10.C. Ficheros.

Sitio: VIRGEN DE LA PAZ
Curso: Programación
Libro: 10.C. Ficheros.

Imprimido por: Cristian Esteban Gómez
Día: miércoles, 31 de mayo de 2023, 11:25

Tabla de contenidos

1. Trabajando con ficheros.
1.1. Escritura y lectura de información en ficheros.
1.2. Ficheros binarios y ficheros de texto (I).
1.3. Ficheros binarios y ficheros de texto (II).
1.4. Modos de acceso. Registros.
1.5. Acceso secuencial.
1.6. Acceso aleatorio.

1. Trabajando con ficheros.

En este apartado vas a ver muchas cosas sobre los ficheros: cómo leer y escribir en ellos, aunque ya hemos visto algo sobre eso, hablaremos de las formas de acceso a los ficheros: secuencial o de manera aleatoria.

Siempre hemos de tener en cuenta que la manera de proceder con ficheros debe ser:

- **Abrir** o bien **crear** si no existe el fichero.

- **Hacer las operaciones** que necesitemos.

- **Cerrar el fichero**, para no perder la información que se haya modificado o añadido.

También es muy importante el **control de las excepciones**, para evitar que se produzcan fallos en tiempo de ejecución. Si intentamos abrir sin más un fichero, sin comprobar si existe o no, y no existe, saltará una excepción.

Para saber más

En el siguiente enlace a la wikipedia podrás ver la descripción de varias extensiones que pueden presentar los archivos.

Extensión de un archivo.

1.1. Escritura y lectura de información en ficheros.

Acabamos de mencionar los pasos fundamentales para proceder con ficheros: abrir, operar, cerrar.

Además de esas consideraciones, debemos tener en cuenta también las clases Java a emplear, es decir, recuerda que hemos comentado que si vamos a tratar con ficheros de texto, es más eficiente emplear las clases de `Reader` `Writer`, frente a las clases de `InputStream` y `OutputStream` que están indicadas para flujos de bytes.

Otra cosa a considerar, cuando se va a hacer uso de ficheros, es la forma de acceso al fichero que se va a utilizar, si va a ser de manera secuencial o bien aleatoria. En un fichero secuencial, **para acceder a un dato debemos recorrer todo el fichero desde el principio hasta llegar a su posición**. Sin embargo, en un fichero de acceso aleatorio podemos posicionarnos directamente en una posición del fichero, y ahí leer o escribir.

Aunque ya has visto un ejemplo que usa `BufferedReader`, insistimos aquí sobre la filosofía de estas clases, que usan la idea de un buffer.

La idea es que cuando una aplicación necesita leer datos de un fichero, tiene que estar esperando a que el disco en el que está el fichero le proporcione la información.

Un dispositivo cualquiera de memoria masiva, por muy rápido que sea, es mucho más lento que la CPU del ordenador.

Así que, es fundamental **reducir el número de accesos al fichero** a fin de mejorar la eficiencia de la aplicación, y para ello se asocia al fichero una memoria intermedia, el buffer, de modo que cuando se necesita leer un byte del archivo, en realidad se traen hasta el buffer asociado al flujo, ya que es una memoria mucho más rápida que cualquier otro dispositivo de memoria masiva.

Cualquier operación de Entrada/Salida a ficheros puede generar una `IOException`, es decir, un error de Entrada/Salida. Puede ser por ejemplo, que el fichero no exista, o que el dispositivo no funcione correctamente, o que nuestra aplicación no tenga permisos de lectura o escritura sobre el fichero en cuestión. Por eso, las sentencias que involucran operaciones sobre ficheros, deben ir en un bloque `try`.

Autoevaluación

Señala si es verdadera o es falsa la siguiente afirmación:

La idea de usar buffers con los ficheros es incrementar los accesos físicos a disco. ¿Verdadero o falso?

○ Verdadero.
○ Falso.

1.2. Ficheros binarios y ficheros de texto (I).

Ya comentamos anteriormente que los ficheros se utilizan para guardar la información en un soporte: disco duro, disquetes, memorias usb, dvd, etc., y posteriormente poder recuperarla. También distinguimos dos tipos de ficheros: los de **texto** y los **binarios**.

En los **ficheros de texto** la información se guarda como caracteres. Esos caracteres están codificados en **Unicode**, o en **ASCII** u otras codificaciones de texto.

