UD12.- Gestión de ficheros

Módulo: Programación
1º DAM

CONTENIDOS

- Introducción
- Clases asociadas a las operaciones de gestión de ficheros
- Flujos o *streams*. Tipos
- Formas de acceso a un archivo
- Operaciones sobre archivos
- Clases para la gestión de flujos

Introducción

- Un fichero o archivo es un **conjunto de bits** almacenado en un dispositivo de memoria de forma permanente.

- Los archivos tienen una serie de **propiedades**: nombre, extensión, tamaño, ruta donde están ubicados, etc.

- Los archivos están formados por **registros** y cada registro por **campos**.

La clase File

- Un objeto de la clase F*ile* representa un archivo o directorio del sistema de archivos.

Constructor	Descripción
File(String pathname)	Este constructor crea un objeto F*ile* al que se le debe pasar la ruta a un archivo o directorio.
File(String parent, String child)	En este constructor se debe pasar la ruta y el nombre del archivo o directorio.
File(File pariente, String child)	Como el objeto F*ile* puede representar un directorio o un archivo, el primer parámetro funcionará como directorio y el segundo parámetro será el nombre del archivo.

- La propiedad estática ***File.separator*** permite indicar el separador correcto de cada Sistema Operativo.
  - Ejemplo de rutas en Windows: “C:\\Windows\\System32\\”
  - Ejemplo de rutas en Unix: “/home/ggascon/”

Métodos de la clase File (I)

Método	Descripción
boolean isDirectory()	Sirve para saber si estamos trabajando con un archivo o con un directorio.
boolean isFile()	Complementario del anterior, sirve para saber si estamos trabajando con un archivo o con un directorio.
boolean exists()	Para asegurarnos de que el archivo existe realmente.
boolean delete()	Permite eliminar el archivo o directorio al que hace referencia el objeto F*ile*.
boolean renameTo(File dest)	Permite renombrar el archivo o directorio al que hace referencia el objeto F*ile*. Se le pasa como parámetro un objeto File con el nuevo nombre.
boolean createNewFile()	Crea el fichero vacío, si no existe y devuelve true. Si ya existe devuelve false. Puede generar IOException.

Métodos de la clase File (II)

Método	Descripción
boolean canRead()	Nos permite averiguar si tenemos permiso de lectura sobre el archivo.
boolean canWrite()	Nos permite averiguar si tenemos permiso de escritura sobre el fichero.
String getPath()	Devuelve la ruta relativa del archivo.
String getAbsolutePath()	Devuelve la ruta absoluta del archivo (incluye el nombre del archivo).
String getName()	Devuelve el nombre del archivo.

Métodos de la clase File (III)

Método	Descripción
String getParent()	Devuelve el directorio padre (del que cuelga el archivo o directorio).
long length()	Devuelve el tamaño del archivo en bytes.
boolean mkdir()	Crea un directorio.
String[] list()	Devuelve un array de objetos S*tring* con los nombres de los archivos / directorios que contiene el directorio.
String[] list(FilenameFilter filter)	Devuelve un array de objetos S*tring* con los nombres de los archivos / directorios que contiene el directorio que cumplen el filtro indicado (interfaz).
File[] listfiles()	Devuelve un array de objetos F*ile* con los nombres de los archivos / directorios que contiene el directorio.

La clase Files (I)

- La clase Files existe desde la versión 7 del JDK y permite realizar operaciones comunes con archivos de forma sencilla a través de sus métodos estáticos:
  - Crear

    Files.createFile(), Files.createDirectory(), Files.createDirectories(), Files.createLink(), Files.createSymbolicLink(), Files.createTempFile() Files.createTempDirectory().
  - Copiar

    Files.copy()
  - Borrar

    Files.delete(), Files.deleteIfExists()
  - Saber si existe

    Files.exists()
  - Buscar

    Files.find()

La clase Files (II)

- Existen otros muchos métodos en esta clase que puede ser muy útiles, però nosotros no los vamos a utilizar en este tema.

- El motivo es que gran parte del código Java existente está hecho con la API común de Java que es estándar y funciona en todas las versiones. Es por ello, que nos interesa practicar la parte estándar.

- Por supuesto, cuando tengáis que hacer nuevos proyectos podéis utilizar la clase Files sin ningún problema, pero para realizar los ejercicios intentaremos evitarla.

Formas de acceso a un archivo

- Acceso **secuencial**:
  - Si se quiere acceder a un dato, se debe acceder previamente a las anteriores.
  - Las inserciones se hacen al final, no es posible escribir entre los datos.

- Acceso **aleatorio**:
  - Se puede acceder a la información en cualquier orden.
  - Las lecturas y escrituras se pueden hacer donde sean necesarias.

Operaciones sobre ficheros (I)

- **Creación**
  - El fichero se crea en disco con un nombre.
  - Se hace la operación una única vez.
- **Apertura**
  - Para poder trabajar con el contenido del archivo se debe abrir previamente.
- **Cierre**
  - El archivo debe cerrarse cuando acabamos de trabajar con él.

Operaciones sobre ficheros (II)

- **Lectura** de datos
  - Transferir información de archivo en la memoria principal, a través de variables.

- **Escritura** de datos
  - Transferir información de la memoria en el archivo, a través de variables.

Operaciones sobre ficheros (III)

- **Altas**
  - Añadir un nuevo registro en el archivo.
- **Bajas**
  - Eliminar del archivo algún registro
    - Eliminación lógica: cambiar el valor de algún campo del registro
    - Eliminación física: reescribir el archivo en otro sin incluir el registro y renombrar el original.

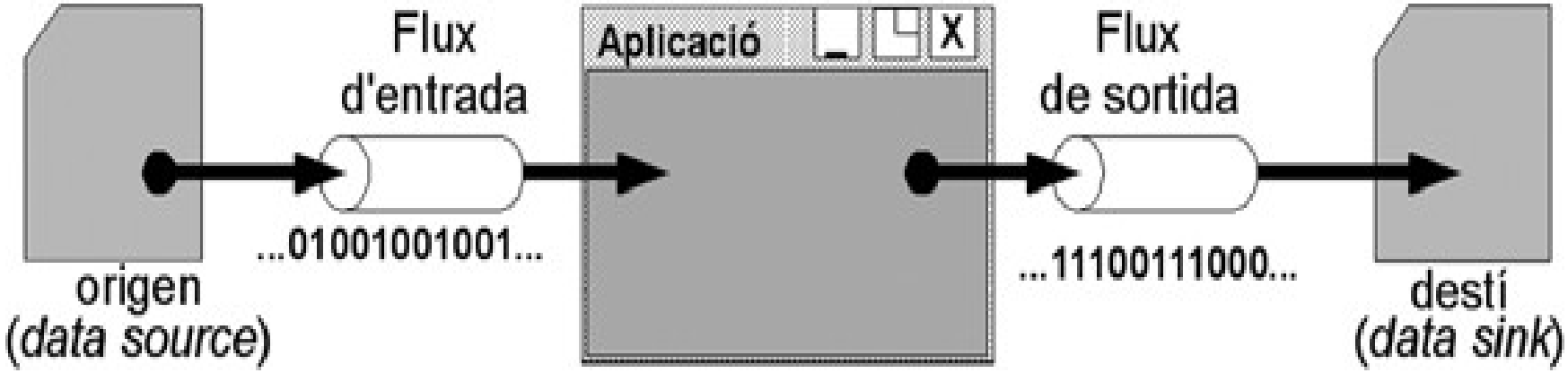
Operaciones sobre ficheros (IV)

- **Modificaciones**
  - Cambiar el contenido de algún registro.
  - Implica la búsqueda del registro.

