1.- Ficheros: Manejo de ficheros y carpetas.
Manenejo de entrada/salida

1 Manejo de ficheros y carpetas

1.1 La clase File.

La clase File es una representación abstracta de la ruta de un fichero o carpeta. Cuando creamos en Java un objeto de la clase File en representación de un fichero o carpeta concretos, no creamos el fichero al que se representa.

Así, la instrucción

```
File f = new File("c:\\alumnos.dat");
```

no crea el archivo alumnos.dat en la raíz del disco c, sino un objeto f que representa la ruta y nombre de dicho fichero.

El fichero indicado podría existir o no existir en el disco. Podríamos, por ejemplo, utilizar el objeto f anterior para crear el fichero indicado y escribir datos en el (en cuyo caso el fichero no existiría cuando se define el objeto f) o, por ejemplo, para eliminar un fichero existente con ese nombre y en esa ruta (en cuyo caso tendría que existir).

Los interfaces de usuario y los sistemas operativos utilizan Strings para representar las rutas de los ficheros y carpetas. Estos Strings son **dependientes** del sistema operativo y, así, por ejemplo, en Windows un fichero lo representaríamos con un String de la forma "c:\documentos\carta.doc", mientras que en sistemas basados en Linux "/documentos/carta.doc". A este respecto la clase File almacena la ruta del archivo o carpeta de una forma **independiente** del sistema como una secuencia de carpetas o directorios que componen la ruta más el nombre del fichero o carpeta. En el momento de convertir dicha ruta en un String se utilizará el sistema operativo huésped para determinar cuál es el carácter separador que hay que utilizar y se producirá un String con una ruta dependiente del sistema. El carácter separador se puede consultar a través de la propiedad del sistema *file.separator* o a través de los atributos estáticos *File.separator* o *File.separatorChar* de la clase *File.*

La clase File dispone de métodos que permiten realizar determinadas operaciones sobre los ficheros a los que representa. Podríamos, por ejemplo, crear un objeto de tipo File que represente a *c:\datos\libros.txt* y, a través de ese objeto File, realizar consultas relativas al fichero libros.txt, como su tamaño, atributos, etc, o realizar operaciones sobre él: borrarlo, renombrarlo,, …

1.2 Constructores.

La clase File tiene varios constructores, que permiten referirse, de varias formas, al archivo o carpeta que queremos representar:

public File (String ruta)	Crea el objeto File a partir de la ruta indicada. Si se trata de un archivo tendrá que indicar la ruta y el nombre.
public File (String ruta, String nombre)	Permite indicar de forma separada la ruta del archivo y su nombre
public File (File ruta, String nombre)	Permite indicar de forma separada la ruta del archivo y su nombre. En este caso la ruta está representada por otro objeto File.
public File (URI uri)	Crea el objeto File a partir de un objeto URI (UniformResourceIdentifier). Un URI permite representar un elemento siguiendo una sintaxis concreta, un estándar.

1.3 Métodos.

Aquí exponemos algunos métodos interesantes. Hay otros que puedes consultar en la documentación de Java

Relacionados con el nombre del fichero	
String getName()	Devuelve el nombre del fichero o directorio al que representa el objeto. (Solo el nombre, sin la ruta)
String getPath()	Devuelve la ruta del fichero o directorio. La ruta obtenida es dependiente del sistema, es decir, contendrá el carácter de separación de directorios que esté establecido por defecto. Este separador está definido en public static final String separator
String getAbsolutePath()	Devuelve la ruta absoluta del fichero o directorio.
String getParent()	Devuelve la ruta del directorio en que se encuentra el fichero o directorio representado. Devuelve null si no hay directorio padre.
Para hacer comprobaciones	
boolean exists() boolean canWrite() boolean canRead() boolean isFile() boolean isDirectory()	Permiten averiguar, respectivamente, si el fichero existe, si se puede escribir en el, si se puede leer de él, si se trata de un fichero o si se trata de un directorio
Obtener información de un fichero	
long length	Devuelve el tamaño en bytes del archivo. El resultado es indefinido si se consulta sobre un directorio o una unidad.
long lastModified	Devuelve la fecha de la última modificación del archivo. Devuelve el número de milisegundos transcurridos desde el 1 de enero de 1970
Para trabajar con directorios	
boolean mkdir()	Crea el directorio al cual representa el objeto File.
boolean mkdirs()	Crea el directorio al cual representa el objeto File,

	incluyendo todos aquellos que sean necesarios y no existan.
String[] list()	Devuelve un array de Strings con los nombres de los ficheros y directorios que contiene el directorio al que representa el objeto File.
String[] list(FileNameFilter filtro)	Devuelve un array de Strings con los nombres de los ficheros y directorios que contiene el directorio al que representa el objeto File y que cumplen con determinado filtro.
public File[] listFiles()	Devuelve un array de objetos File que representan a los archivos y carpetas contenidos en el directorio al que se refiere el objeto File.
Para hacer cambios	
boolean renameTo(File nuevoNombre)	Permite renombrar un archivo. Hay que tener en cuenta que la operación puede fracasar por muchas razones, y que será dependiente del sistema: Que no se pueda mover el fichero de un lugar a otro, que ya exista un fichero que coincide con el nuevo, etc. El método devuelve true solo si la operación se ha realizado con éxito. Existe un método move en la clase Files para mover archivos de una forma independiente del sistema.
boolean delete()	Elimina el archivo o la carpeta a la que representa el objeto File. Si se trata de una carpeta tendrá que estar vacía. Devuelve true si la operación tiene éxito.
boolean createNewFile()	Crea un archivo vacío. Devuelve true si la operación se realiza con éxito.
File createTempFile(String prefijo, String sufijo)	Crea un archivo vacío en la carpeta de archivos temporales. El nombre llevará el prefijo y sufijo indicados. Devuelve el objeto File que representa al nuevo archivo. El fichero creado no se borra automáticamente salvo que se invoque al método deleteOnExit del objeto File correspondiente.
void deleteOnExit()	Se asegura de que el fichero al que representa el objeto se eliminará cuando termine la ejecución de la máquina virtual de java.

