

viewnext.adrformacion.com © ADR Infor SL
VICTOR TENA PALOMARES

Gestión de ficheros

© ADR Infor SL

viewnext.adrformacion.com © ADR Infor SL
VICTOR TENA PALOMARES

viewnext.adrformacion.com © ADR Infor SL
VICTOR TENA PALOMARES

Indice

Competencias y Resultados de Aprendizaje desarrollados en esta unidad	3
Gestión de ficheros	4
Objetivos	4
Flujos	4
Tipos de flujos	4
Metodología de trabajo	5
Flujos de bytes	6
Flujo de lectura de bytes	6
Ejemplos	7
Flujo de escritura de bytes	9
Ejemplos	11
Flujos de caracteres	13
Flujo de lectura de caracteres	13
Ejemplos	14
Flujo de escritura de caracteres	18
Clase File	19
Ejercicios	23
Ejercicio 1. Leer un fichero	23
Recomendaciones	23
Lo que se necesita para comenzar	23
Datos a mostrar por consola:	23
Ejercicio 2. Copiar ficheros	24
Recomendaciones	24
Lo que se necesita para comenzar	24
Datos a mostrar por consola	24
Ejercicio 3. Creación, edición y borrado de ficheros.	25
Lo que se necesita para comenzar	25
Datos a mostrar por consola	25
Ejercicio 4. Manejo de ficheros	26
Datos a mostrar por consola	26
Ejercicio 5. Manejo de carpetas	27
Datos a mostrar por consola	28
Recursos	29
Enlaces de Interés	29
Glosario.	29

Competencias y Resultados de Aprendizaje desarrollados en esta unidad

Competencia:

Manejar correctamente los ficheros desde los programas Java

Resultados de Aprendizaje:

- Entender la lectura del contenido de ficheros desde un programa Java
- Leer un fichero desde un programa Java
- Entender la escritura de contenido en un fichero desde un programa Java
- Copiar ficheros desde un programa Java
- Crear, editar y borrar ficheros desde un programa Java
- Manejar ficheros desde un programa Java
- Manejar carpetas desde un programa Java

Gestión de ficheros

Objetivos

Se va a estudiar, fundamentalmente, la metodología de trabajo para que un programa Java realice las siguientes operaciones:

- Lectura de datos almacenados en un fichero local.
- Escritura de datos en un fichero, que no tiene que existir a priori.

Flujos

Los flujos, surgen por la necesidad que tienen las aplicaciones Java de interaccionar con el exterior. Hay dos posibles formas de hacerlo:

- **Enviando datos a la consola del DOS, a un fichero, etc.**
- **Capturando datos procedentes del teclado, de ficheros, de páginas web, etc.**

Concepto de flujo: **es como un río.**

El agua en movimiento es el flujo, su contenido son los datos.

Lo que permite que esos datos viajen de un lugar a otro es el agua, es decir, el flujo.

Java modela flujos mediante clases del paquete java.io.

Tipos de flujos

Son los siguientes:

De entrada o de lectura

Envían datos desde un origen al programa.

El origen suele ser un fichero aunque no siempre, ya que puede ser también un **socket** como se verá más adelante.

Estos flujos se emplean para que el programa pueda acceder y leer datos del origen.

De salida o de escritura

Envían datos desde el programa a un destino.

El destino, como antes, suele ser un fichero o un socket.

Se emplean para que el programa pueda escribir en el destino.

Dentro de cada tipo de flujo puede haber dos subtipos en función de los datos que manejan:

Flujos de bytes

Trabajan con bytes, es decir, con cadenas de unos y ceros agrupadas de ocho en ocho.
Ejemplos de datos que se expresan mediante bytes: son **números enteros**, **ficheros ejecutables** (.exe), ficheros .class, ficheros de **mapa de bits** (.bmp), ficheros de **imágenes** (.gif, .jpg, etc.), etc.

Flujos de caracteres

Trabajan con caracteres expresados según el código Unicode (engloba a Ascii).
Ejemplos de datos expresados mediante caracteres: **ficheros de texto** (.rtf, .txt, .doc, etc.), **ficheros .java**, **ficheros .html**, etc.
En realidad, los caracteres son también bytes por lo que no sería necesario trabajar con flujos de caracteres.
No obstante, a veces no se puede trabajar directamente con bytes o no conviene, por esto se introducen este tipo de flujos.

Metodología de trabajo

Se podría resumir en los siguientes pasos:

Identificar

Identificar el tipo de flujo a utilizar

Crear

Crear el flujo mediante las clases y constructores adecuados.

Asignar

Asignar un filtro o búfer que optimice el trasvase de datos.
Actúa como un aditivo para la gasolina, mejorando el rendimiento del motor. Podría interpretarse como un contenedor intermedio que almacena los datos con los que va a trabajar un programa, leyéndolos o escribiéndolos, y cuya finalidad es realizar estas operaciones del modo más eficaz posible.
Una analogía simplista podría ser la siguiente: un amigo compra una bolsa de pipas, te apetece comer unas cuantas y le pides. Tu amigo te ofrece, pero sacándolas de la bolsa y de una en una (esto sería leer datos sin filtro o búfer). Al final, le dices que te dé un montón en la mano para que te las puedas comer más cómodamente sin necesidad de ir cogiendo de la bolsa de una en una (esto sería leer datos con filtro o búfer).
Para el caso de escribir datos con filtro o sin él se podría considerar el siguiente ejemplo: las cáscaras de las pipas tienen que tirarse en una papelera que está un poco alejada del lugar donde te las estás comiendo. En vez de ir a la papelera cada vez que te comes una pipa (escribir datos sin filtro o búfer), es preferible verterlas en tu mano y, cuando se llene, llevarlas a la papelera (escribir datos con filtro o búfer).

Leer o escribir datos

Leer o escribir datos desde el filtro, en el caso de que se vaya a utilizar, o directamente desde el flujo mediante los métodos adecuados.

Cerrar

Cerrar el flujo o el filtro.

