

Unidad 13. Ficheros en Java.

Programación funcional con ficheros

Contenido

1	Actividad inicial	2
2	Introducción	2
3	Ficheros. Actividad de exposición	2
3.1	CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS ...	2
3.2	FLUJOS O STREAMS. TIPOS.....	7
3.2.1	Flujos de caracteres (Character streams).....	9
3.3	FORMAS DE ACCESO A UN FICHERO.	10
3.4	OPERACIONES SOBRE FICHEROS.	10
3.5	CLASES PARA GESTIÓN DE FLUJOS DE DATOS DESDE/HACIA FICHEROS.	13
3.5.1	Ficheros de texto. Actividad guiada ficheros de texto	13
3.6	Notas Cornell	15
3.6.1	Objetos en ficheros binarios. Actividad guiada ficheros binarios	21
3.6.2	Actividad independiente ficheros binarios	25
3.6.3	ACTIVIDAD INDEPENDIENTE FICHEROS ALEATORIOS	32
3.7	TRABAJO CON FICHEROS XML.	33
3.7.1	Acceso a ficheros XML con DOM. Actividad guiada XML	35
3.7.2	ACTIVIDAD INDEPENDIENTE XML	40
3.7.3	Acceso a ficheros XML con SAX. Actividad guiada SAX	40
4	Programación funcional en ficheros	44
5	La API java.nio.file. Actividad de ampliación	44
5.1	Package java.nio.file	45
5.2	La clase Files	45
5.2.1	Path.....	46
5.2.2	Directory Stream.....	47
5.2.3	Opciones para crear ficheros en con la utilidad Files de Java. .	50
5.2.4	Creando ficheros y directorios. Ejemplo	52
5.2.5	Atributos en Files	55
5.3	El interface FileVisitor	60
5.4	Ejemplos de lectura y escritura de ficheros con la API Files y Java	870
5.4.1	Actividad independiente lectura y escritura de ficheros de texto	75
5.4.2	Actividad independiente de lectura y escritura de ficheros. ...	77
5.5	Almacenando objetos como bytes usando Files.	77
5.5.1	Convertir un Serializable a byte[] y viceversa.....	78
5.5.2	Lambdas y serializable. Muy importante. Actividad guiada lambdas serializables	78
6	Patrones de diseño en ficheros.	84
6.1	Patron Proxy. Actividad guiada patrón proxy	84
6.2	Ejemplo de patrón proxy para ficheros	88
6.3	Patrón visitor. Actividad guiada patrón Visitor	96
7	Bibliografía y referencias web.....	109

1 Actividad inicial

Se proporciona un **archivo binario generado en Java** a los alumnos y se les pide que lo abran con el **bloc de notas**. Tras ver el contenido **se les pregunta porque se ve así y que contiene el fichero**

2 Introducción

El manejo de **archivos (persistencia)**, es un tema fundamental en cualquier lenguaje de programación. Pues nos permite interactuar con los dispositivos de almacenamiento externo para poder mantener la información en el tiempo. Java no es una excepción.

Cuando se desarrollan applets para utilizar en red, hay que tener en cuenta que la entrada/salida directa a fichero es una violación de seguridad de acceso. Muchos usuarios configurarán sus navegadores para permitir el acceso al sistema de ficheros, pero otros no.

Por otro lado, si se está desarrollando una aplicación Java para uso interno, probablemente será necesario el acceso directo a ficheros.

3 Ficheros. Actividad de exposición

Para realizar operaciones sobre los ficheros, necesitamos contar con la información referente sobre un fichero (archivo). La clase **File** proporciona muchas utilidades relacionadas con ficheros y con la obtención de información básica sobre esos ficheros.

Un fichero o archivo es un conjunto de bits almacenados en un dispositivo, como por ejemplo un disco duro. La ventaja de utilizar ficheros es que los **datos que guardamos permanecen en el dispositivo aun cuando apaguemos el ordenador**, es decir, no son volátiles. Los ficheros tienen un nombre y se ubican en directorios o carpetas, el nombre debe ser único en ese directorio; es decir, no puede haber dos ficheros con el mismo nombre en el mismo directorio. Por convención cuentan con diferentes extensiones que por lo general suelen ser de 3 letras (PDF, DOC, GIF, ...) y nos permiten saber el tipo de fichero.

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

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

3.1 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. <i>src\\ejemplos\\VerInf.java</i>
String getAbsolutePath()	Devuelve el camino absoluto del fichero/directorio. <i>D:\\CLASE\\eclipsePhoton\\ADAT\\UNIDAD1\\src\\ejemplos\\VerInf.java</i>
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 File asignándole <i>nuevonombre</i> .
boolean delete()	Borra el fichero o directorio asociado al objeto File .
boolean createNewFile()	Crea un nuevo fichero, vacío, asociado a File si y sólo 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:

```
import java.io.*;
public class VerDir {
    public static void main(String[] args) {
        String dir = "."; //directorio actual
        File f = new File(dir);
        String[] archivos = f.list();

        System.out.printf("Ficheros en el directorio actual: %d %n",
                           archivos.length);

        for (int i = 0; i < archivos.length; i++) {
            File f2 = new File(f, archivos[i]);
            System.out.printf("Nombre: %s, es fichero?: %b,
                               es directorio?:
                               %b %n", archivos[i], f2.isFile(), f2.isDirectory());
        }
    }
}
```

Un ejemplo de ejecución de este programa mostraría la siguiente salida:

```
Ficheros en el directorio actual: 3
Nombre: VerDir.class, es fichero?: true, es directorio?: false
Nombre: VerDir.java, es fichero?: true, es directorio?: false
Nombre: VerInf.java, es fichero?: true, es directorio?: false
```

Dentro del directorio src del proyecto eclipse

tengo el paquete ejemplos (directorio)

```
String dir = ".\\src";
```

Ficheros en el directorio actual: 1

Nombre: ejemplos, es fichero?: false, es directorio?: true

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

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

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

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

ACTIVIDAD 1.1

Realiza un programa Java que utilice el método `listFiles()` para mostrar la lista de ficheros en un directorio cualquiera, o en el directorio actual. (leer directorio desde teclado) . **comprobar si existe**. Asumimos **directorio actual** cuando no introducimos nada

Realiza un programa Java que muestre los ficheros de un directorio. El nombre del directorio se pasará al programa desde los argumentos de `main()`. Si el directorio no existe se debe mostrar un mensaje indicándolo. **Si el nombre de directorio tiene blancos escribir comillas dobles "C:\Program Files"**

Al ejecutar en línea de comandos no olvidar quitar la sentencia package de los java antes de compilarlos(1h)

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

```
import java.io.*;
public class VerInf {
    public static void main(String[] args) {
        System.out.println("INFORMACIÓN SOBRE EL FICHERO:");
        File f = new File("D:\\ADAT\\UNI1\\VerInf.java");

        if(f.exists()){
            System.out.println("Nombre del fichero    : "+f.getName());
            System.out.println("Ruta            : "+f.getPath());
            System.out.println("Ruta absoluta   : " +
                               f.getAbsolutePath());
            System.out.println("Se puede leer    : "+f.canRead());
            System.out.println("Se puede escribir : "+f.canWrite());
            System.out.println("Tamaño          : "+f.length());
            System.out.println("Es un directorio : "+f.isDirectory());
            System.out.println("Es un fichero    : "+f.isFile());
            System.out.println("Nombre del directorio padre: " +
                               f.getParent());
        }
    }
}
```

Visualiza la siguiente información del fichero:

```
INFORMACIÓN SOBRE EL FICHERO:
Nombre del fichero : VerInf.java
Ruta               : D:\ADAT\UNI1\VerInf.java
Ruta absoluta      : D:\ADAT\UNI1\VerInf.java
Se puede leer      : true
Se puede escribir  : true
Tamaño            : 824
Es un directorio   : false
```

```
Es un fichero      : true
Nombre del directorio padre: D:\ADAT\UNI1
```

```
File f = new File("src\\ejemplos\\VerInf.java");
```

```
File f = new File("src\\VerInf.java");
```

INFORMACIÓN SOBRE EL FICHERO:

```
Nombre del fichero : VerInf.java
Ruta               : src\\ejemplos\\VerInf.java
Ruta absoluta     :
D:\\CLASE\\eclipsePhoton\\ADAT\\UNIDAD1\\src\\ejemplos\\VerInf.java
Se puede leer     : true
Se puede escribir : true
Tamaño           : 854
Es un directorio  : false
Es un fichero     : true
Nombre del directorio padre: src\\ejemplos
```

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

```
import java.io.*;
public class CrearDir {
    public static void main(String[] args) {
        File d = new File("NUEVODIR"); //directorio que creo
        File f1 = new File(d,"FICHERO1.TXT");
        File f2 = new File(d,"FICHERO2.TXT");

        d.mkdir(); //CREAR DIRECTORIO

        try {
            if (f1.createNewFile())
                System.out.println("FICHERO1 creado correctamente...");
            else
                System.out.println("No se ha podido crear FICHERO1...");

            if (f2.createNewFile())
                System.out.println("FICHERO2 creado correctamente...");
            else
                System.out.println("No se ha podido crear FICHERO2...");
        } catch (IOException ioe) {ioe.printStackTrace();}

        f1.renameTo(new File(d,"FICHERO1NUEVO")); //renombro FICHERO1

        try {
            File f3 = new File("NUEVODIR/FICHERO3.TXT");
            f3.createNewFile(); //crea FICHERO3 en NUEVODIR
        }
```

```

    } catch (IOException ioe) {ioe.printStackTrace();}
}
}

```

Para borrar un fichero o un directorio usamos el método ***delete()***, en el ejemplo anterior **no podemos borrar el directorio creado porque contiene ficheros**, antes habría que eliminar estos ficheros. Para borrar el objeto *f2* escribimos:

```

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

```

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

3.2 FLUJOS O STREAMS. TIPOS.

(PDF 22 - PROGRAMACION) 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***. La razón de ser de estas clases es la internacionalización; la antigua jerarquía de flujos de E/S sólo soporta flujos de 8 bits no manejando caracteres Unicode de 16 bits que se utilizaba con fines de internacionalización.

1.3.1. Flujos de bytes (Byte streams)

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

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

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

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

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

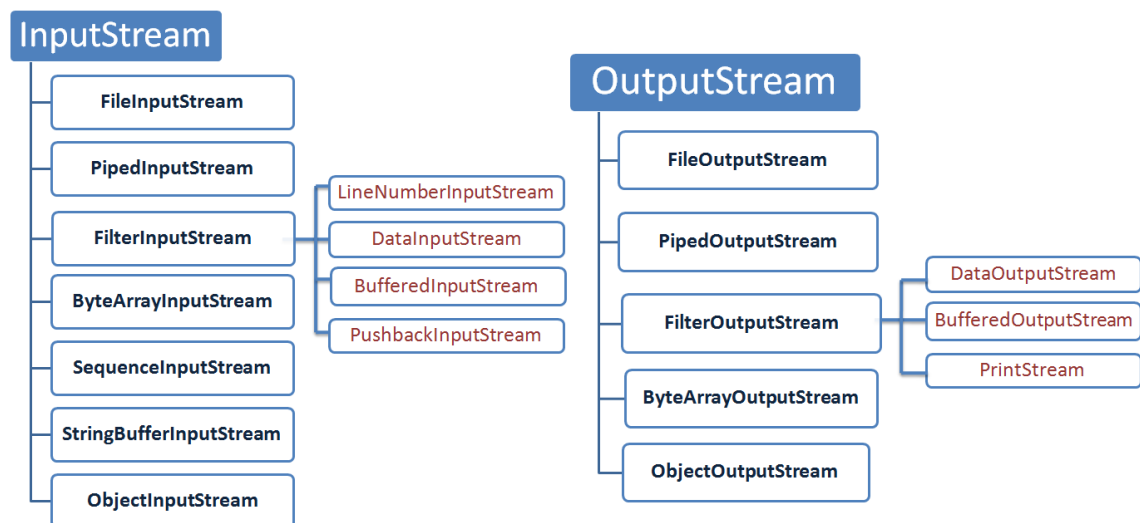


Figura 1.1. Jerarquía de clases para lectura y escritura de bytes.

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

3.2.1 Flujos de caracteres (Character streams)

Las clases **Reader** y **Writer** manejan flujos de caracteres Unicode.

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

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

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

La Figura 1.2 muestra la jerarquía de clases para lectura y escritura de flujos de caracteres.

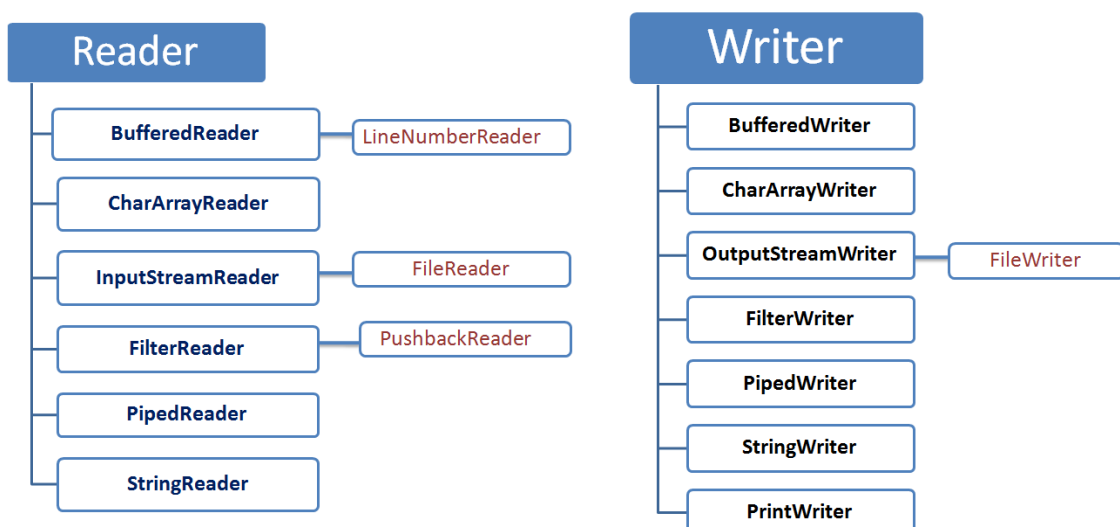


Figura 1.2. Jerarquía de clases para lectura y escritura de flujos de caracteres.

Las clases de flujos de caracteres más importantes son:

- Para acceso a ficheros, lectura y escritura de caracteres en ficheros: **FileReader** y **FileWriter**.
- Para acceso a caracteres, leen y escriben un flujo de caracteres en un array de caracteres: **CharArrayReader** y **CharArrayWriter**.
- Para buferización de datos: **BufferedReader** y **BufferedWriter**, se utilizan para evitar que cada lectura o escritura acceda directamente al fichero, ya que utilizan un buffer intermedio entre la memoria y el stream. (Nos permitirán leer o escribir líneas completas de texto).

3.3 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**.

3.4 OPERACIONES SOBRE FICHEROS.

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

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

alguna variable o variables de nuestro programa en las que se depositarán los datos extraídos del fichero.

- **Escritura de datos en el fichero.** En este caso el proceso consiste en transferir información de la memoria (por medio de las variables del programa) al fichero.

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

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

1.5.1. Operaciones sobre ficheros secuenciales.

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

Veamos como 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 sólo se permite añadir datos al final del fichero.
- **Bajas:** para dar de baja un registro de un fichero es necesario leer todos los registros uno a uno y escribirlos en un fichero auxiliar, salvo el que deseamos dar de baja. Una vez reescritos hemos de borrar el fichero inicial y renombrar el fichero auxiliar dándole el nombre del fichero original.
- **Modificaciones:** consiste en localizar el registro a modificar, efectuar la modificación y reescribir el fichero inicial en otro fichero auxiliar que incluya el registro modificado. El proceso es similar a las bajas.

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

La **desventaja** es que no se puede acceder directamente a un registro determinado, hay que leer antes todos los anteriores; es decir, no soporta acceso aleatorio. **Otra**

desventaja es el proceso de actualización, la mayoría de los ficheros secuenciales no pueden ser actualizados, habrá que reescribirlos totalmente. Para las aplicaciones interactivas que incluyen peticiones o actualizaciones de registros individuales, los ficheros secuenciales ofrecen un rendimiento pobre.

1.5.2. Operaciones sobre ficheros aleatorios.

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

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

- **Consultas:** para consultar un determinado registro necesitamos saber su clave, aplicar la función de conversión a la clave para obtener la dirección y leer el registro ubicado en esa posición. Habría que comprobar si el registro buscado está en esta posición, si no está se buscaría en la zona de excedentes.
- **Altas:** para insertar un registro necesitamos saber su clave, aplicar la función de conversión a la clave para obtener la dirección y escribir el registro en la posición devuelta. Si la posición está ocupada por otro registro, en ese caso el registro se insertaría en la zona de excedentes.
- **Bajas:** las bajas suelen realizarse de forma lógica, es decir, se suele utilizar un campo del registro a modo de switch que tenga el valor 1 cuando el registro exista y le damos el valor 0 para darle de baja, físicamente el registro no desaparece del disco. Habría que localizar el registro a dar de baja a partir de su campo clave y reescribir en este campo el valor 0.
- **Modificaciones:** para modificar un registro hay que localizarlo, necesitamos saber su clave para aplicar la función de conversión y así obtener la dirección, modificar los datos que nos interesen y reescribir el registro en esa posición.

Una de las principales ventajas de los ficheros aleatorios es el rápido acceso a una posición determinada para leer o escribir un registro. El gran inconveniente es establecer la relación entre la posición que ocupa el registro y su contenido; ya que a veces al aplicar la función de conversión para obtener la posición se obtienen posiciones ocupadas y hay que recurrir a la zona de excedentes. Otro inconveniente es

que se puede desaprovechar parte del espacio destinado al fichero ya que se pueden producir huecos (posiciones no ocupadas) entre un registro y otro.

3.5 CLASES PARA GESTIÓN DE FLUJOS DE DATOS DESDE/HACIA FICHEROS.

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

3.5.1 Ficheros de texto. Actividad guiada ficheros de texto

Los ficheros de texto, los que normalmente se generan con un editor, almacenan caracteres alfanuméricos en un formato estándar (ASCII, UNICODE, UTF8, etc). Para trabajar con ellos usaremos las clases **FileReader** para leer caracteres y **FileWriter** para escribir los caracteres en el fichero. Cuando trabajamos con ficheros, cada vez que leemos o escribimos en uno debemos hacerlo dentro de un manejador de excepciones **try-catch**.

Al usar la clase **FileReader** se puede generar la excepción **FileNotFoundException** (porque el nombre del fichero no exista o no sea válido) y al usar la clase **FileWriter** la excepción **IOException** (el disco está lleno o protegido contra escritura).

Los métodos que proporciona la clase **FileReader** para lectura son los siguientes, estos métodos devuelven un número (que si se hace un cast es el carácter leído) 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. (entero es el ascii del carácter)
<code>int read(char[] buf)</code>	Lee hasta <code>buf.length</code> caracteres de datos de una matriz de caracteres (<code>buf</code>). Los caracteres leídos del fichero se van almacenando en <code>buf</code> .
<code>int read(char[] buf, int desplazamiento, int n)</code>	Lee hasta <code>n</code> caracteres de datos de la matriz <code>buf</code> comenzando por <code>buf[desplazamiento]</code> y devuelve el número leído de caracteres.

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

El siguiente ejemplo lee cada uno de los caracteres del fichero de texto de nombre *LeerFichTexto.java* (localizado en la carpeta *C:\EJERCICIOS\UN1*) y los muestra en pantalla, los métodos **read()** pueden lanzar la excepción **IOException**, por ello en **main()** se ha añadido **throws IOException** ya que no se incluye el manejador **try-catch**:

```
import java.io.*;
public class LeerFichTexto {
```

```

public static void main(String[] args) throws IOException {
    //declarar fichero
    File fichero =
        new File("C:\\EJERCICIOS\\UNI1\\LeerFichTexto.java");
    //crear el flujo de entrada hacia el fichero
    FileReader fic = new FileReader(fichero);

    int i;
    while ((i = fic.read()) != -1) //se va leyendo un carácter
        System.out.println((char) i);

    fic.close(); //cerrar fichero

    //lo mismo de antes
    System.out.println("DE OTRA MANERA (SEUDOCODIGO):");
    fic = new FileReader(fichero);
    i = fic.read();
    while (i != -1){ //MIENTRAS NO SEA FF
        System.out.print((char) i);
        i = fic.read();
    }

    fic.close(); //cerrar fichero

    System.out.println("\nSE VAN LEYENDO DE 5 EN 5:");
    System.out.println("=====");
    fic = new FileReader(fichero);
    char [] b= new char[5];
    while ((i = fic.read(b)) != -1){
        System.out.println(b);
        b= new char[5];
    }
    fic.close(); //cerrar fichero
}
}

```

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

```

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

```

Para ir leyendo de 20 en 20 caracteres escribimos:

```

char [] b= new char[20];
while ((i = fic.read(b)) != -1) System.out.println(b);
// para que limpie el buffer y no repita caracteres añadir b = new char[20];

```

3.6 Notas Cornell

Comentar las líneas de código señaladas con el marcador y explicar que hacen paso a paso. Apoyaros en los apuntes para generar los comentarios.

