ProgramaciónCFGS DAM

Entrada / Salida

JAVA 8

Carlos Serrano Sánchez
IES FRANCISCO DE LOS RÍOS

Tabla de contenido

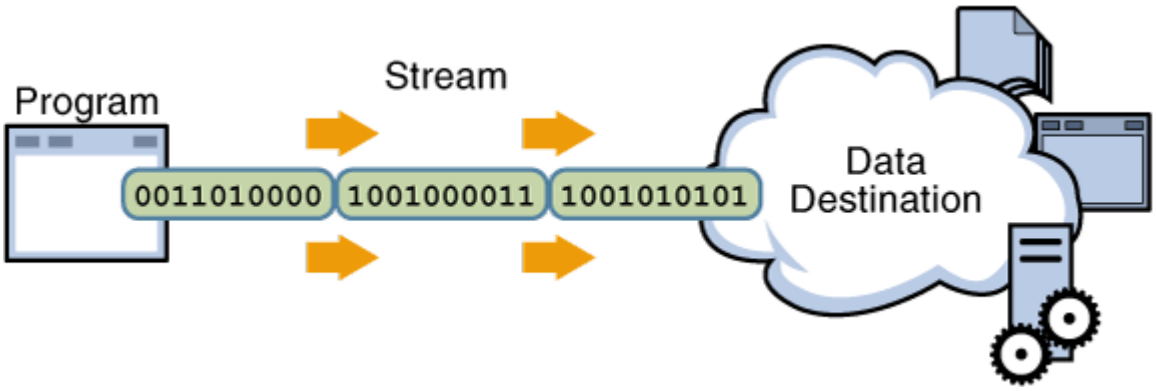
1 Introducción

A lo largo del curso ya hemos empleado en varias ocasiones el sistema de entrada/salida que dispone de Java: cuando hemos escrito por pantalla, cuando hemos leído del teclado, incluso cuando hemos escrito y leído de un archivo, en plano, serializando objetos o, incluso escribiendo XML.
Pero sin duda, no lo hemos visto en detalle dado que para explicar cómo Java trata estos flujos (streams) de entrada y salida se necesitas conceptos avanzados: clases, herencia, poliformismo… Ahora es el momento de abordar este tema con profundidad.

El paquete estándar que gestiona estos flujos es java.io y existen, fundamentalmente, dos tipos de wrapper para estos flujos:

- Orientado a bytes, los cuales se emplean para lectura y escritura de archivos, sockets u otro tipo de dispositivos, sin formato de texto, es decir, byte a byte. Por ejemplo, en la transferencia de una imagen.
- Orientado a caracteres, lo cuales se emplean de igual forma que los anteriores, pero en entorno donde el contenido está en formato texto y nos interesa poder leer un carácter o una línea, por ejemplo, transmitiendo texto.

Aunque un sistema tenga comunicación bidireccional (escritura y lectura) las clases que gestionan el flujo de lectura (input) son distintas a las de escritura (output) y se deben crear ambos sistemas de gestión de flujos si se desea una comunicación bidireccional.



A la hora de emplear flujos siempre debemos tener la siguiente imagen en mente:

- Lectura
  1. Abrir un flujo a una fuente de datos (creación del objeto stream)
     - Teclado
     - Fichero
     - Socket remoto
  2. Mientras existan datos disponibles
     - Leer datos
  3. Cerrar el flujo (método close)
- Escritura
  1. Abrir un flujo a una fuente de datos (creación del objeto stream)
     - Pantalla
     - Fichero
     - Socket local
  2. Mientras existan datos disponibles
     - Escribir datos
  3. Cerrar el flujo (método close)

- ***Nota: para los flujos estándar ya se encarga el sistema de abrirlos y cerrarlos***
- Un fallo en cualquier punto produce la excepción `IOException`

Existen una serie de clases y wrappers para la gestión de estas comunicaciones. Veamos algunas de ellas.

2 Stream de bytes

Como hemos explicado anteriormente, empleamos este tipo de clases cuando la información que intercambiamos no está formateada como texto, o simplemente no nos importa la lectura controlada de caracteres, sino la lectura por bytes.

Para el leer de una fuente de datos en bytes existe la clase InputStream. De aquí derivan los demás wrappers y clases que permiten ampliar su funcionalidad, pero InputStream es la clase primitiva. Veamos, algunos de los métodos que publica:

Método	Descripción
int available ()	Devuelve la cantidad de bytes de entrada disponibles para leer.
void close ()	Cierra la fuente de entrada. Si se intenta leer incluso después de cerrar el flujo, se genera IOException.
void mark(int numBytes)	Coloca una marca en el punto actual del flujo de entrada que seguirá siendo válida hasta que se lean los bytes 'numBytes'
boolean markSupported()	Devuelve true si mark()/reset() son compatibles con el flujo de invocación
int read()	Devuelve una representación entera del siguiente byte de entrada disponible, se devuelve -1 cuando se intenta leer al final del flujo.
int read (byte buffer [])	Intenta leer los bytes de buffer.length en el búfer y devuelve el número total de bytes leídos con éxito. Devuelve -1 cuando se intenta leer al final del flujo.
int read (byte buffer [], int offset, int numBytes)	Intenta leer los bytes de 'numBytes' en el búfer comenzando en el búfer [offset] y devuelve el número total de bytes leídos con éxito. Devuelve -1 cuando se intenta leer al final del flujo.
byte[] readAllBytes()	Lee y devuelve, en forma de una matriz de bytes, todos los bytes disponibles en el flujo. Un intento de leer al final del flujo da como resultado una matriz vacía. [Añadido en JDK9]
int readNBytes(byte byffer[], int offset, int numBytes)	Intenta leer hasta numBytes bytes en el búfer comenzando en el búfer [offset], devolviendo la cantidad de bytes leídos con éxito. Un intento de leer al final del flujo da como resultado la lectura de cero bytes. [Añadido en JDK9]
void reset ()	Restablece el puntero de entrada a la marca establecida previamente.
long skip (long numBytes)	Ignora (es decir, omite) numBytes bytes de entrada, devolviendo la cantidad de bytes ignorados.
long transferTo(OutputStream outStrm)	Copia los contenidos del flujo de invocación a outStrm, devolviendo la cantidad de bytes copiados. [Añadido en JDK9]

