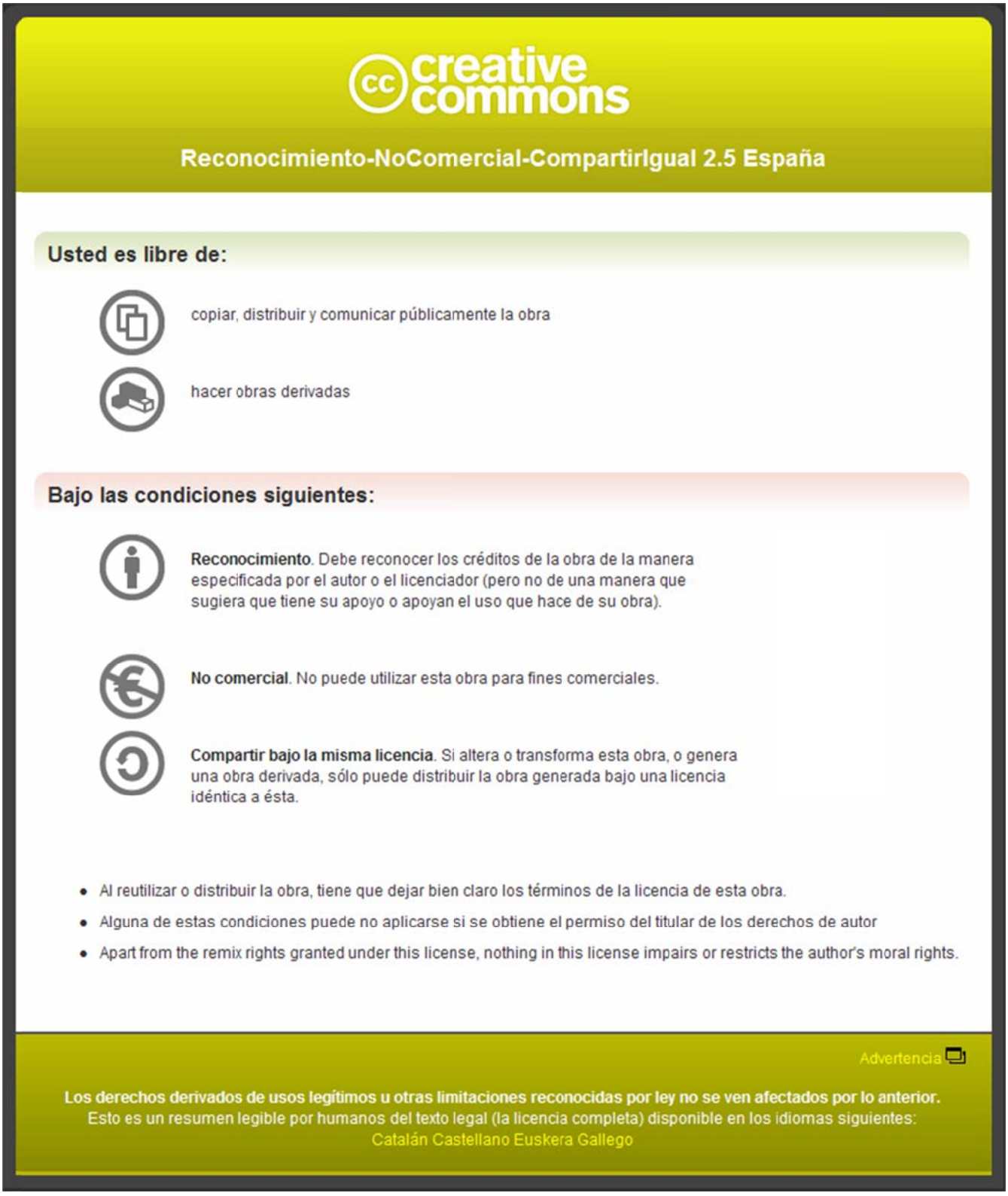
Unidad 10: Entrada y Salida en Java. Ficheros

Fundamentos de Programación. 1º de ASI

Esta obra está bajo una licencia de Creative Commons.
Autor: Jorge Sánchez Asenjo (año 2010) http://www.jorgesanchez.net
e-mail:info@jorgesanchez.net

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de CreativeCommons
Para ver una copia de esta licencia, visite:
http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es
o envíe una carta a:
Creative Commons, 559 Nathan Abbot



Para ver una copia completa de la licencia, acudir a la dirección
http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es

(10) entrada y salida en Java. ficheros

esquema de la unidad

(10.1) entrada y salida en Java

El paquete **java.io** contiene todas las clases relacionadas con las funciones de entrada (**input**) y salida (**output**). Se habla de E/S (o de I/O) refiriéndose a la entrada y salida. En términos de programación se denomina **entrada** a la posibilidad de introducir datos hacia un programa; **salida** sería la capacidad de un programa de mostrar información al usuario.

Todas las clases relacionadas con la entrada y salida de datos están en el paquete **java.io.**

(10.2) archivos

(10.2.1) introducción

Todas las estructuras comentadas en los temas anteriores para almacenar los datos, residen en memoria. De hecho cuando el programa termina, desaparecen.

Para evitar este problema existen los archivos. Se trata de un elemento que pone a disposición el Sistema Operativo para almacenar información de todo tipo bajo un **nombre** y una **extensión**. La extensión indica el tipo de archivo que tenemos.

