7 RESUMEN UNIDAD DIDÁCTICA: LECTURA Y ESCRITURA DE INFO.

Contenido

LECTURA Y ESCRITURA DE INFORMACIÓN.

Ciclo Formativo de Grado Superior **Desarrollo de Aplicaciones Multiplataforma**
Módulo: **Programación**

José Manuel Martínez del Hoyo Cañizares

I.E.S. Maestre de Calatrava. Ciudad Real.
https://iesmaestredecalatrava.es/

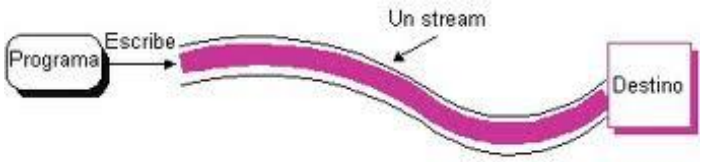
1. Flujos de datos *(Streams)*

1.1. Concepto de *Stream*

Un **stream** *(traducido como flujo)* representa un ***"canal"*** que comunica nuestra aplicación Java con cualquier entidad externa que **produzca** o **consuma** información.

- Si la entidad produce información, la aplicación podrá leerla a través del **stream de Lectura**.

- Si consumen información, la aplicación podrá escribir en ella a través de un **stream de Escritura**.



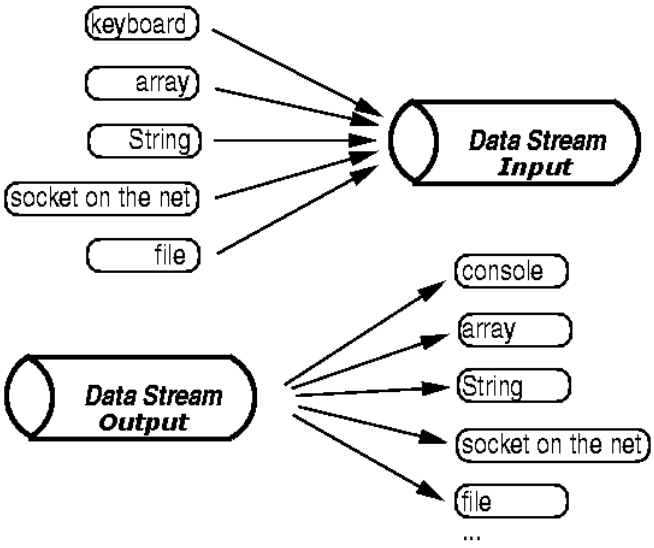
Cualquier aplicación Java que necesite llevar a cabo una **operación de E/S**, lo hará a través de un stream.

Por tanto, un stream hace de **intermediario entre la aplicación y el origen o destino** de la información.

El uso de streams **simplifica la programación**, porque el desarrollador implementará la lectura o escritura de datos de la misma manera, sin importar donde se leen o escriben físicamente dichos datos.

El origen o destino de los datos será un elemento como:

- un **fichero** *(file)* en cualquier dispositivo de almacenamiento.
- un **dispositivo de entrada**; por ejemplo, el Teclado *(keyboard)*.
- un **dispositivo de salida**; por ejemplo, la pantalla *(console)*.
- una **conexión de red** *(socket)* con otra aplicación.



La **vinculación del stream** con el **dispositivo físico**, la hace **el sistema de E/S de Java (java.io)** y, a más bajo nivel, la **Máquina Virtual de Java** y el **Sistema Operativo**

java.io

JVM

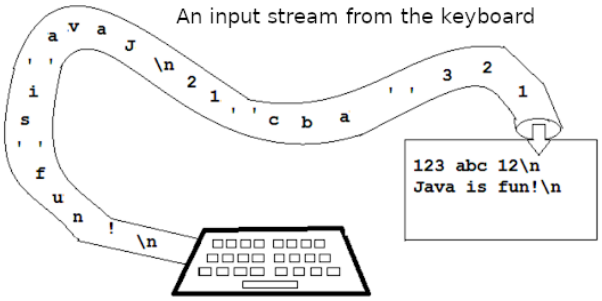
S.O.

Este diseño es muy eficaz, porque **las clases de java.io se utilizan de la misma manera sin importar el dispositivo implicado**.

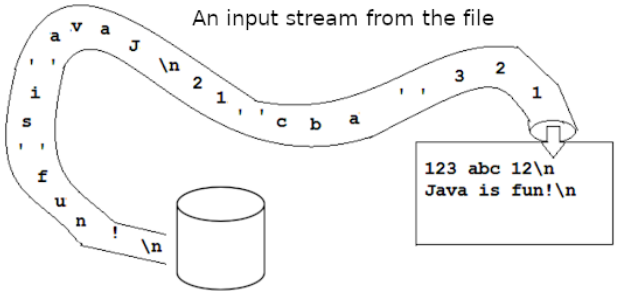
Luego la JVM, y en última instancia el S.O., sabrán si tienen que tratar con el teclado, el monitor, un sistema de ficheros o un socket de red, liberando a nuestro código de tener que saber con quién está interactuando.

Ejemplo

Si una aplicación desea obtener una cadena tecleada por el usuario, el **origen** será el **teclado** y el **stream** a crear será de **lectura** (Input) porque la *aplicación **lee** el dato tecleado*.



Si una aplicación desea obtener una cadena que existe en un fichero en disco, el **origen** será el **fichero** y el **stream** a crear será de **lectura** (Input) porque la *aplicación **lee** el dato del fichero*.



