Tema 9: Entrada - Salida

Todos los lenguajes de programación tienen alguna forma de interactuar con los sistemas de ficheros locales; Java no es una excepción.

Por otro lado, si se está desarrollando una aplicación Java para uso interno, probablemente será necesario el acceso directo a ficheros.

9.1 FICHEROS.-

Antes de realizar acciones sobre un fichero, necesitamos un poco de información sobre ese fichero. La clase *File* proporciona muchas utilidades relacionadas con ficheros y con la obtención de información básica sobre esos ficheros.

9.1.1. CREACIÓN DE UN OBJETO *File*.-

Para crear un objeto File nuevo, se puede utilizar cualquiera de los tres constructores siguientes:

```
    File miFichero;
    miFichero = new File( "/etc/kk" );
```

o

```
  miFichero = new File( "/etc","kk" );
```

o

```
    File miDirectorio = new File( "/etc" );
    miFichero = new File( miDirectorio,"kk" );
```

El constructor utilizado depende a menudo de otros objetos *File* necesarios para el acceso. Por ejemplo, si sólo se utiliza un fichero en la aplicación, el primer constructor es el mejor. Si en cambio, se utilizan muchos ficheros desde un mismo directorio, el segundo o tercer constructor serán más cómodos. Y si el directorio o el fichero es una variable, el segundo constructor será el más útil.

9.1.2. COMPROBACIONES Y UTILIDADES.-

Una vez creado un objeto *File*, se puede utilizar uno de los siguientes métodos para reunir información sobre el fichero:

- **Nombres de fichero**

```
String getName()
String getPath()
String getAbsolutePath()
String getParent()
boolean renameTo( File nuevoNombre )
```

- **Comprobaciones**

```
boolean exists()
boolean canWrite()
boolean canRead()
boolean isFile()
boolean isDirectory()
boolean isAbsolute()
```

- **Información general del fichero**

```
long lastModified()
long length()
```

- **Utilidades de directorio**

```
boolean mkdir()
String[] list()
```

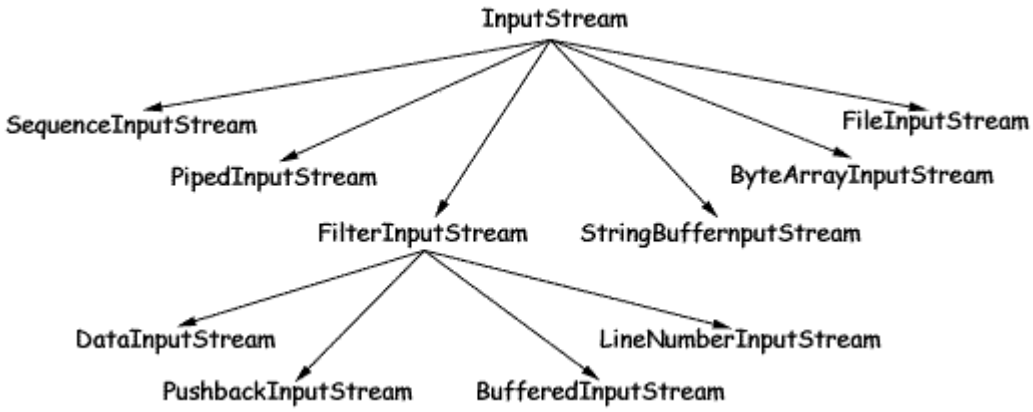
Vamos a desarrollar una pequeña aplicación que muestra información sobre los ficheros pasados como argumentos en la línea de comandos, *InfoFichero.java*:

```
import java.io.*;
class InfoFichero {

    public static void main( String args[] ) throws IOException {
        if( args.length > 0 )
            {
            for( int i=0; i < args.length; i++ )
                {
                File f = new File( args[i] );
                System.out.println( "Nombre: "+f.getName() );
                System.out.println( "Camino: "+f.getPath() );
                if( f.exists() )
                    {
                    System.out.print( "Fichero existente " );
                    System.out.print( (f.canRead() ?
                        " y se puede Leer" : "" ) );
                    System.out.print( (f.canWrite() ?
                        " y se puese Escribir" : "" ) );
                    System.out.println( "." );
                    System.out.println( "La longitud del fichero son "+
                        f.length()+" bytes" );
                    }
                else
                    System.out.println( "El fichero no existe." );
                }
            }
        else
            System.out.println( "Debe indicar un fichero." );
        }
    }
```

