MEDAC
Instituto Oficial de Formación Profesional

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMAS

Flujos

02

ÍNDICE

 ISBN: 978-84-18983-18-4

OBJETIVOS

Identificar Clases de flujos basadas en el manejo de bytes.

Identificar Clases de flujos basadas en el manejo de caracteres.

Saber elegir la clase de flujo correspondiente acorde al requerimiento.

Saber diferenciar entre clases de flujos de lectura y escritura.

/ 1. Introducción y contextualización práctica

EEn el siguiente tema realizaremos un repaso de todas las clases del paquete "java.io" relacionadas con el tema de los flujos de datos o "Streams". En el tema anterior ya hicimos una breve explicación de algunos de ellos como FileInputStream, FileOutputStream, FileReader, FileWriter, BufferedInputStream... dada su clara relación con el tema de ficheros. En el tema actual profundizaremos en este tipo de clases dependiendo del uso que queramos hacer de ellas: Tuberías, Buffers, bloques de información, filtrado, etc.

Fig. 1: Tipos de flujos

Escucha el siguiente audio en el que planteamos el caso práctico que iremos resolviendo a lo largo de esta unidad.

Audio Intro. Selector de líneas

https://bit.ly/2N1nR1f

/ 2. Definición y tipos de ficheros

Definimos flujo de datos o "Streams" como una secuencia ordenada de información que posee un recurso de entrada (flujo de entrada) y un recurso de salida (flujo de salida). Los Streams se basan en la unidireccionalidad, es decir, se usan sólo para leer o sólo para escribir, pero no ambas situaciones simultaneas.

Según el tipo de información a tratar, podemos dividir los Streams en diferentes categorías:

- Tratamiento de ficheros (estudiado en el tema 1)
- Tratamiento con Buffer (estudiado en el tema 1)
- Tratamiento con Arrays.
- Tratamiento con tuberías.
- Tratamiento con análisis (Parsing).
- Tratamiento con bloques de información.

Según su funcionalidad y usabilidad exponemos una tabla a modo de esquema para tener una imagen global de los distintos tipos de clases que nos podremos encontrar:

	Método	
USABILIDAD	**BYTES (E/S)**	**CARACTERES (E/S)**
Ficheros	FileInputStream, FileOutputStream	FileReader, FileWriter
Arrays	ByteArrayInputStream, ByteArrayOutputStream	CharArrayReader, CharArrayWriter
Tuberías	PipedInputStream, PipedOutputStream	PipedReader, PipedWriter
Buffer	BufferedInputStream, BufferedOutputStream	BufferedReader, BufferedWriter
Análisis	PushbackInputStram, StreamTokenizer	PushbackReader, LineNumberReader
Información	DataInputStream, DataOutputStream, PrintStream	PrintWriter

Tabla 1: Streams según su usabilidad.

/ 3. Clases de flujos basados en tuberías

En primer lugar, tendremos en cuenta que el concepto de tuberías en lenguaje de programación Java es distinto a otros sistemas. Las tuberías en lenguaje de programación Java, proporcionan la capacidad de **comunicar dos hilos** (threads) ejecutándose en la misma máquina virtual (JVM). Esto significa que las tuberías pueden ser tanto orígenes, como destinos de datos. En sistemas Unix como Linux, dos procesos que se ejecutan en lugares distintos de memoria, pueden comunicarse a través de una tubería. Este no es el caso. En Java, las partes que se ejecutan deben pertenecer al mismo proceso y deben ser hilos diferentes.

