El paquete java.io.

Manejo de las I/O.

Leo Suarez

leo@javahispano.com

http://www.javahispano.com

Julio 2001

El paquete java.io. Manejo de las I/O.

En este artículo presentamos el paquete que el API estándar de Java nos proporciona para gestionar las operaciones de I/O tanto del sistema como desde/a ficheros.

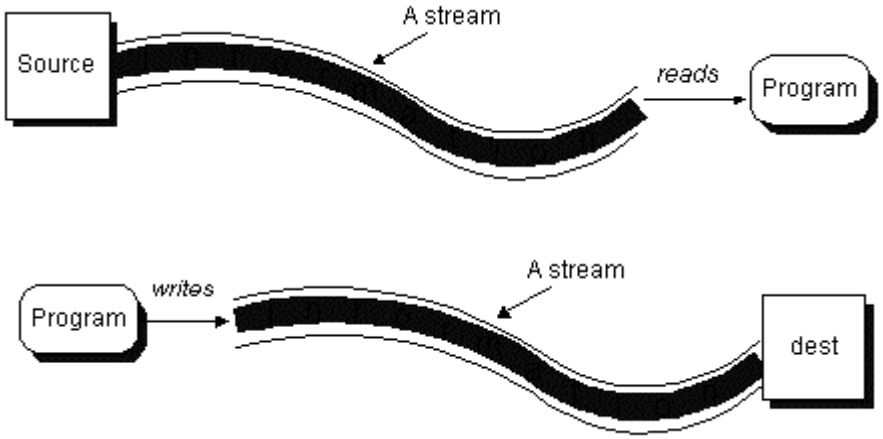
Como sabemos, Java es un lenguaje orientado a red, cuyo potencial radica en todo aquello que tenga que ver con la ejecución de aplicaciones a través de la red y, por tanto, este paquete se ajusta perfectamente a esta condición permitiendo la gestión de ficheros desde sitios remotos. Evidentemente, Java va mucho más allá del típico println(), cuyo uso más normal y natural es el de hacer los "debugging" de la aplicación que estemos creando.

Por último, para los diseñadores de applets, recalcar que Java nos permite escribir sobre un fichero si previamente hemos autentificado y/o firmado nuestro applet.

Índice.

Los streams.

Cualquier programa realizado en Java que necesite llevar a cabo una operación de I/O lo hará a través de un **stream**. Un stream, cuya traducción literal es "flujo", es una abstracción de todo aquello que produzca o consuma información. Podemos ver a este stream como una entidad lógica. La vinculación de este stream al dispositivo físico la hace el sistema de entrada y salida de Java. Se ve pues la eficacia de esta implementación pues las clases y métodos de I/O que necesitamos emplear son las mismas independientemente del dispositivo con el que estemos actuando, luego, el núcleo de Java, sabrá si tiene que tratar con el teclado, el monitor, un sistema de ficheros o un socket de red liberando a nuestro código de tener que saber con quién está interactuando.



Java2 define dos tipos de streams:

- **Byte streams** : Nos proporciona un medio adecuado para el manejo de entradas y salidas de bytes y su uso lógicamente está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene gobernado por dos clases abstractas que son **InputStream** y **OutputStream**. Cada una de estas clases abstractas tienen varias subclases concretas que controlan las diferencias ente los distintos dispositivos de I/O que se pueden utilizar. Así mismo, estas dos clases son las que definen los métodos que sus subclases tendrán implementados y, de entre todas, destacan las clases *read()* y *write()* que leen y escriben bytes de datos respectivamente.