- En ambos casos **se podrían utilizar las mismas clases y los mismos métodos**.

- La única diferencia que existe es al crear el stream: en el primer caso hay que indicarle que es el **teclado** y en el segundo el **nombre y ruta del fichero**.

El **teclado** constituye la **entrada estándar** del sistema y la **consola** la **salida estándar**. Por este motivo, ya existen objetos creados en la clase **java.lang.System**, que representan a ambos dispositivos:

- para el teclado existe **System.in**.
- para la consola existe: **System.out**.

1.2. Clases Java relativas a streams

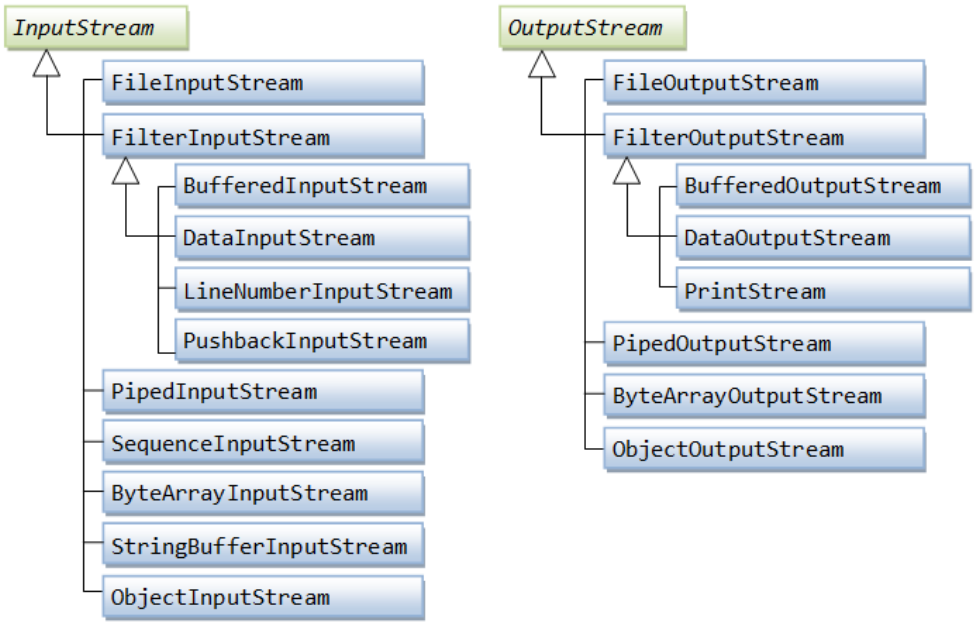
El sistema de E/S de Java está constituido por un **conjunto de clases organizadas jerárquicamente**, que representan los streams.

Estás clases están todas en el paquete **java.io** de la biblioteca de clases estándar de Java.

Java define dos tipos de streams:

- **Byte Streams**
  Proporciona un medio para la lectura y escritura de **datos binarios** (bytes).

  Este tipo de streams se implementan con una serie de clases que heredan de dos clases abstractas: **InputStream** y **OutputStream**.

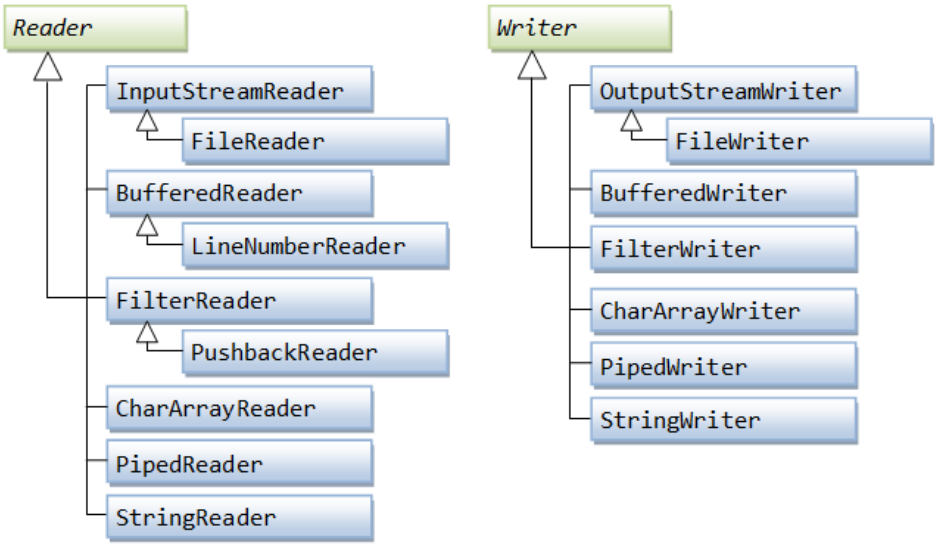


- **Character Streams**
  Proporciona un medio para la lectura y escritura de **caracteres**.

  Este es un modo que Java nos proporciona para manejar caracteres, pero al nivel más bajo todas las operaciones de E/S son orientadas a byte.

  Estos streams usan codificación **Unicode** y, por tanto, se pueden internacionalizar.

  Este tipo de streams se implementan con una serie de clases que heredan de dos clases abstractas: **Reader** y **Writer**.



1.3. Operaciones básicas

Siempre que se desee utilizar la E/S en una aplicación, se deben realizar tres operaciones generales:

- **Creación y apertura del stream**
  El sistema procederá a realizar todas las tareas necesarias para preparar la comunicación con el dispositivo concreto.

