

Este resumen ha sido elaborado para ser leído después de haber asistido a clase: pretende fijar y profundizar en los conceptos ya explicados.

Streams y manejo de entrada y salida (Input & Output)

Introducción

Streams

Los lenguajes de programación usan con frecuencia el término *stream* (que puede traducirse como 'flujo') como una abstracción que representa cualquier fuente o consumidor de datos como un objeto capaz de leer o escribir información.

Esta abstracción, aunque a primera vista parece complicar las cosas, resulta muy útil, pues permite usar el mismo conjunto de clases y métodos para leer y escribir a estas fuentes o consumidores de datos.

Las librería de clases de Java para manejar *streams* se pueden agrupar a) según sirven para leer o escribir a un *stream*; y, b) según el tipo de dato que leen o escriben (bytes o caracteres). Las clases base de la librería son `InputStream` y `Reader`, y `OutputStream` y `Writer`, como se resume en la tabla.

	Bytes	Caracteres
Leen de un <i>stream</i>	<code>InputStream</code>	<code>Reader</code>
Escriben a un <i>stream</i>	<code>OutputStream</code>	<code>Writer</code>

Usando herencia (*inheritance*), toda las clases que derivan de `InputStream` o de `Reader` tienen un método básico `read()` (que permite leer un byte¹ o caracter de un *stream*), y todas las clases que derivan de `OutputStream` o de `Writer` tienen un método `write()` (que permite escribir un byte o caracter a un *stream*).

Streams de bytes y streams de caracteres

Para entender la diferencia entre entre *streams* de bytes y *streams* de caracteres, necesitamos entender la diferencia entre un byte² y un caracter.

-
- 1 El método `read()` tiene también una versión sobrecargada que permite leer un arreglo de bytes, pero para no complicar la explicación no lo tendremos en cuenta en el resumen.
 - 2 Un *byte* equivale a 8 bits (bit es un número binario, que puede ser 0 ó 1). Por tanto, un byte puede representar $2^8 = 256$ valores distintos (0-255).

Es muy frecuente utilizar números en base 16 (hexadecimal o hex, para abreviar) en vez de números en base 10 para representar bytes. La razón es que un dígito en base 16 puede representar los primeros 4 bits del byte, y otro dígito en base 16 puede representar los otro cuatro bits.

Los dígitos en base 16 son 0, 1, 2, ..., 9, A, B, C, ..., F (A=10, B=11, ..., F=16). Así, un byte de valor 16 decimal se puede representar como 0F en hex, que en binario es 0000 1111 (los cuatro '1' son la F). Se suele anteponer un 0x, o una H, para indicar que el número está en base 16: 0x0F. Un valor bastante usado: 255 = 0xFF.

Los bytes se usan para representar la información tal como están en el dispositivo en el que se almacenan (por ejemplo, un disco duro o en memoria RAM). En cambio, un caracter es la representación de una letra del alfabeto o de un signo. Cuando uno almacena un caracter en un archivo (por ejemplo, la letra 'A'), finalmente usaremos bytes. Pero hace falta un estándar que diga que valor es el que representa la letra 'A'.

Un estándar bastante común es el ASCII³. En ASCII, la letra 'A' se representa con un sólo byte de valor 65⁴. En otras palabras, el estándar ASCII es una tabla de equivalencias entre cada letra (caracter) y su valor. Pero no es el único estándar para representar caracteres. Por ejemplo, Java usa Unicode para sus String y caracteres⁵, las páginas web suelen usar UTF-8 (una versión de Unicode) o ISO-8859-15 (una variante del ASCII que incluye el símbolo para el Euro, €).