9.2. STREAMS DE ENTRADA.-

Hay muchas clases dedicadas a la obtención de entrada desde un fichero. Este es el esquema de la jerarquía de clases de entrada por fichero:



9.2.1. OBJETOS *FileInputStream*.-

Los objetos *FileInputStream* típicamente representan ficheros de texto accedidos en orden secuencial, byte a byte. Con *FileInputStream*, se puede elegir acceder a un byte, varios bytes o al fichero completo.

Apertura de un FileInputStream

Para abrir un *FileInputStream* sobre un fichero, se le da al constructor un *String* o un objeto *File*:

```
FileInputStream mi FicheroSt;
miFicheroSt = new FileInputStream( "/etc/kk" );
```

También se puede utilizar:

```
File miFichero FileInputStream miFicheroSt;
miFichero = new File( "/etc/kk" );
miFicheroSt = new FileInputStream(miFichero );
```

Lectura de un FileInputStream

Una vez abierto el *FileInputStream*, se puede leer de él. El método *read()* tiene muchas opciones:

```
int read();
```

Lee un byte y devuelve -1 al final del stream.

```
int read( byte b[] );
```

Llena todo el array, si es posible. Devuelve el número de bytes leídos o -1 si se alcanzó el final del stream.

```
int read( byte b[],int offset,int longitud );
```

Lee longitud bytes en *b* comenzando por *b*[*offset*]. Devuelve el número de bytes leídos o -1 si se alcanzó el final del stream.

Cierre de FileInputStream

Cuando se termina con un fichero, existen dos opciones para cerrarlo: explícitamente, o implícitamente cuando se recicla el objeto (el garbage collector se encarga de ello).

Para cerrarlo explícitamente, se utiliza el método *close():*

```
miFicheroSt.close();
```

Ejemplo: Visualización de un fichero

Si la configuración de la seguridad de Java permite el acceso a ficheros, se puede ver el contenido de un fichero en un objeto *TextArea*. El código siguiente contiene los elementos necesarios para mostrar un fichero:

```
FileInputStream fis;
TextArea ta;

public void init() {
    byte b[] = new byte[1024];
    int i;

    // El buffer de lectura se debe hacer lo suficientemente grande
    // o esperar a saber el tamaño del fichero
    String s;

    try {
        fis = new FileInputStream( "/etc/kk" );
    } catch( FileNotFoundException e ) {
        /* Hacer algo */
```

```
        }

    try {
        i = fis.read( b );
    } catch( IOException e ) {
        /* Hacer algo */
        }

    s = new String( b,0 );
    ta = new TextArea( s,5,40 );
    add( ta );
    }
```

Hemos desarrollado un ejemplo, *Agenda.java*, en el que partimos de un fichero agenda que dispone de los datos que nosotros deseamos de nuestros amigos, como son: nombre, teléfono y dirección. Si tecleamos un nombre, buscará en el fichero de datos si existe ese nombre y presentará la información que se haya introducido. Para probar, intentar que aparezca la información de Pepe.

9.2.2. OBJETOS *DataInputStream*.-

Los objetos *DataInputStream* se comportan como los *FileInputStream*. Los streams de datos pueden leer cualquiera de las variables de tipo nativo, como *floats*, *ints* o *chars*. Generalmente se utilizan *DataInputStream* con ficheros binarios.

Apertura y cierre de *DataInputStream*

Para abrir y cerrar un objeto *DataInputStream*, se utilizan los mismos métodos que para *FileInputStream*:

```
DataInputStream miDStream;
FileInputStream miFStream;

// Obtiene un controlador de fichero
miFStream = new FileInputStream "/etc/ejemplo.dbf" );
//Encadena un fichero de entrada de datos
miDStream = new DataInputStream( miFStream );

// Ahora se pueden utilizar los dos streams de entrada para
// acceder al fichero (si se quiere...)
miFStream.read( b );
i = miDStream.readInt();

// Cierra el fichero de datos explícitamente
//Siempre se cierra primero el fichero stream de mayor nivel
miDStream.close();
miFStream.close();
```

Lectura de un *DataInputStream*

Al acceder a un fichero como *DataInputStream*, se pueden utilizar los mismos métodos *read()* de los objetos *FileInputStream*. No obstante, también se tiene acceso a otros métodos diseñados para leer cada uno de los tipos de datos:

```
byte readByte()
int readUnsignedByte()
short readShort()
int readUnsignedShort()
char readChar()
int readInt()
long readLong()
float readFloat()
double readDouble()
String readLine()
```

Cada método leerá un objeto del tipo pedido.

