Tabla de contenido

¿Qué es un fichero? 3

¿Cómo es un fichero por dentro? 4

Se puede decir que no existe una forma predeterminada de cómo es la estructura de un fichero ya que cada desarrollador puede diseñarlo como quiera. 4

Codificaciones de caracteres 5

Sistema de Ficheros 8

Tipos de ficheros 9

Rutas 10

Ejemplo de árbol de directorios 10

Extensión de un fichero 11

Permisos sobre ficheros 11

Operaciones con ficheros 12

Formas de acceso a un fichero 14

Secuencial 14

Aleatorio 15

Ventajas e inconvenientes de las distintas formas de acceso 16

Clases Java de acceso a un fichero 16

Clases asociadas a operaciones de gestión de ficheros y directorios 17

Librerías específicas utilizadas en la gestión de ficheros 18

1. java.io (Java Input Output) 19

 Clase File (Java.io.File) 21

 Manejo de ficheros en Java con la clase File 23

 Constantes de la clase File 25

 Métodos de la clase File 26

 Creación de un filtro 28

2. java.nio (Java Non-blocking Input Output) 29

 Métodos de la clase Files 31

3. FileUtils 32

Operaciones básicas de ficheros 33

Creación de un fichero 34

 Utilizando java.io.file 34

 Utilizando java.nio.file.File 35

Creación de un directorio 36

 Utilizando java.io.file 36

 Utilizando java.nio.file.File 37

Borrado de un fichero 38

 Utilizando java.io.file 38

 Utilizando java.nio.file.File 39

Copia 41

 Utilizando fileutils 41

 Utilizando java.nio.file.File 42

 Utilizando java.io.file 42

Movimiento 45

 Utilizando fileutils 45

 Utilizando java.nio.file.File 45

 Utilizando java.io.file 46

EJERCICIOS BLOQUE 1 47

Gestión de flujos de datos: Streams – Acceso secuencial 48

Streams binarios o de bytes 51

Constructores InputStream 51

Constructores OutputStream 52

Ejemplo convertir un objeto en un fichero 53

Ejemplo convertir un fichero en un objeto 55

Streams de texto 58

Constructores InputStream 58

Constructores OutputStream 58

Ejemplos 58

EJERCICIOS BLOQUE 2 58

Gestión de flujos de datos - Streams – Acceso Aleatorio 60

Flujos de salida de bytes 60

Flujos de entrada de bytes 64

Flujos de salida de texto 64

Flujos de entrada de texto 64

Control de excepciones 66

Sentencia try-catch-finally 66

Sobreescribir o añadir al final (overwrite or append): 75

TEMA 1 – Manejo de Ficheros

En este tema aprenderemos conceptos básicos que hay que tener en cuenta a la hora de trabajar con ficheros en Java.

# ¿Qué es un fichero?

Un ordenador, lo único que se es capaz de manejar es un conjunto de bits (0, 1). Estos bits se agrupan en la unidad mínima, el byte. Un byte equivale a 8 bits.



Un fichero es una archivo que contendrá un conjunto de caracteres o bytes que se almacenarán en un dispositivo en una ruta y con un nombre y extensión concretos.

En Java se sigue el concepto de un fichero como un conjunto de bytes.

Los ficheros suelen llevar asociado una ruta de acceso, un nombre, y una extensión para saber de que tipo de fichero se trata. El conjunto de los 3 elementos debe ser único.

Los ficheros se guardan en dispositivos de almacenamiento fijo como discos duros, pendrives, tarjetas, CD, etc…

La manera en que se organizan los datos dentro de un fichero depende únicamente del diseñador de ese tipo de archivo.

NOTA: Un archivo y un fichero en el contexto de la informática son exactamente lo mismo.

## ¿Cómo es un fichero por dentro?

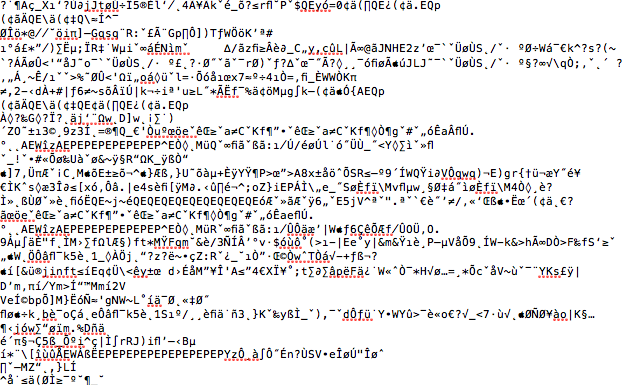
## Se puede decir que no existe una forma predeterminada de cómo es la estructura de un fichero ya que cada desarrollador puede diseñarlo como quiera.

Lo importante es que estará formado por registros.

Los registros son bloques de bytes que guardan la información deseada.

Gracias a la extensión y el software correspondiente el SO puede interpretar el fichero y mostrarlo de forma correcta.

NOTA: Intenta abrir una imagen en formato texto y comprobarás que no es legible.

+

## Codificaciones de caracteres

Las codificaciones de archivos, también denominadas codificaciones de caracteres, especifican cómo se representan los caracteres durante el procesamiento de texto. Una codificación puede ser preferible sobre otra por lo que se refiere a los caracteres del lenguaje que puede o no controlar, aunque normalmente se prefiere Unicode.

Cuando se leen o escriben archivos, la correspondencia incorrecta de las codificaciones de archivo puede producir excepciones o resultados incorrectos. Los caracteres no son legibles y no podemos interpretar la información. Este sería un ejemplo de mala interpretación de un encoding:

H‰ÄTMoÓ@1⁄4ï ̄xGûàõ~z× ̈aÔ ́© ̈(8¤=Ç)AÔn£Â¿ç1⁄2õ¦IH‘‘ƒ3Ÿ3ã

Se denomina ***encoding*** al sistema utilizado para transformar los caracteres que usa cada lenguaje en un símbolo que un ordenador puede interpretar.

Para poder hacer la equivalencia entre byte y símbolo (número, letra, símbolo especial, etc.) se utiliza la tabla ASCII.

La tabla ASCII permite hasta 128 símbolos, ver tabla ASCII en <https://www.ascii-code.com/es.>

Existen varios tipos de codificaciones de caracteres:

* ISO-8859: Este tipo de encoding es bastante conocido y se caracteriza por incluir letras, símbolos y caracteres, pero además también los acentos y los símbolos de interrogación y exclamación. Este tipo de codificación utiliza 8 bits, por tanto, tiene una capacidad de 256 caracteres, lo que la hace más amplia que ASCII. Incluye los 128 caracteres de ASCII, pero se añaden símbolos matemáticos y letras griegas, entre otros. Con el tiempo, se han ido creando diferentes especializaciones de esta codificación. Podemos encontrar también estos otros:
* ISO 8859-1 (Latin-1), para la zona de Europa occidental.
* ISO 8859-2 (Latin-2), para la zona de Europa occidental y Centroeuropa.
* ISO 8859-3 (Latin-3), para la zona de Europa occidental y Europa del sur.
* ISO 8859-4 (Latin-4), para la zona de Europa occidental y países bálticos (lituano, estonio y lapón).
* ISO 8859-5, para el alfabeto cirílico.
* ISO 8859-6, para el alfabeto árabe.
* ISO 8859-7, para el alfabeto griego.
* ISO 8859-8, para el alfabeto hebreo.
* ISO 8859-9 (Latin-5), para la zona de Europa occidental con los caracteres del alfabeto turco.
* ISO 8859-10 (Latin-6), para la zona de Europa occidental, incluye los caracteres del alfabeto nórdico, lapón y esquimal.
* ISO 8859-11, incorpora caracteres del alfabeto tailandés.
* ISO 8859-13 (Latin-7), incorpora caracteres para los idiomas bálticos y el  polaco.
* ISO 8859-14 (Latin-8), incorpora caracteres para los idiomas celtas.
* ISO 8859-15 (Latin-9), añade el símbolo del euro.
* ISO 8859-16, incorpora caracteres para los idiomas polaco, checo, eslovaco, húngaro, albano, rumano, alemán e italiano.
* Unicode: Es una norma de codificación creada en 1991 para unificar los tipos de codificación. Como hemos visto, existen multitud de variantes de codificación para abarcar diferentes idiomas. La creación de este encoding pretendía organizar en un mismo estándar los diferentes caracteres dentro de una misma codificación, para poder abarcar diferentes idiomas tanto de alfabetos europeos como de chinos, japoneses, coreanos o lenguas ya extinguidas con alfabetos diferentes. Para realizar las tablas de equivalencias, “Unicode” asigna un identificador numérico a cada carácter, pero también irá acompañado de información como la direccionalidad, la capitalización y otros atributos. Nuestro ordenador, según su arquitectura, utilizará diferentes bloques de 8, 16 o 32 bits para interpretar y representar los números. Estos tres diferentes bloques han creado diferentes codificaciones:
* UTF-8.
* UTF-16.
* UTF-32.

Hoy en día, la codificación más usada es la codificación de caracteres UTF-8.

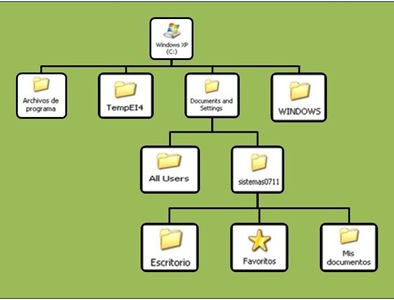
Tanto los editores de texto, como los IDE (programa para desarrollar nuestra aplicación) normalmente dan la posibilidad de configurar qué tipo de codificación queremos usar.

En el caso de que queramos usar diferentes encodings, será necesario usar un fichero binario, es decir, un fichero que almacenará bytes con la información.

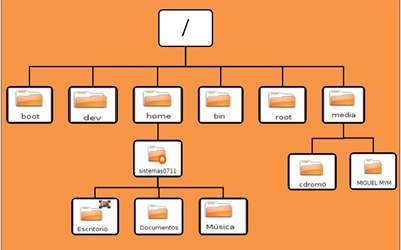
## Sistema de Ficheros

Los ficheros se suelen organizar de forma jerárquica.

