I/O

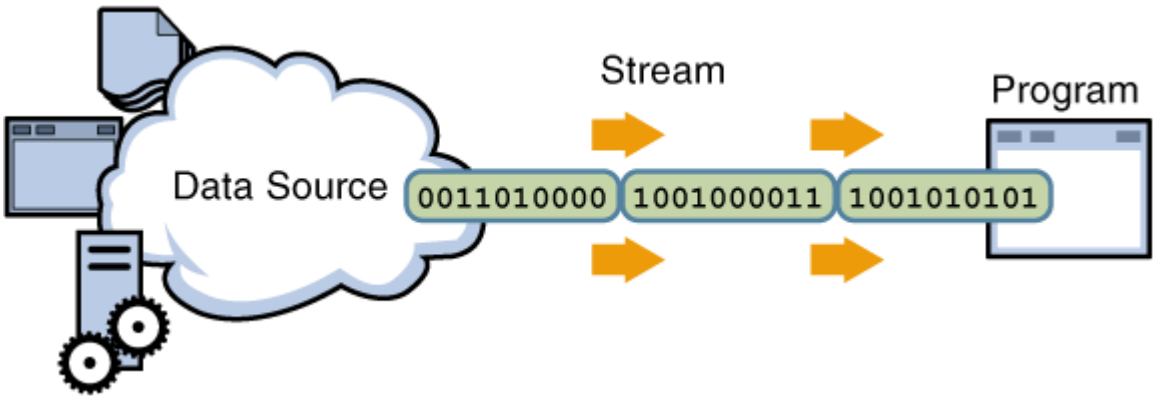
Cualquier programa realizado en Java que necesite llevar a cabo una operación de I/O lo hará a través de un **stream**. Un stream, cuya traducción literal es "flujo", es una abstracción de todo aquello que produzca o consuma información.

I/O Streams

Un *Stream* I/O representa una fuente de entrada o un destino de salida. Un *Stream* puede representar diversas clases de fuentes y de destinos, incluyendo archivos en disco, dispositivos, otros programas, y arreglos en memoria.

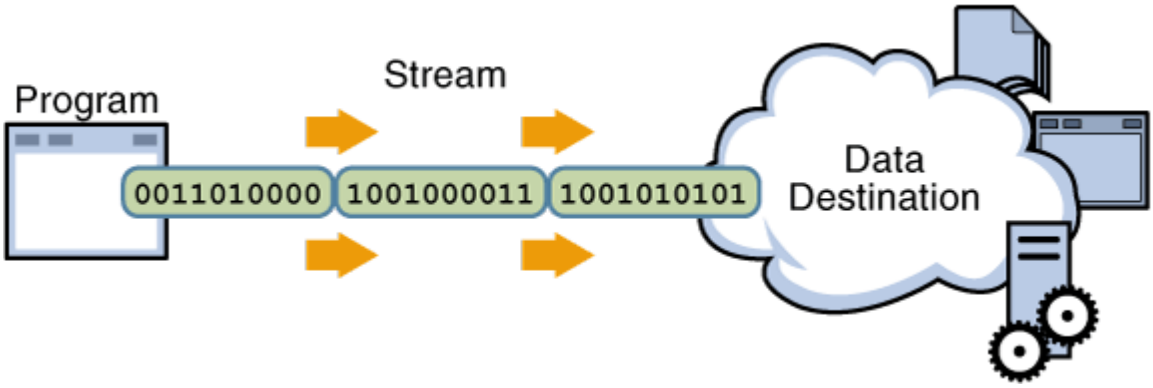
Los Stream soportan diversas clases de datos, incluyendo bytes simples, tipos de datos primitivos, caracteres localizados, y objetos. Algunos Stream simplemente pasan datos; otros manipulan y transforman los datos de forma útil.

No importa cómo trabajan internamente, todos los streams presentan a los programas que los utilizan el mismo modelo simple: Un *Stream* es una secuencia de datos. Un programa utiliza input stream (un flujo de entrada) a los datos leídos de una fuente, leyendo un ítem a la vez:



Información de la lectura en un programa.

Un programa utiliza un output stream (flujo de salida) para escribir los datos a un destino, un dato a la vez:



Información de escritura en un programa.

La fuente de datos (data source) y el destino de los datos (data destination) de los diagramas anteriores puede ser cualquier cosa que genera o consume datos. Esto incluye obviamente archivos de disco, pero una fuente o un destino puede ser otro programa, un dispositivo (impresora, periférico, etc.), un socket de red o un arreglo.

Java2 define dos tipos de streams:

1. **Byte streams**

Proporciona un medio adecuado para el manejo de entradas y salidas de bytes y su uso lógicamente está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene gobernado por dos clases abstractas que son **InputStream** y **OutputStream**.

