



TEMA 11

**PROGRAMACIÓN
CFGS DAM**

2023/2024

Versión: 240326.2316

FICHEROS EN JAVA

ÍNDICE DE CONTENIDO

1. INTRODUCCIÓN.....	3
2. GESTIÓN DE FICHEROS.....	4
2.1 La clase File.....	4
2.2 Rutas absolutas y relativas.....	6
2.3 Métodos de la clase File.....	8
2.3.1 Obtención de la ruta.....	8
2.3.2 Comprobaciones de estado.....	10
2.3.3 Propiedades de ficheros.....	11
2.3.4 Gestión de ficheros.....	12
2.3.5 Listado de archivos.....	14
3. LECTURA Y ESCRITURA DE FICHEROS.....	15
3.1 Ficheros orientados a carácter.....	16
3.2 Lectura de fichero (clase Scanner).....	17
3.3 Escritura en fichero (clase FileWriter).....	20
4. BIBLIOGRAFÍA.....	24
5. Licencia.....	24
5.1 Agradecimientos.....	24

TEMA 11 – FICHEROS EN JAVA

1. INTRODUCCIÓN

La principal función de una aplicación informática es la manipulación y transformación de datos. Estos datos pueden representar cosas muy diferentes según el contexto del programa: notas de estudiantes, una recopilación de temperaturas, las fechas de un calendario, etc. Las posibilidades son ilimitadas. Todas estas tareas de manipulación y transformación se llevan a cabo normalmente mediante el almacenamiento de los datos en variables, dentro de la memoria del ordenador, por lo que se pueden aplicar operaciones, ya sea mediante operadores o la invocación de métodos.

Desgraciadamente, todas estas variables solo tienen vigencia mientras el programa se está ejecutando. Una vez el programa finaliza, los datos que contienen desaparecen. Esto no es problema para programas que siempre tratan los mismos datos, que pueden tomar la forma de literales dentro del programa. O cuando el número de datos a tratar es pequeño y se puede preguntar al usuario. Ahora bien, imagínense tener que introducir las notas de todos los estudiantes cada vez que se ejecuta el programa para gestionarlas. No tiene ningún sentido. Por tanto, en algunos casos, aparece la necesidad de poder registrar los datos en algún soporte de memoria externa, por lo que estas se mantengan de manera persistente entre diferentes ejecuciones del programa, o incluso si se apaga el ordenador.

La manera más sencilla de lograr este objetivo es almacenar la información aprovechando el sistema de archivos que ofrece el sistema operativo. Mediante este mecanismo, es posible tener los datos en un formato fácil de manejar e independiente del soporte real, ya sea un soporte magnético como un disco duro, una memoria de estado sólido, como un lápiz de memoria USB, un soporte óptico, cinta, etc.

En esta unidad didáctica se explican distintas clases de Java que nos permiten crear, leer, escribir y eliminar ficheros y directorios, entre otras operaciones. También se introduce la serialización de objetos como mecanismo de gran utilidad para almacenar objetos en ficheros para luego recuperarlos en tiempo de ejecución.

2. GESTIÓN DE FICHEROS

Entre las funciones de un sistema operativo está la de ofrecer mecanismos genéricos para gestionar sistemas de archivos. Normalmente, dentro de un sistema operativo moderno (o ya no tanto moderno), se espera disponer de algún tipo de interfaz o explorador para poder gestionar archivos, ya sea gráficamente o usando una línea de comandos de texto. Si bien la forma en que los datos se guardan realmente en los dispositivos físicos de almacenamiento de datos puede ser muy diferente según cada tipo (magnético, óptico, etc.), la manera de gestionar el sistema de archivos suele ser muy similar en la inmensa mayoría de los casos: una estructura jerárquica con carpetas y ficheros.

Ahora bien, en realidad, la capacidad de operar con el sistema de archivos no es exclusiva de la interfaz ofrecida por el sistema operativo. Muchos lenguajes de programación proporcionan bibliotecas que permiten acceder directamente a los mecanismos internos que ofrece el sistema, por lo que es posible crear código fuente desde el que, con las instrucciones adecuadas, se pueden realizar operaciones típicas de un explorador de archivos. De hecho, las interfaces como un explorador de archivos son un programa como cualquier otro, el cual, usando precisamente estas librerías, permite que el usuario gestione archivos fácilmente.

Pero es habitual encontrar otras aplicaciones con su propia interfaz para gestionar archivos, aunque solo sea para poder seleccionar qué hay que cargar o guardar en un momento dado: editores de texto, compresores, reproductores de música, etc.

Java no es ninguna excepción ofreciendo este tipo de biblioteca, en forma del conjunto de clases incluidas dentro del *package java.io*. Mediante la invocación de los métodos adecuados definidos de estas clases es posible llevar a cabo prácticamente cualquier tarea sobre el sistema de archivos.

2.1 La clase File

La pieza más básica para poder operar con archivos, independientemente de su tipo, en un programa Java es la **clase File**. Esta clase pertenece al *package java.io*. Por lo tanto será necesario importarla antes de poder usarla.

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

Esta clase permite manipular cualquier aspecto vinculado al sistema de ficheros. Su nombre ("archivo", en inglés) es un poco engañoso, ya que no se refiere exactamente a un archivo.

⚡ La clase File representa una ruta dentro del sistema de archivo

Sirve para realizar operaciones tanto sobre rutas al sistema de archivos que ya existan como no existentes. Además, se puede usar tanto para manipular archivos como directorios.

Como cualquier otra clase **hay que instanciarla para que sea posible invocar sus métodos**. El constructor de File recibe como argumento una cadena de texto correspondiente a la ruta sobre la que se quieren llevar a cabo las operaciones.

```
File f = new File (String ruta);
```

Una ruta es la forma general de un **nombre de archivo o carpeta**, por lo que identifica únicamente su localización en el sistema de archivos.

Cada uno de **los elementos de la ruta pueden existir realmente o no, pero esto no impide poder inicializar File**. En realidad, su comportamiento es como una declaración de intenciones sobre qué ruta del sistema de archivos se quiere interactuar. No es hasta que se llaman los diferentes métodos definidos en File, o hasta que se escriben o se leen datos, que realmente se accede al sistema de ficheros y se procesa la información.

