# UD07.2. Ficheros Binarios

# Apuntes

DAM1-Programación 2024-25 (Paraninfo Capítulo 11)



**Otras fuentes:**

- **Programación Java: Teoría**
  - La Clase File
  - Ficheros de Texto
  - Ficheros Binarios
  - Ficheros de Acceso Aleatorio
  - Persistencia y Serialización de objetos
  - Seleccionar archivos de forma gráfica con JFileChooser
- **Píldoras Informáticas: Curso de Java desde 0**
  - (Vídeos 152 a 160)
- **w3schools Java Tutorial**
  - Java Files

# Introducción

En la unidad anterior vimos que hay dos tipos de flujos de datos en Java, los binarios (también llamados *de bytes*) y los de texto. Allí nos dedicamos a estudiar con detalle los de texto. Ahora nos ocuparemos de los flujos de datos de tipo `byte`, que nos van a permitir guardar (o transferir) y recuperar (o recibir) cualquier tipo de datos usados en un programa. No olvidemos que, para usar cualquier flujo en Java, debemos importar las clases del paquete `java.io`.

Cuando se trata de escribir (o leer) bytes en un flujo, existen dos clases básicas, `FileOutputStream` y `FileInputStream`. Pero nosotros no solemos manejar bytes individuales en nuestros programas, sino datos (eso sí, formados por bytes) más complejos, ya sean de tipos primitivos u objetos. Por eso necesitamos un intermediario capaz de convertir los datos complejos en series planas de bytes o reconstruir los datos a partir de series de bytes, en procesos de **serialización** y de **deserialización** de datos, respectivamente. Esos intermediarios son flujos llamados *envoltorio:* `ObjectOutputStream` y `ObjectInputStream`, que se crean a partir de flujos de bytes planos, como `FileOutputStream` y `FileInputStream`.

# 1. Flujos de salida binarios

**Clases del paquete java.io**

- File

- FileInputStream
- FileOutputStream
- ObjectInputStream
- ObjectOutputStream

Supongamos que queremos grabar en disco los enteros guardados en una tabla. Para ello empezaremos creando un flujo de salida de tipo binario, asociado al fichero donde vamos a grabarlos, que llamaremos *enteros.dat*.

```
FileOutputStream archivo = new FileOutputStream("enteros.dat");
```

El constructor puede arrojar una excepción del tipo `FileNotFoundException`, que hereda de `IOException`. La sentencia creará en el disco el archivo *enteros.dat*. Si ya existía, borrará la versión anterior y lo sustituirá por una nueva.

Como ocurría con los archivos de texto, el nombre del archivo puede incluir una ruta de acceso, con los requisitos que vimos en la Unidad 10. Una vez creado este flujo, lo «envolvemos» en un objeto de la clase `ObjectOutputStream`.

```
ObjectOutputStream out = new ObjectOutputStream(archivo);
```

El constructor de `ObjectOutputStream` puede arrojar una excepción `IOException` —excepción de entrada-salida—. Por tanto, debe ir encerrado en una estructura `try-catch`, que puede englobar también al constructor del objeto `FileOutputStream`.

La clase `ObjectOutputStream` tiene una serie de métodos que permiten la escritura de datos complejos de cualquier tipo o clase, serializándolos antes de enviarlos al flujo de salida. Para ello, las clases de estos datos deben tener implementada la interfaz `Serializable`, que no es más que una especie de sello que declara a sus objetos como susceptibles de ser serializados, es decir, convertibles en una serie plana de bytes.

Las clases implementadas por Java, como `String`, las colecciones (que veremos en la Unidad 12) y las tablas, traen implementadas la interfaz `Serializable`. Los objetos de

estas clases, así como los datos de tipo primitivo, son serializados automáticamente por Java. En cambio, las clases definidas por el usuario deben declararse como serializables en su definición, sin que esto nos obligue a implementar ningún método especial.

```
class miClase implements Serializable {
    //cuerpo de la clase
}
```

Con esto, `miClase` ya es serializable, y sus objetos susceptibles de ser enviados por un flujo binario.

`ObjectOutputStream` dispone de los siguientes métodos para la escritura de datos en un flujo de salida:

- `void writeBoolean(boolean b)`: escribe un valor `boolean` en el flujo.
- `void writeChar(int c)`: escribe el valor `char` que ocupa los dos bytes menos significativos del valor entero que se le pasa como parámetro.
- `void writeInt(int n)`: escribe un entero.
- `void writeLong(long n)`: escribe un entero largo.
- `void writeDouble(double d)`: escribe un número de tipo `double`.
- `void writeObject(Object o)`: escribe un objeto serializable.

