2. Uso de ficheros

Hasta este momento, las estructuras de datos utilizadas para el almacenamiento de información residen en memoria, lo que supone algunas limitaciones:

- Los datos no se almacenan de forma permanente. Una vez que se sale del programa los datos se pierden.
- La cantidad de los datos no puede ser muy grande debido a la limitación del tamaño de la memoria.

Para evitar estos inconvenientes, existen unas estructuras que utilizan la memoria secundaria para el almacenamiento de la información, estas estructuras se denominan *ficheros* y se identifican bajo un nombre y una extensión.

2.1. Ficheros de datos

Podemos definir los ficheros de datos como un conjunto de datos, sobre un mesmo tema, tratado como una unidad de almacenamiento y organizado de forma estructurada para la obtención de un dato individual.

Podemos considerar al fichero como una estructura dinámica ya que su tamaño puede variar durante la ejecución del programa.

Como ya se comentó, la información está almacenada siguiendo una determinada estructura. A esta estructura se le denomina *registro*.

Podemos definir un registro como un *conjunto de datos que forman una unidad de información y que se manipulan como un bloque*. Cada registro está compuesto de unidades elementales de información más pequeñas que se denominan *campos*.

2.2. Tipos de acceso

Como el fichero está almacenado en memoria secundaria no es posible acceder a toda su información de forma inmediata, sino que de cada vez se puede acceder a un registro del fichero.

Dependiendo de cómo se acceda a cada uno de estos registros podemos distinguir dos tipos de acceso:

- *Acceso secuencial*: para leer el fichero debe comenzarse siempre desde el primer registro, después la lectura avanza hacia adelante leyendo los registros de uno en uno.
- *Acceso aleatorio*: podemos acceder directamente a cualquier posición del fichero, y por lo tanto, leer cualquier registro.

2.3. Tipos de ficheros

Existen varias clasificaciones de los ficheros, la clasificación por tipo de fichero se basa en la *forma en la que se guarda la información* dentro del fichero:

- *Ficheros de texto*: la información se almacenas en formato de caracteres, tal y como se vería por pantalla.
- *Ficheros binarios*: la información se guarda en el fichero tal y como está en memoria principal, por lo que estará compuesta por unos y ceros.

Otra clasificación *que se puede hacer* sobre os ficheros es la que distingue entre:

- *Ficheros físicos*: es el fichero que reconoce el sistema operativo y que está almacenado en memoria secundaria bajo un nombre y una extensión.
- *Ficheros lógicos*: el programa debe poder hacer referencia al fichero físico para trabajar con él. De esta manera los lenguajes de programación proporcionan mecanismos para poder crear un tipo de dato de tipo fichero. En Java, se queremos crear una variable que apunta a un fichero físico debemos utilizar a clase *File*. De esta forma el fichero lógico será un objeto de la clase *File*.

También podemos clasificar los ficheros según la *forma en la que se abren* para proceder a su manipulación:

- *Ficheros de lectura*: son aquellos que se abren y sobre los que solo se puede leer la información que contienen. No se puede modificar.
- *Ficheros de escritura*: son los ficheros en los que se puede leer y modificar su contenido.

2.4. Operaciones sobre ficheros

Las operaciones que se deben realizar sobre un fichero para poder trabajar con el son las siguientes:

- *Apertura del fichero*: consiste en crear la asociación entre el fichero lógico y el fichero físico. Para realizar esta operación, debemos conocer, además del nombre y la extensión del fichero, la ruta en memoria secundaria donde está almacenado. En este momento es cuando se decide si el fichero se va a abrir solo para lectura o también para escritura.
- *Operaciones de manipulación*: según la forma en la que se abra el fichero se pueden realizar las siguientes operaciones:
    - Lectura do fichero: permite recuperar la información guardada en el fichero y ponerla en memoria para poder trabajar con ella. La forma de leer el fichero dependerá del tipo en el que esté guardada la información, que puede ser de tipo texto o de tipo binaria.
    - Escritura en fichero: esta operación permite guardar la información que está en la memoria en el fichero, para poder utilizarla posteriormente.
- *Cierre del fichero*: cuando se acabe de realizar todas las operaciones necesarias sobre el fichero, éste debe *cerrarse siempre*. En el caso de Java, el cierre permite liberar el flujo de datos creado en la apertura del fichero.

