

CLASE 6 Entrada/ Salida de Datos

Los programas necesitan comunicarse con su entorno, tanto para recoger datos e información que deben procesar, como para devolver los resultados obtenidos.

La manera de representar estas entradas y salidas en Java es a base de streams (flujos de datos). Un stream es una conexión entre el programa y la fuente o destino de los datos. La información se traslada en serie (un carácter a continuación de otro) a través de esta conexión. Esto da lugar a una forma general de representar muchos tipos de comunicaciones.

Por ejemplo, cuando se quiere imprimir algo en pantalla, se hace a través de un stream que conecta el monitor al programa. Se da a ese stream la orden de escribir algo y éste lo traslada a la pantalla. Este concepto es suficientemente general para representar la lectura/ escritura de archivos, la comunicación a través de Internet o la lectura de la información de un sensor a través del puerto en serie.

6.1 Clases de Java para lectura y escritura de Datos

El package `java.io` contiene las clases necesarias para la comunicación del programa con el exterior. Dentro de este package existen dos familias de jerarquías distintas para la entrada/ salida de datos.

La diferencia principal consiste en que una opera con bytes y la otra con caracteres (el carácter de Java está formado por dos bytes porque sigue el código Unicode). En general, para el mismo fin hay dos clases que manejan bytes (una clase de entrada y otra de salida) y otras dos que manejan caracteres.

Desde Java 1.0, la entrada y salida de datos del programa se podía hacer con clases derivadas de `InputStream` (para lectura) y `OutputStream` (para escritura). Estas clases tienen los métodos básicos `read()` y `write()` que manejan bytes y que no se suelen utilizar directamente. La Figura 6.1 muestra las clases que derivan de `InputStream` y la Figura 6.2 las que derivan de `OutputStream`.

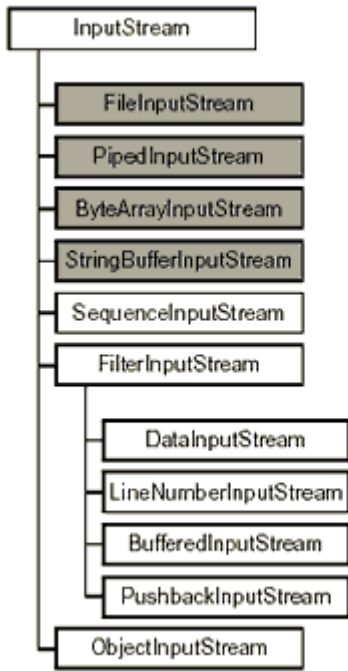


Figura 6.1. Jerarquía de clases InputStream.

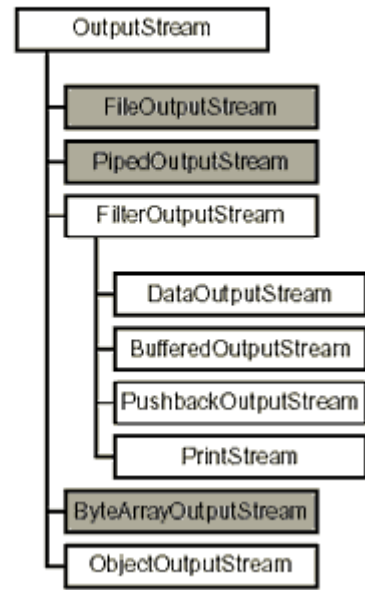


Figura 6.2. Jerarquía de clases OutputStream.

En Java 1.1 aparecieron dos nuevas familias de clases, derivadas de Reader y Writer, que manejan caracteres en vez de bytes. Estas clases resultan más prácticas para las aplicaciones en las que se maneja texto.

Las clases que heredan de Reader están incluidas en la Figura 6.3 y las que heredan de Writer en la Figura 6.4.