Flujos de bytes

En este paquete hay varias clases abstractas situadas en lo más alto de su jerarquía, de las que se nutren, vía herencia, otras. Así, por ejemplo, para modelar los flujos de entrada y de salida de bytes se dispone de las clases abstractas **java.io.InputStream** y **java.io.OutputStream** respectivamente.

Estas clases cuentan con métodos de lectura y escritura de datos muy generales que pueden emplear todas sus subclases.

Se van a estudiar algunas subclases muy útiles de estas clases abstractas pero primero se tratarán los flujos de bytes y luego los flujos de caracteres.

Flujo de lectura de bytes

Clase

java.io.FileInputStream.

Esta clase modela un flujo de lectura de bytes.

Se emplea para **leer datos en forma de bytes** procedentes de un origen que suele ser un fichero.

Clase asociada

La Clase **FileInputStream** pertenece al paquete **java.io** y hereda de la clase abstracta **InputStream** que también pertenece al paquete **java.io**.

Constructores principales

- **FileInputStream(String origen):** crea un flujo de lectura de bytes que puede leer el origen que se le pasa al argumento (habitualmente un fichero).
- **FileInputStream(File origen):** igual que el anterior pero pasándole un objeto **File** que debe crearse previamente. La clase **File** modela ficheros, luego el argumento va a estar asociado a un origen que es un fichero. Mediante métodos adecuados de **File** puede obtenerse cierta información sobre el mismo: su ruta, tamaño, fecha de creación, etc.

Estos constructores lanzan una **FileNotFoundException** que debe capturarse. **FileNotFoundException** hereda de **IOException**.

Métodos principales

- **int read():** devuelve el entero asociado al primer byte leído del flujo de lectura de bytes sobre el que se aplica. Si no hay ningún byte, devuelve **-1**. **MUY USADO**.
- **int read(byte[] array):** devuelve un entero asociado al número de bytes leídos del flujo de lectura de bytes sobre el que se aplica. Si ese flujo de lectura de bytes está asociado a un fichero con un número de bytes inferior al de elementos del array, el entero que devuelve no coincide con el número de elementos del array. En caso contrario, sí. Este array se declara, construye e inicializa con el valor por defecto asociado a la variable primitiva **byte**, es decir, **0**. **NO MUY USADO**.

- `int read(byte[] array, int índiceInicial, int índiceFinal)`: ídem anterior, pero leyendo sólo los bytes del array de bytes del primer argumento, a partir del índice del segundo y hasta el índice del tercero. Devuelve el número de bytes leídos. NO MUY USADO.
- `void close()`: cierra el flujo de lectura de bytes. Una vez utilizado un recurso externo, siempre conviene romper toda relación con el mismo con el fin de que el resto del código funcione del modo más óptimo. No cerrar un flujo de lectura de bytes sería algo así como aparcar un coche dejándolo en marcha. No tiene mucho sentido ya que se gastan recursos sin necesidad. MUY USADO.

Estos métodos lanzan una `IOException` de gestión obligatoria.

No suele leerse directamente mediante el `FileInputStream` sino que se usa un filtro asociado a la clase **`BufferedInputStream`**. Ir a la API y consultar los constructores.

En el tema se va a emplear el constructor que admite como argumento un objeto `InputStream`

Como `FileInputStream` hereda de `InputStream` el uso de un objeto `FileInputStream` como argumento del constructor es consistente.

Ejemplos

El objetivo del siguiente ejemplo es aprender a utilizar los flujos y filtros de lectura de bytes y el empleo del método `read()`.

Se pretende obtener el entero comprendido entre 0 y 255 asociado al primer byte del fichero local incluido responsable de la ejecución del programa `MSPaint`.

La ruta de este fichero dependerá del sistema operativo que tengamos instalado y su nombre es **`mspaint.exe`**.

Hay que tener en cuenta que el separador de directorios en los programas Java se escribe como dos contrabarras (`\`) o como dos barras normales (`/`) o como una barra normal (`.`).

```

import java.io.*;

public class FlujoLecturaBytes1 {
    public static void main(String args[]){
        String rutaFichero="c:\\Windows\\System32\\mspaint.exe";
        try{

            //Crear flujo de lectura y asociarlo a un fichero
            FileInputStream fis=new FileInputStream(rutaFichero);

            //Asignar filtro para optimizar la lectura
            BufferedInputStream bis=new BufferedInputStream(fis);

            //Leer datos del fichero
            int primerByte=bis.read();
            if(primerByte==-1)
                System.out.println("El fichero no contiene ningun byte");
            else
                System.out.println("Entero asociado al primer byte del "+
                    "fichero= "+primerByte);

            //Cerrar filtro. No hace falta cerrar el flujo
            bis.close();
        }catch(IOException e){
            System.out.println("Error---"+e.toString());
        }
    }
}

```

Por consola:**Hay que tener en cuenta varias cosas:**

- Si el fichero no se encuentra en la ruta especificada se lanza una `FileNotFoundException`.
- Si el fichero se encuentra en el mismo directorio que los `.class` del código fuente compilado, no es necesario especificar toda la ruta en el argumento del constructor de `FileInputStream`: basta con especificar su nombre.
- Si se elimina una de las contrabarras en el caso de que se haya optado por ellas, se generará un error al compilar debido a que dentro de un `String` se considera carácter de escape.
- Recuerda que Linux es case sensitive mientras que Windows no lo es.

Hecho el ejemplo, se creará una copia en el mismo directorio donde se encuentra el original. Se le asignará el nombre de pinturas.exe.

A continuación, se abrirá con el Bloc de notas y se eliminará su contenido.

Finalmente, se sustituirá el argumento del constructor de FileInputStream por la ruta asociada a pinturas.exe, se recompilará y se volverá a ejecutar.

