Tema 1: Manejo de Ficheros

Índice

1. La clase File

En Java la clase File[1] representa, de forma abstracta, un fichero o directorio dentro del sistema de archivos del sistema operativo. Esta clase permite tanto obtener información (nombre, ruta, fecha de creación) como realizar operaciones sobre el mismo (copiar, mover, borrar, renombrar, ...) pero no permite ni leer ni escribir información en él.

Un objeto File puede usar, entre otros, los siguientes constructores:

```
File f1 = new File (String rutaDelFichero/nombreFichero)
File f2 = new File (String rutaFichero, String nombreFichero)
```

Entre otros, los métodos más relevantes del objeto de tipo File son:

- getName: obtiene el nombre del fichero/directorio.
- length: retorna el tamaño del fichero en bytes.
- getPath: devuelve, como cadena, la ruta relativa del fichero/directorio.
- getAbsolutePath: devuelve, como cadena, la ruta absoluta del fichero/directorio.
- exists: comprueba si el fichero o directorio existe.
- mkdir: crea el directorio con el nombre indicado en el objeto File.
- mkdirs: crea el directorio y todos los padres necesarios con el nombre indicado en el objeto File.
- renameTo: renombra el fichero o directorio.
- delete: borra el fichero o directorio.
- isDirectory: comprueba si el objeto File es un directorio.
- isFile: comprueba si el objeto File es un fichero.
- canRead: comprueba si se puede leer del fichero (por ejemplo se tienen los permisos suficientes).
- canWrite: comprueba si se puede escribir en el fichero (si se tienen los permisos suficientes).
- listFiles: devuelve un array de objetos de tipo File, con los ficheros que contiene el directorio.
- list: devuelve un array de Strings, con la ruta de los ficheros que contiene el directorio.
- getParent / getParentFile: se obtiene el directorio padre, o null si este no existe.

[1] https://docs.oracle.com/javase/8/docs/api/java/io/File.html

Ejemplo 1: Obtener información de un fichero

```
File fichero = new File ("c:/textos/fichero.txt");
```

A partir del objeto fichero podemos, usando sus métodos, obtener información del fichero

```
System.out.println ("Nombre: " + fichero.getName());
System.out.println ("¿Existe?: " + fichero.exists());
System.out.println ("Ruta absoluta: " + fichero.getAbsolutePath());
```

El resultado será:

```
Nombre: fichero.txt
Directorio padre: true
Ruta absoluta: c:\textos\ejemplo.txt
```

Actividad 1

Muestra, por separado, la lista de ficheros y directorios de un directorio dado.

2. Flujos de datos

En java cualquier origen (entrada - in) o destino (salida - out) de datos se conoce como flujo: Stream[2]. Pueden representar orígenes y/o destinos de datos de diferente tipo: ficheros, dispositivos externos, sockets, …

Por ejemplo, para trabajar con un fichero no se accede directamente a él sino que se utiliza un flujo como un intermediario entre nuestro programa y el fichero. Son una capa de abstracción intermedia que oculta todos los detalles de cómo trabajan internamente (la clase asociada al flujo se encarga de realizar todas las operaciones necesarias de forma transparente) ofreciéndonos una interfaz, más o menos, común a todos ellos.

Java utiliza los flujos de entrada-salida, sobre ficheros, como un mecanismo para conseguir la persistencia de los datos, que consiste en que los datos son almacenados entre ejecuciones de un programa sin perderse para su uso en posteriores ejecuciones del mismo, usando para ello el paquete java.io[3].

Para la utilización de flujos, en Java, debemos, de forma general, seguir los siguientes pasos:

- Abrir el flujo sobre el fichero deseado invocando su constructor.
- Realizar las operaciones de lectura y/o escritura que necesitemos sobre él.
- Una vez terminadas las operaciones anteriores se ha de cerrar el flujo para liberar recursos

[2] https://docs.oracle.com/javase/tutorial/essential/io/streams.html

[3] https://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html

Los flujos sobre ficheros pueden ser de dos tipos:

- Flujos binarios[4]: se utilizan para leer/escribir datos binarios usando bytes (8 bits). Soporta distintos tipos de datos como pueden ser: bytes, tipos primitivos (int, float, …), objetos, …
- Flujos de caracteres[5]: utilizados leer/escribir caracteres (unicode, 16 bits).

Los ficheros se pueden clasificar en función de:

Contenido	Ficheros de caracteres (16 bits)	Contienen únicamente texto[6].
	Ficheros binarios (8 bits)	Están compuesto de secuencias bytes (que pueden representar distintos tipos de información: texto, imágenes, música, videos, …), este tipo de fichero no es reconocible directamente por humanos.
Modo de acceso	Ficheros secuenciales	El acceso a los datos guardados en el fichero se hace de forma lineal desde el principio del fichero hacia el final.
	Ficheros de acceso aleatorio	Se puede acceder de forma directa a cualquier posición del fichero.

3. Ficheros secuenciales

3.1. Ficheros de caracteres

Tal y como se vio en el punto anterior un fichero de caracteres es aquel formado únicamente por texto y que puede ser interpretado directamente tanto por humanos como por un editor de texto cualquiera.

Se verán primero las clases base que permiten trabajar con caracteres individuales para más tarde trabajar con aquellas que permiten trabajar con cadenas de texto.

	Entrada Leer datos del fichero	Salida Escribir datos al fichero
Caracteres	FileReader fin = new FileReader(file)	FileWriter fout = new FileWriter (file [, añadir])
Cadenas	Scanner sc = new Scanner(file)	PrintWriter fout = new PrintWriter(file)

[4] https://docs.oracle.com/javase/tutorial/essential/io/bytestreams.html
[5] https://docs.oracle.com/javase/tutorial/essential/io/charstreams.html
[6] https://es.wikipedia.org/wiki/Archivo_de_texto

3.1.1. Leer y escribir caracteres

FileReader[7] y FileWriter[8] son las clases base que permiten leer y escribir caracteres en ficheros secuenciales. FileWriter además permite escribir cadenas.

