Programación. 1º DAW

Unidad 6: Lectura y escritura de información

Lectura y escritura de la información

- Flujos de datos.
- Tipos de flujos.
- Clases relativas a flujos.
  - El paquete java.io.
  - Flujos de Bytes.
    - Flujos de datos
  - Flujos de caracteres.
    - Entrada desde teclado.
    - Salida a pantalla.
- Ficheros de datos. Registros y métodos de acceso.
- Clase File: Archivos y directorios
  - Creación y eliminación de ficheros y directorios.
  - Escritura y lectura de información en ficheros.
  - Apertura y cierre de ficheros.
  - Utilización de los sistemas de ficheros de texto.

Introducción

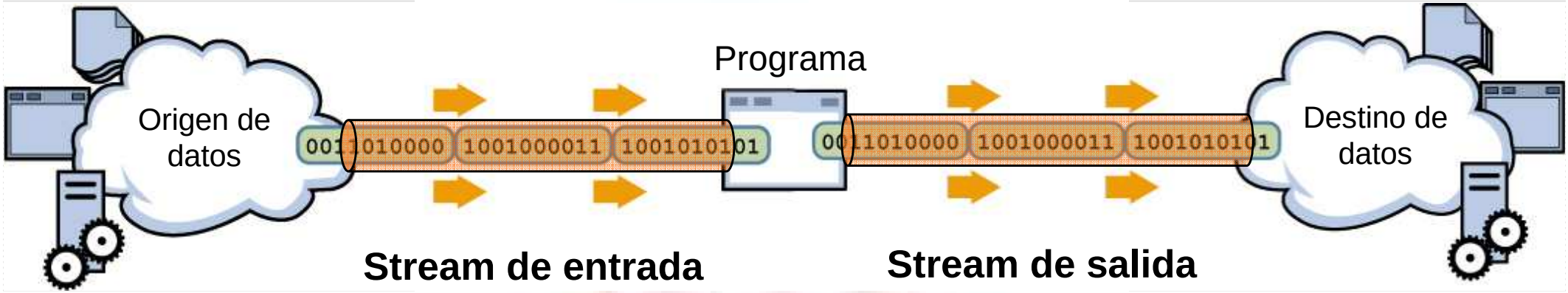
- El acceso a disco y el flujo de información en Java se controla utilizando las clases del paquete **java.io**
- Hay dos grupos de clases que hay que aprender a utilizar:
  - Clase **File** para las correspondientes a ***archivos y directorios***
  - Las *clases abstractas* **InputStream** y **OutputStream** correspondientes a ***corrientes de datos*** (streams)
  - Las corrientes de datos se pueden generar de muchas formas:
    - Usando archivos
    - A partir de vectores de bytes, cadenas, direcciones de internet, sockets de comunicación, etc...

Flujos de datos

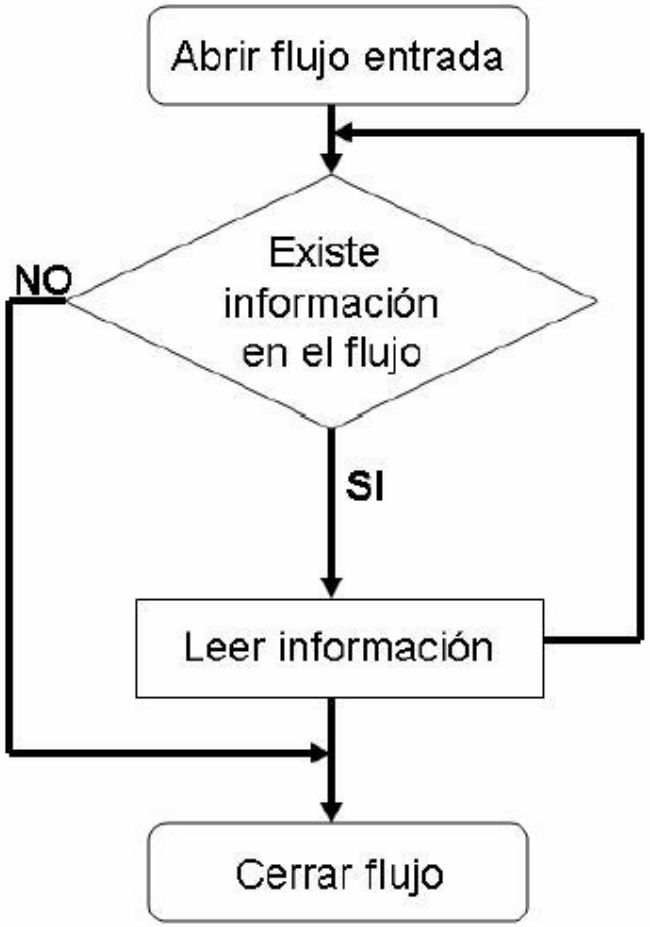
- Los flujos de datos son flujos de información entre el programa y el origen o destino de la información.
- Se tratan en Java mediante objetos stream.
- Sirven para abstraer y simplificar la programación.
- Los programas se encargan de leer y escribir en flujos sin importarles dónde se leen y se escriben los datos.

Flujos de datos

- La entrada y salida de un programa se refiere a las **comunicaciones** que tiene dicho programa con "su entorno" (usuario, ficheros, red, pantalla, otros programas…).
- Al igual que en C y C++, Java usa el concepto de **stream** para gestionar las comunicaciones.
- Un **stream** representa a una *corriente* o *flujo* de datos que sale o entra en nuestro programa.