Ahora si, partiendo de la base del ejemplo anterior, se pidiera mostrar por consola todos los enteros comprendidos entre 0 y 255 asociados a todos los bytes que forman parte del fichero anterior, ¿qué modificaciones deberían hacerse?

```
import java.io.*;

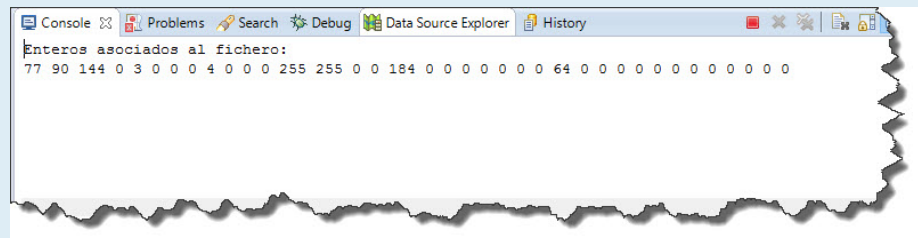
public class FlujoLecturaBytes2{
    public static void main(String args[]){
        String rutaFichero="C:\\ProgramData\\Microsoft\\Windows\\Start Menu\\Programs\\
Accessories\\paint.exe";
        try{
            FileInputStream fis=new FileInputStream(rutaFichero);
            BufferedInputStream bis=new BufferedInputStream(fis);           //Leer datos del fi
chero

            System.out.println("Enteros asociados al fichero:");
            int siguienteByte;

            //Se emplea un bucle para leer todo el fichero
            //Es la forma típica de trabajar
            while((siguienteByte=bis.read())!=-1)
                System.out.print(siguienteByte+" ");
            bis.close();
        }catch(IOException e){
            System.out.println("Error---"+e.toString());
        }
    }
}
```

Por consola:

Aparece una lista de enteros asociados a los bytes que componen el fichero.



Flujo de escritura de bytes

Clase**java.io.FileOutputStream.**

Esta clase Modela un flujo de escritura de bytes.

Se emplea para **escribir datos en forma de bytes** en un destino que suele ser un fichero.

Clase asociada

La clase **FileOutputStream** pertenece al paquete **java.io** y hereda de la clase abstracta **OutputStream** del mismo paquete **java.io**.

Constructores principales

- `FileOutputStream(String destino)`: crea un flujo de escritura de bytes (feb, a partir de ahora) que puede escribir en el destino que se le pasa al argumento (habitualmente un fichero). Si el fichero no existe, se crea automáticamente. Si ya existe, al escribir en él, se destruye su contenido anterior.
- `FileOutputStream(String destino, boolean b)`: ídem anterior, pero permitiendo conservar el contenido del fichero si se le pasa true al segundo argumento.
- `FileOutputStream(File destino)`: ídem anterior, pero pasándole un objeto File que debe crearse previamente.

Estos métodos constructores lanzan una excepción de tipo **FileNotFoundException** que debe capturarse. Esta excepción hereda de **IOException**.

Métodos principales

- `void write(int num)`: escribe el byte del argumento en el feb sobre el que se aplica.
- `void write(byte[] array)`: ídem anterior, pero escribiendo los bytes del array que se le pasa al argumento.
- `void write(byte[] array, int índiceInicial, int índiceFinal)`: ídem anterior, pero escribiendo los bytes comprendidos entre los dos índices del segundo y tercer argumento.
- `void close()`: cierra el feb.

Estos métodos lanzan una **IOException** que debe capturarse.

No suele escribirse directamente mediante el `FileOutputStream` sino que **se usa un filtro asociado a la clase `BufferedOutputStream`**. Ir a la API y consultar los constructores

En el tema se va a emplear el constructor que admite como argumento un objeto `OutputStream`.

Como `FileOutputStream` hereda de `OutputStream` el uso de un objeto `FileOutputStream` como argumento del constructor es consistente.

La clase `OutputStream` cuenta con un método muy empleado cuando se trabaja con flujos y filtros de escritura. Es el método **`void flush()`**; su misión es asegurar el vaciado del filtro sobre el que se aplica.

Después de vaciar el filtro debe cerrarse con `close()`.

Ejemplos

Este ejemplo muestra como se escribe en un fichero de texto, que no existe inicialmente en el disco duro local, un dato asociado al entero del argumento del método.

Al ser el destino un fichero de texto, la salida va a ser el correspondiente carácter Unicode.

```

import java.io.*;

public class FlujoEscrituraBytes1 {
    public static void main(String args[]){
        String rutaFichero="c:\\cursos\\lolo.txt";
        try{
            FileOutputStream fos=new FileOutputStream(rutaFichero, true);
            BufferedOutputStream bos=new BufferedOutputStream(fos);
            bos.write(72);

            //Vaciar el filtro
            bos.flush();

            //Cerrar el filtro
            bos.close();
        }catch(IOException e){
            System.out.println("Error---"+e.toString());
        }
    }
}

```

Al ejecutar, se crea el fichero **lolo.txt** en la carpeta **cursos** del disco duro local **C**. Su contenido es el carácter Unicode correspondiente al entero 72, es decir, la H.

Si se cambia el número del argumento se obtienen otros caracteres Unicode. Por ejemplo, el 125 es }, el 300 es la coma (,), etc.

Si lo que queremos es escribir más de un byte a la vez deberemos añadir un bucle:

```

import java.io.*;

public class FlujoEscrituraBytes2{
    public static void main(String args[]){
        String rutaFichero="c:\\cursos\\lolo.txt";
        try{
            FileOutputStream fos=new FileOutputStream(rutaFichero);
            BufferedOutputStream bos=new BufferedOutputStream(fos);

            //Se escriben los caracteres Unicode del alfabeto occidental
            for(int i=0;i<256;i++)
                bos.write(i);
            bos.flush();
            bos.close();
        }catch(IOException e){
            System.out.println("Error---"+e.toString());
        }
    }
}

```

Al ejecutar se sustituye el contenido del fichero lolo.txt por una línea con los caracteres Unicode occidentales.

