Ciclo de Desarrollo de Aplicaciones Multiplataforma

Grado Superior

U.T.7: LECTURA/ESCRITURA DE LA INFORMACIÓN

Módulo de Programación

Departamento de Informática

I.E.S. Monte Naranco

7.1. ENTRADA/SALIDA DE DATOS EN JAVA

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 mediante lo que denominamos Streams (flujo de datos). Un stream es un canal o vía de comunicación entre el programa y la fuente o destino de los datos. La vinculación de este stream al dispositivo físico lo hace java. Es quien sabe si tiene que tratar con el teclado, monitor, ficheros, la red, etc, gracias al uso de las diferentes clases que posee. Los Streams no son bidireccionales, es decir, debemos crear uno para lectura y otro distinto para escritura.

La información se traslada en serie (una detrás de otra) a través de este canal. Por tanto, el flujo representa una secuencia de bytes entre el programa y la fuente o el destino de información. Así a la hora de escribir o leer datos, en o desde un fichero, haremos uso de un flujo, de forma que cuando queramos escribir ubicaremos el dato en el flujo, y este lo redirigirá hacia el dispositivo correspondiente y a la inversa, si deseamos leer información, el dispositivo ubicará la información en el flujo para que llegue al programa que debe procesarla.

Los Streams sirven para:

1. Enviar información desde un programa java a un sitito remoto en la red, a través de internet (tcp/ip).
2. Acceder a ficheros externos para poder leer la información que contienen, o bien para poder escribir en ellos información.
3. Escribir desde el teclado, enviar al monitor...
4. Leer información a través de un puerto serie (ratón).

Tipos de flujos o streams

Se pueden clasificar en base a la información que fluye entre el dispositivo y el programa, y en base a su funcionalidad.

En base a la información que fluye

1. **Flujo de bytes o binarios (ByteStreams):** Se usa para intercambiar datos binarios entre el dispositivo de entrada/salida y el programa. Por ellos viajan bytes. El tratamiento del flujo de bytes viene gobernado por dos clases abstractas: InputStream y OutputStream. Cada una de ellas tiene varias subclases que controlan las diferencias entre los distintos dispositivos de entrada/salida que se pueden usar.

2. **Flujo de caracteres (CharacterStreams):** Se usa para intercambiar texto (caracteres) entre el dispositivo de entrada/salida y el programa. Por ellos viajan caracteres Unicode (cada carácter Unicode utiliza 2 bytes). El flujo de caracteres viene gobernado por dos clases abstractas: Reader y Writer, que manejan los caracteres Unicode.

En base a su funcionalidad

1. **Flujo de origen o destino de datos:** Están conectados directamente con la fuente o destino de los datos.
2. **Flujo de procesamiento:** Son pasos intermedios en el proceso de la información.

Operaciones a realizar en un stream

Cuando se trabaja con streams, son tres las operaciones a realizar:

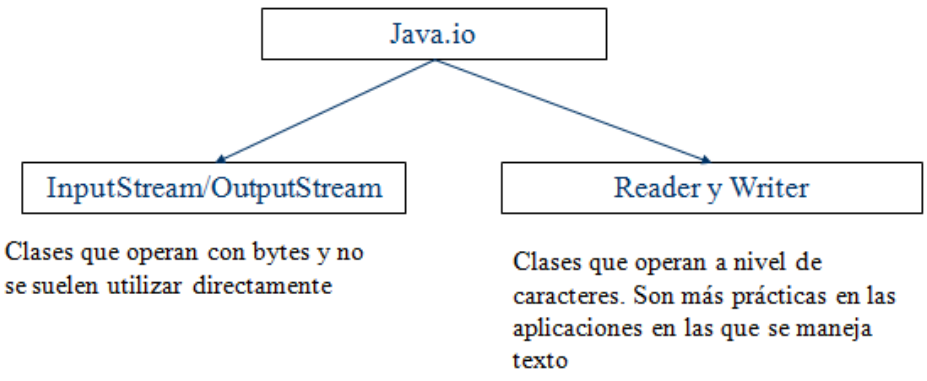
1. **Crear y abrir el stream (por lectura o escritura):** Se debe crear un objeto de la clase Stream invocando al constructor correspondiente y abrirlo, para así relacionar nuestro programa con el exterior (por ejemplo con un fichero almacenado en disco a través de su nombre).

