Tema 13 Ficheros

Contenido

1 Ficheros. Introducción

Un fichero o archivo es un conjunto de información sobre un mismo tema, tratada como unidad de almacenamiento y organizada de forma estructurada para la búsqueda y recuperación de un dato individual. Es por tanto la estructura que necesitamos usar para poder guardar los datos e informaciones en soportes de almacenamiento masivo o permanente, tales como discos duros, CD's, memorias Flash, cintas magnéticas, etc. y poder recuperarlos cada vez que queramos usarlos en una aplicación sin tener que volver a introducirlos de nuevo. Cada uno de estos dispositivos trabaja de forma distinta y también almacena la información de forma distinta.

Nuestra aplicación hará la petición de trabajar con ficheros tanto para lectura como para escritura, al Sistema Operativo, y éste se encargará de enmascarar los detalles de funcionamiento del dispositivo de entrada/salida, de forma que la aplicación siempre leerá o escribirá sobre ficheros más o menos de la misma forma.

2 Flujos (Streams)

Para trabajar con ficheros en Java se utiliza una abstracción que es concepto **de Flujo** o **Stream.**

Para Java, los datos son algo que fluye en una corriente desde un origen (productor) hasta un destino (consumidor).

Básicamente considera dos grandes tipos de flujos de datos:

- **De entrada (InputStream).** En ellos nuestra aplicación recibe los datos. Es el **consumidor** que retira los datos del flujo, que puede provenir de un teclado, o de un fichero, por ejemplo.
- **De salida (OutputStream).** En ellos nuestra aplicación **produce** los datos, y los envía al flujo para que lleguen hasta su destino, que puede ser el monitor, la impresora, o un fichero, por ejemplo.

Tanto **InputStream** como **OutputStream** son clases abstractas.

En Java, en los flujos se escriben o leen bytes, en principio, de forma que **cualquier información que quiero escribir en un flujo tengo que ir descomponiéndola en bytes, y escribiendo cada uno de ellos en el flujo de salida**. De la misma forma, **para recuperar esa información, tendré que ir leyendo uno a uno los bytes que me va proporcionando el flujo, y a partir de ellos recomponer la información original.**

El flujo es como una especie de "tubería unidireccional" en la que la aplicación va metiendo la información byte a byte si el flujo es de salida (escritura), o de la que la aplicación va recibiendo la información byte a byte si el flujo es de entrada (lectura).

La aplicación siempre ve el mismo extremo de la tubería, que siempre es igual. Pero la tubería en sí puede conectarse por el otro extremo a muy distintos orígenes o destinos de datos, a muy distintos dispositivos, sin que la aplicación tenga por qué saber nada sobre ellos, ni sobre sus características, ni sobre su forma de trabajar. Sólo debe tratar con el flujo, escribiendo bytes en la tubería de salida, o leyendo bytes de la tubería de entrada.

Es el flujo el que se comunica con el Sistema Operativo. De esta forma para que las operaciones de entrada/salida de nuestro programa funcionen usando otro dispositivo de entrada/salida o en cualquier otro sistema operativo no tenemos que cambiar nada en nuestra aplicación, que va a ser independiente tanto de los dispositivos físicos de almacenamiento como del sistema operativo sobre el que se ejecuta.

Cuando se ejecuta cualquier aplicación Java se crean automáticamente tres objetos que son flujos:

- **System.err**. Es un flujo de salida definido en la clase **System** y que representa la salida de error estándar. Por defecto, es el monitor, aunque es posible redireccionarlo a otro dispositivo de salida.
- **System.out**. Es el flujo de salida definido en la clase **System** que representa la salida estándar. Por defecto también es el monitor, aunque puede redireccionarse a otro dispositivo.
- **System.in** Está definido en la clase **System** como flujo de entrada estándar. Por defecto es el teclado, pero puede redirigirse para cada host o cada usuario, de forma que se corresponda con cualquier otro dispositivo de entrada.

