élogos

UTILIZACIÓN DE LAS LIBRERÍAS BÁSICAS DE JAVA

élogos

UTILIZACIÓN DE LAS LIBRERÍAS BÁSICAS DE JAVA

Trabajando con streams Java. Flujos y Ficheros

Flujos de memoria

Está representado por dos clases:

- **ByteArrayInputStream**, sirve para trabajar con array de bytes de forma interna. Permite leer los datos de un array como si fuera un Stream.

Tiene el siguiente constructor:

```
ByteArrayInputStream (byte [] b)

String str = "Mensaje a impresora."
ByteArrayInputStream in = new ByteArrayInputStream(str.getBytes());
imprime (in);

void imprime (InputStream in) { . . . }
```

- **ByteArrayOutputStream**, permite actuar como un buffer en memoria que se usa antes de enviarlo al dispositivo.

El constructor es el siguiente:

```
ByteArrayOutputStream()

byte [] <ByteArrayOutputStream>.toByteArray()
```

Devuelve los datos introducidos en la estructura de memoria generada.

String <ByteArrayOutputStream>.toString()

Devuelve los datos de dentro de la estructura y los introduce dentro de un String.

Void <ByteArrayOutputStream>.writeTo(OutputStream os)

Envía los datos del buffer al dispositivo que se le indique. Al ser parámetro clase base se le puede enviar a cualquier derivada.

Manejo de Ficheros

- File

La clase que define los ficheros es File. File representa un fichero o directorio dentro del sistema de ficheros. Los constructores de File son los siguientes:

File (String filename)
File (String dir, String filename)
File (File dir, String filename)

```
File f1  = new File ("c:\\windows\\win.exe");
File dir = new File ("c:\\windows");
File f2  = new File (dir,"win.exe");
```

Tiene las siguientes constantes a nivel de clase y contiene respectivamente los símbolos que se utilizan para separar los paths dependiendo de la plataforma que se utilice.

static final String <File>.pathSeparator > (;)
static final String <File>.separator > (\)

```
File f2 = new File ("c:\\windows\\win.exe"); // path absoluto
File f3 = new File (".\\datos\\bd.dat"); // path relativo
```

String <File>.getPath()

Devuelve el texto del path que anteriormente se le ha pasado al constructor de File.

String <File>.getAbsolutePath()

Devuelve el path completo

```
File f1 = new File (".\\windows\\win.exe");
System.out.println(f1.getAbsolutePath());
        // c:\java\..\windows\win.exe
```

String <File>.GetCanonicalPath() throws IOException

Devuelve el path normalizado. Lanza excepciones ya que comprueba la existencia del fichero o directorio dentro de los sistemas de ficheros.

boolean <File>.isFile()	Devuelve true si es fichero.
boolean <File>.isDirectory()	Devuelve true si es directorio.
boolean <File>.exists()	Devuelve true si existe el fichero o directorio.
boolean <File>.canRead()	Si permite la lectura del fichero.

boolean <File>.canWrite()	Si permite la escritura del fichero.
long <File>.length()	Tamaño del fichero en bytes.

boolean <File>.mkDir()

Intenta crear un directorio con el nombre que se le pasa en el constructor. Si lo crea correctamente devuelve true.

boolean <File>.renameTo (String name)	Renombra el fichero o directorio.
boolean <File>.delete()	Elimina el fichero o directorio.

- FileInputStream y FileOutputStream

Ambas clases sirven para acceder al contenido de los ficheros.

FileInputStream, se crea a partir del nombre del fichero que se va a tratar. Realiza la lectura del fichero. Los constructores son los siguientes:

FileInputStream (String Filename) throws FileNotFoundException
FileInputStream (File Filename) throws FileNotFoundException

int <InputStream>.read() throws IOException

Realiza la lectura del fichero.

Protected <FileInputStream>.finalize() throws IOException

Redefine este método que procede de la clase Object. Este método realiza una llamada a close() antes de hacer la recogida de basura. De todas formas es mejor utilizar close() cuando se ha terminando de trabajar con el fichero.