- **Character streams** : Proporciona un medio conveniente para el manejo de entradas y salidas de caracteres. Dichos flujos usan codificación Unicode y, por tanto, se pueden internacionalizar. Una observación: Este es un modo que Java nos proporciona para manejar caracteres pero al nivel más bajo todas las operaciones de I/O son orientadas a byte. Al igual que la anterior el flujo de caracteres también viene gobernado por dos clases abstractas: **Reader** y **Writer**. Dichas clases manejan flujos de caracteres Unicode. Y también de ellas derivan subclases concretas que implementan los métodos definidos en ellas siendo los más destacados los métodos *read()* y *write()* que, en este caso, leen y escriben caracteres de datos respectivamente.

En ambos casos, los métodos son sobrecargados por las clases derivadas de ellos.

Los streams predefinidos.

Para comprender un poco mejor cómo funciona el paquete de I/O, veamos como trabaja realmente la clase System.

System define tres campos que son in, out y err que corresponden a la entrada estandar, la salida estandar y la salida estandar de errores. System.in es un objeto de tipo **InputStream** mientras que los otros dos son de tipo **PrintStream**. La pregunta que ahora te harás es ¿cómo es posible que si yo uso la clase Sytem para mandar texto , por ejemplo a la salida, sea de tipo Byte? Pues bien, esto se soluciona haciendo un “wrapping” (envoltorio) de la clase para convertirla a un flujo de caracteres. Veamoslo en el siguiente ejemplo:

Lectura de consola.

La entrada de consola, como sabemos, la obtenemos leyendo de System.in. Para conseguir un flujo de caracteres envolvemos dicha clase en un objeto del tipo **BufferedReader**, el cual soporta un flujo de entrada buferizado.

Atendiendo a las especificaciones de esta clase, el parámetro que se le pasa es el stream de entrada que es de tipo **Reader**, el cual, como sabemos es abastracto por lo que recurriremos a una de sus subclases, en nuestro caso será **InputStreamReader** que convierte bytes a caracteres. Otra vez más, atendiendo a la especificación de esta última clase vemos que el parámetro que se le pasa es de tipo **InputStream**, o sea, la entrada orientada a byte que en nuestro caso es System.in. Ya está, ya hemos asociado un dispositivo físico (el teclado) a un stream orientado a caracteres mediante el wrapping de la clase System con la clase BufferedReader. Resumamos los pasos:

1. BufferedReader (Reader input); clase que recibe un flujo de caracteres de entrada.
2. InputStreamReader (InputStream input2) ; clase que convierte de byte a carácter.
3. BufferedReader br = new BufferedReader(new InputStreamReader(System.in); br es un Character Stream que se linka a la consola a traves de la clase System.in la cual hemos tenido que wrappear para convertir de byte a char.

Ejemplo 1. Lectura de consola.

```
//En este ejemplo leemos de consola cadenas de caracteres.
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
public class LecturaStringConsola {
 public static void main(String args[]) throws IOException {
  String cadena;
  BufferedReader br;
  //Creamos un BufferedReader a trevés de System.in
  br = new BufferedReader(new InputStreamReader(System.in));
  System.out.println("Empieza a escribir, 'stop' para salir");
  //leemos cadena de caracteres mediante readLine().
  do {
   cadena = br.readLine();
   System.out.println(cadena);
  } while(!cadena.equals("stop"));
 }
}//Fin LecturaStringConsola
```

Imprimir a consola.

Lo que siempre hacemos es imprimir mediante System.out pero en una aplicación más formal haremos uso de la clase **PrintWriter** que soporta flujos de caracteres. Esta clase contiene los métodos *print()* y *println()* para todos los tipos por lo que se podrán usar igual que lo hacíamos con System.out. Si un argumento no es de un tipo simple, entonces, los métodos de PrintWriter llaman al método *toString()* y luego imprimen el resultado.

Ejemplo 2. Imprimir a pantalla.