* Esta imagen muestra el árbol de archivos en Windows XP:

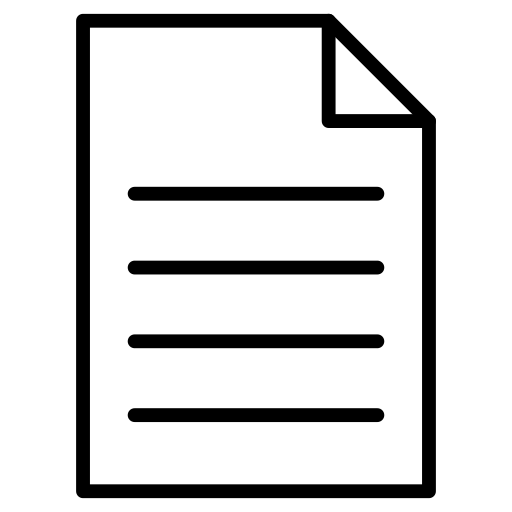
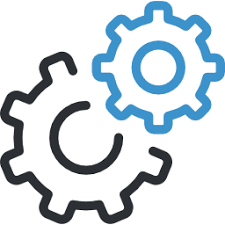


* Este otra imagen muestra el árbol de ficheros en Linux :



## Tipos de ficheros

Existen diferentes tipos de ficheros:

*  Fichero estándar: es un archivo que contiene todo tipo de datos: caracteres, imagen, audio, vídeo, etcétera. Normalmente son ficheros que contienen información de cualquier tipo.
* Directorios o carpetas: son ficheros que albergan más archivos en su interior. Su principal utilidad es mantener un orden o jerarquía en nuestros sistemas.
* Ficheros especiales: son todos esos archivos que usa nuestro sistema operativo y que se utilizan para controlar los dispositivos o periféricos de nuestro ordenador.

En este tema profundizaremos en el tipo de ficheros estándar y en los directorios. Como explicaremos más adelante, este tipo de ficheros nos permitirán realizar diferentes acciones para tratar los ficheros y para mantener un orden y jerarquía con las carpetas.  Podemos destacar dos tipos de ficheros de datos:

* Los ficheros de bytes: también conocidos como ficheros binarios, son archivos que usan los programas para leer o escribir información.
* Los ficheros de caracteres: también conocidos como ficheros de texto, nos permitirán leer o escribir la información que contengan.

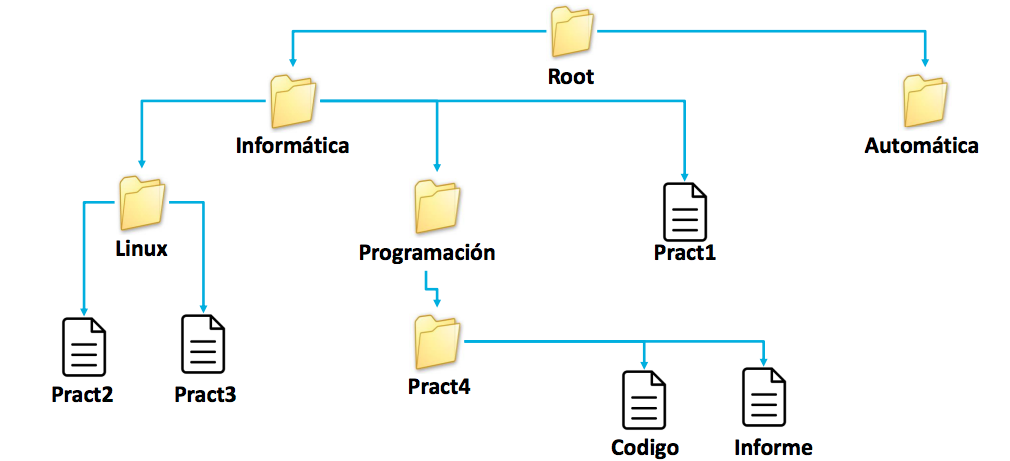
Java puede operar con ficheros de cualquier tipo.

## Rutas

* Para acceder a un determinado fichero, se utiliza la ruta (path). Una ruta indica la dirección del fichero en el sistema de archivos.
* En una ruta, cada nivel de la jerarquía se representa delimitado por el símbolo /. En Windows, el símbolo separador es \ .
* Además de /, existen 2 elementos especiales en la ruta:
* . Un punto representa al directorio actual.
* .. Dos puntos representan al directorio padre en la jerarquía.
* Existen 2 tipos de rutas:
* Absoluta: Ruta al fichero desde el directorio principal (root). Ej: /home/Documentos/ejemplo.txt
* Relativa: Ruta al fichero desde el directorio actual. Ej: Estando en home, ./Documentos/ejemplo.txt

En Java para trabajar con ficheros, se utiliza la ruta relativa parte de la ubicación del fichero ejecutable. Pero también se permiten rutas absolutas.

## Ejemplo de árbol de directorios

****

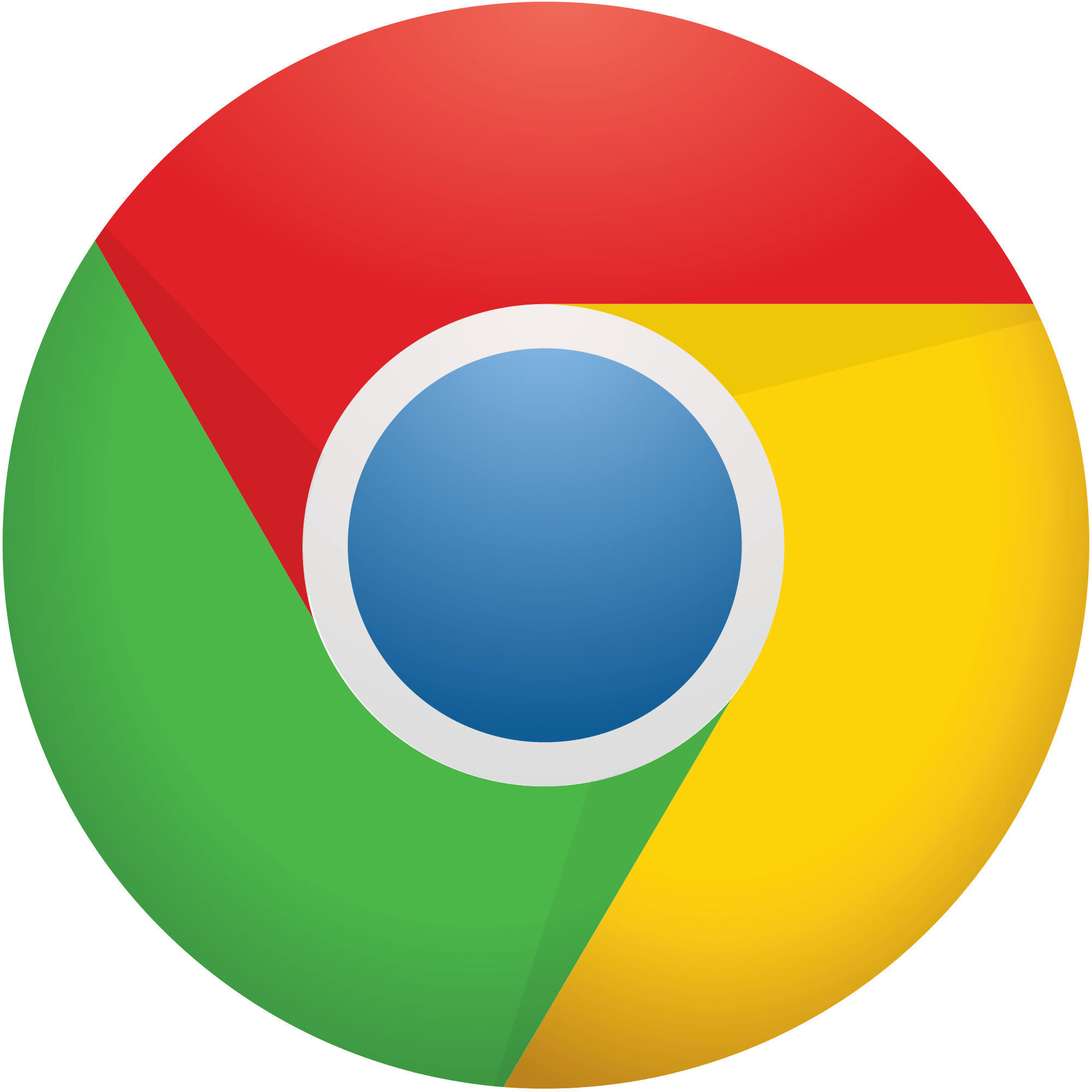
## Extensión de un fichero

La extensión sirve para asociar un determinado fichero con un programa/s que lo ejecuten o interpreten.

* Indica cómo debe ser interpretado el fichero.



* + doc, docx, odt



* + html, xml, jsp



* + mp3, avi, wma
* Se considera extensión lo existente a partir del último punto del nombre del archivo:
  + HolaMundo.doc -> extensión =doc
  + HolaMundo.doc.mp3 -> extensión= mp3

## Permisos sobre ficheros

El usuario que crea el fichero tiene derecho a decidir quien y cómo accede a su fichero.

Existen 3 grupos para los que se les puede definir permisos:

* u. Propietario
* g. Grupo
* o. Resto de usuarios

Los permisos que se pueden dar son los siguientes:

* r. Lectura
* w. Escritura
* x. Ejecución

## Operaciones con ficheros

* Creación: Crea un nuevo fichero con un nombre, extensión y ruta determinados.
* Apertura: El programa abre el fichero y se prepara para leerlo o escribirlo. Suele “reservar” el fichero para sí.
  + Lectura: Lee el fichero o una de sus partes. Se debe disponer del permiso de lectura sobre el fichero. Es una operación NO bloqueante (varios programas pueden leer a la vez el mismo fichero).
  + Escritura: Permite escribir en el fichero, ya sea añadiendo datos o sobrescribiendo los ya existentes. Se debe disponer del permiso de escritura sobre el fichero. Es una operación bloqueante (un solo programa va a poder escribir al mismo tiempo sobre el mismo fichero). La escritura de datos con lleva consigo tres operaciones:
    - Altas: Añadir info al fichero.
    - Modificaciones: Modificar info del fichero.
    - Bajas: Dependiendo del tipo de acceso se realizará de una forma o de otra (ver “*Formas de Acceso a un Fichero”*):



* Bajas Secuenciales: Se crea un nuevo archivo sin los datos que se desean.
* Bajas Aleatorios: Se desactiva el registro para posteriormente cuando se haga una modificación sobreescribirlo.
  + - Valor 1: el registro está activo y no se puede sobreescribir.
    - Valor 0: el registro NO está activo y se puede sobreescribir.

NOTA: Esta segunda opción es más optima porque no hay que copiar todo el fichero de un sitio a otro, pero en esta se gastan más bits. Nos deberíamos de preguntar que es mejor perder mucho tiempo, perder un poco de memoria? Actualmente como la memoria no es problema… mejor perder un poco de memoria.