Para el método *String readLine(),* se marca el final de la cadena con *\n*, *\r*, *\r\n* o con *EOF*.

Para leer un *long*, por ejemplo:

```
long numeroSerie;
...
numeroSerie = miDStream.readLong();
```

9.2.3. STREAM DE ENTRADA URLs.-

Además del acceso a ficheros, Java proporciona la posibilidad de acceder a *URLs* como una forma de acceder a objetos a través de la red. Se utiliza implícitamente un objeto URL al acceder a sonidos e imágenes, con el método *getDocumentBase()* en los applets:

```
String imagenFich = new String( "imagenes/pepe.gif" );
imagenes[0] = getImage( getDocumentBase(),imagenFich );
```

No obstante, se puede proporcionar directamente un *URL*, si se quiere:

```
URL imagenSrc;
imagenSrc = new URL( "http://enterprise.com/~info" );
imagenes[0] = getImage( imagenSrc,"imagenes/pepe.gif" );
```

Apertura de un Stream de entrada de URL

También se puede abrir un stream de entrada a partir de un *URL*. Por ejemplo, se puede utilizar un fichero de datos para un applet:

```
ImputStream is;
byte buffer[] = new byte[24];
is = new URL( getDocumentBase(),datos).openStream();
```

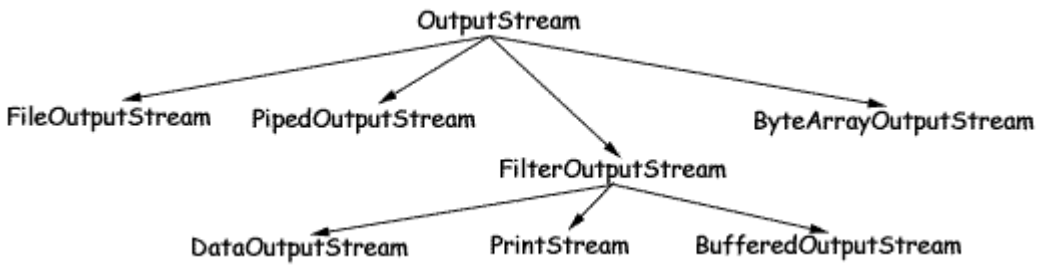
Ahora se puede utilizar is para leer información de la misma forma que se hace con un objeto *FileInputStream*:

```
is.read( buffer,0,buffer.length );
```

Debe tenerse muy en cuenta que algunos usuarios pueden haber configurado la seguridad de sus navegadores para que los applets no accedan a ficheros.

9.3. STREAMS DE SALIDA.-

La contrapartida necesaria de la lectura de datos es la escritura de datos. Como con los *Streams* de entrada, las clases de salida están ordenadas jerárquicamente:



Examinaremos las clases *FileOutputStream* y *DataOutputStream* para complementar los streams de entrada que se han visto.

9.3.1. OBJETOS *FileOutputStream*.-

Los objetos *FileOutputStream* son útiles para la escritura de ficheros de texto. Como con los ficheros de entrada, primero se necesita abrir el fichero para luego escribir en él.

Apertura de un FileOutputStream

Para abrir un objeto *FileOutputStream*, se tienen las mismas posibilidades que para abrir un fichero stream de entrada. Se le da al constructor un *String* o un objeto *File*.

```
    FileOutputStream miFicheroSt;
    miFicheroSt = new FileOutputStream( "/etc/kk" );
```

Como con los streams de entrada, también se puede utilizar:

```
File miFichero FileOutputStream miFicheroSt;

    miFichero = new File( "/etc/kk" );
    miFicheroSt = new FileOutputStream( miFichero );
```

Escritura en un *FileOutputStream*

Una vez abierto el fichero, se pueden escribir bytes de datos utilizando el método *write()*. Como con el método *read()* de los streams de entrada, tenemos tres posibilidades:

```
void write( int b );
```

Escribe un byte.

```
void write( byte b[] );
```

Escribe todo el array, si es posible.

```
void write( byte b[],int offset,int longitud );
```

Escribe longitud bytes en *b* comenzando por *b[offset]*.