A partir de los ejemplos siguientes se explicará su funcionamiento:

Ejemplo 2: Leer caracteres de un fichero de texto cerrándolo de manera explicita

```
1  public void leeFichero(File fichero) throws IOException {
2     FileReader fichIn=null;
3     try {
4        fichIn= new FileReader(fichero);
5        int i;
6        while ((i = fichIn.read()) != -1) {
7           System.out.print((char) i);
8        }
9     } finally {
10       if (fichIn != null) {
11          fichIn.close();
12       }
13    }
14 }
```

En este ejemplo se lee un fichero de texto, carácter a carácter, y se visualiza por pantalla.

- Línea 4: se define y se abre el flujo de entrada sobre un objeto de tipo File (también se podría usar una cadena de texto con la ruta completa del fichero) pasado como parámetro.
- Línea 5: esta variable, obligatoriamente de tipo entero (32 bits), almacena el carácter leído (en sus últimos 16 bits en caso de caracteres y en sus últimos 8 bits en caso de leer bytes).
- Línea 6: el método read de FileReader devuelve, como entero, el carácter leído o -1 en caso de llegar al final del fichero. Es este valor de -1 el valor que indica que se ha llegado al final de fichero por lo que se utilizará como condición de salida del bucle de lectura.
- Línea 7: se realiza el cast a char para obtener el carácter leído.
- Línea 1: dos excepciones habituales cuando se trabaja con ficheros son:
  - FileNotFoundException: no se ha podido encontrar el fichero.
  - IOException: se ha producido un error de entrada-salida.

  En el código de ejemplo se ha optado por propagar las excepciones.
- Líneas 9 y 11: mediante la cláusula finally nos aseguramos que el flujo, si ha sido abierto, se cierre.

[7] https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html
[8] https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html

A partir la versión 7 de Java se pueden definir flujos directamente en la clausula try sin necesidad de cerrarlos de forma implícita con close. El compilador se encarga de generar de forma automática el código para cerrar los flujos cuando no se necesiten.

Se pueden especificar los flujos que se quieran en el try, situándose todas las definiciones dentro de los paréntesis separándolas por un punto y coma

El ejemplo anterior usado try-with-resources[9] quedaría:

Ejemplo 2B: Leer caracteres de un fichero de texto cerrándolo de manera implícita

```
1  public void leeFichero(File fichero) throws IOException {
2     try (FileReader fichIn= new FileReader(fichero)){
3        int i;
4        while ((i = fichIn.read()) != -1) {
5           System.out.print((char) i);
6        }
7     }
8  }
```

Donde:

- Línea 3: se define el flujo dentro do try. De esta manera el flujo de cierra de forma automática cuando ya no sea necesario. Es decir, no hace falta usar close.

Un posible problema de los dos ejemplos anteriores es que es ineficiente ya que se realizan lecturas de carácter en carácter del fichero. Una forma de aumentar el rendimiento es que cada vez que accede a disco no lea un único carácter sino que intente leer un grupo de ellos: el especificado con el tamaño de un array auxiliar que se usará como un buffer intermedio. Para ello substituiremos el bucle while por el siguiente código:

Ejemplo 3: Leer un grupo de caracteres de un fichero de texto

```
1   char buffer[]= new char[50];
2   while ((i = fichIn.read(buffer)) != -1) {
3     System.out.print(new String(buffer,0,i));
4   }
```

- Línea 1: definimos un array de caracteres que servirá de almacenamiento temporal.
- Línea 2: el funcionamiento de read varía con respecto al ejemplo anterior. Ahora en la variable i no se almacena el carácter leído sino que se almacena en número de caracteres leídos (que como mucho será el tamaño de buffer creado, en nuestro ejemplo 50) y es en la variable buffer donde se almacenan todos los caracteres que se han leido.
- Línea 3: en esta línea se podría visualizar directamente el buffer pero podría dar resultado incorrectos. Imaginemos que estamos leyendo, con el

[9] https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

buffer anterior, un fichero que tiene 75 caracteres. En la primera iteración del bucle visualizaremos los 50 primeros caracteres del fichero, en la segunda iteración leeríamos los 25 caracteres restantes del fichero pero visualizaríamos el buffer completo, por lo que los últimos 25 caracteres visualizados serian incorrectos. Para evitar este problema creamos una cadena que solo posee los i caracteres leídos.

Si lo que se desea es escribir caracteres en un fichero secuencial, se tendría que usar FileWriter tal y como se muestra en el ejemplo siguiente:

Ejemplo 4: Escribir caracteres en un fichero de texto

```
1  public void escribeFichero(File fichero, String cad) throws IOException {
2     FileWriter fichOut=null;
3     try {
4       fichOut= new FileWriter(fichero);
5       for (int i=0;i<cad.length();i++){
6         fichOut.write(cad.charAt(i)); // Se escribe un carácter
7         fichOut.write(System.getProperty("line.separator")); // Se escribe una
8       }                                                      // cadena
9     } finally {
10       if (fichOut != null) {
11          fichOut.close();
12       }
13    }
14 }
```

```
try (FileWriter fichOut = new FileWriter(fichero)){
  for (int i=0;i<cad.length();i++){
    fichOut.write(cad.charAt(i));
    fichOut.write(System.getProperty("line.separator"));
  }
}
```

- Líneas 2 y 4: se usa un objeto FileWriter, en vez de un FileReader, para escribir caracteres.
- Línea 4: si el fichero existe se elimina su contenido y si no se crea vacio. Si se quiere modificar este comportamiento para que añada al final del fichero, en vez de borrar su contenido, habrá que añadir al constructor un 2º parámetro a true. FileWriter (File fichero, boolean append).
- Líneas 6 e 7: en vez de leer del flujo escribimos en él, por lo que usamos alguno de sus métodos write. Tenemos, entre otras, las siguientes posibilidaes:
  - write (String cad): escribe directamente una cadena.
  - write(String cad, int desde, int cantidad): permite escribir parte de una cadena.
- Línea 11: se debe cerrar siempre el flujo para liberar recursos.