- **Lectura o Escritura de datos**
  La lectura o escritura se realiza a través del stream creado. Se podrán realizar tantas operaciones como se desee mientras no se cierre o suceda algún tipo de error no previsto.

- **Cierre del stream**
  Se realiza el cierre cuando no se va a seguir utilizando el stream. El sistema procederá a realizar todas las tareas de liberación de recursos y a la escritura de los datos pendientes.

Cualquier de las operaciones anteriores, podrá lanzar una excepción del tipo **IOException**, que deben ser controlada **obligatoriamente**.

Ejemplo: Lectura de un carácter por teclado

```
System.out.printf("Introduce un carácter cualquiera: ");

try {

     // LECTURA - Origen TECLADO (lectura caracteres)
     // Creo el Stream adecuado:
     InputStreamReader streamTeclado =
            new InputStreamReader (System.in);

     // Leo un carácter
     car = streamTeclado.read();

     // Cierro stream
     streamTeclado.close();

     System.out.printf("Has tecleado %c %n", car);

}
catch (IOException ...){
     ...
}
```

2. Ficheros de Texto

2.1. FileReader y FileWriter

Un fichero de texto es básicamente una **secuencia de caracteres**, codificados en ASCII, Unicode o algún otro tipo de codificación. Además, algunos caracteres son usados para indicar fin de línea, fin de página o fin de fichero.

- Existe el juego de caracteres **ISO-8859-1** ***(Latin1)*** para la codificación **ASCII** más otra serie de caracteres hasta un total de 191.

- Los juegos de caracteres **UTF-8** y **UTF-16** se utilizan para codificar **Unicode**.

➔ Más información sobre Juegos de Caracteres
http://www.vicente-navarro.com/blog/2008/06/15/juegos-de-caracteres-ascii-cp850-iso-8859-15-unicode-utf-8/

Aunque un fichero de texto se puede leer y escribir con las clases **FileInputStream** y **FileOutputStream** (lectura y escritura de bytes), existen clases más adecuadas para el manejo de ficheros de texto. Son las heredadas de **Reader** y **Writer**.

Algunas de estas clases son:

- **FileReader** y **FileWriter**.
- **BufferedReader** y **BufferedWriter**.

Un objeto FileReader se puede crear de varias formas. Una de ellas es la siguiente:

- **FileReader**(**String** ***filename***) **throws** ***FileNotFoundException***
  Crea un stream de lectura con el fichero de texto indicado como parámetro.

  Si el fichero no puede ser abierto por cualquier razón, lanzará una excepción del tipo ***FileNotFoundException.***

Un objeto FileWriter se puede crear de varias formas. Dos de ellas son las siguientes:

- **FileWriter**(**String *filename***) **throws *IOException***
  Crea un stream de escritura con el fichero de texto indicado como parámetro. Si el fichero existe, lo **sobrescribe**.

  Si el fichero no puede ser abierto o creado por cualquier razón, lanzará una excepción del tipo ***IOException.***

- **FileWriter**(**String *filename*, boolean *append***)
  throws *IOException*
  Igual que el anterior, pero si el fichero existe y el segundo parámetro es **true**, comenzará a **añadir caracteres** al final.

Ejemplo: Copia un fichero de texto en otro fichero.

```
final String INPUT_FILENAME = "texto.txt";
final String OUTPUT_FILENAME = "copia.txt";
final int EOF = -1;

 // Uso de la sentencia try-with-resources (cierre automático)
 try (
   // Uso de 2 Character Streams:
     // LECTURA cuyo origen es un Fichero (lectura cars.)
     // Stream: Creo objeto de FileReader (clase especializada)
     // ESCRITURA cuyo destino es un Fichero (escritura cars.)
     // Stream: Creo objeto de FileWriter (clase especializada)

         FileReader fr = new FileReader (INPUT_FILENAME);
         FileWriter fw = new FileWriter (OUTPUT_FILENAME)
     )
 {
     // Leo todos los caracteres del fichero de entrada
     // y los escribo en el de salida
     int car = fr.read();
     while (car != EOF){
         fw.write(car);
         car = fr.read();
     }

 }catch (FileNotFoundException e){
     // Se que no ha encontrado el fichero
     System.out.printf("El fichero NO pudo ser abierto %s %n",
             INPUT_FILENAME);

 }catch (IOException e){
     System.out.println("ERROR E/S: " + e);
 }
```

2.2. Lectura y escritura con Buffer

A pesar de la existencia de las clases anteriores, el manejo de un fichero de texto con esas clases, sigue siendo un método muy rudimentario.

Existen otras clases para leer y escribir ficheros de texto de forma **más eficiente** utilizando un buffer. Estas son **BufferedReader** y **BufferedWriter**.

- Un objeto **BufferedReader** permite leer caracteres de una ***"fuente"*** de texto (por ejemplo, de un fichero), utilizando **buffer** para mejorar la eficiencia, y proporcionando otras operaciones como por ejemplo la **lectura de líneas completas**.

- Un objeto **BufferedWriter** permite escribir caracteres a un ***"destino"*** (por ejemplo, un fichero), utilizando **buffer** para mejorar la eficiencia, y proporcionando otras operaciones como por ejemplo la **escritura de líneas completas**.

Los objetos de estos tipos se **combinan** con otros objetos *stream*.