Para escribir en una fuente de datos mediante bytes empleamos OutputStream y algunos de sus métodos son:

Método	Descripción
void close ()	Cierra el flujo de salida. Los intentos de escritura posteriores generarán una IOException.
void flush ()	Hace que cualquier salida que se haya almacenado en el búfer se envíe a su destino. Es decir, vacía el búfer de salida.
void write (int i)	Escribe un solo byte en un flujo de salida. Tenga en cuenta que el parámetro es un int, que le permite llamar a write() con expresiones sin tener que volver a convertirlas en byte.
void write (byte buffer [])	Escribe una matriz completa de bytes en un flujo de salida.
Void write(bytes buffer[],int offset, int numBytes)	Escribe un subrango de bytes numBytes desde la matriz buffer comenzando en el buffer(offset).

En general, todos los métodos que hacen referencia a input/output lanzan la excepción IOException por lo que debemos hacer un control de errores exhaustivo.
Estas clases, InputStream y OutputStream, son abstractas, por lo que para realizar una instanciación de dichos flujos, existen una serie de clases que derivan. Cada una con algunas peculiaridades y que pueden ser ventajosas en determinados contextos de uso.

Antes de comenzar con diferentes clases que nos permitirán escribir y leer en archivos y sockets, veamos algunas que ya hemos utilizado para escribir y leer por consola.

2.1 System.in y System.out

System.in es una instanciación de InputStream y System.out una de PrintStream (una derivada de OutputStream). Ambas, como es lógico orientadas a bytes.

De hecho, en System.in tenemos:

```
int read( ) throws IOException

int read(byte data[ ]) throws IOException

int read(byte data[ ], int inicio, int max) throws IOException
```

Y en System.out (que tantas veces hemos empleado):

```
void write(int valbyte)

void println(char[] x)
```

Existen más métodos, para más información:
https://docs.oracle.com/javase/8/docs/api/java/lang/System.html
https://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html

Ejemplos:

```
//Leer una matriz de bytes desde el teclado
import java.io.*;
class LeerBytes {
   public static void main(String[] args) throws IOException {
         byte data[]= new byte[20];

         System.out.println("Ingrese algunos caracteres.");
         //Lee un array de bytes desde el teclado
         System.in.read(data);
         System.out.print("Usted ingresó: ");
         for (int i=0; i<data.length; i++)
            System.out.print((char)data[i]);
   }
}
```

```
//Demostración de System.out.write().

class DemoWrite {
   public static void main(String[] args) {
         int b;
         b='X';

       System.out.write(b);
       System.out.write('\n');
   }
}
```

2.2 Lectura y escritura de archivos

Comenzamos anexando algunas de las clases que implementa InputStream y OutputStream.

Clases de flujo de bytes	Significado
BufferedInputStream	Contiene métodos para leer bytes del búfer (área de memoria).
BufferedOutputStream	Contiene métodos para escribir bytes en el búfer.
ByteArrayInputStream	Contiene métodos para leer bytes de un array de bytes
ByteArrayOutputStream	Contiene métodos para escribir bytes en un array de bytes.
DataInputStream	Contiene métodos para leer tipos de datos primitivos de Java.
DataOutputStream	Contiene métodos para escribir tipos de datos primitivos de Java.
FileInputStream	Contiene métodos para leer bytes de un archivo.
FileOutputStream	Contiene métodos para escribir bytes en un archivo.
FilterInputStream	Contiene métodos para leer bytes de otros flujos de entrada que utiliza como fuente básica de datos.
FilterOutputStream	Contiene métodos para escribir en otros flujos de salida.
InputStream	Clase abstracta que describe la entrada del stream.
Output Stream	Clase abstracta que describe la salida del stream.
ObjectInputStream	Contiene métodos para leer objetos.
ObjectOutputStream	Contiene métodos para escribir objetos.
PipedInputStream	Un flujo de entrada canalizado debe conectarse a un flujo de salida canalizado.
PipedOutputStream	Contiene métodos para escribir en un flujo de salida canalizado.
PrintStream	Flujo de salida que contiene print() y println()
PushbackInputStream	Flujo de entrada que permite que los bytes se devuelvan al stream.
SequenceInputStream	Contiene métodos para concatenar múltiples flujos de entrada y luego leer de el flujo combinado.

Para leer o escribir un archivo orientado a bytes se recomienda el uso de FileInputStream y FileOutputStream. Se recomienda consultar sus métodos en la API.

En el siguiente ejemplo se lee el archivo cuyo nombre se pasa como primer argumento.

```
import java.io.*;
class MostrarArchivo {
   public static void main(String[] args) {
int i;
        //fin es inicalizado como nulo
        FileInputStream fin=null;


       //Primero asegúrese de que haya especificado un archivo
       if (args.length!=1){
          System.out.println("Uso: MostrarArchivo.");
          return;
       }

      //El siguiente código abre un archivo,
      // lee caracteres hasta que se encuentra el EOF,
      // y luego cierra el archivo a través de un bloque finally.
      // EOF es un concepto para determinar el final de un archivo
       try {
          fin=new FileInputStream(args[0]);

          do {
             i=fin.read();
             if (i !=-1) System.out.print((char)i);
          }while (i!=-1); //Cuando i es igual a -1, se ha alcanzado el final del archivo
       }catch (FileNotFoundException exc){
          System.out.println("Archivo no encontrado");
       }catch (IOException exc){
          System.out.println("Ha ocurrido un error de E/S");
       }finally {
          //Cerrar archivo en todos los casos
          try{
             //Cierra fin sólo si no es nulo
             if (fin!=null) fin.close();
          }catch (IOException exc){
             System.out.println("Error al cerrar archivo.");
          }
       }
   }
}
```

Es importante, ver cómo se liberan los recursos y cierra el stream una vez finaliza la lectura.