FileOutputStream, realiza la escritura de los datos del fichero. Tiene el siguiente constructor:

FileOutputStream (String Filename) throws IOException

Con este constructor se empieza desde el principio en el fichero, aunque el fichero tenga datos.

FileOutputStream(String Filename, boolean append)
throws IOException

Con este constructor se añadirá información en el fichero. Para ello se pondrá a true la variable append. Si se pone false funcionará como el anterior constructor.

```
import java.io.*;

public class AccesoFichero {
public static void main (String [] args) throws IOException {

   // se crea el fichero de salida.
   FileOutputStream fos = new FileOutputStream("c:\\vem\\text.txt");

   // Cadena a escribir en el fichero.
   String str = "Texto de prueba";

   //Proceso de escritura.
   for (int i = 0; i < str.length(); i ++)
 fos.write(str.charAt((i)));
fos.close(); // Se cierra el fichero de salida.
```

```
    // se crea el fichero de entrada.
      FileInputStream fis = new FileInputStream("c:\\vem\\text.txt");
      int tamano = fis.available();/Longitud del fichero.
      System.out.println("Hay un tamaño de "+tamano+"bytes disponibles");
      byte [] B = new byte [tamano];
      fis.read(B); // Lectura del fichero.
      System.out.println ("Se leyó el texto: "+new String(B));
      fis.close(); // Se cierra el fichero de entrada.
    } // fin de main.
} // fin de clase.
```

- SequenceInputStream

Esta clase permite concatenar varios Streams en uno solo, de forma que cuando recorremos el Stream resultante se leerá consecutivamente uno detrás de otro. Para concatenar tienen dos constructores:

```
SequenceInputStream (InputStream s1, InputStream s2)
```

Concatena dos cadenas.

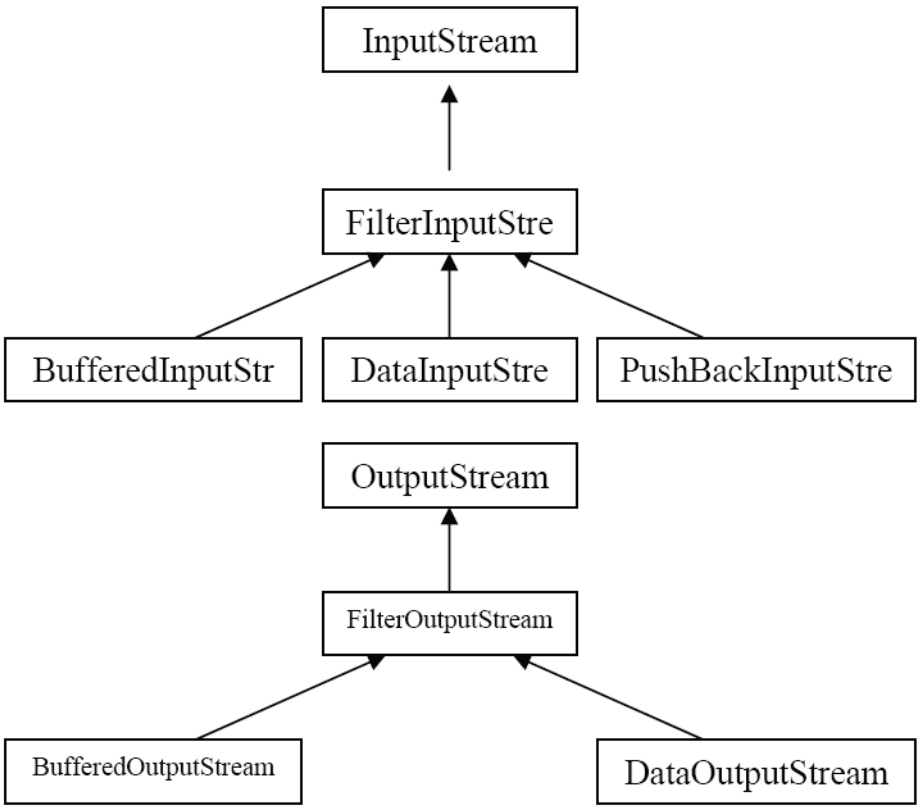
```
SequenceInputStream (Enumeration)
```

