

## Manejo de ficheros.

### Caso práctico



**Ana** está empezando a cursar la Formación en Centros de Trabajo (FCT).

Ya ha tenido unas reuniones con **Juan y María**, para saber cómo se trabaja en BK programación. Aunque está haciendo el módulo de FCT en esta empresa, ya sabe que a veces tendrá que salir a otras empresas acompañada de sus tutores para ver los requisitos de los sistemas que la empresa tenga que informatizar y, en ocasiones, **Antonio**, quizás también se apunte para echar una mano.

Ana está nerviosa, también ilusionada, y tiene muchas ganas de conocer de cerca la realidad de lo que ha estudiado en clase.

Ahora verá el uso de los conocimientos adquiridos de diferentes módulos, y buscará respuestas a posibles dudas que se vayan planteando.

En clase les habían explicado la importancia de los ficheros en el acceso a datos. -Es importante repasar los conceptos -piensa Ana.

# 1.- Introducción.



Si estás estudiando este módulo, es probable que ya hayas estudiado el de programación, por lo que no te serán desconocidos muchos conceptos que se tratan en este tema.

Ya sabes que cuando apagas el ordenador, los datos de la memoria **RAM** se pierden. Un ordenador utiliza ficheros para guardar los datos. Piensa que los datos de las canciones que oyes en mp3, las películas que ves en formato avi, o mp4, etc., están, al fin y al cabo, almacenadas en ficheros, los cuales están grabados en un soporte como son los discos duros, **DVD**, pendrives, etc.

Se llama a los datos que se guardan en ficheros **datos persistentes**, porque persisten más allá de la ejecución de la aplicación que los trata. Los ordenadores almacenan los ficheros en unidades de almacenamiento secundario como discos duros, discos ópticos, etc. En esta unidad veremos, entre otras cosas, cómo hacer con Java las operaciones de crear, actualizar y procesar ficheros.

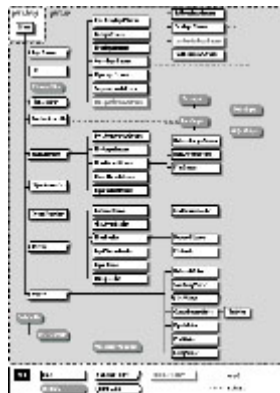
A las operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como Entrada/Salida (E/S).

Las operaciones de E/S en Java las proporciona el paquete estándar de la API de Java denominado **java.io** que incorpora [interfaces](#), clases y excepciones para acceder a todo tipo de ficheros.

La **librería java.io** contiene las clases necesarias para gestionar las operaciones de entrada y salida con Java. Estas clases de E/S las podemos agrupar fundamentalmente en:

- ✓ Clases para leer entradas desde un flujo de datos.
- ✓ Clases para escribir entradas a un flujo de datos.
- ✓ Clases para operar con ficheros en el sistema de ficheros local.
- ✓ Clases para gestionar la serialización de objetos.

En la imagen puedes ver las clases de las que se dispone en **java.io**.



## Autoevaluación

**Indica si la afirmación es verdadera o falsa:**

Los datos persistentes perduran tras finalizar la ejecución de la aplicación que los trata.

Verdadero. ☐ Falso. ☐

## 2.- Clases asociadas a las operaciones de gestión de ficheros y directorios.

### Caso práctico



**Ana** le comenta a **Antonio** -Por lo que nos han comentado, vamos a tener que utilizar bastante los ficheros. ¿Cómo te manejas tú con ellos?

-Pues tan bien como tú -responde **Antonio**, -yo también estudié el módulo de Programación. ¿Es que no recuerdas que en el módulo estudiamos un tema sobre ficheros?

-Sí, pero es que, como había tantos métodos para listar, renombrar archivos, etc., ya casi no me acuerdo; y eso que hace poco que lo

estudiamos -contesta **Ana**.

**Antonio** intenta tranquilizar a **Ana** y le dice que no se preocupe, que en cuanto se les presente la ocasión de tener que programar con ficheros, seguro que no tienen problema y refrescan los conceptos que aprendieron en su día.

En efecto, tal y como dicen Ana y Antonio, hay bastantes métodos involucrados en las clases que en Java nos permiten manipular ficheros y carpetas o directorios.

Vamos a ver la clase **File** que nos permite hacer unas cuantas operaciones con ficheros, también veremos cómo filtrar ficheros, o sea, obtener aquellos con una característica determinada, como puede ser que tengan la [extensión .odt](#), o la que nos interese, y por último, en este apartado también veremos como crear y eliminar ficheros y directorios.



## 2.1.- Clase File.

¿Para qué sirve esta clase, qué nos permite? La clase `File` proporciona una representación abstracta de ficheros y directorios.

Esta clase, permite examinar y manipular archivos y directorios, independientemente de la plataforma en la que se esté trabajando: Linux, Windows, etc.

Las instancias de la clase `File` representan nombres de archivo, no los archivos en sí mismos.

El archivo correspondiente a un nombre puede ser que no exista, por esta razón habrá que controlar las posibles **excepciones**.

Un objeto de clase `File` permite examinar el nombre del archivo, descomponerlo en su rama de directorios o crear el archivo si no existe, pasando el objeto de tipo `File` a un constructor adecuado como `FileWriter(File f)`, que recibe como parámetro un objeto `File`.

Para archivos que existen, a través del objeto `File`, un programa puede examinar los atributos del archivo, cambiar su nombre, borrarlo o cambiar sus permisos. Dado un objeto `File`, podemos hacer las siguientes operaciones con él:

- ✓ **Renombrar** el archivo, con el método `renameTo()`. El objeto `File` dejará de referirse al archivo renombrado, ya que el `String` con el nombre del archivo en el objeto `File` no cambia.
- ✓ **Borrar** el archivo, con el método `delete()`. También, con `deleteOnExit()` se borra cuando finaliza la ejecución de la máquina virtual Java.
- ✓ **Crear** un nuevo fichero con un nombre único. El método estático `createTempFile()` crea un fichero temporal y devuelve un objeto `File` que apunta a él. Es útil para crear archivos temporales, que luego se borran, asegurándonos tener un nombre de archivo no repetido.
- ✓ **Establecer** la fecha y la hora de modificación del archivo con `setLastModified()`. Por ejemplo, se podría hacer: `new File("prueba.txt").setLastModified(new Date().getTime());` para establecerle la fecha actual al fichero que se le pasa como parámetro, en este caso `prueba.txt`.
- ✓ **Crear** un directorio, mediante el método `mkdir()`. También existe `mkdirs()`, que crea los directorios superiores si no existen.
- ✓ **Listar** el contenido de un directorio. Los métodos `list()` y `listFiles()` listan el contenido de un directorio. `list()` devuelve un vector de `String` con los nombres de los archivos, `listFiles()` devuelve un vector de objetos `File`.
- ✓ **Listar** los nombres de archivo de la raíz del sistema de archivos, mediante el método estático `listRoots()`.