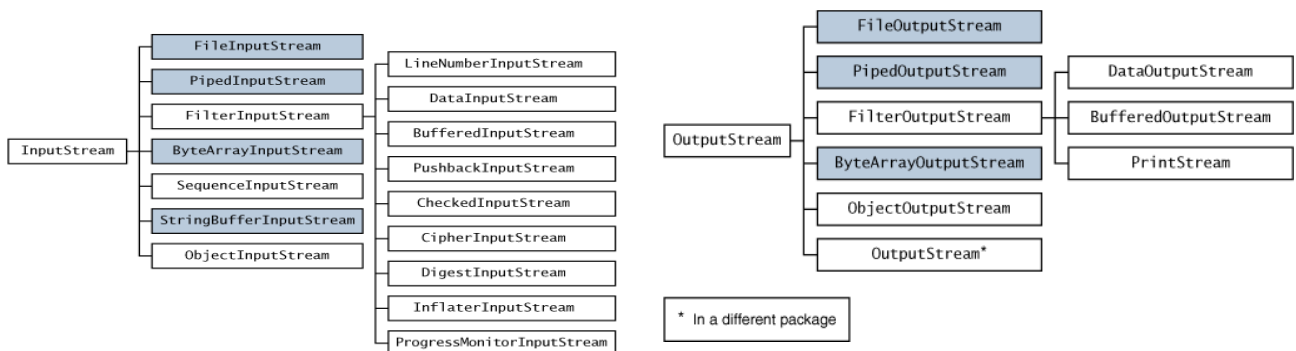
Muchas veces nuestra información estará en caracteres, y podremos escribirla o leerla así del archivo (por ejemplo, un archivo de texto, una página web en HTML, un archivo que contiene la fuente de nuestro programa): en este caso, todos los bytes del archivo se usan para representar caracteres ('letras', en no-técnico).

En otros casos, los bytes del archivo no representan caracteres: una imagen, un programa compilado (un archivo .class, o .exe), un archivo Word (tiene caracteres, pero también tiene otra información que indica el formato, tipo de *font*, etc.). Si leemos estos archivos usando caracteres, obtendríamos unos datos ininteligibles.

Bien, ahora reemplazamos en los párrafos anteriores la palabra 'archivo' por '*stream*'⁶. Java tiene dos árboles de clases distintos: uno que trabajan con streams de bytes (InputStream y OutputStream) y otro que trabaja con caracteres (Reader y Writer).

Los siguientes gráficos, tomados del tutorial de Sun sobre io, muestran la jerarquía de clases⁷:

InputStream y OutputStream (manejan bytes)



3 American Standard Code for Information Interchange. Se debería “un estándar común es el ASCII”, y no “un estándar común es el código ASCII”, del mismo modo que no se dice “Queso Philadelphia Cream Cheese”, ni “Centro Comercial Jockey Plaza Shopping Center”, pero en la práctica hasta en inglés se usa “ASCII Code”.

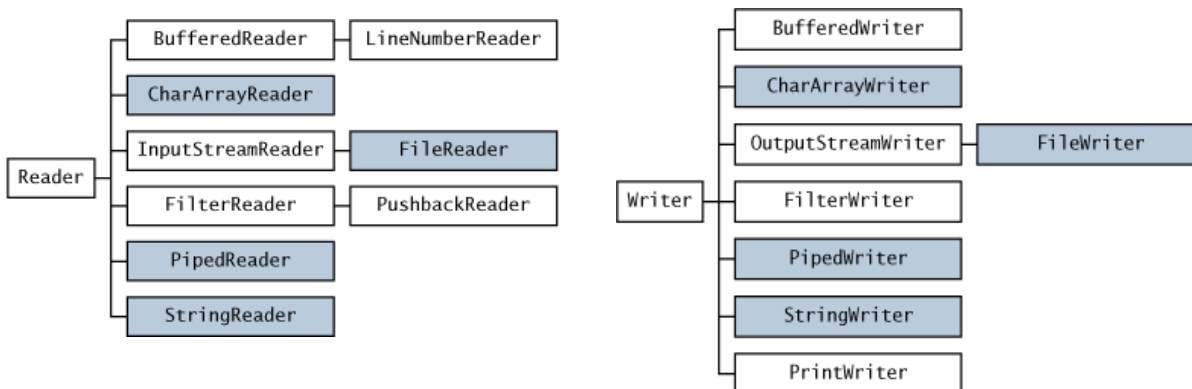
4 La A mayúscula. De modo similar, la B=66, C=67, etc.