Este tipo de clases poseen el método flush que fuerza que se escribir a disco los datos que aun permanezcan en memoria. Este método es invocado de forma automática cuando se cierra el flujo.

3.1.2. Leer y escribir cadenas

La clase Scanner[10] proporciona diversos métodos para leer datos desde distintos tipos de flujos de caracteres: entrada estándar, ficheros. Permite leer líneas completas, palabras simples, tipos primitivos (enteros, boleanos, bytes,…), … en ficheros secuenciales.

El siguiente método permite leer las líneas de un fichero de texto y mostrarlas por pantalla.

Ejemplo 5: Leer cadenas de caracteres

```
1  public void leerLineas (File fichero) throws FileNotFoundException{
2     Scanner sc=null;
3     try {
4        sc= new Scanner(fichero);
5        while (sc.hasNext()) {
6           System.out.println(sc.nextLine());
7        }
8     } finally {
9        if (sc != null) {
10          sc.close();
11       }
12    }
13 }
```

```
try (Scanner sc= new Scanner(fichero)){
   while (sc.hasNext()) {
      System.out.println(sc.nextLine());
   }
}
```

Donde:

- Línea 4: definimos el scanner sobre el fichero pasado como parámetro. Si se le pasa la ruta de un fichero como cadena el Scanner se realizará sobre la propia cadena y no sobre el fichero apuntado por ella.
- Línea 5: mientras haya más datos para leer en el flujo de entrada se sigue leyendo. El método hasNext indica si quedan tokens[11] de cualquier tipo por leer. Se podrían usar métodos más específicos como hasNextLine, hasNextInt, hasNextBoolean, … para leer tipos de datos concretos.
- Línea 6: nextLine lee y devuelve la siguiente línea. Imaginemos que tenemos un fichero de datos repartidos en columnas con datos de tipos int, podríamos usar hasNextInt y nextInt. Next devuelve el siguiente token, de cualquier tipo, por leer. Un token es una cadena de caracteres que, por defecto, está separada por espacios (que incluyen tabuladores y saltos de línea).
- Línea 10: se cierra el scanner.

La clase PrintWriter[12] permite escribir cualquier tipo de dato primitivo al flujo de salida mediante sus métodos print. Asimismo permite con printf y format escribir cadenas de texto con formato. Todos los métodos que terminen con 'ln' añaden un salto de línea al final. Su funcionamiento es exactamente igual a FileWriter.

[10] https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html
[11] https://www.geeksforgeeks.org/java-tokens/
[12] https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html

Ejemplo 6: Escribir cadenas de caracteres

```
1  public void escribeFicheroPw(File fichero, String cad) throws IOException {
2     PrintWriter pw=null;
3     try {
4        pw= new PrintWriter(fichero);
5        pw.println(cad);
6     } finally {
7        if (pw != null) {
8           pw.close();
9        }
10    }
11 }
```

```
try (PrintWriter pw= new PrintWriter(fichero)){
   pw.println(cad);
}
```

- Línea 4: por defecto si el fichero no existe se crea vacio y si existe se borra su contenido. Si se desea añadir al final del mismo se debe sustituir el objeto File en la definición de PrintWriter por un objeto FileWriter con el segundo parámetro, append, a true.

```
PrintWriter fich = new PrintWriter(new FileWriter(fichero, true));
```

- Línea 5: se escribe una cadena con salto de línea.
- Linea 8: se cierra el flujo de escritura si este no es nulo.

3.2. Ficheros binarios

Como se comento al principio del tema los ficheros binarios son aquellos que almacenan secuencias de bytes y que, normalmente, no podemos entenderlos de forma directa sino que necesitan de un programa que interprete su contenido y lo muestre. Ejemplos de ficheros binarios puede ser un fichero de imagen en formato jpg, un fichero de audio en formato mp3, un archivo comprimido zip, un fichero de video en formato mp4, ...

Se verán primero las clases base, que son aquellas que permiten trabajar con bytes individuales para más tarde trabajar con aquellas que permiten trabajar con datos primitivos y objetos.

	Entrada (Leer datos de un fichero)	Salida (Escribir datos en un fichero)
Bytes	FileInputStream in = new FileInputStream(fichero)	FileOutputStream out = new FileOutputStream(fichero)
Tipos primitivos	DataInputStream dIn = new DataInputStream (in)	DataOutputStream dOut = new DataOutputStream(out)
Objetos	ObjectInputStream oIn = new ObjectInputStream (in)	ObjectOutputStream oOut = new ObjectOutputStream (out)

3.2.1. Leer y escribir bytes

Los dos clases base para trabajar con ficheros binarios secuenciales son FileOutputStream[13] y FileInputStream[14]. Estas clases se utilizan exactamente igual que las equivalentes de caracteres, FileWriter y FileReader, pero utilizando bytes en vez de caracteres.

Ejemplo 7: Leer y escribir en ficheros binarios

```
1  public void ficherosBinarios(File fileIn, File fileOut) throws IOException{
2    FileInputStream in = null;
3    FileOutputStream out = null;
4
5    try {
6      in = new FileInputStream(fileIn);
7      out = new FileOutputStream(fileOut, true);
8      int c;
9      while ((c = in.read()) != -1) {
10       out.write(c);
11     }
12   } finally {
13     if (in != null) in.close();
14     if (out != null) out.close();
15   }
16 }
```

```
try (FileInputStream in = new FileInputStream(fileIn);
    FileOutputStream out=new FileOutputStream(fileOut,true)){
     int c;
     while ((c = in.read()) != -1) {
       out.write(c);
     }
}
```

Este ejemplo permite copiar un fichero binario en otro, donde:

- Línea 1: el tratamiento de excepciones es exactamente igual a los vistos en puntos anteriores.
- Línea 6 y 7: se define el flujo binario de entrada y de salida.
- Línea 8: al igual que en el ejemplo de flujos de caracteres creamos una variable que almacenara el valor leído.
- Línea 9: mientras haya más datos en el flujo de entrada para leer, valor de c distinto a -1, se sigue leyendo. El valor leído se almacena en la variable c.
- Línea 10: el valor leído se escribe, con el método write, en el flujo de salida. En este caso al tener el segundo parámetro a true cada vez que se escribe en el fichero se añade al final. Si queremos que al escribir en el flujo se borre el contenido del fichero de destino o bien omitimos el segundo parámetro o bien se establece a false (en el constructor en la línea 7):

  `out = new FileOutputStream(fileOut)` ou `out = new FileOutputStream(fileOut, false)`

- Líneas 13 y 14: cerramos los flujos se han sido abiertos.

