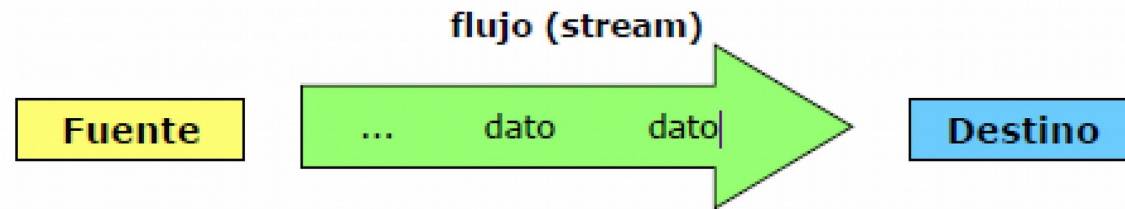


Flujos en Java

¿Qué es un Stream?

Un 'stream' o flujo es una abstracción implementada en Java8 para tratar la comunicación de información entre el programa y el exterior. Tanto la fuente como el destino pueden ser ficheros en disco, sockets de red o procesos; entre otros. Los 'streams' soportan varios tipos de datos, como pueden ser los bytes simples, datos primitivos e incluso objetos de Java.



Flujos en Java

La acción de leer desde una fuente es conocida como **Input**, mientras que la acción de escribir a un destino se la llama **Output**; ambos procesos quedan definidos dentro del paquete java.io.

Los ‘streams’ son unidireccionales, lo que significa que un flujo se podrá usar como lectura o como escritura, pero no para ambas acciones al mismo tiempo.

En Java se accede a la E/S estándar a través de campos estáticos de la clase java.lang.System

- Flujo de entrada (System.in): Instancia de la superclase InputStream, implementa la entrada estándar
- Flujo de salida (System.out): Instancia de la superclase OutputStream, implementa la salida estándar
- Flujo de error (System.err): Se emplea para lanzar mensajes a un log o consola, implementa la salida de error.

Flujos en Java



El algoritmo empleado para leer o escribir información a través de 'streams' es el siguiente:

- 1) Abrir el flujo.
- 2) Leer/escribir información mientras exista.
- 3) Cerrar el flujo.

Cualquier fallo en el uso de los 'streams' produce una excepción de tipo IOException.

Flujos en Java

Ejemplo de uso de flujo

```
import java.io.*;

class LecturaDeLinea {

    public static void main( String args[] ) throws IOException {

        int c;

        int contador = 0;

        // se lee hasta encontrar el fin de línea

        while( (c = System.in.read() ) != '\n' )

        {

            contador++;

            System.out.print( (char) c );

        }

        System.out.println(); // Se escribe el fin de línea

        System.err.println( "Contados "+ contador +" bytes en total." );

    }

}
```

Flujos en Java

Utilización de los flujos

Los flujos se implementan en las clases del paquete java.io

Esencialmente todos funcionan igual, independientemente de la fuente de datos

Clases java.io.Reader y java.io.Writer

```
int read()  
int read(char buffer[])  
int read(char buffer[], int offset, int length)  
  
int write(int aCharacter)  
int write(char buffer[])  
int write(char buffer[], int offset, int length)
```

Flujos en Java

Utilización de los flujos

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**

Flujos en Java

Entrada de texto desde un fichero

```
try {  
    BufferedReader reader = new BufferedReader(new FileReader("nombrefichero"));  
    String linea = reader.readLine();  
    while(linea != null) {  
        // procesar el texto de la línea  
        linea = reader.readLine();  
    }  
    reader.close();  
}  
catch(FileNotFoundException e) {  
    // no se encontró el fichero  
}  
catch(IOException e) {  
    // algo fue mal al leer o cerrar el fichero  
}
```

Flujos en Java

Clasificación de los Streams:

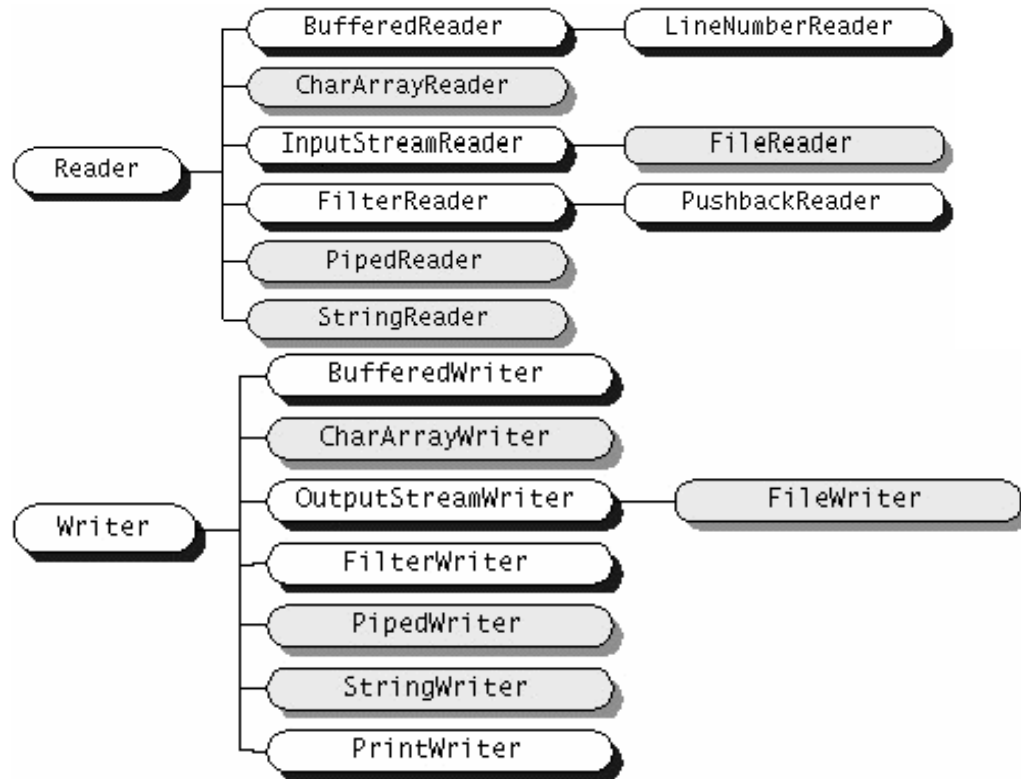
Existen multitud de tipos de 'streams', los cuales pueden ser clasificados en dos grandes grupos dependiendo de la representación de la información que manipulan:

- **Representación de la información**
 - Streams orientados a carácter: clases *Reader* y *Writer*
 - Se puede pasar de un flujo de bytes a uno de caracteres con *InputStreamReader* y *OutputStreamWriter*
 - Streams orientados a bytes: clases *InputStream* y *OutputStream*
- **Propósito**
 - Entrada: *InputStream*, *Reader*
 - Salida: *OutputStream*, *Writer*
 - Lectura/Escritura: *RandomAccessFile*
 - Transformación de los datos
 - Realizan algún tipo de procesamiento sobre los datos (p.e. *buffering*, conversiones, filtrados): *BufferedReader*, *BufferedWriter*
- **Acceso**
 - Secuencial
 - Aleatorio - (*RandomAccessFile*)

Flujos en Java

- **Streams orientados a carácter**. Operan con caracteres de 2 bytes (16 bits) como unidad de trabajo. Suelen ser utilizados para leer y escribir información almacenada en texto, como archivos con extensión TXT, CSV o XML; entre otros. La superclase abstracta utilizada para leer 'streams' orientados a carácter es la clase **Reader**, mientras que la superclase abstracta para escribir 'streams' orientados a carácter es la clase **Writer**.

