Asignatura .Programación de Aplicaciones

# Anexo I **Introducción al tratamiento de flujos de datos y acceso a archivos secuenciales en Java**

# Índice de contenidos

# 1. El paquete **java.io**

En este anexo se analiza el paquete que el API estándar de Java proporciona para gestionar las operaciones de Entrada/Salida.

El paquete no se será cubierto en su totalidad a nivel de clases, pero si a nivel funcional y organizativo.

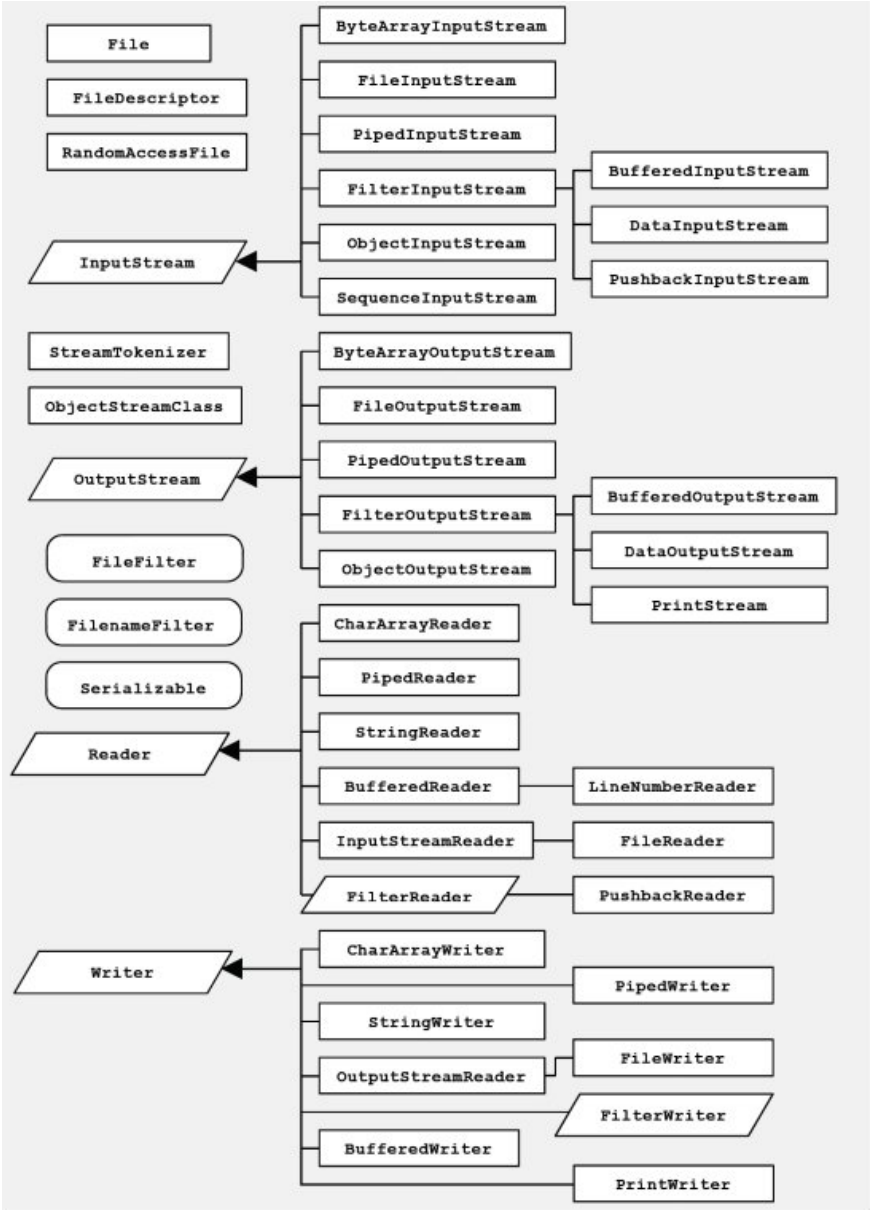
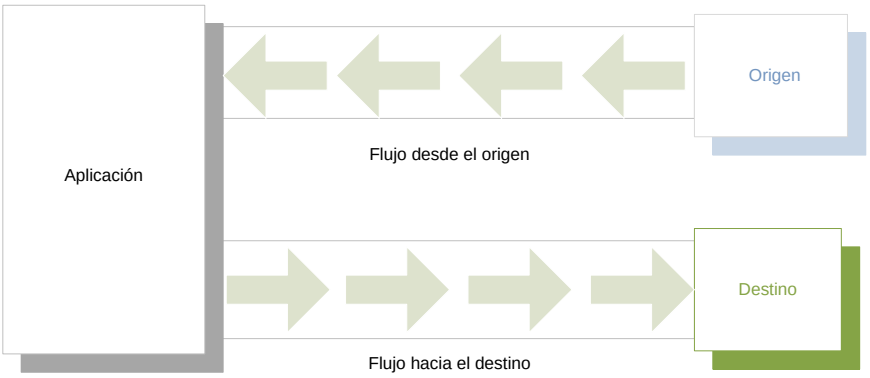



**Figura 1** Organización de clases del paquete java.io

## 2. Tratamiento de flujos de entrada y salida de datos en Java

Una aplicación normalmente necesitará obtener información desde un origen o enviar información a un destino.

La comunicación entre el origen de cierta información y el destino, **se realiza usando** lo que denomina ***flujo*** de información (en inglés *stream*).




Un ***flujo*** **es un objeto que hace de intermediario entre el programa, y el origen o el destino de la información.** Esto es, el programa leerá o escribirá en el *flujo* sin importarle desde dónde viene la información o a dónde va y tampoco importa el tipo de los datos que se leen o escriben. Este nivel de abstracción hace que el programa no tenga que saber nada ni del dispositivo ni del tipo de información, lo que se traduce en una facilidad más a la hora de escribir programas.

Para que un programa pueda obtener información desde un origen **tiene que abrir un flujo** y leer la información. De la misma forma, para que un programa pueda enviar información a un destino tiene que abrir un flujo y escribir la información.