Concatena a partir de una lista de objetos InputStream.

```
Interface Enumeration {
  public boolean hasMoreElements(); //Controla si hay mas elementos

  public Object nextElement();
}
```

Filtro de Streams

Los filtros van a permitir modificar los datos que están dentro del Stream. Para construir filtros se tienen las siguientes clases:



Estas clases no trabajan con dispositivos finales sino que trabajan sobre Streams ya creados. En el constructor de cada una de las clases se recibe un Stream de la clase base (InputStream / OutputStream).

Para realizar el acceso de estas clases se utilizan los siguientes atributos que apuntan al Stream ya creado:

protected InputStream <FileInputStream> in;

protected OutputStream <FileOutputStream> out;

Los atributos son protected para que sólo puedan acceder las clases FilterOutputStream y FilterInputStream y sus derivadas.

- BufferedInputStream y BufferedOutputStream

Estas clases sirven para poner un buffer (memoria RAM) tanto para los Streams de entrada como para los de salida antes de pasarlo al dispositivo final.

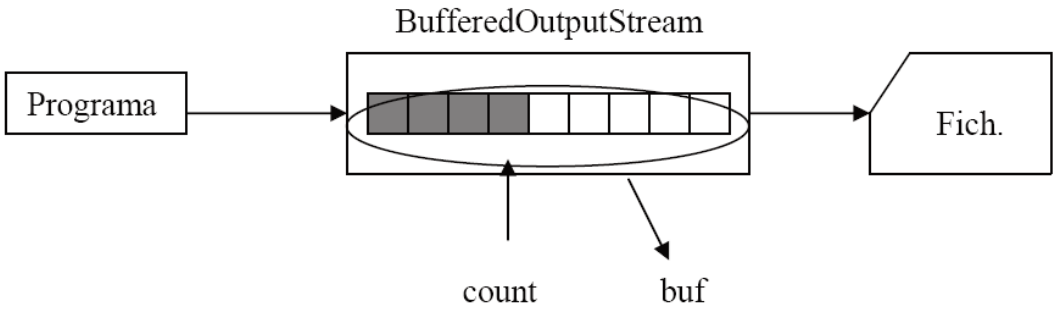
En el constructor de estas clases se le pasa el Stream y al Stream se le añade un buffer:

protected byte [] <BufferedOutputStream> buf;

Array de datos a cargar en el buffer.

protected int <BufferedOutputStream> count;

Indica hasta dónde está cargado el buffer.



Cuando el buffer está lleno se envían automáticamente los datos cargados al fichero.

Para forzar que se vacíe de información el buffer se tiene el método flush() de la clase base.

protected [] byte <BufferedInputStream> buf;

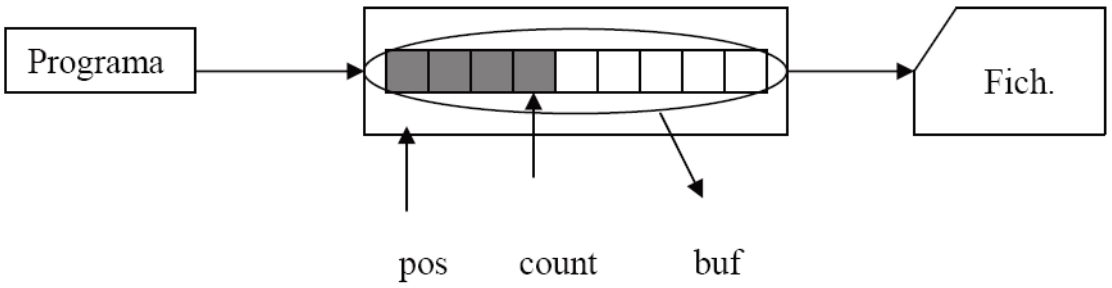
Array de datos del buffer.

protected int <BufferedInputStream> count;

Indica hasta dónde está cargado el buffer.

protected int <BufferedInputStream> pos;

Indica la siguiente posición a liberar del buffer.