Ejemplo: Mostrar información y contenido de una carpeta

```
package _02Ejemplos;
import java.io.*;
importjava.util.*;

publicclass _01InformacionCarpeta {
    publicstaticvoid main(String[] args) {

        Scanner tec = new Scanner(System.in);
        System.out.println("Introduce ruta absoluta de una carpeta");
        StringnombreCarpeta = tec.nextLine();
        //Creamosobjeto File pararepresentar a lacarpeta
        File car = new File(nombreCarpeta);
        //Comprobamossiexiste
        if (car.exists()){
            //¿Esunacarpeta?
            if(car.isDirectory()){
                if(car.canRead()) System.out.println("Lectura permitida");
```

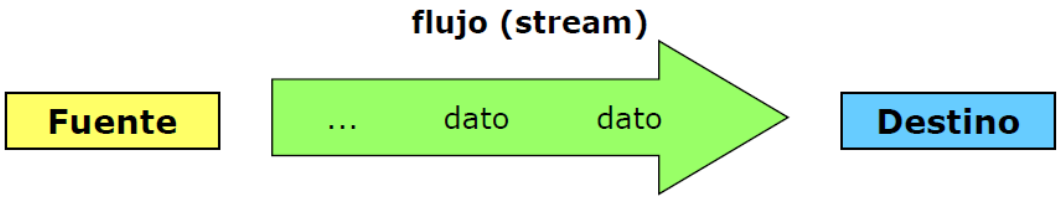
```
                else System.out.println("Lectura no permitida");

                if(car.canWrite()) System.out.println("Escritura permitida");
                else System.out.println("Escritura no permitida");

                if(car.isHidden()) System.out.println("Carpeta oculta");
                else System.out.println("Carpeta visible");

                System.out.println("---- Contenido de la carpeta ----");
                File[] contenido = car.listFiles();
                for(File f: contenido){
                    System.out.println(f.getName());
                }
            }
            else System.out.println(car.getAbsolutePath() + "No es una carpeta" );
        } else System.out.println("No existe la carpeta" + car.getAbsolutePath());
    }
}
```

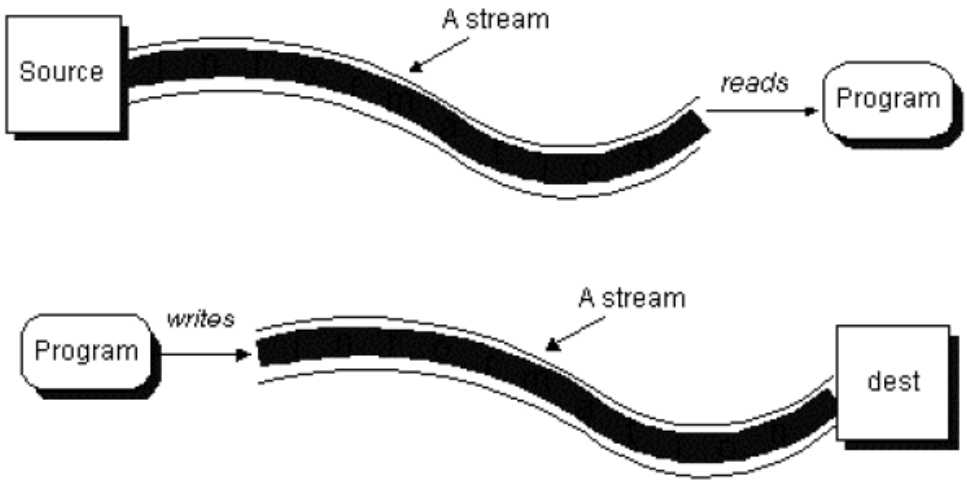
2 Entrada y salida

2.1 Streams

Los programas Java realizan las operaciones de entrada y salida a través de lo que se denominan ***Streams*** (traducido, flujos).

Un stream es una abstracción de todo aquello que produzca o consuma información. Podemos ver a este stream como una entidad lógica que, por otra parte, se encontrará vinculado con un dispositivo físico. La eficacia de esta forma de implementación radica en que las operaciones de entrada y salida que el programador necesita manejar son las mismas independientemente del dispositivo con el que estemos actuando. Será Java quien se encargue de manejar el dispositivo concreto, ya se trate del teclado, el monitor, un sistema de ficheros o un socket de red, etc., liberando a nuestro código de tener que saber con quién está interactuando.