La estructura general de los algoritmos usados para leer y escribir datos de flujos datos son siempre más o menos los mismos. La siguiente tabla muestra el pseudocódigo de estos algoritmos.

| Leer | Escribir |
|---|---|
| *Abrir un flujo desde el origen*<br>*Mientras haya información*<br>  *Leer información*<br>*Cerrar el flujo* | *Abrir un flujo hacia el destino*<br>*Mientras haya información*<br>  *Escribir información*<br>*Cerrar el flujo* |

**Tabla 1.** Algoritmos de lectura y escritura desde flujos

Debido a que todas las clases relacionadas con los flujos pertenecen al paquete **java.io** de la biblioteca estándar de Java, un programa que utilice flujos de E/S tendrá que importar este paquete.

El paquete **java.io** de la biblioteca estándar de Java, contiene una colección de clases que soportan estos algoritmos para leer y escribir. Estas clases **se dividen en dos grupos distintos**, según se muestra en la figura siguiente (Figura 1). El grupo de la izquierda ha sido diseñado para trabajar con datos de tipo **byte** (8 bits) y el de la derecha con datos de tipo **char** (16 bits). Ambos grupos presentan clases análogas que **tienen interfaces casi idénticas**, por lo que se utilizan de la misma manera

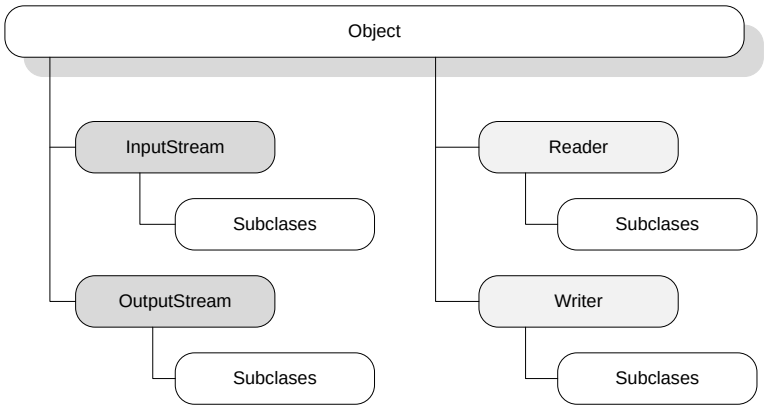
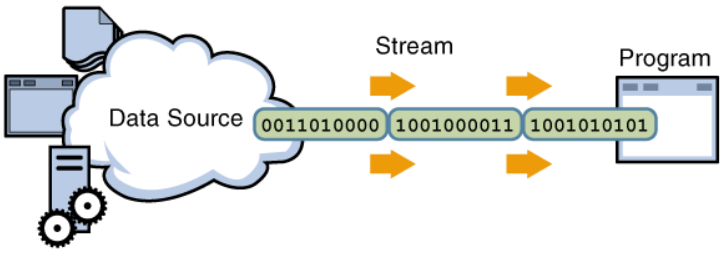



Figura 1Clasificación de clases para trabajar con flujos

## 3. Flujos abstractos de entrada de datos

Un programa usa un flujo de entrada para leer datos de alguna fuente de datos. Donde una fuente de datos puede ser un archivo en un sistema de archivos, un dispositivo, un socket, estructuras de datos en memoria, etc.

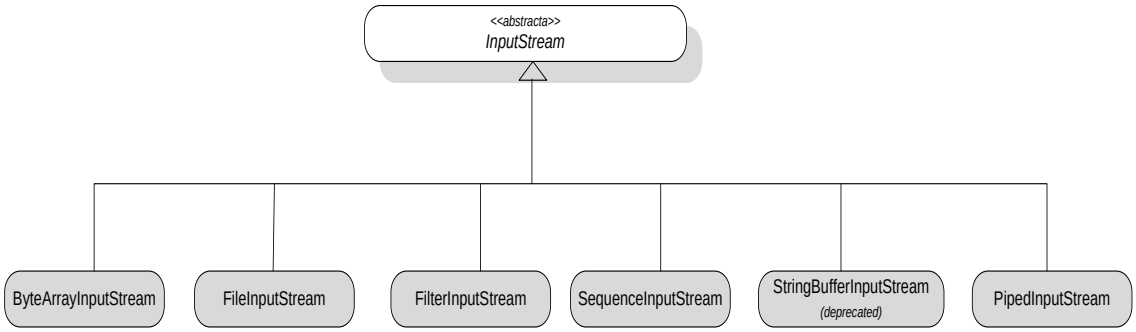



Tipos de flujos de entrada:

a) Flujo abstracto orientado a bytes : **clase InputStream**
b) Flujo abstracto orientado a caracteres : **clase Reader**

## 3.1.Flujo abstracto orientado a bytes. **Clase InputStream**

El siguiente diagrama muestra la jerarquía de clases de flujos de entrada orientados a byte




La clase **InputStream** es una **clase abstracta** que es superclase de todas las clases que representan un flujo en el que el destino **lee bytes de un origen**. Cuando una aplicación define un flujo de entrada, la aplicación es destino de ese flujo de bytes, y con este modelo es todo lo que necesita saber.

El **método más importante** de esta clase es **read**. Este método presenta los siguientes prototipos:

```
public int read() throws IOException

public int read(byte[] b) throws IOException

public int read(byte[] b,int off,int len) throws IOException
```

La primera versión de **read** simplemente lee bytes individuales de un flujo de entrada; concretamente lee el siguiente byte de datos disponibles. Devuelve un entero(int) correspondiente al valor ASCII del carácter leído, al número de bytes leídos si se lee una matriz, o bien -1 cuando en un intento de leer se alcanza el final del flujo ( esto es no hay más datos).