Para mostrar el ejemplo de escritura, concretamente un programa que copia el contenido de un archivo (lee) a otro(escribe), vamos a introducir el bloque try-with-resources disponible a partir de la JDK7 y que es muy útil.

```
try (especificacion-recursos) {
// usa el recurso
}
```

Dentro de este bloque try se ponen tantas instrucciones como queramos y que sean suceptibles de lanzar la excepción. Si no dan la excepción se ejecuta su bloque de contenido limitado por {}, si no se ejecuta el bloque catch o finally, si los hubiera.

```
import java.io.*;
class CopiarArchivo {
   public static void main(String[] args) throws Exception {
      int i;

      //Primero, asegúrese de que ambos archivos hayan sido especificados.
      if(args.length!=2){
         System.out.println("Uso: CopiarArchivo de - a -");
         return;
      }

      //Copiar y gestionar dos archivos con try
      try ( FileInputStream fin= new FileInputStream(args[0]);
          FileOutputStream fout=new FileOutputStream(args[1]);){
         do {
            i=fin.read();
            if (i!=-1)fout.write(i);
         }while (i!=-1);
      }catch (IOException exc){
         System.out.println("Error de E/S: "+exc);
      }

   }
}
```

3 Stream de caracteres

Como hemos dicho emplearemos este sistema cuando los datos que transmitimos tengan formato de texto y necesitamos controlar su acceso por caracteres o por líneas. Las clases abstractas que getionan este tipo de flujos son Reader y Writer. A continuación, mostramos algunos métodos de Reader:

Método	Descripción
abstract void close()	Cierra la fuente de entrada. Los intentos de lectura posteriores generarán un IOException.
void mark(int numChars)	Coloca una marca en el punto actual del flujo de entrada que seguirá siendo válida hasta que se lean los caracteres numChars.
boolean markSupported()	Devuelve true si mark()/reset() son compatibles en este flujo.
int read()	Devuelve una representación entera del siguiente carácter disponible desde el flujo de entrada invocado. -1 se devuelve cuando se intenta leer al final del flujo.
int read(chat buffer[])	Intente leer los caracteres buffer.length en el búfer y devuelve el número real de caracteres que se leyeron con éxito. -1 se devuelve cuando se intenta leer al final del flujo.
abstract int read(char buffer[], int offset, int numChars)	Intenta leer los caracteres numChars en el búfer comenzando en el búfer [offset], devolviendo la cantidad de caracteres que se leyeron con éxito. -1 se devuelve cuando se intenta leer al final del flujo.
int read (CharBuffer buffer)	Intenta llenar el búfer especificado por el búfer, devolviendo la cantidad de caracteres leídos con éxito. -1 se devuelve cuando se intenta leer al final del flujo. CharBuffer es una clase que encapsula un flujo de caracteres, como una cadena.
boolean ready()	Devuelve true si la siguiente solicitud de entrada no esperará. De lo contrario, es devuelto false.
void reset()	Restablece el puntero de entrada a la marca establecida previamente.
long skip(long numChars)	Omite los caracteres de entrada numChars, devolviendo la cantidad de caracteres omitidos.

Y en el caso de Writer:

Método	Descripción
Writer append(char ch)	Se agrega ch al final del flujo de salida de invocación. Devuelve una referencia al flujo de invocación.
Writer append(CharSequence chars)	Agrega caracteres al final del flujo de salida de invocación. Devuelve una referencia a la secuencia de invocación. CharSequence es una interfaz que define operaciones de solo lectura en un flujo de caracteres.
Writer append(CharSequence chars, int begin, int end)	Añade la secuencia de caracteres comenzando desde begin y deteniéndose con end hasta el final del flujo de salida de invocación. Devuelve una referencia al flujo de invocación. CharSequence es una interfaz que define operaciones de solo lectura en un flujo de caracteres.
abstract void close()	Cierra el flujo de salida. Los intentos posteriores de escritura generarán una IOException
abstract void flush()	Hace que cualquier salida que se haya almacenado en el búfer se envíe a su destino. Es decir, vacía el buffer de salida.
void write(int ch)	Escribe un solo caracter en el flujo de salida invocante. Tenga en cuenta que el parámetro es un int, que le permite llamar a write() con expresiones sin tener que volver a convertirlas en char.
void write(char buffer[])	Escribe una matriz completa de caracteres en el flujo de salida invocante.
abstract void write (char buffer[], int offset, int numChars)	Escribe un subintervalo de caracteres numChars del búfer de la matriz, comenzando en el búfer (offset) al flujo de salida invocante.
void write (String str)	Escribe str en el flujo de salida invocante.
void write (String str, int offset, int numChars)	Escribe un subrango de caracteres numChars desde la matriz str, comenzando en el offset especificado.