Cierre de *FileOutputStream*

Cerrar un stream de salida es similar a cerrar streams de entrada. Se puede utilizar el método explícito:

```
    miFicheroSt.close();
```

O, se puede dejar que el sistema cierre el fichero cuando se recicle *miFicheroSt*.

Ejemplo: Almacenamiento de Información

Este programa, *Telefonos.java*, pregunta al usuario una lista de nombres y números de teléfono. Cada nombre y número se añade a un fichero situado en una localización fija. Para indicar que se ha introducido toda la lista, el usuario especifica "Fin" ante la solicitud de entrada del nombre.

Una vez que el usuario ha terminado de teclear la lista, el programa creará un fichero de salida que se mostrará en pantalla o se imprimirá. Por ejemplo:

```
    95-4751232,Juanito
    564878,Luisa
    123456,Pepe
    347698,Antonio
    91-3547621,Maria
```

El código fuente del programa es el siguiente:

```
import java.io.*;

class Telefonos {
    static FileOutputStream fos;
    public static final int longLinea = 81;

    public static void main( String args[] ) throws IOException {
        byte tfno[] = new byte[longLinea];
        byte nombre[] = new byte[longLinea];

        fos = new FileOutputStream( "telefono.dat" );
        while( true )
            {
            System.err.println( "Teclee un nombre ('Fin' termina)" );
            leeLinea( nombre );
            if( "fin".equalsIgnoreCase( new String( nombre,0,0,3 ) ) )
                break;

            System.err.println( "Teclee el numero de telefono" );
            leeLinea( tfno );
            for( int i=0; tfno[i] != 0; i++ )
                fos.write( tfno[i] );
            fos.write( ',' );
            for( int i=0; nombre[i] != 0; i++ )
                fos.write( nombre[i] );
            fos.write( '\n' );
            }
        fos.close();
        }

    private static void leeLinea( byte linea[] ) throws IOException {
        int b = 0;
        int i = 0;

        while( (i < ( longLinea-1) ) &&
            ( ( b = System.in.read() ) != '\n' ) )
            linea[i++] = (byte)b;
        linea[i] = (byte)0;
        }
    }
```

9.3.2 STREAMS DE SALIDA CON BUFFER.-

Si se trabaja con gran cantidad de datos, o se escriben muchos elementos pequeños, será una buena idea utilizar un stream de salida con buffer. Los streams con buffer ofrecen los mismos métodos de la clase *FileOutputStream*, pero toda salida se almacena en un buffer. Cuando se llena el buffer, se envía a disco con una única operación de escritura; o, en caso necesario, se puede enviar el buffer a disco en cualquier momento.

Creación de *Streams* de salida con buffer

Para crear un stream BufferedOutput, primero se necesita un stream FileOutput normal; entonces se le añade un buffer al stream:

```
    FileOutputStream miFileStream;
    BufferdOutpurStream miBufferStream;
    // Obtiene un controlador de fichero
    miFileStream = new FileOutputStream( "/tmp/kk" );
    // Encadena un stream de salida con buffer
    miBufferStream = new BufferedOutputStream( miFileStream );
```

Volcado y Cierre de *Streams* de salida con buffer

Al contrario que los streams *FileOutput*, cada escritura al buffer no se corresponde con una escritura en disco. A menos que se llene el buffer antes de que termine el programa, cuando se quiera volcar el buffer explícitamente se debe hacer mediante una llamada a *flush():*

```
// Se fuerza el volcado del buffer a disco
miBufferStream.flush();

// Cerramos el fichero de datos. Siempre se ha de cerrar primero el
// fichero stream de mayor nivel
miBufferStream.close();
miFileStream.close();
```

9.3.3. STREAMS *DataOutput*.-

Java también implementa una clase de salida complementaria a la clase *DataInputStream*. Con la clase *DataOutputStream*, se pueden escribir datos binarios en un fichero.