2.5. Clase File

La clase *File*, al igual que otras clases relacionadas con la entrada y salida de información en Java, se encuentra en el paquete java.io. Esta clase permite realizar operaciones sobre el fichero a nivel del sistema de archivos: tal como listar los archivos, crear directorios, borrar ficheros, modificar su nombre, etc.

Los sistemas operativos utilizan un nombre de ruta para hacer referencia a los archivos y directorios que dependen del propio sistema operativo. Esta clase permite obtener una representación abstracta de la ruta que es independente del sistema.

Este nombre de ruta puede ser de dos tipos:

- *Ruta absoluta*: en este caso la ruta contiene la información completa para poder acceder al archivo. Podemos decir que la ruta absoluta está formada por un prefijo, un listado de directorios y el nombre del directorio o archivo al que queremos acceder.
    - En Windows, el prefijo está compuesto por el nombre de la unidad, seguido del carácter ":" seguido por "/" o "\\". El listado de directorios para llegar al destino se separará también con "/" o "\\".

```
C:\carpeta\subcarpeta\fichero.txt
```

    - Para sistemas Unix, el prefijo está composto siempre por "/" que representa la raíz del sistema de directorios. El listado de directorios también se separan con "/".

```
/home/usuario/carpeta/subcarpeta /fichero.txt
```

- *Ruta relativa*: en este caso se indicará el nombre del archivo o directorio al que queremos acceder. Usando la ruta relativa se busca dentro directorio actual, que es el directorio del archivo Java en el que se está trabajando.

Para crear un objeto de tipo archivo a partir del sistema de ficheros del sistema operativo, la clase *File* proporciona 3 constructores:

- Un solo constructor con un argumento de tipo cadena que representa la ruta al sistema de archivos.

```
//Ruta absoluta Windows
File archivo = new File("C:/carpeta/subcarpeta/fichero.txt");

//Ruta absoluta Unix
File archivo = new File("/home/usuario/carpeta/fichero.txt");

//Ruta relativa
File archivo = new File("fichero.txt");
```

- Dos constructores:
    - El primero argumento representa a la ruta padre, puede ser de tipo *File* o una cadena de caracteres.

- El segundo argumento es de tipo cadena y representa una ruta hija, que a partir de la ruta padre localiza el archivo.

```
//Ruta padre
File directorio = new File("C:/carpeta");

//Ruta padre + ruta hija
File archivo = new File(directorio,"/subcarpeta/fichero.txt");
```

Cuando se crea un archivo hay que tener en cuenta que no se devuelve una excepción si el archivo no existe. La creación de un objeto de tipo archivo sólo devuelve una excepción cuando la ruta o la ruta hija son nulas (NullPointerException).

Dentro de la clase *File*, contamos con el método *exists()*, esta función determina si existe o no un archivo o directorio denotado por el nombre del archivo abstracto. La función devuelve *true* si la ruta del archivo abstracto existe o devuelve *false*.

La *Máquina Virtual de Java* permítenos ejecutar aplicaciones Java independientemente de la plataforma desde la que se realiza a ejecución. Esta versatilidad de Java la debemos tener en cuenta cuando especificamos la ruta del archivo y no deberíamos utilizar nombres de ruta que dependan directamente del sistema operativo a utilizar.

Para evitar este problema debemos usar el separador "/" que es válido tanto para Windows como para Linux, también podemos utilizar las variables estáticas que tiene *File*.

separatorChar, que representa al carácter (**char**) separador de nombres de archivos y carpetas.

separator, lo mismo que el anterior, pero en forma de **String**

```
String ruta="carpeta\subcarpeta\fichero.txt";
ruta=ruta.replace("/", File.separator);
```

Dentro de la clase File podemos encontrar:

- *Métodos generales*: proporcionan información general sobre el objeto. Permite saber, entre otras cosas, si el fichero existe, si se puede leer, si está oculto o si dos ficheros son iguales.