5 Como un byte sólo puede representar 256 valores distintos, ASCII está limitado y no puede usarse para muchos caracteres de lenguas no latinas. En cambio, el estándar usa dos bytes para cada caracter, de modo que puede representar 65536 caracteres distintos.

6 Un archivo es un tipo particular de *stream*... una subclase de *stream*.

7 El por qué unos recuadros están sombreados y otros no se entenderá más adelante.

Reader y Writer (manejan caracteres)



Como se puede ver en los gráficos, los nombres de las subclases se han escogido de modo que den a entender de cuál es la superclase y la función específica de la subclase: `FileInputStream` es un subclase de `InputStream` que lee de un file.

Uso de las clases de io

Las superclases de cada uno de los árboles de clases de las figuras de arriba definen pocos métodos, que son heredados por todas las subclases. Las clases que leen de *streams* implementan el método `read()`, y las clases que escriben a *streams* el método `write()`, que leen o escriben un byte o un carácter a la vez.

La clase `InputStream`⁸ tiene la siguiente definición:

```
public abstract int read() throws IOException
```

Podemos deducir:

a) es un método abstract, de modo que no podemos usar la clase directamente. Tendremos que usar necesariamente alguna subclase (de la librería de Java, o definida directamente por nosotros);

b) Aunque lee bytes, devuelve el valor leído en un `int`. La documentación también dice (no lo copiamos aquí) que si ya no hay más bytes que leer del stream el valor devuelto es -1. Esto nos será útil como condición de fin en nuestros programas.

c) Arroja una `IOException` en caso de error: cuando usemos `read()`, tendremos que ponerlo dentro de un bloque `try/catch`, o decir que nuestro método arroja una excepción.

`InputStream` también define un método `close()`, sin parámetros, que es importante llamar cuando hayamos terminado de trabajar con el *stream*⁹.

`OutputStream` define el método `write()` del siguiente modo:

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

donde `int b` contiene el byte que se escribirá. A diferencia de `read()`, no devuelve ningún valor.

Las subclases de las clases base de io hacen un *override* de estos métodos, e implementan

⁸ `InputStream`, repetimos, lee bytes de un *stream*.

⁹ Sobre todo si estamos escribiendo a un *stream*, pues muchas veces los datos no se escriben inmediatamente al llamar a `write()`. Si no se llama a `close()`, cabe el riesgo de que se pierda información.

comportamientos especializados. Por ejemplo, como ya hemos dicho, la clase `FileInputStream` y `FileOutputStream` leen y escriben a archivos. Además, algunas veces implementan alguna funcionalidad adicional.

Java define un buen número de clases de io, y eso hace que su manejo a veces sea confuso. Pero su uso no es complicado, si se sabe cómo se relacionan las clases entre sí. Para eso, es **esencial** fijarse en tres cosas:

- a) qué tipo de objeto espera un método o un constructor como parámetro;
- b) qué tipo de objeto devuelve un método. Los constructores, ya sabemos, devuelven objetos del tipo de la clase que están construyendo¹⁰.
- c) las subclasses son también clases del tipo de su superclase¹¹. Para combinar dos clases de io, nos fijamos si son compatibles sus superclases.

Por ejemplo: queremos escribir un método que lea bytes de un archivo. Usaremos un `FileInputStream`, porque `InputStream` lee bytes, y `File` porque nuestro stream será un archivo en este caso¹².

`FileInputStream` tiene varios constructores:

Constructor Summary	
<code>FileInputStream(File file)</code>	Creates a <code>FileInputStream</code> by opening a connection to an actual file, the file named by the <code>File</code> object <code>file</code> in the file system.
<code>FileInputStream(FileDescriptor fdObj)</code>	Creates a <code>FileInputStream</code> by using the file descriptor <code>fdObj</code> , which represents an existing connection to an actual file in the file system.
<code>FileInputStream(String name)</code>	Creates a <code>FileInputStream</code> by opening a connection to an actual file, the file named by the path name <code>name</code> in the file system.