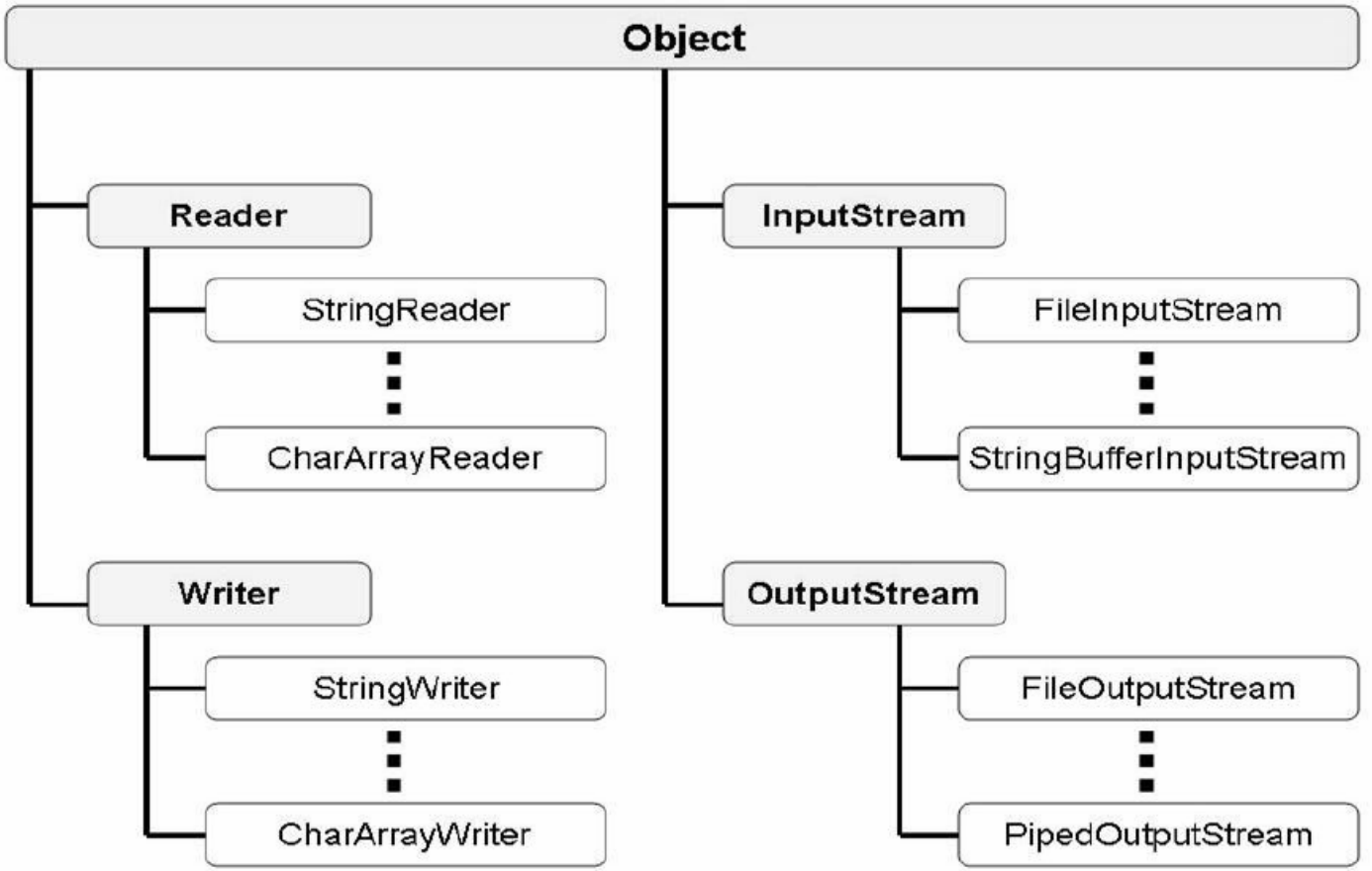
Flujos de datos



Tipos de flujos de datos

- **Clasificación de los Streams:**
  - Según la *dirección* a la que viajan los datos:
    - De **entrada**
    - De **salida**
  - Según el *tipo de dato* que manejan:
    - De **bytes** (8 bits)
    - De **caracteres** (en *Unicode*: 16 bits o más)
    - De **líneas de caracteres**
    - De **datos**
    - De **objetos**
  - Según su *comportamiento*:
    - **Transportadores**: se limitan al transporte de los datos.
    - **Transformadores:** cambian los datos y los transportan.
    - **Retenedores:** almacenan datos y los transportan.

Clases relativas a flujos: El paquete java.io

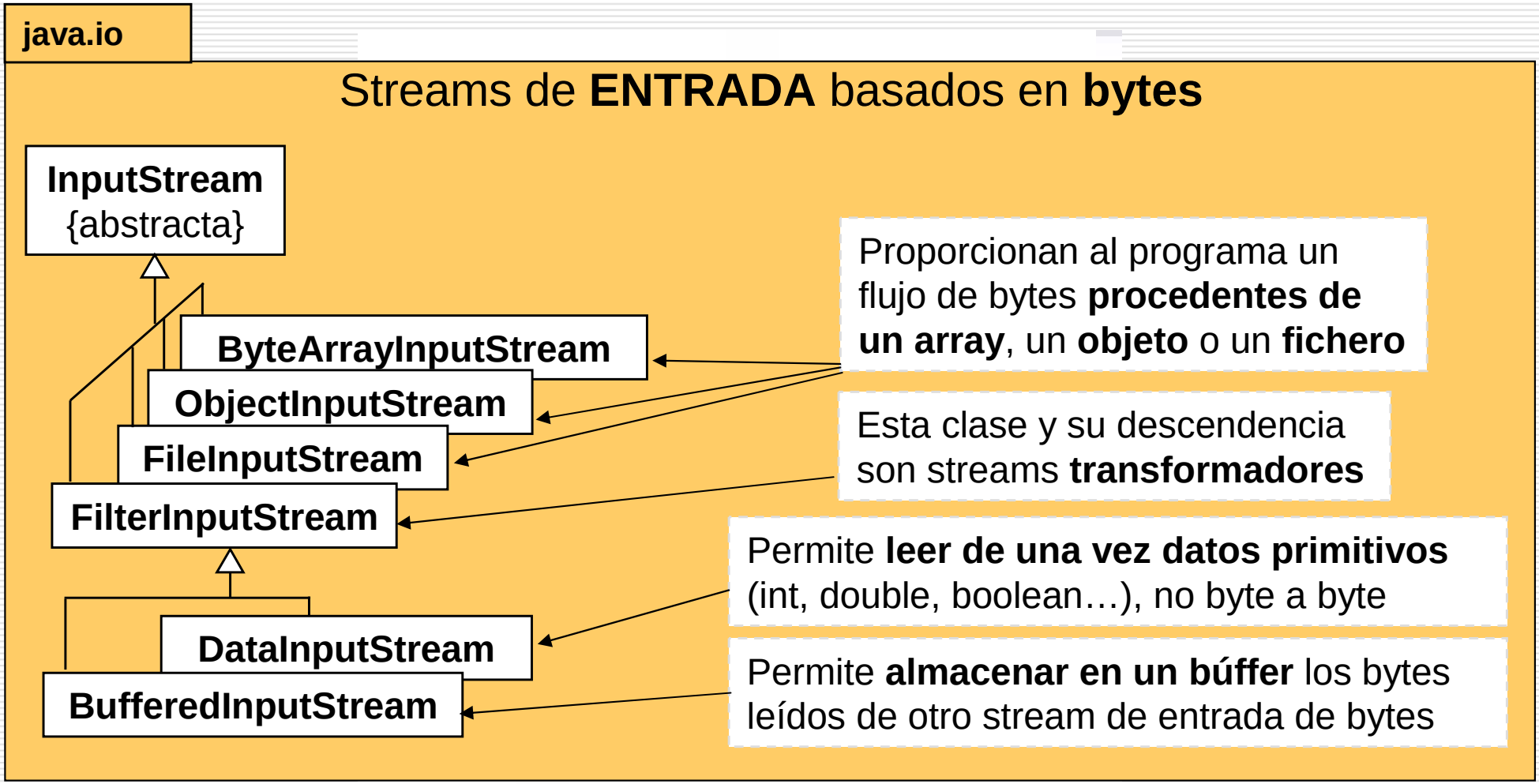


NOTA: Si se le pasa un null como argumento a un constructor o método en las clases o interfaces de este paquete, el programa lanzará una excepción del tipo NullPointerException.