Si se quiere mantener el contenido **FileOutputStream** **fos=new FileOutputStream(rutaFichero,true);**

Flujos de caracteres

Para modelar los flujos de lectura y escritura de caracteres se dispone de las clases abstractas **java.io.Reader** y **java.io.Writer** respectivamente.

Estas clases cuentan con unos métodos de lectura y escritura de datos muy generales que pueden emplear todas sus subclases.

Se van a estudiar algunas subclases de estas clases abstractas.

Flujo de lectura de caracteres

Clase
java.io.FileReader. Modela un flujo de lectura de caracteres. Se emplea para leer datos en forma de caracteres procedentes de un origen que suele ser un fichero.
Clase asociada
La clase FileReader hereda de InputStreamReader que, a su vez, hereda de Reader . Todas ellas pertenecen al paquete java.io.

Constructores principales

- `FileReader(String origen)`: crea un flujo de lectura de caracteres (flc, a partir de ahora) que puede leer el origen que se le pasa al argumento (habitualmente un fichero).
- `FileReader(File origen)`: ídem anterior, pero pasándole un objeto `File` que debe crearse previamente.

Estos constructores lanzan una excepción de tipo **FileNotFoundException** que debe capturarse. Esta excepción hereda de **IOException**.

Métodos principales

- `int read()`: devuelve el entero asociado al primer carácter leído del flc sobre el que se aplica. Si no hay ningún carácter, devuelve `-1`. MUY USADO.
- `int read(byte[] array)`: devuelve un entero asociado al número de caracteres leídos del flc sobre el que se aplica. Si ese flc está asociado a un fichero con un número de caracteres inferior al de elementos del array, el entero que devuelve no coincide con el número de elementos del array. En caso contrario, sí. NO MUY USADO.
- `int read(byte[] array, int índiceInicial, int índiceFinal)`: ídem anterior, pero leyendo sólo los caracteres del array de caracteres del primer argumento, a partir del índice del segundo y hasta el índice del tercero. Devuelve el número de caracteres leídos. NO MUY USADO.
- `void close()`: cierra el flujo de lectura de bytes. MUY USADO.

Estos métodos lanzan una **IOException** de gestión obligatoria.

No suele leerse directamente mediante el `FileReader` sino que se usa un filtro asociado a la clase **BufferedReader**. Ir a la API y consultar los constructores.

En el tema se va a emplear el constructor que admite como argumento un objeto `Reader`

Como `FileReader` hereda de `InputStreamReader` que, a su vez, hereda de `Reader`, el uso de un objeto `FileReader` como argumento del constructor es consistente.

Además, el uso del filtro permite la utilización del método `String readLine()` para leer el contenido del fichero línea a línea en vez de carácter a carácter.

Devuelve **null** en el momento en que se ha leído todo el contenido.

Ejemplos

El siguiente ejemplo muestra como se lee un fichero de texto línea a línea empleando el filtro `BufferedReader` y el método `readLine()`.

Para realizar el ejemplo, se creará un fichero de nombre **ejemplo.txt** y se guardará en el disco duro.

Su contenido serán las tres líneas siguientes:

Hola, estoy en el curso de java.

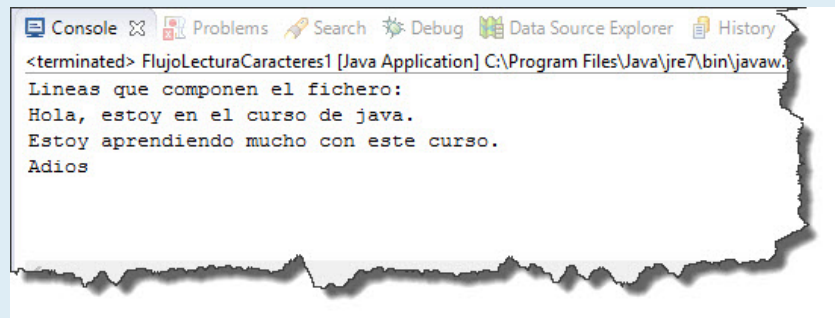
Estoy aprendiendo mucho con este curso.

Adios

```
import java.io.*;

public class FlujoLecturaCaracteres1 {
    public static void main(String args[]) {
        String rutaFichero="c:\\cursos\\ejemplo.txt";
        try {
            FileReader fr=new FileReader(rutaFichero);
            BufferedReader br=new BufferedReader(fr);
            String siguienteLinea;
            System.out.println("Lineas que componen el fichero:");
            while((siguienteLinea=br.readLine())!=null)
                System.out.println(siguienteLinea);
            br.close();
        } catch (IOException e) {
            System.out.println("Error---"+e.toString());
        }
    }
}
```

Por consola:



En el siguiente ejemplo se realiza una comparativa del tiempo de ejecución de tres códigos que leen un fichero de texto de unos 700 KB.

Los modos de lectura del fichero son:

- Sin filtro y leyendo carácter a carácter
- Con filtro y leyendo carácter a carácter
- Con filtro y leyendo línea a línea

Para saber el tiempo de ejecución de los códigos se usa el método estático de System **long currentTimeMillis()** que devuelve los milisegundos transcurridos desde un origen de tiempos con fecha del 1/1/1970 a las 0:00:00, hasta el momento de ejecución de la línea en la que aparece el método.