Nombre	Ta...	Tipo
SoftwareDistribution		Carpeta de archivos
srchassst		Carpeta de archivos
Sun		Carpeta de archivos
system		Carpeta de archivos
system32		Carpeta de archivos
Tasks		Carpeta
Temp		Carpeta de archivos
twain_32		Carpeta de archivos
wBEM		Carpeta de archivos
Web		Carpeta de archivos
WinSxS		Carpeta de archivos
_default	1 KB	Acceso directo al pr...
0.log	0 KB	Documento de texto
A pescar.bmp	17 KB	Imagen de mapa de...
Abanicos.bmp	27 KB	Imagen de mapa de...



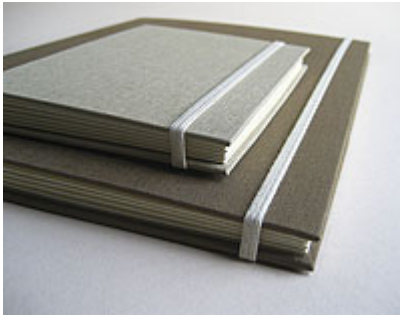
### Autoevaluación

Señala si la afirmación es verdadera o falsa:

Podemos establecer la fecha de modificación de un archivo mediante el método `renameTo()`.

Verdadero. ☐ Falso. ☐

## 2.1.1.- Clase File (II).



Mediante la clase **File**, podemos ver si un fichero cualquiera, digamos por ejemplo **texto.txt**, existe o no. Para ello, nos valemos del método **exists()** de la clase **File**. Hacemos referencia a ese fichero en concreto con el código siguiente:

```
File f = new File("C:\\texto.txt");
```

En el siguiente recurso puedes descargar una pequeña aplicación en la que se usa **File**. Como verás, permite listar el contenido de la carpeta que pongas en un campo de texto, introduciendo el resultado en un **JList**.

[Listar archivos.](#) (0.01 MB)

Pero, ¿qué pasa si nos interesa copiar un fichero, cómo lo haríamos?

Con la clase **File** no es suficiente, necesitamos saber más, en concreto, necesitamos hablar de los flujos, como vamos a ver más adelante.

### Para saber más

En este enlace puedes ver ejemplos para obtener las propiedades de los ficheros, usando la clase **File**:

[Propiedades de los ficheros](#)

[Ejemplo creando carpetas o directorios en Java](#)



### Autoevaluación

Señala la opción correcta. Con la clase **File** podemos:

- ☐ Crear ficheros temporales.
- ☐ Crear directorios.
- ☐ Renombrar un archivo.
- ☐ Todas son correctas.



## 2.2.- Interface FilenameFilter.

Hemos visto como obtener la lista de ficheros de una carpeta o directorio. A veces, nos interesa ver no la lista completa, sino los archivos que encajan con un determinado criterio.

Por ejemplo, nos puede interesar un filtro para ver los ficheros modificados después de una fecha, o los que tienen un tamaño mayor del que el que indiquemos, etc.

El interface **FilenameFilter** se puede usar para crear filtros que establezcan criterios de filtrado relativos al nombre de los ficheros. Una clase que lo implemente debe definir e implementar el método:



```
boolean accept(File dir, String nombre)
```

Este método devolverá verdadero en el caso de que el fichero cuyo nombre se indica en el parámetro **nombre** aparezca en la lista de los ficheros del directorio indicado por el parámetro **dir**.

En el siguiente ejemplo vemos cómo se listan los ficheros de la carpeta **c:\datos** que tengan la extensión **.txt**. Usamos **try** y **catch** para capturar las posibles excepciones, como que no exista dicha carpeta.

[Listar ficheros de una carpeta, filtrando.](#)

### Para saber más

En el ejemplo anterior se utiliza la función **endsWith**. Por si no sabes para que se emplea, y para ver otras más sobre tratamiento de cadenas, sigue este enlace:

[Operaciones con cadenas.](#)

## 2.3.- Rutas de los ficheros



En los ejemplos que vemos en el tema, estamos usando la ruta de los ficheros tal y como se usan en MS-DOS, o Windows, es decir, por ejemplo:

```
C:\\datos\\Programacion\\fichero.txt
```

Cuando operamos con rutas de ficheros, el carácter separador entre directorios o carpetas suele cambiar dependiendo del sistema operativo en el que se esté ejecutando el programa.

Para evitar problemas en la ejecución de los programas cuando se ejecuten en uno u otro sistema operativo y, por tanto, persiguiendo que nuestras aplicaciones sean lo más portables posibles, se recomienda usar en Java: **File.separator**.

Podríamos hacer una función que, al pasarle una ruta, nos devolviera la adecuada según el separador del sistema actual, del siguiente modo:

```

import java.io.*;
import java.util.*;

public class Ruta {
    // Separador de rutas
    private static String separator;

    // Constructor
    public Ruta() {
        // Obtener el separador de rutas del sistema operativo
        separator = System.getProperty("file.separator");
    }

    // Método para obtener el separador de rutas
    public static String getSeparator() {
        return separator;
    }
}

```

Código de separador de rutas.



### Autoevaluación

**Cuando trabajamos con fichero en Java, no es necesario capturar las excepciones, el sistema se ocupa automáticamente de ellas.**

Verdadero. ☐ Falso. ☐

## 2.4.- Creación y eliminación de ficheros y directorios.

Cuando queramos **crear un fichero**, podemos proceder del siguiente modo:

```

1 // Creamos un fichero que almacenará el directorio
2 File fichero = new File("C:\\Programas\\Directorio.txt");
3 // A partir del objeto File creamos el directorio
4 if (fichero.createNewFile())
5 {
6     System.out.println("Fichero se ha creado correctamente");
7 }
8 else
9 {
10    System.out.println("No se puede crear porque ya existe");
11 }
12 catch (Exception e) {
13    e.printStackTrace();
14 }
15

```

Código de crear un fichero.