Clases relativas a flujos: Flujos de Bytes

- El paquete java.io tiene varias clases que son derivadas de dos clases básicas:
  - InputStream: **corrientes de entrada** de bytes provenientes de una *fuente*
    - La **fuente** de una *corriente de entrada* o InputStream puede ser un archivo, el teclado, un array de bytes en la memoria RAM, una conexión a un URL en internet o muchas otras cosas.
  - OutputStream: **salida** de bytes que tienen un *destino* definido
    - El **destino** de una *corriente de salida* o OutputStream puede ser un archivo, una impresora, un array de bytes, una conexión a un URL en internet o muchas otras cosas

Clases relativas a flujos: Flujos de Bytes



Clases relativas a flujos: Flujos de Bytes

Métodos de InputStream	
int	**available**() Devuelve el número de bytes que se pueden leer o saltar sin bloquear la corriente.
void	**close**() Cierra la corriente de entrada y libera los recursos del sistema que esté usando.
void	**mark**(int readlimit) Marca la posición actual de la corriente de entrada.
boolean	**markSupported**() Prueba si esta corriente de entrada soporta los métodos `mark` y `reset`.
abstract int	**read**() Lee el siguiente byte de la corriente de entrada.
int	**read**(byte[] b) Lee bytes de la corriente de entrada y los deposita en el vector `b`.
int	**read**(byte[] b, int off, int len) Lee hasta len bytes de la corriente de entrada y los deposita en `b` a partir de `off`.
void	**reset**() Coloca la corriente de entrada en la posición que tenía la última vez que se invocó `mark`.
long	**skip**(long n) Salta y descarta los siguientes n bytes de la corriente de entrada.

Clases relativas a flujos: Flujos de Bytes

- Los tres métodos read sirven para leer bytes.

  public abstract int read() throws IOException;

  - Lee un solo byte y devuelve su valor como un entero entre 0 y 255.

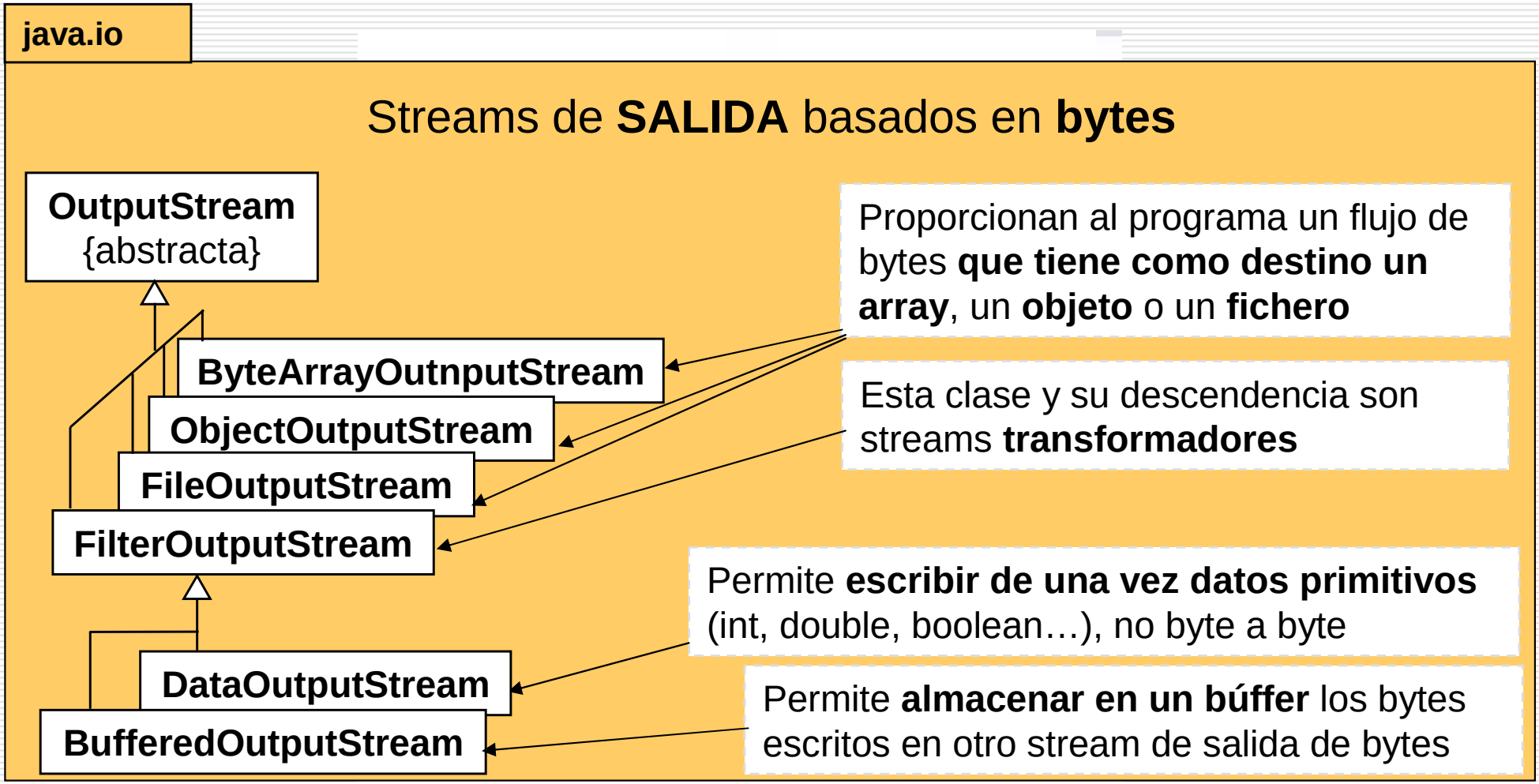
  public int read(byte[] b) throws IOException;

  - Lee tantos bytes como pueda hasta llenar el vector (array) b que se le pasa como parámetro o hasta que se acabe la corriente

  public int read(byte[] b,int pos, int n) throws IOException;

  - Lee a lo más n bytes y los pone en el vector b, a partir de la posición pos.
- Si la corriente de entrada se termina el resultado de los métodos read es -1.

Clases relativas a flujos: Flujos de Bytes



Clases relativas a flujos: Flujos de Bytes

Métodos de Output Stream	
void	**close**() Cierra la corriente de salida y libera los recursos del sistema que está utilizando.
void	**flush**() Fuerza a que se escriba inmediatamente todo lo que pueda estar en el buffer de salida.
void	**write**(byte[] b) Escribe todos los b.length bytes del vector `b` a la corriente de salida.
void	**write**(byte[] b, int off, int len) Escribe len bytes del vector `b` comenzando en la posición `off`.
abstract void	**write**(int b) Escribe un byte especificado por el int `b`.