Utilizar un buffer implica una optimización en el tiempo de acceso al fichero.

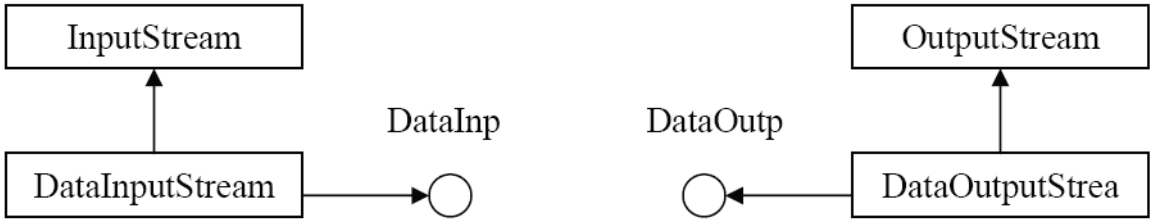
```
FileInputStream     f1  = new FileInputStream ("datos.txt");
BufferedInputStream bf1 = new BufferedInputStream (f1);

// Primeramente se crea el Stream y luego el buffer asociado.
```

En teoría está bien aplicarlo, pero en la práctica los Sistema Operativos ya tienen buffers implementados, entonces el rendimiento casi es el mismo.

De todas formas habrá casos en que utilizar buffers sea interesante.

- DataInputStream y DataOutputStream

Son filtros que permiten solucionar el problema de escribir y leer datos que no sean del tipo fundamental byte.



```
Interface DataInput {
        boolean readBoolean() throws IOException;
    int readInt() throws IOException;
    . . .
}
Interface DataOutput {
        boolean writeBoolean() throws IOException;
    int writeInt() throws IOException;
    . . .
}
```

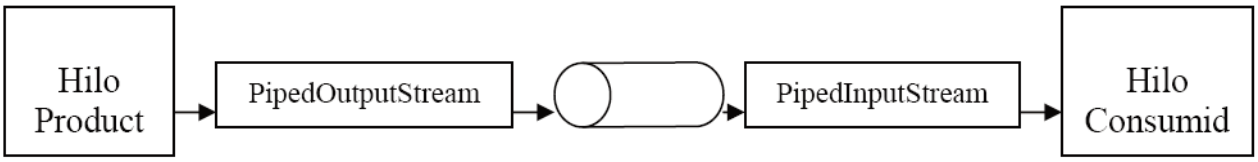
Ambas clases implementan los interfaces DataInput y DataOutput respectivamente.

Ambos interfaces tienen implementados todos los métodos para los tipos de datos fundamentales.

Pipes de Streams

Son una forma de comunicar hilos. Para la comunicación con pipes existen 2 tipos de hilos:

- Hilo productor, que envía datos al pipe.
- Hilo consumidor recibe datos del pipe.



Constructores:

PipedOutputStream (PipedInputStream sink)
PipedOutputStream ()
PipedInputStream (PipedOutputStream src)
PipedInputStream ()

Con estos constructores se puede enlazar de la siguiente forma:

```
PipedInputStream pis = new PipedInputStream();
PipedOutputStream pos = new PipedOutputStream(pis);
// Quedan enlazados y se puede enviar y recibir datos.
```

Existe otra forma de realizar el enlace de dos Stream. Primeramente se crearán con el constructor por defecto para posteriormente conectarlos con los siguientes métodos:

```
void <PipedOutputStream> connect (PipedInputStream sink)
void <PipedInputStream> connect (PipedOutputStream src)

public class Conexion {
  static PipedOutputStream sumidero = new PipedOutputStream();
  static PipedInputStream fuente = new PipedInputStream();
  static { // Inicio de Bloque Estático.
     try {
            fuente.connect(sumidero);
}
catch (IOException ex) {
    System.out.println (“No se pueden conectar los pipes”);
}
  } // fin de Bloque Estático.
} // fin de clase.
```

Esta clase conecta dos pipes con el segundo método explicado.