2. **Escritura o lectura:** Esta operación se debe realizar mientras haya información para escribir o para leer.

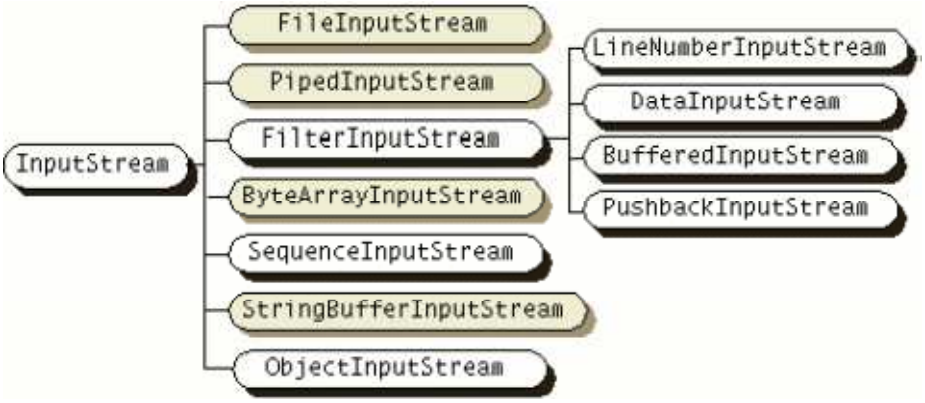
3. **Cerrar el stream:** Se cierra la conexión con el exterior y se liberan los recursos que está usando la misma.

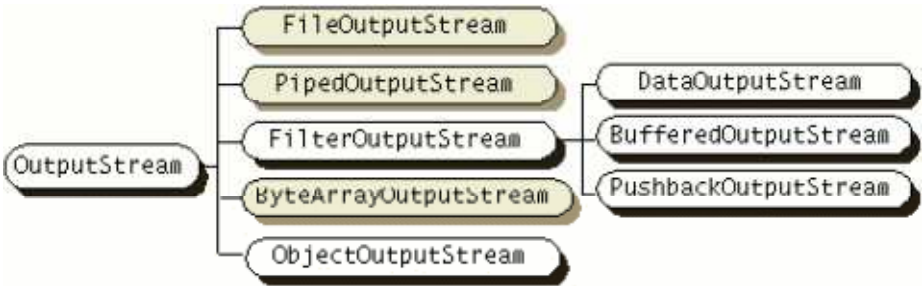
7.2. CLASES DE JAVA RELATIVAS A FLUJOS. JERARQUÍA DE CLASES

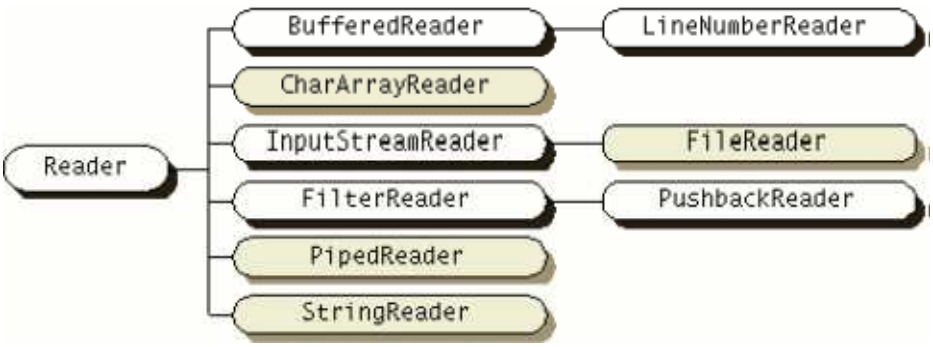
El package java.io contiene las clases necesarias para la comunicación del programa con el exterior, es decir, para trabajar con flujos de datos. Dichas clases se dividen en dos categorías dependiendo si manejan flujos de caracteres o de bytes.