Clases relativas a flujos: Flujos de Bytes

- Los tres métodos de escritura write son análogos a los de lectura.
  - Escribe un byte aunque se le pase como parámetro un entero
  - Convierte el entero a byte y luego lo escribe.
  - Para no perder información sólo se le deben pasar valores n entre 0 y 255

```
public abstract void write(int b) throws IOException;
```

  - Escribe todos los bytes del vector b que se le pasa como parámetro

```
public void write(byte[] b) throws IOException;
```

  - Escribe n bytes del vector b comenzando desde la posición pos

```
public void write(byte[] b,int pos, int n) throws IOException;
```

Clases relativas a flujos: Flujos de Bytes: Datos

- Las clases **InputStream y OutputStream** sólo permiten leer y escribir datos "byte a byte".
- Hay dos interfaces **DataInput y DataOutput** que resultan útiles para leer o escribir datos más complejos que simples bytes.
- Estos son los métodos de **DataInputStream y DataOutputStream** adicionales a los de InputStream y OutputStream respectivamente
- Para usar estas interfaces basta crear subclases DataInputStream y DataOutputStream de cualquier InputStream o OutputStream respectivamente usando los constructores:

```
public DataInputStream(InputStream is);
public DataOutputStream(OutputStream os);
```

Flujos de Bytes: Datos. DataInput

Métodos de la Interface DataInput	
boolean	**readBoolean**() Lee un byte y devuelve true si el byte es distinto de cero y false si es cero.
byte	**readByte**() Lee un byte y lo devuelve.
char	**readChar**() Lee un char y lo devuelve.
double	**readDouble**() Lee ocho bytes y devuelve el double representado por esos ocho bytes.
float	**readFloat**() Lee cuatro bytes y devuelve el float representado por esos cuatro bytes.
void	**readFully**(byte[] b) Lee bytes y los deposita en el vector de bytes b. Lee tantos bytes como caben en b mientras la corriente de entrada no se agote.
void	**readFully**(byte[] b, int off, int len) Lee hasta len bytes y los deposita en el vector b a partir de la posición pos.
int	**readInt**() Lee cuatro bytes y devuelve el int representado por esos cuatro bytes

Flujos de Bytes: Datos. DataInput

String	**readLine**() Lee una línea de texto.
long	**readLong**() Lee ocho bytes y devuelve el long representado por esos ocho bytes.
short	**readShort**() Lee dos bytes y devuelve el short representado por esos dos bytes.
int	**readUnsignedByte**() Lee un byte y lo devuelve en forma de un int no negativo, es decir entre 0 y 255.
int	**readUnsignedShort**() Lee dos bytes y devuelve un int con valor entre 0 y 65535.
String	**readUTF**() Lee una cuerda codificada en formato UTF-8 modificado.
int	**skipBytes**(int n) Intenta saltarse n bytes. Devuelve el número de bytes que realmente logró saltar.

Flujos de Bytes: Datos. DataOutput

Métodos de la Interface DataOutput	
void	**write**(byte[] b) Escribe a la corriente de salida todos los bytes del vector b.
void	**write**(byte[] b, int off, int len) Escribe len bytes del vector b a partir de la posición off.
void	**write**(int b) Escribe el byte (consistente en los ocho bits menos significativos de) b.
void	**writeBoolean**(boolean v) Escribe un valor booleano a la corriente de salida.
void	**writeByte**(int v) Escribe el byte (consistente en los ocho bits menos significativos de) v.
void	**writeBytes**(String s) Escribe la cuerda s a la corriente de salida.
void	**writeChar**(int v) Escribe el char formado por los dos primeros bytes de v.

Flujos de Bytes: Datos. DataOutput

void	**writeChars**(String s) Escribe los char que forman la cadena s, dos bytes por cada caracter.
void	**writeDouble**(double v) Escribe el double v como ocho bytes.
void	**writeFloat**(float v) Escribe el float v como cuatro bytes.
void	**writeInt**(int v) Escribe el int v como cuatro bytes.
void	**writeLong**(long v) Escribe el long v como ocho bytes.
void	**writeShort**(int v) Escribe dos bytes que representan el short v.
void	**writeUTF**(String str) Escribe la cadena str en el formato UTF modificado de Java.

Clases relativas a flujos: Flujos de Caracteres

java.io

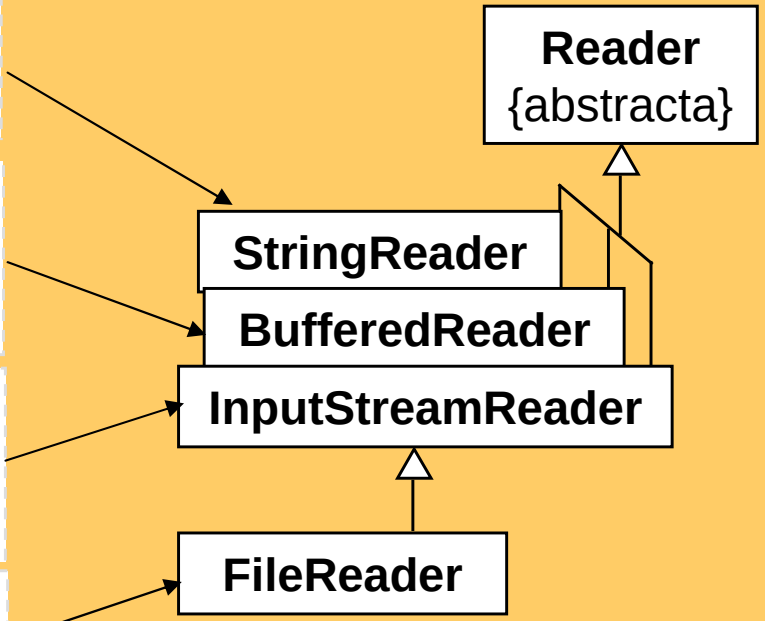
Streams de **ENTRADA** basados en **caracteres**

Proporciona al programa un flujo de caracteres unicode procedentes de un objeto String

Permite **almacenar en un búffer** los caracteres unicode leídos de otro stream de entrada de caracteres

Permite **transformar a caracteres unicode** los bytes de un stream basado en bytes, teniendo en cuenta la codificación local de caracteres