Clasificación de los Streams

En Java los Streams se materializan en un conjunto de clases y subclases, contenidas en el paquete java.io. Todas las clases para manejar streams parten, de cuatro clases abstractas:

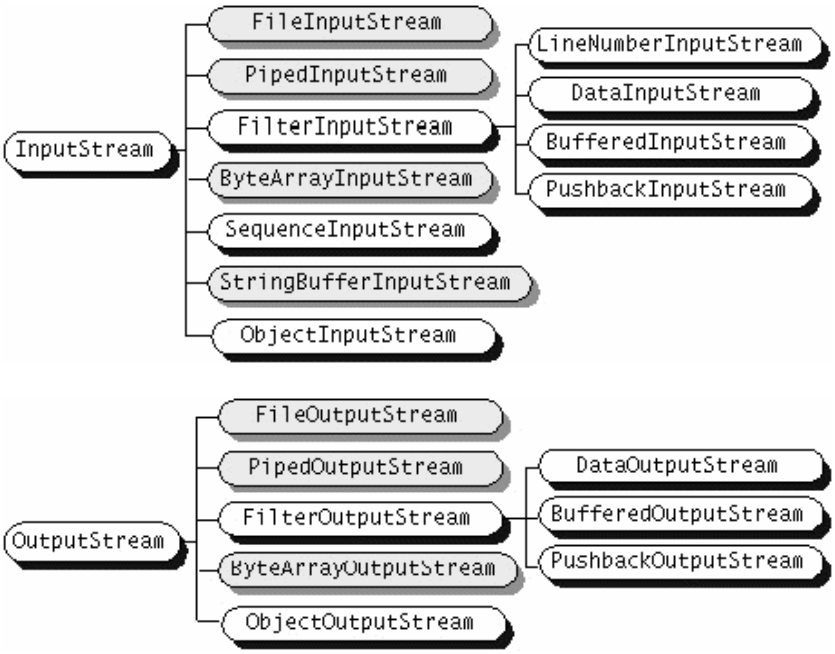
- InputStream
- OutputStream
- Reader
- Writer

Streams	Orientados a **bytes**	Orientados a **carácter**
Para **lectura**	InputStream	Reader
Para **escritura**	OutputStream	Writer

2.1.1 Streams orientados a byte (byte streams)

Proporcionan un medio adecuado para el manejo de entradas y salidas de bytes y su uso lógicamente está orientado a la lectura y escritura de **datos binarios**. El tratamiento del flujo de bytes viene gobernado por dos clases abstractas que son ***InputStream*** y ***OutputStream***.

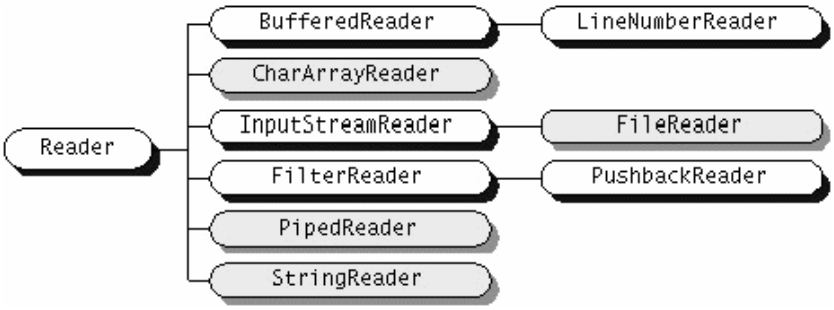
Cada una de estas clases abstractas tiene varias subclases concretas que controlan las diferencias entre los distintos dispositivos de I/O que se pueden utilizar. Así mismo, estas dos clases son las que definen los métodos que sus subclases tendrán implementados y, de entre todas, destacan las operaciones read() y write() que leen y escriben bytes de datos respectivamente.

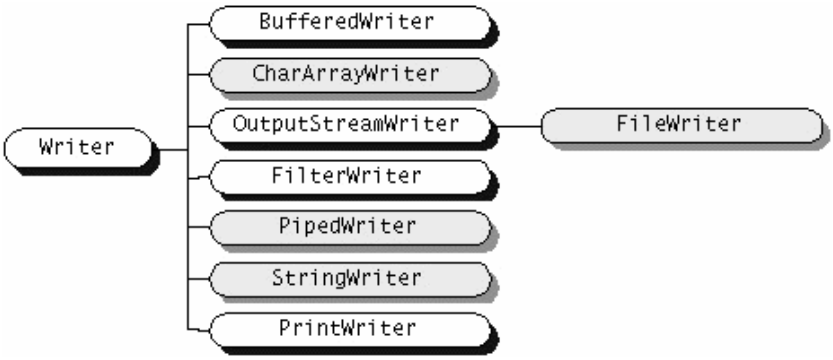


2.1.2 Streams orientados a caracter (character streams)

Proporciona un medio conveniente para el manejo de entradas y salidas de caracteres. Dichos flujos usan codificación Unicode y, por tanto, se pueden internacionalizar.

Una observación: Este es un modo que Java nos proporciona para manejar caracteres pero al nivel más bajo todas las operaciones de I/O son orientadas a byte. Al igual que la anterior el flujo de caracteres también viene gobernado por dos clases abstractas: ***Reader*** y ***Writer***. Dichas clases manejan flujos de caracteres Unicode. Y también de ellas derivan subclases concretas que implementan los métodos definidos en ellas siendo los más destacados los métodos read() y write() que, en este caso, leen y escriben caracteres de datos respectivamente.