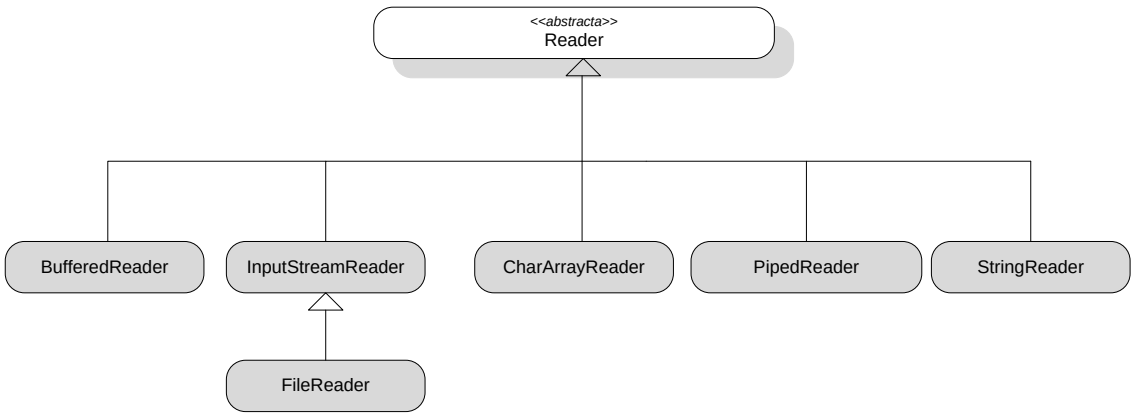
A continuación se detalla de forma general cada una de las subclases

| **Flujo de entrada** | **Descripción** |
|---|---|
| ByteArrayInputStream | Permite leer bytes de un array que esté en memoria |
| FileInputStream | Permite leer bytes de un fichero del sistema local de archivos |
| FilterInputStream | Una clase **FilterInputStream** contiene algún otro flujo de entrada para usarlo como entrada de datos, posiblemente para transformarlo o proporcionar alguna funcionalidad adicional sobre él.<br>Esta clase simplemente sobrescribe todos los métodos de InputStream |
| PipedinputStream | Permite leer bytes desde un tubería ( pipe) creada por un hilo |
| StringBufferInputStream | Permite leer bytes desde una cadena |
| SequenceInputStream | Permite leer bytes de datos desde dos o más flujos de bajo nivel, cambiando a otro flujo al finalizar la lectura de cada uno de ellos |

### 3.2.Flujo abstracto orientado a caracteres .**Clase Reader**

Análogamente la clase **Reader** es una **clase abstracta** que es superclase de todas las clases que representan un flujo para **leer caracteres desde un origen**. Sus métodos son análogos a los de la clase **InputStream** , con la diferencia de que utilizan parámetros de tipo **char** en lugar de **int**.

El siguiente diagrama muestra la jerarquía de clases para lectura de caracteres

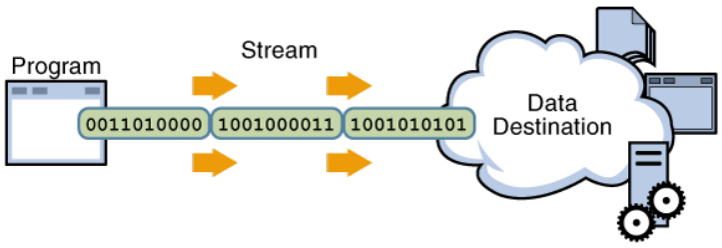



A continuación se detalla de forma general cada una de las subclases

| **Flujo de entrada** | **Descripción** |
|---|---|
| CharArrayReader | Lee de un array de caracteres |
| FileReader | Lee de un fichero en el sistema de ficheros local |
| PipedReader | Lee caracteres de un pipe de comunicación entre hilos |
| StringReader | Lee caracteres de una cadena (String) |
| InputStreamReader | Permite asociar un *reader* a un input stream para poder leerlo |
| | |

## 4. Flujos abstractos de salida de datos

Un programa usa un flujo de salida para escribir datos en algún destino. Donde un destino de datos puede ser un archivo en un sistema de archivos, un dispositivo, un socket, estructuras de datos en memoria, etc.

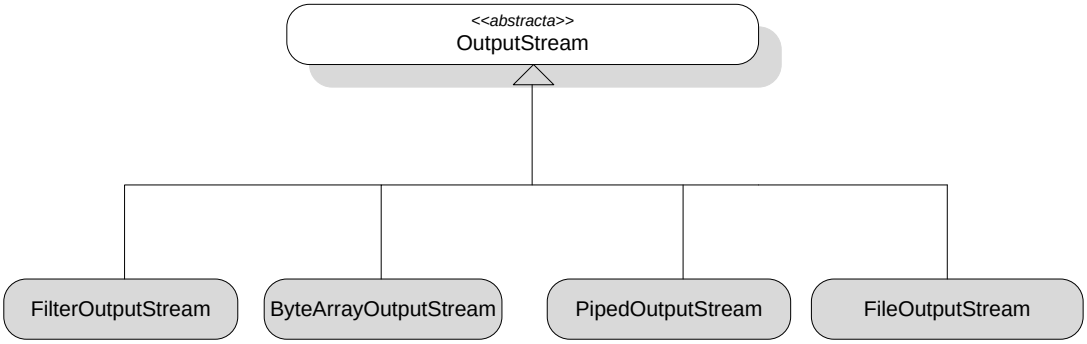



Tipos de flujos de salida

a) Flujo abstracto de salida orientado a bytes : **clase OutputStream**

b) Flujo abstracto de salida orientado a caracteres : **Clase Writer**

## 4.1.Flujo abstracto de salida orientada a bytes .**Clase OutputStream**




La clase **OutputStream es una clase abstracta** que es superclase de todas las clases que representan un flujo en el que un **origen escribe bytes en un destino**. Cuando una aplicación define un flujo de salida, la aplicación es origen de ese flujo de bytes (es la que envía los bytes)

A continuación se detallan cada una de las subclases

| **Flujo de salida** | **Descripción** |
|---|---|
| FilterOutputStream | Esta clase solo sobrescribe todos los métodos de **OutputStream** . la clase usa otro flujo existente para hacer algún tipo de procesamiento o filtrado |
| ByteArrayOutputStream | Escribe bytes a un array que este en memoria |
| FileOutputStream | Escribe bytes de datos a un fichero del sistema local de archivos |
| PipedOutputStream | Escribe bytes de datos a una tubería ( pipe) creada por un hilo |
| | |

El método más importante es **write**. Este método tiene los siguientes prototipos:

- `public int write() throws IOException`
- `public int write(byte[] b) throws IOException`
- `public int write(byte[] b,int off,int len) throws IOException`

La primera versión de **write** simplemente escribe el byte especificado en un flujo de salida. Puesto que su parámetro es de tipo int, lo que se escribe es el valor correspondiente a los 8 bits menos significativos, el resto son ignorados.