Sin filtro y leyendo carácter a carácter:

```
import java.io.*;

public class LecturaFicheroCaracterACaracterSinFiltro {

    public static void main(String args[]) {
        String rutaFichero="c:\\cursos\\setuplog.txt";
        try {
            long ini=System.currentTimeMillis();

            //Lectura de un fichero de 700 KB
            FileReader fr=new FileReader(rutaFichero);
            int siguienteCaracter;
            while((siguienteCaracter=fr.read())!=-1){}
            fr.close();
            System.out.println("Tiempo ejecucion="+
                (System.currentTimeMillis()-ini)+" ms");

            //Se obtienen tiempos de ejecucion del orden de los 5400 ms
        } catch (IOException e) {
            System.out.println("Error---"+e.toString());
        }
    }
}
```

Con filtro y leyendo carácter a carácter:


```

import java.io.*;

public class LecturaFicheroCaracterACaracterConFiltro {
    public static void main(String args[]){
        String rutaFichero="c:\\cursos\\setuplog.txt";
        try{
            long ini=System.currentTimeMillis();

            //Lectura de un fichero de 700 KB
            FileReader fr=new FileReader(rutaFichero);
            BufferedReader br=new BufferedReader(fr);
            int siguienteCaracter;
            while((siguienteCaracter=br.read())!=-1){}
            br.close();
            System.out.println("Tiempo ejecucion="+
                (System.currentTimeMillis()-ini)+" ms");

            //Se obtienen tiempos de ejecucion del orden de los 90 ms
        }catch(IOException e){
            System.out.println("Error---"+e.toString());
        }
    }
}

```

Con filtro y leyendo línea a línea:

```

import java.io.*;

public class LecturaFicheroCaracterACaracterConFiltroLineaALinea {
    public static void main(String args[]){
        String rutaFichero="c:\\cursos\\setuplog.txt";
        try{
            long ini=System.currentTimeMillis();

            //Lectura de un fichero de 700 KB
            FileReader fr=new FileReader(rutaFichero);
            BufferedReader br=new BufferedReader(fr);
            String siguienteLinea;
            while((siguienteLinea=br.readLine())!=null){}
            br.close();
            System.out.println("Tiempo ejecucion="+
                (System.currentTimeMillis()-ini)+" ms");

            //Se obtienen tiempos de ejecucion del orden de los 70 ms
        }catch(IOException e){
            System.out.println("Error---"+e.toString());
        }
    }
}

```

La forma óptima de leer un fichero usando flujos de caracteres es utilizar un filtro asociado a la clase `BufferedReader` y el método de lectura `readLine()` que permite leer línea a línea. También se podría hacer con flujos de bytes.

Flujo de escritura de caracteres

Clase

java.io.FileWriter

Modela un **flujo de escritura** de caracteres.

Se emplea para **escribir datos en forma de caracteres en un destino** que suele ser un fichero.

Clase asociada

La clase `FileWriter` pertenece al paquete **java.io** y hereda de **OutputStreamWriter** que, a su vez, hereda de **Writer**.

Constructores principales

- `FileWriter(String destino)`: crea un flujo de escritura de caracteres (fec, a partir de ahora) que puede escribir en el destino que se le pasa al argumento (habitualmente un fichero). Si el destino no existe, se crea automáticamente. Si ya existe, al escribir en él, se destruye su contenido anterior.
- `FileWriter(String destino, boolean b)`: ídem anterior, pero permitiendo conservar el contenido del destino si se le pasa un `true` al segundo argumento.
- `FileWriter(File destino)`: ídem primero, pero pasándole un objeto `File` que debe crearse previamente.

Estos constructores lanzan una **FileNotFoundException** que debe capturarse. Esta excepción hereda de `IOException`.

Métodos principales

- `void write(int num)`: escribe el carácter asociado al entero del argumento en el fec sobre el que se aplica teniendo en cuenta el código Unicode.
- `void write(char[] array)`: ídem anterior, pero escribiendo los caracteres del array que se le pasa al argumento.
- `void write(char[] array, int índiceInicial, int índiceFinal)`: ídem anterior, pero escribiendo los caracteres comprendidos entre los dos índices del segundo y tercer argumento.
- `void write(String texto)`: escribe la `String` que se le pasa al argumento en el fec sobre el que se aplica.
- `void write(String texto, int índiceInicial, int índiceFinal)`: ídem anterior, pero escribiendo los caracteres de la `String` del primer argumento comprendidos entre los dos índices del segundo y tercer argumento.

- void close(): cierra el fsc.
- void flush(): asegura el vaciado del búfer asociado, si lo hay. Primero se aplica este método y luego se cierra con el close().

Estos métodos lanzan una **IOException** de gestión obligatoria.

No suele escribirse directamente mediante el FileWriter sino que **se usa un filtro asociado a la clase BufferedWriter**. Ir a la API y consultar los constructores.

En el tema se va a emplear el constructor que admite como argumento un objeto Writer

Como FileWriter hereda de OutputStreamWriter que, a su vez, hereda de Writer, el uso de un objeto FileWriter como argumento del constructor es consistente.

La clase BufferedWriter cuenta con el método void flush() ya visto en los flujos de escritura de bytes.

Su misión aquí es la misma: asegurar el vaciado del filtro sobre el que se aplica. Después de vaciar el filtro debe cerrarse con close().

Además, el uso del filtro permite la utilización del método void newLine() para insertar un salto de línea a la hora de escribir en un fichero. En función del sistema operativo con el que se trabaje el método inserta el carácter adecuado de salto de línea.

Clase File

Clase

java.io.File.

Esta clase modela a un fichero o directorio del sistema local de la máquina donde esta instalado el J2SE. Contiene métodos que permiten realizar multitud de operaciones sobre los ficheros y directorios. Suele emplearse junto con las clases asociadas a los flujos de lectura y escritura.

Clase asociada

La clase File hereda directamente de java.lang.Object.

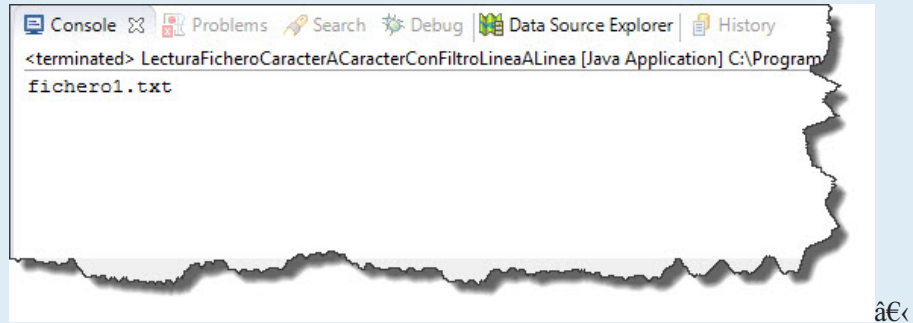
Métodos principales

- String getName(): devuelve nombre del fichero o directorio asociado al objeto File sobre el que se aplica.

