Resumen del tema 2.

# 1.- Introducción.

Ya sabes que cuando apagas el ordenador, los datos de la memoria RAM se pierden. Un ordenador utiliza ficheros para guardar los datos, 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. A las **operaciones**, que constituyen un **flujo de información del programa** con el exterior, se les conoce como **Entrada/Salida (E/S)**.

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**.

**java.nio (Non- Blocking I/O)** fue introducido en el API de Java desde la versión 1.4 como extensión eficiente a los paquetes **java.io** y **java.net**. Java NIO ofrece una forma diferente de trabajar con IO que las API de IO estándar. Se basa en el “**Buffer**” y el “**Channel**”.

java.io vs java.nio:

* Depende de lo que necesitemos **pueden ser complementarios** y utilizaremos conjuntamente ambos paquetes.
* **java.nio** permite manejar múltiples canales (archivos o conexiones de red) con uno o unos pocos hilos.
* En **java.nio** el **procesamiento de datos** es más **complicado** que usar los streams bloqueantes de java.io
* **java.nio** es la opción si necesito **manejar cientos de conexiones** (canales) abiertas y en cada una manejar una **pequeña cantidad de datos**.
* **java.io** es la opción si voy a manejar **pocas conexiones** con un **alto ancho de banda** (envío mucha información a la vez).
* **Clase File**:
  + Hasta JSE7 **java.io.File** era la clase utilizada para realizar operaciones I/O con archivos. Esta clase tenía limitaciones como:
    - **Muchos métodos no lanzaban excepciones** por lo que era muy complicado detectar y resolver errores.
    - **No permitía manejar enlaces simbólicos**.
    - **Presenta problemas de escalabilidad**. Directorios largos provocaban problemas de memoria e incluso de denegación de servicio.
  + **java.nio.file** (incorporado a partir de JSE7) resuelve estos problemas.
    - **Es el que debemos usar** para trabajar con archivos independientemente de si realizamos I/O con streams (java.io) ó con buffers y channels (java.nio).

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

Vamos a ver la clase **File** que nos permite hacer unas cuantas operaciones con ficheros, también veremos cómo filtrar ficheros (por ejemplo, que tengan una determinada extensión). Por último, veremos como crear y eliminar ficheros y directorios.

## 2.1.- Clase File.

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. Si no existen, no se lanzará ningún tipo de excepción ni tampoco serán creados al crearlos con cualquiera de sus constructores.

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** al constructor **FileWriter(File f)**, que recibe como parámetro un objeto **File**.

Los **constructores de la clase File** son:

* public File(String nombreFichero|path);
* public File(String path, String nombreFichero|path);
* public File(File path, String nombreFichero|path);

La ruta o **path** puede ser absoluta o relativa. **Ejemplos de rutas**:

* **"personas.dat"**
* **"ficheros/personas.dat"**
* **"c:/ficheros/personas.dat"**

**Ejemplos de uso de uso del resto de constructores**:

* **Segundo constructor: File f = new File("ficheros", "personas.dat" );** -> indica la ruta en el primer parámetro, y el nombre del archivo en el segundo.
* **Tercer constructor**:  
  **File ruta = new File("ficheros");**  
  **File f = new File(ruta, "personas.dat" );**

Dado un objeto File, podemos hacer las siguientes **operaciones** con él:

|  |  |
| --- | --- |
| **Operación** | **Lo que hace** |
| Boolean canRead() | Devuelve true **si se puede leer el fichero** |
| Boolean canWrite() | Devuelve true **si se puede escribir en el fichero** |
| Boolean createnewFile() | **Crea el fichero**. Devuelve true si se ha podido crear. Para poder crearlo, el fichero no debe existir. Lanza una IOException si algo va mal. |
| Boolean delete() | **Elimina** el fichero o directorio. Si es un directorio, debe estar vacío. |
| Boolean exists() | Devuelve true si el fichero o directorio **existe**. |
| String getName() | Devuelve el **nombre del fichero o directorio**. |
| String getAbsolutePath() | Devuelve la **ruta absoluta** del fichero o directorio. |
| String getCanonicalPath() | Devuelve la **única ruta absoluta asociada al archivo o directorio**. Puede haber varias rutas absolutas asociadas a un File pero solo una única ruta canónica. Lanza una excepción del tipo **IOException**. |
| String getPath() | Devuelve la **ruta con la que se creó el objeto File**. Puede ser relativa o no. |
| String getParent() | Devuelve un **String** conteniendo el **directorio padre del File**. Devuelve null si no tiene directorio padre. |
| File getParentFile() | Devuelve un **objeto File** conteniendo el **directorio padre del File**. Devuelve null si no tiene directorio padre. |
| Boolean isAbsolute() | Devuelve true **si es una ruta absoluta**. |
| Boolean isDirectory() | Devuelve ture **si es un directorio válido**. |
| Boolean isFile() | Devuelve true **si es un fichero válido**. |
| Long lastModified() | Devuelve un **valor en milisegundos** que representa la **última vez que se ha modificado** (medido desde las 00:00:00 GMT, del 1 de Enero de 1970). Devuelve 0 si el fichero no existe o ha ocurrido un error. |
| Long length() | Devuelve el **tamaño en bytes del fichero**. Devuelve 0 si no existe. Devuelve un valor indeterminado si es un directorio. |
| String[] list() | Devuelve un **array** de String con el **nombre de los archivos y directorios que contiene el directorio indicado en el objeto File**. Si no es un directorio devuelve null. Si el directorio está vacío devuelve un array vacío. |
| String[] list(FilenameFilter filtro) | **Similar al anterior**. Devuelve un array de String con el nombre de los archivos y directorios que contiene el directorio indicado en el objeto File **que cumplen con el filtro indicado**. |
| Boolean mkdir() | **Crea el directorio**. Devuelve true si se ha podido crear. |
| Boolean mkdirs() | **Crea el directorio incluyendo los directorios no existentes** especificados en la ruta *padre* del directorio a crear. Devuelve true si se ha creado el directorio y los directorios no existentes de la ruta padre. |
| Boolean renameTo(File dest) | **Cambia el nombre del fichero** por el indicado en el parámetro dest. Devuelve true si se ha realizado el cambio. |

Texto

Descripción generada automáticamente

### 2.1.1.- Existencia y listado de ficheros y carpetas.

|  |  |
| --- | --- |
| **Existencia de fichero** | File f = new File("C:\texto.txt");  f.exists() -> método para comprobar si existe un fichero o carpeta. |
| **Listado de ficheros** | File ruta = new File(jTextField1.getText());  if(ruta.exists()) {  // Obtenemos la lista de archivos  String[] listaArchivos = ruta.list();  // La cargamos en el JList  for(int i=0; i<listaArchivos.length; i++){  System.out.println(listaArchivos[i]);  }  } |

Sin embargo, para copiar un fichero, no nos basta con la clase **File**. Necesitamos saber cómo funcionan los flujos.

**Para saber más**: [Creación y uso de ficheros Properties con Java – Apuntes de Programación (archive.org)](http://web.archive.org/web/20190807160526/http:/programacion.jias.es/2011/10/creacion-de-ficheros-properties-con-java/), [mkdir examples : File Commands « File Input Output « Java (java2s.com)](http://www.java2s.com/Code/Java/File-Input-Output/mkdirexamples.htm)

Texto

Descripción generada automáticamente

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

|  |  |
| --- | --- |
| **Crear ficheros** | try {  // Creamos el objeto que encapsula el fichero  File fichero = new File("c:\\prueba\\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();  } |
| **Eliminar ficheros** | // Borrar fichero  File fichero = new File("c:\\prueba\\miFichero.txt");  if (fichero.exists ())  fichero.delete(); |
| **Crear directorios** | 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());  } |
| **Borrar directorio** | Para borrar un directorio, si posee ficheros, deberemos borrar cada uno de los ficheros y directorios que este contenga. Por ello, se podría recorrer recursivamente el directorio para ir borrando todos los ficheros.  **Por ejemplo**: listamos con:  File [] ficheros = directorio.listFiles ();  Y borramos cada uno. Comprobamos con **isDirectory()** si el elemento es un directorio.  Después de vaciar el directorio, este, se puede borrar importando la librería **FileUtils** y después escribir **FileUtils.deleteDirectory(newFile(destination))**; |

## 2.2.- Interface FilenameFilter.

A veces nos puede interesar ver 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)

Devuelve verdadero si el fichero indicado en el parámetro ***nombre*** aparece en a lista de ficheros del directorio indicado por el parámetro ***dir***.

**Ejemplo de cómo listar ficheros que tengan una extensión determinada**:

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");

}

}

}

En el ejemplo anterior, se utiliza la función **endsWith**. Se usa para el tratamiento de cadenas, en concreto, para comprobar si acaba en cierto String. Más info: [String (Java Platform SE 7 ) (oracle.com)](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html)

**Para saber más**: sobre las [operaciones con cadenas](http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/clases1/string.htm).

## 2.3.- Rutas de los ficheros.

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**.

public class OperacionesConFicheros {

static String substFileSeparator(String ruta) {

String separador = "\\";

try {

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);

}

}

}

Texto

Descripción generada automáticamente

# 3.- Flujos.

**Caso práctico**: Hay dos tipos de flujos: flujos de caracteres y flujos de bytes.

Para realizar cualquier operación de entrada/salida (en adelante E/S), lo haremos mediante un flujo o ***stream***.

