajoz",
25                 "Cáceres","Huelva","Jaén",
26                 "Madrid","Segovia","Soria","Toledo",
27                 "Valladolid","Zamora"};
28
29 fic.write("\n");
30 for(int i=0; i<prov.length; i++) {
31     fic.write(prov[i]);
32     fic.write("\n");
33 }
34 fic.close(); //cerrar fichero
35 }
36 }

```

Tener en cuenta que si el fichero existe cuando vayamos a escribir caracteres sobre él, todo lo que tenía almacenado anteriormente se borrará. Si queremos añadir caracteres al final, usaremos la clase `FileWriter` 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 se utiliza un bloque try-catch:

```

LeerFichTextoBuf.java
1 package UD1;
2
3 import java.io.*;
4
5 public class LeerFichTextoBuf {
6     public static void main(String[] args) {
7         try{
8             File fic = new File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\FichTextoEscrito.txt");//declara fichero
9             BufferedReader fichero = new BufferedReader(new FileReader(fic));
10
11             String linea;
12             while((linea = fichero.readLine())!=null)
13                 System.out.println(linea);
14             fichero.close();
15         }
16         catch (FileNotFoundException fn){
17             System.out.println("No se encuentra el fichero");}
18         catch (IOException io) {
19             System.out.println("Error de E/S ");}
20     }
21 }
22 }
23 }

```

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()`

```
EscibirFichTextoBuf.java
2
3 import java.io.*;
4
5 public class EscibirFichTextoBuf {
6     public static void main(String[] args) {
7         try{
8             File fic = new File ("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\FichTextoBuff.txt");
9             BufferedWriter fichero = new BufferedWriter (new FileWriter(fic));
10            for (int i=1; i<11; i++){
11                fichero.write("Fila numero: "+i); //escribe una línea
12                fichero.newLine(); //escribe un salto de línea
13            }
14            fichero.close();
15        }
16        catch (FileNotFoundException fn ){
17            System.out.println("No se encuentra el fichero");}
18        catch (IOException io) {
19            System.out.println("Error de E/S ");}
20    }
21 }
```

La clase **PrintWriter**, que también deriva de **Writer**, posee los métodos **print(String)** y **println(String)** 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) );
```

Propuesta Ejercicio

Modifica el ejemplo anterior para utilizar la clase `PrintWriter` y el método `println()`

```
EscibirFichTextoPrint.java
2
3 import java.io.*;
4
5 public class EscibirFichTextoPrint {
6     public static void main(String[] args) {
7         try{
8             File fic = new File ("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\FichTextoPrint.txt");
9             PrintWriter fichero = new PrintWriter (new FileWriter(fic));
10            /*PrintWriter fichero = new PrintWriter
11                (new FileWriter("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\FichTextoPrint.txt"));
12            */
13            for (int i=1; i<11; i++){
14                fichero.println("Fila numero: "+i); //escribe una línea
15            }
16            fichero.append("FIN");
17            fichero.close();
18        }
19        catch (FileNotFoundException fn ){
20            System.out.println("No se encuentra el fichero");}
21        catch (IOException io) {
22            System.out.println("Error de E/S ");}
23    }
24 }
```

6.2 Ficheros binarios

Los ficheros binarios almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario como ocurriría 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
<code>int read()</code>	Lee un byte y lo devuelve
<code>int read(byte[] b)</code>	Lee hasta <i>b.length</i> bytes de datos de una matriz de bytes
<code>int read(byte[] b, int desplazamiento, int n)</code>	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