3.1 Lectura y Escritura de caracteres por consola

A lo largo del curso para lectura hemos empleado la clase Scanner (java.util) en infinidad de ocasiones. En este apartado, introducimos otra posible forma de leer desde el teclado controlando caracteres y líneas, empleando la clase BufferedReader (java.io):

```
class LeerCaracteres {
   public static void main(String[] args) throws IOException {
      char c;
      BufferedReader br= new
            BufferedReader(new InputStreamReader(System.in));
      System.out.println("Ingrese caracteres, sequido de punto para parar.");

      //Leer Caracteres
      do {
         c=(char)br.read();
         System.out.println(c);
      }while (c!='.');
   }
}
```

Nota: BufferedReader recibe un InputStreamReader, por ello hay que realizar una instanciación de new InputStreamReader que sí que recibe un InputStream (System.in)
Nota2: las clases BufferedReader y BufferedWriter son aconsejadas siempre que el dispositivo de entrada/salida pueda establecer un retraso en su acceso: teclado, archivos…, pero se desaconseja en sockets.

Si en lugar de leer carácter a carácter, leemos cadenas:

```
import java.io.*;
class ReadLines {
   public static void main(String[] args) throws IOException{
      //Creando un BufferedReader usando System.in
      BufferedReader br=new BufferedReader(new
            InputStreamReader(System.in));
      String str;

      System.out.println("Ingrese una línea de texto.");
      System.out.println("Ingrese 'detener' para detener el programa.");
      do {
         str=br.readLine();
         System.out.println(str);
      }while (!str.equals("detener"));
   }
}
```

En el caso de la salida por pantalla, en la mayoría de ocasiones hemos empleado directamente System.out (como hemos visto en el apartado anterior), sin embargo si deseamos controlar de forma más estrecha el formato de escritura podemos emplear la clase PrintWriter.

```
PrintWriter(OutputStream outputStream, boolean flushingOn)
```

Aquí, outputStream es un objeto de tipo OutputStream y flushingOn controla si Java vacía el flujo de salida cada vez que se llama a un método println() (entre otros). Si flushingOn es true, el flushing se produce automáticamente. Si es false, el flushing no es automático.

> ¿Qué es el flush?
> Fuerza la escritura del buffer en el destino, liberando el buffer para posibles datos. No cierra el stream.

Ahora el ejemplo:

```
//Demostración de PrintWriter
import java.io.*;

class PrintWriterDemo {
   public static void main(String[] args) {
      PrintWriter pw=new PrintWriter(System.out, true);
      int i=10;
      double d=123.65;

      pw.println("Usando PrintWriter.");
      pw.println(i);
      pw.println(d);

      pw.println(i + " + " + d + " es " + (i+d));
   }
}
```

3.2 E/S de archivos empleando flujo de caracteres

En primer lugar, adjuntamos tabla de algunas de las clases que existen:

Clases de flujo de caracteres	Significado
BufferedReader	Contiene métodos para leer caracteres en un búfer.
BufferedWriter	Contiene métodos para escribir caracteres en un búfer.
CharArrayReader	Contiene métodos para leer caracteres de una matriz de caracteres.
CharArrayWriter	Contiene métodos para escribir caracteres en una matriz de caracteres.
FileReader	Contiene métodos para leer desde un archivo.
FileWriter	Contiene métodos para escribir en un archivo.
FilterReader	Contiene métodos para leer caracteres en el flujo de entrada subyacente.
FilterWriter	Contiene métodos para escribir caracteres en el flujo de salida subyacente
InputStreamReader	Contiene métodos para convertir bytes en caracteres.
LineNumberReader	Flujo de entrada que cuenta las líneas.
OutputStreamWriter	Contiene métodos para convertir de caracteres a bytes.
PipedReader	Pipe de entrada.
PipedWriter	Pipe de salida.
PushbackReader	Flujo de entrada que permite que los caracteres se devuelvan al flujo de entrada.
Reader	Clase abstracta que describe la entrada de flujo de caracteres.
Writer	Clase abstracta que describe la salida de flujo de caracteres.
StringReader	Contiene métodos para leer en una cadena.
StringWriter	Contiene métodos para escribir en una cadena.

En el caso de lectura y escritura de archivos se emplea FileReader y FileWriter.

Ejemplo que escribe lo insertado en teclado en un archivo Prueba.txt hasta que se escriba la palabra “detener”:
import java.io.*;

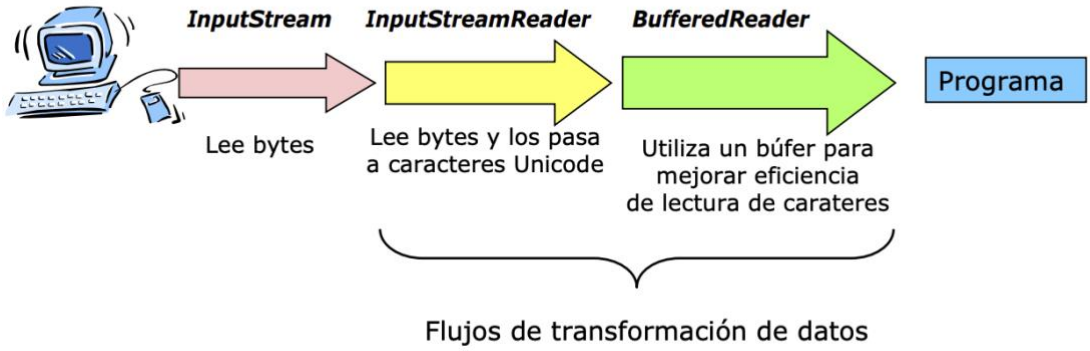
```
class FileWriterDemo {
   public static void main(String[] args) {
      String str;
      BufferedReader br=new BufferedReader(new
            InputStreamReader(System.in));
      System.out.println("Ingrese texto ('detener' para salir)");

      try(FileWriter fw=new FileWriter("Prueba.txt"))//Crea un FileWriter
      {
         do {
            System.out.println(": ");
            str=br.readLine();

            if (str.compareTo("detener")==0) break;
            str=str+"\r\n"; //Añadir nueva línea
            fw.write(str);
         }while (str.compareTo("detener")!=0);
      } catch (IOException e) {
         System.out.println("Error de E/S: "+e);
      }
   }
}
```

Lee el archivo Prueba.txt y lo muestra por pantalla:

```
import java.io.*;
class FileReaderDemo {
   public static void main(String[] args) {
      String s;

      //Cree y use un FileReader incluido en un BufferedReader.
      try (BufferedReader br=new BufferedReader(new FileReader("Prueba.txt"))){
         while ((s=br.readLine())!=null){
            System.out.println(s);
         }
      }catch (IOException exc){
         System.out.println("Error de E/S: "+exc);
      }
   }
}
```

Lo habitual es encontrar diferentes tipos de flujos combinados para ampliar las funcionalidades de nuestra aplicación y emplear uno u otro según necesidades.