Para **borrar un fichero** podemos usar la clase `File`, comprobando previamente si existe el fichero, del siguiente modo:

```

1 File fichero = new File("C:\\Programas\\Directorio.txt");
2 if (fichero.exists())
3 {
4     fichero.delete();
5 }
6

```

Para **crear directorios**, podríamos hacer:

```

1 // Creamos un directorio
2 String directorio = "C:\\Programas";
3 String nombre = "Carpeta\\Carpeta\\Carpeta";
4
5 // Creamos el directorio
6 boolean existe = (new File(directorio).mkdir());
7 if (existe)
8 {
9     System.out.println("Directorio " + directorio + " creado");
10 }
11 else
12 {
13     existe = (new File(nombre).mkdir());
14 }
15 if (existe)
16 {
17     System.out.println("Directorio " + nombre + " creado");
18 }
19 catch (Exception e) {
20     System.out.println("Error " + e.getMessage());
21 }
22

```

Código de crear un directorio.

Para **borrar un directorio** con Java, tendremos que borrar cada uno de los ficheros y directorios que éste contenga. Al poder almacenar otros directorios, se podría recorrer **recursivamente** el directorio para ir borrando todos los ficheros.

Se puede listar el contenido del directorio e ir borrando con:

```
File[] ficheros = directorio.listFiles();
```

Si el elemento es un directorio, lo sabemos mediante el método `isDirectory()`.



### Autoevaluación

Indica si la siguiente afirmación es verdadera o falsa:

Podemos establecer un criterio para listar ficheros mediante `FileNameFilter`.

Verdadero. ☐ Falso. ☐



### 3.- Flujos.

#### Caso práctico

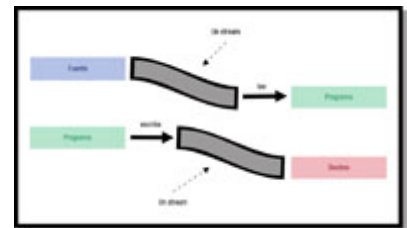


Ana y Antonio saben que van a tener que ayudar a Juan y a María en labores de programación de ficheros en Java. Así que, además, de la clase **File**, van a necesitar utilizar otros conceptos relacionados con la entrada y salida: los flujos o **streams**. Ana recuerda que hay dos tipos de flujos: flujos de caracteres y flujos de bytes.

Un programa en Java, que necesita realizar una operación de entrada/salida (en adelante **E/S**), lo hace a través de un flujo o **stream**.

Un **flujo** es una **abstracción de todo aquello que produce o consume información**.

La **vinculación de este flujo al dispositivo físico la hace el sistema de entrada y salida** de Java.



Las clases y métodos de E/S que necesitamos emplear son las mismas independientemente del dispositivo con el que estemos actuando. Luego, el núcleo de Java sabrá si tiene que tratar con el teclado, el monitor, un sistema de archivos o un socket de red; liberando al programador de tener que saber con quién está interactuando.

Java define dos tipos de flujos en el paquete **java.io**:

- ✓ **Byte streams** (8 bits): proporciona lo necesario para la gestión de entradas y salidas de bytes y su uso está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene determinado por dos clases abstractas que son **InputStream** y **OutputStream**. Estas dos clases definen los métodos que sus subclases tendrán implementados y, de entre todos, destacan **read()** y **write()** que leen y escriben bytes de datos respectivamente.
- ✓ **Character streams** (16 bits): de manera similar a los flujos de bytes, los flujos de caracteres están determinados por dos clases abstractas, en este caso: **Reader** y **Writer**. Dichas clases manejan flujos de caracteres Unicode. Y también de ellas derivan subclases concretas que implementan los métodos definidos en ellas siendo los más destacados los métodos **read()** y **write()** que leen y escriben caracteres de datos respectivamente.

## 3.1.- Flujos basados en bytes.

Para el tratamiento de los flujos de bytes, hemos dicho que Java tiene dos clases abstractas que son **InputStream** y **OutputStream**.

Los archivos binarios guardan una representación de los datos en el archivo, es decir, cuando guardamos texto no guardan el texto en sí, sino que guardan su representación en un código llamado [UTF-8](#).

Las clases principales que heredan de **OutputStream**, para la escritura de ficheros binarios son:

- ✓ **FileOutputStream**: escribe bytes en un fichero. Si el archivo existe, cuando vayamos a escribir sobre él, se borrará. Por tanto, si queremos añadir los datos al final de éste, habrá que usar el constructor `FileOutputStream(String filePath, boolean append)`, poniendo `append` a `true`.
- ✓ **ObjectOutputStream**: convierte objetos y variables en vectores de bytes que pueden ser escritos en un **OutputStream**.
- ✓ **DataOutputStream**, que da formato a los tipos primitivos y objetos `String`, convirtiéndolos en un flujo de forma que cualquier `DataInputStream`, de cualquier máquina, los pueda leer. Todos los métodos empiezan por "write", como `writeByte()`, `writeFloat()`, etc.



De **InputStream**, para lectura de ficheros binarios, destacamos:

- ✓ **FileInputStream**: lee bytes de un fichero.
- ✓ **ObjectInputStream**: convierte en objetos y variables los vectores de bytes leídos de un **InputStream**.

En el siguiente ejemplo se puede ver cómo se escribe a un archivo binario con **DataOutputStream**:

**Escritura a fichero binario.**

[Resumen textual alternativo](#)

[Código de proyecto de escritura a fichero binario.](#) (0.01 MB)

En el siguiente enlace puedes ver cómo leer de un archivo binario mediante la clase **FileInputStream**:

[Leer binario](#)



### Autoevaluación

**Señala si la afirmación es verdadera o falsa:**

Podemos escribir datos binarios a ficheros utilizando el método **append** de la clase **DataOutputStream**.

Verdadero. ☐ Falso. ☐

## 3.2.- Flujos basados en caracteres.



Para los flujos de caracteres, Java dispone de dos clases abstractas: **Reader** y **Writer**.

Si se usan sólo **FileInputStream**, **FileOutputStream**, **FileReader** o **FileWriter**, cada vez que se efectúa una lectura o escritura, se hace físicamente en el disco duro. Si se leen o escriben pocos caracteres cada vez, el proceso se hace costoso y lento por los muchos accesos a disco duro.

Los **BufferedReader**, **BufferedInputStream**, **BufferedWriter** y **BufferedOutputStream** añaden un **buffer** intermedio. Cuando se lee o escribe, esta clase controla los accesos a disco. Así, si vamos escribiendo, se guardarán los datos hasta que haya bastantes datos como para hacer una escritura eficiente.

