PROG06:
Almacenando Datos. Ficheros

Objetivos

Realiza operaciones de entrada y salida de información, utilizando procedimientos específicos del lenguaje y librerías de clases así como el almacenamiento y la recuperación de información de ficheros.

CONTENIDOS

- Concepto de Consola
- Concepto de E/S
- Librerías de E/S
- Métodos de acceso al contenido de ficheros
- Concepto de flujo
- Flujos predefinidos
- Aplicaciones de almacenamiento de información en ficheros
- Sistemas de Ficheros
- Ficheros y directorios
- Almacenamiento de objetos en archivos: Persistencia y serialización

CONCEPTO DE ENTRADA Y SALIDA

Introducción

El esquema general de cualquier programa viene a ser:

Entrada → proceso → Salida

Esta entrada de datos hasta ahora ha sido en variables en la RAM temporal, pero a veces se hace necesario almacenar esta información para volver a utilizarla en otras ocasiones: la solución **FICHEROS** es lo que llamamos **datos persistentes**.

A todas las operaciones que constituyen un flujo de información del programa con el exterior se les conoce como operaciones (E/S) y ya hemos trabajado con alguna de ellas en temas anteriores.

CONCEPTO DE ENTRADA Y SALIDA

Introducción

Distinguimos dos tipos de E/S:

- E/S estándar que se realiza con el terminal del usuario (usada hasta ahora)
- E/S a través de ficheros

Todas las operaciones de E/S se encuentran en la librería **java.io**

CONTROL DE FLUJO Y CLASES RELATIVAS A FLUJOS

Streams o flujos

Un flujo es una abstracción de aquello que produzca o consuma información. Es una entidad lógica.

La clase **Stream** representa un flujo o corriente de datos, es decir, un conjunto secuencial de bytes, como puede ser un archivo, un dispositivo de entrada/salida (en adelante E/S), memoria, un conector TCP/IP (Protocolo de Control de Transmisión/Protocolo de Internet), etc.

Cualquier programa realizado en Java que necesite llevar a cabo una operación de E/S lo hará a través de un **stream**

En Java abrir un archivo supone:

- ***crear un objeto que queda asociado con un flujo/stream***
- ***Que cualquier operación(método) que se haga en un programa es sobre el flujo /stream independientemente del dispositivo al que este asociado (archivo, impresora, teclado…)***

CONTROL DE FLUJO Y CLASES RELATIVAS A FLUJOS

Streams

La entrada/salida de Java se organiza generalmente mediante objetos llamados Streams

Streams es por tanto la generalización (**Clase**) de un fichero:

- Una secuencia de datos con un origen y un destino
- Su origen o destino puede ser un fichero, un string, un dispositivo de E/S (teclado, pantalla,…) o un puerto de comunicación (socket)

Crearemos objetos Stream que asociaremos a los dispositivos de E/S

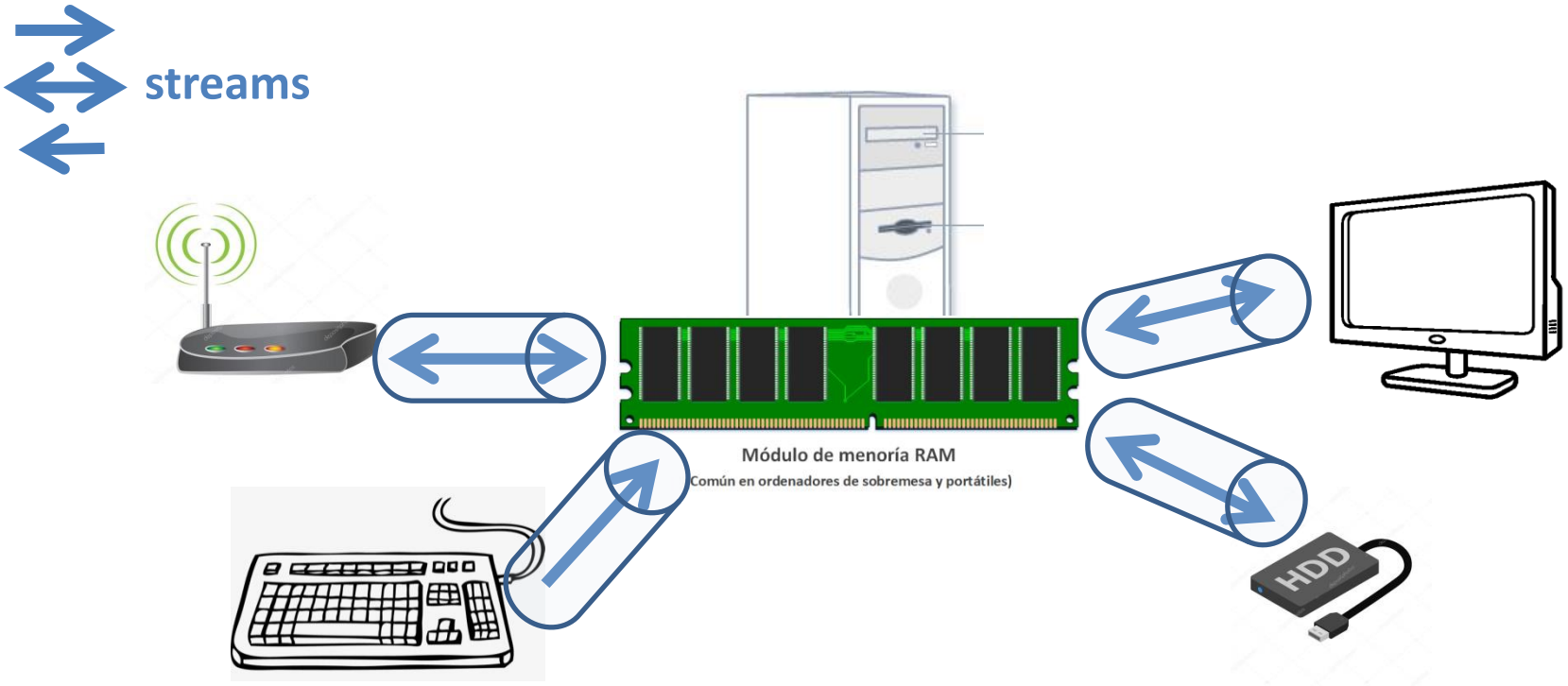
CONTROL DE FLUJO Y CLASES RELATIVAS A FLUJOS

Streams

En java:

toda la información que entra fuera del ordenador a la memoria

o que sale de la memoria fuera del ordenador se hace como **streams**



CONTROL DE FLUJO Y CLASES RELATIVAS A FLUJOS

Clasificación de los Streams

- En función de los datos que transportan:
  - **Binarios** (bytes)
  - **De caracteres(** de texto)

- En función del sentido de flujo de datos:
  - De **entrada**: datos de un dispositivo o fichero hacia el programa (**input**)
  - De **salida**: datos del programa hacia un dispositivo o fichero (**output**)

- Según la cercanía al dispositivo:
  - **Iniciadores**: son los que directamente recogen o vuelcan los datos al dispositivo
  - **Filtro**: se sitúan entre un stream iniciador y el programa para hacer algún tipo de transformación

CONTROL DE FLUJO Y CLASES RELATIVAS A FLUJOS

Tipos de Streams

Existen dos tipos de flujos o Streams:

- **Flujos de bytes** (**byte streams**) **de 8 bits**. Permite leer bytes. Se usan para manipular datos binarios solo legibles por la máquina o un programa. Orientado a lectura y escritura de datos binarios. Son utilizados para leer archivos de extensión jpeg, png, xls, gif,…. Es decir si abrimos el fichero con un editor de texto no entenderíamos nada

  Clases: **InputStream** y **OutputStream**

- **flujos de caracteres** (**characters streams**) **de 16 bits**. Permite leer caracteres. Se usan para manipular datos legibles por humanos (p.e. fichero de texto, csv,…).

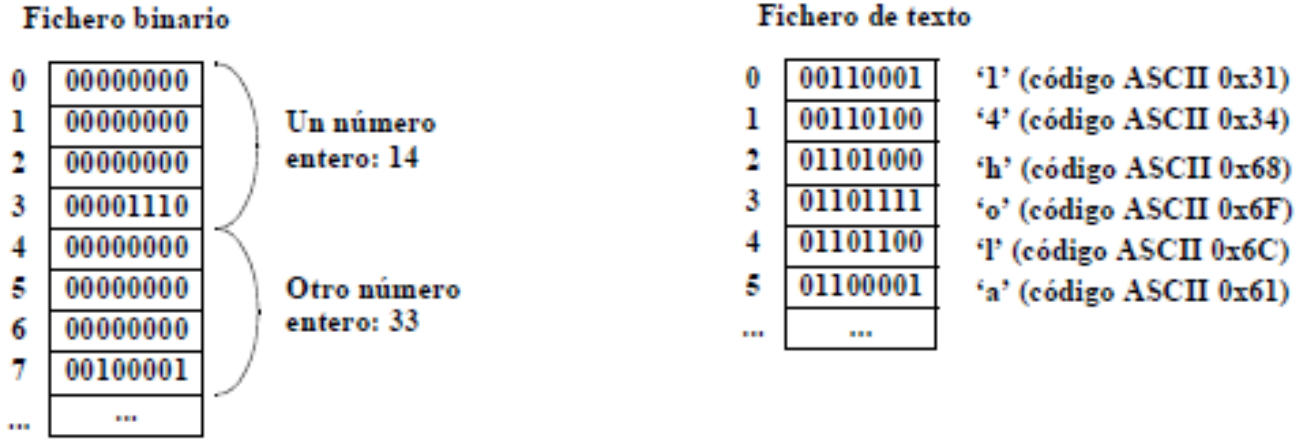
  Clases: **Reader** y **Writer**

  Manejan flujo de caracteres Unicode UTF-8

CONTROL DE FLUJO Y CLASES RELATIVAS A FLUJOS

Clases Relativas a Flujos

- **Flujos de bytes** (byte **streams**) **de 8 bits**. Permite leer bytes y están pensados para ser leídos por un programa
- **flujos de caracteres o de texto** (**characterstreams**) **de 16 bits**. Pensados para ser leídos y/o creados por una persona



Para entender los contenidos de un fichero es necesario conocer que tipos de dato contiene para saber lo que ocupa cada dato y utilizar el método apropiado. Si tenemos un int almacenado en un fichero y lo leemos con el método readShort(), estamos leyendo un menor número de bytes de los que realmente necesita un int, y esto nos dará un dato erróneo y arrastraremos el desfase en bytes a lo largo del resto de ficheros.

CONTROL DE FLUJO Y CLASES RELATIVAS A FLUJOS

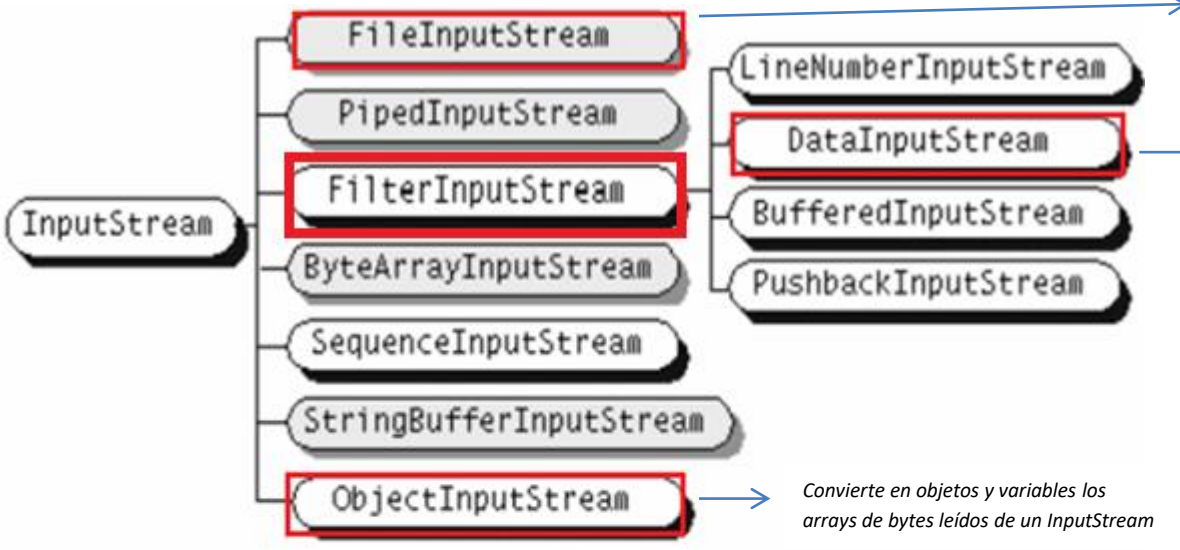
Clases Relativas a Flujos

Las clases del paquete java.io se pueden ver en las diapositivas siguientes. Destacamos las clases relativas a flujos:

- **BufferedInputStream**: permite leer datos a través de un flujo con un buffer intermedio.
- **BufferedOutputStream**: implementa los métodos para escribir en un flujo a través de un buffer.
- **FileInputStream**: permite leer bytes de un fichero.
- **FileOutputStream**: permite escribir bytes en un fichero o descriptor.
- **StreamTokenizer**: esta clase recibe un flujo de entrada, lo analiza (parse) y divide en diversos pedazos (tokens), permitiendo leer uno en cada momento.
- **String Reader**: es un flujo de caracteres cuya fuente es una cadena de caracteres o string.
- **String Writer**: es un flujo de caracteres cuya salida es un buffer de cadena de caracteres, que puede utilizarse para construir un string.
- ….

CONTROL DE FLUJO Y CLASES RELATIVAS A FLUJOS

Clases Flujo de bytes

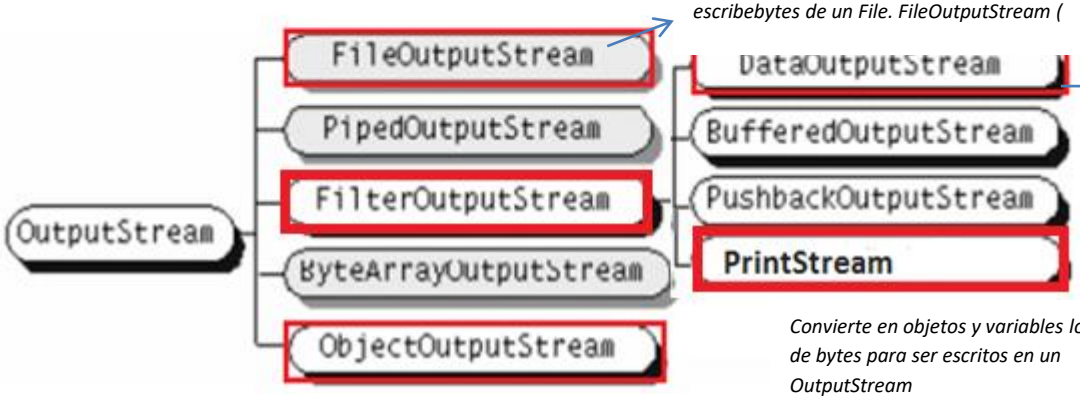


Lee bytes de un File. FileInputStream (Ruta File fichero)

Permite leer datos primitivos en el fichero usando un Obj. InputStream. Metodos readByte(), readLine(), readInt(),….

Convierte en objetos y variables los arrays de bytes leídos de un InputStream

```
FileInputputStrean fis = new FileInputputStream("/ficheros/personas.dat");
ObjectInputputStream  entrada = new ObjectInputputStream(fis);
….
Persona p =  (Persona) entrada.readObject();
```



escribebytes de un File. FileOutputStream (Ruta fichero)

Permite escribir datos primitivos en el fichero usando un Obj. OutputStream.
DataOutputStream x = new DataOutputStream(obj OutputStream)
donde obj podría ser un FileOutputStream o cualquier OutputStream

Convierte en objetos y variables los arrays de bytes para ser escritos en un OutputStream

```
FileOutputStrean fos = new FileOutputStream("/ficheros/personas.dat");
ObjectOutputStream  salida = new ObjectOutputStream(fos);
….
salida.writeObject
```

Para añadir registros seria neceario crear una nueva clase y sobrescribir el método writeStreamHeader sin nada o con reset, y abrir el fichero con esta clase en vez de FileOutputStream,evitando de esta forma volver a escribir la cabecera en el fichero.