- Los flujos se pueden combinar para obtener la funcionalidad deseada



Ejemplos: Escribir caracteres en un archivo mientras se usa un búfer

```
File myFile = new File("targetFile.txt");
PrintWriter writer = new PrintWriter(new BufferedOutputStream(new FileOutputStream(myFile)));
```

Ejemplos: Comprimir y cifrar datos antes de escribirlos en un archivo:

```
Cipher cipher = ... // Initialize cipher
File myFile = new File("targetFile.enc");
BufferedOutputStream outputStream = new BufferedOutputStream(new DeflaterOutputStream(new CipherOutputStream(new FileOutputStream(myFile), cipher)));
```

La clase File

4 Clase File

Una de las clases más útiles a la hora de manejar archivos y directorios es File (java.io)y Files (java.nio).

Antes de comenzar, una aclaración:
La diferencia entre las clases del paquete java.io y java.nio pueden resumirse en la siguiente tabla:

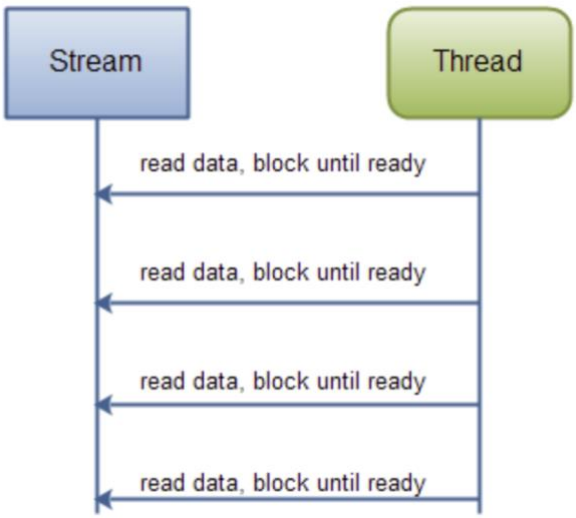
java.io	**java.nio**
Orientado a flujo (Stream)	Orientado a buffer
Bloqueo IO	No bloqueo IO

En java.io se leen los bytes que necesites y no se almacenan en ningún lado. No puedes moverte hacia delante o atrás en ese flujo. Lo leído ya no está disponible y si necesitas un buffer debes crearlo explícitamente. En java.nio existe un buffer el cual cachea los bytes del stream y puedes moverte dentro de ese buffer para ser procesado.

En java.io el hilo queda bloqueado ante una lectura o escritura hasta que se termine o haya algo que recibir. En el caso de java.nio el hilo puede proseguir con su ejecución.
Veamos la diferencia con un ejemplo:

Java.io

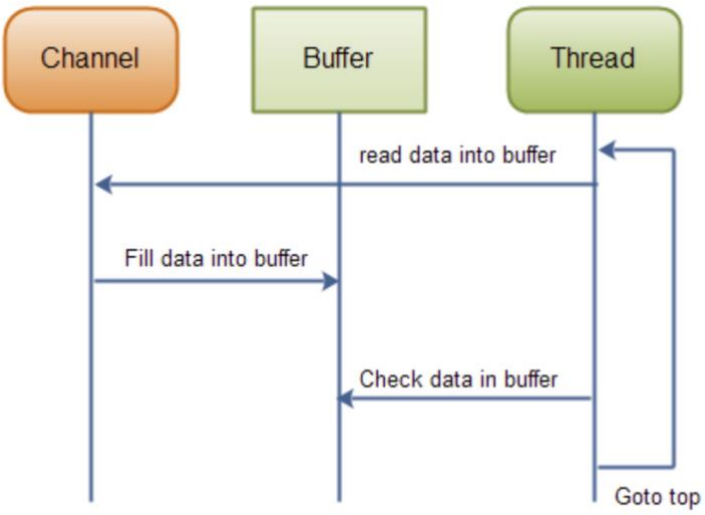
```
InputStream input = ... ; // get the InputStream from the client socket

BufferedReader reader = new BufferedReader(new InputStreamReader(input));

String nameLine   = reader.readLine();
String ageLine    = reader.readLine();
String emailLine  = reader.readLine();
String phoneLine  = reader.readLine();
```



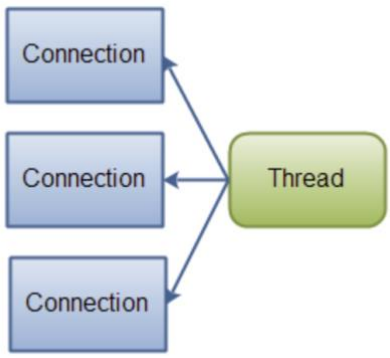
En cada línea raeder.readLine() el hilo se queda bloqueado hasta que no existe una línea para leer y se realiza la lectura.

