

Taller de Lenguajes II

Tema de hoy: Entrada-Salida en JAVA

- Introducción
- Streams estándares: System.in, System.out, System.err
 - Ejemplos
- Clasificación de Streams de E/S
- Streams de caracteres: Reader y Writers
 - Métodos
 - Ejemplos
 - Combinaciones
- Streams de bytes: InputStream y OutputStream
 - Métodos
 - Ejemplos
 - Las clases ObjectOutputStream e ObjectInputStream

Introducción

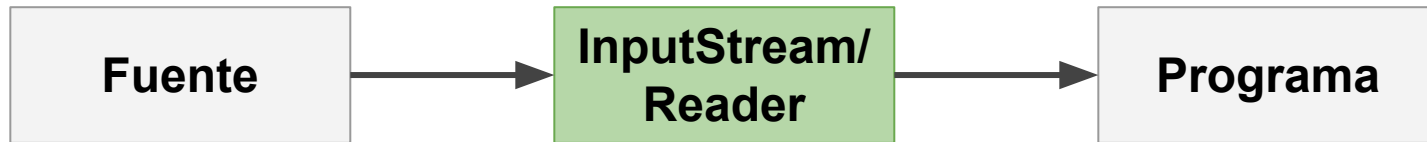
- La mayoría de los programas requieren acceder a **datos externos**: procesar datos de entrada y producir datos de salida de acuerdo a dicha entrada. Ej: leer datos de un archivo o desde la red y escribir en un archivo o devolver la respuesta en la red.
- Los **datos externos** de un programa se **recuperan** ó se **leen** desde un **origen de entrada** o **fuentes** y sus **resultados** se **envían** a un **destino de salida**.
- Una **fuentes** ó **origen de entrada** y la **salida** pueden ser muy **variados**, desde un **archivo almacenado** en el filesystem, el **teclado**, el **monitor**, un **conector de red**, hasta **otros programas**, etc. Ej. leer datos de un archivo o desde la red y escribir la respuesta en un archivo o enviarla a través de la red.

Introducción

Entrada-Salida con *streams*

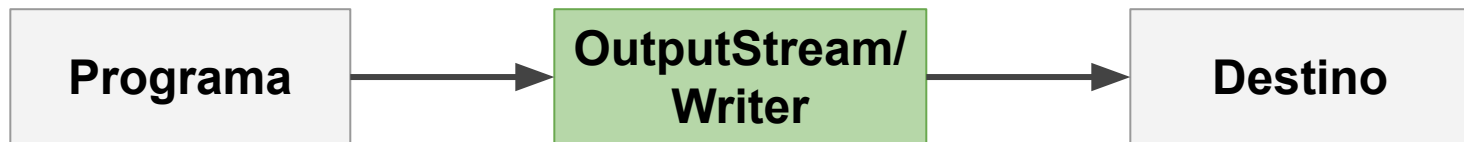
- **Java** define la abstracción llamada ***stream*** como un **flujo de datos** desde el que es posible **leer** y/o **escribir**. Un ***stream*** siempre está conectado a una **fuentes de datos** o a un **destino de datos**.
- Los ***streams*** o **flujos** son una **entidad lógica** que actúan como una **interface** con los **dispositivos de entrada y salida** y, los **programas**:
 - Independencia del tipo de datos y de los dispositivos.
 - Diversidad de dispositivos: archivos, pantalla, teclado, red.
 - Flexibilidad: es posible combinarlos.
 - Diversidad de formas de comunicación: acceso secuencial, aleatorio, la información que se intercambia puede ser binaria, caracteres, líneas, etc.

Entrada-Salida con *streams*



El **programa** que **lee datos** desde una **fuente** necesita un **InputStream** o un **Reader**. Un **stream de entrada** está **conectado** a una **fuente de datos**.

¿**Cuáles podrían ser fuentes de datos?** el teclado, un archivo en el filesystem, un socket remoto.



El **programa** que **escribe datos** en un **destino** necesita un **OutputStream** o un **Writer**. Un **stream de salida** está **conectado** a un **destino de datos**.