ACTIVIDAD INDEPENDIENTE FICHEROS DE TEXTO

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

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

Método	Función
<code>void write(int c)</code>	Escribe un carácter.
<code>void write(char[] buf)</code>	Escribe un array de caracteres.
<code>void write(char[] buf, int desplazamiento, int n)</code>	Escribe n caracteres de datos en la matriz <i>buf</i> comenzando por <i>buf[desplazamiento]</i> .
<code>void write(String str)</code>	Escribe una cadena de caracteres.
<code>void append(char c)</code>	Añade un carácter a un fichero.

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

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

        String cadena ="Esto es una prueba con FileWriter";

        //convierte la cadena en array de caracteres para extraerlos 1 a 1
        char[] cad = cadena.toCharArray();

        for(int i=0; i<cad.length; i++)
            fic.write(cad[i]); //se va escribiendo un carácter

        fic.append('*'); //se añade al final un *

        fic.close(); //cerrar fichero
    }
}
```

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

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

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

(2H)

Hay que 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** de la siguiente manera, colocando en el segundo parámetro del constructor el valor **true**:

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

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

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

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

```
import java.io.*;
public class LeerFichTextoBuf {
    public static void main(String[] args) {
        try{
            BufferedReader fichero = new BufferedReader(
                new FileReader("LeerFichTexto.java"));

            String linea;
            while((linea = fichero.readLine()) != null)
                System.out.println(linea);

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

LEER UN FICHERO DE TEXTO, BUSCAMOS PALABRAS Y TODAS LAS PALABRAS CON AL MENOS UNA LETRA a SE GUARDARÁN EN UN FICHERO DE TEXTO DE SALIDA

EXPRESIÓN REGULAR PARA BUSCAR PALABRAS SEPARADAS POR UNO O MÁS ESPACIOS EN BLANCO EN UNA CADENA

<http://puntocomnoesunlenguaje.blogspot.com/2013/07/ejemplos-expresiones-regulares-java-split.html>


```
String array[] = linea.split("\\s{1,}");  
linea.split("\\s+")
```

\s => representa un caracter en blanco

{1,} => Busca 1 o más en la cadena, en este caso caracteres en blanco \s)

```
File fic = new File("FichTexto.txt");// declara fichero  
File fic2 = new File("SOLO_LETRAS_a.txt");  
  
BufferedReader fichero = new BufferedReader(new FileReader(fic));  
FileWriter fichero2 = new FileWriter(fic2);  
  
String linea;  
while ((linea = fichero.readLine()) != null) {  
    String array[] = linea.split("\\s{1,}"); //array palabras  
    for (int i = 0; i < array.length; i++) {  
        if (array[i].indexOf('a') >= 0) {  
            fichero2.write(array[i]);  
            fichero2.write("\n");  
        }  
    }  
}  
  
fichero.close();  
fichero2.close();  
  
// VISUALIZAR CONTENIDO DE FICHERO2  
BufferedReader fichero2R = new BufferedReader  
    (new FileReader(fic2));  
  
System.out.println("CONTENIDO DEL FICHERO CREADO:");  
  
while ((linea = fichero2R.readLine()) != null)  
    System.out.println(linea);  
  
fichero2R.close();
```

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

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

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

```
import java.io.*;  
public class EscribirFichTextoBuf {  
    public static void main(String[] args) {  
        try{  
            BufferedWriter fichero = new BufferedWriter  
                (new FileWriter("FichTexto.txt"));  
            for (int i=1; i<11; i++){  
                fichero.write("Fila numero: "+i); //escribe una línea  
                fichero.newLine(); //escribe un salto de línea  
            }  
            fichero.close();  
        }  
        catch (FileNotFoundException fn ){  
            System.out.println("No se encuentra el fichero");}  
        catch (IOException io) {  
            System.out.println("Error de E/S ");}  
    }  
}
```

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

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

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

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

1. 6.2 Ficheros binarios.

Los ficheros binarios almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario como ocurría con los ficheros de texto. Tienen la ventaja de que ocupan menos espacio en disco. En Java, las dos clases que nos permiten trabajar con ficheros son **FileInputStream** (para

entrada) y **FileOutputStream** (para salida), estas trabajan con flujos de bytes y crean un enlace entre el flujo de bytes y el fichero.

Los métodos que proporciona la clase **FileInputStream** para lectura son similares a los vistos para la clase **FileReader**, estos métodos devuelven el byte o bytes leídos ~~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:

```
import java.io.*;
public class EscribirFichBytes {
    public static void main(String[] args) throws IOException {
        File fichero = new
            File("C:\\EJERCICIOS\\UN11\\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**, recorreremos 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

```

EJERCICIO QUE LEA UN NOMBRE DE TECLADO Y VUSUALICE SU EDAD.

3.6.1 Objetos en ficheros binarios. **Actividad guiada ficheros binarios**

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

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

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

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

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

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

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

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

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

O bien:

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

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

```
import java.io.*;
public class EscribirFichObject {
    public static void main(String[] args) throws IOException {
        Persona persona;//defino variable persona
        //declara el fichero
        File fichero = new File("FichPersona.dat");
        //crea el flujo de salida
        FileOutputStream fileout = new FileOutputStream(fichero);
        //conecta el flujo de bytes al flujo de datos
        ObjectOutputStream dataOS = new ObjectOutputStream(fileout);

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

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

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

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

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

O bien:

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

El método **readObject()** lee los objetos del flujo de entrada, puede lanzar la excepción **ClassNotFoundException** e **IOException**, por lo que será necesario controlarlas. El proceso de lectura se hace en un bucle **while(true)**, este se encierra en un bloque **try-catch** ya que la lectura finalizará cuando se

llegue al final de fichero, entonces se lanzará la **excepción *EOFException***. El código es el siguiente:

```
import java.io.*;

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

        try {
            while (true) { //lectura del fichero
                persona= (Persona) dataIS.readObject(); //leer una Persona
                System.out.printf("Nombre: %s, edad: %d %n",
                    persona.getNombre(), persona.getEdad());
            }
        } catch (EOFException eo) {
            System.out.println("FIN DE LECTURA.");
        }

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

Problema con los ficheros de objetos:

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

La cabecera se crea cada vez que se pone ***new ObjectOutputStream(fichero)***. Para que no se añadan estas cabeceras lo que se hace es ***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

MODIFICAR CLASE EscribirFichObject, GUARDAR CLASE CON OTRO NOMBRE.
USANDO LA CLASE MiObjectOutputStream
```

3.6.2 Actividad independiente ficheros binarios

1. Realiza un programa Java que cree un fichero binario para guardar datos de departamentos (**Departamentos.dat**), dale el nombre *Departamentos.dat*. Introduce varios departamentos. Los datos por cada departamento son: *Número de departamento (1 y 99): entero, Nombre: String y Localidad: String*. LECTURA DE DATOS POR TECLADO, PROCESO DE LECTURA FINALIZA CUANDO EL NUMERO DE DEPARTAMENTO ES 0. PANTALLA GRAFICA

AÑADIR CONSULTA DE UN DEP. LEIDO de TECLADO.

2. Realiza un programa Java que te permita modificar los datos de un departamento. ~~El programa recibe desde la línea de comandos~~ (**LECTURA POR TECLADO**) el **número de departamento a modificar, el nuevo nombre de departamento y la nueva localidad**. Si el departamento no existe visualiza un mensaje indicándolo. Visualiza también los datos antiguos del departamento y los nuevos datos. **LEER POR TECLADO LOS DATOS A MODIFICAR. EL PROCESO FINALIZA CUANDO EL DEP LEIDO POR TECLADO SEA 0.**

1. 6.4 Ficheros de acceso aleatorio. **Actividad guiada ficheros aleatorios**

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 <code>IOException</code> .
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:

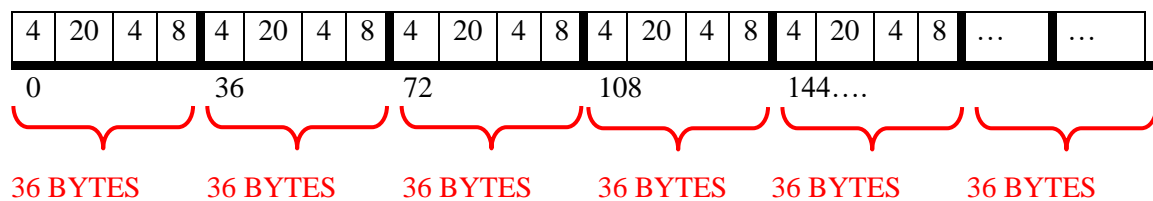
Método	Función
<code>long getFilePointer()</code>	Devuelve la posición actual del puntero del fichero.
<code>void seek(long posicion)</code>	Coloca el puntero del fichero en una posición determinada desde el comienzo del mismo.
<code>long length()</code>	Devuelve el tamaño del fichero en bytes. <code>length()</code> 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 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**.



$$\text{POSICION DE UN EMPLEADO} = (\text{IDENTIFICADOR} - 1) * 36$$

FICHERO DEPARTAMENTOS

Int numero de departamento => (4 bytes)

String nombre de departamento (12 caracteres)

String localidad (15 caracteres)

Longitud del registro: 4 + 24 + 30 => 58

Numero de departamento es el ID DEL REGISTRO

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

(2H)

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
}
}
```

PARA ESCRIBIR CARACTERES USAMOS: **void writeChars(String s)**.
Escribe una cadena en el fichero como una secuencia de caracteres.

PARA LEER UN CARÁCTER USAMOS: **char readChar()**. Lee un carácter del fichero. PARA LAS CADENAS DE CARACTERES VAMOS LEYENDO CARÁCTER A CARÁCTER.

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
            //si recorro el fichero secuencialmente
            //puedo quitar el posicionamiento seek().

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

            //convierto a String el array
            String apellidos = new String(apellido);

            dep = file.readInt(); //obtengo dep
            salario = file.readDouble(); //obtengo salario

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

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

```

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

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

```

La ejecución muestra la siguiente salida:

```

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

```

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

```

int identificador = 5;
//calculo donde empieza el registro
posicion = (identificador - 1 ) * 36;

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

```

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

```

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

```

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

```

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

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

file.seek(posicion); //nos posicionamos

```

```

file.writeInt(id);    //se escribe id

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

file.writeInt(dep);    //insertar departamento
file.writeDouble(salario); //insertar salario

file.close(); //cerrar fichero

```

(1H)

Crear FICHERO DEPARTAMENTOS con datos leídos de teclado HASTA QUE EL departamento ≤ 0 :

Int numero de departamento (1 y 99) \Rightarrow (4)

String nombre de departamento (12 caracteres)

String localidad (15 caracteres)

Longitud del registro: $4 + 24 + 30 \Rightarrow 58$

Numero de departamento es el ID DEL REGISTRO

(COMPROBAR SI EL REG EXISTE, SI EXISTE MUESTRA UN MENSAJE Y NO SE INSERTARA)

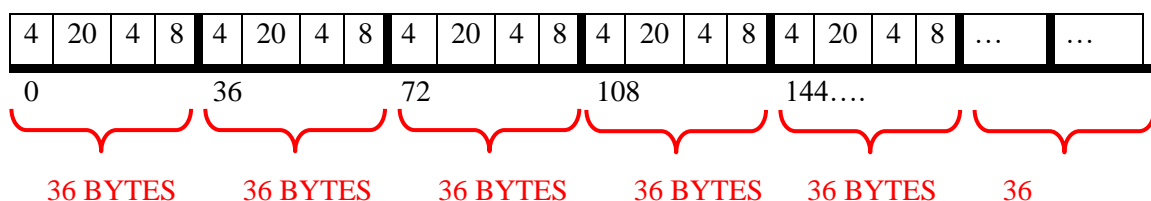
CreaFichAleatorioDepTeclado.java

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
posicion = posicion + 4 + 20; //sumo el tamaño de ID + apellido
file.seek(posicion); //nos posicionamos
file.writeInt(40); //modifico departamento
file.writeDouble(4000.87); //modifico salario

```



3.6.3 ACTIVIDAD INDEPENDIENTE FICHEROS ALEATORIOS

Consulta. Crea un programa Java que consulte los datos de un empleado del fichero aleatorio. Si el empleado existe se visualizarán sus datos, si no existe se visualizará un mensaje indicándolo. **TECLADO HASTA DEP < 0 O = 0**

CREAR EL FICHERO JAR Y EJECUTARLO DESDE A LINEA DE COMANDOS

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 (>0), apellido, departamento (>0) y salario. Antes de insertar se comprobará si el identificador existe, en ese caso se debe visualizar un mensaje indicándolo (NO SE INSERTA); si no existe se deberá insertar.

CREAR EL FICHERO JAR Y EJECUTARLO DESDE A LINEA DE COMANDOS

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
posicion = posicion + 4 + 20; //sumo el tamaño de ID + apellido
file.seek(posicion); //nos posicionamos
file.writeInt(40); //modifico departamento
file.writeDouble(4000.87); //modifico salario
```


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. **EL SALARIO Y EL IDENTIFICADOR TIENEN QUE SER >0**

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 (*IDENTIFICADOR*) y el departamento y salario serán 0. **EL IDENTIFICADOR TIENE QUE SER >0**

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

3.6.4 Actividad de refuerzo ficheros

Crea una aplicación que almacene los datos básicos de un vehículo como la matrícula(String), marca (String), tamaño de deposito (double) y modelo (String) en ese orden y de uno en uno usando la clase `DataInputStream`.

Los datos anteriores datos se pedirán por teclado y se irán añadiendo al fichero (no se sobrescriben los datos) cada vez que ejecutemos la aplicación.

El fichero siempre será el mismo, en todos los casos.

Muestra todos los datos de cada coche en la consola.

3.7 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>
    <dep>10</dep>
    <salario>1000.45</salario>
  </empleado>
  <empleado>
    <id>2</id>
    <apellido>GIL</apellido>
    <dep>20</dep>
    <salario>2400.6</salario>
  </empleado>
  <empleado>
    <id>3</id>
    <apellido>LOPEZ</apellido>
    <dep>10</dep>
    <salario>3000.0</salario>
  </empleado>
</Empleados>
```

Los ficheros XML se pueden utilizar para proporcionar datos a una base de datos, o para almacenar copias de partes del contenido de la base de datos. También se utilizan para escribir ficheros de configuración de programas o en el protocolo SOAP (*Simple Object Access Protocol*), para ejecutar comandos en servidores remotos; la información enviada al servidor remoto y el resultado de la ejecución del comando se envían en ficheros XML.

Para leer los ficheros XML y acceder a su contenido y estructura, se utiliza un procesador de XML o parser. El procesador lee los documentos y proporciona acceso a su contenido y estructura. Algunos de los procesadores más empleados son: **DOM**: *Modelo de Objetos de Documento* y **SAX**: *API Simple para XML*. Son independientes del lenguaje de programación y existen versiones particulares para Java, VisualBasic, C, etc. Utilizan dos enfoques muy diferentes:

- **DOM**: un procesador XML que utilice este planteamiento almacena toda la estructura del documento en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales (que son aquéllos que no tienen descendientes). Una vez creado el árbol, se van recorriendo los diferentes nodos y se analiza a qué tipo particular pertenecen. Tiene su origen en el W3C. Este tipo de procesamiento necesita más recursos de memoria y tiempo sobre todo si los ficheros XML a procesar son bastante grandes y complejos.
- **SAX**: un procesador que utilice este planteamiento lee un fichero XML de forma secuencial y produce una secuencia de eventos (comienzo/fin del documento, comienzo/fin de una etiqueta,

etcétera) en función de los resultados de la lectura. Cada evento invoca a un método definido por el programador. Este tipo de procesamiento **prácticamente no consume memoria**, pero por otra parte, impide tener una visión global del documento por el que navegar.

3.7.1 Acceso a ficheros XML con DOM. **Actividad guiada XML**

Para poder trabajar con DOM en Java necesitamos las clases e interfaces que componen el paquete **org.w3c.dom** (contenido en el JSDK) y el paquete **javax.xml.parsers** del API estándar de Java que proporciona un par de clases abstractas que toda implementación DOM para Java debe extender. Estas clases ofrecen métodos para cargar documentos desde una fuente de datos (fichero, **InputStream**, etc.). Contiene dos clases fundamentales: **DocumentBuilderFactory** y **DocumentBuilder**.

DOM no define ningún mecanismo para generar un fichero XML a partir de un árbol DOM. Para eso usaremos el paquete **javax.xml.transform** que permite especificar una fuente y un resultado. La fuente y el resultado pueden ser ficheros, flujos de datos o nodos DOM entre otros.

Los programas Java que utilicen DOM necesitan estas interfaces (no se exponen todas, sólo algunas de las que usaremos en los ejemplos):

- **Document.** Es un objeto que equivale a un ejemplar de un documento XML. Permite crear nuevos nodos en el documento.
- **Element.** Cada elemento del documento XML tiene un equivalente en un objeto de este tipo. Expone propiedades y métodos para manipular los elementos del documento y sus atributos.
- **Node.** Representa a cualquier nodo del documento.
- **NodeList.** Contiene una lista con los nodos hijos de un nodo.
- **Attr.** Permite acceder a los atributos de un nodo.
- **Text.** Son los datos carácter de un elemento.
- **CharacterData.** Representa a los datos carácter presentes en el documento. Proporciona atributos y métodos para manipular los datos de caracteres.
- **DocumentType.** Proporciona información contenida en la etiqueta **<!DOCTYPE>**.

A continuación vamos **a crear un fichero XML a partir del fichero aleatorio** de empleados creado en el epígrafe anterior. Lo primero que hemos de hacer es importar los paquetes necesarios:

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import java.io.*;
```

A continuación creamos una instancia de **DocumentBuilderFactory** para construir el parser, se debe encerrar entre **try-cath** porque se puede producir la excepción **ParserConfigurationException**:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
try{
    DocumentBuilder builder = factory.newDocumentBuilder();
    . . . . .
}
```

Creamos un documento vacío de nombre *document* con el nodo raíz de nombre *Empleados* y asignamos la versión del XML, la interfaz **DOMImplementation** permite crear objetos **Document** con nodo raíz:

```
DOMImplementation implementation = builder.getDOMImplementation();
Document document = implementation.createDocument
    (null, "Empleados", null);
document.setXmlVersion("1.0"); // asignamos la version de nuestro XML
```

El siguiente paso sería recorrer el fichero con los datos de los empleados y por cada registro crear un nodo empleado con 4 hijos (*id*, *apellido*, *dep* y *salario*). Cada nodo hijo tendrá su valor (por ejemplo: *1*, *FERNANDEZ*, *10*, *1000.45*). Para crear un elemento usamos el método **createElement(String)** llevando como parámetro el nombre que se pone entre las etiquetas menor que y mayor que. El siguiente código crea y añade el nodo **<empleado>** al documento:

```
//creamos el nodo empleado
Element raiz = document.createElement("empleado");
//lo pegamos a la raiz del documento
document.getDocumentElement().appendChild(raiz);
```

A continuación se añaden los hijos de ese nodo (raiz), estos se añaden en el método **CrearElemento()**:

```
//añadir ID
CrearElemento("id", Integer.toString(id), raiz, document);
//añadir APELLIDO
CrearElemento("apellido", apellidoS.trim(), raiz, document);
//añadir DEP
CrearElemento("dep", Integer.toString(dep), raiz, document);
//añadir SALARIO
CrearElemento("salario", Double.toString(salario), raiz, document);
```

Como se puede ver el método recibe el nombre del nodo hijo (*id*, *apellido*, *dep* o *salario*) y sus textos o valores que tienen que estar en formato String (*1*, *FERNANDEZ*, *10*, *1000.45*), el nodo al que se va a añadir (*raiz*) y el documento (*document*). Para crear el nodo hijo (<id> o <apellido> o <dep> o <salario>) se escribe:

```
Element elem = document.createElement(datoEmple); //creamos un hijo
```

Para añadir su valor o su texto se usa el método **createTextNode(String)**:

```
Text text = document.createTextNode(valor); //damos valor
```

A continuación se añade el nodo hijo a la raíz (empleado) y su texto o valor al nodo hijo:

```
raiz.appendChild(elem); //pegamos el elemento hijo a la raiz
elem.appendChild(text); //pegamos el valor al elemento
```

Al final se generaría algo similar a esto por cada empleado:

```
<empleado><id>1</id><apellido>FERNANDEZ</apellido><dep>10</dep><salario>1000.45</salario></empleado>
```

El método es el siguiente:

