Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

El paquete java.io

OBJETIVO

En esta práctica se presenta el paquete `java.io` para realizar las operaciones de entrada y salida de datos, incluyendo las operaciones de lectura/escritura en archivos.

1 Introducción

El lenguaje Java proporciona una serie de clases para facilitar las tareas de lectura y escritura de ficheros. Estas clases consideran muchos de los casos que se pueden dar en este tipo de operaciones.

Los programas Java realizan la entrada y salida de datos a través de ***flujos*** (streams). Un flujo es una abstracción que produce o consume información. Los flujos están relacionados con los dispositivos físicos a través del sistema de entrada/salida de Java. Todos los flujos tienen un comportamiento similar, incluso aunque estén relacionados con distintos dispositivos físicos. Por esta razón, se pueden aplicar las mismas clases y métodos de entrada/salida a cualquier tipo de dispositivo. Esto significa que un flujo de entrada puede abstraer distintos tipos de entrada, desde un archivo de disco a un teclado o a otro dispositivo. Del mismo modo, un destino de salida puede ser una consola, o un archivo de disco. Los flujos son una manera limpia de tratar con la entrada/salida sin necesitar que el código comprenda la diferencia entre los distintos dispositivos.

En el lenguaje de programación C, los flujos (o stream) también se usan con las operaciones de entrada y salida, tanto para el manejo de ficheros como para realizar la entrada y salida, mediante los 3 flujos predefinidos de tipo `FILE *` (`stdin`, `stdout` y `stderr`). La aproximación de Java a los flujos es bastante similar a la de C.

2 Clasificación inicial.

A la hora de trabajar con los ficheros de datos, y como hacen otros lenguajes, Java considera dos tipos:

- *Ficheros binarios*: contienen cualquier tipo de dato en general. Estos tipos de ficfheros pueden contener cualquier tipo de dato: números en cualquier codificación, caracteres de texto, objetos, o/y combinaciones de los anteriores, y usan unidades de datos de 8 bits.

- Ficheros cuyo contenido se identifica con elementos imprimibles: números, letras (mayúsculas o minúsculas), signos de puntuación,.... Son los que comúnmente se conocen como *ficheros de texto*, son datos que se pueden contener en los tipos de datos `char` o `String`, y que usan unidades de datos de 16 bits.

Java proporciona una gran cantidad de clases para la lectura, creación y escritura de ficheros, que se concentran en el paquete `java.io`. Los casos en los que se pueden utilizar son muy amplios.

Para poder utilizar este código, los ficheros Java deben incluir al principio la línea:

```
import java.io.*;
```

3 Flujos predefinidos

Como ya sabemos, todos los programas Java importan automáticamente el paquete `java.lang`. Este paquete define una clase llamada `System` que encapsula algunos aspectos del entorno de ejecución. Por ejemplo, utilizando algunos de sus métodos se puede obtener la hora actual o los valores de algunas propiedades asociadas con el sistema. Además, contiene tres variables con flujos predefinidos llamadas `in`, `out` y `err`. Estas variables están declaradas como `public` y `static` en `System`. Esto significa que se pueden utilizar en cualquier parte del programa y sin tener una referencia a un objeto `System` específico.

`System.out` se refiere al flujo de salida estándar. Por defecto, es la consola. `System.in` hace referencia a la entrada estándar, que es, por defecto, el teclado. `System.err` se refiere al flujo de error estándar, que, por defecto, también es la consola. Sin embargo, estos flujos pueden ser redirigidos a cualquier otro dispositivo de E/S compatible.

En las prácticas anteriores ya se ha utilizado `System.out`. De igual forma se puede utilizar `System.err`. En esta práctica veremos el uso de `System.in`.

4 Ficheros binarios.

Java implementa los flujos dentro de una jerarquía de clases definida en el paquete `java.io`. En la parte superior de la jerarquía hay dos clases abstractas: `InputStream` y `OutputStream`. Java tiene algunas subclases concretas de cada una de ellas para gestionar las diferencias que existen entre los distintos dispositivos, como archivos de disco, conexiones de red y búferes de memoria.

Las clases abstractas `InputStream` y `OutputStream` definen algunos métodos que las otras clases implementan. Dos de los métodos más importantes son `read()` y `write()`, que respectivamente lee bytes del flujo y escribe bytes en el flujo. Ambos métodos están declarados como abstractos dentro de `InputStream` y `OutputStream` y son implementados en las clases derivadas.

Vamos a presentar la forma de trabajar con ficheros binarios. Todas las operaciones con este tipo de ficheros giran en torno a dos clases: `FileInputStream` para lectura de datos de un fichero binario, y `FileOutputStream` para la escritura. También vamos a ver una serie de clases que complementan el funcionamiento de las dos mencionadas añadiendo funcionalidades interesantes para mejorar el rendimiento de los programas y la facilidad de uso de la interfaz por parte del usuario.