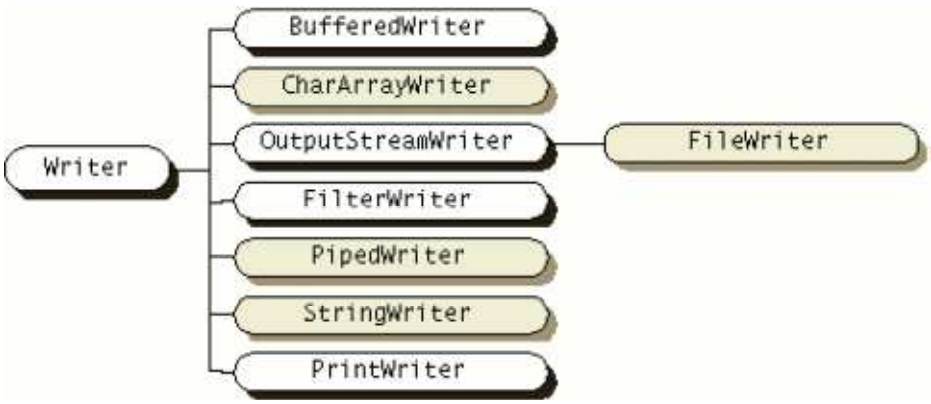


A su vez dichas categorías se dividen en dos grupos en función de que sean clases para leer o clases para escribir (las clases que contienen la palabra Input se refieren a la lectura de datos, mientras que aquellas que contienen la palabra Output indican escritura).









Las clases de fondo sombreado definen de dónde o a dónde se están enviando los datos, es decir, el dispositivo que conecta el stream. Las demás clases (fondo blanco), añaden características particulares a la forma de enviarlos. Combinando ambas se obtiene el comportamiento deseado.

7.3. ALMACENAMIENTO DE INFORMACIÓN EN FICHEROS

Necesidad de manejar ficheros externos

Hasta ahora estuvimos almacenando información en arrays, colecciones y variables. Tanto las variables como los arrays y colecciones se almacenan en la memoria Ram del ordenador. Esta memoria es volátil, lo que significa que cuando se apaga el ordenador la información allí contenida se pierde. Además tiene un tamaño limitado, aunque es rápida.

Si queremos almacenar la información de manera permanente para acceder a ella cuando queramos, o tratar grandes volúmenes de datos, debemos hacerlo en un dispositivo físico (memoria secundaria) como puede ser un disco duro, pendrive, etc. Las estructuras de datos utilizadas para almacenar la información en dispositivos físicos, serán los ficheros o las bases de datos.

Clasificación de los ficheros según el acceso a la información

Podemos clasificar los ficheros siguiendo diferentes criterios, uno de ellos es en función del acceso a la información. Según este criterio se clasifican en:

- Secuenciales.
- Aleatorios o directos.
- Secuencial indexados.

Nos vamos a centrar en los ficheros secuenciales que son los que se usan en java de forma general.

7.3.1. FICHEROS SECUENCIALES

El acceder de forma secuencial a la información que contiene un fichero, implica tener que leer el fichero desde el principio e ir pasando por toda la información que contiene para poder llegar a un determinado dato.