- *Métodos de directorio*: permiten hacer operaciones sobre un directorio, crear un directorio u obtener la lista dos ficheros que contiene.
- *Métodos de archivo*: permiten hacer operaciones sobre un fichero. Entre otros métodos, hay métodos para renombrar un fichero, para borrarlo o para saber su longitud.

  - mkdir vs mkdirs – mkdir crea una carpeta, mkdirs crea toda la ruta.
  - delete – borra la carpeta (y la ruta entera) si no hay archivos
  - getName()
  - getPath()
  - getAbsolutePath()
  - exists()
  - canRead
  - isFile()
  - isDirectory()
  - isAbsolute()
  - lastModified()
  - length()
  - renameTo(File dest);
  - list() – Crea un array de String con nombres de contenido
  - listFiles() – Crea un array de Objetos (carpetas y archivos)
  - list(FilenameFilter filter)

2.6. Manipulación de ficheros de texto

La manipulación de ficheros de texto puede hacerse desde las clases de flujo de bytes *FileInputStream* o *FileOutputStream*.

Sin embargo, cuando se trata de trabajar específicamente con archivos de texto es mejor transformar los flujos anteriores a *Reader* o *Writer*, según corresponda, mediante las clases *InputStreamReader* y *OutputStreamReader*. También se puede trabajar directamente con las clases *FileReader* y *FileWriter*, para hacer la lectura o la escritura del fichero de texto respectivamente. Utilizaremos esta última opción.

Apertura del fichero de texto

La primera operación que tenemos que realizar es la de apertura del fichero. Como ya hemos mencionado, aquí es donde decidimos si queremos abrir el fichero solo para lectura o también para escritura.

Para abrir el fichero solo para lectura usaremos la clase *FileReader* que dispone de los siguientes constructores:

```
                    // FileReader(File archivo);
File archivo = new File ("C:\\archivo.txt");
FileReader fr = new FileReader (archivo);
                    // FileReader(String nomeArchivo);
FileReader fr = new FileReader("datos.dat")
```

Si quisiéramos abrir el fichero también para escritura, deberemos utilizar la clase *FileWriter*. Los constructores disponibles son los mesmos que en la clase *FileReader*, y permiten abrir el fichero directamente para escritura.

Esta clase además incorpora otros dos constructores con un parámetro extra de tipo booleano, que en caso de valer *true*, indica que se abre el archivo para añadir datos, en caso contrario, se abrirá para grabar desde cero con lo que borrará el contenido previo que tuviese el archivo.

```
                    // FileWrite(File archivo);
File archivo = new File ("C:\\archivo.txt");
FileWriter fr = new FileWriter (archivo);          Borrando el contenido previo
FileWriter fr = new FileWriter (archivo, true);    Sin borrando el contenido

                    // FileWrite(String nomeArchivo);
FileWriter fr = new FileWriter ("datos.dat");      Borrando el contenido previo
FileWriter fr = new FileWriter ("datos.dat", true); Sin borrando el contenido
```

Cuando se abre el fichero un cursor apunta al inicio del fichero, indicando la siguiente información del fichero a la que se puede acceder.

Si el fichero no existe, cuando es un directorio en lugar de un fichero o por cualquier otra razón que impida abrir el fichero, se produce una excepción, que puede ser:

- *FileNotFoundException* cando se usan los métodos de *FileReader*.
- *IOException* cando se usan los métodos de *FileWriter*.

Peche do fichero de texto

Una vez que se acabe de utilizar el fichero debe usarse o método *close*, que cierra el flujo y además libera cualquier recurso que sea utilizado por el flujo. Es aconsejable incluír este método dentro del bloque *finally* de forma que siempre se ejecute.

```
try {
        FileReader fichero = new FileReader("fichero.txt")
        //instruciones
} catch (IOException e) {
        //instruciones
} finally {
        fichero.close();
}
```

Desde la *versión 7 de Java* puede utilizarse también la instrucción *try-with-resources*. Esta sentencia declara uno o más recursos, considerando como tal un objeto que debe ser cerrado después de que el programa acabe de usarlo. De esta forma, nos podemos asegurar de que cada recurso será liberado una vez que finalice la sentencia.