Bloque estático: conjunto de instrucciones que queremos que la máquina virtual ejecute en el momento que se carga la clase, es decir la primera vez que se referencia o se nombra la clase en el programa.

La clase Conexion no se instanciará pero en el momento que aparezca en el programa Conexion.sumidero o Conexion.fuente se ejecutará el bloque estático.

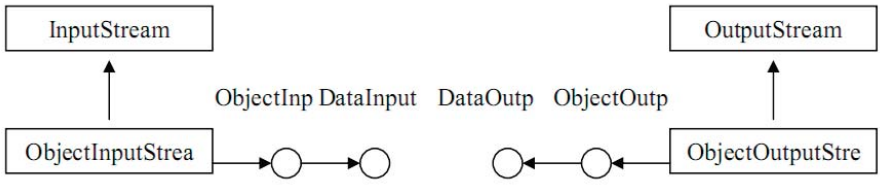
Serialización de Objetos

En un flujo se pueden escribir datos fundamentales como se ha visto anteriormente, a partir de ahora también se van a poder enviar y recoger objetos de un Stream. Para hacer esto, habrá que serializar el objeto, es decir, convertir el objeto en una serie de bytes. Posteriormente se podrá introducir en el Stream. Se tienen dos clases de persistencias:

- **Persistencia temporal**, consiste en hacer que la vida de un objeto supere el tiempo de ejecución de un programa. Para realizarlo antes de cerrar el programa se guardará el objeto en un fichero y la próxima vez que se ejecute el programa se podrá recuperar el objeto guardado.

- **Persistencia espacial**, consiste en que un objeto pase a vivir a una zona de memoria distinta a la zona de memoria donde fue creado. Un objeto que se crea en una máquina se irá a otra máquina (Sistemas Distribuidos).

En ambos casos se necesita serializar el objeto.



```
interface ObjectInput extends DataInput {
        Object ReadObject();
}

interface ObjectOutput extends DataOutput {
        void writeObject(Object ob);
}
```

No se permite por defecto serializar los objetos ya que cualquiera puede hacerlo. Para que un objeto se pueda serializar tiene que implementar el interface **Serializable**. Si el objeto no es serializable y se intenta serializar en ejecución, se lanza la siguiente excepción:

NotSerializableException.

```
Interface Serializable {}
```

Actúa únicamente como un flag para comprobar si un objeto se quiere o no serializar.

```
Class Punto implements Serializable { // Objeto Serializable.
        . . .
}
```

En este momento ya se pueden crear objetos Punto y serializarlos, la forma de hacerlo sería la siguiente:

```
ObjectOutputStream oos = new ObjectOutputStream
        (new FileOutputStream ("datos.dat"));
Punto p = new Punto (2,3);
oos.writeObject(p);
// El objeto Punto se serializa dentro del fichero datos.dat.
```

```
ObjectInputStream ois = new ObjectInputStream
        (new FileInputStream ("datos.dat"));
Punto p = (Punto) in.readObject();
// El objeto Punto serializado se saca del fichero datos.dat.
```

Se puede comprobar que se utiliza el operador cast con el método readObject ya que este método devuelve un objeto tipo Object.

Si una clase contenedora está Serializable y tiene objetos contenidos, los objetos contenidos deben implementar también Serializable (Composición). Si no se hace así se lanza la excepción:

NotSerializableException

Cuando se serializa un objeto, sus atributos se serializan pero no sus métodos. Los atributos estáticos (static) no se serializan ya que pertenecen a la clase y no al objeto. Los atributos con el modificador **transient** tampoco se serializan.

Las razones para serializar un atributo son dos básicamente:

- Por seguridad.

- Para mantener la consistencia. Por ejemplo, FileInputStream no es igual en dos máquinas ya que el sistema de ficheros en cada una de ellas es diferente).