Un aspecto importante a tener presente al inicializar File es tener siempre presente que el formato de la cadena de texto que conforma la ruta puede ser diferente según el sistema operativo sobre el que se ejecuta la aplicación. Por ejemplo, el sistema operativo Windows inicia las rutas por un nombre de unidad (C:, D:, etc.), mientras que los sistemas operativos basados en Unix comienzan directamente con una barra ("/"). Además, los diferentes sistemas operativos usan diferentes separadores dentro de las rutas. Por ejemplo, los sistemas Unix usan la barra ("/") mientras que Windows usa la inversa ("\\").

- Ejemplo de ruta Unix: /usr/bin
- Ejemplo de ruta Windows: C:\Windows\System32

De todos modos Java nos permite utilizar la barra de Unix ("/") para representar rutas en sistemas Windows. Por lo tanto, es posible utilizar siempre este tipo de barra independientemente del sistema, por simplicidad.

Es importante entender que **un objeto representa una única ruta** del sistema de ficheros. Para operar con diferentes rutas vez habrá que crear y manipular varios objetos. Por ejemplo, en el siguiente código se instancian tres objetos File diferentes.

```
File carpetaFotos = new File("C:/Fotos");  
File unaFoto = new File("C:/Fotos/Foto1.png");  
File otraFoto = new File("C:/Fotos/Foto2.png");
```

2.2 Rutas absolutas y relativas

En los ejemplos empleados hasta el momento para crear objetos del tipo File se han usado rutas absolutas, ya que es la manera de dejar más claro a qué elemento dentro del sistema de archivos, ya sea archivo o carpeta, se está haciendo referencia.



Una **ruta absoluta** es aquella que **se refiere a un elemento a partir del raíz** del sistema de ficheros. Por ejemplo "C:/Fotos/Foto1.png"

Las rutas absolutas se distinguen fácilmente, ya que el texto que las representa comienza de una manera muy característica dependiendo del sistema operativo del ordenador. En el caso de los sistemas operativos Windows a su inicio siempre se pone el nombre de la unidad ("C:", "D:", etc.), mientras que en los sistemas operativos Unix, estas comienzan siempre por una barra ("/").

Por ejemplo, las cadenas de texto siguientes representan rutas absolutas en un sistema de archivos de Windows:

- C:\Fotos\Viajes (ruta a una carpeta)
- M:\Documentos\Unidad11\apartado1 (ruta a una carpeta)
- N:\Documentos\Unidad11\apartado1\Actividades.txt (ruta a un archivo)

En cambio, en el caso de una jerarquía de ficheros bajo un sistema operativo Unix, un conjunto de rutas podrían estar representadas de la siguiente forma:

- /Fotos/Viajes (ruta a una carpeta)
- /Documentos/Unidad11/apartado1 (ruta a una carpeta)
- /Documentos/Unidad11/Apartado1/Actividades.txt (ruta a un archivo)

Al instanciar objetos de tipo File usando una ruta absoluta siempre hay que usar la representación correcta según el sistema en que se ejecuta el programa.

Si bien el **uso de rutas absolutas resulta útil para indicar con toda claridad qué elemento dentro del sistema de archivos se está manipulando, hay casos que su uso conlleva ciertas complicaciones**. Suponga que ha hecho un programa en el que se llevan a cabo operaciones sobre el sistema de archivos. Una vez funciona, le deja el proyecto Java a un amigo que lo copia en su ordenador dentro de una carpeta cualquiera y la abre con su entorno de trabajo. Para que el programa funcione perfectamente será necesario que en su ordenador haya exactamente las mismas carpetas que usa en su máquina, tal como están escritas en el código fuente de su programa. De lo contrario, no funcionará, ya que las carpetas y ficheros esperados no existirán, y por tanto, no se encontrarán. Usar rutas absolutas hace que un programa siempre tenga que trabajar con una estructura del sistema de archivos exactamente igual donde quiera que se ejecute, lo cual no es muy cómodo.

Para resolver este problema, a la hora de inicializar una variable de tipo File, también se puede hacer referencia a una ruta relativa.



Una **ruta relativa** es aquella que **no incluye el raíz** y por ello se considera que **parte desde el directorio de trabajo** de la aplicación. Esta carpeta puede ser diferente cada vez que se ejecuta el programa.

Cuando un programa se ejecuta por defecto se le asigna una carpeta de trabajo. Esta carpeta **suele ser la carpeta desde donde se lanza el programa.** En el caso de un programa en Java ejecutado a través de un IDE (como Netbeans), la carpeta de trabajo suele ser la misma carpeta donde se ha elegido guardar los archivos del proyecto.

El formato de una ruta relativa es similar a una ruta absoluta, pero nunca se indica la raíz del sistema de ficheros. Directamente se empieza por el primer elemento escogido dentro de la ruta. Por ejemplo:

- Viajes
- Unidad11\apartado1
- Unidad11\apartado1\Actividades.txt

Una ruta relativa siempre incluye el directorio de trabajo de la aplicación como parte inicial a pesar de no haberse escrito. El rasgo distintivo es que el directorio de trabajo puede variar. Por ejemplo, el elemento al que se refiere el siguiente objeto File varía según el directorio de trabajo.

```
File f = new File ("Unidad11/apartado1/Actividades.txt");
```

Directorio de trabajo	Ruta real
C:/Proyectos/Java	C:/Proyectos/Java/Unidad11/apartado1/Actividades.txt
X:/Unidades	X:/Unidades/Unidad11/apartado1/Actividades.txt
/Programas	/Programas/Unidad11/apartado1/Actividades.txt

Este mecanismo permite facilitar la portabilidad del software entre distintos ordenadores y sistemas operativos, ya que solo es necesario que los archivos y carpetas permanezcan en la misma ruta relativa al directorio de trabajo. Veamos un ejemplo:

```
File f = new File ("Actividades.txt");
```

Dada esta ruta relativa, basta garantizar que el fichero "Actividades.txt" esté siempre en el mismo directorio de trabajo de la aplicación, cualquiera que sea éste e independientemente del sistema operativo utilizado (en un ordenador puede ser "C:\Programas" y en otro "/Java"). En cualquiera de todos estos casos, la ruta siempre será correcta. De hecho, aún más, nótese como **las rutas relativas Java permiten crear código independiente del sistema operativo**, ya que no es necesario especificar un formato de raíz ligada a un sistema de archivos concreto ("C:", "D:", "/", etc.) .