Al leer, la clase leerá más datos de los que se hayan pedido. En las siguientes lecturas nos dará lo que tiene almacenado, hasta que necesite leer otra vez físicamente. Esta forma de trabajar hace los accesos a disco más eficientes y el programa se ejecuta más rápido.

En el siguiente ejemplo, puedes ver cómo se lee un archivo guardado en el directorio raíz, denominado **archivo.txt**, cuando se pulsa el botón de la aplicación. El contenido se introduce en el área de texto.

[Proyecto Java para leer fichero de texto.](#) (0.01 MB)

### Debes conocer

Vídeo sobre el paquete java.io.

Vídeo sobre java.io.

[Resumen textual alternativo](#)

### Para saber más

En este enlace puedes aprender más sobre internacionalización y Unicode.

[Internacionalización](#)



### Autoevaluación

**Mediante las clases que proporcionan buffers se pretende que se hagan lecturas y escrituras físicas a disco, lo antes posible y cuantas más mejor.**

Verdadero. ☐ Falso. ☐

## 4.- Formas de acceso a un fichero.

### Caso práctico



El momento de programar ha llegado, y **Antonio** se pregunta qué opción será mejor para el acceso a los ficheros: si **acceso secuencial o aleatorio**. ¿Qué uso se le va a dar a estos ficheros?, ¿para qué van a servir cuando la aplicación informática esté en funcionamiento? Esa es la cuestión clave, piensa **Antonio**.

Hemos visto que en Java puedes utilizar **dos tipos de ficheros (de texto o binarios) y dos tipos de acceso a los ficheros (secuencial o aleatorio)**. Si bien, y según la literatura que consultemos, a veces se distingue una tercera forma de acceso denominada concatenación, tuberías o pipes.

- ✓ **Acceso aleatorio:** los archivos de acceso aleatorio, al igual que lo que sucede usualmente con la memoria (RAM=Random Access Memory), permiten acceder a los datos en forma no secuencial, desordenada. Esto implica que el archivo debe estar disponible en su totalidad al momento de ser accedido, algo que no siempre es posible.
- ✓ **Acceso secuencial:** En este caso los datos se leen de manera secuencial, desde el comienzo del archivo hasta el final (el cual muchas veces no se conoce a priori). Este es el caso de la lectura del teclado o la escritura en una consola de texto, no se sabe cuándo el operador terminará de escribir.
- ✓ **Concatenación (tuberías o "pipes"):** Muchas veces es útil hacer conexiones entre programas que se ejecutan simultáneamente dentro de una misma máquina, de modo que lo que uno produce se envía por un "tubo" para ser recibido por el otro, que está esperando a la salida del tubo. Las tuberías cumplen esta función.



### Para saber más

En este enlace puedes aprender más sobre tuberías en Java.

[Tuberías](#)

### Citas para pensar

El futuro tiene muchos nombres. Para los débiles es lo inalcanzable. Para los temerosos, lo desconocido. Para los valientes es la oportunidad.

*Víctor Hugo.*

## 4.1.- Operaciones básicas sobre ficheros de acceso secuencial.

Como **operaciones más comunes en ficheros de acceso secuencial**, tenemos el acceso para:

- ✓ Crear un fichero o abrirlo para grabar datos.
- ✓ Leer datos del fichero.
- ✓ Borrar información de un fichero.
- ✓ Copiar datos de un fichero a otro.
- ✓ Búsqueda de información en un fichero.
- ✓ Cerrar un fichero.



Veamos estas operaciones en unos ejemplos comentados.

En este primer ejemplo, vemos cómo se crea el fichero si no existe, o se abre para añadir datos si ya existe:

### Grabar datos en fichero secuencial.

[Resumen textual alternativo](#)

[Grabar en fichero secuencial.](#) (0.01 MB)

En este ejemplo verás cómo podemos **copiar** un archivo origen, a otro destino, desde la línea de comandos.

### Copiar fichero.

[Resumen textual alternativo](#)

[Copiar fichero.](#) (0.01 MB)

En este último ejemplo, has visto cómo usar la clase básica que nos permite utilizar un fichero para escritura de bytes: la clase **FileOutputStream**.

Para mejorar la eficiencia de la aplicación reduciendo el número de accesos a los dispositivos de salida en los que se almacena el fichero, se puede montar un buffer asociado al flujo de tipo **FileOutputStream**. De eso se encarga la clase **BufferedOutputStream**, que permite que la aplicación pueda escribir bytes en el flujo sin que necesariamente haya que llamar al sistema operativo para cada byte escrito.

## Debes conocer

En este enlace puedes ver un ejemplo con buffer.

[Flujos de salida con buffer.](#)

Cuando se trabaja con ficheros de texto se recomienda usar las clases **Reader**, para entrada o lectura de caracteres, y **Writer** para salida o escritura de caracteres. Estas dos clases están optimizadas para trabajar con caracteres y con texto en general, debido a que tienen en cuenta que cada carácter Unicode está representado por dos bytes.

Las subclases de **Writer** y **Reader** que permiten trabajar con ficheros de texto son:

- ✓ **FileReader**, para lectura desde un fichero de texto. Crea un flujo de entrada que trabaja con caracteres en vez de con bytes.
- ✓ **FileWriter**, para escritura hacia un fichero de texto. Crea un flujo de salida que trabaja con caracteres en vez de con bytes.

También se puede montar un buffer sobre cualquiera de los flujos que definen estas clases:

- ✓ **BufferedWriter** se usa para montar un buffer sobre un flujo de salida de tipo **FileWriter**.
- ✓ **BufferedReader** se usa para montar un buffer sobre un flujo de entrada de tipo **FileReader**.





## Autoevaluación

Señala si la afirmación es verdadera o falsa: La clase **Reader** está optimizada para trabajar con ficheros binarios.

Verdadero. ☐ Falso. ☐

## 4.1.1.- Operaciones básicas sobre ficheros de acceso secuencial (II).



Veamos un ejemplo de lectura utilizando un **BufferedReader**.

La clase la puedes descargar de aquí:

[Clase para leer desde fichero con Buffer.](#)

En uno de los ejemplos anteriores, has visto como podemos grabar información a un archivo secuencial, concretamente nombre, apellidos y edad de las personas que vayamos introduciendo.