Por ejemplo suponiendo que tenemos definido un objeto *flujoS* (flujo de salida) de alguna subclase de **OutputStream,** el siguiente código escribe el byte especificado en el destino vinculado con *flujoS*:

```
int n;

//…

flujoS.write(n);
```

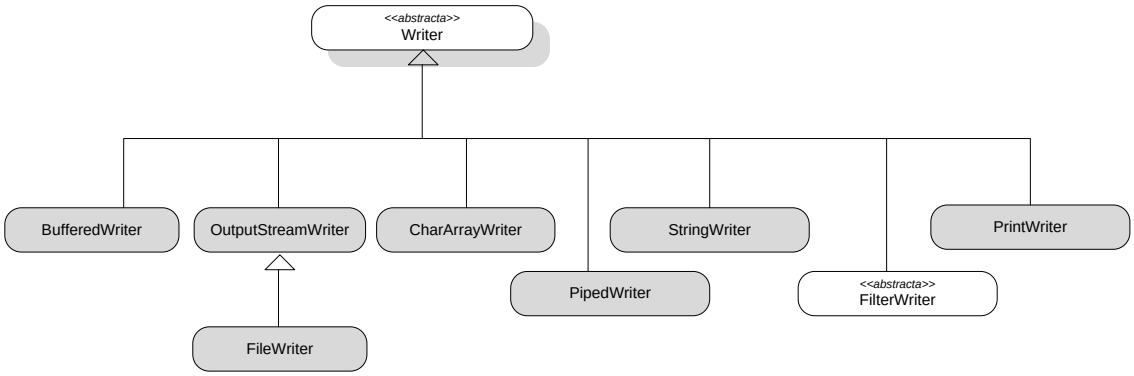
La segunda versión del método **write** escribe los bytes almacenados en la matriz b en un flujo de salida

```
byte[] b= new byte[128]; //matriz 'b' de 128 bytes

flujoS.write(b);
```

La tercera versión del método **write** escribe u máximo de **len** bytes de una matriz b a partir de su posición *off*, en un flujo de salida.

## 4.2. Flujo abstracto de salida orientado a caracteres. **Clase Writer**




A continuación se detallan cada una de las subclases

| **Flujo de salida** | **Descripción** |
|---|---|
| CharArrayWriter | Escribe un array de caracteres |
| FileWriter | Escribe un fichero en el sistema local de archivos |
| PipedWriter | Escribe caracteres de un pipe de comunicación entre hilos |
| StringWriter | Escribe caracteres en un String |
| OutputStremWriter | Permite asociar un *write*r a un flujo de salida *( output stream)* escribiendo en el |
| FilterWriter | Es una clase abstracta que representa un tratamiento de un flujo a nivel de caracteres<br>Las subclases de esta clase deberían ofrecer métodos y atributos adicionales para tratar la salida del flujo |

Análogamente la clase **Writer es una clase abstracta** que es superclase de todas las clases que representan un flujo para **escribir caracteres** a un destino. Sus **métodos son análogos** a los de la clase **OutputStream.**

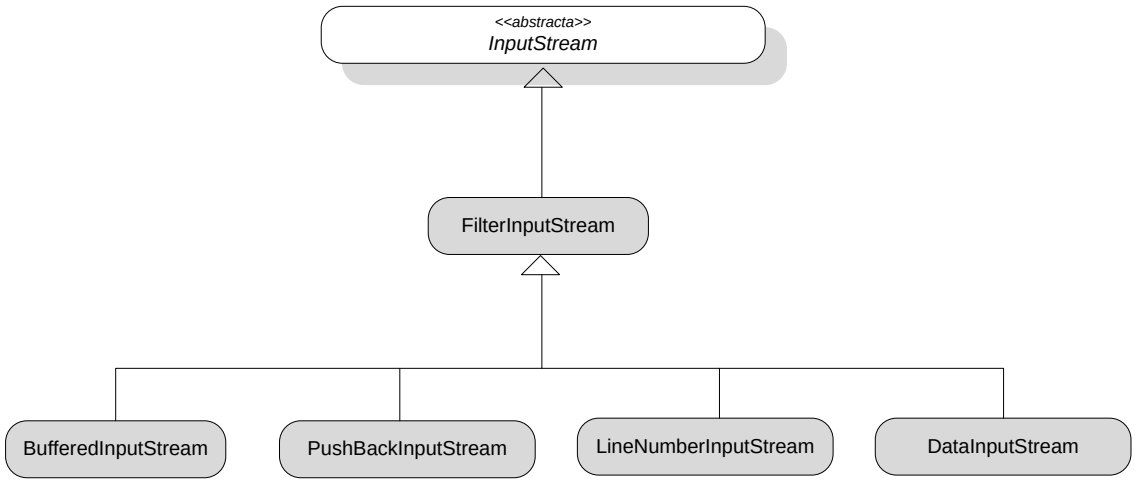
# 5. Flujos abstractos de entrada de datos procesados (o filtrados)

A pesar de que los **flujos de bajo nivel** mostrados permiten leer bytes, **su flexibilidad es limitada** .Los flujos de datos procesados o filtrados añaden funcionalidad a un flujo existente, procesando los datos previamente de alguna forma (por ejemplo almacenándolos en bloques antes de poder leerlos o proporcionando nuevos métodos de acceso. Es importante considerar que para crear un flujo de entrada procesado o filtrado hay que disponer primero de un flujo

de entrada de bajo nivel (vistos anteriormente), ya que el constructor de esta clase de flujos toma como parámetro un flujo existente.