Nos interesan sobre todo el primer constructor (que espera un objeto de tipo `File`¹³ como archivo) y el último (que espera `String` con el nombre del archivo).

El código podría ser algo así:

```
public void leerArchivo( ) {
    abrir un stream de lectura al archivo (FileInputStream)
    mientras queden bytes en el stream
        leer byte
        hacer algo con el byte
    cerrar stream
}
```

En Java:

¹⁰ Es decir, el constructor de `FilterInputStream` devuelve un `FilterInputStream`.
`FilterInputStream in = new FilterInputStream(...);`

¹¹ Todos los gatos son animales, pero no todos los animales son gatos. Así, todos los objetos que derivan de `FilterInputStream` son también `InputStream`.

¹² Por si alguien lo hubiera pensado: la composición de nombres (`File + InputStream`) no es una expresión de Java ni nada parecido. Es simplemente un nombre **convencional**, `FileInputStream` podría llamarse `LeeArchivoDeBytes`, y al compilador le da igual. En Java, para definir `FileInputStream`, la librería usa algo parecido a `class FileInputStream extends InputStream { ... }`.

¹³ Esta clase, como se explicó en clase, permite abstraer la ruta de un archivo y hacer una serie de operaciones sobre él (verificar si existe, etc.). Se puede ver la documentación de la clase `File` en <http://java.sun.com/j2se/1.5.0/docs/api/java/io/File.html>

```

public void leerArchivo( ) {
    int b;    // nuestro byte
    FileInputStream in = new FileInputStream("archivo");
    while ( ( b = in.read( ) ) != -1 )14 {
        // hacer algo con b
    }
    in.close();
}

```

El código de arriba estaría bien, si no fuera por el hecho de que tanto el constructor de `FileInputStream()` como `read()` pueden arrojar excepciones (`IOExceptions`), y por tanto el código no compila como está. Tenemos dos caminos:

a) el fácil, declarar que `leerArchivo` arroja una excepción, y que se encargue de manejarla el método que nos llamó... ¿cómo se le ocurre llamarnos si el archivo no existe?

b) capturar las excepciones dentro de nuestro método.

Vamos a adoptar un camino combinado: capturaremos las excepciones, pero sólo para añadir un poco de palabreo al mensaje y volverla a lanzar al método que nos llamó.

Para eso, primero retocamos la definición de nuestro método:

```

public void leerArchivo( ) throws IOException { ... }

```

También tendremos que encerrar tanto `new FileInputStream` como `in.read()` en un área resguardada `try/catch`. Podríamos usar un solo bloque `try/catch` para todo, pero usaremos un bloque para cada uno, para poder crear excepciones con mensajes apropiados en cada caso.

El método quedaría así:

```

import java.io.*;
public void leerArchivo( ) throws IOException {
    int b;    // nuestro byte
    FileInputStream in15;

    try {
        in = new FileInputStream("archivo");
    } catch (IOException e) {
        throw new IOException("No puedo abrir el archivo! " +
                               e.getMessage( ))16;
    } // end catch

    try {
        while ( ( b = in.read( ) ) != -1 )17 {
            // hacer algo con b

```

¹⁴ Por si no se entiende la expresión `(b = in.read()) != -1)`:

El compilador primero evaluará `in.read()`. Si hay un byte que leer, nos devolverá `in.read()` devolverá un entero mayor o igual a cero; si no hay bytes que leer, devolverá `-1`. El byte o el `-1` se guarda, obviamente, en `b`, y la expresión se simplifica así: `((b) != -1)`. De modo que el `while` seguirá corriendo mientras `in.read()` no devuelva `-1`, o lo que es lo mismo, mientras hayan bytes en el stream.