Apertura y cierre de objetos *DataOutputStream*

Para abrir y cerrar objetos *DataOutputStream*, se utilizan los mismos métodos que para los objetos *FileOutputStream*:

```
DataOutputStream miDataStream;
FileOutputStream miFileStream;
BufferedOutputStream miBufferStream;

// Obtiene un controlador de fichero
miFileStream = new FileOutputStream( "/tmp/kk" );
// Encadena un stream de salida con buffer (por eficiencia)
miBufferStream = new BufferedOutputStream( miFileStream );
// Encadena un fichero de salida de datos
miDataStream = new DataOutputStream( miBufferStream );

// Ahora se pueden utilizar los dos streams de entrada para
// acceder al fichero (si se quiere)
miBufferStream.write( b );
miDataStream.writeInt( i );

// Cierra el fichero de datos explícitamente. Siempre se cierra
// primero el fichero stream de mayor nivel
miDataStream.close();
miBufferStream.close();
miFileStream.close();
```

Escritura en un objeto *DataOutputStream*

Cada uno de los métodos *write()* accesibles por los *FileOutputStream* también lo son a través de los *DataOutputStream*. También encontrará métodos complementarios a los de *DataInputStream*:

```
void writeBoolean( boolean b );
void writeByte( int i );
void writeShort( int i );
void writeChar( int i );
void writeInt( int i );
void writeFloat( float f );
void writeDouble( double d );
void writeBytes( String s );
void writeChars( string s );
```

Para las cadenas, se tienen dos posibilidades: bytes y caracteres. Hay que recordar que los bytes son objetos de 8 bits y

los caracteres lo son de 16 bits. Si nuestras cadenas utilizan caracteres Unicode, debemos escribirlas con *writeChars()*.

Contabilidad de la salida

Otra función necesaria durante la salida es el método *size()*. Este método simplemente devuelve el número total de bytes escritos en el fichero. Se puede utilizar *size()* para ajustar el tamaño de un fichero a múltiplo de cuatro. Por ejemplo, de la forma siguiente:

```
. . .
int numBytes = miDataStream.size() % 4;
for( int i=0; i < numBytes; i++ )
    miDataStream.write( 0 );
 . . .
```

9.4. FICHEROS DE ACCESO ALEATORIO.-

A menudo, no se desea leer un fichero de principio a fin; sino acceder al fichero como una base de datos, donde se salta de un registro a otro; cada uno en diferentes partes del fichero. Java proporciona una clase *RandomAccessFile* para este tipo de entrada/salida.

9.4.1. CREACIÓN DE UN FICHERO DE ACCESO ALEATORIO.-

Hay dos posibilidades para abrir un fichero de acceso aleatorio:

- Con el nombre del fichero:

  ```
  miRAFile = new RandomAccessFile( String nombre,String modo );
  ```

- Con un objeto *File*:

  ```
  miRAFile = new RandomAccessFile( File fichero,String modo );
  ```

El argumento modo determina si se tiene acceso de sólo lectura (*r*) o de lectura/escritura (*r/w*). Por ejemplo, se puede abrir un fichero de una base de datos para actualización:

```
RandomAccessFile miRAFile;
miRAFile = new RandomAccessFile( "/tmp/kk.dbf","rw" );
```

9.4.2. ACCESO A LA INFORMACIÓN.-

Los objetos *RandomAccessFile* esperan información de lectura/escritura de la misma manera que los objetos *DataInput*/*DataOutput*. Se tiene acceso a todas las operaciones *read()* y *write()* de las clases *DataInputStream* y *DataOutputStream*.

También se tienen muchos métodos para moverse dentro de un fichero:

`long getFilePointer();`

Devuelve la posición actual del puntero del fichero

`void seek( long pos );`

Coloca el puntero del fichero en una posición determinada. La posición se da como un desplazamiento en bytes desde el comienzo del fichero. La posición 0 marca el comienzo de ese fichero.

`long length();`

Devuelve la longitud del fichero. La posición *length()* marca el final de ese fichero.