[13] https://docs.oracle.com/javase/8/docs/api/java/io/FileInputStream.html

[14] https://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html

Al igual que con los ficheros de caracteres un fichero binario no se tiene que leer byte a byte sino que se puede trabajar con un buffer intermedio.

Ejemplo 8: Descargar un fichero de Internet

```
1    InputStream is= new URL("http://example.com/cancion.mp3").openStream();
2    FileOutputStream fout=new FileOutputStream("salida.mp3");
3    int i;
4    byte[] buffer = new byte[1000];
5    while ((i = is.read(buffer)) != -1) {
6        fout.write(buffer, 0, i);
7    }
8    fout.close();
9    is.close();
```

Este ejemplo permite descargar un fichero de internet a disco, donde:

- Línea 1: el flujo de entrada es un InputStream[15], que es la clase padre de FileInputStream. Su utilización es idéntica. Este flujo se genera con el método openStream desde una URL[16].
- Línea 2: se define el flujo de salida en donde se guardará el fichero.
- Línea 5: mientras quede información por leer, se leen un grupo de bytes, el tamaño del array auxiliar, y se almacena en la variable buffer. La variable i almacena el número de bytes leídos.
- Línea 6: se escribe en el flujo de salida los datos leídos.
- Líneas 8 y 9: cerramos los flujos se han sido abiertos.

3.2.2. Leer y escribir tipos primitivos

Java también permite escribir tipos primitivos (int, float, double, "Strings", …) de forma transparente en un fichero binario. Para ello se dispone de las clases DataInputStream[17] y DataOutputStream[18].

Estas dos clases siguen poseyendo los métodos genéricos write y read pero además posee métodos específicos para leer y escribir directamente tipos de datos primitivos como pueden ser: readBoolean, writeBoolean, readLong, readInt, writeFloat, readDouble, readUTF, writeUTF, …

Normalmente para poder recuperar datos de un fichero de este tipo hay que recuperarlos en el mismo orden con el que fueron almacenados, se verá un ejemplo de su utilización más adelante.

El constructor de ambos requiere como parámetro un flujo a un fichero y no un objeto tipo file. Se usará un objeto FileInputStream en el caso de DataInputStream y un objeto FileOutputStream en el caso de DataOutputStream.

[15] https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html
[16] https://docs.oracle.com/javase/8/docs/api/java/net/URL.html
[17] https://docs.oracle.com/javase/8/docs/api/java/io/DataInputStream.html
[18] https://docs.oracle.com/javase/8/docs/api/java/io/DataOutputStream.html

Para los ejemplos que se verán a continuación se usara la clase auxiliar Clientes como contenedor de datos:

```
class Cliente {
    private String nombre;
    private int numCompras;
    private float credito;
    public Cliente(String nombre, int numCompras, float credito) {
        this.nombre = nombre;
        this.numCompras = numCompras;
        this.credito = credito;
    }
    public String getNombre()  { return nombre; }
    public int getNumCompras() { return numCompras; }
    public float getCredito()  { return credito; }
}
```

En el siguiente ejemplo escribiremos, como datos primitivos, una lista de clientes a un fichero.

Ejemplo 9: Escribir datos primitivos en un flujo

```
1  public void escribirPrimitivos(File fichero, ArrayList<Cliente> datos)
2                                                          throws IOException{
3     FileOutputStream fos=null;
4     DataOutputStream out=null;
5     try {
6        fos = new FileOutputStream(fichero);
7        out=new DataOutputStream(fos);
8        for (Cliente cl:datos) {
9          out.writeUTF(cl.getNombre());
10         out.writeInt(cl.getNumCompras());
11         out.writeFloat(cl.getCredito());
12       }
13    } catch (IOException e) {
14        System.out.println("Error: "+
15          e.getLocalizedMessage());
16    } finally {
17       if (out != null) out.close();
18       if (fos != null) fos.close();
19    }
  }
```

```
try (FileOutputStream fos = new
FileOutputStream(fichero);
  DataOutputStream out=new DataOutputStream(fos)) {
     for (Cliente cl:datos) {
       out.writeUTF(cl.getNombre());
       out.writeInt(cl.getNumCompras());
       out.writeFloat(cl.getCredito());
     }
} catch (IOException e)
{System.out.println("Error");}
```

Donde:

- Línea 6: se puede forzar a que los datos primitivos se añadan al final del fichero añadiendo un segundo parámetro (append) a true. Si no se añade se borra el contenido del archivo al crear el flujo.
- Línea 7: se define el flujo de salida usando el objeto FileOutputStrean definido anteriormente.
- Líneas 9, 10 e 11: se escribe cada tipo de dato particular con su método correspondiente y en un orden en concreto. Este orden será el mismo con el que posteriormente se lean los datos.
- Línea 13: en este caso se trata la excepción dentro del método, pero también se podría propagar, como se ha hecho en ejemplos anteriores.
- Líneas 17 y 18: se cierran los flujos abiertos.

Para recuperar los datos almacenados en el fichero se usara un flujo DataInputStream con el método read correspondiente al dato que se desea recuperar. Para detectar el final de fichero en este flujo hay que capturar la excepción de fin de fichero, EOFException[19] y no, como hacíamos hasta ahora, un valor de -1 devuelto por un método read.