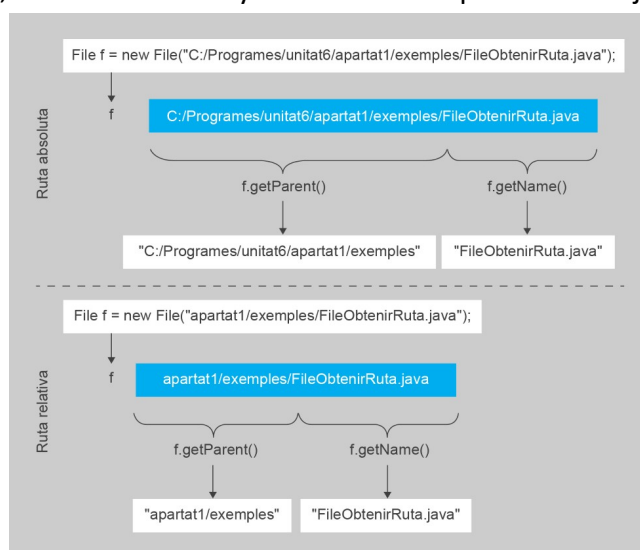
2.3 Métodos de la clase File

File ofrece varios métodos para poder manipular el sistema de archivos u obtener información a partir de su ruta. Algunos de los más significativos para entender las funcionalidades se muestran a continuación, ordenados por tipo de operación.

2.3.1 Obtención de la ruta

Una vez se ha instanciado un objeto de tipo File, puede ser necesario recuperar la información empleada durante su inicialización y conocer en formato texto a qué ruta se está refiriendo, o al menos parte de ella.

- **String getParent()** devuelve la ruta de la carpeta del elemento referido por esta ruta. Básicamente la cadena de texto resultante es idéntica a la ruta original, eliminando el último elemento. Si la ruta tratada se refiere a la carpeta raíz de un sistema de archivos ("C:\", "/", etc.), este método devuelve null. En el caso de tratarse de una ruta relativa, este método no incluye la parte de la carpeta de trabajo.
- **String getName()** devuelve el nombre del elemento que representa la ruta, ya sea una carpeta o un archivo. Es el caso inverso del método getParent(), ya que el texto resultante es solo el último elemento.
- **String getAbsolutePath()** devuelve la ruta absoluta. Si el objeto File se inicializó usando una ruta relativa, el resultado incluye también la carpeta de trabajo.



Veamos un ejemplo de cómo funcionan estos tres métodos. Observe que las rutas relativas se añaden a la ruta de la carpeta de trabajo (donde se encuentra el proyecto):

```
import java.io.File;

public class PruebasFicheros {

    public static void main(String[] args) {
        // Dos rutas absolutas
        File carpetaAbs = new File("/home/lionel/fotos");
        File archivoAbs = new File("/home/lionel/fotos/albanial.jpg");

        // Dos rutas relativas
        File carpetaRel = new File("trabajos");
        File fitxerRel = new File("trabajos/documento.txt");

        // Mostrem sus rutas
        mostrarRutas(carpetaAbs);
        mostrarRutas(archivoAbs);
        mostrarRutas(carpetaRel);
        mostrarRutas(fitxerRel);
    }

    public static void mostrarRutas(File f) {
        System.out.println("getParent()      : " + f.getParent());
        System.out.println("getName()       : " + f.getName() + "\n");
        System.out.println("getAbsolutePath(): " + f.getAbsolutePath());
    }
}
```

Este programa produce la salida:

```
getParent()      : /home/lionel
getName()       : fotos
getAbsolutePath(): /home/lionel/fotos

getParent()      : /home/lionel/fotos
getName()       : albanial.jpg
getAbsolutePath(): /home/lionel/fotos/albanial.jpg

getParent()      : null
getName()       : trabajos
getAbsolutePath(): /home/lionel/NetBeans/Ficheros/trabajos

getParent()      : trabajos
getName()       : documento.txt
getAbsolutePath(): /home/lionel/NetBeans/Ficheros/trabajos/documento.txt
```

2.3.2 Comprobaciones de estado

Dada la ruta empleada para inicializar una variable de tipo `File`, esta puede que realmente exista dentro del sistema de ficheros o no, ya sea en forma de archivo o carpeta. La clase `File` ofrece un conjunto de métodos que permiten hacer comprobaciones sobre su estado y saber si es así.

- **`boolean exists()`** comprueba si la ruta existe dentro del sistema de ficheros. Devolverá *true* si existe y *false* en caso contrario.

Normalmente los archivos incorporan en su nombre una extensión (.txt, .jpg, .mp4, etc.). Aún así, hay que tener en cuenta que la extensión no es un elemento obligatorio en el nombre de un archivo, sólo se usa como mecanismo para que tanto el usuario como algunos programas puedan discriminar más fácilmente el tipo de archivos. Por lo tanto, solo con el texto de una ruta no se puede estar 100% seguro de si esta se refiere a un archivo o una carpeta. Para poder estar realmente seguros se pueden usar los métodos siguientes:

- **`boolean isFile()`** comprueba el sistema de ficheros en busca de la ruta y devuelve *true* si existe y es un fichero. Devolverá *false* si no existe, o si existe pero no es un fichero.
- **`boolean isDirectory()`** funciona como el anterior pero comprueba si es una carpeta.

Por ejemplo, el siguiente código hace una serie de comprobaciones sobre un conjunto de rutas. Para poder probarlo puedes crear la carpeta "Temp" en la raíz "C:". Dentro, un archivo llamado "Documento.txt" (puede estar vacío) y una carpeta llamada "Fotos". Después de probar el programa puedes eliminar algún elemento y volver a probar para ver la diferencia.

```
public static void main(String[] args) {
    File temp = new File("C:/Temp");
    File fotos = new File("C:/Temp/Fotos");
    File documento = new File("C:/Temp/Documento.txt");
    System.out.println(temp.getAbsolutePath()+" ¿existe? "+temp.exists());
    mostrarEstado(fotos);
    mostrarEstado(documento);
}

public static void mostrarEstado(File f) {
    System.out.println(f.getAbsolutePath()+" ¿archivo? "+f.isFile());
    System.out.println(f.getAbsolutePath()+" ¿carpeta? "+f.isDirectory());
}
```

2.3.3 Propiedades de ficheros

El sistema de ficheros de un sistema operativo almacena diversidad de información sobre los archivos y carpetas que puede resultar útil conocer: sus atributos de acceso, su tamaño, la fecha de modificación, etc. En general, todos los datos mostrados en acceder a las propiedades del archivo. Esta información también puede ser consultada usando los métodos adecuados. Entre los más populares hay los siguientes:

- **long length()** devuelve el tamaño de un archivo en bytes. Este método solo puede ser llamado sobre una ruta que represente un archivo, de lo contrario no se puede garantizar que el resultado sea válido.
- **long lastModified()** devuelve la última fecha de edición del elemento representado por esta ruta. El resultado se codifica en un único número entero cuyo valor es el número de milisegundos que han pasado desde el 1 de junio de 1970.