En definitiva es la estructura habitual en la que se almacena información y cuyo manejo parte de que sabemos sus fundamentos en base al sistema operativo que estemos manejando. Su finalidad es la de poder almacenar datos de forma permanente.

(10.2.2) clase File

En el paquete **java.io** se encuentra la clase **File** pensada para poder realizar operaciones de información sobre archivos. No proporciona métodos de acceso a los archivos, sino operaciones a nivel de sistema de archivos (listado de archivos, crear carpetas, borrar ficheros, cambiar nombre,...).

Un objeto **File** representa un archivo o un directorio y sirve para obtener información (permisos, tamaño,...). También sirve para navegar por la estructura de archivos.

construcción de objetos de archivo

Utiliza como único argumento una cadena que representa una ruta en el sistema de archivo. También puede recibir, opcionalmente, un segundo parámetro con una ruta segunda que se define a partir de la posición de la primera.

```
File archivo1=new File("/datos/bd.txt");
File carpeta=new File("datos");
```

El primer formato utiliza una ruta absoluta y el segundo una ruta relativa. En Java el separador de archivos tanto para Windows como para Linux es el símbolo **/**.

Otra posibilidad de construcción es utilizar como primer parámetro un objeto **File** ya hecho. A esto se añade un segundo parámetro que es una ruta que cuenta desde la posición actual.

```
File carpeta1=new File("c:/datos");//ó c\\datos
File archivo1=new File(carpeta1,"bd.txt");
```

Si el archivo o carpeta que se intenta examinar no existe, la clase **File** no devuelve una excepción. Habrá que utilizar el método **exists**. Este método recibe **true** si la carpeta o archivo es válido (puede provocar excepciones **SecurityException**). También se puede construir un objeto **File** a partir de un objeto **URI**.

el problema de las rutas

Cuando se crean programas en Java hay que tener muy presente que no siempre sabremos qué sistema operativo utilizará el usuario del programa. Esto provoca que la realización de rutas sea problemática porque la forma de denominar y recorrer rutas es distinta en cada sistema operativo.

Por ejemplo en Windows se puede utilizar la barra **/** o la doble barra invertida **** como separador de carpetas, en muchos sistemas Unix sólo es posible la primera opción. En general es mejor usar las clases **Swing** (como **JFileDialog**) para especificar rutas, ya que son clases en las que la ruta de elige desde un cuadro y, sobre todo, son independientes de la plataforma.

También se pueden utilizar las **variables estáticas** que posee **File**. Estas son:

propiedad	uso
static char **separatorChar**	El carácter separador de nombres de archivo y carpetas. En Linux/Unix es **/** y en Windows es ****, que se debe escribir como ****, ya que el carácter **** permite colocar caracteres de control, de ahí que haya que usar la doble barra. Pero Windows admite también la barra simple (**/**)
static String **separator**	Como el anterior pero en forma de String
static char **pathSeparatorChar**	El carácter separador de rutas de archivo que permite poner más de un archivo en una ruta. En Linux/Unix suele ser "**:**", en Windows es "**;**"
static String **pathSeparator**	Como el anterior, pero en forma de String

Para poder garantizar que el separador usado es el del sistema en uso:

```
String ruta="documentos/manuales/2003/java.doc";
ruta=ruta.replace('/',File.separatorChar);
```

Normalmente no es necesaria esta comprobación ya que Windows acepta también el carácter / como separador.

métodos generales

método	uso
String **toString()**	Para obtener la cadena descriptiva del objeto
boolean **exists()**	Devuelve **true** si existe la carpeta o archivo.
boolean **canRead()**	Devuelve **true** si el archivo se puede leer
boolean **canWrite()**	Devuelve **true** si el archivo se puede escribir
boolean **isHidden()**	Devuelve **true** si el objeto File es oculto
boolean **isAbsolute()**	Devuelve **true** si la ruta indicada en el objeto File es absoluta
boolean **equals(**File f2**)**	Compara *f2* con el objeto **File** y devuelve verdadero si son iguales.
int **compareTo(**File f2**)**	Compara basado en el orden alfabético del texto (sólo funciona bien si ambos archivos son de texto) *f2* con el objeto **File** y devuelve cero si son iguales, un entero negativo si el orden de *f2* es mayor y positivo si es menor
String **getAbsolutePath()**	Devuelve una cadena con la ruta absoluta al objeto File.
File **getAbsoluteFile()**	Como la anterior pero el resultado es un objeto File
String **getName()**	Devuelve el nombre del objeto File.
String **getParent()**	Devuelve el nombre de su carpeta superior si la hay y si no **null**
File **getParentFile()**	Como la anterior pero la respuesta se obtiene en forma de objeto File.
boolean **setReadOnly()**	Activa el atributo de sólo lectura en la carpeta o archivo.
URL **toURL()** throws MalformedURLException	Convierte el archivo a su notación URL correspondiente
URI **toURI()**	Convierte el archivo a su notación URI correspondiente

métodos de carpeta:

método	uso
boolean **isDirectory()**	Devuelve **true** si el objeto File es una carpeta y **false** si es un archivo o si no existe.
boolean **mkdir()**	Intenta crear una carpeta y devuelve **true** si fue posible hacerlo
boolean **mkdirs()**	Usa el objeto para crear una carpeta con la ruta creada para el objeto y si hace falta crea toda la estructura de carpetas necesaria para crearla.

