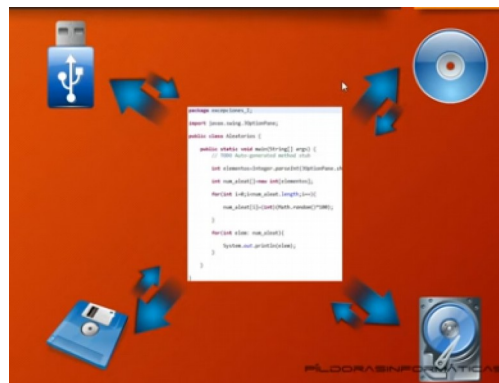


¿Para qué sirve la secuencia o streams, también denominada flujo de datos en Java?

Sirve para diversos propósitos como ejemplo, enviar información desde un programa Java a un sitio remoto a través de la red, para acceder (leer/escribir) a ficheros externos que tengamos en nuestro equipo.

¿Por qué surge la necesidad desde un programa Java de manejar ficheros externos?

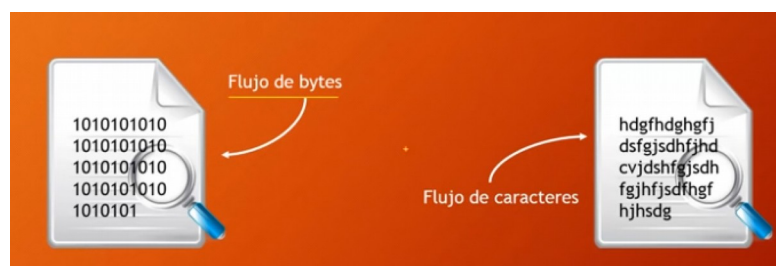
Para almacenar la información de forma permanente, lo tenemos que hacer en un dispositivo físico (HD, DVD, Pen Driver) para hacer esto hay que manejar Streams o flujo de datos.



¿Cómo abordamos en Java estos flujos?

Lo podemos abordar de dos formas:

- a) Como flujos de bytes
- b) Como flujos de caracteres.



¿Cómo saber cuando debemos usar una u otra forma?

Va a depender de lo que queramos hacer. Si lo que vamos a almacenar es un fichero de texto, usaremos flujo de caracteres. Si queremos enviar información a través de la red, lo normal es que se haga como flujo de bytes.

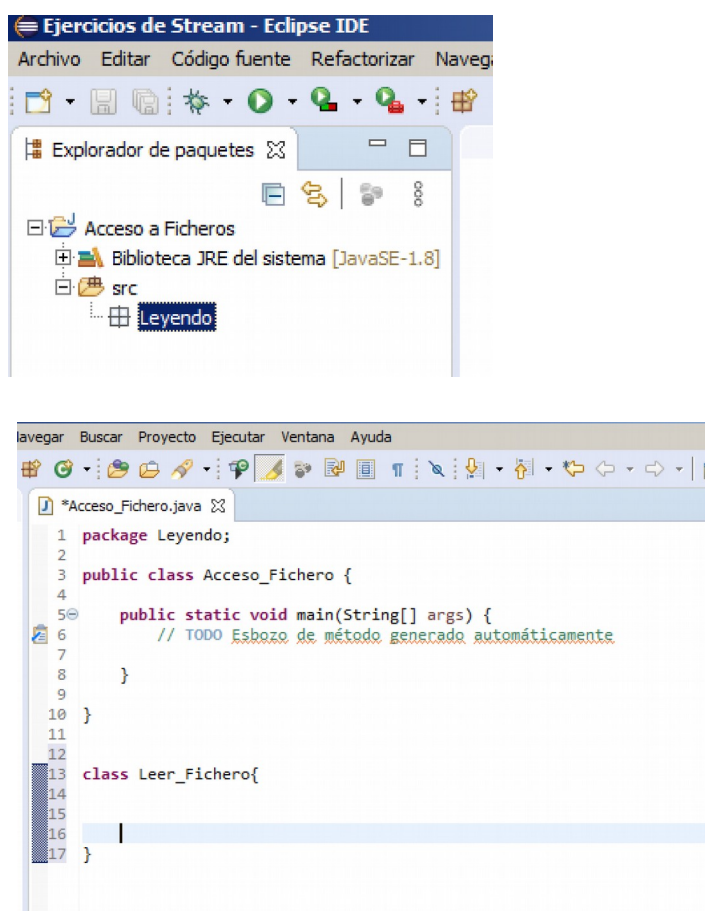
Para el manejo de flujo de bytes, usaremos las clases abstractas “InputStream” y “OutputStream” que pertenecen al paquete java.io