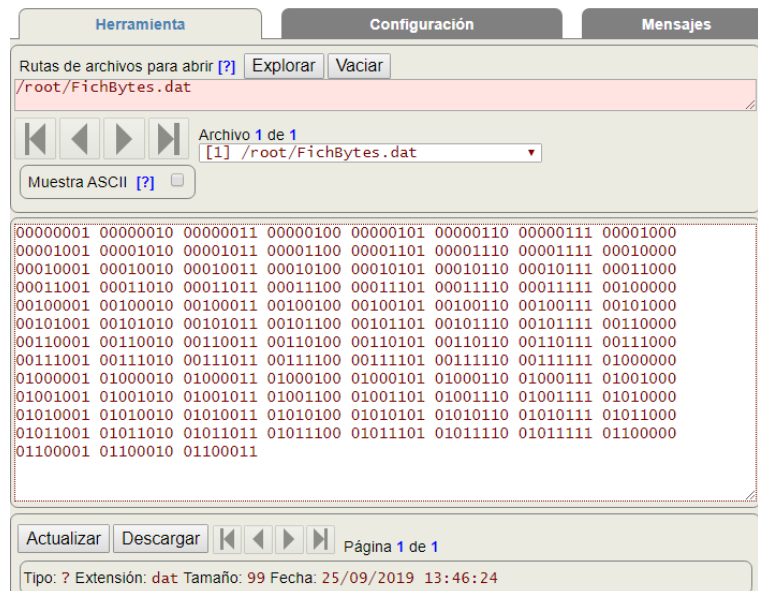
Los métodos que proporciona la clase **FileOutputStream** para escritura son:

Método	Función
<code>void write(int b)</code>	Escribe un byte.
<code>void write(byte[] b)</code>	Escribe <i>b.length</i> bytes.
<code>void write(byte[] b, int, desplazamiento, int n)</code>	Escribe <i>n</i> bytes a partir de la matriz de bytes de entrada comenzando por <i>b[desplazamiento]</i> .

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

```
EscibirFichBytes.java
3 import java.io.*;
4
5 public class EscibirFichBytes {
6     public static void main(String[] args) throws IOException {
7         File fic = new File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\FichBytes.dat");//declara fichero
8         //crea flujo de salida hacia el fichero
9         FileOutputStream fileout = new FileOutputStream(fic,true);
10        //crea flujo de entrada
11        FileInputStream filein = new FileInputStream(fic);
12        int i;
13
14        for (i=1; i<100; i++)
15            fileout.write(i); //escribe datos en el flujo de salida
16        fileout.close(); //cerrar stream de salida
17
18        //visualizar los datos del fichero
19        while ((i = filein.read()) != -1) //lee datos del flujo de entrada
20            System.out.println(i);
21        filein.close(); //cerrar stream de entrada
22    }
23 }
24
```

Si buscas una herramienta en internet que te permita ver el contenido de `FichBytes.dat`, esto es lo que veríamos



Para **añadir** bytes al final del fichero usaremos **FileOutputStream** 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 son:

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));
```

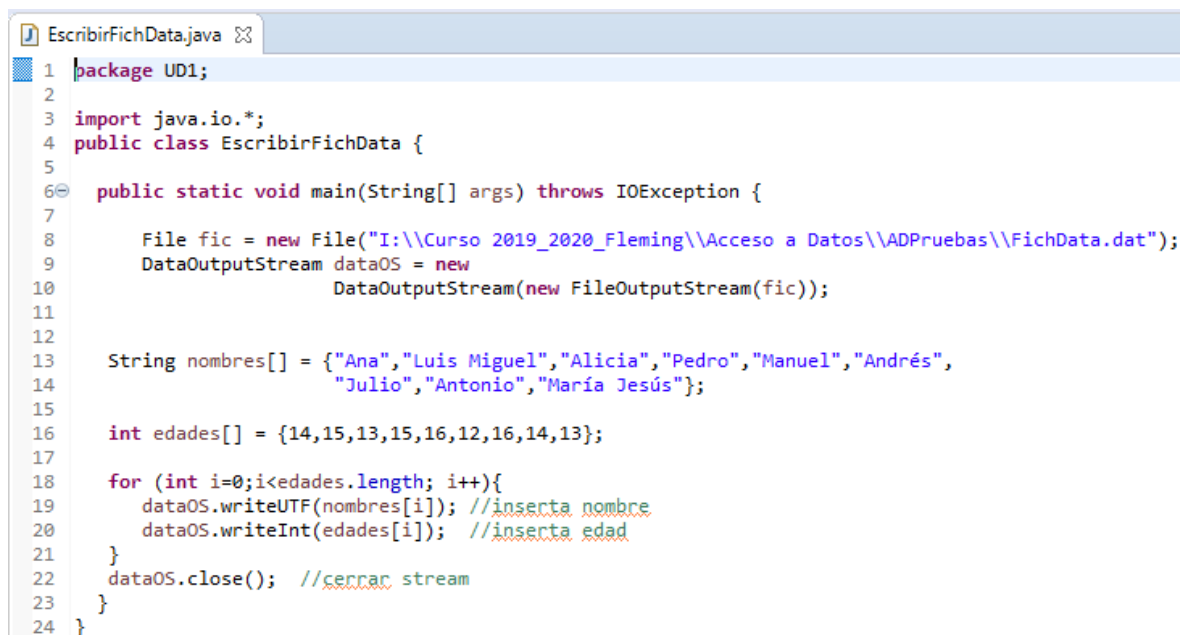
Para abrir un objeto **DataOutputStream**, se utilizan los mismos métodos que para **FileOutputStream**, ejemplo:

```
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, recorreremos los arrays y vamos escribiendo en el fichero el nombre (mediante el método **writeUTF(String)**) y la edad (mediante el método **writeInt(int)**):



```
1 package UD1;  
2  
3 import java.io.*;  
4 public class EscribirFichData {  
5  
6     public static void main(String[] args) throws IOException {  
7  
8         File fic = new File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\FichData.dat");  
9         DataOutputStream dataOS = new  
10             DataOutputStream(new FileOutputStream(fic));  
11  
12  
13         String nombres[] = {"Ana","Luis Miguel","Alicia","Pedro","Manuel","Andrés",  
14             "Julio","Antonio","María Jesús"};  
15  
16         int edades[] = {14,15,13,15,16,12,16,14,13};  
17  
18         for (int i=0;i<edades.length; i++){  
19             dataOS.writeUTF(nombres[i]); //inserta nombre  
20             dataOS.writeInt(edades[i]); //inserta edad  
21         }  
22         dataOS.close(); //cerrar stream  
23     }  
24 }
```

Propuesta Ejercicio

Visualiza los datos grabados anteriormente en el fichero (FichData.dat), se deben recuperar en el mismo orden en el que se escribieron, es decir, primero obtenemos el nombre y luego la edad, utiliza para ello *DataInputStream*. Opten una salida por consola como la siguiente:

```
<terminated> LeerFichData [Java Application] C:\Pro:  
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
```



```

LeerFichData.java
3 import java.io.*;
4
5 public class LeerFichData {
6
7     public static void main(String[] args) throws IOException {
8
9         File fic = new File("G:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\FichData.dat");
10        DataInputStream dataIS = new
11            DataInputStream(new FileInputStream(fic));
12
13        String nom;
14        int ed;
15        try {
16            while (true) {
17                nom = dataIS.readUTF(); //recupera el nombre
18                ed = dataIS.readInt(); //recupera la edad
19                System.out.println("Nombre: " + nom + ", edad: " + ed);
20            }
21        } catch (EOFException eo) {}
22
23        dataIS.close(); //cerrar stream
24    }
25 }

```

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.

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.

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:

```

Persona.java
4
5 import java.io.*;
6
7 public class Persona implements Serializable{
8
9     private String nombre;
10    private int edad;
11
12    public Persona(String nombre,int edad) {
13        this.nombre=nombre;
14        this.edad=edad;
15    }
16
17    public Persona() {
18        this.nombre=null;
19    }
20
21    public void setNombre(String nom){nombre=nom;}
22    public void setEdad(int ed){edad=ed;}
23
24    public String getNombre(){return nombre;}
25    public int getEdad(){return edad;}
26 }//fin Persona

