DESARROLLO DE APLICACIONES WEB

Unidad 10

Lectura y escritura de la información

1r DAW

IES La Mola de Novelda

Departament d'informàtica

Profesor: Joan Carbonell Picó

ÍNDEX

Unidad 10: Lectura y Escritura de la Información

1.- INTRODUCCIÓN

Muchas veces tendremos que interaccionar con alguna fuente de datos, como un archivo de disco duro, una unidad óptica o un dispositivo de red, ya sea para guardar información o para recuperarla. Java implementa una serie de clases llamadas **flujos**, encargadas de comunicarse con los dispositivos de almacenamiento. Desde el punto de vista del programador, el funcionamiento de estos flujos no depende del tipo de dispositivo con el que está asociado, por lo que no tendremos que tener en cuenta las características físicas de cada uno de ellos.

Los flujos pueden ser de entrada o de salida, según sean para guardar o recuperar información. Además, según el tipo de datos que se transmiten, los flujos son de dos tipos:

- Carácter (Texto) → Si se asocia a archivos u otras fuentes de tipo texto.
- Binarios → Si transmiten bytes, enteros entre 0 y 255. Esto, en realidad, permite manipular cualquier tipo de datos.

2.- FICHEROS DE TEXTO

2.1.-FLUJOS DE ENTRADA

Los flujos de entrada de tipo texto heredan de la clase ***InputStreamReader***. Las clases de entrada de texto tienen siempre un nombre que termina en ***Reader***. Usaremos para este tipo de flujos el ***FileReader***, a cuyo constructor se le pasa el nombre del archivo (ruta ce acceso incluida) que queremos asociar al flujo para su lectura:

FileReader (string nombreArchivo)

- Ejemplo:

```
FileReader inWindows = new FileReader(''C:\\programes\\prova.txt'');
FileReader inLinux = new FileReader(''/home/joan/programes/prova.txt'');
```

La obertura de un fichero puede arrojar una excepción del tipo ***IOException*** cuando el archivo no se abre por alguna razón. En verdad, el constructor ***FileReader*** puede arrojar una excepción ***FileNotFoundException*** que hereda de ***IOException***, y que podría usar para la apertura del fichero, separándola del resto del código. Por lo tanto, esta operación siempre deberá ir dentro de la estructura ***try-catch*** correspondiente.

Una vez se ha abierto el archivo para la lectura, el cursor se posiciona al principio, apuntando al primer carácter, y ya estamos preparados para leer el flujo asociado utilizando el siguiente método:

int read() → Lee un carácter (en Unicode) de un fichero no enriquecido (texto plano) y lo devuelve como entero. Eso significa que cada carácter ocupará los dos bytes menos significativos del entero devuelto (un entero ocupa 4 bytes). Por lo tanto, para recuperar el carácter que lleva dentro cada entero, deberemos aplicarle un cast. Cuando la función devuelva un -1, sabremos que hemos llegado al final del archivo, ya que este valor no corresponde con ningún carácter.

A medida que vayamos invocando al método ***read()***, el cursor irá avanzando dentro del archivo, apuntando al siguiente carácter. Una vez que terminemos de leer el flujo, tendremos que cerrarlo.

void close() → Cierra el flujo de entrada con objeto de completar las lecturas pendientes y liberar el archivo.

- Ejemplo: Apertura, lectura y cierre de un archivo.

```
String text = "";
try{
    FileReader in = new FileReader("/home/lliurex/Projectes-Idea-Joan/unitat-10/ArxiusDeText/src/ArxiusDeText.java");
    int c = in.read();
    while (c!=-1){
        text = text + (char)c;
        c = in.read();
    }
    in.close();
}catch(IOException ioe){
    System.out.println(ioe.getMessage());
}
System.out.println("Arxiu llegit:");
System.out.println(text);
```

Podemos ver que siempre tendremos que indicar la ruta absoluta de la ubicación del archivo. Si queremos utilizar la ruta relativa del archivo a leer ya que podemos estar cambiando de ordenador, haremos lo siguiente:

System.getProperty("user.dir") → Nos devuelve un String con el directorio raíz de trabajo donde se ejecuta el programa actual. En nuestro ejemplo, si lo imprimimos en pantalla obtendremos: **/home/lliurex/Projectes-Idea-Joan/unitat-10/ArxiusDeText**

File.separator → Representa el separador de directorios dependiendo el sistema operativo en el que estemos.

La modificación del código anterior será:

```
String directorioRaiz = System.getProperty("user.dir");
String rutaArchivo = directorioRaiz+ File.separator+"src"+File.separator+"ArxiusDeText.java";
String text = "";
try{
    FileReader in = new FileReader(rutaArchivo);
    int c = in.read();
    while (c!=-1){
        text = text + (char)c;
        c = in.read();
    }
    in.close();
}catch(IOException ioe){
    System.out.println(ioe.getMessage());
}
System.out.println("Arxiu llegit:");
System.out.println(text);
```

En el ejemplo podemos ver que la variable texto contiene todo el texto del archivo ArxiusDeText.java, incluidos los cambios de línea. Para ello, estamos accediendo al disco tantas veces como caracteres contiene el archivo. Sin embargo, esta forma de leer de un dispositivo físico no es muy eficiente ya que las operaciones de acceso al disco son extremadamente lentas. Normalmente se usan flujos de la clase ***BufferedReader***.

Los flujos de la clase ***BufferedReader*** no es más que un ***FileReader*** filtrado para asociarle un *búfer* (un búfer es un espacio reservado para el almacenamiento temporal) en memoria. Esto nos permite poder hacer lecturas en el dispositivo físico de grupos de caracteres, en vez de caracteres individuales, los cuales son colocados en cola en el búfer a la espera de que el programa los vaya reclamando. Para crear un ***BufferedReader*** basta con pasarle al constructor un objeto ***FileReader***

BufferedReader in = new BufferedReader(new FileReader(rutaArchivo));

El flujo **in** dispone del método ***read()*** para hacer lectura de caracteres individuales pero además, al tener un *búfer* asociado, puede hacer lecturas de línea completas con el método:

String readLine() → Devuelve una cadena con la línea leída (que concluye en el siguiente carácter fin de línea, el cual lo descarta). Al llegar al final del fichero, la función devuelve un ***null***.