A continuación se muestra la jerarquía de clases de flujos de entrada procesados o filtrados orientados a leer bytes.

## 5.1. Flujo de entrada procesada (orientada a bytes). **Clase FilterInputStream**




A continuación se detalla de forma general cada una de las subclases de InputStream

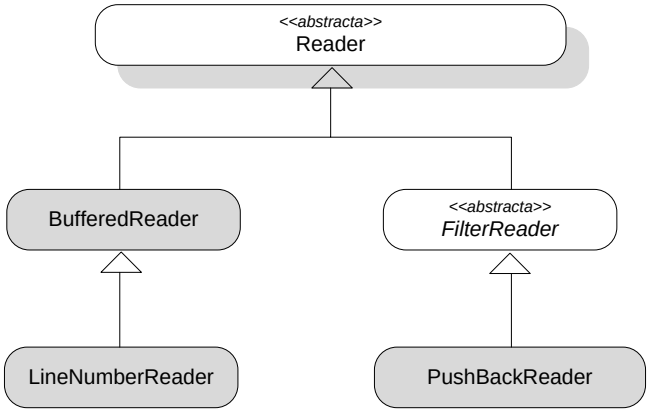
| Flujo de entrada | Descripción |
|---|---|
| BufferedInputStream | Almacena los datos en una memoria intermedia para mejorar la eficiencia de lectura de los mismos |
| DataInputStream | Permite leer tipos de datos primitivos, tales como int, float, doublé e incluso líneas de texto |
| LineNumberInputStream | Mantiene un contador sobre el número de línea que se está leyendo en función de carácter de final e línea |
| PushBackInputStream | Añade a otro flujo de entrada la capacidad de volver a colocar un byte leído ( o array de bytes) otra vez en el flujo. La siguiente operación de lectura volverá a leer el último byte |

Veremos más adelante el flujo de entrada filtrado **BufferedInputStream**.

**Nota:** El constructor de esta clase de flujo toma un flujo existente como entrada. El método constructor de esta clase tiene el siguiente prototipo

```
protected  FilterInputStream ( InputStream)
```

## 5.2. Flujo de entrada procesada (orientada a caracteres)



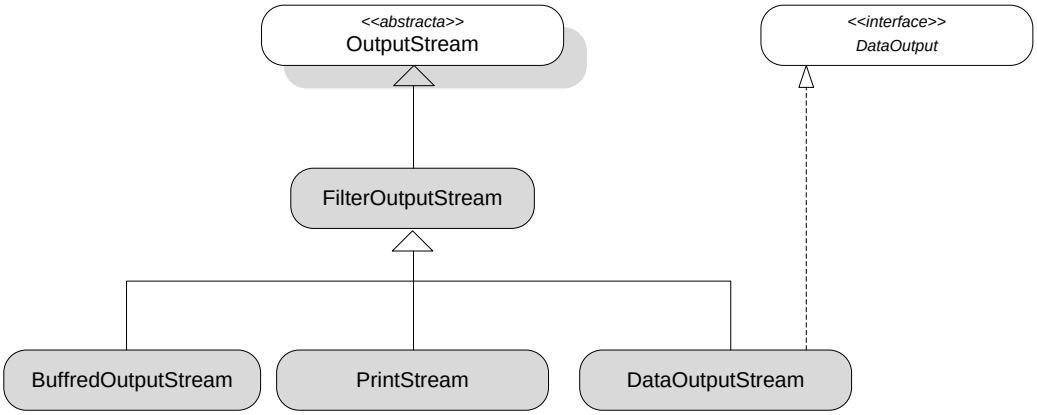

La clase **BufferedReader** se analizará más adelante.

| Flujo de entrada | Descripción |
|---|---|
| BufferedReader | Lee caracteres de un fichero, almacenando los caracteres en una memoria intermedia para que la lectura sea más eficiente |
| PushBackReader | Permite volver a poner en el flujo caracteres leídos |
| LineNumeberReader | Lee caracteres de un fichero utilizando almacenamiento y permitiendo consultar el número de líneas leídas |

# 6. Flujos abstractos de salida de datos procesados (o filtrados).

Al igual que ocurre con los flujos de entrada procesados o filtrados, los flujos de salida procesados tienen un flexibilidad y funcionalidad extendida.

## 6.1. Flujo de salida de datos procesados (orientados a bytes) .**Clase FilterOutputStream**




A continuación se describen cada una de las subclases

| Flujo de entrada | Descripción |
|---|---|
| BufferedOutputStream | Esta clase implementa un flujo de salida con buffer. Lo que permite mejorar la eficiencia en la escritura de datos |
| DataOutputStream | Permite escribir tipos primitivos directamente en el flujo de un modo portable |
| PrintStream | Añade la capacidad de escribir representaciones de diversos datos de forma conveniente. Los métodos de esta clase no lanzan IOException<br>La salida estándar es de este tipo |

## 6.2.PrintStream

La clase **PrintStream** se deriva indirectamente de **OutputStream**, por lo que hereda todos los miembros de esta; por ejemplo el método **write** expuesto anteriormente. Otros métodos de interés que aporta esta clase, que ya hemos utilizado son: **print y println** . La sintaxis de estos métodos es la siguiente

- `print ( tipo argumento);`
- `println([tipo argumento]);`

Los métodos **print y println** son esencialmente los mismos; ambos escriben su argumento en el flujo de salida. La única diferencia entre ellos es que **println** añade el carácter '\n' (avance de línea) al final de su salida.

Los argumentos de **print y println** pueden ser de cualquier tipo primitivo o referenciado: char,int , Object , char [] etc….