En la siguiente porción de código puedes ver cómo para un fichero existente, que en este caso es texto.txt, averiguamos la codificación que posee, usando el método `getEncoding()`

```
FileInputStream fichero;
try {
     // Elegimos fichero para leer flujos de bytes "crudos"
     fichero = new FileInputStream("c:\\texto.txt");
     // InputStreamReader sirve de puente de flujos de byte a caracteres
     InputStreamReader unReader = new InputStreamReader(fichero);
     // Vemos la codificación actual
     System.out.println(unReader.getEncoding());
} catch (FileNotFoundException ex) {

 }
```

El mismo código copiable:

```
FileInputStream fichero;
try {
    // Elegimos fichero para leer flujos de bytes "crudos"
 fichero = new FileInputStream("c:\\texto.txt");
    // InputStreamReader sirve de puente de flujos de byte a caracteres
 InputStreamReader unReader = new InputStreamReader(fichero);
    // Vemos la codificación actual
 System.out.println(unReader.getEncoding());
} catch (FileNotFoundException ex) {
   Logger.getLogger(textos.class.getName()).log(Level.SEVERE, null, ex);
}
```

Para **archivos de texto**, se puede abrir el fichero para leer usando la clase `FileReader`. Esta clase nos proporciona métodos para **leer caracteres**. Cuando nos interese no leer carácter a carácter, sino **leer líneas completas**, podemos usar la clase `BufferedReader` a partir de `FileReader`. Lo podemos hacer de la siguiente forma:

```
File arch = new File ("C:\\fich.txt");
FileReader fr = new FileReader (arch);
BufferedReader br = new BufferedReader(fr);
...
String linea = br.readLine();
```

Para **escribir en archivos de texto** lo podríamos hacer, teniendo en cuenta:

```
FileWriter fich = null;
PrintWriter pw = null;
fich = new FileWriter("c:/fich2.txt");
```

```
pw = new PrintWriter(fichero);
```

```
pw.println("Linea de texto");
```

```
...
```

Si el fichero al que queremos escribir existe y lo que queremos es añadir información, entonces pasaremos el segundo parámetro como true:

```
FileWriter("c:/fich2.txt",true);
```

Para saber más

En el siguiente enlace a wikipedia puedes ver el código ASCII.

Código Ascii.

1.3. Ficheros binarios y ficheros de texto (II).

Los **ficheros binarios** almacenan la información en bytes, codificada en binario, pudiendo ser de cualquier tipo: fotografías, números, letras, archivos ejecutables, etc.

Los archivos binarios guardan una representación de los datos en el fichero. O sea que, cuando se guarda texto no se guarda el texto en sí, sino que se guarda su representación en código UTF-8.

Para **leer datos de un fichero binario**, Java proporciona la clase **FileInputStream**. Dicha clase trabaja con bytes que se leen desde el flujo asociado a un fichero. Aquí puedes ver un ejemplo comentado.

```
package fileinputconbuffer;

import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class leerConBuffer {

   public static void main(String[] args) {
         int tama ;

   try{
     // Creamos un nuevo objeto File, que es la ruta hasta el fichero desde
     File f = new File("C:\\apuntes\\test.bin");

     // Construimos un flujo de tipo FileInputStream (un flujo de entrada desde
     // fichero) sobre el objeto File. Estamos conectando nuestra aplicación
     // a un extremo del flujo, por donde van a salir los datos, y "pidiendo"
     // al Sistema Operativo que conecte el otro extremo al fichero que indica
     // la ruta establecida por el objeto File f que habíamos creado antes. De
     FileInputStream flujoEntrada = new FileInputStream(f);
     BufferedInputStream fEntradaConBuffer = new BufferedInputStream(flujoEntrada);

     // Escribimos el tamaño del fichero en bytes.
     tama = fEntradaConBuffer.available();
     System.out.println("Bytes disponibles: " + tama);

     // Indicamos que vamos a intentar leer 50 bytes del fichero.
     System.out.println("Leyendo 50 bytes....");

     // Creamos un array de 50 bytes para llenarlo con los 50 bytes
```

```
    // que leamos del flujo (realmente del fichero)*/
    byte bytearray[] = new byte[50];

    // El método read() de la clase FileInputStream recibe como parámetro un
    // array de byte, y lo llena leyendo bytes desde el flujo.
    // Devuelve un número entero, que es el número de bytes que realmente se
    // han leído desde el flujo. Si el fichero tiene menos de 50 bytes, no
    // podrá leer los 50 bytes, y escribirá un mensaje indicándolo.
    if (fEntradaConBuffer.read(bytearray) != 50)
        System.out.println("No se pudieron leer 50 bytes");

    // Usamos un constructor adecuado de la clase String, que crea un nuevo
    // String a partir de los bytes leídos desde el flujo, que se almacenaron
    // en el array bytearray, y escribimos ese String.
    System.out.println(new String(bytearray, 0, 50));

    // Finalmente cerramos el flujo.Es importante cerrar los flujos
    // para liberar ese recurso. Al cerrar el flujo, se comprueba que no
    // haya quedado ningún dato en el flujo sin que se haya leído por la aplicación. */
    fEntradaConBuffer.close();

    // Capturamos la excepción de Entrada/Salida. El error que puede
    // producirse en este caso es que el fichero no esté accesible, y
    // es el mensaje que enviamos en tal caso.
   }catch (IOException e){
           System.err.println("No se encuentra el fichero");
   }
   }
}
```

