

Programación

Ficheros

Unidad 12

Jesús Alberto Martínez
versión 0.2



Reconocimiento – NoComercial – CompartirIgual (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.
Basado en los apuntes de WirzJava, del CEEDCV y de los apuntes de programación de Joan Arnedo Moreno (Institut Obert de Catalunya, IOC).



Unidad 12. Ficheros

| | |
|--|----|
| 1 Introducción..... | 3 |
| 2 Gestión de ficheros..... | 3 |
| 2.1 La clase File..... | 4 |
| 2.2 Rutas absolutas y relativas..... | 5 |
| 2.3 Métodos de la clase File..... | 7 |
| 2.3.1. Obtención de la ruta..... | 7 |
| 2.3.2. Comprobaciones de estado..... | 9 |
| 2.3.3. Propiedades de ficheros..... | 10 |
| 2.3.4. Gestión de ficheros..... | 11 |
| 2.3.5. Listado de archivos..... | 13 |
| 3 Lectura y Escritura de Ficheros..... | 15 |
| 3.1 Flujos..... | 15 |
| 3.2 Ficheros orientados a carácter..... | 17 |
| 3.3 Clases para la lectura de ficheros..... | 18 |
| FileReader..... | 18 |
| BufferedReader..... | 20 |
| InputStreamReader..... | 20 |
| Lectura de fichero (clase Scanner)..... | 21 |
| 3.4 Escritura en fichero..... | 25 |
| FileWriter..... | 25 |
| BufferedWriter..... | 27 |
| OutputStreamWriter..... | 27 |
| PrintWriter..... | 28 |
| Otras clases..... | 29 |
| 3.5 Lectura/Escritura con NIO..... | 29 |
| 4 Serialización..... | 30 |
| 4.1 Serial Version UID..... | 31 |
| 5 Clase Properties..... | 33 |
| 5.1 Escribir Fichero de Configuración..... | 33 |
| 5.2 Leer ficheros de configuración..... | 33 |
| 6 Acceso a Recursos Independientemente de la Ubicación en Java..... | 34 |
| 6.1 El problema de las rutas absolutas..... | 35 |
| 6.2 ClassLoader: La solución para recursos independientes de la ubicación..... | 35 |
| ¿Qué es el ClassLoader?..... | 35 |
| Estructura de directorios de recursos..... | 35 |
| 6.3 Métodos para acceder a recursos..... | 36 |
| Obtener recursos como stream..... | 36 |
| Obtener recursos como URL..... | 37 |
| Obtener recursos como URI o File..... | 37 |
| Aplicaciones empaquetadas en JAR..... | 37 |
| 6.4 usar getClassLoader().getResource() o directamente getResource()..... | 38 |
| Ejemplos prácticos:..... | 38 |
| Cuándo usar cada método:..... | 39 |
| Recomendación:..... | 39 |
| 6.5 Patrones de organización de recursos..... | 39 |

| | |
|---|----|
| 6.6 Path API en Java NIO..... | 40 |
| 7 Ampliación: Clases para Lectura / Escritura de Bytes..... | 41 |
| 7.1 Clases para escritura de bytes..... | 41 |
| FileOutputStream..... | 41 |
| DataOutputStream..... | 42 |
| BufferedOutputStream..... | 43 |
| 7.2 Clases para lectura de bytes..... | 44 |
| FileInputStream..... | 44 |
| DataInputStream..... | 45 |
| BufferInputStream..... | 45 |
| 8 Ampliación: Acceso aleatorio a ficheros..... | 46 |
| 9 Referencias y ampliación..... | 48 |

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 archivos.

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 en modo 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 el Windows 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. Además podemos usar la constante **`File.separator`** o el método **`System.getProperty("line.separator")`**

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 el caso de 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\Unitat11\apartado1 (ruta a una carpeta)
- N:\Documentos\Unitat11\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 le funcione perfectamente antes 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 Eclipse, IntelliJ Idea, etc), 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. Veámoslo con 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 a 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

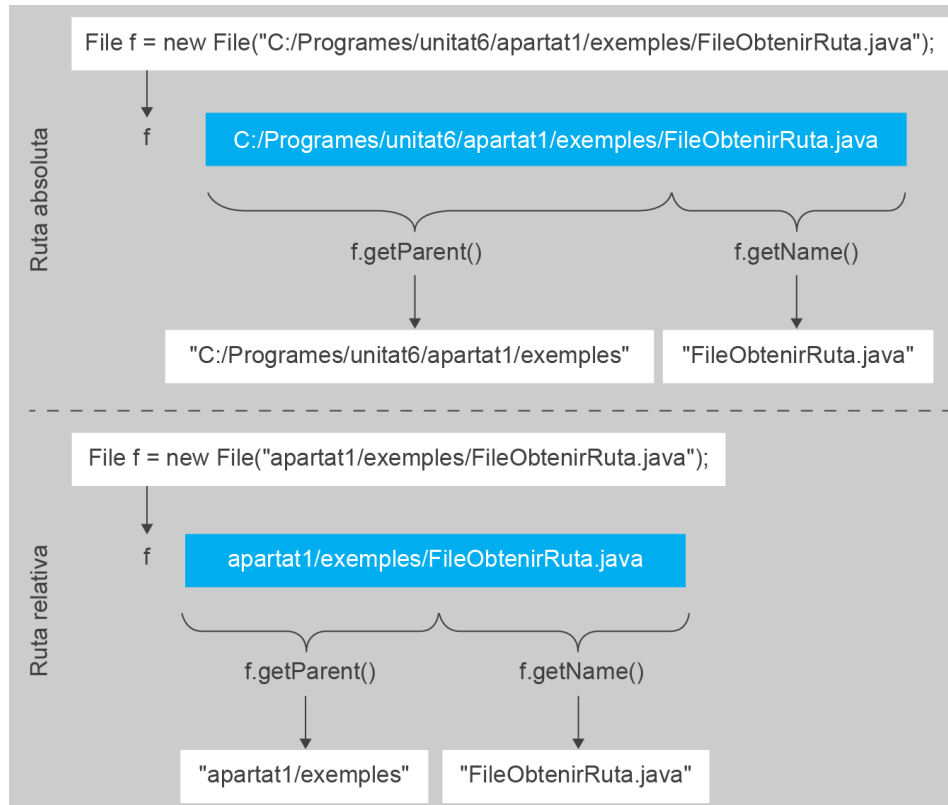
File ofrece varios métodos para poder manipular el sistema de archivos u obtener información a partir de su ruta. Algunos de los más significativos para entender las funcionalidades se muestran a continuación, ordenados por tipo de operación.

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. Obsérvese 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/albania1.jpg");

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

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



```

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

```

Este programa produce la salida:

```

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

getParent()      : /home/lionel/fotos
getName()        : albania1.jpg
getAbsolutePath(): /home/lionel/fotos/albania1.jpg

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

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

```

Ejercicio 1

Prueba el código anterior modificando las rutas absolutas, pero no modifiques las rutas relativas. ¿Cuál es el directorio de trabajo de las rutas relativas? ¿Ha aparecido algún fichero en el entorno del IDE?

El punto representa el directorio actual, prueba la salida con la ruta "." para ver el resultado.

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 "Document.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 document = new File("C:/Temp/Documento.txt");
    System.out.println(temp.getAbsolutePath()+" ¿existe? "+temp.exists());
    mostrarEstado(fotos);
    mostrarEstado(document);
}

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 ejecute 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/Document.txt");
    File destinoDoc = new File("C:/Temp/Document.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 con el contenido del directorio de trabajo.

```
public static void main(String[] args) {  
  
    File dir = new File(".");  
    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("[FIC] " + f.getName());  
        }  
    }  
}
```

Ejercicio 2

La clase File tiene muchos más métodos, busca un enlace donde nos muestre y explique brevemente en español todos los métodos de la clase.

Ejercicio 3

Realiza un programa que cree un nuevo directorio en nuestro directorio de trabajo. El directorio se llamará "archivos", dentro de este directorio crea un nuevo fichero con nombre "prueba.txt".

Copiar un archivo

La clase File no ofrece ningún método para copiar archivos. Podríamos hacerlo "a mano", creando un archivo con un flujo de escritura e ir leyendo del origen y escribiendo en el archivo destino.

Una forma más cómoda de hacerlo es mediante la clase `java.nio.file.Files` que tiene método `copy`, a la que le pasaremos ruta origen y ruta destino. Ejemplo:

```
import java.nio.file.Files;
. . .
File fo = new File("fichOrigen.txt");
File fd = new File("fichDestino.txt");
try {
    if (fd.exists()) fd.delete();
    Files.copy(f.toPath(), b.toPath());
} catch (IOException ex) {System.err.printf("Error:%s",ex.getMessage());}
```

La clase `Files` de `java.nio` proporciona una gran cantidad de nuevos métodos para trabajar con ficheros.