4.1 Lectura de ficheros binarios.

Existe una clase básica que implementa las funciones de lectura, denominada `java.io.FileInputStream` con unos métodos que permiten leer a nivel de byte, y luego una serie de clases que complementan su funcionamiento añadiendo mejoras de rendimiento o de flexibilidad a la hora de realizar las lecturas.

- La clase `FileInputStream` tiene varios constructores, siendo el más interesante el constructor al que se le pasa un `String` con la ruta al fichero que se quiere leer:

```
FileInputStream fich = new FileInputStream(“./lista.bin”);
```

- Los métodos principales son los siguientes:
    - `close()`: cierra el fichero liberando todos los recursos asociados a la lectura.
    - `read`: método para la lectura a nivel de byte. Tiene las siguientes variantes:
        - `read()`: lee un byte del fichero y lo devuelve como resultado de la operación. Si no hay caracteres devuelve -1 si es el final del fichero. Ejemplo:

          ```
          byte datoLeido = fich.read();
          ```

        - `read(byte[] buffer, int despl, int long)`: lee como máximo `long` bytes y los almacena en buffer a partir de la posición `despl`. Devuelve el número de bytes leidos o -1 si no ha podido leer nada por fin de fichero.
        - `read(byte[] buffer)`: lee bytes y los guarda en buffer. Como máximo lee la longitud de buffer. Devuelve el número de bytes leídos o -1 si no ha podido leer nada por fin de fichero.

    Estos no son los únicos métodos de esta clase, algunas clases más se presentan en la siguiente tabla.

Clase	Significado
BufferedInputStream	Flujo de entrada con búfer
BufferedOutputStream	Flujo de salida con búfer
ByteArrayInputStream	Flujo de entrada que lee una matriz de bytes
ByteArrayOutputStream	Flujo de salida que escribe una matriz de bytes
DataInputStream	Flujo de entrada que contiene métodos para leer los tipos básicos de Java.
DataOutputStream	Flujo de salida que contiene métodos para escribir los tipos básicos de Java.
FileInputStream	Flujo de entrada que lee de un archivo
FileOutputStream	Flujo de salida que escribe en un archivo
FilterInputStream	Implementa a InputStream para realizar filtrado.
FilterOutputStream	Implementa a OutputStream para realizar filtrado.
InputStream	Clase abstracta que define un flujo de entrada
LineNumberInputStream	Flujo de entrada que contiene líneas.
OutputStream	Clase abstracta que define un flujo de salida
PipedInputStream	Flujo de entrada de tipo Piped.
PipedOutputStream	Flujo de salida del tipo Piped.
PrintStream	Flujo de salida que contiene los métodos `print()` y `println()`.
PushbackInputStream	Flujo de entrada que, una vez leído un byte, permite que se devuelva de nuevo al flujo de salida.
RandomAccessFile	Permite acceso aleatorio a un archivo de E/S.
SequenceInputStream	Flujo de entrada que es una combinación de dos o más flujos de entrada que serán leídos secuencialmente, uno después de otro.
StreamTokenizer	Divide el flujo de entrada en símbolos delimitados por conjuntos de caracteres.
StringBufferInputStream	Flujo de entrada que lee de una cadena.

Algunas de las clases de este paquete para el manejo de bytes que son de utilidad son las siguientes:

- `BufferedInputStream`: esta clase facilita el problema de velocidad de ejecución por acceso a disco físico implementando una memoria caché intermedia de donde leer los datos. Al constructor se le pasa un objeto de tipo `InputStream`, o cualquier otro objeto que herede de él, como es `FileInputStream`. No hay métodos adicionales de interés a la hora de leer tipos de datos.

- `DataInputStream`: añade métodos que permiten leer tipos de datos primitivos de java, como pueden ser `int`, `long`, `short`, `float`, `double`,...Tiene un solo constructor al que se le pasa un objeto de tipo `InputStream`, o cualquier otro objeto que herede de él, como es `FileInputStream` o `BufferedInputStream`. En cuanto a los métodos añade uno por cada tipo de dato primitivo que lee:

  - `readBoolean()`: devuelve un valor de tipo booleano leído del fichero. Ejemplo:

    ```
    boolean bandera = fich.readBoolean();
    ```

  - `readByte()`: devuelve un valor de tipo byte leído del fichero.

- `readDouble()`: devuelve un valor de tipo `double` leído del fichero.
- `readFloat()`: devuelve un valor de tipo `float` leído del fichero.
- `readInt()`: devuelve un valor de tipo `int` leído del fichero.
- `readShort()`: devuelve un valor de tipo `short` leído del fichero.
- ….

La forma de crear un objeto que nos permitiera leer de un fichero utilizando un buffer intermedio para mejorar la velocidad de acceso y que nos permitiera leer tipos primitivos de java sería la siguiente:

```
FileInputStream fich = new FileInputStream(“./lista.bin”);
BufferedInputStream bis = new BufferedInputStream(fich);
DataInputStream dis = new DataInputStream(bis);
…
int datoEntero = dis.readInt();
```

4.2 Escritura de ficheros binarios

La clase principal que nos permite crear y/o añadir contenidos a un fichero binario es `java.io.FileOutputStream`. Esta clase nos permite escribir en un fichero a nivel de byte.