Para **escribir datos a un fichero binario**, la clase nos permite usar un fichero para escritura de bytes en él, es la clase `FileOutputStream`. La filosofía es la misma que para la lectura de datos, pero ahora el flujo es en dirección contraria, desde la aplicación que hace de fuente de datos hasta el fichero, que los consume.

Autoevaluación

Señala si es verdadera o falsa la siguiente afirmación:

Para leer datos desde un fichero codificados en binario empleamos la clase FileOutputStream. ¿Verdadero o falso?

- ○ Verdadero.
- ○ Falso.

1.4. Modos de acceso. Registros.

En Java no se impone una estructura en un fichero, por lo que conceptos como el de registro que si existen en otros lenguajes, en principio no existen en los archivos que se crean con Java. Por tanto, los programadores deben estructurar los ficheros de modo que cumplan con los requerimientos de sus aplicaciones.

Así, el programador definirá su registro con el número de bytes que le interesen, moviéndose luego por el fichero teniendo en cuenta ese tamaño que ha definido.

Se dice que un fichero es de acceso directo o de organización directa cuando para acceder a un registro **n** cualquiera, no se tiene que pasar por los **n-1** registros anteriores. En caso contrario, estamos hablando de ficheros secuenciales.

Con Java se puede trabajar con **ficheros secuenciales** y con **ficheros de acceso aleatorio**.

En los **ficheros secuenciales**, la información se almacena de manera secuencial, de manera que para recuperarla se debe hacer en el mismo orden en que la información se ha introducido en el archivo. Si por ejemplo queremos leer el registro del fichero que ocupa la posición tres (en la ilustración sería el número 5), tendremos que abrir el fichero y leer los primeros tres registros, hasta que finalmente leamos el registro número tres.

Por el contrario, si se tratara de un **fichero de acceso aleatorio**, podríamos acceder directamente a la posición tres del fichero, o a la que nos interesara.

Autoevaluación

Señala la opción correcta:

- ○ Java sólo admite el uso de ficheros aleatorios.
- ○ Con los ficheros de acceso aleatorio se puede acceder a un registro determinado directamente.
- ○ Los ficheros secuenciales se deben leer de tres en tres registros.
- ○ Todas son falsas.

1.5. Acceso secuencial.

En el siguiente ejemplo vemos cómo se **escriben datos en un fichero secuencial**: el nombre y apellidos de una persona utilizando el método `writeUTF()` que proporciona `DataOutputStream`, seguido de su edad que la escribimos con el método `writeInt()` de la misma clase. A continuación escribimos lo mismo para una segunda persona y de nuevo para una tercera. Después cerramos el fichero. Y ahora lo abrimos de nuevo para ir **leyendo de manera secuencial** los datos almacenados en el fichero, y escribiéndolos a consola.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package escylecsecuen;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 *
 * @author JJBH
 */
public class escylee {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Declarar un objeto de tipo archivo
        DataOutputStream archivo = null ;
        DataInputStream fich = null ;
        String nombre = null ;
        int edad = 0 ;
        try {
            // Creando o abriendo para añadir el archivo
            archivo = new DataOutputStream( new FileOutputStream("c:\\secuencial.dat",true) );