boolean **delete()**	Borra la carpeta y devuelve **true** si puedo hacerlo
String[] **list()**	Devuelve la lista de archivos de la carpeta representada en el objeto File.
static File[] **listRoots()**	Devuelve un array de objetos File, donde cada objeto del array representa la carpeta raíz de una unidad de disco.
File[] **listfiles()**	Igual que la anterior, pero el resultado es un array de objetos File.

métodos de archivos

método	uso
boolean **isFile()**	Devuelve **true** si el objeto File es un archivo y **false** si es carpeta o si no existe.
boolean **renameTo(**File f2**)**	Cambia el nombre del archivo por el que posee el archivo pasado como argumento. Devuelve **true** si se pudo completar la operación.
boolean **delete()**	Borra el archivo y devuelve **true** si puedo hacerlo
long **length()**	Devuelve el tamaño del archivo en bytes (en el caso del texto devuelve los caracteres del archivo)
boolean **createNewFile()** throws IOException	Crea un nuevo archivo basado en la ruta dada al objeto File. Hay que capturar la excepción *IOException* que ocurriría si hubo error crítico al crear el archivo. Devuelve **true** si se hizo la creación del archivo vacío y **false** si ya había otro archivo con ese nombre.
static File **createTempFile(** String prefijo**,** String sufijo**)** throws IOException	Crea un objeto File de tipo archivo temporal con el prefijo y sufijo indicados. Se creará en la carpeta de archivos temporales por defecto del sistema. El *prefijo* y el *sufijo* deben de tener al menos tres caracteres (el sufijo suele ser la extensión), de otro modo se produce una excepción del tipo **IllegalArgumentsException** Requiere capturar la excepción **IOException** que se produce ante cualquier fallo en la creación del archivo
static File **createTempFile(** String prefijo**,** String sufijo**,** File directorio**)**	Igual que el anterior, pero utiliza el directorio indicado.
void **deleteOnExit()**	Borra el archivo cuando finaliza la ejecución del programa

(10.3) clases para la entrada y la salida

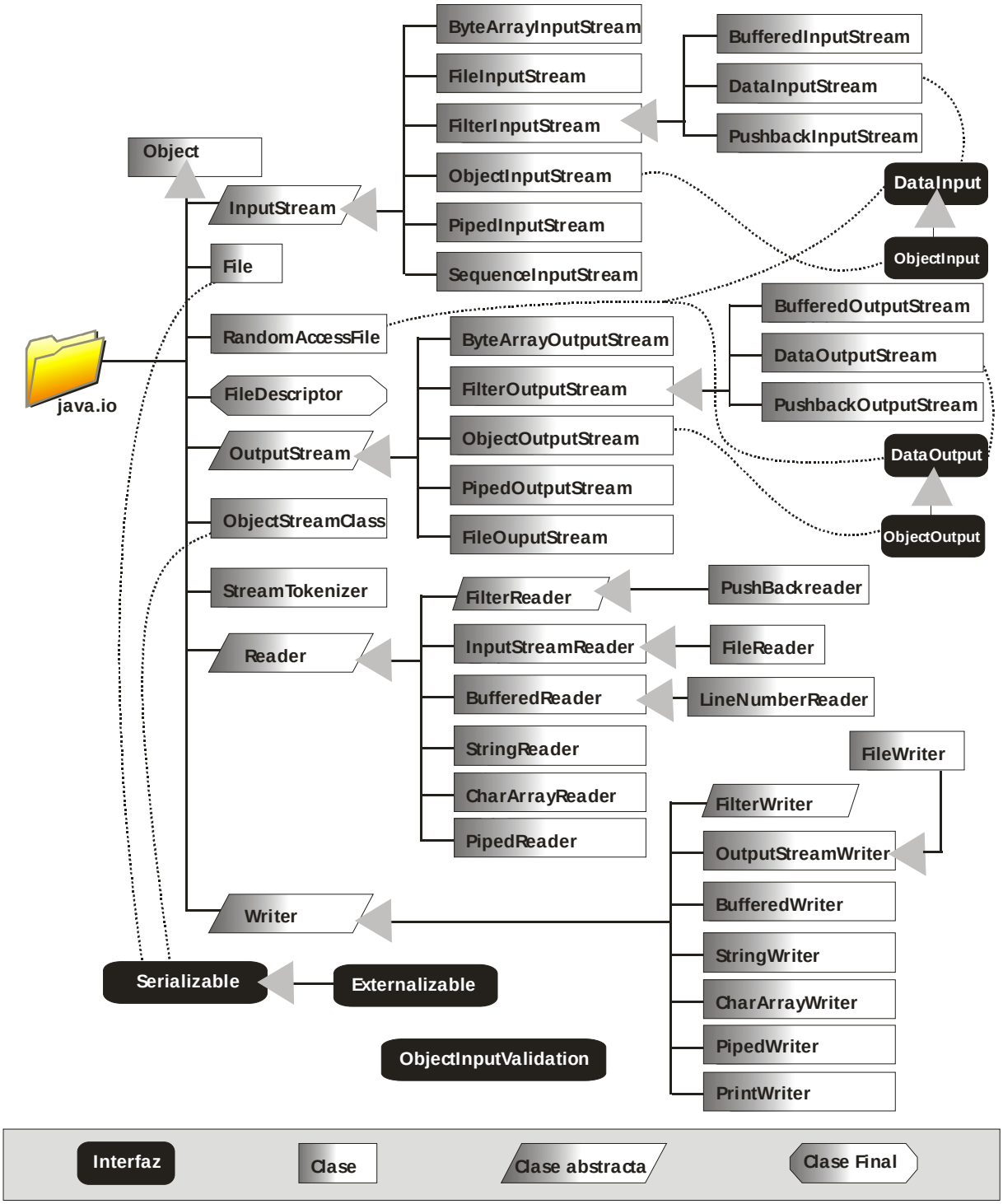


Ilustración 10-1, Clases e interfaces del paquete java.io

Java se basa en las **secuencias de datos** para dar facilidades de entrada y salida. Una secuencia es una corriente de datos entre un emisor y un receptor