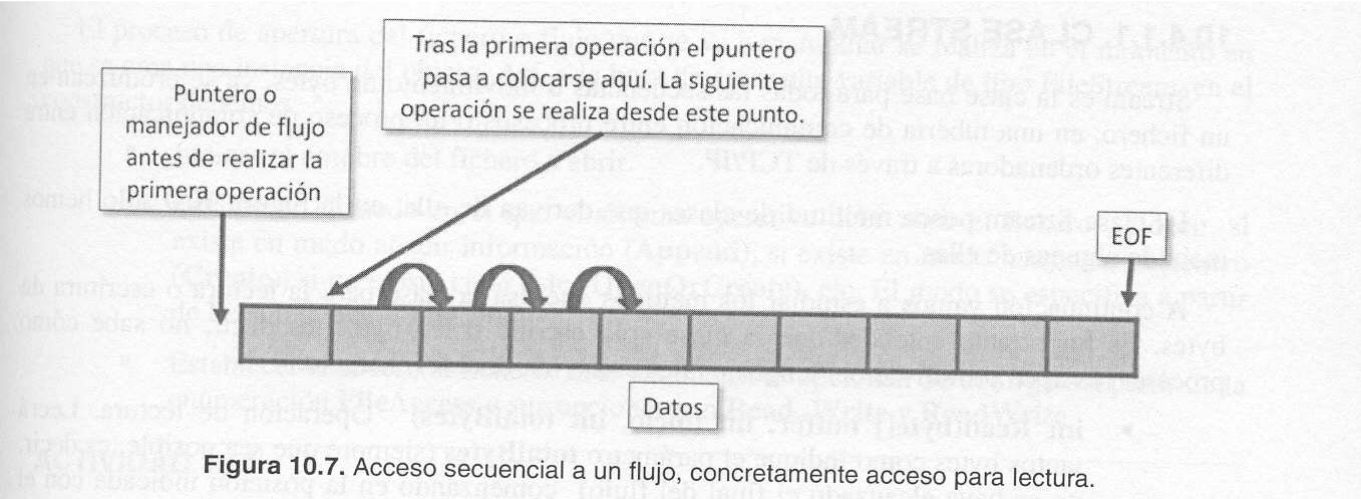


Figura 10.7. Acceso secuencial a un flujo, concretamente acceso para lectura.

Solo es posible realizar una operación de lectura o escritura a la vez, es decir, cuando el fichero está siendo leído no es posible realizar una operación de escritura, así como cuando el fichero está siendo escrito no puede llevarse a cabo ninguna operación de lectura.

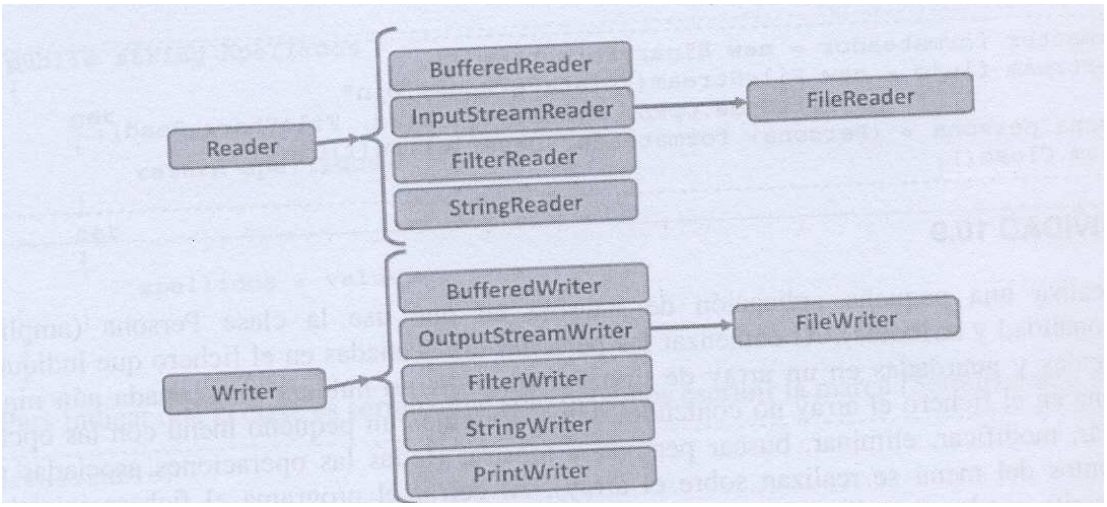
7.3.1.1. FICHEROS DE TEXTO

Un fichero de texto es un fichero legible por nosotros. Está formado por secuencias de caracteres organizados en líneas de igual o distinta longitud. Cada línea sería un registro del fichero.

```
Atún@Alimentación@Lata 92 grs@0.59
Aceitunas@Alimentación@Lata 350 grs@0.8
Chapata@Panadería@Unidad@0.75
```

Los ficheros de texto se pueden crear desde un programa en java y leer con cualquier editor de textos, o bien crear con un editor de textos y leer desde un programa java, o usar un programa java tanto para leerlos como para escribirlos.