```
static void CrearElemento(String datoEmple, String valor,
                          Element raiz, Document document ){
    Element elem = document.createElement(datoEmple); //creamos hijo
    Text text = document.createTextNode(valor);        //damos valor
    raiz.appendChild(elem); //pegamos el elemento hijo a la raiz
    elem.appendChild(text); //pegamos el valor
}
```

En los últimos pasos se crea la fuente XML a partir del documento:

```
Source source = new DOMSource(document);
```

Se crea el resultado en el fichero *Empleados.xml*:

```
Result result = new StreamResult
    (new java.io.File("Empleados.xml")); //fichero XML
```

Se obtiene un **TransformerFactory**:

```
Transformer transformer =
    TransformerFactory.newInstance().newTransformer();
```

Se realiza la transformación del documento a fichero:

```
transformer.transform(source, result);
```

Para mostrar el documento por pantalla podemos especificar como resultado el canal de salida *System.out*:

```
Result console = new StreamResult(System.out);
transformer.transform(source, console);
```

El código completo es el siguiente:

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import java.io.*;

public class CrearEmpleadoXml {
    public static void main(String args[]) throws IOException{
        File fichero = new File("AleatorioEmple.dat");
        RandomAccessFile file = new RandomAccessFile(fichero, "r");

        int id, dep, posicion=0; //para situarnos al principio del fichero
        Double salario;
        char apellido[] = new char[10], aux;
```

```

DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();

try{
    DocumentBuilder builder = factory.newDocumentBuilder();
    DOMImplementation implementation = builder.getDOMImplementation();
    Document document =
        implementation.createDocument(null, "Empleados", null);
    document.setXmlVersion("1.0");

    for(;;) {
        file.seek(posicion); //nos posicionamos
        id=file.readInt();    // obtengo id de empleado
        for (int i = 0; i < apellido.length; i++) {
            aux = file.readChar();
            apellido[i] = aux;
        }
        String apellidos = new String(apellido);
        dep = file.readInt();
        salario = file.readDouble();

        if(id>0) { //id validos a partir de 1
            Element raiz =
                document.createElement("empleado"); //nodo empleado
            document.getDocumentElement().appendChild(raiz);

            //añadir ID
            CrearElemento("id",Integer.toString(id), raiz, document);
            //Apellido
            CrearElemento("apellido",apellidos.trim(), raiz, document);
            //añadir DEP
            CrearElemento("dep",Integer.toString(dep), raiz, document);
            //añadir salario
            CrearElemento("salario",Double.toString(salario), raiz,
                                                                    document);
        }
        posicion= posicion + 36; // me posiciono para el sig empleado

        if (file.getFilePointer() == file.length()) break;
    }
} //fin del for que recorre el fichero

Source source = new DOMSource(document);
Result result =
    new StreamResult(new java.io.File("Empleados.xml"));
Transformer transformer =
    TransformerFactory.newInstance().newTransformer();
transformer.transform(source, result);

}catch(Exception e){ System.err.println("Error: "+e); }

file.close(); //cerrar fichero
} //fin de main

//Inserción de los datos del empleado
static void CrearElemento(String datoEmple, String valor,
                        Element raiz, Document document){
    Element elem = document.createElement(datoEmple);
    Text text = document.createTextNode(valor); //damos valor
    raiz.appendChild(elem); //pegamos el elemento hijo a la raiz
}

```

```

        elem.appendChild(text); //pegamos el valor
    }
} //fin de la clase

```

A PARTIR DEL FICHERO DE OBJETOS FichDepartamentos.dat CREA UN XML:

```

<Departamentos>
  <departamento>
    <dep> ...</dep><nombre>...</nombre><loc>...</loc>
  </departamento>
  <departamento>
    <dep> ...</dep><nombre>...</nombre><loc>...</loc>
  </departamento>
  <departamento>
    <dep> ...</dep><nombre>...</nombre><loc>...</loc>
  </departamento>
  . . . . .
</Departamentos>
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Departamentos>
<departamento><dep>10</dep><nombre>CONTABILIDAD</nombre><loc>MADRID</loc>
</departamento>
<departamento><dep>20</dep><nombre>VENTAS</nombre><loc>SEVILLA</loc>
</departamento>
<departamento><dep>30</dep><nombre>COMERCIO</nombre><loc>TOLEDO</loc>
</departamento>
<departamento><dep>40</dep><nombre>INFORMATICA</nombre><loc>BILBAO</loc>
</departamento>
<departamento><dep>50</dep><nombre>PRODUCCION</nombre><loc>GUADALAJARA</loc>
</departamento>
</Departamentos>

```

Para leer un documento XML, creamos una instancia de **DocumentBuilderFactory** para construir el parser y cargamos el documento con el método **parse()**:

```
Document document = builder.parse(new File("Empleados.xml"));
```

Obtenemos la lista de nodos con nombre *empleado* de todo el documento:

```
NodeList empleados = document.getElementsByTagName("empleado");
```

Se realiza un bucle para recorrer esta lista de nodos. ~~Por cada nodo se obtienen sus etiquetas y sus valores llamando a la función **getNode()**.~~ El código es el siguiente:

```

import java.io.File;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class LecturaEmpleadoXml {
    public static void main(String[] args) {

        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
    }
}

```

```

try {
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document document = builder.parse(new File("Empleados.xml"));
    document.getDocumentElement().normalize();

    System.out.printf("Elemento raiz: %s %n",
        document.getDocumentElement().getNodeName());
    //crea una lista con todos los nodos empleado
    NodeList empleados = document.getElementsByTagName("empleado");
    System.out.printf("Nodos empleado a recorrer: %d %n",
        empleados.getLength());

    //recorrer la lista
    for (int i = 0; i < empleados.getLength(); i++) {
        Node emple = empleados.item(i); //obtener un nodo empleado
        if (emple.getNodeType() == Node.ELEMENT_NODE) { //tipo de nodo
            //obtener los elementos del nodo
            Element elemento = (Element) emple;
            System.out.printf("ID = %s %n",
                elemento.getElementsByTagName("id").
                    item(0).getTextContent());
            System.out.printf(" * Apellido = %s %n",
                elemento.getElementsByTagName("apellido").
                    item(0).getTextContent());
            System.out.printf(" * Departamento = %s %n",
                elemento.getElementsByTagName("dep").
                    item(0).getTextContent());
            System.out.printf(" * Salario = %s %n",
                elemento.getElementsByTagName("salario").
                    item(0).getTextContent());
        }
    }
} catch (Exception e)
{e.printStackTrace();}

} //fin de main
} //fin de la clase

```

LECTURA DEL XML DE DEPARTAMENTOS CREADO ANTERIORMENTE.

INTRODUCIR POR TECLADO UN NUMERO DE DEPARTAMENTO Y VISUALIZAR SUS DATOS (CONSULTA DEL XML ANTERIOR)

EL PROCESO DE LECTURA DE TECLADO ES REPETITIVO HASTA QUE EL DEP SEA 0

3.7.2 ACTIVIDAD INDEPENDIENTE XML

A partir del fichero de objetos Persona utilizado anteriormente crea un documento XML usando DOM.

3.7.3 Acceso a ficheros XML con SAX. Actividad guiada SAX

SAX (*API Simple para XML*) es un conjunto de clases e interfaces que ofrecen una herramienta muy útil para el procesamiento de documentos XML. Permite

analizar los documentos de forma secuencial (es decir, no carga todo el fichero en memoria como hace DOM), esto implica poco consumo de memoria aunque los documentos sean de gran tamaño, en contraposición, impide tener una visión global del documento que se va a analizar. SAX es más complejo de programar que DOM, es un API totalmente escrita en Java e incluida dentro del JRE que nos permite crear nuestro propio parser de XML.

La lectura de un documento XML produce eventos que ocasiona la llamada a métodos, los eventos son encontrar la etiqueta de inicio y fin del documento (*startDocument()* y *endDocument()*), la etiqueta de inicio y fin de un elemento (*startElement()* y *endElement()*), los caracteres entre etiquetas (*characters()*), etc:

Documento XML (alumnos.xml)	Métodos asociados a eventos del documento
<pre><?xml version="1.0"?> <listadealumnos> <alumno> <nombre> Juan </nombre> <edad> 19 </edad> </alumno> <alumno> <nombre> Maria </nombre> <edad> 20 </edad> </alumno> </listadealumnos></pre>	<pre>startDocument() startElement() startElement() startElement() characters() endElement() startElement() characters() endElement() endElement() endElement() startElement() startElement() characters() endElement() startElement() characters() endElement() endElement() endElement() endDocument()</pre>

Vamos a construir un ejemplo sencillo en Java que muestra los pasos básicos necesarios para hacer que se puedan tratar los eventos. En primer lugar se incluyen las clases e interfaces de SAX:

```
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;
```

Se crea un objeto procesador de XML, es decir un **XMLReader**, durante la creación de este objeto se puede producir una excepción (**SAXException**) que es necesario capturar (se incluye en el método *main()*):

```
XMLReader procesadorXML = XMLReaderFactory.createXMLReader();
```

A continuación hay que indicar al **XMLReader** qué objetos poseen los métodos que tratarán los eventos. Estos objetos serán normalmente implementaciones de las siguientes interfaces:

- **ContentHandler**: recibe las notificaciones de los eventos que ocurren en el documento.
- **DTDHandler**: recoge eventos relacionados con la DTD.
- **ErrorHandler**: define métodos de tratamientos de errores.
- **EntityResolver**: sus métodos se llaman cada vez que se encuentra una referencia a una entidad.
- **DefaultHandler**: clase que provee una implementación por defecto para todos sus métodos, el programador definirá los métodos que sean utilizados por el programa. Esta clase es de la que extenderemos para poder crear nuestro parser de XML. En el ejemplo la clase se llama *GestionContenido* y se tratan sólo los eventos básicos: inicio y fin de documento, inicio y fin de etiqueta encontrada, encuentra datos carácter (*startDocument()*, *endDocument()*, *startElement()*, *endElement()*, *characters()*):
 - *startDocument*: se produce al comenzar el procesado del documento XML.
 - *endDocument*: se produce al finalizar el procesado del documento XML.
 - *startElement*: se produce al comenzar el procesado de una etiqueta XML. Es aquí donde se leen los atributos de las etiquetas.
 - *endElement*: se produce al finalizar el procesado de una etiqueta XML.
 - *characters*: se produce al encontrar una cadena de texto.

Para indicar al procesador XML los objetos que realizarán el tratamiento se utiliza alguno de los siguientes métodos incluidos dentro de los objetos **XMLReader**: *setContentHandler()*, *setDTDHandler()*, *setEntityResolver()* y *setErrorHandler()*; cada uno trata un tipo de evento y está asociado con una interfaz determinada. En el ejemplo usaremos *setContentHandler()* para tratar los eventos que ocurren en el documento:

```
GestionContenido gestor = new GestionContenido();
procesadorXML.setContentHandler(gestor);
```

A continuación se define el fichero XML que se va a leer mediante un objeto **InputSource**:

```
InputSource fileXML = new InputSource("alumnos.xml");
```

Por último se procesa el documento XML mediante el método *parse()* del objeto **XMLReader**, le pasamos un objeto **InputSource**:

```
procesadorXML.parse(fileXML);
```

El ejemplo completo se muestra a continuación:

```
import java.io.*;
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
```

```

import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;

public class PruebaSax1 {
    public static void main(String[] args)
        throws FileNotFoundException, IOException, SAXException{

        XMLReader procesadorXML = XMLReaderFactory.createXMLReader();
        GestionContenido gestor = new GestionContenido();
        procesadorXML.setContentHandler(gestor);
        InputSource fileXML = new InputSource("alumnos.xml");
        procesadorXML.parse(fileXML);
    }
}

//fin PruebaSax1

class GestionContenido extends DefaultHandler {
    public GestionContenido() {
        super();
    }
    public void startDocument() {
        System.out.println("Comienzo del Documento XML");
    }
    public void endDocument() {
        System.out.println("Final del Documento XML");
    }
    public void startElement(String uri, String nombre,
        String nombreC, Attributes atts) {
        System.out.printf("\tPrincipio Elemento: %s %n", nombre);
    }
    public void endElement(String uri, String nombre,
        String nombreC) {
        System.out.printf("\tFin Elemento: %s %n", nombre);
    }
    public void characters(char[] ch, int inicio, int longitud)
        throws SAXException {
        String car = new String(ch, inicio, longitud);
        //quitar saltos de línea
        car = car.replaceAll("[\t\n]", "");
        System.out.printf ("\tCaracteres: %s %n", car);
    }
}

//fin GestionContenido

```

En el resultado de ejecutar el programa con el fichero *alumnos.xml* se puede observar como el orden de ocurrencia de los eventos está relacionado con la estructura del documento:

Comienzo del Documento XML	Caracteres:
Principio Elemento: listadealumnos	Principio Elemento: alumno
Caracteres:	Caracteres:
Principio Elemento: alumno	Principio Elemento: nombre
Caracteres:	Caracteres: Maria
Principio Elemento: nombre	Fin Elemento: nombre
Caracteres: Juan	Caracteres:
Fin Elemento: nombre	Principio Elemento: edad
Caracteres:	Caracteres: 20
Principio Elemento: edad	Fin Elemento: edad
Caracteres: 19	Caracteres:
Fin Elemento: edad	Fin Elemento: alumno
Caracteres:	Caracteres:

4 Programación funcional en ficheros

A partir de Java 11 se han introducido unos cuantos cambios en las clases `File`, `Files` y `FileFilter` de la librería `java.io` donde encontramos la API de gestión de ficheros en Java. Los cambios en versión 8 y 11 no tienen gran impacto si los comparamos con otras APIs de java, sólo unos cuantos métodos se han visto afectados, pero en cualquier caso nos proporciona cierto manejo de ficheros como `Stream` e interfaces funcionales. Es lo que básicamente vamos a ver en este tema.

Primero intentaremos entender que es este concepto de APIs java, veremos las APIs más utilizadas de las librerías Java que os harán falta a lo largo de este ciclo formativo.

Posteriormente, nuestra tarea será analizar, describir y utilizar como podemos utilizar java funcional y los patrones de diseño para optimizar el uso de ficheros en nuestros programas, mejorando y refactorizando nuestro código y mejorando nuestro performance. Sin más dilación, nos adentraremos en estos cambios en la API `java.io`.

5 La API `java.nio.file`. Actividad de ampliación

Esta API incluye las clases de plataforma Java utilizadas para E/S básicas. Primero se centra en los *Streams de E/S*, un concepto potente que simplifica enormemente las operaciones de E/S. La lección también examina la serialización, que permite a un programa escribir objetos enteros en secuencias de bytes y leerlos de nuevo. A continuación, la lección examina la E/S de archivos y las operaciones del sistema de archivos, incluyendo las operaciones ac.

Nota: No confundir `IOStream`, `InputStream` y `OutputStream`, de la API de entrada salida java con la API `Stream` de java 8. No están directamente relacionadas.

La mayoría de las clases cubiertas en la sección `Stream` de E/S (`IOStream`) se encuentran en el paquete de `java.io`. La mayoría de las clases cubiertas en la sección E/S de archivo se encuentran en el paquete `java.nio.file`. Este último es el que veremos en los apuntes en profundidad.

I/O Stream incluye:

- **Byte Streams:** maneja E/S de datos binarios sin procesar.
- **Character Streams:** controlan E/S de datos de caracteres, manejando automáticamente la traducción hacia y desde el juego de caracteres local.
- **Buffered Streams:** optimizar la entrada y la salida reduciendo el número de llamadas a la API nativa.
- **Scanning and Formatting:** permite que un programa lea y escriba texto formateado.
- **I/O de línea de comandos:** describe los Standard Streams y las clases y objetos de Consola.
- **Data Streams:** controlar la E/S binaria del tipo de datos primitivo y los valores String.
- **Object Streams** manejar E/S binaria de objetos.

Todo el manejo básico de ficheros y Streams de E/S lo encontrareis en el tema del aula virtual. Vamos a centrarnos en el uso **java funcional de la librería y java.nio**.

5.1 Package *java.nio.file*

La API **java.nio** nos ofrece algunas funcionalidades con ficheros usando **java funcional** que vamos a intentar relatar en este apartado, a partir de la versión 11 de java.

Define interfaces y clases para que la máquina virtual Java acceda a archivos, atributos de archivo y sistemas de archivos.

El paquete **java.nio.file** define las clases para acceder a archivos y sistemas de archivos. La API para acceder a los atributos del sistema de archivos y archivos se define en el paquete `java.nio.file.attribute`. El paquete `java.nio.file.spi` es utilizado por los implementadores del proveedor de servicios que desean ampliar el proveedor predeterminado de la plataforma o para construir otras implementaciones de proveedor.

5.2 La clase *Files*

Vamos a analizar algunos de los métodos y clases que podemos utilizar con colecciones y la API **Stream**. También aprenderemos a manejar la nueva utilidad de Ficheros **Files**, de **java.nio** disponible desde **Java 7**. Empezamos con **Path** y **DirectoryStream** que nos servirá para listar las entradas de un directorio, ficheros y directorios, dentro de un **Path**.

5.2.1 Path

Un Objeto que se puede utilizar para localizar un archivo o directorio en un sistema de archivos. Normalmente representará una ruta de acceso de archivo dependiente del sistema.

Algunos métodos útiles son

<code>boolean endsWith(Path other)</code>	Comprueba si el path acaba con el path pasado como parametro
<code>boolean equals(Object other)</code>	Comprueba si los dos paths son iguales
<code>Path getFileName()</code>	Devuelve el nombre del archivo o directorio denotado por esta ruta de acceso como un objeto path
<code>FileSystem getFileSystem()</code>	Devuelve el sistema de archivos que creó este objeto.
<code>Path getName(int index)</code>	Devuelve un elemento name de esta ruta de acceso como un objeto path a partir del index
<code>int getNameCount()</code>	Devuelve el número de elementos name en la ruta de acceso.
<code>Path getParent()</code>	<i>Devuelve el path padre</i> , or null si no tiene
<code>Path getRoot()</code>	Devuelve el root del path as a Path object, o null si no tiene
<code>boolean isAbsolute()</code>	Tells whether or not this path is absolute.
<code>default Iterator<Path> iterator() Path normalize()</code>	Devuelve un iterator a los names del path Normalize() elimina elementos redundantes
<code>static Path of (String first, String... more)</code>	Devuelve un Path convirtiendo path en cadena, or una secuencia de cadenas que se unen para formar el path
<code>static Path of(URI uri)</code>	Convierte la URI en un path

Creando un path

La clase Path nos permite crear un path a partir de un String o Uri

Con un solo parámetro:

```
Path.of("C:\\Program Files\\Java\\ jre1.8.0_251\\lib") ;
```

Con varios parámetros:

```
Path.of("C:\\Program Files\\Java","jre1.8.0_251", "lib");
```

Podemos usar también un objeto URI, como veis con URI hay que indicar que es un fichero delante. La clase URI se usará en segundo.

```
Path.of("file:///C:/ficheros");
```

Podeis ver el uso de estos métodos en EjemploPath.java

```
import java.nio.file.Path;

public class EjemploPath {

    public static void main(String[] args) {

        Path path= Path.of("C:\\\\Program Files\\\\Java","jre1.8.0_251", "lib");

        path.forEach(System.out::println);

        System.out.println("Root: " + path.getRoot());

        System.out.println("Parent: " + path.getParent());

        System.out.println("Normalize: " + path.normalize());

        System.out.println("Name index 2: " + path.getName(2));

        System.out.println("Name Count: " + path.getNameCount());
    }

}
```

5.2.2 Directory Stream

```
public interface DirectoryStream<T>
extends Closeable, Iterable<T>
```

Nos permite crear un objeto que implementa Iterator para iterar sobre las entradas de un directorio. Una DirectoryStream permite iterar sobre las entradas de un directorio, que pueden ser ficheros u otros directorios.

Iterator en las nuevas versiones de java nos permite recorrer un iterador con un forEach usando consumers. Tenemos tres métodos estáticos en la clase Files para conseguir un objeto de tipo DirectoryStream.