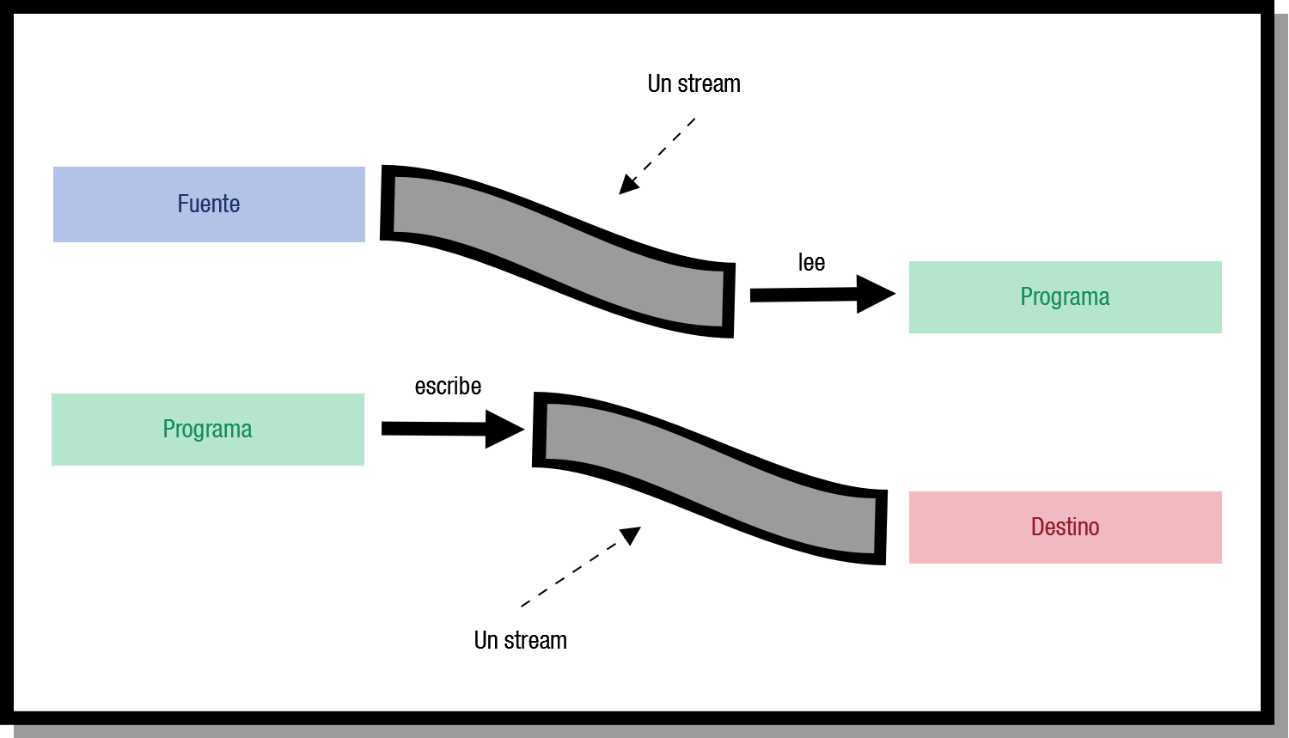
Un **flujo** es una **abstracción de todo aquello que produce o consume información**.

**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 valen para cualquier dispositivo con el que estemos actuando, sea el teclado, el monitor, un sistema de archivos, impresora, o un socket en red. De esta forma, Java libera 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)**: **lectura y escritura de datos binarios**.
  + **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)**: lectura y escritura de caracteres.
  + **Reader** y **Writer**. Dichas clases manejan flujos de caracteres Unicode. También hay clases específicas que derivan de estas. Destacan los métodos **read()** y **write()** que leen y escriben caracteres de datos respectivamente.



## 3.1.- Flujos basados en bytes.

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

Clases principales que heredan de ***OutputStream*** para **escribir ficheros binarios**:

* **FileOutputStream**: escribe bytes en un fichero.
  + **FileOutputStream(String filePath, boolean append)** es el constructor.
    - Si append es true, el contenido se añadirá al final del archivo, de lo contrario, se “machacará” el contenido con el nuevo.
* **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.
* **DataInputStream**.
* **ObjectInputStream**: convierte en objetos y variables los vectores de bytes leídos de un InputStream.

**Por ejemplo**: La escritura de un fichero Binario se hace así:

DataOutputStream fileOut = new DataOutputStream (new FileOutputStream(filename));

fileOut.writeUTF(“el texto que sea”); -> para escribir texto.

fileOut.writeInt(5); -> para escribir números.

**Por ejemplo 2**: escritura a fichero binario.

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

public class CopiaFicheros {

public static void main(String args[]) {

// Copiar ficheros

File origen = new File("origen.txt");

File destino = new File("destino.txt");

try {

InputStream in = new FileInputStream(origen);

OutputStream out = new FileOutputStream(destino);

byte[] buf = new byte[1024];

int len;

while ((len = in.read(buf)) > 0) {

out.write(buf, 0, len);

}

in.close();

out.close();

} catch (IOException ioe) {

ioe.printStackTrace();

}

}

}

<https://chat.openai.com/share/89cefadb-0bb4-44e9-989b-65b73298959a>

Texto

Descripción generada automáticamente

## 3.2.- Flujos basados en caracteres.

Si se leen o escriben pocos caracteres cada vez, el proceso se hace costoso y lento por los muchos accesos a disco duro.

Java proporciona dos **clases abstractas para los flujos de caracteres**. Reader y Writer. Estas solucionan los problemas de FileReader, FileWriter, FileInputStream y FileOutputStream

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.

A diferencia de FileReader, **BufferedReader** sí posee **métodos** que nos permiten **leer líneas completas**.

Afortunadamente, **podemos construir un BufferedReader a partir del FileReader** de la siguiente manera:

File archivo = new File ("C:\\archivo.txt");

FileReader fr = new FileReader (archivo);

BufferedReader br = new BufferedReader(fr);

...

String linea = br.readLine();

**Ejemplo: lectura de un fichero**: en este ejemplo leemos caracteres hasta llegar al fin de archivo (EOF, o End Of File) del flujo.

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

public class LeerFichEOF {

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

BufferedReader br = new BufferedReader(new FileReader("origen.txt")) ;

int codigo = br.read();//lee el primer caracter

char caracter;

//mientras el código no sea -1 (EOF) continuo leyendo

while (codigo != -1) {

caracter = (char) codigo; //casting

System.out.print(caracter);

codigo = br.read(); //lee un caracter

}

}

}

**Debes conocer**: Vídeo sobre el paquete java.io. [Conoce mejor un paquete: java.io y escritura en ficheros - YouTube](https://www.youtube.com/watch?v=-1C-wVnCd3c)

**Para saber más**: [sobre internacionalización y Unicode](https://developers-latam.googleblog.com/2008/04/internacionalizacin-i-unicode.html).

Este documento habla de:

* **Internacionalización I - Unicode**: Un artículo que explica los conceptos básicos de los juegos de caracteres, las codificaciones y el estándar Unicode para el desarrollo de aplicaciones web multilingües.
* **ISO-8859 y UTF-16**: Dos familias de juegos de caracteres que usan uno o dos bytes por carácter, respectivamente, y que cubren algunos idiomas, pero no todos. Tienen problemas de compatibilidad con el ASCII y el C.
* **UTF-8**: Una codificación de tamaño variable que usa entre uno y cuatro bytes por carácter, y que es compatible con el ASCII y el C. Es la recomendación del autor para guardar los textos en la base de datos y en la interfaz web.
* **Consejos prácticos**: El autor ofrece algunos consejos para evitar errores comunes al usar UTF-8, como el BOM, los modificadores y el atributo “lang”. También hace publicidad de Google como un lugar de trabajo genial.

Texto

Descripción generada automáticamente

# 4.- Formas de acceso a un fichero.

En Java podemos utilizar dos tipos de ficheros (de texto o binarios) y dos tipos de acceso a los ficheros (secuencial o aleatorio). Según donde consultemos, a veces se distingue una tercera forma de acceso denominada concatenación, tuberías o pipes.

* **Acceso aleatorio**: **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**: los datos **se leen de manera secuencial**, desde el comienzo del archivo hasta el final (el cual muchas veces no se conoce a priori).
  + **Por ejemplo**: 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.

## 4.1.- Operaciones básicas sobre ficheros de acceso secuencial (buffers).

Operaciones típicas de acceso secuencial en ficheros:

* **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.

**Debes conocer**: la idea de la lectura secuencial de ficheros secuenciales es que tenemos que abrirlo e ir leyendo hasta encontrar el dato que buscamos, si es que lo encontramos.

Scanner sc = new Scanner(System.in);

String busqueda = sc.nextLine();

try {

// Declarar variable

DataInputStream archivo = null;

// Abrir el archivo

archivo = new DataInputStream ( new FileInputStream ("c:\\secuencial.dar") );

// Leer archivo

while (seguir) {

// Leer el nombre

nombre = archivo.readUTF();

// Si el nombre es el que buscamos

if (busqueda.equals (nombre) {

System.out.println ("encontrado");

seguir = false;

}

// Leer los otros campos

apellidos = archivo.readUTF();

edad = archivo.readInt();

}

// Cerrar fichero

archivo.close();

}

catch (FileNotFoundException fne) {

/ \* Archivo no encontrado \* /

}

catch (IOException ioe) {

/ \* Error al escribir \* /

}

Cuando se trabaja con ficheros de texto, se recomienda usar las clases **Reader** (para leer caracteres) y **Writer** (para escribir caracteres), ya que están **optimizadas específicamente para leer y escribir caracteres y texto en general**, y tienen en cuenta que cada carácter Unicode está representado por 2 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.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

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

Es como si fuera una base de datos, donde se salta de un registro a otro.

Java nos proporciona una clase ***RandomAccessFile*** para este tipo de entrada/salida. **Características**:

* 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 **para poder movernos** de un registro a otro del fichero, o **posicionarnos** directamente en una posición concreta del fichero.
  + **seek**(long posicion)
  + **skipBytes**(int desplazamiento)

Por esas características que presenta la clase, **un archivo de acceso directo** (o aleatorio, como se quiera llamar) tiene sus registros de un tamaño fijo o predeterminado de antemano.

**Constructores**:

* RandomAccessFile(File file, String mode).
* RandomAccessFile(String name, String mode).