Los primeros 5 métodos nos permiten escribir en un fichero datos de tipo “primitivo”. El método writeObject() nos permite escribir objetos más complejos como por ejemplo Arrays, Strings u otros objetos de clases definidas por el programador con la condición de que implementen la interfaz Serializable.

La **serialización** en Java es el proceso de convertir un objeto en una secuencia de bytes para almacenarlo en un archivo, enviarlo por red o guardarlo en una base de datos. La deserialización es el proceso inverso: reconstruir el objeto a partir de esos bytes.

Para que un objeto sea serializable:

1. La clase debe implementar la interfaz java.io.Serializable (es una interfaz marcadora, no tiene métodos).
2. Todos los campos deben ser serializables (tipos primitivos, String, otras clases Serializable).
3. Los campos **static** nunca se serializan (son de clase, no de instancia).
4. Los campos marcados como transient no se serializan.

De este modo si tenemos una secuencia de números en una tabla `t` podemos escribirla de golpe en el fichero con una única sentencia:

```
out.writeObject(t)
```

Esta distinción a la hora de guardar los datos en el fichero habrá que tenerla en cuenta después a la hora de leer los datos del mismo fichero.

**E1101.** Escribir en un archivo datos.dat los valores de diez números enteros.

Repite la actividad escribiendo los números enteros en el archivo *datos.dat* como un array y no como números enteros individuales.

**E1102.** Escribe como una cadena, en el fichero binario *cancionPirata.dat*, la siguiente estrofa:

```
Con diez cañones por banda,
viento en popa a toda vela,
no corta el mar, sino vuela
un velero bergantin.
```

Hasta Java 7 los flujos había que cerrarlos con el método `close()`, incluyéndolo en el bloque finally. Esto no es necesario y el cierre será automático si usamos una estructura `try-catch` con recursos.

**E1103.** Pedir un entero **n** por consola y, a continuación, generar **n** números aleatorios de tipo `double`, que iremos insertando en una tabla. Guardar la tabla en un archivo binario.

Implementa un programa que lea la tabla de números guardada en ese fichero de datos y los muestre por pantalla.

# 2. Flujos de entrada binarios

Para leer de fuentes de datos binarios, usaremos flujos de la clase `ObjectInputStream`, construidos a partir de un flujo de bytes planos `FileInputStream`. Por ejemplo, si leemos los datos escritos en el archivo *datos.dat* de las actividades propuesta y resuelta 11.1, crearemos un flujo de entrada asociado al archivo.

```
ObjectInputStream flujoEntrada = new ObjectInputStream(new
                                 FileInputStream("datos.dat"));
```

Esta sentencia puede producir una excepción `IOException`; por tanto, deberá ir encerrada en una estructura `try-catch`. Lo mismo ocurre a la hora de cerrarlo con el método `close()`, aunque nosotros usaremos habitualmente una estructura `try-catch` con recursos.

Los métodos de la clase `ObjectInputStream` permiten leer los mismos datos que grabamos con `ObjectOutputStream`. Por cada método de escritura de esta última hay otro de lectura de la primera. En el caso de que hayamos grabado los 10 enteros de una tabla por separado usando `writeInt()`, los podemos recuperar, también por separado, con el método `readInt()`, que puede arrojar una excepción `IOException` si hay un error de lectura o `EOFException` si se ha llegado al final del fichero.

**E1104.** Lee los números enteros (de tipo `int`) contenidos en el archivo *datos.dat* generado en la actividad E1101 y muéstralos por pantalla

Los métodos más importantes de `ObjectInputStream` son los siguientes:

- `boolean readBoolean()`: lee un booleano del flujo de entrada.
- `char readChar()`: lee un carácter.
- `int readInt()`: lee un entero.
- `long readLong()`: lee un entero largo.
- `double readDuoble()`: lee un número real de tipo `double`.
- `Object readObject()`: lee un objeto.

Dado que las tablas son objetos, si se ha guardado la tabla `t` usando el método `writeObject()`, en vez de un bucle `for` para la lectura, usaremos una sentencia única, ya que lo que hay guardado es un objeto, no una serie de enteros.