InputStream para ver la información almacenada en un fichero.
OutputStream para escribir información en bytes.

Si lo hacemos como secuencia de caracteres, usaremos las clases “Reader” y “Writer” de java.io.

1.- Leer la información de un fichero de texto almacenado en el escritorio.

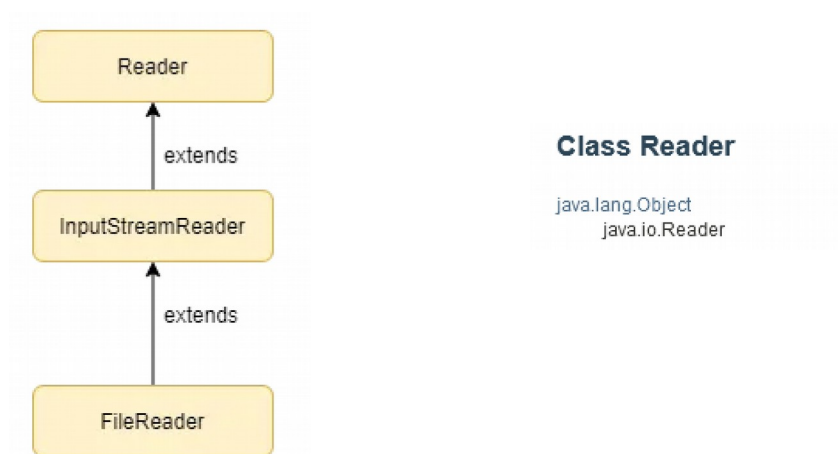
Creamos la siguiente estructura:



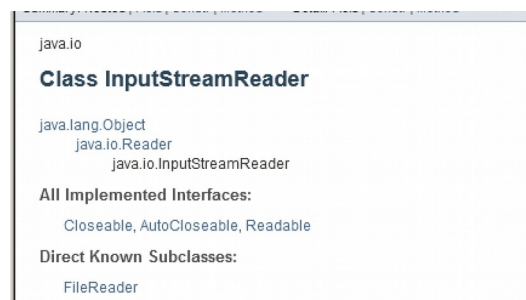
Lo primero que hay que hacer es indicar donde se encuentra nuestro fichero a leer.

Consultamos la Api de java para ver todo el sistema de herencia que parte de la clase Reader.

Reader pertenece al paquete java.io



Vemos las subclases y sobre todo , a nosotros nos interesa la subclase InputStreamReader.



Esta clase, implementa una subclase “FileReader” que es la que usaremos para indicar donde se encuentra el archivo.

Si leemos la información de InputStreamReader, esta clase nos dice:

```
public class InputStreamReader
    extends Reader
```

An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified `charset`. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

Each invocation of one of an InputStreamReader's `read()` methods may cause one or more bytes to be read from the underlying byte-input stream. To enable the efficient conversion of bytes to characters, more bytes may be read ahead from the underlying stream than are necessary to satisfy the current read operation.

For top efficiency, consider wrapping an InputStreamReader within a BufferedReader. For example:

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

Que es un puente entre Stream de bytes a caracteres de bytes, lo que hace es decodificar lo que sería un byte correspondiente a un carácter a su código de carácter (ascii , unicode)

Cuando pulsamos una tecla o queremos especificar una tecla en código, por ejemplo la tecla “A” en bytes sería una sucesión de “100011101”. Lo que hace la clase `InputStreamReader` es traducir esos bytes a su código correspondiente , por ejemplo la “A” (100011101) al código 105 por ejemplo.