- Ejemplo: Modificación del programa anterior implementado con el método ***readLine()***.

```
String directorioRaiz = System.getProperty("user.dir");
String rutaArchivo = directorioRaiz+ File.separator+"src"+File.separator+"ArxiusDeText.java";
String text = "";
 try{
        BufferedReader in = new BufferedReader(new FileReader(rutaArchivo));
        String linia = in.readLine();
        while(linia!=null){
```

```
            text = text + linia + '\n';//La línia no inclou el salt de línia
            linia = in.readLine();
        }
        in.close();
}catch(IOException ioe){
        System.out.println(ioe.getMessage());
}
System.out.println("Arxiu llegit:");
System.out.println(text);
```

2.2.-FLUJOS DE SALIDA

Para escribir en un archivo de texto, necesitaremos un flujo de salida de texto. Para ello, crearemos un objeto de a clase ***FileWriter***, que hereda de ***OutputStreamWriter***. Las clases de salida de texto tiene un nombre que acaba en ***Writer***. Los constructores de ***FileWriter*** son:

- **FileWriter (string nombreArchivo)** → Destruye la versión anterior del archivo y escribe en él desde el principio
- **FileWriter (string nombreArchivo, boolean append)** → Si el valor booleano *append* vale true, nos permite añadir texto al final del archivo, respetando el contenido anterior.

Como hemos visto anteriormente, nombreArchivo puede contener la ruta de acceso.

La apertura de un ***FileWriter*** puede generar una excepción del tipo ***IOException***, que se habrá que tratarlo con un ***try-catch***.

De la misma manera que indicamos con el ***FileReader***, para mejorar el rendimiento usaremos una versión con *búfer*, ***BufferedWriter***, a cuyo constructor se le pasa como parámetro un flujo de salida. Por ejemplo:

BufferedWriter out = new BufferedWriter(new FileWriter(nombreArchivo))

Los métodos de que disponemos son:

void writer (int carácter) →Escribe un carácter en el archivo.

void writer (String cadena) → Escribe una cadena en el archivo.

void newLine () → Escribe una salto de línea en el fichero. Se deben evitar los saltos de línea escribiendo la secuencia de escape ‘\n’, ya que son sensibles a la plataforma utilizada.

void flush () → Vacía el *búfer* de salida, escribiendo en el fichero los caracteres pendientes.

void close () → Cierra el flujo de salida, vaciando el *búfer* y liberando el recurso

correspondiente.

- Ejemplo: Guardar un par de líneas en un archivo.

```
try{
        BufferedWriter out = new BufferedWriter(new FileWriter("quijote.txt"));
        String cad = "En un lugar de la mancha,";
        for (int i =0; i < cad.length();i++){
           out.write(cad.charAt(i));
        }
        cad = "de cuyo nombre no quiero acordarme.";
        out.newLine();
        out.write(cad);;
        out.close();

}catch (IOException e) {
        System.out.println(e.getMessage());
}
```

Es muy común olvidarse de cerrar el flujo. El resultado puede ser encontrarse un archivo vacío, sin ningún carácter escrito, debido a que los caracteres se han guardado en el *búfer*, pero no han llegado a escribirse en el archivo antes de que el programa termine.

A partir de la versión 7 de Java, disponemos de una estructura para cerrar archivos o liberar cualquier recurso, sin necesidad de utilizar ***finally***. Se trata de la estructura ***try-catch-resources***. Por ejemplo:

```
try(BufferedReader in = new BufferedReader(new FileReader("quijote.txt"))){
        String linia = in.readLine();
        while(linia!=null){
           text = text + linia + '\n';
           linia = in.readLine();
        }

}catch(IOException e){
        System.out.println(e.getMessage());
}
System.out.println("QUIJOTE:");
System.out.println(text);
```

En este caso, nos estamos asegurando de que se cerrará el archivo ocurra lo que ocurra. Tanto si se produce la excepción como si no, el flujo **in** se cierra automáticamente al terminar de ejecutarse la estructura ***try-catch***.

2.3.-CIERRE DEL ARCHIVO

Hay que recordar que siempre tendremos que cerrar el objeto creado para trabajar con ficheros. Como hemos visto anteriormente, con la estructura ***try-catch-resources*** se cierra automáticamente. Pero si utilizamos la estructura ***try-catch***, una forma de asegurarnos que se cerrará el archivo es poniéndolo en la cláusula finally. Con ellos estamos haciendo un uso de la

cláusula ***finally*** como cierre de recursos. Por ejemplo:

```
String directorioRaiz = System.getProperty("user.dir");
String rutaArchivo = directorioRaiz+ File.separator+"src"+File.separator+"ArxiusDeText.java";
String text = "";
//LECTURA CARÁCTER A CARÁCTER
FileReader in = null;
try{
        in = new FileReader(rutaArchivo);
        while (c!=-1){
           text = text + (char)c;
           c = in.read();
        }
}catch(IOException ioe){
        System.out.println(ioe.getMessage());
}finally {
        if (in!=null)
           try{
              in.close();
           }catch (IOException ioe){
              ioe.printStackTrace();
           }
}
System.out.println("Arxiu llegit:");
System.out.println(text);
```

3.- EJERCICIOS DE FICHEROS DE TEXTO

3.1.-EJERCICIOS A

1. Realizar un programa que solicite al usuario el nombre de un fichero de texto y muestre su contenido en pantalla. Si no se proporciona ningún nombre de fichero, la aplicación utilizará por defecto *prueba.txt*.

2. Diseñar una aplicación que pida al usuario su nombre y edad. Estos datos deben guardarse en el fichero *datos.txt*. Si este fichero existe, debe borrarse su contenido, y en caso de que no exista, debe crearse.