Permite **la lectura directa de caracteres** de un fichero de texto



Clases relativas a flujos: Flujos de Caracteres

java.io

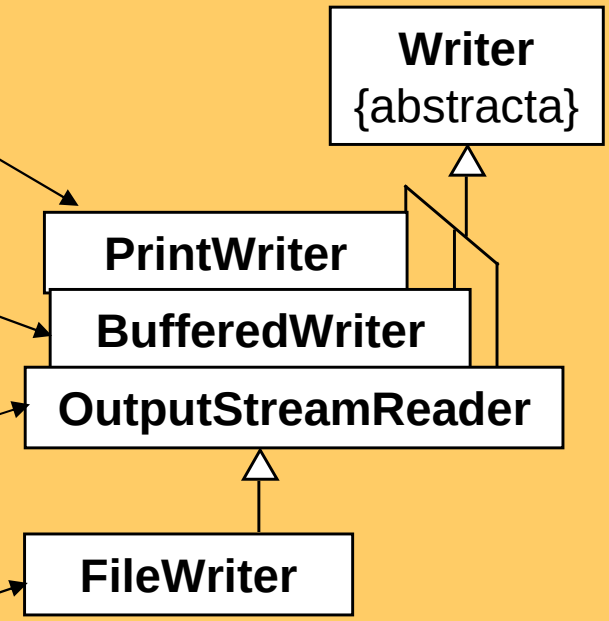
Streams de **SALIDA** basados en **caracteres**

Permite la **escritura formateada** de datos a caracteres unicode

Permite **almacenar en un búffer** los caracteres unicode escritos por otro stream de salida de caracteres

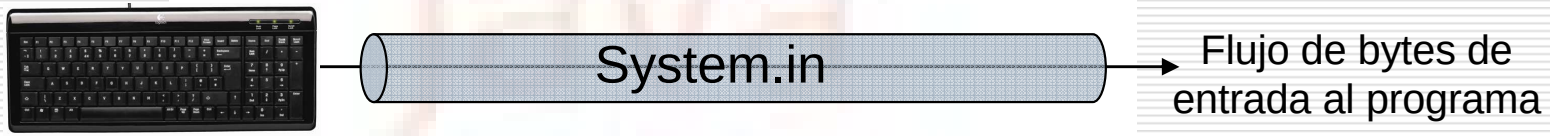
Permite **transformar a bytes** los caracteres unicode de un stream basado en caracteres

Permite **la escritura directa de caracteres** en un fichero de texto



Clases relativas a flujos: Flujos de Caracteres

- **Los streams se pueden conectar unos con otros, combinando sus funcionalidades.** Ejemplo:
  - Partimos de un **flujo de bytes de entrada**
  - Ahora lo combinamos con otro que **convierte los bytes a caracteres unicode**
  - Y por último, lo combinamos con otro que es capaz de **almacenar en un búffer** dichos caracteres.
- Esto es precisamente lo que hay que hacer para poder leer los datos que el usuario introduce por teclado.
  - La clase **System** que representa al sistema nos devuelve un **stream de bytes** a través de su propiedad estática ***in.***



Clases relativas a flujos: Flujos de Caracteres

- A continuación **convertimos el stream de bytes** obtenido a un stream de caracteres unicode **mediante la clase InputStreamReader.**

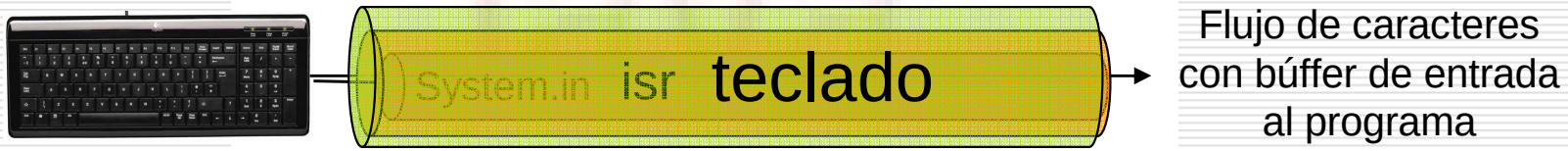
  InputStreamReader isr = **new** InputStreamReader(System.in);

  Con lo que obtendríamos:

- Por último lo combinamos con **BufferedReader** que es capaz de **almacenar** los caracteres leídos **en un búffer**, mejorando la eficiencia en el acceso al teclado.

  BufferedReader teclado = **new** BufferedReader (isr);

  Finalmente obtenemos:



Flujos de Caracteres: Entrada desde teclado

- **BufferedReader:** Para leer texto
  - Tiene el método **readLine()** que lee líneas de texto de manera eficiente y segura realizando las conversiones de caracteres necesarias.
  - Para convertir un InputStream en un BufferedReader es necesario hacerlo en dos pasos:.

```
new BufferedReader(new InputStreamReader(System.in));
```

- Para solicitar un texto al usuario, se hace de la siguiente manera:

```
br= new BufferedReader(new InputStreamReader(System.in));
System.out.print("escriba su nombre:");
String nombre=br.readLine();//Lee hasta detectar enter.
```

Flujos de Caracteres: Salida a pantalla

- **BufferedWriter:** Para escribir texto
  - Tiene los métodos **write(String)** y **newLine()** con los que se pueden escribir líneas de texto una tras otra cómodamente.
  - Para obtener un BufferedReader a partir de un OutputStream, igual que en el caso de la lectura, hay que hacerlo en dos pasos:
    - Se convierte el OutputStream en un OutputStreamWriter
    - Luego éste en un BufferedWriter.

      new BufferedWriter(new OutputStreamWriter(System.out));

- Para escribir un texto en pantalla, se hace de la siguiente manera:

```
BufferedWriter bw;
bw=new BufferedWriter(new OutputStreamWriter(System.out));
bw.newLine();//crea una nueva línea (salto de línea)
bw.write("Cadena a Mostrar");//Envía la cadena al flujo de salida
bw.flush();//muestra el contenido del flujo de salida en pantalla
```

Ejercicios

- Escribir un programa que lea desde teclado un entero, un caracter, un double y una cadena sin utilizar la Clase PedirDatos.

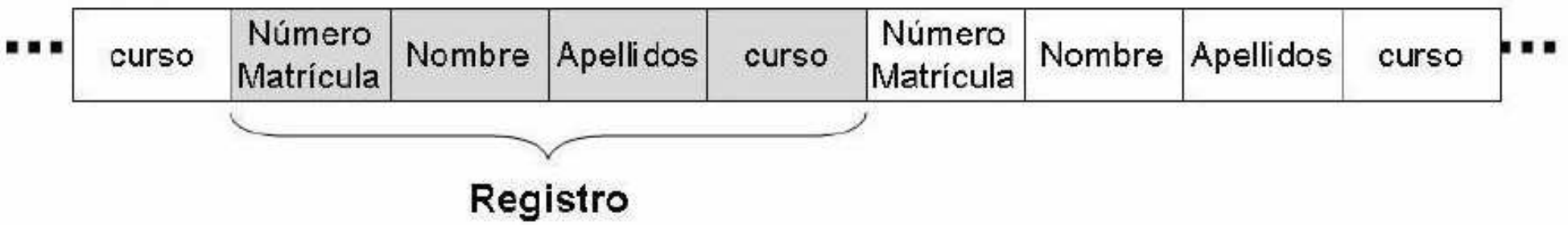
- Escribir un programa que escriba en pantalla una frase utilizando de dos maneras diferentes:
  - Utilizando BufferedWriter
  - Utilizando PrintWriter(System.out)

Ficheros de datos

- Fichero: lugar donde guardamos los datos con los que trabajamos para que perduren en el tiempo. Se almacenan en memoria secundaria (disco duro, pendrive, etc.)
- Existen dos tipos de ficheros:
  - De texto: Se puede leer desde cualquier editor de texto.
  - Binarios: Hay que conocer la estructura interna del fichero para poder interpretar los datos que contiene

Ficheros de datos: Registros

- Un registro es una agrupación de datos.
- Cada uno de estos elementos se denomina Campo.
- Estos campos pueden ser tipos elementales (int, char, etc.) o bien pueden ser a su vez estucturas de datos.
- Es una agrupación generalmente heterogénea de campos tanto por sus tipos de datos como por su contenido.