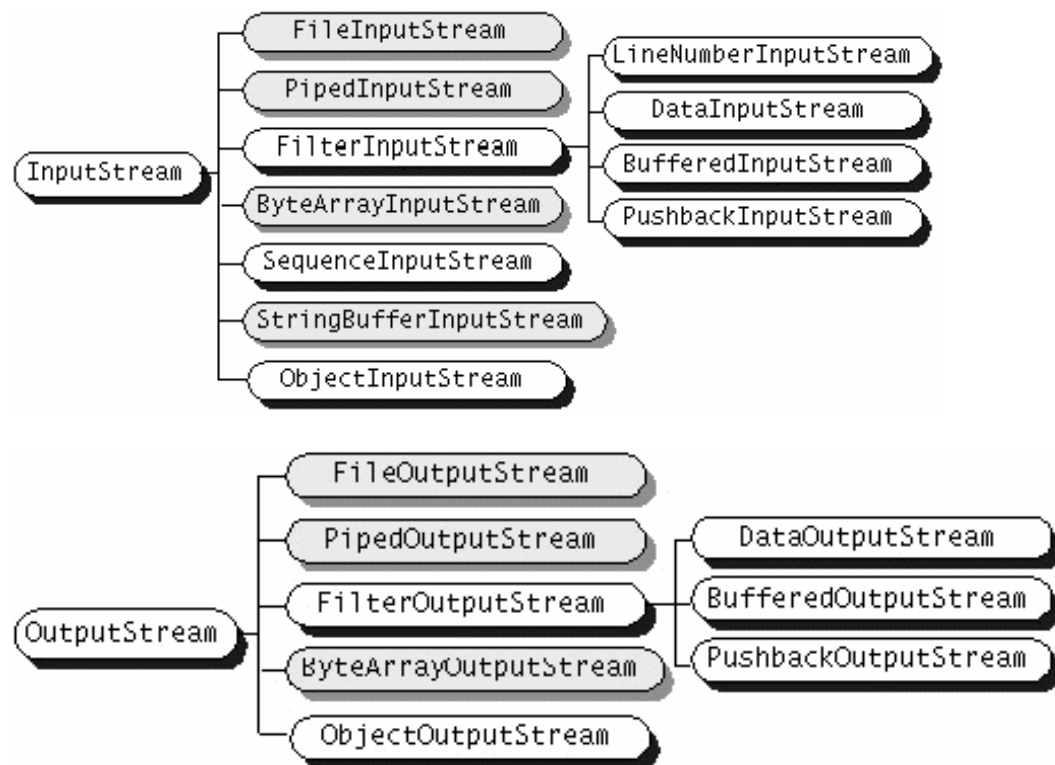
Jerarquía de flujos de caracteres



Flujos en Java

Streams orientados a byte. Operan con bytes (8 bits) dispuestos en forma unitaria. Se emplean para leer y escribir información almacenada en forma binaria, como imágenes o audio; entre otros. La superclase abstracta encargada de leer ‘streams’ orientados a byte es la clase **InputStream**, mientras que la superclase abstracta utilizada para escribir ‘streams’ orientados a byte es la clase **OutputStream**.

Jerarquía de flujos de bytes



Entrada de caracteres

- **InputStreamReader**

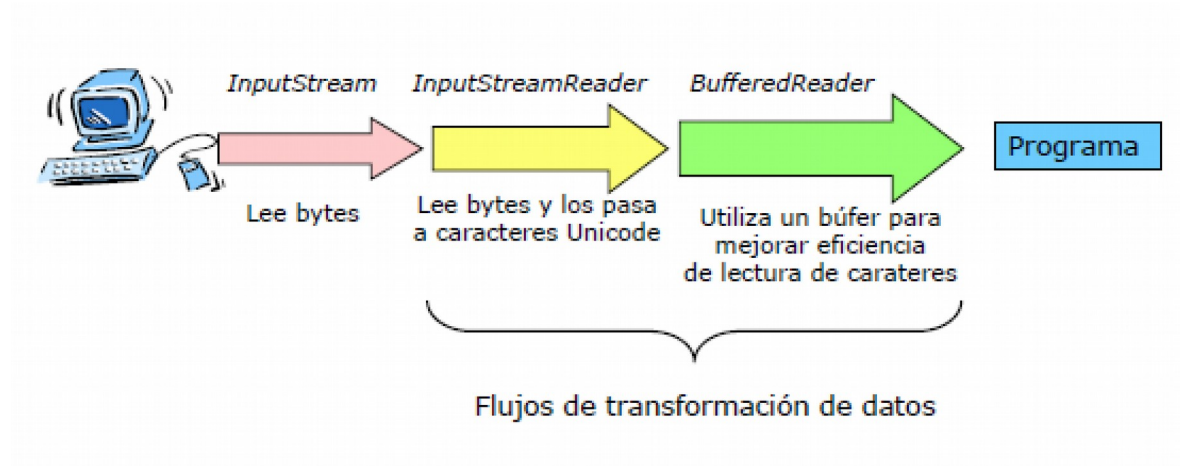
- Lee bytes de un flujo InputStream y los convierte en caracteres Unicode
- Métodos de utilidad
 - *read()* lee un único carácter
 - *ready()* indica cuando está listo el flujo para lectura