Cada una de estas clases abstractas tienen varias subclases concretas que controlan las diferencias entre los distintos dispositivos de I/O que se pueden utilizar. Así mismo, estas dos clases son las que definen los métodos que sus subclases tendrán implementados y, de entre todas, destacan los métodos *read()* y *write()* que leen y escriben bytes de datos respectivamente.

Hay muchas clases Byte Stream. Para demostrar cómo funcionan los byte streams, nos centraremos en los streams, ***FileInputStream*** y en ***FileOutputStream***.

Exploraremos FileInputStream y FileOutputStream por medio de un programa de ejemplo llamado CopyBytes, que utiliza byte streams para copiar frase.txt, un byte a la vez.

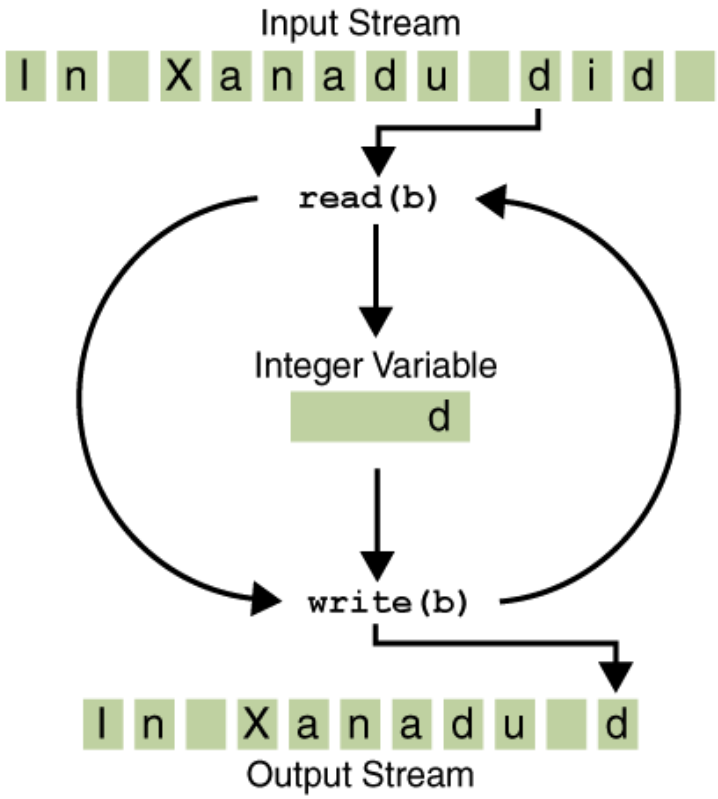
```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("frase.txt");
            out = new FileOutputStream("salidafrase.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }

        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

CopyBytes pasa la mayoría de tiempo en un loop simple que lee el flujo de entrada y escribe en el stream de la salida, un byte a la vez, según se muestra en la figura siguiente.



Un stream simple de entrada y salida

El método read() retorna un dato tipo int. Si el stream de entrada de bytes no se puede leer, el método retorna el valor de -1. Retorna -1 para indicar que llego al final del stream o que no se puede leer.

Siempre se deben cerrar los Streams

Cerrar los streams mientras no se necesiten es muy importante, pues previene bugs futuros en los programas y es una muy buena practica. En el ejemplo CopyBytes usa el bloque finally para garantizar que los streams siempre son cerrados en el caso de que un error ocurra.

Un posible error en el programa CopyBytes ocurre cuando se intenta abrir uno o ambos archivos, esto ocurre si la variable correspondiente al archivo nunca cambia su valor de nulo.

El ejemplo garantiza que antes de llamar el método close() de los objetos streams (in y out)la variable no sea nula.

Todas las clases encargadas de streams de I/O en java se basan en las clases de **Byte streams**, el uso de las clases **Byte streams**, se debe reservar solo para ocasiones en los cuales se necesite trabajar con datos a bajo nivel.

2. Character streams

Proporciona un medio conveniente para el manejo de entradas y salidas de caracteres. Dichos flujos usan codificación Unicode y, por tanto, se pueden internacionalizar.

Observación: Este es un modo que Java proporciona para manejar caracteres, pero al nivel más bajo todas las operaciones de I/O son orientadas a byte.

Al igual que la anterior el flujo de caracteres también viene gobernado por dos clases abstractas: **Reader** y **Writer**. Dichas clases manejan flujos de caracteres Unicode. Y también de ellas derivan subclases concretas que implementan los métodos definidos en ellas siendo los más destacados los métodos *read()* y *write()* que, en este caso, leen y escriben caracteres de datos respectivamente.

En las localidades occidentales, generalmente se usa un carácter ASCII de 8-bit.

Para la mayoría de las aplicaciones, los "character streams" no son mas complicados que los "byte streams". Un programa que usa character streams en lugar de byte streams automáticamente se adapta al código ASCII local y esta listo para la internacionalización.

Usando Character Streams

Todas las classes de "character stream" descienden de las classes Reader y Writer. El siguiente ejemplo CopyCharacters ilustra estas classes.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {
        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("frase.txt");
            outputStream = new FileWriter("salidafrase.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

CopyCharacters es muy similar a CopyBytes. La diferencia más importante es que CopyCharacters utiliza las classes FileReader y FileWriter para la entrada y la salida en lugar de FileInputStream y de FileOutputStream.

Note que CopyBytes y CopyCharacters utilizan una variable "int" para leer y para escribir en los métodos read() y write(). Sin embargo, en CopyCharacters, la variable int (int c;) mantiene un valor carácter (character) de 16 bits; en CopyBytes, la variable int mantiene un byte de 8 bits.

Character Streams que usan Byte Streams