En todos los ejemplos se parte del objeto `f1` construido en el apartado anterior. Apunta a un fichero cuya ruta es `c:\cursos\fichero1.txt`

```
System.out.println(f1.getName());
```

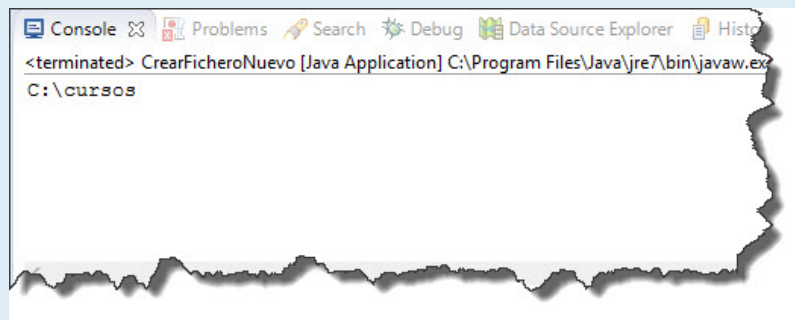
Por consola:



- String `getParent()`: devuelve el nombre del directorio inmediatamente superior que contiene al fichero o directorio sobre el que se aplica. Se incluye su ruta completa.

```
System.out.println(f1.getParent());System.out.println(f1.getParent());
```

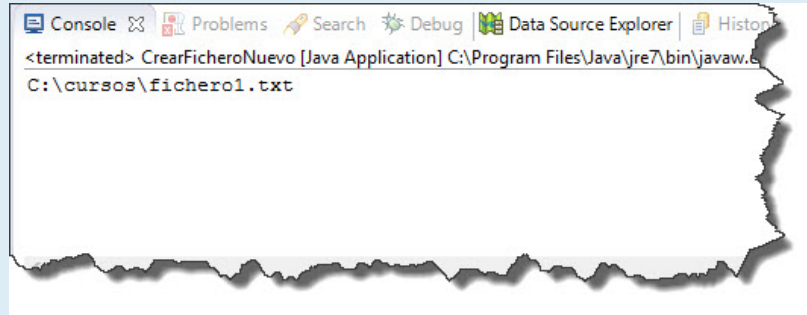
Por consola:



- String `getPath()`: devuelve la ruta asociada al fichero o directorio sobre el que se aplica. En dicha ruta se incluye el propio fichero o directorio.

```
System.out.println(f1.getPath());
```

Por consola:



- boolean `isFile()`: devuelve true si el File sobre el que se aplica representa a un fichero. Si no, false.
- boolean `isDirectory()`: ídem, pero si representa a un directorio.
- boolean `exists()`: devuelve true si el fichero o directorio existen en el sistema de ficheros local.
- boolean `canRead()`: devuelve true si el fichero o directorio puede leerse, y false si no. Un fichero es siempre leíble a menos que no exista en el sistema de ficheros local.
- boolean `setReadOnly()`: establece como de sólo lectura el fichero o directorio. Devuelve true si el proceso se realiza con éxito y false en caso contrario.
- boolean `canWrite()`: devuelve true si el fichero o directorio puede escribirse, y false si no. Por defecto un fichero o directorio son escribibles. Se hacen no escribibles aplicándoles el método “boolean `setReadOnly()`”
- boolean `delete()`: se utiliza para eliminar ficheros o directorios. Devuelve true si la eliminación ha sido correcta y false en caso contrario. Si se aplica sobre un directorio, debe estar vacío.

Si se tiene un fichero de nombre `miFichero` ubicado en `c:\cursos\ficheros`, las líneas:

```
String objetivo = "C:\\cursos\\ficheros\\miFichero.txt";
File f = new File(objetivo);
System.out.println(f.delete());
```

eliminan el fichero del sistema local y muestran por consola true.

Si se tiene un subdirectorio vacío de nombre `subficheros`, y se sustituye la String objetivo por `c:\\cursos\\ficheros\\subficheros`, también se eliminaría del sistema local y devolvería true.

Si el subdirectorio contiene algún fichero u otros directorios, no se elimina y devuelve false. Debe trabajarse junto con el método `File[] listFiles` para conseguir su eliminación. Primero debe eliminarse su contenido y luego se elimina el contenedor, en este caso, el directorio `subficheros`.

- `long length()`: devuelve el tamaño del fichero en bytes.
- `long lastModified()`: devuelve información temporal (el número de milisegundos transcurridos desde el “epoch” o 1 de Enero de 1970 a las 00:00:00 GMT) que permite saber, utilizando el constructor adecuado de `java.util.Date`, cuando se modificó por última vez un fichero o directorio.
- `String[] list()`: devuelve un array de `String` que contiene los nombres de los ficheros o directorios, incluida su ruta, del directorio sobre el que se aplica.
- Si se aplica sobre un objeto `File` que apunte a un directorio vacío, el número de elementos del array es cero. Al intentar acceder a su contenido se produciría una `NullPointerException`.
- Si se aplica sobre un objeto `File` que apunta a un fichero, devuelve `null`.