También con los siguientes métodos se pueden serializar los objetos con encriptación:

```
private void writeObject(ObjectOutputStream oos)throws IOException

private void readObject(ObjectInputStream ois) throws IOException
```

```
class numero implements  Serializable {
        int numero;

        private void writeObject (ObjectOutputStream oos)
                throws IOException {
                        oos.writeInt(numero);
        }

        private void readObject(ObjectInputStream ois)
                throws IOException {
                        numero = ois.readInt();
        }
}
```

Existen también dos métodos para serializar y deserializar de forma estándar, recogen los atributos de la clase y operan con ellas.

```
void <ObjectOutputStream>.defaultWriteObject() throws IOException

void <ObjectInputStream>.defaultReadObject() throws IOException

class Cuenta implements Serializable {
        String nombre, direccion, telefono;
        int codpostal;
        transient int pin;
```

```
        private void writeObject(ObjectOutputStream oos)
                throws IOException {
                oos.defaultWriteObject();
                int numero = pin ^ pin; //Operación XOR.
                oos.writeInt(numero);
        }

        private  void readObject(ObjectInputStream ois)
                throws IOException {
                ois.defaultReadObject();
                pin ^= ois.readInt(pin);
        }
} // fin de clase
```

Cuando hay una clase base que implementa Serializable todas las derivadas implementarán Serializable (Agregación). Si la derivada implementa Serializable la base puede implementarla o no.

Cuando se serializa un objeto de clase derivada, primero se serializan los atributos de la base y posteriormente los atributos de la derivada.

Cuando se deserializa un objeto de clase derivada, primero se hace con los atributos de la base y seguidamente los atributos de la derivada.

- Gestión de versiones

En Java por defecto si la clase de serialización y la clase de deserialización no coinciden no se puede recoger el objeto. La excepción que se lanza cuando se produce este error es la siguiente:

InvalidCastException

```
class Prueba implements Serializable {
        int numero = 0;
        . . .
}
. . . se modifica la clase . . .

class Prueba implements Serializable {
        int numero = 0;
        void NoHaceNada () {};
}
```

A pesar de que el número de atributos no ha cambiado las clases se consideran incompatibles.

Para comprobar las versiones de las clases en Java, Java dispone del número SUID (Serial Unique ID) que es diferente para cada una de las clases que se crean.

Este número SUID lo genera a partir de los bytecode de la clase utilizando la función hash(). Si se modifica algo en una clase ya generada la función hash() genera un nuevo número SUID para la clase modificada.

```
C:\> serialver <Clase> Devuelve el número SUID de una clase
```

Para mantener la compatibilidad entre las distintas versiones de una clase, habrá que declarar la siguiente constante dentro de la clase:

```
static final long serialVersionUID = 6290341731103L; // serialver
```

El valor de serialVersionUID se corresponderá con la primera versión de la clase.

Normalmente el número SUID se pone desde la primera clase generada.

Las clases Reader y Writer

Estas clases sirven para trabajar con Streams de UNICODE. Normalmente se utilizan para Streams de texto. Todos lo anteriores sirven para flujo binario.

Reader y Writer son clases abstractas.

Reader se corresponde con InputStream, tiene todos los métodos descritos anteriormente en InputStream. Además tiene el siguiente método:

```
boolean <Reader>.ready()
```

Devuelve true en caso de que la próxima lectura de un solo byte se queda bloqueada.

Writer se corresponde a OutputStream.

Clases para trabajar con array de caracteres y objetos String

- CharArrayReader y CharArrayWriter

CharArrayReader equivale a ByteArrayInputStream.
CharArrayWriter equivale a ByteArrayOutputStream.

Sirven para trabajar con array de caracteres.

```
char[] letras = new char[10];
CharArrayReader ch = new CharArrayReader(letras);

// Busca el carácter 'c' en un array.
int c  = ch.read();

while (c ¡= -1 && c ¡= 'c')
    c=ch.read(); //Lee hasta al final del array o encuentre 'c'.
```

- StringReader y StringWriter

Son igual que las anteriores sólo que trabajan con objetos String.

```
String letras = “Federico”;
StringReader sr =  new StringReader (letras);

int pos = 0;
int c = sr.read();

while (c ¡= -1 && c ¡= ‘c’) {
    pos++;
    c = sr.read();
}
```

StringWriter introduce los datos en un objeto de tipo StringBuffer mediante las operaciones writer y no sobre el objeto String que como se sabe es constante. StringWriter tiene dos constructores:

StringWriter()
: El objeto StringBuffer se crea con un tamaño de 32 bytes y se redimensiona si se necesita más espacio.