¿**Cuáles podrían ser destinos de datos?** la pantalla, un archivo en el filesystem, un socket local.

Los **streams** soportan **diferentes tipos de datos**, entre ellos **bytes**, **datos primitivos**, **caracteres** localizados y **objetos**. Algunos streams simplemente pasan los datos, otros los **manipulan** y los **transforman**.

Entrada-Salida con *streams*

- **Java** implementa los **streams** o **flujos de datos** a través de las **clases del paquete java.io**.
- Esencialmente **todos los flujos funcionan igual**, independientemente del dispositivo con el que se esté trabajando.
- El paquete IO se ocupa de la **lectura de datos** sin formato desde una **fuentes** y la **escritura de datos** sin formato en un **destino**. Las **fuentes** y **destinos** de datos más típicos son estos:
 - Archivos
 - Sockets
 - Buffers en memoria: arreglos
 - System.in, System.out, System.error

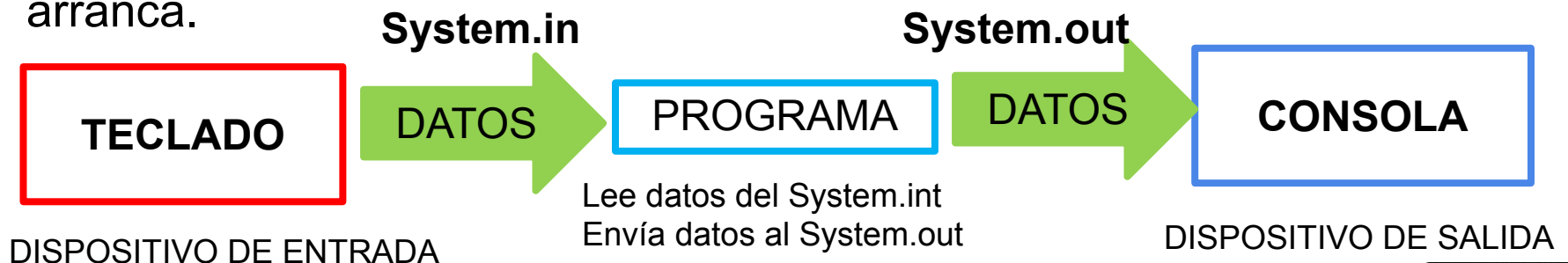
Streams Estándares

Algo conocido....

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

- **System.in** implementa la **entrada** estándar. Es un **InputStream** conectado con el teclado.
- **System.out** implementa la **salida** estándar. Es un **PrintStream** (**OutputStream**) conectado con la consola.
- **System.err** implementa la **salida** de error. Es un **PrintStream** (**OutputStream**) conectado con la consola.

Estos 3 streams están listos para usar, los instancia la JVM cuando arranca.



Streams Estándares

System.in

- Es una instancia de la clase **InputStream**: flujo de bytes de entrada.
- Típicamente conectado al **teclado** para programas de línea de comando.
- Métodos:
 - `read()`: lee un byte de la entrada como un entero
 - `skip(n)`: ignora n bytes de la entrada
 - `available()`: devuelve el número estimado de bytes disponibles para leer de la entrada.

System.out

- Es una instancia de clase **PrintStream (OutputStream)**: flujo de bytes de salida.
- Los datos se escriben en la **consola**. Es usado frecuentemente por programas de línea de comando.
- Métodos para imprimir datos:
 - `print()`, `println()`
 - `flush()`: vacía el buffer de salida y escribe su contenido

System.err

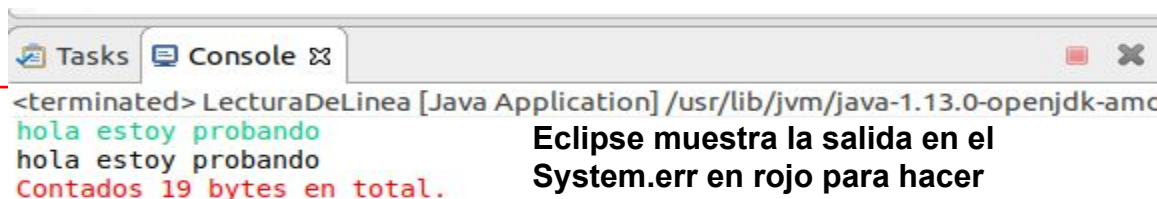
- Es similar al `System.out`.
- Se usa para enviar mensajes de error a la consola.