La clase cuenta con varios constructores siendo de interés para esta práctica los siguientes:

- `FileOutputStream(String nombreFichero)`: se le pasa como argumento un `String` con el camino al fichero a crear o abrir para añadir contenido. Si el fichero no existe lo crea, y si existe lo borra y lo crea de nuevo. Ejemplo:

```
FileOutputStream fich =new FileOutputStream(“./listado.txt”);
```

- `FileOutputStream(String nombreFichero, boolean append)`: se le pasa como argumento un `String` con el camino al fichero a crear o abrir para añadir contenido y una variable de tipo boolean que indica si se añade el contenido al final del fichero en caso de que ya exista el mismo o no. Ejemplo:

```
FileOutputStream fich =new FileOutputStream(“./listado.txt”, true);
```

Los métodos principales de esta clase son:

- `close()`: cierra el fichero liberando todos los recursos asociados a la escritura.
- `write`: método para la escritura de caracteres. Las posibilidades de uso son las siguientes:
    - `void write(int c)`: escribe el byte representado por `c`. Ejemplo:

```
int datoAEscribir = 0xA3;
fich.write(datoAEscribir);
```

    - `void write(byte[] buffer, int desp, int long)`: escribe parte de los bytes incluido en la tabla de `byte` referenciada por `buffer`, comenzando desde la posición `desp`, hasta un total de `long` bytes.
    - `void write(byte[] buffer)`: escribe todos los bytes incluidos en `buffer`.

También existen una serie de clases complementarias que aumentan las funcionalidades de esta clase. Las más interesantes para esta práctica son las siguientes:

- `BufferedOutputStream`: esta clase aumenta la velocidad de ejecución de las aplicaciones usando una memoria caché intermedia y reduciendo el número de accesos físicos para escritura al disco. Al constructor se le pasa un objeto de tipo `OutputStream`, o cualquier otro objeto que herede de él, como es `FileOutputStream`. No hay métodos adicionales de interés a la hora de escribir tipos de datos.

- `DataOutputStream`: incrementa la interfaz de la clase de escritura a ficheros añadiendo métodos para escribir tipos de datos primitivos de java, como pueden ser `int`, `long`, `short`, `float`, `double`,...Tiene un solo constructor al que se le pasa un objeto de tipo `OutputStream`, o cualquier otro objeto que herede de él, como es `FileOutputStream` o `BufferedOutputStream`. En cuanto a los métodos añade uno por cada tipo de dato primitivo:
  - `writeBoolean(boolean dato)`: escribe el valor `dado` de tipo `booleano` en el fichero. Ejemplo:

```
boolean bandera = false;
fich.writeBoolean(bandera);
```

  - `writeByte(byte dato)`: escribe el valor de dato de tipo byte en el fichero.
  - `writeDouble(double dato)`: escribe el valor de dato de tipo `double` en el fichero.
  - `writeFloat(float dato)`: escribe el valor de dato de tipo `float` en el fichero.
  - `writeInt(int dato)`: escribe el valor de `dato` de tipo `int` en el fichero.
  - `writeShort(short dato)`: escribe el valor de `dato` de tipo `short` en el fichero.
  - ….

La forma de crear un objeto en java que aprovechara las ventajas de estas dos clases sería la siguiente:

```
FileOutputStream     fich = new FileOutputStream(“./lista.bin”);
BufferedOutputStream bos  = new BufferedOutputStream(fich);
DataOutputStream     dos  = new DataOutputStream(bis);
…
int datoEntero = 12500;
dos.writeInt(datoEntero);
```

5 Ficheros de texto.

Tal y como ya se ha indicado Java implementa los flujos dentro de una jerarquía de clases definida en el paquete `java.io`. En la parte superior de la jerarquía hay dos clases abstractas: `InputStream` y `OutputStream`, para el manejo de flujos de 8 bits. Y a partir de estas clases Java tiene algunas subclases concretas de cada una de ellas para gestionar las diferencias que existen entre los distintos dispositivos, como archivos de disco, conexiones de red y búferes de memoria.

Para el manejo de flujos de 16 bits o flujo de texto la organización es similar. En este caso las clases abstractas son, Reader y Writer, que contienen métodos similares a las clases abstractas `InputStream` y `OutputStream`, pero en este caso las unidades mínimas gestionadas son de 16 bits.

5.1 Ficheros de Texto

Si para la lectura de ficheros binarios todas las operaciones con este tipo de ficheros giran en torno a dos clases: `FileInputStream` para lectura de datos de un fichero binario, y `FileOutputStream` para la escritura. Para los ficheros de texto las operaciones giran en torno a `FileReader` y `FileWriter`. Por tanto, para la lectura, creación y escritura de ficheros se utilizan las siguientes clases del paquete `java.io`.: `java.io.FileReader` y `java.io.FileWriter` y un conjunto de clases asociadas con estas.