- **BufferedReader**

- Entrada mediante búfer, mejora el rendimiento
- Método de utilidad
 - *readLine()* lectura de una línea como cadena

```
InputStreamReader entrada = new InputStreamReader(System.in);  
BufferedReader teclado = new BufferedReader (entrada);  
String cadena = teclado.readLine();
```

Combinación de flujos



Ejemplo de combinación de flujo

```
import java.io.*;

public class Eco {

    public static void main (String[] args) {

        BufferedReader entradaEstandar = new BufferedReader(new InputStreamReader(System.in));

        String mensaje;

        System.out.println ("Introducir una línea de texto:");

        mensaje = entradaEstandar.readLine();

        System.out.println ("Introducido: \"" + mensaje + "\"");

    }

}
```

La clase Teclado

```
import java.io.*;

public class Teclado {

    /** variable de clase asignada a la entrada estándar del sistema */
    public static BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));

    /** lee una cadena desde la entrada estándar
     * @return cadena de tipo String
     */

    public static String leerString() {
        String cadena="";
        try {
            cadena = new String(entrada.readLine());
        } catch (IOException e) {
            System.out.println("Error de E/S"); }
        return cadena; } // la clase Teclado continua
```


La clase Teclado

```
// ...continuación de la clase teclado
/** lee un numero entero desde la entrada estandar
 * @return numero entero de tipo int
 */
public static int leerInt() {
    int entero = 0;
    boolean error = false;
    do {
        try {
            error = false;
            entero = Integer.valueOf(entrada.readLine()).intValue();
        } catch (NumberFormatException e1) {
            error = true;
            System.out.println("Error en el formato del numero, intentelo de nuevo.");
        } catch (IOException e) {
            System.out.println("Error de E/S");
        }
    } while (error);
    return entero;
}
} // final de la clase Teclado
```

Flujo de bytes especiales

- ***File streams***
 - Para escribir y leer datos en ficheros
- ***Object streams***
 - Para escribir y leer objetos
 - Implementa lo que se denomina serialización de objetos (*object serialization*)
 - Es posible guardar un objeto con una representación de bytes
- ***Filter streams***
 - Permiten filtrar datos mientras se escriben o leen
 - Se construyen sobre otro flujo
 - Permiten manipular tipos de datos primitivos
 - Implementan las interfaces `DataInput` y `DataOutput` y pueden heredar de las clases `FilterInputStream` y `FilterOutputStream`
 - El mejor ejemplo son las clases `DataInputStream` y `DataOutputStream` para leer y escribir datos de tipos básicos

Uso de filter stream

- **Para leer tipos de datos primitivos**
 - *Se puede utilizar un `DataInputStream`*

```
FileInputStream ficheroEntrada = new FileInputStream("precios.cat");  
DataInputStream entrada = new DataInputStream(ficheroEntrada);  
  
double precio= entrada .readDouble();  
  
entrada.close();
```

Uso de filter stream

- **Para escribir tipos de datos primitivos**
 - *Se puede utilizar un `DataOutputStream`*

```
FileOutputStream ficheroSalida = new FileOutputStream("precios.cat");  
DataOutputStream salida = new DataOutputStream(ficheroSalida);  
salida.writeDouble(234.56);
```

```
salida.flush();
```

```
salida.close();
```

Fuerza la
escritura de los
datos

Ficheros de texto

- **FileReader**

- Para leer de ficheros de texto
- Hereda de InputStreamReader, que hereda de Reader
- Constructor: FileReader(String nombreFichero)

- **FileWriter**

- Para escribir en ficheros de texto
- Hereda de OutputStreamWriter, que hereda de Writer
- Constructores
 - FileWriter(String nombreFichero) – reescribe
 - FileWriter(String nombreFichero, boolean añadirFinal) -- añade

- **PrintWriter**

- Implementa un flujo de salida de caracteres
- Métodos de utilidad
 - print(), println(), close()

Ejemplo de ficheros de texto

```
import java.io.*;
public class FicheroTexto {
    public static void main(String args[]) {
        try { // escritura de datos
            PrintWriter salida = new PrintWriter( new BufferedWriter(new FileWriter("prueba.txt")) );
            salida.println("Este es un ejemplo de escritura y lectura de datos");
            salida.println("en un fichero.");
            salida.close();
            // lectura de datos
            BufferedReader entrada = new BufferedReader(new FileReader("prueba.txt"));
            String s, s2 = new String();
            while((s = entrada.readLine()) != null)
                s2 += s + "\n";
            System.out.println("Texto leído:" + "\n" + s2);
            entrada.close();
        } catch (java.io.IOException e) { e.printStackTrace(); }
    }
}
```