Streams Estándares

Ejemplo

```
package taller.entradasalida;
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 +" caracteres en total." );
    }
}
```



The screenshot shows the Eclipse IDE's console window. It has tabs for 'Tasks' and 'Console'. The console output is as follows:

```
<terminated> LecturaDeLinea [Java Application] /usr/lib/jvm/java-1.13.0-openjdk-amc
hola estoy probando
hola estoy probando
Contados 19 bytes en total.
```

Eclipse muestra la salida en el System.err en rojo para hacer más obvio que es un error

Streams de E/S

Clasificación

Representación de la información

- Flujos de bytes: clases **InputStream** y **OutputStream**
- Flujos de caracteres: clases **Reader** y **Writer**

Se puede pasar de un flujo de bytes a uno de caracteres con **InputStreamReader** y **OutputStreamWriter**

Propósito

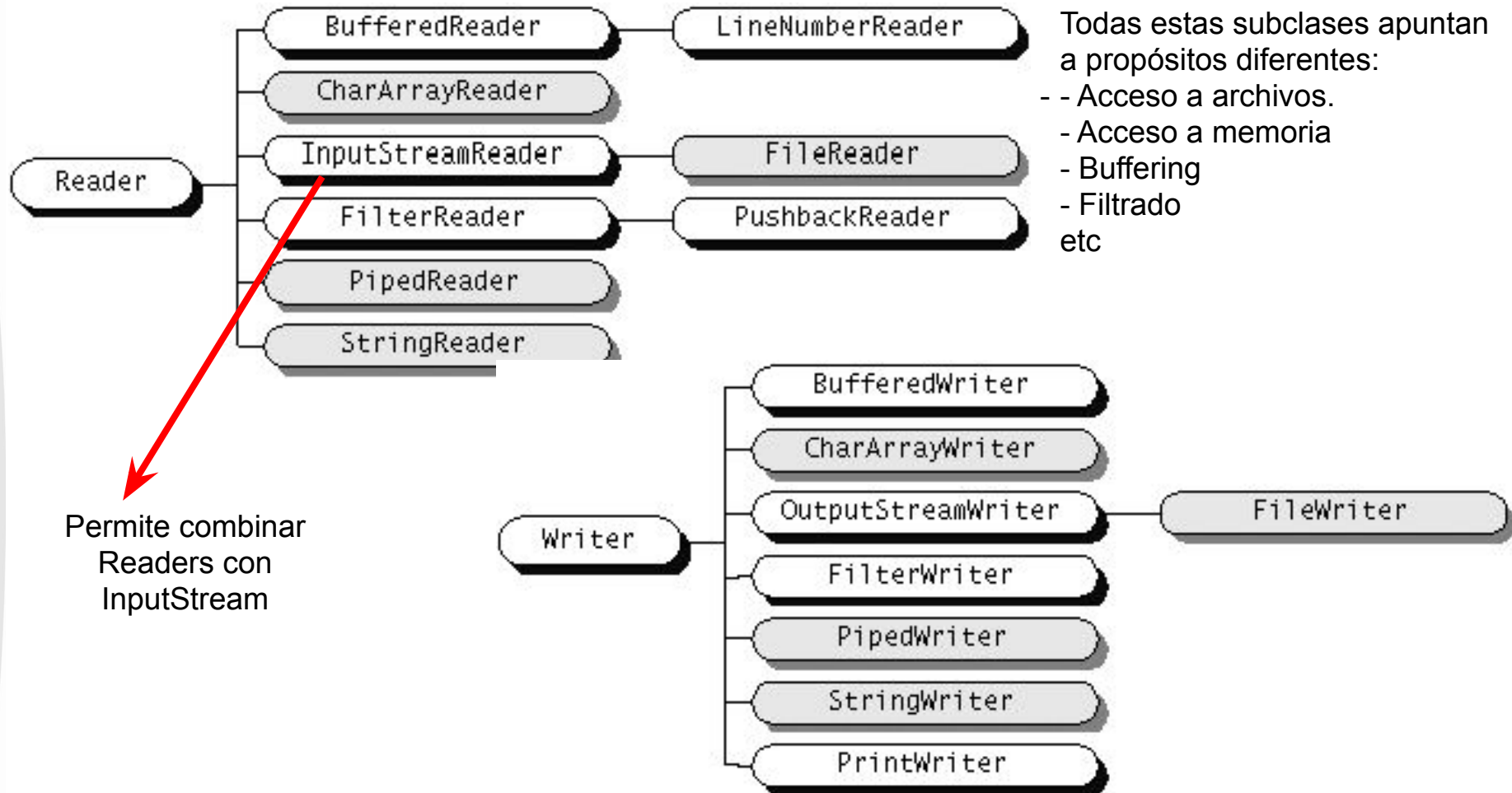
- Entrada: clases **InputStream** y **Reader**
- Salida : clases **OutputStream** y **Writer**
- Entrada/Salida: clase **RandomAccessFile**
- Transformación de datos: realizan algún tipo de procesamiento sobre los datos (p.ej. buffering, conversiones, filtrados): las clases **BufferedReader**, **BufferedWriter**