5.2 Lectura de ficheros de texto

La clase `java.io.FileReader` (o simplemente `FileReader` si hemos incluido la sentencia `import` indicada en un apartado anterior) proporciona una serie de métodos para una lectura básica de un fichero de texto.

Esta clase tiene tres constructores, de los cuales vamos a utilizar el constructor al que se le pasa una cadena de texto en un `String` con la ruta al fichero que queremos abrir. Un ejemplo de cómo se instancia un objeto de esta clase es el siguiente:

```
FileReader fich = new FileReader(“./listin.txt”);
```

Al ejecutar esta sentencia y si todo ha ido bien tenemos una referencia al fichero que podemos utilizar para leer datos en forma de caracteres. Los métodos de estas clases que son de interés son los siguientes:

- `close()`: cierra el fichero liberando todos los recursos asociados a la lectura.
- `read`: método para la lectura de caracteres. Tiene las siguientes variantes:
    - `read()`: lee un carácter del fichero y lo devuelve como resultado de la operación. Si no hay caracteres devuelve -1 si es el final del fichero. Ejemplo:

```
System.out.println(“caracter leido:”+fich.read());
```

    - `read(char[] buffer, int despl, int long)`: lee como máximo long caracteres y los almacena en `buffer` a partir de la posición `despl`. Devuelve el número de caracteres leídos o -1 si no ha podido leer nada por fin de fichero.

- `read(char[] buffer)`: lee caracteres y los guarda en `buffer`. Devuelve el número de caracteres leídos o -1 si no ha podido leer nada por fin del fichero.

Estos no son los únicos métodos de esta clase, si quiere conocer el resto consulta la documentación del paquete java disponible en Internet.

La clase `FileReader` presenta dos problemas:

- Solo permite leer caracteres como unidad de referencia, no permite otros métodos de lectura. Como hemos visto solo presenta un método `read` para acceder a los datos.
- Esta clase cada vez que necesita datos accede físicamente al disco. No utiliza ningún buffer intermedio de datos, ralentizando mucho el acceso al fichero y por tanto el rendimiento de las aplicaciones.

Para solucionar estos dos problemas se utiliza otra clase que complementa las funcionalidades de `FileReader: java.io.BufferedReader`. Esta clase utiliza un Buffer o memoria intermedia antes de acceder a disco por lo que mejora el rendimiento de los accesos. Esta clase tiene un constructor al que se le pasa como argumento el objeto de la clase `FileReader` al que complementa. Ejemplo:

```
BufferedReader fichBuff = new BufferedReader(fich);
```

En lo que se refiere a métodos de la clase, incluye la posibilidad de leer líneas de caracteres completas utilizando el método `readLine()`, al que no se le pasan argumentos y devuelve un `String` con la línea leída del fichero. Ejemplo:

```
String lineaLeida = fichBuff.readline();
```

5.3 Escritura de Ficheros de Texto.

La clase que se utiliza para la escritura de fichero de texto es `java.io.FileWriter`. El funcionamiento es idéntico al de `FileReader` pero orientado a la escritura de ficheros:

- Respecto al constructor, la clase tiene varios constructores posibles, pero nosotros vamos a considerar dos de ellos a la hora de hacer esta práctica:
  - `FileWriter(String nombreFichero)`: se le pasa como argumento un `String` con el camino al fichero a crear o abrir para añadir contenido. Si el fichero no existe lo crea, y si existe lo borra y lo crea de nuevo. Ejemplo:

    ```
    FileWriter fich = new FileWriter("./listado.txt");
    ```

  - `FileWriter(String nombreFichero, boolean append)`: se le pasa como argumento un String con el camino al fichero a crear o abrir para añadir contenido y una variable de tipo `boolean` que indica si se añade el contenido al final del fichero en caso de que ya exista el mismo o no. Ejemplo:

    ```
    FileWriter fich = new FileWriter("./listado.txt", true);
    ```

- Los métodos principales de esta clase son:
- `close()`: cierra el fichero liberando todos los recursos asociados a la escritura.
- `write`: método para la escritura de caracteres. Tenemos las siguientes posibilidades:

- `void write(int c)`: escribe el carácter representado por c.
- `void write(char[] buffer, int desp, int long)`: escribe parte de los caracteres incluidos en la tabla de `char` referenciada por buffer, comenzando desde la posición `desp`, hasta un total de `long` caracteres.
- `void write(String texto, int desp, int long)`: escribe parte de los caracteres incluidos en el `String` referenciado por texto, comenzando desde la posición `desp`, hasta un total de `long` caracteres.

- `FileWriter` tiene los mismos dos problemas que vimos con `FileReader` en lo que se refiere a acceso continuo a disco con los problemas de ralentización de los programas, así como que sus métodos no manejan el concepto de línea.