Una vez que se ejecuten todas las instrucciones dentro del bloque *try* se cerrará el flujo declarado a su comienzo.

```
FileReader fichero = null;
try (fichero = new FileReader("datos.dat")) {
        //instruciones
} catch (IOException e) {
        //instruciones
}
```

Lectura de ficheros de texto

Para realizar la lectura se puede utilizar el método *read()* que puede recibir un array de caracteres donde se almacenarán los caracteres leídos. Este método devuelve un valor de tipo entero que indica el número de caracteres leídos, o bien tendrá valor -1 para indicar que se llegó al final de la lectura.

Este método es un poco rudimentario por lo que lo ideal es convertir el flujo en una clase de tipo *BufferedReader*. Esta clase proporciona los siguientes métodos de lectura:

int read()

Lee un carácter de forma individual y lo devuelve como un entero que estará dentro del rango 0...65535. Devolverá -1 cuando llegue al final del flujo.

int read(char[] buffer, int off, int lonxitude)

Almacena en un array de caracteres lo que se lee. Se debe indicar una posición desde la que comenzará la lectura y un número máximo de caracteres a leer. Devuelve un entero que será el número de caracteres leídos o -1 cuando llegue al final del flujo. Devolverá *null* cuando se alcance el final del flujo.

String readLine()

Lee una línea completa de texto. Se considerará que la línea acabó cuando se encuentra un salto de línea '\n', un retorno de carro '\r' o un retorno de carro seguido inmediatamente por un salto de línea.

Estos métodos lanzan una excepción de tipo *IOException*, que se deberá tener en cuenta a la hora de utilizarlos.

Cada vez que se ejecuta el método *read()* el cursor avanza a la siguiente posición. En caso de estar leyendo de carácter en carácter el cursor avanzará al siguiente carácter, si se lee la línea completa entonces el cursor avanzará a la siguiente línea.

```
try {
    String cadena;
    FileReader f = new FileReader("d:/fichero.txt");
    BufferedReader b = new BufferedReader(f);
    while ((cadena = b.readLine()) != null) {
        System.out.println(cadena);
    }
    b.close();
} catch (Exception e) {
```

Escritura en ficheros de texto

Para poder escribir en el fichero, previamente tendremos que abrirlo para escritura como se vio anteriormente. Para escribir en el fichero podemos utilizar el método *write()*, que permite escribir tanto un único carácter, como un String o un array de bytes.

Al igual que ocurría en la lectura de ficheros, si utilizamos un buffer para escribir en ficheros, con la clase *BufferedWriter*, podemos utilizar además el método *newLine()* que va a escribir un salto de línea en el fichero. Con el uso de este método evitamos el problema de la compatibilidad entre plataformas, ya que los caracteres de cambio de párrafo son distintos según cada sistema operativo.

```
try {
    FileWriter fichero = new FileWriter("d:/fichero.txt");
    fichero.write("grabado línea 1" + "\n");
    fichero.write("grabado línea 2" + "\n");
    fichero.write("grabado línea 3" + "\n");
    fichero.close();
} catch (Exception e) {
```

2.7. Manipulación de ficheros binarios

Un fichero binario está formado por secuencias de bytes. Dependendo de si conocemos la estructura interna del fichero podemos leer su contenido de una forma o de otra. Si no conocemos la estructura entonces debemos leer el fichero byte a byte. En caso de conocer la estructura, de saber el tipo (int, float, char, etc) y el orden de los datos podemos utilizar métodos más específicos para la lectura de cada tipo de dato.

Apertura del fichero binario

La clase que nos permite abrir un fichero binario para su *lectura* es *FileInputStream*

```
FileInputStream fis = new FileInputStream ("C:\\archivo.dat");
```

La clase que nos permite abrir un fichero binario para su *escritura* es *FileOutputStream*

```
FileOutputStream fos = new FileOutputStream ("C:\\archivo.dat");
FileOutputStream fos = new FileOutputStream ("C:\\archivo.dat", true);
```

Los constructores mencionados apartado lanzan una excepción *FileNotFoundException* si el fichero no existe o si no se pudo crear el fichero en caso de abrirlo para escritura.