En ambos casos, el modo se especifica de la siguiente manera:

* “r”: solo lectura
* “rw”: lectura y escritura

**Ejemplo**: Vamos a ver un pequeño ejemplo, **Log.java**, que añade una cadena a un fichero existente. Lo crearía en caso de no existir.

import java.io.IOException;

import java.io.RandomAccessFile;

public class RandomEjemplo {

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

RandomAccessFile miRAFile;

String s = "linea a añadir al final del fichero";

// Abrimos el fichero de acceso aleatorio

miRAFile = new RandomAccessFile( "java.log","rw" );

// Nos vamos al final del fichero

miRAFile.seek( miRAFile.length() );

// Incorporamos la cadena al fichero

miRAFile.writeBytes( s );

// Cerramos el fichero

miRAFile.close();

}

}

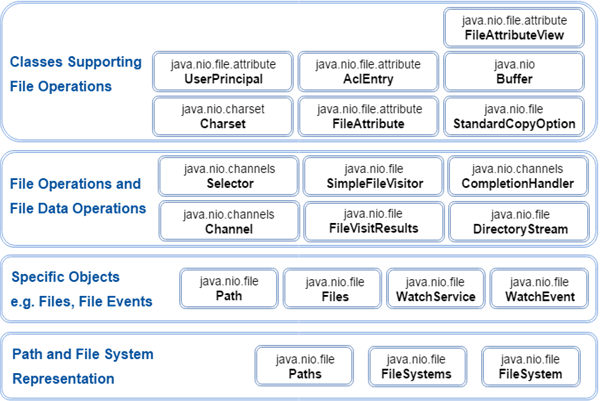
Texto

Descripción generada automáticamente

## 4.3.- Enlaces de interés sobre ficheros.

* [Lesson: Basic I/O (The Java™ Tutorials > Essential Java Classes) (oracle.com)](https://docs.oracle.com/javase/tutorial/essential/io/)
* [Java - Files and I/O (tutorialspoint.com)](https://www.tutorialspoint.com/java/java_files_io.htm)
* [Enhancements in Java I/O (oracle.com)](https://docs.oracle.com/javase/8/docs/technotes/guides/io/enhancements.html)

# 5.- Java NIO.



Desde las versiones iniciales de Java, se han añadido características como:

* añadir programación **asíncrona** de E/S.
* permitir **obtener** información de **atributos** propios del sistema de archivos.
* reconocimiento de **enlaces simbólicos**.
* facilitado de algunas **operaciones básicas**.

**Java.io es la API tradicional**: es muy potente y flexible, pero es complicada de entender. **Java NIO** (New IO) es un nuevo API disponible desde Java7 que nos permite **mejorar el rendimiento**, así como **simplificar** el manejo de muchas cosas.

**Java.nio** define interfaces y clases para que la JVM tenga acceso a archivos, atributos de archivos y sistemas de archivos. Tiene muchísimas clases, pero **solo unas pocas son puntos de entrada al API**, lo que simplifica considerablemente su manejo.

|  |  |  |
| --- | --- | --- |
|  | **Java.io** | **Java.nio** |
| **Lectura/escritura de datos** | Secuencias de bytes y secuencias de caracteres | Canales y búferes. |
| **Bloqueo** | Siempre | Soporta ambas posibilidades: con bloqueo o sin bloqueo |
| **Selectores[[1]](#footnote-1)** | No | Sí |

El estudio de la API NIO completa, excede el módulo, por lo que veremos lo más básico: las clases ***Path*** y ***Files***.

* Java.nio.file.Path: representa un path y las clases que implementen esta interfaz puede utilizarse para localizar ficheros en el sistema de ficheros. Nos permite manejar rutas sea cual sea el S.O en el que estemos trabajando.
* Java.nio.file.Files: es el otro punto de entrada a la librería de ficheros de Java. Es la que nos permite manejar ficheros reales del disco desde Java.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

## 5.1.- Java NIO Path.

Una ruta puede:

* señalar a un archivo o un directorio
* ser absoluta o relativa.

Para crear un objeto de la clase Path, usaremos el siguiente método estático (de los muchos que tiene) de la clase **java.nio.file.Paths**

Path p = Paths.get("/home/ad/mi\_fichero");

Tal como ocurría con los objetos de tipo File, no es necesario **que los ficheros existan de verdad en el disco duro para que se puedan crear los objetos Path correspondientes**

**Otros** de los **métodos** del interfaz **Path** son para obtener el **nombre corto** de un fichero, obtener el **directorio** que lo contiene, resolver paths relativos, etc.

Una instancia de tipo Path refleja el sistema de nombrado del sistema operativo subyacente, por lo que **objetos path de diferentes sistemas operativos no pueden ser comparados fácilmente entre sí**.

**Operaciones con Path**:

* Recuperar **partes de una ruta**
* **Eliminar redundancias** de una ruta
* **Convertir una ruta**
* **Unir dos rutas**
* **Crear una ruta relativa** a otra dada
* **Comparar dos rutas**

**Ejemplo**:

import java.nio.file.Path;

import java.nio.file.Paths;

public class PathEjamplo {

public static void main(String args[]) {

Path path = Paths.get("C:/Users/alumno/PathEjemplo");

System.out.println(" path = " + path);

System.out.println(" is absoute ? = " + path.isAbsolute());

System.out.println(" file short name = " + path.getFileName());

System.out.println(" parent = " + path.getParent());

System.out.println(" uri = " + path.toUri());

path = Paths.get("/home/PathEjemplo");

System.out.println(" path = " + path);

System.out.println(" is absoute ? = " + path.isAbsolute());

System.out.println(" file short name = " + path.getFileName());

System.out.println(" parent = " + path.getParent());

System.out.println(" uri = " + path.toUri());

}

Clase FileSystem

El concepto de **FileSystem define un sistema de ficheros completo.** Mientras que por otro lado el concepto de **Path hace referencia a un directorio, fichero o link que tengamos dentro de nuestro sistema de ficheros**. Por ejemplo, el siguiente código usa FileSystem y Path para obtener el nombre de un fichero así como la carpeta padre en la que se encuentra ubicado:

import java.nio.file.FileSystem;

import java.nio.file.FileSystems;

import java.nio.file.Path;

import java.util.Iterator;

public class EjemploPath2 {

public static void main(String args[]) {

FileSystem sistemaFicheros = FileSystems.getDefault();

Path rutaFichero = sistemaFicheros.getPath("C:\\Users\\alumno\\PathEjemplo");

System.out.println(rutaFichero.getFileName());

System.out.println(rutaFichero.getParent().getFileName());

Path rutaDirectorio = sistemaFicheros.getPath("C:\\Users\\alumno");

Iterator<Path> it = rutaDirectorio.iterator();

while (it.hasNext()) {

System.out.println(it.next().getFileName());

}

}

}

## 5.2.- Clases Java NIO Files.

Esta clase tiene métodos estáticos para el manejo de ficheros. Los métodos de la clase Files trabajan sobre objetos Path.

**Operaciones principales**:

* Verificación de existencia y accesibilidad
* Borrar un archivo o directorio
* Copiar un archivo o directorio
* Mover un archivo o directorio

**Ejemplo 1**: existencia y comprobación de permisos.

import java.nio.file.Path;

import java.nio.file.Paths;

import java.nio.file.Files;

public class FileEjemplo {

public static void main(String args[]) {

Path path = Paths.get("C:\\Users\\alumno\\FileEjamplo\\hola.txt");

System.out.println(" path = " + path);

System.out.println(" exists = " + Files.exists(path));

System.out.println(" readable = " + Files.isReadable(path));

System.out.println(" writeable = " + Files.isWritable(path));

System.out.println(" executeable = " + Files.isExecutable(path))

**Ejemplo 2**: Creación y borrado de directorios.

import java.nio.file.Path;

import java.nio.file.Paths;

import java.nio.file.Files;

import java.io.IOException;

// Crea un nuevo fichero o directorio o lo borra si ya existe

public class FileEjemplo2 {

public static void main(String args[]) {

Path path = Paths.get("C:\\Users\\alumno\\prueba");

try {

if (Files.exists(path))

Files.delete(path);

else

Files.createFile(path);

} catch (IOException e) {

System.err.println(e);

System.exit(1);

}

}

}

El método **delete(Path)** borra el fichero o directorio, o lanza una excepción si el borrado falla.

**Ejemplo 2.1**: como capturar y gestionar las excepciones que pueden producirse en el borrado. Si el fichero o directorio no existe la excepción que se produce es **NoSuchFileException**. Los sucesivos catch permiten determinar por qué ha fallado el borrado.

try {

Files.delete(path);

} catch (NoSuchFileException x) {

System.err.format("%s: no such" + " file or directory%n", path);

} catch (DirectoryNotEmptyException x) {

System.err.format("%s not empty%n", path);

} catch (IOException x) {

// File permission problems are caught here.

System.err.println(x);

}

El metodo deleteIfExists(Path) también borra el fichero o directorio, pero no lanza ningún error en caso de que el fichero o directorio no exista.

**Ejemplo 3**: crear un Directorio.

import java.nio.file.Path;

import java.nio.file.Paths;

import java.nio.file.FileAlreadyExistsException;

import java.nio.file.Files;

import java.io.IOException;

// crea una nueva carpeta

public class FileEjemplo4 {

public static void main(String args[]) {

Path path = Paths.get("C:\\Users\\alumno\\newdir");

try {

Path newDir = Files.createDirectory(path);

} catch(FileAlreadyExistsException e){

// el directorio ya existe

} catch (IOException e) {

//error I/O

e.printStackTrace();

}

}

}

**Ejemplo 4**: copiar directorios.

El método copy(Path, Path, CopyOption...) permite hacerlo. La copia falla si el archivo de destino existe, a menos que se especifique la opción REPLACE\_EXISTING.

import java.nio.file.Path;

import java.nio.file.Paths;

import java.nio.file.FileAlreadyExistsException;

import java.nio.file.Files;

import java.io.IOException;

public class FileEjemplo5 {

public static void main(String args[]) {

Path sourcePath = Paths.get("C:\\Users\\alumno\\origen");

Path destinationPath = Paths.get("C:\\Users\\alumno\\FileEjamplo\\destino");

try {

Files.copy(sourcePath, destinationPath);

//Files.copy(sourcePath, destinationPath, StandardCopyOption.REPLACE\_EXISTING);

} catch (FileAlreadyExistsException e) {

System.out.println("el fichero existe");

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Ejemplo 5**: copiar ficheros.

import java.io.IOException;

import java.nio.file.FileAlreadyExistsException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.nio.file.StandardCopyOption;

public class FileEjemplo7 {

public static void main(String args[]) {

Path sourcePath = Paths.get("C:\\Users\\alumno\\FileEjamplo\\hola.txt");

Path destinationPath = Paths.get("C:\\Users\\alumno\\FileEjamplo\\destino\\hola.txt");

try {

Files.copy(sourcePath, destinationPath, StandardCopyOption.REPLACE\_EXISTING);

} catch (FileAlreadyExistsException e) {

System.out.println("el destino existe");

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Ejemplo 6**: mover ficheros y directorios, cambiando el nombre.

import java.io.IOException;

import java.nio.file.FileAlreadyExistsException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.nio.file.StandardCopyOption;

public class FileEjemplo7 {

public static void main(String args[]) {

Path sourcePath = Paths.get("C:\\Users\\alumno\\FileEjamplo\\hola.txt");

Path destinationPath = Paths.get("C:\\Users\\alumno\\FileEjamplo\\destino\\OtroNombre.txt");

try {

Files.move(sourcePath, destinationPath, StandardCopyOption.REPLACE\_EXISTING);

} catch (FileAlreadyExistsException e) {

System.out.println("el destino existe");

} catch (IOException e) {

e.printStackTrace();

}

}

}

## 5.3.- Escribir contenido en un fichero.

**Modos de acceso, el parámetro OpenOptions**:

Por ejemplo, si el usuario tiene permisos de lectura y escritura sobre un fichero, un programa Java que solo quiera leerlo puede abrir el fichero solo en modo lectura, lo que ayudará a evitar bugs desde el propio lenguaje.