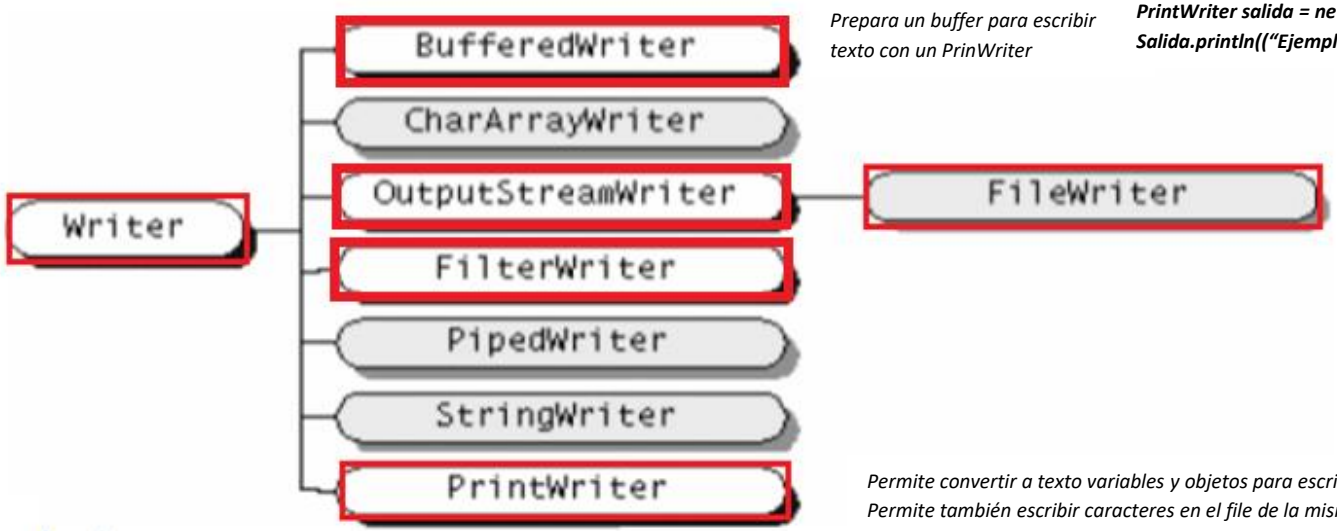
CONTROL DE FLUJO Y CLASES RELATIVAS A FLUJOS

Clases Flujo de caracteres

Lee texto línea a línea de un obj. Reader

BufferReader x = new BufferReader (new FileReader(ruta))
Métodos readLine() , read()

Lee texto de un fichero FileReader (nom_file).
Métodos read() pero lo normal es hacerlo con el BufferReader



Prepara un buffer para escribir texto con un PrinWriter

PrintWriter salida = new PrintWriter(new BufferedWriter (new FileWriter("fich.txt")));
Salida.println(("Ejemplo de escritura de datos");

Escritura de texto en un fichero.
Método write() pero lo normal es usarlo con PrintWriter

Permite convertir a texto variables y objetos para escribirlos en un Writer.
Permite también escribir caracteres en el file de la misma forma que en pantalla

FLUJOS

Utilización de los flujos

Las acciones que se suelen hacer con los flujos de datos son

Lectura

1. Abrir un flujo a una fuente de datos (creación del objeto stream)
   - Teclado
   - Fichero
   - Socket remoto
2. Mientras existan datos disponibles
   - Leer datos
3. Cerrar el flujo (método close)

Escritura

1. Abrir un flujo a una fuente de datos (creación del objeto stream)
   - Pantalla
   - Fichero
   - Socket local
2. Mientras existan datos disponibles
   - Escribir datos
3. Cerrar el flujo (método close)

***Nota**: para los flujos estándar ya se encarga el sistema de abrirlos y cerrarlos.* Un fallo en cualquier punto produce la excepción IOException

FLUJOS

Flujos Predefinidos. Entrada y Salida estándar

En un programa se crean automáticamente 3 tipos de flujo:

- El fichero de entrada estándar (**stdin**) es típicamente el teclado.
- El fichero de salida estándar (**stdout**) es típicamente la pantalla (o la ventana del terminal).
- El fichero de salida de error estándar (**stderr**) también se dirige normalmente a la pantalla, pero se implementa como otro fichero de forma que se pueda distinguir entre la salida normal y (si es necesario) los mensajes de error.

FLUJOS

Flujos Predefinidos. Entrada y Salida estándar

Java tiene acceso a la entrada/salida estándar a través de la clase System. En concreto, los tres ficheros que se implementan son:

- **Stdin**. **System.in** Es un objeto de tipo InputStream, y está definido en la clase System como flujo de entrada estándar. Por defecto es el teclado, pero puede redirigirse para cada host o cada usuario, de forma que se corresponda con cualquier otro dispositivo de entrada.

- **Stdout**. **System.out** implementa stdout como una instancia/objeto de la clase PrintStream. Se pueden utilizar los métodos print() y println() con cualquier tipo básico Java como argumento.

- **Stderr**. Es un objeto de tipo PrintStream. Es un flujo de salida definido en la clase System y representa la salida de error estándar. Por defecto, es el monitor, aunque es posible redireccionarlo a otro dispositivo de salida.

FLUJOS

Flujos Predefinidos. Entrada y Salida estándar

Para la entrada, se usa el método read para leer de la entrada estándar:

- **int System.in.read() ;** -> Lee el siguiente byte (char) de la entrada estándar.

Ej. leer un conjunto de caracteres hasta que se pulse la tecla RETORNO :

```
StringBuffer str=new StringBuffer();
char c;
 try{
            while ((c=(char)System.in.read())!='\n'){  //lee carácter por teclado
                  str.append(c);                       // lo añade al stringBuffer
            }
}catch(IOException ex){};
// Escribir la cadena que se ha ido tecleando
System.out.println("Cadena introducida:   " + str);
```

- **int System.in.read(byte[] b);** -> Leer un conjunto de bytes de la entrada estándar y lo almacena en el vector b.

FLUJOS

Flujos Predefinidos. Entrada y Salida estándar

Para la salida, se usa el método print para escribir en la salida estándar:

- **System.out.print(String);** -> Muestra el texto en la consola.

  Ej. System.out.print ("El precio es de " + precio + " euros");

- **System.out.println(String);** -> Muestra el texto en la consola y seguidamente efectúa un salto de línea. Normalmente, para **leer valores numéricos,** lo que se hace es tomar el valor de la entrada estándar en forma de cadena y entonces usar métodos que permiten transformar el texto a números (int, float, double, etc.) según se requiera.

  Ej. System.out.println ("El precio es de " + precio + " euros");

FLUJOS

Flujos Predefinidos. Entrada y Salida estándar

Funciones de conversión	
Método	Funcionamiento
byte Byte.parseByte(String)	Convierte una cadena en un número entero de un byte
short Short.parseShort(String)	Convierte una cadena en un número entero corto
int Integer.parseInt(String)	Convierte una cadena en un número entero
long Long.parseLong(String)	Convierte una cadena en un número entero largo
float Float.parseInt(String)	Convierte una cadena en un número real simple
double Double.parseInt(String)	Convierte una cadena en un número real doble
boolean Boolean.parseBoolean(String)	Convierte una cadena en un valor lógico

FLUJOS

Flujos Predefinidos. Entrada y Salida estándar. Ejemplo

Leer de teclado mientras no pulsemos el INTRO. Programa completo

```
public class leeEstandar {
public static void main(String[]   args) {
        // Cadena donde iremos almacenando los caracteres que se escriban
        StringBuilder str = new StringBuilder(); char c;
        // Por si ocurre una excepción ponemos el bloque try-cath
        try{
                        // Mientras la entrada de terclado no sea Intro
                        while ((c=(char)System.in.read())!='\n'){
                                        / / Añadir el character leído a la cadena str
                                        str.append(c);
        }
        }catch(IOException ex){System.out.println(ex.getMessage()); }
        // Escribir la cadena que se ha ido tecleando
        System.out.println("Cadena introducida:   " + str);
        }
}
```

FILE STREAM

CLASE File

Los **File Stream** son los streams utilizados para lectura y escritura de (particularmente) archivos. Es una categoría que agrupa tanto a los streams orientados a carácter como a los streams orientados a byte.

Clase File no trabaja con un flujo de Bytes, sino directamente con el fichero y el sistema de ficheros (directorios). Con esta clase no accedemos a los datos del fichero sino que permite obtener y manipular información asociada al mismo (permisos, fecha, si es fichero o directorio, path…..)

La clase File modela tanto archivos como directorios. Para su instanciación se deberá importar **java.io.File.**

Ejemplo de archivo:

```
File archivo = new File ("un_path/un_fichero.txt");
File archivo = new File("C://Usuarios//XXXX//Mis Documentos//MiArchivo.txt”);
File archivo = new File (“MiArchivo.txt/MiAnimacion.gif”); // Carpeta Raiz
```

Ejemplo de directorio:

```
File directorio = new File(“Carpeta Nueva”);
directorio.mkdir();//Se crea el directorio en la carpeta raíz
```

FILE STREAM

CLASE File. Constructores

Los **constructores** de File permiten inicializar el objeto con el nombre de un archivo y la ruta donde se encuentra. También, inicializar el objeto con otro objeto File como ruta y el nombre del archivo.

```
public File(String nombreCompleto)
public File(String ruta, String nombre)
public File(File ruta, String nombre)
```

Por ejemplo:

```
File miFichero = new File("C:\LIBRO\Almacen.dat");
File otro = new File("COCINA", "Enseres.dat");
```

CONSEJO:

Es una buena práctica crear objetos File con el archivo que se va a procesar, para pasar el objeto al constructor del flujo en vez de pasar directamente el nombre del archivo. De esta forma se pueden hacer controles previos sobre el archivo (si existe el fichero o la ruta, …)

FILE STREAM

CLASE File. Métodos

Con los métodos de la clase File se obtiene información relativa al archivo o ruta con que se ha inicializado el objeto.

Los métodos mas útiles para conocer los atributos de un archivo o un directorio:

```
public boolean exists()          true si existe el archivo(o el directorio)
public boolean canWrite()        true si se puede escribir en el archivo
public boolean canRead()         true si es de sólo lectura
public boolean isFile()          true si es un archivo
public boolean isDirectory()     true si es un directorio
public long length()             número de bytes que ocupa el archivo
public String getName()
public String getPath()
public String getAbsolutePath()  cadena con la ruta completa
public boolean delete()
public boolean renameTo(File nuevo)  ren
public String[] list()           devuelve un array de cadenas, cada una
contiene un elemento(archivo o directorio) del directorio  con el
que se ha inicializado el objeto FILE
```

FILE STREAM

CLASE File. Métodos

Con los métodos de la clase File se obtiene información relativa al archivo o ruta con que se ha inicializado el objeto.

Renombrar el archivo, con el método **renameTo().** El objeto File dejará de referirse al archivo renombrado, ya que el String con el nombre del archivo en el objeto File no cambia.

Borrar el archivo, con el método **delete()**. También, con **deleteOnExit()** se borra cuando finaliza la ejecución de la máquina virtual Java.

Crear un nuevo fichero con un nombre único. El método estático **createTempFile()** crea un fichero temporal y devuelve un objeto File que apunta a él. Es útil para crear archivos temporales, que luego se borran, asegurándonos tener un nombre de archivo no repetido.

Establecer la fecha y la hora de modificación del archivo con **setLastModified().** Por ejemplo, se podría hacer: new File("prueba.txt").setLastModified(new Date().getTime()); para establecerle la fecha actual al fichero que se le pasa como parámetro, en este caso *prueba.txt*.

Crear un directorio con el método **mkdir().** También existe **mkdirs(),** que crea los directorios superiores si no existen.

Listar el contenido de un directorio. Los métodos **list()** y **listFiles()** listan el contenido de un directorio list() devuelve un vector de String con los nombres de los archivos, listFiles() devuelve un vector de objetos File.

Listar los nombres de archivo de la raíz del sistema de archivos, mediante el método estático **listRoots().**

FILE STREAM

CLASE File. Ejemplo

```
public class EjemplosClaseFile01 {
   //comprobar si existe un directorio
   public static void main(String[] args) {
      File dir;
      String mensaje;

      dir = new File("c:/prueba");

      if (dir.exists())
         mensaje ="SI existe";
      else
          mensaje ="NO existe";

      System.out.println("El directorio "+dir.getPath()+" "+mensaje);
   }
}
```

FILE STREAM

CLASE File. Ejemplo

```
public class EjemplosClaseFile02 {
   //comprobar si existe un fichero
   public static void main(String[] args) {
      File path,fichero;
      String mensaje;

      path = new File("c:/prueba");
      fichero = new File(path,"fichero.txt");

      //tambien podríamos  fichero = new File("c:/prueba/fichero.txt");

      if (fichero.exists())
         mensaje ="SI existe";
      else
          mensaje ="NO existe";

      System.out.println("El directorio "+fichero.getPath()+" "+mensaje);
   }
}
```

FLUJOS BASADOS EN BYTES

Flujos Basados en BYTES

Flujos Basados en BYTES (1 byte)

- Se utiliza para el manejo de entradas y salidas de bytes. ***Orientado a la lectura y escritura de datos binarios solo legibles por una máquina o un programa.*** Se utiliza para ficheros gif, jpg, png, xls,…

Flujos de caracteres (characters streams) (2 bytes).

- Permite leer caracteres. Se usan para manipular datos legibles por humanos (p.e. fichero de texto, csv,…).

  Manejan flujo de caracteres Unicode (2 bytes)

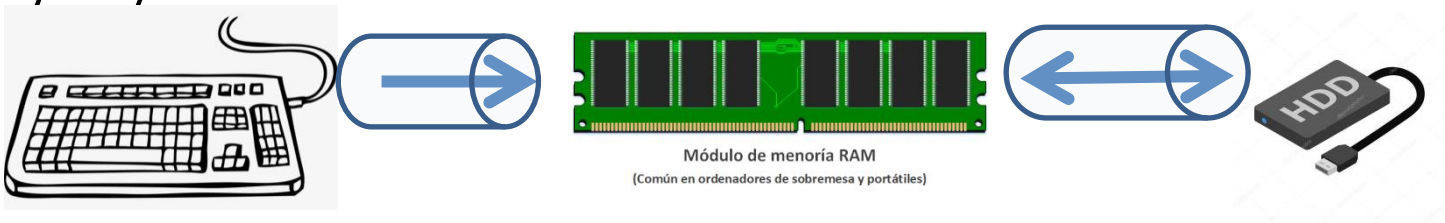
FLUJOS BASADOS EN BYTES

Flujos Basados en BYTES

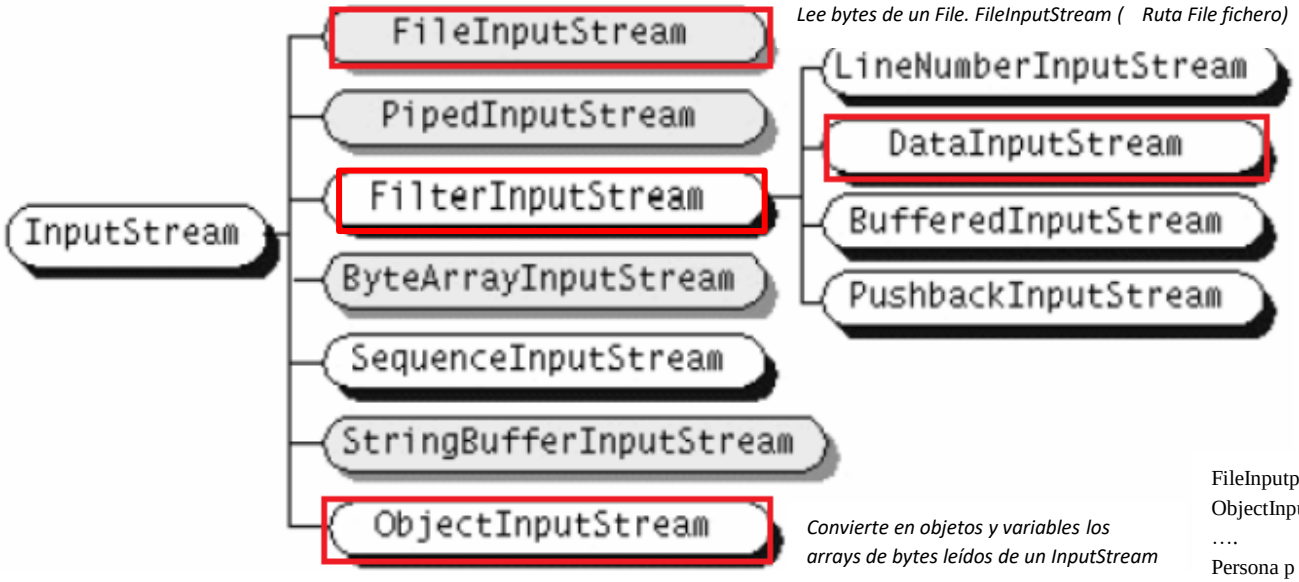
- Se utiliza para el manejo de entradas y salidas de bytes. ***Orientado a la lectura y escritura de datos binarios solo legibles por una máquina o un programa.*** Se utiliza para ficheros gif, jpg, png, xls,…

- tiene 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 E/S que se pueden utilizar.

- Tanto **InputStrean** como **OutputStream** crean un enlace entre el flujo de bytes y el fichero



FLUJOS BASADOS EN BYTES

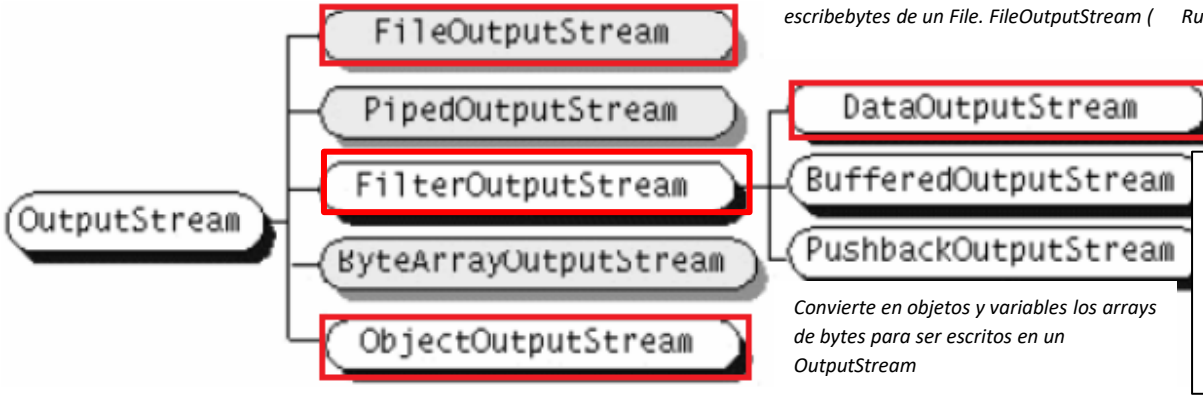


Lee bytes de un File. FileInputStream (Ruta File fichero)

Permite leer datos primitivos en el fichero usando un Obj. InputStream. Metodos readByte(), readLine(), readInt(),….