<code>static DirectoryStream<Path> newDirectoryStream(Path dir)</code>	Abrimos un directorio para iterar a partir de un Path
<code>static DirectoryStream<Path> newDirectoryStream (Path dir, String glob)</code>	Abrimos un directorio a partir de un Path, para iterar sobre el. Glob, es un patrón de búsqueda de sistema operativo Basado en * y ?. Por ejemplo *.txt es el patrón para todos los ficheros De tipo txt
<code>static DirectoryStream<Path> newDirectoryStream (Path dir, DirectoryStream.Filter<? super Path> filter</code>	Abrimos un directorio a partir de un Path, para iterar sobre el. Filter, es un interfaz Predicate que filtra por la condición dada por el tipo

Vamos a ver la creación de `DirectoryStream` con filtros en el siguiente ejemplo, `EjemploDirectoryStream.java`. El primer ejemplo sin filtro y recorreremos las entradas de directorio con un `forEach` y un `Consumer`.

```
directorio = Files.newDirectoryStream(path);
directorio.forEach((entry)-> System.out.println(entry));
```

En el segundo caso usamos un filtro glob para quedarnos sólo con las entradas de directorio con extensión `.txt`.

```
directorio2 = Files.newDirectoryStream(path2, "*.txt");
```

```
directorio2.forEach((entry)-> System.out.println(entry));
```

En el tercer caso usamos un **Filter**. Lo implementamos con una expresión lambda. El filtro se quedamos con las entradas cuyo tamaño sea mayor de 2000 bytes (`Files.size()` devuelve el tamaño en bytes de la entrada) y cuyo path comience por `C:\`. (Ponemos la doble barra porque `\` es carácter reservado, acordaos).

```
directorio3 = Files
    .newDirectoryStream(path2,
        (pathparam)-> Files.size(pathparam)>2000
        &&
        pathparam.startsWith("C:\\")
    );
```

Para probar el ejemplo aseguraos que vuestra JRE coincide con la indicada `jre1.8.0_251`, sino cambiad ese directorio en el código para probar el ejemplo.

Debéis tener esta carpeta o similar en vuestro ordenador ya que tenéis la JVM instalada

`C:\Program Files\Java\jre1.8.0_251`

EjemploDirectoryStream.java

```
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;

public class EjemploDirectoryStream {

    public static void main(String[] args) {

        Path path = Path.of("c:\\Program Files");

        System.out.println("Sin filtro");

        DirectoryStream<Path> directorio=null;
```



```

        try {
            directorio = Files.newDirectoryStream(path);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        directorio.forEach((entry)-> System.out.println(entry));

        System.out.println("Con filtro glob ficheros .txt");

        Path path2= Path.of("C:\\Program Files\\Java","jre1.8.0_251",
"lib");

        DirectoryStream<Path> directorio2=null;
        try {
            directorio2
Files.newDirectoryStream(path2,"*.txt");
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        directorio2.forEach((entry)-> System.out.println(entry));

        System.out.println("Con filtro filter ficheros jar");

        DirectoryStream<Path> directorio3=null;
        try {
            directorio3 = Files
                .newDirectoryStream(path2,
(pathparam)-> Files.size(pathparam)>2000
                &&
pathparam.startsWith("C:\\")
        );
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        directorio3.forEach((entry)-> System.out.println(entry));

    }
}

```

Files tiene multiples métodos y propiedades. Vamos a desgranarlos en los siguientes apartados. Tenéis en cualquier caso la documentación oficial en:

[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html#newDirectoryStream\(java.nio.file.Path\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html#newDirectoryStream(java.nio.file.Path))

Nos permite **crear, copiar, borrar y modificar ficheros, directorios y ficheros temporales**, leer y escribir en ficheros de manera rápida y un largo etc.

5.2.3 Opciones para crear ficheros en con la utilidad Files de Java.

En la siguiente tabla ofrecemos métodos para copiar, borrar ficheros, directorios y ficheros temporales con Files.

Descripción	Descripción
static long copy (Path source, OutputStream out)	Copia todos los bytes de fichero a OutputStream
static Path copy (Path source, Path target , CopyOption ... options)	Copia un fichero origen a uno destino
static Path createDirectories (Path dir, FileAttribute <?>... attrs)	Crea un directorio creando primero todos los directorios primarios inexistentes.
static Path createDirectory (Path dir, FileAttribute <?>... attrs)	Crea un directorio
static Path createFile (Path path, FileAttribute <?>... attrs)	Crea un nuevo fichero si no existe
static Path createTempDirectory (String prefix, FileAttribute <?>... attrs)	Crea un nuevo directorio en el directorio de archivos temporales predeterminado, utilizando el prefijo especificado para generar su nombre.
static Path createTempDirectory (Path dir, String prefix, FileAttribute <?>... a ttrs)	Crea un nuevo directorio en el directorio especificado, utilizando el prefijo especificado para generar su nombre.
static Path createTempFile (String prefix, String suffix, FileAttribute <?> ... attrs)	Crea un archivo vacío en el directorio de archivos temporales predeterminado, utilizando el prefijo y sufijo dados para generar su nombre.
static Path createTempFile (Path dir, String prefix, String suffix, FileAt tribute <?>... attrs)	Crea un nuevo archivo vacío en el directorio especificado, utilizando las cadenas de prefijo y sufijo dadas para generar su nombre.
static void delete (Path path)	Borra un fichero
static Boolean deleteIfExists (Path path)	Borra el fichero si existe

Nota: todos los parámetros con el operador ... pueden ir vacíos. Una característica del atributo ... es que nos permite pasar de cero a n parámetros.

Enum StandardOpenOption

- [java.lang.Object](#)

- [java.lang.Enum<StandardOpenOption>](#)
 - [java.nio.file.StandardOpenOption](#)
- **All Implemented Interfaces:**
[Serializable](#), [Comparable<StandardOpenOption>](#), [OpenOption](#)

Constante Enum	Descripción
ATOMIC_MOVE	Mueve el archivo como una operación atómica del sistema de archivos.
COPY_ATTRIBUTES	Copia atributos en el nuevo archivo.
REPLACE_EXISTING	Reemplaza un archivo existente si existe.

Enum StandardOpenOption

- [java.lang.Object](#)
 - [java.lang.Enum<StandardOpenOption>](#)
 - [java.nio.file.StandardOpenOption](#)
- **All Implemented Interfaces:**
[Serializable](#), [Comparable<StandardOpenOption>](#), [OpenOption](#)

El enumerado **Options** para crear, abrir, cerrar, borrar, escribir o leer en un fichero con las siguientes opciones

Opción	Descripción
APPEND	Si el archivo se abre para el acceso WRITE, los bytes se escribirán al final del archivo en lugar del principio.
CREATE	Crear un nuevo archivo si no existe.
CREATE_NEW	Crear un nuevo archivo, con errores si el archivo ya existe
DELETE_ON_CLOSE	Eliminar al cerrar.
DSYNC	Requiere que cada actualización del contenido del archivo se escriba sincrónicamente en el dispositivo de almacenamiento subyacente.
READ	Abierto para el acceso de lectura.
SPARSE	Sparse file. Intento de utilizar el fichero de forma más eficiente en su almacenamiento cuando el fichero está casi vacío o tiene muy poco contenido
SYNC	Requiere que cada actualización del contenido o metadatos del archivo se escriba de forma sincrónica en el dispositivo de almacenamiento subyacente.
TRUNCATE_EXISTING	Si el archivo ya existe y se abre para el acceso WRITE, su longitud se trunca a 0.

WRITE	Abierto para el acceso de escritura.
-------	--------------------------------------

5.2.4 Creando ficheros y directorios. Ejemplo

Vamos a ver un ejemplo de como crear directorios y ficheros usando estas funcionalidades. Explicaremos el código paso a paso:

Definimos los path donde vamos a colocar nuestro directorio, fichero, y directorios y ficheros temporales

```
Path directorio = Path.of("C:\\pruebasficheros");  
Path fichero = Path.of("C:\\pruebasficheros\\Fichero.txt");  
Path directoriotemporal =  
Path.of("C:\\pruebasficheros\\tmp");
```

Si no existe creo el directorio definido en mi Path directorio.

```
if (!Files.isDirectory(directorio))  
    Files.createDirectory(directorio);
```

Si no existe creo el fichero definido en mi variable Path fichero.

```
if (!Files.exists(fichero))  
    Files.createFile(fichero);
```

Creamos ahora el directorio donde van a ir mis ficheros y directorios temporales

```
if (!Files.exists(directoriotemporal)) {  
    Files.createDirectory(directoriotemporal);  
}
```

Creamos un directorio temporal sobre la carpeta para directorios temporales y un fichero. Observar cuando se crean los directorios y ficheros que añade un número que esta basado en Random y la fecha actual la fecha y hora en milisegundos.

```
Files.createTempDirectory(directoriotemporal, "tmpMio");
```

```
Files.createTempFile(directoriotemporal, "tmp", "filemyapp");
```

partir Vista

Este equipo > Windows (C:) > pruebasficheros > tmp

Nombre	Fecha de modificación
tmpMio10801311670352339985	21/03/2021 20:16
tmp1618079893284812281filemyapp	21/03/2021 20:16

Para finalizar mostramos la entrada de mi directorio base.

```
directorio.forEach((entry)-> System.out.println(entry));
```

Finalmente, con el método `walk`, que estudiaremos posteriormente, generamos un `Stream` para recorrer todo el árbol de directorios desde nuestro directorio base `c:\pruebasficheros\`.

```
Stream<Path> miStream = Files.walk(directorio);
miStream.forEach((entry)->
System.out.println(entry));
```

El resultado de la ejecución será el siguiente:

```
Viendo las entradas de mi directorio base c:/pruebasficheros
pruebasficheros
Viendo las entradas del arbol de directorios cuyo raiz es mi directorio base
c:/pruebasficheros:
C:\pruebasficheros
C:\pruebasficheros\Fichero.txt
C:\pruebasficheros\tmp
C:\pruebasficheros\tmp\tmp1618079893284812281filemyapp
C:\pruebasficheros\tmp\tmp2735514376671543879filemyapp
C:\pruebasficheros\tmp\tmpMio10801311670352339985
C:\pruebasficheros\tmp\tmpMio1084233957266289559
```

EjemplosFicherosNIO.java

```
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.nio.file.attribute.FileAttribute;
import java.nio.file.attribute.PosixFilePermission;
import java.nio.file.attribute.PosixFilePermissions;
import java.util.Set;
import java.util.stream.Stream;
```

```

public class EjemplosFicherosNIO {

    public static void main(String[] args) {

        try {
            Path directorio = Path.of("C:\\pruebasficheros");
            Path fichero = Path.of("C:\\pruebasficheros\\Fichero.txt");
            Path directoriotemporal =
Path.of("C:\\pruebasficheros\\tmp");

            //Estos atributos o permisos sólo estan disponibles en Linux
            // los indico pero
            // no los uso
            Set<PosixFilePermission> perms =
PosixFilePermissions.fromString("rwxr-x--");
            FileAttribute<Set<PosixFilePermission>> atributo =
PosixFilePermissions.asFileAttribute(perms);

            //Estos atributos o permisos sólo estan disponibles en
Linux //los indico pero
            // no los uso
            Set<PosixFilePermission> permstmp =
PosixFilePermissions.fromString("rwxrwx---");
            FileAttribute<Set<PosixFilePermission>> atributotmp =
PosixFilePermissions.asFileAttribute(perms);

            if (!Files.isDirectory(directorio))
                Files.createDirectory(directorio);

            if (!Files.exists(fichero))
                Files.createFile(fichero);

            Files.writeString(fichero, "Escribimos en el fichero\n
veremos más tarde\n en detalle", StandardOpenOption.WRITE);

            if (!Files.exists(directoriotemporal)) {
                Files.createDirectory(directoriotemporal);
            }

            Path pathTmp1 =
Files.createTempDirectory(directoriotemporal, "tmpMio");

            Files.writeString(pathTmp1, " Escribimos fichero tmp \n
creado en:" + fecha+ " milisecs:" + System.currentTimeMillis());

            Files.createTempFile(directoriotemporal,"tmp",
"filemyapp");

```

```

        System.out.println("Viendo las entradas de mi
directorio base c:/pruebaficheros");

        directorio.forEach((entry)->
System.out.println(entry));

        System.out.println("Viendo las entradas del arbol de
directorios cuyo raiz es mi directorio base c:/pruebaficheros:");

        Stream<Path> miStream = Files.walk(directorio);
        miStream.forEach((entry)->
System.out.println(entry));

    } catch (IOException ioe) {

        ioe.printStackTrace();

    }

}
}

```

5.2.5 Atributos en Files

Existen diferentes maneras de establecer o modificar los atributos de ficheros en Java. Vamos a hacerlo para los atributos básicos comunes a todos los sistemas. También lo haremos para Windows. Tenéis en el siguiente enlace información para hacerlo en otros sistemas.

<https://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html>

Atributos básicos de Ficheros

Para la **definición de Atributos** en ficheros, tenemos varios interfaces. El Interfaz base será
Interface BasicFileAttributes

- Los subinterfaces conocidos son:

[DosFileAttributes](#): para atributos en sistemas Windows.

[PosixFileAttributes](#): para atributos en sistemas UNIX.

```
public interface BasicFileAttributes
```

Atributos básicos asociados a un archivo en un sistema de archivos.

Los atributos de archivo básicos son atributos comunes a muchos sistemas de archivos y constan de atributos de archivo obligatorios y opcionales definidos por esta interfaz.

Usage Example:

```
Path file = ...
BasicFileAttributes attrs =
Files.readAttributes(file, BasicFileAttributes.class);
```

En el siguiente ejemplo vemos como usar los atributos básicos para el directorio **PruebaFicheros** y el fichero **Fichero.txt** que hemos creado en los ejemplos anteriores. Como podéis ver en el siguiente ejemplo:

1. Leemos los atributos con `Files.readAttributes` para el directorio **dir** `c:\pruebasficheros\` especificado con un `Path`, y quedan almacenados todos los atributos en un objeto de la clase `BasicFileAttributes`. La clase `BasicFileAttributes.class` es cargada por reflexión.

```
BasicFileAttributes basicAttr = Files.readAttributes(dir,
BasicFileAttributes.class);
```

2. Los atributos básicos son tres y comunes a todos los sistemas de ficheros:
 - a. Fecha de **creación**: `basicAttr.creationTime()`;
 - b. Fecha de **ultimo acceso**: `basicAttr.lastAccessTime()`;
 - c. Fecha de **modificación**: `basicAttr.lastModifiedTime()`;

EjemploBasicoAtributos.java

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.attribute.BasicFileAttributes;
import java.nio.file.attribute.FileTime;

public class EjemploBasicoAtributos {

    public static void main(String[] args) throws IOException {

        System.out.println("Atributos Básicos directorio
```



```

pruebaficheros");
        Path dir = Paths.get("C:\\pruebasficheros\\");
        BasicFileAttributes basicAttr =
Files.readAttributes(dir, BasicFileAttributes.class);
        System.out.println("Fecha de creacion:");
        FileTime creationTime = basicAttr.creationTime();
        System.out.println(creationTime);
        System.out.println("Ultimo acceso:");
        FileTime lastAccessTime =
basicAttr.lastAccessTime();
        System.out.println(lastAccessTime);
        System.out.println("Ultima modificación:");
        FileTime lastModifiedTime =
basicAttr.lastModifiedTime();
        System.out.println(lastModifiedTime);

        System.out.println("Atributos Básicos fichero
Fichero.txt");
        Path file =
Paths.get("C:\\pruebasficheros\\Fichero.txt");

        basicAttr = Files.readAttributes(file,
BasicFileAttributes.class);
        System.out.println("Fecha de creacion:");
        creationTime = basicAttr.creationTime();
        System.out.println(creationTime);
        System.out.println("Ultimo acceso:");
        lastAccessTime = basicAttr.lastAccessTime();
        System.out.println(lastAccessTime);
        System.out.println("Ultima modificación:");
        lastModifiedTime = basicAttr.lastModifiedTime();
        System.out.println(lastModifiedTime);
    }
}

```

Modificando atributos, dosFileAttributeView

Esta clase nos permite recoger los atributos de un fichero ya creado en Windows y modificarlos. Sin entrar a profundizar, vamos a ver como cambiamos los atributos Windows de Fichero.txt en el siguiente ejemplo.

1. Recogemos los atributos del fichero Fichero.txt con el método estático de Files. `Files.getFileAttributeView`. Fijaos que en el segundo parámetro del método debemos indicarle la clase donde recogeremos los atributos. El primer parámetro es el Path del fichero.

```

        DosFileAttributeView fileAttributeView =
Files.getFileAttributeView(file,
        DosFileAttributeView.class,

```

```
LinkOption.NOFOLLOW_LINKS);
```

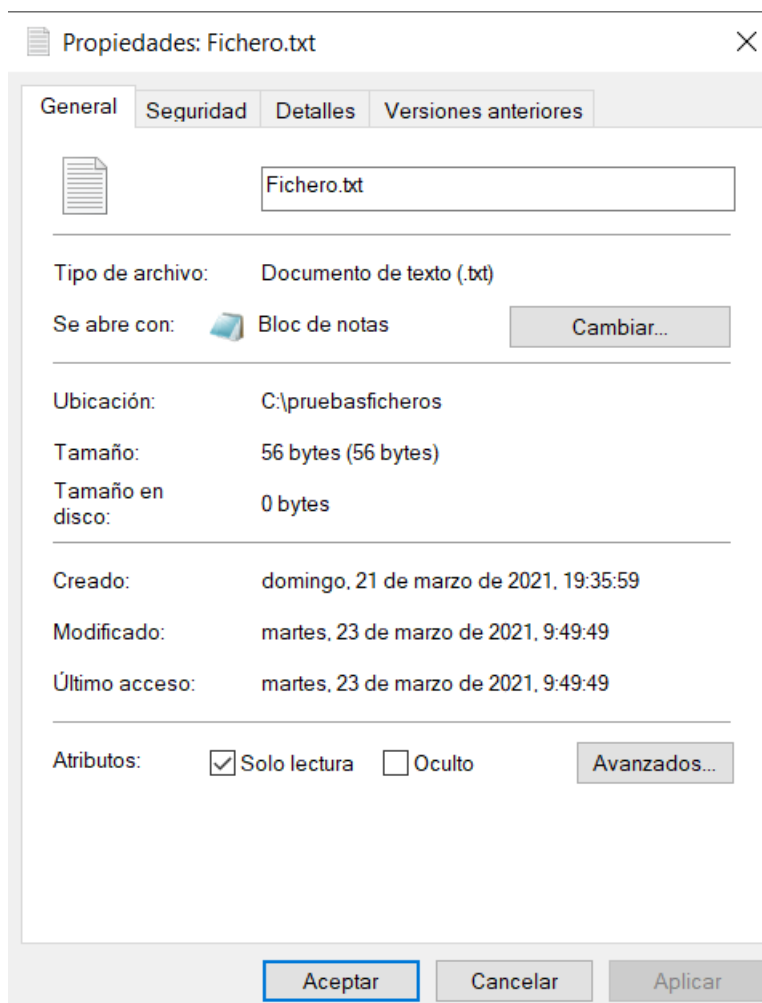
2. Cambiamos los atributos de Fichero.txt, sólo lectura, y archivo oculto.

```
fileAttributeView.setReadOnly(true);  
fileAttributeView.setHidden(false);
```

3. Confirmamos los cambios, cambiando el archivo.

```
fileAttributeView.setArchive(true);
```

Si comprobais los atributos de Fichero.txt tras ejecutar el programa, veréis que los atributos han sido cambiados. La casilla Sólo lectura aparece marcada. Desmarcarla.



EjemploAtributosWindows.java

```
import java.io.IOException;  
import java.nio.file.Files;  
import java.nio.file.LinkOption;  
import java.nio.file.Path;  
import java.nio.file.Paths;  
import java.nio.file.attribute.DosFileAttributeView;
```

```

public class EjemploAtributosWindows {

    public static void main(String[] args) throws IOException {
        System.out.println("Atributos MSDOS fichero Fichero.txt");
        Path file = Paths.get("C:\\pruebasficheros\\Fichero.txt");

        DosFileAttributeView fileAttributeView =
            Files.getFileAttributeView(file,
                                     DosFileAttributeView.class,
                                     LinkOption.NOFOLLOW_LINKS);
        try {
            fileAttributeView.setReadOnly(true);
            fileAttributeView.setHidden(false);

            fileAttributeView.setArchive(true);
        } catch (IOException e) {

            e.printStackTrace();
            //Ignore
        }

        System.out.println("Atributos MSDOS fichero cambiados
Fichero.txt");
    }
}

```

Walk

Una serie de funciones que nos ofrece Files en la versión Java 11 para recorrer directorios y entradas. Los métodos más utilizados son los siguientes.

Nota: Cuando veáis **Lazy evaluation** o **Lazy fill**, en cualquier método significa que la llamada o el procesado se irá realizando poco a poco a demanda, no de una. Si llenamos un Stream de manera Lazy, se iran añadiendo poco a poco los elementos desde fichero o entrada de directorios conforme los vayamos necesitando.

<pre>static <u>Stream</u><<u>Path</u>> <u>walk</u>(<u>Path</u> start, int maxDepth, <u>FileVisitOption</u>.. . options)</pre>	Devuelve un Stream con llenado Lazy para el Path recorriendo por el árbol de archivos cuya raíz en el Path inicial determinado.
<pre>static <u>Stream</u><<u>Path</u>> <u>walk</u> (<u>Path</u> start, <u>FileVisitOption</u>... options)</pre>	Devuelve un Stream con llenado Lazy para el Path recorriendo por el árbol de archivos cuya raíz en el Path inicial determinado. Con FileVisitOption decimos que hacer con los enlaces simbolicos o accesos directos.
<pre>static <u>Path</u> <u>walkFileTree</u></pre>	Recorre un arbol de directorios

(Path start, FileVisitor <? super Path > visitor)	
static Path walkFileTree (Path start, Set < FileVisitOptio n > options, int maxDepth, FileVisitor <? super Path > visitor)	Recorre un arbol de directorios

FileVisitor es un interfaz que se usa para indicar que hacemos con cada fichero y cada directorio cuando recorremos un árbol. Podéis ver la implementación en el siguiente trozo de código, borra el fichero tras visitarlo y borra el directorio tras visitarlo. Haremos un ejemplo y lo explicaremos.

5.3 El interface *FileVisitor*

Para recorrer un árbol de archivos, se debe implementar un `FileVisitor`. Un `FileVisitor` especifica el comportamiento necesario en los puntos clave del proceso de recorrido: cuando se visita un archivo, antes de que se tiene acceso a un directorio, después de que se tiene acceso a un directorio o cuando se produce un error. La interfaz tiene cuatro métodos que corresponden a estas situaciones:

- `preVisitDirectory` - Invocado antes de que se visite las entradas de un directorio.
- `postVisitDirectory` - Invocado después de que se visite todas las entradas de un directorio. Si se encuentra algún error, la excepción específica se pasa al método.
- `visitFile` - Invocado en el archivo que se está visitando.
- `visitFileFailed` - Invocado cuando no se puede acceder al archivo. La excepción específica se pasa al método. Puede elegir si desea producir la

Si no necesita implementar los cuatro métodos `FileVisitor`, en lugar de implementar la interfaz `FileVisitor`, puede ampliar la clase `SimpleFileVisitor`. Esta clase, que implementa la interfaz `FileVisitor`, visita todos los archivos de un árbol o subarbol de directorios y produce un `IOException` cuando se encuentra un error. Se puede ampliar esta clase y utilizar solo los métodos que se necesitan. En el ejemplo posterior, ampliaremos la clase añadiendo gestion de errores, con `public FileVisitResult visitFileFailed`.