Por ello, Java define una serie de modos de acceso a un fichero. La forma más cómoda de utilizar este parámetro es a través del **enum StandardOpenOptions** que puede tomar los siguientes valores (hay más):

* **WRITE**: habilita la escritura en el fichero
* **APPEND**: todo lo escrito al fichero se hará al final del mismo
* **CREATE\_NEW**: crea un fichero nuevo y lanza una excepción si ya existía
* **CREATE**: crea el fichero si no existe y simplemente lo abre si ya existía
* **TRUNCATE\_EXISTING**: si el fichero existe, y tiene contenido, se ignora su contenido para sobreescribirlo desde el principio.

Los métodos que se muestran en los siguientes ejemplos utilizan este parámetro. En la descripción de cada método **en el API se explica cuál es el comportamiento por defecto en caso de no utilizarse este parámetro**.

Escritura desde arrays de bytes:

La escritura a ficheros mediante arrays es la forma más sencilla (y limitada) de escritura de ficheros, y se realiza mediante el método **java.nio.file.Files.write().**

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.nio.file.StandardOpenOption;

public class FileEjemplo10 {

public static void main(String args[]) {

Path inputFile = Paths.get("C:\\Users\\alumno\\FileEjamplo\\origen\\hola.txt");

Path outputFile = Paths.get("C:\\Users\\alumno\\FileEjamplo\\destino\\hola.txt");

try {

byte[] contents = Files.readAllBytes(inputFile);

Files.write(outputFile, contents, StandardOpenOption.WRITE,

StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE\_EXISTING);

} catch (IOException e) {

System.err.println(" ERROR : " + e);

System.exit(1);

}

}

}

Escritura desde buffers

Esta resulta mucho más eficiente que utilizando arrays de bytes para ficheros grandes.

El siguiente programa Java copia ficheros, accediendo al fichero original una   
vez por línea y escribiendo en el fichero destino una línea cada vez. Utiliza las clases de **java.io:** **BufferedReader** y **BufferedWriter**:

import java .nio. file. Path;

import java .nio. file. Paths ;

import java .nio. file. Files ;

import java .io. IOException;

import java .nio. charset . Charset ;

import java .io. BufferedReader;

import java .io. BufferedWriter;

import java .nio. file. StandardOpenOption;

public class FileEjemplo11 {

// Copy a file

public static void main( String args []) {

Path input = Paths . get( "C:\\Users\\alumno\\FileEjamplo\\origen\\hola.txt") ;

Path output = Paths . get("C:\\Users\\alumno\\FileEjamplo\\destino\\hola.txt") ;

try {

BufferedReader inputReader = Files . newBufferedReader(input , Charset .

defaultCharset());

BufferedWriter outputWriter = Files . newBufferedWriter(output , Charset .

defaultCharset() , StandardOpenOption. WRITE , StandardOpenOption.

CREATE , StandardOpenOption. TRUNCATE\_EXISTING);

String line;

while ( ( line = inputReader. readLine ()) != null ) {

outputWriter. write (line , 0, line. length ());

outputWriter. newLine () ;

}

inputReader. close ();

outputWriter. close ();

} catch ( IOException e) {

System .err. println (" ERROR : " + e);

System . exit (1);

}

}

}

**Enlaces de interés**: [Java NIO vs. IO - DZone](https://dzone.com/articles/java-nio-vs-io), [Serialización de objetos en java - ChuWiki (chuidiang.org)](https://chuidiang.org/index.php?title=Serializaci%C3%B3n_de_objetos_en_java)

# 6.- Trabajo con ficheros XML: analizadores sintácticos (parser) y vinculación (binding).

El metalenguaje XML se crea para evitar problemas de **interoperabilidad entre plataformas y redes**. Con él se consigue un **soporte estándar para el intercambio de datos**: no sólo los datos están en un formato estándar sino también la forma de acceder a ellos. Y entre las ventajas de su uso destacamos:

* **Facilita el intercambio de información entre distintas aplicaciones** ya que se basa en **estándares aceptados**.
* **Proporciona una visión estructurada de la información**, lo que permite su posterior tratamiento de forma local.

## 6.1.- Conceptos previos.

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).

Este último paso, se realiza mediante la técnica del “parsing”, o **análisis léxico-sintáctico**. Los programas que lo llevan a cabo se denominan "parsers”.

El “parsing XML” comprueba que sea un documento XML bien formado, y pasa el contenido a una aplicación cliente que necesite consumir dicha información.

**Schema**: un esquema (o esquema) 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 o hijos del padre.

Un documento XML no tiene porqué tener un esquema, pero si lo tiene, debe atenerse a ese esquema para ser un documento XML válido.

**Para saber más**: [XML Schema. - Wikipedia, la enciclopedia libre](https://es.wikipedia.org/wiki/XML_Schema.)

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

## 6.2.- Definiciones.

JAXB (Java Architecture for XML Binding) simplifica el acceso a documentos XML representando la información obtenida de los documentos XML en un programa en formato Java.

JAXB permite (mediante distintos métodos) **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**: consiste en "escanear" el documento y dividirlo o separarlo lógicamente en piezas discretas. El contenido parseado está entonces disponible para la aplicación.
* **Binding**: vincular un esquema (schema) significa generar un conjunto de clases Java que representan el esquema.
* **Schema compiler (compilador de esquema)**: 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 ruintime framework**: proporciona operaciones de unmarshalling y marshalling para acceder, manipular y validar contenido XML usando un esquema derivado o elementos de programa.
* **Marshalling**: proceso de codificación de un objeto en un medio de almacenamiento, normalmente un fichero. Es decir, convierte un árbol de objetos Java JAXB a ficheros XML.
  + Por defecto, el marshaller utiliza 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**: SAX (Simple API for XML) es un API para parsear ficheros XML. Proporciona un mecanismo para leer datos de un documento XML. Otra alternativa es DOM (Document Object Model). [Más información sobre el DOM](https://es.wikipedia.org/wiki/Document_Object_Model).

## 6.3.- “Parser” o analizador XML.

Introducción

Los documentos XML consiguen estructurar la información intercalando una serie de marcas denominadas etiquetas. Esas etiquetas son “como contenedores” de información. Así, una etiqueta puede contener otras etiquetas o información textual. Esto es, **la información se estructura jerárquicamente**.

**Cualquier dato**, ya sea numérica o booleana, **habrá transcribirla en modo texto**, de modo que cualquiera que sea el sistema de representación de datos será posible leer e interpretar correctamente la información contenida en un archivo XML.

Es cierto que los caracteres **se pueden escribir usando también diferentes sistemas de codificación, pero XML ofrece diversas técnicas para evitar que esto sea un problema**. Por ejemplo, es posible incluir en la cabecera del archivo que codificación se ha utilizado durante el almacenamiento, o también se pueden escribir los caracteres de código ASCII superior a 127, utilizando **entities de carácter**, una **forma universal de codificar cualquier símbolo**.

**Se pueden establecer similitudes entre XML y la orientación a objetos**. La información, en las aplicaciones orientadas a objeto, estructura, agrupa y jerarquiza en clases, y en los documentos XML se estructura, organiza y jerarquiza en etiquetas contenidas unas dentro de otras y atributos de las etiquetas.

Analizador (parser) XML

Un **parser** XML es un **módulo**, biblioteca o programa que se ocupa de **analizar, clasificar y transformar un archivo de XML en una representación interna**, extrayendo la información contenida en cada una de las etiquetas y relacionándola de acuerdo con su posición en la jerarquía.

En el caso de XML, como el formato siempre es el mismo, existen muchos **parsers** disponibles **que pueden averiguar si un documento XML cumple con una determinada gramática**. Estos analizadores o parsers **pueden ser**: **secuenciales** como SAX o STAX o **jerarquicos** como DOM.

**Analizadores secuenciales**: también llamados analizadores sintácticos.

* Permiten **extraer el contenido a medida que se van descubriendo las etiquetas** de apertura y cierre
* Son **muy rápidos**, pero cada vez que se necesita acceder a una parte del contenido necesario **releer todo el documento de arriba a abajo**.
* **En Java, existen dos**:
  + SAX (Simple API for XML): muy usado en varias bibliotecas de tratamiento de datos XML, pero **no suele usarse en aplicaciones finales**.
  + STAX (Streaming API for XML): posterior a SAX y que lo ha superado.

**Analizadores jerárquicos**:

Generalmente, las aplicaciones finales que necesitan trabajar con datos XML suelen usar analizadores jerárquicos.

* Realizan un análisis secuencial (igual que los parsers sintácticos).
* Permiten además:
  + Clasificar el contenido.
  + Se almacenan en al RAM siguiendo la estructura jerárquica detectada en el documento.
* Por ello, **facilita mucho las consultas que haya que repetir varias veces**.
* Son ideales para aplicaciones que requieran una consulta continua de los datos.
* El **formato en el que se almacena en RAM** ha sido especificado por el organismo internacional W3C (World Wide Web Consortium): el **DOM** (Document Object Model). HTML y JavaScript la han popularizado mucho.
  + **Java materializa esta especificación en forma de interfaces**. La principal se llama Document y representa todo un documento XML.
    - Al tratarse de una interfaz, puede ser implementada por varias clases.