La **clase análoga** a **PrintStream es PrintWriter**, clase derivada de **Writer,** pero los métodos proporcionados por ambas clases son prácticamente los mismos, por lo no comentaremos esta última. Una diferencia entre ambas es que cuando se ejecuta un método de **PrintStream**, el buffer de salida es vaciado automáticamente, no sucediendo lo mismo con **PrintWriter**; en esta caso habría que forzar el vaciado del *buffer* de salida invocando su método **flush . Por ejemplo:**

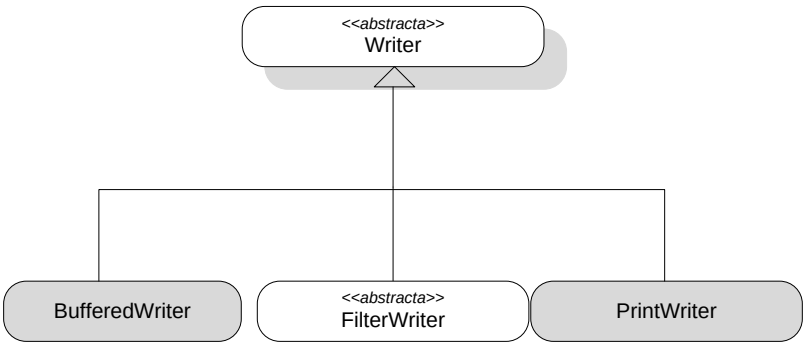
```
PrintWriter flujos= new PrintWriter(System.out);

Int dato=4;

flujos.println(dato);

flujos.flush();
```

## 6.3. Flujo de salida de datos procesados (orientado a carácter)




A continuación se describen cada una de las subclases

| Flujo de entrada | Descripción |
|---|---|
| BufferedWriter | Lee caracteres de un fichero, almacenando los caracteres para que la lectura sea más eficiente |
| PrintWriter | Permite escribir tipos primitivos y objetos a un flujo de salida orientado a caracteres |

# 7. Flujos filtrados. Lectura y escritura con memoria intermedia (buffer)

Existen varias clases de flujos filtrados como hemos visto, en particular vamos a ver aquellos que usan una memoria intermedia para acoplar el origen y destino del flujo. En concreto vamos a ver las subclases **BufferedInputStream y BufferedOutStream**

La mayoría de los ejemplos que hemos visto usan una entrada/salida sin buffers. Esto significa que cada lectura o escritura se maneja directamente por el sistema operativo. Esto hace que el programa sea menos eficiente en relación con estas operaciones, puesto que cada solicitud de lectura o escritura lleva asociado comenzar un nuevo acceso a disco y esta operación es relativamente costosa en términos de rendimiento. Para reducir esta sobrecarga java usa flujos con buffer.

## 7.1. Lectura y escritura con buffer. **Clase BufferedInputStream y BufferedInputStream**

La clase **BufferedInputStream** se deriva (no directamente si no a través de FilterInputStream) de la clase abstracta **InputStream**, por lo tanto hereda todos los miembros de esta; por ejemplo el método **read** expuesto anteriormente. La clase **BufferedInputStream** aunque no aporta ningún método nuevo si **aporta una característica muy interesante** de la que se

benefician todos sus métodos: un ***buffer*** que actúa como memoria intermedia para las lecturas futuras. Para comprender esto observe la siguiente figura:


**Figura 2** Lectura con buffer

Según el esquema anterior, cuando la aplicación ejecute un sentencia de entrada (que solicite datos) los datos obtenidos desde el origen pueden ser depositados en el *buffer* en bloques más grandes que los que realmente está leyendo la aplicación (por ejemplo, cuando se leen datos de un disco la cantidad mínima de información transferida es un bloque equivalente a una unidad de asignación). Esto aumenta la velocidad de ejecución porque la siguiente vez que la aplicación necesite más datos no tendrá que esperar por ellos porque ya los tendrá en el buffer. Por otra parte cuando es una operación de salida, los datos no serán enviados hasta que no se llene el *buffer* (o hasta que se fuerce el vaciado del buffer explícitamente), lo que reduce el número de acceso al dispositivo físico vinculado que siempre resulta mucho más lento que los accesos a memoria.

Ejemplo

```
import java.io.*;

class LectorConFiltro

{

    public static void main(String args[])

    {

        try  {

      FileInputStream fin = new FileInputStream("file.txt");

            BufferedInputStream bis = new BufferedInputStream(fin);


            while (bis.available() > 0)

            {

                System.out.print((char)bis.read());

            }

        }
```

```
        catch (Exception e)
        {
            System.err.println("Error de lectura de archivo: " + e);
        }
    }
}
```

La parte más importante de este ejemplo es la creación de los flujos con un buffer para operar con ellos. A continuación se muestra este fragmento de código:

```
      FileInputStream fin = new FileInputStream("file.txt");
            BufferedInputStream bis = new BufferedInputStream(fin);
```

La clase análoga a **BufferedInputStream** pero que permite trabajar con caracteres directamente es **BufferedReader,** clase derivada de **Reader.**

### Operación de vaciado de buffer

Siempre que se trabaja con flujos de salida con buffer existe la posibilidad de forzar el vaciado del contenido buffer al destino indicado. Por ejemplo esta operación podría darse en una operación crítica que no puede esperar a que el buffer esté lleno para escribir el contenido.

Algunas clases con buffer soportan la operación de **autoflush o forzado de vaciado del buffer automático**. Esto se debe especificar en un parámetro del constructor. Cuando el vaciado automático está activado existen ciertos eventos que causan que el buffer se vacíe. Por ejemplo la clase **PrintWriter** vacía el buffer cada vez que se invoca la operación println() o format.

Para forzar el vaciado manualmente se debe invocar el método **flush()**. Este método es válido para cualquier flujo de salida, pero no tendrá efecto si el flujo no es con buffer.

El siguiente ejemplo muestra la invocación del **método flush()** sobre un flujo con buffer

```
FileOutputStream archivo = new FileOutputStream("archivo");

DataOutputStream flujoSalida = new DataOutputStream(archivo);

String strContent = "Linea de datos hacia el flujo de salida";
flujoSalida.writeBytes(strContent);

flujoSalida.flush();
```

# 8. Acceso secuencial de un archivo

El tipo **de acceso más simple a un fichero de datos es el secuencial**. Un fichero abierto para acceso secuencial es un fichero que puede almacenar registros de cualquier longitud, incluso de un solo byte. Cuando la información se escribe registro a registro, estos son colocados uno a continuación de otro, y cuando se lee, se empieza por el primer registro y se continúa al siguiente hasta alcanzar el final.