* Cierre: Indica que se ha finalizado con las operaciones sobre el fichero. Libera el fichero para que el resto de programas o el SO pueda hacer uso del mismo. NOTA: Siempre que abramos un fichero, sockets para comunicaciones de Internet, a controladores de BBDD a cualquier cosa que os podáis imaginar que abráis desde un programa tendréis que cerrar para dejarlo disponible.

En Java se permiten hacer todas las operaciones con ficheros vistas anteriormente.

Antes de meternos de lleno con cada una de las operaciones, tenemos que hablar de las formas de acceso a un fichero.

# Formas de acceso a un fichero

En este apartado, explicaremos las diferentes maneras de acceder a la información de un fichero. Podemos diferenciar dos tipos de acceso: el acceso secuencial y el acceso aleatorio.

## Secuencial

El acceso se produce desde el primer registro y se va avanzando registro a registro para leer la información. Si deseamos acceder al registro 24 debemos leer los 23 anteriores.

Cómo podemos observar son los menos óptimos ya que es obligatorio recorrer de forma constante el fichero desde el inicio.

Un archivo con acceso secuencial es un archivo donde se guardan los registros en orden, con base en el campo clave de origen. Para leer los datos de un fichero secuencial, se debe acceder a los datos uno después de otro y, una vez consultados, no se podrá acceder a ellos si no se sigue el orden.

Un ejemplo puede ser un VHS, (película antigua) donde para ir a un fragmento de la misma debíamos avanzar o rebobinar sobre la cinta electromagnética para leer su información

Aquí tenemos la simplificación a modo de esquema para aclarar el concepto.

## Disco2015:Users:soniadelgadosierra:Documents:Web-docu:TEMA 1 - img:Acceso Secuencial.png

Más adelante, veremos ejemplos prácticos de lectura y escritura de este tipo de acceso de datos.

## Aleatorio

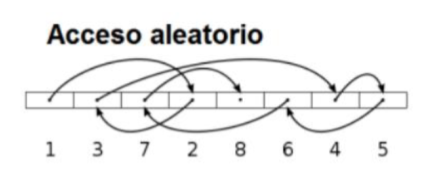
Se puede acceder directamente a un registro sin haber recorrido los anteriores. Casi todos los sistemas actuales utilizan este formato ya que es mucho más rápido que el secuencial.

Un ejemplo es un DVD donde podemos avanzar y retroceder a nuestro antojo desde cualquier punto.

El acceso aleatorio a archivos es un tipo de acceso a datos que permite al programa Java acceder a los datos sin un orden, es decir, a cualquier posición en que se encuentren los datos.

Pasaría lo mismo con un vídeo online, es un fichero al que podremos acceder a cualquiera de sus segundos en cualquier momento sin tener que ver todo el vídeo entero.

A diferencia del acceso secuencial, con el acceso aleatorio no es necesario empezar desde la primera línea del fichero. Se asemeja a los *arrays* de bytes, ya que podemos acceder a cualquier parte de los datos con un puntero de fichero, es decir, indicando al método de acceso en qué posición queremos empezar a tratar los datos. En la siguiente ilustración, podremos apreciar más fácilmente a qué nos referimos con acceso aleatorio de los datos.



Como podemos apreciar, el acceso a los datos es totalmente aleatorio y sin ningún orden, y este tipo de acceso es realmente útil en aquellas aplicaciones de baja latencia que necesitan persistencia. Este tipo de acceso a datos es especialmente útil para situaciones en que el programa ha sufrido un error y es necesario acceder otra vez de manera aleatoria a los datos.

## Ventajas e inconvenientes de las distintas formas de acceso

Las ventajas del acceso de datos secuencial frente al aleatorio es que este tipo de estructura se puede usar para persistir datos de manera más simple.

Por otro lado, las desventajas son evidentes: no es una manera de acceder a los datos de manera eficiente. Nos obliga a acceder a los datos en orden, aunque el resto de datos no nos interesen.

En cambio, el acceso de datos aleatorio proporciona mucho más control y eficiencia a la hora de tratar la información. Por un lado, podemos elegir el punto exacto de lectura de datos, permitiendo ser mucho más rápido y ahorrar consumo de recursos en la aplicación. Como desventaja podemos destacar que puede ser un método de acceso un poco más lento en comparación con el secuencial.

# Clases Java de acceso a un fichero

* Secuencial :
* FileInputStream / FileOutputStream
* FileReader / FileWriter
* Aleatorio o directo :
* RandomAccessFile

# Clases asociadas a operaciones de gestión de ficheros y directorios

Cuando se programa con Java, podemos realizar tareas básicas de gestión de ficheros desde crear hasta leer, borrar, copiar o mover de sitio carpetas o archivos.



Para tratar los diferentes archivos, Java tiene diferentes paquetes que nos pueden ayudar a realizar las operaciones básicas con cualquier tipo de fichero, tal y como veremos a continuación.

Los diferentes objetos que veremos en esta sección pertenecen a diferentes paquetes de Java, algunos de la librería propia de Java (incluidas en el jdk de Java), pero también hay otros que son necesario importación.

## Librerías específicas utilizadas en la gestión de ficheros

En Java, un paquete ó librería se puede entender como un conjunto de clases que facilitan operaciones y tareas ofreciendo al programador funcionalidades ya implementada y listas para ser usadas a través de una Interfaz de Programación de Aplicaciones, API (Application Programming Interface).

Ventaja de los paquetes: Usamos paquetes para evitar conflictos de nombres y para escribir un código más fácil de mantener. Los paquetes se dividen en dos categorías:

* Paquetes integrados (paquetes de la API de Java)
* Paquetes definidos por el usuario (crea tus propios paquetes)

Esto significa que se puede importar una sola clase (junto con sus métodos y atributos) o un paquete completo que contenga todas las clases que pertenecen al paquete especificado.

Para utilizar una clase o un paquete de la biblioteca, se debe utilizar la import palabra clave.

Ejemplo de importación de unas librerías:

import java.io.File;

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

A continuación hablaremos de tres librerías utilizadas en la gestión de ficheros:

* Java.io (Java Input Output)
* Java.nio (Java Non-blocking Input Output)
* FileUtils

### java.io (Java Input Output)

El paquete más utilizado en el lenguaje Java es el paquete ó librería java.io. Se encarga de gestionar las operaciones de entrada y salida de nuestro programa .

Es muy usada, ya que las operaciones System.out y System.err la utilizan para poder mostrar los mensajes.

Trabaja con streams, un flujo de datos (en formato de bytes).

La clase principal se llama File, ya que es la que nos permite abrir o crear ficheros, hay que resaltar que no nos permite ni leer ni escribir pero si la necesitamos para poder realizar estas operaciones..

A continuación se mencionan algunas de las clases más relevantes de esta paquete, java.io :

* java.io.File -> Es la clase File de gestión de ficheros.
* java.io.IOException - > Es un tipo de excepción generalizada que es lanzada cuando algún tipo de operación I/O ha fallado. Esta excepción debe de ser tratada ya sea manejándola con un bloque de código try-catch en el mismo momento que se realice la operación o dejando que se propague hacia arriba en la pila de llamadas utilizando throws IOException.

A continuación haremos hincapié en la clase File, también veremos cómo filtrar ficheros con esta librería java.io, es decir, obtener aquellos ficheros con una característica determinada, como puede ser que tengan la extensión .java, o la que nos interese, y más adelante, también veremos como crear y eliminar ficheros y directorios con esta clase.

#### Clase File (Java.io.File)

La clase File manipula los ficheros en Java. Nos será útil si solo queremos obtener información de un archivo o de una carpeta. Este paquete esta enfocado a la lectura por streams.

Un stream es una librería de java.io que se utiliza para gestionar flujos de datos, ya sea en ficheros, strings o dispositivos. Se encarga de tratar de manera ordenada una secuencia de datos con un origen y un destino.

Los datos que se leen desde este paquete no se guardan en ningún sitio, es decir, no se guardan en caché. Si fuera necesario recorrerlos otra vez, un *stream* no nos sería útil, sino que necesitaríamos cambiar a un búfer, pero ampliaremos información más adelante en el tema.

Con esta clase se pueden hacer un gran número de operaciones sobre un fichero y sus propiedades, pero no se permite leer ni escribir.

La clase File permite realizar un gran número de operaciones:

* Apertura y cierre de un fichero.
* Comprobar si el fichero existe.
* Crear un fichero.
* Borrar un fichero.
* Obtener nombre, rutas (absolutas y relativas) y extensión del fichero.
* Decir si el fichero es un fichero o un directorio.
* Obtener el tamaño en bytes del fichero.
* Consultar y cambiar los permisos del fichero.
* Si es un directorio, obtener la lista de ficheros que contiene el mismo.
* Crear nuevos directorios.
* Renombrar fichero…

Consulta todas la operaciones que se pueden realizar en <https://docs.oracle.com/javase/7/docs/api/java/io/File.html>

El resto de clases que manipulan ficheros parten de la existencia de una clase File, por lo que es la base de cualquier operación de manipulación de ficheros.

#### Manejo de ficheros en Java con la clase File

Para crear un nuevo objeto, debemos declarar una nueva instancia del objeto *File*, al cual le pasaremos el nombre de la ruta, el nombre del fichero que queramos crear y la extensión en un *string*.

Un objeto de la clase File representa un fichero, apunta mediante su ruta a un fichero o directorio del sistema de archivos o también se puede crear un nuevo fichero.

* NO representa el contenido de ningún fichero.

La clase File, nos va a proporcionar una serie de utilidades, evidentemente dependiendo de la plataforma en la que nos encontremos (Linux, Mac o Windows) cambiarán las rutas de los directorios de los archivos, además en los sistemas Linux y Mac habrá que tener en cuenta que en las rutas se distinguen mayúsculas de minúsculas, cosa que no ocurre en los sistemas Windows.

La clase File, puede utilizar cualquiera de los siguientes constructores:

* **File(String path)**
* Linux:

File varFichero = new File (“/directorio/subdirectorio/fichero1.txt”);

* Windows:

File varFichero = new File(“c:\\directorio\\subdirectorio\\fichero1.txt”);

* **File(String path, String name)**
* Linux:

File varFichero= new File(“/directorio/subdirectorio”,”fichero1.txt”);

* **File(File path, String name)**
* Linux:

File varFichero= new File(new File (“/directorio/subdirectorio”),”fichero1.txt”);

Dependiendo de nuestras necesidades podemos utilizar uno u otro.