```

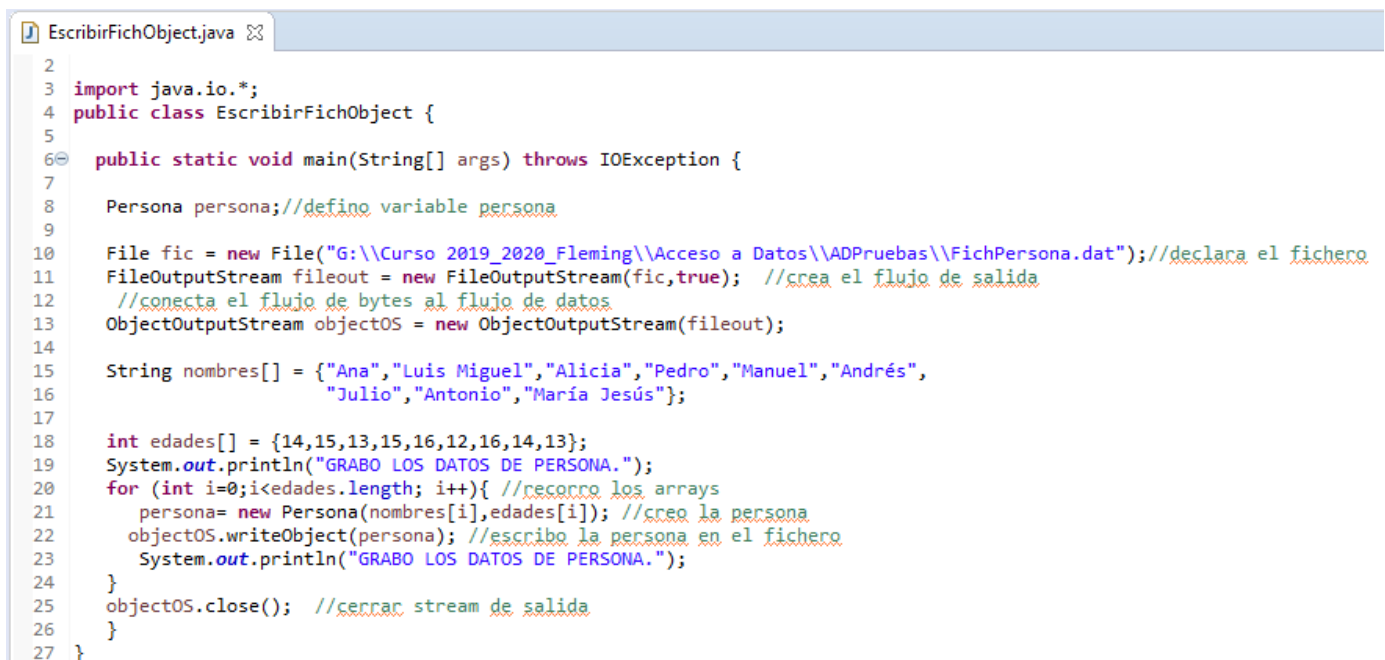
El siguiente **ejemplo** escribe objetos **Persona** en un fichero. Necesitamos crear un flujo de salida a disco con **FileOutputStream** y a continuación se crea el flujo de salida **ObjectOutputStream** que es el que procesa los datos y se ha de vincular al fichero de **FileOutputStream**:

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

O bien:

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

El método **writeObject()** escribe los objetos al flujo de salida y los guarda en un fichero en disco:



```
1  EscribirFichObject.java  
2  
3  import java.io.*;  
4  public class EscribirFichObject {  
5  
6  public static void main(String[] args) throws IOException {  
7  
8      Persona persona; //defino variable persona  
9  
10     File fic = new File("G:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\FichPersona.dat"); //declara el fichero  
11     FileOutputStream fileout = new FileOutputStream(fic,true); //crea el flujo de salida  
12     //conecta el flujo de bytes al flujo de datos  
13     ObjectOutputStream objectOS = new ObjectOutputStream(fileout);  
14  
15     String nombres[] = {"Ana", "Luis Miguel", "Alicia", "Pedro", "Manuel", "Andrés",  
16                          "Julio", "Antonio", "María Jesús"};  
17  
18     int edades[] = {14,15,13,15,16,12,16,14,13};  
19     System.out.println("GRABO LOS DATOS DE PERSONA.");  
20     for (int i=0; i<edades.length; i++){ //recorro los arrays  
21         persona= new Persona(nombres[i],edades[i]); //creo la persona  
22         objectOS.writeObject(persona); //escribo la persona en el fichero  
23         System.out.println("GRABO LOS DATOS DE PERSONA.");  
24     }  
25     objectOS.close(); //cerrar stream de salida  
26 }  
27 }
```

