



# tecnun

CAMPUS TECNOLÓGICO DE LA UNIVERSIDAD DE NAVARRA  
NAFARROAKO UNIBERTSITATEKO CAMPUS TEKNOLOGIKOA  
Escuela Superior de Ingenieros • Ingeniarien Goi Mailako Eskola

## Aprinda Java

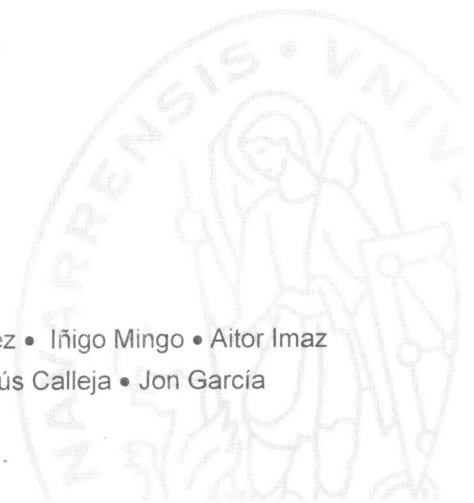
*como si estuviera en primero*



Aprinda Informática ...

San Sebastián, Enero 2000

Javier García de Jalón • José Ignacio Rodríguez • Iñigo Mingo • Aitor Imaz  
Alfonso Brazález • Alberto Larzabal • Jesús Calleja • Jon García



## 9. ENTRADA/SALIDA DE DATOS EN JAVA 1.1

Los programas necesitan comunicarse con su entorno, tanto para recoger datos e información que deben procesar, como para devolver los resultados obtenidos.

La manera de representar estas entradas y salidas en *Java* es a base de *streams* (flujos de datos). Un *stream* es una conexión entre el programa y la fuente o destino de los datos. La información se traslada *en serie* (un carácter a continuación de otro) a través de esta conexión. Esto da lugar a una forma general de representar muchos tipos de comunicaciones.

Por ejemplo, cuando se quiere imprimir algo en pantalla, se hace a través de un *stream* que conecta el monitor al programa. Se da a ese *stream* la orden de escribir algo y éste lo traslada a la pantalla. Este concepto es suficientemente general para representar la lectura/escritura de archivos, la comunicación a través de Internet o la lectura de la información de un sensor a través del puerto en serie.

### 9.1 CLASES DE JAVA PARA LECTURA Y ESCRITURA DE DATOS

El package *java.io* contiene las clases necesarias para la comunicación del programa con el exterior. Dentro de este package existen dos familias de jerarquías distintas para la entrada/salida de datos. La diferencia principal consiste en que una opera con *bytes* y la otra con *caracteres* (el carácter de *Java* está formado por dos bytes porque sigue el código *Unicode*). En general, para el mismo fin hay dos clases que manejan bytes (una clase de entrada y otra de salida) y otras dos que manejan caracteres.

Desde *Java 1.0*, la entrada y salida de datos del programa se podía hacer con clases derivadas de *InputStream* (para lectura) y *OutputStream* (para escritura). Estas clases tienen los métodos básicos *read()* y *write()* que manejan *bytes* y que no se suelen utilizar directamente. La Figura 9.1 muestra las clases que derivan de *InputStream* y la Figura 9.2 las que derivan de *OutputStream*.

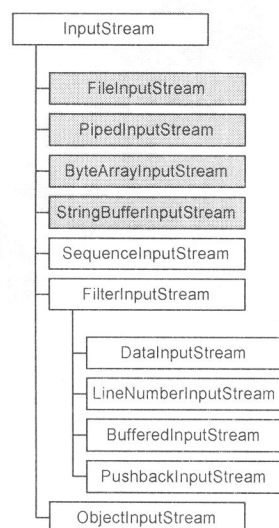


Figura 9.1. Jerarquía de clases *InputStream*.

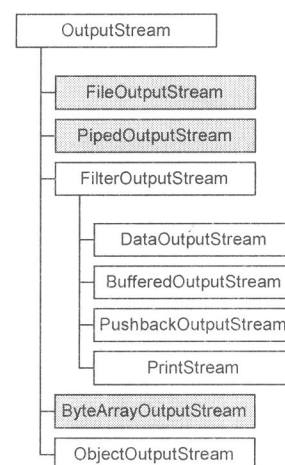


Figura 9.2. Jerarquía de clases *OutputStream*.