  Sí presenta una diferencia en cómo propone soluciones para estas cuestiones, mientras que en el caso anterior usando una única clase `BufferedReader` de complemento solucionábamos las dos circunstancias, en el caso de escritura es necesario utilizar dos clases complementarias, resolviendo cada una de ellas uno de los problemas:

  - `BufferedWriter`: esta clase nos soluciona el problema de los accesos continuos al fichero, al utilizar una zona intermedia de memoria donde acumula datos antes de grabarlos en disco. Al constructor se le pasa un objeto de tipo `FileWriter`. Los métodos son los mismos de `FileWriter`.
  - `PrintWriter`: esta clase proporciona métodos adicionales al `write` para escribir datos en un fichero, incluidos los que introducen el carácter separador de línea. Al constructor se le pasa un objeto de tipo `Writer` (como por ejemplo `BufferedWriter` o `FileWriter`). En cuanto a métodos de escritura en ficheros podemos destacar los siguientes:

- `println(char[] buffer)`: escribe en el fichero los caracteres incluidos en la tabla de caracteres buffer, y añade un salto de línea.
- `println(String cadena)`: escribe el `String` cadena en el fichero y un salto de línea.

  La utilización de las tres clases de manera combinada se haría de la siguiente manera:

```
FileWriter fich = new FileWriter(“./listado.txt”, true);
BufferedWriter bw = new BufferedWriter(fich);
PrintWriter pw = new PrintWriter(bw);
```

6 Tratamiento de errores

A la hora de trabajar con acceso a disco y ficheros se pueden producir muchas circunstancias que pueden dar lugar a errores: intentar leer de ficheros que no existen, intentar crear ficheros que ya están creados, intentar leer un fichero al que no se tiene permiso de acceso, intentar escribir en un fichero pero no hay espacio libre en disco,....

Para tratar todos los posibles errores y situaciones anómalas que puedan producirse en la gestión de ficheros se utiliza el mecanismo de excepciones estándar de java. Las excepciones que se generan en caso de errores son del tipo IOException o de alguna clase que hereda de esta (como FileNotFoundException ó EOFException).

Para poder tratar estas excepciones debemos utilizar las estructuras de control de java try {…} catch {…} finally {…} explicado en una práctica anterior.

Siempre que se trabaja con clases que manejan ficheros, es muy importante para mantener la consistencia de estos ficheros (especialmente cuando trabajamos con zonas de memoria intermedia para mejorar el rendimiento) en cerrar los flujos que generan dichas clases una vez finalizado el uso de los mismos. Este cierre se debe garantizar en cualquier circunstancia, tanto si el funcionamiento ha sido correcto como si se ha producido alguna excepción que ha provocado el final del flujo normal deseable de nuestra aplicación. Es por eso que estas operaciones de cierre se ejecutan siempre dentro de la clausula finally.

No obstante la ejecución del método close() que realiza este cierre de los flujos, también puede generar una excepción del tipo IOException si encuentra algún problema cuando va a cerrar los flujos. Para poder capturar y tratar esta excepción es necesario utilizar nuevamente una estructura try {…} catch {…} anidada con la que estamos tratando. De esta forma la estructura para la gestión de esta situación sería la siguiente:

```
try {
…
FileWriter fichero = new FileWriter(“/home/alumno/ejemplo.txt”);
…
} catch (IOException error1) {
…(Tratamieto error)...
}finally {
   try {
      fichero.close();
   } catch (Exception error2) {
   ...(Tratamiento error de cierre de fichero)...
   }
}
```

Ejemplos entregados.

1. Utilice la herramienta **`make`** para compilar los ejemplos que se van a ver a continuación de la siguiente forma.

   `make compila`

   y ejecute el código de este primer ejemplo en Java de la siguiente forma:

   `make ejecuta01`

   Nota: a partir de ahora en la mayor parte de las ejecuciones, cada prueba de los ejemplos se realiza mediante **`make ejecutaXX`**, con **`XX`** que indica el número del ejercicio que se está probando.

```
package fp2.poo.practica8;

import java.io.InputStream;
import java.io.IOException;

public class Practica8Ejercicio01 {
    public static void main(String args[]){
       InputStream in   = null;
       int cr           = 0;
       int total   = 0;
       int espacio = 0;

       try{
           in = System.in;

           System.out.println("Introduccion de datos desde la entrada estandar.");
           System.out.println("Introduzca datos y pulse la tecla ENTER.");
           System.out.println("La lectura de datos finaliza con el caracter 'q'.");

           for(total = 0; (cr = in.read()) != 'q' ; total++){
               if(Character.isWhitespace((char)cr))
                   espacio++;
           }

           in.close();
           System.out.println(total + " caracteres, "  + espacio + " espacios");
      }catch(IOException e){
         System.out.println(e);
      }
    }
}
```

En este ejemplo, se realiza la lectura de datos con flujos del tipo binario desde la entrada estándar. La lectura finaliza cuando se encuentra con el carácter ‘q’. Por pantalla se muestra el número total de caracteres leídos y el número de espacios en blanco introducidos desde la entrada estándar. Para comprobar si la información proporcionada desde el teclado es un espacio en blanco se utiliza el método estático `isWhitespace` de la clase `Character`.