El siguiente ejemplo complementa el ejemplo 9 de escritura de tipos primitivos:

Ejemplo 10: Leer datos primitivos en un flujo

```
1  public void leerPrimitivos(File fichero) throws IOException {
2   FileInputStream fin=null;
3   DataInputStream in=null;
4   try {
5     fin = new FileInputStream(fichero);
6     in=new DataInputStream(fin);
7     try { // Los datos se recuperan en el mismo orden en que se guardaron
8        while (true){
9           System.out.printf("Nombre: %s  Cantidad: %d  Crédito: %f \n",
10                                    in.readUTF(), in.readInt(), in.readFloat());
11          }
12     } catch (EOFException e) {
13        System.out.println("Fin de fichero");
14     }
15  } finally {
16     if (in != null) in.close();
17     if (fin != null) fin.close();
18  }
19 }
```

Donde:

- Línea 6: se define el flujo de entrada usando el objeto FileInputStream definido en la línea anterior.
- Líneas 8 y 12: se leen datos del fichero mientras no se produzca la excepción EOFException que indica final de fichero.
- Línea 9 y 10: se leen los tipos de datos primitivos, con su método apropiado, en el mismo orden en el que fueron almacenados, para que los datos leídos sean consistentes.
- Líneas 16 y 17: se cierran los flujos abiertos.

```
try ( FileInputStream fin = new FileInputStream(fichero);
     DataInputStream in=new DataInputStream(fin)){
  try {  // Los datos se recuperan en el mismo orden en que se guardaron
    while (true){
      System.out.printf("Nombre: %s  Cantidad: %d  Crédito: %f \n",
                  in.readUTF(), in.readInt(), in.readFloat());
    }
  } catch (EOFException e) {
    System.out.println("Fin de fichero");
  }
}
```

[19] https://docs.oracle.com/javase/8/docs/api/java/io/EOFException.html

El ejemplo siguiente es un ejemplo de la utilización de ambos métodos de forma simultánea:

Ejemplo 11: Lectura y escritura de datos primitivos

```
1  Cliente cliente1=new Cliente("Iria",32,433.3f);
2  Cliente cliente2=new Cliente("Sara",14,142.1f);
3  Cliente cliente3=new Cliente("Xabi",124,251.7f);
4  Cliente cliente4=new Cliente("Pablo",124,251.7f);
5
6  ArrayList<Cliente> clientes=new ArrayList<>();
7  clientes.add(cliente1);
8  clientes.add(cliente2);
9  clientes.add(cliente3);
10 clientes.add(cliente4);
11
12 try {
13     escribirPrimitivos(new File("D:/primitivos.dat"), clientes);
14     leerPrimitivos(new File("D:/primitivos.dat"));
15 } catch (IOException e) {
16     System.out.println(e.getLocalizedMessage());
17 }
```

3.2.3. Leer y escribir objetos. Serialización.

En el punto anterior hemos visto como se almacenan los atributos de cada objeto de forma individual. Sin embargo puede ser más cómodo trabajar directamente con objetos como un todo en vez de con sus atributos de forma individual como hemos hecho en el punto anterior. Para ello Java proporciona las clases ObjectInputStream[20] y ObjectOutputStream[21] que permiten escribir de forma directa objetos en un flujo. La única condición que se tiene que cumplir es que las clases de los objetos que se deseen almacenar, o recuperar, deben implementar la interfaz Serializable[22].

Se van a repetir los métodos del ejercicio anterior usando estas nuevas clases, lo primero que se debe hacer es redefinir la clase Cliente para que acepte la interfaz Serializable:

```
class Cliente implements Serializable{

  // el resto de código es idéntico a la clase cliente definida al principio del punto anterior

}
```

[20] https://docs.oracle.com/javase/8/docs/api/java/io/ObjectInputStream.html
[21] https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html
[22] https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html

La clase escribirObjeto quedaría:

Ejemplo 12: Escribir un objeto en un flujo

```
1  public void escribirObjeto(File fichero, ArrayList<Cliente> datos)
2                                                          throws IOException {
3     FileOutputStream fos=null;
4     ObjectOutputStream out=null;
5     try {
6        fos = new FileOutputStream(fichero);
7        out=new ObjectOutputStream(fos);
8        for (Cliente cl:datos) {
9           out.writeObject(cl); // Se escribe el objeto de forma directa
10       }
11    } finally {
12       if (out != null) out.close();
13       if (fos != null) fos.close();
14    }
   }
```

```
try (FileOutputStream fos=new FileOutputStream(fichero);
 ObjectOutputStream out=new ObjectOutputStream(fos)){
 for (Cliente cl:datos) {
    out.writeObject(cl);
  }
}
```

Donde:

- Línea 7: se define el flujo de salida usando el flujo FileOutputStream definido en la línea anterior, en este caso como vamos a escribir objetos usaremos la clase ObjectOutputStream.
- Línea 9: mediante el método writeObject del flujo se escribe de forma directa un objeto al fichero. Cada vez que se cierra el flujo se añade una cabecera al fichero. Debido a esto una vez cerrado el flujo no se puede volver abrir para añadir más datos ya que se genera la excepción StreamCorruptedException al añadir otra cabecera. Una solución[23] es crear la siguiente clase:

```
class AppendingObjectOutputStream extends ObjectOutputStream {
        public AppendingObjectOutputStream(OutputStream out) throws IOException {
          super(out);
        }
        @Override
        protected void writeStreamHeader() throws IOException {
            reset();// do not write a header
        }
}
```

Y reemplazar la línea 6 del ejemplo número 12 por las siguientes líneas de código:

```
if (fichero.length()==0) out = new ObjectOutputStream(fos);
else out = new AppendingObjectOutputStream(fos);
```

Si el fichero está vacío usamos un flujo ObjectOutputStream para introducir datos por primera vez, pero si lo que queremos es escribir datos a un fichero no vacío usaremos la nueva clase creada AppendingObjectOutputStream.

- Líneas 12 y 13: se cierran los flujos abiertos.