Ejercicio 4

Realiza los ejercicios contenidos en Ejercicios A.

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 solo 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 los paquetes de la API de Java `java.io` y `java.nio`, se encuentra un amplio conjunto de herramientas para la manipulación de la entrada y salida de datos del programa. Entre estas herramientas se encuentran aquellas que permiten tanto la manipulación del contenido de archivos como las que facilitan las comunicaciones en una red. Comenzaremos con las clases incluidas en `java.io`, que son subclases de `InputStream`, `OutputStream`, `Reader` y `Writer`. En el caso del paquete `java.nio`, la "n" que aparece en su nombre hace referencia a que fue introducido posteriormente a `java.io` y por tanto era "new I/O" en el momento de su introducción.

3.1 Flujos

En programación Java, los flujos (también conocidos como streams en inglés) son una secuencia de datos que se mueven de un lugar a otro, por ejemplo, desde un archivo en el disco duro hasta un programa en memoria. En Java, los flujos son representados por clases que proporcionan un conjunto de métodos para leer o escribir datos de entrada y salida.

Además de los flujos de entrada y salida, Java también proporciona flujos de bytes y flujos de caracteres para trabajar con diferentes tipos de datos. Los flujos de bytes se utilizan para trabajar con datos binarios, mientras que los flujos de caracteres se utilizan para trabajar con datos de texto. Las clases que leen o escriben flujos de caracteres contendrán en sus nombres `Reader` o `Writer`, las clases que leen o escriben flujo de bytes contendrán en sus nombres `InputStream` o `OutputStream`.

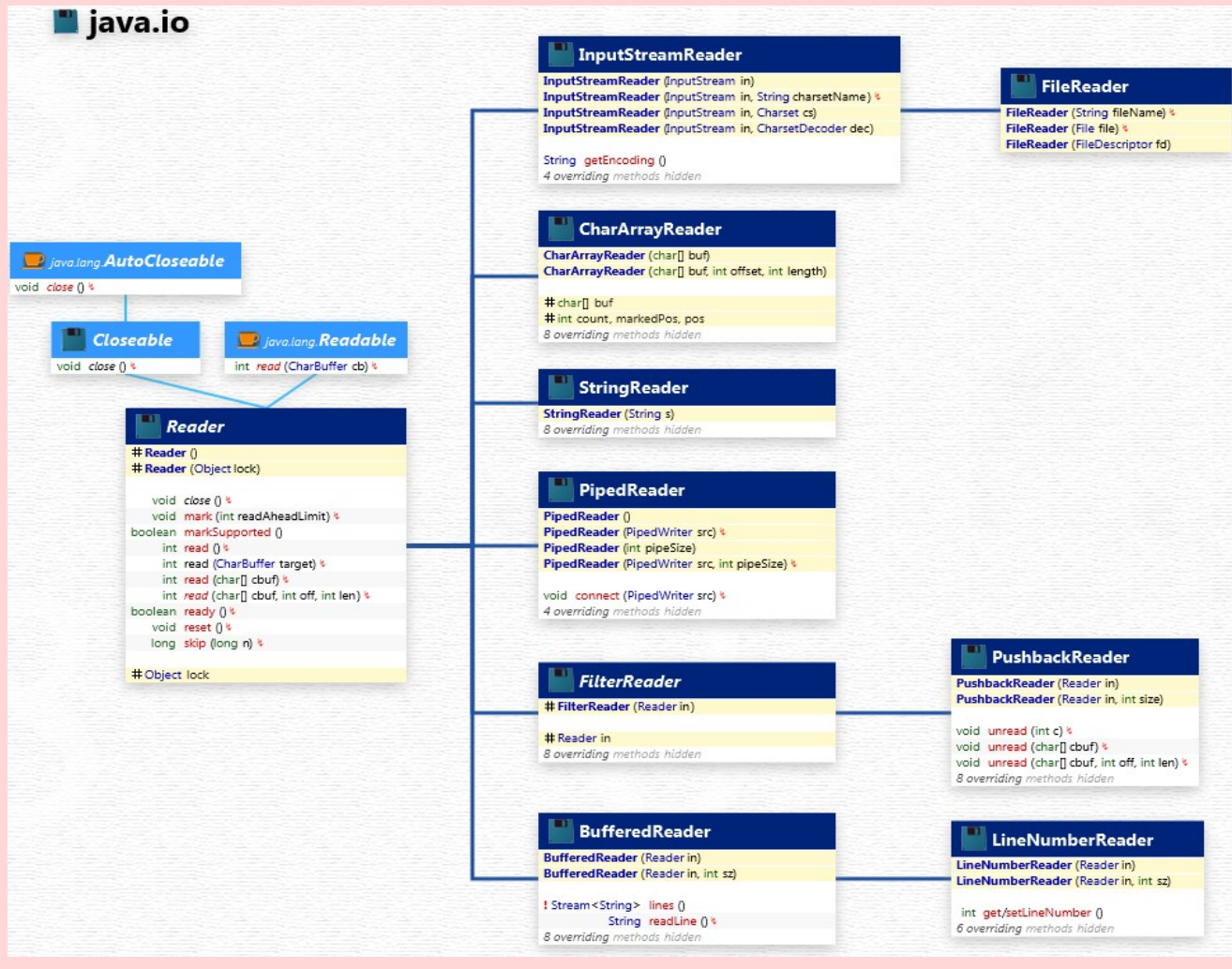
Hasta ahora hemos estado usando los flujos de entrada y salida estándar predeterminados en Java, obtenemos información a través del teclado y enviamos información a la consola, también si se produce algún error se mostrará en la consola, el flujo de error estándar. Estos flujos vienen implementados en los objetos `System.in`, `System.out` y `System.err`.

El tratamiento fundamental de datos de E/S se basa en la idea de la manipulación de los flujos, es decir, una secuencia de datos en la que se puede colocar nueva información (escribir) o consumir la información que contiene (lectura). Cuando en nuestros programas utilizamos clases de `java.io` para, por ejemplo, leer archivos, comunicar procesos en una red o hacer salidas a un terminal, estaremos utilizando diferentes tipos de flujos. En el caso de `java.nio`, aparece el concepto de canal que tendrá las mismas utilidades pero que llevará asociado un buffer (una zona de almacenamiento, que se utiliza de puente, y que intenta aumentar la velocidad de las operaciones de entrada/salida). Brevemente, este es el listado de los tipos de `Stream` que podremos utilizar:

- **`InputStream`, `OutputStream`**: Clases abstractas que definen las funcionalidades básicas para leer o escribir una secuencia de bytes sin estructura. El resto de flujos de bytes se construyen a partir de ellas.
- **`Reader`, `Writer`**: Clases abstractas que definen las funcionalidades básicas para leer o escribir una secuencia de caracteres, con soporte para la codificación Unicode. El resto de flujos de caracteres se construyen a partir de ellas.
- **`InputStreamReader`, `OutputStreamWriter`**: Clases que hacen de puente entre flujos de bytes y de caracteres haciendo las conversiones precisas según la codificación utilizada. Por ejemplo, con la codificación Unicode, un carácter no se corresponde con un byte.
- **`DataInputStream`, `DataOutputStream`**: Clases especializadas en permitir leer y escribir tipos de datos multibyte, como los tipos numéricos primitivos o los objetos `String`.
- **`ObjectInputStream`, `ObjectOutputStream`**: Clases especializadas en permitir serializar objetos y reconstruirlos.
- **`BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, `BufferedWriter`**: Clases especializadas en mejorar la eficiencia de las operaciones de lectura y escritura, tanto de bytes como de caracteres.
- **`PrintStream`, `PrintWriter`**: Clases para simplificar la lectura y escritura de texto.
- **`PipedInputStream`, `PipedOutputStream`, `PipedReader`, `PipedWriter`**: Clases que se usan por parejas para mover datos dentro de una aplicación. Los datos escritos en un `PipedOutputStream` o `PipedWriter` se leen usando el correspondiente `PipedInputStream` o `PipedReader`.
- **`FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter`**: Implementaciones de `InputStream`, `OutputStream`, `Reader` y `Writer` que leen y escriben usando archivos del sistema de archivos local.

Existen muchas más clases en la jerarquía de clases de java.io o java.nio. Busca imágenes donde puedas apreciar esta jerarquía.