- **Consultas**
  - Buscar un registro determinado.

Flujos o *streams*

- Uso: para tratar la comunicación de información entre origen y destino.



Flujos o *streams*. Tipos

- Flujos de **caracteres** (16 bits)
  - Lectura y escritura de caracteres (**texto**)
  - Clases: ***Reader*** y ***Writer***

- Flujos de **bytes** (8 bits)
  - Lectura y escritura de datos **binarios**.
  - Clases: ***InputStream*** y ***OutputStream***

Archivos de texto (I)

- Guardan caracteres alfanuméricos en un formato estándar (ASCII, UNICODE, UTF8, …)
- Clase ***FileReader***: Para leer caracteres
  - *FileNotFoundException* (No existe el nombre del archivo o no es válido)
- Clase ***FileWriter***: Para escribir caracteres
  - *IOException* (El disco está lleno o protegido contra escritura)

Constructores de la clase *FileReader*

Constructor	Descripción
FileReader(File file)	Crea un objeto *FileReader* a partir del objeto File que apunta al archivo desde el que se quiere leer.
FileReader(String fileName)	Crea un objeto *FileReader* a partir del nombre del archivo desde el que se quiere leer. Se le pasa como parámetro la ruta donde se encuentra el archivo.

* Pueden lanzar la excepción *FileNotFoundException*.

Métodos de *FileReader*

Método	Descripción
int read()	Lee un carácter y lo devuelve como un entero. Devuelve -1 cuando llega al final del archivo.
int read(char[] cbuf)	Lee 'n' caracteres (n <= cbuf.length) Devuelve el número de caracteres leídos. Los caracteres leídos están en las posiciones [0..n-1] del array cbuf. Devuelve -1 si el archivo se ha acabado.
int read(char[] cbuf, int off, int len)	Lee 'n' caracteres. Devuelve el número de caracteres leídos. Los caracteres leídos están en las posiciones [off ... off + len] del array cbuf. Devuelve -1 si el archivo se ha acabado.
void close()	Cierra el archivo.

* Todos los métodos anteriores pueden lanzar la excepción *IOException*.

Constructores de la clase *FileWriter*

Constructor	Descripción
FileWriter(File file)	Dado un objeto F*ile* que será el archivo sobre el que se quiere escribir, construye un objeto F*ileWriter* sobre él.
FileWriter(File file, boolean append)	Dado un objeto F*ile* que será el archivo sobre el que se quiere escribir, construye un objeto *FileWriter* sobre él al que se pueden **añadir datos** (No sobrescribe).
FileWriter(String fileName)	Dada la ruta del archivo sobre el que se quiere escribir, construye un objeto *FileWriter* sobre él.
FileWriter(String fileName, boolean append)	Dada la ruta del archivo sobre el que se quiere escribir, construye un objeto *FileWriter* sobre él al que se pueden **añadir datos** (No sobrescribe).

Métodos de *FileWriter*

método	Descripción
void write(int c)	Escribe un carácter
Writer append(char c)	Añadir un carácter a un fichero
void write(char [] cbuf)	Escribe en el fichero del array de caracteres.
void write(char [] cbuf, int off, int len)	Escribe en el fichero los 'n' caracteres del array cbuf a partir de la posición off.
void write(String str)	Escribe en el archivo la cadena 'str'.
void write(String str, int off, int len)	Escribe los caracteres de la cadena 'str' a partir de la posición off.
void flush()	Asegura que todos los caracteres queden bien escritos en disco, sin cerrar el archivo.
void close()	Cierra el fichero, asegurando que todo queda bien escrito en disco.

* Todos los métodos anteriores pueden lanzar la excepción *IOException*.

Archivos de texto (II)

- Pasos para trabajar con archivos de texto:
  1. Invocar a la clase File (este paso es opcional).
  2. Crear el flujo de entrada con FileReader o flujo de salida con FileWriter (puede implicar la creación del archivo)
  3. Operaciones de lectura / escritura
  4. Cerrar los flujos de datos

Ejemplo (I)

```
// Crear el archivo
    File f1 = new File ( "/home/ggascon/fitxer1.txt");
// Crear los flujos de datos
    FileReader lector = new FileReader(f1); // entrada
    FileWriter escritor = new FileWriter(f1); // salida
// Ejemplos de operaciones de lectura / escritura
    int i = lector.read();
    escritor.write("Cadena a escribir");
// Cerramos los flujos de datos
    lector.close();
    escritor.close();
```

Ejemplo (II)

```
// Crear los flujos de datos
    FileReader lector = new FileReader("/home/ggascon/fitxer1.txt"); // entrada
    FileWriter escritor = new FileWriter("/home/ggascon/fitxer1.txt"); // salida

// Ejemplos de operaciones de lectura / escritura
    int i = lector.read();
    escritor.write( "Cadena a escribir");

// Cerramos los flujos de datos
    lector.close();
    escritor.close();
```

Clase BufferedReader

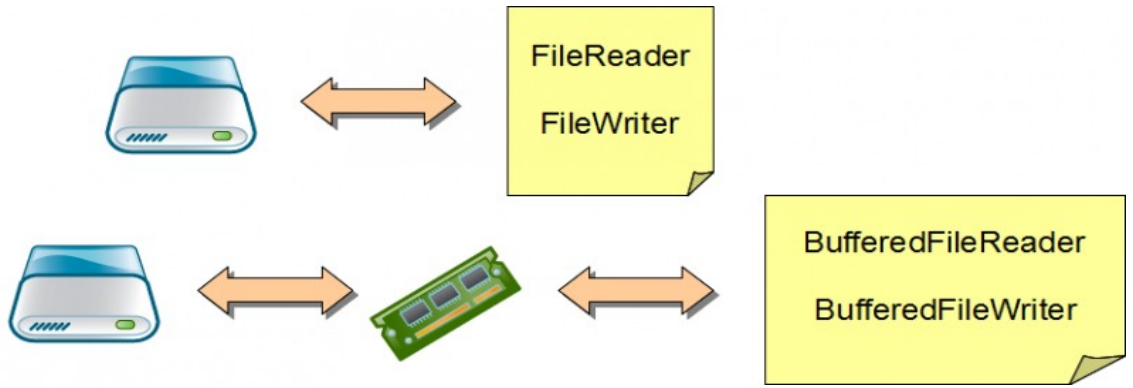
- Las clases FileReader, FileWriter y FileInputStream y FileOutputStream (que veremos para el manejo de archivos binarios) realizan las operaciones de lectura y escritura directamente sobre el dispositivo de almacenamiento.
  - Si la aplicación hace estas operaciones de forma muy repetida, su ejecución será más lenta.
- Para mejorar esta situación, se pueden utilizar junto con las clases anteriores la clase **Buffered**