El ejemplo siguiente muestra cómo funcionan estos métodos. Para probarlos crea el archivo "Documento.txt" en la carpeta "C:\Temp". Primero deja el archivo vacío y ejecuta el programa. Luego, con un editor de texto, escribe cualquier cosa, guarda los cambios y vuelve a ejecutar el programa. Observa cómo el resultado es diferente. Como curiosidad, fíjate en el uso de la clase Date para poder mostrar la fecha en un formato legible.

```
public static void main(String[] args) {  
    File documento = new File("C:/Temp/Documento.txt");  
    System.out.println(documento.getAbsolutePath());  
  
    long milisegundos = documento.lastModified();  
    Date fecha = new Date(milisegundos);  
  
    System.out.println("Última modificación (ms)    : " + milisegundos);  
    System.out.println("Última modificación (fecha): " + fecha);  
    System.out.println("Tamaño del archivo: " + documento.length());  
}
```

Primera salida:

```
C:/Temp/Documento.txt  
Última modificación (ms)    : 1583025735411  
Última modificación (fecha): Sun Mar 01 02:22:15 CET 2020  
Tamaño del archivo: 0
```

Segunda salida:

```
C:/Temp/Documento.txt  
Última modificación (ms)    : 1583025944088  
Última modificación (fecha): Sun Mar 01 02:25:44 CET 2020  
Tamaño del archivo: 7
```

2.3.4 Gestión de ficheros

El conjunto de operaciones más habituales al acceder a un sistema de ficheros de un ordenador son las vinculadas a su gestión directa: renombrar archivos, borrarlos, copiarlos o moverlos. Dado el nombre de una ruta, Java también permite realizar estas acciones.

- **boolean mkdir()** permite crear la carpeta indicada en la ruta. La ruta debe indicar el nombre de una carpeta que no existe en el momento de invocar el método. Por ejemplo, dado un objeto `File` instanciado con la ruta "C:/Fotos/Albania" que no existe, el método `mkdir()` creará la carpeta "Albania" dentro de "C:/Fotos". Devuelve `true` si se ha creado correctamente, en caso contrario devuelve `false` (por ejemplo si la ruta es incorrecta, la carpeta ya existe o el usuario no tiene permisos de escritura).
- **boolean delete()** borra el archivo o carpeta indicada en la ruta. La ruta debe indicar el nombre de un archivo o carpeta que sí existe en el momento de invocar el método. Se podrá borrar una carpeta solo si está vacía (no contiene ni carpetas ni archivos). Devuelve `true` o `false` según si la operación se ha podido llevar a cabo.

Para probar el ejemplo que se muestra a continuación de manera que se pueda ver cómo funcionan estos métodos, primero asegúrate de que en la raíz de la unidad "C:" no hay ninguna carpeta llamada "Temp" y ejecuta el programa. Todo fallará, ya que las rutas son incorrectas (no existe "Temp"). Luego, crea la carpeta "Temp" y en su interior crea un nuevo documento llamado "Documento.txt" (puede estar vacío). Ejecuta el programa y verás que se habrá creado una nueva carpeta llamada "Fotos". Si lo vuelves a ejecutar por tercera vez podrás comprobar que se habrá borrado.

```
public static void main(String[] args) {  
    File fotos = new File("C:/Temp/Fotos");  
    File doc = new File("C:/Temp/Documento.txt");  
  
    boolean mkdirFot = fotos.mkdir();  
  
    if (mkdirFot) {  
        System.out.println("Creada carpeta " + fotos.getName() + "? " +  
mkdirFot);  
    }  
    else {  
        boolean delCa = fotos.delete();  
        System.out.println("Borrada carpeta " + fotos.getName() + "? " +  
delCa);  
        boolean delAr = doc.delete();  
        System.out.println("Borrado archivo " + doc.getName() + "? " + delAr);  
    }  
}
```

Desde el punto de vista de un sistema operativo la operación de "mover" un archivo o carpeta no es más que cambiar su nombre desde su ruta original hasta una nueva ruta destino. Para hacer esto también hay un método.

- **boolean renameTo(File destino)** el nombre de este método es algo engañoso ("renombrar", en inglés), ya que su función real no es simplemente cambiar el nombre de un archivo o carpeta, sino cambiar la ubicación completa. El método se invoca el objeto File con la ruta origen (donde se encuentra el archivo o carpeta), y se le da como argumento otro objeto File con la ruta destino. Devuelve true o false según si la operación se ha podido llevar a cabo correctamente o no (la ruta origen y destino son correctos, no existe ya un archivo con este nombre en el destino, etc.). Nótese que, en el caso de carpetas, es posible moverlas aunque contengan archivos.

Una vez más, veamos un ejemplo. Dentro de la carpeta "C:/Temp" crea una carpeta llamada "Media" y otra llamada "Fotos". Dentro de la carpeta "Fotos" crea dos documentos llamados "Documento.txt" y "Fotos.txt". Después de ejecutar el programa, observa como la carpeta "Fotos" se ha movido y ha cambiado de nombre, pero mantiene en su interior el archivo "Fotos.txt". El archivo "Documento.txt" se ha movido hasta la carpeta "Temp".

```
public static void main(String[] args) {  
  
    File origenDir = new File("C:/Temp/Fotos");  
    File destinoDir = new File("C:/Temp/Media/Fotografias");  
    File origenDoc = new File("C:/Temp/Media/Fotografias/Documento.txt");  
    File destinoDoc = new File("C:/Temp/Documento.txt");  
  
    boolean res = origenDir.renameTo(destinoDir);  
    System.out.println("Se ha movido y renombrado la carpeta? " + res);  
    res = origenDoc.renameTo(destinoDoc);  
    System.out.println("Se ha movido el documento? " + res);  
}
```

Como ya se ha comentado este método también sirve, implícitamente, para renombrar archivos o carpetas. Si el elemento final de las rutas origen y destino son diferentes, el nombre del elemento, sea archivo o carpeta, cambiará. Para simplemente renombrar un elemento sin moverlo de lugar, simplemente su ruta padre sea exactamente la misma. El resultado es que el elemento de la ruta origen "se mueve" en la misma carpeta donde está ahora, pero con un nombre diferente.