Java.nio

```
ByteBuffer buffer = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buffer);

while(! bufferFull(bytesRead) ) {
   bytesRead = inChannel.read(buffer);
}
```



Para entender mejor el concepto se recomienda investigar sobre un hilo que crea varias conexiones gracias a java.nio



La clase File nos permite gestionar los archivos y carpetas locales de forma fácil y rápida. Veamos un ejemplo donde se lista los archivos de una carpeta local:

```
String sCarpAct = System.getProperty("user.dir");
File carpeta = new File(sCarpAct);
File[] archivos = carpeta.listFiles();
if (archivos == null || archivos.length == 0) {
   System.out.println("No hay elementos dentro de la carpeta actual");
   return;
}
else {
   SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
   for (int i=0; i< archivos.length; i++) {
      File archivo = archivos[i];
      System.out.println(String.format("%s (%s) - %d - %s",
            archivo.getName(),
            archivo.isDirectory() ? "Carpeta" : "Archivo",
            archivo.length(),
            sdf.format(archivo.lastModified())
            ));
   }
}
```

En el caso de la clase Files nos proporciona mecanismos similares, algunos más avanzados, pero sin bloqueo.

Veámoslo en ejemplos:

Listar todos los archivos

```
Files.list(Paths.get("."))
      .forEach(System.out::println);

 Output:

.\filename1.txt
.\directory1
.\filename2.txt
.\Employee.java
```

Listar solo los archivos

```
Files.list(Paths.get("."))
      .filter(Files::isRegularFile)
      .forEach(System.out::println);

 Output:

.\filename1.txt
.\filename2.txt
.\Employee.java
```

Listar solo archivos ocultos

```
final File[] files = new File(".").listFiles(file -> file.isHidden());
//or
final File[] files = new File(".").listFiles(File::isHidden);
```

Nota: para entender los ejemplos anteriores hay que estudiar Java 8 lambda expression y Java 8 stream

5 Archivos en Java 8

Todo lo explicado anteriormente es absolutamente válido para Java 8, pero este nuevo JDK proporciona algunas herramientas más versátiles y sencillas de aplicar. Veamos algunos ejemplos:

Lo primero es declarar una función que crea un buffer de lectura orientado a caracteres desde un InputStream. Lee todas las líneas y las devuelve como String:

```
private String readFromInputStream(InputStream inputStream) throws IOException {
   StringBuilder resultStringBuilder = new StringBuilder();
   try (BufferedReader br
     = new BufferedReader(new InputStreamReader(inputStream))) {
       String line;
       while ((line = br.readLine()) != null) {
          resultStringBuilder.append(line).append("\n");
       }
   }
  return resultStringBuilder.toString();
}
```

Ahora las empleamos para leer un archivo desde la Classpath:

```
public void givenFileNameAsAbsolutePath_whenUsingClasspath_thenFileData() {
InputStream inputStream = null;
try {
   File file = new File(classLoader.getResource("file.txt").getFile());
   inputStream = new FileInputStream(file);

   //tu codigo para el proceso
}
finally {
   if (inputStream != null) {
      try {
         inputStream.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
}
```

En el caso de leer archivos por bytes:

```
public void givenFilePath_whenUsingFilesLines_thenFileData() {
   Path path = Paths.get(getClass().getClassLoader()
     .getResource("file.txt").toURI());

   StringBuilder data = new StringBuilder();
   Stream<String> lines = Files.lines(path);
```

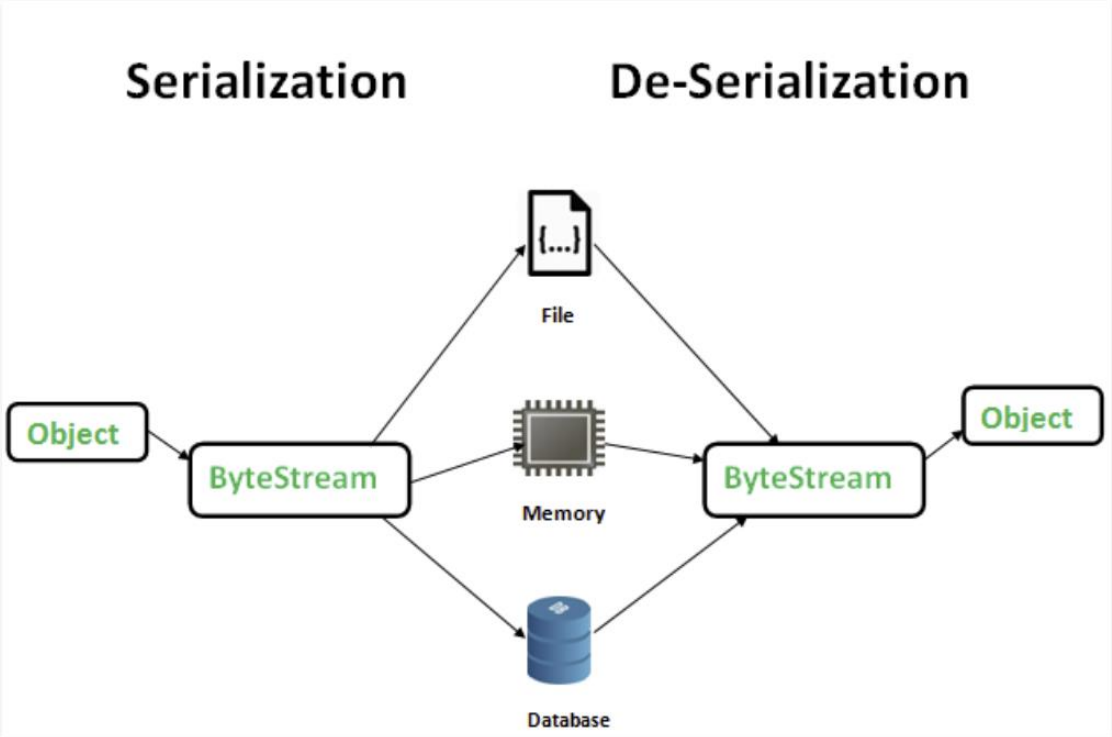
```
    lines.forEach(line -> data.append(line).append("\n"));
    lines.close();
}
```

Leer contenido de una URL:

```
public void givenURLName_whenUsingURL_thenFileData() {
    URL urlObject = new URL("/");
    URLConnection urlConnection = urlObject.openConnection();
    InputStream inputStream = urlConnection.getInputStream();
    String data = readFromInputStream(inputStream);
}
```

6 Object Streams

Uno de los aspectos más interesantes que Java permite es la serialización de objetos. Es decir, que nuestra JDK convierta una instancia a una serie de bytes para que puedan ser transmitidos y/o almacenados con la peculiaridad de que posteriormente puede ser revertido el cambio, es decir, pasar de esos bytes a la instancia original (deserialización).