En Java el separador de archivos tanto para Windows como para Linux es el símbolo /, aunque las rutas se escriben en: Linux / y en Windows \\

Al crear el constructor, la variable varFichero es un objeto con los datos del fichero que se encuentra en la ruta pasada por parámetro, si es que existe:

* La ruta puede ser absoluta o relativa.
* Para cerrar un fichero, se usa la siguiente sentencia:

varFichero.close();

Con esta sentencia se cierra el fichero y la variable varFichero pasará a ser nula (null).

#### Constantes de la clase File

La clase File tenemos un par de constantes muy útiles, que nos servirán si usamos nuestra aplicación en varios sistemas operativos. Nos permitirá mostrar el separador entre ficheros entre rutas de cada sistema operativo, por ejemplo, para Windows, se separan las rutas con “;” y los directorios con “\”. y en Unix/Linux se separan con “:” y “/” respectivamente.

Los métodos son:

* pathSeparator
* separator

String path = File.separator + "var"+ File.separator + "temp"

String listOfFiles = ...

String[] filePaths = listOfFiles.split(File.pathSeparator);

Utilizar estas constantes nos evitará duplicar código a la hora de trabajar con ficheros dependiendo si es un sistema Windows o Unix.

#### Métodos de la clase File

A diferencia de otras clases que manejan ficheros, en la clase File no es necesario controlar excepciones en la mayoría de los métodos. Si hay alguna excepción a manejar lo indicaremos.

Estos son los métodos mas conocidos:

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Descripción | Ejemplo | Dato devuelto |
| Exists () | Indica si existe o no el fichero. |  | boolean |
| isDirectory () | Indica si el objeto File es un directorio. |  | boolean |
| isFile () | Indica si el objeto File es un fichero. |  | boolean |
| isHidden () | Indica si el objeto File esta oculto. |  | boolean |
| getPath () | Obtiene la ruta del fichero, el inicio es el punto de ejecución del programa | File ficheros = new File("ficheros"); System.out.println(ficheros.getPath()); | String |
| getAbsolutePath () | Devuelve una cadena con la ruta absoluta del fichero o directorio. | File ficheros = new File("ficheros"); System.out.println(ficheros. getAbsolutePath()); | String |
| canRead () | Indica si se puede leer. | File ficheros = new File("ficheros"); System.out.println(ficheros. canRead()); | boolean |
| canWrite () | Indica si se puede escribir. | File ficheros = new File("ficheros"); System.out.println(ficheros. canWrite()); | boolean |
| canExecute () | Indica si se puede ejecutar. |  | boolean |
| getName () | Devuelve una cadena con el nombre del fichero o directorio. |  | String |
| getParent () | Devuelve una cadena con el directorio padre. | File ficheros = new File("ficheros"); System.out.println(ficheros. getParent()); | String |
| listFiles () | Devuelve un array de File con los directorios hijos. Solo funciona con directorios. | File ficheros = new File("ficheros"); ficheros.listFiles(); | Array de File |
| list () | Devuelve un array de String con los directorios hijos. Solo funciona con directorios. | File ficheros = new File("ficheros"); ficheros.list(); | Array de String |
| mkdir () | Permite crear el directorio en la ruta indicada. Solo se creara si no existe. | File ficheros = new File("ficheros"); System.out.println(ficheros. mkdir()); | boolean |
| mkdirs () | Permite crear el directorio en la ruta indicada, también crea los directorios intermedios. Solo se creara si no existe. |  | boolean |
| createNewFile () | Permite crear el fichero en la ruta indicada. Solo se creara si no existe. Debemos controlar la excepcion con IOException. | File ficheros = new File("ficheros"); System.out.println(ficheros. createNewFile()); | Boolean |
| delete () | Elimina un fichero o directorio, si es un directorio debe estar vacío | File ficheros = new File("ficheros"); System.out.println(ficheros. delete()); | Boolean |
| renameTo () | Renombra un fichero o directorio | File ficheros = new File("ficheros"); System.out.println(ficheros.renameTo(new File("eliminados"))); | Boolean |

#### Creación de un filtro

Un filtro es un objeto de una clase que implementa el interface *FilenameFilter*, y tiene que redefinir la única función del [interface](http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/herencia/interfaces.htm) denominada *accept*. Esta función devuelve un dato de tipo boolean. En este caso, la hemos definido de forma que si el nombre del archivo termina con una determinada extensión devuelve true en caso contrario devuelve false. La función *endsWith* de la clase *String* realiza esta tarea tal como se ve en la porción de código que viene a continuación. La extensión se le pasa al constructor de la clase *Filtro* para inicializar el miembro dato *extension*.

import java.io.\*;

public class Filtro implements FilenameFilter {

String extension;

Filtro(String extension){

this.extension=extension;

}

public boolean accept(File dir, String name){

return name.endsWith(extension);

}

}

Para obtener la lista de archivos con extensión .java en el directorio actual, creamos un objeto de la clase *Filtro* y se lo pasamos a la función *list* miembro de la clase *File*.

listaArchivos=fichero.list(new Filtro(".java"));

for ( int i=0; i<listaArchivos.length; i++ ) {

System.out.println(listaArchivos[i]);

}

### java.nio (Java Non-blocking Input Output)

Es una librería importante que encontraremos en la versión de Java 8. Es otra alternativa para la creación de ficheros, una de las más recomendadas. Tiene un gran repertorio de métodos para la creación, copia, borrado, escritura y lectura de datos.

Esta segunda librería supone una mejora en la forma en la que se realizan las operaciones. Ambas pueden ser utilizadas, pero java.nio corrigió muchas de las deficiencias de java.io.

Entrando en detalle de este paquete, nos permite un enfoque diferente a la hora de gestionar los datos: java.nio.file permite el almacenaje de la información del fichero en un búfer.

Un búfer es un bloque de memoria que permite almacenar temporalmente los datos y recorrerlos tantas veces como se desee para tratarlos.

Tal y como hemos comentado en la definición de búfer, los datos que se guardan en él se pueden volver a leer según se necesite. Nos proporciona un poco más de flexibilidad a la hora de tratar con un fichero.

A continuación se mencionan algunas de las clases más relevantes de esta paquete:

* java.nio.file.Path -> Sirve para manejar las rutas de los ficheros.
* java.nio.file.Files -> Sirve para controlar las operaciones básicas de los ficheros.
* java.nio.file.FileSystem -> Sirve para obtener referencias al sistema de archivos.

Esta librería consta con métodos para la manipulación de ficheros que nos serán realmente útiles.

Esta clase tiene métodos estáticos (recordad “método estático” se puede acceder a él sin crear un objeto de la clase) para el manejo de ficheros, los métodos de la clase Files trabajan sobre objetos Path.

Las operaciones principales a realizar con archivos y directorios son:

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

#### Métodos de la clase Files

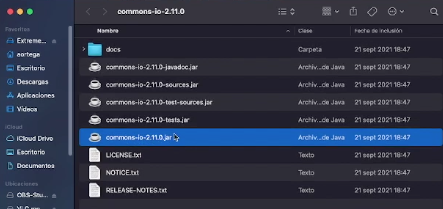
|  |  |  |
| --- | --- | --- |
| Método | Descripción | Dato devuelto |
| createDirectory(path) | Crea un directorio en la ruta indicada. | path |
| createFile(path) | Crea un fichero en la ruta indicada. | path |
| copy(path origen, path destino) | Crea una copia de un fichero origen a una ruta destino que le indiquemos. | path |
| createTempDirectory(String prefijo) | Crea una carpeta temporal en la aplicación con el nombre que le pasemos como string al llamar al método. | Path |
| createTempFile(String prefijo, String sufijo) | Este método se encarga de crear un fichero temporal en la carpeta temporal del programa, usando el prefijo y el sufijo que le hemos indicado por parámetro. | path |
| delete(path) | Borra un fichero de la ruta indicada. | void |
| deleteIfExists(path) | Borra un fichero siempre que exista en la ruta indicada. | boolean |
| exists(path) | Comprueba si el fichero que le indicamos existe en la ruta indicada. | boolean |
| isDirectory(path) | Comprueba si ese fichero indicado en la ruta es una carpeta o no. | boolean |
| size(path) | Devuelve el tamaño del fichero que le indiquemos en la ruta. | long |
| walkFileTree(path) | Este método se encarga de recorrer el árbol de directorios de una ruta recursivamente. | Path |
| readAllLines(path) |  |  |
| readAllBytes(path) |  |  |
| write(path, byte[]) |  |  |
| writeString(path,CharSequence) |  |  |

### FileUtils

Para poder realizar más operaciones sobre los ficheros, disponemos de la clase FileUtils, de la librería de Apache Commons.

No viene con el jdk por defecto por lo que si deseamos utilizarla debemos añadirla de forma manual a nuestro proyecto. Se puede descargar buscando en el navegador “fileutils download jar” y llegamos a “Download Apache Commons IO“. En la sección

Apache Commons IO 2.16.1 (requires Java 8) -> Binaries nos bajamos el .gz ó el .zip haciendo click. Y se nos abrirá una pantalla como esta:



El fichero .jar seleccionado en la imagen tendremos que añadir a nuestro proyecto como una libreria. Depende del IDE (Visual Code, NetBeans, …) de trabajo se añadirá de una forma u otra.

Para explorar todas las funciones que admite FileUtils consultar [FileUtils.html](https://commons.apache.org/proper/commons-io/apidocs/org/apache/commons/io/FileUtils.html).

Está nos ofrece un conjunto de mecanismos para poder controlar los ficheros de forma muy sencilla, por ejemplo las funciones:

* Copia
* Movimiento

## Operaciones básicas de ficheros

Existen 4 operaciones sobre los ficheros esenciales para la gestión de los mismos:

* Creación
* Eliminación
* Copia
* Movimiento

Por desgracia no todas ellas se pueden realizar con el API básico de Java.io.

Existe otra operación que no está incluida en este bloque pero me gusta hacer hincapié en ella, el renombramiento de ficheros:

* rename

import java.io.File;

public class fileRename {

public static void main (String[] args) {

System.out.println("Iniciando programa...");

File file = new File("ficheros/fileIO\_1.txt");

File rename = new File("ficheros/fileIO\_2.txt");

if (file.renameTo(rename)){

System.out.println("Fichero renombrado con éxito");

}else{

System.out.println("Fichero NO renombrado");

}

System.out.println("Finalizando programa...");

}

}

A continuación trataremos las distintas operaciones que se pueden realizar con las distintas librerías.