¹⁵ Definimos la variable `in` fuera del bloque `try/catch`, porque todas las variables que se definen dentro de las dos `{ }` dejan de existir al salir del bloque. Nosotros necesitamos `in` para leer el archivo. Ése es también el motivo por el que puedo definir nuevamente la misma variable `e` para cada bloque de excepciones.

¹⁶ `e.getMessage` da el mensaje de la excepción anterior, que es el mensaje por defecto de `IOException`. Simplemente lo añadimos al mensaje de la nueva excepción que estamos creando.

¹⁷ Por si no se entiende la expresión `(b = in.read()) != -1)`:

El compilador primero evaluará `in.read()`. Si hay un byte que leer, nos devolverá `in.read()` devolverá un entero mayor o igual a cero; si no hay bytes que leer, devolverá `-1`. El byte o el `-1` se guarda, obviamente, en `b`, y la expresión se simplifica así: `((b) != -1)`. De modo que el `while` seguirá corriendo mientras `in.read()` no devuelva `-1`, o lo que es lo mismo, mientras hayan bytes en el stream.

```

    }
    } catch (IOException e) {
        in.close( ); // cierro el stream
        throw new IOException("Ha ocurrido un error al leer el
archivo! " + e.getMessage( ));
    }
    in.close();
}

```

También se podría haber usado el constructor de `FileInputStream` que espera un objeto `File` como parámetro. Habría que decir:

```

try {
    in = new FileInputStream( new File("archivo") );
}

```

No necesitamos cambiar ninguna línea del resto del código. Por supuesto, el método sería más útil si el nombre del archivo fuera un parámetro `String` del método.

Cuándo usar streams de bytes y cuándo usar streams de caracteres

La regla es muy sencilla: *siempre que se pueda, usar stream de caracteres*. *Streams* de caracteres son, por ejemplo, un archivo con el código fuente de un programa (.java), una página web, los *feeds* de un blog, etc. *Streams* de bytes son un programa compilado (.class, .exe), archivos de imágenes (.jpg), audio (.wav, .mp3, .ogg), video (.mp4, .ogg, .mov)¹⁸.

Las clases que leen caracteres, `Reader` y `Writer`, tienen métodos `read()` y `write()` que funcionan igual que los de `InputStream` y `OutputStream`. De modo que si quisiéramos cambiar nuestro ejemplo anterior para que lea de un archivo de caracteres, bastaría cambiar `FileInputStream` por `FileReader`, sin tocar ninguna otra línea¹⁹:

```

FileReader in;
try {
    in = new FileReader("archivo");
}

```

Decorators

Fijémonos en los constructores de algunas de las subclases de `Reader` y `Writer`:

`BufferedReader` es una subclase de `Reader` que implementa el método `readLine()`²⁰, que lee de línea en línea en vez de carácter por carácter²¹. Su constructor está definido

18 Como veremos párrafos más abajo, la entrada del teclado (`System.in`) y la salida a la consola (`System.out`) son streams de bytes, no de caracteres.

19 Cabe que nos preguntemos: ¿y qué pasa que si intento leer un archivo mp3 con un stream de caracteres? Bueno, es cuestión de hacer la prueba. Sucederán varias cosas: a) los bytes serán interpretados como caracteres, y en Java cada carácter pueden ser 2 bytes; b) No me servirá de nada la información leída ¿para qué me servirá representar como caracteres (letras) lo que no son caracteres? Si lo trato de imprimir con `System.out.println`, probablemente la pantalla se llenará de garabatos. Pero es cuestión de hacer la prueba.

20 `readLine()` está definido así: `public String readLine() throws IOException`. Devuelve un `String` con la línea leída, o `null` si no hay más líneas que leer. El típico loop se escribiría así:

```

while ( ( s = in.readLine( ) ) != null ) {
    // hacer algo
}

```

Se entiende que habría que manejar adecuadamente la `IOException` que podría generarse.