StringWriter(int size)
: Se le indica el tamaño, con esto se hace que la operación sea más rápida.

StringBuffer <StringWriter>.getBuffer()
: Se saca lo que se ha escrito.

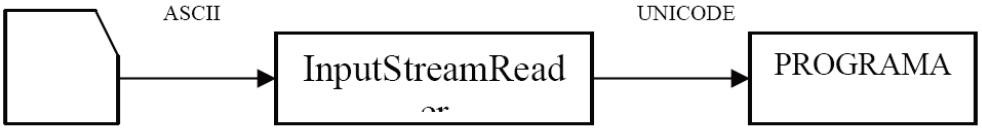
String <StringWriter>.toString()
: Lo que está contenido se guarda en un String.

InputStreamReader y OutputStreamReader

Conversión de código UNICODE a ASCII y viceversa. Actúan como filtro para la traducción.

InputStreamReader

InputStreamReader (InputStream is) De ASCII a UNICODE



El problema surge en cómo traducir de ASCII a UNICODE los caracteres nacionales (0..127 Mundiales / 128..255 Nacionales). Este problema se resuelve con el siguiente constructor.

InputStreamReader (InputStream is, String encode)

El segundo parámetro representa el código ISO que se quiere utilizar. En el caso de España es “8859-1”.

En el primer constructor como se recibe el encode se utiliza la propiedad fileencoding.

System.out.println (System.getProperty(“fileencoding”);

- **OutputStreamReader**

Realiza la conversión de UNICODE a ASCII.

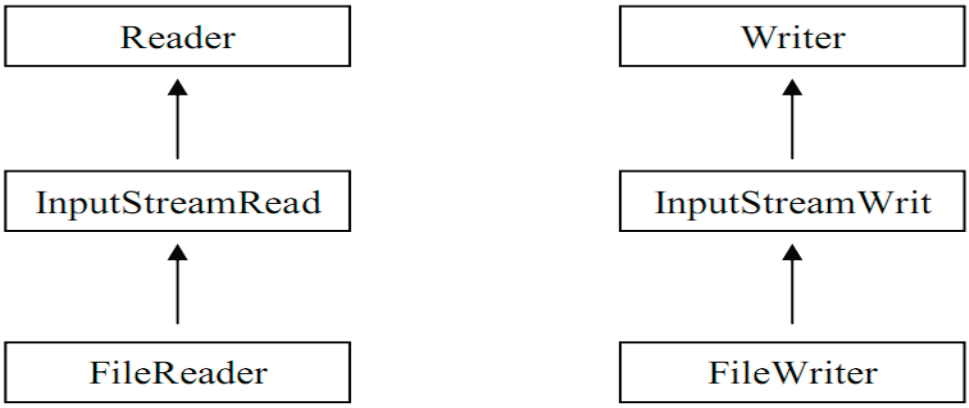
OutputStreamWriter (OutputStream os)

OutputStreamWriter (OutputStream os, String encode)

Igual que lo definido en InputStreamReader.

```
FileInputStream f_binario = new FileInputStream ("prueba.dat");

InputStreamReader f_tex = new InputStreamReader
                          (f_binario,"8859-1");
```

Las clases FileReader y FileWriter



Como se puede comprobar son derivadas de las anteriores que sólo valen para ficheros.

FileReader (String filename)
FileReader (File f)

FileWriter (String filename)
FileWriter (File f)
FileWriter (String Filename, boolean append)

A continuación puedes ver un extracto del API oficial de la página de SUN Microsystems para que tengas a mano un resumen con los métodos más usados en lo que se refiere a streams y ficheros java.