```
Path start = ...
```

```

Files.walkFileTree(start, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file,
BasicFileAttributes attrs)
        throws IOException
    {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }
    @Override
    public FileVisitResult postVisitDirectory(Path dir,
IOException e)
        throws IOException
    {
        if (e == null) {
            Files.delete(dir);
            return FileVisitResult.CONTINUE;
        } else {
            // directory iteration failed
            throw e;
        }
    }
});

```

Opciones para devolver en los métodos del FileVisit

- **CONTINUE** – Indica que el **walking o recorrido del archivo debe continuar**. Si devuelve el método `preVisitDirectory` **CONTINUE**, se visita el directorio.
- **TERMINATE** – **Aborta inmediatamente el recorrido** del archivo. No se invocan más métodos de recorrido de archivos después de devolver este valor.
- **SKIP_SUBTREE** – Cuando `preVisitDirectory` devuelve este valor, se **omiten el directorio especificado y sus subdirectorios**. Esta rama se "poda" del árbol de directorios.
- **SKIP_SIBLINGS** – Cuando `preVisitDirectory` devuelve este valor, **el directorio especificado no se visita**, `postVisitDirectory` no se invoca, y **no se visitan más hermanos no visitados**. Si se devuelve en el método `postVisitDirectory` no se visitan más hermanos. Básicamente, no se realiza ninguna acción más sobre el directorio especificado.

Con **FileVisitOptions** podemos indicar entre otras cosas que el recorrido siga los enlaces simbólicos en Linux.

```
FileVisitOption.FOLLOW_LINKS
```

Vamos a ver cómo usar este método con su Interfaz **FileVisitor** en el siguiente ejemplo **EjemploWalk.java**. Para ello lo primero es recoger el **Path** del directorio **pruebasficheros**.

```
Path directorio = Path.of("C:\\pruebasficheros");
```

Empezamos viendo el recorrido sencillo, usando el método `Walk` que nos devuelve un `Stream` de `Paths`, de entradas de directorio. Como veis en la ejecución va recorriendo primero el directorio `root`, `pruebasficheros`, todas sus entradas, y luego los subdirectorios, y sus entradas, en este caso `tmp`.

```
Files.walk(directorio).forEach(System.out::println);
```

```
Recorremos el directorio pruebas normal
C:\pruebasficheros
C:\pruebasficheros\Fichero.txt
C:\pruebasficheros\tmp
C:\pruebasficheros\tmp\tmp17520194810180034790filemyapp
C:\pruebasficheros\tmp\tmpMio18388745732298653726
```

Vamos a ver ahora el recorrido con el método `walkFileTree`. Nos permite un tratamiento específico para ficheros y directorios en el recorrido usando un interfaz `SimpleFileVisitor<Path>` del que creamos una clase anónima. Esta implementación sigue el patrón de diseño `Visitor` que veremos posteriormente.

```
Files.walkFileTree(directorio, new SimpleFileVisitor<Path>() {
```

Implementamos tres métodos del `SimpleFileVisitor`.

1. En la implementación del `FileVisitResult` escribimos el contenido del fichero con el método de `Files` `readAllLines` que veremos más adelante. Es el tratamiento para ficheros que realizamos en nuestro recorrido.

```
Files.readAllLines(file).forEach(System.out::println);
return FileVisitResult.CONTINUE;
```

y continuamos `return FileVisitResult.CONTINUE;`

```
@Override
    public FileVisitResult visitFile(Path file,
BasicFileAttributes attrs)
        throws IOException
    {
        System.out.println("Leemos el fichero pasando por
el FileVisitor " +file);
Files.readAllLines(file).forEach(System.out::println);
return FileVisitResult.CONTINUE;
    }
```

2. En la implementación del `postVisitDirectory`, indicamos que hemos visitado completamente ese directorio, si quedan directorios hermanos o subdirectorios por visitar pasaremos al siguiente, hasta que se termine el árbol de directorios.

- a. Indicamos que el directorio ha sido completamente visitado.

```
System.out.println("Directorio completo visitado "+ dir.getFileName());
```

b. Continuamos con el recorrido.

```
return FileVisitResult.CONTINUE;
```

```
@Override
public FileVisitResult postVisitDirectory(Path dir,
IOException e)
    throws IOException
{
    if (e == null) {
        System.out.println("Directorio completo
visitado "+ dir.getFileName());
        return
FileVisitResult.CONTINUE;
    } else {
        // directory iteration failed
        throw e;
    }
}
```

3. Por ultimo, implementamos el método visitFileFailed que controlará los errores en el recorrido.

- a. Como veis podemos tener dos tipos de errores, que haya un ciclo o que no podamos abrir el archivo:

```
System.err.println("ciclo detectado: " + file);
    } else {
        System.err.format("Imposible abrir el
fichero:" + " %s: %s%n", file, exc);
```

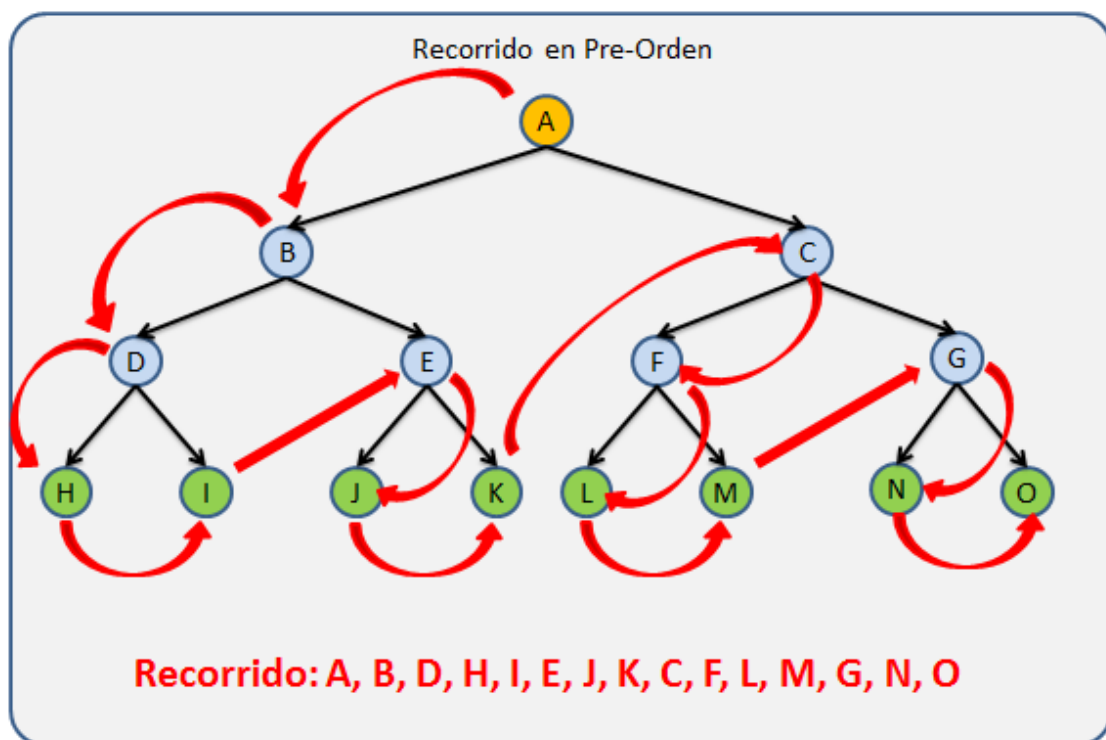
- b. En caso de error, evitamos dejar de recorrer ese subárbol, haciendo SKIP del subárbol, devolviendo SKIP_SUBTREE

```
return FileVisitResult.SKIP_SUBTREE;
```

```
@Override
public FileVisitResult
visitFileFailed(Path file,
IOException exc) {
    if (exc instanceof FileSystemLoopException) {
        System.err.println("ciclo detectado: " +
file);
    } else {
        System.err.format("Imposible abrir el
fichero:" + " %s: %s%n", file, exc);
    }
    return FileVisitResult.SKIP_SUBTREE;
}

});
```

Podeis observar en la ejecución que, vamos leyendo primero todos los subdirectorios y los ficheros dentro de esos subdirectorios. Cuando hemos completado un directorio y todas sus entradas, entre ellas subdirectorios, entonces indicamos que ha sido totalmente visitado. Por esto el walk, el recorrido se hace de los directorios más interiores, hacia los directorios padre. Además, el recorrido primero completa la rama de la izquierda siempre, y los nodos del árbol más interiores, visitando primero los ficheros y después marcando los directorios como visitados. Es lo que se conoce en Teoría de arboles como recorrido en Pre-orden. Se puede observar en la siguiente figura.



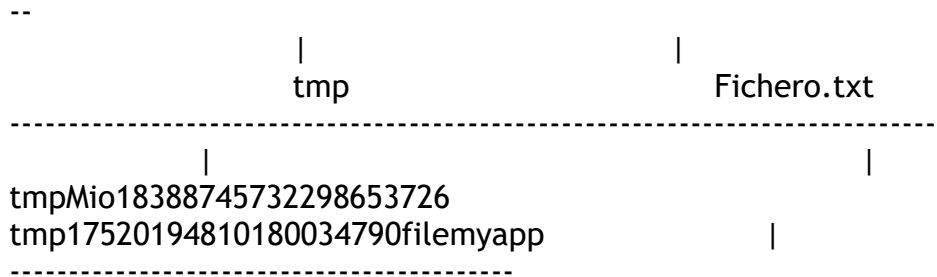
La ejecución nos daría este resultado

```

Recorremos el directorio Con un FileVisitor
Leemos el fichero pasando por el FileVisitor C:\pruebasficheros\Fichero.txt
Escribimos en el fichero
veremos más tarde
en detalle
Leemos el fichero pasando por el FileVisitor
C:\pruebasficheros\tmp\tmp17520194810180034790filemyapp
Escribimos fichero tmp
creado en:23/3/21 9:49 milisecs:1616489389939
Directorio completo visitado tmpMio18388745732298653726
Directorio completo visitado tmp
Directorio completo visitado pruebasficheros
  
```

La estructura de directorios y ficheros es:

Pruebasficheros



En la visita primero lee el **Fichero.txt**, luego el fichero temporal **tmp17520194810180034790filemyapp**, y luego los directorios.

Leemos el fichero pasando por el FileVisitor C:\pruebasficheros\Fichero.txt

Escribimos en el fichero

veremos más tarde

en detalle

Leemos el fichero pasando por el FileVisitor

C:\pruebasficheros\tmp\tmp17520194810180034790filemyapp

Escribimos fichero tmp

creado en:23/3/21 9:49 milisecs:1616489389939

Los directorios:

1. Como el directorio está vacío **tmpMio18388745732298653726** no tiene que seguir recorriendo subdirectorios, lo marca como visitado:
2. Como ya ha visitado el fichero **tmp17520194810180034790filemyapp** y el directorio **tmpMio18388745732298653726**, marca tmp como visitado.
3. Para finalizar como ya ha visitado todos los ficheros y directorios dentro de pruebasficheros, marca pruebasficheros como visitado.

Directorio completo visitado tmpMio18388745732298653726

Directorio completo visitado tmp

Directorio completo visitado pruebasficheros

En resumen, el recorrido es primero ficheros, y luego marca como visitados los subdirectorios más interiores en el árbol de ficheros de izquierda a derecha en el árbol. Finalmente, el directorio principal es marcado como visitado.

EjemploWalk.java

```

import java.io.IOException;
import java.nio.file.FileSystemLoopException;
import java.nio.file.FileVisitResult;
import java.nio.file.Files;
import java.nio.file.LinkOption;
import java.nio.file.Path;
import java.nio.file.SimpleFileVisitor;
import java.nio.file.attribute.BasicFileAttributes;

public class EjemploWalk {

    public static void main(String[] args) {

```

```

        Path directorio = Path.of("C:\\pruebasficheros");

        try {

            System.out.println("Recorremos el directorio pruebas
normal");

            //Recorremos el directorio pruebas normal
            Files.walk(directorio).forEach(System.out::println);

            System.out.println("Recorremos el directorio Con un
FileVisitor");

            Files.walkFileTree(directorio, new
SimpleFileVisitor<Path>() {
                @Override
                public FileVisitResult visitFile(Path file,
BasicFileAttributes attrs)
                    throws IOException
                {
                    System.out.println("Leemos el fichero pasando por
el FileVisitor " +file);
                    Files.readAllLines(file).forEach(System.out::println);
                    return FileVisitResult.CONTINUE;
                }
                @Override
                public FileVisitResult postVisitDirectory(Path dir,
IOException e)
                    throws IOException
                {
                    if (e == null) {
                        System.out.println("Directorio completo
visitado "+ dir.getFileName());
                        return
FileVisitResult.CONTINUE;
                    } else {
                        // directory iteration failed
                        throw e;
                    }
                }

                @Override
                public FileVisitResult
visitFileFailed(Path file,
IOException exc) {
                    if (exc instanceof FileSystemLoopException) {
                        System.err.println("ciclo detectado: " +
file);
                    } else {
                        System.err.format("Imposible abrir el
fichero:" + " %s: %s%n", file, exc);
                    }
                    return FileVisitResult.SKIP_SUBTREE;
                }
            }
        }
    }

```

```

    });

    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}
}

```

Vamos a ofrecer otro ejemplo de Walk muy similar al anterior, pero en este caso en la visita a ficheros vamos a borrar el fichero si es un fichero temporal. Lo podéis ver en EjemploWalk2.java. Hemos sobrescrito en este caso el método visitFile, para que si el fichero es temporal, borre el fichero.

```

        if (file.getFileName().toString().contains("filemyapp")) {
            System.out.println("Fichero temporal
borrado " +file);
            Files.delete(file);
        }

```

```

@Override
    public FileVisitResult visitFile(Path file,
BasicFileAttributes attrs)
        throws IOException
    {
        System.out.println("Borramos el fichero sólo si es
temporal " +file);
        if
(file.getFileName().toString().contains("filemyapp")) {
            System.out.println("Fichero temporal
borrado " +file);
            Files.delete(file);
        }
        return FileVisitResult.CONTINUE;
    }

```

En la ejecución podéis ver que sólo se borra el fichero temporal, aunque visitamos dos ficheros, porque nuestros ficheros temporales acaban con filemyapp.

```
Recorremos el directorio pruebas normal
C:\pruebasficheros
C:\pruebasficheros\Fichero.txt
C:\pruebasficheros\tmp
C:\pruebasficheros\tmp\tmp17520194810180034790filemyapp
C:\pruebasficheros\tmp\tmpMio18388745732298653726
Recorremos el directorio Con un FileVisitor
Borramos el fichero sólo si es temporal C:\pruebasficheros\Fichero.txt
Borramos el fichero sólo si es temporal 
C:\pruebasficheros\tmp\tmp17520194810180034790filemyapp
Fichero temporal borrado
C:\pruebasficheros\tmp\tmp17520194810180034790filemyapp
Directorio completo visitado tmpMio18388745732298653726
Directorio completo visitado tmp
Directorio completo visitado pruebasficheros
```

EjemploWalk2.java

```
import java.io.IOException;
import java.nio.file.FileSystemLoopException;
import java.nio.file.FileVisitResult;
import java.nio.file.Files;
import java.nio.file.LinkOption;
import java.nio.file.Path;
import java.nio.file.SimpleFileVisitor;
import java.nio.file.attribute.BasicFileAttributes;

public class EjemploWalk2 {

    public static void main(String[] args) {

        Path directorio = Path.of("C:\\pruebasficheros");

        try {

            System.out.println("Recorremos el directorio pruebas
normal");

            //Recorremos el directorio pruebas normal
            Files.walk(directorio).forEach(System.out::println);

            System.out.println("Recorremos el directorio Con un
FileVisitor");

            Files.walkFileTree(directorio, new
SimpleFileVisitor<Path>() {
                @Override
                public FileVisitResult visitFile(Path file,
BasicFileAttributes attrs)
                    throws IOException
                {
                    System.out.println("Borramos el fichero sólo si es
temporal " +file);
                    if
(file.getFileName().toString().contains("filemyapp")) {
```

```

        System.out.println("Fichero temporal
borrado " +file);
        Files.delete(file);
    }
    return FileVisitResult.CONTINUE;
}
@Override
public FileVisitResult postVisitDirectory(Path dir,
IOException e)
    throws IOException
    {
        if (e == null) {
            System.out.println("Directorio completo
visitado "+ dir.getFileName());
            return
FileVisitResult.CONTINUE;
        } else {
            // directory iteration failed
            throw e;
        }
    }

@Override
public FileVisitResult
visitFileFailed(Path file,
IOException exc) {
    if (exc instanceof FileSystemLoopException) {
        System.err.println("ciclo detectado: " +
file);
    } else {
        System.err.format("Imposible abrir el
fichero:" + " %s: %s%n", file, exc);
    }
    return FileVisitResult.SKIP_SUBTREE;
}

});

} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

}
}

```

5.4 Ejemplos de lectura y escritura de ficheros con la API Files y Java 8

Para la lectura y escritura en ficheros tenemos los siguientes métodos en la API Files. Tenemos dos tipos de métodos, los que escriben caracteres, los que escriben bytes, y los que escriben objetos en ficheros como bytes. Vamos a ver un ejemplo de cada uno de ellos.

Opciones para leer y escribir en Ficheros con la utilidad Files de Java.

En esta tabla presentamos los métodos más comunes para leer y escribir en ficheros en Files.

<code>static Stream<String> lines(Path path)</code>	Lea todas las líneas de un archivo como una Stream.
<code>static Stream<String> lines(Path path, Charset cs)</code>	Lea todas las líneas de un archivo como un Stream.
<code>static String readString(Path path)</code>	Lee todo el contenido de un archivo en una cadena, descodificando de bytes a caracteres mediante el conjunto de caracteres UTF-8.
<code>static String readString (Path path, Charset cs)</code>	Lee todos los caracteres de un archivo en una cadena, descodificando de bytes a caracteres mediante el conjunto de caracteres especificado.
<code>static byte[] readAllBytes(Path path)</code>	Lee todos los bytes de un archivo.
<code>static List<String> readAllLines(Path path)</code>	Lea todas las líneas de un archivo.
<code>static List<String> readAllLines (Path path, Charset cs)</code>	Lea todas las líneas de un archivo.
<code>static BufferedReader newBufferedReader(Path path)</code>	Abre un archivo para su lectura, devolviendo un <code>BufferedReader</code> para leer texto del archivo de una manera eficaz.
<code>static BufferedReader newBufferedReader (Path path, Charset cs)</code>	Abre un archivo para su lectura, devolviendo un <code>BufferedReader</code> que se puede usar para leer texto del archivo de una manera eficaz.
<code>static BufferedWriter newBufferedWriter (Path path, Charset cs, OpenOption ion... options)</code>	Abre o crea un archivo para escribir, devolviendo un <code>BufferedWriter</code> que se puede usar para escribir texto en el archivo de una manera eficaz.
<code>static BufferedWriter newBufferedWriter (Path path, OpenOption... options)</code>	Abre o crea un archivo para escribir, devolviendo un <code>BufferedWriter</code> para escribir texto en el archivo de una manera eficaz.
<code>static Path write(Path path, byte[] bytes, OpenOption... options)</code>	Escribe bytes en un archivo.
<code>static Path</code>	Escriba líneas de texto en un archivo.

<code>write(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options)</code>	
<code>static Path write(Path path, Iterable<? extends CharSequence> lines, OpenOption... options)</code>	Escriba líneas de texto en un archivo.
<code>static Path writeString(Path path, CharSequence csq, Charset cs, OpenOption... options)</code>	Escriba un <code>CharSequence</code> en un archivo.
<code>static Path writeString(Path path, CharSequence csq, OpenOption... options)</code>	Escribir un <code>CharSequence</code> en un archivo.

Vamos a usarlos, explicarlos y desarrollarlos con ejemplos.

Ejemplo de fichero de texto. **Actividad guiada lectura y escritura de ficheros de texto**

En primer lugar, vamos a **generar ficheros usando las funciones para ficheros de texto y cadenas**. Como siempre lo vamos a explicar a través de un ejemplo que es más gráfico.

Usamos el **interfaz `CharSequence` para escribir caracteres**. Un `CharSequence` es una **secuencia legible de valores char**. Esta interfaz proporciona acceso uniforme y de solo lectura a muchos tipos diferentes de **secuencias char**. Un valor `char` representa un carácter en el plano *multilingüe básico (BMP)* o un suplente, con `String`. Podemos sustituirlo en muchos casos por una cadena, no os preocupéis.

En el ejemplo tenemos una Lista con cadenas:

```
List<String> listaCadenas = Arrays.asList("En este fichero", "vamos a escribir", "unas cuantas cadenas", "de una lista de cadenas", "intentando probar ", "los métodos que nos ofrece", "la clase Files", " Y SU API");
```

También tenemos dos ficheros de texto que vamos a crear y trabajar con ellos.