Las clases para poder manejar archivos de texto desde un programa java son:



De todas ellas utilizaremos FileReader y FileWriter, que nos van a permitir instanciar las clases abstractas Reader (permite leer ficheros de caracteres) y Writer (nos permite escribir la información en caracteres en un archivo), pertenecientes al paquete java.io.

Clase FileReader

Cuando instanciamos esta clase, lo que hace el programa java es crear un flujo de caracteres (crea un objeto de la clase invocando al constructor correspondiente) y abrirlo para lectura. Una vez abierto podremos comunicarnos con el fichero (que se envía como parámetro) para leer la información que contiene de forma secuencial, es decir, carácter a carácter para procesarla. Al final debemos cerrar el flujo de caracteres.

Para crear un objeto de la clase FileReader podemos utilizar los constructores siguientes:

- **FileReader (String Fichero) :** Recibe como parámetro el nombre del fichero que se quiere abrir para leer.

  Ejm: FileReader FicheroOrigen =new FileReader ("primero.txt"); // El fichero primero.txt se encuentra en el directorio raíz de la carpeta que contenga nuestro proyecto java.

  Ejm: FileReader FicheroOrigen =new FileReader ("C:/Users/Mj/Desktop/FichJava/primero.txt"); // El fichero primero.txt se encuentra almacenado en el directorio FichJava.

  Ejm: FileReader FicheroOrigen =new FileReader (".\src\test\primero.txt"); // El fichero primero.txt se encuentra en la carpeta del paquete test del proyecto.

- **FileReader (File objFichero):** Recibe un objeto File, el cual representa el fichero con el que queremos trabajar.

Estos constructores lanzan la excepción **FileNotFoundException**, si el fichero no existe cuando se quiere acceder a él.

La clase FileReader, implementa entre otros los siguientes métodos heredados: read, método sobrecargado, ready y close.

- **int read():** Nos permite leer un carácter del archivo de texto. Este método nos devuelve un entero (el código del carácter leído) correspondiente al siguiente carácter que hay después de donde esté situado el puntero del fichero. Dicho puntero se va moviendo secuencialmente según vayamos leyendo los caracteres. Cuando llega al final del archivo devuelve -1.

- **int read (char[ ] cbuf):** No lee un solo carácter, sino que lee hasta "**cbuf.length**" caracteres del fichero, depositándolos en el array **cbuf** que se envía por parámetro. Este método devuelve el número de caracteres leídos, o -1 si no hay ningún carácter más que leer (se llegó al final del fichero).

- **int read (char [ ] cbuf, int offset, int length):** Lee hasta "**length"** caracteres del fichero, depositándolos en el array cbuf, a partir de offset. Este método devuelve el número de caracteres leídos.

- **boolean ready():** Nos indica si el flujo está listo para leer del mismo, es decir, si la siguiente llamada a read devolverá algo.

- **void close():** Nos permite cerrar el Stream abierto. Su uso se aconseja para liberar recursos asociados con el Stream (no se podrá invocar a read(), ready(),ni a ningún otro método).

Estos métodos lanzan la excepción **IOException,** si ocurre un error de entrada/salida.

Clase FileWriter

Cuando instanciamos esta clase, lo que hace el programa java es crear un flujo de caracteres (crea un objeto de la clase invocando al constructor correspondiente) y abrirlo para escritura. Una vez abierto podremos comunicarnos con el fichero (que se envía como parámetro) para guardar o escribir información de forma secuencial, es decir, carácter a carácter. Al final debemos cerrar el flujo de caracteres.

Para crear un objeto de la clase FileWriter podemos utilizar los constructores siguientes:

- **FileWriter (String FicheroDestino) :** Recibe como parámetro el nombre del fichero que se quiere abrir para escribir.

  Ejm: FileWriter FicheroDestino =new FileWriter ("salida.txt");

- **FileWriter (File Fichero):** Recibe un objeto File, el cual representa el fichero con el que queremos trabajar.