### Creación de un fichero

#### Utilizando java.io.file

A continuación, veremos cómo crear un archivo *File* pasándole como parámetro al constructor la ruta absoluta, el nombre y la extensión del fichero que queremos crear. El constructor, lo que realmente necesita es un *string*, que será la ruta compuesta.

import java.io.File;

import java.io.IOException;;

public class fileCreator {

public static void main (String[] args) {

System.out.println("Iniciando programa...");

File directorio= new File("ficheros");

if (!directorio.mkdir()){

System.err.println("No se ha podido crear el directorio, puede que exista");

}

File ficheroDatos = new File(directorio,"fichero1.txt");

try{

if (ficheroDatos.createNewFile()){

System.out.println("Creado el fichero "+ ficheroDatos);

}

else {

System.out.println("No se ha creado el fichero, puede que exista.");

}

}catch(IOException ex){

System.err.println("No se ha podido crear el fichero");

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

System.exit(-1);

}

System.out.println("Finalizando programa...");

}

}

#### Utilizando java.nio.file.File

En este caso no utilizamos un *string* para definir la ruta del fichero que queremos crear, sino que utilizamos el objeto *Path*. Este es un objeto que se utiliza para localizar un fichero dentro de un sistema de archivos y se encargará de representar una ruta del fichero de nuestro sistema. La librería *Files* necesitará que se le pase este objeto para crear el fichero. Para crear el fichero, llamaremos al método *createFile()*, el cual devuelve un *path* con la ruta del fichero creado.

Al definir el *path*, tenemos que asegurarnos de que la ruta absoluta que utilizamos sea correcta, si no es así, se producirá un error cuando el programa ejecute esta parte del código porque no encontrará la ruta de carpetas indicada.

public class fileCreatorNIO {

public static void main (String[] args) {

System.out.println("Iniciando programa...");

Path directorio = Paths.get("./ficheros/fileNIO\_1.txt");

if (Files.notExists(directorio)) { //Comprueba que no exista el fichero

try{

directorio = Files.createFile(directorio); //Crea el fichero

System.out.println("Creado el fichero "+ directorio);

}catch(IOException ex){

System.err.println("No se ha creado el fichero");

System.exit(-1);

}

}

System.out.println("Finalizando programa...");

}

}

### Creación de un directorio

#### Utilizando java.io.file

Para la creación de directorios se usa la misma clase *File* que hemos comentado anteriormente, pero el método para crearlo es algo distinto a la creación de ficheros. Si nos fijamos en el ejemplo:

import java.io.File;

public class dirCreatorIO {

public static void main (String[] args) {

System.out.println("Iniciando programa...");

File directorio= new File("directorioIO1");

if (directorio.mkdir()){ //Crea el directorio

System.out.println("Creado el directorio"+ directorio);

} else {

System.out.println("No se ha podido crear el directorio");

}

System.out.println("Finalizando programa...");

}

}

Como podemos ver, para crear un directorio el objeto *File* tiene un método que permite la creación de directorios: *mkdir()*. En el caso de que se realice con éxito, el método devuelve un booleano *true*.

Algunos de los problemas que nos podemos encontrar con este método son que, si no existe la carpeta indicada, o si la ruta absoluta que utilizamos para crear nuestro directorio no está bien, dará un error en la ejecución de nuestro programa. Debemos tener en cuenta esta posibilidad y controlar los errores.

#### Utilizando java.nio.file.File

El objeto java.nio.file.Filestambién nos ofrece posibilidad de creación de directorios a través del método *createDirectories()*, al cual le tendremos que pasar la ruta, como en otros objetos. Aquí podemos apreciar un ejemplo de cómo usar este objeto con el método de creación de carpetas.

public class dirCreatorNIO {

public static void main (String[] args) {

System.out.println("Iniciando programa...");

String nombre = ".pepe/directorioNIO";

Path ruta = Paths.get(nombre);

try{

Files.createDirectories(ruta);

System.out.println("Creado el directorio"+ ruta);

}catch(IOException ex){

System.err.println("Se ha producido un error");

System.exit(-1);

}

System.out.println("Finalizando programa...");

}

}

Este tipo de tareas son especialmente útiles para diferenciar carpetas y tener organización dentro de un ordenador o un SFTP para gestionar archivos que se van generando.

Un SFTP (protocolo de transferencia de archivos) es un programa estandarizado para transferir archivos entre ordenadores de cualquier sistema operativo.

Es un protocolo cliente servidor que será útil para transferir archivos que, por ejemplo, un usuario sube en una página web y que queremos transferir a un servidor para que queden almacenados. Una de las posibles utilidades sería crear carpetas por día, y crear nuevos archivos en esa carpeta.

### Borrado de un fichero

#### Utilizando java.io.file

En el objeto *File*, tenemos dos métodos que nos serán realmente útiles para el borrado:

* delete(): borrará el fichero que le indiquemos según la ruta o lanzará un error si no lo encuentra.

import java.io.File;

public class fileDeleteIO {

public static void main (String[] args) {

System.out.println("Iniciando programa...");

File ficheroDatos = new File("./ficheros/fileIO\_1.txt");

if(ficheroDatos.delete()) {

System.out.println("Hemos borrado un fichero, "+ ficheroDatos);

}else{

System.out.println("No se puede borrar el fichero, " + ficheroDatos);

}

System.out.println("Finalizando programa...");

}

}

Como podemos ver, no es necesario crear un nuevo objeto para borrar el fichero en cuestión, solo necesitamos indicarle la ruta y el nombre del fichero, aunque también existe la posibilidad de borrar un fichero ya creado con el mismo método. En Java hay muchas maneras de poder borrar ficheros, os mostramos las más utilizadas.

* deleteOnExit(): este método, en cambio, borrará el fichero o el directorio que le indiquemos por ruta absoluta cuando la máquina virtual finalice.

File archivo = new File("src/main/resources/crearFicheros fichero4.txt");

archivo.deleteOnExit()) {

System.out.println("Hemos borrado un fichero");

}

#### Utilizando java.nio.file.File

La librería java.nio.Files también nos ofrece la posibilidad de borrar ficheros, aunque el procedimiento es algo distinto. También cuenta con un método *delete()* para borrar los ficheros. Para realizar un borrado, se debe realizar de este modo:

Para el borrado de directorios, se puede usar los mismos métodos que hemos indicado anteriormente. En el caso de directorios, antes de borrar se debe comprobar que esté vacío, si no, no va a borrarse y se lanzara una excepción *NoSuchFileException*. Hablaremos de las excepciones más adelante.

El objeto java.nio.files.Filetambién ofrece posibilidad de borrado de carpetas.

String nombrePath = "src/main/resources/crearFicheros/directorio";

Path carpeta = Paths.get(nombrePath);

Files.delete(carpeta);

//Primero crearemos una carpeta, en este caso se crea porque no existe y sino nos dara error.

String nombrePath = "src/main/resources/crearFicheros/directorio";

File ficheroCarpeta = new File(nombrePath); ficheroCarpeta.mkdir();

//Indicamos la ruta con el directorio y el fichero que queremos crear

String nombrefichero = "src/main/resources/crearFicheros /ejercicio/fichero.txt";

File ejemplo = new File(nombrefichero);

ejemplo.createNewFile();

//Cogemos la ruta y lo borramos. Si no existe dara error

Path path = Paths.get(nombrefichero);

try {

Files.delete(path);

}catch (IOException e) {

System.out.println("No existe la carpeta" + ficheroCarpeta);

}

### Copia

Otra de las acciones que podemos realizar con los ficheros o directorios es el copiado. Este método nos será útil para copiar ficheros de un origen a un destino, y tratarlos sin modificar los datos de origen.

Tal y como hemos mostrado anteriormente, veremos ejemplos de los dos objetos Java más utilizados que disponen del método para copiar ficheros.

#### Utilizando fileutils

import java.io.File;

import java.io.IOException;

import org.apache.commons.io.FileUtils;

public class filecopyFU {

public static void main (String[] args) {

System.out.println("Iniciando programa...");

File destDir = new File("./directorioIO");

File srcFile = new File("./ficheros/fileIO\_2.txt");

try{

FileUtils.copyFileToDirectory(srcFile, destDir);

System.out.println("Fichero copiado");

} catch (IOException ex)

{ System.err.println("Error al copiar el archivo");

ex.printStackTrace();

}

System.out.println("Finalizando programa...");

}

}

#### Utilizando java.nio.file.File

Para empezar, la manera más fácil de copiar un archivo es con la API de java.nio. El objeto *Files* utiliza el método *copy()*, el cual necesita que se le pase por parámetro la ruta de origen y la ruta de destino, con un objeto *Path*. La copia fallará si el fichero destino ya existe, pero si utilizamos la opción *REPLACE\_EXISTING* el fichero se sobrescribirá. Veremos en el ejemplo cómo usar esta opción.

Este método también se usa para los directorios, pero debemos tener en cuenta que solo se copiará la carpeta, no el contenido de esa carpeta.

Path origen = Paths.get("src/main/resources/copiarFicheros/ejemploCopia.txt");

Path ejemploOrigen = Files.createFile(origen);

Path destino = Paths.get("src/main/resources/copiarFicheros/ /destino");

Path ejemploDestino = Files.createFile(destino);

Files.copy(origen, destino, StandardCopyOption.REPLACE\_EXISTING);

#### Utilizando java.io.file

Por otro lado, el objeto *Files* que se encuentra en la API java.io ofrece también la opción de copiar ficheros. Actúa de una manera parecida, pero se implementa de modo distinto. Para realizar el copiado, es necesario crear búferes.

Aquí podemos ver un ejemplo práctico:

//Primero se crean los ficheros

File archivoOrigen = new File("src/main/resources/copiarrFicheros/origen.txt");

archivoOrigen.createNewFile();

File archivoDestino = new File("src/main/resources/copiarFicheros/destino.txt");

archivoDestino.createNewFile();

try {

// Se lee el origen

InputStream origen = new BufferedInputStream(new FileInputStream(archivoOrigen));

// Fichero destino

OutputStream destino = new BufferedOutputStream(new FileOutputStream(archivoDestino));

byte[] buffer = new byte[1024];

int lengthRead;

while ((lengthRead = origen.read(buffer)) > 0) {

// se escriben los datos de un fichero a otro

destino.write(buffer, 0, lengthRead);

// Se cierra el proceso

destino.flush();

}

// Ejemplo de copiado de datos con la api java.nio

Path orig = Paths.get("src/main/resources/copiarFicheros/ejemploCopia.txt");

Path ejemploOrigen = Files.createFile(orig);

Path dest = Paths.get("src/main/resources/copiarFicheros/destino");

Path ejemploDestino = Files.createFile(dest);

Files.copy(orig, dest, StandardCopyOption.REPLACE\_EXISTING);

System.out.println("se ha realizado la copia de ficheros");

} catch (Exception e) {

e.getCause();

}

Tal y como vemos en el ejemplo, primero tendremos que crear un archivo origen y otro destino, con las rutas correspondientes. A continuación, necesitamos crear un *InputStream* con un nuevo *BufferedInputStream*, que se encargará de leer los datos del archivo de origen. Profundizaremos más adelante en estas clases, pero las clases *InputStream* y *BufferedInputStream* se caracterizan por leer archivos, y en este ejemplo nos serán útiles para coger el contenido del fichero origen y traspasarlo al fichero destino. En cambio, las clases *OutputStream*, *BufferedOutputStream* y *FileOutputStream* se encargan de escribir ficheros.