Convierte en objetos y variables los arrays de bytes leídos de un InputStream

```
FileInputputStrean fis = new FileInputputStream("/ficheros/personas.dat");
ObjectInputputStream  entrada = new ObjectInputputStream(fis);
….
Persona p =  (Persona) entrada.readObject();
```



escribebytes de un File. FileOutputStream (Ruta fichero)

Permite escribir datos primitivos en el fichero usando un Obj. OutputStream.
DataOutputStream x = new DataOutputStream(obj OutputStream)
donde obj podría ser un FileOutputStream o cualquier OutputStream

Convierte en objetos y variables los arrays de bytes para ser escritos en un OutputStream

```
FileOutputStrean fos = new FileOutputStream("/ficheros/personas.dat");
ObjectOutputStream  salida = new ObjectOutputStream(fos);
….
salida.writeObject
```

Para añadir registros seria neceario crear una nueva clase y sobrescribir el método writeStreamHeader sin nada o con reset, y abrir el fichero con esta clase en vez de FileOutputStream,evitando de esta forma volver a escribir la cabecera en el fichero.

FLUJOS BASADOS EN BYTES

Flujos Basados en BYTES

OutputStream: escritura de ficheros binarios

- FileOutputStream (iniciador): escribe bytes en un fichero
- ObjectOutputStream (filtro): convierte objetos y variables en arrays de bytes que pueden ser escritos en un OutputStream

InputStream: lectura de ficheros binarios

- FileInputStream (iniciador): lee bytes de un fichero
- ObjectInputStream (filtro): convierte los arrays de bytes leídos de un InputStream en objetos y variables

FLUJOS BASADOS EN BYTES

Flujos Basados en BYTES FileInputStream/FileOutputStream

Tanto InputStream como OutputStream (clases abstractas) tiene varias subclases concretas, que controlan las diferencias entre los distintos dispositivos de E/S que se pueden utilizar.

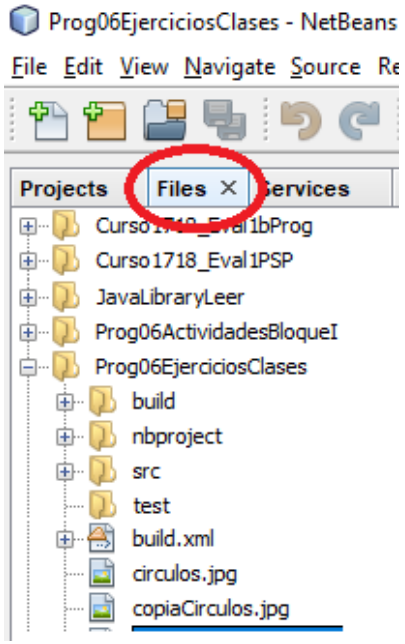
FileInputStream/FileOutputStream

Sus Constructores:

```
class   FileInputStream extends InputStream {
        FileInputStream  (String fichero)   throws FileNotFoundException;  // fichero = nomFichero
        FileInputStream  (File fichero)   throws FileNotFoundException;  //fichero objeto File
        …..
}

class   FileOutputStream extends OutputStream {
        FileOutputStream (String fichero) throws IOException;   // fichero = nomFichero
        FileOutputStream  (File fichero)   throws IOException;  //Fichero objeto File
        FileOutputStream (String fichero, boolean sw) throws IOException;  //si sw =true añade al final
        …..
}
```

FLUJOS BASADOS EN BYTES

Flujos Basados en BYTES FileInputStream/FileOutputStream



FileInputStream (String fichero); // fichero = nomFichero

FileInputStream (File fichero); //fichero objeto File

…..

FileOutputStream (String fichero); // fichero = nomFichero

FileOutputStream (File fichero); //Fichero objeto File

FileOutputStream (String fichero, boolean sw); //si sw =true añade al final

…..

IMPORTANTE: Si no especificamos el path del archivo, por defecto:

- Si vamos a leer el fichero/archivo deberá de estar almacenado en la misma carpeta donde está el proyecto
- Si vamos a grabar información el fichero se creará y grabará en la misma carpeta donde esta el proyecto.

En netbeans no se verá en la pestaña de PROJECTS, tendremos que verlo en la pestañas de FILES

FLUJOS BASADOS EN BYTES

Flujos Basados en BYTES FileInputStream/FileOutputStream

Métodos más utilizados:

Métodos FileInputStream:

- **int read() throws IOException; //devuelve -1 si ha llegado al EOF***
- **int read(byte[] s) throws IOException;**

Métodos FileOutputStream:

- **void write(byte b) throws IOException;**
- **void write(int b) throws IOException;**
- **void write(byte[] s) throws IOException;**

Ambas clases disponen del método **close()** para cerrar el flujo/stream

***EOF → End Of File**

FLUJOS BASADOS EN BYTES

Métodos mas usuales de InputStream y OutputStream

InputStream	OutputStream
public **int read()** throws IOException Lee un byte de dato desde el flujo de entrada. Returns: el siguiente byte de dato o -1 si no es final de fichero EOF (End Of File) Throws: IOException – si ocurre error E/S	public void **write(int b)** throws IOException Escribe el byte en el flujo de salida Parameters:b – el byte a ser escrito. Throws: IOException - if an I/O error occurs.
public **int read(byte[] b)** throws IOException Lee hasta b.length bytes de datos desde el flujo de entrada dentro de un array de bytes **Overrides:** read en la clase InputStream Parameters: b – es el buffer dentro del cual el dato es leído Returns: el núm total de bytes leídos dentro del buffer o -1 si no hay mas datos por EOF Throws: IOException - if an I/O error occurs.	public void **write(byte[] b)** throws IOException Escribe b.length bytes desde el byte especificado del array hasta el flujo de salida **Overrides:** write en la clase OutputStream Parameters:b – el dato a escribir Throws: IOException - if an I/O error occurs.

FLUJOS BASADOS EN BYTES

Detección del EOF (End Of File) en un fichero de bytes

Cuando leemos un fichero de bytes con read(), si llegamos al final de fichero **devuelve -1**

Problema: ¿Que pasa si el fichero de bytes es de números enteros y entre ellos está el -1 como elemento de dicho fichero?

5 -30 7 9 -1 4 2 8 10 -22

Respuesta: → en este caso al leer el -1 el consideraría que se ha llegado al EOF, cuando realmente el -1 es un valor del fichero y no el EOF.

¿Cómo solucionaríamos el problema?

FLUJOS BASADOS EN BYTES

Detección del EOF (End Of File) en un fichero de bytes

Solución : en este caso la forma de leer el final de fichero sería forzando a que saltara una excepción EOFException

```
.....
try{
        InputStream  fentrada = new FileInputStream(origen);
        while  (true) {// Leer el flujo de bytes del fichero origen
                        int n = fentrada.read();
                       //……. Instrucciones para Procesar la información leída
       }
  } catch(EOFException ex) {
        System.out.println("-------final de fichero ---------") ;
   catch(IOException ex) {
        System.out.println(ex.getMessage()) ;
   }  finally{
        fentrada .close();
   }
```

Podríamos eliminar el catch del EOFException y poner el mensaje en el catch del IOException.

FLUJOS BASADOS EN BYTES

Flujos Basados en BYTES

OutputStream y de **InputStream** y todas sus subclases, *reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de entrada o salida*. Tanto OutputStream como InputStream son flujos iniciadores, es decir son los que directamente vuelcan o recogen los datos del dispositivo

En el ejemplo siguiente será:

String origen para la entrada y String destino para el fichero salida

```
void copia (String origen, String destino) throws IOException {
    try{
            // Obtener los nombres de los ficheros de origen y destino
            // y abrir la conexión a los ficheros.
            (*)InputStream fentrada = new FileInputStream(origen);
            (*)OutputStream fsalida = new FileOutputStream(destino);
            ….
    }
```


(*) Al haber herencia en las clases puedo instanciar un objeto declarado de una clase superior usando una clase hija (ver hoja de ficheros)

FLUJOS BASADOS EN BYTES

Por ejemplo, podemos copiar el contenido de un fichero en otro optimizando la escritura:

```
void  copia  (String origen,  String destino)   throws IOException {
 try{
        InputStream  fentrada = new FileInputStream(origen);
        OutputStream  fsalida = new FileOutputStream(destino);
        // Crear una variable para leer el flujo de bytes del origen y optimizar la lectura y escritura
        byte[]  buffer= new byte[256];   //estruct. datos temporal para almacenar lo leido del origen
        while  (true) {
                        // Leer el flujo de bytes del fichero origen y lo mete en buffer
                         int n = fentrada.read(buffer);
                        if  (n < 0) break;   // Si no queda nada por leer,  salir del while
                        // Escribir el flujo de bytes leídos  y almacenados en buffer al fichero destino
                        fsalida.write(buffer,   0, n); }
                        // Cerrar los ficheros
                        fentrada.close();
                        fsalida.close();
}catch(IOException ex) { System.out.println(ex.getMessage()) ;  }
```

FLUJOS BASADOS EN BYTES

Por ejemplo, podemos copiar de un fichero a otro fichero, byte a byte:

```
public class CopiarFicheroBytes01 {
   public static void main(String[] args) throws IOException
   {
        File archivoEntrada = new File("logo.gif");
        File archivoSalida = new File("destino.gif");

      // Instancia un  FileInputStream  y un  FileOutputStream  que se encargaran de leer y escribir archivos respectivamente
      FileInputStream lector = new FileInputStream(archivoEntrada);
      FileOutputStream escritor = new FileOutputStream(archivoSalida);
```

FLUJOS BASADOS EN BYTES

Por ejemplo, podemos copiar de un fichero a otro fichero:

```
    // Instancia una variable que contendrá el byte a leer, se lee byte a byte
     int unByte ;

    // Informa que se está copiando el archivo
     System.out.println("\n\tEl archivo está siendo copiado....");

     // Lee el archivoEntrada y guarda la informacion en el archivoSalida hasta que hemos llegado a EOF (-1)
     while ( (unByte = lector.read()) != -1){
       escritor.write(unByte);
      }
     // Cierra los archivos
     lector.close();
     escritor.close();

     // Informa que se ha copiado el archivo
     System.out.println("\tEl archivo ha sido copiado con éxito....\n");
   }
}
```

FLUJOS BASADOS EN BYTES

CLASES FILTRO: Leer y escribir datos de tipo primitivo y/o complejos

Datos primitivos	byte, int, long, float, double,….	Cada dato primitivo ocupa varios bytes, dependiendo de los valores que sea capaz de representar
Datos complejos	objetos	Un objeto esta formado por atributos de distinto tipo y su tamaño será la suma de todos los bytes de cada tipo y lo que ocupan sus métodos

TIPOS PRIMITIVOS (sin métodos; no son objetos; no necesitan una invocación para ser creados)

NOMBRE	TIPO	OCUPA	RANGO APROXIMADO
byte	Entero	1 byte	-128 a 127
short	Entero	2 bytes	-32768 a 32767
int	Entero	4 bytes	$2*10^9$
long	Entero	8 bytes	Muy grande
float	Decimal simple	4 bytes	Muy grande
double	Decimal doble	8 bytes	Muy grande
char	Carácter simple	2 bytes	---
boolean	Valor true o false	1 byte	---

FLUJOS BASADOS EN BYTES

CLASES FILTRO: FileInputStream y FileOutputStream

Datos primitivos	byte, int, long, float, double,….	Cada dato primitivo ocupa varios bytes, dependiendo de los valores que sea capaz de representar
Datos complejos	objetos	Un objeto esta formado por atributos de distinto tipo y su tamaño será la suma de todos los bytes de cada tipo y lo que ocupan sus métodos

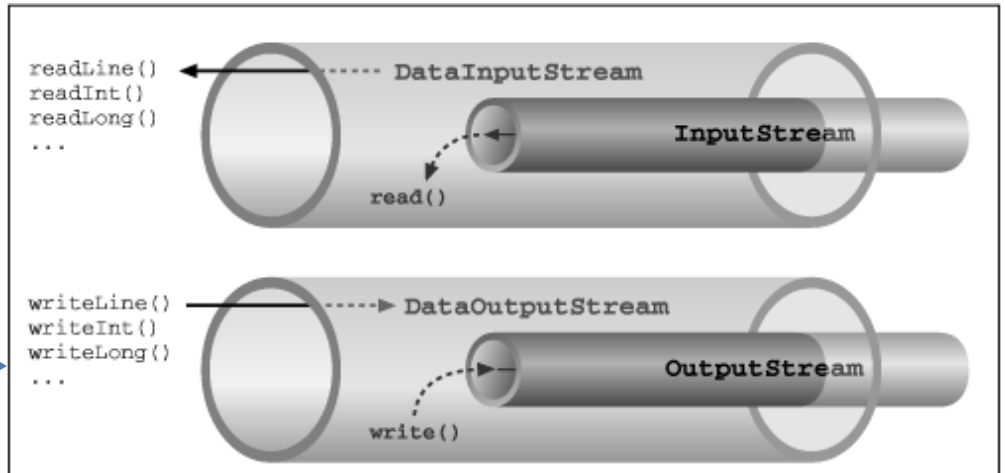
Son elementos que ponemos entre el programa y dispositivo de E/S para:

- En el programa los datos de tipo primitivo(secuencia de bytes) se convierten en bytes para escribir en el fichero o dispositivo de S
- y viceversa leer del dispositivo de E, bytes y formar secuencia de bytes para crear datos de tipo primitivo

Obtenemos dato tipo primitivo u objeto

Programa

transformaremos dato tipo primitivo u objeto



Leemos bytes

dispositivo E/S

Escribimos bytes

FLUJOS BASADOS EN BYTES

CLASES FILTRO: FileInputStream y FileOutputStream

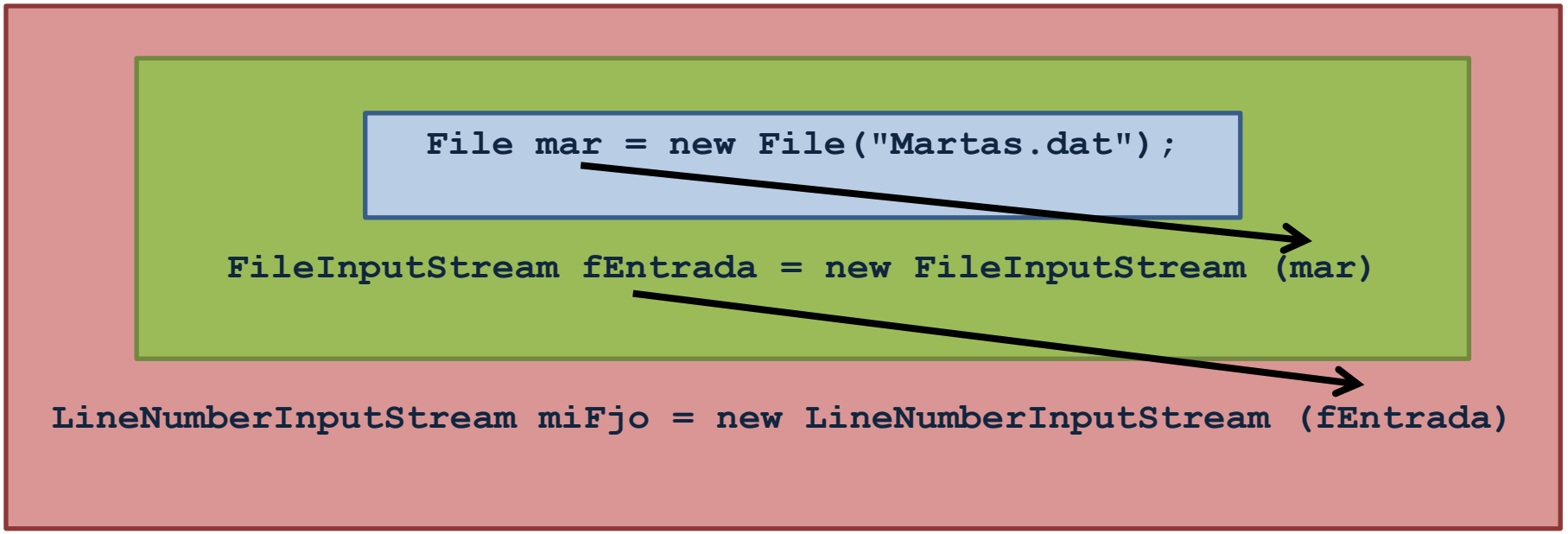
- Los flujos filtro leen secuencias de bytes, ***pero organizan internamente estas secuencias para formar datos de los tipos primitivos (int, long, double …)***.
- Los objetos stream filtro leen de un flujo que previamente ha tenido que ser escrito por otro objeto stream filtro de salida. ***Es decir, para poder leer datos primitivos usando filtros, estos tuvieron que ser escritos usando filtros.***
- Las clases para manejar filtros de datos de tipo primitivo son **FilterInputStream** y **FilterOutputStream** (derivan de InputSream y OutputStream).
- Los objeto stream filtro siempre están enlazados con secuencias de bytes. Al crear un objeto filtro se pasa como argumento un objeto stream que representa la secuencia de bytes que transformará (filtrará) el objeto creado

Por ejemplo:

```
File mar = new File("Martas.dat");
FileInputStream fEntrada = new FileInputStream (mar);
LineNumberInputStream miFjo = new LineNumberInputStream (fEntrada)
  //Mantiene un seguimiento de los números de línea en el flujo de entrada;
```

