ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMAS

Flujos

Flujo de datos ("*Streams*"): secuencia ordenada de información que posee un recurso de entrada (flujo de entrada) y un recurso de salida (flujo de salida). **Unidireccionalidad:** 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**
- Tratamiento con **Buffer**
- Tratamiento con **Arrays**.
- Tratamiento con **tuberías**.
- Tratamiento con **análisis** (Parsing).
- Tratamiento con **bloques** de información.

Según su **funcionalidad y usabilidad**:

STREAMS		
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**

Tuberías

En Java, proporcionan la capacidad de **ejecutar dos hilos** (threads) ejecutándose en la **misma máquina virtual** (JVM). Esto significa que las tuberías pueden ser **tanto orígenes**, **como destinos** de datos. En **Java**, las partes que se ejecutan deben **pertenecer al mismo proceso** y deben ser **hilos diferentes**. De modo que proceso se relaciona con un mismo lugar de memoria. En otras palabras, para comunicar 2 tuberías, una de entrada y otra de salida (por ejemplo), deberán de ejecutarse bajo el mismo proceso y con 2 threads independientes.

Ejemplo:

escribir datos desde un hilo y a leerlos desde otro, dentro del mismo proceso:

```
package streamscontuberias;
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;
import java.util.logging.Level;
import java.util.logging.Logger;
public class StreamsConTuberias {
    public static void main(String[] args) throws IOException {
        /*
        Instanciamos los objetos para entrada y salida de flujos
        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 chavales!”.
         */
        final PipedOutputStream salida = new PipedOutputStream();
        /*
        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.
         */
        final PipedInputStream entrada = new PipedInputStream(salida);
        String textoSalida = "Hola chavales!";
        /*
```

```
 Creamos 2 hilos:
    hilo1 para escribir
    hilo2 para leer
En el hilo1 escribimos los bytes referenciados
 */
 Thread hilo1 = new Thread(new Runnable() {
    @Override
    public void run() {
       try {
          /*
            Dentro del hilo escribimos los bytes en la salida
          */
          salida.write(textoSalida.getBytes());
       } catch (IOException ex) {
          Logger.getLogger(StreamsConTuberias.class.getName()).log(Level.SEVERE, null, ex);
       }
    }
 });
 /*
En el hilo2 leremos "al mismo tiempo" en la misma tubería
el objeto PipedInputStream nombrado entrada
 */
 Thread hilo2 = new Thread(new Runnable() {
    @Override
    public void run() {
       try {
          /*
          Como en entrada tenemos como argumento salida,
          podremos leer lo escrito en salida:
          leemos directamente del PipedInputStream cuya variable
          definimos “entrada”, referenciada directamente a la salida,
          por lo cual tenemos acceso a ella.
```

```
            Básicamente lo que hacemos es leer directamente Byte a Byte
            del fichero de salida al mismo tiempo que se está escribiendo.
             */
            int unByte = entrada.read();
              while (unByte != -1) {
                 System.out.print((char) unByte);
                 unByte = entrada.read();
              }
           } catch (IOException e) {
           }
        }
      });
      /*
     Sólo nos queda lanzar los dos hilos
     ejecutamos ambos hilos, teniendo en cuenta que “hilo2” va a tener acceso
     de lectura a “hilo1” y, por lo tanto, va a mostrar por pantalla justo
     lo que “hilo1” está escribiendo.
      */
     hilo1.start();
      hilo2.start();
   }
}
```

Flujos basados en arrays

• Acceso basado en caracteres: **CharrArrayReader, CharArrayWriter**.

• Acceso basado en bytes: **ByteArrayInputStream**, **ByteArrayOutputStream**.

caracteres

CharrArrayReader y CharrArrayWriter

CharrArrayReader: 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.