Los character Streams que utilizan los byte streams con a menudo conocidos como "wrappers" (envolturas), los character streams utilizan los byte streams para realizar el I/O a un nivel físico, mientras que character streams se encarga de hacer la traducción de los caracteres ASCII a bits.

FileReader usa FileInputStream, mientras que FileWriter usa FileOutputStream.

I/O Orientado a Líneas.

El I/O sobre caracteres normalmente ocurre en unidades más grandes que de carácter a carácter. Una unidad común es la línea: que se concibe como una cadena de caracteres con un terminador de línea al final de la cadena, se conoce también como un carácter de salto de línea.

Un adaptador, terminador o carácter de salto de línea puede ser una secuencia de retorno de carro o un salto de línea o ambos, ejemplo :

Salto de carro / salto de línea = ("\r\n")
Salto de carro = ("\r")
o Salto de línea = ("\n")

A continuación se modifica la clase CopyCharacters para usar I/O orientado a líneas, se usaran las classes BufferedReader y PrintWriter que aun no se han visto, pero solo es para ilustrar el ejemplo.

La nueva clase CopyLines invoca BufferedReader.readLine y PrintWriter.println para hacer la entrada y salida de una línea a la vez.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream =
                                        new   BufferedReader(new
FileReader("xanadu.txt"));
            outputStream =
                                              new   PrintWriter(new
FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

La llamada al método readLine retorna una línea del texto CopyLines copia cada línea usando el método println y añade el adaptador o terminador de línea para el sistema operativo sobre el cual se este ejecutando la aplicación.

Éste adaptador puede no ser el mismo que el usado para el archivo de entrada. Hay muchas maneras de estructurar la entrada y la salida de texto más allá de caracteres y de líneas. Esto se vera en un apartado llamado Scanning y Formatting.

Buffered Streams

Hasta el momento se han colocado ejemplos de programas que leen o escriben directamente realizando una petición que es dirigida al OS subyacente. Esto hace que un programa sea menos eficiente, por que cada petición de lectura o escritura acciona el acceso de disco, o en acceso a la red lo que aumenta la actividad de la red, o acciona cierta operación que sea relativamente costosa.

Para reducir esta clase de consumos de recursos indirectos, la plataforma de Java implementa los buffered Streams de I/O. Que no es mas que leer datos en un área de memoria intermedia conocida como buffer ; el API de lectura - read() - nativo solamente es llamado cuando el buffer intermediario esta vacío. De forma similar, Los datos de salida se escriben a un buffer intermedio, y se llama al API de escritura - write() – nativo solamente cuando el buffer intermediario esta lleno.

Un programa que usa streams sin buffer se puede convertir a un programa con buffer stream usando los wrapping (envolturas) que me brinda Java , donde el objeto unbuffered stream se pasa al constructor de una clase buffered stream.

Un ejemplo de la modificacion es cambiar el ejemplo CopyCharacters para utilizar I/O con Buffered Streams

```
inputStream =    new BufferedReader(new FileReader("frase.txt"));

outputStream =  new BufferedWriter(new fileWriter("salidafrase.txt"));
```

Hay cuatro clases buffered stream usadas para envolver streams unbuffered: BufferedInputStream y BufferedOutputStream crean buffered byte streams, mientras que BufferedReader y BufferedWriter crean buffered character streams.

Flushing Buffered Streams

Es comun colocar marcas de escritura en el buffer en los puntos criticos y no tener que esperar a que el buffer de escritura se llene para que se realize un accion en el OS. Esta accion se conoce como hacer flush al buffer

Algunas classes de buffer de salida soportan el autoflush, especificando un parámetro opcional en el constructor. Cuando se permite el autoflush, ciertos eventos hacen que el buffer sea limpiado haciendo flush. Por ejemplo, un objeto de PrintWriter con autoflush hace flush en cada invocación de los metodos println o format.

Para hacer flush sobre un stream manualmente, basta con llamar al metodo flush(). El metodo flush es valido en cualquier stream de salida, pero no tiene ningun efecto hacerlo si el streams no usa un buffer