//En el siguiente ejemplo básico imprimimos a consola diferentes tipos de datos primitivos.

```
import java.io.PrintWriter;
public class Imprimir {
 public static void main(String args[]) {
  /*
  El constructor toma como parámetros un objeto de tipo OutputStream del
cual deriva PrintStream, por tanto, ya tenemos
  linkada la clase con el dispositivo de salida (la consola).
  */
  PrintWriter pw = new PrintWriter(System.out, true);
  pw.println("Imprime una cadena de texto");
  int i = 15;
  pw.println("Imprime un entero " + i);
  double d = 6.8e-9;
  pw.println("Imprime un double " + d);
 }
}
```

Lectura y escritura de ficheros.

En Java, todos los ficheros son orientados a byte por lo que nos proporciona métodos para leer y escribir desde/a un fichero. No obstante, también nos permite hacer “wrapping” de dicho flujo orientado a byte para convertirlo a un objeto basado en caracteres.

Las dos principales clases que nos posibilitan trabajar con los ficheros son **FileInputStream** y **FileOutputStream** las cuales crean un enlace entre el flujo de bytes y el fichero. Para abrir un fichero simplemente le pasamos al constructor de estas clases el nombre de éste, luego con los métodos *read()* y *write()* actuaremos sobre él y finalmente mediante el método *close()* cerramos la sesión con el fichero. Más adelante se detallan un poco más estas clases y las acompañamos de algunos ejemplos útiles.

ADVERTENCIA: No confundas las dos clases anteriores con la clase **File**. Ésta, para empezar, no trabaja sobre un flujo de bytes sino que trata directamente con el fichero y con el sistema de ficheros. Es decir, con esta clase no accedemos a los datos de los ficheros, está orientada a obtener y/o manipular la información asociada a éste como, por ejemplo, permisos, fechas, si es un fichero o un directorio, el path, etc... Así cuando creamos una instancia de File, lo que estamos haciendo es establecer un enlace con un archivo o directorio físico al que luego le podremos consultar todas sus propiedades. Lo recomendable, para poder acceder a los datos del fichero y a sus propiedades es crear una instancia a la que le pasamos un objeto File de la siguiente manera:

```
File f = new File(“ruta_del_fichero”); instancia del descriptor del
fichero.
FileInputStream fis = new FileInputStream( f ); instancia de la clase que
nos permite leer los datos.
```

Ejemplo 3. Obtención de las propiedades de un fichero.

```
/*
En este ejemplo presentamos algunos de los métodos que la clase File nos
proporciona para conocer los detalles de un fichero.
*/
import java.io.File;
import java.io.IOException;
import java.net.MalformedURLException;
class FileDemo {
  static void imprimir(String s) {
  System.out.println(s);
 }
 public static void main(String args[]) {
  File fich = new File("pon la ruta del fichero que quieras");
  imprimir("Nombre del fichero : " + fich.getName());
  imprimir("Path relativo : " + fich.getPath());
  imprimir("Path Absoluto : " + fich.getAbsolutePath());
  imprimir("Directorio padre : " + fich.getParent());
  imprimir(fich.exists() ? "existe" : "no existe");
  imprimir("Ultima modificación del fichero : " + fich.lastModified());
  imprimir(fich.canWrite() ? "es de escritura" : "no es de escritura");
  imprimir(fich.canRead() ? "es de lectura" : "no es de lectura");
  imprimir(fich.isFile() ? "fichero normal" : "no normal");
  imprimir("Tamaño del fichero : " + fich.length() + "Bytes");
  try {
  imprimir("URL del fichero : " +fich.toURL());
  } catch (MalformedURLException urle) {imprimir(urle.getMessage());}
  fich.setReadOnly();//también podemos modificar sus atributos.
 }
}//Fin clase FileDemo
```

Las clases orientadas a flujo de bytes (Bytes Stream).

En apartados anteriores habíamos discutido que en la cima de la jerarquía de estas clases estaban las clases abstractas **InputStream** y **Outputstream**. Veámos ahora las subclases concretas más importantes:

ByteArrayInputStream y ByteArrayOutputStream.