de datos en cada extremo. Normalmente las secuencias son de bytes, pero se pueden formatear esos bytes para permitir transmitir cualquier tipo de datos.

Los datos fluyen en serie, byte a byte. Se habla entonces de un **stream** (corriente de datos, o mejor dicho, corriente de bytes). Pero también podemos utilizar streams que transmiten caracteres Java (tipo **char** Unicode, de dos bytes), se habla entonces de un **reader** (si es de lectura) o un **writer** (escritura).

En el caso de las excepciones, todas las que provocan las excepciones de E/S son derivadas de **IOException** o de sus derivadas. Además son habituales ya que la entrada y salida de datos es una operación crítica porque Con lo que la mayoría de operaciones deben ir inmersas en un **try**.

(10.3.1) corrientes de bytes. InputStream/ OutputStream

Los **Streams** de Java son corrientes de datos binarios accesibles byte a byte. Estas dos clases **abstractas**, definen las funciones básicas de lectura y escritura de una secuencia de bytes pura (sin estructurar). Estas corrientes de bits, no representan ni textos ni objetos, sino datos binarios puros. Poseen numerosas subclases; de hecho casi todas las clases preparadas para la lectura y la escritura, derivan de estas.

Los métodos más importantes son **read** (leer) y **write** (escribir), que sirven para leer un byte del dispositivo de entrada o escribir un byte respectivamente.

métodos de InputStream

método	uso
int **available()**	Devuelve el número de bytes de entrada
void **close()**	Cierra la corriente de entrada. Cualquier acceso posterior generaría una **IOException**.
void **mark(**int bytes**)**	Marca la posición actual en el flujo de datos de entrada. Cuando se lea el número de bytes indicado, la marca se elimina.
boolean **markSupported()**	Devuelve verdadero si en la corriente de entrada es posible marcar mediante el método **mark.**
int **read()**	Lee el siguiente byte de la corriente de entrada y le almacena en formato de entero. Devuelve -1 si estamos al final del fichero
int **read(**byte[] búfer**)**	Lee de la corriente de entrada hasta llenar el array *búfer*.
void **reset()**	Coloca el puntero de lectura en la posición marcada con **mark**.
long **skip()**	Se salta de la lectura el número de bytes indicados

métodos de OutputStream

método	uso
void **close()**	Cierra la corriente de salida. Cualquier acceso posterior generaría una **IOException**.
void **flush()**	Vacía los búferes de salida de la corriente de datos
void **write(**int byte**)**	Escribe un byte en la corriente de salida
void **write(**byte[] bufer**)**	Escribe todo el array de bytes en la corriente de salida
void **write(** byte[] buffer**,** **int** posInicial**,** **int** numBytes **)**	Escribe el array de bytes en la salida, pero empezando por la posición inicial y sólo la cantidad indicada por *numBytes*.

(10.3.2) Reader/Writer

Clases abstractas que definen las funciones básicas de escritura y lectura basada en texto Unicode. Se dice que estas clases pertenecen a la jerarquía de lectura/escritura orientada a caracteres, mientras que las anteriores pertenecen a la jerarquía orientada a bytes.

Aparecieron en la versión 1.1 y no substituyen a las anteriores. Siempre que se pueda es más recomendable usar clases que deriven de estas.

Poseen métodos **read** y **write** adaptados para leer arrays de caracteres.

métodos de Reader

método	uso
void **close()**	Cierra la corriente de entrada. Cualquier acceso posterior generaría una **IOException**.
void **mark(**int bytes**)**	Marca la posición actual en el flujo de datos de entrada. Cuando se lea el número de bytes indicado, la marca se elimina.
boolean **markSupported()**	Devuelve verdadero si en la corriente de entrada es posible marcar mediante el método **mark.**
int **read()**	Lee el siguiente byte de la corriente de entrada y le almacena en formato de entero. Devuelve -1 si estamos al final del fichero
int **read(**byte[] búfer**)**	Lee de la corriente de entrada bytes y les almacena en el búfer. Lee hasta llenar el búfer.
int **read(** byte[] bufer, int posInicio, int despl**)**	Lee de la corriente de entrada bytes y les almacena en el búfer. La lectura la almacena en el array pero a partir de la posición indicada, el número máximo de bytes leídos es el tercer parámetro.
boolean **ready()**	Devuelve verdadero si la corriente de entrada está lista.