Ficheros de datos: Métodos de acceso

- **Acceso secuencial**.
  - El acceso al registro n implica la lectura previa de los registros 1 al n-1.
- **Acceso directo**.
  - A los registros se acceden expresando su dirección en el fichero.
- **Acceso por índice**.
  - El acceso a los datos se hace mediante una clave. La clave se busca en una tabla,la cual tiene asociados clave y dirección relativa de los registros. Una vez que se conoce la dirección relativa se accede a los datos.
- **Acceso dinámico**.
  - Se puede acceder a los datos mediante cualquiera de las formas anteriormente citadas.

Clase File
Archivos y Directorios

- En Java tanto los archivos de datos como los directorios se representan como objetos de una misma clase, la clase File (Igual que en UNIX).
- Un objeto File no es más que una dirección de un disco.
  - Puede representar un archivo, un directorio o simplemente una dirección donde por ejemplo se podría crear un archivo o un directorio.
- La clase File sirve para representar sólo los aspectos externos de los archivos, no sirve ni para leer ni para escribir en ellos.

Clase File
Archivos y Directorios

- Manejar ficheros en Java es muy sencillo. En el paquete **java.io** encontramos las herramientas que necesitamos:
  1. La clase **File** representa a un fichero o directorio (que no tiene por qué existir todavía). **File** nos permite averiguar mucha información útil (ruta del fichero, si se puede escribir…)
  2. Por otro lado, si queremos tratar con ficheros de texto usaremos **streams basados en caracteres**, para binarios utilizaremos **streams basados en bytes**.

Clase File
Archivos y Directorios

- Hay tres constructores.
  - El tercero pide como parámetro una dirección de un directorio o archivo.
  - El segundo pide la dirección de un directorio y el nombre de un subdirectorio o un archivo.
  - El primero pide un objeto de la clase File, que debe corresponder a un directorio, y el nombre de un subdirectorio o archivo.
- Para llamar estos constructores no es necesario que los archivos o directorios existan.

Constructores de File
File(File parent, String child) Crea un fichero de nombre child en el directorio representado por parent.
File(String pathname) Crea un fichero con la dirección pathname.
File(String parent, String child) Crea un fichero o directorio de nombre child en el directorio pathname.

Clase File
Archivos y Directorios

- Las constantes pathSeparatorChar y separatorChar son los caracteres que el sistema operativo usa como separador de directorios en la variable PATH y como separador de directorios en los nombres de archivos.
  - Windows pathSeparator=";" y separatorChar="\"
  - En Unix pathSeparator=":" y separatorChar="/".

Variables de File	
static String	**pathSeparator** El caracter que se usa en el sistema operativo como separador de direcciones, en forma de cadena. (En Windows es ; y en Unix es :)
static char	**pathSeparatorChar** El caracter que se usa en el sistema operativo como separador de direcciones, en forma de char. (En Windows es ; y en Unix es :)
static String	**separator** El caracter que se usa en el sistema operativo como separador de directorios en una dirección de archivo, en forma de una cadena. (En Windows es \ mientras que en Unix es /).
static char	**separatorChar** El caracter que se usa en el sistema operativo como separador de directorios en una dirección de archivo, en forma de una cadena. (En Windows es \ mientras que en Unix es /).

Clase File
Archivos y Directorios

Algunos métodos de la clase File	
boolean	**canRead**() Prueba si la aplicación actual puede leer el fichero.
boolean	**canWrite**() Prueba si la aplicación actual puede modicar el fichero.
int	**compareTo**(File pathname) Compara lexicográficamente la dirección de este File con pathname.
boolean	**createNewFile**() Crea un archivo vacío si el actual no existe.
boolean	**delete**() Elimina el archivo o directorio denotado por este File.
boolean	**equals**(Object obj) Prueba la igualdad de este File con el objeto obj.
boolean	**exists**() Pruebe si el fichero o directorio existe.
String	**getAbsolutePath**() Devuelve la dirección absoluta del fichero o directorio.
String	**getCanonicalPath**() Devuelve la dirección canónica del fichero o directorio.
String	**getName**() Devuelve el nombre del fichero o directorio.

Clase File
Archivos y Directorios

String	**getName**() Devuelve el nombre del fichero o directorio.
String	**getParent**() Devuelve la dirección del directorio "padre" del actual fichero o directorio. Devuelve null si la dirección del fichero no tiene un padre explícito.
String	**getPath**() Devuelve la dirección de este fichero o directorio.
boolean	**isAbsolute**() Prueba si la dirección del fichero o directorio es absoluta.
boolean	**isDirectory**() Prueba si el `File` es un directorio.
boolean	**isFile**() Prueba si el `File` es un directorio.
boolean	**isHidden**() Prueba si el archivo o directorio es "escondido".
long	**lastModified**() Devuelve el momento en que el archivo o directorio fue modificado por última vez.
long	**length**() Devuelve el tamaño (en bytes) del archivo.

Clase File
Archivos y Directorios

String[]	**list**() Devuelve un vector con los nombres de todos los archivos y directorios que contiene este directorio.
String[]	**list**(FilenameFilter filter) Devuelve un vector con los nombres de todos los archivos y directorios que contiene este directorio y satisfacen el filtro filter.
static File[]	**listRoots**() Devuelve las direcciones de todas las "raíces" del sistema, esto incluye todos los discos.
boolean	**mkdir**() Creat el directorio con este nombre siempre y cuando el directorio padre exista.
boolean	**mkdirs**() Crea el directorio con este nombre y todos los necesarios que aún no existan.
boolean	**renameTo**(File dest) Cambia el nombre del archivo o directorio al de `dest`.
boolean	**setLastModified**(long time) Cambia la fecha y hora del archivo a la determinada por `time`.
boolean	**setReadOnly**() Marca este archivo o directorio como de "sólo lectura".
URL	**toURL**() Convierte la dirección de este archivo o directorio en una dirección URL.