```
Path file = Path.of("C:\\pruebasficheros", "ficherocadenas.txt");
Path file2 = Path.of("C:\\pruebasficheros", "ficherocadenas2.txt");
```

Vamos a ver como usamos los métodos para añadir una cadena de texto.

1. **Escribiendo una cadena:** El `Charset.forName("UTF-8")` nos devuelve el `Charset` con el que queremos que se codifique el fichero. Fijaos que en este caso pasamos una cadena como parámetro. Con `StandardOpenOption.CREATE` le indicamos que cree el fichero. Con la opción `APPEND` añadimos al fichero ya creado el texto que deseamos.

```
Files.writeString(file, "Escribimos la primera cadena que no esta en la lista", Charset.forName("UTF-8"), StandardOpenOption.CREATE);
```

```
Files.writeString(file, "Escribimos la segunda cadena, " + "le pasaremos un iterable a Files para escribir la lista de Strings", Charset.forName("UTF-8"), StandardOpenOption.APPEND);
```

2. **Escribiendo un iterable:** en este segundo caso usamos la función `write` para escribir en el fichero, un iterable de cadenas o `CharSequence`. Como ya sabéis las colecciones implementan `Iterable`. Escribirá todas las cadenas de la `listaCadenas` línea a línea. Una línea por elemento de la colección.

```
Files.write(file, listaCadenas);
```

Podéis comprobarlo en la ejecución. En amarillo lo escrito por `writeString`, en azul lo escrito por `write`, la lista de cadenas.

```
Escribimos en el fichero ficheroCadenas.txt
Leemos el fichero en una sola cadena
Fichero en una sola cadena En este fichero
vamos a escribir
unas cuantas cadenas
de una lista de cadenas
intentando probar
los métodos que nos ofrece
la clase Files
Y SU API
```

3. **Con `readString`** leemos todo el fichero en una sola cadena de texto.

```
String cadenaCompleta = Files.readString(file, Charset.forName("UTF-8"));
```

4. **Con el método `lines`** leemos línea a línea del fichero de texto, teniendo en cuenta que el método devuelve un `Stream` de cadenas, `Stream<String>`.

```
Files.lines(file, Charset.forName("UTF-8")).forEach(System.out::println);
```


5. El método `readAllLines` devuelve una lista de cadenas, un `List<String>`, podemos obtener un `Stream` de tipo `String` a partir de él con `.stream()`.

```
Files.readAllLines(file,Charset.forName("UTF-8")).stream().forEach((linea)->
System.out.print(linea+"|"));
```

6. Por ultimo creamos un nuevo fichero `ficheroCadenas2.txt`, usando como entrada los datos escritos en `ficheroCadenas.txt`, y usando el método `lines` para leer `ficheroCadenas.txt`, y `write` para escribir el fichero `ficheroCadenas2.txt`. Para que veáis que es un iterable el parámetro de entrada de `write` lo hacemos en dos pasos.

```
Iterable<CharSequence> it = Files.lines(file,Charset.forName("UTF-8"))
    .map(line-> (CharSequence)
line).collect(Collectors.toList());

Files.write(file2,
    it , StandardOpenOption.CREATE);
```

Pero podemos hacerlo en un solo paso, pasando directamente el `readAllLines`, que devuelve un iterable de tipo `List<String>`.

```
Files.write(file2,
    Files.readAllLines(file,Charset.forName("UTF-8")) ,
StandardOpenOption.CREATE);
```

EjemploLecturaEscrituraCadenas.java

```
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class EjemploLecturaEscrituraCadenas {

    public static void main(String[] args) throws IOException {

        List<String> listaCadenas = Arrays.asList("En este fichero",
"vamos a escribir",
, "unas cuantas cadenas", "de una lista de
cadenas",
"intentando probar ", "los métodos que nos
ofrece",
"la clase Files", " Y SU API");

        Path file = Path.of("C:\\pruebasficheros",
"ficheroCadenas.txt");

        Path file2 = Path.of("C:\\pruebasficheros",
```

```

"ficherocadenas2.txt");

        System.out.println("Escribimos en el fichero
ficherocadenas.txt");

        Files.writeString(file, "Escribimos la primera cadena que no
esta en la lista", Charset.forName("UTF-8"),
StandardOpenOption.CREATE);

        Files.writeString(file, "Escribimos la segunda cadena, "
+ "le pasaremos un iterable a Files para
escribir la lista de Strings", Charset.forName("UTF-8"),
StandardOpenOption.APPEND);

        Files.write(file, listaCadenas,
StandardOpenOption.APPEND);

        System.out.println("Leemos el fichero en una sólo cadena");

        String cadenaCompleta =
Files.readString(file,Charset.forName("UTF-8"));

        System.out.println("Fichero en una sólo cadena " +
cadenaCompleta);

        System.out.println("Leemos el fichero con un Stream");

        Files.lines(file,Charset.forName("UTF-
8")).forEach(System.out::println);

        System.out.println("Leemos el fichero con una lista, lo
colocamos en una sola linea con separador |");

        Files.readAllLines(file,Charset.forName("UTF-
8")).stream().forEach((linea)-> System.out.print(linea+"|"));

        System.out.println("escribimos en un nuevo fichero con una
lista");

        Iterable<CharSequence> it =
Files.lines(file,Charset.forName("UTF-8"))
.map(line-> (CharSequence)
line).collect(Collectors.toList());

        Files.write(file2,
it , StandardOpenOption.CREATE);

        Files.write(file2,
Files.readAllLines(file,Charset.forName("UTF-8")) ,
StandardOpenOption.CREATE);

```

```

        System.out.println("Leemos ficherocadenas2");

        Files.lines(file,Charset.forName("UTF-
8")).forEach(System.out::println);

    }

}

```

5.4.1 Actividad independiente lectura y escritura de ficheros de texto

Se pedirá a los alumnos:

1. Que creen un fichero de texto fichText.txt
2. Que lean de la consola y vayan escribiendo en el fichero hasta recibir la letra "P"
3. Que abran el fichero y lo muestren por pantalla

Lectura y escritura de Bytes en ficheros usando la API Files.

Actividad guiada lectura y escritura de Bytes.

Vamos a repasar con otro ejemplo métodos usados para escribir y leer bytes en ficheros. Realizamos un sencillo ejemplo, que nos vendrá bien para el curso que viene. En el ejemplo vamos a escribir una cadena transformada en bytes a fichero. Luego vamos a recuperar los bytes de fichero y transformarlo nuevamente en cadena.

Tenemos un Path para el fichero binario `binario.dat`, y dos cadenas, `cadenaBytes` para escribirla en el fichero, `cadenaRecupera` para transformar los bytes recuperados del fichero a cadena.

```

Path fileBytes = Path.of("C:\\pruebasficheros", "binario.dat");
String cadenaBytes = "Vamos a guardar en el fichero la
cadena como Bytes";
String cadenaRecupera="";

```

1. Con el write escribimos los bytes de la cadena en el fichero tras transformar la cadena a bytes con `cadenaBytes.getBytes()`.

```
Files.write(fileBytes, cadenaBytes.getBytes(), StandardOpenOption.CREATE);
```

2. Leemos los bytes escritos de fichero con `readBytes` y los colocamos en un array de bytes.

```
byte[] bytesFich= Files.readAllBytes(fileBytes);
```

3. Transformamos los bytes en cadena llamando al constructor de String al que le pasamos como parámetro los bytes leídos de fichero.

```
cadenaRecupera = new String(bytesFich);
```

La ejecución del programa sería la siguiente:

Guardamos un fichero una cadena como Byte

Recuperamos la cadena del fichero

Bytes recuperados[B@4769b07b

Transformamos a cadena los bytes con resultado: Vamos a guardar en el fichero la cadena como Bytes

EjemploLecturaBytesFiles.java

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import modelo.*;

public class EjemploLecturaBytesFiles {

    public static void main(String[] args) throws IOException {

        Path fileBytes = Path.of("C:\\pruebasficheros",
"binario.dat");
        String cadenaBytes = "Vamos a guardar en el fichero la
cadena como Bytes";
        String cadenaRecupera="";

        System.out.println("Guardamos un fichero una cadena como
Byte");

        Files.write(fileBytes, cadenaBytes.getBytes(),
StandardOpenOption.CREATE);

        System.out.println("Recuperamos la cadena del fichero");

        byte[] bytesFich= Files.readAllBytes(fileBytes);

        System.out.println("Bytes recuperados" + bytesFich);

        cadenaRecupera = new String(bytesFich);
        System.out.println("Transformamos a cadena los bytes con
resultado: " + cadenaRecupera);

    }

}
```

5.4.2 Actividad independiente de lectura y escritura de ficheros.

Se pedirá a los alumnos:

1. Que creen un array de 1024 bytes arrayBytes.
2. Que creen un fichero binario fichero.bin
3. Que escriban los bytes en el fichero binario.
4. Que lean los bytes del fichero binario y comparen para ver si la lectura ha sido correcta

5.5 Almacenando objetos como bytes usando Files.

Por último, vamos a ver como almacenar objetos como bytes en ficheros usando Files. Para ello vamos a usar el modelo de Usuarios del tema anterior, introduciendo una pequeña modificación. Vamos a hacer que Usuario sea Serializable, condición necesaria para poder transformar un objeto en Bytes con un OutputStream y transformar de bytes a objetos con un InputStream. Las clases que vamos a utilizar para realizar esta labor serán ByteArrayOutputStream, ObjectOutputStream para transformar los objetos a bytes y ByteArrayInputStream y ObjectInputStream para transformar de bytes a objetos. El proceso es muy similar a como se realiza en los apuntes base del curso.

Para que un objeto sea serializable basta con que implemente la interfaz Serializable. Como la interfaz Serializable no obliga a implementar métodos, es muy sencillo implementarla, basta con un implements Serializable y nada más. Por ejemplo, la clase Usuario siguiente es Serializable y java sabe perfectamente enviarla o recibirla por red, a través de socket o de rmi. También java sabe escribirla en un fichero o reconstruirla a partir del fichero.

Si dentro de la clase hay atributos que son otras clases, éstos a su vez también deben ser Serializable. Con los tipos de java (String, Integer, etc.) no hay problema porque son Serializables. Si ponemos como atributos nuestras propias clases, éstas a su vez deben implementar Serializable.

Serial Version UID

Cuando pasamos objetos Serializable de un lado a otro tenemos un pequeño problema. Si la clase que queremos pasar es de tipo clase objeto_serializable, lo normal, salvo que usemos Carga dinámica de clases, es que en ambos lados (el que envía y el que recibe la clase o el que escribe en fichero o lee de fichero), tengan su propia copia del fichero objeto_serializable.class. Es posible que en distintas versiones de nuestro programa la clase objeto_serializable cambie, de forma que es posible que un lado tenga una versión más antigua que en el otro lado. Si sucede esto, la

reconstrucción de la clase en el lado que recibe es imposible.

Para evitar este problema, se aconseja que la clase objeto_serializable tenga un atributo privado llamado serialVersionUID.

```
private static final long serialVersionUID = 8799656478674716638L;
```

de forma que el numerito que ponemos al final debe ser distinto para cada versión de compilado que tengamos.

De esta forma, java es capaz de detectar rápidamente que las versiones de objeto_serializable.class en ambos lados son distintas.

5.5.1 Convertir un Serializable a byte[] y viceversa

Podemos convertir cualquier objeto Serializable a un array de bytes y viceversa. Normalmente esto no es necesario que lo hagamos explícitamente en el código para enviar el objeto por un socket o escribirlo en un fichero puesto que contamos con las clases ObjectOutputStream y ObjectInputStream que se encargan de ello. Sin embargo, en ocasiones, por ejemplo, al intentar enviar un objeto por un socket udp o guardar objetos en fichero con la clase Files, sí es necesario hacerlo manualmente.

Para hacer esta conversión, podemos usar este código

De objeto a byte[]

```
ByteArrayOutputStream bs= new ByteArrayOutputStream();
ObjectOutputStream os = new ObjectOutputStream (bs);
os.writeObject(unObjetoSerializable); // this es de tipo DatoUdp
os.close();
byte[] bytes = bs.toByteArray(); // devuelve byte[]
```

De byte[] a objeto

```
ByteArrayInputStream bs= new ByteArrayInputStream(bytes); //
bytes es el byte[]
ObjectInputStream is = new ObjectInputStream(bs);
ClaseSerializable unObjetoSerializable =
(ClaseSerializable)is.readObject();
is.close();
```

Una utilidad de estos dos códigos es el realizar copias "profundas" de un objeto, es decir, se obtiene copia del objeto, copias de los atributos y copias de los atributos de los atributos. Basta convertir el objeto a byte[] y luego reconstruirlo en otra variable.

5.5.2 Lambdas y serializable. Muy importante. Actividad guiada

lambdas serializables

Lo estudiaremos en el ejemplo, para la clase Usuario hemos añadido una lambda que calcula el rating del usuario. Cada lambda que usemos en un objeto que queramos llevar a bytes necesita marcarse como Serializable porque son las lambdas son una clase anónima en si mismas. Por tanto deben ser Serializable, ya sea para ficheros, base de datos o la red o ya sea la red, como veréis el curso que viene.

IMPORTANTE

La manera de Serializar una lambda es castearla de la siguiente manera `(Function<Usuario,Double> &Serializable)`, al Interfaz funcional a castear le añadimos al final `&Serializable`. Si no lo hacéis no os va a funcionar el guardado de bytes en fichero a partir de un objeto que contenga expresiones lambda.

```
private          Function<Usuario,Double>          rating=  
(Function<Usuario,Double>          &Serializable)          (u)->  
(u.getNumConexiones()*0.30 + u.getHorasDeUso()*0.70)/100;
```

Ejemplo de guardar objetos como bytes. Practica guiada de guardar en fichero.

Lo primero es cambiar en nuestro modelo de usuarios la clase Usuario.java y convertirla en Serializable. En amarillo los cambios para hacerla Serializable. En azul alguna mejora que se ha introducido. GeneraUsuarios.java y GeneraCamposAleatorios.java no se ven afectadas por los cambios.

```
public class Usuario implements Comparable<Usuario>, Serializable  
{  
  
    private static final long serialVersionUID = -  
1234664055657290718L;  
    private int id;  
    private String nombre;  
    private String apellidos;  
    private Integer Edad;  
    private Double horasDeUso;  
    private int numConexiones;  
  
    private          Function<Usuario,Double>          rating=  
(Function<Usuario,Double>          & Serializable)          (u)->  
(u.getNumConexiones()*0.30 + u.getHorasDeUso()*0.70)/100;  
  
    public Usuario() {
```

```

    }

    public Usuario(int id, String nombre, String apellidos,
Integer edad, Double horasDeUso, int numConexiones) {

        this.id=id;
        this.nombre = nombre;
        this.apellidos = apellidos;
        Edad = edad;
        this.horasDeUso = horasDeUso;
        this.numConexiones = numConexiones;
    }

    public int getId() {
        // TODO Auto-generated method stub
        return id;
    }
    public String getNombre() {
        return nombre;
    }
    public String getApellidos() {
        return apellidos;
    }
    public Integer getEdad() {
        return Edad;
    }
    public Double getHorasDeUso() {
        return horasDeUso;
    }
    public int getNumConexiones() {
        return numConexiones;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }
    public void setEdad(Integer edad) {
        Edad = edad;
    }
    public void setHorasDeUso(Double horasDeUso) {
        this.horasDeUso = horasDeUso;
    }
    public void setNumConexiones(int numConexiones) {
        this.numConexiones = numConexiones;
    }

    public Double getRating() {

        return rating.apply(this);
    }

```



```

    }

    @Override
    public String toString() {
        return "Usuario [id=" + id + ", nombre=" + nombre +
            ", apellidos=" + apellidos + ", Edad=" + Edad
            + ", horasDeUso=" + horasDeUso + ",
            numConexiones=" + numConexiones + "]\n";
    }

    @Override
    public int compareTo(Usuario o) {
        // TODO Auto-generated method stub

        return
            this.getNumConexiones() > o.getNumConexiones() ? 1 : (this.getNumConexiones() == o.getNumConexiones() ? 0 : -1);
    }

}

```

Y ahora vamos a ver como guardamos los objetos como Bytes en ficheros y como recuperamos esos objetos. Para ello vamos a estudiar el siguiente ejemplo, **EjemploLecturaBytesSerialize.java**.