Acceso

- Secuencial
- Aleatorio (clase **RandomAccessFile**)

Jerarquía de streams de caracteres

Readers y **Writers** proporcionan un medio para el manejo de entradas y salidas de **caracteres**. Dichos flujos usan codificación **Unicode**. Se usan para leer/escribir texto.



Streams de caracteres

Las subclases de **Reader** y **Writer** implementan streams específicos.

- Aquellos que leen desde o escriben en memoria (fondo gris): arreglos, archivos, strings, etc.
- Aquellos que realizan algún procesamiento o transformación (fondo blanco): buffering, filtrado, ect.

Streams de caracteres

Los métodos de **Reader**:

int read()

retorna un carácter Unicode o -1 en caso de error o fin de archivo.

int read(char[] cbuf)

int read(char[] cbuf, int offset, int length)

leen y almacenan el resultado en un arreglo. Los parámetros *offset* y *length* son usados para indicar un sub-rango en el arreglo destino que necesita ser completado.

Streams de caracteres

Ejemplo: lectura

```
public class LeerFileReader {  
    public static void main(String[] args) throws FileNotFoundException, IOException {  
        Reader reader = new FileReader("/home/claudia/Documentos/txtfile.txt");  
        int data;  
        try {  
            data = reader.read();  
            while(data != -1){  
                char dataChar = (char) data;  
                System.out.print(dataChar);  
                data = reader.read();  
            }  
        } catch (IOException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        } finally {  
            if (reader!=null) reader.close();  
        }  
    }  
}
```

Streams de caracteres

Los métodos de **Reader** (cont.):

void close(): cierra el flujo y libera cualquier recurso asociado a él.

boolean ready(): indica si el stream está listo para ser leído.

long skip (long n): omite n caracteres.

boolean markSupported(): retorna *true* si el stream soporta el método `mark()`.

void mark(int readAheadLimit): marca el byte del *stream* sobre el que estamos posicionados.

void reset(): nos reposiciona en el byte del stream que marcamos .

Streams de caracteres

Los métodos de **Writer**:

void write(int c)

void write(char[] cbuf)

void write(char[] cbuf, int offset, int length)

void write(String string)

void write(String string, int offset, int length)

Escriben en el flujo de datos un carácter, un arreglo de caracteres (o parte de él) o un string (o parte de él).

void close(): cierra el flujo y libera cualquier recurso asociado a él.

void flush(): vacía el flujo.