Ficheros

- **Clase *File***

- **Constructores**

- File(String ruta)
 - File(String ruta, String nombre)
 - File(File directorio, String nombre)

- **Métodos**

- canRead() comprueba si el fichero se puede leer
 - canWrite() comprueba si el fichero se puede escribir
 - delete() borra dicho fichero
 - getPath() devuelve la ruta del fichero
 - mkdir() crea un directorio con la ruta del objeto que lo recibe
 - isDirectory() comprueba si dicho fichero es un directorio

- **Constructores de otras clases**

- FileReader(File fichero)
 - FileWriter(File fichero)

Copia de ficheros

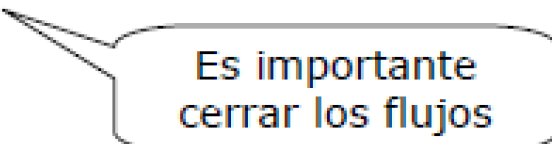
```
import java.io.*;

public class CopiaFicheros {
    public static void main(String[] args) throws IOException {
        File ficheroEntrada = new File("original.txt");
        File ficheroSalida = new File("copia.txt");

        FileReader entrada = new FileReader(ficheroEntrada);
        FileWriter salida = new FileWriter(ficheroSalida);
        int dato;

        while ( (dato = entrada.read()) != -1 )
            salida.write(dato);

        entrada.close();
        salida.close();
    }
}
```



Es importante
cerrar los flujos

Serialización de objetos

- **Serializar** es almacenar objetos directamente en como una secuencia de bytes, por ejemplo en un fichero
 - Sirve para guardar objetos y reconstruirlos posteriormente (*persistencia*)
- **Flujos**
 - **Clase *ObjectOuputStream***
 - Método – writeObject()
 - Ejemplo: flujoSalida.writeObject (objetoClase);
 - **Clase *ObjectInputStream***
 - Método – readObject()
 - Ejemplo: objetoClase = (Clase) flujoEntrada.readObject();

Interfaz Serializable

- Cualquier clase que desee poder serializar sus objetos debe implementar la interfaz Serializable
 - En esta implementación el objeto define cómo debe almacenarse o recuperarse de un fichero con los métodos
 - **writeObject**: responsable de escribir el estado del objeto en el flujo
 - **readObject**: responsable de recuperar el estado del objeto desde el flujo
 - Si se trata de serializar un objeto que no lo implementa se obtiene la excepción `NotSerializableException`

```
public interface Serializable {  
    private void writeObject(java.io.ObjectOutputStream out)  
        throws IOException  
    private void readObject(java.io.ObjectInputStream in)  
        throws IOException, ClassNotFoundException;  
}
```

Serialización de objetos

Ejemplo: serialización de un objeto que guarda un calendario

```
GregorianCalendar calendario = new GregorianCalendar();
ObjectOutputStream out = new ObjectOutputStream
    (new FileOutputStream("calendario.dat"));
out.writeObject(calendario);
out.close();
```

```
public class java.util.GregorianCalendar extends java.util.Calendar { ...
```

```
public class java.util.Calendar extends java.lang.Object implements
    java.lang.Cloneable, java.io.Serializable { ...
```

Nota: Como Calendar implementa Serializable, GregorianCalendar también.

Deserialización de objetos

- Utilizando la clase *ObjectInputStream*
 - Hay que respetar el orden en el que se guardaron los elementos de estado del objeto para poder hacer un casting al tipo correcto

```
ObjectInputStream in = new ObjectInputStream (new FileInputStream("calendario.dat"));
GregorianCalendar calendario = (GregorianCalendar)in.readObject();
in.close();
```

Resumen

- **La E/S en Java sigue el mismo modelo que en Unix:**
 - Abrir, usar, cerrar flujo
 - Flujos estándar: *System.in*, *System.out* y *System.err*
- **Dos tipos de clases de E/S:**
 - Readers y Writers para texto
 - Basados en el tipo char
 - *Streams (InputStream y OutputStream)* para datos binarios
 - Basados en el tipo byte
 - *Los flujos de E/S se pueden combinar para facilitar su uso*
 - *La E/S suele ser propensa a errores*
 - Implica interacción con el entorno exterior
 - Excepción *IOException*