3 Jerarquía de clases para Flujos en Java

Además de las clases base para entrada y salida orientada a byte, que son **InputStream** y **OutputStream**, existen otras dos clases base para entrada y salida orientada a caracteres y texto. Esas clases son **`Reader`** para entrada o lectura de caracteres y **`Writer`** para salida o escritura de caracteres. Estas clases están optimizadas para trabajar con caracteres y con texto en general, debido a que tienen en cuenta que cada carácter Unicode está representado por **dos bytes**.

En Java hay una amplia jerarquía de clases para trabajar con Streams que se indican en las tablas que hay a continuación. Sólo unas pocas identifican **orígenes o destinos de datos, es decir, distintos "dispositivos"** con los que se conecta el Stream. Aparecen marcadas en las tablas con **sombreado**. Las otras clases sólo añaden características que permiten modificar la forma de trabajar con el flujo, y la forma de enviar los datos hacia o desde el flujo.

Jerarquía de clases **`InputStream`**	Jerarquía de clases **`OutputStream`**
`InputStream`	`OutputStream`
`FileInputStream`. Origen de datos: un fichero.	**`FileOutputStream`** (*)
`PipedInputStream`. Origen de datos: aplicación que se ejecuta concurrentemente en otro hilo.	`PipedOutputStream` (*)
`ByteArrayInputStream`. Origen de datos: la memoria.	`ByteArrayOutputStream` (*)
`StringBufferInputStream`. Origen de datos: un flujo de caracteres desde un Stream.	`FilterOutputStream`
`SequenceInputStream`	`DataOutputStream`
`FilterInputStream`	`PrintStream`
`DataInputStream`	**`BufferedOutputStream`**
`LineNumberInputStream`	`PushbackOutputStream`
`BufferedInputStream`	**`ObjectOutputStream`**
`PushbackInputStream`	
`ObjectInputStream`	

Cada aumento de la sangría izquierda indica que son subclases de la clase que hay encima en el nivel anterior.

(*)Análogamente se pueden considerar como destino s de datos las clases similares de la jerarquía **`OutputStream`**, o las clases similares de las jerarquías **`Reader`** y **`Writer`**.

En la siguiente tabla se muestra la jerarquía de clases de `Reader` y `Writer`.

Jerarquía de clases **`Reader`**	Jerarquía de clases **`Writer`**
`Reader`	`Writer`
`CharArrayReader`. Origen de datos: un array de caracteres en memoria.	`CharArrayWriter` (*)
`PipedReader`. Origen de datos: aplicación que se ejecuta concurrentemente en otro hilo.	`PipedWriter` (*)
`StringReader`. Origen de datos: un String.	`StringWriter` (*)
`BufferedReader`. Es un "envoltorio" de FileReader. Proporciona un buffer para leer grupos de caracteres.	**`BufferedWriter`** (*)
`LineNumberReader`	`OutputStreamWriter`
`InputStreamReader`	**`FileWriter`**
`FileReader`. Sus métodos pueden **leer caracteres de uno en uno** en un flujo de caracteres asociado a un archivo.	`FilterWriter`
`FilterReader`	`PrintWriter`
`PushbackReader`	

4 Clase File

Las clases que nos permiten trabajar con ficheros en Java se encuentran disponibles en el paquete **java.io**, por lo que **lo primero que debemos hacer en toda aplicación que use las clases involucradas en la entrada/salida de ficheros es escribir la sentencia...**

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

Entre ellas se encuentra la clase **File**, que nos permite obtener información importante sobre el fichero con el que vamos a trabajar, como su ruta, **si es un fichero o un directorio**, el tamaño, última fecha de modificación, etc.

Esta clase (File) **NO** permite acceder a la información que contiene el propio fichero.

4.1 Constructores de File:

Constructor	**Explicación**
File(String nombre)	Recibe en la instanciación del objeto la ruta completa donde está el fichero junto con el nombre. Por defecto, si no se indica, lo busca en la carpeta del proyecto. **nombre** puede ser también la ruta a un directorio, sin indicar al final el nombre de ningún fichero.