método	uso
void **reset()**	Coloca el puntero de lectura en la posición marcada con **mark**.
long **skip()**	Se salta de la lectura el número de bytes indicados

métodos de Writer

método	uso
void **close()**	Cierra la corriente de salida. Cualquier acceso posterior generaría una **IOException**.
void **flush()**	Vacía los búferes de salida de la corriente de datos
void **write(**int byte**)**	Escribe un byte en la corriente de salida
void **write(**byte[] bufer**)**	Escribe todo el array de bytes en la corriente de salida
void **write(** byte[] buffer**,** **int** posInicial**,** **int** numBytes **)**	Escribe el array de bytes en la salida, pero empezando por la posición inicial y sólo la cantidad indicada por *numBytes*.
void **write(**String texto**)**	Escribe los caracteres en el **String** en la corriente de salida.
void **write(** **String** buffer**,** **int** posInicial**,** **int** numBytes **)**	Escribe el String en la salida, pero empezando por la posición inicial y sólo la cantidad indicada por *numBytes*.

(10.3.3) InputStreamReader/ OutputStreamWriter

Son clases que sirven para adaptar la entrada y la salida. La razón es que las corrientes básicas de E/S son de tipo **Stream**. Estas clases consiguen adaptarlas a corrientes **Reader/Writer**.

Puesto que derivan de las clases **Reader** y **Writer**, ofrecen los mismos métodos que éstas.

Para ello poseen un constructor que permite crear objetos **InputStreamReader** pasando como parámetro una corriente de tipo **InputStream** y objetos **OutputStreamWriter** partiendo de objetos **OutputStream**.

(10.3.1) DataInputStream/DataOutputStream

Leen corrientes de datos de entrada en forma de byte, pero adaptándola a los tipos simples de datos (**int**, **short**, **byte**,..., **String**). Tienen varios métodos **read** y **write** para leer y escribir datos de todo tipo.

Ambas clases construyen objetos a partir de corrientes **InputStream** y **OutputStream** respectivamente.

métodos de DataInputStream

método	uso
boolean **readBoolean()**	Lee un valor booleano de la corriente de entrada. Puede provocar excepciones de tipo **IOException** o excepciones de tipo **EOFException**, esta última se produce cuando se ha alcanzado el final del archivo y es una excepción derivada de la anterior, por lo que si se capturan ambas, ésta debe ir en un catch anterior (de otro modo, el flujo del programa entraría siempre en la **IOException**).
byte **readByte()**	Idéntica a la anterior, pero obtiene un byte. Las excepciones que produce son las mismas
... **readChar(), readShort(), readLong(), readFloat(), readDouble()**	Como las anteriores pero devolviendo el tipo de datos adecuado
String **readLine()**	Lee de la entrada caracteres hasta llegar a un salto de línea o al fin del fichero y el resultado le obtiene en forma de **String**
String **readUTF()**	Lee un String en formato UTF (codificación norteamericana). Además de las excepciones comentadas antes, puede ocurrir una excepción del tipo **UTFDataFormatException** (derivada de **IOException**) si el formato del texto no está en UTF.

métodos de OutputStreamWriter

La idea es la misma, los métodos son: **writeBoolean, writeByte, writeBytes** (para Strings)**, writeFloat, writeShort, writeUTF, writeInt, writeLong.** Todos poseen un argumento que son los datos a escribir (cuyo tipo debe coincidir con la función).

(10.3.2) ObjectInputStream/ObjectOutputStream

Filtros de secuencia que permiten leer y escribir objetos de una corriente de datos orientada a bytes. Sólo tiene sentido si los datos almacenados son objetos. Tienen los mismos métodos que la anterior, pero aportan un nuevo método de lectura:

- **readObject.** Devuelve un objeto Object de los datos de la entrada. En caso de que no haya un objeto o no sea serializable, da lugar a excepciones. Las excepciones pueden ser:
  - **ClassNotFoundExcepcion**
  - **InvalidClassExcepcion**
  - **StreamCorruptedException**
  - **OptionalDataException**
  - **IOException** genérica.

La clase ObjectOutputStream posee el método de escritura de objetos **writeObject** al que se le pasa el objeto a escribir. Este método podría dar lugar en caso de fallo a excepciones **IOException**, **NotSerializableException** o **InvalidClassException**.

(10.3.3) BufferedInputStream/BufferedOutputStream/ BufferedReader/BufferedWriter

La palabra **buffered** hace referencia a la capacidad de almacenamiento temporal en la lectura y escritura. Los datos se almacenan en una memoria temporal antes de ser realmente leídos o escritos. Se trata de cuatro clases que trabajan con métodos distintos pero que suelen trabajar con las mismas corrientes de entrada que podrán ser de bytes (**Input/OutputStream**) o de caracteres (**Reader/Writer**).

La clase **BufferedReader** aporta el método **readLine** que permite leer caracteres hasta la presencia de **null** o del salto de línea.

(10.3.4) PrintWriter

Clase pensada para secuencias de datos orientados a la impresión de textos. Es una clase escritora de caracteres en flujos de salida, que posee los métodos **print** y **println**, que otorgan gran potencia a la escritura.

(10.3.5) PipedInputStream/PipedOutputStream

Permiten realizar canalizaciones entre la entrada y la salida; es decir lo que se lee se utiliza para una secuencia de escritura o al revés.