Ahora vamos a ver cómo **buscar** en un archivo secuencial, usando ese mismo ejemplo. La idea es que al ser un fichero secuencial, tenemos que abrirlo e ir leyendo hasta encontrar el dato que buscamos, si es que lo encontramos.

```
String búsqueda = jTextField1.getText().trim();
try {
    FileReader reader = new FileReader("datos.txt");
    BufferedReader br = new BufferedReader(reader);
    String linea;
    while ((linea = br.readLine()) != null) {
        String[] datos = linea.split(",");
        if (datos.length == 3) {
            String nombre = datos[0].trim();
            String apellidos = datos[1].trim();
            String edad = datos[2].trim();
            if (búsqueda.equals(nombre)) {
                System.out.println("Nombre encontrado: " + nombre);
            }
        }
    }
    System.out.println("No se encontró el nombre.");
} catch (IOException e) {
    e.printStackTrace();
}
```

El proyecto completo lo puedes descargar de aquí:

[Proyecto Java para buscar en fichero secuencial.](#) (0.01 MB)

### Para saber más

Aunque están en inglés, hay muchísimos ejemplos e información sobre la clase **java.io**.

[Java I/O \(Input/Output\)](#)

## 4.2.- Operaciones básicas sobre ficheros de acceso aleatorio.

A menudo, no necesitas leer un fichero de principio a fin, sino simplemente acceder al fichero como si fuera una base de datos, donde se salta de un registro a otro; cada uno en diferentes partes del fichero. Java proporciona una clase **RandomAccessFile** para este tipo de entrada/salida.



Esta clase:

- ✓ Permite leer y escribir sobre el fichero, no es necesario dos clases diferentes.
- ✓ Necesita que le especifiquemos el modo de acceso al construir un objeto de esta clase: sólo lectura o bien lectura y escritura.
- ✓ Posee métodos específicos de desplazamiento como `seek(long posicion)` o `skipBytes(int desplazamiento)` para poder movernos de un registro a otro del fichero, o posicionarnos directamente en una posición concreta del fichero.

Por esas características que presenta la clase, un archivo de acceso directo tiene sus registros de un tamaño fijo o predeterminado de antemano.

La clase posee dos constructores:

- ✓ `RandomAccessFile(File file, String mode).`
- ✓ `RandomAccessFile(String name, String mode).`

En el primer caso se pasa un objeto **File** como primer parámetro, mientras que en el segundo caso es un **String**. El modo es: "r" si se abre en modo lectura o "rw" si se abre en modo lectura y escritura.

A continuación puedes ver una presentación en la que se muestra cómo abrir y escribir en un fichero de acceso aleatorio. También, en el segundo código descargable, se presenta el código correspondiente a la escritura y localización de registros en ficheros de acceso aleatorio.

### Escribir en ficheros de acceso aleatorio.

[Resumen textual alternativo](#)

[Clase para escribir en ficheros de acceso aleatorio.](#)

[Descargar proyecto para escribir y localizar datos en ficheros de acceso aleatorio.](#) (0.01 MB)




## Autoevaluación

**Indica si la afirmación es verdadera o falsa:**

Un objeto de la clase **RandomAccessFile** necesita el modo de acceso al crear el objeto.

Verdadero. ☐ Falso. ☐

## Caso práctico



## 5.1.- Conceptos previos.

¿Cómo se trabaja con datos XML desde el punto de vista del desarrollador de aplicaciones?

Una aplicación que consume información XML debe:

- ✓ Leer un fichero de texto codificado según dicho estándar.
- ✓ Cargar la información en memoria y, desde allí...
- ✓ Procesar esos datos para obtener unos resultados (que posiblemente también almacenará de forma persistente en otro fichero XML).



El proceso anterior se enmarca dentro de lo que se conoce en informática como **"parsing"** o **análisis léxico-sintáctico**, y los programas que lo llevan a cabo se denominan "parsers" o analizadores (léxico-sintácticos). Más específicamente podemos decir que el "parsing XML" es el proceso mediante el cual se lee y se analiza un documento XML para comprobar que está bien formado para, posteriormente, pasar el contenido de ese documento a una aplicación cliente que necesite consumir dicha información.

**Schema:** Un esquema (o schema) es una especificación XML que dicta los componentes permitidos de un documento XML y las relaciones entre los componentes. Por ejemplo, un esquema identifica los elementos que pueden aparecer en un documento XML, en qué orden deben aparecer, qué atributos pueden tener, y qué elementos son subordinados (esto es, son elementos hijos) para otros elementos. Un documento XML no tiene por qué tener un esquema, pero si lo tiene, debe atenerse a ese esquema para ser un documento XML válido.

### Para saber más

En el siguiente enlace de la wikipedia puedes ver el concepto de esquema y algún que otro ejemplo:

[Esquemas XML](#)



### Autoevaluación

**Di si la afirmación es verdadera o falsa:**

XML hace más fácil el intercambio de información entre sistemas.

Verdadero. ☐ Falso. ☐

## 5.2.- Definiciones.



¿Qué es y para qué sirve **JAXB** (Java Architecture for XML Binding)? JAXB simplifica el acceso a documentos XML representando la información obtenida de los documentos XML en un programa en formato Java, o sea, proporciona a los desarrolladores de aplicaciones Java, una forma rápida para vincular esquemas XML a representaciones Java.

JAXB proporciona métodos para, a partir de documentos XML, obtener árboles de contenido (generados en código Java), para después operar con ellos o manipular los mismos en una aplicación Java y generar documentos XML con la estructura de los iniciales, pero ya modificados.

**Parsear** un documento XML consiste en "escanear" el documento y dividirlo o separarlo lógicamente en piezas discretas. El contenido parseado está entonces disponible para la aplicación.

**Binding:** Binding o vincular un esquema (schema) significa generar un conjunto de clases Java que representan el esquema.

**Compilador de esquema** o schema compiler: liga un esquema fuente a un conjunto de elementos de programa derivados. La vinculación se describe mediante un lenguaje de vinculación basado en XML.

>**Binding runtime framework:** proporciona operaciones de unmarshalling y marshalling para acceder, manipular y validar contenido XML usando un esquema derivado o elementos de programa.

**Marshalling:** es un proceso de codificación de un objeto en un medio de almacenamiento, normalmente un fichero. Proporciona a una aplicación cliente la capacidad para convertir un árbol de objetos Java JAXB a ficheros XML. Por defecto, el marshaller usa codificación UTF-8 cuando genera los datos XML.

**Unmarshalling:** proporciona a una aplicación cliente la capacidad de convertir datos XML a objetos Java JAXB derivados.

### Para saber más

Hay muchos "parsers" conocidos, como **SAX** (Simple API for XML). SAX es un **API** para parsear ficheros XML. Proporciona un mecanismo para leer datos de un documento XML. Otra alternativa es **DOM** (Document Object Model).