File(String ruta, String nombre)	Recibe en la instanciación del objeto la ruta completa donde está el fichero, como primer parámetro, y el nombre del fichero como segundo parámetro.
File(File ruta, String nombre)	Recibe en la instanciación del objeto un objeto de tipo File, que hace referencia a un directorio, como primer parámetro y el nombre del fichero como segundo parámetro.

4.2 Algunos métodos de File

Para manejo de ficheros y directorios

Método	Explicación
boolean canRead()	Informa si se puede leer la información que contiene.
boolean canWrite()	Informa si se puede guardar información.
boolean exists()	Informa si el fichero o el directorio existen.
boolean isFile()	Informa si es un archivo.
boolean isDirectory()	Devuelve true si el objeto File corresponde a un directorio.
long lastModified()	Retorna la fecha de la última modificación.
String getName()	Devuelve el nombre del fichero o directorio.
String getPath()	Devuelve la ruta relativa.
String getAbsolutePath()	Devuelve la ruta absoluta.
String getParent()	Devuelve el nombre del directorio padre o null si no existe.
String[] list()	Devuelve un array de Strings que contiene los nombres de los archivos y directorios que contiene el directorio.
File[] listFiles()	Devuelve un array que contiene referencias a los archivos que contiene el directorio.

Sólo para manejo de ficheros

Método	Explicación
delete()	Borra el fichero.
long length()	Devuelve el tamaño del archivo en bytes.
boolean renameTo(File)	Cambia el nombre del fichero por el nombre del archivo pasado como argumento.

Sólo para manejo de directorios

Método	Explicación
boolean mkdir()	Crea el directorio.
String[] list()	Devuelve un listado de los archivos/directorios que se encuentran en el directorio.
File[] listFiles()	Devuelve un array que contiene referencias a los archivos que se encuentran en el directorio.

Hay que tener en cuenta que se puede:

- indicar el nombre de un fichero sin la ruta: se buscará el fichero en el directorio actual.
- indicar el nombre de un fichero con la ruta relativa.
- indicar el nombre de un fichero con la ruta absoluta.

Ejemplo:

```
import java.io.File;
public class EjemClaseFile {
    public static void main(String[] args) {
        File fichero;
        String resp, nombre;
        resp = Leer.pedirCadena("\n¿Nombre de fichero para ver si existe?\n "
                + "Escribe 'S ' para si- y cualquier otro caracter para no\t");
        while (resp.equalsIgnoreCase("S")) {
            nombre = Leer.pedirCadena("\n\tIndica el nombre de fichero a buscar: ");
            fichero = new File(nombre);
            if (fichero.isFile()) {
                System.out.println("\t\t El fichero existe");
            } else {
                System.out.println("\t\t El fichero no existe");
            }
            resp = Leer.pedirCadena("\nNombre de fichero para ver si existe?\n "
                    + "Escribe 'S ' para si- y cualquier otro caracter para no\t");
        }
    }
}
```

5 ESCRITURA Y LECTURA DE INFORMACIÓN EN FICHEROS

Para trabajar con flujos de bytes se usan las clases:

- FileInputStream
- FileOutputStream

Para trabajar con flujos de caracteres

- FileReader
- FileWriter

5.1 FileInputStream y BufferedInputStream

La clase **`FileInputStream`** es la clase básica **para leer datos desde un fichero**. Sirve para leer todo tipo de datos de todo tipo de ficheros. Esta clase trabaja con bytes, es decir, se leen las informaciones por bytes desde el flujo, que estará asociado a un fichero. Define un origen de datos, que es un fichero, y `BufferedInputStream` lo único que hace es modificar la forma de trabajar con ese flujo. En concreto, lo que hace es añadirle un buffer al flujo `FileInputStream`. Cuando una aplicación debe leer datos de un fichero, tiene que estar esperando a que el disco en el que está el fichero le suministre la información. Cualquier dispositivo de memoria masiva, por rápido que sea, es infinitamente más lento que la CPU del ordenador. Si nuestra aplicación accede con mucha frecuencia al dispositivo, estará obligando a la CPU a permanecer parada, a la espera de que le llegue desde el fichero el dato solicitado. Por tanto, resulta útil minimizar el número de accesos al fichero a fin de mejorar la eficiencia de la aplicación, y para ello se asocia al fichero una memoria intermedia (hace de intermediaria entre el fichero y la aplicación), de forma que cuando se necesita leer un byte del archivo, en realidad se traen hasta el buffer asociado al flujo (asociado a su vez al fichero) tanta información como le quepa. Mientras quede información en el buffer, nuestra aplicación leerá