Para entender bien este tipo de clase de flujos, estudiaremos un ejemplo, el cual nos ayudará a escribir datos desde un hilo y a leerlos desde otro, dentro del mismo proceso.

```
final PipedOutputStream salida = new PipedOutputStream();
final PipedInputStream  entrada  = new PipedInputStream(salida);
Thread thread1 = new Thread(new Runnable() {
   @Override
   public void run() {
      try {
         salida.write("Hola por aquí!".getBytes());
      } catch (IOException e) {
      }
   }
});
```

Código 1: Uso Streams con tuberías I

En la imagen anterior podemos observar la instanciación de ambos objetos Streams. El primero de ellos hace referencia a un objeto de salida, el cual, podría ser por ejemplo una escritura de un fichero, pero en este caso simplemente simularemos la escritura de la frase: "Hola por aquí!". Justo a continuación instanciamos la tubería de entrada pasándole por parámetro la salida creada previamente. De esta forma la "tubería" quedará conectada y tendremos acceso al fichero de salida que está realizando la escritura. Por último, instanciamos un hilo "thread" y en su interior escribimos los bytes referenciados a dicha frase expuesta con anterioridad.

```
Thread thread2 = new Thread(new Runnable() {
   @Override
   public void run() {
      try {
         int unByte = entrada.read();
         while(unByte!= -1){
            System.out.print((char) unByte);
            unByte = entrada.read();
         }
      } catch (IOException e) {
      }
   }
});
thread1.start();
thread2.start();
```

Código 2: Uso Streams con tuberías II

Audio 1. Implementación PipedStreams
https://bit.ly/3e7LRM8

/ 4. Flujos basados en arrays

En este apartado podremos analizar algunas clases basadas en el flujo de Arrays. Como de costumbre, dividiremos el acceso a dichas clases entre:

- **Acceso basado en bytes:** ByteArrayInputStream, ByteArrayOutputStream.
- **Acceso basado en caracteres:** CharrArrayReader, CharArrayWriter.

Hablaremos en este caso de las clases orientadas y basadas en el funcionamiento con caracteres. Para ello veremos como trabajar con las clases CharrArrayReader y CharrArrayWriter:

La clase CharrArrayReader nos permite leer el contenido de un array de caracteres (char) como un Stream de caracteres. Esta clase nos será útil en casos que tengamos información en un array de caracteres y necesitemos pasarlo a algún componente, el cual solo pueda ser leído desde una clase Reader o una subclase de la misma. En este caso simplemente encapsularemos el array en un CharArrayReader y lo pasaremos al componente.

```
char[] chars = "hola amigo".toCharArray();
CharArrayReader charArrayReader = new CharArrayReader(chars);
int data = 0;
try {
   data = charArrayReader.read();
   while(data != -1) {
      //OPERACIONES
      System.out.println((char)data);
      data = charArrayReader.read();
   }
} catch (IOException e) {
   e.printStackTrace();
}
charArrayReader.close();
```

Código 3: Uso de CharArrayReader

Básicamente para este ejemplo instanciaríamos un array de caracteres, en este caso, llamado "chars". Después pasaríamos por parámetro dicho array para construir un nuevo elemento, concretamente, sería construir un objeto de la clase CharArrayReader. De esta forma, tendríamos en esta clase el contenido de todo el array. Después, y esto ya es a nuestra elección, podríamos ir leyendo el objeto "charArrayReader" byte a byte para realizar lo que se desee en la parte del comentario "OPERACIONES". En este caso se ha imprimido por pantalla el array.

Por último ejecutaríamos el método close() liberando los recursos, tal y como estamos habituados a hacer.

Vídeo 1. " CharArrayReader"

https://bit.ly/37wfkgi

/ 5. Caso práctico 1: "Tuberías"

Fig. 2: Threads

Planteamiento: Pedro necesita elaborar un aplicativo que le permita leer y escribir información de forma efectiva bajo un mismo proceso. Lo que quiere es, básicamente, leer la información para manipularla al mismo tiempo que la escribe.

Nudo: Después de haber investigado cuáles son sus opciones, Pedro encuentra las clases PipedInputStream y PipedOutputStream.

Desenlace: nuestro compañero necesitará tener en funcionamiento 2 hilos bajo un mismo proceso; en este caso, deberá instanciar 2 objetos. Primero instanciará un PipedOutputStream:

```
final PipedOutputStream output = new PipedOutputStream()
```

Después necesitará incluir este output en un PipedInputStream:

```
final PipedInputStream  input  = new PipedInputStream(output)
```

Una vez creadas ambas instancias, deberá encapsular las operaciones de escritura y lectura en 2 threads. El primer hilo irá escribiendo en un documento o por pantalla (según decida Pedro); el segundo, al estar conectado al primero por la instancia que hemos visto anteriormente, realizará las operaciones de lectura.