Implementan un flujo de entrada y salida de bytes respectivamente teniendo como fuente un array de bytes en el caso de la clase de entrada. En el caso de la clase que maneja la salida existen dos tipos de constructores: el 1º crea un buffer de 32 bytes y el 2º crea un buffer del tamaño que le digamos en el parámetro.

FileInputStream.

Como ya vimos, esta clase crea un **InputStream** que usaremos para leer los bytes del fichero. Define varios constructores de los cuales destacan el que le pasamos como parámetro un String que contiene la ruta completa del fichero y otro al que le pasamos un objeto de tipo File. En todos los casos, si el fichero no existe se debe lanzar la excepción FileNotFoundException. La sintáxis de los constructores más habituales es:

```
FileInputStream(String ruta_fichero);
FileInputStream(File fichero);//fichero es la descripción del archivo.
```

FileOutputStream.

Este, al contrario que el anterior crea un **OutputStream** para enviar un flujo de bytes a un fichero. Igualmente al constructor le podemos pasar la ruta completa del archivo o un objeto de tipo File. Dado que esta operación es de riesgo, la clase puede lanzar una **SecurityException**. Así mismo, se pueden lanzar las excepciones **IOException** y/o **FileNotFoundException** si, por ejemplo, el fichero no se puede crear. La sintáxis de los constructores más habituales es:

```
FileOutputStream(String ruta_fichero);
FileInputStream(File fichero);//fichero es la descripción del archivo.
```

IMPORTANTE: Si el fichero existe, cuando vayamos a escribir sobre él, se borrará a menos que le digamos que añada los datos al final de éste mediante el constructor FileOutputStream(String *filePath*, boolean *append*), poniendo *append* a true.

Antes de mostrarte algun ejemplo, veamos 1º los streams de bytes filtrados para luego combinarlos en el ejemplo.

Las clases orientadas al filtrado del flujo de bytes (Filtered Byte Stream).

Estas clases son simplemente wrappers de la entrada o salida de los streams. Dichas clases proporcionan de manera transparente a estos flujos una funcionalidad de un nivel superior dado que las clases orientadas a flujo de bytes quedan algo limitadas para "trabajar" sobre los datos leidos o sobre los datos que vamos a escribir.

Lo que se hace es acceder al flujo de bytes por medio de alguna de las clases o métodos comentados en los apartados anteriores (normalmente InputStream) y, luego, se "envuelven" con alguna de las clases orientadas a filtrarlos para realizar operaciones algo más complejas como, por ejemplo, buferizar los datos, traslación de caracteres o datos crudos o, una de las más importantes, convertir los bytes leidos a alguno de los tipos primitivos de Java (byte, short, int, etc...).

Las clases más representativas de este tipo son **FilterInputStream** y **FilterOutputStream** aunque a continuación pasaré a explicar las más útiles.

DataInputStream.

La importancia de esta clase radica en que nos permite convertir la lectura de un flujo de bytes en uno de los tipos primitivos que Java nos proporciona según la forma que nos interese "empaquetar" esos bytes leídos. El único constructor crea un FilterInputStream y guarda el argumento que se le pasa para usarlo posteriormente. Su sintáxis es:

```
DataInputStream(InputStream in);
```

DataOutputStream.

Esta clase hace lo inverso de la anterior. Escribe a un flujo de salida de bytes con alguno de los tipos de datos primitivos de Java. La sintáxis del constructor en este caso es:

```
DataOutputStream(OutputStream out);
```

BufferedInputStream.