Clase B*uffered*

- La palabra B*uffered* hace referencia a la capacidad de almacenamiento temporal en la lectura y escritura:
  - En las operaciones de lectura se guardan más datos de los que realmente se necesitan en un momento determinado, de forma que en una operación posterior de lectura, es posible que los datos ya estén en memoria y no sea necesario acceder de nuevo al dispositivo.
  - En las operaciones de escritura, los datos se guardan en memoria y no se vuelcan a disco hasta que no haya una cantidad suficiente de datos para mejorar el rendimiento. Cuidado esto puede ser peligroso sino se gestionan bien las excepciones.

Flujos de caracteres (*Character Streams*)

- Clase Reader
  - FileReader
  - BufferedReader *
- Clase Writer
  - FileWriter
  - BufferedWriter *



* Para evitar que cada lectura o escritura acceda directamente al archivo, se puede utilizar un *buffer* intermedio entre el disco y el *stream*.

Clase BufferedReader

- La clase **BufferedReader** dispone de métodos para leer líneas completas.

Constructor	Descripción
BufferedReader(Reader in) * Necesitamos un FileReader	Crea un buffer para almacenar los caracteres del flujo de entrada (a través del cual se lee la información del archivo).
métodos	**Descripción**
String readLine()	Lee una línea de texto.
int read()	Lee un solo carácter.
void close()	Cierra el *buffer*.

* Los métodos anteriores pueden lanzar la excepción IOException.

Clase BufferedWriter

- La clase **BufferedWriter** dispone de métodos para escribir líneas completas.

Constructor	Descripción
BufferedWriter(Writer out) * Necesitamos un FileWriter	Crea un buffer para almacenar los caracteres del flujo de salida (a través del cual se escribirá en el archivo).
métodos	**Descripción**
void newLine()	Escribe una línea en blanco.
void write(int c)	Escribe un carácter.
void write(String s)	Escribe una cadena.
void close()	Cierra el *buffer*.

* Los métodos anteriores pueden lanzar la excepción IOException.

Ejemplo

```
// Crear el archivo
        File f1 = new File("/home/ggascon/fitxer1.txt");
// Crear los flujos de datos
        FileReader lector = new FileReader(f1); // entrada
// Crear el buffer
        BufferedReader dadesLector = new BufferedReader(lector);
// Operaciones lectura
        String linea = dadesLector.readLine();
// Cerramos el buffer
        dadesLector.close();
```

Constructores de la clase PrintWriter

Constructor	Descripción
PrintWrite(File file)	Crea un objeto de tipo *PrintWrite* a partir de un fichero.
PrintWrite(Writer out)	Crea un objeto de tipo *PrintWrite* a partir de un *FileWriter*.
PrintWrite(String fileName)	Crea un objeto de tipo *PrintWrite* a partir de la ruta del archivo.

Métodos de la clase PrintWriter

Métodos	Descripción
void print(String s)	Escribe una cadena.
void println()	Escribe un salto de línea.
void print(int x)	Escribe un entero.
void println(char x)	Escribe un carácter y un salto de línea.

* Los métodos anteriores pueden lanzar la excepción IOException.

* print () y println() son similares a los de System.out

Ficheros binarios (I)

- Almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario como ocurre con los archivos de texto.

- Clases:
  - **FileInputStream:** para lectura de datos.
  - **FileOutputStream:** para escritura de datos.

Constructores de las clases FileInputStream

Constructor	Descripción
FileInputStream(File file)	Crea un objeto de tipo *FileInputStream* para leer datos desde el objeto F*ile* creado previamente.
FileInputStream(String fileName)	Crea un objeto de tipo *FileInputStream* para leer datos. Recibe como parámetro la ruta donde se encuentra el fichero con los datos.

Constructores de las clases FileOutputStream

Constructor	Descripción
FileOutputStream(File file)	Crea un objeto de tipo *FileOutputStream* para escribir en el fichero al que hace referencia el objeto File.
FileOutputStream(File file, boolean append)	Crea un objeto de tipo *FileOutputStream* para añadir datos (no sobreescribir) el archivo al que hace referencia el objeto File.
FileOutputStream(String fileName)	Crea un objeto de tipo *FileOutputStream* para escribir datos. Recibe como parámetro la ruta donde se encuentra el archivo.
FileOutputStream(String fileName, boolean append)	Crea un objeto de tipo *FileOutputStream* para añadir datos. Recibe como parámetro la ruta donde se encuentra el archivo.

Métodos de FileInputStream

Método	Descripción
int read()	Lee un byte y lo devuelve pasado a entero. Devuelve -1 cuando llega al final del archivo
int read(byte[] b)	Lee 'n' bytes (n <= b.length) Devuelve el número de bytes leídos. Los bytes leídos están en las posiciones [0..n-1] del array b. Devuelve -1 si el archivo se ha acabado.
int read(byte[] b, int off, int len)	Lee 'instalan' bytes. Devuelve el número de bytes leídos. Los bytes leídos están en las posiciones [off ... off + len] del array b. Devuelve -1 si el archivo se ha acabado.
void close()	Cierra el archivo.
int available()	Una estimación del número de bytes que quedan por leer.

* Todos los métodos anteriores pueden lanzar la excepción IOException.

Métodos de FileOutputStream

Método	Descripción
void write(int b)	Escribe un byte.
void write(byte[] b)	Escribe en el fichero del array de bytes.
void write(byte[] b, int off, int len)	Escribe los 'instalan' bytes del array b en el fichero, a partir de la posición off.
void close()	Cierra el fichero, asegurando que todo está bien escrito en disco.

* Todos los métodos anteriores pueden lanzar la excepción IOException.

Ficheros binarios (II)

- Pasos para trabajar con ficheros binarios:
  - Invocar a la clase File (opcional)
  - Crear el flujo de entrada con FileInputStream o flujo de salida con FileOutputStream (implica la creación del archivo)
  - Operaciones de lectura / escritura
  - Cerrar el archivo

DataInputStream y DataOutputStream

- Para guardar y recuperar datos primitivos.
- Descienden de las clases FilterInputStream y FilterOutputStream.
  - El objetivo principal de las clases filtro en Java es la modificación / transformación de los datos.
- Necesitan un FileInputStream o un FileOutputStream.

