

Tema 5. Flujos en Java

Metodología y Desarrollo de Programas

- Flujos
- Class File
- Entrada Orientada a Bytes
- Salida Orientada a Bytes
- Entrada Orientada a Caracteres
- Salida Orientada a Caracteres
- Entrada / Salida por Consola
- Serialización
- Lectura de un Archivo desde un Servidor de Internet

Flujos

- Un flujo es una abstracción que se refiere a un *flujo o corriente* de datos que fluyen entre un origen o fuente (productor) y un destino (consumidor).



- Entre el origen y el destino debe existir un canal por el que circulan los datos.
- Ejemplo:
 - ❖ A la entrada estándar System.in, a la salida estándar System.out, a la salida de error estándar (System.err)
 - ❖ Fichero binarios o ficheros de texto
 - ❖ Conexiones de red

Flujos II

- Creación automática de tres objetos al comenzar un programa:
 - ❖ `System.in`: entrada de flujos de bytes desde teclado
 - ❖ `System.out`: salida de datos por pantalla
 - ❖ `System.err`: salida de errores por pantalla
- La E/S (entrada/salida) de Java se define en términos de flujos (streams) de datos
- El paquete `java.io`, es el que contiene todas las clases que representan estos flujos.

Flujos III

- E/S puede estar basada:
 - ❖ En texto: **streams** de caracteres legibles
Ejemplo: el código fuente de un programa
 - ❖ En datos: **streams** de datos binarios
Ejemplo: patrón de bits de una imagen
- Los **streams de caracteres** se utilizan en la E/S basada en texto.
 - ❖ Se denominan lectores ([reader](#)) y escritores ([writer](#))
- Los **streams de bytes** se utilizan en la E/S basada en datos.
 - ❖ Se denominan [InputStreams](#) y [OutputStreams](#)
- La librería java.io lo implementa:
 - ❖ [Streams de caracteres](#): caracteres Unicode de 16 bits
 - ❖ [Streams de bytes](#): 8 bits

Clase File

- Representa realmente un camino de acceso, no necesariamente un archivo.
- Constructores:
 - ❖ File (String viaAcceso)
 - ❖ File (String directorio, String fichero)
 - ❖ File (File directorio, String fichero)
- Métodos para crear y borrar archivos o directorios, cambiar el nombre de un archivo, leer el nombre del directorio, consultar si un nombre representa un fichero o directorio, listar el contenido de un directorio (String [] list()), ...

➤ Métodos

- ❖ boolean canRead()
- ❖ boolean canWrite()
- ❖ boolean cancreateNewFile()
- ❖ boolean delete()
- ❖ boolean exists()
- ❖ String getAbsolutePath()
- ❖ String getName()
- ❖ String getParent()
- ❖ String getPath()
- ❖ boolean isAbsolute()
- ❖ boolean isDirectory()
- ❖ boolean isFile()
- ❖ boolean isHidden()
- ❖ long lastModified()
- ❖ long length()
- ❖ String[] list()
- ❖ File[] listFiles()
- ❖ boolean mkdir()
- ❖ boolean renameTo(File d)
- ❖ boolean
 setLastModifiedDate()
- ❖ boolean setReadOnly()
- ❖ String toString()
- ❖ String getFreeSpace()
- ❖ String getTotalSpace()
- ❖ String getUsableSpace()

Ejemplo: File

```
class DatosArchivo{
    public static void main (String [] args){
        File f = new File(/home/usuario/prueba.txt);
        System.out.println(
            "Ruta absoluta: " + f.getAbsolutePath()+
            "\n Puede leer: " + f.canRead()+
            "\n Puede escribir: " + f.canWrite()+
            "\n Nombre del fichero: " + f.getName()+
            "\n Padre del fichero: " + f.getParent()+
            "\n Ruta del fichero: " + f.getPath()+
            "\n Longitud: " + f.length()+
            "\n Ultima modificación: " + f.lastModified());
        if (f.isFile())
            System.out.println("Es un archivo");
        else if (f.isDirectory())
            System.out.println("Es un directorio");
    }
}
```


Ejemplo File

➤ Ejemplo

```
public class RenameFileExample {
    public static void main(String[] args)    {
        try{
            File oldfile =new File("oldfile.txt");
            File newfile =new File("newfile.txt");

            if(oldfile.renameTo(newfile)){
                System.out.println("Rename succesful");
            }else{
                System.out.println("Rename failed");
            }
        }catch(Exception e){ // if any error occurs
            e.printStackTrace();
        }
    }
}
```

```
public class FileDemo {
    public static void main(String[] args) {
        File f = null;
        boolean bool = false;
        try{
            // create new file
            f = new File("test.txt");
            // tries to delete a non-existing file
            bool = f.delete();
            // prints
            System.out.println("File deleted: "+bool);
        }catch(Exception e){
            // if any error occurs
            e.printStackTrace();
        }
    }
}
```

Ejemplo: File

➤ Ejemplo: Mostrar los ficheros de un directorio

❖ ¿Cómo funciona args?

```
import java.io.*;
import java.util.*;
public class DirList {
    public static void main(String[] args) {
        File path = new File(".");
        String[] list;
        if(args.length == 0)
            System.out.println("Desconozco el directorio a mostrar");
        else {
            list = path.list();
            Arrays.sort(list, new AlphabeticComparator());
            for(int i = 0; i < list.length; i++)
                System.out.println(list[i]);
        }
    }
}
```

Ejemplo File

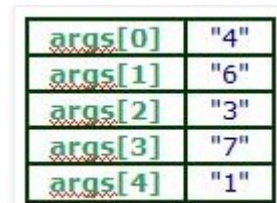
➤ args

- ❖ El parámetro args es un array de Strings que debe aparecer obligatoriamente como argumento del método main en un programa Java.