Con estos constructores el fichero se crea, si ya existe su contenido se pierde. Si lo que necesitamos es abrir un fichero de texto existente sin perder su contenido y añadir nueva información al final, utilizaremos los constructores siguientes:

- **FileWriter (String FicheroDestino, boolean append)**

  Ejm: FileWriter FicheroDestino =new FileWriter ("/salida.txt", true);

- **FileWriter (File objFichero, boolean append)**

Si append es true, significa que los datos se van a añadir a los existentes. Si es false, los datos existentes se pierden.

Estos constructores pueden lanzar la excepción **FileNot FoundException**, si se produce algún problema cuando se abre el stream (disco lleno, protegido, etc).

La clase FileWriter, implementa entre otros los siguientes métodos heredados: write, método sobrecargado, flush y close.

- **void write (int b):** Escribe el carácter que recibe por parámetro en el fichero.
- **void write (char[ ] cbuf):** Escribe "**cbuf.length"** caracteres en el fichero, tomándolos del Array "**cbuf".**
- **void write (char [ ] cbuf, int off, int len):** Escribe **"len"** caracteres en el fichero, obteniéndolos del array **"cbuf",** a partir de **"off".**
- **void write (String str):** Escribe una cadena de caracteres en el fichero.
- **void write (String str, int off, int len):** Escribe "**len**" caracteres en el fichero, obteniéndolos de una cadena de caracteres, a partir de "**off**".
- **void flush():** Permite vaciar los caracteres que quedan en el stream y los escribe en el fichero.

- **void close():** Nos permite cerrar el Stream abierto. Su uso se aconseja para liberar recursos asociados con el Stream (no se podrá invocar a write(), ni a ningún otro método).

Estos métodos lanzan la excepción **IOException,** si ocurre un error de entrada/salida.

- **String getEncoding():** Retorna el carácter codificado usado en el Stream.

El método write no escribe la información en el archivo físico hasta que no se cierra el flujo de datos con el método close o se use el método flush.

Clase PrintWriter

Es una clase con una amplia variedad de métodos, para hacer más cómoda la escritura de diferentes tipos de elementos java.

Para crear un objeto de la clase PrintWriter podemos utilizar los constructores siguientes:

- **PrintWriter (String fileName) :** Recibe como parámetro el nombre del fichero que se quiere abrir para escribir.

  Ejm: PrintWriter FicheroDestino =new PrintWriter ("salida.txt");

- **PrintWriter (File file):** Recibe un objeto File, el cual representa el fichero con el que queremos trabajar.

Estos constructores pueden lanzar la excepción **FileNot FoundException**, si se produce algún problema cuando se abre el stream (disco lleno, protegido, etc).

- **PrintWriter (Writer out):** Recibe por parámetro una instancia de la clase.

  Ejm:

  ```
  PrintWriter entrada=new PrintWriter(new FileWriter("/salida.txt"));
  ```

  Este constructor no lanza ninguna excepción.

Además de los métodos heredados de Writer como, write, flush y close vistos anteriormente, tiene otros propios como:

- **void print(Tipo t):** Escribe un valor t, donde Tipo puede ser char, boolean char[ ], double, int, String, long, float u Object.
- **void println():** Produce un salto de línea.
- **void println(Tipo t):** Escribe un objeto de tipo boolean, char, char[ ], String, double, float, int, long u Object y después salta línea.

Estos métodos no lanzan ninguna excepción.

Buffers

Cuando los archivos de texto son pequeños podemos leerlos o escribirlos carácter a carácter, pero cuando los archivos son grandes, leerlos o escribirlos carácter a carácter es poco eficiente y lento, puesto que consume muchos recursos. La solución pasa por crear un buffer (es el modo habitual).

Un buffer (se le denomina también filtro), es una especie de memoria interna que se coloca entre el programa de java y el archivo externo a donde queremos acceder, de tal forma que la información contenida en el archivo de texto se vuelque o almacene íntegramente en el buffer para ser leída por el programa, o se vaya depositando en el buffer para ser escrita en el archivo. De esta forma se gana en eficiencia y rapidez.