Para el objeto destino, debemos crear un *OutputStream* que contendrá un *BufferedOutputStream* para albergar el resultado de la copia. Con un bucle, recorreremos el fichero origen dato a dato y lo escribiremos al objeto destino. Al finalizar, se llamará al método *flush()* para terminar con el proceso. Este método se encargará de vaciar el *OutputStream* y se guardarán los archivos de salida en un búfer.

Para realizar la copia de los archivos es necesario muchas más líneas de código en comparación con el otro ejemplo, por eso recomendamos usar la API java.nio.

### Movimiento

Java proporciona funciones para mover ficheros entre carpetas. Hay diferentes maneras de hacerlo, y veremos las posibilidades que tenemos según la API que queramos usar.

#### Utilizando fileutils

import java.io.File;

import java.io.IOException;

import org.apache.commons.io.FileUtils;

public class fileMoveFU {

public static void main (String[] args) {

System.out.println("Iniciando programa...");

File destDir = new File("./directorioIO");

File srcFile = new File("./ficheros/fileIO\_2.txt");

try {

FileUtils.moveFileToDirectory(srcFile, destDir, true);

System.out.println("Fichero movido");

} catch (IOException ex) {

System.err.println("Error al mover el archivo");

ex.printStackTrace();

}

System.out.println("Finalizando programa...");

}

}

#### Utilizando java.nio.file.File

La API java.nio es la que ofrece una opción más rápida para mover un fichero. Usa el método *move()*, al cual le tendremos que pasar por parámetro el origen e indicarle un destino. Como se hacía en la copia, tenemos la posibilidad de indicarle por parámetro *REPLACE*\_EXISTING. Este parámetro le indica al método que, si existe un dichero con ese nombre, lo deberá sobrescribir. Aquí tenemos un ejemplo de cómo se debe implementar:

Path destino = Files.move(Paths.get("c:/ejercicio/ejemplo.txt"), Paths.get("c:/ejercicio/destinoEjemplo.txt"), StandardCopyOption.REPLACE\_EXISTING);

#### Utilizando java.io.file

Otra manera de realizar esta acción es con el objeto *File* de la API java.io. Este objeto tiene una forma más rudimentaria de realizar el movimiento. Básicamente, se encarga de renombrar el fichero a uno nuevo y borrar el fichero de origen. Recomendamos por su rapidez usar el objeto *Files*.

File origenArchivo = new File("src/main/resources/crearFicheros/carpetaEjemplo/ficheroOrigen.txt");

origenArchivo.createNewFile();

if(origenArchivo.renameTo(new File(src/main/resources/crearFicheros/destino.txt"))){

origenArchivo.delete();

System.out.println("Se ha movido el fichero.");

} else {

System.out.println("Se ha producido un error.");

}

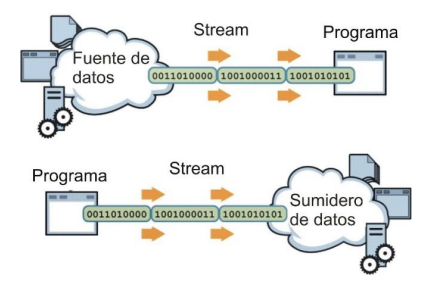
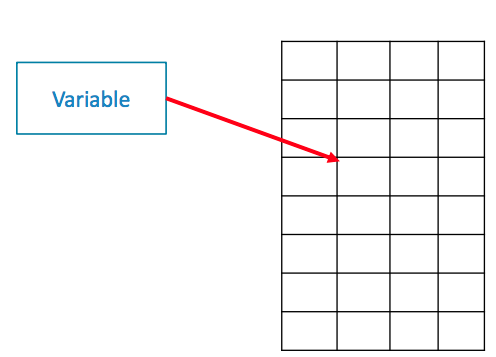
# EJERCICIOS BLOQUE 1

1. Crea un directorio llamado ”dam-accesodatos”
2. Crea un fichero llamado ejercicio1, dentro del directorio “dam-accesodatos”
3. Muestra por pantalla la longitud del fichero con nombre “ejercicio1”
4. Crea un fichero llamado ejercicio2 , dentro del directorio “dam-accesodatos”
5. Muestra todos los ficheros del directorio “dam-accesodatos”
6. Elimina el fichero llamado ejercicio1
7. Muestra todos los ficheros del directorio “dam-accesodatos”
8. Elimina nuevamente el fichero llamado ejercicio1.

# Gestión de flujos de datos: Streams – Acceso secuencial

Para poder realizar las operaciones de lectura y escritura de ficheros, Java establece lo que se conoce como un Stream de datos.

Crea una vía de comunicación entre programa y fichero que permite “moverse” por las distintas partes del fichero. Existe un puntero que apunta a las partes del fichero.



Los flujos de datos (streams en inglés) son una abstracción empleada en muchos lenguajes de programación, entre ellos Java, para representar cualquier fuente que produzca o consuma información.

Su nombre (flujo) viene de que pueden considerarse como una cadena de datos continúa con una longitud que, posiblemente, es desconocida, al igual que la "longitud" de un flujo de agua que está corriendo.

Los flujos de datos o streams representan cualquier fuente que proporcione datos al programa, o cualquier sumidero que tome datos del programa.

Existen dos tipos de flujos de datos:

* los binarios o de bytes: la información que fluye está en formato binario. Se basan en enviar la información en bloques de 8 bits, 1 byte, también son conocidos como ficheros binarios.

*“Un fichero Word es un*

*fichero BINARIO. Además de caracteres tiene tablas, imágenes… si se abre con el notepad no será ilegible”*

**

*“Un fichero PDF también es un fichero BINARIO.”*

* los de caracteres o de texto: la información que fluye está en formato texto. Se basan en enviar la información en bloques de 16 bits, 2 bytes, esto es debido a la codificación Unicode que usa 16bits para representar cada carácter(UTF-16), también conocidos como archivos de caracteres.

Cada una de estas dos categorías puede volver a subdividirse en:

* flujos de datos de entrada: serían flujos que nos proporcionan datos, es decir, entradas de nuestro programa.
* flujos de datos de salida: flujos en los cuales nuestro programa escribe datos, es decir, salidas de nuestro programa.

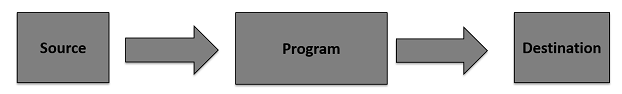
Es muy importante determinar que tipo de stream necesitamos y hacer la programación adecuada. La elección nos hará utilizar unas clases u otras.

En este apartado vamos a ver las principales clases para representar flujos binarios y de texto, de entrada y de salida que proporciona Java.

## Streams binarios o de bytes

Los streams de 8 bits, 1 byte, utilizan las clases:

* InputStream para entrada de datos: nos permite leer bytes de un array, de un String, de un fichero, etc.
* OutputStream para salida de datos: nos permite todo lo contrario hacer salida de datos a un array o un fichero.



### Constructores InputStream

ByteArrayInputstream

Permite leer un array de bytes de la RAM

ObjectInputStream

Permite convertir bytes en objetos

FileInputStream

Permite leer un fichero de bytes

InputStream

* ByteArrayInputStream:
* ByteArrayInputStream(byte[] buf)
* ByteArrayInputStream(byte[] buf, int offset, int lenght)
* ObjectInputStream:
* ObjectInputStream(InputStream in)
* FileInputStream:
* FileInputStream(String name)
* FileInputStream(File file)

buf -> Buffer de entrada

offset -> Desplazamiento inicial dentro del buffer

lenght -> Número máximo de bytes leídos del buffer

in -> InputStream con los datos para leer

name -> Nombre de fichero

file -> Fichero de entrada

### Constructores OutputStream

Permitir convertir objetos en bytes

Permitir escribir bytes en ficheros

FileOutputStream

La clase debe implementar la interfaz

Serializable

Permitir escribir un array de bytes almacenado en la RAM

ByteArrayOutputStream

ObjectOutputStream

OutputStream

* ByteArrayOutputStream:
* ByteArrayOutputStream(int size)
* ObjectOutputStream:
* ObjectOutputStream(OutputStream output)
* FileOutputStream:
* FileOutputStream(String name)
* FileOutputStream(String name, boolean append)
* FileOutputStream(File file)
* FileOutputStream(File file, boolean append)

size -> Capacidad inicial

output -> OutputStream donde escribir los datos

name -> Nombre del fichero de salida

append -> opción para escribir al inicio o final de fichero

file -> Fichero de salida

### Ejemplo convertir un objeto en un fichero

private static void guardarObjeto(File file){

try{

file.createNewFile();

}catch (IOException ex){

System.err.println("No se ha podido crear el fichero " + file.getName());

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

System.exit(-1);

}

//1.Crear el objeto que deseamos guardar

//Pedimos los datos de Persona por consola

Scanner scanner= new Scanner(System.in);

System.out.println("Introduce el id de la persona");

int id=scanner.nextInt();

scanner.nextLine();

System.out.println("Introduce el nombre de la persona");

String nombre=scanner.nextLine();

System.out.println("Introduce la edad de la persona");

int edad= scanner.nextInt();

scanner.nextLine();

System.out.println("Introduce la dirección de la persona");

String direccion=scanner.nextLine();

Persona persona= new Persona(id,nombre,edad,direccion);

//2.Crear el fileOutStream para guardar información en el fichero -> flujo de datos de salida

FileOutputStream fileOutputStream=null;

try{

fileOutputStream=new FileOutputStream(file);

}catch(FileNotFoundException ex){

System.err.println("No se ha encontrado el fichero: " + file.getName());

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

System.exit(-2);

}

//Convertir el objeto en bytes

ObjectOutputStream objectOutputStream=null;

try{

//Utiliza el fileOutputStream para saber donde escribir

objectOutputStream=new ObjectOutputStream(fileOutputStream);