En *Java 1.1* aparecieron dos nuevas familias de clases, derivadas de *Reader* y *Writer*, que manejan *caracteres* en vez de *bytes*. Estas clases resultan más prácticas para las aplicaciones en las que se maneja texto. Las clases que heredan de *Reader* están incluidas en la Figura 9.3 y las que heredan de *Writer* en la Figura 9.4.

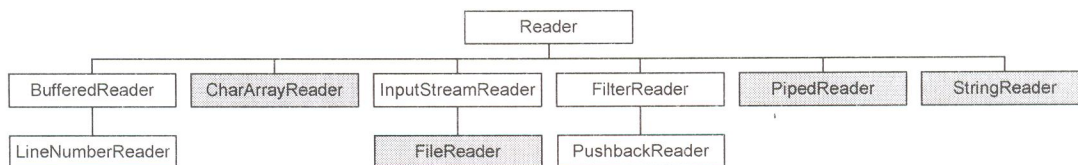


Figura 9.3. Jerarquía de clases Reader.

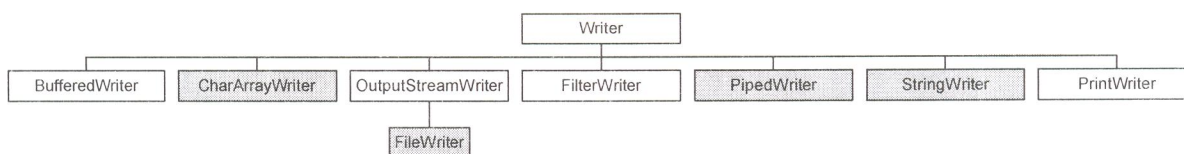


Figura 9.4. Jerarquía de clases Writer.

En las cuatro últimas figuras las clases con *fondo gris* definen de dónde o a dónde se están enviando los datos, es decir, el dispositivo con que conecta el *stream*. Las demás (*fondo blanco*) añaden características particulares a la forma de enviarlos. La intención es que se combinen para obtener el comportamiento deseado. Por ejemplo:

```
BufferedReader in = new BufferedReader(new FileReader("autoexec.bat"));
```

Con esta línea se ha creado un *stream* que permite leer del archivo *autoexec.bat*. Además, se ha creado a partir de él un objeto *BufferedReader* (que aporta la característica de utilizar *buffer*<sup>6</sup>). Los caracteres que lleguen a través del *FileReader* pasarán a través del *BufferedReader*, es decir utilizarán el *buffer*.

A la hora de definir una comunicación con un dispositivo siempre se comenzará determinando el origen o destino de la comunicación (*clases en gris*) y luego se le añadirán otras características (*clases en blanco*).

Se recomienda utilizar siempre que sea posible las clases *Reader* y *Writer*, dejando las de *Java 1.0* para cuando sean imprescindibles. Algunas tareas como la *serialización* y la *compresión* necesitan las clases *InputStream* y *OutputStream*.

### 9.1.1 Los nombres de las clases de java.io

Las clases de *java.io* siguen una nomenclatura sistemática que permite deducir su función a partir de las palabras que componen el nombre, tal como se describe en la Tabla 9.1.

<sup>6</sup> Un *buffer* es un espacio de memoria intermedia que actúa de "colchón" de datos. Cuando se necesita un dato del disco se trae a memoria ese dato y sus datos contiguos, de modo que la siguiente vez que se necesite algo del disco la probabilidad de que esté ya en memoria sea muy alta. Algo similar se hace para escritura, intentando realizar en una sola operación de escritura física varias sentencias individuales de escritura.

Palabra	Significado
InputStream, OutputStream	Lectura/Escritura de bytes
Reader, Writer	Lectura/Escritura de caracteres
File	Archivos
String, CharArray, ByteArray, StringBuffer	Memoria (a través del tipo primitivo indicado)
Piped	Tubo de datos
Buffered	Buffer
Filter	Filtro
Data	Intercambio de datos en formato propio de Java
Object	Persistencia de objetos
Print	Imprimir

Tabla 9.1. Palabras identificativas de las clases de java.io.