Instanciaremos un **CharArrayWriter** y escribiremos un texto. Después **pasaríamos por parámetro** dicho objeto 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 objeto CharArrayWriter. Después podríamos ir **leyendo el objeto "charArrayReader" byte a byte** para realizar lo que se desee en la parte del comentario "BLOQUE". En este caso se ha **imprimido por pantalla** el array leído desde el objeto CharArrayReader.

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

```
package escrituralecturaarraycaractereschararrayreader.writer;

import java.io.CharArrayReader;

import java.io.CharArrayWriter;

import java.io.IOException;

import java.util.logging.Level;

import java.util.logging.Logger;

public class EscrituraLecturaArrayCaracteresCharArrayReaderWriter {

    public static void main(String[] args) {

        CharArrayWriter writer = new CharArrayWriter();

        CharArrayReader reader;

        /*

        en data asignamos la información que leeremos

         */

        int data = 0;

        try {

            /*

            Escribimos en el escritor de array de caracteres

             */
```

```
            writer.write("Esto es escrito en un objeto CharArrayWriter");
            /*
            Usamos CharArrayReader a partir del mismo CharArrayWriter writer
            pasándolo como parámetro, convertido a array de caracteres
             */
            reader = new CharArrayReader(writer.toCharArray());
            /*
            leemos el primer byte, para luego seguir leyendo mientras
            no lleguemos al final del array de caracteres
             */
            data = reader.read();
            while (data != -1) {
                // BLOQUE
                System.out.println((char) data);
                // incrementamos el caracter leido para la próxima iteración
                data = reader.read();
            }
            /*
            Como no usaremos más el flujo de lectura de caracteres lo cerramos
             */
            reader.close();
        } catch (IOException ex) {
            Logger.getLogger(EscrituraLecturaArrayCaracteresCharArrayReaderWriter.class.getName()).log(Level.SEVERE, null, ex);
        }
        // Si no lo usamos más lo cerramos
        writer.close();
    }
}
```

Clases de análisis para flujos de datos

PushBackInputStream, StreamTokenizer, PushbackReader, LineNumberReader.

PushBackInputStream y PushbackReader: 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 (ya leídos con el método read()), serán **de nuevo 'empujados' a la secuencia**, para que posteriormente, los leamos cuando se haga de nuevo read(). Es como invalidar el último read() para volver a leer ese byte dependiendo del valor obtenido, es decir, nos da una segunda oportunidad de leer un byte, pero en este caso ya lo habremos podido leer.

- **PushbackInputStream: byte** a byte.
- **PushbackReader: caracteres**.

En el siguiente código: un byte podrá ser leídos con anterioridad, y más tarde devuelto a la secuencia o al flujo de bytes:

```
package retrocesodereaderpushbackreader;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.io.PushbackReader;

import java.io.Writer;

import java.util.logging.Level;

import java.util.logging.Logger;

public class RetrocesoDeReaderPushbackReader {

   public static void main(String[] args) {

      // Creamos un fechero para luego leerlo con PushbackReader

      File fichero = new File("fichero");

      try {

         fichero.createNewFile();

         Writer escritor = new FileWriter(fichero);

         escritor.write("Este es el contenido del fichero");
```

```
        } catch (IOException ex) {
            Logger.getLogger(RetrocesoDeReaderPushbackReader.class.
                    getName()).log(Level.SEVERE, null, ex);
        } finally {
            try {
                escritor.close();
            } catch (IOException ex) {
                Logger.getLogger(RetrocesoDeReaderPushbackReader.class.getName()).log(Level.SEVERE,
null, ex);
            }
        }
        try {
            // instanciamos un objeto PushBackReader
            PushbackReader pushBackReader = new PushbackReader(
            /*
             para ello necesitamos a su vez instanciar un objeto FileReader
            indicando la ruta del fichero a tratar
            la primera letra del fichero es “E”.
            */
                    new FileReader(fichero));
            /*
            usamos el método read() y justo
           después mostramos por pantalla el byte leído haciendo casting de
           “char”.
           mostramos por pantalla la letra “E”.
            */
            int data = pushBackReader.read();
            System.out.println((char) data);
            /*
            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".
             */
             pushBackReader.unread(data);
             data=pushBackReader.read();
             System.out.println((char)data);


        } catch (FileNotFoundException ex) {
            Logger.getLogger(RetrocesoDeReaderPushbackReader.class.
                    getName()).log(Level.SEVERE, null, ex);
        } catch (IOException ex) {
                Logger.getLogger(RetrocesoDeReaderPushbackReader.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }
}
```