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 objctIS = new ObjectInputStream(filein);
```

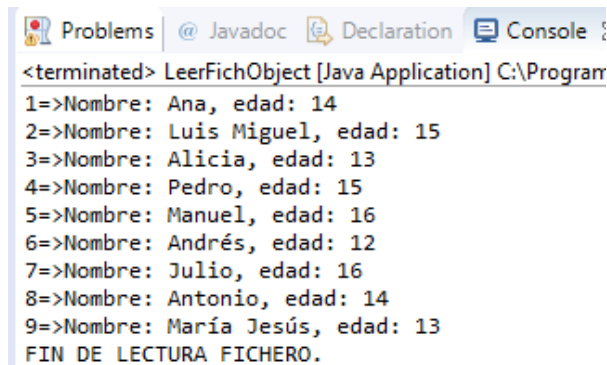
O bien:

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

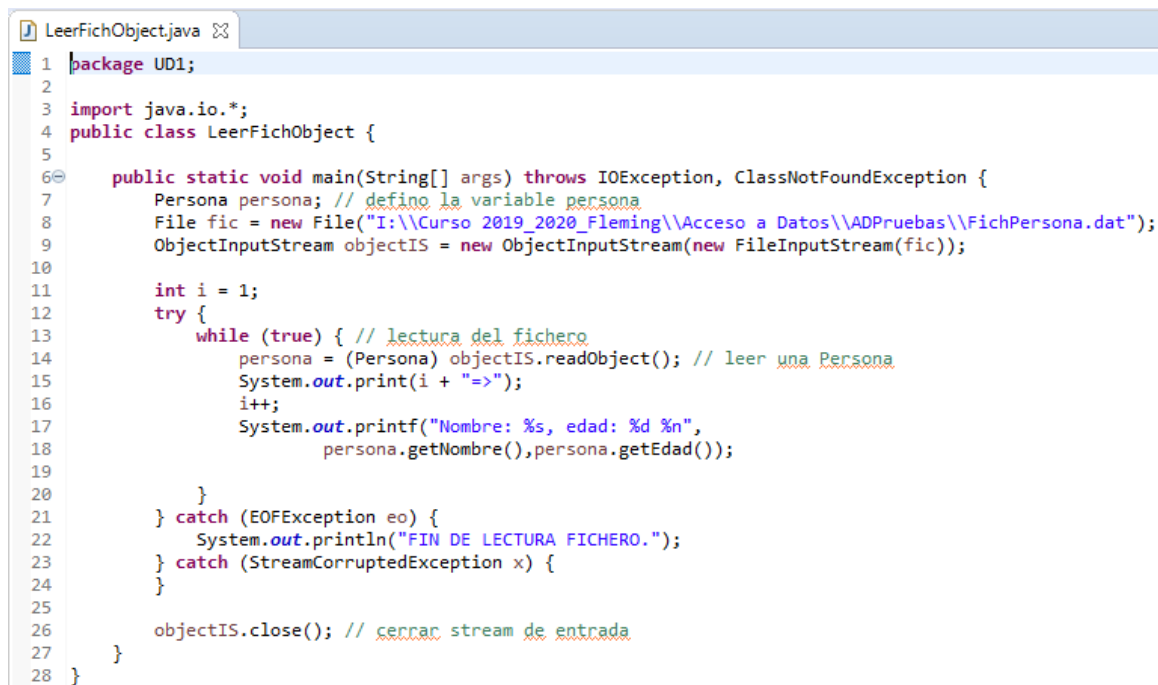
El método **readObject()** lee los objetos del flujo de entrada, puede lanzar la excepción **ClassNotFoundException** e **IOException**, por lo que será necesario controlarlas.

Propuesta Ejercicio

Visualiza los datos grabados anteriormente en el fichero (FichPersona.dat). El proceso de lectura se puede hacer en un bucle while(true), que 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. Opten una salida por consola como la siguiente:



```
<terminated> LeerFichObject [Java Application] C:\Program
1=>Nombre: Ana, edad: 14
2=>Nombre: Luis Miguel, edad: 15
3=>Nombre: Alicia, edad: 13
4=>Nombre: Pedro, edad: 15
5=>Nombre: Manuel, edad: 16
6=>Nombre: Andrés, edad: 12
7=>Nombre: Julio, edad: 16
8=>Nombre: Antonio, edad: 14
9=>Nombre: María Jesús, edad: 13
FIN DE LECTURA FICHERO.
```