Streams de caracteres

Ejemplo: escritura

```
public class EscribirFileWriter {  
  
    public static void main(String[] args) throws IOException {  
        Writer writer = new FileWriter("/home/claudia/Documentos/file-output.txt");  
        writer.write("Hola mundo Writer");  
        writer.close();  
    }  
  
}
```


Streams de caracteres

Clases básicas

La clase InputStreamReader:

Lee bytes de un flujo `InputStream` y los convierte en caracteres `UNICODE`. Es un puente entre flujos de bytes y flujos de caracteres.

```
Reader reader = new InputStreamReader(inputStream);
```

La clase OutputStreamWriter:

Los caracteres escritos en el `OutputStreamWriter` son codificados a bytes. Es un puente entre flujos de bytes y flujos de caracteres.

```
Writer writer = new OutputStreamWriter(outputStream);
```

Streams de caracteres

Clases básicas

Las clases **BufferedReader** y **BufferedWriter**

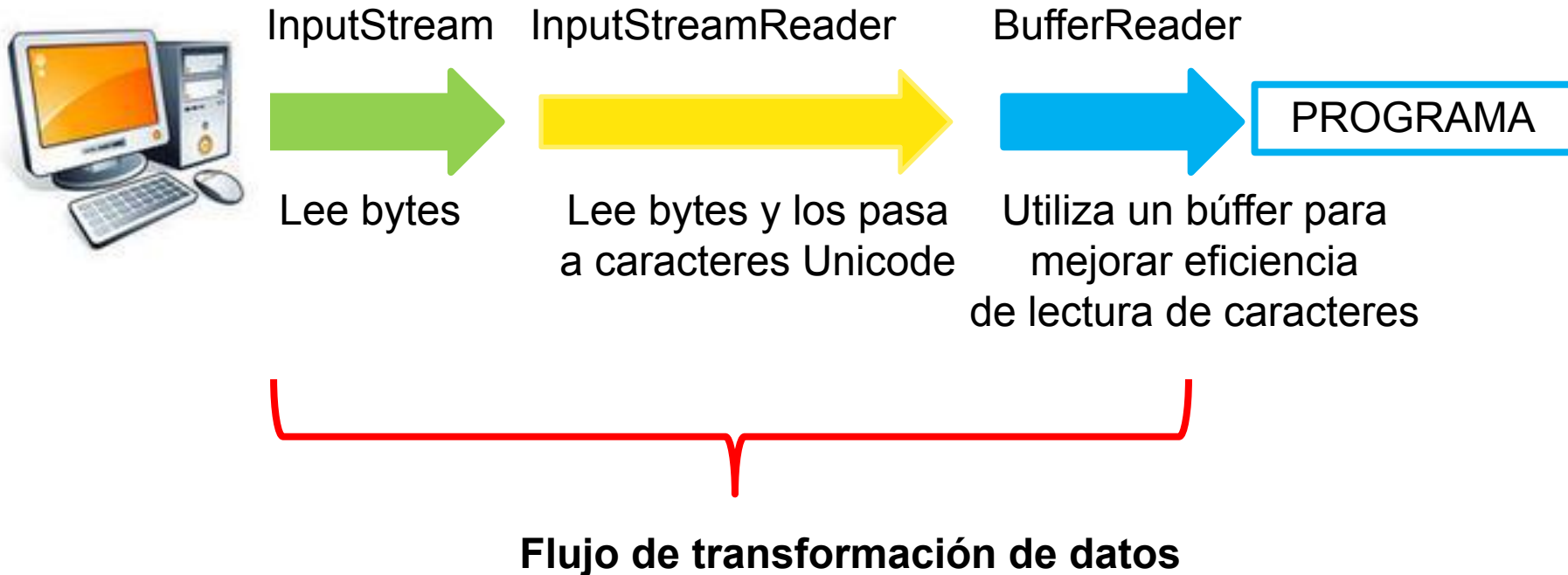
Incrementan la eficiencia de la lectura y escritura de los streams de caracteres usando técnicas de *buffering*.

```
package taller.entradasalida;
import java.io.*;
public class Eco {
    public static void main(String[] args) throws IOException {
        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 + "\"");
    }
}
```