los datos directamente de la memoria principal, donde se encuentra el buffer, que es mucho más rápida que cualquier dispositivo de memoria masiva. Así, reducimos el número de accesos al fichero, y mejoramos la eficiencia de nuestra aplicación, que tendrá distraída a la CPU el menor tiempo posible.

Ver: `EjemploFileInputStream`

Al instanciar un objeto de la clase **`FileInputStream`** se abre el fichero en modo lectura. Ya se puede leer byte a byte de modo secuencial.

Constructores de `FileInputStream`:

Constructor	Explicación	Excepción que lanza
FileIntputStream(String nombreFich)	Recibe como parámetro el nombre del fichero a abrir.	FileNotFoundException si el fichero no existe.
FileIntputStream(File fichero)	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar.	FileNotFoundException si el fichero no existe.

Métodos más usados de `FileInputStream`:

Método	Explicación
int read()	Devuelve el código ASCII del siguiente byte que hay después de donde está situado el puntero del fichero. Dicho puntero se va moviendo secuencialmente por el fichero, según vamos leyendo los bytes. Devuelve **-1** si no hay ningún byte más que leer.
int read(byte cadByte[])	Lee hasta cadByte.length bytes guardándolos en la tabla que se envía como parámetro. Devuelve -1 si no hay ningún byte más que leer.
void close()	Cierra el fichero.

5.2 FileOutputStream y BufferedOutputStream

La clase básica que nos permite usar un fichero para salida o escritura de bytes en él, es la clase `FileOutputStream`.

Un flujo de tipo **`FileOutputStream`**, puede construirse a partir de un objeto **`File`** o directamente a partir de un **`String`** que especifique un path. No obstante, también en este último caso se crea el objeto **`File`**, pero de forma implícita por el compilador, que es el que lo utiliza.

Al igual que antes, para mejorar la eficiencia de la aplicación reduciendo el número de accesos a los dispositivos de salida en los que se almacena el fichero, podemos montar un buffer asociado al flujo de tipo **`FileOutputStream`**. De eso se **encarga la clase `BufferedOutputStream`, que permite que la aplicación pueda escribir bytes en el flujo sin que necesariamente haya que llamar al sistema operativo subyacente para cada byte escrito.**

Ver: `EjemploFileOutputStream`

Constructores de `FileOutputStream`:

Constructor	Explicación
FileOutputStream(String nombreFich)	Recibe como parámetro el nombre del fichero a abrir y borra el contenido previo del fichero comenzando a escribir desde el principio.
FileOutputStream(File fichero)	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar y borra el contenido previo del fichero comenzando a escribir desde el principio.
FileOutputStream(String nombreDeFich, boolean append)	Recibe como parámetro el nombre del fichero a abrir y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.
FileOutputStream(File fichero, boolean append)	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.

Métodos más usados de `FileOutputStream`:

Método	Explicación
int write(int byte)	Escribe el byte que recibe como argumento en el fichero.
int write(byte cadByte[])	Escribe todos los bytes que contiene el array cadByte.
void close()	Cierra el fichero.

5.3 DataInputStream y DataOutputStream

Con las clases **FileInputStream** y **FileOutputStream** cada byte que lee o escribe del fichero lo va cogiendo o enviando al fichero de uno en uno, lo cual hace que dicha tarea sea lenta.

DataInputStream y DataOutputStream poseen más métodos, como la posibilidad de leer y escribir datos de una sola vez, como short, int, String, etc. Sin necesidad de tener que ir byte a byte.

