Ficheros de texto

Introducción

La información en los ficheros se puede guardar en formato binario o en formato texto. Dado que la información en un ordenador siempre se representa en binario, el significado de la frase anterior es que en la codificación binaria de un fichero de texto lo que se representan son caracteres, mientras que en otros tipos de ficheros pueden codificarse objetos de cualquier clase.

En un fichero de texto la información se almacena como una secuencia de caracteres. Algunos caracteres de esa secuencia tienen un significado especial, por ejemplo, como marcas de fin de línea (CR/LF).

Como con cualquier otro tipo de fichero, con la información almacenada en un fichero de texto, solo se pueden hacer dos cosas: "leerla", cargándola en objetos en la RAM para procesarla; o "escribirla", almacenando en el fichero información presente en objetos en la RAM.

En ambos casos, se necesita:

1. Un fichero físico, identificado por un nombre.
2. Un objeto de alguna clase específica que permita manipular el fichero físico desde el programa
3. Establecer una conexión entre el fichero físico y el objeto que lo maneja.
4. Operaciones para tratar los elementos del fichero; generalmente, con un bucle que actúe mientras queden elementos por tratar.
5. Finalizar (cerrar) la conexión entre el fichero físico y el objeto que lo maneja, a fin de que el fichero físico quede disponible para otros usos.

En este documento se ilustra el manejo de ficheros de texto a través de una clase, *TextFileDemo*, para la que se desarrollan varios métodos que leen y escriben información.

```
import java.io.*;

public class TextFilesDemo {

        …

}
```

La clase *TextFilesDemo* importa el paquete *java.io.** porque los diferentes métodos a desarrollar necesitarán manejar objetos de varias clases incluidas en dicho paquete.

Lectura de datos de un fichero de texto

Lectura de un texto

Vamos a empezar por desarrollar un método llamado *showText* que lea un texto almacenado en un fichero y lo muestre en pantalla. Al método *showText* se le pasa el nombre del fichero a tratar.

La implementación de *showText* que se muestra en el Ejemplo 1 utiliza un objeto de la clase *Scanner* para acceder al fichero, por lo que, previamente, se tendrá que haber añadido a la clase la cláusula "*import java.util.Scanner*". La clase Scanner proporciona una forma cómoda de leer de un fichero.

```
1.   public static void showText(String name) {
2.       try {
3.           Scanner input =
4.               new Scanner(new BufferedReader(new FileReader(name)));
5.           while (input.hasNextLine()) {
6.               String line = br.nextLine();
7.               System.out.println(line);
8.           }
9.           input.close();
10.      } catch (FileNotFoundException e) {
11.          System.out.println("Fichero no encontrado");
12.      }
13.  }
```

Ejemplo 1

El primer paso es obtener acceso al fichero externo, lo que se hace en la línea 4:

```
new Scanner(new BufferedReader(new FileReader(name)));
```

En esta línea se crea un objeto *Scanner* (referenciado por la variable *input*) usando un objeto de la clase *BufferedReader* creado "al vuelo" a partir de un objeto de la clase *FileReader* creado, a su vez, usando el nombre del fichero pasado como parámetro al método *showText*. *FileReader* es una clase diseñada para facilitar la lectura de ficheros de caracteres. Como no se puede leer un fichero que no existe, *FileReader* lanza una excepción de la clase *FileNotFoundException* si no lo encuentra, razón por la que se ha encerrado la sentencia, junto con el resto, que solo se pueden ejecutar si se ha encontrado el fichero, en un bloque *try* con el correspondiente bloque *catch*.

La clase *BufferedReader*, por su parte, sirve para "*bufferizar*" las operaciones de lectura, utilizando un área de memoria (*buffer*) para cargar por adelantado porciones del texto, evitando que todas las operaciones de lectura individuales tengan que acceder directamente al dispositivo externo.

Una vez creado el objeto *Scanner*, el bucle de las líneas de la 5 a la 8, lee, una a una, las líneas del fichero usando el método *nextLine* del *Scanner*, que lee todos los caracteres de una línea y los devuelve en una *String*, exceptuando los que marcan el fin de línea. El bucle está controlado por el método *hasNextLine*, que indica si quedan, o no, líneas por leer.

Una vez que se ha terminado de trabajar con el fichero, hay que cerrarlo, con el método *close*, para finalizar la conexión con el fichero físico.