Cierre del fichero binario

Al igual que ocurre con los ficheros de texto, los flujos de datos binarios deben ser cerrados para liberar cualquier recurso que se estuviese utilizando. Usaremos el mesmo método que en el caso de ficheros de texto, *close()*.

Se hai un error de entrada/saída, este método lanza una excepción de tipo *IOException*.

Se puede incluir la llamada al método *close()* dentro del bloque *finally*. También se puede usar declarando el flujo con una instrucción *try-with-resources*.

Escritura en ficheros binarios

La clase utilizada para abrir un flujo de bytes en modo escritura, *FileOutputStream*, proporciona el método *write()* para escribir bytes en el fichero. Puede lanzar excepciones de tipo *IOException*.

```
FileOutputStream fos = new FileOutputStream ("C:\\archivo.dat");
fos.write(datos):
```

Si queremos escribir en el fichero datos de tipo primitivo, como por ejemplo datos de tipo *int*, podemos crear un objeto *DataOutputStream* a partir del objeto *FileOutputStream*, ya que esta clase proporciona métodos específicos para la escritura de datos de tipo primitivo en el fichero.

Lectura de ficheros binarios

La clase *FileInputStream*, proporciona el método *read()* para leer bytes desde el fichero. Cuando usemos este método hay que tener en cuenta que lanza una excepción de tipo *IOException*.

Si queremos leer datos de tipo primitivo podemos crear un objeto *DataInputStream* que proporciona métodos específicos para a lectura de datos de tipo primitivo.

```
try(FileOutputStream fos=new FileOutputStream("D:\\fichero_bin.ddr")){
    String texto="Esto es una prueba para ficheros binariosssss";
    byte codigos[]=texto.getBytes(); //Copia  texto en array de bytes
    fos.write(codigos);
}catch(IOException e){
    System.out.println("Fichero inaccesible " + e.getMessage());
}
```

```java
try(FileInputStream fis=new FileInputStream("D:\\fichero_bin.ddr")){
    int valor=fis.read();
    while(valor!=-1){
        System.out.print((char)valor);
        valor=fis.read();
    }
}catch(IOException e){
    System.out.println("Fichero inaccesible " + e.getMessage());
}
```

2.8. Ficheros de acceso aleatorio

Los tipos de ficheros vistos hasta ahora son secuenciales, lo que quiere decir que para acceder a un determinado registro debemos recorrer secuencialmente desde el inicio del fichero y pasar por todos los registros hasta encontrar el registro que nos interese.

Los ficheros de acceso aleatorio tienen las siguientes características:

- No se trata de un flujo de datos como en los ficheros de texto o binarios.
- Almacena un conjunto de bytes.
- Permite el acceso aleatorio a un determinado registro, de esta forma no es necesario recorrer los registros anteriores para acceder a un registro concreto.
- Se puede abrir para lectura o para lectura y escritura.
- Tiene un cursor o índice llamado *file pointer* (puntero), que señala hacia un punto del fichero. Inicialmente apunta al comienzo del fichero y se puede mover a otra posición diferente. Las operaciones de lectura comienzan a leer bytes a partir de la posición a la que apunta el puntero en ese instante y lo desplazan tantos bytes como fueron leídos. En el caso de las operaciones de escritura, los bytes se empiezan a escribir a partir de la posición indicada por el puntero y este avanza hasta pasados los bytes que fueron escritos.
- Dispone de operaciones de lectura y escritura basadas en el tipo de dato a leer o a escribir en el fichero.

La clase que permite el uso de ficheros de acceso aleatorio es la clase *RandomAccessFile* que también se encuentra en el paquete java.io. Los constructores de esta clase son:

```
RandomAccessFile(File archivo, String modo);
RandomAccessFile(String nombre, String modo);
```

Los dos constructores lanzan la excepción *FileNotFoundException* que hay que tratar.