### 9.1.2 Clases que indican el origen o destino de los datos

La Tabla 9.2 explica el uso de las clases que definen el lugar con que conecta el *stream*.

Clases	Función que realizan
FileReader, FileWriter, FileInputStream y FileOutputStream	Son las clases que leen y escriben en <b>archivos</b> de disco. Se explicarán luego con más detalle.
StringReader, StringWriter, CharArrayReader, CharArrayWriter, ByteArrayInputStream, ByteArrayOutputStream, StringBufferInputStream	Estas clases tienen en común que se comunican con la <b>memoria</b> del ordenador. En vez de acceder del modo habitual al contenido de un String, por ejemplo, lo leen como si llegara carácter a carácter. Son útiles cuando se busca un modo general e idéntico de tratar con todos los dispositivos que maneja un programa.
PipedReader, PipedWriter, PipedInputStream, PipedOutputStream	Se utilizan como un “tubo” o conexión bilateral para transmisión de datos. Por ejemplo, en un programa con dos threads pueden permitir la comunicación entre ellos. Un thread tiene el objeto PipedReader y el otro el PipedWriter. Si los streams están conectados, lo que se escriba en el PipedWriter queda disponible para que se lea del PipedReader. También puede comunicar a dos programas distintos.

Tabla 9.2. Clases que indican el origen o destino de los datos.



### 9.1.3 Clases que añaden características

La Tabla 9.3 explica las funciones de las clases que alteran el comportamiento de un *stream* ya definido.

Clases	Función que realizan
BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream	Como ya se ha dicho, añaden un <b>buffer</b> al manejo de los datos. Es decir, se reducen las operaciones directas sobre el dispositivo (lecturas de disco, comunicaciones por red), para hacer más eficiente su uso. <b>BufferedReader</b> por ejemplo tiene el método <b>readLine()</b> que lee una línea y la devuelve como un String.
InputStreamReader, OutputStreamWriter	Son clases puente que permiten <b>convertir</b> streams que utilizan bytes en otros que manejan caracteres. Son la única relación entre ambas jerarquías y no existen clases que realicen la transformación inversa.
ObjectInputStream, ObjectOutputStream	Pertencen al mecanismo de la <b>serialización</b> y se explicarán más adelante.
FilterReader, FilterWriter, FilterInputStream, FilterOutputStream	Son clases base para aplicar diversos <b>filtros</b> o procesos al stream de datos. También se podrían extender para conseguir comportamientos a medida.
DataInputStream, DataOutputStream	Se utilizan para escribir y leer datos directamente en los formatos propios de Java. Los convierten en algo ilegible, pero independiente de plataforma y se usan por tanto para <b>almacenaje</b> o para <b>transmisiones</b> entre ordenadores de distinto funcionamiento.
PrintWriter, PrintStream	Tienen métodos adaptados para <b>imprimir</b> las variables de Java con la apariencia normal. A partir de un boolean escriben "true" o "false", colocan la coma de un número decimal, etc.

Tabla 9.3. Clases que añaden características.

## 9.2 ENTRADA Y SALIDA ESTÁNDAR (TECLADO Y PANTALLA)

En *Java*, la entrada desde teclado y la salida a pantalla están reguladas a través de la clase **System**. Esta clase pertenece al package *java.lang* y agrupa diversos métodos y objetos que tienen relación con el sistema local. Contiene, entre otros, tres objetos *static* que son:

**System.in:** Objeto de la clase **InputStream** preparado para recibir datos desde la entrada estándar del sistema (habitualmente el teclado).

**System.out:** Objeto de la clase **PrintStream** que imprimirá los datos en la salida estándar del sistema (normalmente asociado con la pantalla).

**System.err:** Objeto de la clase **PrintStream**. Utilizado para mensajes de error que salen también por pantalla por defecto.

Estas clases permiten la comunicación alfanumérica con el programa a través de los métodos incluidos en la Tabla 9.4. Son métodos que permiten la entrada/salida a un nivel muy elemental.