21 Leer un carácter de un stream, dejar pasar un tiempo, y luego leer otro, es ineficiente. Siempre que se pueda, es

así:

```
public BufferedReader(Reader in)
```

Lo interesante aquí es que **BufferedReader sólo tiene constructores con parámetros de tipo Reader**. ¿Cómo podríamos leer, por ejemplo, un archivo? No hay ningún **BufferedReader(File file)** definido.

La solución es pasar como parámetro de **BufferedReader** un objeto de tipo **Reader** que sepa cómo leer un stream de un archivo. Esa clase es **FileReader**.

FileReader tiene los siguientes constructores:

```
FileReader( String fileName )  
FileReader( File file )
```

Podemos usar cualquiera de los dos. Escojamos el que acepta el nombre del archivo como un **String**.

```
new FileReader22( String fileName )
```



```
BufferedReader in = new BufferedReader( )
```

El constructor de **FileReader** devuelve un objeto **FileReader** que, no olvidemos, es también de tipo **Reader**. Por tanto, puede ser usado como parámetro de **BufferedReader**.

Ahora que hemos visto cómo se usa **BufferedReader**, comentemos algunas cosas que están pasando. ¿Qué hace realmente **BufferedReader**? **BufferedReader** añade a cualquier objeto de tipo **Reader** que se le pase como parámetro una funcionalidad específica: la capacidad de leer del stream de línea en línea, en vez de carácter en carácter²³. Este nuevo *comportamiento* se concreta en el método **readLine()**.

Este “diseño” responde a un *pattern*²⁴ llamado **Decorator**²⁵. Un *decorator* pretende añadir

mejor leer un bloque de datos y luego procesarlos. Es lo que hace la clase **BufferedReader**: implementa un buffer (una memoria de almacenamiento intermedio) entre el *stream* y nosotros, y lee la información por bloques. Nosotros podemos manipularla de modo transparente usando **read()** y **readLine()** sin que se deteriore el rendimiento, pues en realidad los datos ya fueron leídos antes de que nosotros los pidamos por el **BufferedReader**.

El mismo razonamiento se aplica a **BufferedWriter**: es más eficiente escribir los datos por paquetes, que escribir carácter por carácter.

22 Recordemos que **new FileReader(...)** devuelve un objeto de tipo **Reader**.

23 Además, como hemos dicho, hace de almacenamiento intermedio, etc., etc. Pero lo ponemos entre paréntesis para no enredar la explicación.

24 Un *pattern* (patrón) describe un problema que ocurre una y otra vez, y luego describe el núcleo de la solución de fondo al problema, de un modo tal que se puede usar la misma solución mil veces, aunque no se implemente siempre igual. Dicho de otro modo, un *pattern* describe un problema común (en este caso, un problema de diseño de software) y luego explica el fondo de la solución, para que nosotros veamos cómo implementarla.

En la programación orientada a objetos, es frecuente encontrar problemas recurrentes. Los buenos diseñadores y programadores saben identificar estos patrones y resolver el problema con una solución ya probada y que han usado frecuentemente.

El libro clásico que recoge muchos *patterns* de aplicables al diseño de software es *Design Patterns. Elements of Reusable Object-Oriented Software*. ERICH GAMMA, RICHARD HELM, RALPH JOHNSON Y JOHN VLISSIDES. No es el libro, sino el libro. La primera edición es de 1977, y la número 20 de mayo del 2000. A los autores de este libro se les conoce como *The Gang of Four*.

25 cfr. ERICH GAMMA ET AL., *op. cit.*, p. 175 y ss.

responsabilidades adicionales a un objeto de modo dinámico²⁶. No estamos extendiendo la funcionalidad de la clase, sino de un objeto que ha sido creado durante la ejecución del programa. **Los *decorators* permiten extender la funcionalidad de una clase sin tener que definir un subclase que implemente esa funcionalidad.**

Por eso, para que `FileReader` pueda leer líneas en vez de sólo caracteres, no hemos derivado una subclase de `FileReader` usando `extends`, sino que hemos usado este pattern. La librería de io de Java espera que varias de sus clases se usen de este modo.