2. Ejecute el siguiente ejemplo de la siguiente forma:

`make ejecuta02`

```
/**
 * Descripcion: En este ejemplo se realiza la lectura de bytes de la entrada
 * estándar. Es el mismo caso que el de Practica8Ejercicio01.java, pero en este
 * caso el flujo de entrada se "envuelve" en un flujo de tipo InputStreamReader
 * para realizar la conversion de 8 a 16 bits.
 *
 * version 1.0 Mayo 2014
 * Fundamentos de Programacion II
 */

package fp2.poo.practica8;

import java.io.InputStream;
import java.io.IOException;
import java.io.InputStreamReader;

public class Practica8Ejercicio02 {
    public static void main(String args[]){
       InputStreamReader in   = null;
       int cr           = 0;
       int total   = 0;
       int espacio = 0;

       try{
           in = new InputStreamReader(System.in);
           System.out.println("Introduccion de datos desde la entrada estandar.");
           System.out.println("Introduzca datos y pulse la tecla ENTER.");
           System.out.println("La lectura de datos finaliza con el caracter 'q'.");

           for(total = 0; (cr = in.read()) != 'q' ; total++){
               if(Character.isWhitespace((char)cr))
                   espacio++;
           }

           in.close();
           System.out.println(total + " caracteres, "  + espacio + " espacios");
      }catch(IOException e){
         System.out.println(e);
       }
    }
}
```

El resultado de la ejecución de este segundo ejemplo es el mismo que el del primero. En este caso se ha añadido la transformación de un flujo de 8 bit en uno de 16 bits, mediante el uso del constructor de la clase `InputStreamReader`. Esta clase construye una “envolvente” al flujo que se le proporciona como parámetro, en este caso `System.in`.

3. Este ejemplo se puede ejecutar de dos formas: proporcionando un argumento en línea de comando que será el fichero a leer, o bien sin argumento en línea de comandos donde los datos se proporcionarán desde teclado. Para realizar la ejecución del programa sin argumentos en línea de comandos, teclee:

`make ejecuta03SinFichero`

Para realizar la ejecución del programa mediante un argumento en línea de comandos, teclee:

`make ejecuta03ConFichero`

Que invoca a:

`java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio03`

y

`java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio03 ./ficherosEntrada/listado.txt`

respectivamente.

```
package fp2.poo.practica8;

import java.io.IOException;
import java.io.OutputStream;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;

/**
 * Recuenta el numero de bytes de un fichero.
 * Se proporciona el nombre como argumento en
 *  linea de comandos. Sino se proporciona fichero se lee
 *  de la entrada estándar (System.in)
 */
public class Practica8Ejercicio03 {
    public static void main( String args[] ) {

      InputStream in = null;
      int total           = 0;

      try{
         if(args.length == 0){
            in = System.in;
        }else{
            in = new FileInputStream(args[0]);
        }
        int c = 0;

        while( (c = in.read()) != -1 ) {
            System.out.write((char)c);
            total++;
        }

        System.out.println("Numero total de caracteres leidos: " + total);
      }catch(FileNotFoundException e){
          System.out.println("fichero no encontrado "+e);
      }catch(IOException e){
          System.out.println(e);
      }finally {
          in.close();
      }
    }
}
```

En el caso de realizar la lectura del fichero, este se toma de **`./ficherosEntrada/listado.txt`**.

Nota: Para finalizar la lectura desde teclado pulse las teclas **Control**+**d**.

4. Ejecute el siguiente ejemplo de la siguiente forma:

`make ejecuta04`

que ejecuta la siguiente orden:

```
java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio04 ./ficherosSalida/FicheroSalidaPrueba04.txt
```

```
package fp2.poo.practica8;

import java.io.IOException;
import java.io.OutputStream;
import java.io.FileOutputStream;

public class Practica8Ejercicio04 {

   public static void main(String[] args) {
      byte[] b = {'F', 'u', 'n', 'd', 'a','m', 'e', 'n', 't', 'o','s',' ',
                  'd','e',' ',
                  'P', 'r', 'o', 'g', 'r', 'a', 'm', 'a', 'c', 'i', 'o', 'n', ' ',
                  'I', 'I', '\n'};
      OutputStream os = null;
      try {
         os = new FileOutputStream(args[0]);
         for (int i = 0; i < b.length; i++) {
            os.write(b[i]);
            System.out.print("" + (char)b[i]);
         }
          os.close();
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

En este ejemplo se realiza la escritura en un archivo utilizando clases que manejan información de tipo binario.

El nombre del archivo se proporciona como argumento en línea de comandos. El flujo es del tipo: `FileOutputStream`, y usa el método `write` para realizar la escritura.

5. Para realizar la ejecución del programa teclee:

`make ejecuta05`

La llamada se realiza de la siguiente forma:

```
java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio05 ./ficherosSalida/FicheroSalidaPrueba05.txt
```