Tienes más información sobre DOM en la wikipedia:

[DOM](#)

En este enlace se ve un ejemplo de cómo parsear un fichero XML mediante SAX.

**Parsear**

[Resumen textual alternativo](#)

## 5.3.- Introducción a JAXB.

JAXB permite mapear clases Java a representaciones en XML y viceversa.

JAXB proporciona dos principales características:

- ✓ La capacidad de serializar (marshalling) objetos Java a XML.
- ✓ Lo inverso, es decir, deserializar (unmarshalling) XML a objetos Java.



O sea que **JAXB permite almacenar y recuperar datos en memoria en cualquier formato XML, sin la necesidad de implementar un conjunto específico de rutinas XML** de carga y salvaguarda para la estructura de clases del programa.

El compilador de JAXB (schema compiler) permite generar una serie de clases Java que podrán ser llamadas desde nuestras aplicaciones a través de métodos sets y gets para obtener o establecer los datos de un documento XML.

El **funcionamiento esquemático** al usar JAXB sería:

- ✓ Crear un esquema (fichero .xsd) que contendrá la estructura de las clases que deseamos utilizar.
- ✓ Compilar con el JAXB compiler (bien con un IDE como NetBeans o desde línea de comandos con el comando **xjc**) ese fichero .xsd, de modo que nos producirá los **POJOs**, o sea, una clase por cada uno de los tipos que hayamos especificado en el fichero .xsd. Esto nos producirá los ficheros .java.
- ✓ Compilar esas clases java.
- ✓ Crear un documento XML: validado por su correspondiente esquema XML, o sea el fichero .xsd, se crea un árbol de objetos.
- ✓ Ahora se puede parsear el documento XML, accediendo a los métodos gets y sets del árbol de objetos generados por el proceso anterior. Así se podrá modificar o añadir datos.
- ✓ Después de realizar los cambios que se estimen, se realiza un proceso para sobrescribir el documento XML o crear un nuevo documento XML.



### Autoevaluación

**Un schema permite validar un documento XML.**

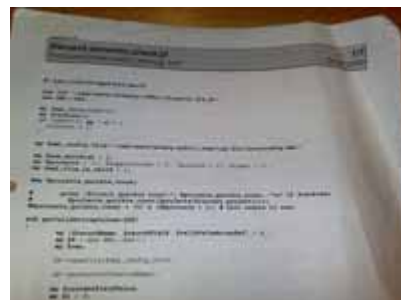
Verdadero. ☐ Falso. ☐



## 5.4.- Funcionamiento de JAXB.

Para construir una aplicación JAXB necesitamos tener un esquema XML.

Tras obtener el esquema XML, seguimos los **siguientes pasos para construir la aplicación JAXB**:



1. **Escribir el esquema:** es un documento XML que contiene la estructura que se tomará como indicaciones para construir las clases. Estas indicaciones pueden ser, por ejemplo, el tipo primitivo al que se debe unir un valor de atributo en la clase generada.
2. **Generar los ficheros fuente de Java:** para esto usamos el compilador de esquema, ya que éste toma el esquema como entrada de información. Cuando se haya compilado el código fuente, podremos escribir una aplicación basada en las clases que resulten.
3. **Construir el árbol de objetos Java:** con nuestra aplicación, se genera el árbol de objetos java, también llamado árbol de contenido, que representa los datos XML que son validados con el esquema. Hay dos formas de hacer esto:
  - a. **Instanciando** las clases generadas.
  - b. Invocando al método `unmarshall` de una clase generada y pasarlo en el documento. El método `unmarshall` toma un documento XML válido y construye una representación de árbol de objetos.
4. **Acceder al árbol de contenido** usando nuestra aplicación: ahora podemos acceder al árbol de contenido y modificar sus datos.
5. **Generar un documento XML** desde el árbol de contenido. Para poder hacerlo tenemos que invocar al método `marshall` sobre el objeto raíz del árbol.

Aunque estos pasos que acabamos de comentarte te parezcan algo complicados, vamos a ver un ejemplo sencillo, en el que clarificaremos todo esto, comprobando que no es tan difícil como parece.

Vamos a suponer una estructura de un archivo que podría usarse en un almacén distribuidor de medicamentos. El fichero **albaran.xsd** tiene la estructura que el almacén usa cuando envía un pedido de medicamentos que le ha hecho una farmacia.

Puedes descargarlo para examinarlo aquí:

[Fichero albaran.xsd](#) (1 KB)

Vamos a ver el desarrollo del proyecto en la siguiente presentación:

### Ejemplo de proyecto sencillo con JAXB.

[Resumen textual alternativo](#)

El proyecto resultante tras seguir los pasos de la presentación se encuentra aquí:

[Proyecto JAXB](#) (0.03 MB)

## Para saber más

En el siguiente enlace puedes ver un ejemplo paso a paso en el que:

- ✓ Se crea el fichero de esquema XML.
- ✓ Se crea un proyecto nuevo con NetBeans.
- ✓ Se añade un JAXB Binding utilizando el fichero XSD creado anteriormente.
- ✓ Se añade un servicio web y se utiliza las clases Java JAXB Binding como un tipo de objeto.

[Ejemplo con JAXB](#)

## 6.- Librerías para conversión de documentos XML a otros formatos.

### Caso práctico

**María** le dice a **Juan** que una buena práctica para **Antonio** y **Ana** sería que hicieran los informes que necesitan para la aplicación de la farmacia.

-Tan solo tenemos que instruirles un poco en JasperReport e indicarles unos cuantos tutoriales, seguro que lo hacen bien -le comenta **María** a **Juan**.

**María** decide, además, consultar a su amiga **Blanca**, que es experta en JasperReport y trabaja en otra empresa, qué tutoriales de Internet o de cualquier otra fuente recomendaría a unos principiantes en esta materia.



En la mayoría de aplicaciones informáticas, hay que mostrar la información resultante de los procesos que se ejecutan, sobre todo en aplicaciones que generan información que implica tomar decisiones comerciales. Dicha información está almacenada normalmente en bases de datos o en archivos.

Hoy en día, XML está muy extendido, y muchas empresas guardan la información en ficheros o bases de datos con ese formato.

Hay muchos productos o herramientas informáticas que permiten convertir documentos XML a otros formatos.



En nuestro caso, vamos a optar por una herramienta que permite generar informes de todo tipo en Java de una forma sencilla: JasperReport.

