

Java2

incluye Swing, Threads,
programación en red,
JavaBeans, JDBC y
JSP / Servlets

Autor: Jorge Sánchez (www.jorgesanchez.net) año 2004

Basado en el lenguaje Java definido por Sun
(<http://java.sun.com>)

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.

clases para la entrada y la salida

Java se basa en las secuencias para dar facilidades de entrada y salida. Cada secuencia es una corriente de datos con un emisor y un receptor de datos en cada extremo. Todas las clases relacionadas con la entrada y salida de datos están en el paquete **java.io**.

Los datos fluyen en serie, byte a byte. Se habla entonces de un **stream** (corriente de datos, o mejor dicho, corriente de bytes). Hay otro stream que lanza caracteres (tipo char Unicode, de dos bytes), se habla entonces de un reader (si es de lectura) o un writer (escritura).

Los problemas de entrada / salida suelen causar excepciones de tipo **IOException** o de sus derivadas. Con lo que la mayoría de operaciones deben ir inmersas en un **try**.

InputStream/ OutputStream

Clases **abstractas** que definen las funciones básicas de lectura y escritura de una secuencia de bytes pura (sin estructurar). Esas son corrientes de bits, no representan ni textos ni objetos. Poseen numerosas subclases, de hecho casi todas las clases preparadas para la lectura y la escritura, derivan de estas.

Aquí se definen los métodos **read()** (Leer) y **write()** (escribir). Ambos son métodos que trabajan con los datos, byte a byte.

Reader/Writer

Clases **abstractas** que definen las funciones básicas de escritura y lectura basada en caracteres 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.

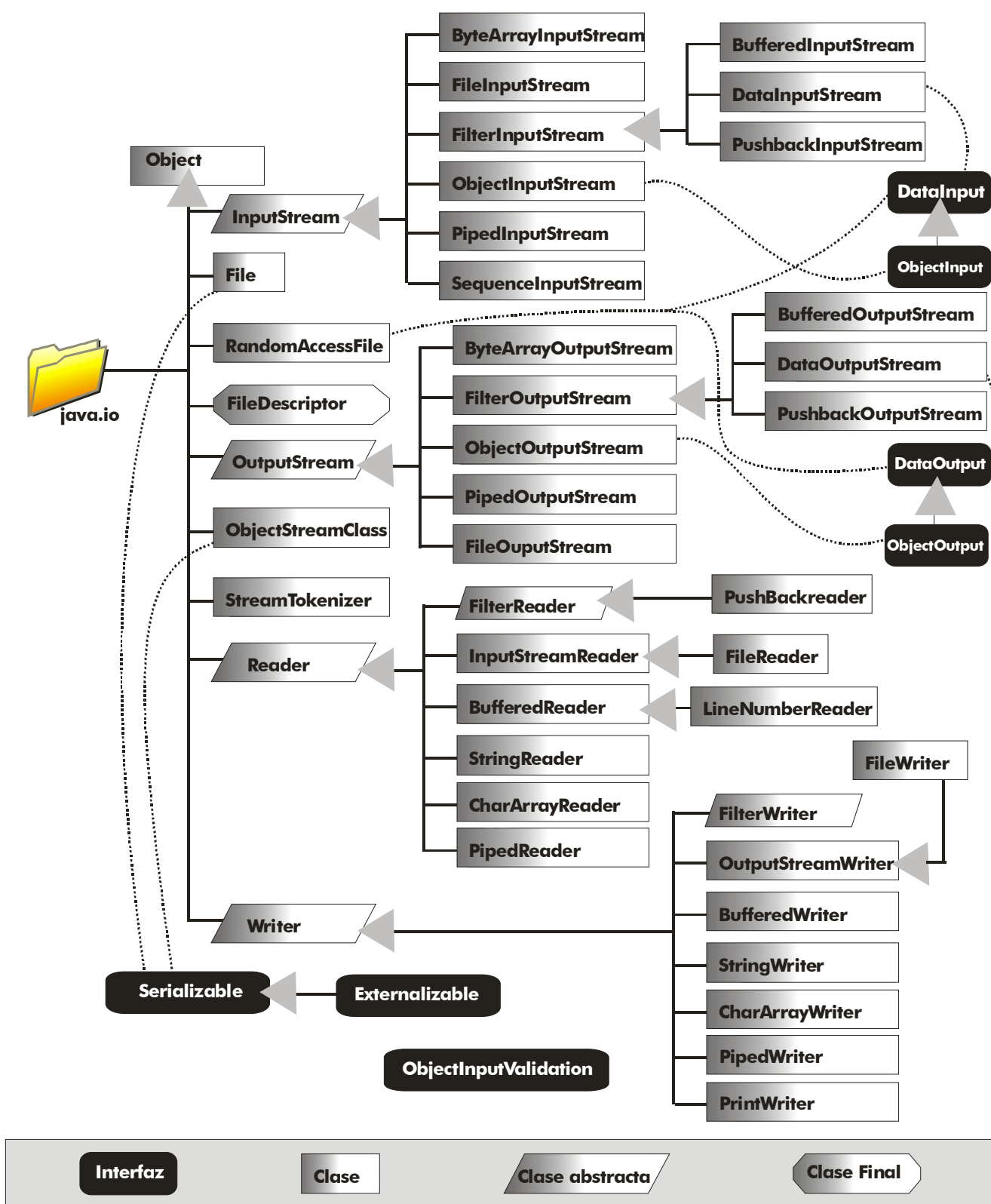
Posee métodos **read** y **write** adaptados para leer arrays de caracteres.