FLUJOS BASADOS EN BYTES

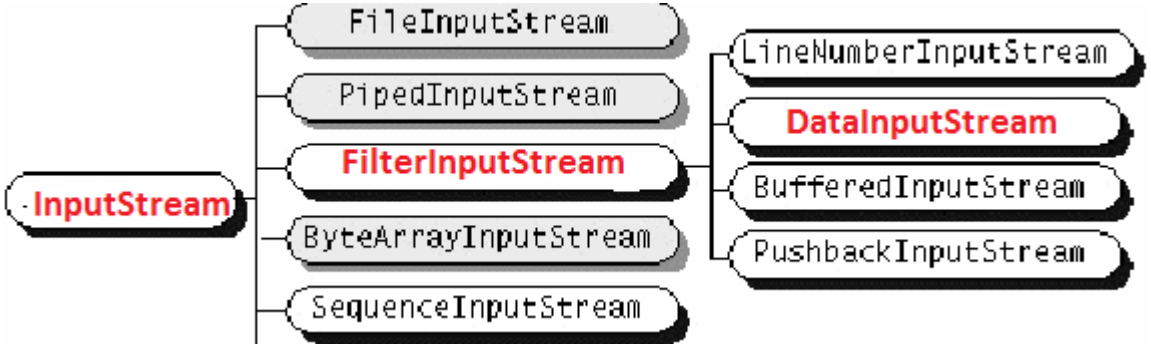
CLASES FILTRO: FileInputStream y FileOutputStream

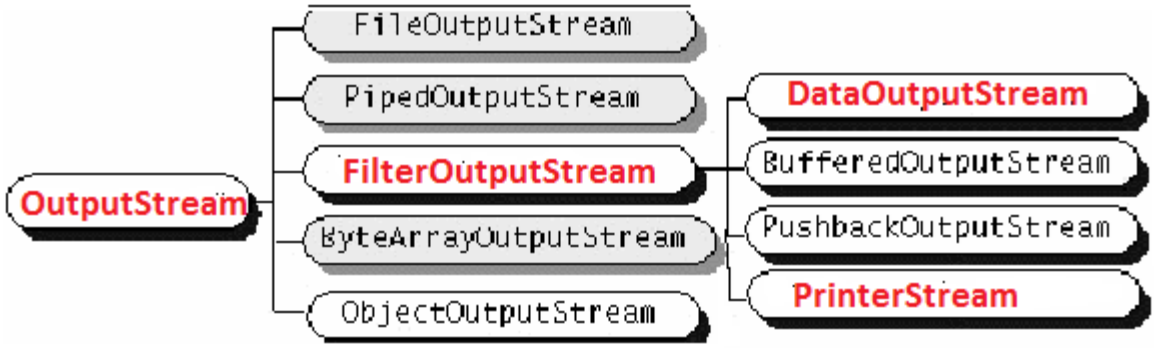


FLUJOS BASADOS EN BYTES

CLASES FILTRO: DataInputStream y DataOutputStream

- La clases **`DataInputStream`** y **`DataOutputStream`** *filtran una secuencia de bytes y organizan los bytes para formar datos de tipo primitivo*. De tal forma que se puedan escribir o leer directamente datos de tipo: `char, byte, short, int, long, float, double, boolean, String .`





FLUJOS BASADOS EN BYTES

CLASES FILTRO: DataInputStream y DataOutputStream

El objetivo de las clases filtro es facilitar la manipulación (leer o grabar) de los datos de una forma más sencilla.

Sabemos que los Streams mueve bytes, entonces si vamos a grabar a un fichero o leer de un fichero número enteros tenemos que ser conscientes de los bytes que ocupa un int (4 bytes) , por lo tanto tendríamos que controlar el leer o grabar 4 bytes y interpretar esos 4 bytes como un valor que sería el entero. Es decir

byte byte byte byte byte byte **byte byte byte byte** …… **byte byte byte byte**

int int

Pero como programador debemos de controlar que leemos 4 bytes y no menos ni más.

Con los **filtros** encapsulamos la información que llega en bytes y es el filtro es que se encarga de leer/escribir el número de bytes necesarios en función al tipo de dato facilitando la labor al programador. Por ejemplo en el caso de leer enteros

Fich = new DataInputStream (FileInputStream (…)) **fich.readInt();**

byte byte byte byte byte byte byte byte | byte byte byte byte | byte byte byte byte

int

FLUJOS BASADOS EN BYTES

CLASES FILTRO: DataInputStream y DataOutputStream

Constructores:

DataInputStream(InputStream in**)** : Permite crear un objeto sobre un flujo de tipo InputStream

DataOutputStream(OutputStream **out)** : Permite crear un objeto sobre un flujo de tipo OutputStream

De esta forma podemos manejar los ficheros tanto para grabar como para leer de una forma mucho más fáciles cuando la información con la que vamos a trabajar son datos primitivos

Ejem.

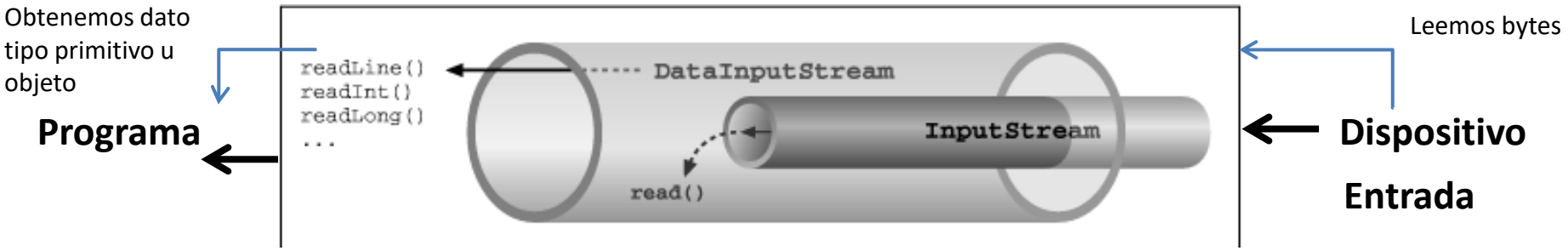
```
FileOutputStream fos = new FileOutputStream("cities.dat");
DataOutputStream dos = new DataOutputStream(fos);

FileInputStream fis = new FileInputStream("cities.dat");
DataInputStream dos = new DataInputStream(fis);
```

FLUJOS BASADOS EN BYTES

CLASES FILTRO: DataInputStream y DataOutputStream

- **DataInputStream** declara el comportamiento de los flujos de entrada, con métodos que leen cualquier tipo primitivo básico: readInt(), readDouble(), readUTF(), ... Este tipo de flujos leen bytes de otro flujo de bajo nivel (flujo de bytes); por esa razón se asocian a un flujo de tipo InputStream, generalmente un flujo FileInputStream .

```
File fich = new File("nubes.dat");
DataInputStream fent = new DataInputStream(new FileInputStream(fich));
```



En una palabra lee varios bytes del dispositivo de entrada para para formar un dato de tipo primitivo (int, long, double, float,……)

Constructor :

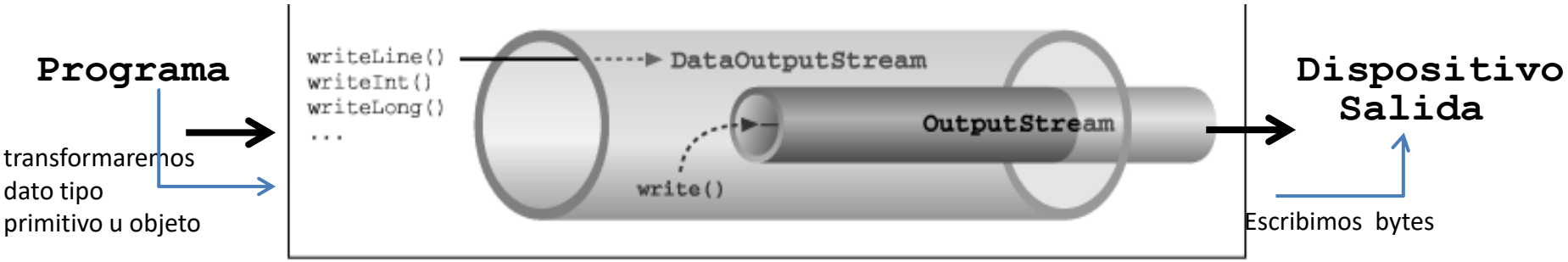
```
public DataInputStream(InputStream entrada) throws IOException
```

FLUJOS BASADOS EN BYTES

CLASES FILTRO DataInputStream y DataOutputStream

- Los flujos de salida, **DataOutputStream**, se asocian o enlazan con otro flujo de salida de bajo nivel, de bytes. Los métodos de este tipo de flujos permiten escribir cualquier valor de tipo de dato primitivo, `int double ... y String` como bytes en un fichero

```
File fich = new File("nubes.dat");
DataOutputStream fsal=new DataOutputStream(new FileOutputStream(fich));
```



- Constructor:

```
public DataOutputStream(InputStream entrada) throws IOException
```

FLUJOS BASADOS EN BYTES

CLASES FILTRO DataInputStream

Métodos DataInputStream: (lee un archivo que previamente fue escrito con un flujo de salida DataOutputStream). Cuando llega al final del fichero produce una **EOFException** que habrá que gestionar

public final boolean readBoolean() throws IOException *Devuelve el valor de tipo boolean leído*

public final byte readByte() throws IOException Devuelve *el valor de tipo byte leído*

public final short readShort() throws IOException *Devuelve el valor de tipo short leído*

public final char readChar() throws IOException *Devuelve el valor de tipo char leído*

public final int readInt() throws IOException *Devuelve el valor de tipo int leído*

public final long readLong() throws IOException *Devuelve el valor de tipo long leído*

public final float readFloat() throws IOException *Devuelve el valor de tipo float leído*

public final double readDouble() throws IOException *Devuelve el valor de tipo double leído*

public final String readUTF()throws IOException *Devuelve una cadena escrita en formato UTF.*

FLUJOS BASADOS EN BYTES

CLASES FILTRO DataInputStream y DataOutputStream

```
File fich = new File("nubes.dat");
DataInputStream fentrada
                =new DataInputStream(new FileInputStream(fich));
try{
    …
    while (true) {
        System.out.println( fentrada.readUTF());
    }
    ….
}catch(EOFException ex){ //gestión fin de fichero  EOF
    System.out.println("Fin de fichero");
    fentrada.close();
}
```

FLUJOS BASADOS EN BYTES

CLASES FILTRO DataOutputStream

Métodos DataOutputStream: para escribir en un fichero variables primitivas como bytes

public final void writeBoolean(boolean v) throws IOException *Escribe el dato de tipo boolean*

public final void writeByte(int v) throws IOException *Escribe el dato v como un byte*

public final void writeShort(int v) throws IOException *Escribe el dato v como un short*

public final void writeChar(int v) throws IOException *Escribe el dato v como un carácter*

public final void writeChars(String v) throws IOException *Escribe la secuencia de caracteres de la cadena v*

public final void writeInt(int v) throws IOException *Escribe el dato de tipo int v*

public final void writeLong(long v) throws IOException *Escribe el dato de tipo long v*

public final void writeFloat(float v) throws IOException *Escribe el dato de tipo float v*

public final void writeDouble(double v) throws IOException *Escribe el valor de tipo double v*

public final void writeUTF(String cad) throws IOException *Escribe la cadena cad en formato UTF. Escibe los caracteres de la cadena y dos bytes adicionales con longitud de la cadena.*

FLUJOS BASADOS EN BYTES

CLASES FILTRO DataInputStream y DataOutputStream

Consideraciones:

- Cuando queramos guardar datos primitivos int, long, float,…. Y Strings (caso especial no dato primitivo propiamente dicho) tendremos que utilizar un filtro DataOutputStream para transformar los valores primitivos a bytes

- Para leer un fichero con datos primitivos, tendremos que conocer la estructura del registro con la que fue escrito y leerlo según ese orden

Si se graba : un string + real +real + int + real + real + int + real + real + int

Para Leer : un string + (**real + real + int**) y esto ultimo hasta que finalice el fichero

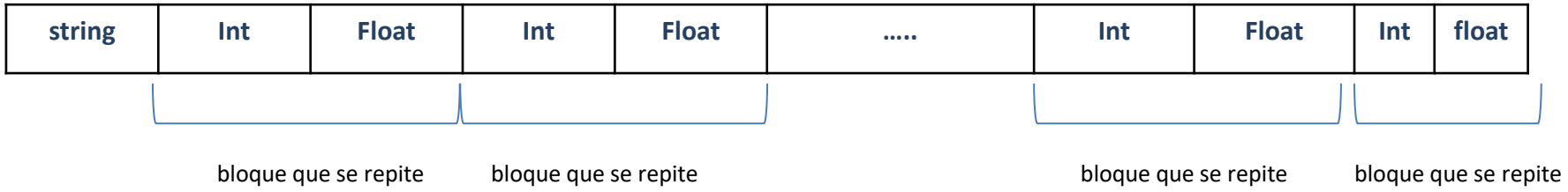
Recordar que cada dato ocupa una cantidad de bytes distintas en función de si es int o float o long. ***Una mala lectura de un dato de la secuencia puede acarrear una información errónea y arrastrar dicho error en la lectura de los siguiente datos, leyendo bytes de un dato que no son suyos.***

FLUJOS BASADOS EN BYTES

CLASES FILTRO Ejemplo

Hacer un programa que permita escribir un archivo que almacene una mensaje cualquiera y a continuación guarde 10 bloques de números formados por un entero entre 1 y 6 y un float entre 0 y 1

En el fichero la información se almacenará de la siguiente forma:



Tendremos que :

- Grabar primero un String
- Mediante un bucle escribir un bloque formado por :
  - Escribir int
  - Escribir float

FLUJOS BASADOS EN BYTES

CLASES FILTRO Ejemplo

```
public class FicheroDatosPrimitivos01 {
   public static void main(String[] args) throws FileNotFoundException {
    //programa que lee de un fichero
    //un mensaje =" Ejemplo mensaje" en un fichero
    //y a continuación 20 números (1 numero real y número entero alternativamente )
      int numInt;
      float numFloat;
      String mensaje = "Ejemplo Mensaje";
      File ficheroSalida = new File("FicheroDatosPrimitivos.dat");
      DataOutputStream fsal = new DataOutputStream(new FileOutputStream(ficheroSalida));

      try {
         fsal.writeUTF(mensaje); //escribimos el string
         System.out.println(‘’Ya se ha grabado el mensaje en el fichero ’’+ mensaje);
```

FLUJOS BASADOS EN BYTES

CLASES FILTRO Ejemplo

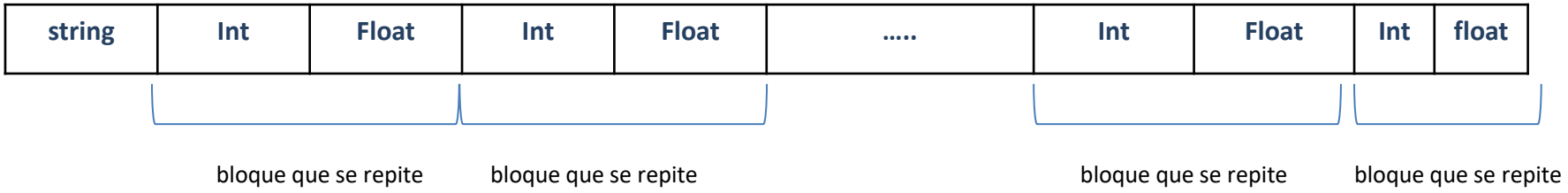
```
            //escribimos 10 números reales aleatorios
            for (int i = 0; i < 10; i++) {
               //calculo los números
               numInt = (int) (Math.random() * 6 + 1);
               numFloat = (float) Math.random();
               System.out.println(numInt+ " "+ numFloat);
               //escribimos en el fichero
               fsal.writeInt(numInt);
               fsal.writeFloat(numFloat);
            }
         } catch (IOException ex) {            System.out.println("Error en la escritura del fichero");}
          finally {
            try {          fsal.close();          }
            catch (IOException ex) {
               Logger.getLogger(FicheroDatosPrimitivos01.class.getName()).log(Level.SEVERE, null, ex);
            }
         }
      }
   }
```

FLUJOS BASADOS EN BYTES

CLASES FILTRO Ejemplo

Hacer un programa que permita Leer el archivo anterior, sabiendo que se ha almacenado un mensaje y a continuación bloques de números formados por un entero un float, entero y float,…..