(10.4) entrada y salida estándar

(10.4.1) las clases in y out

java.lang.System es una clase que poseen multitud de pequeñas clases relacionadas con la configuración del sistema. Entre ellas están la clase **in** que es un objeto de tipo **InputStream** que representa la entrada estándar (normalmente el teclado) y **out** que es un **OutputStream** que representa a la

salida estándar (normalmente la pantalla). Hay también una clase **err** que representa a la salida estándar para errores. El uso podría ser:

```
InputStream stdin =System.in;
OutputStream stdout=System.out;
```

El método **read()** permite leer un byte. Este método puede lanzar excepciones del tipo **IOException** por lo que debe ser capturada dicha excepción.

```
int valor=0;
try{
   valor=System.in.read();
}
catch(IOException e){
   ...
}
System.out.println(valor);
```

No tiene sentido el listado anterior, ya que **read()** lee sólo un byte de la entrada estándar (del teclado), y lo normal es escribir textos largos; por lo que el método read no es el apropiado. El método **read** puede poseer un argumento que es un array de bytes que almacenará cada carácter leído y devolverá el número de caracteres leído

```
InputStream stdin=System.in;
int n=0;
byte[] caracter=new byte[1024];
try{
   n=System.in.read(caracter);
}
catch(IOException e){
   System.out.println("Error en la lectura");
}
for (int i=0;i<=n;i++)
   System.out.print((char)caracter[i]);
```

El lista anterior lee una serie de bytes y luego los escribe. La lectura almacena el código del carácter leído, por eso hay que hacer una conversión a **char**.

Para saber que tamaño dar al array de bytes, se puede usar el método **available** de la clase **InputStream** la tercera línea del código anterior sería:

```
byte[] carácter=new byte[System.in.available()];
```

(10.4.2) conversión a forma de Reader

El hecho de que las clases **InputStream** y **OutputStream** usen el tipo byte para la lectura, complica mucho su uso. Desde que se impuso Unicode y con él las

clases **Reader** y **Writer**, hubo que resolver el problema de tener que usar las dos anteriores.

La solución fueron dos clases: **InputStreamReader** y **OutputStreamWriter.** Se utilizan para convertir secuencias de byte en secuencias de caracteres según una determinada configuración regional. Permiten construir objetos de este tipo a partir de objetos **InputStream** u **OutputStream.** Puesto que son clases derivadas de **Reader** y **Writer** el problema está solucionado.

El constructor de la clase **InputStreamReader** requiere un objeto **InputStream** y, opcionalmente, una cadena que indique el código que se utilizará para mostrar caracteres (por ejemplo "ISO-8914-1" es el código Latín 1, el utilizado en la configuración regional). Sin usar este segundo parámetro se construye según la codificación actual (es lo normal).

Lo que hemos creado de esa forma es un objeto *convertidor.* De esa forma podemos utilizar la función **read** orientada a caracteres Unicode que permite leer caracteres extendidos. Está función posee una versión que acepta arrays de caracteres, con lo que la versión **writer** del código anterior sería:

```
InputStreamReader stdin=new InputStreamReader(System.in);
char caracter[]=new char[1024];
int numero=-1;
try{
   numero=stdin.read(caracter);
}
catch(IOException e){
   System.out.println("Error en la lectura");
}
for(int i=0;i<numero;i++)
   System.out.print(caracter[i]);
```

(10.4.3) lectura con readLine

El uso del método **read** con un array de caracteres sigue siendo un poco enrevesado, además hay que tener en cuenta que el teclado es un dispositivo con buffer de lectura. Por ello para leer cadenas de caracteres se suele utilizar la clase **BufferedReader**. La razón es que esta clase posee el método **readLine** que permite leer una línea de texto en forma de String, que es más fácil de manipular. Esta clase usa un constructor que acepta objetos **Reader** (y por lo tanto **InputStreamReader**, ya que desciende de ésta) y, opcionalmente, el número de caracteres a leer.

Hay que tener en cuenta que el método **readLine** (como todos los métodos de lectura) puede provocar excepciones de tipo **IOException** por lo que, como ocurría con las otras lecturas, habrá que capturar dicha lectura.

Ejemplo:

```
String texto="";
try{
    //Obtención del objeto Reader
    InputStreamReader conv=new InputStreamReader(System.in);
    //Obtención del BufferedReader
    BufferedReader entrada=new BufferedReader(conv);
    texto=entrada.readLine();
}
catch(IOException e){
    System.out.println("Error");
}
System.out.println(texto);
```

(10.5) lectura y escritura en archivos

(10.5.1) clases FileInputStream y FileOutputStream

Se trata de las clases que manipulan archivos. Son herederas de **Input/OutputStream**, por lo que manejan corrientes de datos en forma de bytes binarios. La diferencia es que se construyen a partir de objetos de tipo **File**.

(10.5.2) lectura y escritura byte a byte de un archivo

Para leer necesitamos un archivo del que dispongamos permisos de escritura y su ruta o bien un objeto File que le haga referencia. Con ello creamos una corriente de tipo **FileInputStream**:

```
FileInputStream fis=new FileInputStream(objetoFile);
FileInputStream fos=new FileInputStream("/textos/texto25.txt");
```

La construcción de objetos **FileOutputStream** se hace igual, pero además se puede indicar un parámetro más de tipo booleano que con valor **true** permite añadir más datos al archivo (normalmente al escribir se borra el contenido del archivo, valor **false**).