}catch(IOException ex){

System.err.println("No se ha podido crear el ObjectOutputStream:");

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

System.exit(-3);

}

//Guardar los bytes en el fichero

try{

//Escribimos el objeto en el fichero ligado al FileOutputStream

objectOutputStream.writeObject(persona);

}catch (IOException ex){

System.err.println("No se ha podido escribir en el fichero: " + file.getName());

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

System.err.println(ex);

System.exit(-4);

}

//Cerrar el fichero

try{

fileOutputStream.close();

objectOutputStream.close();

}catch (IOException ex){

System.err.println("No se ha podido cerrar correctamente el fichero: " + file.getName());

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

System.exit(-5);

}

}

### Ejemplo convertir un fichero en un objeto

private static void recuperarObjeto(File file){

FileInputStream fileInputStream=null;

try {

//De donde debe leer

fileInputStream=new FileInputStream(file);

} catch (FileNotFoundException ex) {

System.err.println("Error al abrir el FileInputStream");

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

System.exit(-6);

}

ObjectInputStream objectInputStream=null;

try {

//Utiliza el FileInputStream para saber de donde leer

objectInputStream=new ObjectInputStream(fileInputStream);

} catch (IOException ex) {

System.err.println("Error al crear el ObjectInputStream");

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

System.exit(-7);

}

try {

//Lee el fichero y lo convierte en una persona

//IMPORTANTE:Hacer el casteo ->(Persona)

Persona persona=(Persona) objectInputStream.readObject();

System.out.println(persona);

} catch (IOException | ClassNotFoundException ex) {

System.err.println("Error al recuperar el objeto del fichero");

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

System.exit(-8);

}

try {

objectInputStream.close();

fileInputStream.close();

} catch (IOException ex) {

System.err.println("Error al cerrar los InputStream");

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

System.exit(-9);

}

}

En la clase Main implementaríamos lo siguiente:

public static void main(String[] args) {

//Crear el fichero donde guardaremos el fichero

File file=new File("ficheros/objeto");

guardarObjeto(file);

recuperarObjeto(file);

}

Y también deberíamos implementar la clase Persona.

package ficherosBinarios;

import java.io.Serializable;

public class Persona implements Serializable{

private int id;

private String nombre;

private int edad;

private String direccion;

public Persona(int id, String nombre, int edad, String direccion) {

this.id = id;

this.nombre = nombre;

this.edad = edad;

this.direccion = direccion;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

public int getEdad() {

return edad;

}

public void setEdad(int edad) {

this.edad = edad;

}

public String getDireccion() {

return direccion;

}

public void setDireccion(String direccion) {

this.direccion = direccion;

}

@Override

public String toString() {

return "Persona [id=" + id + ", nombre=" + nombre + ", edad=" + edad + ", direccion=" + direccion + "]";

}

}

NOTA: Para guardar un objeto en un fichero la clase que deseamos guardar o el objeto que deseamos guardar debe implementar la interfaz Serializable .

NOTA: Existen diferentes tipos de Exceptions que deben ser controladas:

* IOException: Excepción producida al leer o escribir datos.
* FileNotFoundException: Excepción producida cuando no encuentra el fichero solicitado .
* ClassNotFoundException: Excepción producida cuando la JVM no es capaz de recuperar la clase que se está indicando.

## Streams de texto

Los streams de 16 bits, 2 bytes, orientados a caracteres utilizan unas clases que ya están preparadas para esta lectura de datos.

* Reader
* Writer

De estás clases utilizaremos las implementaciones de FileReader y FileWriter para nuestras operaciones de lectura y escritura respectivamente para tratar los ficheros.

Al crear un objeto de la clase FileWriter se puede pasar un segundo parámetro de tipo boolean, este especificará si escribirá en el fichero desde el inicio (false) o continuará desde el último punto (true).

### Constructores FileWriter

* FileWriter(String name)
* FileWriter(String name, boolean append)
* FileWriter(File file)
* FileWriter(File file, boolean append)
* FileWriter(String name, Charset charset)
* FileWriter(String name, Charset charset, boolean append)
* FileWriter(File file, Charset charset)
* FileWriter(File file, Charset charset, boolean append)

### Constructores FileReader

* FileReader(String name)
* FileReader(String File)
* FileReader(String name, Charset charset)
* FileReader(File file, Charset charset)

charset -> Es la codificación de los caracteres. Nos permite leer caracteres que de forma inicial no están previstos como símbolos de otros lenguajes.

output -> OutputStream donde escribir los datos

name -> Nombre del fichero de salida

append -> opción para escribir al inicio o final de fichero

file -> Fichero de salida

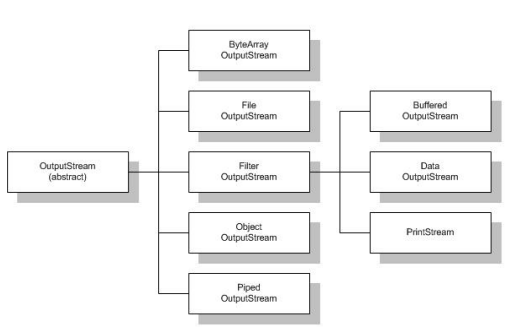
### Ejemplos

# EJERCICIOS BLOQUE 2

1. Crea una nueva clase llamada Persona con los atributos (id,nombre,edad, dni).
   1. Crea una función para guardar un objeto Persona en un fichero con el nombre persona1
   2. Crea una función para recuperar un objeto Persona del fichero persona1
   3. Modifica sus propiedades y vuelve a guardarlo en el fichero persona 1
2. Crea un fichero de texto utilizando la clase FileWriter
   1. El fichero debe contener la información: “Esto es un texto de prueba, Estamos creando nuestro primer fichero de texto https://codigonline.com”
3. Lee un fichero de tipo imagen y muestra su contenido por pantalla.
   1. ¿Se puede leer de forma correcta?

# Gestión de flujos de datos - Streams – Acceso Aleatorio

## Flujos de salida de bytes



Jerarquía de los flujos de salida de bytes de Java.

Como podemos observar en la figura, la clase padre de todos los flujos de salida de Java es OutputStream. Se trata de una clase abstracta (por tanto no vamos a poder crear objetos de ella porque su funcionalidad está "incompleta") que representa un flujo de datos de salida binario cualquiera.

Los métodos de la clase OutputStream son los siguientes:

* close(): Cierra el flujo de datos y libera cualquier recurso que el flujo de datos pudiera estar consumiendo. Por ejemplo, este método permite liberar los recursos del sistema operativo consumidos por un fichero al cual hemos terminado de escribir información.
* flush(): Sincroniza este flujo de datos con el dispositivo al cual se están escribiendo los datos. Este método es necesario porque, habitualmente, los OutputStream tienen un buffer de datos interno al cual van escribiendo la información antes de enviarla al dispositivo de entrada y salida correspondiente. De este modo, se pueden agrupar varias operaciones de escritura y efectuarlas de un modo más eficiente. Este método hace que los últimos cambios realizados sobre el buffer de memoria se sincronicen con el dispositivo de entrada y salida.
* write(byte[] b): Escribe el array b de bytes que se le pasa como argumento al flujo de salida.
* write(byte[] b, int off, int len): Escribe len bytes del array b al flujo de salida, empezando a escribirlos en el offset indicado por off .
* abstract void write(int b): Escribe 1 byte al flujo de salida. Dado que la clase OutputStream es abstracta nunca vamos a poder crear un objeto de ella. Tendremos que crear objetos de alguna de sus subclases que, como podemos ver en la figura 2, son bastantes. Resumamos, brevemente, cuál es el propósito de cada una de ellas:
* ByteArrayOutputStream: Como su nombre indica, este flujo de salida representa un array de bytes que se almacena en memoria. A partir de él puede obtenerse una cadena de caracteres que represente todos los datos escritos. Obviamente, no soluciona el problema de almacenar información de modo persistente.
* FileOutputStream: Flujo de salida para la escritura de datos a un objeto de tipo File.
* FilterOutputStream: Esta clase encapsula a otro objeto de tipo OutputStream e intercepta todas las operaciones de escritura para, posiblemente, realiza alguna transformación sobre los datos que se están escribiendo. Una de sus subclases, DataOutputStream: Resulta particularmente útil para escribir distintos tipos de datos primitivos a un flujo de datos de salida.
* ObjectOutputStream: Encapsula otro objeto de tipo OutputStream y permite escribir objetos Java completos al flujo de datos de salida representado por el OutputStream correspondiente.
* PipedOutputStream: Junto con la clase PipedInputStream permite emplear pipes para comunicar (habitualmente) dos threads. Como podemos observar, cada una de las clases hijas de OutputStream proporciona una funcionalidad más específica a la clase padre. Dos de ellas (FilterOutputStream y ObjectOutputStream) actúan como decoradores sobre cualquier OutputStream; es decir, encapsulan un objeto de tipo OutputStream y le proporcionan funcionalidad adicional (aplicar algún tipo de filtrado sobre los datos, o permitir escribir objetos Java). Cuando queremos escribir datos en binario a un archivo lo más habitual es comenzar creando un objeto File que represente a dicho archivo y creando un flujo de datos de salida mediante la clase FileOutputStream:

File f2 = new File ("C:/datos.dat");

FileOutputStream out = new FileOutputStream(f2);

El siguiente paso depende de qué tipo de datos queramos escribir. Hay dos escenarios habituales:

* queremos escribir tipos de datos primitivos (float, double, boolean...),
* queremos escribir objetos Java completos.