Empezaremos con la clase **InputStream:**

MÉTODOS

int **available**()
Returns the number of bytes that can be read (or skipped over) from this input stream without blocking by the next caller of a method for this input stream.

void **close**()
Closes this input stream and releases any system resources associated with the stream.

void **mark**(int readlimit)
Marks the current position in this input stream.

boolean **markSupported**()
Tests if this input stream supports the mark and reset methods.

abstract int **read**()
Reads the next byte of data from the input stream.

int **read**(byte[] b)
Reads some number of bytes from the input stream and stores them into the buffer array b.

int **read**(byte[] b, int off, int len)
Reads up to len bytes of data from the input stream into an array of bytes.

void **reset**()
Repositions this stream to the position at the time the mark method was last called on this input stream.

long **skip**(long n)
Skips over and discards n bytes of data from this input stream.

Clase OutputStream:

MÉTODOS

void **close**()
Closes this output stream and releases any system resources associated with this stream.

void **flush**()
Flushes this output stream and forces any buffered output bytes to be written out.

void **write**(byte[] b)
Writes b.length bytes from the specified byte array to this output stream.

void **write**(byte[] b, int off, int len)
Writes len bytes from the specified byte array starting at offset off to this output stream.

abstract void **write**(int b)
Writes the specified byte to this output stream.

La clase File:

MÉTODOS

boolean **canRead**()
Tests whether the application can read the file denoted by this abstract pathname.

boolean **canWrite**()
Tests whether the application can modify to the file denoted by this abstract pathname.

int **compareTo**(File pathname)
Compares two abstract pathnames lexicographically.

int **compareTo**(Object o)
Compares this abstract pathname to another object.

boolean **createNewFile**()
Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.

static File **createTempFile**(String prefix, String suffix)
Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.

static File **createTempFile** (String prefix, String suffix, File directory)
Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name.

boolean **delete**()
Deletes the file or directory denoted by this abstract pathname.

void **deleteOnExit**()
Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.

boolean **equals**(Object obj)
Tests this abstract pathname for equality with the given object.

boolean **exists**()
Tests whether the file denoted by this abstract pathname exists.

File **getAbsoluteFile**()
Returns the absolute form of this abstract pathname.

String **getAbsolutePath**()
Returns the absolute pathname string of this abstract pathname.

File **getCanonicalFile**()
Returns the canonical form of this abstract pathname.

String **getCanonicalPath**()
Returns the canonical pathname string of this abstract pathname.

String **getName**()
Returns the name of the file or directory denoted by this abstract pathname.

String **getParent**()
Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.

File **getParentFile**()
Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.

String **getPath**()
Converts this abstract pathname into a pathname string.

int **hashCode**()
Computes a hash code for this abstract pathname.

boolean **isAbsolute**()
Tests whether this abstract pathname is absolute.

boolean **isDirectory**()
Tests whether the file denoted by this abstract pathname is a directory.

boolean **isFile**()
Tests whether the file denoted by this abstract pathname is a normal file.

boolean **isHidden**()
Tests whether the file named by this abstract pathname is a hidden file.

long **lastModified**()
Returns the time that the file denoted by this abstract pathname was last modified.

long **length**()
Returns the length of the file denoted by this abstract pathname.

String[] **list**()
Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

String[] **list**(FilenameFilter filter)
Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.

File [] **listFiles**()
Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.

File [] **ListFiles** (FileFilter filter)
Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.

File[] **ListFiles** (FilenameFilter filter)
Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.

static File[] **listRoots**()
List the available filesystem roots.

boolean **mkdir**()
Creates the directory named by this abstract pathname.

boolean **mkdirs**()
Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories.

boolean **renameTo**(File dest)
Renames the file denoted by this abstract pathname.

boolean **setLastModified**(long time)
Sets the last-modified time of the file or directory named by this abstract pathname.

boolean **setReadOnly**()
Marks the file or directory named by this abstract pathname so that only read operations are allow

String **toString**()
Returns the pathname string of this abstract pathname.

URL **toURL**()
Converts this abstract pathname into a file: URL.