Esta herramienta permite generar informes electrónicos en formato pdf, quizás el formato más usado debido a su portabilidad entre sistemas conservando la apariencia. Pero existen muchos más: xls, html, rtf, csv, xml, etc.

### Para saber más

En el siguiente enlace puedes ver un tutorial de otra herramienta de conversión de XML a otros formatos.

[XSL](#)



### Autoevaluación

JasperReport permite generar documentos en formato html.

Verdadero. ☐ Falso. ☐

## 6.1.- Introducción a JasperReport.

### JasperReports

En Java, durante un tiempo, la generación de informes fue uno de los puntos débiles del lenguaje, pero hoy en día, existen muchas librerías y herramientas dedicadas (varias de ellas, de [código abierto](#)) para la rápida generación de informes. JasperReports, es una de las más conocidas.

JasperReports es una herramienta que consta de un poderoso motor para la generación de informes. Está empaquetada en un archivo JAR y puede ser utilizada como una librería, la cuál podemos integrar en cualquier IDE de desarrollo en Java para desarrollar nuestras aplicaciones. Está escrita totalmente en Java, su código es abierto y es totalmente gratuita bajo los términos de la licencia GPL (Licencia Pública General).

Si visitas el siguiente enlace podrás acceder a la página de descarga de JasperReports para todas las plataformas. Encontrarás una lista con todas las versiones disponibles, no es necesario identificarte para poder bajarte la que desees.

[Zona de descarga de JasperReports](#)

La descarga y el contenido de lo descargado lo podemos ver en la siguiente presentación:

### Debes conocer

Descarga de JasperReports e integración en NetBeans.

[Resumen textual alternativo](#)

En la presentación que acabas de ver, al descomprimir el fichero de la descarga, has visto que en el mismo hay varios directorios o carpetas. Comentamos brevemente qué contiene cada una:

- ✓ **build:** es la librería JasperReports sin empaquetar, con todas las clases que incluye.
- ✓ **demo:** podemos encontrar algunos ejemplos de utilización de la librería. Estos ejemplos están preparados para ser compilados con la herramienta "**ant**". Puedes inspeccionar el código Java e intentar compilarlos y ejecutarlos.

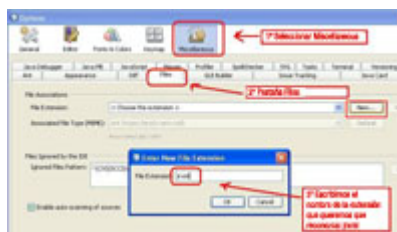
- ✓ **dist:** es donde se encuentra realmente la librería empaquetada en un fichero JAR (`jasperreports-3.7.4.jar`) y algunos ficheros JAR que no utilizaremos. También podemos acceder a la documentación tipo javadoc.
- ✓ **docs:** es la referencia rápida en formato XML.
- ✓ **lib:** Diferentes librerías necesarias por JasperReports, como algunas para exportar a distintos formatos, para incluir gráficos, etc.
- ✓ **src:** Ficheros fuente de la librería.

## Para saber más

En este enlace tienes la documentación en línea de la API de JasperReports.

[API de JasperReports](#)

## 6.2.- Diseñar y compilar la plantilla.



Las plantillas de los informes de JasperReports son sencillamente ficheros XML con la extensión .jrxml. Podemos hacer que NetBeans reconozca este tipo de ficheros como XML, para que cuando los editemos en el editor se muestren los mismos códigos de colores en las etiquetas y demás elementos de la sintaxis de XML.

En la imagen se ilustra cómo conseguirlo: en NetBeans pinchamos en el menú Tools, y ahí en Options. Ahí seleccionamos Miscellaneous, luego la pestaña Files. Entonces pulsamos en el botón New... para añadir la

nueva extensión.

Los pasos a seguir para trabajar con JasperReport serían:

**Paso 1: Diseñar la plantilla del informe:** un fichero .jrxml. El documento de diseño está representado por un archivo XML que mantiene la estructura de un archivo **DTD** (Document Type Definition) definido por el motor de JasperReports.

La generación de un diseño implica editar un archivo XML validado mediante:

```
<!DOCTYPE jasperReport PUBLIC "-//JasperReports//DTD Report Design//EN" "http://jasperreports
```

Estos documentos XML cuentan con una estructura similar a la de cualquier documento de texto. Fundamentalmente se siguen estas secciones:

- ✓ title Título del informe.
- ✓ pageHeader Encabezado del documento.
- ✓ columnHeader Encabezado de las columnas.
- ✓ detail Detalle del documento. Cuerpo
- ✓ columnFooter Pie de la columna.
- ✓ pageFooter Pie del documento.
- ✓ sumary Cierre del documento.

**Paso 2: Compilación:** Una vez que se ha realizado el diseño, se compila antes de poder iniciar el proceso de carga de datos. La compilación se lleva a cabo a través del método **compileReport()**.

En este proceso, el diseño se transforma en un objeto serializable de tipo **net.sf.jasperreports.engine JasperReport**, que luego se guarda en disco.



### Autoevaluación

¿Es correcta la afirmación siguiente?

Los documentos XML que se usan en el diseño de los informes cuentan con una serie de secciones.

Verdadero. ☐ Falso. ☐

## 6.3.- Rellenar el informe con datos, exportar el informe.

**Paso 3: Rellenar el informe con datos:** mediante los métodos `fillReportXXX()`, se puede realizar la carga de datos del informe, pasándole como parámetros el objeto de diseño (o bien, el archivo que lo representa en formato [serializado](#)) y la conexión `JDBC` a la base de datos desde donde se obtendrá la información que necesitamos.

Como resultado de este proceso, se obtiene un objeto que representa un documento listo para ser impreso, un objeto serializable de tipo **JasperPrint**. Este objeto puede guardarse en disco para su uso posterior, o bien puede ser impreso, enviado a la pantalla o transformado en PDF, XLS, CSV, etc.

### Paso 4: Visualización

Ahora podemos optar por mostrar un informe por pantalla, imprimirlo, o bien obtenerlo en algún tipo específico de fichero, como PDF, etc.

- ✔ Para mostrar un informe por pantalla se utiliza la clase `JasperViewer`, la cual, a través de su método `main()`, recibe el informe a mostrar.
- ✔ Para imprimir el informe usaremos los métodos `printReport()`, `printPage()` o `printPages()`, contenidos en la clase `JasperPrintManager`.
- ✔ Para exportar los datos a un formato de archivo específico podemos utilizar los métodos `exportReportXXX()`.