Como ejemplo, la parte de Reader en <https://falkhausen.de/java-8/java.io/Reader.html>



3.2 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 un archivo siempre vamos a tener que realizar las siguientes operaciones:

- Abrir el archivo para iniciar la lectura/escritura
- Leer / Escribir en el archivo (de forma repetitiva hasta llegar al final del mismo o aleatoria a un punto concreto)
- Cerrar el archivo.

Todas estas operaciones pueden producir excepciones de tipo IOException por lo que los programas y clases que trabajen con ficheros deberán realizar estas operaciones en bloques try-catch capturando este tipo de excepciones.

3.3 Clases para la lectura de ficheros


FileReader

Es la clase base, la que da acceso a un fichero del sistema de archivos.

```
import java.io.FileReader;
import java.io.IOException;

public class FileReader {
    public static void main(String[] args) {
        FileReader reader = null;
        try {
            reader = new FileReader("ruta/de/l/archivo.txt");
            int character;
            while ((character = reader.read()) != -1) {
```

}

-  **try-with-resources:** Es una forma “especial” de usar try de modo que el cierre del flujo se realiza automáticamente tanto si ha habido alguna excepción como si no. Y por lo tanto no se hace necesaria la llamada al método close(). La sintaxis es la habitual de un try, pero añade la definición de los recursos entre paréntesis justo después de la palabra reservada try y antes de la llave de comienzo de bloque.
- ```
try (FileReader f = new FileReader("prueba.txt")) { . . . }
```

}

## BufferedReader

Permite leer texto sobre un flujo de texto, pero haciendo uso de un buffer intermedio haciendo que el rendimiento aumente.

Normalmente las clases que implementan un buffer van a trabajar sobre otro flujo de entrada/salida, es como si 'envolvieran' el flujo de entrada/salida original y añadieran una capa nueva con las funcionalidades del buffer.

Tendremos los siguientes métodos:

- **readLine():** Devuelve una línea entera del fichero.
- **mark():** Permite establecer una marca en el fichero para en el momento que queramos, volver a situarnos en la posición marcada.
- **reset():** Nos volvemos a posicionar en una posición marcada previamente.

Veamos un ejemplo de su uso con FileReader. **Será la estructura que más emplearemos para leer archivos de texto:**


```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Lectura {
 public static void main(String[] args) {
 try (BufferedReader br = new BufferedReader(new FileReader("datos.txt"))) {
 String linea;
 while ((linea = br.readLine()) != null) {
 System.out.println(linea);
 }
 } catch (IOException e) {
 System.err.println("Error al leer el archivo: " + e.getMessage());
 }
 }
}
```

## InputStreamReader

Permite leer un flujo de bytes convirtiéndolo a un flujo de caracteres, pudiendo especificar el juego de caracteres utilizado en la conversión.

```
File f = new File("fichero.txt");
String cadena;
try (FileInputStream fis = new FileInputStream(f);
 InputStreamReader isr = new InputStreamReader(fis, "UTF-8"); // "ISO-8859-1"
 BufferedReader bfr = new BufferedReader(isr)) {
 while ((cadena = bfr.readLine()) != null) {
 System.out.printf("%s\n", cadena);
 }
} catch (IOException ex) {
 System.err.printf("Error: %s\n", ex.getMessage());
}
```

 **La codificación de un fichero** se refiere a la forma en que se representan los caracteres en binario dentro del archivo. En otras palabras, es la forma en que se mapean los caracteres de un lenguaje a una secuencia de bits que pueda ser almacenada en un medio físico o electrónico.

Existen diferentes tipos de codificaciones, como UTF-8, UTF-16, ISO-8859-1, entre otras. Cada codificación utiliza diferentes patrones de bits para representar diferentes caracteres. Al leer un archivo, es importante saber la codificación utilizada para interpretar correctamente los caracteres en el archivo y mostrarlos de forma legible para el usuario.

## Lectura de fichero (clase Scanner)

Otra 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.

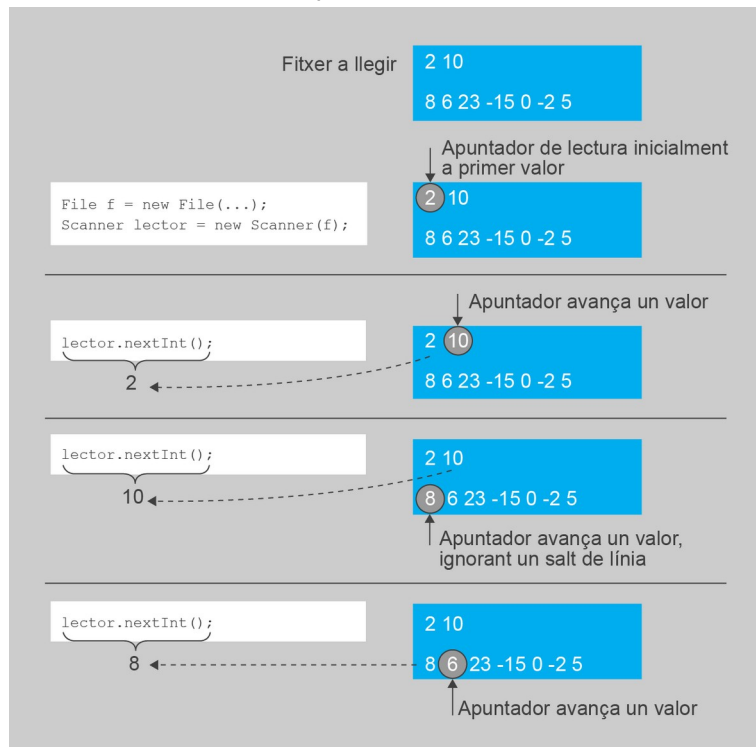
Por ejemplo, para crear un objeto de tipo Scanner de modo que permita leer datos desde el archivo ubicado en la ruta "C:\Programas\Unidad11\Documento.txt", habría que hacer:

```
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, recalando 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. Pero recuerda usar **`try-for-resources`** en la apertura de fijos, así evitaremos tener que cerrarlos nosotros.

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). Para probarlo, crea el archivo e introduce valores enteros separados por espacios en blanco o saltos de línea.

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

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

 // Intentamos abrir el fichero
 File f = new File("enteros.txt");
 try (Scanner lector = new Scanner(f)){
 // Si llega aquí es que ha abierto el fichero :)
 while (lector.hasNextInt()){
 int valor = lector.nextInt();
 System.out.println("El valor leído es: " + valor);
 }
 } catch (FileNotFoundException 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:

1. 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.
2. 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.

Otro ejemplo, supongamos que tenemos un archivo con datos en formato "csv" que representa un nombre, una edad y un número de teléfono, separados por algún carácter, en este caso ":" y queremos obtener sus datos para procesarlos.

Juan:28:5551234  
María:35:5555678  
Pedro:42:5559012  
Laura:19:5553456

```
String linea = "";
File fr = new File("datos.csv");
try (Scanner sc = new Scanner(fr)) {
 // Usamos try-for-resources
 while (sc.hasNextLine()) { // mientras que queden líneas
 linea = sc.nextLine();
 // Separamos los elementos de la línea especificando el carácter de separación
 // Nos devuelve un array
 String[] partes = linea.split(":");
 if (partes.length == 3) {
 System.out.printf("%nNombre:%s, edad:%s, teléfono:%s",
 partes[0], partes[1], partes[2]);
 }
 }
} catch (IOException ex) {
 System.err.printf("%nError:%s", ex.getMessage());
}
```

## Archivos csv

Un archivo 'csv' (comma-separated values) es un archivo de texto separado cada campo por un símbolo (en muchos casos, un punto y coma) y cada línea termina por un salto de línea. Es un formato habitual para convertir hojas de cálculo a archivos de texto.

La clase String dispone de un método llamado split () que obtiene de una cadena un array con las sub-cadenas comprendidas entre el delimitador pasado como parámetro. Ejemplo:

```
String[] partes = linea.split(",");
if (partes.length == 3)
 System.out.printf("%s,%s,%s\n",partes[0],partes[1],partes[2]);
else System.out.println("Error de formato");
```

El parámetro que recibe split puede ser una **expresión regular**, permite especificar patrones, por ejemplo si queremos obtener las palabras que componen una frase, el separador será el espacio en blanco pero una o varias veces: String[] partes = linea.split(" +"); Si queremos indicar que el punto o la coma también son separadores: =línea.split("[., ]+");



## 3.4 Escritura en fichero

### 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
```