Si vemos la información de `FileReader` y más concretamente los constructores que tiene

Constructor Summary	
Constructors	
Constructor and Description	
<code>FileReader(File file)</code>	Creates a new <code>FileReader</code> , given the <code>File</code> to read from.
<code>FileReader(FileDescriptor fd)</code>	Creates a new <code>FileReader</code> , given the <code>FileDescriptor</code> to read from.
<code>FileReader(String fileName)</code>	Creates a new <code>FileReader</code> , given the name of the file to read from.

Tiene sobrecarga de constructores. ¿Cuál usamos? , el que más se adapte a lo que estamos haciendo. Para nuestro ejemplo será el primero `FileReader(File file)`.

Si hacemos clic sobre este constructor para ver la información de él

FileReader
<pre>public FileReader(File file) throws FileNotFoundException</pre>
Creates a new <code>FileReader</code> , given the <code>File</code> to read from.
Parameters:
<code>file</code> - the <code>File</code> to read from
Throws:
<code>FileNotFoundException</code> - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

Observamos que lanza la exception `FileNotFoundException`,

java.io

Class FileNotFoundException

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.io.IOException
        java.io.FileNotFoundException
```

All Implemented Interfaces:

Serializable

```
public class FileNotFoundException
extends IOException
```

Signals that an attempt to open the file denoted by a specified pathname has failed.

This exception will be thrown by the `FileInputStream`, `FileOutputStream`, and `RandomAccessFile` constructors when a file with the specified pathname does not exist. It will also be thrown by these constructors if the file does exist but for some reason is inaccessible, for example when an attempt is made to open a read-only file for writing.

Vemos que esta exception hereda de `IOException`, por lo que estamos obligado a manejar dicha exception.

Como `FileNotFoundException` hereda `IOException`, si manejamos `IOException` no es necesario, en este caso manejar la exception `FileNotFoundException`. Probarlo en el ejercicio.

Probar a manejar solo `FileNotFoundException`, luego a maneja `IOException` y por último a manejar las dos.

Para leer el fichero, vamos a la API de java y vemos los métodos que tiene la clase `FileReader`.

Method Summary

Methods inherited from class java.io.InputStreamReader

`close`, `getEncoding`, `read`, `read`, `ready`

Methods inherited from class java.io.Reader

`mark`, `markSupported`, `read`, `read`, `reset`, `skip`

Methods inherited from class java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Vemos que hay varios métodos `read`. Unos heredan de una clase y otros de otra clase. Vemos el método que hereda de `InputStreamReader`.

read

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

Reads a single character.

Overrides:

`read` in class `Reader`

Returns:

The character read, or -1 if the end of the stream has been reached

Throws:

`IOException` - If an I/O error occurs

Comprobamos que este método devuelve un valor entero y lanza la exception IOException.

The character read, or -1 if the end of the stream has been reached

También nos indica que lo que devuelve es el código del carácter leído o -1 si hemos llegado al final de la secuencia de caracteres.



Es advertencia es debido a que tenemos abierto un flujo y no lo hemos cerrado.
Todo flujo debe ser cerrado con el método close().
Incluimos la instrucción “entrada.close()” después del bucle.

¿Porqué no incluimos esta instrucción en un finally?

No es necesario, porque el flujo se abre dentro del try y si todo va bien y lo cerrará cuando finalice de leer. Cosa distinta si abrimos el flujo antes del try, deberíamos incluir la clausula “finally” , ya que al estar fuera del try, el flujo se abriría tanto si encuentra el fichero como si no.

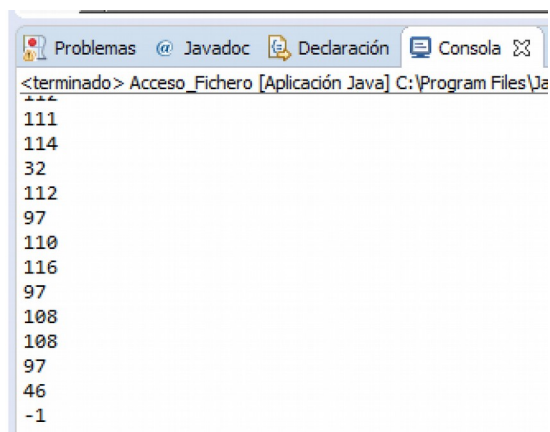
Otro inconveniente es que close() lanza una exception IOException por lo que hay que meterlo dentro de un try-catch.

Sería de la siguiente forma:

```
public void lee() {  
  
    //Declaramos una instancia de FileReader  
    FileReader entrada= new FileReader("C:\\Users\\Internet\\Desktop\\Ejercicio1.txt");  
  
    try {  
  
        //int c =entrada.read(); //Variable "c" para almacenar el código del carácter leído  
        int c=entrada.read();  
        while(c!=-1) {  
            char letra=(char)c;
```

```
almacenado                System.out.print(letra); // obtenemos todos los códigos unicode del texto
                            c=entrada.read();
                            }
                            } catch (IOException e) {
                                e.printStackTrace(); //Imprime el contenido de la pila
                                System.out.println("Archivo no encontrado"); // Indicamos información que nos de una pista
                            } finally {
                                try {
                                    entrada.close();
                                } catch (IOException e) {
                                    System.out.println("No se ha podido cerrar el flujo");
                                }
                            }
                        }
```

¿Qué salida obtendremos?



```
<terminado> Acceso_Fichero [Aplicación Java] C:\Program Files\Ja
111
114
32
112
97
110
116
97
108
108
97
46
-1
```

Sería todos los códigos correspondiente a los caracteres y el (-1) se correspondería al final de la cadena. Para obtener los caracteres, es necesario hacer un casting

2.- Como escribir caracteres en un fichero (Writer).

Clase Writer

Para escribir una frase en un fichero que no existe, usaremos la clase `FileWriter`, si vamos a la API de java. Esta clase tiene el siguiente sistema de herencia:

```
java.io
Class FileWriter
java.lang.Object
  java.io.Writer
    java.io.OutputStreamWriter
      java.io.FileWriter
All Implemented Interfaces:
  Closeable, Flushable, Appendable, AutoCloseable
```

Constructores:

Constructor Summary

Constructors

Constructor and Description

FileWriter(File file)

Constructs a FileWriter object given a File object.

FileWriter(File file, boolean append)

Constructs a FileWriter object given a File object.

FileWriter(FileDescriptor fd)

Constructs a FileWriter object associated with a file descriptor.

FileWriter(String fileName)

Constructs a FileWriter object given a file name.

FileWriter(String fileName, boolean append)

Constructs a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

Usaremos el primero donde solo debemos especificar un argumento de tipo archivo. Debemos indicarle el fichero donde queremos escribir.

FileWriter

```
public FileWriter(File file)
    throws IOException
```

Constructs a FileWriter object given a File object.

Parameters:

file - a File object to write to.

Este constructor lanza una excepción de tipo `IOException`, por lo que cuando instanciamos esta clase, debemos ponerla dentro de un try-catch.

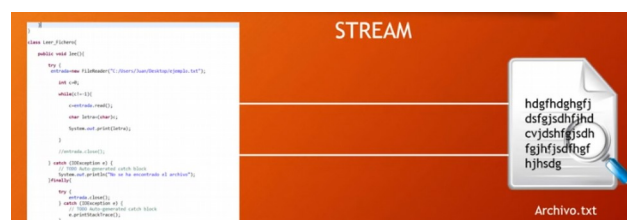
Si queremos escribir en un fichero existente, tenemos que usar otro constructor.