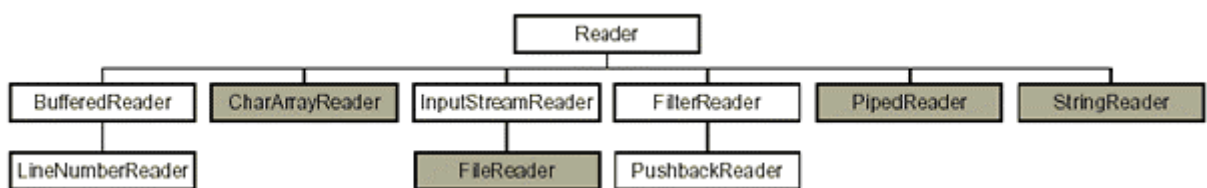


Figura 6.3: Jerarquía de clases Reader

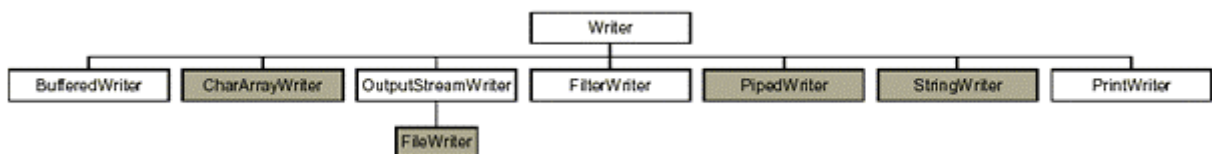


Figura 6.4: Jerarquía de clases Writer

En las cuatro últimas figuras las clases con fondo gris definen de dónde o a

dónde se están enviando los datos, es decir, el dispositivo con que conecta el stream. Las demás (fondo blanco) añaden características particulares a la forma de enviarlos. La intención es que se combinen para obtener el comportamiento deseado. Por ejemplo:

```
BufferedReader in = new BufferedReader(new  
FileReader("autoexec.bat"));
```

Con esta línea se ha creado un stream que permite leer del archivo autoexec.bat. Además, se ha creado a partir de él un objeto `BufferedReader` (que aporta la característica de utilizar buffer).

Los caracteres que lleguen a través del `FileReader` pasarán a través del `BufferedReader`, es decir utilizarán el buffer. A la hora de definir una comunicación con un dispositivo siempre se comenzará determinando el origen o destino de la comunicación (clases en gris) y luego se le añadirán otras características (clases en blanco).

Se recomienda utilizar siempre que sea posible las clases `Reader` y `Writer`. Algunas tareas como la serialización y la compresión necesitan las clases `InputStream` y `OutputStream`.

6.1.1 Los nombres de las clases de java.io

Las clases de `java.io` siguen una nomenclatura sistemática que permite deducir su función a partir de las palabras que componen el nombre, tal como se describe en la siguiente tabla.

Palabra	Significado
InputStream, OutputStream	Lectura/ Escritura de bytes
Reader, Writer	Lectura/ Escritura de caracteres
File	Archivos
String, CharArray, ByteArray, StringBuffer	Memoria (a través del tipo primitivo indicado)
Piped	Tubo de datos
Buffered	Buffer
Filter	Filtro
Data	Intercambio de datos en formato propio de Java
Object	Persistencia de objetos
Print	Imprimir

Tabla 9.1. Palabras significativas de java.io.

6.1.2 Clases que indican el origen o destino de los datos

La Tabla 6.2 explica el uso de las clases que definen el lugar con que conecta el stream.

Clases	Función que realiza
FileReader, FileWriter, FileInputStream y FileOutputStream	Son las clases que leen y escriben en archivos de disco. Se explicarán luego con más detalle
StringReader, StringWriter, CharArrayReader, CharArrayWriter, ByteArrayInputStream, ByteArrayOutputStream, StringBufferInputStream	Estas clases tienen en común que se comunican con la memoria del ordenador. En vez de acceder del modo habitual al contenido de un String, por ejemplo, lo leen como si llegara caracter a caracter. Son útiles cuando se busca un modo general e idéntico de tratar con todos los dispositivos que maneja un programa.
PipedReader, PipedWriter, PipedInputStream, PipedOutputStream	Se utilizan como un tubo o conexión bilateral para transmisión de datos. Por ejemplo, en un programa con dos threads pueden permitir la comunicación entre ellos. Un thread tiene el objeto PipedReader y el otro PipedWriter. Si los streams están conectados, lo que se escriba en el PipedWriter queda disponible para que se lea del PipedReader. También puede comunicar a dos programas distintos.

