Ficheros Binarios

Trabajar con un fichero de texto es sencillo. No obstante, suponte que ahora tenemos que almacenar en un fichero un audio o una imagen. Para estos casos, se emplea otro juego de clases diferente, que nos permite almacenar secuencias de bytes.

Como ya hemos mencionado, este apartado se asemeja bastante a los contenidos de la asignatura de PSP de 2ndo del apartado de los **sockets**.

Fichero binario – escritura (bytes)

La clase **FileOutputStream** nos permite definir un flujo que escribe **bytes** en un fichero de modo secuencial. Esta clase es similar a **FileWriter** en lo relativo a sus constructores y modos de apertura; no los vamos a repetir aquí.

La única diferencia notable con FileWriter es que el flujo de FileOutputStream requiere de un buffer. El método de escritura principal es **write ()**.

Un ejemplo de escritura secuencial de un fichero binario podría ser:

```
public void writeBinaryFile() {
    String texto = "Información a almacenar";
    File file = null;
    FileOutputStream output = null;
    try {
        // Create a File
        file = new File("example.txt");

        // Create a FileOutputStream
        output = new FileOutputStream(file);

        // Turn the info into bytes...
        byte[] bytes = texto.getBytes();

        // Write all info
        output.write(bytes);
    } catch (Exception e) {
        System.out.println("An error occurred.");
    } finally {
        // Close the FileOutputStream
        try {
            if (output != null)
                output.close();
        } catch (IOException e) {
            // Nothing to do here...
        }
    }
}
```

Si abres el fichero con el Notepad, verás que ha escrito el texto de forma legible. Esto es porque un carácter no deja de ser 4 bytes.

Fichero binario – lectura (bytes)

De forma similar, la lectura de un fichero binario se realiza mediante la clase **FileInputStream**, que define un flujo de lectura de un fichero secuencial. Obviamente, esta clase lee **bytes**. Sus constructores equivalen al **FileReader.**

De nuevo, el método de lectura principal es **read ()**, que devuelve un array de bytes (buffer).

Un ejemplo de lectura secuencial de un fichero binario podría ser:

```
public void readBinaryFile() {
    File file = null;
    FileInputStream input = null;
    try {
        // Create a File
        file = new File("example.txt");

        // Create a FileInputStream
        input = new FileInputStream(file);

        // Loop to read a bytes until the end of file
        int i = 0;
        while ((i = input.read()) != -1) {
            System.out.println(i);
        }
    } catch (Exception e) {
        System.out.println("An error occurred.");
    } finally {
        // Close the reader
        try {
            if (input != null)
                input.close();
        } catch (IOException e) {
            // Nothing to do here...
        }
    }
}
```

En este caso lo que obtenemos es una secuencia de números, en realidad, el valor de cada byte leído del fichero.

Ejercicio Propuesto - 4

Crea un programa en Java que duplique imágenes. Al arrancar el programa, se lee una imagen y se genera otro ficho con el nombre _copia con el mismo contenido. Genera la copia mediante flujos binarios (bytes).

Fichero binario – escritura (tipos primitivos)

Si estuviésemos leyendo y escribiendo bytes de ficheros continuamente, nuestro programa sería bastante ineficiente. Y lo que es peor, podríamos meter la pata y no darnos cuenta.

¿Te acuerdas cuando he dicho que un carácter son 4 bytes? Bueno, para empezar, no siempre es así. Suponiendo que lo sea, lo que viene a significar esto es que, si esos 16 bytes de un fichero son caracteres, eso es que hay 16/4 = 4 caracteres en el fichero.

Guay, ¿verdad?

Pues no, porque:

- En Java 4 bytes también es int. Luego en el fichero podría haber 4 enteros en vez de 4 caracteres.

- Un long son 8, así que, podrían ser 6/8 = 2 números long.

Intentar hacer esta conversión manualmente (coger tú los bytes y decir que son un int, o un long, o lo que sea) es peligroso y tedioso dado que tenemos que estar al tanto de qué tipo de dato es a la hora de programar.

Pero hay una solución.

La clase **DataOutputStream** nos permite definir un flujo que escribe **bytes** en un tipo primitivo que le especifiquemos. Esta clase es similar a **FileWriter** en lo relativo a sus constructores y modos de apertura.

El flujo de DataOutputStream no dispone de un método genérico write (), sino de un tipo específico para cada tipo de dato primitivo de java **writeUTF (), writeInt ()**, etc.

Fichero binario – lectura (tipos primitivos)

De forma similar, la clase **DataInputStream** nos permite definir un flujo que escribe **bytes** en un fichero de modo secuencial, pero retornando tipos primitivos de Java. Esto quiere decir que lee bytes, pero retorna un int, un double, etc. según necesites, sin más conversiones. Esta clase es similar a **FileReader** en lo relativo a sus constructores y modos de apertura.