**Comparativa analizadores**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CARACTERÍSTICA** | **STAX** | **SAX** | **DOM** | **TRAX** |
| **Tipo de API** | Pull, streaming | Push, streaming | En memoria | Regla XSLT |
| **Facilidad de uso** | Alta | Media | Alta | Media |
| **Capacidad XPath** | No | No | Si | Si |
| **Eficiencia de CPU y memoria** | Buena | Buena | Varia | Varia |
| **Solo hacia adelante** | Si | Si | No | No |
| **Lee XML** | Si | Si | Si | Si |
| **Escribe XML** | Si | No | Si | Si |
| **Crear, Leer, Modificar, Borrar** | No | No | Si | No |

## 6.4.- DOM (Document Object Model).

El estándar W3C define la especificación de la clase ***DocumentBuilder*** con el propósito de poder **instanciar estructuras DOM** a partir de un XML. La clase ***DocumentBuilder*** es una clase abstracta, y para que se pueda adaptar a las diferentes plataformas, puede necesitar fuentes de datos o requerimientos diversos.

La W3C también proporciona una implementación de esta interfaz, mediante la clase ***DocumentBuilderFactory***.

**Las instrucciones necesarias para leer un archivo XML y crear un objeto Document** serían las siguientes:

DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();

DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();

Document doc = dBuilder.parse(new File("fitxer.xml"));

**Transformer**: Es capaz de pasar la información contenida en un objeto Document a un archivo de texto en formato XML . También sería capaz de hacer la operación inversa, pero el mismo DocumentBuilder ya se encarga de ello.

También es una clase abstracta y requiere de una *fábrica* para poder ser instanciada.

La clase ***Transformer*** puede trabajar con multitud de contenedores, ya que en realidad trabaja con un par de tipos adaptadores (clases que hacen compatibles jerarquías diferentes), que se llaman **Source** y **Result**. Las clases que implementen estas interfaces se encargarán de hacer compatible un tipo de contenedor específico a lo que le pida la clase ***Transformer*** (sea convertir de fichero a objeto, o de objeto a fichero).

**Por ejemplo**: las clases **DOMSource, SAXSource o StreamSource** como adaptadores del contenedor de la fuente de información (DOM, SAX o Stream respectivamente). **DOMResult, SAXResult o StreamResult** son los adaptadores equivalentes del contenedor destino.

**Cómo realizar una transformación de DOM**:

DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();

DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();

Document doc = dBuilder.parse(new File("fitxer.xml"));

// ... Hacer las modificaciones al DOM que se deseen

// Creación de una instancia Transformer

Transformer trans = TransformerFactory.newInstance.newTransformer ;

// Creación de los adaptadores Source y Results a partir de un Documento

// y un File.

File file = new File ("fitxer2.xml") ;

StreamResult result = new StreamResult(file) ;

DOMSource source = new DOMSource(doc) ;

trans.transform(source, result) ;

Con el fin de rebajar la complejidad, vamos a crear una clase que llamaremos **XmlCtrlDomcon** utilidades genéricas que nos simplifiquen el traspaso de archivos XML a DOM o viceversa.

import java.io.File;

import java.io.IOException;

import javax.xml.parsers.DocumentBuilderFactory;

import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;

import org.xml.sax.SAXException;

import javax.xml.transform.Transformer;

import javax.xml.transform.TransformerFactory;

import javax.xml.transform.TransformerException;

import javax.xml.transform.OutputKeys;

import javax.xml.transform.dom.DOMSource;

import javax.xml.transform.stream.StreamResult;

public class XmlCtrlDom {

     public static Document createDocument() throws ParserConfigurationException {

          Document doc = null;

          DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();

          DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();

          doc = dBuilder.newDocument();

          return doc;

     }

     public static void writeDocumentToXmlFile(Document doc, File file) throws TransformerException {

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

          transformer.setOutputProperty(OutputKeys.INDENT, "yes");

          StreamResult result = new StreamResult(file);

          DOMSource source = new DOMSource(doc);

          transformer.transform(source, result);

     }

     public static Document createDocumentFromFile(File xmlFile) throws ParserConfigurationException, SAXException, IOException {

          Document doc = null;

          DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();

          DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();

          doc = dBuilder.parse(xmlFile);

          doc.getDocumentElement().normalize();

          return doc;

     }

     //...}

### 6.4.1.- La estructura DOM.

La estructura DOM toma la forma de un árbol, donde cada parte del XML se encontrará representada en forma de nodo.

Puede haber **diferentes tipos de nodos** en función de la posición del mismo en el documento XML.

* **El nodo principal que representa todo el XML entero denomina documento**
* las diversas**etiquetas**, incluida la etiqueta raíz, se conocen como **nodos elemento.**
* El **contenido textual**de una etiqueta se instancia como**nodo de tipo TextElement**
* los **atributos**como nodos de**tipo Atribute.**

**Cada nodo específico** dispone de **métodos para acceder a sus datos concretos** (nombre, valor, nodos hijos, nodo padre, etc.).

**El DOM resultante obtenido de un XML termina siendo una copia exacta del archivo, pero con una disposición distinta**. Tanto el DOM como el XML tendrán información no visible, como los **retornos de carro**, que **deben tenerse en cuenta para** saber cómo **procesar correctamente el contenido** y evitar sorpresas un poco comprensibles.

**Al mapear XML**, debes tener en cuenta las siguientes diferencias:

* los retornos de carro se interpretan en DOM como un *hijo*
* el contenido textual de las etiquetas se plasma como el primer hijo de la etiqueta contenedora.

La interfaz **Document** contempla un conjunto de **métodos para seleccionar diferentes partes del árbol** a partir del nombre de la etiqueta o de un atributo identificador. De este modo, podremos ir explorando partes del árbol sin necesidad de tener que pasar por todos los nodos.

**Interfaces DOM** más comunes:

* **Node** − Representa a cualquier nodo del documento.
* **Element** − expone propiedades y métodos para manipular los elementos del documento y sus atributos.
* **Attr** − Representa un atributo de un elemento.
* **Text** − Contenido de un elemento o atributo.
* **Document** − Representa al documento XML completo. Generalmente nos referiremos a el como árbol DOM. Proporciona información del documento. Permite crear nuevos nodos en el documento.
* **NodeList**. Colección de nodos a los que se puede acceder por medio de un índice.

**Métodos DOM** más usuales:

* **Document.getDocumentElement()** − Retorna el elemento raiz del documento.
* **Node.getFirstChild()** − Retorna el primer nodo hijo del nodo.
* **Node.getLastChild()** − Retorna el ultimo nodo hijo del nodo.
* **Node.getNextSibling()** − Retorna el siguiente hermano de un nodo.
* **Node.getPreviousSibling()** − Retorna el hermano anterior de un nodo.
* **Node.getAttribute(attrName)** − Retorna el atributo del nodo con el nombre que se pasa como atributo.

Para facilitar la obtención del contenido de un **Element**, **ampliaremos las utilidades** de la clase **XmlCtrlDom** **añadiendo dos métodos más**.

public static String getValorEtiqueta(String tag, Element element) {

     Node nValue = element.getElementsByTagName(tag).item(0);

     return nValue.getChildNodes().item(0).getNodeValue();

}

public static Element getElementEtiqueta(String tag, Element element) {

     return (Element) element.getElementsByTagName(tag).item(0);

}

1. El primero recibe el nombre de la etiqueta y el elemento (o nodo parcial del árbol) a partir del cual se desea realizar la **búsqueda del elemento**
   1. Conseguimos todos los nodos que tengan por nombre el valor del parámetro etiqueta. Si solo existe un único nodo con el nombre del parámetro, éste ocupará la primera posición de la lista. Por ello accedemos con el método item, indicando que nos interesa el primer elemento (posición cero).
   2. **Este método devuelve el valor o nodo de tipo texto que tenga este nodo**.
2. El segundo método es muy **similar** al primero, pero en vez de recuperar el texto, **obtendremos el nodo con todos los hijos que tenga**. Es útil para aplicar a nodos intermedios (no textuales).

**Otros métodos**: de los objetos tipo Element:

* **appendChild()**: permite añadir nuevos hijos.
* **setAttribute()**: permite asignar el valor a un atributo.
* **getAttribute()**: permite consultar el valor de un atributo.
* **getParentNode()**: permite obtener el elemento padre.
* **getFirstChild/getLastChild()**: permiten obtener el primer/último hijo.
* **getNextSibling()**: para navegar de hermano en hermano.

**Para crear nuevos elementos**:

El objeto Document hará de factory para cualquier nodo del documento.

* **createElement**: permite crear una nueva etiqueta (elemento).
* **createTextNode**: crea un nodo de tipo texto (contenido).
* **createComment**: crea un comentario dentro del nodo padre.

Para añadir esos nodos que hemos creado al árbol, utilizaremos el método **appendChild** que ya hemos visto sobre un elemento padre.

### 6.4.2.- Ejemplo completo.

import org.w3c.dom.\*;

import javax.xml.parsers.\*;

import java.io.\*;

public class EjemploCompleto {

     public static void main(String[] args) {

          // Crea un SAXBuilder

          DocumentBuilderFactory factory =

          DocumentBuilderFactory.newInstance();

          DocumentBuilder builder = factory.newDocumentBuilder();

          // Crea un documento para el flujo de datos

          StringBuilder xmlStringBuilder = new StringBuilder();

          xmlStringBuilder.append("<?xml version="1.0"?> <class> </class>");

          ByteArrayInputStream input = new ByteArrayInputStream(

          xmlStringBuilder.toString().getBytes("UTF-8"));

          Document doc = builder.parse(input);

          // Obtiene el elemento raíz

          Element rootElement = doc.getDocumentElement();

          // Examinar atributos

          String attribute = rootElement.getAttribute("attributeName");

          // Examinar hijos

          NodeList nodes = rootElement.getChildNodes();

     }

}

Para ver el **ejemplo original** de la plataforma, [consultar la plataforma](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Acceso%20a%20datos/Evaluacion%201/Tema%202/Apuntes/AD02/web/html/642_ejemplo_completo.html).