3. Crear un programa que duplique el contenido de un fichero. Realizar dos versiones:

    1. Duplicaremos el fichero *original.txt* en una que se llame *copia.txt*.

    2. Pedir el nombre del fichero fuente y duplicarlo en un fichero con el mismo nombre con el prefijo «*copia_de_».*

4. Realizar un programa que lea un fichero de texto llamado *carta.txt*. Tenemos que contar los caracteres, las líneas y las palabras. Para facilitar el procesamiento supondremos que cada palabra está separada de otra por un único espacio en blanco.

5. En el archivo *numeros.txt* disponemos de una serie de números (uno por cada línea). Diseñar un programa que procese el fichero y nos muestre el menor y el mayor.

6. Un libro de firmas es útil para recoger todas las personas que han pasado por un determinado lugar. Crear un aplicación que permita mostrar el libro de firmas o insertar un nuevo nombre (comprobando que no se encuentre repetido). Llamaremos al fichero *firmas.txt*.

7. En Linux disponemos del comando *more*, al que se le pasa un fichero y lo muestra poco a poco: cada 24 líneas. Implementar un programa que funcione de forma similar

8. Disponemos de dos ficheros, *perso1.txt* y *perso2.txt* con nombres de personas (ambos ordenados). Realizar un programa que lea ambos ficheros y cree un tercer fichero (*todos.txt*) con todos los nombres ordenados alfabéticamente.

9. Un encriptador es una aplicación que transforma un texto haciéndolo ilegible para aquellos que desconocen el código. Diseñar una programa que lea un fichero de texto, lo codifique y cree un nuevo archivo con el mensaje cifrado. El alfabeto de codificación se encontrará en el fichero *codec.txt*. Un ejemplo de alfabeto de codificación es:

Alfabeto	a	b	c	d	e	f	g	h	i	j	k	l	m	n	→
Cifrado	e	m	s	r	c	y	j	n	f	x	i	w	t	a	→

→	o	p	q	r	s	t	u	v	w	x	y	z
→	k	o	z	d	l	q	v	b	h	u	p	g

10. Utilizando el fichero *codec.txt* del ejercicio anterior, diseñar un decodificador.

11. El fichero *matriz.txt*, contiene los datos de una matriz cuadrada. Leer dicha matriz y mostrarla transpuesta en pantalla.

12. Algunos sistemas operativos disponen del comando *comp*, que compara dos archivos y nos dice si son iguales o distintos. Diseñar este comando de forma que, además, nos diga en qué línea y carácter se encuentra la primera diferencia. Utilizar los ficheros *texto1.txt* y *texto2.txt*.

13. Diseñar una pequeña agenda, que muestre el siguiente funcionamiento:

    1. Nuevo contacto.

    2. Buscar por nombre.

    3. Mostrar todos.

4. Salir.

En la agenda, guardaremos el nombre y el teléfono de un máximo de 20 personas. La opción 1 nos permitirá introducir un nuevo contacto siempre y cuando la agenda no esté llena, comprobando que el teléfono no se encuentra insertado ya (mismo teléfono). La opción 2 muestra todos los teléfonos que coinciden con la cadena a buscar. Por ejemplo, si tecleamos «**Pe**», mostrará el teléfono de Pedo, de Pepe y de Petunia. La opción 3 mostrará un listado con toda la información (nombres y teléfonos) ordenados alfabéticamente por el nombre. Por último, la opción 4 guarda todos los datos de la agenda en el archivo *agenda.txt*.

La próxima vez que se ejecute la agenda, se debe comprobar si hay datos guardados y cargarlos.

4.- FICHEROS BINARIOS

Un fichero binario o de datos está formado por secuencias de *bytes*. Nos van a permitir guardar (o transferir) y recuperar (o recibir) cualquier tipo de datos usados en un programa.

Cuando se trata de escribir o leer *bytes* en un fichero, existen dos clases básicas: ***FileOutputStream*** y ***FileInputStream***. Pero lo habitual no es trabajar con *bytes* individuales en nuestros programas, sino con datos (formado por *bytes*) más complejos, ya sean de tipo primitivos, estructuras estáticas/dinámicas u objetos. Por eso, necesitaremos un intermediario capaz de convertir los datos en series planas de *bytes* o reconstruir los datos a partir de series de *bytes*. Estos dos procesos se llaman **serialización** y **deserialización de datos**, respectivamente. Esos intermediarios son flujos llamados *envoltorios*, ***ObjectOutputStream*** y ***ObjectInputStream***, que se crean a partir de flujos de *bytes* planos, como ***FileOutputStream*** y ***FileOutputStream***.

4.1.-FLUJO DE SALIDA

Supongamos que queremos utilizar un flujo de salida de datos para grabar en disco una secuencia de enteros almacenados en una estructura. En primer lugar crearemos el flujo de salida de tipo binario asociado a un fichero llamado *enteros.dat* (igual que hemos hecho con los archivos de texto, el nombre del archivo puede incluir una ruta de acceso):

FileOutputStream archivo = new FileOutputStream(“enteros.dat”);

Una vez creado este flujo, lo envolvemos en un objeto de la clase ***ObjectOutputStream***:

ObjectOutputStream out = new ObjectOutputStream(archivo);

El constructor de ***ObjectOutputStream*** puede lanzar una excepción ***IOExcepcion***, por lo que debe ir encerrado de la estructura ***try-catch***.

La clase ***ObjectOutputStream*** tiene una serie de métodos que permiten la escritura de datos complejos de cualquier tipo o clase. En todos caso, para escribir un objeto, su clase debe tener implementada la interfaz ***Serializable***, que no es más que una marca que declara al objeto en cuestión como susceptible de ser serializado, es decir, convertible en una serie plana de bytes.

Las clases implementadas por Java como ***String***, las ***Collections*** y los ***arrays***, traen implementadas la interfaz ***Serializable*** y no hay que preocuparse de ellas. Lo mismo ocurre con los datos primitivos. 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 esta definición, **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 les pasa

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.