En el fichero la información se almacenará de la siguiente forma:



Tendremos que :

- Leer primero un String
- Mediante un bucle leer un bloque formado por :
  - Leer int
  - Lerr float

FLUJOS BASADOS EN BYTES

CLASES FILTRO Ejemplo

```
public class FicheroDatosPrimitivos02 {
   public static void main(String[] args) throws FileNotFoundException {
      //programa que graba un mensaje =" Ejemplo mensaje" en un fichero
      //y a continuación escribe 20 números: 1 numero real entre 0 y
      // y otro  número entero entre  1 y 6 ,así alternativamente
      int numInt;
      float numFloat;
      String mensaje;
      boolean eof = false;

      File ficheroEntrada = new File("FicheroDatosPrimitivos.dat");
      DataInputStream fin = new DataInputStream(new FileInputStream(ficheroEntrada));

      try {
         mensaje = fin.readUTF(); //leemos el string
         System.out.println("mensaje: " + mensaje);
```

FLUJOS BASADOS EN BYTES

CLASES FILTRO Ejemplo

```
            //leemos  los 10 bloques de números real y entero
            while (!eof) {  //producirá una excepción cuando llegue al final del fichero
               //leemos en el fichero numero entero y numero real en cada iteracion
               numInt = fin.readInt();
               numFloat = fin.readFloat();
               System.out.println("Número entero: " + numInt + " Numero real: " + numFloat);
            }
         } catch (EOFException ex) {       System.out.println("Se ha llegado al final del fichero"); }
          catch (IOException ex) {         System.out.println("Error en la lectura del fichero"); }
          finally {
            try {
               fin.close();
            } catch (IOException ex) {            System.out.println(ex.toString()); }
         }
      }
   }
```

FLUJOS BASADOS EN BYTES

CLASES FILTRO PrintStream

PrintStream:

- Deriva de la clase **FilterOutputStream**, y añade una nueva funcionalidad a los flujos de salida permitiendo escribir/grabar representaciones de distintos tipos de valores (int, long cadenas,…)
- Todos los caracteres escritos por un PrintStream son *convertidos a bytes utilizando la codificación de caracteres por defecto*.
- Los flujos/Stream tipo **PrintStream** son de salida que se asocian a otro flujo de bajo nivel de bytes, que a su vez se crea asociado a un archivo externo.
- El destino puede ser tanto la pantalla como un fichero

```
PrintStream fps = new PrintStream (new FileOutputStream(''Fichero.dat''));
```

Constructores:

```
public PrintStream (OutputStream destino);
public PrintStream(OutputStream destino, boolean s)  //s = true, volcado automático
```

FLUJOS BASADOS EN BYTES

CLASES FILTRO PrintStream

PrintStream:

Métodos más comunes:

public void print (Object obj);

public void print (String cadena);

public void print (xxx var); → donde xxx puede ser int, double, float, char,…..

public void println (String cadena);

public void println (xxx var); → donde xxx puede ser int, double, float, char,…..

void println(Object o);

public void flush(); //vuelva el flujo actual

System.out es de tipo **PrintStream**, asociado normalmente con la pantalla, de hecho **out** y **err** son variables objeto de tipo PrintStream a los cuales le podemos aplicar sus métodos print y println

System.**out**.print(….);

System.**out**.println(……);

System.**err**.print(….);

System.**err**.println(….);

FLUJOS BASADOS EN BYTES

CLASES FILTRO PrintStream

Hacer un programa que permita volcar en pantalla un mensaje y un número usando PrintStream

```
public class EjemploPrintStream01 {
   public static void main(String[] args) {
        int c = 15;
        // create printstream object
        PrintStream ps = new PrintStream(System.out);

        // print an object and change line
       ps.println(c);
       ps.print("Nueva linea");

       // flush the stream
       ps.flush(); //vuelca el flujo actual, en este caso a la pantalla
       }
}
```

Si en vez de System.out, ponemos el nombre de un archivo el volcado del println y de print se haría sobre dicho archivo

FLUJOS BASADOS EN BYTES

CLASES FILTRO PrintStream

Hacer un programa que permita volcar en un fichero, usando printSream un carácter, un entero, un real, una fecha, una nueva línea y un string y acceder después al proyecto y comprobar si existe el fichero y es legible al abrirlo con el bloc de notas

```
public class EjemploPrintStream02 {
public static void main(String[] args) {
   File file = new File("filePrintStream.txt");   //creamos un objeto file
   FileOutputStream fileOutputStream = null;
   PrintStream printStream = null;
   try {
      fileOutputStream = new FileOutputStream(file);  //creamos un fichero de salida usando el valor de file como nombre
      printStream = new PrintStream(fileOutputStream);  //creamos un obj PrintStream donde imprimir
      printStream.print('A'); //Imprimimos char value
      printStream.print(100); //Imprimimos int value
      printStream.print(45.451); //Imprimimos double value
       printStream.print(new Date());  //Imprimimos una fecha date
```

FLUJOS BASADOS EN BYTES

CLASES FILTRO PrintStream

```
            printStream.println(); //Imprimimos newline
            printStream.println("Esto es un ejemplo de la clase PrintStream "); //Imprimimos String
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (fileOutputStream != null) {                fileOutputStream.close();                }
                if (printStream != null) {                printStream.close();                }
            } catch (Exception e) {                e.printStackTrace();                }
        }
    }
}
```

ALMACENAR OBJETOS EN FICHEROS: Serialización

Serialización.

- **Serialización**: Permite almacenar objetos en un fichero, consiste en convertir los distintos objetos de la memoria a bytes que se guardarán en un fichero.

- Se denomina **persistencia** a la posibilidad de almacenar los objetos en un fichero, de forma que estos persistan después de que la aplicación ha finalizado.

- Las clases **ObjectInputStream** y **ObjectOutputStream** están diseñadas para crear flujos de entrada y salida de objetos persistentes .

  Objeto → serializable → byte → grabar al fichero

  Fichero → leen bytes → serializable → objeto

- La clase cuyos objetos van a persistir debe de implementar el interface **Serializable** del paquete java.io.

  class Persona **implements Serializable** {...}

ALMACENAR OBJETOS EN FICHEROS: Serialización

Serialización

- La **serialización** de un objeto consiste en generar una secuencia de bytes lista para su almacenamiento o transmisión.
- la **deserialización**, se encargará de hacer el paso contrario de bytes → objetos

Para que un objeto sea serializable, ha de implementar la interfaz java.io.Serializable (que lo único que hace es marcar el objeto como serializable, **sin que tengamos que implementar ningún método**, algo que se hace siempre con los interfaces –Interfaces se verán más adelante-). Simplemente se usa para 'marcar' aquellas clases cuyas instancias pueden ser convertidas a secuencias de bytes (y posteriormente reconstruídas).

- Si una clase es serializable y tiene atributos complejos (objetos de otra clase) estas clases deberán de implementar **Serializable** también.

- Todos los tipos primitivos en java son Serializables

ALMACENAR OBJETOS EN FICHEROS: Serialización

ObjectOutputStream. (flujos de bytes). Grabar Fich. Objetos

- se utiliza para grabar objetos persistentes. El método `writeObject()` escribe cualquier objeto de una clase *serializable* en **el flujo de bytes asociado**; puede elevar excepciones del tipo `IOException` que es necesario procesar.

```
public void writeObject(Object obj) throws IOException;
```

- El constructor de la clase espera un argumento de tipo `OutputStream`, que es la clase base de los flujos de salida a nivel de bytes.

```
FileOutputStream bn = new FileOutputStream("fileObjetos.dat");
ObjectOutputStream fobj = new ObjectOutputStream(bn);
```

A continuación se puede escribir cualquier tipo de objeto en el flujo:

```
Persona p = new Persona("Juan Perez",27);
fobj.writeObject(p);
String sr = new String("Cadena de favores");
```

ALMACENAR OBJETOS EN FICHEROS: Serialización

Problemas a tener en cuenta con ObjectOutputStream.

Cuando se abre un fichero de objetos la primera vez automáticamente el ObjectOutputStream añade una cabecera y a continuación podemos grabar todos los registros que queramos hasta que finalicemos la sesión

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fichero));

oos.writeObject(objeto1); //grabamos un objeto
oos.writeObject(objeto2); //grabamos un objeto…….

Contenido del fichero							
Primera sesión de grabación de registros en el fichero							
cabecera	Persona	Persona	Persona	Persona	Persona	Persona	Persona

ALMACENAR OBJETOS EN FICHEROS: Serialización

Problemas a tener en cuenta con ObjectOutputStream.

Sin embargo cuando otro día queramos añadir más registros, si utilizamos el mismo método con el modo true (append) , se generara una nueva cabecera justo al final

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fichero,true))

Esto produce un error **StreamCorruptedException** al leer el fichero, ya que cuando termine de leer el grupo de registros que grabamos en la primera sesión empezara a leer la cabecera en vez de un nuevo registro produciéndose un error.

Contenido del fichero

Primera sesión con el fichero					Segunda sesión con el fichero. Le añadimos datos			
cabecera	Persona	Persona	Persona	Persona	cabecera	Persona	Persona	Persona

Tendríamos que eliminar esa segunda cabecera y empezar a grabar directamente objetos

ALMACENAR OBJETOS EN FICHEROS: Serialización

Problemas a tener en cuenta con ObjectOutputStream.

Una forma de solucionar esto es creando una nueva clase nuestra que sea igual que la ObjectOutputStream pero que sobrescriba el método **writeStreamHeader** (el encargado de escribir dicha cabecera) de forma que cuando tengamos ya un fichero creado con objetos si queremos añadir, en vez de definir el objeto ObjectOutputStream utilizaremos la nueva clase diseñada por nosotros que tiene modificado el método writeStreamHeader para que no escriba nada en la cabecera (dejándolo vacío o utilizando la opción reset).

```
public class MyAppendingObjectOutputStream extends ObjectOutputStream {
    public MyAppendingObjectOutputStream(OutputStream out) throws IOException {
    super(out);
  }
   @Override
  protected void writeStreamHeader() throws IOException {
        reset(); // no escribe la cabecera
  }
}
```

ALMACENAR OBJETOS EN FICHEROS: Serialización

Problemas a tener en cuenta con ObjectOutputStream.

De tal forma que cuando vayamos a **añadir** en un fichero serializable nuevos objetos cuando se abra el fichero en vez de hacerlo con :

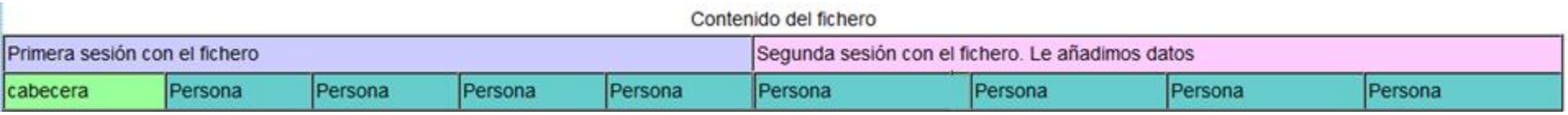
```
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fichero,true));
```

Lo haremos con la nueva clase que hemos creado donde hemos redefinido el método writeStreamHeader para que no escriba la cabecera, de esta forma al final ya se empezará a añadir directamente los objetos.

```
MyAppendingObjectOutputStream oos =
        new MyAppendingObjectOutputStream(new FileOutputStream(fichero, true));
```

Así, podemos añadir y no se añadirá la cabecera en la nueva sesión

```
oos.writeObject( objeto1 );  //grabamos un objeto
```



ALMACENAR OBJETOS EN FICHEROS: Serialización

Grabar objetos en un fichero. Serialización y Persistencia

```
…. la Clase Coche y Clase Motor tienen implements Serializable…….
FileOutputStream fos = null;
ObjectOutputStream salida = null;
Coche coche;
Scanner teclado = new Scanner(System.in);
char continuar = 's';
try {
    //Se crea el fichero
    fos = new FileOutputStream("coches.dat");
salida = new ObjectOutputStream(fos);
do {
    //Se crea el primer objeto Persona
    System.out.print("Introduce la marca: ");
    String marca = teclado.nextLine();
    System.out.print("Introduce el modelo: ");
       String modelo = teclado.nextLine();
    System.out.print("Introduce los caballos: ");
    int cv = teclado.nextInt();
    teclado.nextLine(); /vacio buffer
```

ALMACENAR OBJETOS EN FICHEROS: Serialización

Grabar objetos en un fichero. Serialización y Persistencia

```
                coche = new Coche(marca, modelo, cv);
                //Se escribe el objeto en el fichero
                System.out.println(coche.toString());
                salida.writeObject(coche);

                System.out.println("Quieres continuar (s/n):");
}while (((continuar=(char)System.in.read())!='n') &&
                    ((continuar=(char)System.in.read())!='N'));
fos.close();
 …….
```

ALMACENAR OBJETOS EN FICHEROS: Serialización

Añadir objetos en un fichero. Serialización y Persistencia

Crearemos una clase para sobrescribir el metodo writeStreamHeader, para evitar que escriba una nueva cabecera cuando vayamos a añadir nuevos objetos al fichero

```
public class MyAppendingObjectOutputStream extends ObjectOutputStream {
        //bloque que sobreescribe el metodo writeStreamHeader para que no vuelva a
        //escribir la cabecera del fichero cada vez qeu abramos el fichero para
        //añadir. Sobreescribirmos el método writeStremHeader con reset() y asi
        //no pone nueva cabecera al nuevo bloque de registros que escribamos
    public MyAppendingObjectOutputStream(OutputStream out) throws IOException
{
            super(out);
        }
        @Override
        protected void writeStreamHeader() throws IOException {
                reset();
            }
    }
```

ALMACENAR OBJETOS EN FICHEROS: Serialización

Añadir objetos en un fichero. Serialización y Persistencia

```
….
ObjectOutputStream fsalida = null;
        Coche coche;
        Scanner teclado = new Scanner(System.in);
        char continuar = 's';
        try {
          //creamos el ObjetOutputStream como  MyAppendingObjetStream  para que no escriba la cabecera del nuevo bloque de registros
           MyAppendingObjectOutputStream oos =
                 new MyAppendingObjectOutputStream(new FileOutputStream("coches.dat", true));
          do {
             //Se crea el  objeto Persona
             System.out.print("Introduce la marca: ");
             String marca = teclado.nextLine();
             System.out.print("Introduce el modelo: ");
             String modelo = teclado.nextLine();
```

ALMACENAR OBJETOS EN FICHEROS: Serialización

Añadir objetos en un fichero. Serialización y Persistencia

```
….
        System.out.print("Introduce los caballos: ");
        int cv = teclado.nextInt();
        //vacio el \n del entero
        teclado.nextLine();

        coche = new Coche(marca, modelo, cv);
        //Se escribe el objeto en el fichero
        System.out.println(coche.toString());
        oos.writeObject(coche);

        System.out.println("Quieres continuar (s/n):");
      }while (((continuar=(char)System.in.read())!='n') && ((continuar=(char)System.in.read())!='N'));
      oos.close();
…….
```

ALMACENAR OBJETOS EN FICHEROS: Serialización

ObjetInputStream. (clase flujo de bytes). Lectura Fich. Objetos

- Permite recuperar información de un fichero de objetos.
- El método mas interesante definido por la clase **`ObjectInputStream`** es **`readObject`**`()`, lee un objeto del flujo de entrada (del archivo asociado al flujo de bajo nivel); este objeto se escribió en su momento con el método **`writeObject`**`()`.

```
public Object readObject() throws IOException;
```

- El constructor de flujos **`ObjectInputStream`** tiene como entrada otro flujo, de bajo nivel, de cualquier tipo derivado de **`InputStream`**, por ejemplo **`FileInputStream`**, asociado con el archivo de objetos.

```
ObjectInputStream obje = new ObjectInputStream(
                         new FileInputStream("archivoObjets.dat"));
Persona persona = (Persona) obje.readObjetc();
```

- El **constructor** levanta una excepción si, por ejemplo, el archivo no existe,..., del tipo `ClassNotFoundException`, o bien `IOException`; es necesario poder capturar estas excepciones.
- Devuelve null cuando ha llegado al final del fichero EOF

ALMACENAR OBJETOS EN FICHEROS: Serialización

Lectura y Detección del EOF (End Of File) en un fichero de Objetos

Opción 1:

```
ObjectInputStream ois = new ObjectInputStream(new FileInputStream ("fichero.obj");
try{
  Vehiculo vehiculo;
   while (true) {  //forzamos a que se produzca un error cuando se acaba el fichero e intenta leer
          vehiculo = (vehiculo) ois.readObject();
           System.out.println(vehiculo.toString());
  }
} catch (EOFException ex) { System.out.println("FINAL DE FICHERO");   }
catch(  IOException ex){   System.out.println("error en la lectura del registro");      }
finally{
        ois.close();
}
……
```

ALMACENAR OBJETOS EN FICHEROS: Serialización

Lectura y Detección del EOF (End Of File) en un fichero de Objetos

Opción 2:

```
FileInputStream fis = null;
ObjectInputStream fentrada = null;
Coche coche;
try {
    fentrada = new ObjectInputStream(new FileInputStream("coches.dat"));
     while (( coche = (Coche) fentrada.readObject()) != null)  { //si EOF → excepción
          if (coche instanceof Coche) //compruebo si es un objeto de la clase coche
                System.out.println(coche.toString());     //visualizo el objeto
      }  //fin del while
   }
catch(  IOException e){  //podríamos poner tb antes otro catch con la EOFException
        System.out.println("------fin de fichero----- o posible error en la lectura del registro");
 } finally {
      fis.close();
 }
```

CONTROL DE FLUJO

Tipos de Streams

Comentamos que había dos tipos de flujos o Streams:

- **Flujos de bytes** (**byte streams**) **de 8 bits**. Permite leer bytes. Se usan para manipular datos binarios solo legibles por la máquina o un programa. Orientado a lectura y escritura de datos binarios. Son utilizados para leer archivos de extensión jpeg, png, xls, gif,…. Es decir si abrimos el fichero con un editor de texto no entenderíamos nada

  Clases: **InputStream** y **OutputStream**

- **flujos de caracteres** (**characters streams**) **de 16 bits**. Permite leer caracteres. Se usan para manipular datos legibles por humanos (p.e. fichero de texto, csv,…).

  Clases: **Reader** y **Writer**

  Manejan flujo de caracteres Unicode

Veamos estos últimos….