Constructor	Explicación
DataOutputStream(OutputStream out)	Crea un nuevo flujo de salida de datos para escribir datos en el OutputStream indicado "out".
DataInputStream(InputStream in)	Crea un nuevo flujo de entrada de datos para leer datos en el InputStream indicado "in".

Para abrir un fichero con estas clases:

```
FileOutputStream fichero = new FileOutputStream("fich.dat");
DataOutputStream salida = new DataOutputStream(fichero);


FileInputStream fichero = new FileInputStream("fich.dat");
DataInputStream entrada = new DataInputStream(fichero);
```

Ver: `EjemploDataOutputStream y EjemploDataInputStream`

Métodos más usados:

DataInputStream	DataOutputStream
• readShort() • readInt() • readFloat() • readUTF() • readLine() • ...	• writeShort(short dato) • writeInt(int dato) • writeUTF(String dato) • ...

Al leer un fichero hay que utilizar el mismo formato con el que se ha escrito porque de no ser así daría errores en la ejecución al no corresponderse los tipos.

Para saber que se ha alcanzado el final del fichero, los métodos lanzan la excepción **`EOFException`**, así que hay que **recogerla** y **tratarla** correctamente.

5.4 Clases FileWriter, FileReader, BufferedWriter y BufferedReader

Existen dos clases base para entrada y salida orientada a caracteres y texto, que es recomendable usar cuando los datos que se manipulan son de tipo texto. Esas clases son **`Reader`** para entrada o lectura de caracteres y **`Writer`** para salida o escritura de caracteres. Estas clases están optimizadas para trabajar con caracteres y con texto en general, debido a que tienen en cuenta que cada carácter Unicode está representado por dos bytes.

Las subclases de **`Writer`** y **`Reader`** que permiten trabajar con ficheros de texto son:

- **`FileReader`**, para lectura desde un fichero de texto. Crea un flujo de entrada que trabaja con caracteres en vez de con bytes. El método read() recupera un carácter cada vez. Ver `EjemploFileReader`
- **`FileWriter`**, para escritura o salida hacia un fichero de texto. Crea un flujo de salida que trabaja con caracteres en vez de con bytes. Ver `EjemploFileWriter`

Estas clases sólo se podrán usar para manejar ficheros de texto, y además es recomendable usarlas en este caso, porque son más eficientes.

Por lo demás, la forma de trabajar con ellas es muy similar a la forma de trabajar con las clases vistas hasta ahora, aunque evidentemente disponen de sus propios métodos.

También se puede montar un buffer sobre cualquiera de los flujos que definen estas clases:

- **`BufferedWriter` se usa para montar un buffer sobre un flujo de salida de tipo `FileWriter`.**
- **`BufferedReader`** se usa para montar un buffer sobre un flujo de entrada de tipo **`FileReader`**. Ver `EjemploBufferedReader`

Métodos de `FileWriter`:

Constructores:

Constructor	Explicación
FileWriter(String nombreFich)	Recibe como parámetro el nombre del fichero a abrir y borra el contenido previo del fichero comenzando a escribir desde el principio.

FileWriter(File fichero)	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar y borra el contenido previo del fichero comenzando a escribir desde el principio.
FileWriter(String nombreDeFich, boolean append)	Recibe como parámetro el nombre del fichero a abrir y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.
FileWriter(File fichero, boolean append)	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.

Métodos más usados de `FileWriter`:

Método	Explicación
void write(int c)	Escribe un carácter.
void write(char[] buf)	Escribe un array de caracteres.
void write(String str)	Escribe una cadena de caracteres.
append(char c)	Añade un carácter.

Métodos de `FileReader`:

Constructores:

Constructor	Explicación
FileReader(File fichero)	Lanza la excepción FileNotFoundException si el fichero del parámetro no existe.
FileReader(String fichero)	Lanza la excepción FileNotFoundException si el fichero del parámetro no existe.