- Ejemplo: Grabar en un archivo un *array* de números enteros.

```
int[] t = new int[10];
try{
        ObjectOutputStream flujoSalida = null;
        flujoSalida = new ObjectOutputStream(new FileOutputStream("datos.dat"));
        //Inicializamos el array
        for(int i=0;i<10;i++)
          t[i] = i;
        //Escribimos en el flujo cada entero
        for(int i=0;i<10;i++)
          flujoSalida.writeInt(t[i]);
        //Cerramos el flujo para que se vacíe el búfer
        //y se escriban los datos pendientes en el archivo
        flujoSalida.close();
}catch(IOException e){
        System.out.println(e.getMessage());
}
```

En este ejemplo hemos obtenido y grabado los enteros por separado. Como un *array* es un objeto en Java, podríamos haberlo escritos como un objeto, por medio del método ***writeObject()***. Por lo tanto el segundo bucle puede ser sustituido por la sentencia:

flujoSalida.writeObject(t);

En esta sentencia le hemos pasado como parámetro la referencia al objeto que queremos grabar, es decir, el *array* **t**. En este caso hemos grabado el *array* como objeto, que no es lo mismo que grabar los enteros por separado. Esta distinción será importante a la hora de recuperarlos.

Igualmente, para guardar una cadena de caracteres se usa la función ***writeObject***, ya que una cadena es un objeto de la clase ***String***. Por ejemplo:

String cadena = “Sancho Panza”;

flujoSalida.writeObject(cadena);

4.2.-FLUJOS DE ENTRADA

Para leer de fuentes de datos, tales como los archivos, usaremos flujos de la clase ***ObjectInputStream***. Por ejemplo, si leemos los datos escritos en el archivo “*datos.dat*” del apartado anterior, crearemos un flujo de entrada asociado al archivo:

ObjectInputStream flujoEntrada = new ObjectInputStream(new FileInputStream(“datos.dat”));

Esta sentencia deberá de ir englobada dentro de la estructura ***try-catch*** ya que puede producir una excepción.

- Ejemplo: Usando los métodos de la clase ***ObjectInputStream*** vamos a leer los datos que grabamos con ***ObjectOutputstream***. Hemos de tener en cuenta que si grabamos los enteros usando el método ***writeInt()***, los podemos recuperar también por separado con el método ***readInt()***, el cual puede lanzar un ***IOException*** si hay un error de lectura o ***EOFException*** si ha llegado al final del archivo.

```
int[] t2 = new int[10];
try{
        ObjectInputStream flujoEntrada = new ObjectInputStream(new FileInputStream("datos.dat"));
        for(int i=0;i<10;i++){
          //Leemos los enteros del archivo binario datos.dat
          //y los metemos en el array de enteros t2
          t2[i]=flujoEntrada.readInt();
        }
        flujoEntrada.close();
        System.out.println(Arrays.toString(t2));
}catch(IOException e){
        System.out.println(e.getMessage());
}
```

En cambio, las cadenas de texto son objetos, y se deben recuperar utilizando el método ***readObject()***. Por ejemplo:

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

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 readDouble() → Lee un número real double.

final Object readObject() → Lee un objeto

En el ejemplo anterior, podemos ver que la estructura ***try-catch*** está asociada a la excepción ***ClassNotFoundException***, que rodea al método ***readObject()***. Esto se debe a que cuando leemos un objeto de un flujo de entrada, puede ocurrir que intentemos asignarlo a una variable de distinta clase, en cuyo caso se arrojaría la excepción mencionada. Por esa razón podemos ver que se le ha aplicado un ***cast*** delante de la sentencia de lectura, en este caso ***String***. Es necesario ya que el

método ***readObject()*** devuelve un ***Object*** y nosotros lo asignamos a una referencia de tipo ***String***, lo cual supone una conversión de estrechamiento. Esto ocurre cuando a una variable de una determinada clase (en este caso ***String***) se le intenta asignar un objeto de una superclase (en este caso ***Object***).

- Ejemplo: Supongamos el ejemplo visto anteriormente. Como los ***arrays*** son objetos, si se ha guardado l'***array* t** usando el método **writeObject(t)**, el bucle *for* usado en la lectura debe ser sustituido por una sentencia única de lectura, ya que lo que hay guardado es un objeto, no una serie de enteros

```
try{
        ObjectInputStream flujoEntrada = new ObjectInputStream(new FileInputStream("datos.dat"));
        t2 = (int[])flujoEntrada.readObject();
        flujoEntrada.close();
        System.out.println(Arrays.toString(t2));
}catch(ClassNotFoundException e){
        System.out.println(e.getMessage());
}catch(IOException e){
        System.out.println(e.getMessage());
}
```

Muchas veces desconocemos el número de datos guardados en un archivo. En este caso, para recuperarlos todos no podemos usar un *for* controlado por un contador, sino que tenemos que leer hasta que se llegue al final del fichero, es decir, hasta que salte la excepción ***EOFException***.

- Ejemplo: Supongamos que un fichero contiene una lista de enteros y no sabemos cuántos hay. Para recuperarlos y mostrarlos todos por pantalla usamos un bucle infinito, el cual será finalizado cuando se produzca la excepción ***EOFException*** de fin de fichero.

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

5.- CIERRE DE FLUJOS

Cuando dejamos de necesitar un flujo, ya sea de entrada o de salida, siempre debemos cerrarlo para provocar el vaciado de lo *búferes* asociados y liberar el recurso. Esto se consigue con el método ***close()***, disponible en todas las clases de entrada y salida que hemos visto anteriormente. Se recomienda incluir el método ***close()*** en un bloque ***finally*** para asegurarnos de que se ejecuta independientemente de si han saltado excepciones o de si todo ha ido correctamente. Por ejemplo, un programa tipo que trabaje con flujos, tendría una estructura similar a la siguiente:

```
//null es necesario para la comparación finally
ObjectOutputStream out = null;
try{
    out = new ObjectOutputStream(new FileOutputStream("archivo.dat"));
    … //sentencias que manipulan el flujo
}catch(IOExcepction e){
    System.out.println(e.getMessage());
}finally{
     try{
          if(out!=null) //si el flujo está abierto
            out.close()
     }catch(IOException e){
          System.out.println(e.getMessage());
     }
}
```