```
LeerFichObject.java
1 package UD1;
2
3 import java.io.*;
4 public class LeerFichObject {
5
6     public static void main(String[] args) throws IOException, ClassNotFoundException {
7         Persona persona; // defino la variable persona
8         File fic = new File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\FichPersona.dat");
9         ObjectInputStream objectIS = new ObjectInputStream(new FileInputStream(fic));
10
11         int i = 1;
12         try {
13             while (true) { // lectura del fichero
14                 persona = (Persona) objectIS.readObject(); // leer una Persona
15                 System.out.print(i + "=>");
16                 i++;
17                 System.out.printf("Nombre: %s, edad: %d %n",
18                                 persona.getNombre(), persona.getEdad());
19             }
20         } catch (EOFException eo) {
21             System.out.println("FIN DE LECTURA FICHERO.");
22         } catch (StreamCorruptedException x) {
23         }
24
25         objectIS.close(); // cerrar stream de entrada
26     }
27 }
28 }
```

Problema con los ficheros de objetos:

Existe un problema con los ficheros de objetos y es que 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, salta 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{
    }
}

```

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 objectOS;

if (!fichero.exists())
{
    //Si el fichero no existe crea un ObjectOutputStream, la primera vez
    FileOutputStream fileout;
    fileout = new FileOutputStream(fichero);
    objectOS = new      ObjectOutputStream(fileout);
}
else
{
    // Si ya existe el fichero creará un ObjectOutputStream con el método
    //writeStreamHeader redefinido (sin hacer nada)
    objectOS = new MiObjectOutputStream(new
    FileOutputStream(fichero,true));
}

```

Propuesta Ejercicio

Crea *EscribirFichObject2.java*, para que se puedan añadir varios objetos Persona y se pueda leer el fichero sin que salte la excepción *StreamCorruptedException* debido a la inclusión de cabeceras. Para leer el fichero utilizar *LeerfichObjet.java*, cambiando el nombre del fichero por el nuevo FichPersona2.dat

(Si aún no está claro ver ejemplo internet. (colgarlo en el campus)

6.4 Ficheros de acceso aleatorio

Hasta ahora, todas las operaciones que hemos realizado sobre los ficheros se realizaban de forma secuencial.

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

El **ejemplo** siguiente 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.

```

EscribirFichAleatorio.java
3 import java.io.*;
4
5 public class EscribirFichAleatorio {
6     public static void main(String[] args) throws IOException {
7         File fic = new File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\AleatorioEmple.dat");
8         //declara el fichero de acceso aleatorio
9         RandomAccessFile file = new RandomAccessFile(fic, "rw");
10        //arrays con los datos
11        String apellido[] = {"FERNANDEZ","GIL","LOPEZ","RAMOS",
12                             "SEVILLA","CASILLA", "REY"}; //apellidos
13        int dep[] = {10, 20, 10, 10, 30, 30, 20}; //departamentos
14        Double salario[]={1000.45, 2400.60, 3000.0, 1500.56,
15                           2200.0, 1435.87, 2000.0}; //salarios
16
17        StringBuffer buffer = null; //buffer para almacenar apellido
18        int n=apellido.length; //numero de elementos del array
19
20        for (int i=0;i<n; i++){ //recorro los arrays
21            file.writeInt(i+1); //uso i+1 para identificar empleado
22            buffer = new StringBuffer( apellido[i] );
23            buffer.setLength(10); //10 caracteres para el apellido
24            file.writeChars(buffer.toString()); //insertar apellido
25            file.writeInt(dep[i]); //insertar departamento
26            file.writeDouble(salario[i]); //insertar salario
27        }
28        file.close(); //cerrar fichero
29    }

```

A continuación **ejemplo** que a partir del fichero creado en el ejemplo anterior, visualiza todos los registros. Inicialmente se posiciona el puntero al comienzo (0), a continuación hay que ir avanzando el tamaño del registro (4+20+4+8=36).