Métodos más usados de `FileReader`:

Método	Explicación
int read()	Lee un carácter y lo devuelve.
int read(char[] buf)	Lee hasta buf.length caracteres de datos de la matriz buf pasada como parámetro.
int read(char[] buf, int desplazamiento, int n)	Lee hasta n caracteres de datos de una matriz buf comenzando por buf[desplazamiento] y devuelve el número leído de caracteres.

6 Acceso aleatorio: RandomAccessFile

Esta clase implementa las interfaces **`DataInput`** y **`DataOutput`**. Con lo cual, puede hacer uso de los métodos `read()` y `write()` para cada tipo de dato.

Constructores de RandomAccessFile:

Constructor	Explicación
RandomAccessFile(String nomFich, String oper)	**nomFich** es el nombre del fichero y **oper** es el argumento que determina si el contenido del fichero se va a poder solo leer (**r**) o leer y escribir (**rw**).
RandomAccessFile(File nomFich, String oper)	**nomFich** es el objeto fichero y **oper** es el argumento que determina si el contenido del fichero se va a poder solo leer (**r**) o leer y escribir (**rw**).

Métodos más usados de RandomAccessFile:

Método	Explicación
long getFilePointer()	Indica dónde está situado el puntero del fichero.
void seek(long pos)	Desplaza el puntero 'pos' bytes desde el inicio.
long length()	Devuelve la longitud del fichero.
int skipbytes(int d)	Desplaza el puntero del fichero –desde su posición actual- tantos bytes como indica 'd'.
readBoolean(), readByte(), readChar(), reading(), readDouble(), readFloat(), readUTF(), readLine ()	Leen un dato del tipo indicado.
writeBoolean, writeByte, writeBytes, writeChar, writeChars, writeInt, writeDouble, writeFloat, writeUTF, writeLine	Todos reciben como parámetro, el dato a escribir.

7 Serialización: Interface Serializable y clases ObjectInputStream y ObjectOutputStream

Java proporciona un mecanismo para poder escribir y leer directamente objetos en los flujos. Ese mecanismo recibe el nombre de **Serialización** y, para usarlo hay que tener en cuenta que:

- La **Serialización** consiste en la **descomposición automática,** por parte de la máquina virtual, **de un objeto en una secuencia (serie) de bytes** para poder escribirlos en un flujo asociado a un fichero o a cualquier otro tipo de canal (por ejemplo para enviar un objeto Java desde un servidor web hasta una máquina cliente). En aplicaciones distribuidas los objetos *serializables* nos permitirán mover estructuras de datos entre diferentes máquinas sin que el desarrollador tenga que preocuparse de la codificación y transmisión de los datos.
- Evidentemente también es posible hacer el proceso contrario de "deserialización", que lee del flujo asociado al fichero la cadena de bytes, y a partir de ellos reconstruye el objeto original en memoria.