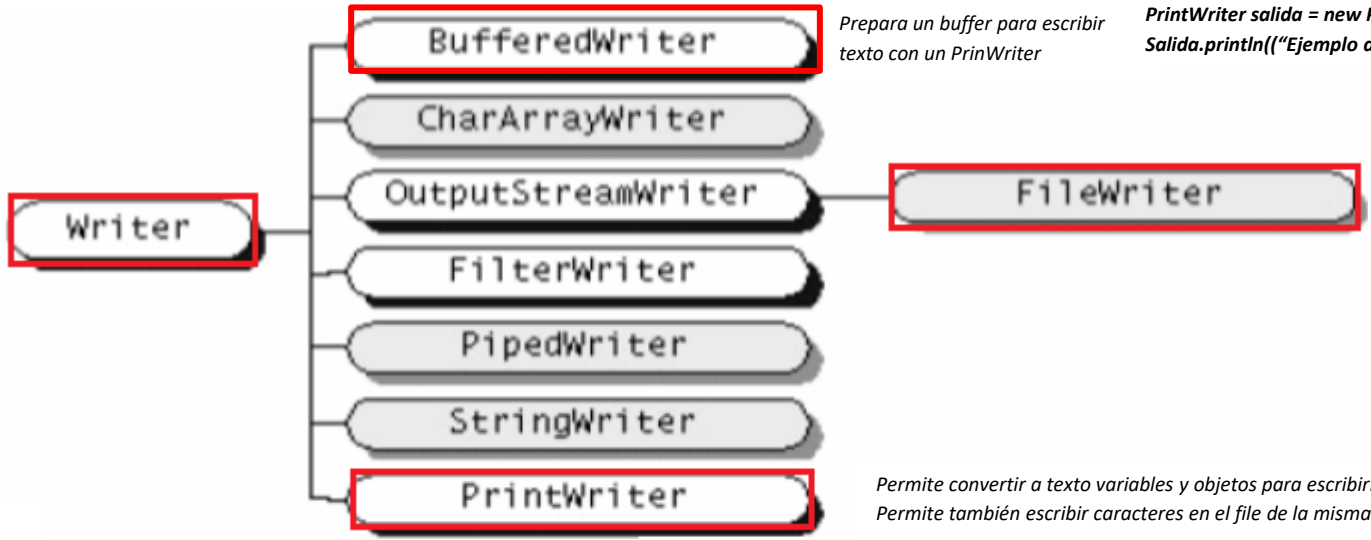
CONTROL DE FLUJO Y CLASES RELATIVAS A FLUJOS

Clases Flujo de caracteres

Lee texto línea a línea de un obj. Reader

BufferReader x = new BufferReader (new FileReader(ruta))
Métodos readLine() , read()

Lee texto de un fichero FileReader (nom_file).
Métodos read() pero lo normal es hacerlo con el BufferReader



Prepara un buffer para escribir texto con un PrinWriter

PrintWriter salida = new PrintWriter(new BufferedWriter (new FileWriter("fich.txt")));
Salida.println(("Ejemplo de escritura de datos");

Escritura de texto en un fichero.
Método write() pero lo normal es usarlo con PrintWriter

Permite convertir a texto variables y objetos para escribirlos en un Writer.
Permite también escribir caracteres en el file de la misma forma que en pantalla

FLUJOS BASADOS EN CARACTERES

Flujos Basados en CARACTERES (16 bits)

Las clases orientadas a bytes no permiten trabajar con caracteres UNICODE (2 bytes) .

Para trabajar con ellos Java dispone de dos clases abstractas : **Reader** y **Writer**

Estas clases así como sus subclases, reciben en el constructor el objeto que representa el flujo de datos (Stream) para el dispositivo de entrada o salida, pudiendo envolverlo en otras clases para manipularlo de una forma más sencilla para el programador.

FLUJOS BASADOS EN CARACTERES

Clases de Flujos Basados en CARACTERES (16 bits)

- **Reader** es la clase base, abstracta, de la jerarquía de clases para leer un carácter o una secuencia de caracteres.
- **Writer** es la clase base, abstracta, de la jerarquía de clases diseñadas para escribir caracteres

Clases y subclases:

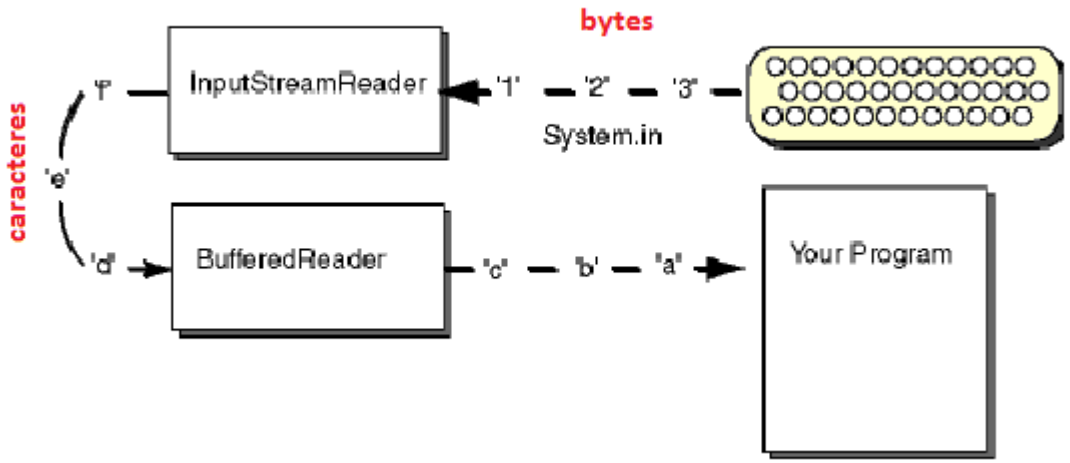
Reader *(lectura)*
- BufferedReader
- InputStreamReader
  - FileReader
- FilterReader

Writer *(escritura)*
- BufferedWriter
- OutputWriter
  - FileWriter
- FilterWriter
- PrintWriter

FLUJOS BASADOS EN CARACTERES

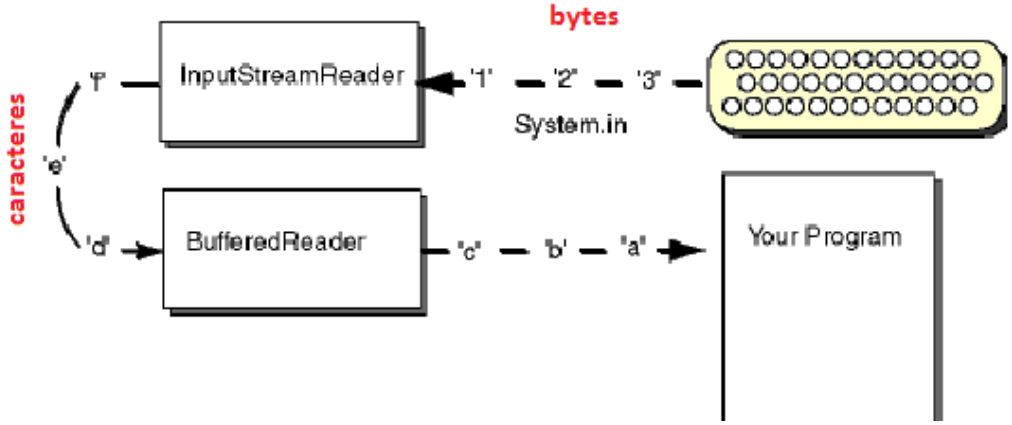
Lectura de Flujos Basados en CARACTERES (16 bits)

- La clase Reader permite leer caracteres y dispone de métodos para leer caracteres, sin embargo si queremos leer por teclado (leemos bytes) seguimos teniendo **System.in**, que es un **InputStream** (basado en bytes) y no un **Reader**(basado en caract).
- Los flujos de la clase **InputStreamReader** *envuelven* a un flujo de bytes; convierte la secuencia de bytes en secuencia de caracteres. Generalmente se utilizan estos flujos como entrada en la construcción de flujos con buffer.



FLUJOS BASADOS EN CARACTERES

Lectura de Flujos Basados en CARACTERES (16 bits)



- En el siguiente ejemplo se crea el flujo entradaChar que puede leer caracteres del flujo de bytes System.in (asociado con el teclado):

Lo que introducimos por teclado se lee como bytes y InputStreamReader lo convierte a caracteres

InputStremReader entradaChar = new InputStreamReader(System.in);

BufferedReader ent = new BufferedReader(entradaChar); ***// para utilizar el método readLine()***

String cadena = ent.readLine();

A partir de este momento ya podemos leer del teclado cadenas de caracteres a variables usando el método readLine() –leería cadena hasta \n - o read()-leer carácter-

Leemos bytes que se convierten en caracteres para almacenarlos en una variable

FLUJOS BASADOS EN CARACTERES

Lectura de Flujos Basados en CARACTERES (16 bits)

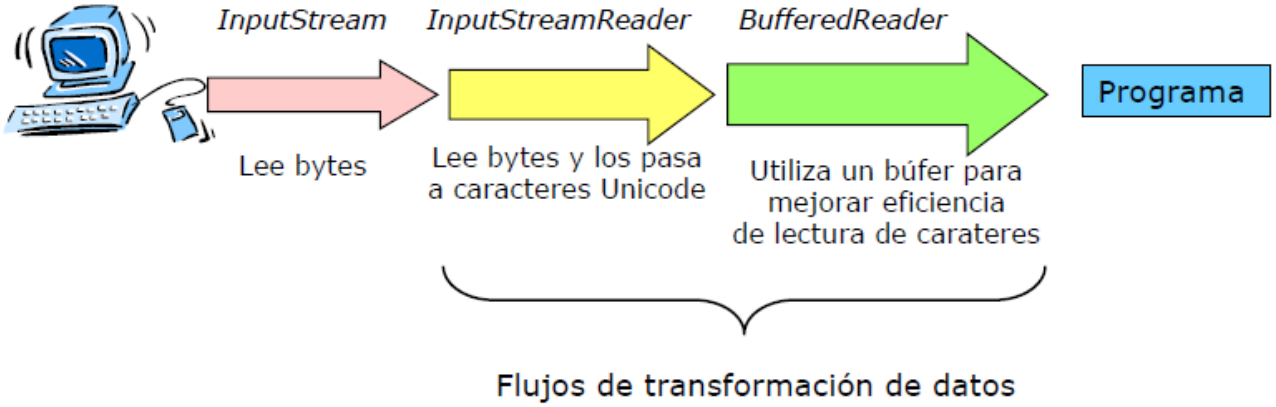
- Los flujos los podemos combinar para obtener la funcionalidad deseada

Lo que introducimos por teclado se lee como bytes, es un InputStream

```
InputStremReader entradaChar = new InputStreamReader(System.in);
BufferedReader ent = new BufferedReader(entradaChar); // ya puedo utilizar el método readLine()
String cadena = ent.readLine();
```

A partir de este momento ya podemos leer del teclado cadenas de caracteres a variables usando el método readLine() –leería cadena hasta /n - o read()-leer carácter-

Leemos bytes que se convierten en caracteres para almacenarlos en una variable



FLUJOS BASADOS EN CARACTERES

Lectura de Flujos Basados en CARACTERES (16 bits)

Lectura desde Teclado

```
public class EjemploLecturaTecladoInputStreamReader {
   public static void main(String[] args) throws IOException {
      InputStreamReader teclado = new InputStreamReader(System.in);
      BufferedReader lectorTeclado = new BufferedReader(teclado); //Ya tenemos el "lector"
      System.out.println("Por favor ingrese su nombre");
      String nombre = lectorTeclado.readLine(); //lee el nombre con readLine() que retorna un String con el dato
      System.out.println("Bienvenido " + nombre + ". Por favor ingrese su edad");
      String entrada = lectorTeclado.readLine();    //siempre lee Cadenas de caracteres
    //readLine siempre retorna String y la clase BufferedReader...no tiene un método para leer enteros, así que debemos convertirlo.
      int edad = Integer.parseInt(entrada)  //transformo la cadena en número
      System.out.println("Gracias " + nombre + " en 10 años usted tendrá " + (edad + 10) + " años.");
   }
}
```

FLUJOS BASADOS EN CARACTERES

Lectura de un Fichero de datos de caracteres: FileReader

- Para leer archivos de texto, archivos de caracteres, se puede crear un flujo del tipo **FileReader**. Esta clase deriva de **InputStreamReader**, hereda los métodos read() para lectura de caracteres. El constructor tiene como entrada una cadena con el nombre del archivo.

```
public FileReader(String miFichero) throws FileNotFoundException;
```

Por ejemplo,

```
FileReader fr = new FileReader("C:\cartas.dat");
```

- FileReader no permite leer líneas completas, solo carácter a carácter .
- El EOF se detecta por que al leer devuelve un -1
- IMPORTANTE: *No resulta eficiente, en general, leer directamente de un flujo FileReader; se aconseja utilizar un flujo BufferedReader envolviendo al flujo FileReader.*

FLUJOS BASADOS EN CARACTERES

Lectura de un Fichero de datos de caracteres: FileReader

Lectura de un fichero usando directamente el FileReader,

Con FileReader solo podemos leer carácter a carácter

```
public class LecturaFileWriter {
    public static void main(String[] args) {
        FileReader   fw = null;
        boolean eof = false;
        int c = 0;
        File file = new File("lecturaFileWriter.txt"); // Solo es un objeto
```

FLUJOS BASADOS EN CARACTERES

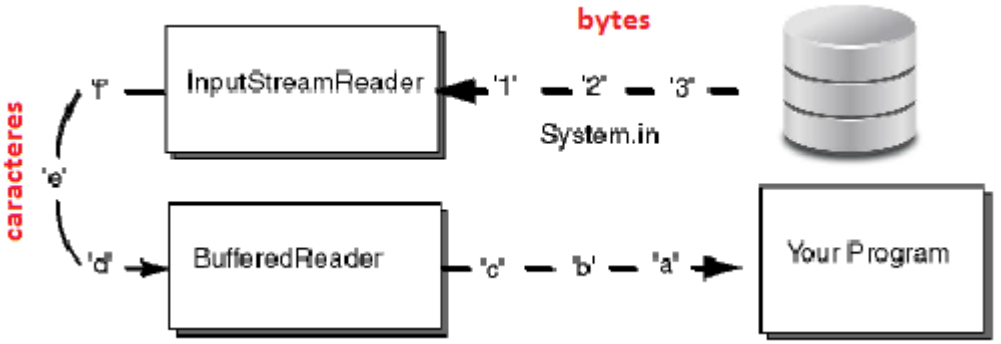
Lectura de un Fichero de datos de caracteres: FileReader

```
if (file.exists()) {
        try {
           fw = new FileReader(file);          // Creamos un fichero actual y un objeto FileReader
           c = fw.read();                      //leo el primer caracter de fichero lo lee como int
           while (c != -1) {                   //si no es eof leo el resto
              System.out.print((char) c);      //lo convierto en caracter
              c = fw.read();                   //read lee int del fichero
           }
           fw.close();                         // Cerramos el archivo cuando todo ha terminado
           System.out.println("\nEl fichero se ha leido con un mensaje");
        } catch (FileNotFoundException ex) {
           Logger.getLogger(LecturaFileWriter.class.getName()).log(Level.SEVERE, null, ex);
        } catch (IOException ex) {
           System.out.println("error en lectura");
        }
     } else {
        System.out.println("El fichero no existe");
     }
  }
}
```

FLUJOS BASADOS EN CARACTERES

Lectura de un Fichero de datos de caracteres: BufferReader

- Aunque se puede leer directamente de un fichero, para optimizar la lectura de archivos de texto, los caracteres no se leen directamente del archivo, sino que su lectura se realiza a un flujo que almacena en un buffer intermedio y los caracteres se leen del *buffer*; consiguiendo mas eficiencia en las operaciones de entrada .
- **`BufferReader`** permite crear flujos de caracteres con buffer, es una forma de organizar el flujo básico de caracteres; esto se manifiesta en que al crear el flujo `BufferReader` se inicializa con un flujo de caracteres de tipo (`InputStreamReader` ...) .
- *Permite leer líneas completas*



FLUJOS BASADOS EN CARACTERES

Lectura de un Fichero de datos de caracteres: BufferReader

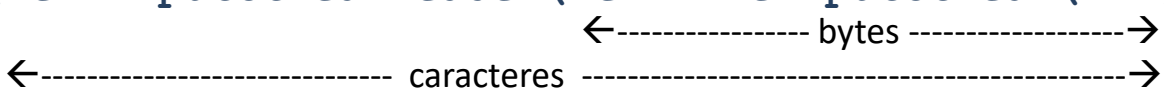
- El constructor de la clase tiene un argumento de tipo `Reader`.