En el tenemos una lista de usuarios que creamos con 100 usuarios aleatorios y un fichero que vamos a crear usuarios.dat, un fichero binario.

```

Path fileBytes = Path.of("C:\\pruebasficheros", "usuarios.dat");
List<Usuario> listaUsu =
GeneraUsuarios.devuelveUsuariosLista(100);

```

Para escribir la lista la debemos transformar a bytes. Usaremos las clases **ByteArrayOutputStream**, y crear el **ObjectOutputStream** pasandole el **ByteArrayOutputStream** como parámetro. Escribimos el objeto de tipo **List<Usuario>** con **writeObject**. Después de escribir realizamos un **flush** para vaciar el buffer, y cerramos el **ObjectOutputStream** con **close**. Para escribir en el fichero los bytes obtenidos usamos el **método Files.write**. Es importante

cerrar los flujos de entrada y salida cuando hemos terminado de usarlos, pero no antes.

```
ByteArrayOutputStream bos = new ByteArrayOutputStream();

    System.out.println("Escribimos en el fichero");

    ObjectOutputStream out = new ObjectOutputStream(bos);
    out.writeObject(listaUsu);
    out.flush();
    byte[] bytes = bos.toByteArray();

    Files.write(fileBytes, bytes,
StandardOpenOption.CREATE);

    out.close();
```

Para leer realizaremos un proceso parecido pero con `ObjectInputStream`. Primero leemos los bytes de fichero con `Files.readAllBytes`, luego transformamos esos bytes a objeto de tipo `List<Usuario>`. Creamos un `ObjectInputStream` a partir de un `ByteArrayInputStream` al que le hemos pasado como parámetro los bytes leídos de fichero. Finalmente transformamos a objeto con el método `readObject` de `ObjectInputStream`. Finalmente cerramos el `ObjectInputStream` con el método `close`.

```
byte[] bytesRecuperados = Files.readAllBytes(fileBytes);
ByteArrayInputStream bis = new
ByteArrayInputStream(bytesRecuperados);
ObjectInputStream in = new
ObjectInputStream(bis);
List<Usuario> listaUsuRecuperada=null;
try {
    listaUsuRecuperada =(List<Usuario>)
in.readObject();
} catch (ClassNotFoundException | IOException e)
{
    // TODO Auto-generated catch block
    e.printStackTrace();
}

in.close();
```

Montad el ejemplo y probarlo, necesitarías las clases para generar usuarios del tema anterior. En mi caso las he añadido a un paquete llamado `modelo`.

```
import modelo.*;
```

EjemploLecturaBytesSerialize.java

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.util.List;

import modelo.*;

public class EjemploLecturaBytesSerialize {

    public static void main(String[] args) throws IOException {

        Path fileBytes = Path.of("C:\\pruebasficheros",
"usuarios.dat");
        List<Usuario> listaUsu =
GeneraUsuarios.devuelveUsuariosLista(100);

        System.out.println("Creamos la lista de usuarios");
        listaUsu.stream().forEach(System.out::println);

        ByteArrayOutputStream bos = new
ByteArrayOutputStream();

        System.out.println("Escribimos en el fichero");

        ObjectOutputStream out = new
ObjectOutputStream(bos);
        out.writeObject(listaUsu);
        out.flush();
        byte[] bytes = bos.toByteArray();

        Files.write(fileBytes, bytes,
StandardOpenOption.CREATE);

        out.close();

        System.out.println("leemos de fichero los bytes y
transformamos a objeto");

        byte[] bytesRecuperados =
Files.readAllBytes(fileBytes);
```

```

        ByteArrayInputStream bis = new
ByteArrayInputStream(bytesRecuperados);
        ObjectInputStream in = new
ObjectInputStream(bis);
        List<Usuario> listaUsuRecuperada=null;
        try {
            listaUsuRecuperada =(List<Usuario>)
in.readObject();
        } catch (ClassNotFoundException |
IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        in.close();
        System.out.println("Recuperamos la lista de
usuario de fichero");

        listaUsuRecuperada.forEach(System.out::println);
    }
}

```

6 Patrones de diseño en ficheros.

En este tema vamos a ver dos patrones de diseño. El patrón proxy y el patrón visitor aplicados a Ficheros. Se pueden aplicar en especial el patrón proxy a otros ámbitos como base de datos, muy habitual con una versión del patrón de proxy llamada DAO o patrón DAO, Data Access Object. Es una versión de Proxy específica para acceso a base de datos. También veremos el patrón Visitor. Hemos usado en este tema un ejemplo de patrón visitor de la API de Java java.nio, en la implementación del interfaz SimpleFileVisitor.

6.1 Patron Proxy. **Actividad guiada patrón proxy**

Proxy es un patrón de diseño estructural que permite proporcionar un sustituto o puntero y manejador de otro objeto. Un proxy controla el acceso al objeto original, lo que le permite realizar algo antes o después de que la solicitud llegue al objeto original. Muy útil para establecer conexiones con B.D, antes de hacer las peticiones al objeto específico de Base de Datos, cuando nuestra aplicación guarda los datos en base de datos. Util también para crear los ficheros y mantenerlos, cuando queremos guardar los datos de

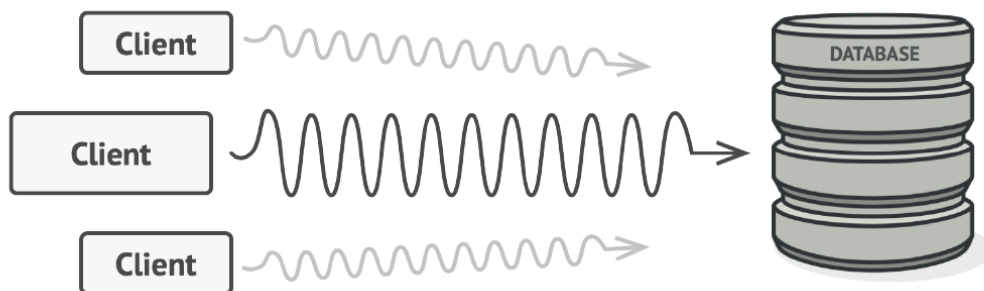
nuestra aplicación en ficheros.

Intent

El Intent del proxy-patrón es proporcionar un puntero o marcador hacia otro objeto para controlar el acceso a él. Introduce un nivel adicional de indirección.

Problema

¿Para que queremos controlar el acceso a un objeto? Razón: tenemos un **objeto complejo de manejar** que consume una gran cantidad de recursos del sistema, por ejemplo, un **objeto específico de base de datos**. Además se **necesita de manera infrecuente**, cuando queremos recuperar datos o guardar datos, pero no siempre.

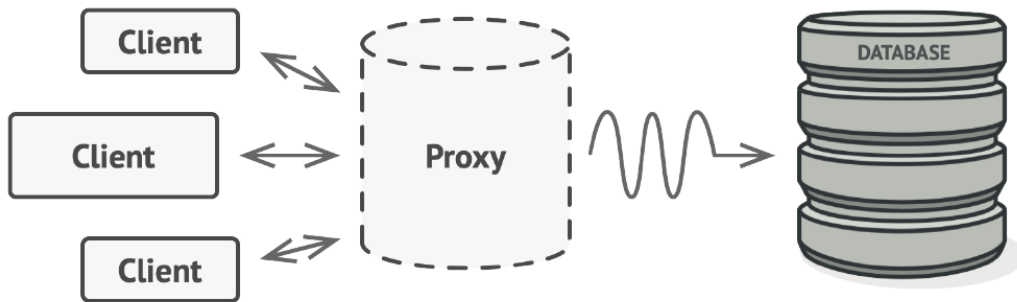


Las razones anteriormente expuestas nos obligan a implementar una **inicialización lazy**, bajo demanda: se crea este objeto solo cuando realmente sea necesario. Otro problema es que todos los clientes del objeto tendrían que ejecutar algún código de inicialización diferido. Desafortunadamente, esto probablemente **causaría una gran duplicación de código**.

Lo ideal sería introducir este **código común de inicialización directamente en la clase de nuestro objeto específico**, pero eso no siempre es posible. Lo habitual es que estos objetos específicos de acceso a base de datos, por ejemplo jdbc de MySQL u ORACLE, formen parte de una librería de terceros, una API externa.

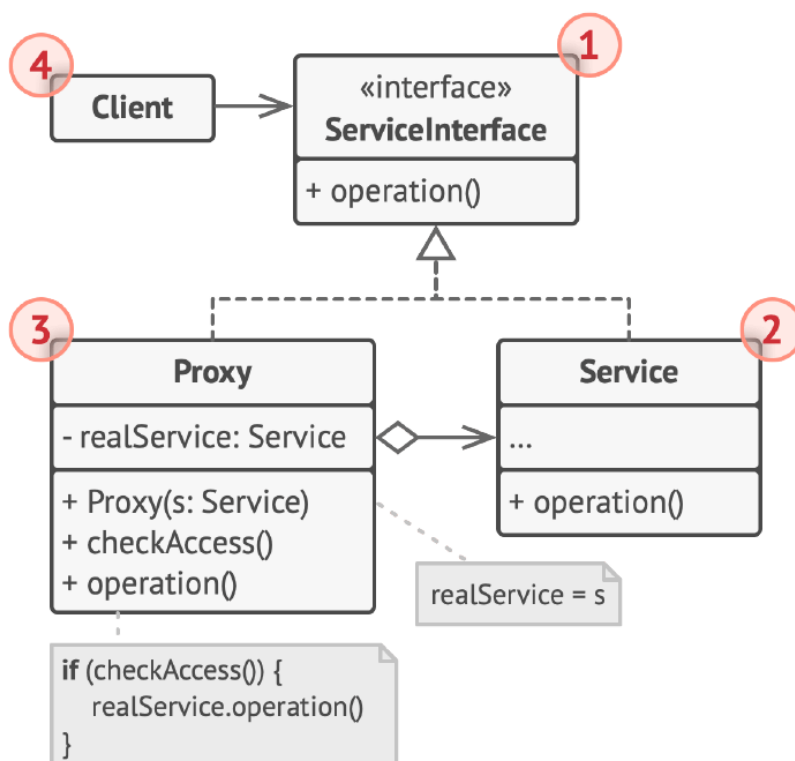
Solución

El patrón Proxy sugiere que cree una nueva clase de proxy con la misma interfaz que el objeto original. A continuación, se modifica la aplicación para que todas las peticiones al objeto original pasen por el objeto Proxy. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real, el original, y delega todo el trabajo en él objeto específico, pasándole o redirigiendo las peticiones.



¿Pero cuál es el beneficio? Si se necesita ejecutar cualquier código antes o después de la lógica principal de la clase, como inicializar la base de datos, crear un driver para acceder a ella o crear ficheros, el objeto proxy te permite hacerlo sin cambiar esa clase específica de base de datos. Puesto que el proxy implementa la misma interfaz que la clase original, se puede pasar a cualquier cliente que espera un objeto de servicio real.

Estructura

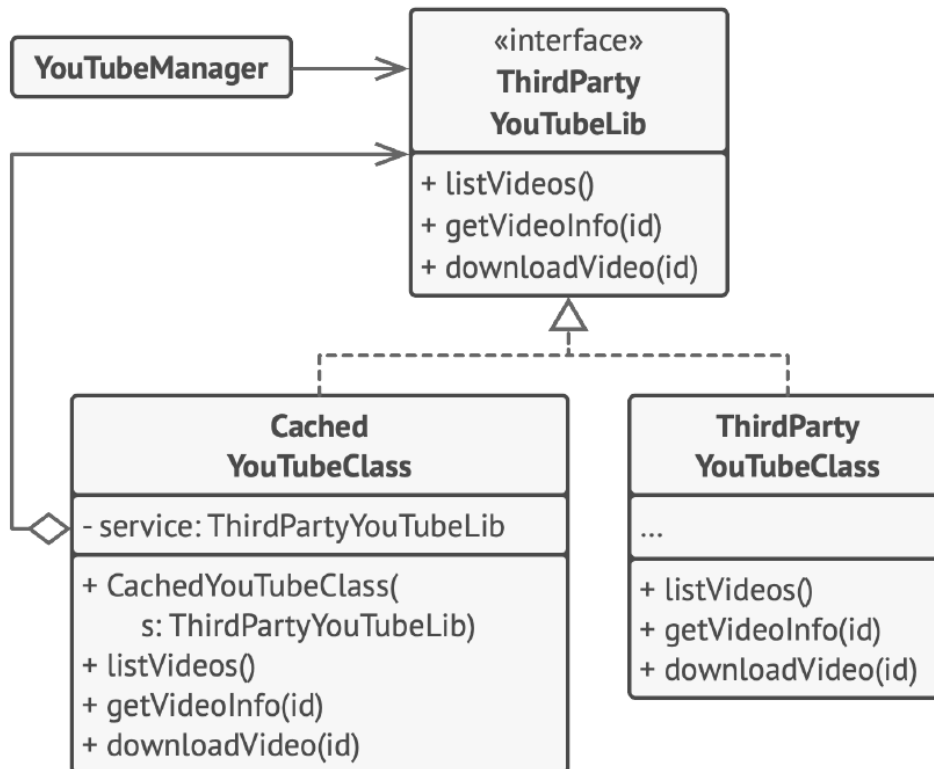


1. La interfaz **ServiceInterface** declara la interfaz del servicio. El proxy debe seguir esta interfaz para poder disfrazarse de objeto de servicio.
2. El servicio es una clase que proporciona cierta lógica de negocio útil.

3. La clase Proxy **tiene un campo de referencia que apunta a un servicio Objeto**. Después de que el proxy termine su procesamiento (por ejemplo, inicialización diferida, registro, control de acceso, almacenamiento en caché, etc.), **pasa la solicitud al objeto de servicio**. Por lo general, **los proxies administran el ciclo de vida de sus objetos de servicio**. Cuando crear la conexión con la base de datos, cuando cerrarla, cuando abrir ficheros cuando cerrarlo.
4. El **Cliente debe trabajar con los servicios y los servidores proxy a través de la misma interfaz**. De esta manera se puede pasar un proxy a cualquier código que espere un objeto de servicio.

Pseudocódigo

En este ejemplo se muestra cómo el patrón proxy puede ayudar a introducir la inicialización lazy y el almacenamiento en caché a una librería o API de terceros que integra una librería YouTube.



La librería original nos proporciona la clase de descarga de vídeo. Sin embargo, es muy ineficiente. Si la aplicación cliente solicita el mismo vídeo varias veces, la biblioteca lo descarga repetidas veces, en lugar de almacenar en caché y reutilizar el primer archivo descargado.

La clase proxy implementa la misma interfaz que el gestor de descargas original y delega todo el trabajo de descarga en él. Sin embargo, realiza un seguimiento de los archivos descargados y devuelve el resultado almacenado en caché cuando la aplicación solicita el mismo vídeo varias veces.

6.2 Ejemplo de patrón proxy para ficheros

En el siguiente ejemplo de Proxy vamos a crear un proyecto con la siguiente estructura de paquetes

```

v proxy.datos
  > GestorDatosFicheros.java
  > IService.java
v proxy.main
  > MainProxyUsuarios.java
v proxy.modelo
  > GeneraCamposAleatorios.java
  > GeneraUsuarios.java
  > Usuario.java

```

Usamos las clases de modelo que hemos definido en este tema `Usuario.java`, y en anteriores `GeneraUsuarios.java` y `GeneraCamposAleatorios.java`. El objetivo dentro de la aplicación del patrón Proxy es el manejo de los ficheros necesarios para almacenar y recuperar una lista de objetos `Usuario` de/desde ficheros.

`GestorDatosFicheros` será el objeto Proxy que delegará el trabajo en la clase `Files`, es decir el almacenado y recuperación de datos de/a fichero. Para ello va a implementar el interfaz `IService` que ofrece una interfaz parecida al objeto original `Files`. Podemos realizar un `write` de un objeto y un `read` de ese objeto. En este caso el objeto será de tipo `List<Usuario>`.

`IService.java`

```
import java.io.IOException;
import java.util.List;
import java.util.Optional;

import proxy.modelo.Usuario;

public interface IService {

    public boolean write(List<Usuario> listaUsuarios) throws
IOException ;
    public Optional<List<Usuario>> read() ;

}
```

Vamos a analizar la clase `GestorDatosFicheros.java`. Tenemos un método estático que inicializa el Gestor, `inicializaGestorSingleton()`, creando el fichero y directorio necesario para guardar los datos, la lista de usuarios,

de nuestra aplicación. Como indicábamos el Proxy se encarga de estas tareas necesarias para usar el servicio original.

Además, vamos a crear una única instancia del objeto proxy usando **Singleton**. Es habitual *mezclar patrones para dar una solución a un problema* de programación. En lugar de null, como hemos usado en el patrón Singleton en temas anteriores para comprobar que existe instancia o no del objeto, vamos a usar **Optional** en su lugar, para guardar la instancia del Objeto Proxy.

El directorio y fichero a crear son los siguientes. Comprobar que se crean cuando ejecutéis el programa.

```
private static String dir="usuariodatos";  
    private static String fich="FicheroUsuarios.Dat";  
private static Path directorio = Path.of("c:\\"+dir+"\\");  
  
    private static Path fichero = Path.of("c:\\"+dir+"\\",  
fich);
```

Para escribir el objeto lista de usuarios seguimos los siguientes pasos.

1. Creamos el directorio y el fichero necesario al inicializar el Proxy.

```
public static GestorDatosFicheros inicializaGestorSingleton() {
```

```
Files.createDirectory(directorio);
```

```
Files.createFile(fichero);
```

2. Además, se ha aplicado el patrón singleton para que se genere una sólo instancia de GestorDatosFicheros. Es interesante hacerlo así con cualquier componente de acceso a datos, sea ficheros o bases de datos. Por hacer el patrón Singleton más Java 8, como yahemos comentado, hemos usado un Optional de tipo GestorDatosFicheros. Si el Optional está vacío creamos una nueva instancia de GestorDatosFicheros

```
private static Optional<GestorDatosFicheros> instanciaGestor =  
Optional.empty();
```

```
if (GestorDatosFicheros.instanciaGestor.isEmpty())  
    instanciaGestor = Optional.of(new  
    GestorDatosFicheros());
```

3. Sino devolvemos la instancia ya creada con un `get()`, que está almacenada en la variable de tipo `Optional`.

```
return instanciaGestor.get();
```

4. Tenemos además el método `write` para escribir la Lista de Usuarios a fichero, que usa del servicio original `Files`. Aplicando métodos privados y **Single Responsibility Principle** hemos creado dos métodos privados, para transformar de bytes a `List<Usuario>` y viceversa:

`deListaABytes(listaUsuarios)` y `deBytesALista(bytesFichero)`

```
public boolean write(List<Usuario> listaUsuarios) {  
    // TODO Auto-generated method stub  
    try {  
        Files.write(fichero,  
        deListaABytes(listaUsuarios), StandardOpenOption.CREATE);
```

Para leer los bytes de fichero y transformar a lista de usuarios seguimos los siguientes pasos:

1. Usamos el método `read()` que lee la lista del fichero. Primero lee los bytes

```
byte[] bytesFichero= Files.readAllBytes(fichero);
```

2. Los transforma a lista

```
List<Usuario> listaResultado = deBytesALista(bytesFichero);
```

3. Y devolvemos un `Optional<List<Usuario>>`. El uso de un objeto `Optional` en lugar de un objeto `List<Usuario>` es muy útil y evita muchos errores con el `NullPointerException`.

```
optResultado= Optional.of(listaResultado);
```

```
return optResultado;
```

```
@Override
public Optional<List<Usuario>> read() {

    Optional<List<Usuario>> optResultado =
Optional.empty();
    // TODO Auto-generated method stub
    try {
        byte[] bytesFichero=
Files.readAllBytes(fichero);

        List<Usuario> listaResultado =
deBytesALista(bytesFichero);
        optResultado= Optional.of(listaResultado);
    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
        return optResultado;
    }

    return optResultado;
}
```

GestorDatosFicheros.java

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.nio.file.Files;
import java.nio.file.LinkOption;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.nio.file.attribute.DosFileAttributeView;
import java.util.List;
import java.util.Optional;

import proxy.modelo.Usuario;

public class GestorDatosFicheros implements IService {
```

```

        private static String dir="usuariodatos";
        private static String fich="FicheroUsuarios.Dat";

        private static Path directorio = Path.of("c:\\"+dir+"\\");

        private static Path fichero = Path.of("c:\\"+dir+"\\",
fich);

        private static Optional<GestorDatosFicheros>
instanciaGestor = Optional.empty() ;

        private GestorDatosFicheros() {

        }

        public static GestorDatosFicheros
inicializaGestorSingleton() {

            try {

                if (!Files.exists(directorio))
                    Files.createDirectory(directorio);
                DosFileAttributeView fileAttributeView =
                if (!Files.exists(fichero))
                    Files.createFile(fichero);

                if
(GestorDatosFicheros.instanciaGestor.isEmpty())

                    instanciaGestor = Optional.of(new
GestorDatosFicheros());

            } catch (Exception e) {

                e.printStackTrace();

            }

            return instanciaGestor.get();

        }

        @Override
        public boolean write(List<Usuario> listaUsuarios) {
            // TODO Auto-generated method stub
            try {

                Files.write(fichero,
deListaABytes(listaUsuarios), StandardOpenOption.CREATE);

```

```

        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            return false;
        }

        return true;
    }

    @Override
    public Optional<List<Usuario>> read() {

        Optional<List<Usuario>> optResultado =
Optional.empty();
        // TODO Auto-generated method stub
        try {
            byte[] bytesFichero=
Files.readAllBytes(fichero);

            List<Usuario> listaResultado =
deBytesALista(bytesFichero);
            optResultado= Optional.of(listaResultado);
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            return optResultado;
        }

        return optResultado;
    }

    private byte[] deListaABytes (List<Usuario> listaUsu)
throws IOException {

        ByteArrayOutputStream bos = new
ByteArrayOutputStream();
        ObjectOutputStream out = new
ObjectOutputStream(bos);
        out.writeObject(listaUsu);
        out.flush();
        return bos.toByteArray();
    }

    private List<Usuario> deBytesALista(byte[] bytes) {

        List<Usuario> listaUsuRecuperada=null;
        ByteArrayInputStream bis = new
ByteArrayInputStream(bytes);

```

```

        ObjectInputStream in;

        try {
            in = new ObjectInputStream(bis);

            listaUsuRecuperada = (List<Usuario>)
in.readObject();
        } catch (IOException |
ClassNotFoundException e) {
            // TODO Auto-generated catch
block
            e.printStackTrace();
        }
        return listaUsuRecuperada;
    }

}

```

Vamos a ver ahora el programa principal **MainProxyUsuarios.java**. En el vamos a realizar los siguientes pasos:

1. Creamos una lista con 100 usuarios.

```

List<Usuario> listaUsu =
GeneraUsuarios.devuelveUsuariosLista(100);

```

2. Creamos e inicializamos una instancia de nuestro objeto proxy.

```

GestorDatosFicheros gestorProxy =
GestorDatosFicheros.inicializaGestorSingleton();

```

3. Escribimos la lista en fichero.

```

gestorProxy.write(listaUsu);

```

4. Leemos la lista de ficheros como un Optional

```
Optional<List<Usuario>> optLista = gestorProxy.read();
```

5. **Controlamos los errores.** Si el Optional está vacío, ha habido un error en la lectura de los datos. Si está lleno, todo ha ido correctamente. Finalmente, mostramos la lista.

```
if (optLista.isEmpty()) {  
    System.out.println("Los datos no se recuperaron  
correctamente");  
} else {  
    System.out.println("La lista de usuarios  
recuperada de fichero es: ");  
    optLista.get().stream().forEach(System.out::println);  
}
```

Nota: Fijaos como en ningún momento usamos ficheros o Paths o la clase Files, en el programa principal, y no sabemos en qué directorio o ficheros se están guardando los datos. Una de las ventajas del patrón Proxy, es que encapsula, y enmascara los detalles de implementación. Al usuario le da igual en qué fichero se guardan nuestros datos, le importa que se guarden.