El flujo de DataInputStream no dispone de un método genérico read (), sino de un tipo específico para cada tipo de dato primitivo de java **readUTF (), readInt ()**, etc.

Un ejemplo de lectura y escritura secuencial de un fichero binario podría ser:

```
public void writeBinaryFile() {
    File file = null;
    FileOutputStream output = null;
    DataOutputStream outputStream = null;
    try {
        // Create a File
        file = new File("example.txt");

        // Create a FileOutputStream
        output = new FileOutputStream(file);

        // Create a DataOutputStream
        outputStream = new DataOutputStream(output);

        // Write all info
        for (int i = 0; i < cesta.length; i++) {
            outputStream.writeUTF(cesta[i]); // inserta nombre
            outputStream.writeInt(cantidad[i]); // inserta edad
        }

    } catch (Exception e) {
        System.out.println("An error occurred.");
    } finally {
        // Close the DataOutputStream
        try {
            if (outputStream != null)
                outputStream.close();
        } catch (IOException e) {
            // Nothing to do here...
        }
        // Close the FileOutputStream
        try {
            if (output != null)
                output.close();
        } catch (IOException e) {
            // Nothing to do here...
        }
    }
}
```

Fíjate que estamos almacenando DOS tipos de datos primitivos en el mismo fichero binario, uno de ellos, String (en formato UTF). Por tanto, mientras “recordemos” la secuencia en la que vamos introduciendo datos, es posible almacenar una gran variedad de información.

```java
public void readBinaryFile() {
    File file = null;
    FileInputStream input = null;
    DataInputStream inputStream = null;
    try {
        // Create a File
        file = new File("example.txt");

        // Create a FileInputStream
        input = new FileInputStream(file);

        // Create a DataOutputStream
        inputStream = new DataInputStream(input);

        // Loop to read a bytes until the end of file
        while (input.getChannel().position() < input.getChannel().size()) {
            String producto = inputStream.readUTF();
            int cant = inputStream.readInt();
            System.out.println("Producto: " + producto + ", cant: " + cant);

        }
    } catch (Exception e) {
        System.out.println("An error occurred.");
    } finally {
        // Close the DataInputStream
        try {
            if (inputStream != null)
                inputStream.close();
        } catch (IOException e) {
            // Nothing to do here...
        }
        // Close the reader
        try {
            if (input != null)
                input.close();
        } catch (IOException e) {
            // Nothing to do here...
        }
    }
}
```

De nuevo, date cuenta de que este código funciona porque “nos acordamos” de la secuencia en la que introdujimos los datos: primero String, luego entero, luego String, luego entero…

No es que sea la cosa más práctica del mundo, pero estamos trabajando a bajo nivel, lo que nos da mucha flexibilidad. Mientras tengamos clara la secuencia, podemos leer y escribir todo lo que queramos en un fichero binario.

Ejercicio Propuesto - 5

Realiza un programa que almacene y lea facturas en un fichero. Cada factura tiene una id (int), un cliente (String), un pagador (String) y un importe (long). Permite añadir y mostrar todas las facturas del fichero.

Fichero binario – escritura (objetos)

Te habrás dado cuenta de que, primero empezamos a trabajar con bytes, luego con tipos primitivos, luego lógicamente ahora tendremos que pasar a **objetos**. Java dispone de dos clases para esto, la clase **ObjectOutputStream** y la clase **ObjectInputStream**. Ambas están derivadas de **OutputStream** e **InputStream**, y su manejo es similar.