Clases de análisis para flujos de datos

StreamTokenizer

Un token viene a estar relacionado con un "fragmento" o "**trozo**" … La clase StreamTokenizer tiene la capacidad de **analizar el fichero por** 'fragmentos'. Dichos fragmentos más tarde tendremos que evaluarlos y comprobar si son palabras o números. 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.

Ejemplo

```
package analisisfragmentosficherostreamtokenizer;
/*
 La clase StreamTokenizer tiene la capacidad de analizar el fichero por «trozos»
o «fragmentos». Evaluaremos dichos fragmentos y comprobaremos si son palabras
o números. Reconoce identificadores: números, comillas, espacios, etc.,
lo cual nos puede ser de gran utilidad al analizar ficheros dependiendo de si
su tipología sea de números o de caracteres.
 */
```

```
import java.io.IOException;
import java.io.StreamTokenizer;
import java.io.StringReader;
import java.util.logging.Level;
import java.util.logging.Logger;
public class AnalisisFragmentosFicheroStreamTokenizer {
    public static void main(String[] args) {
        /*
         instaciamos el objeto StreamTokenizer usando el constructor por medio
        del cual le pasamos un StringReader.
         */
        StreamTokenizer streamTokenizer = new StreamTokenizer(new StringReader(""
                + "Este es el texto de un StreamTokenizer"));
        /*
         algunos de sus métodos estáticos 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: indica que el token evaluado es un número o una asociación
        de ellos.
         */
        try {
            while (streamTokenizer.nextToken() != StreamTokenizer.TT_EOF) {
                // si el token es de tipo palabra, mostraremos por pantalla dicha palabra.
                if (streamTokenizer.ttype == StreamTokenizer.TT_WORD) {
                    System.out.println(streamTokenizer.sval);
                    // si es de tipo número, mostraremos por pantalla el número.
                } else if (streamTokenizer.ttype == StreamTokenizer.TT_NUMBER) {
                    System.out.println(streamTokenizer.nval);
                    // si es de tipo final de línea se imprimirá retorno de carro.
                } else if (streamTokenizer.ttype == StreamTokenizer.TT_EOL) {
```

```
                System.out.println();
            }
          }
        } catch (IOException ex) {
            Logger.getLogger(AnalisisFragmentosFicheroStreamTokenizer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Clases de análisis para flujos de datos

LineNumberReader

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**.
- **readLine**(): nos devolverá un conjunto de caracteres al cual se le debe asignar a una variable String.
- **read**(): lee un buffer.

Veamos un código con el uso de esta clase:

```
package contadorlineaslinenumberreader;
/*
LineNumberReader es hija de BufferedReader.
Almacena y cuenta el número de líneas leídas de caracteres.
Está orientada a trabajar y analizar líneas completas.
Empieza leyendo por la primera línea con el contador a 0 y cada vez que
encuentra un retorno de carro incrementa su valor en +1.
 */
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
```