Solo quedaría iniciar ambos hilos con la línea:

- nombreDelThread1.start();
- nombreDelThread2.start();

De esta forma, la lógica comenzaría a funcionar automáticamente, escribiendo el contenido encapsulado en la definición del thread1 y leyendo y manipulando la información escrita en el thread2.

/ 6. Clases de análisis para flujos de datos I

En este punto estudiaremos algunas clases que nos serán de utilidad para analizar ciertas partes de los flujos de datos. A continuación, profundizaremos en las clases: **PushBackInputStream, StreamTokenizer, PushbackReader, LineNumberReader.**

Las clases **PushBackInputStream y PushbackReader** están diseñadas para el análisis de datos previo de un InputStream. En algunas ocasiones se necesita leer algunos bytes con anticipación para saber qué se aproxima, para así poder interpretar el byte actual.

Estos bytes leídos con anterioridad, serán de nuevo 'empujados' a la secuencia, para que posteriormente, los leamos cuando se haga de nuevo read().

La diferencia básica entre las dos clases mencionadas anteriormente, es que básicamente PushbackInputStream está orientada a trabajar byte a byte y PushbackReader a leer directamente con caracteres.

A continuación, mostraremos un fragmento de código donde veremos cómo algunos bytes pueden ser leídos con anterioridad, y más tarde devueltos a la secuencia o al flujo de bytes.

```
try {
   PushbackReader pushbackReader = new PushbackReader(
         new FileReader("C:\\temp\\pruebas\\pruebas2.txt"));
   int data = pushbackReader.read();
   System.out.println((char)data);
   pushbackReader.unread(data);
   data = pushbackReader.read();
   System.out.println((char)data);
   pushbackReader.close();
} catch (FileNotFoundException e) {
   e.printStackTrace();
} catch (IOException e) {
   e.printStackTrace();
}
```

Código 4: PushbackReader

Vemos como instanciamos un objeto *PushBackReader*, y para ello necesitamos a su vez instanciar un objeto FileReader. Este objeto FileReader tendremos que instanciarlo a su vez indicando la ruta del fichero a tratar. Imaginemos que la primera letra del fichero es "D".

En las siguientes líneas usamos el conocido método read() y justo después mostramos por pantalla el byte leído haciendo casting de "char". En este punto veremos por pantalla la letra "D". Justo en la siguiente línea nos aparece un método nuevo **unread(data)**. Es el método clave de esta clase definida. Este método devuelve al stream de datos el byte que hemos leído con anterioridad, de tal forma que si volvemos a hacer un read() y mostramos por pantalla, obtendremos el mismo byte que antes. En este caso la letra "D".

/ 7. Clases de análisis para flujos de datos II

En este punto del tema analizaremos la clase StreamTokenizer. Para entender la funcionalidad de dicha clase y el significado de la misma, debemos de entender qué es un "token". Un token viene a estar relacionado con un "fragmento", "una ficha", "un trozo" ... La **clase StreamTokenizer** tiene la capacidad de analizar el fichero por 'trozos' o 'fragmentos'. Dichos fragmentos más tarde tendremos que evaluarlos y comprobar si son palabras o números. De hecho, StreamTokenizer es capaz de reconocer identificadores: números, comillas, espacios, etc., lo cual nos puede ser de gran utilidad en aplicativos de análisis de ficheros según su tipología sea números o caracteres.

```
StreamTokenizer streamTokenizer = new StreamTokenizer(
    new StringReader("Hola mi edad es 45"));
  try {
    while(streamTokenizer.nextToken() != StreamTokenizer.TT_EOF){
      if(streamTokenizer.ttype == StreamTokenizer.TT_WORD) {
        System.out.println(streamTokenizer.sval);
      } else if(streamTokenizer.ttype == StreamTokenizer.TT_NUMBER) {
        System.out.println(streamTokenizer.nval);
      } else if(streamTokenizer.ttype == StreamTokenizer.TT_EOL) {
        System.out.println();
      }
    }
  } catch (IOException e) {
    e.printStackTrace();
  }
```

Código 5: StreamTokenizer

En el cuadro de código anterior podemos ver los distintos casos de uso de un objeto StreamTokenizer. En primer lugar, instaciamos el objeto StreamTokenizer usando el constructor por medio del cual le pasamos un StringReader.