            // Escribir el nombre y los apellidos
```

```
            archivo.writeUTF( "Antonio López Pérez     " );

            archivo.writeInt(33) ;

            archivo.writeUTF( "Pedro Piqueras Peñaranda" );

            archivo.writeInt(45) ;

            archivo.writeUTF( "José Antonio Ruiz Pérez " ) ;

            archivo.writeInt(51) ;

            // Cerrar fichero

            archivo.close();


            // Abrir para leer

            fich = new DataInputStream( new FileInputStream("c:\\secuencial.dat") );

            nombre = fich.readUTF() ;

            System.out.println(nombre) ;

            edad = fich.readInt() ;

            System.out.println(edad) ;

            nombre = fich.readUTF() ;

            System.out.println(nombre) ;

            edad = fich.readInt() ;

            System.out.println(edad) ;

            nombre = fich.readUTF() ;

            System.out.println(nombre) ;

            edad = fich.readInt() ;

            System.out.println(edad) ;

            fich.close();



        } catch(FileNotFoundException fnfe) { /* Archivo no encontrado */ }

          catch (IOException ioe) { /* Error al escribir */ }

          catch (Exception e) { /* Error de otro tipo*/

            System.out.println(e.getMessage());}


      }


}
```

Fíjate al ver el código, que hemos tenido la precaución de ir escribiendo las cadenas de caracteres con el mismo tamaño, de manera que sepamos luego el tamaño del registro que tenemos que leer.

Por tanto para **buscar información en un fichero secuencial**, tendremos que abrir el fichero e ir leyendo registros hasta encontrar el registro que buscamos.

¿Y si queremos **eliminar un registro en un fichero secuencial**, qué hacemos? Esta operación es un problema, puesto que no podemos quitar el registro y reordenar el resto. Una opción, aunque costosa, sería crear un nuevo fichero. Recorremos el fichero original y vamos copiando registros en el nuevo hasta llegar al registro que queremos borrar. Ese no lo copiamos al nuevo, y seguimos copiando hasta el final, el resto

de registros al nuevo fichero. De este modo, obtendríamos un nuevo fichero que sería el mismo que teníamos pero sin el registro que queríamos borrar. Por tanto, si se prevé que se va a borrar en el fichero, no es recomendable usar un fichero de este tipo, o sea, secuencial.

Autoevaluación

Señala si es verdadera o es falsa la siguiente afirmación:

Para encontrar una información almacenada en la mitad de un fichero secuencial, podemos acceder directamente a esa posición sin pasar por los datos anteriores a esa información. ¿Verdadero o Falso?

- ○ Verdadero.
- ○ Falso.

1.6. Acceso aleatorio.

A veces no necesitamos leer un fichero de principio a fin, sino acceder al fichero como si fuera una base de datos, donde se accede a un registro concreto del fichero. Java proporciona la clase `RandomAccessFile` para este tipo de entrada/salida.

La clase `RandomAccessFile` permite utilizar un fichero de **acceso aleatorio** en el que el programador define el formato de los registros.

```
RandomAccessFile objFile = new RandomAccessFile( ruta, modo );
```

Donde ruta es la dirección física en el sistema de archivos y modo puede ser:

- "r" para sólo lectura.

- "rw" para lectura y escritura.

La clase `RandomAccessFile` implementa los interfaces `DataInput` y `DataOutput`. Para abrir un archivo en modo lectura haríamos:

```
RandomAccessFile in = new RandomAccessFile(“input.dat”, “r”);
```

Para abrirlo en modo lectura y escritura:

```
RandomAccessFile inOut = new RandomAccessFile(“input.dat”, “rw”);
```

Esta clase permite leer y escribir sobre el fichero, no se necesitan dos clases diferentes.

Hay que especificar el modo de acceso al construir un objeto de esta clase: sólo lectura o lectura/escritura.

Dispone de métodos específicos de desplazamiento como `seek` y `skipBytes` para poder moverse de un registro a otro del fichero, o posicionarse directamente en una posición concreta del fichero.

No está basada en el concepto de flujos o streams.

Autoevaluación

Indica si es verdadera o es falsa la siguiente afirmación:

Para decirle el modo de lectura y escritura a un objeto RandomAccessFile debemos pasar como parámetro "rw". ¿Verdadero o Falso?

○ Verdadero.
○ Falso.