Las clases orientadas a buferizar los flujos de bytes asignan un buffer de memoria a los streams de I/O. Este buffer le permite a Java realizar operacioes de I/O sobre más de un byte a la misma vez con lo que estas clases incrementan y optimizan las prestaciones a la hora de trabajar con estos datos de esta forma. Por tanto, esta clase nos permite "envolver" cualquier InputStream en un stream buferizado para optimizar el uso de la memoria. La sintáxis de los dos constructores que nos ofrece es:

```
BufferedInputStream (InputStream in);
BufferedInputStream (InputStream in, int bufSize);
```

La primera opción crea un stream buferizado usando el tamaño por defecto del buffer mientras que la segunda opción usa un bufer del tamaño indicado en *bufSize*. Una buena elección de este tamaño sería escoger un múltiplo aunque ésto depende de ciertos factores como el sistema operativo sobre el que se ejecute, la cantidad de memoria disponible, etc.

Una última observación respecto a las clases orientadas a buferizar datos es que implementan los métodos *mark()* y *reset()*, los cuales nos permiten acceder a datos del buffer ya leídos.

- mark(int *limite*), como su nombre indica hace una marca en el byte del stream sobre el que estemos situados en ese momento. Con *limite* indicamos el número de bytes que tienen que transcurrir antes de dejar de tener efecto la marca que hemos hecho.
- reset(); al llamar a este método, Java nos "reposiciona" en el byte del stream sobre el que habíamos hecho la marca.

BufferedOutputStream.

Esta clase es similar a OutputStream con la diferencia de que incorpora el método *flush()* con el cual aseguramos que los datos en el buffer de salida son transferidos físicamente al dispositivo de escritura. En este método es precisamente donde radica la ventaja de esta clase frente a otras destinadas a las escritura, reducir el número de veces que el sistema realmente escribe sobre el dispositivo. La diferencia es clara, no es lo mismo escribir byte a byte que enviar un flujo de N bytes (tamaño del buffer) de una tacada al dispositivo. La sintáxis de los constructores es similar al de la clase anterior:

```
BufferedOutputStream (OutputStream in);
BufferedOutputStream (OuputStream in, int bufSize);
```

Combinación de las clases.

A continuación veremos una serie de ejemplos que combinan las clases que tratan directamente sobre los flujos de entrada o de salida con las clases que las envuelven.

Ejemplo 4. Lectura de datos crudos (raw data).

```
/*
En este ejemplo leemos un fichero que contiene un flujo de bytes que
representa una imagen
2D de tamaño 128x128. Cada pixel viene codificado por 2 bytes que indican
el nivel de gris de dicho pixel. Así
mismo, escribimos en el fichero salida.txt los niveles de grises con un
"formato" de array 2D.
Este programa lee de manera correcta cualquier short, aunque exceda de
32767.
*/
import java.io.*;
class RawData {

 final static int WIDTH = 128;
 final static int HEIGHT = 128;
  public static void main (String args[]) {

   InputStream in = null;
   DataInputStream dis = null;
   FileOutputStream fout = null;
   PrintStream miSalida = null;
   int shortLeido = 0;
    try {
      in = new FileInputStream("Rana_128.035");//Leemos un stream de bytes
     dis = new DataInputStream(in);//Envolvemos la clase de entrada (in)
para acceder a las funcionalidades de DataInput
      fout =  new FileOutputStream("salida.txt");
      miSalida = new PrintStream(fout);//convertimos a PrintStream
    }
    catch(IOException e) {
      System.out.println("Error al abrir el fichero");
      System.exit(0);
    }
    catch (Exception e) {
      System.out.println(e.getMessage());
      System.exit(0);
    }

    int contX =0, contY=0;
    for (contX = 0; contX < HEIGHT; contX++) {
     miSalida.print("\n");
     for (contY = 0; contY < WIDTH; contY++) {
       try {
        shortLeido = dis.readUnsignedShort();//Al leer short sin signo
podemos tratar datos hasta 65535.
        miSalida.print("[ " + shortLeido + " ]  ");
       }
```

```
       catch(IOException e) {
         System.out.println("Se leyó todo el fichero " + e);
       }
       catch (Exception e) {
         System.out.println("excepción desconocida: " + e);
         System.exit(0);
       }
     }
    }  // fin for exterior
    try {
    in.close();
    fout.close();
    } catch (IOException ioe) {System.out.println("No se pudo cerrar alguno
de los ficheros");}

  }  // end main

} //Fin RawData
```