[23] https://stackoverflow.com/questions/1194656/appending-to-an-objectoutputstream

Un ejemplo de lectura de objetos es el siguiente método, este es equivalente al visto en la sección de lectura de tipos primitivos, pero en esta ocasión se leen directamente objetos.

Ejemplo 13: Leer un objeto de un flujo

```
1  public void leerObjetos(File fichero)
2                                  throws IOException, ClassNotFoundException {
3   FileInputStream fin=null;
4   ObjectInputStream in=null;
5   try {
6      fin = new FileInputStream(fichero);
7      in=new ObjectInputStream(fin);
8      try {
9         Cliente cl;
10        while (true){
11           cl=(Cliente)in.readObject();
12           System.out.printf("Nombre: %s  Cantidad: %d  Crédito: %f \n",
13                             cl.getNombre(), cl.getNumCompras(), cl.getCredito());
14        }
15     } catch (EOFException e) {
16        System.out.println("Fin de fichero");
17     }
18  } finally {
19    if (in != null) in.close();
20    if (fin != null) fin.close();
21  }
}
```

```
try (FileInputStream fin = new FileInputStream(fichero);
    ObjectInputStream in=new ObjectInputStream(fin)){
 try {
   Cliente cl;
   while (true){
    cl=(Cliente)in.readObject();
    System.out.printf("Nombre: %s  Cantidad: %d Crédito: %f"+
    "\n", cl.getNombre(), cl.getNumCompras(), cl.getCredito());
   }
 } catch (EOFException e) {
      System.out.println("Fin de fichero");
 }
}
```

Donde:

- Línea 7: se define el flujo de entrada usando el objeto FileInputStream definido en la línea anterior, en este caso como vamos a leer objetos usaremos la clase ObjectInputStream.
- Líneas 10 y 15: se leen datos del fichero mientras no se produzca la excepción EOFException la cual indica el final de fichero.
- Línea 11: se lee un objeto mediante el método readObjetct. Como se lee un objeto genérico hay que realizar un cast al tipo de objeto que esperamos leer. Si se produce algún tipo de error se puede lanzar la excepción ClassNotFoundException.
- Línea 13: al tener un objeto de tipo Cliente se pueden usar sus métodos get.
- Líneas 19 y 20: se cierran los flujos abiertos.

Ya que en un mismo fichero se puede almacenar más de un tipo de objeto tendremos que determinar, antes de realizar el cast, cual es el tipo del objeto que estamos leyendo. Tenemos dos opciones:

- Usar el objeto de tipo java.lang.Class asociado a la clase que hemos leído. Por lo que la línea 10 se sustituiría por el siguiente código:

```
// Obtenemos el objeto del flujo y se lo asociamos a una variable
Object obj= in.readObject();
```

```
// Vemos el tipo de objeto que es comparando sus clases
if (obj.getClass() == Cliente.class) {
   Cliente cliente=(Cliente)obj;      // Hacemos el cast correspondiente
   …
}
```

- Usar instanceof:

```
// Obtenemos el objeto del flujo y se lo asociamos a una variable
Object obj= in.readObject();

// Vemos el tipo de objeto comprobando si obj es una instancia de Cliente
if (obj instanceof Cliente) {
   Cliente cliente=(Cliente)obj; …      // Hacemos o cast correspondiente
   …
}
```

3.3. Ficheros con Buffer

El problema de las opciones vistas hasta ahora es que cada lectura o escritura accede a memoria secundaria con la penalización en eficiencia que esto conlleva.

Para solucionar esto Java proporciona cuatro clases que encapsulan a las clases base vistas añadiéndole, de forma transparente, un buffer intermedio que persigue la optimización del acceso a disco:

- Cuando se leen datos del disco no solo se leerán los datos solicitados sino que se leerá una cantidad mayor almacenándose en un buffer intermedio. Mientras haya datos en el buffer los datos se obtienen de él pero cuando este se vacíe se vuelve a acceder al disco para leer otro grupo de datos.
- Cuando escribimos datos en el flujo realmente no se escriben directamente a disco sino que se van almacenado en este buffer intermedio y es cuando este se llena cuando realmente se vuelca a disco.

  Los métodos siguientes permiten forzar el volcado de los datos, almacenados en el buffer intermedio, a disco sin esperar a que este se llene:

  - flush: fuerza el volcado a disco.
  - close: cerrando el flujo se hace una llamada implícita a flush.

Las cuatro clases que Java nos proporcionan para el uso de buffers son:

Tipo de flujo	Contenedor	Clase contenida
Flujo de caracteres	BufferedReader[24]	FileReader
	BufferedWriter[25]	FileWriter
Flujo binario	BufferedInputStream[26]	FileInputStream
	BufferedOutputStream[27]	FileOutputStream

Los constructores de estos contenedores pueden aceptar dos parámetros:

- El primero es el flujo a encapsular.
- El segundo es opcional. Si se especifica usara este valor como tamaño del buffer y en caso contrario tomara el valor establecido por java como defecto[28].

La utilización de estos buffers es idéntica a la de la clase que encapsulan (a excepción de BufferedReader que nos proporciona un método para leer líneas).

Ejemplos de su definición son:

Ejemplo 14: Definición de ficheros con buffer

```
1  File fichero = new File("d:/datos.txt");

2  FileReader fr = new FileReader (fichero);
3  BufferedReader bfr = new BufferedReader(fr); // lectura fichero de caracteres

4  FileWriter fw= new FileWriter(fichero);
5  BufferedWriter bfo = new BufferedWriter(fw); // escritura en fichero de charact.

6  FileInputStream fin = new FileInputStream(fichero);
7  BufferedInputStream bin = new BufferedInputStream(fin); // lectura fichero binar.

8  FileOutputStream fOut = new FileOutputStream (fichero);
9  BufferedOutputStream bout = new BufferedOutputStream(fOut); // escritura
                                                               // fichero binario
```

Donde:

- Línea 3: se define el flujo con buffer binario de entrada que encapsula a su flujo correspondiente de entrada definido en la línea anterior.
- Línea 5: se define el flujo con buffer binario de salida que encapsula a su flujo correspondiente de salida definido en la línea anterior.
- Línea 7: se define el flujo con buffer de caracteres de entrada que encapsula a su flujo correspondiente de entrada definido en la línea anterior.