También podemos prescindir de todas estas precauciones, ya que es posible crear el flujo asociándolo, entre paréntesis, al bloque ***try***, dejando al programa encargado del cierre incondicional al final de la ejecución de la estructura. Se dice que estamos usando una estructura ***try*** con recursos (dichos recursos están sujetos al cierre automático). Utilizando el ejemplo anterior quedaría:

```
try(ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("archivo.dat"))){
    … //sentencias que manipulan el flujo
}catch(IOExcepction e){
    System.out.println(e.getMessage());
}
```

Podemos ver que la declaración y construcción del flujo se coloca entre paréntesis, tras la palabra clave ***try*** y se prescinde tanto del método ***close()***, como del bloque ***finally.***

6.- EJERCICIOS DE FICHEROS BINARIOS

6.1.-EJERCICIOS A

1. Pedir un *double* por consola y guardarlo en un archivo binario.
2. Abrir el archivo del ejercicio anterior, leer el *double* y mostrarlo por pantalla.
3. Pedir números enteros positivos por consola, y guardarlos en un fichero binario hasta que se introduzca un número negativo. Leer del fichero todos los enteros guardados y mostrarlos por pantalla.
4. Pedir un entero *n* por consola. A continuación, pedir *n* números *double*, que iremos guardando en un *array*. Guardar el *array* en un archivo binario.
5. Leer de un fichero binario un *array* de números *double*. Mostrar el contenido del *array* por consola.
6. En un fichero binario, sabemos que se ha guardado una serie de números *double*, pero no

sabemos cuántos. Implementar un programa que los lea todos y los muestre por pantalla.

7. Escribir por teclado una frase y guardarla en un archivo binario. A continuación, recuperarla del archivo y mostrarla por pantalla.

8. Escribir un texto, línea a línea, de forma que cada vez que se pulse **Intro**, se guarde la línea en un archivo binario. El proceso se termina cuando introduzcamos una línea vacía. Leer el texto completo del archivo y mostrarlo por pantalla.

9. Crear un *array* de 10 números enteros aleatorios menores que 100, ordenados de menor a mayor y guardarlos en un fichero binario. A continuación, recuperarlos y mostrarlos por consola.

10. Por motivos puramente estadísticos se desea llevar constancia del número de llamadas recibidas en una oficina. Para ello, al terminar cada jornada laboral se guarda dicho número al final de un archivo binario. Implementar una aplicación con un menú, que nos permita añadir el número correspondiente cada día y ver la lista completa en cualquier momento.

11. Disponemos de dos ficheros binarios que guardan números enteros ordenados de forma creciente (*numeros1.dat* y *numeros2.dat*). Fusionar ambos ficheros en un tercero (*numeros.dat*), de forma que todos los datos sigan ordenados. Para probar el algoritmo se pueden utilizar los ficheros generados en el ejercicio 3, introduciendo números ordenados.

12. En un comercio desean mantener los datos de sus clientes. Implementar una aplicación que permita guardar y recuperar los datos de los clientes. Para ello, definir la clase **Cliente** que tendrá los siguientes atributos:

    id: identificador de cliente (entero).

    nombre: nombre y apellidos del cliente.

    telefono: número de teléfono del cliente.

    Para realizar las distintas operaciones, la aplicación tendrá el siguiente menú:

    1. Añadir nuevo cliente.
    2. Modificar datos.
    3. Dar de baja cliente.

4. Listar los clientes

La información se guardará en un fichero binario, que se cargará en memoria al iniciar la aplicación y se grabará en disco, actualizada, al salir de la aplicación.

Un posible ejemplo de valores podría ser el siguiente:

id cliente	Nombre	Teléfono
4	Pepe Pérez	12345
2	José Sánchez	54321
5	Ana Gómez	11223

7.- PERSISTENCIA Y SERIALIZACIÓN (*MARSHALLING*)

Como hemos dicho anteriormente, la **serialización** consiste en transformar un objeto en una secuencia o serie de ***bytes*** de tal manera que represente el estado de dicho objeto. Una vez tenemos serializado un objeto, se puede enviar a un fichero.

La **persistencia** se consigue al tener el objeto seriado y almacenado en un fichero, porque sería posible recomponer el objeto. El estado de un objeto es básicamente el estado de cada uno de los campos. Imaginemos que un campo es a su vez otro objeto, en ese caso debería de ser serializado para serializar el primer objeto.

Hemos visto que para que un objeto de una clase sea serializado, la clase debe implementar la interfaz ***java.io.Serializable***. Dicha interfaz no define ningún método, el objetivo es marcar las clases que vamos a convertir en secuencias de bytes.

```
public class Amigo implements Serializable {
//atributos y métodos de la clase
}
```

El objeto Amigo se ha marcado como serializable, ahora Java se encargará de realizar la serialización de forma automática

Es posible no serializar algunos de los atributos de un objeto. Esto se realiza utilizando el modificador ***transient***:

protected transient int dato;

En este caso, el campo **dato** no interesa que sea persistente.

La clase ***ObjectInputStream*** y la clase ***ObjectOutputStream*** se encargan de realizar estos procesos. Son las encargadas de escribir o leer el objeto de un archivo. Son herederas de ***InputStream*** y ***OutputStream***. Lo métodos ***readObject*** y ***writeObject*** son los que permiten grabar directamente objetos.

7.1.-EJEMPLOS

7.1.1.- Lectura de un archivo que guarda coches.

```
try {
            FileInputStream fis   = new FileInputStream("nuevo.out");
            ObjectInputStream ois = new ObjectInputStream(fis);

            Coche c;
            boolean finalArchivo = false;

            while (!finalArchivo) {
                        c = (Coche) ois.readObject();
                        System.out.println( c );
            }

} catch (EOFException e) {
            System.out.println("Se alcanzó el final.");
} catch (ClassNotFoundException e) {
            System.out.println("Error, el tipo de objeto no es compatible.");
} catch (FileNotFoundException e) {
            System.out.println("No se encontró el archivo.");
} catch (IOException e) {
            System.out.println("Error " + e.getMessage());
            e.printStackTrace();
}
```

Los métodos ***readObject*** y ***writeObject*** usan objetos de tipo ***Object***, readObject los devuelve y writeObject los recibe como parámetro. Ambos métodos lanzan excepciones del tipo ***IOException*** y ***readObject*** además lanza excepciones del tipo ***ClassNotFoundException***.