Métodos de DataInputStream y DataOutputStream

DataInputStream	DataOutputStream
boolean readBoolean()	void writeBoolean()
byte readByte()	void writeByte()
char readChar()	void writeChar()
short readShort()	void writeShort()
int read()	void write()
long readLong()	void writeLong()
float readFloat()	void writeFloat()
double readDouble()	void writeDouble()
String readUTF()	void writeUTF()
	void writeBytes() Escribe una cadena como una secuencia de bytes
	void writeChars() Escribe una cadena como una secuencia de caracteres

Ficheros de acceso aleatorio (I)

- La clase **RandomAccessFile** permite el acceso directo cualquier posición de un archivo binario.

- Permite abrir los ficheros:
  - En modo lectura ("r")
  - En modo escritura ("w")
  - En modo lectura / escritura ("rw")

Constructores de la clase RandomAccessFile

Constructor	Descripción
RandomAccessFile(File file, String modo)	Se crea el objeto a partir de un objeto File. El modo puede ser "r", "w" o "rw".
RandomAccessFile(String name, String modo)	Se crea el objeto pasándole como parámetro la ruta del archivo. El modo puede ser "r", "w" o "rw".

Ficheros de acceso aleatorio (II)

- El lugar del archivo al que se accede para leer o escribir está marcado con un puntero:
  - El puntero señala el carácter a partir del cual se hará la operación de lectura o escritura.
  - Cada vez que se hace una lectura o escritura, el puntero se posiciona automáticamente en el siguiente carácter.

Métodos de la clase RandomAccessFile (I)

Método	Descripción
long getFilePointer()	Devuelve la posición actual del puntero.
void seek(long pos)	Coloca el puntero en la posición 'pos'. El primer carácter se encuentra en la posición 0.
long length()	Devuelve la longitud del archivo en bytes.
int skipBytes(int n)	Desplaza el puntero desde la posición actual el número de bytes indicado en el parámetro (n bytes).

Métodos de la clase RandomAccessFile (II)

Método	Descripción
int read()	Devuelve el byte leído en la posición marcada por el puntero.
final String readLine()	Devuelve la cadena de caracteres leída desde la posición actual del puntero hasta el primer salto de línea que encuentre.
public void write(int b)	Escribe en el fichero el byte pasado como parámetro.
public final void writeBytes(String s)	Escribe en el archivo la cadena pasada como parámetro. No incluye salto de línea, por lo tanto si se quiere añadir, se deberá incluir en la cadena de caracteres: writeBytes(cadena + "\ n").

Métodos de la clase RandomAccessFile (III)

- Hay un método para cada tipo de datos básico: readChar, readInt, readDouble, readBoolean, ...

- También existe un método write para cada tipo de datos básico: writeChar, writeInt, writeDouble, writeBoolean, ...

Los objetos

- Los objetos tienen:
  - comportamiento (métodos) → "viven" en la clase
  - estado (atributos) → "viven" en el objeto

- Cuando nos puede interesar guardar el estado de un objeto? Por ejemplo ...
  - Guardar el estado de una partida (en un juego).
  - Guardar el estado de un archivo antes de hacer una actualización.

Guardar el estado de un objeto

- Con lo que hemos visto hasta ahora, escribiríamos el valor de cada instancia "a mano".

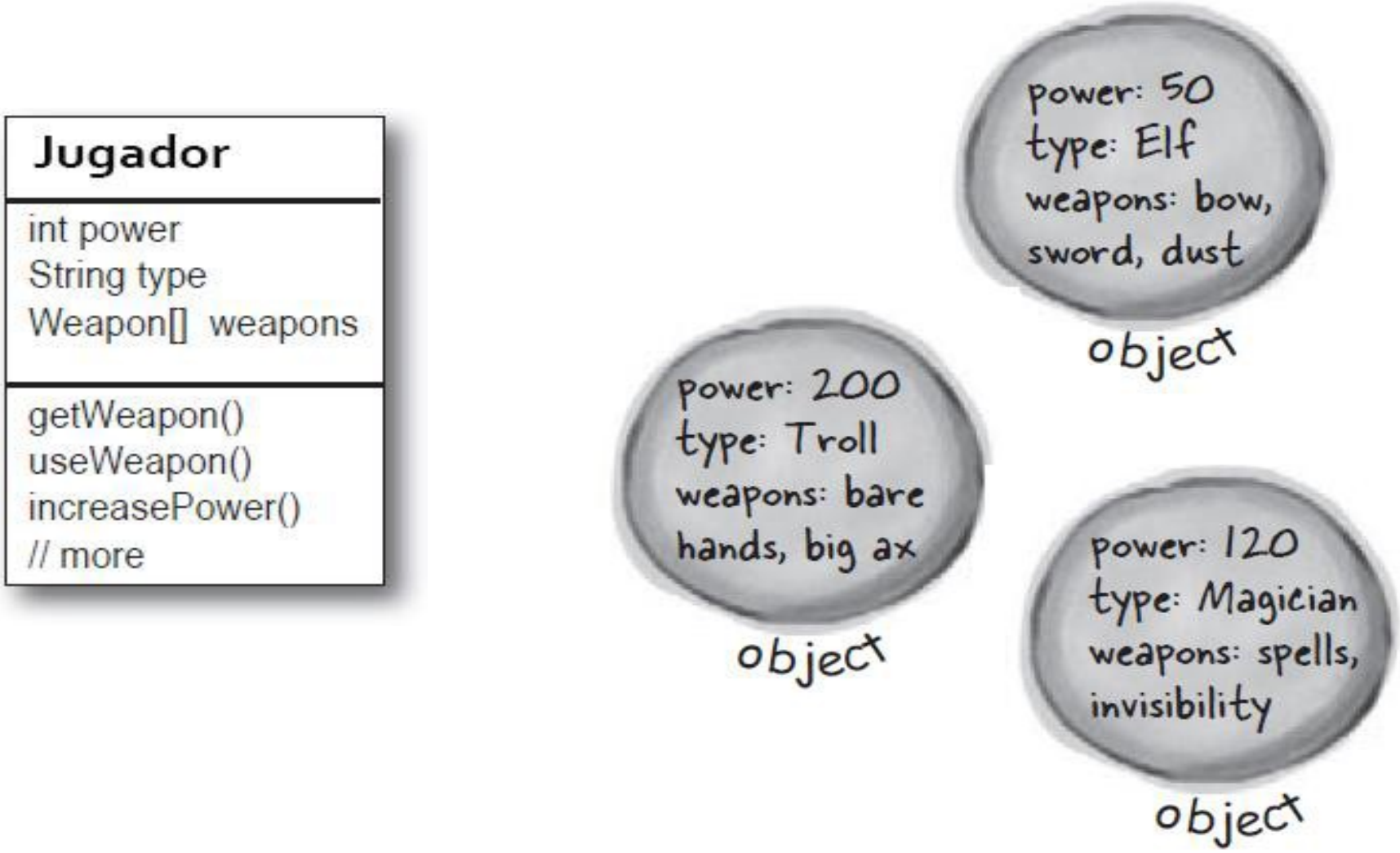
- Pero podemos hacerlo de una forma OO "fácil".