[24] https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html
[25] https://docs.oracle.com/javase/8/docs/api/java/io/BufferedWriter.html
[26] https://docs.oracle.com/javase/8/docs/api/java/io/BufferedInputStream.html
[27] https://docs.oracle.com/javase/8/docs/api/java/io/BufferedOutputStream.html
[28] https://stackoverflow.com/questions/19561167/buffer-size-for-bufferedinputstream

- Línea 9: se define el flujo con buffer de caracteres de salida que encapsula a su flujo correspondiente de entrada definido en la línea anterior.

El comportamiento por defecto cuando se abre el flujo para escribir es de borrarlo. Si se quiere añadir al final hay que añadir a FileOutputStream o a FileWriter el segundo parámetro, Append, a true.

Los siguientes son dos ejemplos de su uso:

Ejemplo 15:Escribimos cadenas de texto en un fichero usando Buffers

```
1  public boolean escribeFichero(File fichero, String[] cadenas)  {
2     try (FileWriter fw= new FileWriter(fichero, true);
3         BufferedWriter bfo = new BufferedWriter(fw) ){
4         for(String s:cadenas) {
5           bfo.write(s);
6           bfo.newLine();
7         }
8         return true;
9     } catch (IOException ex) {
10       return false;
11    }
12 }
```

Donde:

- Líneas 2 y 3: se define los flujos necesarios para la escritura. Como se está usando una estructura try-with-resources los flujos se cierran automáticamente.
- Líneas 4: se recorre el array pasado como parámetro.
- Línea 5: se escribe una cadena de texto en el fichero.
- Línea 5: se escribe un salto de línea.

Ejemplo 16: Leemos cadenas de texto en un fichero usando Buffers

```
1  public void leedFichero(File fichero) {
2     try (FileReader fr = new FileReader (fichero);
3         BufferedReader bfr = new BufferedReader(fr) ){
4         String linea;
5         while ((linea = bfr.readLine()) != null) {
6           System.out.println(linea);
7         }
8     } catch (IOException ex) {
9        System.out.println(ex.getLocalizedMessage());
10    }
11 }
```

Donde:

- Líneas 2 y 3: se define los flujos necesarios para la lectura. Como se está usando una estructura try-with-resources los flujos se cierran automáticamente.
- Líneas 4: mientras la línea leída sea distinta de null se siguen leyendo líneas.

4. Ficheros de acceso aleatorio

Al contrario que los ficheros secuenciales, en los que los datos se han de leer desde el principio hacia el final de forma lineal, los ficheros de acceso aleatorio permiten moverse a cualquier posición del fichero para leer o escribir. Se podría ver como un array en disco que internamente tiene un puntero que indica la posición actual en el fichero.

Se usara para ello la clase RandomAccessFile[29] que nos proporciona, además de métodos específicos para movernos por el fichero, los mismos métodos que DataInputStream y DataOutputStream para leer y escribir tipos primitivos.

El constructor de RandomAccessFile necesita de dos parámetros:

- El fichero en el que se va a leer o escribir.
- El modo[30] de acceso al fichero. Los modos más usados son:
  - "r": solo permite leer del fichero. Si se intenta escribir se produce una IOException.
  - "rw": se permite leer y/o escribir. Ambas operaciones se pueden realizar con el mismo flujo.

Destacaremos los siguientes métodos específicos:

- length: indica el tamaño del fichero en bytes.
- getFilePointer: devuelve la posición actual en el fichero en bytes. La posición al principio del fichero es 0, es la posición en la que se encuentre el puntero al abrir el fichero, y el final es length.
- seek: mueve el puntero del fichero a la posición indicada en bytes desde el principio del fichero.
- skipBytes: desplaza el puntero, desde la posición actual, los bytes indicados en su parámetro.
- setLength: establece un nuevo tamaño para el fichero. Pueden producirse dos situaciones:
  - Si el tamaño nuevo es mayor que el tamaño actual del fichero este crece con un contenido no definido.
  - Si el tamaño nuevo es menor que el tamaño actual del fichero este se trunca al tamaño nuevo.

Cuando se leen datos (readByte, readChar, readUTF, …) o se escriben (writeFloat, writeInt, writeUTF…) el puntero interno avanzara tantas posiciones en el fichero como bytes se hayan leído o escrito. Por ejemplo si leemos un byte el puntero avanza una posición pero si leemos un entero avanzara cuatro. Internamente todos los datos se almacenan como bytes.

[29] https://docs.oracle.com/javase/8/docs/api/java/io/RandomAccessFile.html
[30] https://docs.oracle.com/javase/8/docs/api/java/io/RandomAccessFile.html#mode

Para guardar y leer cadenas se deben usar los métodos readUTF y writeUTF dejando que RandomAccessFile se encargue de calcular las longitudes correctas de las cadenas a leer y/o escribir.

Al igual que cuando escribíamos datos primitivos se debe guardar un orden cuando leemos o escribirnos datos de distinto tipo.