Para ello, pasamos por parámetro del StringReader la cadena de texto: "Hola mi edad es 45".

- StreamTokenizer dispone de algunos métodos estáticos que nos dan información de la tipología de los distintos Tokens:
    - **TT_EOF:** indica el final del fichero (End Of File)
    - **TT_EOL:** indica el final de la línea (End Of Line)
    - **TT_WORD:** indica que el token es de tipo palabra, conjunto de letras.
    - **TT_NUMBER: i**ndica que el token evaluado es un número o una asociación de ellos.

Por último, comentar la lógica básica de este código: disponemos de un bucle de tipo "while" que estará ejecutándose mientras el siguiente token no sea el final de nuestro fichero. A continuación, se observan una serie de condicionales:

- El primero: si el token es de tipo palabra, mostraremos por pantalla dicha palabra.
- El segundo condicional: si es de tipo número, mostraremos por pantalla el número.
- El tercero: si es de tipo final de línea se imprimirá retorno de carro.

/ 8. Clases de análisis para flujos de datos III

En este último punto dedicada al análisis de flujos de datos, veremos la clase **LineNumberReader.**

Esta clase es básicamente un objeto de tipo BufferedReader que almacena y cuenta el número de líneas leídas de caracteres. Está orientada a trabajar y analizar líneas completas.

Básicamente esta clase empieza leyendo por la primera línea con el contador a 0 y cada vez que encuentra un retorno de carro incrementa su valor sumando 1.

Existen varios métodos clave para esta clase:

- **getLineNumber():** este método como su nombre indica, nos devolverá el número de la línea en la que estemos actualmente leyendo.
- **setLineNumber():** este método es realmente interesante ya que se le puede pasar como parámetro un entero, y se convertirá en la línea actual.

Comentar que uno de los métodos clave de esta clase es el método: readLine() el cual nos devolverá un conjunto de caracteres al cual se le debe asignar a una variable String.

Evidentemente dispone de su método read() como ya hemos visto en clases anteriores. Veamos un ejemplo de esta clase:

```
try {
  LineNumberReader lineNumberReader =
        new LineNumberReader(new FileReader("C:\\temp\\pruebas\\pruebas2.txt"));
  String line = lineNumberReader.readLine();
  while(line != null) {
     System.out.println("Contenido de la linea numero:"+ lineNumberReader.getLineNumber());
     System.out.println(line);
     line = lineNumberReader.readLine();
  }
  lineNumberReader.close();
} catch (IOException e) {

  e.printStackTrace();

}
```

Código 6: LineNumberReader

Este ejemplo de código de clase LineNumberReader es un caso muy básico. En primer lugar, se instancia el objeto con un nuevo objeto FileReader, el cual es instanciado con la ruta de un fichero que se ha preparado con anterioridad. Este es un fichero con varias líneas.

Justo después de la instanciación vemos la ejecución del método readLine(), con su asignación a una variable de tipo String. Ahí quedará almacenado el contenido de la primera línea. A continuación, se ejecutará el bucle while mientras se tengan líneas que recorrer, e se irá mostrando por pantalla tanto el número de la línea en la que se está posicionado, como su contenido, a través de: getLineNumber() y Syestem.out.println(line). De esta forma iremos mostrando el contenido del fichero completo línea tras línea.