--lectura directamente desde el fichero

```
File mf =  new File("C:\listados.txt");
FileReader fr = new FileReader(mf);
BufferedReader bra = new BufferedReader(fr);
```

--lectura del fichero a través de un BufferReader (más óptimo y aconsejable).

```
File mfz =  new File("Complejo.dat");
 BufferedReader brz =
   new BufferedReader(new InputStreamReader(new FileInputStream(mfz)));
```



- El método más importante es **`readLine();`** lee una línea de caracteres, termina con el carácter de fin de línea, y devuelve una cadena con la línea leída (no incluye el carácter fin de línea). **Puede devolver NULL si lee la marca de fin de archivo EOF**. Lee líneas completas

```
public String readLine() throws IOException;
```

FLUJOS BASADOS EN CARACTERES

Lectura de un Fichero de datos de caracteres

Lectura con BufferReader desde el fichero

```
public class LecturaBufferReader {
   public static void main(String[] args) {
      FileReader fr = null;
      String sCadena ;
      try {
         fr = new FileReader("lecturaFileWriter.txt");
         BufferedReader bf = new BufferedReader(fr);
       //tambien podriamos poner      BufferedReader bf = new BufferedReader(new FileReader("datos.txt"));

         while ((sCadena = bf.readLine())!=null) {  //lee hasta EOF , devuelve null
         System.out.println(sCadena);
       }
      } catch (FileNotFoundException ex)
```

FLUJOS BASADOS EN CARACTERES

Lectura de un Fichero de datos de caracteres

Lectura con BufferReader desde el fichero

```
{
            Logger.getLogger(LecturaBufferReader.class.getName()).log(Level.SEVERE, null, ex);
        } catch (IOException ex) {
            System.out.println("Excepcion en la lectura");
        } finally {
            try {
                fr.close();
                System.out.println("Fin de lectura del fichero");
            } catch (IOException ex) {
                Logger.getLogger(LecturaBufferReader.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

FLUJOS BASADOS EN CARACTERES

Lectura de un Fichero de datos de caracteres

Lectura con BufferReader usando flujo de bytes de un archivo de texto

```
public class LecturaBufferReaderFileInputStream {
   public static void main(String[] args) {
      try {

         File file = new File("lecturaFileWriter.txt");                    // Solo es un objeto

          FileInputStream fstream  = new FileInputStream(file); // Abrimos el archivo

         // Creamos el objeto de entrada
         DataInputStream entrada = new DataInputStream(fstream);

         // Creamos el Buffer de Lectura
         BufferedReader buffer = new BufferedReader(new InputStreamReader(entrada));

         //Sería equivalente a
//BufferedReader buffer = new BufferedReader(new InputStreamReader(new DataInputStream (new FileInputStream("lecturaFileWriter.txt"))));
```

FLUJOS BASADOS EN CARACTERES

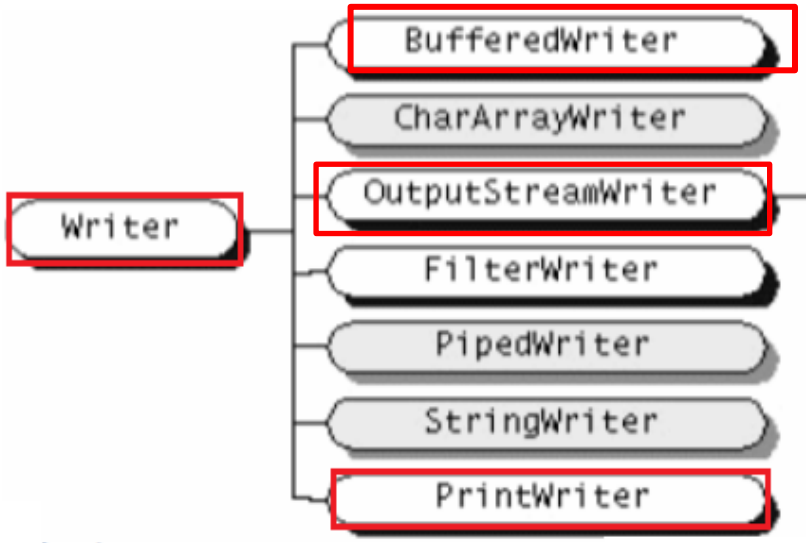
Lectura de un Fichero de datos de caracteres

Lectura con BufferReader usando flujo de bytes de un archivo de texto

```
 String strLinea;
// Leer el archivo linea por linea
          while ((strLinea = buffer.readLine()) != null)   {
             // Imprimimos la línea por pantalla
             System.out.println (strLinea);
          }
       } catch (FileNotFoundException ex) {

Logger.getLogger(LecturaBufferReaderFileInputStream.class.getName()).log(Level.SEVERE,
null, ex);
       } catch (IOException ex) {
          System.out.println("error en lectura");
       }
    }
}
```

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres: Writer



Prepara un buffer para escribir texto con un PrinWriter

PrintWriter salida = new PrintWriter(new BufferedWriter (new FileWriter("fich.txt")));
Salida.println(("Ejemplo de escritura de datos");

Escritura de texto en un fichero.
Método write() pero lo normal es usarlo con PrintWriter

Permite convertir a texto variables y objetos para escribirlos en un Writer.
Permite también escribir caracteres en el file de la misma forma que en pantalla

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres: Writer

- La clase **`Writer`** define métodos **`write()`** para escribir arrays de caracteres o cadenas (`String`).

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres:

OutputStreamWriter

- De `Writer` deriva `OutputStreamWriter` que permite escribir caracteres en un flujo de bytes al cual se asocia en la creación del objeto o flujo., igual que hacíamos con la lectura. Básicamente es un puente entre un flujo de caracteres y un flujo de bytes.
- Cuando hacemos write() sobre un objeto OutputStreamWrites convierte los caracteres a bytes que son acumulados en un buffer antes de ser escritor en el flujo de salida.

←----------buffer de caracteres a escribir ----------→

←-----escribe los caract. en un flujo de bytes----------→

←flujo bytes→

```
Writer out = new BufferedWriter(new OutputStreamWriter(System.out));
```

out.write(mensaje); //con el método write(…) procede a imprimir en pantalla (como bytes) el mensaje

---

```
OutputStreamWriter ot =
                new OutputStreamWriter(new FileOutputStream(archivo));
ot.write(mensaje);  //escrite en el fichero como bytes el mensaje
```

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres:

FileWriter

- la clase **`FileWriter`** es una extensión de **`OutputStreamWriter`**, diseñada para escribir en un archivo de caracteres. Los flujos de tipo **`FileWriter`** escriben caracteres, método **`write()`**, en el archivo asociado el flujo cuando se crea el objeto.

  Contructores:

  - **FileWriter(String nombreFichero) -- reescribe**
  - **FileWriter(String nombreFichero, boolean añadirFinal) -- añade**

Ejem:

```
FileWriter nr = new FileWriter("cartas.dat");
nr.write("Estimado compañero de fatigas");
```

- Los flujos más utilizados para salida de caracteres son de tipo **`PrintWriter`**

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres:

FileWriter

- Contructores:

FileWriter(String path)
FileWriter(File objetoFile);

El fichero se crea y si ya existe su contenido se pierde.

FileWriter(String path, boolean append)
FileWriter(File objetoFile, boolean append)

Si queremos añadir mas contenido al final el parámetro append debe ser true

Ej.

```
FileWriter fw = null;
File file = new File("lecturaFileWriter.txt"); // Solo es un objeto
fw = new FileWriter(file, true); // abrimos el fichero para añadir - TRUE
```

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres:

FileWriter

- Métodos **write(),…**
- Se suele utilizar con las clases **PrintWriter** o **BufferedWriter**
- Con el **BufferedWriter** nos ayuda a manejar los stream en forma de buffer con métodos muy sencillos. Este buffer necesitará saber cual es el fichero. Esto se lo proporcionamos desde la clase FileWriter.

Ejem.

```
String sFichero = "fichero.txt";

BufferedWriter ficheroEscritura = new BufferedWriter (new FileWriter(sFichero));
for (int x=0;x<10;x++)
        ficheroEscritura.write("Fila numero " + x + "\r\n");
```

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres:

Escritura de nuevo fichero solo con FileWriter

```
File file = new File("lecturaFileWriter.txt");
FileWriter  fw = new FileWriter(file); // Creamos un fichero actual y un objeto FileWriter
fw.write("Hola como estas\r\n yo me llamo Roberto\r\n");// Escribimos caracteres en el fichero
fw.flush();                    // Limpiamos
fw.close();                    // Cerramos el archivo cuando todo ha terminado
.....
```

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres:

Añadir información a un fichero con FileWriter

```
FileWriter fw = null;
File file = new File("lecturaFileWriter.txt");
fw = new FileWriter(file, true); // Creamos un fichero actual y un objeto FileWriter
fw.write("esta es la nueva linea a añadir a lo existente");
fw.flush();                      // Limpiamos
fw.close();                      // Cerramos el archivo cuando todo ha terminado
.....
```

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres:

Optimizar de escritura de ficheros de caracteres con FileWriter

- Se suele utilizar con las clases **PrintWriter** o **BufferedWriter**, optimizando el proceso
- Con el **BufferedWriter** nos ayuda a manejar los stream en forma de buffer con métodos muy sencillos. Este buffer necesitará saber cual es el fichero. Esto se lo proporcionamos desde la clase FileWriter.

```
……..
  String sFichero = "filewriterConBufferedWriter.txt";
  File file = new File(sFichero);
   FileWriter fw = new FileWriter(file);  //si quisieramos añadir usar el otro constructor
  BufferedWriter fbw = new BufferedWriter(fw);
  for (int i = 0; i < 5; i++) {
    String cadena ="cadena " + i+"\r\n";
    fbw.write(cadena );
  }
  fbw.close();
```

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres:

PrintWriter

- *Permite escribir de forma formateada* la representación de objetos en un flujo de caracteres.

- Con **PrintWriter** podemos escribir en un fichero igual que lo haríamos en pantalla utilizando los métodos clásicos de impresión en pantalla **print() y println(),** también podemos usar **write() y append()**

- Aunque podemos usarlo directamente, se aconseja utilizar un objeto FileWriter

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres:

PrintWriter

Constructores:

PrintWriter(File file) crea un objeto PrintWriter usando un fichero instanciado como objeto File

PrintWriter(File file, String csn) Creates a new PrintWriter, without automatic line flushing, with the specified file and charset.

PrintWriter(OutputStream out) crea un objeto PrintWriter usando un objeto OutputStream

PrintWriter(OutputStream out, boolean autoFlush) Creates a new PrintWriter from an existing OutputStream.

PrintWriter(String fileName) crea un objeto PrintWriter usando Sstring como nombre de fichero
PrintWriter(String fileName, String csn) Creates a new PrintWriter, without automatic line flushing, with the specified file name and charset.

PrintWriter(Writer out) Crea un PrintWriter, without automatic line flushing, usando un objeto Writer

PrintWriter(Writer out, boolean autoFlush) Crea un PrintWriter con volvado automático

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres:

PrintWriter

Métodos entre otros:

void **print**(tipo f) donde tipo puede ser int, float, char, long, string,…..

void **println**(tipo f) donde tipo puede ser int, float, char, long, string,…..

void **write**(int c) Writes a single character.

void **write**(String s) Writes a string.

void **close**() Closes el stream

void **flush**() Flushes the stream.

FLUJOS BASADOS EN CARACTERES

Escritura de un Fichero de datos de caracteres:

Optimizar de escritura de ficheros de caracteres con PrintWriter

- Con **PrintWriter** podemos escribir en un fichero igual que lo haríamos en pantalla utilizando los métodos clásicos de impresión en pantalla **print() y println(),** también podemos usar **write() y append()**
- Aunque podemos usarlo directamente, se aconseja utilizar un objeto FileWriter

Usado directamente

```
PrintWriter file=
        new PrintWriter("Amigos.txt");
file.println("Juan");
file.println("Jesús");
file.println("Miguel");
file.close();
```

Usando un PrintWriter

```
FileWriter fw =
          new FileWriter((«Amigos.txt");
PrintWriter  file =
          new PrintWriter( fw);
file.println("Juan");
file.println("Jesús");
file.println("Miguel");
file.close();
```

Usando **BufferedWriter**

```
File f =
   new File("FicheroPrintWriterConBufferedWriter.txt");
FileWriter fw = new FileWriter(f,true);
BufferedWriter bw = new BufferedWriter(fw);
 PrintWriter wr = new PrintWriter(bw);
 wr.println("Maria");//escribimos en el archivo
 wr.write("Ana"); //asi tb podemos escribir
 wr.close();
 bw.close();
```

FLUJOS BASADOS EN CARACTERES

Flujos Basados en CARACTERES (16 bits)

Otro uso de PrintWriter

Ejemplo: Escritura de la salida estándar (pantalla) a un fichero.

Leemos un flujo de datos (bytes) por teclado (InputStremReader) y lo transformamos en un flujo de caracteres (BufferReader) y lo copiamos en un fichero de salida, y se hace hasta que escribamos la palabra **salir**

```
try{
        PrintWriter out = null;
        out = new PrintWriter(new FileWriter("salida.txt", true));
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String s;
         while  (!(s = br.readLine()).equals("salir")) { out.println(s); }
         out.close();
}
catch(IOException ex){   System.out.println(ex.getMessage());   }
```

FLUJOS BASADOS EN CARACTERES

Ficheros CSV

StreamTokenizer y StringTokenizer

Los archivos **CSV** (del inglés *comma-separated values*) son un tipo de documento en formato abierto sencillo para representar datos en forma de tabla, en las que las columnas se separan por comas (o punto y coma en el caso de que algún datos use la coma como separador decimal) ..) y las filas por saltos de línea. Suelen tener una extensión .csv o .txt o .dat

El formato CSV es muy sencillo y no indica un juego de caracteres concreto, ni cómo van situados los bytes , ni el formato para el salto de línea.

Son ficheros fáciles de hacer, leer y exportar para trabajar con ellos en distintas aplicaciones.

personas.csv

Nom, apel, edad

Ana, Perez, 23

Juan, Gonzalez, 40

Pedro, Iglesias, 21

coches.txt

año, marca, modelo, descripcion, precio

1997, Ford, E350, "ac, abs, moon", 3000.00

1999, Chevy, "Venture ""Extended Edition""", "", 4900.00

1999, Chevy, "Venture ""Extended Edition, Very Large""", , 5000.00

1996, Jeep, Grand Cherokee, "MUST SELL! air, moon roof, loaded", 4799.00

FLUJOS BASADOS EN CARACTERES

Ficheros CSV

Clase StreamTokenizer

StreamTokenizer solo funciona con objetos **InputStream (bytes)** y permite trocear un string en tokens en función a una serie de constantes de separador ya predefinidas

La clase coge un InputStream y parsea en "tokens", permitiendo que esos tokens sean leídos una vez. El StreamTokenizer puede reconocer identificadores, numeros, comillas , EOF y EOL

Constructores:

StreamTokenizer(Reader r) // r tiene que ser un objeto de caract generado a partir de un inputStream

Ej. Partimos que el fichero fue grabado como un OutputStream (ObjectOutputStream)

```
// crear un ObjectInputStream que será el fichero en bytes de objetos sobre el que vamos a trabajar
ObjectInputStream ois = new ObjectInputStream(new FileInputStream("ficheroToken1.txt"));
 // create a new tokenizer
 Reader file = new BufferedReader(new InputStreamReader(ois));
 StreamTokenizer st = new StreamTokenizer(file);
```

FLUJOS BASADOS EN CARACTERES

Ficheros CSV

StreamTokenizer

Métodos: entre otros el más utilizado

- int nextToken() : devuelve un int que permite conocer el tipo de token leído del InputStream
- String sval(): devuelve una cadena si el token es un String
- double nval(): devuelve un número si el token es un numero

Hay una serie de constantes definidas para determinar el tipo de token:

StreamTokenizer.TT_WORD indica que el token es una palabra.

StreamTokenizer.TT_NUMBER indica que el token es un número.

StreamTokenizer.TT_EOL indica que se ha leído el fin de línea.

StreamTokenizer.TT_EOF indica que se ha llegado al fin del flujo de entrada.

var_objeto_StreamTokenizar.ttype: devuelve el tipo de token

FLUJOS BASADOS EN CARACTERES

Ficheros CSV

StreamTokenizer

```
.....
String text = "Hola. Esto es un fichero de texto\n que será dividido en tokens. 1+1=2";
   try {
     // creamos un fichero de objetos con el texto de text
     FileOutputStream out = new FileOutputStream("ficheroToken1.txt");
     ObjectOutputStream oout = new ObjectOutputStream(out);
     oout.writeUTF(text);
     oout.flush();

       // creamos un  ObjectInputStream para leer el fichero de objetos en bytes
      ObjectInputStream ois = new ObjectInputStream(new FileInputStream("ficheroToken1.txt"));
       // create a new tokenizer
       Reader file = new BufferedReader(new InputStreamReader(ois));
       StreamTokenizer st = new StreamTokenizer(file);

       // print the stream tokens
     boolean eof = false;
```

FLUJOS BASADOS EN CARACTERES

Ficheros CSV

StreamTokenizer

```
..... do {
        int token = st.nextToken();
        switch (token) {
          case StreamTokenizer.TT_EOF:
            System.out.println("EOF encontrado");
            eof = true;      break;
          case StreamTokenizer.TT_EOL:
            System.out.println("Fin de linea");      break;
          case StreamTokenizer.TT_WORD:
            System.out.println("Palabra: " + st.sval);          break;
          case StreamTokenizer.TT_NUMBER:
            System.out.println("Number: " + st.nval);      break;
          default:
            System.out.println((char) token + " encontrado.");
        }
      } while (!eof);
        ois.close();
.....
```

FLUJOS BASADOS EN CARACTERES

Ficheros CSV

StringTokenizer

StringTokenizer es parecido al StreamTokenizer, nos ayuda a dividir un string en substrings o tokens, en base a otro string (normalmente un carácter) como separador entre ellos denominado **delimitador**.

Devuelve todos los símbolos contenidos en una cadena de caracteres de uno en uno, y separados por un delimitador que habrá que especificar.

Los delimitadores de fin de línea pueden ser:

- espacio en blanco → delimitador por defecto
- tabulador \t
- salto de línea \n
- retorno \r
- avance de página \f

FLUJOS BASADOS EN CARACTERES

Clase StringTokenizer. Ficheros CSV

StringTokenizer

Constructores para definir un objeto StreamTokenizer:

- **StringTokenizer var_obj = new StringTokenizer (obj_String);** //delimitador por defecto esp. blanco

- **StringTokenizer var_obj = new StringTokenizer (obj_String, "delimitador");**

Ej.

StringTokenizer v_tokens = new StringTokenizer(linea, ","); // el separador será la ,

Ana, Perez, 24

Ej.

StringTokenizer v_tokens = new StringTokenizer(linea); // el separador será espacio

Ana Juan "Maria Antonia"

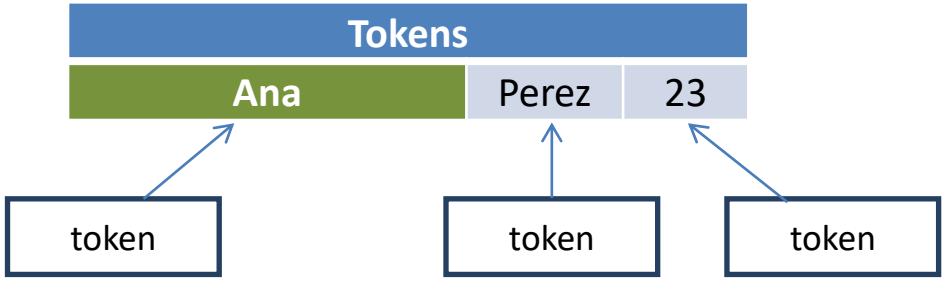
FLUJOS BASADOS EN CARACTERES

Clase StringTokenizer. Ficheros CSV

StringTokenizer

Token → es cada uno de los elementos que separa el delimitador

```
StringTokenizer v_tokens = new StringTokenizer(linea, ",");  // el separador será la ,
```



Al crear la variable objeto v_tokens, esta se sitúa en el primer token de la línea

FLUJOS BASADOS EN CARACTERES

Clase StringTokenizer. Ficheros CSV

StringTokenizer

Métodos:

- **countTokens()** : devuelve un int. Cuenta el número de tokens que tiene el StringTokenizer desde la posición en la que estamos recorriendo.

- **hasMoreTokens():** boolean . Método que chequea si hay más tokens en el String de tokens que tiene StringTokenizer.

- **nextToken():** String. Devuelve el siguiente token

Ej.