Por ejemplo, si utilizamos "C:/Trabajos/Doc.txt" como ruta origen y "C:/Trabajos/File.txt" como ruta destino, el archivo "Doc.txt" cambiará de nombre a "File.txt" pero permanecerá en la misma carpeta "C:/Trabajos".

2.3.5 Listado de archivos

Finalmente, sólo en el caso de las carpetas, es posible consultar cuál es el listado de archivos y carpetas que contiene.

- **File[] listfiles()** devuelve un vector de tipo File (File[]) con todos los elementos contenidos en la carpeta (representados por objetos File, uno por elemento). Para que se ejecute correctamente la ruta debe indicar una carpeta. El tamaño del vector será igual al número de elementos que contiene la carpeta. Si el tamaño es 0, el valor devuelto será null y toda operación posterior sobre el vector será errónea. El orden de los elementos es aleatorio (al contrario que en el explorador de archivos del sistema operativo, no se ordena automáticamente por tipo ni alfabéticamente).

Veamos un ejemplo. Antes de ejecutarlo, crea una carpeta "Temp" en la raíz de la unidad "C:". Dentro crea o copia cualquier cantidad de carpetas o archivos.

```
public static void main(String[] args) {  
  
    File dir = new File("C:/Temp");  
    File[] lista = dir.listFiles();  
    System.out.println("Contenido de " + dir.getAbsolutePath() + " :");  
  
    // Recorremos el array y mostramos el nombre de cada elemento  
    for (int i = 0; i < lista.length; i++) {  
        File f = lista[i];  
        if (f.isDirectory()) {  
            System.out.println("[DIR] " + f.getName());  
        } else {  
            System.out.println("[ARX] " + f.getName());  
        }  
    }  
}
```

3. LECTURA Y ESCRITURA DE FICHEROS

Normalmente las aplicaciones que utilizan archivos no están centradas en la gestión del sistema de archivos de su ordenador. El objetivo principal de usar ficheros es poder almacenar datos de modo que entre diferentes ejecuciones del programa, incluso en diferentes equipos, sea posible recuperarlos. El caso más típico es un editor de documentos, que mientras se ejecuta se encarga de gestionar los datos relativos al texto que está escribiendo, pero en cualquier momento puede guardarlo en un archivo para poder recuperar este texto en cualquier momento posterior, y añadir otros nuevos si fuera necesario. El fichero con los datos del documento lo puede abrir tanto en el editor de su ordenador como en el de otro compañero.

Para saber cómo tratar los datos de un fichero en un programa, hay que tener muy claro cómo se estructuran. Dentro de un archivo se pueden almacenar todo tipo de valores de cualquier tipo de datos. La parte más importante es que estos valores se almacenan en forma de secuencia, uno tras otro. Por lo tanto, como pronto veréis, la forma más habitual de tratar ficheros es secuencialmente, de forma parecida a como se hace para leerlas del teclado, mostrarlas por pantalla o recorrer las posiciones de un array.



Se denomina **acceso secuencial** al procesamiento de un conjunto de elementos de manera que sólo es posible acceder a ella de acuerdo a su orden de aparición. Para procesar un elemento es necesario procesar primero todos los elementos anteriores.

Java, junto con otros lenguajes de programación, diferencia entre dos tipos de archivos según cómo se representan los valores almacenados en un archivo.



En los **ficheros orientados a carácter**, los datos se representan como una secuencia de cadenas de texto, donde cada valor se diferencia del otro usando un delimitador. En cambio, en los **ficheros orientados a byte**, los datos se representan directamente de acuerdo a su formato en binario, sin ninguna separación.

En esta unidad didáctica solo veremos el procesamiento de ficheros orientados a carácter.

3.1 Ficheros orientados a carácter

Un fichero orientado a carácter no es más que un documento de texto, como el que podría generar con cualquier editor de texto simple. Los valores están almacenados según su representación en cadena de texto, exactamente en el mismo formato que ha usado hasta ahora para entrar datos desde el teclado. Del mismo modo, los diferentes valores se distinguen al estar separados entre ellos con un delimitador, que por defecto es cualquier conjunto de espacios en blanco o salto de línea. Aunque estos valores se puedan distribuir en líneas de texto diferentes, conceptualmente, se puede considerar que están organizados uno tras otro, secuencialmente, como las palabras en la página de un libro.

El siguiente podría ser el contenido de un fichero orientado a carácter donde hay diez valores de tipo real, 7 en la primera línea y 3 en la segunda:

```
1,5 0,75 -2,35 18 9,4 3,1416 -15,785
-200,4 2,56 9,3785
```

Y este el de un fichero con 3 valores de tipo String ("Había", "una" y "vez..."):

```
Había una vez...
```

En un fichero orientado a carácter **es posible almacenar cualquier combinación de datos de cualquier tipo** (int, double, boolean, String, etc.).

```
7 10 20,5 16,99
Había una vez...
true false 2020 0,1234
```

La principal ventaja de un fichero de este tipo es que resulta muy sencillo inspeccionar su contenido y generarlos de acuerdo a nuestras necesidades.

Para el caso de los ficheros orientados a carácter, hay que usar dos clases diferentes según si lo que se quiere es leer o escribir datos en un archivo. Normalmente esto no es muy problemático, ya que **en un bloque de código dado solo se llevarán a cabo operaciones de lectura o de escritura sobre un mismo archivo, pero no los dos tipos de operación a la vez.**



Para tratar de manera sencilla ficheros orientados a carácter, Java proporciona las clases **Scanner** (para lectura) del **package java.util**, y **FileWriter** (para escritura) del **package java.io**.

3.2 Lectura de fichero (clase Scanner)

La clase que permite llevar a cabo la lectura de datos desde un fichero orientado a carácter es exactamente la misma que permite leer datos desde el teclado: **Scanner**. Al fin y al cabo, los valores almacenados en los archivos de este tipo se encuentran exactamente en el mismo formato que ha usado hasta ahora para entrar información en sus programas: una secuencia de cadenas de texto. La única diferencia es que estos valores no se piden al usuario durante la ejecución, sino que se encuentran almacenados en un fichero.

Para procesar datos desde un archivo, **el constructor de la clase Scanner permite como argumento un objeto de tipo File** que contenga la ruta a un archivo.