/ 9. Clases para el tratamiento de información I

En este caso hablaremos de una clase que nos permite procesar tipos primitivos de Java: int, float, long etc. Nos referimos a la clase DataInputStream, que nos ofrece una gran ventaja en comparación con InputStream, ya que esta última sólo procesará los bytes sin asociación de ninguna tipología.

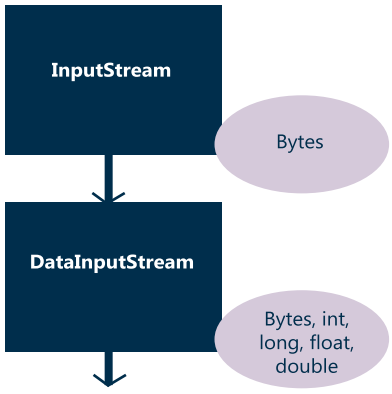

Fig. 3: Esquema tipos primitivos

Esta clase nos será de utilidad siempre y cuando nuestra lectura esté orientada a los tipos primitivos de Java citados en el párrafo anterior. Se suele usar la clase DataInputStream para leer ficheros que previamente han sido escritos con DataOutputStream. DataInputStream al ser una subclase de InputStream hereda los métodos que ésta pone a su disposición. Es por esto que si queremos leer byte a byte con el método read(), evidentemente lo tendremos también disponible, así como la lectura con un array de bytes.Algo a tener en cuenta cuando leemos tipos de datos primitivos es que no hay forma de distinguir la lectura de un número -1 a la lectura de fin de flujo que es también -1, por lo tanto, es muy importante en este tipo de lectura saber qué tipo de datos vamos a leer y qué orden llevan.

Veamos un ejemplo de uso de DataInputStream con tipos de datos primitivos:

```
try {
   DataInputStream dataInputStream = new DataInputStream(
       new FileInputStream("C:\\temp\\pruebas\\ficheroBinario.data"));
   int    aByte   = dataInputStream.read();
   int    anInt   = dataInputStream.readInt();
   float  aFloat  = dataInputStream.readFloat();
   double aDouble = dataInputStream.readDouble();
   dataInputStream.close();
} catch (FileNotFoundException e) {
   e.printStackTrace();
} catch (IOException e) {
   e.printStackTrace();
}
```

Código 7: DataInputStream tipos primitivos.

/ 10. Clases para el tratamiento de información II

En este apartado continuaremos profundizando en la clase DataInputStream. Este objeto se instancia con un objeto de la clase FileInputStream como origen de datos, después de esto, los tipos primitivos de Java serán leídos desde este origen. A continuación, veremos una puesta en práctica de cómo usar **DataOutputStream y DataInputStream**:

```
try {
   DataOutputStream dataOutputStream = new DataOutputStream(
       new FileOutputStream("C:\\temp\\pruebas\\data.bin"));
   dataOutputStream.writeInt(123);
   dataOutputStream.writeFloat(123.45F);
   dataOutputStream.writeLong(789);
   dataOutputStream.close();
```

Código 8: DataOutputStream modo de uso.

Tal y como dijimos con anterioridad, para poder realizar una lectura de un fichero por medio de la clase DataInputStream, antes hemos tenido que realizar la escritura ordenada de dicho fichero y debemos ser conocedores del tipo de datos y la cantidad de ellos que ha sido insertada. Para ello usamos DataOutputStream.

El objeto es instanciado en su constructor con un FileOutputStream con la ruta del fichero que vamos a escribir.

En las líneas que siguen podemos observar como escribimos en el fichero distintos tipos primitivos de Java usando diferentes métodos:

- **writeInt() :** con este método introduciremos enteros, por lo tanto solo aceptara entero como parámetro.
- **writeFloat() :** método que acepta como parámetro un objeto de tipo float. Igual que en el caso anterior solo podremos pasarle dicho tipo primitivo.
- **writeLong() :** en este caso solo podremos introducir un objeto de tipo long.