Para que un objeto pueda ser serializable debe implementar la interfaz java.io.Serializable. Para escribir/leer objetos serializables contamos con:

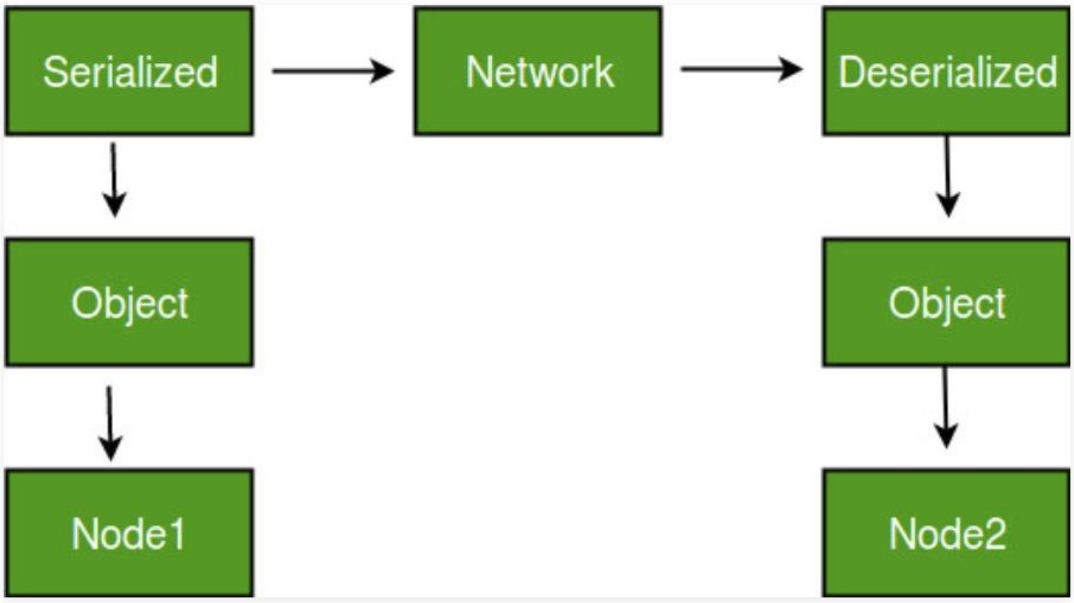
ObjectOutputStream:

```
public final void writeObject(Object obj)
                        throws IOException
```

ObjectInputStream:

```
public final Object readObject()
                   throws IOException,
                ClassNotFoundException
```

Esta funcionalidad nos permite almacenar objetos o incluso transmitirlos por la red:



La única pega es que la serialización es intrínseca a Java por lo que su compatibilidad con otros sistemas se ve comprometida. Por ello, es habitual emplear otros sistemas y protocolos de transmisión más universales, como son XML o JSON.

La máquina virtual de Java socia un número a cada clase serializable (long). Este identificador es empleado para comprobar que el objeto que se serializó y deserializó son del mismo tipo. Este UID puede ser generado por los propios IDE. Si el programador no lo declara, la JVM lo generará, pero es recomendable declarar explícitamente el *serialVersionUID de cada clase.*

Veamos un ejemplo:

```
public class Person implements Serializable {
   private static final long serialVersionUID = 1L;
   static String country = "ITALY";
   private int age;
   private String name;
   transient int height;

   // getters and setters
}
```

Ahora usemos la clase en una prueba:

```
public void whenSerializingAndDeserializing_ThenObjectIsTheSame() ()
 throws IOException, ClassNotFoundException {
   Person person = new Person();
   person.setAge(20);
   person.setName("Joe");

   FileOutputStream fileOutputStream
    = new FileOutputStream("yourfile.txt");
   ObjectOutputStream objectOutputStream
    = new ObjectOutputStream(fileOutputStream);
```

```
    objectOutputStream.writeObject(person);
    objectOutputStream.flush();
    objectOutputStream.close();

    FileInputStream fileInputStream
      = new  FileInputStream("yourfile.txt");
    ObjectInputStream objectInputStream
      = new  ObjectInputStream(fileInputStream);
    Person p2 = (Person) objectInputStream.readObject();
    objectInputStream.close();

    assertTrue(p2.getAge() == p.getAge());
    assertTrue(p2.getName().equals(p.getName()));
}
```

7 ANEXO

7.1 Lectura recomendada

https://howtodoinjava.com/java8/

7.2 Stream a lo Java 8

A pesar de que stream de Java 8 sale del objetivo de este tema (tratamiento de flujo y archivos), en las últimas versiones de la JDK han surgido conceptos que permiten facilitar todo el tratamiento de estas funcionalidades.