- Tendremos dos operaciones:
  - Serializar
  - Deserializar

Guardar el estado de un objeto

- Si el objeto será utilizado por el programa Java que lo ha creado:
  - Podemos utilizar serialización Java
  - Podemos utilizar algún formato estándar de transporte de datos JSON o XML
- Si los datos serán utilizados por otros programas
  - Utilizaremos el formato estándar de transporte de datos JSON o XML
- Los datos en formato JSON o XML pueden ser parseados de forma sencilla en el destino.

Ejemplo: Juego de aventura



Ejemplo: Juego de aventura

- Como ya hemos dicho, tendremos dos opciones:

① Opción 1

Guardar los 3 objetos de los personajes serializados en un fichero

Se crea el fichero y se guarda el objeto serializado. El fichero si lo leemos como texto, no tendrá sentido:

```
"ÌsrGameCharacter
"%gê8MÛIpowerLjava/lang/
String;[weaponst[Ljava/lang/
String;xp8tlfur[Ljava.lang.String;≠“VÁ
È{Gxptbowtswordtdustsq~»tTrolluq~tb
are handstbig axsq~xtMagicianuq~tspe
llstinvisibility
```

② Opción 2

Escribir un fichero JSON

```
{jugadores: [
  {
     power: 50,
     type: Elf,
     weapons: [bow, sword, dust]
  },
  {
     power: 200,
     type: Troll,
     weapons: [bare, hands, big ax]
  }, .....
```

Serializar objetos

Si el fichero no existe, lo creará automáticamente

1 Crear un **FileOutputStream**

```
FileOutputStream fileStream = new FileOutputStream("MyGame.ser");
```

Crea un objeto FileOutputStream, que sabe cómo crear y conectarse a un fichero

2 Crear un **ObjectOutputStream**

```
ObjectOutputStream os = new ObjectOutputStream(fileStream);
```

La clase ObjectOutputStream puede escribir en un fichero, pero no sabe conectarse directamente a un fichero

3 Escribe el objeto

```
os.writeObject(characterOne);
os.writeObject(characterTwo);
os.writeObject(characterThree);
```

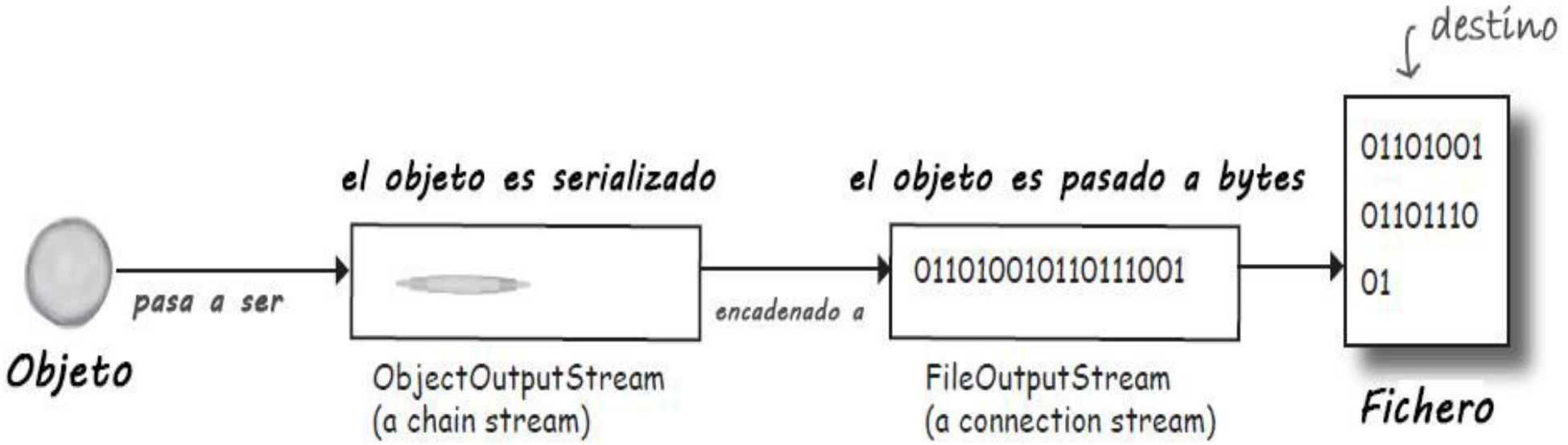
serializa los objetos serializados por las variables characterOne, characterTwo y characterThree, y lo graba en el fichero "MyGame·ser"