`PrintWriter` es una subclase interesante de `Writer`²⁷. Decora un objeto `Writer` con los métodos `print`, `println` y `printf`²⁸ (entre otros).

Algunos constructores de `PrintWriter` son:

```
public PrintWriter(Writer out)
public PrintWriter(OutputStream out)
public PrintWriter(String fileName) throws FileNotFoundException
public PrintWriter(File file) throws FileNotFoundException
```

Como se ve, `PrintWriter` puede escribir tanto a streams de caracteres como de bytes²⁹. Además `PrintWriter` tiene dos constructores (los dos últimos) que nos permiten escribir a un archivo sin necesidad de crear un objeto `FileWriter` o `FileOutputStream`³⁰.

Standard i/o: `System.out`, `System.in`, `System.err`

El término *standard i/o* se refiere a la idea, tomada del sistema operativo UNIX, del programa que usa un stream único de información. En ese contexto, un programa recibe toda la información de la entrada estándar (*standard input*), escribe toda su salida a la salida estándar (*standard output*) y por último todos sus mensajes de error van a *standard error*³¹.

Muchos lenguajes de programación usan este modelo³², que permite hacer programas cuya salida sirven de entrada para otro programa³³. Java también sigue este modelo, y tiene unos streams que están siempre abiertos cuando el programa empieza a correr: `System.out`, `System.in`, `System.err`. Estos tres *streams* **son streams de bytes**, lo que quiere decir que para trabajar con ellos hay que usar `InputStream`, `OutputStream` y sus subclases.

`System.out` y `System.err` vienen “pre-empaquetados” en un `PrintStream`, y es por eso que podemos decir `System.out.println(“Hola”)`³⁴, sin necesidad de crear un objeto `PrintStream` o

26 Podemos suponer que dinámico en este contexto quiere decir durante la ejecución del programa (*late binding*).

27 También existe la clase `PrintStream`, que escribe a un `OutputStream`.

28 La documentación de esta clase se puede encontrar en

<http://java.sun.com/j2se/1.5.0/docs/api/java/io/PrintWriter.html>. `printf` es un método similar al `printf` del C/C++: permite dar formato con precisión a los objetos que se quieren escribir al stream. La documentación sobre las distintas opciones de formato se encuentra en

<http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html#syntax>

29 Primer y segundo constructor, que tienen como parámetro un `Writer` y un `OutputStream`, respectivamente.

30 Como tendríamos que hacer si usamos el primer o segundo constructor. Por ejemplo, `PrintWriter out = new PrintWriter(new FileWriter(“Archivo.txt”));` o, nos queremos poner dramáticos, `PrintWriter out = new PrintWriter(new FileWriter(new File(“Archivo.txt”));`

31 ¿”error estándar”?

32 De hecho, el C/C++ tiene `stdin`, `stdout`, `stderr` (o, en la versión de streams del C++, `cin`, `cout` y `cerr`)

33 En Linux, podríamos decir en una ventana de shell: `$ dir | less`. `dir` lista el directorio, pero en vez de mostrarlo directamente en pantalla, el listado de directorio se convierte en la entrada del programa `less` (que evita que si el listado es más largo que una pantalla “se pase” sin que el usuario lo pueda ver... es un paginador).

34 ¿Qué sucede si decimos `System.err.println(“Hola”)` ;? En BlueJ la impresión sale en la mitad inferior de la ventana de terminal, donde se muestran los errores del programa. Esto es particularmente útil para depurar programas.

PrintWriter. En cambio, System.in es un InputStream *raw* (“crudo”, tal cual).

Si nos interesa leer caracteres del teclado en vez de bytes (suele ser lo normal), nos encontramos que BufferedReader lee de un Reader, no de un InputStream. Necesitamos un convertido de bytes a caracteres, es decir, una clase que sea subclase de Reader pero que pueda leer de un objeto InputStream.