**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. Ten en cuenta que si ya existía un archivo con ese nombre, quedará totalmente sobrescrito.

```
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
 // MEJOR USAR TRY-FOR-RESOURCES

 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);** Pruébalo y verás que ya no se sobrescribe el fichero, sino que se añaden los 20 números al final.

## BufferedWriter

Igual que el `BufferedReader` pero para la escritura. Dispone, entre otros, de métodos para escribir un salto de línea `newLine()`, y para escribir una cadena `write(String cadena)`. Por otra parte, el `close()` de `BufferedWriter` hace el cierre del fichero que envuelve.

Un ejemplo de uso:

```
BufferedWriter bfw=null;
try {
 File f = new File("fichero.txt");
 FileWriter fw = new FileWriter(f);
 bfw = new BufferedWriter(fw);
 bfw.write("Esto es un texto");
 bfw.newLine();
 bfw.write("Esto es otro texto");
 bfw.write(" que se escribirá en la misma línea");
} catch (IOException ex) {
 System.err.printf("Error:%s ",ex.getMessage());
}
finally {
 try{ if (bfw != null) bfw.close(); }
 catch(IOException e){
 System.err.printf("Error:%s",e.getMessage()); }
}
```

Si empleamos try-with-resources queda una estructura más sencilla. **Esta será la estructura habitual que podemos emplear:**

```
File f = new File("fichero.txt");
try(FileWriter fw = new FileWriter(f);
 BufferedWriter bfw = new BufferedWriter(fw)) {
 bfw.write("Esto es un texto");
 bfw.newLine();
 bfw.write("Esto es otro texto");
 bfw.write(" que se escribirá en la misma línea");
} catch (IOException ex) {
 System.err.printf("Error:%s",ex.getMessage());
}
```

## OutputStreamWriter

Al igual que se clase equivalente en lectura (`InputStreamReader`) esta clase es un puente entre un flujo de bytes y un flujo de caracteres. Va a convertir los bytes a caracteres, permitiendo escribir caracteres, array de caracteres y cadenas.

Por lo tanto, en su constructor deberemos enviarle un flujo de bytes, pudiendo abrir un fichero para añadir datos (como en la clase `InputStreamReader` podemos indicar el juego de caracteres a utilizar).

```
FileOutputStream fos = new FileOutputStream("fichero.txt",true);
OutputStreamWriter osw = new OutputStreamWriter(fos,"UTF-8");
```

Al igual que indicamos antes, vamos a poder 'envolver' los flujos para dar más funcionalidades:

```
FileOutputStream fos = new FileOutputStream("fichero.txt",true);
OutputStreamWriter osw = new OutputStreamWriter(fos,"UTF-8");
BufferedWriter bfw = new BufferedWriter(osw);
```

Este sería un ejemplo de como quedaría:

```
File f = new File("fichero.txt");
try (FileOutputStream fos = new FileOutputStream(f, true);
 OutputStreamWriter osw = new OutputStreamWriter(fos, "ISO-8859-1");
 BufferedWriter bfw = new BufferedWriter(osw)) {
 bfw.write("Esto es un texto");
 bfw.newLine();
 bfw.write("Esto es otro texto con eñe");
} catch (IOException ex) {
 System.err.printf("Error:%s", ex.getMessage());
}
```

## PrintWriter

Permite enviar cadenas de caracteres con un formato a un flujo de salida (Writer, File, OutputStream o Cadena con nombre del fichero) con métodos que ya utilizamos para la salida por consola: print(), println(), printf() .

Tiene varios constructores en los que le podemos pasar como parámetro un File o un OutputStream/FileWriter. En el primer caso se le puede pasar adicionalmente el tipo de codificación y en segundo caso un boolean que representa si el 'autoflush' está activado o no. Ejemplos:

```
File fichero = new File("fichero.txt");
try(PrintWriter pw = new PrintWriter(fichero,"UTF-8")){}
```

Los métodos de escritura de esta clase no lanzan excepciones IOException en caso de error. El programador debe hacer uso de los métodos:

**checkError():** Para comprobar si la llamada al método produjo un error.

**clearError():** Una vez gestionado el error se debe 'limpiar' para poder chequear un posterior.

Si está habilitado el 'autoflush', cada vez que se invoque a los métodos 'println', 'printf' o 'format' se escribirá en el fichero. Sino será necesario llamar al método 'flush' o cerrar el flujo para que se envíen.

Como en casos anteriores, podemos hacer que un FileWriter sea 'envuelto' por un 'BufferedWriter' para hacer uso de un buffer y aumentar el rendimiento y a su vez, el BufferedWriter será 'envuelto' por un PrintWriter para poder enviar al flujo de salida diferentes tipos de datos, como flotantes, booleano o cualquiera de los que soporta la clase PrintWriter.

De esta forma nos evitamos tener que estar convirtiendo los datos a enviar al flujo.

```
double num = 42.12;
try(FileWriter fw = new FileWriter("fichero.txt", true); //añade datos
 BufferedWriter bfw = new BufferedWriter(fw);
 PrintWriter pw = new PrintWriter(bfw, true))
{
 pw.printf("num=%06.1f\n", num);
 pw.println("línea nueva");
} catch (IOException ex) {
 System.err.printf("Error:%s", ex.getMessage());}
```

## Otras clases

Existen muchas otras clases con características y métodos específicos que pueden ser útiles en determinados contextos: CharArrayWriter, PipedWriter, StringWriter, etc.

## 3.5 Lectura/Escritura con NIO

Usando las librerías de java.nio usamos la interface Path y la clase Paths para hacer referencia a una ruta. Y usamos la clase Files con la ruta para leer, usamos streams para un uso más eficiente.

```
1 public class _351_LeerFicheroNIO {
2 public static void main(String[] args) {
3 String rutaArchivo = "Documentos/frases.txt";
4
5 // Leer el archivo línea por línea
6 try {
7 URI resourceUri = _351_LeerFicheroNIO.class.getClassLoader().getResource(rutaArchivo).toURI();
8 Path ruta = Paths.get(resourceUri);
9 Files.lines(ruta).forEach(System.out::println);
10 } catch (IOException | URISyntaxException e) {
11 System.err.println("Error al leer el archivo: " + e.getMessage());
12 }
13
14 // Introducir todas las líneas en una lista de String
15 try {
16 URI resourceUri = _351_LeerFicheroNIO.class.getClassLoader().getResource(rutaArchivo).toURI();
17 Path ruta = Paths.get(resourceUri);
18 List<String> lineas = Files.readAllLines(ruta);
19 for (String linea : lineas) {
20 System.out.println(linea);
21 }
22 } catch (IOException | URISyntaxException e) {
23 e.printStackTrace();
24 }
25
26 }
27 }
```

Ejemplo de escritura de archivo usando java.nio

```
1 public class _352_EscribirFicheroNIO {
2 public static void main(String[] args) {
3 Path path = Paths.get("Descargas/pruebaME.md");
4 List<String> lineas=List.of("uno", "dos", "tres");
5
6 // Escribimos todas las líneas como lista
7 try {
8 Files.write(path,lineas);
9 } catch (IOException e) {
10 e.printStackTrace();
11 }
12
13 // Para añadir líneas a un fichero usamos modificadores
14 try {
15 Files.writeString(path, "Nueva linea", StandardOpenOption.APPEND);
16 } catch (IOException e) {
17 e.printStackTrace();
18 }
19 }
20 }
```

## 4 Serialización

En Java disponemos de clases que permiten establecer un flujo de entrada (lectura) / salida (escritura) de objetos. **La serialización es el proceso por el que un objeto se convierte en una secuencia de bytes, permitiendo guardar su contenido en un archivo**

- Cuando escribimos un objeto a disco lo que hace la clase `ObjectOutputStream` es convertir el contenido de cada uno de los campos a binario y lo guarda en disco.
- Cuando leemos un objeto de disco, lo que hace la clase `ObjectInputStream` es leer un flujo de bytes que después, mediante un cast, guardará en cada uno de los campos del objeto.

**Para que un objeto pueda ser serializado debe de implementar la interface `java.io.Serializable`.** Esta interface no define ningún método, pero toda clase que la implemente, informará a la JVM que el objeto será serializado.

Todos los tipos primitivos de datos en Java son serializables al igual que los arrays. Si una clase tiene como 'dato' algún objeto de otra clase, esa otra clase debe implementar la interface `Serializable`.

Podemos indicarle a Java que un atributo de una clase no sea serializado (y por lo tanto no lo guardará) con la palabra clave **transient** de la forma:

```
private transient String ejemplAtributo;
```



#### Importante:

- La **serialización no permite añadir objetos a un archivo** conservando los que tuviera previamente. Esto es debido a que cada vez que se añade un objeto habiendo cerrado el archivo, se añade una cabecera. Si añadiésemos, se crearía una nueva cabecera cada vez, y los archivos tendrían un número indeterminado de cabeceras. Un `ObjectInputStream` solo va a leer una cabecera.
- **Cuidado con el método `readObject()`**. Este método no indica cuando se acaba el fichero, por lo que si estamos haciendo un bucle `while` leyendo objetos, la condición del `while` para salir podría ser llamando al método `available()` de un flujo de bytes asociado al fichero (leeríamos mientras el método devuelva un valor mayor que 0). Otra forma sería capturar la excepción `EOFException`.
- Cuando leemos un objeto, tenemos que hacer un cast a la clase a la que pertenece. Sin embargo, aplicando el **polimorfismo**, podemos hacer un cast a la clase 'común' pero aún así, el objeto tendrá toda la información de su clase original, de tal forma que si lo convertimos a su clase original (con otro cast) podremos acceder a todos sus métodos y propiedades.

## 4.1 Serial Version UID

Las clases que implementan la interfaz `Serializable` en Java deben incluir un campo `serialVersionUID`. Este campo es un identificador único de versión para la clase serializable que se utiliza durante la deserialización para garantizar que el objeto recibido sea compatible con la versión de la clase que se está utilizando. Si no se proporciona un `serialVersionUID` explícito, el compilador de Java generará uno automáticamente, pero esto puede ser problemático si se cambia la clase después de que los objetos hayan sido serializados.

Es importante destacar que el `serialVersionUID` debe ser único para cada versión de la clase serializable, es decir, cada vez que se realice una actualización en la clase, se debe actualizar también su `serialVersionUID`. De lo contrario, se pueden producir errores de deserialización y otros problemas.

Veamos en un ejemplo práctico. Partiendo de la siguiente clase:

```
1 public class _41_Persona implements Serializable{
2 private static final long serialVersionUID = 1L;
3 public String nombre;
4 public String telefono;
5 public float sueldo;
6
7 public _41_Persona(String nombre, String telefono, float sueldo) {
8 this.nombre = nombre;
9 this.telefono = telefono;
10 this.sueldo = sueldo;
11 }
12
13 @Override
14 public String toString() {
15 return String.format("Nombre:%s Telefono:%s Sueldo:%.2f%n",
16 nombre, telefono, sueldo);
17 }
18 }
```

Para escribir en fichero una lista de objetos de la clase Persona:

```
1 public static void escritura() {
2 List<_41_Persona> personas = new ArrayList<>();
3 personas.add(new _41_Persona("Pedro", "981222333", 1234.34f));
4 personas.add(new _41_Persona("Juan", "982444555", 2222.34f));
5
6 Path filePath = Paths.get(ruta);
7 try (ObjectOutputStream oos = new ObjectOutputStream(Files.newOutputStream(filePath))) {
8 for (_41_Persona persona : personas) {
9 oos.writeObject(persona);
10 }
11 } catch (IOException ex) {
12 System.err.println("Error al escribir el archivo: " + ex.getMessage());
13 }
14 }
```

La operación contraria, leer las instancias de Persona desde el disco sería como se muestra a continuación:

```
1 public static void lectura() {
2 List<_41_Persona> personas = new ArrayList<>();
3 Path filePath = Paths.get(ruta);
4
5 try (ObjectInputStream ois = new ObjectInputStream(Files.newInputStream(filePath))) {
6 while (true) {
7 try {
8 _41_Persona persona = (_41_Persona) ois.readObject();
9 personas.add(persona);
10 } catch (EOFException e) {
11 break;
12 }
13 }
14 } catch (IOException | ClassNotFoundException ex) {
15 System.err.println("Error al leer el archivo: " + ex.getMessage());
16 }
17
18 // Imprimir las personas leídas
19 personas.forEach(System.out::println);
20 }
```

Ejemplo completo en `_41_LecturaEscritura`



## 5 Clase Properties

---

Java dispone de librerías específicas para trabajar con ficheros de configuración, esto es ficheros que se componen de parejas de variables y valores, típicos en casi cualquier aplicación, servicio, etc.

Como todos siguen patrón similar, es la librería la que se encarga de acceder al fichero a bajo nivel y nosotros sólo tenemos que indicar la propiedad a leer/escribir.

Ejemplo:

```
Fichero de configuración
Thu Feb 13 10:49:39 CET 2020
user=usuario
password=mypassword
server=localhost
port=3306
```

### 5.1 Escribir Fichero de Configuración

---

Se crea el objeto Properties, se establecen los pares de valores que queremos guardar, y llama al método store para escribir el fichero.

```
1 // (import.java.util.Properties)
2 Properties config = new Properties();
3 config.setProperty("user", "pepito");
4 config.setProperty("password", "palotes");
5 config.setProperty("server", "llegamos");
6 config.setProperty("port", "3123");
7 try {
8 config.store(new FileOutputStream("config.props"), "Fichero de config.");
9 } catch (IOException ioe) {
10 ioe.printStackTrace();
11 }
```

### 5.2 Leer ficheros de configuración

---

A la hora de leerlo, en vez de tener que recorrer todo el fichero como suele ocurrir con los ficheros de texto, simplemente tendremos que cargarlo e indicar de qué propiedad queremos obtener su valor con getProperty(String).

```

1 Properties config = new Properties();
2 try {
3 config.load(new FileInputStream("config.props"));
4 String usuario = config.getProperty("user");
5 String password = config.getProperty("password");
6 String servidor = config.getProperty("server");
7 int puerto = Integer.valueOf(config.getProperty("port"));
8 System.out.println("Propiedades:"+usuario+": "+password+": "+servidor+": "+puerto);
9 } catch (IOException ioe) {
10 ioe.printStackTrace();
11 }

```

Tanto para escribir como para leer este tipo de ficheros, hay que tener en cuenta que, al tratarse de ficheros de texto, toda la información se almacena como si de un String se tratara. Por tanto, todos aquellos tipos Date, boolean o incluso cualquier tipo numérico serán almacenados en formato texto.

Así, habrá que tener en cuenta las siguientes consideraciones:

- Para el caso de las fechas, deberán ser convertidas a texto cuando se quieran escribir y nuevamente reconvertidas a Date cuando se lea el fichero y queramos trabajar con ellas
- Para el caso de los tipos boolean, podemos usar el método `String.valueOf(boolean)` para pasarlos a String cuando queramos escribirlos. En caso de que queramos leer el fichero y pasar el valor a tipo boolean podremos usar el método `Boolean.parseBoolean(String)`
- Para el caso de los tipos numéricos (integer, float, double) es muy sencillo ya que Java los convertirá a String cuando sea necesario al escribir el fichero. En el caso de que queramos leerlo y convertirlos a su tipo concreto, podremos usar los métodos `Integer.parseInt(String)`, `Float.parseFloat(String)` y `Double.parseDouble()`, según proceda.
- Las líneas que comienzan por '#' o '!' se interpretan como comentarios.
- En las parejas de propiedad valor el separador entre la clave y el valor puede ser '=' o bien ':'.

## 6 Acceso a Recursos Independientemente de la Ubicación en Java

En el desarrollo de aplicaciones Java, uno de los desafíos recurrentes es gestionar el acceso a recursos externos como archivos de configuración, imágenes, plantillas, archivos de datos, entre otros. El problema fundamental radica en que las rutas absolutas a estos recursos pueden variar dependiendo del entorno de ejecución, lo que complica la portabilidad del código.

Este documento explora las técnicas y mecanismos proporcionados por Java para acceder a recursos de manera independiente de la ubicación física, garantizando que nuestras aplicaciones funcionen correctamente independientemente de dónde se ejecuten: desde un entorno de desarrollo (IDE), como aplicación de escritorio, aplicación web o distribuida como un archivo JAR.

## 6.1 El problema de las rutas absolutas

---

Consideremos un escenario común: una aplicación Java necesita cargar un archivo de configuración llamado `config.properties`. Podemos acceder usando una ruta absoluta:

```
File configFile = new File("C:/MiAplicacion/config.properties");
```

Este código presenta varios problemas evidentes:

- Solo funcionará en sistemas Windows con esa estructura de directorios específica
- Fallará en entornos Linux o MacOS donde las rutas de archivos son diferentes
- No funcionará si la aplicación se distribuye como un JAR
- Requiere que el usuario tenga exactamente esa estructura de directorios

Una mejora parcial sería utilizar rutas relativas:

```
File configFile = new File("./resources/config.properties");
```

Aunque este enfoque es más flexible, sigue presentando problemas:

- La carpeta de trabajo actual (./) depende de cómo se inicie la aplicación
- Si se ejecuta desde diferentes ubicaciones, la ruta relativa puede no ser válida
- No funciona bien cuando la aplicación está empaquetada en un JAR

## 6.2 ClassLoader: La solución para recursos independientes de la ubicación

---

### ¿Qué es el ClassLoader?

El sistema de carga de clases de Java (ClassLoader) se encarga de cargar las clases, pero además también proporciona un mecanismo para acceder a recursos empaquetados junto con el código. Este enfoque permite:

- Acceso a recursos independientemente del sistema operativo
- Portabilidad entre diferentes entornos de ejecución
- Funcionamiento correcto cuando la aplicación está empaquetada en un JAR
- Organización coherente de recursos junto con el código

### Estructura de directorios de recursos

En un proyecto Java típico, los recursos suelen organizarse siguiendo una estructura como esta:

```

MiProyecto/
├── src/
│ ├── main/
│ │ ├── java/ # Código fuente Java
│ │ │ ├── com/
│ │ │ │ ├── miempresa/
│ │ │ │ │ ├── miapp/
│ │ │ │ │ │ ├── Main.java
│ │ │ │ │ │ └── ...
│ │ │ └── resources/ # Recursos de la aplicación
│ │ │ ├── config.properties
│ │ │ ├── images/
│ │ │ │ ├── logo.png
│ │ │ │ └── templates/
│ │ │ │ └── report.html
│ │ └── test/
│ │ ├── java/ # Código de prueba
│ │ └── resources/ # Recursos para pruebas
│ └── ...

```

Al compilar y empaquetar la aplicación, los recursos se copian junto a las clases compiladas, manteniendo la misma estructura relativa.

## 6.3 Métodos para acceder a recursos

### Obtener recursos como stream

El método más común para acceder a recursos es a través de un `InputStream`, usaremos el método `getResourceAsStream`:

```

// Usando el ClassLoader
InputStream inputStream = getClass().getClassLoader().getResourceAsStream("config.properties");

// O directamente desde la clase (si el recurso está en el mismo paquete o subpaquete)
InputStream inputStream = getClass().getResourceAsStream("config.properties");

```

Ejemplo completo para cargar un archivo de propiedades:

```

Properties properties = new Properties();
try (InputStream inputStream =
 getClass().getClassLoader().getResourceAsStream("config.properties")) {
 if (inputStream == null) {
 System.err.println("No se pudo encontrar el archivo config.properties");
 return;
 }
 properties.load(inputStream);
 String dbUrl = properties.getProperty("database.url");
 System.out.println("URL de la base de datos: " + dbUrl);
} catch (IOException e) {
 e.printStackTrace();
}

```

## Obtener recursos como URL

A veces, necesitamos la URL del recurso en lugar del contenido, esto nos lo proporciona el método `getResource`:

```
URL resourceUrl = getClass().getClassLoader().getResource("images/logo.png");
if (resourceUrl != null) {
 // Usar la URL, por ejemplo para cargar una imagen
 ImageIcon icon = new ImageIcon(resourceUrl);
}
```

## Obtener recursos como URI o File

Para recursos accesible como archivos podemos convertir la URL/URI de recurso a un objeto `File`

```
try {
 URL resourceUrl = getClass().getClassLoader().getResource("config.properties");
 if (resourceUrl != null) {
 File file = new File(resourceUrl.toURI());
 // Trabajar con el archivo
 }
} catch (URISyntaxException e) {
 e.printStackTrace();
}
```

**Aviso** Este uso no funcionará si la aplicación está empaquetada como JAR, ya que los recursos dentro de un JAR no son accesibles directamente como archivos.

## Aplicaciones empaquetadas en JAR

Cuando la aplicación se distribuye como JAR, los recursos se empaquetan dentro. El acceso a través del `ClassLoader` funciona perfectamente:

```
InputStream inputStream =
getClass().getClassLoader().getResourceAsStream("templates/report.html");
```

Sin embargo, no se puede convertir directamente a un objeto `File`, ya que los recursos están dentro del archivo JAR. Una solución es copiar el recurso a un archivo temporal:

```
try (InputStream inputStream =
getClass().getClassLoader().getResourceAsStream("templates/report.html")) {
 if (inputStream != null) {
 Path tempFile = Files.createTempFile("report", ".html");
 Files.copy(inputStream, tempFile, StandardCopyOption.REPLACE_EXISTING);
 // Usar tempFile...
 // No olvidar eliminar el archivo temporal cuando ya no se necesite
 tempFile.toFile().deleteOnExit();
 }
} catch (IOException e) {
 e.printStackTrace();
}
// Procesar el recurso
}
```

## 6.4 usar `getClassLoader().getResource()` o directamente `getResource()`

---

La diferencia radica principalmente en cómo interpretan las rutas a los recursos:

### 1. `class.getResource(String path)`:

- Interpreta las rutas **relativas al paquete de la clase** actual
- Si la ruta comienza con `/`, se considera absoluta desde la raíz del classpath
- Es útil cuando quieres acceder a recursos que están en el mismo paquete o en relación con la clase actual

### 2. `class.getClassLoader().getResource(String path)`:

- Siempre interpreta las rutas como **absolutas desde la raíz del classpath**
- No reconoce rutas que empiezan con `/` (o las interpreta sin el slash inicial)
- Es útil cuando quieres acceder a recursos con una ruta completa desde la raíz

### Ejemplos prácticos:

Supongamos que tienes una clase `com.miempresa.app.MiClase` y un recurso `config.properties` ubicado en diferentes lugares:

#### Caso 1: Recurso en el mismo paquete que la clase

```
src/main/resources/com/miempresa/app/config.properties
```

```
// Funciona - Ruta relativa al paquete de la clase
MiClase.class.getResource("config.properties");

// Funciona - Ruta absoluta desde la raíz
MiClase.class.getClassLoader().getResource("com/miempresa/app/config.properties");

// Funciona - Ruta absoluta usando /
MiClase.class.getResource("/com/miempresa/app/config.properties");
```

#### Caso 2: Recurso en la raíz del classpath

```
src/main/resources/config.properties
```

```
// NO funciona - Busca en el paquete de la clase
MiClase.class.getResource("config.properties");

// Funciona - Ruta absoluta desde la raíz
MiClase.class.getClassLoader().getResource("config.properties");

// Funciona - Ruta absoluta usando /
MiClase.class.getResource("/config.properties");
```

## Cuándo usar cada método:

1. Usa `class.getResource()` cuando:
  - Quieres acceder a recursos en el mismo paquete que tu clase
  - Prefieres usar rutas relativas al paquete actual
  - Necesitas indicar explícitamente rutas absolutas usando `/`
2. Usa `class.getClassLoader().getResource()` cuando:
  - Quieres acceder a recursos con su ruta completa desde la raíz
  - Necesitas consistencia en el acceso independientemente de la clase llamante
  - Trabajas con rutas estandarizadas en toda tu aplicación

## Recomendación:

Como regla general:

- Para recursos que están "junto" a la clase, usa `class.getResource("archivo.txt")`
- Para recursos con ubicación fija en la aplicación, usa `class.getClassLoader().getResource("ruta/al/archivo.txt")`

Esta elección contribuye a un código más mantenible y a una organización lógica de los recursos relacionados con cada clase.

## 6.5 Patrones de organización de recursos

---

Es recomendable seguir estas prácticas para organizar los recursos:

1. Estructura por tipo: Agrupar recursos por tipo (imágenes, configuraciones, plantillas)

```
resources/
├── config/
├── images/
└── templates/
```

2. Estructura por paquete: Reflejar la estructura de paquetes para asociar recursos con clases

```
resources/
├── com/
│ └── miempresa/
│ └── miapp/
│ ├── ui/
│ │ └── icons/
│ └── config/
```

## 6.6 Path API en Java NIO

---

Java NIO (desde Java 7) ofrece la interfaz Path y la clase Paths para trabajar con rutas de archivos de manera más flexible:

```
// Obtener un recurso como URL y convertirlo a Path (si es posible)
try {
 URL resourceUrl = getClass().getClassLoader().getResource("data/users.csv");
 if (resourceUrl != null) {
 Path resourcePath = Paths.get(resourceUrl.toURI());
 // Usar el Path
 List<String> lines = Files.readAllLines(resourcePath);
 }
} catch (URISyntaxException | IOException e) {
 e.printStackTrace();
}
```

Para recursos dentro de un JAR, se puede utilizar el sistema de archivos ZIP:

```
try {
 URI uri = getClass().getClassLoader().getResource("data/users.csv").toURI();
 Path path;

 if (uri.getScheme().equals("jar")) {
 FileSystem fileSystem = FileSystems.newFileSystem(uri, Collections.emptyMap());
 path = fileSystem.getPath("/data/users.csv");
 } else {
 path = Paths.get(uri);
 }

 // Usar el Path
 List<String> lines = Files.readAllLines(path);
} catch (URISyntaxException | IOException e) {
 e.printStackTrace();
}
```

**Ejemplo completo en la solución del ejercicio Ejercicio\_B1v3**



## 7 Ampliación: Clases para Lectura / Escritura de Bytes

---

Vamos a ver cuales son las clases para la lectura y escritura de bytes. Todas tienen como parte de su nombre XXXXStream para indicar que tenemos un flujo de bytes. Dependiendo de si leemos o escribimos, tendremos InputXXX o OutputXXX. Siempre que veamos una clase con parte de su nombre Stream sabremos que es una clase para lectura / escritura de bytes.

Para poder crear ficheros binarios o ver su contenido podemos hacer uso de cualquier editor binario de los que hay por la red, por ejemplo, okteta.

### 7.1 Clases para escritura de bytes

---

Van a permitir operaciones de escritura en forma de bytes.

En casi todas las clases de escritura de datos vamos a encontrar los siguientes métodos:

- **write(int byte):** Escribe un solo byte.
- **write(byte[] b):** Escribe de una sola vez los bytes indicados por el tamaño del array.
- **write(byte[] b, int offset, int length):** Escribe de una sola vez el número de bytes indicado por length desde la posición offset del array.

#### FileOutputStream

Permite escribir bytes a un archivo. El constructor en su primer parámetro contendrá una cadena con el nombre/ruta del archivo o bien un objeto de tipo File.

Opcionalmente, dispone de un segundo parámetro, de tipo boolean que en caso de valer true añade los datos al final del fichero y en caso de ser false (o no estar presente este segundo parámetro) borrará los datos que existiesen previamente.

Ejemplos:

- `FileOutputStream fs=new FileOutputStream ("fichero.dat");` Esta instrucción crea el fichero con nombre texto.txt. Si ya existiera borraría su contenido.
- `FileOutputStream fs=new FileOutputStream ("fichero.dat",true);` Esta instrucción si no existiera el fichero, lo crea y si ya existe añade datos al final.
- `FileOutputStream fs=new FileOutputStream (nombreFile);` La clase File incorpora nuevos métodos que nos permitirán comprobar si el fichero ya existe, antes de crearlo evitando que se pueda perder la información.

El método `write(int valor)` permite enviar un entero, pero lo que se guardará será un cast a un byte del valor enviado.

Ejemplo de uso:

```
byte[] datos = {1,2,3,4,5,6};
try(FileOutputStream fos = new FileOutputStream("/tmp/fichero.dat",true)){
 fos.write(257); // Escribirá el valor 01, ya que es un byte (byte) 257;
 fos.write(datos,2,3); // Guarda los valores 3,4 y 5
} catch (FileNotFoundException ex) {
 System.err.printf("%nError:%s",ex.getMessage());
} catch (IOException ex) {
 System.err.printf("%nError:%s",ex.getMessage());}
```

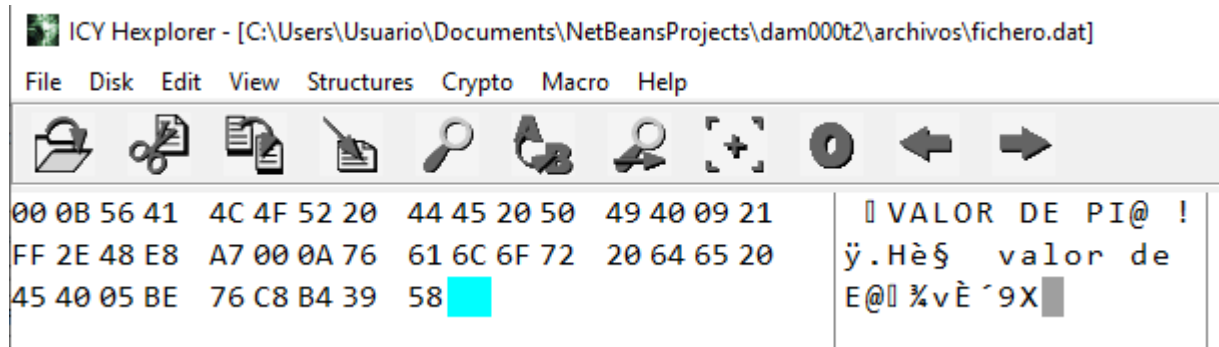
## DataOutputStream

Permite escribir en un fichero tipos de datos primitivos de Java. El constructor de esta clase espera recibir un OutputStream. Dispondremos de múltiples métodos writeXXXX() siendo XXXX un tipo primitivo de Java comenzando en mayúscula (Double, Float, Int, Char, Boolean)

Ejemplo de uso:

```
try (FileOutputStream fos = new FileOutputStream("archivos/fichero.dat");
 DataOutputStream dos = new DataOutputStream(fos);) {
 dos.writeUTF("VALOR DE PI");
 dos.writeDouble(3.1416);
 dos.writeUTF("valor de E");
 dos.writeDouble(2.718);
} catch (IOException ex) {
 System.err.printf("%nError:%s", ex.getMessage());
}
```

Obteniendo un fichero binario con los cuatro valores escritos. Como son datos binarios necesitaremos un visor de datos hexadecimales (por ejemplo, ICY Hexplorer en Windows):



Analicemos el archivo:

- Por cada carácter está utilizando dos bytes. Además, al usar el método writeUTF() envía en primer lugar la longitud de la cadena guardada.
  - 00 0B => 11 caracteres/bytes que ocupa la cadena y luego la cadena

- x56 => V x41 => A, x4C=L, etc...
- Después va el valor de PI. Como es un double, ocupa 8 bytes.
  - 40 09 21 FF 2E 48 E8 A7
- Y otra cadena
  - 00 0A => 10 caracteres/bytes que ocupa la cadena y luego la cadena
  - x76 => v x61 => a, x6C=l, etc...
- Y finalmente el valor de E:
  - 40 05 BE 76 C8 B4 39 58

Normalmente en un fichero binario de datos no vamos a tener 'saltos de línea'. Lo que tendremos es una serie de registros con un tamaño determinado para cada campo en los que van a estar todos seguidos uno detrás de otro. Sabiendo la longitud de cada campo, podremos leerlos.

## BufferedOutputStream

Al igual que todas las clases BufferXXX vistas, añade la funcionalidad de un buffer aumentando el rendimiento. Disponemos del método flush() para forzar a vaciar el buffer y que se escriba a disco.

Ejemplo de uso:

```
byte[] datos = {1,2,3,4,5,6,7};
try(FileOutputStream fis = new FileOutputStream("/tmp/datos.dat");
 BufferedOutputStream bfos = new BufferedOutputStream(fis);
) {
 bfos.write(15); // 0xF en hexadecimal
 bfos.write(datos,2,3); // Pos2 del array (desde índ. 3, empieza en 0) escribe 3 bytes
} catch (FileNotFoundException ex) {
 System.err.printf("%nError:%s", ex.getMessage());
} catch (IOException ex) {
 System.err.printf("%nError:%s", ex.getMessage());
}
```

Aunque la estructura habitual que utilizaremos será la siguiente:

```
try(FileOutputStream fis = new FileOutputStream("fichero.dat");
 BufferedOutputStream bfos = new BufferedOutputStream(fis);
 DataOutputStream dos = new DataOutputStream(fis);
) {
 dos.writeUTF("VALOR DE PI");
 dos.writeDouble(3.1416);
 dos.writeUTF("valor de E");
 dos.writeDouble(2.718);
} catch (IOException ex) {
 System.err.printf("%nError:%s", ex.getMessage());
}
```

## 7.2 Clases para lectura de bytes

---

Como norma general debemos de tener en cuenta que los Reader van a permitir leer caracteres en base a alguna codificación y nos van a suministrar caracteres, mientras que los Stream van a leer bytes que tendremos que convertir a caracteres (en caso necesario).

En casi todas las clases de lectura de datos vamos a encontrar los siguientes métodos:

- **read ()**: Lee un solo byte.
- **read(byte[] b)**: Lee de una sola vez los bytes indicados por el tamaño del array y devuelve el número de bytes leídos.
- **read(byte[] b,int offset, int length)**: Lee de una sola vez el número de bytes indicado por length y los guarda en el array pero a partir de la posición indicada por offset. Hay que tener cuidado que la suma de length y offset no sobrepase el tamaño del array.

En este enlace encontraréis una pequeña lista en la que se muestran las diferentes cabeceras de diferentes archivos binarios.

### FileInputStream

Obtiene bytes de un fichero del sistema operativo. El constructor tendrá un parámetro que contendrá un String con el nombre (y ruta) del fichero o bien un objeto de tipo File.

Si utilizamos una cadena para indicar el nombre del fichero debemos de controlar la excepción de que el fichero no exista (FileNotFoundException).

Disponemos del método available() que nos informa del número de bytes estimado que quedan por leer. Por tanto, si abrimos el fichero y llamamos al método antes de leer ningún dato nos informará del tamaño del fichero.

Queremos leerlo utilizando esta clase, dando como resultado la suma de todos ellos.