Este **tipo de acceso generalmente se utiliza con ficheros de texto** en los que se escribe toda la información desde el principio hasta el final y se lee de la misma forma.

Los **ficheros de texto no son los más apropiados para almacenar grandes series de números**, porque cada número es almacenado como una secuencia de bytes; esto significa que un número de nueve dígitos ocupa nueve bytes en lugar de los cuatro requeridos para un entero

A continuación se muestran los distintos tipos de flujos: de bytes y caracteres para el tratamiento de texto, y de datos para el tratamiento de números.

## 8.1. Flujos sobre archivos ( orientados a bytes)

Los datos pueden ser escritos o leídos de un fichero byte a byte utilizando flujos de las clases **FileOutputStream y FileInputStream**

### 8.1.1. Flujos de salida para archivos orientados a bytes. **Clase FileOutputStream**

Un flujo de la clase **FileOutputStream** permite escribir bytes a un fichero

**Constructores**

- `FileOutputStream ( String nombre)`
- `FileOutputStream ( String nombre, bool añadir)`
- `FileOutputStream ( File fichero)`

**Ejemplo**

```
import java.io.*;
public class CEscribirBytes
{
  public static void main (String[] args)
  {
    FileOutputStream fs = null;
    byte[] buffer = new byte[81];
    int nbytes;

    try
    {
      System.out.println(
        "Escriba el texto que desea almacenar en el fichero:");
      nbytes = System.in.read(buffer);
      fs = new FileOutputStream("texto.txt");
      fs.write(buffer, 0, nbytes);
    }
    catch(IOException e)
    {
      System.out.println("Error: " + e.toString());
    }
  }
```

Siempre que se trabaje con un flujo, es casi una práctica obligatoria cerrar el flujo del archivo cuando se ha terminado de usar. Para ello se puede añadir el siguiente código justo después del bloque catch del ejemplo anterior.

```
finally
{
  try
  {
    // Cerrar el fichero
    if (fs != null) fs.close();
  }
  catch(IOException e)
  {
    System.out.println("Error: " + e.toString());
  }
}
```

### 8.1.2. Flujos de entrada para archivos orientados a bytes. **Clase FileInputStream**

Un flujo de la clase **FileInputStream** permite leer bytes desde un fichero. Además de los métodos que esta clase hereda de **InputStream,** la clase proporciona los siguientes constructores:

- `FileInputStream ( String nombre)`
- `FileInputStream ( String nombre, bool añadir)`
- `FileInputStream ( File fichero)`

**Ejemplo.**

El siguiente ejemplo es una aplicación Java que lee el texto guardado en el fichero *text.txt* creado en la aplicación anterior y lo almacena en una matriz denominada buffer.

La aplicación definida por la **clase CLeerBytes** mostrada a continuación, realiza lo siguiente:

1.- Define una matriz buffer de 81 *bytes*

2.-Define un flujo( *fe)* desde un fichero denominado *texto.txt*. Si no existe el fichero, se lanzará una excepción

3.- Lee el texto desde el flujo y lo almacena en *buffer*

4.- Crea un objeto String con los datos leídos

```
import java.io.*;

public class CLeerBytes
{

  public static void main (String[] args)
  {
    FileInputStream fe = null;
    byte[] buffer = new byte[81];
    int nbytes;

    try
    {
      fe = new FileInputStream("texto.txt");
      nbytes = fe.read(buffer, 0, 81);
      String str = new String(buffer, 0, nbytes);
      System.out.println(str);
    }
    catch(IOException e)
    {
      System.out.println("Error: " + e.toString());
    }
    finally
    {
      try
      {
        // Cerrar el fichero
        if (fe != null) fe.close();
      }
      catch(IOException e)
      {
        System.out.println("Error: " + e.toString());
      }
    }
  }
}
```

## 8.2.Flujos sobre archivos ( orientados a caracteres)

Una vez que sabemos cómo trabajar con flujos de bytes, hacerlo con flujos de caracteres es prácticamente lo mismo. Las clases que definen estos flujos (de entrada y de salida) son subclases de **Reader** y **Writer** y son **FileWriter y FileReader.**

### 8.2.1. Flujo de salida para archivos orientados a caracteres .**Clase FileWriter**

Un flujo de la clase **FileWriter** permite escribir caracteres (char) en un fichero

- `FileWriter (String nombre)`
- `FileWriter (String nombre,boolean añadir)`

- `FileWriter (File fichero)`

**Ejemplo**

```
import java.io.*;

public class CEscribirCars
{
  public static void main (String[] args)
  {
    FileWriter fs = null;
    byte[] buffer = new byte[81];
    int nbytes;
    String nombreFichero = null;
    File fichero = null;

    try
    {
      System.out.print("Nombre del fichero: ");
      nbytes = System.in.read(buffer);
      nombreFichero = new String(buffer, 0, nbytes-2); // menos CR+LF
      fichero = new File(nombreFichero);

      char resp = 's';
      if (fichero.exists())
      {
        System.out.print("El fichero existe ¿desea sobreescribirlo? (s/n) ");
        resp = (char)System.in.read();
        // Saltar los bytes no leídos del flujo in
        System.in.skip(System.in.available());
      }
```

```
      if (resp == 's')
      {
        System.out.println(
          "Escriba el texto que desea almacenar en el fichero:");
        nbytes = System.in.read(buffer);
        String str = new String(buffer, 0, nbytes);
        fs = new FileWriter(fichero);
        fs.write(str, 0, str.length());
      }
    }
    catch(IOException e)
    {
      System.out.println("Error: " + e.toString());
    }
    finally
    {
      try
      {
        // Cerrar el fichero
        if (fs != null) fs.close();
      }
      catch(IOException e)
      {
        System.out.println("Error: " + e.toString());
      }
    }
  }
}
```

## 8.3. Flujo de entrada para archivos orientados a caracteres. **Clase FileReader**

Un flujo de la clase **FileReader** permite leer caracteres desde un fichero. Los constructores son los siguientes:

- `FileReader (String nombre)`

- `FileReader (File fichero)`

**Ejemplo**

```
public class CLeerCars
{
  public static void main (String[] args)
  {
    byte[] nomFich = new byte[81];
    String nombreFichero = null;
    File fichero = null;
    int nbytes, ncars;
    FileReader fe = null;
    char[] buffer = new char[81];

    try
    {
      do
      {
        System.out.print("Nombre del fichero: ");
        nbytes = System.in.read(nomFich);
        nombreFichero = new String(nomFich, 0, nbytes-2); // menos CR+LF
        fichero = new File(nombreFichero);
      }
      while (!fichero.exists());

      fe = new FileReader(fichero);
      ncars = fe.read(buffer, 0, 81);
      System.out.println(buffer);
    }
    catch(IOException e)
    {
      System.out.println("Error: " + e.toString());
    }
    finally
    {
      try
      {
        // Cerrar el fichero
        if (fe != null) fe.close();
      }
      catch(IOException e)
      {
        System.out.println("Error: " + e.toString());
      }
    }
  }
```

# 9. Clase File

El los ejemplos anteriores se utiliza un **String** para referirse al fichero. Se podría haber usado un objeto de la clase **File.** Un objeto de esta clase representa el nombre de un fichero o de un directorio que existe en el sistema de ficheros, sus métodos permitirán interrogar al sistema sobre todas las características de ese fichero o directorio. Los constructores son los siguientes

- `public File (String ruta_completa)`
- `public File(String ruta,String nombre)`
- `public File (File ruta,String nombre)`

Ejemplo

```
File fichero = new File("proyecto\\texto.txt");

System.out.println("Nombre del fichero: " + fichero.getName());
System.out.println("Directorio padre:   " + fichero.getParent());
System.out.println("Ruta relativa:      " + fichero.getPath());
System.out.println("Ruta absoluta:      " +
                                  fichero.getAbsolutePath());
```

**NOTA IMPORTANTE**

La última versión de Java en el momento de la redacción de este documento es JDK 6 a punto de salir JDK 7 con importantes modificaciones en la librería I/O de Java.

En concreto existe una nueva clase denominada **Path** que ofrece muchas más funcionalidades que File, si bien File no será desaconsejada (deprecated) además existe una forma de trabajar con ambas.

Para más información
JSRs: Java Specification Requests JSR 203: **More New I/O APIs for the JavaTM Platform** ("NIO.2") http://www.java.net/external?url=http://jcp.org/en/jsr/detail?id=203

# 10. Entrada y salida orientadas a líneas de caracteres

La entrada y salida con caracteres normalmente se realiza agrupando los caracteres en líneas. Una línea es un conjunto de caracteres con un carácter de control que indica su terminación, llamado terminador de línea. Un terminador de línea puede ser un secuencia de retorno de carro y retorno de línea ("\r\n"), un simple "retorno de carro" ("\r") o un simple retorno de línea ("\n"). Soportar todos y cada uno d los terminadores permite a un programa leer ficheros de texto creados en cualquier sistema operativo, ya que estos difieren en los distintos sistemas operativos.

A continuación se va a modificar la clase `CopyCharacters (usada en un ejemplo anterior)` para usar lectura y escritura de líneas del archivo de texto. Para hacer esto, vamos a usar dos clases, que ya hemos visto `BufferedReader y PrintWriter`.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream =
                new BufferedReader(new FileReader("xanadu.txt"));
            outputStream =
                new PrintWriter(new
FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
```

```
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

La invocación de `readLine()` devuelve una línea de texto. La clase `CopyLines` escribe la línea en el flujo de salida usando el método `println`, la cual añade el terminador de línea para el sistema operativo que se esté utilizando en ese momento, qué podría no ser el mismo que se está usando en la entrada.

**NOTA:** existen varias formas de estructurar el texto usado como entrada o salida. A continuación se verá esta cuestión en más detalle

# 11. Lectura de la información de entrada con un analizador (Scanner)

La programación de aplicaciones que hacen uso de operaciones de entrada /salida implica frecuentemente la transformación de la información desde o hacia un formato .para asistir al desarrollador en esta tarea Java proporciona el API scanner que permite dividir una entrada en

El analizador proporcionado por Java en su *API scanner* divide la entrada en tokens individuales de forma que sea más sencillo su posterior asignación a unidades de datos independientes.

## 11.1. Análisis de entrada con analizador (Scanner)

Los objetos de tipo Scanner son útiles para dividir la entrada en tokens en función de su tipo.

Por defecto, un Analizador (Scanner) usa el espacio en blanco como separador de tokens (los espacios en blanco pueden ser debidos a caracteres de espaciado como tabuladores). Para más información consultar la documentación para el siguiente método de la clase `Character` `Character.isWhitespace`.

A continuación se muestra un ejemplo de una lectura de un flujo de entrada usando un analizador que usa el carácter espaciador como carácter separador de elementos .

### Ejemplo

```
import java.io.*;
import java.util.Scanner;

public class ScanXan {
    public static void main(String[] args) throws IOException {
        Scanner s = null;
        try {
            s = new Scanner(new BufferedReader(new
FileReader("xanadu.txt")));

            while (s.hasNext()) {
                System.out.println(s.next());
```

```
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

Observe que `ScanXan` invoca el método de cierre del analizador cuando termina el proceso. Aunque un analizador no es un flujo, se necesita cerrarlo explícitamente para indicar que ya no va a trabajar con el flujo subyacente.

La salida obtenida es la siguiente:

```
In
Xanadu
did
Kubla
Khan
A
stately
pleasure-dome
...
```

Para usar diferentes caracteres de separación de elementos en el Analizador, se debe usar `useDelimiter()`, especificando como parámetro un expresión regular. Por ejemplo, suponiendo que se usa como separador de elementos una coma separada con un espacio la expresión regular sería "`,\\s*`" y su uso con este método sería como sigue

```
s.useDelimiter(",\\s*");
```

# 12. Bibliográfica


- Java 2 Curso de programación. Fco Javier Ceballos .Editorial RA_MA
- The java Tutorials. .Basic IO . http://download.oracle.com/javase/tutorial/essential/io/