Sea un directorio **cursos** que cuelga del disco duro, que contiene tres ficheros 1.txt, 2.txt, 3.txt y dos subdirectorios sub1 y sub2 que contienen ficheros.
Si se aplica `list()` a un objeto `File` que apunta al directorio **cursos** se obtiene un array de `String` cuyos elementos son:

1.txt 2.txt 3.txt sub1 sub2

- `File[] listFiles()`: ídem anterior, pero devolviendo un array de objetos `File`. Ojo que si se aplica sobre un objeto `File` que apunta a un fichero, devuelve `null`. MUY USADO.
- `boolean mkdir()`: sirve para crear un directorio en el sistema local asociado al objeto `File` sobre el que se aplica. Devuelve `true` si el proceso se ha realizado correctamente.
- Si ya existe un directorio en el sistema local con la misma ruta que el que se pretende crear, no se elimina el del sistema local, simplemente, no se crea el asociado al objeto `File`.
- `URL toURL()`: crea un objeto de `java.net.URL` en base al objeto `File` sobre el que se aplica. Este paquete se estudiará en temas posteriores.

Ejercicios

Ejercicio1. Leer un fichero

30

Crear una clase pública de nombre **EjercicioFicheros1** que contenga sólo al método main y que haga lo siguiente:

- **Pedir** al usuario la **ruta** de un **fichero**. El **usuario** introducirá por teclado `c:\cursos\pruebaFicheros.txt`.
- **Leer** el **fichero** línea a línea y **mostrarlas** por consola **excepto aquellas que empiecen con el carácter '+' o '-'**, ya que éstas **deben escribirse en un fichero de texto** ubicado en el disco duro de nombre **resultados.txt**.
- **Mostrar** por consola el **número de líneas leídas** en el fichero inicial y el **número de líneas escritas** en el fichero `c:\resultados.txt`

Recomendaciones

Para leer, usar un flujo de lectura de caracteres, un filtro y el método adecuado. No olvidar cerrar el flujo.

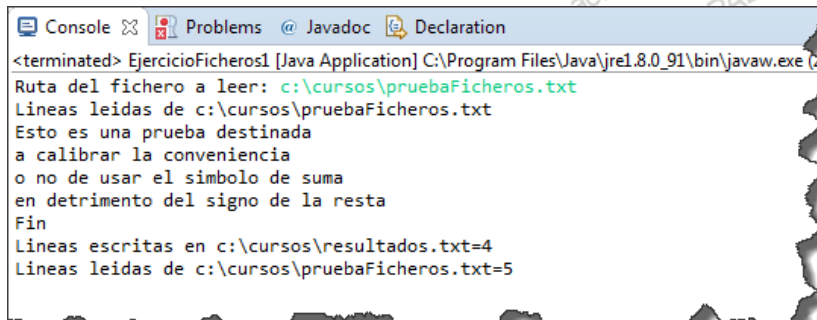
Para escribir, usar un flujo de escritura de caracteres, un filtro y el método adecuado. No olvidar vaciar y cerrar el filtro.

Lo que se necesita para comenzar

Para poder hacer es ejercicio se creará el fichero `c:\pruebaFicheros.txt`. Su contenido será el siguiente:

Esto es una prueba destinada
a calibrar la conveniencia
o no de usar el simbolo de suma
+ como herramienta de trabajo
en detrimento del signo de la resta
- Cucurrucucu
-Hola estudiante
+Adios estudiante
Fin

Datos a mostrar por consola:



```

<terminated> EjercicioFicheros1 [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe
Ruta del fichero a leer: c:\cursos\pruebaFicheros.txt
Lineas leidas de c:\cursos\pruebaFicheros.txt
Esto es una prueba destinada
a calibrar la conveniencia
o no de usar el simbolo de suma
en detrimento del signo de la resta
Fin
Lineas escritas en c:\cursos\resultados.txt=4
Lineas leidas de c:\cursos\pruebaFicheros.txt=5

```

Contenido del fichero resultados.txt:

- + como herramienta de trabajo
- Cucurruccucu
- Hola caracola
- +Adios caracola

Ejercicio 2. Copiar ficheros

20

Se trata de **copiar el contenido** de un fichero local existente que se le pide al usuario que **introduzca por teclado, en otro que todavía no existe** y cuya ruta se le pide también al usuario.

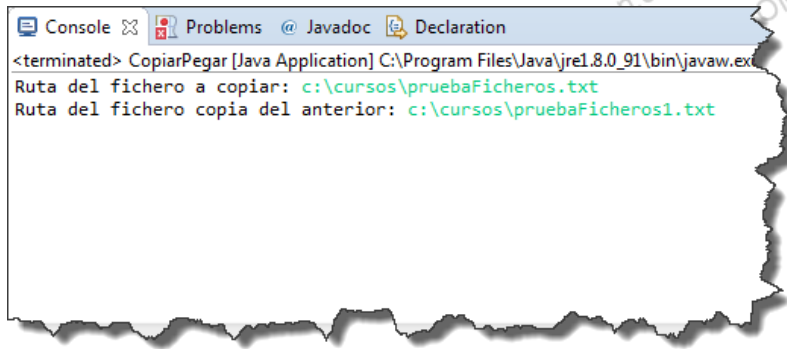
Recomendaciones

Para leer y escribir se utilizarán flujos y filtros de bytes.

Lo que se necesita para comenzar

El código tendrá una clase pública de nombre **CopiarPegar** y contendrá sólo al método main.

Datos a mostrar por consola



Ejercicio 3. Creación, edición y borrado de ficheros.

30

Realizar un programa que contenga una clase pública de nombre **EjercicioFile1** con un solo método: el main. Su código debe hacer lo siguiente:

- Crear dos objetos File f1 y f2 asociados a los ficheros **cucu.doc** y **borrame.txt** de contenido libre. Ambos se guardarán en **C:\carpeta1\carpeta2** (se creará la estructura de directorios previamente).
- Comprobar que los dos ficheros forman parte del sistema de ficheros local mediante el método exists() y:
 - Mostrar por consola sus tamaños en bytes y sus nombres originales.
 - Comprobar mediante un if y el método adecuado si el fichero cucu.doc puede leerse.
 - Comprobar mediante un if y el método adecuado si el fichero cucu.doc puede escribirse.
 - Mostrar por consola la carpeta que contiene a este fichero. Si dicha carpeta empieza por “c” debe mostrarse por consola el mensaje “El directorio padre empieza por c”; si no, nada.
 - Mostrar por consola la ruta asociada a los dos ficheros.
 - Eliminar el fichero “borrame.txt” comprobando que se ha eliminado correctamente.