```
import java.util.logging.Level;
import java.util.logging.Logger;
public class ContadorLineasLineNumberReader {
   public static void main(String[] args) {
      try {
         /*
         Se instancia LineNumberReader con un nuevo objeto FileReader, el cual
         es instanciado con un fichero que se ha agregado al proyecto.
          */
         LineNumberReader lineNumberReader
               = new LineNumberReader(new FileReader("prueba.txt"
               ));
         // quedará almacenado el contenido de la primera línea con readLine()
         String line = lineNumberReader.readLine();
         /*
         mientras se tengan líneas que recorrer, 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
         System.out.println(line). De esta forma iremos mostrando el
         contenido del fichero completo línea tras línea.
          */
         while (line != null) {
            System.out.println("Línea número "
                  + lineNumberReader.getLineNumber()
                  + ": "
                  + line);
            line = lineNumberReader.readLine();
         }
         lineNumberReader.close();
      } catch (FileNotFoundException ex) {
            Logger.getLogger(ContadorLineasLineNumberReader.class.getName()).log(Level.SEVERE,
null, ex);
```

```
        } catch (IOException ex) {
            Logger.getLogger(ContadorLineasLineNumberReader.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }
}
```

Clases para el tratamiento de información

Nos permiten **procesar tipos primitivos de Java**: int, float, long etc.

DataInputStream

Nos ofrece una gran ventaja en comparación con InputStream:

Veamos un **ejemplo** de uso de **DataInputStream y DataOutputStream** con tipos de **datos primitivos**:

```
package lecturatiposprimitivosdatainputstream;
/*
Útil para  lectura de tipos primitivos.
Se suele usar la clase DataInputStream para leer ficheros que previamente han
sido escritos con DataOutputStream.
Hereda métodos de InputStream.
Si queremos leer byte a byte con el método read(), evidentemente lo tendremos
también disponible, así como la lectura con un array de bytes.
Atencion: cuando leemos tipos de datos primitivos 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.
 */
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
import java.util.logging.Level;
import java.util.logging.Logger;
public class LecturaTiposPrimitivosDataInputStream {
    public static void main(String[] args) {
        try {
            // vamos a introducir datos en el fichero
            /*
            para leer de un fichero por medio de la clase DataInputStream,
            antes hemos tenido que realizar la escritura ordenada del fichero y
            debemos conocer el tipo de datos y la cantidad de ellos que ha sido
            insertada. Usamos DataOutputStream.
            El objeto es instanciado en su constructor con un FileOutputStream
            con la ruta del fichero que vamos a escribir.
             */
            DataOutputStream salida = new DataOutputStream(
                    new FileOutputStream("datos.bin"));
            /*
            Escribimos en el fichero distintos tipos primitivos de Java usando
            diferentes métodos:
            • writeInt() : solo aceptara entero como parámetro.
            • writeFloat() : solo aceptara float como parámetro.
            • writeDouble() : solo aceptara double como parámetro.
             */
            salida.writeInt(10);
            salida.writeFloat(10.10F);
            salida.writeDouble(10.1234);
            // cerramos nuestro flujo y liberamos recursos.
            salida.close();
            // ahora vamos a crear el flujo de lectura de datos primitivos
            DataInputStream dataInputStream = new DataInputStream(
                    new FileInputStream("datos.bin"));
            int entero = dataInputStream.readInt();
```

```
            float numFloat = dataInputStream.readFloat();
            double numDouble = dataInputStream.readDouble();
            dataInputStream.close();
            System.out.println("El número entero es: " + entero);
            System.out.println("El número float es: " + numFloat);
            System.out.println("El número double es: " + numDouble);
        } catch (FileNotFoundException ex) {
            Logger.getLogger(LecturaTiposPrimitivosDataInputStream.class.getName()).log(Level.SEVERE
, null, ex);
        } catch (IOException ex) {
            Logger.getLogger(LecturaTiposPrimitivosDataInputStream.class.getName()).log(Level.SEVERE
, null, ex);
        }
    }
}
```

Clases para el tratamiento de información

DataInputStream

Un objeto de esta clase 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. En el código anterior vimos **cómo usar DataOutputStream** y **DataInputStream**:

Oracle web oficial:

https://docs.oracle.com/javase/8/docs/api/?java/io