MainProxyUsuarios.java

```
import java.io.IOException;  
import java.util.List;  
import java.util.Optional;  
  
import proxy.datos.GestorDatosFicheros;  
import proxy.modelo.*;  
  
public class MainProxyUsuarios {  
    public static void main(String[] args) {  
        List<Usuario> listaUsu =  
        GeneraUsuarios.devuelveUsuariosLista(100);  
  
        System.out.println("Creamos la lista de usuarios");  
    }  
}
```

```

        listaUsu.stream().forEach(System.out::println);

        GestorDatosFicheros gestorProxy =
        GestorDatosFicheros.inicializaGestorSingleton();
        gestorProxy.write(listaUsu);

        Optional<List<Usuario>> optLista =
        gestorProxy.read();

        if (optLista.isEmpty()) {
            System.out.println("Los datos no se
recuparon correctamente");
        } else {
            System.out.println("La lista de usuarios
recuperada de fichero es: ");
            optLista.get().stream().forEach(System.out::println);
        }
    }
}

```

6.3 Patrón visitor. **Actividad guiada patrón Visitor**

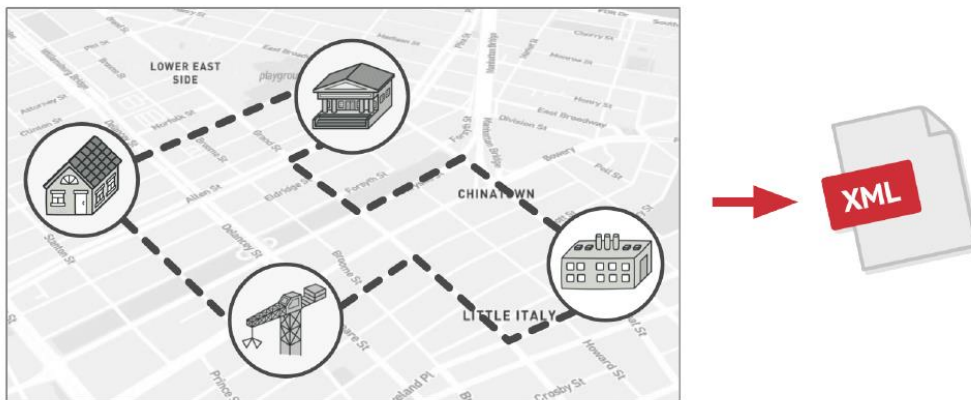
Visitor es un patrón de diseño de comportamiento que permite separar los algoritmos de los objetos en los que operan esos algoritmos.

Intent.

Representa una operación que se va a realizar en los elementos de una estructura de objetos. Visitor te permite definir una nueva operación sin cambiar las clases de los elementos en los que opera.

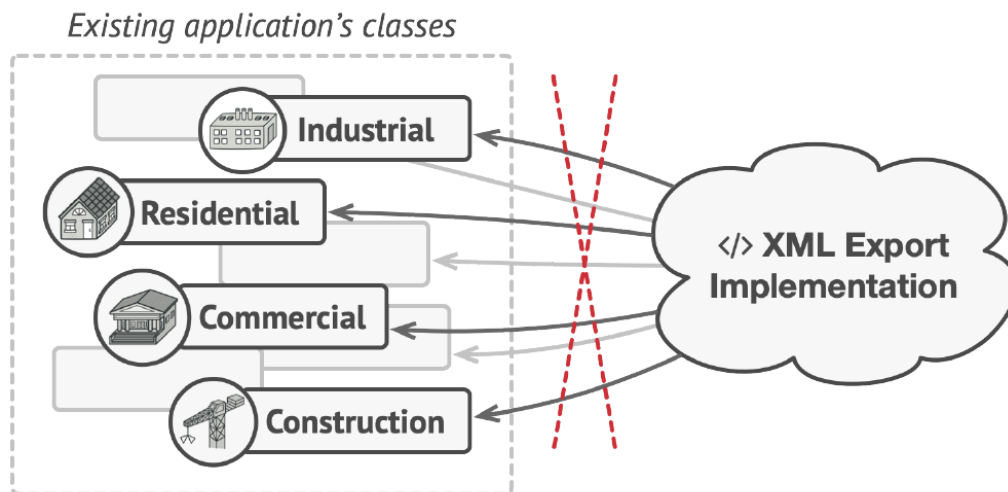
Problema

Imagina que tu equipo desarrolla una aplicación que funciona con información geográfica estructurada como un gráfico de gran tamaño. Cada nodo del gráfico puede representar una entidad compleja como una ciudad, pero también entidades más simples como industrias, áreas turísticas, etc. Los nodos están conectados entre sí, si hay una carretera entre los objetos reales que representan. En la implementación subyacente, cada tipo de nodo está representado por su propia clase, mientras que cada nodo específico es un objeto. Para ciudad clase Ciudad, para industrias clase Industria, etc. son clases diferentes, pero todos deben ser visitados en nuestro programa.



Durante el desarrollo de la aplicación, se requiere la tarea de implementar la exportación del gráfico en formato XML. El trabajo parece bastante sencillo. Se planea agregar un método de exportación a cada clase de nodo y, a continuación, aprovechar la recursividad para pasar cada nodo del gráfico, ejecutando el método de exportación. La solución es simple y elegante: gracias al polimorfismo, no estabas acoplando el código que llamaba el método de exportación a cada clase concreta que puede ser un nodo del grafo.

Desafortunadamente, el arquitecto del sistema se niega a permitirte modificar las clases que pueden ser nodo del grafo. Afirma que el código ya estaba en producción y que no quería arriesgarse a cambiar las clases pues pueden surgir errores en los cambios. Además, es mala práctica modificar una clase de modelo de datos para realizar una tarea externa a ella. Es mejor llevar esa tarea a una clase aparte.



Además, el arquitecto cuestiona si tiene sentido añadir el código para exportar a XML dentro de las clases de nodo. El trabajo principal de estas clases era trabajar con datos geográficos. El comportamiento de exportación XML es ajeno a la función de las clases.

Había otra razón para la negativa. Era muy probable que después de que se implemente esta característica, alguien del departamento de marketing pida que se proporcione la capacidad de exportar a un formato diferente. Esto obligaría a cambiar esas clases de nuevo.

En resumen, en nuestras clases de modelo que en Java llamaríamos POJO, no es aconsejable añadir implementaciones para una solución específica como transformar o exportar a XML. Esa implementación se debe separar de nuestro modelo original.

Solucion

El patrón Visitor nos sugiere que coloquemos el nuevo comportamiento en una clase independiente denominada **visitor**, en lugar de intentar integrarlo en las clases existentes. El objeto original que tiene que implementar el comportamiento de exportar a XML ahora se pasa a uno de los métodos del objeto visitor como argumento, proporcionando métodos de acceso a todos los datos necesarios contenidos en el objeto.

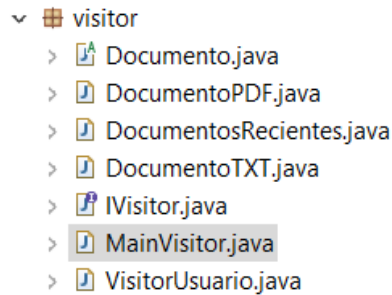
Ahora, ¿qué pasa si ese comportamiento se puede ejecutar sobre objetos de diferentes clases? Por ejemplo, en nuestro caso con la exportación XML, la implementación real debe ser un pelin diferente para las diferentes clases que pueden ser nodos del grafo. Por lo tanto, la clase visitor puede definir no uno, sino un conjunto de métodos, cada uno de los cuales podría tomar argumentos de diferentes tipos. En el SimpleFileVisitor habeis visto como se da un tratamiento distinto a ficheros y directorios con métodos diferentes.

Ejemplo de implementación de visitor

Para implementar el patrón visitor hemos creado una pequeña aplicación con tres tipos de documentos. DocumentoPDF, DocumentoTXT, y DocumentoRecientes (recientes). Recientes puede contener una lista con

documentos de tipo PDF o TXT. La idea es que el usuario a través de un objeto `UsuarioVisitor` puede visitar PDF's TXT's o recientes. Cuando visitamos un documento se abre. Si es PDF se abrirá el fichero en el acrobar reader o lector PDF que tengais como predefinido, si es TXT se abrirá el block de notas, y si es REC abrirá toda la lista de documentos recientes, ya sean PDF o TXT. Tres comportamientos diferentes en las visitas a documentos.

Para crear esta aplicación vamos a necesitar las siguientes clases e interfaces:



```
visitor
├── Documento.java
├── DocumentoPDF.java
├── DocumentosRecientes.java
├── DocumentoTXT.java
├── IVisitor.java
├── MainVisitor.java
└── VisitorUsuario.java
```

Empezamos por la clase abstracta `Documento`. Es muy sencilla no requiere explicacion, tenemos el Path del documento su nombre, y el tipo con sus getters y setters.

Documento.java

```
public abstract class Documento {

    private String documentoPath;
    private String nombre;

    public Documento(String documentoPath, String nombre) {
        super();
        this.documentoPath = documentoPath;
        this.nombre = nombre;
    }

    public String getDocumentoPath() {
        return documentoPath;
    }

    public void setDocumentoPath(String documentoPath) {
        this.documentoPath = documentoPath;
    }

    public String getNombre() {
```

```

        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    @Override
    public String toString() {
        return "Documento [documentoPath=" + documentoPath
+ ", nombre=" + nombre + "];"
    }

}

```

Continuamos con los documentos específicos. Heredan de Documento `extends Documento` y llaman al constructor de la clase padre.
DocumentoPDF.java

```

public class DocumentoPDF extends Documento {

    public DocumentoPDF(String documentoPath, String nombre) {
        super(documentoPath, nombre);
        // TODO Auto-generated constructor stub
    }

}

```

DocumentoTXT.java

```
public class DocumentoTXT extends Documento {

    public DocumentoTXT(String documentoPath, String nombre) {
        super(documentoPath, nombre);
        // TODO Auto-generated constructor stub
    }

}
```

Reseñable en DocumentosRecientes.java es que tenemos una lista de documentos recientes a la que podemos añadir objetos de tres tipos Documentos, TXT, PDF o de el mismo de Documentos Recientes. ¿Que patrón ya visto estamos usando aquí?

```
private List<Documento> listaRecientes;
```

```
import java.util.ArrayList;
import java.util.List;

public class DocumentosRecientes extends Documento {

    private List<Documento> listaRecientes;

    public DocumentosRecientes(String documentoPath, String
nombre) {
        super(documentoPath, nombre);

        listaRecientes = new ArrayList<Documento> ();
    }

    public void addDocumentoReciente(Documento documento) {

        listaRecientes.add(documento);
    }
}
```

```

    }

    public List<Documento> getListaRecientes() {
        return listaRecientes;
    }

}

```

Estos son los documentos a visitar por el usuario. Necesitamos el visitor. Para establecer el visitor usamos un interfaz IVisitor con tres métodos. Cada método debe ser implementado para visitar un tipo de documento diferente.

IVisitor.java

```

public interface IVisitor {
    public void visit(DocumentoTXT txt) ;
    public void visit(DocumentoPDF pdf);
    public void visit(DocumentosRecientes rec);
}

```

El objeto que implementa las visitas, es VisitorUsuario. Como veréis almacena información del usuario, el nombre del usuario, pero nos vamos a centrar en la implementación de las visitas. Implementa las visitas para tres documentos diferentes.

Para documentos de tipo TXT tenemos el método `public void visit(DocumentoTXT txt)`. Abre el fichero en el block de notas, con la clase Desktop, y su método `Desktop.getDesktop().open`.

```

Desktop.getDesktop().open(myFile);

```

```

@Override
    public void visit(DocumentoTXT txt) {
        System.out.println(
            "Usuario: " + this.nombreusuario + " Abriendo

```

```

documento de texto" + txt.getNombre());

        System.out.println("Abrimos el Acrobat Reader y
mostramos el documento");

        File myFile = new
File(txt.getDocumentoPath()+txt.getNombre());

        try {
            Desktop.getDesktop().open(myFile);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}

```

Para documentos PDF, abre el documento en el lector de PDF de igual manera, lo tenéis marcado en el código.

```

@Override
    public void visit(DocumentoPDF pdf) {

        System.out.println(
            "Usuario: " + this.nombreusuario + " Abriendo
documento pdf"+ pdf.getNombre());

        System.out.println("Abrimos el Acrobat Reader y
mostramos el documento");

        File myFile = new
File(pdf.getDocumentoPath()+pdf.getNombre());

        try {
            Desktop.getDesktop().open(myFile);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}

```

Finalmente, para documentos recientes, el método `public void visit(DocumentosRecientes rec)` recorre la lista de documentos y abre

todos.

El truco de orientación a objetos aquí es adivinar el tipo de documento y llamar al visit adecuado, pasando el tipo de objeto adecuando. Para ello estamos usando instanceof, y una nueva característica de Java 14, llamada Pattern Matching para instance of. Además de comprobar de que tipo de clase es realmente el objeto guardado en una variable de la Superclase documento, nos permite crear una variable de la subclase específica.

Observar: documento instanceof DocumentoPDF docPDF , si documento contiene un objeto de tipo DocumentoPDF, además de comprobarlo con instanceof lo almacenamos a la variable docPDF que es del tipo de la subclase específica.

Cuando hacemos la llamada al visit, la realizamos con su objeto específico, con lo que escogerá la versión de visit para PDF.

```
if (documento instanceof DocumentoPDF docPDF)

    visit((DocumentoPDF) documento);
```

Exactamente pasa lo mismo para documentos TXT .

```
else if (documento instanceof DocumentoTXT docTxt)

    visit(docTxt);
```

Y para recientes exactamente lo mismo. Analizar como tenemos Documentos Simples como PDF, y documentos compuestos de otros documentos como Documentos Recientes. ¿Qué patrón estamos usando?

```
else if (documento instanceof DocumentosRecientes docReciente)

    visit( docReciente);
```

Override

```
public void visit(DocumentosRecientes rec) {
    // TODO Auto-generated method stub

    for (Documento documento:
rec.getListasRecientes()) {

        if (documento instanceof DocumentoPDF docPDF)
```



```

        visit( docPDF);
    else if (documento instanceof DocumentoTXT docTxt)
        visit(docTxt);
    else if (documento instanceof DocumentosRecientes
docReciente)
        visit( docReciente);
    }
}

```

VisitorUsuario.java

```

import java.awt.Desktop;
import java.io.File;
import java.io.IOException;

public class VisitorUsuario implements IVisitor{

    public VisitorUsuario(String nombreusuario) {
        super();
        this.nombreusuario = nombreusuario;
    }

    private String nombreusuario;

    public String getNombreusuario() {
        return nombreusuario;
    }

    public void setNombreusuario(String nombreusuario) {
        this.nombreusuario = nombreusuario;
    }

    @Override
    public void visit(DocumentoTXT txt) {
        System.out.println(
            "Usuario: " + this.nombreusuario + " Abriendo
documento de texto" + txt.getNombre());

        System.out.println("Abrimos el Acrobat Reader y
mostramos el documento");
    }
}

```

```

        File myFile = new
File(txt.getDocumentoPath()+txt.getNombre());

        try {

            Desktop.getDesktop().open(myFile);
        } catch (IOException e) {
            // TODO Auto-generated catch
block
            e.printStackTrace();
        }

    }

    @Override
    public void visit(DocumentoPDF pdf) {

        System.out.println(
            "Usuario: " + this.nombreusuario + " Abriendo
documento pdf"+ pdf.getNombre());

        System.out.println("Abrimos el Acrobat
Reader y mostramos el documento");

        File myFile = new
File(pdf.getDocumentoPath()+pdf.getNombre());

        try {

            Desktop.getDesktop().open(myFile);
        } catch (IOException e) {
            // TODO Auto-generated catch
block
            e.printStackTrace();
        }

    }

    @Override
    public void visit(DocumentosRecientes rec) {
        // TODO Auto-generated method stub

        public void visit(DocumentosRecientes rec) {
            // TODO Auto-generated method stub

```

```

        for (Documento documento:
rec.getListaRecientes() ) {

        if (documento instanceof DocumentoPDF docPDF)

        visit( docPDF);
    else if (documento instanceof DocumentoTXT docTxt)

        visit(docTxt);
    else if (documento instanceof DocumentosRecientes
docReciente)

        visit( docReciente);

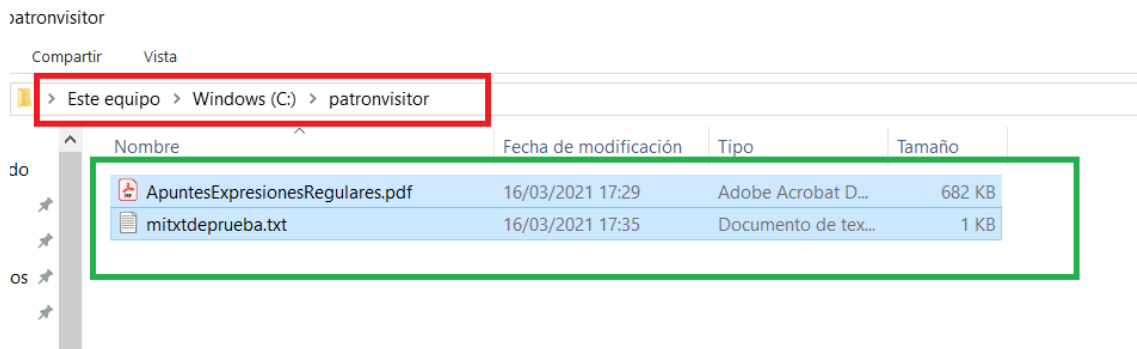
    }

}

}

```

Para finalizar vamos a revisar la clase principal. Pero antes aseguraos que tenéis **estos ficheros y el directorio creado** en vuestro ordenador antes de probar el proyecto. **Debéis crearlos vosotros, o copiarlos**. El pdf son los apuntes de Expresiones regulares que se proporcionaron en el curso. El **txt** podéis crearlo vosotros mismos.



En **MainVisitor.java**, nuestro programa principal vamos a realizar varios pasos.

1. Creamos el visitor, un documento PDF y otro TXT.

```

IVisitor v = new VisitorUsuario("Alumno");

DocumentoPDF pdf = new DocumentoPDF("c:\\patronvisitor\\",

```

```
"ApuntesExpresionesRegulares.pdf");  
    DocumentoTXT txt = new DocumentoTXT("c:\\patronvisitor\\",  
"mitxtdeprueba.txt");
```

2. Los visitamos con el visitor

```
v.visit(pdf);  
v.visit(txt);
```

3. Creamos un documento de recientes con los dos documentos anteriores

```
recientes.addDocumentoReciente(pdf);  
recientes.addDocumentoReciente(txt);
```

4. Lo visitamos

```
v.visit(recientes);
```

Para probar que la visita a documentos recientes funciona. Cerrar el pdf y el txt. Después comentar las dos líneas `v.visit(pdf);` y `v.visit(txt);` y volved a ejecutar.

MainVisitor.java

```
public class MainVisitor {  
  
    public static void main(String[] args) {  
  
        DocumentosRecientes recientes = new  
DocumentosRecientes("", "recientes");  
        IVisitor v = new VisitorUsuario("Alumno");  
  
        DocumentoPDF pdf = new  
DocumentoPDF("c:\\patronvisitor\\",  
"ApuntesExpresionesRegulares.pdf");  
        DocumentoTXT txt = new  
DocumentoTXT("c:\\patronvisitor\\", "mitxtdeprueba.txt");  
  
        v.visit(pdf);  
        v.visit(txt);  
  
        recientes.addDocumentoReciente(pdf);  
        recientes.addDocumentoReciente(txt);  
  
        v.visit(recientes);  
  
    }  
}
```

```
}
```

7 Bibliografía y referencias web

Tutoriales de Java Jacob Jenkov

<http://tutorials.jenkov.com/>

Tutoriales de Java arquitectura Java

<https://www.arquitecturajava.com/>

Repositorio Github Venkat Subramanian

<https://github.com/venkats>

Tutoriales Java geeksforgeeks

<https://www.geeksforgeeks.org/>

Tutoriales Java Baeldung

<https://www.baeldung.com/>

Bibliografía

Programación, Alfonso Jiménez Pérez, Francisco Manuel Pérez Montes, Paraninfo, 1ª edición, 2021

Acceso a Datos, Alicia Ramos Martín, Garceta 2ª Edición, 2018