Ejemplos:

Leer líneas de un fichero

```
Path path = Paths.get("fichero.txt");

try (Stream<String> stream = Files.lines(path)) {
  stream.forEach(System.out::println);
} catch (IOException e) {
  e.printStackTrace();
}
```

Escribir líneas en un archivo:

```
String[] lines = new String[] { "line 1", "line 2", "line 2" };
    Path path = Paths.get("outputfile.txt");
    try (BufferedWriter br = Files.newBufferedWriter(path,
        Charset.defaultCharset(), StandardOpenOption.CREATE)) {
      Arrays.stream(lines).forEach((s) -> {
        try {
          br.write(s);
          br.newLine();
        } catch (IOException e) {
          throw new UncheckedIOException(e);
        }

      });
    } catch (Exception e) {
      e.printStackTrace();
    }
```

Uno más complejo:
Lee de un archivo
Ordenador.java

```
package com.arquitecturajava;

public class Ordenador {

        private int numero;
        private String sistema;
        private double precio;
        public int getNumero() {
```

```
		return numero;
	}
	public void setNumero(int numero) {
		this.numero = numero;
	}
	public String getSistema() {
		return sistema;
	}
	public void setSistema(String sistema) {
		this.sistema = sistema;
	}
	public double getPrecio() {
		return precio;
	}
	public void setPrecio(double precio) {
		this.precio = precio;
	}
	public Ordenador(int numero, String sistema, double precio) {
		super();
		this.numero = numero;
		this.sistema = sistema;
		this.precio = precio;
	}
	public static Ordenador buildFromArray(String[] elementos) {

		return new Ordenador
(Integer.parseInt(elementos[0]),elementos[1],Double.parseDouble(elementos[2]));

	}
}
```

Principal.java

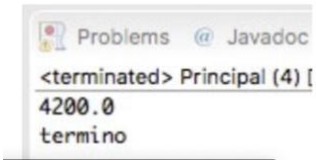
```
package com.arquitecturajava;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class Principal {

	public static void main(String[] args) {

		String ficheros = "datos.txt";

		try (Stream<String> stream = Files.lines(Paths.get(ficheros))) {

			stream.map(linea -> linea.split(","))
				.map(Ordenador::buildFromArray)
				.mapToDouble(o -> o.getPrecio())
				.onClose(() -> System.out.println("termino"))
				.reduce(Double::sum)
```

```
                                        .ifPresent(System.out::println);

                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }

}
```

Imaginemos el archivo datos.txt con el siguiente contenido:

```
1,pc,1000
2,mac,2000
3,android,300
4,ios,900
```

El resultado sería:



Para más información se puede consultar https://www.oracle.com/technetwork/es/articles/java/procesamiento-streams-java-se-8-2763402-esa.html

7.3 Flujos predefinidos

Como sabes, todos los programas Java importan automáticamente el paquete java.lang. Este paquete define una clase llamada System, que encapsula varios aspectos del entorno de tiempo de ejecución. Entre otras cosas, contiene tres variables de flujo predefinidas, llamadas in, out y err. Estos campos se declaran como públicos (public), finales (final) y estáticos (static) dentro de System. Esto significa que pueden ser utilizados por cualquier otra parte de tu programa y sin referencia a un objeto específico de System.

System.out se refiere a la secuencia de salida estándar. Por defecto, esta es la consola. System.in se refiere a la entrada estándar, que es por defecto el teclado. System.err se refiere a la secuencia de error estándar, que también es la consola por defecto. Sin embargo, estas secuencias se pueden redireccionar a cualquier dispositivo de E/S compatible.

System.in es un objeto de tipo InputStream; System.out y System.err son objetos del tipo PrintStream. Se trata de flujos de bytes, aunque normalmente se utilizan para leer y escribir caracteres desde y hacia la consola. La razón por la que son bytes y no flujos de caracteres es que los flujos predefinidos eran parte de la especificación original para Java, que no incluía los streams de caracteres. Como verás, es posible ajustarlos dentro de streams basadas en caracteres si lo desea.

- ***System.in***
  - Instancia de la clase ***InputStream***: flujo de bytes de entrada
  - Metodos
    - ***read()*** permite leer un byte de la entrada como entero
    - ***skip(n)*** ignora n bytes de la entrada
    - ***available()*** número de bytes disponibles para leer en la entrada
- ***System.out***
  - Instancia de la clase ***PrintStream***: flujo de bytes de salida
  - Metodos para impresión de datos
    - ***print(), println()***
    - ***flush() vacía el buffer de salida escribiendo su contenido***
- ***System.err***
  - Funcionamiento similar a ***System.out***
  - ***Se utiliza para enviar mensajes de error*** (por ejemplo a un fichero de log o a la consola)

8 Bibliografía

https://javadesdecero.es/avanzado/io-entrada-salida/
https://howtodoinjava.com/java8/
http://tutorials.jenkov.com/java-nio
https://www.arquitecturajava.com/

9 Ejercicio propuesto

Programa que por consola proporcione las siguientes operaciones:
1- Listar la carpeta indicada (se pide insertar path de carpeta)
2. Mostrar el contenido de un archivo por pantalla (se pide archivo)
3- Crear archivo y pide el nombre, la extensión y el contenido.
4- Copiar un archivo: pide archivo origen y destino (mediante inputstream and outputstream)
5- Copiar una carpeta entera (pide carpeta origen y destino)
6- Borrar un archivo con confirmación
7- Borrar una carpeta con confirmación
8- Comprimir un archivo
9- Comprimir una carpeta
10 - Cifrar un archivo
11 - Cifrar una carpeta
12- Salir
(del 1 al 7 obligatorios) (del 8 al 11 opcionales)

10 Ampliación

Proyecto recomendado
Crear servidor y cliente que se transmiten archivos (imágenes) y texto.
https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/SocketProgramming/SocketProgram.html#overview

GITHUB:
https://github.com/Developodo/Java/tree/master/JavaIO

Vídeo-Tutorial:
https://www.youtube.com/watch?v=uk0HYJmmVJA