Ejemplo 5. Lectura a través de la red. Carga de un fichero desde un applet.

```
/*
En este ejemplo cargamos un fichero a través de la red. Este ejemplo es la
manera que se suele
emplear para cargar un fichero. Como nuestra intención es leer una ristra
de caracteres hacemos
un "wrapping" tal y como explicamos.
*/
/*
<applet code = ReadFromApplet.class width = 400 height = 400>
</applet>
*/
import java.applet.Applet;
import java.io.InputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.net.URL;
import java.net.MalformedURLException;
public class ReadFromApplet extends Applet {

 URL url;
 InputStream in;
 BufferedReader br;
 public void init() {
   try {
    url = new URL(this.getCodeBase() + "Propiedades.java");
   } catch (MalformedURLException mue)
{System.out.println(mue.getMessage());}

   try {
    in = url.openStream();//abre un flujo de bytes

    //hacemos el wrapping de la clase orientada a byte para leer caracteres
    br = new BufferedReader(new InputStreamReader(in));

    String linea;
```

```
    while ((linea = br.readLine()) != null) {
     System.out.println(linea);
    }

    br.close();
   } catch (IOException ioe) {System.out.println(ioe.getMessage());}
 }
}
```

Las clases orientadas a flujo de caracteres (Character Stream).

Aunque las clases orientadas al flujo de bytes nos proporcionan la suficiente funcionalidad para realizar cualquier tipo de operación de entrada o salida, éstas no pueden trabajar directamente con caracteres Unicode. Es por ello que fue necesario la creación de las clases orientadas al flujo de caracteres para ofrecernos el soporte necesario para el tratamiento de caracteres.

Como ya dijimos al principio, estas clases nacen de las clases abstractas **Reader** y **Writer**. Las clases concretas derivadas de ellas tienen su homónimo en las clases concretas derivadas de InputStream y OutputStream, por tanto, la explicación hecha para todas ellas tiene validez para las orientadas a caracter salvo quizás algunos matices que podrás completar con las especificaciones por lo que omitiremos la explicación detallada de cada una de ellas.

No obstante, os daré una relación de las clases más importantes. Son las siguientes:

1. Acceso a fichero: **FileReader** y **FileWriter.**
2. Acceso a caracter: **CharArrayReader** y **CharArrayWriter**.
3. Buferización de datos: **BufferedReader** y **BufferedWriter**.

Serialización.

La serialización es el proceso de escribir el estado de un objeto a un flujo de bytes. La utilidad de esta operación se manifiesta cuando queremos salvar el estado de nuestro programa en un sitio de almacenamiento permanente o, en otras palabras, cuando queremos hacer la persistencia de nuestro programa. Así, en un momento posterior dado podemos recuperar estos objetos deserializándolos.

Otra de las situaciones en las que necesitamos recurrir a la serialización es cuando hacemos una implementación RMI. La Invocación de Métodos Remotos consiste en que un objeto Java de una máquina pueda llamar a un método de un objeto Java que está en otra máquina diferente. Entonces, la máquina que lo invoca serializa el objeto y lo transmite mientras que la máquina receptora lo deserializa.

Dado que cuando un objeto se serializa, éste puede tener referencias a otros objetos que a la vez lo tendrán a otros, los métodos para la serialización y deserialización de objetos contemplan esta posibilidad. Así, cuando serializamos un objeto que está en la cima del grafo de objetos, todos los objetos a los que se hace referencia son también serializados. El proceso inverso de recuperación de objetos hará justo lo contrario.

La interface Serializable.

Para que un objeto sea almacenado y recuperado mediante las herramientas de serialización debe de implementar esta interface. Serializable no define campos ni métodos, tan solo es un identificador de que el objeto se puede serializar. Si una clase es serializable, entonces todas sus subclases lo son también.