La lectura del fichero línea a línea se podría haber hecho también directamente con el objeto de la clase BufferedReader, como muestra el Ejemplo 2, pero la clase *Scanner* ofrece más opciones, cómo se muestra en la siguiente sección. La clase *BufferedReader* solo ofrece operaciones para leer líneas, caracteres, o arrays de caracteres.

```
1.  public static void showText1(String name) {
2.      BufferedReader br = null;
3.      try {
4.          br = new BufferedReader(new FileReader(name));
5.          while (br.ready()) {
6.              String line = br.readLine();
7.              System.out.println(line);
8.          }
9.      } catch (FileNotFoundException e) {
10.         System.out.println("Fichero no encontrado");
11.     } catch (IOException e) {
12.         System.out.println("Error leyendo el fichero");
13.     } finally {
14.         if (br != null) {
15.             try {
16.                 br.close();
17.             } catch (IOException e) {
18.                 System.out.println("Error al cerrar el fichero");
19.             }
20.         }
21.     }
22. }
```

Ejemplo 2

De forma similar a como se hacía en el Ejemplo 1, la línea 4 del Ejemplo 2 crea un objeto de la clase *BufferedReader* (referenciado por la variable *br*), conectado con un fichero físico a través de un objeto de la clase *FileReader*, y a continuación un bucle se encarga de realizar la lectura. Las diferencias en

esta parte están en las operaciones para leer y para ver si se puede seguir leyendo (control de bucle). Además, estas operaciones pueden lanzar *IOException*, por lo que en la línea 11 se ha añadido un bloque *catch* para recogerla. A diferencia de *FileNotFoundException*, *IOException*, en este ejemplo, ocurriría cuando el fichero ya ha sido abierto, por lo que la operación para cerrarlo, en lugar de estar a continuación del *while*, se ha trasladado a la línea 16, dentro de un bloque *finally*, para cerrarlo tanto si ocurre un excepción como si no. El problema es que el bloque *finally* se ejecutará también si ocurre *FileNotFoundException*, en cuyo caso el objeto de la clase *BufferedReader* no habrá sido creado, por lo que, antes de intentar cerrarlo, hay que preguntar si su valor es distinto de *null* (línea 14). Para poder hacer esta pregunta y cerrar luego el fichero, la declaración de la variable *br* se ha sacado a la línea 2, de manera que sea accesible en todos los bloques, y no solo en el bloque *try*, como sucedería si se hubiese mantenido en la línea 4. Finalmente, la operación *close* aplicada a un objeto de la clase *BufferedReader* puede, a su vez, lanzar *IOException*, por lo que se ha encerrado en otro bloque *try* (línea 15, con un *catch* en la línea 17).

Lectura de un texto por componentes

La clase *Scanner* ofrece otras formas de leer un texto, aparte de por líneas. Una primera opción es la lectura por *tokens*. A los efectos, el texto se considera formado por *tokens* y delimitadores, siendo un *token* cualquier secuencia de caracteres que no contenga un delimitador. Por omisión, se considera delimitador cualquier carácter perteneciente a la categoría "*Java whitespace*", lo que incluye espacios, tabulaciones, separadores de línea, etc. El Ejemplo 3 muestra, entre corchetes, cada *token* del fichero cuyo nombre se pasa como parámetro. Las únicas diferencias respecto al Ejemplo 1 son que se usa *hasNext* en vez de *hasNextLine* para ver si quedan elementos por leer, y *next* en vez de *nextLine* para leer.

```
 1.      public static void showTextTokens(String name) {
 2.          try {
 3.              Scanner input =
 4.                  new Scanner(new BufferedReader(new FileReader(name)));
 5.              while (input.hasNext()) {
 6.                  String token = input.next();
 7.                  System.out.println("[" + token + "]");
 8.              }
 9.              input.close();
10.          } catch (FileNotFoundException e) {
11.              System.out.println("Fichero no encontrado");
12.          }
13.      }
```

Ejemplo 3

La clase *Scanner* permite, además, leer variables de cualquier tipo primitivo, excepto *char*. Simplemente, si el *token* que se encuentra a continuación en el fichero tiene la forma adecuada para un literal del tipo en cuestión, se puede leer usando una operación específica para cada tipo.

```
1.   public static void showInts(String name) {
2.       try {
3.          Scanner input =
4.              new Scanner(new BufferedReader(new FileReader(name)));
5.          while (input.hasNext()) {
6.              if (input.hasNextInt()) {
7.                  int token = input.nextInt();
8.                  System.out.println("[" + token + "]");
9.              } else {
10.                 input.next();
11.             }
12.         }
13.         input.close();
14.      } catch (FileNotFoundException e) {
15.          System.out.println("Fichero no encontrado");
16.      }
17.  }
```