Como hemos comentado, tanto en Reader como en InputStream encontramos un método read(), en concreto *public int read().* Observar la diferencia entre estos dos métodos nos ayudará a comprender la diferencia entre los flujos orientados a byte y los orientados a carácter.

método	devuelve
int InputStream.read()	valor entre 0 y 255
int Reader.read()	valor entre 0 y 65535

A pesar de llamarse igual, InputStream.read() devuelve el siguiente **byte** de datos leído del stream. El valor que devuelve está entre 0 y 255 (ó -1 si se ha llegado al final del stream). Reader.read(), sin embargo, devuelve un valor entre 0 y 65535 (ó -1), correspondiente al siguiente **carácter** simple leído del stream.

Stream estándar

Existen una serie de stream de uso común a los cuales se denomina stream estándar. El sistema se encarga de crear estos stream automáticamente.

- **System.in**
  - Instancia de la clase InputStream: flujo de bytes de entrada
  - Metodos
    - read() permite leer un byte de la entrada como entero
    - skip(n) ignora n bytes de la entrada
    - available() número de bytes disponibles para leer en la entrada
- **System.out**
  - Instancia de la clase PrintStream: flujo de bytes de salida
  - Metodos
    - para impresión de datos: print(), println()
    - flush() vacía el buffer de salida escribiendo su contenido
- **System.err**
    - Funcionamiento similar a System.out
    - Se utiliza para enviar mensajes de error (por ejemplo a un fichero de log o a la consola)

Por defecto, System.in, System.out y System.err se encuentran asociados a la consola (teclado y pantalla), pero es posible redirigirlos a otras fuentes o destinos, como por ejemplo a un fichero.

Utilización de Streams

Para utilizar un stream hay que seguir una serie de pasos:

- Lectura
  - Abrir el stream asociado a una fuente de datos (creación del objeto stream)
    - Teclado
    - Fichero
    - Socket remoto
  - Mientras existan datos disponibles
    - Leer datos
  - Cerrar el stream (método close)
- Escritura
  - Abrir el stream asociado a una fuente de datos (creación del objeto stream)
    - Pantalla
    - Fichero
    - Socket local
  - Mientras existan datos disponibles
    - Escribir datos
  - Cerrar el stream (método close)

Nota: Los Stream estándar ya se encarga el sistema de abrirlos y cerrarlos

Nota: Un fallo en cualquier punto del proceso produce una IOException

Las clases InputStream y OutputStream

Como hemos dicho anteriormente, proporcionan métodos para leer y escribir, respectivamente, **un byte** de información, a través de sus métodos read() y write()

Clase	métodos	descripción
InputStream	int read()	Lee un byte de información y lo devuelve como un entero cuyo valor estará entre 0 y 255. Si se detecta el final de los datos de entrada, devuelve -1
OutputStream	write(int b)	Escribe un byte de información en el Stream. El parámetro es entero, pero si su valor es superior a 255 se escriben los 8 bits de menos peso (los más a la derecha)

Reader y Writer

Permiten, respectivamente, leer y escribir **un carácter** en el stream.

Clase	métodos	descripción
Reader	int read()	Lee un carácter unicode de información y lo devuelve como un entero cuyo valor estará entre 0 y 65565. Si se detecta el final de los datos de entrada, devuelve -1
Writer	write(int c)	Escribe un carácter unicode en el Stream. El parámetro es entero y corresponderá al código Unicode del carácter que se escribe.

InputStreamReader y OutputStreamWriter

Son clases que actúan de puente entre streams orientados a bytes y streams orientados a carácter. Podemos, por ejemplo, crear un InputStreamReader asociado a un InputStream y leer caracteres del InputStream asociado, a través del InputStreamReader

Clase	métodos	Descripción
InputStreamReader	InputStreamReader(InputStream) int read()	Constructor: El objeto se crea a partir de un InputStream (orientado a byte). Leerá información del inputStream asociado y la devolverá en forma de caracteres. Se puede indicar el charset a utilizar. Lee un carácter del InputStream asociado.
OutputStreamWriter	OutputStreamWriter(OutputStream)	Constructor: Crea el objeto asociándolo a un OutputStream, en el que escribirá bytes. Se

		puede indicar el charset a usar.
	write(int c)	Escribe el carácter indicado en el OutputStream asociado.

Buffering

Las clases BufferedReader, BufferedWritter, BufferedInputStream y BufferedOutputStream permiten realizar buffering.

Situadas "por delante" de un stream acumulan las operaciones de lectura y escritura en una memoria o buffer y cuando hay suficiente información las operaciones se realizan finalmente sobre el dispositivo físico.