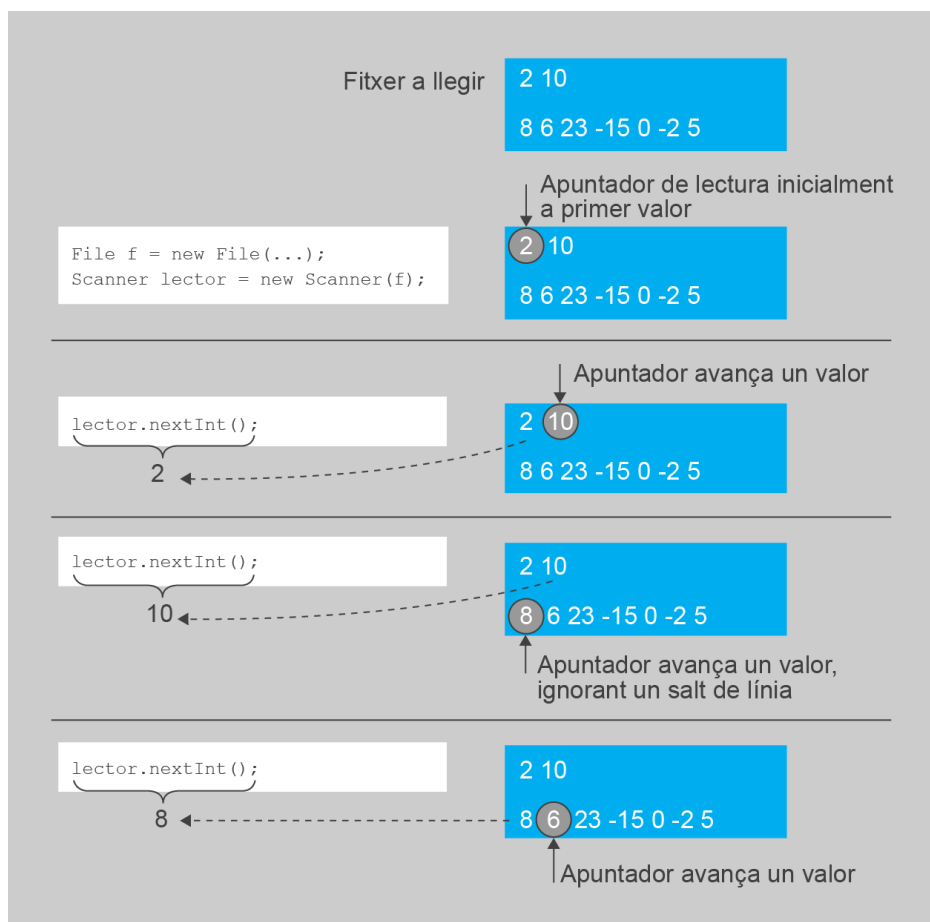
Por ejemplo, para crear un objeto de tipo Scanner de modo que permita leer datos desde el archivo ubicado en la ruta "C:\Programas\Unidad11\Documento.txt", habría que hacer:

```
import java.io.File;
import java.util.Scanner;
...
File f = new File("C:\Programas\Unidad11\Documento.txt");
Scanner lectorArchivo = new Scanner(f);
...
```

Una vez instanciado el objeto Scanner **podemos utilizar sus métodos exactamente igual que si leyéramos de teclado**: `hasNext()`, `next()`, `nextLine()`, `nextInt()`, `nextDouble()`, `nextBoolean()`, etc. La única diferencia es que el objeto Scanner leerá secuencialmente el contenido del archivo.

Es importante entender que en el caso de un archivo, **el objeto Scanner gestiona internamente un apuntador que indica sobre qué valor actuarán las operaciones de lectura**. Inicialmente el apuntador se encuentra en el primer valor dentro del archivo. **Cada vez que se hace una lectura el apuntador avanza automáticamente hasta el siguiente valor dentro del archivo y no hay ninguna manera de hacerlo retroceder**. A medida que invocamos métodos de lectura el apuntador sigue avanzando hasta que hayamos leído tantos datos como queramos, o hasta que no podamos seguir leyendo porque hemos llegado al final del fichero.

A continuación se muestra un pequeño esquema de este proceso, recalcando cómo avanza el apuntador a la hora de realizar operaciones de lectura sobre un archivo que contiene valores de tipo entero.



Es importante recordar la diferencia entre el método `next()` y `nextLine()`, ya que ambos evalúan una cadena de texto. El método `next()` sólo lee una palabra individual (conjuntos de caracteres, incluidos dígitos, que no están separados por espacios o saltos de línea, como por ejemplo "casa", "hola", "2", "3,14", "1024", etc.). En cambio, `nextLine()` lee todo el texto que encuentre (espacios incluidos) hasta el siguiente salto de línea. En tal caso el apuntador se posiciona al inicio de la siguiente línea.

Una vez se ha finalizado la lectura del archivo, ya sean todas o solo una parte, es imprescindible ejecutar un método especial llamado `close()`. Este método indica al sistema operativo que el archivo ya no está siendo utilizado por el programa. Esto es muy importante ya que mientras un archivo se considera en uso, su acceso puede verse limitado. Si no se utiliza `close()` el sistema operativo puede tardar un tiempo en darse cuenta de que el archivo ya no está en uso.

⚡ Siempre hay que cerrar los archivos con `close()` cuando se ha terminado de leer o escribir en ellos.

Es importante saber que al instanciar el objeto `Scanner` se lanzará una excepción de tipo `java.io.FileNotFoundException` si el fichero no existe. Siempre habrá que manejar dicha excepción

mediante un try-catch. Scanner también **puede lanzar otras excepciones**, por ejemplo si se intenta leer el tipo de dato incorrecto (llamamos a `nextInt()` cuando no hay un entero, como sucede en la entrada por teclado,) o si hemos llegado al final del fichero e intentamos seguir leyendo (podemos comprobarlo mediante el método `hasNext()` de Scanner, que devuelve `true` si aún hay algún elemento que leer).