Lo que se necesita para comenzar

Se deben tener creadas las rutas y los dos ficheros.

Datos a mostrar por consola

```

<terminated> EjercicioFile1 [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe
Tamaño del fichero cucu.doc=11527 bytes
Tamaño del fichero borrame.txt=25 bytes
Fichero leible
Fichero escribible
Directorio padre: C:\carpeta1\carpeta2
El directorio padre empieza por c
Ruta: C:\carpeta1\carpeta2\cucu.doc
Fichero "borrame.txt" borrado

```

Ejercicio 4. Manejo de ficheros

30

Realizar un programa que contenga una clase pública de nombre **EjercicioFile2** con un solo método: el main. Su código debe hacer lo siguiente:

- Crear un objeto File asociado al directorio carpeta1 que cuelga del disco duro en el sistema de ficheros local. Guardar en dicho directorio dos ficheros de texto de contenido libre a los que se asignarán los nombres mortadelo.txt y filemon.txt
- Después de comprobar, con un if y el método adecuado, que carpeta1 se ha creado:
 1. Mostrar por consola su contenido.
 2. Crear un directorio llamado hola que cuelgue de carpeta1 y utilizando un if y el valor de retorno asociado al método que permite crear el directorio, comprobar que ha sido correctamente creado, mostrando por consola el mensaje "Directorio "hola" correctamente creado". Si no ha sido así, mostrar "Directorio "hola" no creado".
 3. Crear otra carpeta llamada adios que cuelgue de hola y comprobar del mismo modo que antes que ha sido correctamente creada.

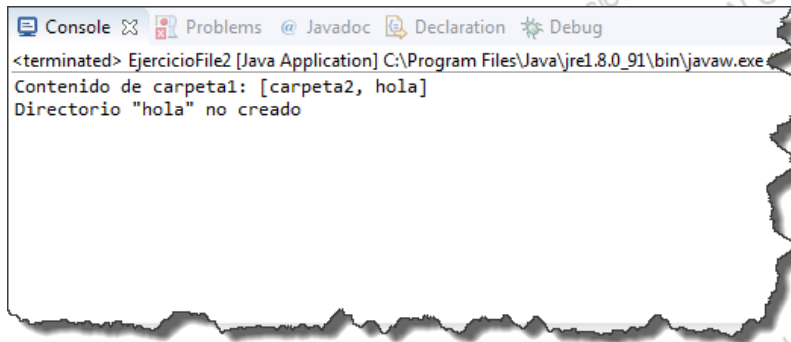
Datos a mostrar por consola

```

<terminated> EjercicioFile2 [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe
Contenido de carpeta1: [carpeta2]
Directorio "hola" correctamente creado
Directorio "adios" creado

```

Si se vuelve a ejecutar:



```
<terminated> EjercicioFile2 [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe
Contenido de carpeta1: [carpeta2, hola]
Directorio "hola" no creado
```

Ejercicio 5. Manejo de carpetas

25

Realizar un programa que contenga una clase pública de nombre **EjercicioFile3** con un solo método: el main. Su código debe hacer lo siguiente:

- Mostrar por consola el número de elementos del array de objetos File asociado al contenido de **carpeta1**. **Para obtener el array se empleará el método listFiles()**.
- Mostrar por consola el contenido del array.
- Eliminar el contenido del directorio **carpeta1** que cuelga del disco duro del sistema de ficheros local siempre y cuando esté vacío. Si la carpeta tiene ficheros no se elimina. Teniendo en cuenta que su contenido es el siguiente:
 - El directorio carpeta2, que deberá estar vacío. **Si hay algún fichero, se eliminará a mano.**
 - El directorio hola, que no está vacío.
 - El fichero mortadelo.txt.
 - El fichero filemon.txt.

Debe ocurrir lo siguiente:

- Sólo se eliminarán carpeta2, mortadelo.txt y filemon.txt
- No se elimina hola ya que no está vacío.
- Repetir el primer y segundo apartado y observar que el número de elementos del array es el mismo que antes.

Datos a mostrar por consola

```
<terminated> EjercicioFile3 [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe  
Numero elementos array=4  
c:\carpeta1\carpeta2 false c:\carpeta1\filemon.txt true c:\carpeta1\hola false c:\carpeta1\mortadelo.txt true  
Numero elementos despues de eliminar=4  
c:\carpeta1\carpeta2 c:\carpeta1\filemon.txt c:\carpeta1\hola c:\carpeta1\mortadelo.txt
```

Si se vuelve a ejecutar:

```
<terminated> EjercicioFile3 [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (2)  
Numero elementos array=1  
c:\carpeta1\hola false  
Numero elementos despues de eliminar=1  
c:\carpeta1\hola
```

Recursos

Enlaces de Interés



<http://www.javahispano.org/portada/2011/8/1/java-y-las-redes-introduccion-a-las-redes-a-javaio-javazip.html>

<http://www.javahispano.org/portada/2011/8/1/java-y-las-redes-introduccion-a-las-redes-a-javaio-javazip.html>

Clases de java.io. Gestión de ficheros



<http://programacion.com/java/cursos.htm#basico>

<http://programacion.com/java/cursos.htm#basico>

Paquete java.io



<http://docs.oracle.com/javase/tutorial/essential/io/index.html>

<http://docs.oracle.com/javase/tutorial/essential/io/index.html>

Clases de java.io. Gestión de ficheros

Glosario.

- **Búfer:** En informática, un bufer o buffer de datos es un espacio de la memoria en un disco o en un instrumento digital reservado para el almacenamiento temporal de información digital, mientras que está esperando ser procesada. Por ejemplo, un analizador TRF tendrá uno o varios buffers de entrada, donde se guardan las palabras digitales que representan las muestras de la señal de entrada. El Z-Buffer es el usado para el renderizado de imágenes 3D.