Clase	métodos	Descripción
BufferedReader BufferedWriter	String readLine() void newLine() write(String s)	Además de los métodos heredados, encontramos otros que permiten leer Strings completos, escribir una línea completa de texto y hacer saltos de línea
BufferedInputStream, BufferedOutputStream		Mantienen las mismas operaciones de lectura y escritura que sus clases padre pero, como hemos dicho, reducen el número de accesos al dispositivo físico por el uso de buffers

DataInputStream y DataOutputStream

Realizan una transformación de la información antes de ser escrita o después de ser leída. Los bytes leídos o escritos se interpretan como datos correspondientes a los tipos primitivos de Java

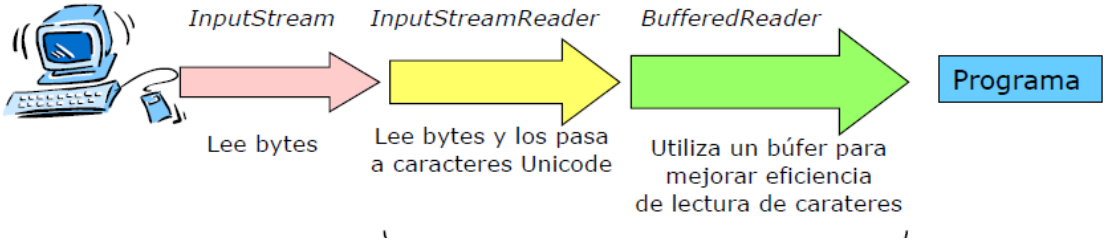
Clase	métodos	Descripción
DataInputStream, DataOutputStream	read(), write() readInt(), writeInt() readDouble(), writeDouble() readUTF(), writeUTF() …	Que permiten leer y escribir información correspondiente a los distintos tipos de datos de Java

PrintWriter

clase	métodos	Descripción
PrintWriter	print() println()	Esta clase (a la que pertenece System.out) tiene los conocidos métodos print y println, que escriben en el stream de salida datos binarios representados en forma de cadenas de caracteres.

Combinación de Streams

En muchas ocasiones, una sola clase de las vistas no nos da la funcionalidad necesaria para poder hacer la tarea que se requiere. En tales casos es necesario combinar varios Streams de manera que unos actúan como origen de información de los otros, o unos escriben sobre los otros.



En este caso tendríamos que combinar tres clases:

BufferedReader b = new BufferedReader(new InputStreamReader(System.in));

2.2 Ficheros

En ocasiones necesitamos que los datos que introduce el usuario o que produce un programa **persistan** cuando éste finaliza, es decir, que se conserven cuando el programa termina su ejecución. Para ello es necesario el uso de una base de datos o de **ficheros**, que permitan guardar los datos en un almacenamiento secundario como un pendrive, disco duro, DVD, etc.

En esta unidad se abordan distintos aspectos relacionados con el almacenamiento en ficheros:

- Introducción a conceptos básicos como los de registro y campo
- Clasificación de los ficheros según el contenido y forma de acceso.
- Operaciones básicas con ficheros de distinto tipo.

Campos y registros

Llamamos ***campo*** a un dato en particular almacenado en una base de datos o en un fichero. Un campo puede ser en nombre de un cliente, la fecha de nacimiento de un alumno, el número de teléfono de un comercio. Los campos pueden ser de distintos tipos: alfanuméricos, numéricos, fechas, etc.

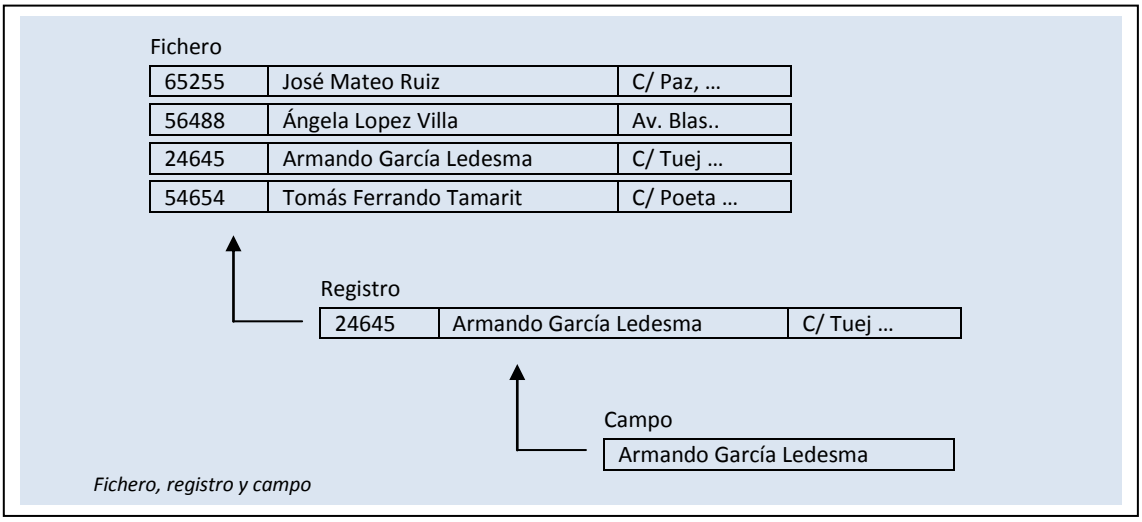
La agrupación de uno o más campos forman un ***registro.*** Un registro de alumno podría consistir, por ejemplo, de los siguientes campos

1. Número de expediente.
2. Nombre y apellidos
3. Domicilio
4. Grupo al que pertenece.

Un ***fichero*** puede estar formado por registros, lo cual lo dotaría al archivo de estructura. En un fichero de alumnos tendríamos un registro por cada alumno. Los campos del registro serían cada uno de los datos que se almacena del alumno: Nº expediente, nombre, etc …

En **Java** no existen específicamente los conceptos de campo y registro. Lo más similar que conocemos son las clases (similares a un registro) y, dentro de las clases, los atributos (similares a campos).