Clase File
Archivos y Directorios

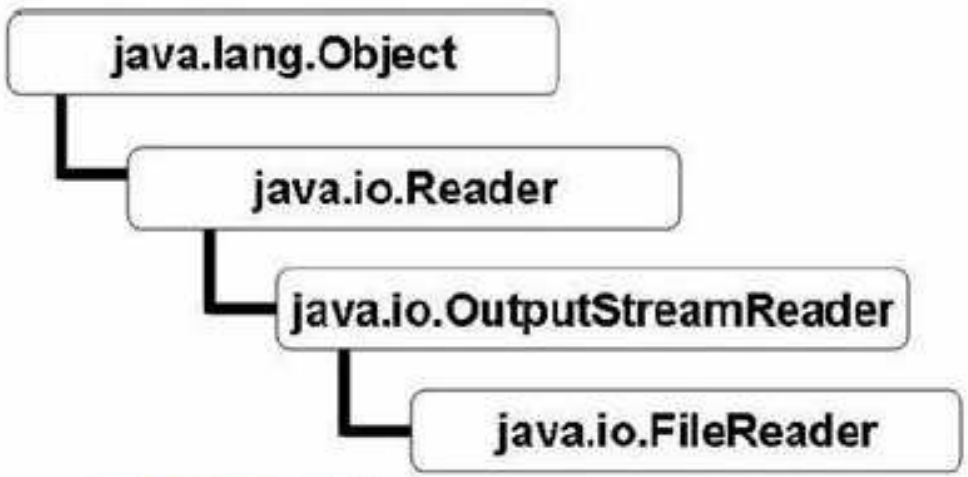
- **exists()**
  - nos informa de la existencia de un archivo
- **mkdirs()**
  - crea todos los directorios necesarios para que un objeto de la clase File corresponda a un directorio real en el disco.
- **isFile() e isDirectory()**
  - nos informan si el objeto de la clase File es un archivo o un directorio.
- **getName()**
  - devuelve el nombre del archivo o directorio
- **getPath()**
  - devuelve la dirección relativa al directorio actual

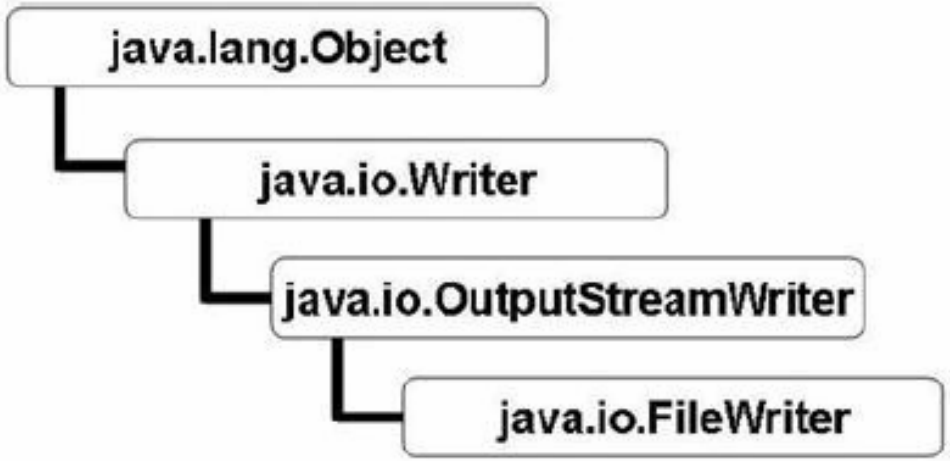
Clase File
Archivos y Directorios

- **getAbsolutePath()**
  - devuelve la dirección absoluta.
- **delete()**
  - borra el directorio o archivo del disco
- **renameTo(String File dest)**
  - cambia y/o mueve el archivo (o directorio) a la nueva dirección.
- **canRead() y canWrite()**
  - nos informan si el archivo se puede leer y si se puede escribir en él respectivamente. Ambos métodos devuelven false si se trata de un directorio.
- **list()**
  - devuelve los nombres de todos los archivos y subdirectorios contenidos en File si se trata de un directorio.

Ejercicio: Crear un programa que recibe como parámetro la dirección de un archivo o directorio y devuelva sus datos obtenidos usando los métodos de la clase File

Escritura y lectura de información en ficheros



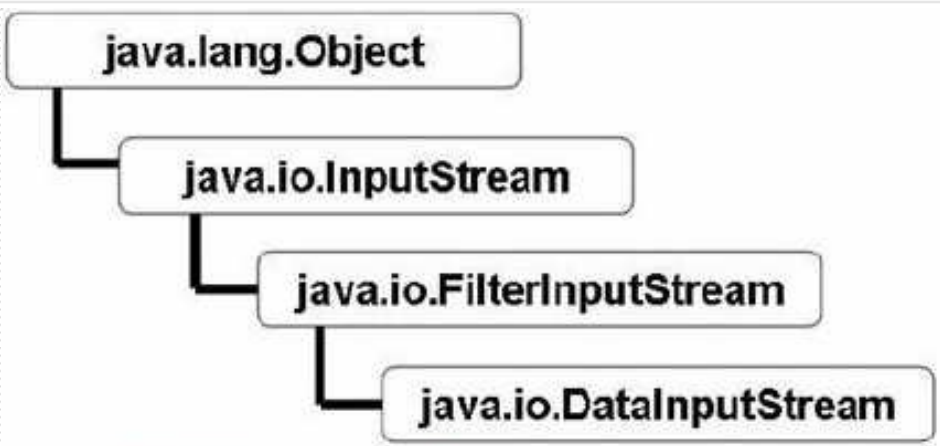


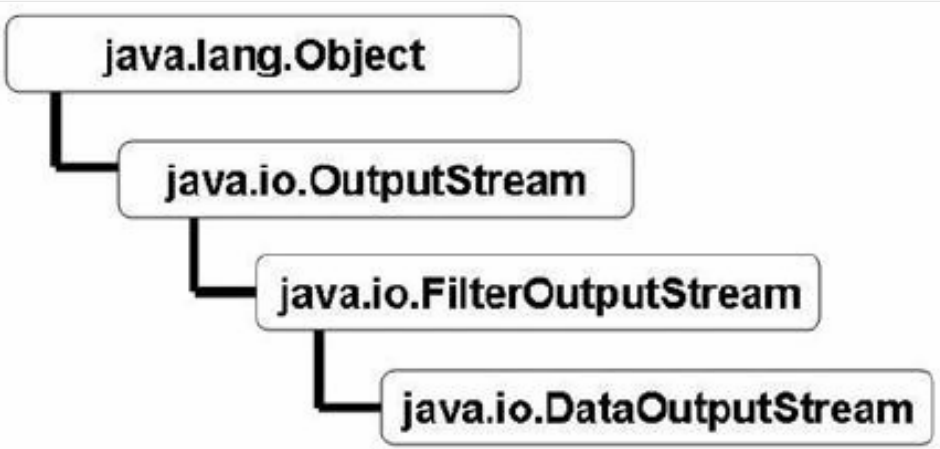
Escritura y lectura de información en ficheros