```
package fp2.poo.practica8;

import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.io.FileOutputStream;

public class Practica8Ejercicio05 {
    public static void main(String args[]){
       InputStream  in   = null;
       OutputStream out   = null;
       int cr            = 0;

       try{
           in  = System.in;
           out = new FileOutputStream(args[0]);

           for( ; (cr = in.read()) != -1 ;  ){
              out.write((byte)cr);
              System.out.print("" + (char)cr);
         }
          out.close();
          in.close();
      } catch ( IOException e ) {
         e.printStackTrace();
      }
   }
}
```

En este ejemplo se proporciona el nombre del fichero donde se va a realizar la escritura como argumento en línea de comandos (`./ficherosSalida/FicheroSalidaPrueba05.txt`).

Los datos leídos se proporcionan desde la entrada estándar. Y se muestran (a modo de eco) por pantalla.

Para finalizar pulse **Control-d.**

6. Para realizar la ejecución del programa teclee:

`make ejecuta06`

La llamada se realiza de la siguiente forma:

```
java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio06 ./ficherosEntrada/listado.txt
./ficherosSalida/FicheroSalidaPrueba06.txt
```

```
package fp2.poo.practica8;

import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.Reader;

public class Practica8Ejercicio06 {
   public static void main(String[] args) {

      try {
         InputStream  in = new FileInputStream( args[0]);
         OutputStream os = new FileOutputStream(args[1]);
         int          cr = 0;

        System.out.println("Se han abierto los dos ficheros " + args[0] + " y " + args[1] );
        System.out.println("Comienza la copia de un fichero en otro. ");

        while((cr = in.read()) != -1 ) {
            System.out.print("" + (char) cr);
            os.write((byte)cr);
         }

      } catch ( FileNotFoundException e) {
          System.err.println("Fichero no encontrado");
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

Este ejemplo realiza la copia de un fichero de entrada y que existe en otro fichero de salida.

7. Para realizar la ejecución del programa teclee:

`make ejecuta07`

```
/**
 * Descripción: Uso de un flujo buferado de un fichero.
 *
 *
 * version 1.0 Mayo 2014
 * Fundamentos de Programacion II
 */

package fp2.poo.practica8;

import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.BufferedInputStream;
import java.io.FileNotFoundException;
import java.io.Reader;

public class Practica8Ejercicio07 {
    public static void main(String[] args) {
        FileInputStream     fin   = null;
        BufferedInputStream bis   = null;
        int                 c     = 0;

        try {
            if ( args.length == 1 ){

                fin = new FileInputStream(args[0]);
                bis = new BufferedInputStream(fin);

                while ( ( c = bis.read()) != -1 ) {
                    System.out.print( (char)c );
                }
            } else {
                System.err.println("Proporcione el nombre del fichero.");
            }
        } catch (FileNotFoundException e) {
            System.err.println(e);
        } catch (IOException e) {
            System.err.println(e);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

Este programa muestra un ejemplo de buferado de los datos de un fichero. El nombre del fichero se le proporciona como argumento en línea de comandos. El buferado se realiza proprocionándole como parámetro a la clase `BufferedInputStream`, el flujo de tipo `FileInputStream` asociado al archivo.

8. Para realizar la ejecución del programa teclee:

`make ejecuta08`

```
package fp2.poo.practica8;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;

public class Practica8Ejercicio08 {
    public static void main(String[] args) {

        FileInputStream       is    = null;
        BufferedInputStream   bis   = null;
        BufferedOutputStream  bos   = null;
        FileOutputStream      os    = null;

             int                  value = -1;

        try{
            is   = new FileInputStream( args[0] );
            bis  = new BufferedInputStream(is);

            os = new FileOutputStream ( args[1] );
            bos  = new BufferedOutputStream(os);

            while ((value = bis.read()) != -1) {
                bos.write(value);
            }

            /* Invoca a flush para forzar que los bytes se escriban en el flujo. */
            bos.flush();
      }catch(IOException e){
         // if any IOException occurs
         e.printStackTrace();
      }finally{
          try{
              if(is!=null)
                  is.close();
              if(bis!=null)
                  bis.close();
              if(os!=null)
                  os.close();
              if(bos!=null)
                  bos.close();
          }catch(IOException e){
             // if any IOException occurs
             e.printStackTrace();
          }
      }
   }
}
```

En este ejemplo se realiza el buferado de ambos flujos, el de entrada y el de salida.

9. Para realizar la ejecución del programa teclee:

`make ejecuta09`

para la ejecución se esta forma se invoca de la siguiente forma:

`java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio09 a A`

```
/*
 *  @(#)Practica8Ejercicio09.java
 *
 *  Fundamentos de Programacion II. GITT.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 *
 */


/**
 * Descripcion:
 *
 * version 1.0 Mayo 2014
 * Fundamentos de Programacion II
 */

package fp2.poo.practica8;

import java.io.IOException;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

/**
 *  Programa que copia su entrada en su salida,
 *  transformando un valor particular en otro.
 *  Con valores proporcionados desde linea de comandos.
 */
public class Practica8Ejercicio09 {
    public static void main( String args[] ) {

      try{
          byte desde = (byte)args[0].charAt(0);
          byte hacia = (byte)args[1].charAt(0);
          int b  = 0;

          while( (b = System.in.read()) != -1)
              System.out.write( (b == desde) ? hacia : b);