Tampoco en Java los ficheros están formados por registros. Java considera los archivos simplemente como flujos secuenciales de bytes. Cuando se abre un fichero se asocia a él un ***flujo*** (***stream***) a través del cual se lee o escribe en el fichero.



Fichero, registro y campo

Ficheros de texto vs ficheros binarios.

Desde un punto de vista a muy bajo nivel, un fichero es un conjunto de bits almacenados en memoria secundaria, accesibles a través de una ruta y un nombre de archivo.

Este punto de vista a bajo nivel es demasiado simple, pues cuando se recupera y trata la información que contiene el fichero, esos bits se agrupan en unidades mayores que las dotan de significado. Así, dependiendo de cuál es el contenido del fichero (de cómo se interpretan los bits que contiene el fichero), podemos distinguir dos tipos de ficheros:

- Ficheros de texto (o de caracteres)
- Ficheros binarios (o de bytes)

Un ***fichero de texto*** está formado únicamente por caracteres. Los bits que contiene se interpretan atendiendo a una tabla de caracteres, ya sea ASCII o Unicode. Este tipo de ficheros se pueden abrir con un editor de texto plano y son, en general, legibles. Por ejemplo, los ficheros .java que contienen los programas que elaboramos, son ficheros de texto.

Por otro lado, los ***ficheros binarios*** contienen secuencias de bytes que se agrupan para representar otro tipo de información: números, sonidos, imágenes, etc. Un fichero binario se puede abrir también con un editor de texto plano pero, en este caso, el contenido será ininteligible. Existen muchos ejemplos de ficheros binarios: el archivo .exe que contiene la versión ejecutable de un programa es un fichero binario.

Las operaciones de lectura/escritura que utilizamos al acceder desde un programa a un fichero de texto están orientadas al carácter: leer o escribir un carácter, una secuencia de caracteres, una línea de texto, etc. En cambio las operaciones de lectura/escritura en ficheros binarios están orientadas a byte: se leen o escriben datos binarios, como enteros, bytes, double, etc.

Acceso secuencial vs acceso directo.

Existen dos maneras de acceder a la información que contiene un fichero:

- Acceso secuencial
- Acceso directo (o aleatorio)

Con ***acceso secuencial,*** para poder leer el byte que se encuentra en determinada posición del archivo es necesario leer, previamente, todos los bytes anteriores. Al escribir, los datos se sitúan en el archivo uno a continuación del otro, en el mismo orden en que se introducen. Es decir, la nueva información se coloca en el archivo a continuación de la que ya hay. No es posible realizar modificaciones de los datos existentes, tan solo añadir al final.

Sin embargo, con el **acceso directo**, es posible acceder a determinada posición (dirección) del fichero de manera directa y, posteriormente, hacer la operación de lectura o escritura deseada.

No siempre es necesario realizar un acceso directo a un archivo. En muchas ocasiones el procesamiento que realizamos de sus datos consiste en la escritura o lectura de todo el archivo siguiendo el orden en que se encuentran. Para ello basta con un acceso secuencial.

Streams para trabajar con ficheros.

Para trabajar con ficheros disponemos de las siguientes clases

Streams para ficheros	Ficheros binarios	Ficheros de texto
Para **lectura**	FileInputStream	FileReader
Para **escritura**	FileOutputStream	FileWriter

FileReader proporciona operaciones para leer de un fichero uno o varios caracteres

FileWriter permite escribir en un fichero uno o varios caracteres o un String.

FileInputStream permite leer bytes de un fichero.

FileOutputStream permite escribir bytes de un fichero.

Consulta en la documentación los distintos constructores disponibles para estas clases.

2.2.1 Crear un fichero

En el siguiente ejemplo vemos como crear un fichero de texto y escribir una frase en el.

```
package _02Ejemplos;
import java.io.*;
public class _01CrearFicheroTexto {
   public static void main(String[] args) {
      FileWriter f = null;
      try {
         f = new FileWriter("texto.txt");
         f.write("Este texto se escribe en el fichero\n\r");

      } catch (IOException e) {
         System.out.println("Problema al abrir o escribir ");
      } finally {
         if (f != null)
            try {
               f.close();
            } catch (IOException e) {
               System.out.println("Problema al cerrar el fichero");
            }
      }
   }
}
```

----- *Ejemplo 1* -----

La creación del FileWriter puede provocar IOException, lo mismo que el método write. Por ello las instrucciones se encuentran en un bloque try-catch

Al finalizar su uso, y tan pronto como sea posible, hay que **cerrar** los streams (close)

2.2.2 Sobreescribir un fichero

Es muy importante tener en cuenta que cuando se crea un FileWriter o un FileOutputStream y se escribe en él …

- … si el fichero no existe se crea
- … si el fichero existe, **su contenido se reemplaza** por el nuevo. El contenido previo que tuviera el fichero se pierde.