4 Cierra el **ObjectOutputStream**

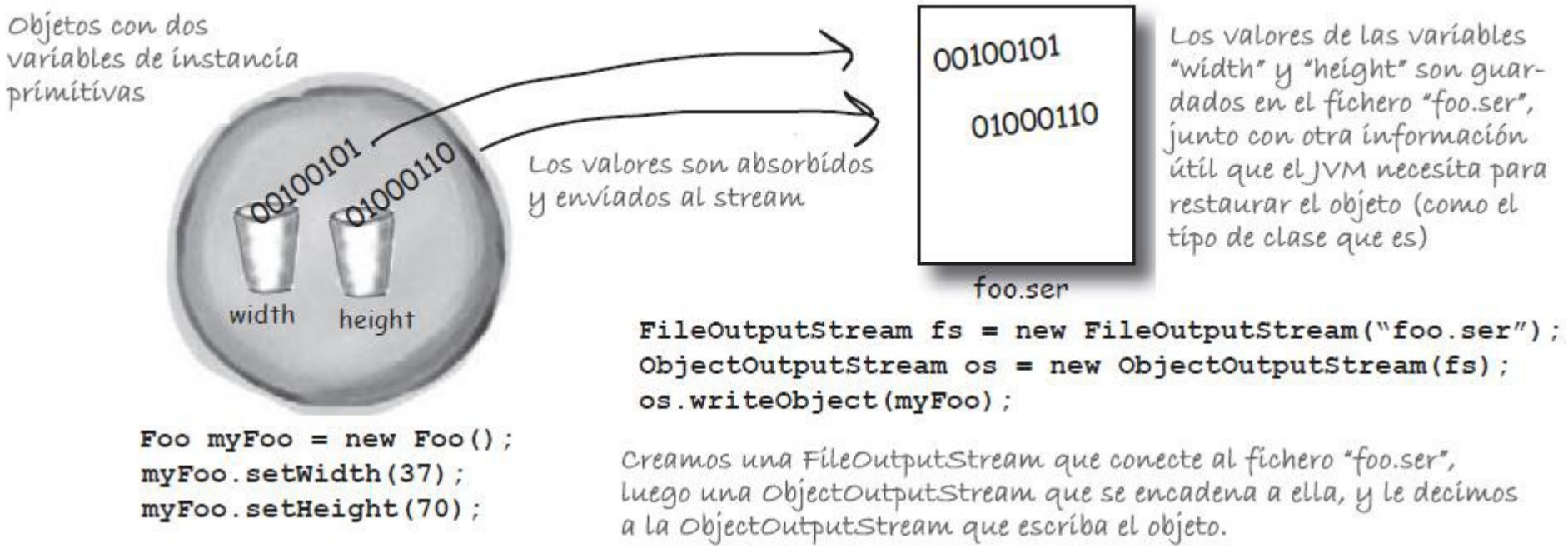
```
os.close();
```

Cierra el stream y el FileOutputStream cerrará automáticamente el fichero

FileOutputStream y ObjectOutputStream



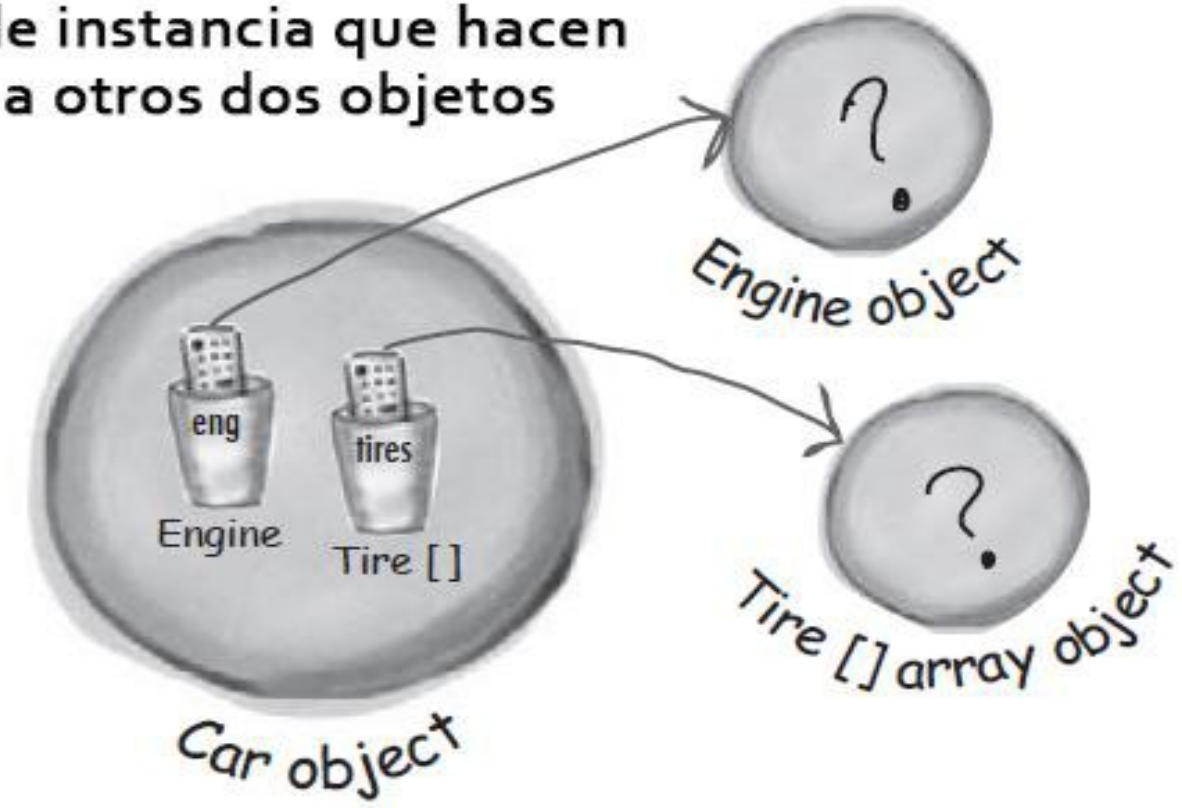
Serializar objetos



El estado del objeto son los valores de las variables de instancia. Pueden ser también variables de referencia a otro objeto.

¿Qué pasaría en este caso?

El objeto de tipo Car tiene dos variables de instancia que hacen referencia a otros dos objetos



¿Qué pasa cuando salvamos el objeto de tipo Car?

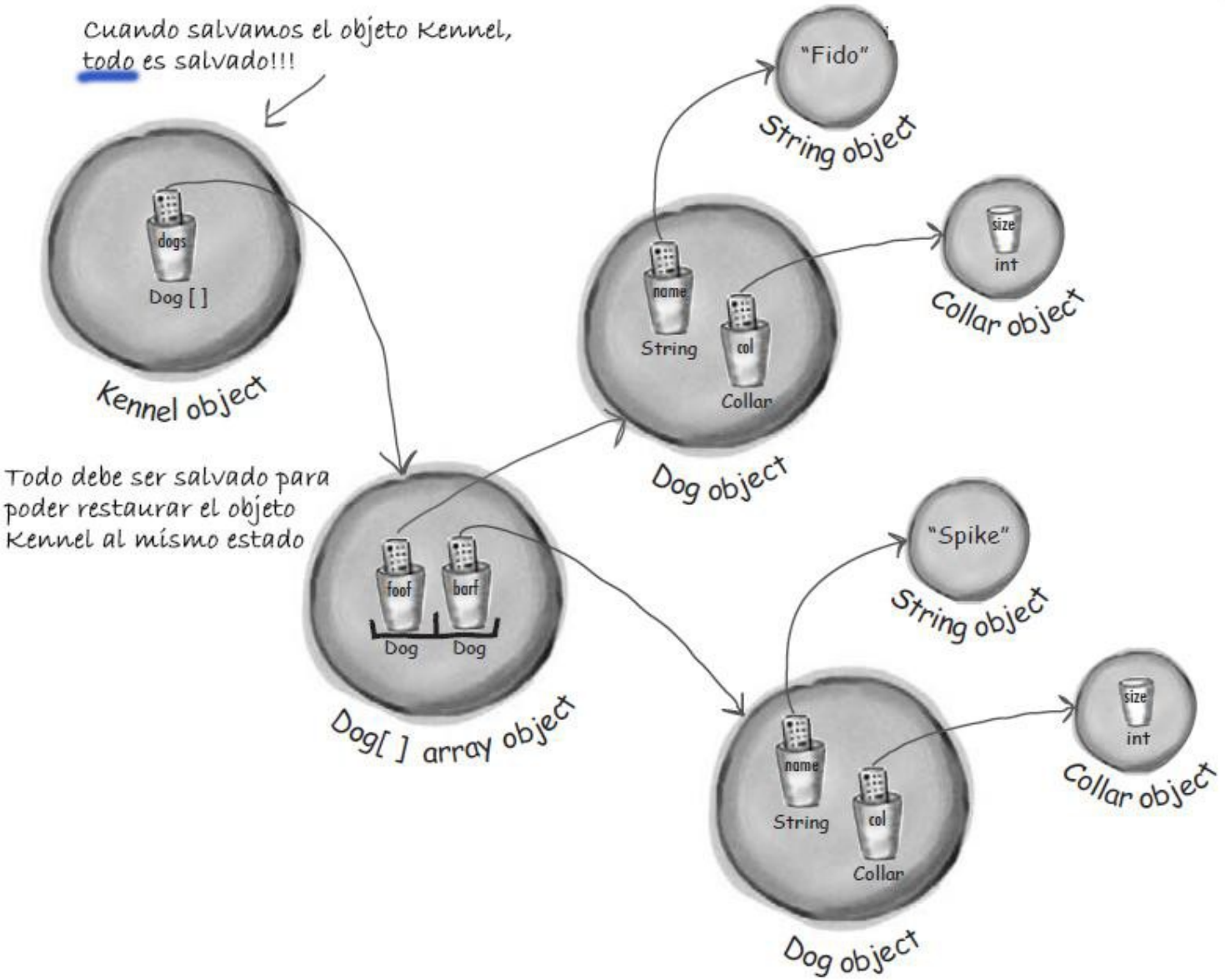
Reacción en cadena ...