- Si la fuente de texto es un fichero de texto, se crea un objeto **FileReader** y con él, creas el objeto **BufferedReader**.

```
BufferedReader br =
        new BufferedReader (new FileReader(FILENAME)))
```

- Si el destino del texto es un fichero de texto, se crea un objeto **FileWriter** y con él, creas el objeto **BufferedWriter**.

```
BufferedWriter bw =
        new BufferedWriter (new FileWriter(FILENAME))
```

Estas **clases especializadas** que necesitan un objeto de otro tipo de *stream* más genérico, se suelen denominar ***"Filtros"***.

Algunos métodos de BufferedReader

```
public String readLine()throws IOException
```

Lee una linea completa, devolviendo el contenido en un String. Devuelve **null** si se leyeron todos los datos del *stream*.

```
public int read(char[] cbuf,int off,int len)
                                        throws IOException
```

Lee un conjunto de caracteres en una porción de un array.

Algunos métodos de BufferedWriter

```
public void write(String str) throws IOException
```

Escribe el contenido del String pasado.

```
public void newLine() throws IOException
```

Escribe un salto de linea.

```
public void write(int c) throws IOException
```

Escribe un carácter individual.

```
public void write(String s,int off,int len)
                                        throws IOException
```

Escribe una porción del String pasado.

```
public void flush() throws IOException
```

Fuerza la escritura del buffer.

Ejemplo: Comprueba si la primera linea contiene una palabra dada.

```
// Uso de la sentencia try-with-resources
try ( BufferedReader br = new BufferedReader(
                new FileReader (FILENAME)) )
{
    // Leo la primera linea y examino si contiene la palabra
    String linea = br.readLine();

    if (linea != null && linea.contains(PALABRA) ){
       System.out.printf("SI contiene la palabra);
    }
    else {
         System.out.printf("NO contiene la palabra);
    }

}catch (FileNotFoundException e){
    // Se que no ha encontrado el fichero
    System.out.printf("El fichero NO pudo ser abierto %s %n",
            FILENAME);

}catch (IOException e){
    System.out.println("ERROR E/S: " + e);
}
```

3. Ficheros Binarios

3.1. FileInputStream y FileOutputStream

Un fichero siempre es un **conjunto de bytes**, por tanto, todos los ficheros son binarios. Lo que representen esos bytes puede ser muy diverso.

Para manejar adecuadamente los ficheros hay que **conocer perfectamente su formato**, es decir, qué datos contiene, en qué orden están organizados, etc.

- Si son **caracteres** en algún tipo de codificación, ya sabemos que son **ficheros de texto**.

- Si **no son caracteres**, esos bytes constituirán **cualquier información** que tenga que manejar una aplicación. Por ejemplo podría ser audio, video, un gráfico, una colección de números reales, un documento Word o PDF, etc.

Las clases que manejan ficheros binarios son heredadas de **InputStream** y **OutputStream**. En concreto son:

- **FileInputStream**: Stream de entrada (lectura).
- **FileOutputStream**: Stream de salida (escritura).

Un objeto FileInputStream se puede crear de varias formas. Una de ellas es la siguiente:

- **FileInputStream** (**String** ***filename***) **throws** ***FileNotFoundException***
  Crea un stream de lectura con el fichero indicado como parámetro.

  Si el fichero no puede ser abierto por cualquier razón, lanzará una excepción del tipo ***FileNotFoundException.***

Un objeto FileOutputStream se puede crear de varias formas. Dos de ellas son las siguientes:

- **FileOutputStream** (**String *filename***) **throws *FileNotFoundException***
  Crea un stream de escritura con el fichero indicado como parámetro. Si el fichero existe, lo **sobrescribe**.

  Si el fichero no puede ser abierto por cualquier razón, lanzará una excepción del tipo ***FileNotFoundException.***

- **FileOutputStream** (**String *filename*, boolean *append***) **throws *FileNotFoundException***
  Igual que el anterior, pero si el fichero existe y el segundo parámetro es **true**, comenzará a **añadir bytes** al final.

Los métodos de lectura y escritura de las clases anteriores, son los heredados de las clases InputStream y OutputStream:

```
public void write(int b) throws IOException
```

Escribe el byte especificado en el fichero.

```
public int available() throws IOException
```

Devuelve el nº de bytes que quedan por leer.

```
public int read() throws IOException
```

Lee el siguiente byte del fichero. Se devuelve como un valor entero entre 0 y 255. Si se llegó al final del fichero, devuelve el valor -1.

```
public long skip(long n)throws IOException
```

Salta un número de bytes del fichero sin tener que leerlos, avanzando el puntero de lectura/escritura.

Ejemplo: Verificar si una imagen es PNG.

```
        final int CAB_BYTES = 3;
        StringBuffer cabecera = new StringBuffer("");

        final String IMG_PNG = "img/super-mario.png";

        System.out.println("Examinando la imagen...");

        try ( FileInputStream fis = new FileInputStream(IMG_PNG) )
        {
            // Salto el primero byte
            fis.skip(1);

            for (int i=0;i<CAB_BYTES;i++){
                // Leo un byte, lo convierto a caracter y
                char unByte = (char) fis.read();
                // lo añado
                cabecera.append(unByte);
            }

            if (cabecera.charAt(0) == 'P' &&
                cabecera.charAt(1) == 'N' &&
                cabecera.charAt(2) == 'G'){
                   System.out.println("Es PNG");
            }

        }
        catch (FileNotFoundException e){
            System.out.println("ERROR: " + e);
        }
        catch (IOException e){
            System.out.println("ERROR: " + e);
        }
    }
```