**E1105.** Leer una tabla de enteros de un archivo *datos.dat*.

Al utilizar `readObject()` hay que tener en cuenta que el valor devuelto es de tipo `Object` por lo que normalmente será necesario hacer un *cast* o conversión de estrechamiento de tipos para procesarlo adecuadamente.

Por otra parte, llama la atención la excepción `ClassNotFoundException`, que puede ser arrojada por el método `readObject()`. Esto se debe a que, cuando leemos un objeto de un flujo de entrada, puede ocurrir que la clase a la que pertenece no sea visible desde el lugar del código donde se invoca `readObject()`, debido a que no esté en el mismo paquete ni haya sido importada de otro.

Como ya vimos, las cadenas de texto son objetos y, si se guardaron como tales, se deben recuperar utilizando el método `readObject()`.

```
try {
        String cadena = (String) flujoEntrada.readObject();
} catch (ClassNotFoundException ex) {
        System.out.println(ex.getMessage());
}
```

**E1106.** Recuperar la estrofa del archivo *cancionPirata.dat* de la actividad E1102 y mostrar la por consola.

A menudo desconocemos el número de datos guardados en un archivo. En este caso, para recuperarlos todos, no podemos usar un bucle `for` controlado por contador, sino que tenemos que leer hasta que se llegue al final del fichero, es decir, hasta que salte la excepción `EOFException`. Por ejemplo, si un fichero contiene una lista de enteros y no sabemos cuántos hay, para recuperarlos todos, usamos un bucle infinito del que solo nos puede sacar la excepción `EOFException` de fin de fichero.

```
try {
     while (true) {
             System.out.println(in.readInt());
     }
}catch(EOFException ex) {
     System.out.println("Fin de fichero");
}
```

Cuando se haya leído el último entero, se habrá llegado al final del fichero. Entonces se arrojará la excepción y el programa saldrá del bucle `while` y del bloque `try` para continuar en el bloque `catch`.

**E1107.** Grabar en el fichero *numeros.dat* una serie de números enteros no negativos introducidos por teclado. La serie acabará cuando escribamos -1. Abrir de nuevo *numeros.dat* para lectura y leer todos los números hasta el final del fichero, mostrándolos por pantalla y copiándolos en un segundo fichero *numerosCopia.dat*.

# 3. Ficheros binarios y objetos complejos

Los objetos que queremos guardar en un archivo binario no siempre son tan simples como una cadena de caracteres. La mayoría pertenecen a clases con atributos, que muchas veces son también objetos. Los valores de estos atributos, en realidad, son solo referencias a los objetos propiamente dichos. Entonces se plantea la siguiente cuestión: ¿qué se guarda en el fichero, el valor del objeto referenciado o solo la referencia? Si fuera solo la referencia, no estaríamos guardando la información que nos interesa, ya que las referencias cambian en cada ejecución del programa. Si leyéramos el archivo al día siguiente en una nueva ejecución del programa, no recuperaríamos la información del objeto, sino la dirección de memoria que tenía cuando se guardó.

Supongamos que queremos guardar una tabla de objetos de la clase `Socio`. Pasaremos al método `writeObject()` la variable `tablaSocios`, que guarda la referencia a la tabla en la memoria. Pero no olvidemos que cada componente de la tabla guarda, a su vez, la referencia a un objeto de la clase `Socio`, no el objeto propiamente dicho. ¿Qué se guarda realmente en el archivo? La respuesta es: toda la información necesaria para reconstruir la tabla cuando se vuelva a leer del archivo. Esto incluye: la propia tabla y los objetos referenciados en cada componente, con la información sobre su clase y los valores de los atributos, incluidos aquellos que referencian otros objetos, como el nombre o el `dni`. Los atributos pueden ser incluso otras tablas, como la lista de los familiares del socio, que se guardarían de la misma forma. Java rastrea todas las referencias a objetos hasta construir la estructura completa de los datos, y guarda toda la información necesaria para reconstruir de nuevo el objeto guardado, junto con todos los objetos referenciados desde él, cuando se lea más tarde con `readObject()`.

**E1108.** Implementar un programa que guarde en el fichero socios.dat una tabla de objetos Socio. Después se abrirá de nuevo el fichero en modo lectura para recuperar la tabla de socios, mostrándose por pantalla.