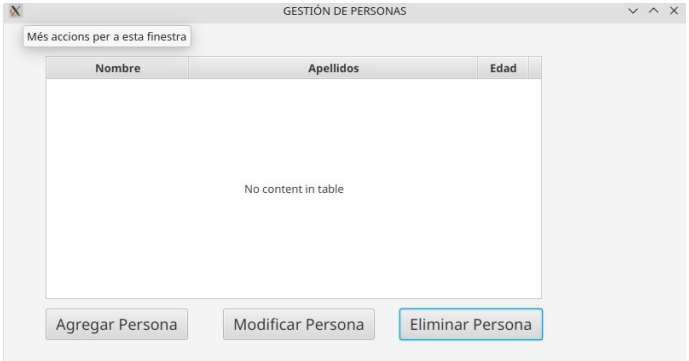
Obsérvese en el ejemplo como la excepción ***EOFException*** ocurre cuando se alcaza el final del archivo al igual que ocurre con los flujos binarios de datos.

7.1.2.- Escritura y lectura de objetos en un fichero

```
import java.io.Serializable;

public class Amigo implements Serializable {

            protected String nombre;
            protected long telefono;

            public Amigo(String nombre, long telefono) {
                        this.nombre   = nombre;
                        this.telefono = telefono;
            }

            public void print() {
                        System.out.println(nombre + " -> " + telefono);
            }
}
```

```
import java.io.*;
public class AmigoGuardarFichero {

     public static void main(String[] args) {

          String[] amigos  = {"Paco Pérez", "Pedro Ruiz", "Isaac Nacher", "Ricardo García"};
          long[] telefonos = {666777888, 678999000, 632444111, 654111777};

          try {
             FileOutputStream fs    = new FileOutputStream("amigos.txt");
             ObjectOutputStream oos = new ObjectOutputStream(fs);

             for (int i = 0; i < 4; i++) {
                    Amigo a = new Amigo(amigos[i], telefonos[i]);
                    oos.writeObject(a);
             }
             if (oos != null) {
                    oos.close(); fs.close();
             }
          } catch (IOException e) {
                    e.printStackTrace();
          }
     }
}

//Lectura del fichero
try {
           File f = new File("src/amigos.txt");

           if (f.exists()) {
                     FileInputStream fis   = new FileInputStream(f);
                     ObjectInputStream ois = new ObjectInputStream(fis);

                     while (true) {
                               Amigo a = (Amigo) ois.readObject();
                               a.print();
                     }
           }
           else {
                     System.out.println("El fichero no existe");
           }

} catch (EOFException e) {
           System.out.println("Se alcanzó el final.");
} catch (ClassNotFoundException e) {
           System.out.println("Error, el tipo de objeto no es compatible.");
} catch (IOException e) {
           System.out.println("Error " + e.getMessage());
}
```

7.2.-AMPLIA EL SIGUIENTE EJERCICIO PARA QUE SE ALMACENEN LAS PERSONAS AGREGADAS EN UN ARCHIVO Y SE RECUPEREN UNA VEZ EJECUTADO EL PROGRAMA.



7.3.- INSERTAR AL FINAL DE UN BINARIO DE OBJETOS

Hay que tener cuidado si queremos “añadir” (insertar) objetos en un fichero binario de objetos ya existente, con objetos **serializados**, podemos tener un problema.

Cuando se escribe un objeto en un fichero binario, con ***ObjectOutputStream***, esta clase crea una cabecera al principio del fichero, identificando varios datos propios. Cuando cerramos el fichero, y luego nuevamente volvemos a añadir un objeto, añade nuevamente otra cabecera, con lo que el fichero resultante tiene dos cabeceras, y al volver a abrirlo posteriormente, da un error por no entender la estructura.

Podemos hacer que, al añadir, no se cree esta segunda cabecera. Para ello vamos a crear nuestra propia versión de la clase ***ObjectOutputStream***, sobreescribiendo el método que crea esta cabecera. Veamos como sería:

```
//Clase propia que va a heredar de ObjectOutputStream y por lo tanto hace lo mismo que esta, menos el método que sobrescribimos
public MiObjectOutputStream extends ObjectOutputStream {
    // tendremos un atributo modo que nos indique si escribirmos desde inicio o insertamos
    int modo;
    public final static int DESDE_INICIO=0;
    public final static int INSERTAR=1;

    //Se sobrescribie el método que crea la cabecera
    @Override
    protected void writeStreamHeader() throws IOException {
        if(modo==DESDE_INICIO){
            // Si es DESDE_INICIO, se llama al método que escribe la cabecera
            super.writeStreamHeader();
        }
        // Si no es DESDE_INICIO no se hace nada, luego no escribe nueva cabecera
    }

    //Llamada al constructor normal de ObjectOutputStream
    public MiObjectOutputStream(OutputStream out, int modo) throws IOException{
        super(out);
        this.modo=modo;
    }
}
```

Ejemplo de uso de nuestra clase:

```
FileOutputStream fos1 = new FileOutputStream("ejem.dat");
MiObjectOutputStream moos1 = new MiObjectOutputStream(fos1,MiObjectOutputStream.DESDE_INICIO);
moos1.writeObject(objetoPersona1);
moos1.writeObject(objetoPersona1);
moos1.close();
// Ahora en el segundo paso creamos el FileOutputStream con segundo parámetro true, para insertar al final
FileOutputStream fos2 = new FileOutputStream("ejem.dat",true);
MiObjectOutputStream moos2 = new MiObjectOutputStream(fos2,MiObjectOutputStream.INSERTAR);
moos2.writeObject(objetoPersona3);
moos2.close();
```

7.4.-USANDO SERIALVERSIONID

El ***serialVersionUID*** es un número de versión que se aplica a cada clase Serializable. La clase usa este número en la deserialización, para verificar que el emisor y el receptor de un objeto serializado mantienen una compatibilidad en lo que a serialización se refiere con respecto a la clases que cada uno usó en el proceso.