```
StringTokenizer st = new StringTokenizer("Ana, Perez, 23",  ",");
while (st.hasMoreTokens()){
        System.out.println(st.nextToken()); //mostraria todos los tokens de la línea
}
```

FLUJOS BASADOS EN CARACTERES

Clase StringTokenizer. ¿Cuando usar bucle para recorrer una línea de tokens y cuando no?

Si el registro del fichero texto tiene una estructura fija NO es necesario usar el bucle con el hasMoreTokens(),aunque se puede usar	**Si el registro del fichero de texto NO tiene una estructura fija, SI es necesario usar el bucle con el hasMoreTokens()**
nombre, apellido, edad nombre, apellido, edad nombre, apellido, edad ….	Cadena, cadena Cadena, cadena, cadena Cadena Cadena, cadena, cadena, cadena, cadena
Cada línea leída tiene el mismo núm de tokens	Cada línea leída NO tiene el mismo núm de tokens
StringTokenizer tokens = new StringTokenizer(linea, ","); String nombre= tokens.nextToken() ; String apellido= tokens.nextToken(); String edad= tokens.nextToken(); //la edad como String *//cuidado si edad tiene espacios " "→ no podríamos parsear* *(Excepcion) sería necesario eliminar previamente los espacios para parsear*	StringTokenizer tokens = new StringTokenizer(linea, ","); //no sabemos cuantos tokens hay en la línea while (tokens.hasMoreTokens()) { System.out.println(tokens.nextToken()); } //estaría leyendo en el bucle tokens hasta que no queden mas tokens

FLUJOS BASADOS EN CARACTERES

Clase StringTokenizer. Ficheros CSV

AVISO: Teniendo una línea de tokens, cada vez que hacemos nextToken() → devuelve un String aunque el token sea un núm, y cuidado si hay espacio antes del numero

pepe, perez, 24

```
StringTokenizer tokens = new StringTokenizer(linea, ",");
String nombre= tokens.nextToken() ;
String apellido= tokens.nextToken();
String edadComoString= tokens.nextToken();  " 24" → con un espacio delante pepe, perez, 24
```

IMPORTANTE: Si lo quiero convertir a num → tengo que eliminar los espacios para evitar la excepción NumberFormatExceptio y eso se hace con el método trim() que tiene la Clase String y despues hacer un parset para convertirlo a Entero

```
int edadComoNum = Integer.parseInt( (tokens.nextToken()).trim())
                  ←-String con posibles espacios--→
                  ←--------String con posibles SIN  espacios------→
←-----------------------int ----------------------------------------------------------------→
```

FLUJOS BASADOS EN CARACTERES

Clase StringTokenizer. Ficheros CSV

Grabar un fichero tipo csv de varias líneas con nombre, apellido y edad

nombre, apellido, edad
nombre, apellido, edad
nombre, apellido, edad
nombre, apellido, edad
……

FLUJOS BASADOS EN CARACTERES

Clase StringTokenizer. Ficheros CSV

```
Persona p;
FileWriter fw = null;
try {
   fw = new FileWriter("nombres.csv");
   BufferedWriter fsalida = new BufferedWriter(fw);
    p = new Persona("Dolores", "Fuertes", 32);
   fsalida.write(p.getNombre() + "," + p.getApellido() + "," + p.getEdad()+"\r\n");

    p = new Persona("Onofre", "Nadol", 25);
   fsalida.write(p.getNombre() + "," + p.getApellido() + "," + p.getEdad()+"\r\n");
          …….

   p = new Persona("Agustin", "Dormido", 33);
   fsalida.write(p.getNombre() + "," + p.getApellido() + "," + p.getEdad()+"\r\n");
  fsalida.close();
 } catch(  ……   )….
```

FLUJOS BASADOS EN CARACTERES

Clase StringTokenizer. Ficheros CSV

Leer el fichero csv creado anteriormente sabiendo que tiene el siguiente formato

nombre, apellido, edad
nombre, apellido, edad
nombre, apellido, edad
nombre, apellido, edad
……

FLUJOS BASADOS EN CARACTERES

Clase StringTokenizer. Ficheros CSV

```
public class LeerFicheroCSV {

//método para tokenizar
 public static void tokenizar(String linea){
      StringTokenizer tokens = new StringTokenizer(linea, ",");

      while(tokens.hasMoreTokens()){ //no seria necesario al ser una linea solo y ya  especificar todos los tokens
         String nom = tokens.nextToken() ;
         String apel = tokens.nextToken();
         int   edad = Integer.parseInt(tokens.nextToken());
         Persona persona = new Persona (nom, apel, edad);
         System.out.println(persona.toString());
      }
   }
.....
```

FLUJOS BASADOS EN CARACTERES

Clase StringTokenizer. Ficheros CSV

```
public static void main(String[] args) {
  try {
          FileReader fr = null;
          fr = new FileReader("nombres.csv");
          BufferedReader registro = new BufferedReader(fr);
           String cadena = registro.readLine();              //leemos el primer registro
          while (cadena != null) {
             tokenizar(cadena);  //llamamos al método que nos permite tokenizar
             cadena = registro.readLine(); //leemos el siguiente registro
          }
       } catch (FileNotFoundException ex) {
          Logger.getLogger(LeerFicheroCSV.class.getName()).log(Level.SEVERE, null, ex);
       } catch (IOException ex) {
          Logger.getLogger(LeerFicheroCSV.class.getName()).log(Level.SEVERE, null, ex);
      }
  }
}
```

FLUJOS BASADOS EN CARACTERES

Clase StringTokenizer. Ficheros CSV

Crear un fichero de texto con el bloc de notas, llamado DATOS.CSV separando los valores por campos (nom, apel1, apel2, direccion, codpostal, ciudad, provincia) y los registros con salto de línea (intro). Introducir varias líneas

Realizar un programa en java que permita visualizar cada una de las líneas del fichero, debiendo mostrar al final cuantos registros (líneas) hay.

¿Si los campos están separados por ; en vez de por comas, como se lo harías saber al programa?

RUTA DE LOS FICHEROS

Especificar PATH del Fichero

Según el SO el separador de los Path o rutas de los ficheros varía.

La clase **File.separator**: favorece la portabilidad de aplicaciones entre distintos S.O.

Podríamos hacer una función que al pasarle una ruta nos devolviera la adecuada según el separador del sistema actual, del siguiente modo:

```
static String substFileSeparator(String ruta){
        String separador = "\\";
        try{            // Si estamos en Windows
                        if  ( File.separator.equals(separador) ) separador =  "/" ;
                        // Reemplaza todas las cadenas que coinciden con la expresión
                        // regular dada oldSep por la cadena File.separator
                        return ruta.replaceAll(separador, File.separator);
        }catch(Exception e){      // Por si ocurre una java.util.regex.PatternSyntaxException
                        return ruta.replaceAll(separador + separador, File.separator); }
}
```

RECORDAR

USO DE LOS Buffered

Si usamos sólo **FileInputStream**, **FileOuputStream**, **FileReader** o **FileWriter**, cada vez que hagamos una lectura o escritura, se hará físicamente en el disco duro. Si escribimos o leemos pocos caracteres cada vez, el proceso se hace costoso y lento, con muchos accesos a disco duro.

Los **BufferedReader**, **BufferedInputStream**, **BufferedWriter** y **BufferedOutputStream** añaden un buffer intermedio. Cuando leamos o escribamos, esta clase controlará los accesos a disco.

Si vamos escribiendo, se guardará los datos hasta que tenga bastantes datos como para hacer la escritura eficiente.

Si queremos leer, la clase leerá muchos datos de golpe, aunque sólo nos dé los que hayamos pedido. En las siguientes lecturas nos dará lo que tiene almacenado, hasta que necesite leer otra vez.

Esta forma de trabajar hace los accesos a disco más eficientes y el programa correrá más rápido. La diferencia se notará más cuanto mayor sea el fichero que queremos leer o escribir.

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Modo de acceso a los registros

Permite determinar la forma de acceder a los distintos registros de un fichero. En función del acceso a la información podemos distinguir entre:

- **ficheros secuenciales**, la información se almacena de manera secuencial, de manera que para recuperarla se debe hacer en el mismo orden en que la información se ha introducido en el archivo. Si por ejemplo queremos leer el registro del fichero que ocupa la 3, tendremos que abrir el fichero y leer los primeros tres registros, hasta que finalmente leamos el registro número tres.

- **fichero de acceso aleatorio,** podríamos acceder directamente a la posición tres del fichero, o a la que nos interesara

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros secuenciales y acceso secuencial en Java

Teniendo en cuenta las distintas formas de grabar y leer información tanto de los flujos orientados a bytes como a carácter, podemos decir que:

Byte:

FileOutputStream y FileInputStream, así como DataInputStream y DataOutputStream podemos considerarlos como de acceso secuencial.

Carácter:

BufferedReader y PrintWriter así como Scanner podemos considerarlos como de acceso secuencial.

Pensar en las operaciones de modificar o borrar un registro, como hacerlo….

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccesFile

Los ficheros con los que hemos trabajado hasta ahora (ya sean ficheros de texto o ficheros binarios con objetos serializados) no resultan adecuados para muchas aplicaciones en las que hay que trabajar eficientemente con un subconjunto de los datos almacenados en disco

La clase Java **RandomAccessFile** se utiliza para acceder a un fichero de forma aleatoria, permitiéndonos acceder a un registro concreto del fichero y trabajar con él.

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccesFile

Constructores:

- RandomAccessFile(String path, String modo);
- RandomAccessFile(File objetoFile, String modo);

Donde

modo	Significado
"r"	Abre el fichero en **modo solo lectura**. El fichero debe existir. Una operación de escritura en este fichero lanzará una excepción IOException.
"rw"	Abre el fichero en **modo lectura y escritura**. Si el fichero no existe se crea.

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

Ejemplo: abrir un fichero aleatorio para **lectura** Se abre el fichero clientes.dat para lectura usando el primer constructor. El fichero debe de existir.

```
RandomAccessFile fichero = new RandomAccessFile("/ficheros/clientes.dat", "r");
```

Ejemplo : abrir un fichero aleatorio para lectura/escritura

Se abre el fichero personas.dat para **lectura/escritur**a usando el segundo constructor. Si el fichero no existe se crea.

```
File f = new File ("/ficheros/personas.dat");
RandomAccessFile fichero = new RandomAccessFile(f, "rw");
```

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

Métodos de acceso a los registro de un fichero RandomAccesFile

long getFilePointer();	Devuelve la posición actual del puntero del fichero. Indica la posición (en bytes) donde se va a leer o escribir.
long length();	Devuelve la longitud del fichero en bytes.
void seek(long pos);	Coloca el puntero del fichero en una posición pos determinada. La posición se da como un desplazamiento **en bytes** desde el comienzo del fichero. La posición 0 indica el principio del fichero. La posición length() indica el final del fichero. Ojo hay que conocer la estructura del registro y lo que ocupa en bytes cada dato del registro, para hacer los cálculos de la posición a la que debemos ir.
public int read()	Devuelve el byte leído en la posición marcada por el puntero. Devuelve -1 si alcanza el final del fichero. Se debe utilizar este método para leer los caracteres de un fichero de texto.
public final String readLine()	Devuelve la cadena de caracteres que se lee, desde la posición marcada por el puntero, hasta el siguiente salto de línea que se encuentre.
public xxx readXxx()	hay un método read para cada tipo de dato básico: **readChar**, **readInt**, **readDouble**, **readBoolean, etc**.
public void write(int b)	Escribe en el fichero el byte indicado por parámetro. Se debe utilizar este método para escribir caracteres en un fichero de texto.
public final void writeBytes(String s)	Escribe en el fichero la cadena de caracteres indicada por parámetro.
public final void writeXxx(argumento)	También existe un método write para cada tipo de dato básico: **writeChar**, **writeInt**, **writeDouble**, **writeBoolean, etc**.

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

Ejemplo: Crear un fichero de acceso aleatorio de números enteros que se introducirán por teclado. Indicar en cada momento la longitud del fichero

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

```
public class grabarRandomAccessFileInteger {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      RandomAccessFile fichero = null;
      int pos, numero;
      long size;
      char continuar;
      try {
         fichero = new RandomAccessFile("randomAccessFile.txt", "rw");
       do {
         size = fichero.length(); //calcular cuántos enteros tiene el fichero
         size = size / 4;   //cada int ocupa 4 bytes y length() nos da el tamaño del fichero en bytes
         System.out.println("El fichero tiene " + size + " enteros");
          System.out.println("Introduce nuevo valor: "); //pedimos que se introduzca el nuevo valor
         numero = sc.nextInt();
```

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

```
            //escribimos el entero
            fichero.writeInt(numero);
            System.out.println("grabado");
            System.out.println("Continuar (s/n): ");
          }while (((continuar=(char)System.in.read())!='n') && ((continuar=(char)System.in.read())!='N'));
      } catch (FileNotFoundException ex) {        System.out.println("Fichero no encontrado");
      } catch (IOException ex) {       System.out.println("Error en la escritura del fichero");
      }
      finally {
         try {
            fichero.close();
         } catch (IOException ex) {         System.out.println("error al cerrar el  fichero");
         }
      }
   }
}
```

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

Ejemplo: Leer el fichero creado anteriormente

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

```
public class LeerRandomAccessFileInteger {
   public static void main(String[] args) {
      RandomAccessFile fichero = null;
      int pos = 0;
      long size;
      try {
         fichero = new RandomAccessFile("randomAccessFile.txt", "r");
         //calcular cuántos enteros tiene el fichero
         size = fichero.length();
         size = size / 4; //cada entero ocupa 4 bytes
         System.out.println("El fichero tiene " + size + " enteros");
```

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

```
//leemos desde el principio
        do {    //nos situamos en la posición (byte de inicio) y vamos leyendo hasta el final
           System.out.println(" " + fichero.readInt());
           pos++;
        } while (pos < size);
      } catch (FileNotFoundException ex) {      System.out.println(“Fichero no existe”);
      } catch (IOException ex) {          System.out.println("Error en acceso a fichero");
      } finally       {
        try {
           fichero.close();
        } catch (IOException ex) {
           Logger.getLogger(LeerRandomAccessFileInteger.class.getName()).log(Level.SEVERE, null, ex);
        }
      }
   }
}
```

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

Ejemplo: Modificar un número de una determinada posición del fichero. Mostrar el fichero y pedir al usuario que especifique la posición del número que quiere modificar y el valor del nuevo número y modificar dicha posición.

Visualizar el fichero resultante.

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

```
public class ModificarUnaPosicionDeUnRandomAccessFileDeEnteros {
   public static void leerRandomAccess(RandomAccessFile raf) {
      int totalEnteros;
      try {
         //calculamos cuantos enteros hay en el fichero, sabiendo 1 int = 4 bytes
         // y que length nos muestra el total de bytes del fichero
         totalEnteros = (int) (raf.length() / 4);
         System.out.println("\nEl fichero tiene " + totalEnteros + " enteros");
         raf.seek(0); //me situo al principio del fichero
         for (int pos = 0; pos < totalEnteros; pos++) {
            System.out.print(" " + raf.readInt());
            System.out.flush();
         }
      } catch (IOException ex) {          System.out.println("Error en la lectura del fichero");
      }
   }
```

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

```
public static void modificarPosicRandomAccessFile(RandomAccessFile raf) {
    long size;
    int pos, numero;
    Scanner sc = new Scanner(System.in);
    try {          //calcular cuántos enteros tiene el fichero
       size = raf.length();  //total de bytes del fichero
       size = size / 4;  //cada entero ocupa 4 bytes
       System.out.println("\nEl fichero tiene " + size + " enteros");
       //Modificar el entero que se encuentra en una posición determinada
       do {
          System.out.println("Introduce una posición entre 1 y  " + size + " donde quieras modificar: ");
          pos = sc.nextInt();
       } while (pos < 1 || pos > size + 1);
       pos--;  //la posición 1 realmente es la 0
       //nos situamos en la posición (byte de inicio) del entero a modificaren Java un entero ocupa 4 bytes
       raf.seek(pos * 4);
```

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

```
if (size != 0) {  //el fichero no esta vacio leemos y mostramos el valor actual
        System.out.println("Valor actual: " + raf.readInt());
    }
    System.out.println("Introduce nuevo valor: ");
    numero = sc.nextInt(); //pedimos que se introduzca el nuevo valor
    //nos situamos de nuevo en la posición del entero a modificar
    //esto es necesario porque después de la lectura que hemos realizado para mostrar
    //el valor el puntero de lectura/escritura ha avanzado al siguiente entero del fichero.
    //si no hacemos esto escribiremos sobre el siguiente entero
    raf.seek(pos * 4);
    //escribimos el entero
    raf.writeInt(numero);
  } catch (IOException ex) {
    System.out.println("Error en la escritura del fichero");
  }
}
```

FICHEROS SECUENCIALES Y ACCESO ALEATORIO

Ficheros Acceso Aleatorio: RandomAccessFile

```
public static void main(String[] args) {
      RandomAccessFile fichero = null;
      int pos, numero;
      long size;
      try {
         fichero = new RandomAccessFile("randomAccessFile.txt", "rw");
          leerRandomAccess(fichero); //leemos el fichero
          modificarPosicRandomAccessFile(fichero); //procedemos a modificar un valor concreto del fichero
        leerRandomAccess(fichero);   //mostramos el fichero modificado
        try {      fichero.close();
            } catch (IOException ex) {              System.out.println("error en fichero");
            }
      } catch (FileNotFoundException ex) {       System.out.println("Fichero no se encuentra");
      } catch (IOException ex) {                 System.out.println("Error en la E/S del fichero");
     }
  }
}
```