La interface Externalizable.

Las herramientas de serialización han sido diseñadas para que la mayoría del trabajo se haga de manera automática. No obstante, hay situaciones en la que el programador estará interesado en tener el control de este proceso y para ello surge la interface Externalizable. Para ello, esta clase nos proporciona dos métodos para que en la clase que implementa la interface se tome el control del formato y contenido sobre el stream.

Las interfaces ObjectInput y ObjectOutput.

Estas interfaces extienden a DataOutput y DataInput respectivamente y soportan serialización. En otras palabras son clases de entrada y salida de flujos de datos pero con la particularidad de que están diseñadas para serializar los objetos. La implementación concreta de ellas son las siguientes clases:

ObjectOutputStream

Esta es la clase responsable de escribir los objetos a streams. El constructor de esta clase tiene la siguiente sintáxis:

```
ObjectOutputStream( OutputStream outStream);
```

El argumento es el flujo de salida sobre el cual el objeto serializado será escrito.

ObjectInputStream

Esta es la clase responsable de leer los objetos desde un stream. El constructor de esta clase tiene la siguiente sintáxis:

```
ObjectInputStream( InputStream inStream);
```

El argumento es el flujo de entrada desde el cual el objeto serializado se lee.

Ejemplo 6. Serialización de un objeto.

```
import java.io.*;
public class Serializacion {
 public static void main(String args[]) {

  //Serializamos el objeto
  try {
   MiClase obj1 = new MiClase("String", 15);
   System.out.println("Objeto 1: " + obj1);
   FileOutputStream fos = new FileOutputStream("serial");
   ObjectOutputStream oos = new ObjectOutputStream(fos);
   oos.writeObject(obj1);
   oos.flush();
   oos.close();
  } catch (Exception e)
{System.out.println(e.getMessage());System.exit(0);}

  //Deserialización del objeto
  try {
   MiClase obj2;
   FileInputStream fis = new FileInputStream("serial");
   ObjectInputStream ois = new ObjectInputStream(fis);
   obj2 = (MiClase)ois.readObject();
   ois.close();
   System.out.println("Objeto 2: " + obj2);
  } catch(Exception e) {System.out.println(e.getMessage());System.exit(0);}
 }
}
class MiClase implements Serializable {
 String s;
 int i;

 public MiClase(String s, int i) {
  this.s = s;
  this.i = i;
 }

 public String toString() {
  return "s=" + s + "; i=" + i;
 }
}
```

Conclusión.

De lo visto en este mini-tutorial se desprende que el paquete java.io nos ofrece mucho más juego del que en principio se podría pensar. Hemos visto cómo Java trabaja con los datos que lee o que va a escribir y también hemos aprendido como envolver una clase con otra que le pueda dar una funcionalidad mayor o, simplemente, una funcionalidad distinta a la que tienen las clases que inicialmente actuan con el fichero.

Habrás observado que a este documento lo he calificado como mini-tutorial. Considerarlo como tutorial sería dar por sentado que se ha tratado con todo lujo de detalles las clases y métodos que conforman este paquete y ciertamente eso no es así. Con este texto lo que he pretendido es que te hagas una idea bastante aproximada de las herramientas de que dispones para trabajar con ficheros y qué es lo que puedes hacer con ellas.

A continuación te doy una pequeña bibliografía con la que puedes completar este aprendizaje en caso que lo explicado aquí no te fuera del todo suficiente.

Bibliografía.

Java2: The Complete Reference. Autor: Patrick Naughton, Herbert Schildt.

The Java Tutorial: A practical guide for programmers. URL: http://java.sun.com/docs/books/tutorial/essential/io/index.html.

Java® 2 SDK, Standard Edition Documentation. URL: http://java.sun.com/j2se/

Leo Suarez
leo@javahispano.com
Julio 2001.