Es posible escribir en un fichero indicando que la información se añada a la que ya hay y no se reescriba el fichero. Para ello usaremos el constructor que recibe el parámetro ***append*** y lo pasaremos a *true.*

El siguiente ejemplo muestra como añadir una línea al final de un fichero de texto.

```
package _02Ejemplos;
import java.io.*;
public class _02AnyadirAFicheroTexto {
   public static void main(String[] args) {
      try (FileWriter f = new FileWriter("texto.txt", true);) {
         f.write("Este texto se añade en el fichero\n\r");

      } catch (IOException e) {
         System.out.println("Problema al abrir o escribir ");
      }
   }
}
```

----- *Ejemplo 2* -----

En este ejemplo se ha utilizado la nueva sintaxis disponible para los bloques try-catch: lo que se denomina "*try with resources*". Esta sintaxis permite crear un objeto en la cabecera del bloque try. El objeto creado se cerrará automáticamente al finalizar. El objeto debe pertenecer al interface Closeable, es decir, debe tener método close().

2.2.3 Lectura y escritura de información estructurada.

Si observamos la documentación de las clases FileInputStream y FileOutputStream veremos que las operaciones de lectura y escritura son muy básicas y permiten únicamente leer o escribir uno o varios bytes. Es decir, son operaciones de muy bajo nivel. Si lo que queremos es escribir información binaria más compleja, como por ejemplo un dato de tipo double o boolean o int, tendríamos que hacerlo a través de un stream que permitiese ese tipo de operaciones y asociarlo al FileOutputStream o FileOutputStream. Podríamos, por ejemplo, asociar un DataInputStream a un FileInputStream para leer del fichero un dato de tipo int.

En ejemplos posteriores se ilustrará cómo asociar un stream a un File…Stream.

Operaciones con ficheros de acceso secuencial.

Como hemos comentado anteriormente el acceso secuencial a un fichero supone que para acceder a un byte es necesario leer previamente los anteriores. Suele utilizarse este tipo de acceso cuando es necesario leer un archivo de principio a fin.

Vamos a ver una serie de ejemplos que muestren cómo leer y escribir secuencialmente un fichero.

Lectura de un fichero secuencial de texto.

Leer un fichero de texto y mostrar el número de vocales que contiene.

```
package _03Ejemplos;
import java.io.*;
public class _04ContarVocales {
   final static String VOCALES = "AEIOUaeiou";
   public static void main(String[] args) {
      try (FileReader f = new FileReader(new File("texto.txt"));) {
         int contadorVocales = 0;
         int caracter;
         while((caracter=f.read())!=-1){
            char letra = (char) caracter;
            if(VOCALES.indexOf(letra)!=-1) contadorVocales++;
         }
         System.out.println("Numero de vocales: " + contadorVocales);
      } catch (FileNotFoundException e) {
         System.out.println("Probrema al abrir el fichero");
      } catch (IOException e){
         System.out.println("Problema al leer");

      }
   }
}
```

----- Ejemplo 3 -----

Observa que:

- Para leer el fichero de texto usamos un InputReader.
- Al crear el stream (InputReader) es posible indicar un objeto de tipo File

- La operación read() devuelve un entero. Para obtener el carácter correspondiente tenemos que hacer una conversión explícita de tipos.
- La operación read() devuelve -1 cuando no queda información que leer del stream.
- La guarda del bucle while combina una asignación con una comparación. En primer lugar se realiza la asignación y luego se compara carácter con -1.
- *FileNotFoundException* sucede cuando el fichero no se puede abrir (no existe, permiso denegado, etc), mientras que *IOException* se lanzará si falla la operación read()

Escritura de un fichero secuencial de texto.

Dada una cadena escribirla en un fichero en orden inverso

```
package _02Ejemplos;
import java.io.*;
public class _04EscribirCadenaInvertida {
	final static String CADENA = "En un lugar de la mancha...";
	public static void main(String[] args) {
		try (FileWriter f = new FileWriter(new File("texto.txt"));) {
			for(int i=CADENA.length()-1; i>=0; i--){
				f.write(CADENA.charAt(i));
			}
			System.out.println("FIN");
		} catch (FileNotFoundException e) {
			System.out.println("Probrema al abrir el fichero");
		} catch (IOException e){
			System.out.println("Problema al escribir");

		}
	}
}
```

----- Ejemplo 4 -----

Observa que:

- Para escribir el fichero de texto usamos un FileWriter.
- Tal y como se ha creado el stream, el fichero (si ya existe) se sobreescribirá.
- El manejo de excepciones es como el del caso previo.

Ficheros con buffering.

Cualquier operación que implique acceder a memoria externa es muy costosa, por lo que es interesante intentar reducir al máximo las operaciones de lectura/escritura que realizamos sobre los ficheros, haciendo que cada operación lea o escriba muchos caracteres. Además, eso también permite operaciones de más alto nivel, como la de leer una línea completa y devolverla en forma de cadena.

En el libro Head First Java, describe los buffers de la siguiente forma: *"Si no hubiera buffers, sería como comprar sin un carrito: debería llevar los productos uno a uno hasta la caja. Los buffers te dan un lugar en el que dejar temporalmente las cosas hasta que está lleno. Por ello has de hacer menos viajes cuando usas el carrito."*

Las clases BufferedReader, BufferedWritter, BufferedInputStream y BufferedOutputStream permiten realizar buffering. Situadas "por delante" de un stream de fichero acumulan las

operaciones de lectura y escritura y cuando hay suficiente información se llevan finalmente al fichero.