En la última línea podemos observar cómo simplemente cerramos nuestro flujo y liberamos recursos.

```
DataInputStream dataInputStream =
        new DataInputStream(
              new
FileInputStream("C:\\temp\\pruebas\\data.bin"));

   int   entero    = dataInputStream.readInt();
   float numeroFloat = dataInputStream.readFloat();
   long  numeroLong    = dataInputStream.readLong();
   dataInputStream.close();
   System.out.println("El numero entero es: " + entero);
   System.out.println("El objeto de tipo float es: " + numeroFloat);
   System.out.println("El objeto long es: " + numeroLong);
} catch (FileNotFoundException e) {
   e.printStackTrace();
} catch (IOException e) {
   e.printStackTrace();
}
```

Código 9: DataInputStream recuperando datos.

A continuación, mostraremos como recuperar esa información que hemos almacenado en el fichero binario "data.bin".

Video 2. "DataOutputStream"
https://bit.ly/3d7r2ik

/ 11. Caso práctico 2: "Extracción de números"

Planteamiento: un documento contiene información sobre el nombre, los apellidos y la edad de una serie de usuarios de una base de datos de una web comercial. Queremos obtener tan solo las edades que están escritas en números para, posteriormente, procesarlas.

Fig. 4: Tokens de piezas

Nudo: tras realizar un estudio de las clases que nos pueden ser útiles para este caso llegamos a la conclusión de que la clase StreamTokenizer nos puede ser de gran ayuda.

Desenlace: en primer lugar, tendremos que instanciar un objeto de clase reader apuntando al fichero:

```
Reader reader = new FileReader("ruta/fichero.bin")
```

Después crearemos una instancia de nuestra clase StreamTokenizer y le pasaremos el objeto reader que hemos creado previamente.

Una vez instanciado nuestro StreamTokenizer, simplemente tendremos que recorrer dicho fichero en busca de las edades en las que estamos interesados.

Recorremos el fichero completo de la siguiente forma:

```
while(streamTokenizer.nextToken() != StreamTokenizer.TT_EOF)
```

Después de preparar el bucle while solo nos faltaría acotar las apariciones de los números referentes a las edades dentro del fichero. Para hacer esto, usaremos el siguiente condicional:

```
if(streamTokenizer.ttype == StreamTokenizer.TT_NUMBER)
```

Evidentemente, otro tipo de dato que no sea número no entrará en esta comprobación. Por lo tanto, en este punto tendremos la información que queremos. Justo ahí, dentro de las llaves de este condicional, debemos indicar las operaciones que queramos realizar con dichas edades.

/ 12. Resumen y resolución del caso práctico de la unidad

En esta unidad hemos navegado a través de las distintas clases que nos sirven de utilidad para tratar flujos. Hemos profundizado en clases que nos sirven de base para tratar Arrays, también hemos aprendido a realizar el manejo de flujos con Buffer, así como conectar tuberías de entrada y salida con varios threads.

Más adelante se estudiaron algunas clases que nos servirán de análisis de flujos de datos con funcionalidades diferentes, funcionalidades que serán de utilidad para analizar la información con cierta previsión de flujo, o para dividirla en fragmentos a los cuales hemos llamado tokens.

Por último, aprendimos a leer y escribir ficheros de tipo binario en cuyo interior podemos encontrar objetos de tipo primitivo de Java. Es por todo ello, que, con el conocimiento aprendido, debemos de saber elegir la clase y funcionalidad que se adecue al modelo de implementación que queremos desarrollar.

Resolución del caso práctico de la unidad.

Juán afronta el reto que se le plantea a modo de desarrollo. Nuestro compañero deber realizar el acceso a un fichero en el que tendrá que ir escogiendo cierta información. En este caso no deberá de leer el documento íntegro y su labor consiste en escoger las líneas 0,2,4... y así hasta el final del fichero.

Tras realizar un estudio de las clases que puede utilizar se escoge la clase LineNumberReader.

Con esta clase Juan podrá indicar el fichero que quiere recorrer, y podrá coger el contenido de las líneas intercaladas que él desee.

/ 13. Webgrafía

Oracle web oficial:
https://docs.oracle.com/javase/8/docs/api/?java/io"Fig. 4: Tokens de piezas" en la página 12