### 6.4.3.- Creación de un fichero XML a partir de un documento.

El **código básico para realizar una transformación de DOM archivo de texto XML** sería el siguiente:

import org.w3c.dom.\*;

import javax.xml.parsers.\*;

import java.io.\*;

import java.util.logging.Level;

import java.util.logging.Logger;

import javax.xml.transform.Result;

import javax.xml.transform.Source;

import javax.xml.transform.Transformer;

import javax.xml.transform.TransformerConfigurationException;

import javax.xml.transform.TransformerException;

import javax.xml.transform.TransformerFactory;

import javax.xml.transform.dom.DOMSource;

import javax.xml.transform.stream.StreamResult;

public class XMLPruebas {

     public static void main(String[] args) {

          // Instanciar el documento DOM en memoria

          DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

          DocumentBuilder builder = factory.newDocumentBuilder();

          DOMImplementation implementation = builder.getDOMImplementation();

          Document document = implementation.createDocument(null, "personas", null);

          // Crear el nodo raíz y colgarlo del documento

          Element raiz = document.getDocumentElement();

          // Crear un nodo persona y colgarlo de la raíz

          Element persona = document.createElement("persona");

          persona.appendChild(document.createTextNode("Santiago"));

          raiz.appendChild(persona);

          // Crear un nodo nombre y colgarlo de persona

          Element nombre = document.createElement("nombre");

          persona.appendChild(nombre);

          // Guarda el XML en un fichero

          Source source = new DOMSource(document);

          Result result = new StreamResult(new java.io.File("personas.xml"));

          Transformer transformer;

          try {

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

               transformer.transform(source, result);

          } catch (TransformerConfigurationException ex) {

               System.out.println(ex.getMessage());

          } catch (TransformerException ex) {

               System.out.println(ex.getMessage());

          }

     }

}

**Además:** puede encontrar otro ejemplo completo en el siguiente enlace: [Java - Crear Archivo XML - decodigo.com](http://decodigo.com/java-crear-archivo-xml)

## 6.5.- SAX.

**SAX** (API Simple para XML), es una interfaz simple de aplicaciones XML, fácil e intuitiva. El parser **trabaja de la siguiente forma**:

1. **Lee un fichero XML de forma secuencial**, produciendo eventos de manera secuencial en función de los resultados de lectura.
2. **Cada evento invoca a un método que ha sido realizado por el programador procesando el documento poco a poco y no consumiendo prácticamente memoria**, **aunque** por otra parte **impide tener visión general** de todo el documento XML.

Permite analizar el documento XML de forma secuencial, es decir, **va cargando en memoria diferentes partes del mismo**, cosa contraria a lo que hace DOM que carga todo el documento de golpe en memoria.

**¿Cuándo es conveniente SAX?**:

* Cuando el **documento XML** es considerablemente **grande**.
* Cuando **no se requiere una modificación estructural**.
* **Si se quiere parsear diferentes partes** del documento.

La lectura de un documento XML produce eventos los cuales invocan a métodos. Los eventos son:

* **Inicio y fin de un archivo XML**:
  + startDocument()
  + endDocument()
* **Inicio y fin de un elemento**:
  + startElement()
  + endElement()

**SAX genera un objeto procesador** de XML llamado **XMLReader**, para después decir **qué objetos tienen métodos para transferir los eventos**. Estos objetos implementan los siguientes **interfaces**:

* **ContentHandler**, el cual recibe las notificaciones de los eventos del fichero XML.
* **DTDHandler**, que recoge los eventos del DTD del fichero.
* **ErrorHandler**, crea el tratamiento de errores.
* **EntityResolver**, se utilizan siempre que se referencia a otra entidad.
* **DefaultHandler**, la cual implementa por defecto los métodos, siendo el programador quien los defina. **Con esta clase se podrá crear el parser XML**. Se compone de los siguientes eventos básicos:
  + **startDocument**: se llama cuando se detecta que el documento empieza. Aquí deben indicarse las acciones a realizar al inicio del documento.
  + **endDocument**: se llama cuando se detecta que el documento ha acabado. Por lo tanto, aquí deben indicarse las acciones a realizar al finalizar el documento.
  + **startElement**: Se llama cuando encuentra un nuevo elemento, nodo, etiqueta, tag, etc. Aquí debe indicarse el tratamiento que se deberá realizar sobre cada nuevo elemento, como la recogida de información de sus atributos.
  + **endElement**: Se llama una vez ha leído el elemento. Aquí se encuentra la información del nodo y del contenido del nodo por lo que es aquí donde, normalmente, se recoge la información.
  + **Characters**: se invoca para encontrar una cadena de texto.

Ejemplo: leer ficheros XML con SAX

import org.xml.sax.Attributes;

import org.xml.sax.SAXException;

import org.xml.sax.helpers.DefaultHandler;

import org.xml.sax.helpers.XMLReaderFactory;

import org.xml.sax.XMLReader;

public class XMLSAXParserExample {

    public static void main(String[] args) {

        try {

            XMLReader xmlReader = XMLReaderFactory.createXMLReader();

            XMLHandler handler = new XMLHandler();

            xmlReader.setContentHandler(handler);

            xmlReader.parse("archivo.xml");

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}

class XMLHandler extends DefaultHandler {

    @Override

    public void startElement(String uri, String localName, String qName, Attributes attributes) throws SAXException {

        System.out.println("Elemento Inicio: " + qName);

    }

    @Override

    public void characters(char[] ch, int start, int length) throws SAXException {

        String contenido = new String(ch, start, length).trim();

        if (!contenido.isEmpty()) {

            System.out.println("Contenido: " + contenido);

        }

    }

    @Override

    public void endElement(String uri, String localName, String qName) throws SAXException {

        System.out.println("Elemento Fin: " + qName);

    }

}

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

## 6.6.- StAx.

**Es una API basada en Java para analizar documentos XML**de forma similar a como lo hace el analizador SAX. Pero hay **dos diferencias principales entre las dos API**:

* **StAX es una API PULL, mientras que SAX es una API PUSH**. Esto significa que en StAX, una aplicación cliente necesita solicitar al analizador StAX que **obtenga** información de XML siempre que lo necesite. Mientras que en SAX, se requiere una aplicación cliente para obtener información cuando el analizador SAX notifica a la aplicación cliente que la información está disponible.
* **StAX API puede leer y escribir documentos XML**. Usando SAX API, un archivo XML solo se puede leer.

**StAX consta realmente de 2 distintas API**: ([Aclaración ChatGPT](https://chat.openai.com/share/7e660f3d-5d25-41f8-86f0-a8b84a360d6b))

* **API Cursor**: Este cursor puede apuntar un elemento a la vez y siempre se mueve hacia adelante hasta el final, nunca hacia atrás.
* **API Iterator**: Representa un flujo de un documento XML como un conjunto de objetos de eventos discretos. La aplicación (la que lee el XML) saca estos eventos en el mismo orden que los proporciona el parser al leerlos del documento XML. Por cada paso de iteración se obtiene un objeto XMLEvent, que contiene información sobre el evento generado en el proceso de lectura de la información.
  + Mediante la referencia a este objeto, podemos extraer la información de los eventos generados durante la lectura, con los métodos que proporciona esta clase.

**Diferencias entre Cursor e Iterator**:

* Cuando se usa la ***Iterator***, es posible ir hacia atrás en la jerarquía, cosa que no puede realizarse cuando usamos ***Cursor***.
* Sin embargo la ***Cursor API*** es **más eficiente en cuanto a memoria**, así que la selección de que estilo de API utilizar dependerá de las necesidades.

### 6.6.1.- Características de StAX.

* **Lee un documento XML de arriba a abajo**, reconociendo los tokens que componen un documento XML bien formado.
* **Los tokens se procesan en el mismo orden en que aparecen** en el documento.
* **Informa a la aplicación de la naturaleza de los tokens** que el analizador ha encontrado a medida que ocurren.
* **El programa proporciona un lector de "eventos"** que **actúa como un iterador** e itera sobre el evento para obtener la información requerida. **Otro lector disponible es "cursor"** que actúa como un puntero a los nodos XML.
* **A medida que se identifican los eventos, los elementos XML pueden recuperarse del objeto de evento** y pueden procesarse más.

**Debemos usar un parset StAX cuando**:

* Puede **procesar el documento XML de forma lineal de arriba a abajo**.
* El **documento no está profundamente anidado**.
* **Está procesando un documento XML muy grande cuyo árbol DOM consumiría demasiada memoria**. Las implementaciones DOM típicas usan diez bytes de memoria para representar un byte de XML.
* El problema a resolver involucra **solo una parte del documento XML**.
* Los **datos** están **disponibles** **tan pronto como el analizador los vea**, por lo que StAX **funciona bien para** un documento XML que llega a través de **una transmisión**.

**Para poder utilizar clases e interfaces de ambas API**necesitamos un origen de datos XML valido. Esto lo conseguimos con la factoria**XMLInputFactory:**

XMLInputFactory xmlif = XMLInputFactory.newInstance();

XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader("books.xml"));

O bien:

XMLInputFactory inputFactory = XMLInputFactory.newInstance();

InputStream in = new FileInputStream("books.xml");

Cómo funciona

El analizador crea diferentes tipos de eventos a medida que avanza leyendo el documento XML de origen.

Algunos de los **tipos de eventos importantes** son:

1. Comience el documento
2. Elemento de inicio
3. Comentarios
4. Caracteres
5. Elemento final
6. Documento final

**Por ejemplo**: [ver ejemplo aquí](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Acceso%20a%20datos/Evaluacion%201/Tema%202/Apuntes/AD02/web/html/661_caractersticas_de_stax.html).