La librería de io de Java proporciona estas clases: InputStreamReader, que sirve de puente entre un InputStream y un Reader, y OutputStreamReader, que hace lo mismo entre un OutputStream y un Writer.

Un ejercicio:

Con la siguiente información, haga un programa que lea líneas del teclado y las escriba a la pantalla. El programa termina cuando el usuario escribe una línea en blanco.

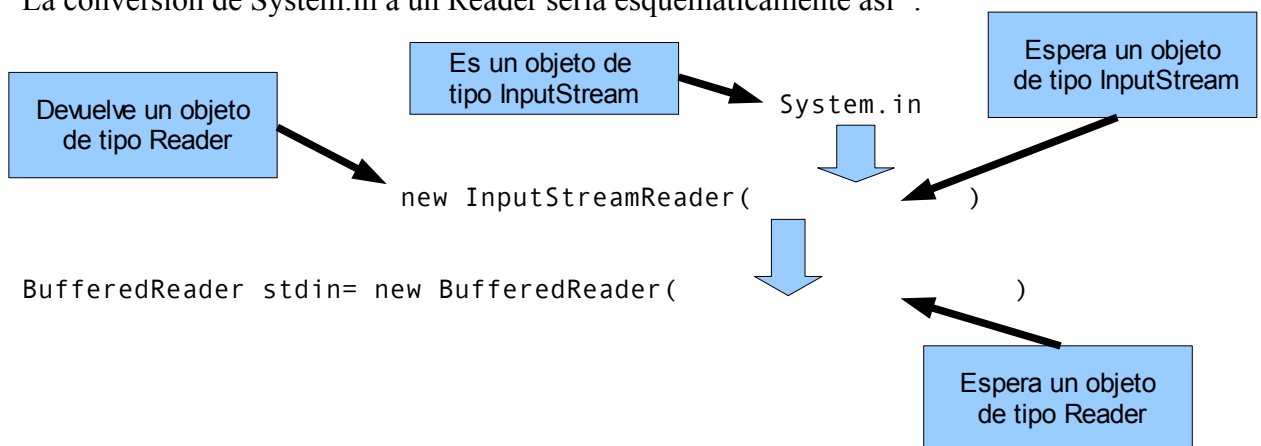
BufferedReader(Reader in)	constructor de la clase BufferedReader, proporciona el codiciado método readLine()
System.in	la entrada estándar (en nuestro caso, el teclado) que está siempre abierta para lectura, de tipo InputStream
InputStreamReader(InputStream in)	Constructor de la clase InputStream, que es una subclase de Reader.

Solución:

Usaremos el método readLine() de BufferedReader para leer del stream. El loop principal del programa sería algo así³⁵:

```
while( (s = stdin.readLine( ) ) != null && (s.length() != 0)) {  
    System.out.println(s);  
}
```

La conversión de System.in a un Reader sería esquemáticamente así³⁶:



Por tanto, nuestro objeto Reader sería:

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

35 La expresión `s.length() == 0` es para ver si el usuario escribió una línea en blanco (una de las condiciones de fin).

36 Cuando tengamos dudas de cómo conectar los objetos de io entre sí, puede ayudar hacer un esquema parecido.

Ahora podemos hacer nuestro programa:

```
import java.io.*;

public class Teclado {
    public static void main(String[] args) throws37 IOException {
        BufferedReader stdin = new BufferedReader(
            new InputStreamReader( System.in ) );
        String s;
        System.out.print("> ");    // el "prompt"
        while ( (s=stdin.readLine() ) != null && (s.length() != 0)) {
            System.out.println(s);
            System.out.print("> ");
        }
    }
}
```

Comentarios, correcciones y sugerencias: Roberto Zoia (roberto.zoia@gmail.com)

This work is licensed under the Creative Commons Attribution-ShareAlike 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

³⁷ Para simplificar, hacemos que el método `main` arroje las excepciones (que en este caso irán al usuario) que puedan generarse dentro del método.