3.2. Lectura y Escritura de Datos

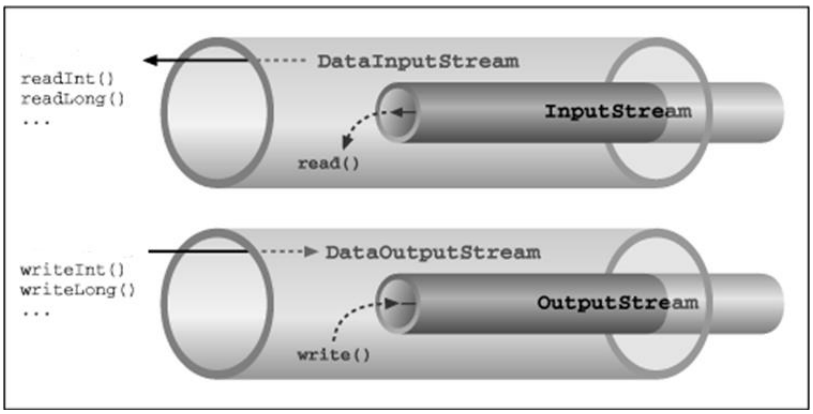
A veces se hace necesario escribir y leer datos que maneja un objeto de mi aplicación. Con las clases anteriores sería difícil escribirlos o leerlos a nivel de byte. **¿Cómo se codifica un double? ¿Qué bytes tengo que escribir o leer? ¿Y si en lugar de un double es un int?**

Para este propósito, existen las siguientes clases ***"Filtro"***:

- **DataInputStream**
  Permite leer datos en forma de byte, pero **adaptándola a los tipos simples** (double, int, short, byte,..., String).

- **DataOutputStream**
  Permite escribir datos en forma de byte, que estén **almacenados en cualquier tipo simple** (double, int, short, byte,..., String).

Ambas clases construyen objetos a partir de streams de tipo **InputStream** y **OutputStream** respectivamente.



- Un objeto **DataOutputStream** permite escribir datos a un ***"destino"*** (por ejemplo un fichero), utilizando el método apropiado **según el tipo donde está almacenado el datos** a escribir (writeLong, writeInt, writeDouble, writeUTF, ...).

- Un objeto **DataInputStream** permite leer datos de una ***"fuente"*** (por ejemplo un fichero), utilizando el método apropiado **según el método utilizado al escribir** (readLong si se escribió con writeLong; readDouble si se escribió con writeDouble, ...).

Como ya sabemos, los objetos de estos tipos se **combinan** con otros objetos *stream*.

- Si el destino de los datos es un fichero, se crea un objeto **FileOutputStream** y con él, creas el objeto **DataOutputStream**.

```
DataOutputStream dos =
        new DataOutputStream (
                new FileOutputStream (FILENAME))
```

- Si la fuente de datos es un fichero, se crea un objeto **FileInputStream** y con él, creas el objeto **DataInputStream**.

```
DataInputStream dis =
        new DataInputStream (
                new FileInputStream(FILENAME))
```

Ejemplo 1 : Escritura y Lectura de un String y un double.

```
String nombre = "Juan";
double peso   = 65.678;
```

```
// ESCRIBO EN EL FICHERO los dos datos
try ( DataOutputStream dos =
      new DataOutputStream(new FileOutputStream("pesos.dam")) )
{

    dos.writeUTF(nombre);
    dos.writeDouble(peso);

}catch (IOException e){
     ...
}
```

```
// El fichero está cerrado. Ahora LEO los datos.
try ( DataInputStream dis =
        new DataInputStream(new FileInputStream("pesos.dam")) )
{

    String nom = dis.readUTF();
    double kg  = dis.readDouble();
    System.out.printf("Nombre: %s, %.2f Kg %n", nom, kg);

}catch (FileNotFoundException e){
     ...
}
 catch (IOException e){
     ...
}
```

EOFException

Hay que tener en cuenta que los métodos de **DataInputStream** que realizan la **lectura de datos**, lanzan la excepción **EOFException** (subclase de IOException) si se alcanzó el final del fichero, por lo que será necesario capturarla.

Por ejemplo:

```
public final double readDouble()throws IOException
```

Devuelve los siguientes ocho bytes leídos, interpretados como un valor double. Si se alcanza el final del fichero, lanza **EOFException**.

Ejemplo 2 : Lectura completa de un fichero que almacena N valores double.

```
final String FICHERO = "valores.dat";
double valor = 0.0;

// try-with-resources: cierre automático
try (
    // Apertura
    DataInputStream dis = new DataInputStream(
           new FileInputStream(FICHERO))
    )
{
    // Leo los datos (double)
    while(true){
        // Leo valor del fichero
        valor = dis.readDouble();
        // TRAZA
        System.out.printf("Valor leido: %.2f %n", valor);
    }

}catch(FileNotFoundException e){
    System.out.printf("Fichero no encontrado %n");

}catch(EOFException e){
    System.out.printf("Final de Fichero %n");

}catch(IOException e){
    System.out.printf("ERROR: %s", e);
}
```

4. Serialización de Objetos

4.1. Concepto

La serialización consiste en **enviar una colección de objetos** *"en serie"*, hacia un destino que podría ser un fichero, la red, etc.

Si se escriben en un fichero, se logra la **persistencia de objetos** para poder recuperarlos en otro momento o por otra aplicación.

Ya conoces cómo escribir un dato en un fichero: dependiendo del tipo que se desee escribir, hay un método diferente en **DataOutputStream**, que lo que hace es **convertir un valor dado a una secuencia de bytes**.