      }catch(IOException e){
          System.out.println(e);
      }catch(IndexOutOfBoundsException e){
          System.out.println(e);
      }
    }
}
```

En este ejemplo se toman el primer carácter de los dos argumentos en línea de comando. El ejemplo realiza un cambio de todas las ocurrencias del primer carácter y las transforma en el del otro carácter.

10. Para realizar la ejecución del programa teclee:

`make ejecuta10`

```
package fp2.poo.practica8;

import java.io.IOException;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

/**
 *  Un flujo Pushback nos permite enviar de vuelta
 *  cuando la aplicación lo requiera.
 *  Pushback es generalmente útil para dividir la entrada en tokens.
 *  Por ejemplo, los exploradores léxicos a menudo sólo saben que un token
 *  ha terminado (como por ejemplo un identificador) cuando leen el primer
 *  carácter que lo sige. Una vez visto ese carácter, el explorador debe
 *  devolverlo al flujo de entrada, para que quede disponible como el
 *  primer carácter del siguiente token.
 *
 *  Este ejercicio usa un flujo PushbackInputStream para indicar la secuencia
 *  consecutiva más larga con el mismo byte de entrada.
 * Los resultados los muestra por pantalla.
 */
import java.io.IOException;
import java.io.PushbackInputStream;

public class Practica8Ejercicio10 {
    public static void main(String[] args) throws IOException{
        PushbackInputStream in =  new PushbackInputStream (System.in);
        int max = 0;   // secuencia más larga encontrada
        int maxB= -1;  // el byte de esa secuencia
        int b;         // byte actual de la entrada

        do{
            int cnt;
            int b1 = in.read();   // primer byte de la secuencia
            for(cnt = 1; (b = in.read()) == b1 ; cnt++)
                ;
            if(cnt > max){
                max = cnt;   //recuerda la longitud
                maxB= b1;    //recuerda el valor del byte
            }
            in.unread(b);    //devuelve el inicio de la siguiente sec.
        }while(b != -1);     //hasta el final de la entrada.
        System.out.println(max + " bytes de " + (char)maxB);
    }
}
```

Este programa lee la secuencia de caracteres idénticos más larga de un flujo. Usa el método `unread`, de la clase `PushbackInputStream`, para devolver datos al flujo.

11. Para realizar la ejecución del programa teclee:

`make ejecuta11SinFichero`

o bien

`make ejecuta11ConFichero`

Para realizar la lectura de teclado o bien del fichero.

```
package fp2.poo.practica8;

import java.io.IOException;
import java.io.InputStreamReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.Reader;

public class Practica8Ejercicio11 {
    public static void main(String args[]){
       Reader in   = null;
       int cr      = 0;
       int total   = 0;
       int espacio = 0;
      try{

       if( args.length == 0){
           in = new InputStreamReader(System.in);
       }else{
           in = new FileReader(args[0]);
       }
       for(total = 0; (cr = in.read()) != -1 ; total++){
           if(Character.isWhitespace((char)cr))
               espacio++;
       }
       in.close();
       System.out.println(total + " caracteres, "  + espacio + " espacios");
      }catch(FileNotFoundException e){
         System.out.println(e);
      }catch(IOException e){
         System.out.println(e);
      }
    }
}
```

Realiza la lectura de ficheros mediante la clase `FileReader`.

12. Para realizar la ejecución del programa teclee:

`make ejecuta12`

esta invoca al programa de la siguiente forma

java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio12 ./ficherosSalida/FicheroSalidaPrueba12.txt

```
package fp2.poo.practica8;

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.FileWriter;
import java.io.FileNotFoundException;
import java.io.Writer;

public class Practica8Ejercicio12 {
    public static void main(String args[]){
        InputStream       inp  = null;
        InputStreamReader in   = null;
        Writer            out  = null;
        int               cr   = 0;

        try{
            inp  = System.in;
            out = new FileWriter( args[0] );
            in  = new InputStreamReader(System.in);

            for( ; (cr = in.read()) != -1 ;  ){
                out.write((byte)cr);
                System.out.print("" + (char)cr);
            }
            out.close();
            in.close();
        } catch ( IOException e ) {
            e.printStackTrace();
        }
    }
}
```

En este programa se realiza la escritura en un fichero de tipo `FileWriter.`

Trabajo a entregar

Implemente la clase **`TrabajoAEntregar`** que en el paquete **`fp2.poo.practica8.XXX`** (siendo XXX el uvus).

En la clase **`TrabajoAEntregar`**, se debe realizar la lectura de datos con flujos del tipo binario desde la entrada estándar. La lectura finalizará cuando se encuentra con el carácter 'q'. Por pantalla se muestra el número total de caracteres leídos y el número de espacios en blanco introducidos desde la entrada estándar. Para comprobar si la información proporcionada desde el teclado es un espacio en blanco se utiliza el método estático `isWhitespace` de la clase `Character`.

Comprima el código en un archivo de nombre el uvus del alumno y entréguelo en la Actividad de la práctica.