Ejemplo 4

El Ejemplo 4 muestra en pantalla todos los *tokens* de un fichero de texto que representan números enteros. Para hacerlo, el bucle (línea 5) sigue estando controlado por el método *hasNext*, que indica si queda algún *token* por leer, pero dentro se ha incluido una pregunta para ver si el siguiente *token* representa un número entero, en cuyo caso se lee y se muestra; si no es así, sólo se lee y se desecha al no hacer nada con él.

Escritura de un fichero de texto

Al igual que para leerla, existen varias clases que permiten escribir información en un fichero de texto, de las cuales una muy conveniente es la clase *PrintStream*, que proporciona varios métodos de escritura, entre ellos dos llamados *print* y *println* que se diferencian en que el segundo inserta una marca de fin de línea detrás de la información que escribe. El Ejemplo 5 ilustra el uso de la clase *PrintStream*.

```
1.  public static void createFile(String name) {
2.      try {
3.          PrintStream ps = new PrintStream(name);
4.          Scanner input = new Scanner(System.in);
5.          while (input.hasNext()) {
6.              String line = input.nextLine();
7.              ps.println(line);
8.          }
9.          input.close();
10.         ps.close();
11.     } catch (IOException e) {
12.         System.out.println("Error al crear el fichero");
13.     }
14. }
```

Ejemplo 5

El Ejemplo 5 crea en la línea 3 un objeto de la clase *PrintStream* asociado a un fichero creado con el nombre pasado como parámetro. En este fichero se van a ir escribiendo (línea 7) las líneas leídas de la consola (*System.in*) usando un objeto de la clase *Scanner* creado en la línea 4. El bucle de lectura-escritura (líneas 5 a 8), controlado por el método *hasNext* de la clase *Scanner*, puede interrumpirse pulsando la secuencia de teclas *Ctrl-z (en Windows) o Ctrl-d (en linux)*, que actúa a modo de "marca de fin". Una vez terminado el bucle, hay que cerrar tanto el objeto de la clase *PrintStream*, como el objeto de la clase *Scanner*.

Nótese que la creación del objeto de la clase *Scanner* se ha hecho después de crear el objeto de la clase *PrintStream*. De esta manera, si se produce una excepción al intentar crear este último, no es necesario cerrar el primero, ya que aún no se ha creado. Recuérdese que la creación de un objeto de la clase *Scanner* no produce ninguna excepción, salvo las que pudiesen derivarse de la creación de los objetos usados para crear el *Scanner* (como un *FileReader*), lo que no es el caso, dado que, en este caso, el *Scanner* se crea usando *System.in*.

Formateo de la salida

La clase *PrintStream* ofrece otros dos métodos, llamados *format* y *printf*, que permiten formatear la salida usando una *String* de formato que incluye secuencias de caracteres empezando por '%' para indicar diferentes opciones de formato. En el Ejemplo 6, la sentencia de las líneas de 8 a la 11 utiliza el método *printf* para escribir tres datos (número de línea, tamaño de la línea, y la línea propiamente dicha) con una *String* de formato en la que se muestra un mensaje en medio del que se inserta esta información; los dos primeros datos como números enteros (*%d*) y el tercero como ristra (*%s*). Además, se inserta una marca de fin de línea al final (*%n*). El resultado, si el usuario teclea como tercera línea la palabra "cinco", podría ser algo como:

"La línea 3 tiene 5 caracteres (cinco)"

```
1.   public static void createFileWithFormat(String name) {
2.       try {
3.           PrintStream ps = new PrintStream(name);
4.           Scanner input = new Scanner(System.in);
5.           int lineCount = 1;
6.           while (input.hasNext()) {
7.               String line = input.nextLine();
8.               ps.printf("La línea %d tiene %d caracteres (%s)%n",
9.                       lineCount++,
10.                      line.length(),
11.                      line);
12.          }
13.          ps.close();
14.          input.close();
15.      } catch (IOException e) {
16.          System.out.println("Error al crear el fichero");
17.      }
18.  }
```

Ejemplo 6

Se puede obtener más información sobre las opciones de formateo en la siguiente *url*, que proporciona ayuda de la clase *Formatter*: http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html.