Una vez convertido, es posible:

- **escribirlo** para poder recuperarlo más tarde (con FileOutputStream por ejemplo)

- **enviarlo** por la red a otra aplicación

Para convertir un **Objeto completo** a una secuencia de bytes, es necesario que la clase implemente el **interface java.io.*Serializable***.

- Al poder convertir el objeto a bytes, ese objeto se puede **enviar a través de red**, **guardarlo en un fichero**, y después leerlo para que sea reconstruido en otra parte o en otro momento.

- Como la interfaz Serializable no tiene métodos, es muy sencillo implementarla, basta con un **implements java.io.*Serializable*** y nada más, Java ya sabe cómo convertir ese objeto a una secuencia de bytes y cómo poder reconstruir el objeto a partir de dicha secuencia.

4.2. ObjectInputStream y ObjectOutputStream

El proceso de serialización y des-serialización lo realizan las clases **ObjectOutputStream** y **ObjectInputStream**.

Ambas son casi iguales que DataOutputStream/DataInputStream, solo que incorporan los métodos **writeObject** y **readObject**.

Recuerda que funcionan como "Filtros":

- si se desea **escribir un objeto en un fichero**, se usará un objeto de la clase **FileOutputStream**.

- si se desea **leer un objeto de un fichero**, se usará un objeto de la clase **FileInputStream**.

Ejemplo:

El ejemplo se compone de tres clases:

- clase de los objetos que deseamos escribir/leer (serializable). Pertenece a la **capa de negocio** (model)

- clase que representa un objeto de la **capa de persistencia**, que permite **escribir** objetos

- clase que representa un objeto de la **capa de persistencia**, que permite **leer** objetos

Clase **Alumno**:

```
package es.maestredam.model;

/**
 * Clase que permite crear objetos Alumno.
 * @author jmartinez
 */
public class Alumno implements java.io.Serializable{
    ...
}
```

Clase **EscritorAlumnos**:

```
package es.maestredam.persistence;

/**
 * Clase que permite guardar objetos Alumno
 * @author josema
 */
public class EscritorAlumnos {
    /**
     * Atributo: ruta/nombre fichero
     */
    private String filename;

     /**
     * Constructor
     *
     * @param filename ruta/nombre del fichero.
     */
    public EscritorAlumnos(String filename){
        this.filename = filename;
    }

   /**
     * Escribe en el fichero una lista de objetos Alumno.
     *
     * @param lista array con la lista de objetos
     * @return nº de objetos escritos; -1 si error.
     */
    public int escribir(Alumno lista[]){
        int ret = 0;

        // Abro recurso: fichero de objetos
        try (ObjectOutputStream oos =
                new ObjectOutputStream (
                        new FileOutputStream(filename))){

            for (Alumno al : lista) {
                oos.writeObject(al);
                ret++;
            }

        } catch(IOException e){
            // Escribo información sobre la excepción
            System.out.printf(" > ERROR E/S.%s %n", e);
            ret = -1;

        } finally {
            // El return se realizará siempre.
            return ret;

        } // fin try-with-resources
    }
```

Clase **LectorAlumnos**:

```
package es.maestredam.persistence;

/**
 * Clase que permite leer objetos Alumno de un fichero.
 * @author jmartinez
 */
public class LectorAlumnos {
    /* Atributo: ruta/nombre fichero */
    private String filename;
    /* Constructor */
    public LectorAlumnos(String filename){
        this.filename = filename;
    }

    /**
     * Lee el primer objeto del fichero.
     *
     * @return objeto Alumno; null en caso de error.
     */
    public Alumno leer(){
        Alumno alum = null;

        try (ObjectInputStream ois =
                new ObjectInputStream(
                        new FileInputStream(filename)))
        {
           // Leo el primer objeto
           alum = (Alumno) ois.readObject();

           // TRAZA:
           System.out.printf("  > leido: %s %n", alum);

        } catch (ClassNotFoundException cnfe) {
            System.out.printf(" > ERROR en Clase %n");

        } catch (EOFException e){
            System.out.printf(" >EOF %n");

        } catch (FileNotFoundException e){
            System.out.printf(" > No encontrado %n");

        } catch (IOException e){
            System.out.printf(" > ERROR E/S.%s %n", e);
        }

        finally {
            // Devuelve el valor con o sin error
            return alum;
        }

    } // fin método
```

4.3. Serial versión ID

Cuando tratamos con **objetos Serializables** en distintas aplicaciones, lo normal (salvo que usemos carga dinámica de clases), es que ambas partes tengan **su propia copia del fichero .class** correspondiente a la clase.

Esta clase se **modificará**, de forma que es posible que un lado tenga una versión más antigua que en el otro lado. Si sucede esto, la reconstrucción de la clase en el lado que recibe es imposible.

Para evitar este problema, se aconseja que la clase Serializable tenga un atributo privado de esta forma:

```
private static final long serialVersionUID = 8799656478674716638L;
```

- de forma que el número que ponemos al final debe ser distinto para cada versión de compilado que tengamos.

- Así Java es capaz de detectar rápidamente que las versiones de del fichero .class en ambos lados son distintas.

Algunos IDEs, como **Eclipse**, dan un warning si una clase que implementa Serializable (o hereda de una clase que a su vez implementa Serializable) no tiene definido este atributo.

Es más, **puede generarlo automáticamente**, número incluido, si se lo pedimos en las opciones que nos da para solucionar el warning.