Tabla 6.2. Clases que indican el origen o destino de los datos.

6.1.3 Clases que añaden características

La Tabla 6.3 explica las funciones de las clases que alteran el comportamiento de un stream ya definido.

Clases	Función que realizan
BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream	Como ya se ha dicho, añaden un buffer al manejo de los datos. Es decir se reducen las operaciones directas sobre el dispositivo (lecturas de disco, comunicaciones por red), para hacer más eficiente su uso. BufferedReader por ejemplo tiene un método readLine() que lee una línea y la devuelve como un String
InputStreamReader, OutputStreamWriter	Son clases puente que permiten convertir streams que utilizan bytes en otros que manejan caracteres. Son la única relación entre ambas jerarquías y no existen clases que realicen la

	transformación inversa
ObjectInputStream, ObjectOutputStream	Pertencen al mecanismo de la serialización y se explicarán más adelante
FilterReader, FilterWriter, FilterInputStream, FilterOutputStream	Son clases base para aplicar diversos filtros o procesos al stream de datos. También se podrían extender para conseguir comportamientos o medida
DataInputStream, DataOutputStream	Se utilizan para escribir y leer datos directamente en los formatos propios de Java. Los convierten en algo ilegible, pero independiente de la plataforma y se usan por tanto para almacenaje o para transmisiones entre ordenadores de distinto funcionamiento
PrintWriter, PrintStream	Tiene métodos adaptados para imprimir las variables de Java con la apariencia normal. A partir de un boolean escriben "true" o "false", colocan la coma de un número decimal, etc.

Tabla 6.3. Clases que añaden características.

6.2 Entrada y Salida estándar (Teclado y Pantalla)

En Java, la entrada desde teclado y la salida a pantalla están reguladas a través 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.

Estas clases permiten la comunicación alfanumérica con el programa a través de lo métodos incluidos en la Tabla 6.4. Son métodos que permiten la entrada/salida a un nivel muy elemental.

Métodos de System.in	Función que realizan
int read()	Lee un carácter y lo devuelve como int.
Métodos de System.out y System.err	Función que realizan
int print(cualquier tipo)	Imprime en pantalla el argumento que se le pase. Puede recibir cualquier tipo primitivo de variable de Java
int println(cualquier tipo)	Como el anterior, pero añadiendo "\n" al final

Tabla 9.4. Métodos elementales de lectura y escritura.

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.

6.2.1 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.

6.2.2 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`.

6.2.3 Método práctico para leer desde teclado

Para facilitar la lectura de teclado se puede conseguir que se lea una línea entera con una sola orden si se utiliza un objeto `BufferedReader`. El método `String readLine()` perteneciente a `BufferedReader` lee todos los caracteres hasta

encontrar un '\n' o '\r' y los devuelve como un String (sin incluir '\n' ni '\r'). Este método también puede lanzar java.io.IOException.

System.in es un objeto de la clase InputStream. BufferedReader pide un Reader en el constructor. El puente de unión necesario lo dará InputStreamReader, que acepta un InputStream como argumento del constructor y es una clase derivada de Reader. Por lo tanto si se desea leer una línea completa desde la entrada estándar habrá que utilizar el siguiente código:

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
// o en una línea:
// BufferedReader br2 = new BufferedReader(
//         new InputStreamReader(System.in) );
String frase = br2.readLine(); // Se lee la línea con una llamada
```

Así ya se ha leído una línea del teclado. El thread que ejecute este código estará parado en esta línea hasta que el usuario termine la línea (pulse return). Es más sencillo y práctico que la posibilidad anterior.

¿Y qué hacer con una línea entera? La clase java.util.StringTokenizer da la posibilidad de separar una cadena de caracteres en las "palabras" (tokens) que la forman (por defecto, conjuntos de caracteres separados por un espacio, '\t', '\r', o por '\n'). Cuando sea preciso se pueden convertir las "palabras" en números.

La Tabla 6.5 muestra los métodos más prácticos de la clase StringTokenizer.

Métodos	Función que realizan
StringTokenizer(String)	Constructor a partir de la cadena que hay que separar
boolean hasMoreTokens()	¿Hay más palabras disponibles en la cadena?
String nextToken()	Devuelve el siguiente token de la cadena
int countTokens()	Devuelve el número de tokens que se pueden extraer de la frase

Tabla 6.5. Métodos de StringTokenizer.

La clase StreamTokenizer de java.io aporta posibilidades más avanzadas que StringTokenizer, pero también es más compleja. Directamente separa en tokens lo que entra por un InputStream o Reader.

Se recuerda que la manera de convertir un String del tipo "3.141592654" en el valor double correspondiente es crear un objeto Double a partir de él y luego extraer su valor double:


```
double pi = (Double.valueOf("3.141592654")).doubleValue();
```

El uso de estas clases facilita el acceso desde teclado, resultando un código más fácil de escribir y de leer. Además tiene la ventaja de que se puede generalizar a la lectura de archivos.

6.3 Lectura y Escritura de Archivos

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.

6.3.1 Clases `File` y `FileDialog`

Un objeto de la clase `File` puede representar un archivo o un directorio. Tiene los siguientes constructores:

```
File(String name)  
File(String dir, String name)  
File(File dir, String name).
```

Se puede dar el nombre de un archivo, el nombre y el directorio, o sólo el directorio, como path absoluto y como path relativo al directorio actual. Para saber si el archivo existe se puede llamar al método boolean `exists()`.

```
File f1 = new File("c:\\windows\\notepad.exe"); // La barra '\' se escribe '\\'
File f2 = new File("c:\\windows"); // Un directorio
File f3 = new File(f2, "notepad.exe"); // Es igual a f1
```

Si `File` representa un archivo que existe los métodos de la Tabla 6.6 dan información de él.

Métodos	Función que realizan
<code>boolean isFile()</code>	true si el archivo existe
<code>long length()</code>	tamaño del archivo en bytes
<code>long lastModified()</code>	fecha de la última modificación
<code>boolean canRead()</code>	true si se puede leer
<code>boolean canWrite()</code>	true si se puede escribir
<code>delete()</code>	borrar el archivo
<code>rename To(File)</code>	cambiar el nombre

Tabla 6.6. Métodos de File para archivos.

Si representa un directorio se pueden utilizar los de la Tabla 6.7:

Métodos	Función que realizan
<code>boolean isDirectory()</code>	true si existe el directorio
<code>mkdir()</code>	crear el directorio
<code>delete()</code>	borrar el directorio
<code>String [] list()</code>	devuelve los archivos que se encuentran en el directorio

Tabla 6.7. Métodos de File para directorios.

Por último, otros métodos incluidos en la Tabla 9.8 devuelven el path del archivo de distintas maneras.

Métodos	Función que realizan
<code>String getPath()</code>	Devuelve el path que contiene el objeto File
<code>String getNamer()</code>	Devuelve el nombre del archivo
<code>String getAbsolutePath()</code>	Devuelve el path absoluto (juntando el relativo al actual)
<code>String getParent()</code>	Devuelve el directorio padre

Tabla 6.8. Métodos de File que devuelven el path.

Una forma típica de preguntar por un archivo es presentar un caja de diálogo. La clase `java.awt.FileDialog` presenta el diálogo típico de cada sistema operativo para guardar o abrir ficheros. Sus constructores son:

```
FileDialog(Frame fr)
FileDialog(Frame fr, String title)
FileDialog(Frame fr, String title, int type)
```

donde `type` puede ser `FileDialog.LOAD` o `FileDialog.SAVE` según la operación que se desee realizar.

Es muy fácil conectar este diálogo con un `File`, utilizando los métodos `String getFile()` y `String getDirectory()`. Por ejemplo:

```
FileDialog fd = new FileDialog(f, "Elija un archivo");
fd.show();
File f = new File(fd.getDirectory(), fd.getFile());
```

6.3.2 Lectura de archivos de texto

Se puede crear un objeto `BufferedReader` para leer de un archivo de texto de la siguiente manera:

```
BufferedReader br = new BufferedReader(new FileReader("archivo.txt"));
```

Utilizando el objeto de tipo `BufferedReader` se puede conseguir exactamente lo mismo que en las secciones anteriores utilizando el método `readLine()` y la clase `StringTokenizer`. En el caso de archivos es muy importante utilizar el buffer puesto que la tarea de escribir en disco es muy lenta respecto a los procesos del programa y realizar las operaciones de lectura de golpe y no de una en una hace mucho más eficiente el acceso. Por ejemplo:

```
// Lee un archivo entero de la misma manera que de teclado
```

```
String texto = new String();
try {
    FileReader fr = new FileReader("archivo.txt");
    entrada = new BufferedReader(fr);
    String s;
    while((s = entrada.readLine()) != null)
        texto += s;
    entrada.close();
}
catch(java.io.FileNotFoundException fnfex) {
    System.out.println("Archivo no encontrado: " + fnfex);
}
catch(java.io.IOException ioex) { }
```

6.3.3 Escritura de archivos de texto

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 le 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();
}
catch(java.io.IOException ioex) { }
```

6.3.4 Archivos que no son de texto

`DataInputStream` y `DataOutputStream` son clases que no han sido modificadas hasta ahora. Para leer y escribir datos primitivos directamente (sin convertir a/de `String`) siguen siendo las más útiles.

Son clases diseñadas para trabajar de manera conjunta. Una puede leer lo que la otra escribe, que en sí no es algo legible, sino el dato como una secuencia de bytes. Por ello se utilizan para almacenar datos de manera independiente de la plataforma (o para mandarlos por una red entre ordenadores muy distintos).

El problema es que obligan a utilizar clases que descienden de `InputStream` y `OutputStream` y por lo tanto algo más complicadas de utilizar. El siguiente código primero escribe en el fichero `prueba.dat` para después leer los datos escritos:

```
// Escritura de una variable double
DataOutputStream dos = new DataOutputStream(
    new BufferedOutputStream(new FileOutputStream("prueba.dat")));
double d1 = 17/7;
dos.writeDouble(d);
dos.close();
// Lectura de la variable double
DataInputStream dis = new DataInputStream(
```

```
new BufferedInputStream(new FileInputStream("prueba.dat"));
double d2 = dis.readDouble();
```

6.4 Serialización

La serialización es un proceso por el que un objeto cualquiera se puede convertir en una secuencia de bytes con la que más tarde se podrá reconstruir dicho objeto manteniendo el valor de sus variables. Esto permite guardar un objeto en un archivo o mandarlo por la red.

Para que una clase pueda utilizar la serialización, debe implementar la interface `Serializable`, que no define ningún método. Casi todas las clases estándar de Java son serializables. La clase `MiClase` se podría serializar declarándola como:

```
public class MiClase implements Serializable { }
```

Para escribir y leer objetos se utilizan las clases `ObjectOutputStream` y `ObjectInputStream`, que cuentan con los métodos `writeObject()` y `readObject()`. Por ejemplo:

```
ObjectOutputStream objout = new ObjectOutputStream(
    new FileOutputStream("archivo.x"));
String s = new String("Me van a serializar");
objout.writeObject(s);
ObjectInputStream objin = new ObjectInputStream(
    new FileInputStream("archivo.x"));
String s2 = (String)objin.readObject();
```

Es importante tener en cuenta que `readObject()` devuelve un `Object` sobre el que se deberá hacer un casting para que el objeto sea útil. La reconstrucción necesita que el archivo `*.class` esté al alcance del programa (como mínimo para hacer este casting).