Streams de caracteres

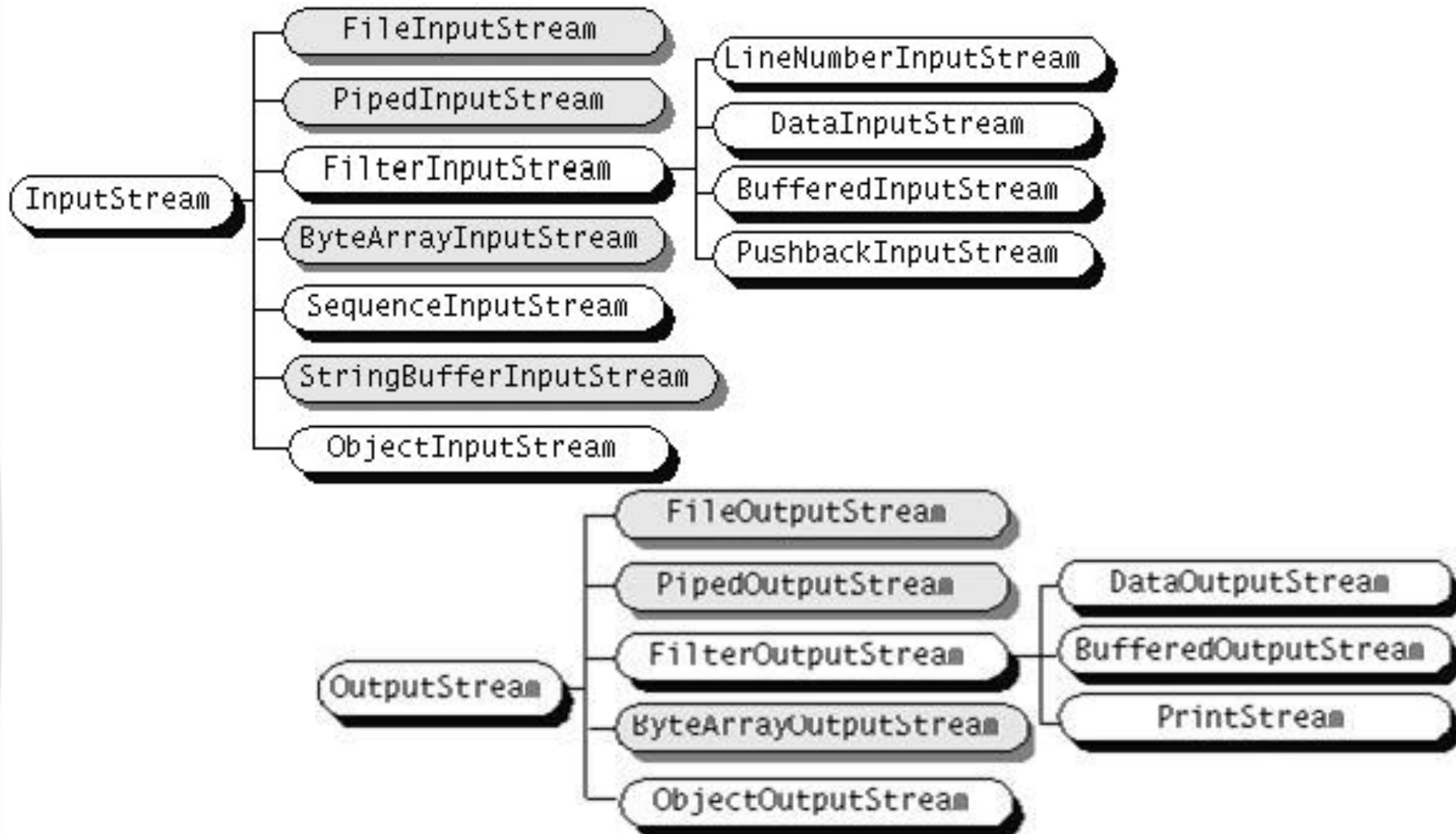
Clases básicas



Los flujos se pueden combinar para obtener la funcionalidad deseada

Jerarquía de flujos de bytes

Proporcionan un medio adecuado para el manejo de **entradas y salidas de bytes** y su uso está orientado a la **lectura y escritura de datos binarios**.