PROGRAMA JAVA	STREAM	BUFFER	STREAM	ARCHIVO DE TEXTO

Para utilizar un buffer debemos usar la clase BufferedReader (permite crear un buffer para lectura), o BufferedWrite (permite crear un buffer para escritura). El tamaño del buffer se puede especificar, o podemos utilizar el que viene definido por defecto.

Clase BufferedReader

Para crear un buffer de lectura, es decir un objeto de la clase BufferedReader, podemos utilizar los **constructores** siguientes:

- **BufferedReader (Reader in, int tamaño):** Recibe por parámetro una instancia de la clase Reader y el tamaño del buffer.

  Ejm:

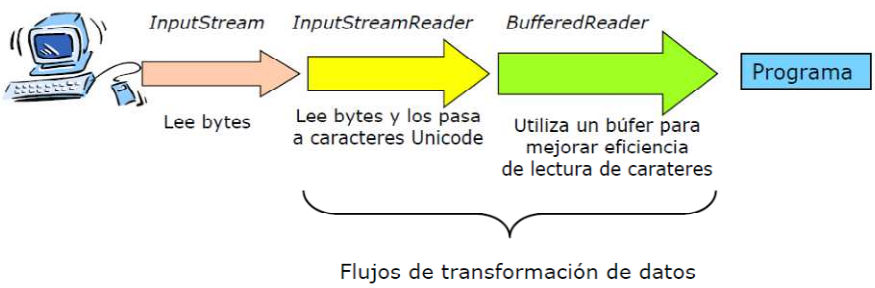
  ```
  FileReader fr=new FileReader("/salida.txt");
  BufferedReader entrada=new BufferedReader(fr,1000);
  ```

  Este constructor lanza la excepción **IllegalArgumentException,** si el tamaño del buffer es menor o igual a cero. Esta excepción es de tipo RuntimeException, lo que significa que no es obligatorio capturarla.

- **BufferedReader (Reader in):** Recibe por parámetro una instancia de la clase Reader. El tamaño del buffer utilizado será el definido por defecto.

  Ejm:

  ```
  BufferedReader entrada=new BufferedReader(new FileReader("/salida.txt"));
  ```



Además de los métodos heredados de Reader como, read, ready y close vistos anteriormente, tiene otros propios como:

- **String readLine():** Lee una línea de texto que será devuelta. Una línea de texto se considera terminada cuando hay un salto de línea (\n) o un retorno de carro (\r). Cuando el texto se ha leído totalmente, el método en lugar de devolver la línea devuelve el valor nulo (null). Este método es muy cómodo para leer ficheros línea a línea.

El método readLine lanza la excepción **IOException,** si ocurre un error de lectura.

Clase BufferedWriter

Para crear un buffer de escritura, es decir un objeto de la clase BufferedWriter, podemos utilizar los **constructores** siguientes:

- **BufferedWriter (Writer out, int tamaño):** Recibe por parámetro una instancia de la clase Writer y el tamaño del buffer.

  Ejm:

  ```
  FileWriter fr=new FileWriter("/salida.txt");
  BufferedWriter entrada=new BufferedWriter(fr,1000);
  ```

  Este constructor lanza la excepción **IllegalArgumentException,** si el tamaño del buffer es menor o igual a cero. Esta excepción es de tipo RuntimeException, lo que significa que no es obligatorio capturarla.

- **BufferedWriter (Writer out):** Recibe por parámetro una instancia de la clase Writer. El tamaño del buffer utilizado será el definido por defecto.

  Ejm:

  ```
  BufferedWriter entrada=new BufferedWriter(new FileWriter("/salida.txt"));
  ```

Además de los métodos heredados de Writer como, write, flush y close vistos anteriormente, tiene otros propios como:

- **newLine():** Permite escribir un separador de líneas.

El método newLine lanza la excepción **IOException,** si ocurre un error de escritura.