```
FileWriter escritura=new FileWriter("C:\\Users\\Internet\\Desktop\\Ejercicio2Escritura.txt", true);
```

En este constructor se le añade un “true”, esto lo que hace que en caso de que exista el fichero, añade el texto pertinente. Si quitamos el “true”, lo que hace es sobrescribir el fichero existente.

3.- Manejo de archivos. Acceso a Ficheros (Buffers)

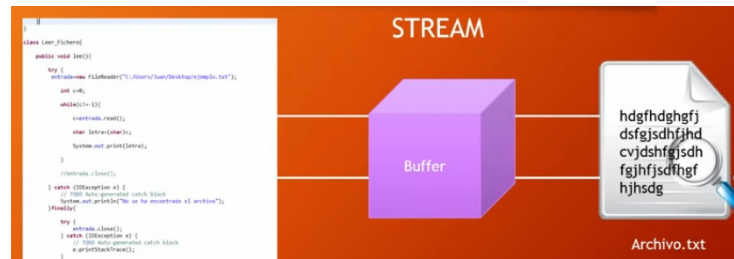
Para comunicarnos desde un programa java a un fichero externo, debemos de construir un Stream.



¿Qué sucede cuando el archivo externo es muy grande?

En este caso, el sistema visto anteriormente se vuelve poco eficiente porque cada vez que escribimos o leemos de un archivo de texto, estamos solicitando comunicación con ese archivo para leer o escribir un carácter y en el caso de que tengamos millones de caracteres, nuestro programa tendría que repetir esa misma operación millones de veces (consumiría mucho recurso y se volvería una tarea lenta).

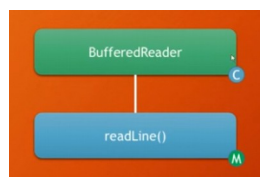
La solución a este problema de eficiencia es la creación de un Buffer.



Un buffer sería una memoria intermedia entre el programa de java y el fichero.
(Recordar “videos en streaming”)

El objetivo es que el programa java en lugar de acceder directamente al archivo, lo haga a la memoria interna (buffer), de forma que la información que contiene el fichero de texto, se vuelca en su totalidad al buffer y se almacena, una vez que el contenido del fichero está totalmente en el buffer, es el programa java el que accede al buffer para descargar poco a poco esa información. El buffer lo podemos usar tanto para leer como para escribir información en un archivo externo.

Para utilizar los buffer debemos usar la clase BufferedReader para leer información o BufferedWriter para escribir.



Además la clase BufferedReader tiene métodos interesantes para nuestra tarea como por ejemplo el método readLine() que lee línea a línea el texto existente en un archivo (devuelve una línea entera).

```
java.io
Class BufferedReader

java.lang.Object
  java.io.Reader
    java.io.BufferedReader

All Implemented Interfaces:
  Closeable, AutoCloseable, Readable

Direct Known Subclasses:
  LineNumberReader

public class BufferedReader
  extends Reader

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. For example,

    BufferedReader in
    = new BufferedReader(new FileReader("foo.in"));

will buffer the input from the specified file. Without buffering, each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.