Streams de bytes

Las subclases de **InputStream** y **OutputStream** implementan streams específicos

- Aquellos que leen desde o escriben en lugares de memoria (fondo gris).
- Aquellos que realizan algún procesamiento (fondo blanco).
- **ObjectInputStream** y **ObjectOutputStream** son utilizados para la persistencia de objetos.
- Los métodos son similares a los de las clases **Reader** y **Writer** pero para elementos del tipo byte.

Streams de bytes

File Streams

Para leer y escribir datos desde archivos.

Object Streams

Para escribir y leer objetos.

Implementa serialización de objetos: permite guardar un objeto como una representación de bytes.

Filter Streams

Se construyen sobre otro *stream*.

Permiten manipular tipos de datos primitivos.

Ejemplos son **DataInputStream** y **DataOutputStream** para leer y escribir datos primitivos Java.

Implementa serialización de objetos: permite guardar un objeto como una representación de bytes.

Streams/flujos de bytes

File Streams

Combinar InputStreamReader y FileInputStream

```
package entradasalida;
import java.io.*;
public class LeeLineaDeArchivo {
    public static void main(String[] args) throws IOException{
        FileInputStream fstream = new
        FileInputStream("/home/claudia/Documentos/file-output.txt");
        BufferedReader br = new BufferedReader(new InputStreamReader(fstream));
        String strLine;
        while ((strLine = br.readLine()) != null) {
            //Se procesa la línea
            System.out.println (strLine);
        }
        fstream.close();
    }
}
```

La interface Serializable

Cualquier clase que desee persistir en archivos sus objetos **debe** implementar la interfaz **Serializable**.

Serializable es una interface **marca**, no contiene métodos, simplemente **indica** que las **instancias** de la clase que la implementa **pueden pasar a estado persistente**.

Asimismo una clase destinada a la serialización debe contener una variable `private static final long` variable de nombre **serialVersionUID**.

```
public class Persona implements Serializable{}
```

```
public class Direccion implements Serializable {}
```


Streams de bytes

Object Streams

Clase Persona

```
import java.io.Serializable;

public class Persona implements Serializable {
    private static final long serialVersionUID = 1L;
    private String nombre;
    private String apellido;
    private int edad;
    private Direccion domicilio;
    public Persona(){ }
    public Persona(String nom, String ape,
        int edad, Direccion dir) {
        this.nombre = nom;
        this.apellido = ape;
        this.edad = edad;
        this.domicilio = dir;
    }
    //getters y setters
}
```

```
import java.io.Serializable;

public class Direccion implements Serializable
{
    private static final long serialVersionUID = 1L;
    private String calle;
    private String numero;
    private String localidad;
    public Direccion(){ }
    public Direccion(String calle,
        String num, String loc) {
        this.calle = calle;
        this.numero = num;
        this.localidad = loc;
    }
    //getters y setters
}
```

Object Streams: Escritura de objetos

```
package entradasalida;
import java.io.*;

public class LeerEscribirObjetos {
    public static void main(String[] args) {
        ObjectOutputStream salida=null;
        ObjectInputStream entrada=null;
        try {
            //Persistir un objeto
            salida = new ObjectOutputStream(new
                FileOutputStream("/home/claudia/Documentos/miArchivo.dat"));
            salida.writeObject("Datos de una persona");
            Persona persona = new Persona("Juan","Pereyra",19, new Direccion("50", "1234", "La
                Plata"));
            //El objeto debe implementar la interface Serializable
            salida.writeObject(persona);
```

Continúa en la siguiente

Object Streams: Leer objetos persistidos

//Leer un objeto persona desde un archivo

```
entrada=new ObjectInputStream(new
FileInputStream("/home/claudia/Documentos/miArchivo.dat"));
String texto=(String) entrada.readObject();
Persona unaPersona=(Persona)entrada.readObject();
System.out.println(unaPersona.getApellido());
System.out.println(unaPersona.getNombre());
System.out.println(unaPersona.getEdad());
System.out.println(unaPersona.getDomicilio().getLocalidad());
} catch (IOException ex) {
    ex.printStackTrace();
} catch (Exception ex) {
    ex.printStackTrace();
} finally { //TODO }
}
```