Vamos a construir un ejemplo comentado en la siguiente presentación:

### Ejemplo con JasperReports.



[Resumen textual alternativo](#)

Seguimos desarrollando el ejemplo. Ahora vamos a hacer que obtenga datos de una base de datos. En concreto, de la base de datos derby que se incluye al instalar el `jdk`.

### Seguimos con con JasperReports.

[Resumen textual alternativo](#)

## Para saber más

El principal inconveniente que puedes encontrarte al trabajar con JasperReports sin más, es sin duda el diseño del informe. Por ello, para facilitar el diseño de los mismos, y hacerlos de manera visual y cómoda se pueden usar otros productos como iReport, que es también una herramienta de software libre.

Hay mucha documentación sobre iReport en la red, aquí te adjuntamos dos:

[Tutorial iReport](#)

[Crear informes con iReport](#)



## Anexo I.- Listar ficheros de una carpeta, filtrando.

---

```
import java.io.File;
import java.io FilenameFilter;

public class Filtrar implements FilenameFilter {
    String extension;
    // Constructor
    Filtrar(String extension){
        this.extension = extension;
    }
    public boolean accept(File dir, String name){
        return name.endsWith(extension);
    }
    public static void main(String[] args) {
        try {
            // Obtendremos el listado de los archivos de ese directorio
            File fichero=new File("c:\\datos\\.");
            String[] listadeArchivos = fichero.list();
            // Filtraremos por los de extension .txt
            listadeArchivos = fichero.list(new Filtrar(".txt"));
            // Comprobamos el número de archivos en el listado
            int numarchivos = listadeArchivos.length ;
            // Si no hay ninguno lo avisamos por consola
            if (numarchivos < 1)
                System.out.println("No hay archivos que listar");
            // Y si hay, escribimos su nombre por consola.
            else
            {
                for(int conta = 0; conta < listadeArchivos.length;
                    conta++)
                    System.out.println(listadeArchivos[conta]);
            }
        }
        catch (Exception ex) {
            System.out.println("Error al buscar en la ruta indicada");
        }
    }
}
```

## Anexo II.- Código de separador de rutas.

---

```
String substFileSeparator(String ruta){
    String separador = "\\";
        try{
            // Si estamos en Windows
                if ( File.separator.equals(separador) )
                    separador = "/" ;
            // Reemplaza todas las cadenas que coinciden con la expresión
            // regular dada oldSep por la cadena File.separator
            return ruta.replaceAll(separador, File.separator);
        }catch(Exception e){
            // Por si ocurre una java.util.regex.PatternSyntaxException
            return ruta.replaceAll(separador + separador, File.separator);
        }
    }
```

## Anexo III.- Código de crear un fichero.

---

```
try {  
    // Creamos el objeto que encapsula el fichero  
    File fichero = new File("c:\\prufba\\miFichero.txt");  
    // A partir del objeto File creamos el fichero físicamente  
    if (fichero.createNewFile())  
        System.out.println("El fichero se ha creado correctamente");  
    else  
        System.out.println("No ha podido ser creado el fichero");  
} catch (Exception ioe) {  
    ioe.getMessage();  
}
```



## Anexo IV.- Código de crear un directorio.



---

```
try {  
    // Declaración de variables  
    String directorio = "C:\\\\prueba";  
    String varios = "carpeta1/carpeta2/carpeta3";  
  
    // Crear un directorio  
    boolean exito = (new File(directorio)).mkdir();  
    if (exito)  
        System.out.println("Directorio: " + directorio + " creado");  
    // Crear varios directorios  
    exito = (new File(varios)).mkdirs();  
    if (exito)  
        System.out.println("Directorios: " + varios + " creados");  
}catch (Exception e){  
    System.err.println("Error: " + e.getMessage());  
}
```

## Anexo.- Licencias de recursos.

### Licencias de recursos utilizados en la Unidad de

Recurso (1)	Datos del recurso (1)	Recurso (2)	
	Autoría: dado83. Licencia: CC-by-nc. Procedencia: <a href="http://www.flickr.com/photos/dado83/3406962115/">http://www.flickr.com/photos/dado83/3406962115/</a>		Auto Licer Proc <a href="http://www.flickr.com/photos/dado83/3406962115/">http://www.flickr.com/photos/dado83/3406962115/</a>
	Autoría: Art3mis4. Licencia: CC-by-nc-sa. Procedencia: <a href="http://www.flickr.com/photos/art3mis4/4910243349/">http://www.flickr.com/photos/art3mis4/4910243349/</a>		Auto Licer Proc de M
	Autoría: Kasaa. Licencia: CC-by-nc. Procedencia: <a href="http://www.flickr.com/photos/kasaa/2693784352/">http://www.flickr.com/photos/kasaa/2693784352/</a>		Auto Licer Proc
	Autoría: d\$мд. Licencia: CC-by-nc. Procedencia: <a href="http://www.flickr.com/photos/asmamirza/2599581983/">http://www.flickr.com/photos/asmamirza/2599581983/</a>		Auto Licer Proc <a href="http://www.flickr.com/photos/asmamirza/2599581983/">http://www.flickr.com/photos/asmamirza/2599581983/</a>
	Autoría: Ornellaswouldgo. Licencia: CC-by. Procedencia: <a href="http://www.flickr.com/photos/ornellas/4257487503/">http://www.flickr.com/photos/ornellas/4257487503/</a>		Auto Licer Proc is/71
	Autoría: aldoaloz. Licencia: CC-by-nc-sa. Procedencia: <a href="http://www.flickr.com/photos/aldoaloz/3895614433/#/">http://www.flickr.com/photos/aldoaloz/3895614433/#/</a>		Auto Licer Proc <a href="http://www.flickr.com/photos/aldoaloz/3895614433/#/">http://www.flickr.com/photos/aldoaloz/3895614433/#/</a>
	Autoría: Fartese. Licencia: CC-by-sa. Procedencia: <a href="http://www.flickr.com/photos/fartese/4214174953/">http://www.flickr.com/photos/fartese/4214174953/</a>		Auto Licer Proc
	Autoría: Roland. Licencia: CC-by. Procedencia: <a href="http://www.flickr.com/photos/roland/4358742850/">http://www.flickr.com/photos/roland/4358742850/</a>		Auto Licer Proc boy/

Recurso (1)	Datos del recurso (1)	Recurso (2)	
	<p>Autoría: José Javier Bermúdez Hernández. Licencia: GNU GPL v2. Procedencia: Montaje sobre Captura de pantalla del programa NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.</p>		Auto Licer Proc