El siguiente código es un ejemplo de su utilización:

```
    Ejemplo 17:Uso de fichero de acceso aleatorio
1   String vocales="aeiou";
2   RandomAccessFile fileAle=null;
3   try {
4      fileAle=new RandomAccessFile("D:/random.txt", "rw");
5      for (int i=0;i<vocales.length();i++){
6         fileAle.writeInt(i);
7         fileAle.writeChar(vocales.charAt(i));
8      }
9      fileAle.seek(0);
10     while (fileAle.getFilePointer()<fileAle.length()){
11        System.out.printf("%d %s %n",fileAle.readInt(),fileAle.readChar());
12     }
13
14     fileAle.writeInt(7);
15     fileAle.writeChars("r");
16
17     int nuevaPos=2;
18     fileAle.seek(nuevaPos*4+nuevaPos*2);
19     System.out.printf("%n%d %s %n%n",fileAle.readInt(),fileAle.readChar());
20
21     fileAle.writeInt(9);
22     fileAle.writeChar('j');
23
24     fileAle.seek(0);
25     while (fileAle.getFilePointer()<fileAle.length()){
26        System.out.printf("%d %s %n",fileAle.readInt(),fileAle.readChar());
27     }
28  } catch (IOException e) {
29     System.out.println(e.getLocalizedMessage());
30  }
31  finally {
32     if (fileAle != null) {
33        try {
34           fileAle.close();
35        } catch (IOException e) {}
36     }
37  }
```

Donde:

- Línea 4: se define el flujo de acceso aleatorio en modo lectura/escritura.
- Líneas 6 y 7: se escribe un entero con la posición de la vocal en la cadena más el carácter con la vocal en esa posición en la cadena.
- Línea 9: el puntero del fichero se posiciona al principio del mismo.
- Líneas 10 y 11: se recorre el fichero visualizando su contenido en el mismo orden en el que fue almacenado.

- Líneas 14 y 15: al terminar de visualizar el contenido del fichero el puntero esta al final del mismo. Se le añade un entero y un carácter al final del mismo.
- Líneas 17, 18 y 19: se puede considerar que el conjunto de un entero y un carácter es un registro. Se quiere visualizar el registro número 2, para ello se ha de calcular la posición de comienzo de dicho registro teniendo en cuenta que un entero ocupa 4 bytes y un char (UTF-16) ocupa 2.
- Líneas 21 y 22: luego de posicionarse en el registro 2 y visualizar su contenido la posición del puntero es la que corresponde con el registro 3 que se sobrescribe con los nuevos valores.
- Línea 24: nos volvemos a situar al principio del fichero.
- Línea 25 y 26: se vuelve a visualizar el fichero desde el principio.
- Línea 34: se cierra el flujo.

5. Apéndice I: try-with-resources[31]

A partir la versión 7 de Java se pueden definir flujos directamente en la clausula try sin necesidad de cerrarlos de forma implícita con close. El compilador se encarga de generar de forma automática el código para cerrar los flujos cuando no se necesiten.

Ejemplo 20: try-with-resources

```
1   Charset charset = Charset.forName("US-ASCII");
2   String s = "cadena de texto";
3   try (BufferedWriter writer = new BufferedWriter(file_writer)) {
4      writer.write(s);
5   } catch (IOException x) {
6      System.err.format("IOException: %s%n", x);
7   }
```

Se pueden especificar los flujos que se quieran en el try, se sitúan todas las definiciones dentro de los paréntesis y separadas por un punto y coma.

[31] https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

6. Apéndice II: Propiedades del sistema[32]

Las propiedades del sistema son conjuntos de pares clave/valor que contienen información sobre el entorno en el cual se está ejecutando la maquina virtual de Java.

Para obtener su valor se utiliza el método getProperty de la clase System pasándole como parámetro la clave que se quiere obtener.

Un ejemplo de su utilización es el siguiente:

Ejemplo 18:

```
1 String separadorFichero=System.getProperty("file.separator");
2 String sistemaOperativo=System.getProperty ("os.name");
```

Alguna de las claves más interesantes son:

- file.separator: indica el carácter que se utiliza para separar directorios en un archivo.
- line.separator: indica el carácter o caracteres usados para separar líneas en ficheros.
- os.name: nombre del sistema operativo.
- user.dir: directorio de trabajo actual.
- user.home: raíz del directorio personal del usuario.

Para obtener todos los valores de las propiedades, en el entorno actual, se puede usar el siguiente código:

Ejemplo 19:Obtención de todas las propiedades del sistema

```
1    Properties propiedades = System.getProperties();
2    Enumeration<Object> valores = propiedades.keys();
3
4
5    String valor;
6    while (valores.hasMoreElements()) {
7       valor=valores.nextElement().toString();
8       System.out.printf("%s: %s %n",valor, propiedades.get(valor));
     }
```

[32] https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#getProperties--

7. Bibliografía

1. Clase File
2. Try with resources
3. Paquete Java IO (input - output)
4. Flujos en java - Streans
    1. Flujos
    2. Flujos de caracteres
    3. Flujos binarios
    4. Flujos de caracteres
        1. Archivo de texto
        2. FileReader: flujo para leer caracteres
        3. FileWriter: flujo para escribir caracteres
        4. Scanner: flujo para leer cadenas
        5. ¿Que es un token?
        6. PrintWriter: flujo para escribir cadenas
    5. Flujos binarios
        1. FileInputStream: flujo para leer bytes
        2. FileOutputStream: flujo para escribir bytes
        3. Clase InputStream
        4. Clase URL
        5. Flujos de tipos primitivos
            1. DataInputStream: flujo para leer tipos primitivos
            2. DataOutputStream: flujo para escribir tipos primitivos
            3. EOFException: excepción que surge al intentar leer en el final de fichero
        6. Flujos de Objetos
            1. Serializable: requisito para que una clase pueda ser almacenada en un fichero binario con ObjectOutputStream
            2. ObjectInputStream: flujo para leer objetos java
            3. ObjectOutputStream: flujo para escribir objetos java
            4. Añadir datos a ObjectOutputStream
        7. Flujos con buffer
            1. BufferedReader: clase que encapsula a fileReader añadiendole un buffer
            2. BufferedWriter: clase que encapsula a fileWriter añadiendole un buffer
            3. BufferedInputStream: clase que encapsula a fileInputStream añadiendole un buffer
            4. BufferedOutputStream: clase que encapsula a fileOutputStream añadiendole un buffer
            5. Tamaño del buffer
    6. Flujos a ficheros de acceso aleatorio:
        1. RandomAccessFile: fichero de acceso aleatorio
        2. Modos en RandomAccessFile
5. Propiedades del sistema
6. Try with resources