Programs that use DataInputStreams for textual input can be localized by replacing each DataInputStream with an appropriate BufferedReader.
```

Viendo la descripción nos indica que podemos especificar el tamaño del buffer o bien usar el tamaño por defecto y que este es suficiente para la mayoría de los casos.

Constructores

Constructor Summary	
Constructors	
Constructor and Description	
<code>BufferedReader(Reader in)</code>	Creates a buffering character-input stream that uses a default-sized input buffer.
<code>BufferedReader(Reader in, int sz)</code>	Creates a buffering character-input stream that uses an input buffer of the specified size.

Tiene dos constructores uno donde nos pide que se le pase por parámetro una instancia de la clase `Reader` o bien que **herede** de la clase `Reader` y otro que además nos pide que le pasemos por parámetro el tamaño del buffer.

Si vemos los métodos,

Method Summary	
Methods	
Modifier and Type	Method and Description
<code>void</code>	<code>close()</code> Closes the stream and releases any system resources associated with it.
<code>void</code>	<code>mark(int readAheadLimit)</code> Marks the present position in the stream.
<code>boolean</code>	<code>markSupported()</code> Tells whether this stream supports the <code>mark()</code> operation, which it does.
<code>int</code>	<code>read()</code> Reads a single character.
<code>int</code>	<code>read(char[] cbuf, int off, int len)</code> Reads characters into a portion of an array.
<code>String</code>	<code>readLine()</code> Reads a line of text.
<code>boolean</code>	<code>ready()</code> Tells whether this stream is ready to be read.
<code>void</code>	<code>reset()</code> Resets the stream to the most recent mark.
<code>long</code>	<code>skip(long n)</code> Skips characters.

El método `readLine` devuelve un `String` y no un `char`.

readLine
<pre>public String readLine() throws IOException</pre>
Reads a line of text. A line is considered to be terminated by any one of a line feed (' <code>\n</code> '), a carriage return (' <code>\r</code> '), or a carriage return followed immediately by a linefeed.
Returns: A <code>String</code> containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached
Throws: <code>IOException</code> - If an I/O error occurs

Este método lanza una excepción de tipo IOException por lo que hay que capturarla; Además nos indica que una línea se considera terminada cuando encuentra (\n) un salto de línea o bien con (\r) o un retorno de carro. Cuando no encuentre línea, devolverá null.

La clase BufferedWriter es igual.

Ejemplo Construir un Buffer para leer la información que tenemos en un archivo externo.

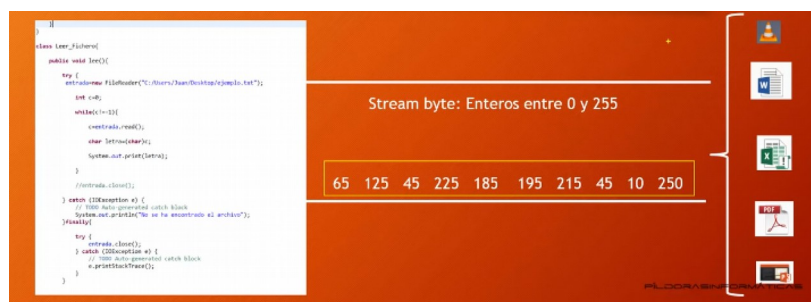
```
FileReader entrada= new FileReader("C:\\Users\\Internet\\Desktop\\Ejercicio1.txt");
```

```
BufferedReader mibuffer=new BufferedReader(entrada);
```

BufferedReader nos pide que le pasemos como parámetro una instancia de tipo Reader que en este caso es el objeto “entrada”, con esta instrucción lo que hacemos es almacenar en la memoria intermedia el fichero de texto que está en la instancia “entrada”

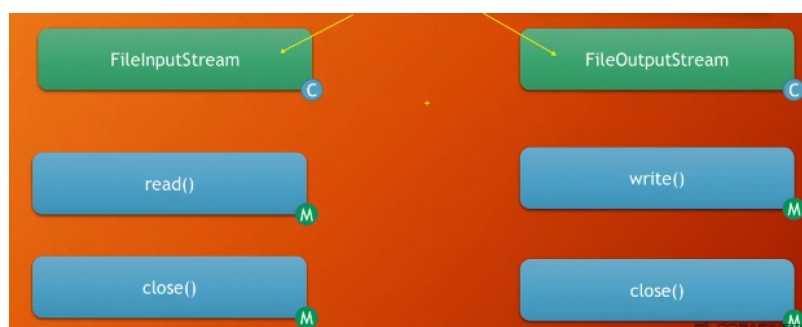
4.- Manejo de archivos. Streams Byte.

Esto tiene la utilidad para poder enviar o recibir cualquier tipo de archivo desde un programa java.



En java, cualquier tipo de archivo se puede convertir en bytes.

Para hacer esto, usaremos las clases FileInputStream y FileOutputStream



El manejo de estas clases es similar a las clases FileReader y FileWriter vista anteriormente.

Ejemplo:

Este ejemplo consistirá en leer un archivo cualquiera (en este caso lo haremos con un archivo de imagen) y escribiremos una copia en una carpeta cualquiera.

Para realizar esto, es necesario conocer cuantos bytes forma parte de la imagen. Esto lo haremos con un contador (a cada vuelta del bucle, lee un byte).

Una vez conocido los bytes que compone la imagen, con esos bytes debemos crear una imagen con ellos.

Para ello, necesitamos almacenar todos esos bytes en algún sitio para posteriormente crear el archivo con esos bytes.

Usaremos un Array para almacenar los bytes.