- Para lectura o escritura en ficheros **de texto** usamos *FileReader* y *FileWriter*. Ejemplo:

```
FileReader fichLec = new FileReader (cadNombre1);
FileWriter fichEsc = new FileWriter (cadNombre2);
…
char caracter = fichLec.read();
fichEsc.write(caracter);
…
```

Escritura y lectura de información en ficheros





- Como se puede observar, ambas clases descienden de una clase filtro (FilterInputStream y FilterOutputStreeam).
- Las clases filtro en Java es la modificación/transformación de los datos.
- FilterInputStream y FilterOutputStream son subclases de InputStream y OutputStream que sirven para filtrar o transformar los datos. Las clases sólo añaden los constructores:

```
public FilterInputStream(InputStream is);
public FilterOutputStream(OutputStream out);
```

Escritura y lectura de información en ficheros

- Para lectura o escritura **de datos** en ficheros usamos DataInputStream y DataOutputStream. Ejemplo:

```
DataInputStream fichLec = new DataInputStream (cad1);
DataOutputStream fichEsc = new DataOutputStream (cad1);
…
int codArticulo = fichLec.readInt();
int numUnidades = fichLec.readInt();
double precio = fichLec.readDouble();
fichEsc.writeInt (codArticulo);
fichEsc.writeInt (numUnidades);
fichEsc.writeDouble (precio);
…
```

Escritura y lectura de información en ficheros

- Todos los **constructores de streams de ficheros** admiten como parámetro de entrada un objeto de la **clase File**.
- Se aconseja que el **acceso a los fichero se haga mediante un buffer** de memoria para mejorar la eficiencia. Para ello debemos usar un stream con capacidad de almacenamiento.
- Ejemplo:

```
File fichero = new File ("pedidos.dat");
DataOutputStream dos = new DataOutputStream (fichero);
BufferedOutputStream stream = new BufferedOutputStream
(dos);
stream.writeInt (codArticulo);
stream.writeDouble (precio);
```

Apertura y cierre de ficheros. Acceso secuencial

- BufferedInputStream y BufferedOutputStream son subclases que se obtienen usando los constructores

```
public BufferedInputStream(InputStream is);
public BufferedInputStream(InputStream is,int tamaño);
public BufferedOutputStream(OutputStream is);
public BufferedOutputStream(OutputStream is,int tamaño);
```

  - Lo que hacen es usar un "buffer" o almacén temporal con el objeto de mejorar la eficiencia de la lectura o la escritura. En el caso de la escritura es importante llamar al método flush() antes de cerrar la corriente. En los constructores tamaño especifica el número de bytes que se usarán en el *buffer*.

Apertura y cierre de ficheros. Acceso secuencial

- Se pueden crear corrientes de entrada y salida a partir de diversas fuentes y con diversos destinos.
  - Por ejemplo, se pueden crear InputStreams y OutputStreams a partir de archivos usando los constructores de la clase FileInputStream y FileOutputStream respectivamente:

    public FileInputStream(File fuente);

    public FileOutputStream(File destino);

- También se puede crear un InputStream usando una matriz de bytes como fuente. Para ello se utiliza uno de los constructores de la clase ByteArrayInputStream:

  public ByteArrayInputStream(byte[] b);

  public ByteArrayInputStream(byte[] b,int pos,int n);

Apertura y cierre de ficheros. Acceso aleatorio

- La clase **RandomAccessFile** proporciona todos los métodos necesarios para leer y escribir en archivos en forma aleatoria.
- La clase RandomAccessFile **no** es una extensión de File. Son clases complementarias.
  - File representa *directorios* y *archivos*, pero no proporciona métodos para acceder al interior de los *archivos*.
  - RandomAccessFile sólo representa archivos y proporciona precisamente métodos para leer su contenido y escribir en ellos diferentes tipos de datos y en cualquier sitio.
- Esta clase es útil para aplicaciones que leen y escriben datos en archivos de un disco local

Apertura y cierre de ficheros. Acceso aleatorio

- Los constructores de RandomAccessFile son:
  public RandomAccessFile(String nombre,String modo) throws IOException
  public RandomAccessFile(File f,String modo) throws IOException
- El primer parámetro debe ser la dirección completa de un archivo o bien un objeto de tipo File que deberá ser un archivo y no un directorio
- El segundo parámetro en ambos casos debe ser **"r"** o **"rw"**
- Llamar a uno de estos constructores *abre* el archivo si ya existe o lo crea si aún no existe
- **Todos** los métodos de RandomAccessFile arrojan IOException, por lo cual todo lo que se programe usando esta clase debe escribirse dentro de una construcción:
  try { ... } catch (IOException ioe) {}

Apertura y cierre de ficheros. Acceso aleatorio

- RandomAccessFile tiene unos métodos para saber el *tamaño* del archivo en bytes y poder *cerrarlo* cuando se ha terminado de trabajar con él:

  public long length() throws IOException
  public void close() throws IOException

- y otros que permiten saber la localización del *puntero* del archivo y colocarlo en una posición arbitraria:

  public long getFilePointer() throws IOException
  public void seek(long pos) throws IOException

- Para leer y escribir RandomAccessFile tiene todos los métodos de las interfaces DataInput y DataOutput que se utilizan también con Streams.

Utilización de los sistemas de ficheros: Apertura y lectura de fichero de texto

```
//Crea el fichero si no existe.
File fentrada=new File("C:\fentrada.txt");

//Abre el fichero en modo lectura
BufferedReader br=new BufferedReader(new FileReader(fentrada));

//Lee un linea o registro del fichero de texto
String registro;
registro=br.readLine();

//recorre el fichero de lectura hasta que no encuentra mas registros
while(registro==null){
        System.out.println(registro);
        //Lee linea a linea el fichero de texto
        registro=br.readLine();
};

//Cierra el buffer y el fichero  de entrada
br.close();
```

Utilización de los sistemas de ficheros: Escritura y cierre de fichero de texto

```
//Crea el fichero de salida si no existe
File fsalida=new File("C:\fsalida.txt");

//Abre el fichero en modo escritura
BufferedWriter bw=new BufferedWriter(new FileWriter(fsalida));

//Escribe una linea en el buffer de salida
bw.write("Nueva Linea");

//Realiza un salto de linea en el fichero de salidaInserta una linea en
bw.newLine();

//Escribe otra linea en el buffer de salida
bw.write("Otra Linea");

//Envia el buffer de salida al fichero de salida
bw.flush();

//Cierra el buffer y el fichero de salida
bw.close();
```

Copyright ©

Para la confección de parte de esta documentación se ha utilizado material con derechos reservados del Copyright del autor **Enrique José Royo Sánchez**, autorizando este su uso a **Antonio Blázquez Pérez** como material didáctico en el IES Polígono Sur de Sevilla