Todo eso significa, que se verifica que una clase Persona que se serializó en fichero, cuando se deserializa, se hace para usar una clase Persona que es la misma que se usó para serializar, y no otra tercera que no tenga nada que ver.

No es obligatorio usar un ***serialVersionUID***. Si la clase no especifica un ***serialVersionUID***, durante el proceso de serialización se calculará un ***serialVersionUID*** por defecto, basándose en varios aspectos de la clase.

Es muy recomendable que se declare un ***serialVersionUID*** en las clases serializables, ya que el cálculo del ***serialVersionUID*** es muy sensible a detalles de la clase, que pueden variar con el tiempo incluso entre compiladores.

Si el proceso de comprobación de serialización falla (tanto si es con ***serialVersionUID*** por defecto o explícito) se produce una ***InvalidClassException*** durante el proceso de deserialización. Por lo tanto, es altamente recomendable declarar un valor explícito del ***serialVersionUID***. Éste es de tipo ***long***, y también es aconsejable que sea privado para que afecte únicamente a la clase que lo ha declarado y no a las clases hijas (subclases) que heredan de ella, forzando de alguna manera a cada hija a declarar su propia ***serialVersionUID***.

Aquí tenéis un ejemplo:

```
class Figura implements Serializable {
  private static final long serialVersionUID = 344234342234L;
  .....
```

8.- TRATAMIENTO DE DOCUMENTOS XML

8.1.-INTRODUCCIÓN

La serialización (*marshalling*) consiste en un proceso de codificación de un objeto en un medio de almacenamiento (como puede ser un archivo, o un búffer de memoria) con el fin de transmitirlo a través de una conexión en red como una serie de bytes o en un formato humanamente más legible como XML o JSON, entre otros. La serialización es un mecanismo ampliamente usado para transportar objetos a través de una red, para hacer persistente un objeto en un archivo o base de datos, o para distribuir objetos idénticos a varias aplicaciones o localizaciones.

8.2.-FICHEROS XML CON DOM

El Modelo de objetos de documento (DOM - *Document Object Model*) es una recomendación

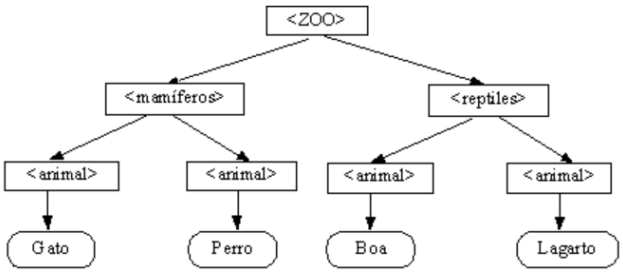
oficial del *World Wide Web Consortium* (W3C). Define una interfaz que permite a los programas acceder y actualizar el estilo, la estructura y el contenido de los documentos XML, independientemente del lenguaje. Los analizadores XML (XML parsers) que admiten DOM implementan esta interfaz. Por lo tanto, DOM es un estándar de la WWW, que define cómo representar documentos HTML, XHTML y XML. Además define qué tipo de etiquetas y atributos puede haber, en qué orden y con que estructura

Java posee clases para trabajar con ficheros XML o HTML tratándolos como objetos DOM. La mecánica es leer el fichero HTML o XML y almacenarlo en un objeto *Document*, y procesar éste, o escribir un objeto Document en un fichero XML/HTML.

8.3.-VENTAJAS

Cuando se analiza un documento XML con un analizador DOM, se recupera una estructura de árbol que contiene todos los elementos de su documento. El DOM proporciona una variedad de funciones que se pueden usar para examinar el contenido y la estructura del documento. Por ejemplo, a continuación podemos ver un documento XML y el modelo de una implementación de DOM:

```
<zoo>
   <mamiferos>
      <animal> Gato </animal>
      <animal> Perro </animal>
   </mamiferos>
   <reptiles>
      <animal> Boa </animal>
      <animal> Lagarto </animal>
   </reptiles>
</zoo>
```



El DOM es una interfaz común para manipular estructuras de documentos. Uno de sus objetivos de diseño es que el código Java escrito para un analizador compatible con DOM debe ejecutarse en cualquier otro analizador compatible con DOM sin tener que hacer ninguna modificación.

8.4.-DOM INTERFACES

El DOM define varias interfaces de Java. Aquí están las interfaces más comunes:

- Node → Es **cualquier objeto** que haya en un documento DOM. A su vez, hay varios tipos de Node (Element, textos de nodo, comentarios, el propio Document entero, todos ellos son ante todo Nodes). Por lo tanto, es el tipo de datos base del DOM.

- Element → Es un **tipo específico de Node**. Casi todas las etiquetas de un XML son

Elements, y por lo tanto, también Node, lo que a veces lleva a confundir los objetos. Se pueden usar indistintamente, pero cada clase tiene sus métodos y puede hacer sus cosas. De hecho, se puede hacer un casting entre una clase y otra para convertir un objeto de uno en otro. Por ejemplo:

Element unElement = (Element) unNodo;

Node otroNodo = (Node) otroElement;

La gran mayoría de los objetos con los que tratarás son Elementos.

- Attr → Representa un atributo de un elemento.
- Text → Representa el contenido real de un Elemento o Atributo.
- Document → Un documento DOM (*Document*) consiste en una jerarquía de nodos en modo de árbol, en el que cada nodo tiene hermanos, un padre, o hijos. Java proporciona esta clase donde se puede ir añadiendo nodos y estructurarlos como el árbol que deseemos. Por lo tanto, representa todo el documento XML. Un objeto de documento a menudo se conoce como un árbol DOM.
- NodeList → Es un conjunto de nodos, que pueden extraerse con métodos que devuelven cosas como “hijos de X”, “todos los de nombre X” , etc.

8.5.-CLASES Y MÉTODOS BÁSICOS PARA TRABAJAR CON DOM.