- Para que los objetos de una clase puedan ser serializados, esa clase debe implementar el interface `Serializable`.
- El interface `Serializable` no requiere implementar ningún método. Es sólo una "etiqueta" que hay que poner a una clase para que el compilador sepa que debe proporcionar para ella el mecanismo de "descomposición en secuencias de bytes" de sus objetos.
- Al serializar un objeto, también se serializan todos los objetos que lo compongan, siempre y cuando también pertenezcan a clases serializables. Si al serializar un objeto la máquina virtual se encuentra que uno de sus campos es una referencia de un tipo serializable, seguirá esa referencia serializando también el objeto al que apunta.
- Por ejemplo, al serializar un objeto Persona, se serializa también el nombre de la persona, que es de tipo String, ya que la clase String también es serializable. Esto es una característica poderosa. Para grabar en fichero toda una lista enlazada de objetos Persona, por ejemplo, no sólo no tenemos que ir guardando de forma individual cada uno de los datos que componen a cada persona de la lista, es que ni siquiera tenemos que ir guardando las personas una a una. Basta con que escribamos en el flujo la referencia a la primera persona de la lista, y toda la lista entera será serializada y guardada correctamente en el fichero. Para leer los datos de esas personas del fichero, bastará con asignar el resultado de la lectura a la referencia que apunta a la primera persona de la lista para que tengamos reconstruida en memoria la lista completa. En realidad, para Java la referencia que apunta a la primera persona es como si no apuntara a la primera persona, si no a un único objeto "lista enlazada de personas" que es el que se serializa.
- Para serializar debemos usar flujos `ObjectOutputStream` y `ObjectInputStream`, sobre los que podremos escribir y leer objetos directamente. Realmente estos flujos modifican la forma de trabajar con los flujos sobre los que se montan, para que se puedan escribir objetos en ellos en vez de bytes.
- Como la serialización realmente transforma los objetos en cadenas de bytes, que son lo que se envía al fichero, los flujos `ObjectOutputStream` deben montarse sobre un flujo `FileOutputStream` y los flujos `ObjectInputStream` deben montarse sobre un flujo `FileInputStream`. **No es posible usar Serialización** con flujos de las jerarquías **`Writer`** y **`Reader`**.
- Los flujos `ObjectOutputStream` y `ObjectInputStream` proporcionan métodos `writeObject`() y `readObject`() para la ecritura y lectura de objetos de los flujos, pero además proporcionan toda una serie de métodos para trabajar con datos de tipo primitivo. Así tendremos un método write y un método read para cada tipo primitivo. Por ejemplo, `writeInt`() y `readInt`() permitirán escribir y leer valores int de los flujos.
- Al **leer los objetos del flujo** debemos **aplicar un casting explícito** a la clase del objeto que queremos leer. A fin de cuentas, lo que llega a la aplicación desde el flujo no es más que una serie de bytes, que pueden representar cualquier cosa. Al hacer el casting, realmente le estamos proporcionando a la máquina virtual la información que necesita sobre qué es lo que debe intentar ver en esa cadena de bytes, de forma que si debe buscar un objeto Persona, buscará un nombre (String), una edad (int) y un sexo (char) en esa cadena de bytes, y en ese orden. El casting le proporciona el nombre de la clase en la que "buscar los planos" para construir el objeto a partir de la secuencia de bytes que recibe.

- Ya que al **leer los objetos del flujo** es necesario usar la clase del objeto para el casting explícito, esta clase debe estar accesible, y si no lo está se producirá una `ClassNotFoundException`, que deberá ser capturada convenientemente mediante un bloque `try-catch`.
- Sólo son serializables los objetos. Si necesitamos guardar en el flujo como parte importante de la información alguna variable de tipo `static`, debe guardarse de forma independiente, aunque en el mismo flujo.
- La serialización resulta de utilidad en algunos aspectos avanzados de Java como RMI (invocación remota de métodos) y JavaBeans (creación de componentes reutilizables), así como para proporcionar una característica denominada "persistencia ligera" de objetos. Se puede profundizar mucho más sobre Serialización, pero nos hemos limitado a ver cómo almacenar y recuperar datos desde ficheros, cuando esos datos están constituidos por objetos de clases que se han definido como serializables.
- Muchas clases de la API de Java son *serializable*s, como por ejemplo las colecciones. Si tenemos una serie de elementos en una lista, podríamos serializar la lista completa, y de esa forma guardar todos nuestros objetos, con una única llamada a writeObject.
- Cuando una clase implemente la interfaz Serializable veremos que Eclipse nos da un *warning* si no añadimos un campo serialVersionUID. Este es un código numérico que se utiliza para asegurarnos de que al recuperar un objeto serializado éste se asocie a la misma clase con la que se creó. Así evitamos el problema que puede surgir al tener dos clases que puedan tener el mismo nombre, pero que no sean iguales (podría darse el caso de que una de ellas esté en una máquina cliente, y la otra en el servidor). Si no tuviésemos ningún código para identificarlas, se podría intentar recuperar un objeto en una clase incorrecta.
- Eclipse nos ofrece dos formas de generar este código pulsando sobre el icono del *warning*: con un valor por defecto, o con un valor generado automáticamente. Será recomendable utilizar esta segunda forma, que nos asegura que dos clases distintas tendrán códigos distintos.

Ver: `EjemploObjetos`