No vamos a extendernos con esto. Aquí tienes un ejemplo de lectura y escritura con objetos tipo Mensaje (POJOS).

```
public void writeBinaryFile(Mensaje mensaje) {
    File file = null;
    FileOutputStream outputStream = null;
    ObjectOutputStream objectOutputStream = null;
    try {
        // Create a File
        file = new File("example.dat");

        // Create a FileOutputStream
        outputStream = new FileOutputStream(file);

        // Create a ObjectOutputStream
        objectOutputStream = new ObjectOutputStream(outputStream);

        // Write all info
        objectOutputStream.writeObject(mensaje);

    } catch (Exception e) {
        System.out.println("An error occurred.");
    } finally {
        // Close the ObjectOutputStream
        try {
            if (objectOutputStream != null)
                objectOutputStream.close();
        } catch (IOException e) {
            // Nothing to do here...
        }
        // Close the FileOutputStream
        try {
            if (outputStream != null)
                outputStream.close();
        } catch (IOException e) {
            // Nothing to do here...
        }
    }
}
```

```
public void readBinaryFile() {
    File file = null;
    FileInputStream inputStream = null;
    ObjectInputStream objectInputStream = null;
    try {
        // Create a File
        file = new File("example.dat");

        // Create a FileInputStream
        inputStream = new FileInputStream(file);

        // Create a ObjectInputStream
        objectInputStream = new ObjectInputStream(inputStream);

        // Loop to read
        while (inputStream.getChannel().position()
                < inputStream.getChannel().size()) {
            Mensaje mensaje = (Mensaje) objectInputStream.readObject();
            System.out.println(mensaje.toString());
        }
    } catch (Exception e) {
        System.out.println("An error occurred.");
    } finally {
        // Close the ObjectInputStream
        try {
            if (objectInputStream != null)
                objectInputStream.close();
        } catch (IOException e) {
            // Nothing to do here...
        }
        // Close the FileInputStream
        try {
            if (inputStream != null)
                inputStream.close();
        } catch (IOException e) {
            // Nothing to do here...
        }
    }
}
```

Lo importante de todo este ejemplo es la clase Mensaje. Internamente, no es más que un POJO con sus getters, setters, toString… Si te fijas, se usa como si fuese un molde. Lee del fichero unos bytes, y los va 'encajando' como piezas en un objeto Mensaje. Esto funciona perfectamente mientras el fichero haya sido escrito mediante un 'molde' que sea **exactamente igual**. Lógico… ¿no?

Pues no, nos faltan un par de cosas.

La **serialización** es la transformación de un objeto en una secuencia de bytes. Esa secuencia de bytes puede ser posteriormente leídos de vuelta para reconstruir el objeto original. A esto se llama **deserialización**.

Para poder transformar el objeto en una secuencia de bytes, el objeto debe ser Serializable, para lo que tiene que implementar la interface Serializable.

Si un objeto contiene atributos que son referencias a otros objetos éstos a su vez deben ser serializables. Todos los tipos básicos Java son serializables, así como los arrays y los String. Los demás, deberías indicarlos tú siempre.

De nuevo, esto de la serialización no es única de los ficheros. En la asignatura de PSP de 2ndo en el apartado de los **sockets**, para enviar un objeto a través de uno, también hace falta que dicho objeto sea serializable.

En nuestro ejemplo:

```
public class Mensaje implements Serializable {

    private static final long serialVersionUID = 378393508666255153L;

    private int id = 0;
    private String emisor = null;
    private String receptor = null;
    private String mensaje = null;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    .
    .
    .
```

Por cierto…

El **serialVersionUID** es un identificador único que se utiliza en la **serialización y deserialización** de objetos en Java. Sirve para verificar que la clase usada para **guardar** el objeto es compatible con la clase usada para **leerlo**. Es decir, que el programa que escribe un Mensaje el fichero tiene la misma versión que el que lee Mensajes del fichero.

La idea es parecida a las versiones de Java. Si intentas usar una versión de Java incompatible, saldrá un error. Aquí, saltará una **InvalidClassException**.

Si no lo añades, no importa porque Java genera uno; ahora, te expones a que el día de mañana con cualquier cambio menor que hagas, el serialVersionUID cambie, dando errores al tratar de leer el fichero.

Hay que tener especialmente cuidado con esto cuando trabajamos son servidores, por ejemplo (de nuevo, Sockets). Si enviamos un Mensaje de la versión 4 y el servidor dispone de la 3, habrá un error de compatibilidad y no podremos acceder a la información del Mensaje.

Práctica - 2

Crea un sistema visual que permita trabajar con las Notas de la asignatura de ADT de los alumnos de 2DAM. Cada Nota se registra de la siguiente manera:

- Id: tipo de dato int.
- Nombre: tipo de dato String.
- Apellido: tipo de dato String.
- Fecha: tipo de dato Date.
- Nota: tipo de dato Integer.

Dispondrás de las siguientes opciones:

- Añadir: Cada vez que se pulse añadir, se mostrarán los datos en un JTable. Debo de poder añadir tantas Notas como quiera.

- Guardar: Se volcará el contenido de lo que haya en ese momento en el JTable en un fichero binario Notas.dat.

- Cargar: Se recuperará el contenido del fichero y se cargarán sus datos en la JTable.

Adicionalmente:

- Actualizar: Al darle actualizar, en lugar de volcar el contenido del JTable y reescribir el fichero, se añadirán al fichero solamente las filas que NO existan previamente en el fichero.

- Ordenar: Ordenará de mayor o menor las Notas del fichero por Id, y luego refrescará el JTable.