Métodos de uso en DOM (para crear XML)

CLASE	DEVUELVE	METODOS	DESCRIPCION
Document	Node	getDocumentElement()	Devuelve el nodo raiz de un documento `nivel1 = miDocDom.createElement("personal");`
Document	Node	createElement("texto")	Crea un nodo cuyo nombre es el texto del parametro `nivel11 = miDocDom.createElement("empleado");`
Node	void	setAttribute("texto", "contenido")	Crea, en el Element aplicado, un atributo de nombre "texto" con valor "contenido" `((Element) unnodo).setAttribute("codigoEmp", "2323");`
Document	Node	createTextNode("texto")	Crea un nodo de texto (el valor de una etiqueta) `nodotexto = miDocDom.createTextNode("Pedro");`
Node	void	appendChild(otronodo)	Añade otroNodo como un nodo hijo del nodo llamante `unnodo.appendChild(sunodohijo);`

Métodos de uso en DOM (para leer XML)

CLASE	DEVUELVE	METODOS	DESCRIPCION
Document	Node	getDocumentElement()	Devuelve el nodo raiz de un documento `nivel1 = miDocDom.createElement("personal");`
Node	NodeList	getElementsByTagName("texto")	Devuelve una lista con los nodos que tienen por nombre "texto" `NodeList listaDeNodosDeEmpleados = nodoRaiz.getElementsByTagName("empleado");`
NodeList	int	getLength()	Devuelve el tamaño de una NodeList `int m listaNodosHijosDeEmpleado.getLength();`
Node	NodeList	getChildNodes()	Devuelve una lista con los nodos hijos del nodo al que se aplica el metodo `NodeList listaNodosHijos = nodoEmpleado.getChildNodes();`
NodeList	Node	item(num)	Devuelve el nodo num-esimo de la lista de nodos a la que se aplica el metodo `Node dato = listaNodos.item(m);`
Node	String	getNodeName()	Devuelve el nombre del nodo al que se aplica el metodo `if (dato.getNodeName().equals("nombre"))`
Node	Node	getFirstChild()	Devuelve el primer hijo del nodo al que se aplica el metodo `Node datoContenido = dato.getFirstChild();`
Node	String	getNodeValue()	Devuelve el valor (el contenido) del nodo al que se aplica el metodo `String atrib= nodo.getAttributes().getNamedItem("cod").getNodeValue()`
Node	NamedNodeMap	getAttributes()	Devuelve todos los atributos de nodo al que se aplica el metodo `NamedNodeMap mapanodo = nodoEmpleado.getAttributes()`
NamedNodeMap	Node	getNamedItem(texto)	Devuelve el elemento llamado "texto" de los que existen en la colección `Node unnodo = nodoEmpleado.getAttributes().getNamedItem("codDptoPertenecia")`

8.6.- EJEMPLOS

8.6.1.- FicheroXML, con una lista simple de objetos de la clase Alumno

1. Primero creamos la clase **Alumno**, que es una clase POJO (***Plain Old Java Objects***) normal (POJO es una clase que solo tiene atributos, constructor completo, getters y setters). Es obligatorio que tenga un constructor vacío. Sus atributos privados serán:

   private String nombre;
   private String apellidos;
   private int edad;
   private String curso;
   private char sexo;

   - Crear un constructor con todos los atributos.
   - Crear un constructor vacío.
   - Sobreescribir el método toString.
   - Generar todos los setter y getters

2. El documento XML que vamos a escribir con DOM es el siguiente:

Este es el XML que vamos a escribir con DOM

```
<Alumnos>
    <alumno edad="20">
        <nombre nacionalidad="española">Paco</nombre>
        <apellidos>Gomez</apellidos>
        <curso>Musica</curso>
        <sexo>M</sexo>
    </alumno>
    <alumno edad="30">
        <nombre nacionalidad="española">Maria</nombre>
        <apellidos>Castillo</apellidos>
        <curso>Pintura</curso>
        <sexo>F</sexo>
    </alumno>
    <alumno edad="25">
        <nombre nacionalidad="española">Alejandro</nombre>
        <apellidos>Martin</apellidos>
        <curso>Ajedrez</curso>
        <sexo>M</sexo>
    </alumno>
    <alumno edad="50">
        <nombre nacionalidad="española">Lisa</nombre>
        <apellidos>Simpson</apellidos>
        <curso>Musica</curso>
        <sexo>F</sexo>
    </alumno>
</Alumnos>
```

3. A continuación, creamos la **clase Main**, donde construimos una lista con la información que deseamos guardar en el fichero XML.

4. Por último creamos un método que reciba una lista y la escriba en un fichero XML.

8.6.2.- Ejemplo DOM con un XML más complejo

1. A partir de lo que has hecho en el ejercicio anterior, crea un archivo XML partiendo de una lista simple de objetos de la clase Libro, los cuales tienen o pueden tener varios objetos de la clase Autor.

libros con más de un autor....

```
<libros>
        <libro año="1994">
                <titulo>TCP/IP Illustrated</titulo>
                <autor>
                        <apellido>Stevens</apellido>
                        <nombre>W.</nombre>
                </autor>
                <editorial>Addison-Wesley</editorial>
                <precio> 65.95</precio>
        </libro>
        <libro año="1992">
                <titulo>Advan Programming for Unix environment</titulo>
                <autor>
                        <apellido>Stevens</apellido>
                        <nombre>W.</nombre>
                </autor>
                <editorial>Addison-Wesley</editorial>
                <precio>65.95</precio>
        </libro>
        <libro año="2000">
                <titulo>Data on the Web</titulo>
                <autor>
                        <apellido>Abiteboul</apellido>
                        <nombre>Serge</nombre>
                </autor>
                <autor>
                        <apellido>Buneman</apellido>
                        <nombre>Peter</nombre>
                </autor>
                <autor>
                        <apellido>Suciu</apellido>
                        <nombre>Dan</nombre>
                </autor>
                <editorial>Morgan Kaufmann editorials</editorial>
                <precio>39.95</precio>
        </libro>
</libros>
```