9.4.3. ACTUALIZACIÓN DE LA INFORMACIÓN.-

Se pueden utilizar ficheros de acceso aleatorio para añadir información a ficheros existentes:

```
    miRAFile = new RandomAccessFile( "/tmp/kk.log","rw" );
    miRAFile.seek( miRAFile.length() );
    // Cualquier write() que hagamos a partir de este punto del código
    // añadirá información al fichero
```

Vamos a ver un pequeño ejemplo, *Log.java*, que añade una cadena a un fichero existente:

```
import java.io.*;

// Cada vez que ejecutemos este programita, se incorporara una nueva
// linea al fichero de log que se crea la primera vez que se ejecuta
//
class Log {
    public static void main( String args[] ) throws IOException {
        RandomAccessFile miRAFile;
        String s = "Informacion a incorporar\nTutorial de Java\n";

        // Abrimos el fichero de acceso aleatorio
        miRAFile = new RandomAccessFile( "/tmp/java.log","rw" );
        // Nos vamos al final del fichero
        miRAFile.seek( miRAFile.length() );
        // Incorporamos la cadena al fichero
        miRAFile.writeBytes( s );
        // Cerramos el fichero
        miRAFile.close();
        }
    }
```

9.5.ENTRADA Y SALIDA ESTÁNDAR (TECLADO Y PANTALLA).-

En ***Java***, la entrada desde teclado y la salida a pantalla están reguladas a traves de la clase ***System***. Esta clase pertenece al package ***java.lang*** y agrupa diversos métodos y objetos que tienen relación con el sistema local. Contiene, entre otros, tres objetos ***static*** que son:

- **System.in:** Objeto de la clase ***InputStream*** preparado para recibir datos desde la entrada estándar del sistema (habitualmente el teclado).
- **System.out:** Objeto de la clase ***PrintStream*** que imprimirá los datos en la salida estándar del sistema (normalmente asociado con la pantalla).
- **System.err:** Objeto de la clase ***PrintStream***. Utilizado para mensajes de error que salen también por pantalla por defecto.

Existen tres métodos de ***System*** que permiten sustituir la entrada y salida estándar. Por ejemplo, se utiliza para hacer que el programa lea de un archivo y no del teclado.

```
System.setIn(InputStream is);
System.setOut(PrintStream ps);
System.setErr(PrintStream ps);
```

El argumento de ***setIn()*** no tiene que ser necesariamente del tipo ***InputStream***. Es una referencia a la clase base, y por tanto puede apuntar a objetos de cualquiera de sus clases derivadas (como ***FileInputStream***). Asimismo, el constructor de ***PrintStream*** acepta un ***OutputStream***, luego se puede dirigir la salida estándar a cualquiera de las clases definidas para salida.

Si se utilizan estas sentencias con un compilador de ***Java 1.1*** se obtiene un mensaje de método obsoleto (***deprecated***) al crear un objeto ***PrintStream***. Al señalar como obsoleto el constructor de esta clase se pretendía animar al uso de ***PrintWriter***, pero existen casos en los cuales es imprescindible un elemento ***PrintStream***. Afortunadamente, ***Java 1.2*** ha reconsiderado esta decisión y de nuevo se puede utilizar sin problemas.

9.5.1.Salida de texto y variables por pantalla.-

Para imprimir en la pantalla se utilizan los métodos System.out.print() y System.out.println(). Son los primeros métodos que aprende cualquier programador. Sus características fundamentales son:

1. Pueden imprimir valores escritos directamente en el código o cualquier tipo de variable primitiva de Java.

```
System.out.println("Hola, Mundo!");
System.out.println(57);
double numeroPI = 3.141592654;
System.out.println(numeroPI);
String hola = new String("Hola");
System.out.println(hola);
```

2. Se pueden imprimir varias variables en una llamada al método correspondiente utilizando el operador + de concatenación, que equivale a convertir a String todas las variables que no lo sean y concatenar las cadenas de caracteres (el primer argumento debe ser un String).

```
System.out.println("Hola, Mundo! " + numeroPI);
```

Se debe recordar que los objetos System.out y System.err son de la clase PrintStream y aunque imprimen las variables de un modo legible, no permiten dar a la salida un formato a medida.

El programador no puede especificar un formato distinto al disponible por defecto.