La forma en la que indicamos el archivo que queremos abrir en modo de acceso aleatorio lo podemos hacer creando un objeto de tipo *File* o bien indicando la cadena de texto que corresponde con el fichero.

Ambos constructores tienen un parámetro obligatorio que indica el modo en el que se abrirá el fichero. Valores permitidos:

Modo	*Descripción*
"r"	Abre el fichero para solo lectura. Se utilizamos alguno de los métodos de escritura sobre el objeto creado se lanzará una excepción *IOException*.
"rw"	Abre el fichero para lectura y escritura. Si el fichero no existe entonces intentará crearlo.
"rws"	Abre el fichero para lectura y escritura y además requiere que cada actualización del contenido o metadatos del archivo se escriba de forma sincronizada en el dispositivo de almacenamiento subyacente.
"rwd"	Abre el fichero para lectura y escritura y además requiere que cada actualización del contenido del archivo se escriba de forma sincronizada en el dispositivo de almacenamiento subyacente.

Si se utiliza para *lectura* del archivo (modo "r"), dispone de métodos para leer elementos de cualquier tipo primitivo: *readInt()*, *readLong()*, *readDouble()*, *readLine()* y otros. Así mismo, cuando se utiliza el modo de *lectura y escritura* (modo "rw") se puede utilizar los métodos de escritura para escribir los tipos de datos de forma similar a como se pueden leer, con los métodos: *writeInt()*, *writeLong()*, *writeDouble()*, *writeBytes()*, ...

Respecto al funcionamiento de los métodos de escritura, hay que aclarar, que cuando se utiliza un método *write* por el medio del fichero, es decir, cuando la posición del puntero no esté al final del fichero, entonces el método lo que va a hacer es sobrescribir la información ya existente en el archivo, no la añade. Si la posición del puntero está al final del archivo y utilizamos un método de escritura, entonces se escribirá al final del archivo e incrementará su longitud.

Método	*Descripción*
getFilePointer()	Devuelve la posición actual del puntero medida desde el comienzo del archivo, a partir del cual se hará la siguiente lectura o escritura.
length()	Devuelve la longitud del fichero.
seek(long posicion)	Sitúa la posición de la próxima operación de lectura o escritura en el byte especificado.

Todos los métodos de la tabla devuelven una excepción *IOException* si ocurre algún error de entrada/salida.

2.9. Serialización

La serialización consiste en la transformación de un objeto en una secuencia de bytes que posteriormente pueden ser leídos para reconstruir el objeto original. Una vez que tenemos la secuencia de bytes, esta pude ser almacenada en un fichero.

El hecho de guardar un objeto y que siga existiendo más allá de cuando se acaba la ejecución de la aplicación es lo que se conoce como *persistencia*.

Un detalle importante para que un objeto pueda transformarse en una secuencia de bytes es que éste debe ser serializable. Para ello, tiene que implementar la interface de Java *Serializable* que se encuentra en el paquete java.io. Esta interface está vacía y no contiene ningún método.

La clase *Serializable* solo sirve para indicarle a la máquina virtual de Java que el objeto que la implementa se puede serializar.

Todos los tipos básicos de Java, junto con los arrays y los Strings, son serializables.

Escritura de un objeto serializable (a1) en un fichero.

```
FileOutputStream fs = new FileOutputStream("d:\\agenda.ser");  //Creamos el archivo
ObjectOutputStream os = new ObjectOutputStream(fs);            //Creamos el objeto
os.writeObject(a1);              //método writeObject() serializa y escribe el objeto en el archivo
os.close();                      //Hay que cerrar siempre el archivo
```

Lectura del objeto anteriormente serializado.

```
FileInputStream fs = new FileInputStream("d:\\agenda.ser");      //Creamos el archivo
ObjectInputStream ois = new ObjectInputStream(fs);          //Creamos el objeto
a1 = (Agenda) ois.readObject();   //El método readObject() recupera el objeto
ois.close();                              //Hay que cerrar siempre el archivo
```