- Cuando un objeto es serializado, todos los objetos a los que apuntan sus variables de instancia (también si son variables de referencia) también son serializados.

- Se produce una reacción en cadena **automáticamente**!!

- La serialización guarda el esquema de objetos entero.

Reacción en cadena ...



Interfaz *Serializable*

- Cuando un objeto se quiere serializar, deberá implementar la interfaz *Serializable*.
- Es una interfaz sin métodos a sobrescribir.
- Sus subclases también serán serializables, aunque no se especifique.

Interfaz *Serializable*

```
objectOutputStream.writeObject(myBox);
```

Todo lo que entra aquí, DEBERÁ implementar la interfaz Serializable o dará error en tiempo de ejecución

Necesitamos importar el paquete java.io.*

```
import java.io.*;

public class Box implements Serializable {
```

No tiene métodos para implementar. Cuando implementamos la interfaz, el JVM ya sabe que: "Es correcto serializar objetos de este tipo".

```
    private int width;
    private int height;
```

Estos dos valores serán guardados

```
    public void setWidth(int w) {
       width = w;
    }

    public void setHeight(int h) {
       height = h;
    }

public static void main (String[] args) {

      Box myBox = new Box();
      myBox.setWidth(50);
      myBox.setHeight(20);
      try {
         FileOutputStream fs = new FileOutputStream("foo.ser");
         ObjectOutputStream os = new ObjectOutputStream(fs);
         os.writeObject(myBox);
         os.close();
      } catch(Exception ex) {
          ex.printStackTrace();
      }
   }
}
```

Las operaciones de E/S pueden lanzar excepciones

Conecta a un fichero llamado "foo.ser" si existe. Y sino, crea un nuevo fichero con ese nombre.

Crea un ObjectOutputStream encadenado a una chain connection. Le decimos que escriba el objeto.

Serialización: todo o nada

- ¿Qué pasaría si no se guarda correctamente el estado de los objetos? Sería un desastre ...
- O el esquema entero es serializado completamente o la serialización falla.
- Si un objeto al que apunta otro objeto a serializar no es serializable, fallará.
- Obtendríamos una: ***java.io.NotSerializableException***

Cuando falla la serialización ...

```
import java.io.*;

public class Pond implements Serializable {

    private Duck duck = new Duck();

    public static void main (String[] args) {
       Pond myPond = new Pond();
       try {
         FileOutputStream fs = new FileOutputStream("Pond.ser");
         ObjectOutputStream os = new ObjectOutputStream(fs);

         os.writeObject(myPond);
         os.close();

       } catch(Exception ex) {
           ex.printStackTrace();
       }
    }
}
```

Los objetos de tipo Pond pueden ser serializados

la clase Pond tiene una variable de instancia de tipo Duck

Cuando serializamos myPond (un objeto de tipo Pond) su variable de instancia de tipo Duck automáticamente es serializada

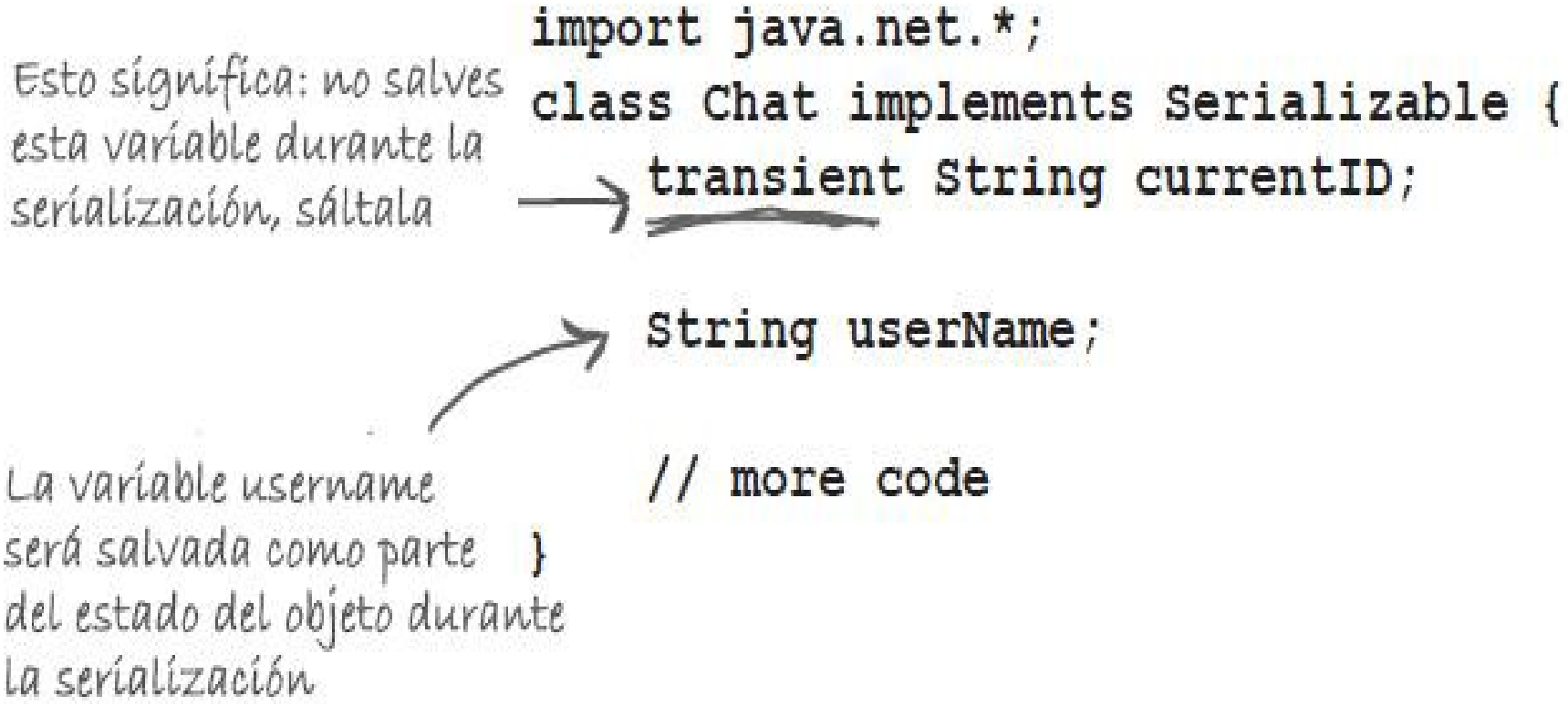
```
public class Duck {
   // duck code here
}
```