El programa siguiente muestra un ejemplo de cómo leer diez valores enteros de un archivo llamado "Enteros.txt" ubicado en la carpeta de trabajo (debería ser la carpeta del proyecto Netbeans). Para probarlo, crea el archivo e introduce exactamente 10 valores enteros separados por espacios en blanco o saltos de línea.

```
import java.io.File;
import java.util.Scanner;

public class PruebasFicheros {

    public static final int NUM_VALORES = 10;

    public static void main(String[] args) {

        try {
            // Intentamos abrir el fichero
            File f = new File("Enteros.txt");
            Scanner lector = new Scanner(f);

            // Si llega aquí es que ha abierto el fichero :)
            for (int i = 0; i < NUM_VALORES; i++) {
                int valor = lector.nextInt();
                System.out.println("El valor leído es: " + valor);
            }

            // ¡Hay que cerrar el fichero!
            lector.close();

        } catch (Exception e) {
            // En caso de excepción mostramos el error
            System.out.println("Error: " + e);
            e.printStackTrace();
        }

    }

}
```

Una diferencia importante a la hora de tratar con archivos respecto a leer datos del teclado es que las operaciones de lectura no son producto de una interacción directa con el usuario, que es quien escribe los datos. Solo se puede trabajar con los datos que hay en el archivo y nada más. Esto tiene dos efectos sobre el proceso de lectura:

- Por un lado, recuerda que **cuando se lleva a cabo el proceso de lectura de una secuencia**

de valores, siempre hay que tener cuidado de usar el método adecuado al tipo de valor que se espera que venga a continuación. Qué tipo de valor se espera es algo que habréis decidido vosotros a la hora de hacer el programa que escribió ese archivo, por lo que es vuestra responsabilidad saber qué hay que leer en cada momento. De todos modos nada garantiza que no se haya cometido algún error o que el archivo haya sido manipulado por otro programa o usuario. Como operamos con ficheros y no por el teclado, no existe la opción de pedir al usuario que vuelva a escribir el dato. Por lo tanto, el programa debería decir que se ha producido un error ya que el archivo no tiene el formato correcto y finalizar el proceso de lectura.

- Por otra parte, **también es necesario controlar que nunca se lean más valores de los que hay disponibles para leer.** En el caso de la entrada de datos por el teclado el programa simplemente se bloqueaba y espera a que el usuario escribiera nuevos valores. Pero con ficheros esto no sucede. Intentar leer un nuevo valor cuando el apuntador ya ha superado el último disponible se considera erróneo y lanzará una excepción. Para evitarlo, **será necesario utilizar el método hasNext() antes de leer, que nos devolverá true si existe un elemento a continuación.** Una vez se llega al final del archivo ya no queda más remedio que invocar close () y finalizar la lectura.

Para verlo, intenta ejecutar el ejemplo anterior modificando el archivo de forma que alguno de los valores no sean de tipo entero, o haya menos de 10 valores.

3.3 Escritura en fichero (clase FileWriter)

Para escribir datos a un archivo la clase más sencilla de utilizar es FileWriter. Esta clase tiene dos constructores que merece la pena conocer:

- public FileWriter(File file)
- public FileWriter(File file, boolean append)

El primer constructor es muy parecido al del Scanner. Solo hay que pasarle un objeto File con la ruta al archivo. Al tratarse de escritura la ruta puede indicar un fichero que puede existir o no dentro del sistema. **Si el fichero no existe, se creará uno nuevo.** Pero **si el fichero ya existe, su contenido se borra por completo, con tamaño igual a 0.** Esto puede ser peligroso ya que si no se maneja correctamente puede producir la pérdida de datos valiosos. Hay que estar completamente seguro de que se quiere sobrescribir el fichero.

```
import java.io.File;
import java.io.FileWriter;
...
File f = new File("C:\\Programas\\Unidad11\\Documento.txt");
FileWriter writer = new FileWriter(f);
```


El segundo constructor tiene otro **parámetro de tipo booleano llamado “append” (añadir)**

que nos permite indicar si queremos escribir al final del fichero o no. Es decir, si le pasamos "false" hará lo mismo que el constructor anterior (si el archivo ya existe, lo sobrescribirá), pero si le pasamos "true" abrirá el archivo para escritura en modo "append", es decir, **escribiremos al final del fichero sin borrar los datos ya existentes**.

```
import java.io.File;
import java.io.FileWriter;

File f = new File("C:\\Programas\\Unidad11\\Documento.txt");
FileWriter writer = new FileWriter(f, true);
```

La escritura secuencial de datos en un fichero orientado a carácter es muy sencilla. Solo es necesario utilizar el siguiente método **void write(String str)** que escribirá la cadena str en el fichero. Si se desea agregar un **final de línea** se puede agregar "\n".

 Tanto el constructor de **FileWriter** como el método **write()** pueden lanzar una excepción **IOException** si se produce algún error inesperado.

Es importante tener en cuenta que **para que el método write() escriba texto correctamente es imprescindible pasarle como argumento un String**. Está permitido utilizar datos o variables distintas a String, pero se escribirá directamente su valor en bytes, no como texto. Veamos dos ejemplos ilustrativos.

```
writer.write("8"); // Escribe el carácter 8
writer.write(8);   // Escribe 8 como byte, es un carácter no imprimible

writer.write("65"); // Escribe dos caracteres, el 6 y el 5
writer.write(65);   // Escribe 65 como byte, es el carácter A
```

Por lo tanto cuando queremos escribir el valor de variables que no sean String será necesario pasárselas a write() como String. Esto es muy sencillo, solo hay que concatenar un String vacío con la variable (Java siempre convierte a String la concatenación de cadenas de texto con cualquier otro elemento):