En el siguiente codigo se usan buffers para leer líneas de un fichero y escribirlas en otro convertidas a mayúsculas

```
package _02Ejemplos;
import java.io.*;
public class _05DuplicadoEnMayusculas {
    final static String ENTRADA = "texto.txt";
    final static String SALIDA = "textoMayusculas.txt";
    public static void main(String[] args) {
        try (
                BufferedReader fe = new BufferedReader(new FileReader(ENTRADA));
                BufferedWriter fs = new BufferedWriter(new FileWriter(SALIDA))
        ){
            String linea;
            while ((linea = fe.readLine())!=null){
                fs.write(linea.toUpperCase());
                fs.newLine();
            }
            System.out.println("FIN");
        } catch (FileNotFoundException e) {
            System.out.println("Probrema al abrir el fichero");
        } catch (IOException e){
            System.out.println("Problema al leer o escribir");
        }
    }
}
```

----- *Ejemplo 5* -----

Observa que:

- Usamos buffers tanto para leer como para escribir. Esto permite minimizar los accesos a disco.
- Los buffers quedan asociados a un FileReader y FileWriter respectivamente. Realizamos las operaciones de lectura/escritura sobre las clases Buffered… y cuando es necesario la clase accede internamente al stream que maneja el fichero.
- Es necesario escribir explícitamente los saltos de línea. Esto se hace mediante el método newLine(). newLine() permite añadir un salto de línea sin preocuparnos de cuál es el carácter de salto de línea. El salto de línea es distinto en distintos sistemas: en unos es /n, en otros /r, en otros /n/r, …
- BufferedReader dispone de un método para leer líneas completas (readLine()). Cuando se llega al final del fichero este método devuelve null.
- Fíjate como el bloque *try with resources* creamos varios objetos. Si la creación de cualquiera de ellos falla, se cerrarán todos los stream que se han abierto.

Escritura de un fichero binario secuencial.

Ya hemos visto que con FileInputStream y FileOutputStream se puede leer y escribir bytes de información de/a un archivo.

Sin embargo esto puede no ser suficiente cuando la información que tenemos que leer o escribir es más compleja y los bytes se agrupan para representar distintos tipos de datos.

Imaginemos por ejemplo que queremos guardar en un fichero “jugadores.dat”, el año de nacimiento y la estatura de cinco jugadores de baloncesto:

```
package _02Ejemplos;
import java.io.*;
public class _06JugadoresBaloncesto {
    public static void main(String[] args) {
        Scanner tec = new Scanner(System.in);
        try (
                DataOutputStream fs = new DataOutputStream(
                                      new BufferedOutputStream(
                                          new FileOutputStream("jugadores.dat")));
        ){
            for(int i = 1 ; i<=5 ; i++){
                //Pedimos datos al usuario
                System.out.println(" ---- Jugador " + i + " -----");
                System.out.print("Nombre: ");
                String nombre = tec.nextLine();

                System.out.print("Nacimiento: ");
                int anyo= tec.nextInt();

                System.out.print("Estatura: ");
                double est= tec.nextDouble();
                //Vaciar salto linea
                tec.nextLine();

                //Volcamos información al fichero
                fs.writeUTF(nombre);
                fs.writeInt(anyo);
                fs.writeDouble(est);
            }
        } catch (FileNotFoundException e) {
            System.out.println("Probrema al abrir el fichero");
        } catch (IOException e){
            System.out.println("Problema al leer o escribir");
        }
   }}
```

----- *Ejemplo 6* -----

Observa que:

- Para escribir información binaria usamos un DataInputStream asociado al stream. La clase tiene métodos para escribir int, byte, double, boolean, etc, etc.
- Además, como hemos hecho en ejemplos previos, usamos un buffer. Fíjate como en el constructor se enlazan unas clases con otras.
- A pesar de que en Java los ficheros son secuencias de bytes, estamos dotando al fichero de cierta estructura: primero aparece el nombre, luego el año y finalmente la estatura. Cada uno de estos tres datos constituirían un registro de formado por tres campos. Para poder recuperar información de un fichero binario es necesario conocer cómo se estructura ésta dentro del fichero.

Lectura de un fichero binario secuencial.

```
package _02Ejemplos;
import java.io.*;
public class _07LeerNombresJugadores {
   public static void main(String[] args) {
      Scanner tec = new Scanner(System.in);
```

```
      try (
            DataInputStream fe = new DataInputStream(
                                    new BufferedInputStream(
                                        new FileInputStream("jugadores.dat")));
      ){
         while(true){
            //Leemos nombre
            System.out.println(fe.readUTF());

            //leemos y desechamos resto de datos
            fe.readInt();
            fe.readDouble();
         }
      } catch(EOFException e){
         //Se lanzará cuando se llegue al final del fichero
      } catch (FileNotFoundException e) {
         System.out.println("Probrema al abrir el fichero");
      } catch (IOException e){
         System.out.println("Problema al leer o escribir");
      }
   }
}
```

----- Ejemplo 7 -----

Observa que:

- A pesar de que necesitamos solamente el nombre de cada jugador, es necesario leer también el año y la estatura. No es posible acceder al nombre del primer jugador sin leer previamente todos los datos del primer jugador.
- La lectura se hace a través de un bucle infinito (while (true)), que finalizará cuando se llegue el final del fichero y al leer de nuevo se produzca la excepción EOFException