¡Ay, La clase Duck no es serializable! Esta clase no implementa la interfaz Serializable, por eso cuando se intenta serializar un objeto tipo Pond falla porque la variable de instancia de tipo Duck del objeto del tipo Pond no puede ser guardada.

```
File Edit Window Help Regret
% java Pond
java.io.NotSerializableException: Duck
        at Pond.main(Pond.java:13)
```

Variables *transient*

- Aquellas que no queremos guardar (por irrelevante o por demasiado relevante).



Deserialización de objetos

- Al serializar un objeto, podemos restaurar el estado original más tarde, en una ejecución posterior (persistencia).
- Es como la serialización, pero al revés.

Deserialización de objetos

1 Crear un **FileInputStream**

Si el fichero MyGame.ser no existe, dará una excepción

```
FileInputStream fileStream = new FileInputStream("MyGame.ser");
```

Creamos un objeto FileInputStream. Este objeto sabe cómo conectar con el fichero existente.

2 Crear un **ObjectInputStream**

```
ObjectInputStream os = new ObjectInputStream(fileStream);
```

ObjectInputStream permite leer objetos, pero no puede conectar directamente con el fichero. Necesita estar conectado a una connection stream (FileInputStream)

3 Leer los objetos

```
Object one = os.readObject();
Object two = os.readObject();
Object three = os.readObject();
```

Cada vez que llamamos a readObject(), obtenemos el siguiente objeto del stream. Los leeremos en el mismo orden en el que los hemos escrito. Obtendremos una gran excepción si intentamos leer más objetos de los que hemos escrito.

4 Hacer casting a los objetos

```
GameCharacter elf = (GameCharacter) one;
GameCharacter troll = (GameCharacter) two;
GameCharacter magician = (GameCharacter) three;
```

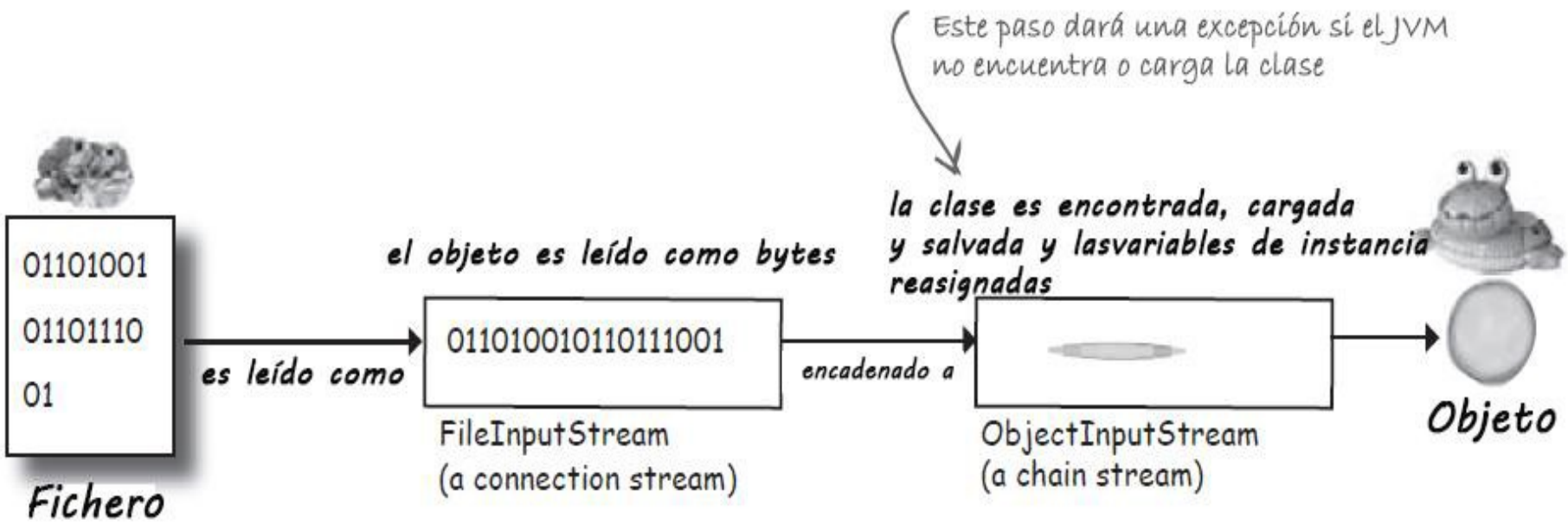
El valor devuelto por readObject() es de tipo Object y debemos hacer un casting para volver al tipo original.

5 Cerrar el **ObjectInputStream**

```
os.close();
```

Cerramos el stream y el FileOutputStream cerrará automáticamente el fichero

FileInputStream y ObjectInputStream



Deserialización de objetos

- Las variables de instancia del objeto contendrán los valores del objeto serializado.

- Las variables *transient* tendrán valores por defecto para variables primitivas y valores *null* para las referencias a objetos.

- Las variables *static* no se guardan (no forman parte del objeto).

Resumen de cosas importantes

- Podemos guardar el estado de un objeto serializandolo.
- Para serializar un objeto necesitamos un ObjectOutputStream, del paquete java.io.*.
- Para serializar un objeto en un fichero se necesita un FileOutputStream y encadenar a un ObjectOutputStream.
- Para serializar un objeto llamamos al método **writeObject(o)** mediante el ObjectOutputStream. No necesitamos llamar a más métodos.
- Aquellas clases que queramos que sean serializadas deben implementar la interfaz ***Serializable.***
- Si una superclase es ***Serializable***, La subclase automáticamente también lo es, aunque no lo especificamos explícitamente.
- Cuando un objeto es serializado, el grafo entero (el esquema de objetos) es serializado.

Resumen de cosas importantes

- Si algún objeto del grafo no es serializable, se producirá una excepción a menos que la variable sea *transient*.
- Marcamos una variable como *transient* si no es necesario serializarla. Las variables restauradas tendrán valores por defecto.
- En la deserialización leemos objetos con el método **readObject(o)**, en el mismo orden en el que se guardaron inicialmente.
- El tipo de retorno del método readObject(o) es de tipo Object, por lo tanto se deberá hacer un *casting* a las variables para que tengan su tipo real.
- Las variables *static* no se serializan. No tiene sentido guardar una variable que no forma parte del estado de cada objeto.