Al serializar un objeto, automáticamente se serializan todas sus variables y objetos miembro. A su vez se serializan los que estos objetos miembro puedan tener (todos deben ser serializables). También se reconstruyen de igual manera. Si se serializa un `Vector` que contiene varios `Strings`, todo ello se convierte en una serie de bytes. Al recuperarlo la reconstrucción deja todo en el lugar en que se guardó.

Si dos objetos contienen una referencia a otro, éste no se duplica si se escriben o leen ambos del mismo stream. Es decir, si el mismo `String` estuviera contenido dos veces en el `Vector`, sólo se guardaría una vez y al recuperarlo sólo se crearía un objeto con dos referencias contenidas en el vector.

6.4.1 Control de la serialización

Aunque lo mejor de la serialización es que su comportamiento automático es bueno y sencillo, existe la posibilidad de especificar cómo se deben hacer las

cosas.

La palabra clave `transient` permite indicar que un objeto o variable miembro no sea serializado con el resto del objeto. Al recuperarlo, lo que esté marcado como `transient` será 0, `null` o `false` (en esta operación no se llama a ningún constructor) hasta que se le dé un nuevo valor. Podría ser el caso de un `password` que no se quiere guardar por seguridad.

Las variables y objetos `static` no son serializados. Si se quieren incluir hay que escribir el código que lo haga. Por ejemplo, habrá que programar un método que serialice los objetos estáticos al que se llamará después de serializar el resto de los elementos. También habría que recuperarlos explícitamente después de recuperar el resto de los objetos.

Las clases que implementan `Serializable` pueden definir dos métodos con los que controlar la serialización.

No están obligadas a hacerlo porque una clase sin estos métodos obtiene directamente el comportamiento por defecto. Si los define serán los que se utilicen al serializar:

```
private void writeObject(ObjectOutputStream stream) throws IOException  
private void readObject(ObjectInputStream stream) throws IOException
```

El primero permite indicar qué se escribe o añadir otras instrucciones al comportamiento por defecto. El segundo debe poder leer lo que escribe `writeObject()`. Puede usarse por ejemplo para poner al día las variables que lo necesiten al ser recuperado un objeto. Hay que leer en el mismo orden en que se escribieron los objetos.

Se puede obtener el comportamiento por defecto dentro de estos métodos llamando a `stream.defaultWriteObject()` y `stream.defaultReadObject()`.

Para guardar explícitamente los tipos primitivos se puede utilizar los métodos que proporcionan `ObjectInputStream` y `ObjectOutputStream`, idénticos a los de `DataInputStream` y `DataOutputStream` (`writeInt()`, `readDouble()`, ...) o guardar objetos de sus clases equivalentes (`Integer`, `Double`...).

Por ejemplo, si en una clase llamada `Tierra` se necesita que al serializar un objeto siempre le acompañe la constante `g` (9,8) definida como `static` el código podría ser:

```
static double g = 9.8;  
private void writeObject(ObjectOutputStream stream) throws IOException {  
    stream.defaultWriteObject();  
    stream.writeDouble(g);  
}  
private void readObject(ObjectInputStream stream) throws IOException {  
    stream.defaultReadObject();
```

```
        g = stream.readDouble(g);
    }
```

6.4.2 Externalizable

La interface Externalizable extiende Serializable. Tiene el mismo objetivo que ésta, pero no tiene ningún comportamiento automático, todo se deja en manos del programador.

Externalizable tiene dos métodos que deben implementarse.

```
interface Externalizable {
    public void writeExternal(ObjectOutput out) throws IOException;
    public void readExternal(ObjectInput in) throws IOException,
    ClassNotFoundException;
}
```

Al transformar un objeto, el método writeExternal() es responsable de todo lo que se hace.

Sólo se guardará lo que dentro de éste método se indique. El método readExternal() debe ser capaz de recuperar lo guardado por writeExternal(). La lectura debe ser en el mismo orden que la escritura. Es importante saber que antes de llamar a este método se llama al constructor por defecto de la clase.

Como se ve el comportamiento de Externalizable es muy similar al de Serializable.

6.5 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);  
}
```