En el primer caso, debemos emplear la clase DataOutputStream; al crear el objeto de esta clase le pasaremos a su constructor el FileOutputStream que representa el fichero al cual queremos escribir los tipos de datos primitivos:

DataOutputStream out = new DataOutputStream(fout);

DataOutputStream tiene un conjunto de métodos con nombre writeXXX donde XXX son los nombres de los distintos tipos de datos primitivos existentes en Java. Cada uno de esos métodos permite escribir el tipo de dato primitivo correspondiente al flujo de datos de salida:

out.writeBoolean(true);

out.writeInt(45);

out.writeDouble(4.8);

Si lo que queremos escribir son objetos Java, debemos emplear la clase ObjectOutputStream y, al crear el objeto de esta clase, le pasaremos a su constructor el FileOutputStream que representa el fichero al cual queremos escribir los objetos:

FileOutputStream fout2 = new FileOutputStream(f2);

ObjectOutputStream out2 = new ObjectOutputStream(fout2);

out2.writeObject(new Date());

La última sentencia está escribiendo un objeto de tipo java.util.Date al fichero representado por el File f2 . Finalmente, siempre que terminemos de trabajar con un flujo de datos debemos cerrarlo para liberar los recursos que dicho flujo de datos está consumiendo. Esto se hace invocando a su método close() . En el siguiente ejemplo podemos ver un código que hace uso tanto de un DataOutputStream como de un ObjectOutputStream. Ambos están asociados con un File que se creará en C:, el nombre del primer fichero será datos.dat y el del segundo objetos.dat. Observa como las operaciones de creación de los flujos de datos y las escrituras están dentro de un bloque try-catch. Si se produce una excepción, mostraremos un mensaje de error por la consola. En la cláusula finally cerramos los ficheros y los flujos de datos. Observa que estas sentencias también están dentro de un bloque try-catch, ya que también lanzan excepciones, que en el caso que nos atañe, son excepciones del tipo IOException (Input-Output Exception). Presta especial atención al modo de cerrar todos los recursos que, dependiendo de los distintos cauces de ejecución del programa (es decir, de si se ha lanzado alguna excepción o no), pueden haberse adquirido durante su ejecución.

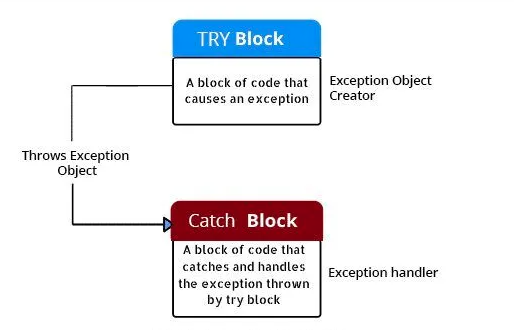
## Flujos de entrada de bytes

## Flujos de salida de texto

## Flujos de entrada de texto

# Control de excepciones

Java tiene un sistema para controlar los errores que se pueden producir durante la ejecución, llamado control de excepciones .



Se basa en que cuando se produce un error que se quiere controlar en ese punto, se lanza una excepción. Y en otro punto, se controla la excepción y se maneja el error en consecuencia.

Facilita el control de errores, ya que distintos tipos de excepciones se pueden controlar de la misma forma.

La obligación de control de excepciones garantiza un mejor control de los errores.

## Sentencia try-catch-finally

Para controlar las excepciones, se utiliza la sentencia try-catch.

Dentro del bloque try, se ejecutan las sentencias sobre las que se quiere controlar las excepciones.

Dentro del bloque catch, se muestran las sentencias que se deben ejecutar en caso de error.



Como parámetro, el catch pone el tipo de excepción a controlar. Si se quieren controlar todas, se usa Exception.

La sentencia finally es opcional, y contiene las sentencias que se van a ejecutar exista o no una excepción.

Ejemplo Escribir en Fichero con control de excepciones:

import java.io.\*;

public class EscribirFichero {

public static void main(String[] args) {

File fichero = null;

FileWriter writer = null;

PrintWriter pw = null;

try {

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

writer = new FileWriter(fichero);

pw = new PrintWriter(writer);

for (int i = 0; i < 10; i++) {

pw.println("Linea " + i);

}

} catch (Exception e) {

e.printStackTrace();

} finally {

try {

if (null != fichero) {

fichero.close();

}

}

catch (Exception e2) {

e2.printStackTrace();

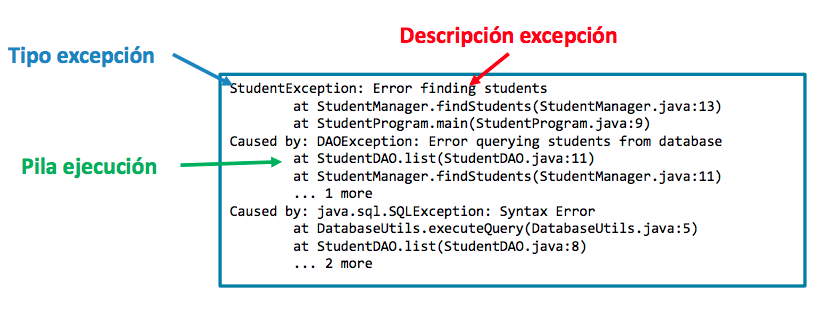
}

}

}

}

Si las excepciones no se controlan, se muestra el detalle por consola. Muestra la pila de ejecución.



**Práctica 1 – Crear un fichero**

Para crear un fichero en JAVA usamos la [clase File](https://docs.oracle.com/javase/7/docs/api/java/io/File.html) de Java, hay que tener en cuenta que podemos generar una [IOException](https://docs.oracle.com/javase/7/docs/api/java/io/IOException.html), por lo que habrá que gestionarla.

En el siguiente ejemplo vamos a crear un fichero llamado **practica1.java**:

import java.io.File;

import java.io.IOException;

public class fileCreate {

public static void main(String[] args) {

try {

File ficheroDatos = new File("prueba1.java"); if (ficheroDatos.createNewFile()) {

System.out.println("Fichero "+ ficheroDatos.getName() +" creado");

}

else {

System.out.println("No se ha podido crear el fichero. Probablemente ya exista.");

}

} catch(IOException error){

System.out.println("Error al crear el fichero.");

error.printStackTrace();

}

}

}

**Práctica 2 – Obtener información de un fichero**

Creamos un objeto fichero de la clase File, pasándole el nombre del archivo, en este caso, el nombre del archivo código fuente anterior practica1.java.

Si este archivo existe, es decir, si la función exists devuelve true, entonces se obtiene información acerca del archivo:

* getName devuelve el nombre del archivo
* getPath devuelve el camino relativo
* getAbsolutePath devuelve el camino absoluto.
* canRead nos indica si el archivo se puede leer.
* canWrite nos indica si el archivo se puede escribir
* length nos devuelve el tamaño del archivo, si dividimos la cantidad devuelta entre 1024 obtenemos el tamaño del archivo en KB.

File fichero=new File("practica1.java");

If ( fichero.exists() ) {

System.out.println("Nombre del archivo "+fichero.getName());

System.out.println("Camino "+fichero.getPath());

System.out.println("Camino absoluto "+fichero.getAbsolutePath());

System.out.println("Se puede escribir "+fichero.canRead());

System.out.println("Se puede leer "+fichero.canWrite());

System.out.println("Tamaño "+fichero.length());

}

**Práctica3** – Obtener lista de archivos del directorio actual

Para ello se crea un nuevo objeto de la clase *File.*

Para obtener la lista de los archivos que contiene este directorio se llama a la función miembro *list*, la cual nos devuelve un array de strings.

fichero=new File(".");

String[] listaArchivos=fichero.list();

For ( int i=0; i<listaArchivos.length; i++ )

{

System.out.println(listaArchivos[i]);

}

**Práctica 4 – Eliminar un archivo**

Para eliminar un fichero podemos usar la clase [clase File](https://docs.oracle.com/javase/7/docs/api/java/io/File.html) con su método **delete()**.

El método delete devolverá cierto si ha podido borrar el fichero y falso si no ha podido, probablemente en ese caso el fichero no existía.

En el ejemplo siguiente borramos el fichero practica1.java y avisamos al usuario si se ha podido borrar o no:

import java.io.File;

public class fileDelete {

public static void main(String[] args) {

File ficheroDatos = new File("datos.txt");

Boolean ficheroBorrado = ficheroDatos.delete();

if(ficheroBorrado){

System.out.println("Se ha borrado el fichero.");

}else{

System.out.println("No se ha borrado el fichero."); }

}

}

**Práctica 5 – Crear un directorio**

Para crear un directorio, la clase [clase File](https://docs.oracle.com/javase/7/docs/api/java/io/File.html) dispone del método **mkdir()**. En el siguiente ejemplo creamos el directorio **/home/sonia/midirectorio/**:

import java.io.File;

public class fileDirectory {

public static void main(String[] args) {

File ficheroDatos = new File("/home/profe/directorioTemporal/"); if (ficheroDatos.mkdir()) {

System.out.println("Directorio "+ ficheroDatos.getName() +" creado"); }

else {

System.out.println("No se ha podido crear el directorio. Probablemente ya exista.");

}

}

}

**Práctica 6 – Eliminar un directorio**

Es exactamente igual que eliminar un fichero pero poniendo la ruta del directorio en lugar de la de un fichero. Por ejemplo para eliminar el directorio /home/sonia/midirectorio/ escribiremos:

import java.io.File;

public class fileDelete {

public static void main(String[] args) {

File directorioDatos = new File("/home/usuario/directorioTemporal/"); Boolean directorioBorrado = directorioDatos.delete();

if(directorioBorrado){

System.out.println("Se ha borrado el directorio."); }else{

System.out.println("No se ha borrado el directorio."); }

}

}

**Práctica 7 – Escribir en un fichero**

Usaremos la clase FileWriter de Java para escribir en un fichero. En el siguiente ejemplo vamos a escribir información en el fichero **practica1.java**:

import java.io.FileWriter;

import java.io.IOException;

public class fileWrite {

public static void main(String[] args) {

try{

FileWriter escritorDeFichero = new FileWriter("practica1.java");

escritorDeFichero.write("Información.");

escritorDeFichero.close();

System.out.println("Fichero escrito con éxito.");

}catch (IOException error){

System.out.println("Error escribir en el fichero."); error.printStackTrace();

}

}

}

## Sobreescribir o añadir al final (overwrite or append):

Con FileWriter podemos escribir un fichero borrando todo lo que tenía en él utilizando el constructor FileWriter("namefile") o añadir información al final de un fichero que ya tiene algo escrito, con el método FileWriter("namefile, True).

En el ejemplo anterior si queremos escribir información a continuación de la que ya hay debemos substituir la línea:

FileWriter escritorDeFichero = new FileWriter("practica1.java");

por esta otra:

FileWriter escritorDeFichero = new FileWriter("practica1.java",true);

**Práctica 8 – Leer un fichero**

Para leer de fichero utilizaremos la clase [Scanner](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html) de Java. Cuando leemos, si no existe el fichero, podemos provocar una excepción FileNotFoundException que debemos controlar.

El siguiente código lee la información del fichero practica1.java y la muestra por consola:

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class fileRead {

public static void main(String[] args) {

try {

File ficheroDatos = new File("practica1.java");

Scanner lectorDatos = new Scanner(ficheroDatos);

while (lectorDatos.hasNextLine()){

String datos = lectorDatos.nextLine();

System.out.println(datos);

}

lectorDatos.close();

}catch(FileNotFoundException error){

System.out.println("Error al abrir el fichero para lectura."); error.printStackTrace();

}

}

}