```

LeerFichAleatorio.java
3 import java.io.*;
4
5 public class LeerFichAleatorio {
6     public static void main(String[] args) throws IOException {
7         File fic = new File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\AleatorioEmple.dat");
8         //declara el fichero de acceso aleatorio
9         RandomAccessFile file = new RandomAccessFile(fic, "r");
10        int id, dep, posicion;
11        Double salario;
12        char apellido[] = new char[10];
13        char aux;
14        boolean finArchivo = false;
15
16        posicion = 0; //para situarnos al principio
17
18        //for(;;){ //recorro el fichero
19        do {
20            try {
21                file.seek(posicion); //nos posicionamos en posicion
22                id = file.readInt(); // obtengo id de empleado
23
24                //recorro uno a uno los caracteres del apellido
25                for (int i = 0; i < apellido.length; i++) {
26                    aux = file.readChar();
27                    apellido[i] = aux; //los voy guardando en el array
28                }
29
30                //convierto a String el array
31                String apellidos = new String(apellido);
32                dep = file.readInt(); //obtengo dep
33                salario = file.readDouble(); //obtengo salario
34
35                if(id > 0)
36                    System.out.printf("ID: %s, Apellido: %s, Departamento: %d, Salario: %.2f %n",
37                                      id, apellidos.trim(), dep, salario);
38
39                //me posiciono para el sig empleado, cada empleado ocupa 36 bytes
40                posicion= posicion + 36;
41            } catch (EOFException eof){
42                finArchivo=true; //Se que he llegado a fin fichero
43                file.close();
44            }
45            //Si he recorrido todos los bytes salgo del for
46            // if (file.getFilePointer() == file.length())break;
47
48            // } //fin bucle for
49            // file.close(); //cerrar fichero
50            }while (!finArchivo);
51        }
52    }
53

```

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):

```
LeerFichAleatorioUnReg.java
15 int registro=5;
16 //Calculamos donde empieza el registro
17 posicion = (registro -1 ) * 36;
18 if(posicion >= file.length() )
19     System.out.printf("ID: %d, NO EXISTE EMPLEADO...",registro);
20 else{
21     file.seek(posicion); //nos posicionamos
22     id=file.readInt(); // obtengo id de empleado
23     //Obtener el resto de los datos
```

Propuesta Ejercicio

Crea *LeerFichAleatorioUnReg.java*, para que se puedan consultar un determinado registro pasando como parámetro el nº de registro a consultar.

Si el registro no existe, nos muestra un mensaje:

```
Problems @ Javadoc Declaration
<terminated> LeerFichAleatorioUnReg [Java Ap
ID: 20, NO EXISTE EMPLEADO...
```

Si existe nos muestra todos los datos del registro:

```
Problems @ Javadoc Declaration Console
<terminated> LeerFichAleatorioUnReg [Java Application] C:\Program Files\Java
ID: 7, Apellido: REY, Departamento: 20, Salario: 2000.0
```

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:


```

11
12  StringBuffer buffer = null; //buffer para almacenar apellido
13  String apellido="GONZALEZ";
14  Double salario=1230.87;
15  int dep=10;
16
17  //INSERTAR UN REGISTRO AL FINAL DEL FICHERO
18  /*long posicion= file.length() ;
19  file.seek(posicion);
20  int id= (int) ((posicion + 36) / 36);
21  */
22  //insertar reg con id 20
23
24  int id=20;
25  long posicion = (id -1 ) * 36;
26  file.seek(posicion); //nos posicionamos
27
28  file.writeInt(id); //uso id para identificar empleado
29  buffer = new StringBuffer( apellido);
30  buffer.setLength(10); //10 caracteres para el apellido
31  file.writeChars(buffer.toString()); //insertar apellido
32  file.writeInt(dep); //insertar departamento
33  file.writeDouble(salario); //insertar salario
34
35  file.close(); //cerrar fichero
36  }
37  }

```

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 . Modificamos el departamento que pasa a ser 40 y el salario que pasa a ser 4000.87

```

1  ModificarFichAleatorio.java
2
3  import java.io.*;
4  public class ModificarFichAleatorio {
5
6  public static void main(String[] args) throws IOException {
7      File fichero = new File("I:\\Curso 2019_2020_Fleming\\Acceso a Datos\\ADPruebas\\AleatorioEmple.dat");
8      //declara el fichero de acceso aleatorio
9      RandomAccessFile file = new RandomAccessFile(fichero, "rw");
10
11      int registro = 4 ; //id a modificar
12
13      long posicion = (registro -1 ) * 36; //(4+20+4+8) calcular la posición
14      posicion=posicion+4+20; //sumo el tamaño de ID+apellido
15      file.seek(posicion); //nos posicionamos
16      file.writeInt(40); //modif departamento
17      file.writeDouble(4000.87); //modif salario
18      file.close(); //cerrar fichero
19  }
20  }

```

Realizar las Actividades 1.3 y 1.4