InputStreamReader/ OutputStreamWriter

Son clases que sirven para adaptar la entrada y la salida. El problema está en que las clases anteriores trabajan de forma muy distinta y ambas son necesarias. Por ello **InputStreamReader** convierte una corriente de datos de tipo **InputStream** a forma de **Reader**.

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. En el caso de **DataInputStream** son:

Ilustración 16, Clases e interfaces del paquete `java.io`

- **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**).
- **readByte()**. Idéntica a la anterior, pero obtiene un byte. Las excepciones que produce son las mismas
- **readChar, readShort, readInt, readLong, readFloat, readDouble**. Como las anteriores, pero leen los datos indicados.
- **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.

Por su parte, los métodos de `DataOutputStream` son:

- **writeBoolean, writeByte, writeDouble, writeFloat, writeShort, writeUTF, writeInt, writeLong**. Todos poseen un argumento que son los datos a escribir (cuyo tipo debe coincidir con la función).

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. Aporta 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: **ClassNotFoundException**, **InvalidClassException**, **StreamCorruptedException**, **OptionalDataException** o **IOException** a secas.

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**.

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 clase 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.

PrintWriter

Secuencia pensada para impresión de texto. Es una clase escritora de caracteres en flujos de salida, que posee los métodos **print** y **println** ya comentados anteriormente, que otorgan gran potencia a la escritura.

FileInputStream/FileOutputStream/FileReader/FileWriter

Leen y escriben en archivos (File=Archivo).

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.

entrada y salida estándar

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 **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 un byte de la entrada estándar, y en esta entrada se suelen enviar caracteres, 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];
```

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]);
```

Ficheros

Una aplicación Java puede escribir en un archivo, salvo que se haya restringido su acceso al disco mediante políticas de seguridad. La dificultad de este tipo de operaciones está en que los sistemas de ficheros son distintos en cada sistema y aunque Java intenta aislar la configuración específica de un sistema, no consigue evitarlo del todo.

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,...).

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 archiv1=new File("/datos/bd.txt");  
File carpeta=new File("datos");
```

El primer formato utiliza una ruta absoluta y el segundo una ruta relativa. La ruta absoluta se realiza desde la raíz de la unidad de disco en la que se está trabajando y la relativa cuenta desde la carpeta actual de trabajo.

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 archiv1=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 **URL**.

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 se 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
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.
String separator	Como el anterior pero en forma de String
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 ";"
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.
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.

método	uso
URL toURL() throws MalformedURLException	Convierte el archivo a su notación URL correspondiente
URI toURI()	Convierte el archivo a su notación URI correspondiente

métodos de carpetas

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
boolean createNewFile() Throws IOException	Crea un nuevo archivo basado en la ruta dada al objeto File. Hay que capturar la excepción <i>IOException</i> 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.

método	uso
static File createTempFile(String prefijo, String sufijo)	<p>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.</p> <p>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 IllegalArgumentException</p> <p>Requiere capturar la excepción IOException que se produce ante cualquier fallo en la creación del archivo</p>
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

secuencias de archivo

lectura y escritura byte a byte

Para leer y escribir datos a archivos, Java utiliza dos clases especializadas que leen y escriben orientando a byte (Véase tema anterior); son **FileInputStream** (para la lectura) y **FileOutputStream** (para la escritura).

Se crean objetos de este tipo construyendo con un parámetro que puede ser una ruta o un objeto File:

```
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 segundo parámetro 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**. 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.

Otra posibilidad, más interesante, es utilizar las clases **DataInputStream** y **DataOutputStream**. Estas clases están mucho más preparadas para escribir datos de todo tipo.