La generación del SerialVersionUID la realiza una **Utilidad del JDK llamada serialver**.

Ejemplo:

```
> serialver es.maestredam.samples.Circulo

es.maestredam.samples.Circulo:    private static final
long serialVersionUID = 6578138543862045339L
```

5. Uso de Sockets

5.1. Introducción

Los sockets son un mecanismo que nos permite establecer un **enlace entre dos programas** que se ejecutan

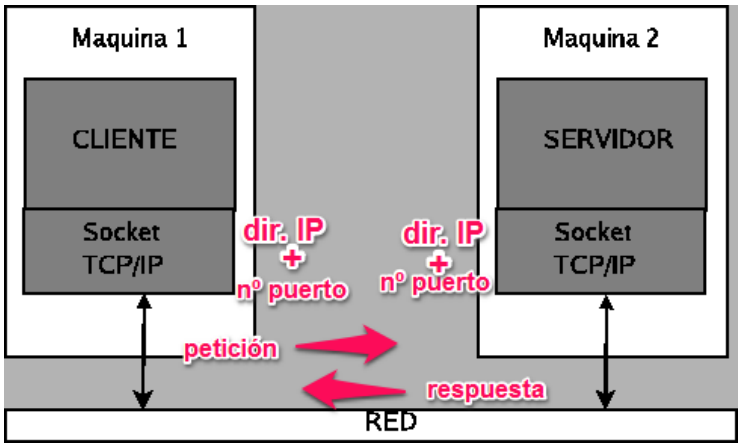
- **independientes** el uno del otro,
- en **distintas máquinas** conectadas en una red.

Permite construir **aplicaciones de red** que sigan el paradigma Cliente-Servidor: uno de los programas actúa de **servidor** y el otro de **cliente**.

- **Servidor**: es el programa que permanece a la escucha y responde a las peticiones de uno o varios clientes.

  Normalmente, son procesos que se están ejecutando indefinidamente.

- **Cliente**: es el programa que inicia la comunicación, enviando una o varias peticiones al servidor



Requisitos:

- Que un programa sea capaz de localizar al otro. Para ello un socket identifica una **dirección IP** y un **número de puerto**.

- Que ambos programas sean capaces de **intercambiar secuencias de datos** (bytes) que sean relevantes para su funcionamiento.

5.2. Clases ServerSocket y Socket

El lenguaje Java implementa dos clases para manejar sockets y crear clientes y servidores de red. Estas son:

- **ServerSocket**: representa un socket de servidor.
- **Socket**: representa un socket de un cliente.

Servidor

1) Un programa servidor creará un **objeto ServerSocket** indicando un **nº de puerto**.

2) El servidor comenzará a **escuchar** mediante el método **accept**, que devolverá un objeto de la clase **Socket**, que representa la conexión con un determinado cliente.

```
public static void main(String[] args) {

    final int PUERTO = 4444;
    Socket clientSocket = null;

    try (
        // Abro Recurso:
        // Socket de servidor para escuchar peticiones.
        ServerSocket serverSocket =
                                new ServerSocket(PUERTO))
    {
        while (true){
            // Esperando peticiones de algún cliente
            clientSocket = serverSocket.accept();

            // ¡Cliente conectado!
            System.out.println(" > Cliente en linea");

            procesarPeticion(clientSocket);

            System.out.println(" > El cliente finalizó");

           } // Fin while

    } catch (IOException e) {
        System.out.printf("ERROR en serverSocket: "+e);
    }
```

3) El servidor procesará la petición del cliente. Esto implica un **intercambio de información** entre cliente y servidor.

   Para ello, se utiliza el concepto de **stream** como vimos en los apartados anteriores.

   - **Output stream**: para enviar datos al cliente.
   - **Input stream**: para recibir datos al cliente.

   Los streams se obtienen del **objeto Socket** que devolvió el método **accept** al establecerse la conexión del cliente con el servidor.

```
private static boolean procesarPeticion(Socket clientSocket)
{
    final String RESPUESTA = "Bye!";

    try (
        // Abro recursos:
        // Data streams con el cliente (de entrada y salida)
        DataOutputStream toClient = new
             DataOutputStream(clientSocket.getOutputStream());

        DataInputStream fromClient = new
             DataInputStream(clientSocket.getInputStream())
         )
    {

        // Servidor RECIBE texto del cliente
        String texto = fromClient.readUTF();

        // Servidor ENVÍA respuesta al cliente
        toClient.writeUTF(RESPUESTA);

    }catch(IOException e){
        System.out.println("ERROR E/S: " + e);

    }  // fin try-with-resources
```

Cliente

1) Un programa cliente creará un **objeto Socket** indicando el **host** o máquina donde está el programa servidor y un **nº de puerto**.

   El host se indica mediante su dirección IP o su nombre.

2) Si la conexión se establece con éxito, creará los **streams** correspondientes para el intercambio de información, igual que se hizo en el servidor.

   - **Output stream**: para enviar datos al servidor.
   - **Input stream**: para recibir datos del servidor.