9.5.2. Lectura desde teclado.-

Para leer desde teclado se puede utilizar el método System.in.read() de la clase InputStream. Este método lee un carácter por cada llamada. Su valor de retorno es un int. Si se espera cualquier otro tipo hay que hacer una conversión explícita mediante un cast.

```
char c;
c=(char)System.in.read();
```

Este método puede lanzar la excepción java.io.IOException y siempre habrá que ocuparse de ella, por ejemplo en la forma:

```
try {
c=(char)System.in.read();
}
catch(java.io.IOException ioex) {
// qué hacer cuando ocurra la excepción
}
```

Para leer datos más largos que un simple carácter es necesario emplear un bucle while o for y unir los caracteres. Por ejemplo, para leer una línea completa se podría utilizar un bucle while guardando los caracteres leídos en un String o en un StringBuffer (más rápido que String):

```
char c;
String frase = new String(""); // StringBuffer frase=new StringBuffer("");
try {
while((c=System.in.read()) != '\n')
frase = frase + c; // frase.append(c);
}
catch(java.io.IOException ioex) {}
```

Una vez que se lee una línea, ésta puede contener números de coma flotante, etc. Sin embargo, hay una manera más fácil de conseguir lo mismo: utilizar adecuadamente la librería java.io.

9.6. LECTURA DE UN ARCHIVO EN UN SERVIDOR DE INTERNET.-

Teniendo la dirección de Internet de un archivo, la librería de Java permite leer este archivo utilizando un stream. Es una aplicación muy sencilla que muestra la polivalencia del concepto de stream.

En el package java.net existe la clase URL, que representa una dirección de Internet. Esta clase tiene el método InputStream openStream(URL dir) que abre un stream con origen en la dirección de Internet.

A partir de ahí, se trata como cualquier elemento InputStream. Por ejemplo:

```
//Lectura del archivo (texto HTML)
URL direccion = new URL("http://www1.ceit.es/subdir/MiPagina.htm");
String s = new String();
String html = new String();
try {
      BufferedReader br = new BufferedReader(
                  new InputStreamReader(
                        direccion.openStream()));
      while((s = br.readLine()) != null)
      html += s + '\n';
      br.close();
}
catch(Exception e) {
      System.err.println(e);
}
```

9.7. LECTURA Y ESCRITURA DE FICHEROS DE TEXTO.-

Aunque el manejo de archivos tiene características especiales, se puede utilizar lo dicho hasta ahora para las entradas y salidas estándar con pequeñas variaciones. Java ofrece las siguientes posibilidades:

Existen las clases FileInputStream y FileOutputStream (extendiendo InputStream y OutputStream) que permiten leer y escribir bytes en archivos. Para archivos de texto son preferibles FileReader (desciende de Reader) y FileWriter (desciende de Writer), que realizan las mismas funciones. Se puede construir un objeto de cualquiera de estas cuatro clases a partir de un String que contenga el nombre o la dirección en disco del archivo o con un objeto de la clase File que representa dicho archivo. Por ejemplo el código

```
FileReader fr1 = new FileReader("archivo.txt");
es equivalente a:
File f = new File("archivo.txt");
FileReader fr2 = new FileReader(f);
```

Si no encuentran el archivo indicado, los constructores de FileReader y FileInputStream pueden lanzar la excepción java.io.FileNotFoundException. Los constructores de FileWriter y FileOutputStream pueden lanzar java.io.IOException. Si no encuentran el archivo indicado, lo crean nuevo. Por defecto, estas dos clases comienzan a escribir al comienzo del archivo. Para escribir detrás de lo que ya existe en el archivo (“append”), se utiliza un segundo argumento de tipo boolean con valor true:

```
FileWriter fw = new FileWriter("archivo.txt", true);
```

Las clases que se explican a continuación permiten un manejo más fácil y eficiente que las vistas hasta ahora.

9.7.1. Lectura de un fichero de texto en java.-

Podemos *abrir un fichero de texto* para leer usando la clase *FileReader*. Esta clase tiene métodos que nos permiten leer caracteres. Sin embargo, suele ser habitual querer las líneas completas, bien porque nos interesa la línea completa, bien para poder analizarla luego y extraer campos de ella. *FileReader* no contiene métodos que nos permitan leer líneas completas, pero sí *BufferedReader*. Afortunadamente, podemos construir un *BufferedReader* a partir del *FileReader* de la siguiente forma:

```
File archivo = new File ("C:\\archivo.txt");
FileReader fr = new FileReader (archivo);
```

```
BufferedReader br = new BufferedReader(fr);
...
String linea = br.readLine();
```

La apertura del fichero y su posterior lectura pueden lanzar excepciones que debemos capturar. Por ello, la apertura del fichero y la lectura debe meterse en un bloque *try-catch*.