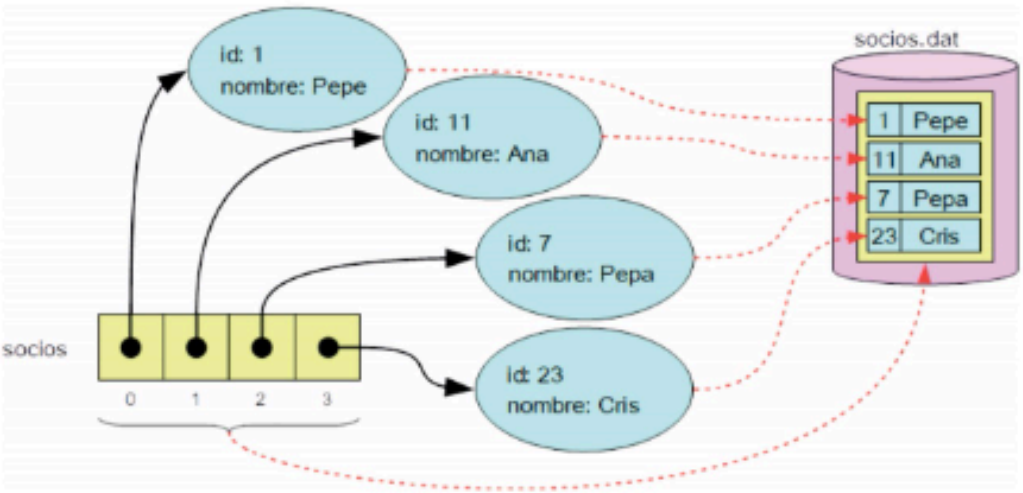


**Figura 11.1.** *Se guarda la tabla* `socios` *en el fichero socios.dat*

**Recuerda**

En una estructura `try-catch`, el bloque catch puede capturar más de un tipo de excepción. Para ello, basta escribir en el paréntesis todos los tipos separados por barras verticales, poniendo al final el nombre del parámetro que referencia la excepción, que funcionará como variable local dentro del bloque.

**E1109.** Implementar un programa que registra la evolución temporal de la temperatura en una ciudad. La aplicación mostrará un menú que permite añadir nuevos registros de temperatura y mostrar el listado de todos los registros históricos. Cada registro constará de la temperatura en grados centígrados, introducida por teclado, y la fecha y hora, que se leerá del sistema en el momento de la creación del registro.

**Ejercicio: Personajes de rol**

Escribe un programa que genere 10 personajes de un juego de rol y los almacene en un archivo binario.

Escribe otro programa que lea 10 personajes de un juego de rol de un archivo binario, los almacene en un array y los muestre por pantalla.

```
// Genera y escribe 10 Personajes de rol en un fichero binario
try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("personajeRol.dat"))) {

    for (int i = 0; i < 10; i++) {
        out.writeObject(new Personaje("Jugador" + i));
    }

    out.close();
```

```
} catch (IOException ex) {
    System.out.println("Escribiendo... Error Entrada/Salida");
}


Personaje[] tabla = new Personaje[10];
// Lee 10 Personajes de rol de un fichero binario
try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream("personajeRol.dat"))) {

    for (int i = 0; i < 10; i++){
        tabla[i] = (Personaje) in.readObject();
    }

} catch (EOFException ex) {
    System.out.println("Leyendo... Excepción fin de fichero");
} catch (IOException ex) {
    System.out.println("Leyendo... Error Entrada/Salida");
} catch (ClassNotFoundException ex) {
    System.out.println("ClassNotFoundException");
}

for (Personaje p: tabla)
    System.out.println(p);
```

**Comentarios adicionales.**

1. **Manejo de versiones:** Si se modifica la clase serializada después de guardar datos, es posible que los objetos guardados dejen de ser compatibles y se produzcan errores. Para evitar estos errores será necesario gestionar las posibles versiones de clases y objetos.
   Investiga sobre el **serialVersionUID** para gestionar distintas versiones de clases serializables.

2. **Seguridad:** La deserialización de objetos puede ser un riesgo de seguridad. Investiga como evitar el casting crudo o sin comprobación al leer objetos del fichero binario.

3. **Alternativas a ficheros binarios:** La serialización Java es útil para persistencia rápida de objetos, pero tiene limitaciones. Para datos más complejos o persistentes:
   - Bases de datos (SQLite, MySQL, etc.)
   - Ficheros de texto JSON (con librerías como Gson, Jackson, etc.)
   - Ficheros de texto XML (con JAXB)