Estos constructores intentan abrir el archivo, generando una excepción del tipo **FileNotFoundException** si el archivo no existiera u ocurriera un error en la apertura. Los métodos de lectura y escritura de estas clases son los heredados de las clases **InputStream** y **OutputStream**; fundamentalmente los métodos **read** y **write** son los que permiten leer y escribir. El método **read** devuelve **-1** en caso de llegar al final del archivo.

Este método lee el archivo de forma absolutamente binaria los archivos y sólo es válido cuando deseamos leer toda la información del archivo.

Ejemplo, suponiendo que existe un archivo de texto llamado prueba.txt, este código le muestra por pantalla:

```
File f=new File("prueba.txt");
try {
   FileInputStream fis=new FileInputStream(f);
   int x=0;
   while(x!=-1){
         x=fis.read();
         System.out.print((char)x);
   }
}
catch (IOException e) {
   e.printStackTrace();
}
```

(10.6) lectura y escritura de archivos de texto

Como ocurría con la entrada estándar, se puede convertir un objeto **FileInputStream** o **FileOutputStream** a forma de **Reader** o **Writer** mediante las clases **InputStreamReader** y **OutputStreamWriter**. Y esto es más lógico cuando manejamos archivos de texto.

Existen además dos clases que manejan caracteres en lugar de bytes (lo que hace más cómodo su manejo), son **FileWriter** y **FileReader.**

La construcción de objetos del tipo FileReader se hace con un parámetro que puede ser un objeto File o un String que representarán a un determinado archivo.

La construcción de objetos FileWriter se hace igual sólo que se puede añadir un segundo parámetro booleano que, en caso de valer **true**, indica que se abre el archivo para añadir datos; en caso contrario se abriría para grabar desde cero (se borraría su contenido).

Para escribir se utiliza **write** que es un método void que recibe como parámetro lo que se desea escribir en formato int, String o en forma de array de caracteres.

Por ejemplo este el código de un programa que lee por teclado texto hasta que el usuario deja vacía la línea y todo lo escrito lo vuelca en un archivo llamado **salida.txt**:

```
File f=new File("d:\\salida.txt");
try {
   FileWriter fw=new FileWriter(f);
   BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
   String texto=" ";
   while(texto.length()>0){
         texto=br.readLine();
         fw.write(texto+"\r\n");
   }
   fw.close();
}
catch (IOException e) {
   e.printStackTrace();
}
```

Para leer se utiliza el método **read** que devuelve un **int** y que puede recibir un array de caracteres en el que se almacenarían los caracteres leídos. Ambos métodos pueden provocar excepciones de tipo **IOException**.

No obstante sigue siendo un método todavía muy rudimentario. Por ello lo ideal es convertir el flujo de las clases File en clases de tipo **BufferedReader** y **BufferedWriter** vistas anteriormente . Su uso sería:

```
File f=new File("D:/salida.txt");
int x=105;
try{
   FileReader fr=new FileReader(f);
   BufferedReader br=new BufferedReader(fr);
   String s="";
   br.readLine();
   while(s!=null){
        System.out.println(s);
        s=br.readLine();
   }
}
catch (IOException e) {
   e.printStackTrace();
}
```

Éste código muestra el archivo por pantalla. **readLine** devuelve **null** cuando se llega al final del archivo. Sin embargo es conveniente adelantar la primera lectura antes del bucle para conseguir que no salga por pantalla el propio null (a esto se le llama bucle de lectura adelantada).

La escritura se realiza con el método **write** que permite grabar caracteres, Strings y arrays de caracteres. **BufferedWriter** además permite utilizar el método **newLine** que escriba un salto de línea en el archivo; lo que arregla el problema de la compatibilidad entre plataformas por que los caracteres para el cambio de párrafo son distintos según cada sistema operativo (o incluso por diferentes circunstancias).

(10.7) archivos binarios

Para archivos binarios se suelen utilizar las clases **DataInputStream** y **DataOutputStream**. Estas clases están mucho más preparadas para escribir datos de todo tipo.

(10.7.1) escritura en archivos binarios

El proceso sería:

(1) Crear un objeto **FileOutputStream** a partir de un objeto **File** que posea la ruta al archivo que se desea escribir (para añadir usar el segundo parámetro del constructor indicando **true**)

(2) Crear un objeto **DataOutputStream** asociado al objeto anterior. Esto se realiza mediante el constructor de **DataOutputStream**.

(3) Usar el objeto del punto 2 para escribir los datos mediante los métodos **write***Tipo* donde *tipo* es el tipo de datos a escribir (**int**, **double**, ...). A este método se le pasa como único argumento los datos a escribir.

(4) Se cierra el archivo mediante el método **close** del objeto **DataOutputStream.**

Ejemplo:

```
File f=new File("d:/prueba.out"); Random r=new Random();
double d=18.76353;
try{
  FileOutputStream fis=new FileOutputStream(f);
  DataOutputStream dos=new DataOutputStream(fis);
  for (int i=0;i<234;i++){ //Se repite 233 veces
    dos.writeDouble(r.nextDouble());//Nº aleatorio
  }
  dos.close();
}
catch(FileNotFoundException e){
  System.out.println("No se encontro el archivo");
}
catch(IOException e){
  System.out.println("Error al escribir");
}
```

(10.7.2) lectura en archivos binarios

El proceso es análogo. Sólo que hay que tener en cuenta que al leer se puede alcanzar el final del archivo. Al llegar al final del archivo, se produce una excepción del tipo **EOFException** (que es subclase de **IOException**), por lo que habrá que controlarla.