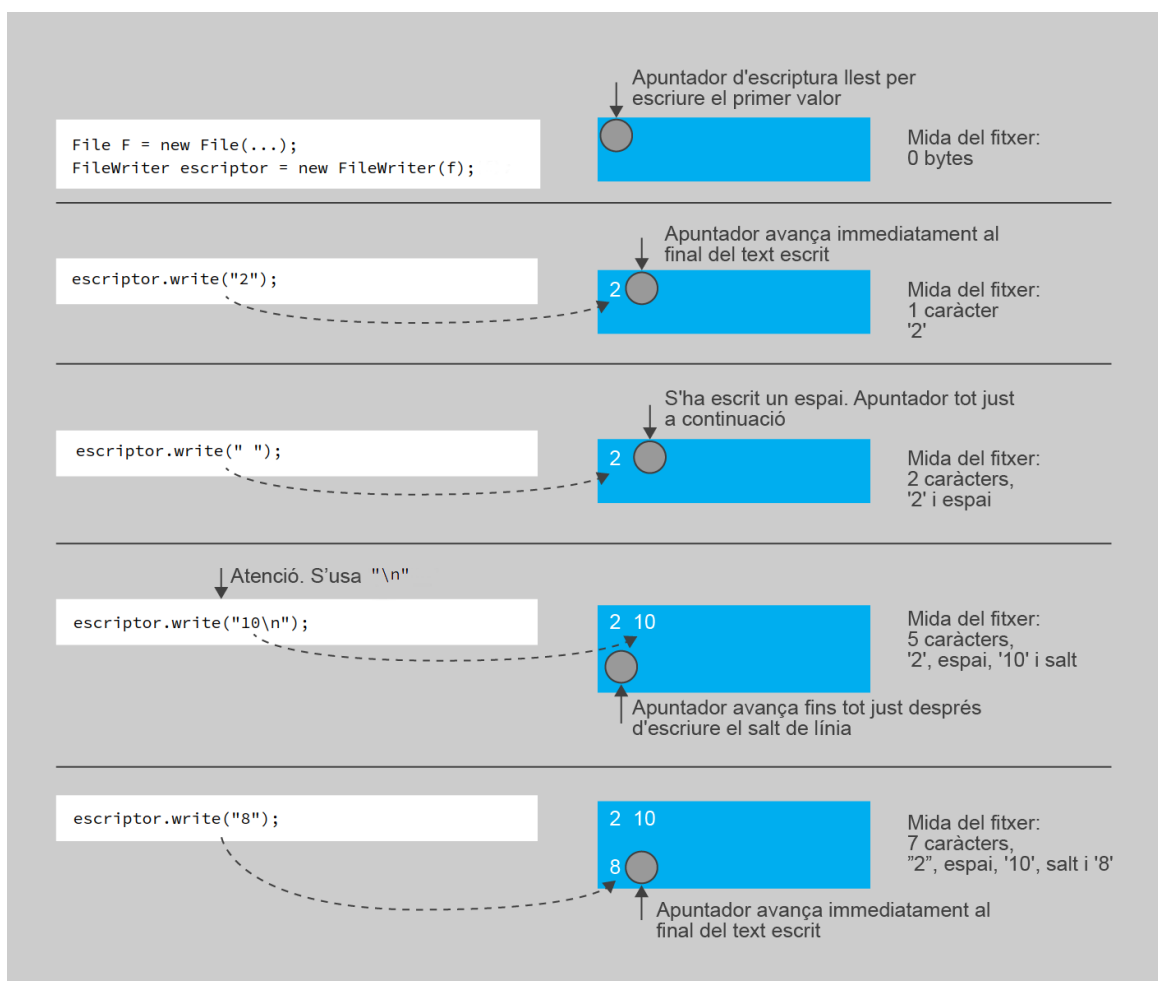
"" + variable

```
int edad = 35;
writer.write("" + edad); // escribe el texto "35"
```

La escritura de datos en fichero tiene la particularidad de que una vez se ha escrito un dato ya no hay marcha atrás. No es posible escribir información antes o en medio de valores que ya están escritos.

Como en el caso de la lectura, la clase `FileWriter` también gestiona un apuntador que le permite saber a partir de qué posición del texto debe ir escribiendo. Cada vez que se invoca uno de sus métodos de escritura, el apuntador avanza automáticamente y no es posible hacerlo retroceder. A efectos prácticos este apuntador siempre está al final del archivo, de modo que a medida que se van escribiendo datos el archivo va incrementando su tamaño.


A continuación se muestra un esquema del funcionamiento de la escritura en fichero.



La escritura no genera automáticamente un delimitador entre valores. Los espacios en blanco o saltos de línea que se deseen incorporar deben escribirse explícitamente. De lo contrario los valores quedarán pegados y en una posterior lectura se interpretarán como un único valor. Por ejemplo, si se escribe el valor 2 y luego el 4, sin espacio, en el fichero se habrá escrito el valor 24. Si se leyera mediante un `nextInt()` nos devolvería un único valor, no dos.

Al escribir en ficheros el cierre con `close()` es todavía más importante que en la lectura. Esto se debe a que los sistemas operativos a menudo actualizan los datos de forma diferida. Es decir, el hecho de ejecutar una instrucción de escritura no significa que inmediatamente se escriba en el archivo. Puede pasar un intervalo de tiempo variable. Solo al ejecutar el método `close()` se fuerza al

sistema operativo a escribir los datos pendientes (si los hubiera).

 Al terminar la escritura también **es imprescindible invocar el método `close()`** para cerrarlo y asegurar la correcta escritura de datos.

El código siguiente sirve como ejemplo de un programa que escribe un archivo llamado "Enteros.txt" dentro de la carpeta de trabajo. Se escriben 20 valores enteros, empezando por el 1 y cada vez el doble del anterior. Pruébalo para ver su funcionamiento. Ten en cuenta que si ya existía un archivo con ese nombre, quedará totalmente sobrescrito. Después, puedes intentar leerlo con el programa del ejemplo anterior para leer 10 valores enteros y mostrarlos por pantalla.

```
public static void main(String[] args) {
    try {
        File f = new File("Enteros.txt");
        FileWriter fw = new FileWriter(f);

        int valor = 1;

        for (int i = 1; i <= 20; i++) {
            fw.write("" + valor); // escribimos valor
            fw.write(" ");        // escribimos espacio en blanco
            valor = valor * 2;    // calculamos próximo valor
        }

        fw.write("\n"); // escribimos nueva línea

        fw.close(); // cerramos el FileWriter

        System.out.println("Fichero escrito correctamente");
    } catch (IOException e) {
        System.out.println("Error: " + e);
    }
}
```

Prueba a ejecutar el código varias veces. Verás que el archivo se sobrescribe y siempre queda igual. Luego, modifica la instanciación del `FileWriter` agregando el segundo argumento ("append") a `true`: `FileWriter fw = new FileWriter(f, true);` Probarlo y ver que ya no se sobrescribe el fichero, sino que se añaden los 20 números al final.

4. BIBLIOGRAFÍA

Parte del contenido de esta unidad didáctica es una traducción al castellano de los apuntes de programación de Joan Arnedo Moreno (Institut Obert de Catalunya, IOC).

También se ha utilizado como referencia las siguientes fuentes:

- [1] Apuntes de programación de Jose Luis Comesaña (sitiolibre.com).
- [2] Apuntes de programación de Natividad Prieto, Francisco Marqués y Javier Piris (E.T.S. de Informática, Universidad Politécnica de Valencia).

5. LICENCIA



[CC BY-NC-SA 3.0 ES](https://creativecommons.org/licenses/by-nc-sa/3.0/es/) Reconocimiento – No Comercial – Compartir Igual (by-nc-sa)

No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original. Esta es una obra derivada de la obra original de Carlos Cacho y Raquel Torres.

5.1 Agradecimientos

Apuntes actualizados y adaptados al CEEDCV a partir de la siguiente documentación:

- [1] Apuntes Programación de José Antonio Díaz-Alejo. IES Camp de Morvedre.