Además, el fichero hay que cerrarlo cuando terminemos con él, tanto si todo ha ido bien como si ha habido algún error en la lectura después de haberlo abierto. Por ello, se suele poner al *try-catch* un bloque *finally* y dentro de él, el *close()* del fichero.

El siguiente es un código completo con todo lo mencionado.

```
import java.io.*;

class LeeFichero {
   public static void main(String [] arg) {
      File archivo = null;
      FileReader fr = null;
      BufferedReader br = null;

      try {
         // Apertura del fichero y creacion de BufferedReader para poder
         // hacer una lectura comoda (disponer del metodo readLine()).
         archivo = new File ("C:\\archivo.txt");
         fr = new FileReader (archivo);
         br = new BufferedReader(fr);

         // Lectura del fichero
         String linea;
         while((linea=br.readLine())!=null)
            System.out.println(linea);
      }
      catch(Exception e){
         e.printStackTrace();
      }finally{
         // En el finally cerramos el fichero, para asegurarnos
         // que se cierra tanto si todo va bien como si salta
         // una excepcion.
         try{
            if( null != fr ){
               fr.close();
            }
         }catch (Exception e2){
            e2.printStackTrace();
         }
      }
   }
}
```

Como opción para leer un fichero de texto línea por línea, podría usarse la clase *Scanner* en vez de el *FileReader* y el *BufferedReader*.

9.7.2. Escritura de un fichero de texto en java.-

La clase PrintWriter es la más práctica para escribir un archivo de texto porque posee los métodos print(cualquier tipo) y println(cualquier tipo), idénticos a los de System.out (de clase PrintStream).

Un objeto PrintWriter se puede crear a partir de un BufferedWriter (para disponer de buffer), que se crea a partir del FileWriter al que se la pasa el nombre del archivo. Después, escribir en el archivo es tan fácil como en pantalla. El siguiente ejemplo ilustra lo anterior:

```
try {
      FileWriter fw = new FileWriter("escribeme.txt");
      BufferedWriter bw = new BufferedWriter(fw);
      PrintWriter salida = new PrintWriter(bw);
      salida.println("Hola, soy la primera línea");
      salida.close();

      // Modo append
      bw = new BufferedWriter(new FileWriter("escribeme.txt", true));
      salida = new PrintWriter(bw);
      salida.print("Y yo soy la segunda. ");
      double b = 123.45;
      salida.println(b);
      salida.close();
}
cacth(java.io.IOException ioex) { }
```

EJERCICIOS PROPUESTOS.-

1º Las notas de cada una de las asignaturas de los alumnos de una clase se han almacenado en un fichero cuyo nombre coincide con el de la asignatura y están dadas en forma tal que la línea i-ésima se da la nota del alumno i-ésimo, cuyo nombre se incluye en el fichero "nombres", también uno por línea.

Se trata de hacer un programa Java que genere una tabla y almacene en un fichero con la lista de notas de forma que en las columnas se incluyan los nombres de los alumnos, ordenados alfabéticamente, y las notas obtenidas en las asignaturas con los encabezamientos correspondientes.

2º Escribir un programa que se encargue de leer el contenido de un fichero de texto y crear un nuevo fichero en el que escriba el texto resultante de cambiar cada carácter del texto inicial por el siguiente en la tabla ASCII.

3º Dado un fichero que contiene texto, se pide:

a) Escribir un programa que liste todas las palabras que contiene y el número de veces que se repite cada una de ellas. Se considerarán equivalentes las letras mayúsculas y minúsculas.

b) Suponiendo que se dispone de un fichero con todas las palabras del diccionario, buscar todas las palabras del texto del fichero anterior que tienen capicúa, es decir, aquellas palabras tales que al invertirlas dan lugar a nuevas palabras (contenidas en el diccionario).

c) Detectar las palabras que no están en el diccionario y buscar las nuevas palabras contenidas en él y que resultan de éstas modificando un letra. Informar al usuario dando la lista de las mismas.