Aspectos importantes

* **Los eventos se crean en el orden en que se encuentran los elementos XML correspondientes en el documento**, incluida la anidación de elementos, la apertura y el cierre de elementos, el orden de los atributos, el inicio del documento y el final del documento, y así sucesivamente.
* Al igual que con las etiquetas XML (apertura y cierre), **todos los elementos del contenedor tienen los correspondientes eventos de inicio y final**; por ejemplo, cada StartElement tiene un EndElement correspondiente, incluso para elementos vacíos.
* **Los eventos de atributo se tratan como eventos secundarios**, y se accede a ellos desde su evento StartElement correspondiente.
* **Los eventos de Espacio de nombres** también **se tratan como secundarios**, pero aparecen dos veces y **son accesibles dos veces en la secuencia de eventos**, primero desde su StartElement correspondiente y luego desde su EndElement correspondiente.
* **Los eventos de caracteres se especifican para todos los elementos, incluso si esos elementos no tienen datos de caracteres**.
  + Del mismo modo, los eventos de carácter se pueden dividir entre eventos.
* El analizador StAX mantiene una **pila de espacio de nombres**, que contiene **información sobre todos los espacios de nombres XML definidos para el elemento actual y sus antecesores**. Se puede acceder a esta pila, a través de la interfaz **javax.xml.namespace.NamespaceContext** , mediante el prefijo del espacio de nombres o el URI.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

### 6.6.2.- Ejemplo de API Cursor.

Utilizamos el documento books.XML que podemos encontrar [aquí](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Acceso%20a%20datos/Evaluacion%201/Tema%202/Apuntes/AD02/web/html/662_ejemplo_de_api_cursor.html).

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.util.ArrayList;

import javax.xml.stream.FactoryConfigurationError;

import javax.xml.stream.XMLInputFactory;

import javax.xml.stream.XMLStreamException;

import javax.xml.stream.XMLStreamReader;

import javax.xml.stream.events.XMLEvent;

// El programa java que recorre el documento para extraer los titulos y el atributo lang es el siguiente

public class ListaLibros {

     public static void main(String[] args) throws FileNotFoundException, XMLStreamException {

          // Creamos el flujo

          XMLInputFactory xmlif = XMLInputFactory.newInstance();

          XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader("books.xml"));

          String tag = null;

          int eventType;

          System.out.println("Lista de libros");

          // Iteramos con el cursor a lo largo del documento

          while (xmlsr.hasNext()) {

               eventType = xmlsr.next();

               switch (eventType) {

                    case XMLEvent.START\_ELEMENT:

                         tag = xmlsr.getLocalName();

                         if (tag.equals("title")) {

                              System.out.println(xmlsr.getElementText() + " idioma&nbsp; " + xmlsr.getAttributeValue(0));

                         }

                         break;

                    case XMLEvent.END\_DOCUMENT:

                         System.out.println("Fin del documento");

                         break;

               }

          }

     }

}

### 6.6.3.- Ejemplo de API Event.

Utilizamos el documento books.XML del ejemplo anterior.

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.InputStream;

import java.util.Iterator;

import javax.xml.namespace.QName;

import javax.xml.stream.XMLEventReader;

import javax.xml.stream.XMLInputFactory;

import javax.xml.stream.XMLStreamConstants;

import javax.xml.stream.XMLStreamException;

import javax.xml.stream.events.Attribute;

import javax.xml.stream.events.StartElement;

import javax.xml.stream.events.XMLEvent;

public class EventReader {

     public static void main(String[] args) {

          // primero crea un nuevo XMLInputFactory

          XMLInputFactory inputFactory = XMLInputFactory.newInstance();

          // Configura un nuevo eventReader a partir del fichero XML

          InputStream in = null;

          try {

               in = new FileInputStream("books.xml");

          } catch (FileNotFoundException e1) {

               e1.printStackTrace();

          }

          try {

               XMLEventReader eventReader = InputFactory.createXMLEventReader(in);

               // Bucle que recorre todos los eventos

               while (eventReader.hasNext()) {

                    XMLEvent event = eventReader.nextEvent();

                    // si el evento es el inicio del nodo titulo

                    // avanzo un evento para obtener el titulo del libro

                    if (event.getEventType() == XMLStreamConstants.START\_ELEMENT) {

                         StartElement startElement = event.asStartElement();

                         if (startElement.getName().getLocalPart() == "title") {

                              Iterator iterator = ((StartElement) event).getAttributes();

                              while (iterator.hasNext()) {

                                   Attribute attribute = (Attribute) iterator.next();

                                   QName name = attribute.getName();

                                   String value = attribute.getValue();

                                   System.out.println("Atributo name/valor: " + "/" + value);

                              }

                              event = eventReader.nextEvent();

                              System.out.println((String) event.asCharacters().getData());

                         }

                    } else if (event.getEventType() == XMLStreamConstants.END\_DOCUMENT) {

                         System.out.println("fin del documento");

                    }

               }

          } catch (XMLStreamException e) {

               e.printStackTrace();

          }

     }

}

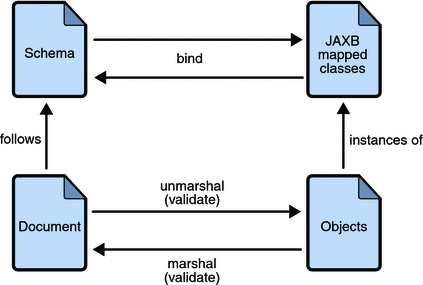
## 6.7.- Binding.

Técnica que consiste en vincular clases Java con formatos específicos de almacenamiento de manera automatizada.

En Java, existen varias bibliotecas para hacer **binding**: JAXB, JiBX, XMLBinding, etc. Desde la versión 6.0 se ha incorporado en el JDK estándar JAXB, una potente biblioteca.

**JAXB** permite a los desarrolladores Java asignar clases de Java a representaciones XML. Esto es, **proporciona**:

* la capacidad de serializar las referencias de objetos Java a XML
* y la inversa, es decir, deserializar XML en objetos Java



### 6.7.1.- Configuración con anotaciones.

Las**Anotaciones pueden asociarse a un paquete, a una clase, a un atributo o incluso a un parámetro**.

Estas **clases especiales se declaran**en el código de la aplicación **anteponiendo el símbolo @ en el nombre de la Anotación.**

**Cuando el compilador de Java detecta una Anotación crea una instancia y la inyecta en el elemento estructural afectado** (paquete, clase, método, atributo, etc.). Esto hace que éstas **no aparezcan como atributos o métodos propios** del objeto, y por eso decimos que no interacciona con el modelo de datos, **pero las aplicaciones que lo necesiten pueden obtener la instancia inyectada y usarla**.

Las Anotaciones **pueden declararse con parámetros o sin ellos.** En caso de que tengan parámetros, estos pueden ser otros Anotaciones, valores constantes o valores literales, de modo que **estén disponibles en tiempo de compilación** (que es cuando el compilador realiza la inyección).

**Anotaciones principales**:

* **@XmlRootElement(namespace = "namespace", name = "nombre" )**: Define la raíz del XML.
* **@XmlType(propOrder = { "field2", "field1",.. }, name = "nombre")**: Permite definir en que orden se van escribir los elementos dentro del XML. Esta anotación deberán tenerla todas las clases que no mapean la raiz del XML.
* **@XmlElement(name = "nombre")**: Define el elemento de XML que se va usar. Si el atributo de la clase tiene el mismo nombre que la etiqueta XML, podemos omitirlo. Tenemos que escribir esta anotación antes del método setter correspondiente. Mirar ejemplos en el siguiente subcapitulo.
* **@XmlAttribute**: se utiliza para mapear atributos a los nodos XML.
* **@XmlElementWrapper**: ofrece la posibilidad de crear un contenedor alrededor de una representación XML. Este contenedor puede contener una colección de elementos.

El parámetro **name** = “nombre” permite especificar el nombre de la etiqueta XML.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

### 6.7.2.- Ejemplos con anotaciones.

import java.util.ArrayList;

import javax.xml.bind.annotation.XmlElement;

import javax.xml.bind.annotation.XmlElementWrapper;

import javax.xml.bind.annotation.XmlRootElement;

//Esto significa que la clases "Libreria.java" es el elemento raiz

@XmlRootElement

public class Libreria {

//Wrapper

@XmlElementWrapper(name = "ListaLibro")

@XmlElement(name = "Libro")

private ArrayList<Libro> ListaLibro;

private String nombre;

private String lugar;

public ArrayList<Libro> getListaLibro() {

return ListaLibro;

}

public void setListaLibro(ArrayList<Libro> ListaLibro) {

this.ListaLibro = ListaLibro;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

public void setLugar(String lugar) {

this.lugar = lugar;

}

public String getNombre() {

return nombre;

}

public String getLugar() {

return lugar;

}

}

Código que mapea cada nodo de <Libro> individualmente:

import javax.xml.bind.annotation.XmlType;

@XmlType(propOrder = { "autor", "nombre", "editorial", "isbn" })

public class Libro {

     private String nombre;

     private String autor;

     private String editorial;

     private String isbn;

     public String getNombre() {

          return nombre;

     }

     public String getAutor() {

          return autor;

     }

     public String getEditorial() {

          return editorial;

     }

     public String getIsbn() {

          return isbn;

     }

     public void setNombre(String nombre) {

          this.nombre = nombre;

     }

     public void setAutor(String autor) {

          this.autor = autor;

     }

     public void setEditorial(String editorial) {

          this.editorial = editorial;

     }

     public void setIsbn(String isbn) {

          this.isbn = isbn;

     }

}

### 6.7.3.- Funcionamiento de JAXB.

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

* **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.
* **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.
* **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:
  + **Instanciando las clases generadas**.
  + **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.
* **Acceder al árbol de contenido usando nuestra aplicación**: ahora podemos acceder al árbol de contenido y modificar sus datos.
* **Generar un documento XML desde el árbol de contenido**. Para poder hacerlo tenemos que invocar al método marshall sobre el objeto raíz del árbol.

JAXB proporciona **dos principales características**:

* Capacidad de serializar (marshalling) objetos Java a XML.
* Lo inverso, es decir, deserializar (unmarshalling) XML a objetos Java.

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.

**Pasos del proceso de enlace JAXB**:

1. Generar clases. Podemos generarlas manualmente o automáticamente mediante el uso de JAXB utilizando el esquema .xsd correspondiente al XML.
2. Deben compilarse todas las clases generadas. Los archivos fuente y el código de la aplicación.
3. Aplicar **Marshall** a los objetos de las clases mapeadas **para crear XML**.
4. Aplicar **UnMarshall** al XML para **obtener objetos** de las clases mapeadas.

Marshall

JAXBContext jaxbContext = JAXBContext.newInstance(Libreria.class); // Contexto

// Generar un objeto Marshaller

Marshaller marshaller = jaxbContext.createMarshaller();

marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);

// Pasarle una salida a nuestro marshaller

File libreria-jaxb = this.getFile();

if (libreria-jaxbl != null) {

marshaller.marshal(libreria, libreria-jaxb);

}

Unmarshall

Tenemos varios inputs y utilizaremos el fichero por ahora para poder parsear. Partiendo de que tenemos ya un contexto, el cual reutilizaremos en esta parte.

Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();

Libreria libreria = (Libreria) unmarshaller.unmarshal( libreria-jaxb);

Así de simple, recuperaremos o pasaremos de nuestro fichero xml a un objeto el cual tenemos ya **mapeado**.

**Para saber más**: [Guide to JAXB | Baeldung](https://www.baeldung.com/jaxb)

### 6.7.3.- Ejemplo: Marshall, Unmarshall.

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import javax.xml.bind.JAXBContext;

import javax.xml.bind.JAXBException;

import javax.xml.bind.Marshaller;

import javax.xml.bind.Unmarshaller;

public class JavaJAXB {

private static final String LIBRERIA\_XML = "./libreria-jaxb.xml";

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

// Lista de LIbros

ArrayList<Libro> libroLista = new ArrayList<Libro>();

// Creamos varios libros

Libro libro1 = new Libro();

libro1.setIsbn("978-0060554736");

libro1.setNombre("The Game");

libro1.setAutor("Neil Strauss");

libro1.setEditorial("Harpercollins");

libroLista.add(libro1);

Libro libro2 = new Libro();

libro2.setIsbn("978-3832180577");

libro2.setNombre("Feuchtgebiete");

libro2.setAutor("Charlotte Roche");

libro2.setEditorial("Dumont Buchverlag");

libroLista.add(libro2);

// Se crea La libreria y se le asigna la lista de libros

Libreria libreria = new Libreria();

libreria.setNombre("LIbreria sin limite");

libreria.setLugar("Barrio Obrero");

libreria.setListaLibro(libroLista);

// Creamos un contexto de la clase JAXB y lo intanciamos

JAXBContext context = JAXBContext.newInstance(Libreria.class);

Marshaller m = context.createMarshaller();

m.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, Boolean.TRUE);

// Lo creamos con system out

m.marshal(libreria, System.out);

// Escribimos en el archivo

m.marshal(libreria, new File(LIBRERIA\_XML));

// Obtenemos las variables obtenidas del XML creado anteriormente

System.out.println();

System.out.println("Salida del XML: ");

Unmarshaller um = context.createUnmarshaller();

Libreria libreria2 = (Libreria) um.unmarshal(new FileReader(LIBRERIA\_XML));

ArrayList<Libro> lista = libreria2.getListaLibro();

for (Libro libro : lista) {

System.out.println("Libro: " + libro.getNombre() + " de " + libro.getAutor());

}

}

}

**Para saber más**: [JAXB Tutorial for Java XML Binding – The ULTIMATE Guide (PDF Download) (javacodegeeks.com)](https://www.javacodegeeks.com/2014/12/jaxb-tutorial-xml-binding.html#marshal)

### 6.7.5.- Generación automática de clases Java a partir del esquema .xsd.

JAXB es una parte de la plataforma Java SE y una de las APIs de la plataforma Java EE, y es parte del Java Web Services Development Pack (JWSDP). También es uno de los fundamentos para WSIT. JAXB es parte de la versión 1.6 SE.

**La herramienta "xjc" se puede utilizar para convertir un XML Schema** y otros tipos de archivo de esquemas (en Java 1.6, RELAX NG, XML DTD y WSDL son compatibles experimentalmente) **a representaciones de clase**.​

* Las clases son marcadas usando anotaciones del espacio de nombres javax.xml.bind.annotation.\* , por ejemplo, **@XmlRootElement** y **@XmlElement**.
* Las secuencias de listas XML se representan con atributos de tipo java.util.List.
* JAXB incluye la herramienta “schemagen”, que puede llevar a cabo la inversa de "xjc", creando un XML Schema a partir de un conjunto de clases anotadas.

La siguiente tabla muestra las asignaciones de tipos de datos XML Schema (XSD) a partir de tipos de datos Java en JAXB:

|  |  |
| --- | --- |
| **Tipo de XML Schema** | **Tipo de dato Java** |
| **xsd:string** | **java.lang.String** |
| **xsd:positiveInteger** | **java.math.BigInteger** |
| **xsd:int** | **int** |
| **xsd:long** | **long** |
| **xsd:short** | **short** |
| **xsd:decimal** | **java.math.BigDecimal** |
| **xsd:float** | **float** |
| **xsd:double** | **double** |
| **xsd:boolean** | **boolean** |
| **xsd:byte** | **byte** |
| **xsd:QName** | **javax.xml.namespace.QName** |
| **xsd:dateTime** | **javax.xml.datatype.XMLGregorianCalendar** |
| **xsd:base64Binary** | **byte[]** |
| **xsd:hexBinary** | **byte[]** |
| **xsd:unsignedInt** | **long** |
| **xsd:unsignedShort** | **int** |
| **xsd:unsignedByte** | **short** |
| **xsd:unsignedLong** | **java.math.BigDecimal** |
| **xsd:time** | **javax.xml.datatype.XMLGregorianCalendar** |
| **xsd:date** | **javax.xml.datatype.XMLGregorianCalendar** |
| **xsd:g** | **javax.xml.datatype.XMLGregorianCalendar** |
| **xsd:anySimpleType (for xsd:element of this type)** | **java.lang.Object** |
| **xsd:anySimpleType (for xsd:attribute of this type)** | **java.lang.String** |
| **xsd:duration** | **javax.xml.datatype.Duration** |
| **xsd:NOTATION** | **javax.xml.namespace.QName** |

**Los pasos para obtener las clases**:

1. Descargar la distribución: [Download standalone distribution (maven.org)](https://repo1.maven.org/maven2/com/sun/xml/bind/jaxb-ri/2.3.0/jaxb-ri-2.3.0.zip), Download standalone distribution
2. **Descomprimimos el fichero**.
3. **Añadir todas las clases que están en la carpeta** \jaxb-ri-2.3.0\jaxb-ri\lib, jaxb-api.jar, jaxb-core.jar, jaxb-impl.jar, jaxb-jxc.jar, jaxb-xjc.jar
4. **Obtener el esquema .xsd a partir del XML**.
5. Crear un proyecto, y **en la carpeta scr del proyecto situar el esquema y en la carpeta raíz del proyecto situamos el fichero XML**.
6. En Eclipse, **para generar las clases de forma automática** clicamos el botón derecho del ratón sobre el esquema, **generate -> JAXB Classes**.

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

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**: [Hojas de Estilos XSL | Manual de XML (archive.org)](http://web.archive.org/web/20191208125135/https:/www.mundolinux.info/hojas-de-estilo-xls.htm)

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

## 7.1.- Introducción a JasperReport.

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.

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. Está escrita totalmente en Java, su código es abierto y es totalmente gratuita bajo los términos de la licencia GPL (Licencia Pública General).

[JasperReports Library - Browse /jasperreports at SourceForge.net](https://sourceforge.net/projects/jasperreports/files/jasperreports/)

[Presentación de como añadir la librería a Netbeans](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Acceso%20a%20datos/Evaluacion%201/Tema%202/Apuntes/AD02/web/pdf/ad02_cont_r42_descargajasper.pdf).

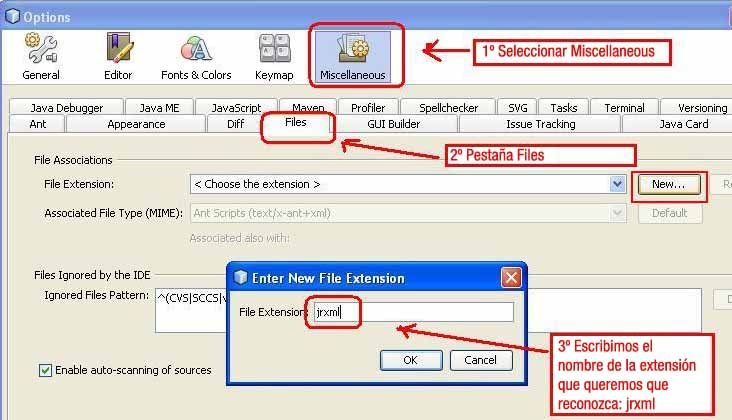
Aal descomprimir el fichero de la descarga, has visto que en el mismo hay varios directorios o carpetas. Comentamos brevemente qué contiene cada una:

* **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**: [Overview (JasperReports 6.20.6 API) (sourceforge.net)](https://jasperreports.sourceforge.net/api/index.html)

## 7.2.- 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.



**Cómo trabajar con JasperReport**:

**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](https://ead.murciaeduca.es/pluginfile.php/1431832/mod_resource/content/4/AD02_Contenidos_Web/72_disear_y_compilar_la_plantilla.html#te30b4ea9-3ca3-fae9-2fb4-2e980058518d)(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.sourceforge.net/dtds/jasperreport.dtd">

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 cargar los 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.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

## 7.3.- 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](https://ead.murciaeduca.es/pluginfile.php/1431832/mod_resource/content/4/AD02_Contenidos_Web/73_rellenar_el_informe_con_datos_exportar_el_informe.html#t8c71fbf0-4a94-141a-cad4-e609e56cc91f)) y la conexión JDBC a la base de datos desde donde se obtendrá la información que necesitemos.

Como resultado de este proceso, se obtiene un objeto que representa un documento listo para ser impreso (ya sea en una impresora, en PDF, en XLS, en CSV, etc.), un objeto serializable de tipo **JasperPrint**.

**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().

**En el siguiente**[**enlace**](https://www.youtube.com/watch?v=SvHWBFrLhPs)**se muestra un video de como intalar JasperReports con Netbeans.**

**Para saber más**: para facilitar el diseño de los diseños de los informes, 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:

[Informes en Java con iReports (archive.org)](http://web.archive.org/web/20120118183221/https:/www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=ireport)

[Crear informes con iReport](http://mundobyte.wordpress.com/2008/01/05/creacion-de-reportes-en-java-parte-i/)

# 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 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());

}

1. Selector: es un objeto que puede gestionar múltiples canales para eventos (como: conexión abierta, datos recibidos, etc.).  Un Selector permite que un solo hilo maneje múltiples canales. [↑](#footnote-ref-1)