```
int dato = 0,suma = 0;
try (FileInputStream fis = new FileInputStream("/tmp/fichero.dat")){
 while((dato=fis.read())!=-1)
 suma += dato;
 System.out.print(suma);
} catch (FileNotFoundException ex) {
 System.err.printf("%nError:%s",ex.getMessage());
} catch (IOException ex) {
 System.err.printf("%nError:%s",ex.getMessage());
}
```

Veamos ahora otro ejemplo utilizando un array:

```

int numBytesLeídos = 0, suma = 0;
byte[] buf = new byte[5];
try (FileInputStream fis = new FileInputStream("/tmp/fichero.dat")){
 while((numBytesLeídos=fis.read(buf))≠-1){
 for(int cont=0;cont<numBytesLeídos;cont++){
 suma += buf[cont]& 0xFF;;
 }
 }
 System.out.print(suma);
} catch (FileNotFoundException ex) {
 System.err.printf("%nError:%s", ex.getMessage());
} catch (IOException ex) {
 System.err.printf("%nError:%s", ex.getMessage());
}

```

Recordar que el tipo byte lleva signo y por lo tanto el número 255 representa el valor -1. Si queremos pasarlo a decimal, debemos de hacer una operación de 'and' con el valor 255. En el ejemplo anterior no hizo falta hacerlo ya que guardamos la lectura en un entero y por tanto el valor guardado es el de 255. Si leemos datos de texto con esta clase, debemos de tener en cuenta que leeremos de byte en byte.

Tendremos que ser nosotros los que convirtamos a cadena con un cast ((char)byteleído)) teniendo en cuenta los problemas que podemos tener dependiendo del charset del fichero.

Tener en cuenta que, a pesar de ser un flujo de bytes, podríamos utilizar esta clase para que la clase InputStreamReader la 'envuelva' y podamos utilizarla como lectura de caracteres.

Esto ya lo comentamos al hablar de la clase InputStreamReader. Por ejemplo:

```

InputStreamReader isr = new InputStreamReader(new FileInputStream(nomFich),"UTF-8");

```

## DataInputStream

Esta clase permite leer tipos primitivos de datos. Su constructor espera recibir un InputStream y dispone de diferentes métodos readXXX() para leer tipos de datos de Java.

A nivel de cadenas es muy útil el método readUTF() que permite leer cadenas de caracteres guardadas con el juego de caracteres modificado UTF-8.

## BufferInputStream

Al igual que todas las clases BufferXXX implementa un buffer (un array internamente) para aumentar el rendimiento a la hora de realizar operaciones de lectura a disco. Recordar que puede ser utilizada por otras clases para que 'envuelvan' el buffer y aumentar su rendimiento.

Dispone de los métodos: read(), mark(), skip() y available().

Es importante destacar que cuando llega al final de fichero se provoca una excepción EOFException, que puede ser utilizada para determinar cuando acabamos la lectura del fichero.

Esta será la estructura que emplearemos habitualmente. En este ejemplo, leemos el fichero escrito previamente con `BufferedOutputStream`:

```
boolean eof = false;
try(FileInputStream fis = new FileInputStream("fichero.dat");
 BufferedInputStream bfis = new BufferedInputStream(fis);
 DataInputStream dis = new DataInputStream(bfis)) {
 while (!eof){
 String txt = dis.readUTF();
 double val = dis.readDouble();
 System.out.printf("%s→ %f\n", txt, val);
 }
}
catch (EOFException e) {eof = true;}
catch (IOException ex) {System.err.printf("%nError:%s", ex.getMessage());}
```

## 8 Ampliación: Acceso aleatorio a ficheros

Hasta ahora hemos recorrido los ficheros de forma secuencial. Esto es, para posicionarnos en una determinada posición del fichero debemos de leer previamente todas las posiciones anteriores.

Pero si tenemos un fichero en el que guardamos registros de datos, con un tamaño fijo determinado, puede ser útil que podemos posicionarnos en un registro determinado sin necesidad de pasar por los anteriores.

La clase que permite realizar estas operaciones es la clase `RandomAccessFile`. Además de esta ventaja, esta clase permite realizar operaciones de lectura/escritura sobre un archivo. No se necesitan dos clases como hasta ahora.

En el constructor enviaremos un objeto de la clase `File` o una cadena, que va a indicar el archivo a abrir, y un segundo parámetro (mode) que es una cadena que indica el modo de acceso al fichero:

"r": Lectura

"rw": Lectura / Escritura

Como podemos comprobar en el gráfico del principio, la clase `RandomAccessFile` incorpora las interfaces `DataInput` y `DataOutput`. Esto lleva consigo que esta clase va a disponer de los métodos `writeXXX` y `readXXX` de los tipos primitivos de Java, así como `writeUTF` y `readUTF`, pasando dichos datos a bytes.

Métodos más importantes:

- **`getFilePointer()`**: Devuelve la posición actual en el fichero.
- **`seek(long pos)`**: Permite posicionarse en la posición indicada por pos.
- **`length()`**: Tamaño del fichero.
- **`skipBytes(int num)`**: Desplaza la posición del puntero 'num' bytes.

Para añadir datos a un fichero debemos posicionarnos al final del mismo y llamar al método writeXXXX que queramos. Si nos situamos en la posición de un registro existente, los datos serán reemplazados al escribir.

Para ver un ejemplo, supongamos un registro con: String nombre (46 chars),int edad (4bytes) = 50 Bytes. El código para escribir añadir registros podría ser así:

```
try (RandomAccessFile raf = new RandomAccessFile("fichero.dat", "rw");) {
 raf.seek(raf.length());
// Nos posicionamos al final;
 raf.write(String.format("%46s", "Jose Perez").substring(0, 46).getBytes());
 raf.writeInt(20);
} catch (IOException ex) {
 System.err.printf("%nError:%s", ex.getMessage());
}
```

En el ejemplo anterior, vemos como el nombre (String) se almacena como un conjunto constante de 50B y no como vimos anteriormente, con dos Bytes previos que indicaban la longitud. Así logramos que todos los registros sean del mismo tamaño.

Veamos ahora el ejemplo inverso, que lee una posición determinada del archivo, en este caso la décima. Fíjate como tratamos la cadena sin los dos bytes iniciales que indicarían el tamaño (no podemos usar readUTF):

```
int pos = 10; int TAM_REGISTRO= 50;
try (RandomAccessFile raf = new RandomAccessFile("fichero.dat", "r");) {
 CantReg = (int) (raf.length()/50); //50 tamaño registro
 if (CantReg ≥ pos)
//ver si tiene al menos 10 reg
 raf.seek((pos-1)*TAM_REGISTRO);
 int edad = raf.readInt();
 byte[] modeloArray = new byte[46];
 raf.read(modeloArray);
 String nombre = new String(modeloArray);
} catch (IOException ex) {System.err.printf("%nError:%s",ex.getMessage());}
```

### Aspectos a tener en cuenta:

- Siempre hay que calcular el tamaño real de cada registro (sumando los bytes respectivos a los distintos campos que sean numéricos ej: 8 bytes si es un long, 4 si es int, tc....) Es necesario para posicionarse para leer y escribir con seek(). Cada caracter de un String ocupa generalmente 1 byte aunque depende de la codificación.
- Para borrar físicamente los registros que se desean dar de baja, hay que crear un fichero temporal para grabar en él los registros que siguen de alta ,luego se borra el fichero inicial con el

método delete() y a continuación se renombra el fichero temporal actualizado con el método rename().

- Es necesario una variable de tipo int ó long que contendrá la cantidad de registros del fichero, para no acceder fuera de rango, y se podrá calcular por medio de la siguiente fórmula:  $=(int) \text{Math.ceil}(\text{fichero.length()}/\text{tamañoRegistro})$ .

## 9 Referencias y ampliación

---

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