Ejemplo, leer el contenido del archivo del ejemplo anterior:

```
boolean finArchivo=false;//Para provocar bucle infinito
try{
   FileInputStream fis=new FileInputStream(f);
   DataInputStream dis=new DataInputStream(fis);
   double d;
    while (!finArchivo){
            d=dis.readDouble();
            System.out.println(d);
    }
   dis.close();
}
catch(EOFException e){
   finArchivo=true;
}
catch(FileNotFoundException e){
   System.out.println("No se encontr el archivo");
}
catch(IOException e){
   System.out.println("Error al leer");
}
```

En este listado, obsérvese como el bucle **while** que da lugar a la lectura se ejecuta indefinidamente (no se pone como condición a secas **true** porque casi ningún compilador lo acepta), se saldrá de ese bucle cuando ocurra la excepción **EOFException** que indicará el fin de archivo.

Para manejar adecuadamente los archivos hay que conocer perfectamente su contenido.

(10.8) archivos de acceso aleatorio

Hasta ahora los archivos se están leyendo secuencialmente. Es decir desde el inicio hasta el final. Pero es posible leer datos de una zona concreta del archivo.

Por supuesto esto implica necesariamente dominar la estructura del archivo, pero además permite crear programas muy potentes para manejar archivos de datos binarios.

(10.8.1) RandomAccessFile

Esta clase permite leer archivos en forma aleatoria. Es decir, se permite leer cualquier posición del archivo en cualquier momento. Los archivos anteriores son llamados secuenciales, se leen desde el primer byte hasta el último.

Esta es una clase primitiva que implementa las interfaces **DataInput** y **DataOutput** y sirve para leer y escribir datos.

La construcción requiere de una cadena que contenga una ruta válida a un archivo o de un archivo File. Hay un segundo parámetro obligatorio que se llama **modo**. El modo es una cadena que puede contener una **r** (lectura), **w** (escritura) o ambas, **rw**.

Como ocurría en las clases anteriores, hay que capturar la excepción **FileNotFound** cuando se ejecuta el constructor para el caso de que haya problemas al crear el objeto File.

```
File f=new File("D:/prueba.out");
RandomAccessFile archivo = new RandomAccessFile( f, "rw");
```

Los métodos fundamentales son:

- **void seek(long pos).** Permite colocarse en una posición concreta, contada en bytes, en el archivo. Lo que se coloca es el puntero de acceso que es la señal que marca la posición a leer o escribir.
- **long getFilePointer().** Posición actual del puntero de acceso
- **long length().** Devuelve el tamaño del archivo
- **readBoolean, readByte, readChar, readInt, readDouble, readFloat, readUTF, readLine.** Funciones de lectura, equivalentes a las disponibles en la clase **DataInputStream**. Leen un dato del tipo indicado. En el caso de **readUTF** lee una cadena en formato Unicode.
- **writeBoolean, writeByte, writeBytes, writeChar, writeChars writeInt, writeDouble, writeFloat, writeUTF, writeLine.** Funciones de escritura. Todas reciben como parámetro, el dato a escribir. Escriben encima de lo ya escrito. Para escribir al final hay que colocar el puntero de acceso al final del archivo.

(10.9) serialización

Es una forma automática de guardar y cargar el estado de un objeto. Se basa en la interfaz **Serializable** (en el paquete **java.io**) que es la que permite esta operación. Si una clase implementa esta interfaz puede ser guardado y restaurado directamente en un archivo

Cuando se desea utilizar un objeto para ser almacenado con esta técnica, debe ser incluida la instrucción **implements Serializable** la cabecera de

clase. Esta interfaz no posee métodos, pero es un requisito obligatorio para hacer que un objeto sea serializable.

La clase **ObjectInputStream** y la clase **ObjectOutputStream** se encargan de realizar este procesos. Son las encargadas de escribir o leer el objeto de un archivo. Son herederas de **InputStream** y **OutputStream**, de hecho son casi iguales a **DataInput/OutputStream** sólo que incorporan los métodos **readObject** y **writeObject** que son los que permiten grabar directamente objetos. Ejemplo:

```
try{
 FileInputStream fos=new FileInputStream("d:/nuevo.out");
 ObjectInputStream os=new ObjectInputStream(fos);
 Coche c;
 boolean finalArchivo=false;

 while(!finalArchivo){
    c=(Coche) readObject();//Casting necesario porque devuelve Object
     System.out.println(c);
  }
}
catch(EOFException e){
   System.out.println("Se alcanzó el final");
}
catch(ClassNotFoundException e){
   System.out.println("Error el tipo de objeto no es compatible");
}
catch(FileNotFoundException e){
   System.out.println("No se encontró el archivo");
}
catch(IOException e){
   System.out.println("Error  "+e.getMessage());
   e.printStackTrace();
}
```

El listado anterior podría ser el código de lectura de un archivo que guarda coches. Los métodos **readObject** y **writeObject** usan objetos de tipo Object, readObject les devuelve y writeObject les recibe como parámetro. Ambos métodos lanzan excepciones del tipo **IOException** y **readObject** además lanza excepciones del tipo **ClassNotFoundException**.

Obsérvese en el ejemplo como la excepción **EOFException** ocurre cuando se alcanzó el final del archivo al igual que ocurre con las corrientes binarias de datos