   Los streams se obtienen del **objeto Socket** creado y que establece la conexión con el servidor.

```
public static void main(String[] args) {

    final String HOST    = "merlin";
    final int PUERTO     = 4444;
    final String mensaje = "Hello";

    try (
        // Abro recursos:
        // socket con servidor + Data streams
        Socket socket = new Socket(HOST, PUERTO);
        DataOutputStream outputToServer = new
                 DataOutputStream(socket.getOutputStream());
        DataInputStream inputFromServer = new
                 DataInputStream(socket.getInputStream()))
    {
        // Cliente ENVÍA texto al servidor
        outputToServer.writeUTF(mensaje);

        // Cliente RECIBE respuesta del servidor
        String respuesta = inputFromServer.readUTF();

    } catch (UnknownHostException e) {
        System.err.printf("ERROR: Host desconocido);

    } catch (IOException e) {
        System.out.println("ERROR E/S: " + e);

    } // fin try-with-resources
}
```

5.3. Servidor multithread

Un **thread** o **hilo de ejecución** es simplemente una tarea que puede ser ejecutada **al mismo tiempo que otra tarea** del mismo programa.

Para que un servidor pueda atender a varios clientes simultáneamente, es necesario que el servidor se ocupe de la **conexión de cada cliente en un thread**.

Servidor

1) Un programa servidor creará un **objeto ServerSocket** indicando un **nº de puerto** *(igual que ante*s).

2) El servidor comenzará a **escuchar** mediante el método **accept**, que devolverá un objeto de la clase **Socket** *(igual que antes)*.

```
public static void main(String[] args) {

    final int PUERTO = 4444;
    Socket clientSocket = null;
    int numClient = 0;

    try (
        // Abro Recurso:
        // Socket de servidor para escuchar peticiones.
        ServerSocket Recurs =
                              new ServerSocket(PUERTO))
    {
        while (true){
            System.out.println("Esperando cliente...");

            // Esperando que algún cliente se conecte
            clientSocket = serverSocket.accept();

              ...

        } // Fin while

    } catch (IOException e) {
        System.out.printf("ERROR en serverSocket: "+e);
    }
```

3) El servidor procesará la petición del cliente. Para ello:

   - le asigna un **número** consecutivo al cliente conectado

   - crea un **nuevo thread** para atender todas las peticiones del cliente *(más adelante se explica cómo crear el thread).*

   - la ejecución del **servidor continua** y **vuelve a hacer accept**, permaneciendo a la espera hasta que se conecte otro cliente, siguiente el mismo proceso que en el paso anterior.

```
...

while (true){
    System.out.println("Esperando cliente...");

    // Esperando que algún cliente se conecte
    clientSocket = serverSocket.accept();

    // ¡Cliente conectado!
    if (numClient == Integer.MAX_VALUE){
        numClient = 0;
    }
    numClient++;

    // Creo un nuevo THREAD para el cliente
    ThreadServer thread = new ThreadServer(
                                    clientSocket,
                                    numClient);

        thread.start();

} // Fin while
```

4) La creación de un **nuevo thread** se hace

   - creando un objeto de una **clase que hereda** de la clase del API llamada **java.lang.Thread**.

   - invocando al método **start** del objeto anterior para ponerlo en ejecución.

   En esta nueva clase, debes sobrescribir el **método run()**, que tendrá todo el **intercambio de información** entre cliente y servidor.

   - En el ejemplo, esta clase se llama **ThreadServer**.

   - Su constructor tiene dos parámetros: el nº asignado al cliente y el socket para comunicarse con él.

```
public class ThreadServer extends java.lang.Thread {

    private Socket clientSocket;
    private int numClient;

     /**
      * Constructor
      *
      * @param clientSocket socket del cliente
      * @param numClient número asignado al cliente
      */
     public ThreadServer(Socket clientSocket, int numClient){
         super("Cliente " + numClient);
         this.clientSocket = clientSocket;
         this.numClient    = numClient;
     }

    @Override
    public void run(){

        // ¡Cliente conectado!
        System.out.printf(" > Cliente en linea,
                              número: %d %n", numClient);

        // Proceso cada petición del cliente,
        // devolviendo una respuesta
        procesarPeticiones();

        System.out.printf(" > El cliente número %d
                                 finalizó %n", numClient);
     }
```

```java
    /**
     * Método que procesa todas las petición de un cliente.
     * <ul>
     *   <li>Lee una cadena enviada por el cliente.</li>
     *   <li>Escribe otra cadena al cliente como respuesta.</li>
     * </ul>
     *
     * @return true si no ha sucedido alguna excepción, para
     * que continue el servidor escuchando; false para parar
     * servidor al producirse una IOException.
     */
    private boolean procesarPeticiones(){
        final String RESPUESTA = "Bye la!";
        boolean terminar = false;

        // Uso de try-with-resources
        try (
            // Abro recursos:
            // Data streams con el cliente (de entrada y salida)
            DataOutputStream mensajeParaCliente = new
                DataOutputStream(clientSocket.getOutputStream());
            DataInputStream mensajeDelCliente = new
                DataInputStream(clientSocket.getInputStream())
            )
        {
            boolean salir = false;
            while (!salir){
                // LEO una palabra
                String palabra = mensajeDelCliente.readUTF();

                // ESCRIBO Respuesta
                mensajeParaCliente.writeUTF(RESPUESTA);

                // Hasta que el cliente NO decida salir,
                // continua atendiendo al cliente.
                if (palabra.equalsIgnoreCase("BYE")){
                    // El cliente pidió finalizar
                    salir = true;
                }
            }

        }catch (EOFException e){
            System.out.println(" > El cliente finalizó
                                    inesperadamente.");
        }catch(IOException e){
            System.out.println(" > ERROR en servidor: " + e);
            System.out.println(" > La aplicación se cerrará.");
            terminar = true; // Cierro la aplicación

        } // fin try-with-resources

        return terminar;
    }
}
```