7.4. MANIPULAR FICHEROS Y DIRECTORIOS. CLASE FILE

Las clases utilizadas en los apartados anteriores, nos permiten el tratamiento del contenido de los ficheros.

Para manipular ficheros y directorios a nivel del sistema operativo (crearlos, borrarlos, obtener información....) , debemos usar la clase **File.**

Un objeto de la clase File representa un archivo o directorio. Crear un objeto File, no significa que deba existir ese archivo o directorio o que la ruta indicada sea correcta. En cualquiera de estos casos no se lanzan excepciones que haya que capturar de forma obligatoria, ni tampoco serán creados.

Al especificar una ruta, podemos:

- Indicar el nombre del fichero sin indicar el camino. En este caso se buscará el fichero en la carpeta del proyecto.
- Indicar la ruta y el fichero, pero sin empezar por "/" (path relativo).
- Camino completo.

La clase File tiene varios constructores que nos permiten crear objetos:

- **File (String nombreFichero|directorio):** Recibe en la instanciación del objeto el camino donde está el fichero o directorio junto con el nombre. Por defecto, si no se indica camino, lo buscará en la carpeta del proyecto (la ruta puede ser absoluta o relativa).

  Ejemplos:

  - ✓ File f=new File("personas.dat") /*Se supone que el fichero se encuentra en el directorio de trabajo al no indicar la ruta*/
  - ✓ File f=new File("ficheros/personas.dat") /*Ruta relativa al directorio actual de trabajo*/

- **File (String camino, String nomFiche|directorio):** Recibe en la instanciación como primer parámetro la ruta donde está el objeto, y el nombre del fichero o directorio como segundo parámetro.

  Ejemplos:

  - ✓ File f=new File("ficheros", denominacion) /*denominacion es una variable de tipo String, cuyo valor es el nombre de un fichero o directorio asignado previamente*/
  - ✓ File f=new File("/ficheros",personas.dat") /*El fichero personas.dat está en el directorio ficheros dentro del directorio actual*/

- **File (File camino, String nomFiche|directorio):** Recibe en la instanciación como primer parámetro, un objeto de tipo File que hace referencia a un directorio, y el nombre del fichero o directorio como segundo parámetro.

  Ejemplo:

  ```
  File ruta=new File("/ficheros")
  File f=new File (ruta,"personas.dat") /*Se crea un objeto File, cuya ruta se indica a través de otro objeto File*/
  ```

Los métodos más representativos de la clase son:

- **getAbsolutePath():** Nos devuelve un String que representa la ruta absoluta de un archivo o directorio.
- **getAbsoluteFile():** Hace referencia a la manipulación de rutas.
- **getPath():** Hace referencia a la manipulación de rutas.
- **getCanonicaPath():** Hace referencia a la manipulación de rutas.
- **getCanonicalFile():** Hace referencia a la manipulación de rutas.
- **exists():** Nos devuelve un boolean, que indica si un archivo o directorio existe en la ruta especificada.
- **list():** Devuelve un array de String, con los nombres de archivos y directorios que se encuentran en el directorio.
- **isDirectory():** Devuelve un boolean que nos indica si lo que estamos examinando es un directorio o no.
- **isFile():** Devuelve un boolean que nos indica si lo que estamos examinando es un fichero o no.
- **canRead():** Devuelve un boolean que indica si el archivo o directorio que estamos examinando es de lectura o no lo es.
- **canWrite():** Devuelve un boolean que nos indica si el archivo o directorio que estamos examinando es de escritura o no lo es.
- **canExecute:** Devuelve un boolean que nos indica si el archivo se puede ejecutar.
- **delete():** Borra un archivo o directorio. Devuelve un boolean.
- **length():** Devuelve un long que almacena el tamaño del archivo en bytes.
- **lastModified():** Devuelve un long que almacena la fecha de la última modificación.
- **renameTo(File):** Cambia el nombre del fichero. Devuelve un boolean.
- **mkdir():** Crea un directorio. Devuelve un boolean.
- **createNewFile():** Crea un archivo si no existe. Si existe no lo vuelve a crear. Este método lanza una excepción de tipo IOException que hay que capturar. Devuelve un boolean.