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

- ❖ Aunque se le suele dar el nombre args, no es obligatorio que este parámetro se llame así.
- ❖ Un programa java se puede ejecutar desde la línea de comandos del sistema operativo y con esta opción se puede pasar parámetros al programa desde la línea de comando

✓ java ordenar 4 6 3 7 1



args[0]	"4"
args[1]	"6"
args[2]	"3"
args[3]	"7"
args[4]	"1"

- ❖ La propiedad length del array args (args.length) contiene el número de valores enviados al programa.

Entrada Orientada a Bytes

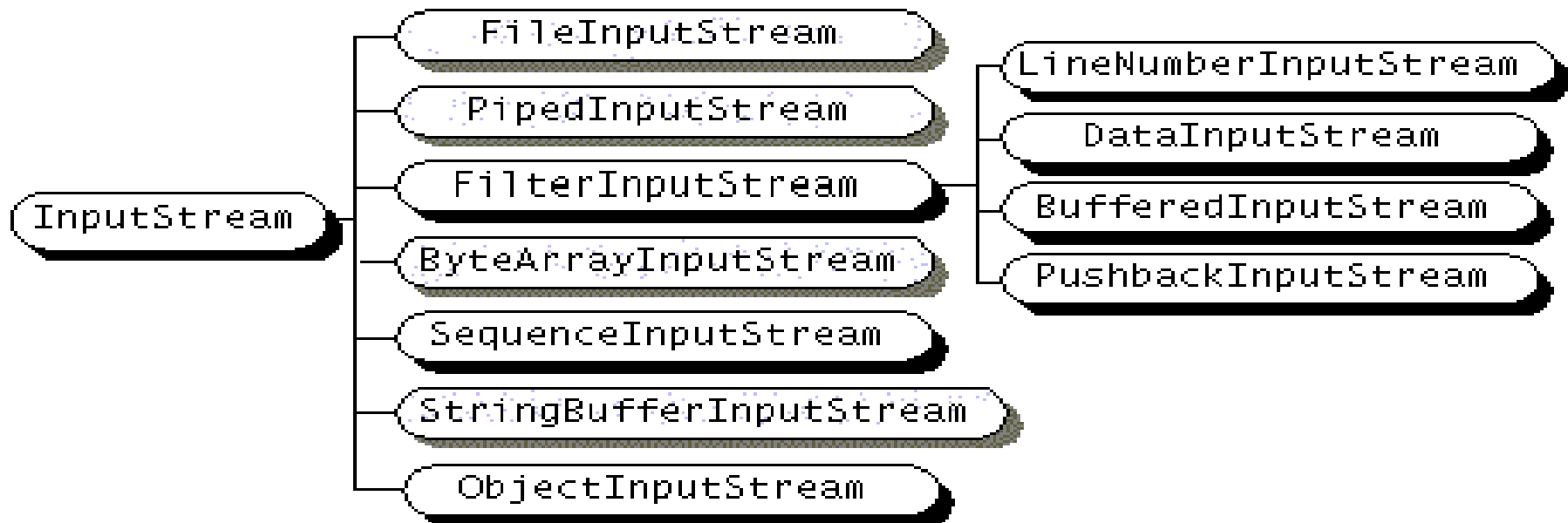
➤ Clase **InputStream** (Abstracta)

Método	Descripción
read()	Lee el siguiente byte del flujo de entrada y lo devuelve como un entero . Cuando alcanza el final del flujo de datos, devuelve -1.
read(byte b[])	Lee múltiples bytes y los almacena en la matriz b. Devuelve el número de bytes leídos o -1 cuando se alcanza el final del flujo.
read(byte b[], int off, int long)	Lee hasta len bytes de datos del flujo de entrada, empezando desde la posición indicada por el desplazamiento off, y los almacena en una matriz.
available()	Devuelve el número de bytes que se pueden leer de un flujo de entrada sin que se produzca un bloqueo por causa de una llamada a otro método que utiliza el mismo flujo de entrada.
skip(long n)	Omite la lectura de n bytes de datos de un flujo de entrada y los descarta.
close()	Cierra un flujo de entrada y libera los recursos del sistema utilizados por el flujo de datos.

Entrada Orientada a Bytes (II)

➤ Clase **InputStream** (Abstracta)

- ❖ Cada subclase debe proporcionar una implementación de estos métodos
- ❖ Algunas Subclases:
 - ✓ **ByteArrayInputStream**: permite leer bytes de un array que esté en memoria.
 - ✓ **FileInputStream**: permite leer bytes de un fichero del sistema local de ficheros.
 - ✓ **PipedInputStream**: permite leer bytes desde un pipe creado por un hilo.
 - ✓ **StringBufferInputStream**: permite leer bytes desde un String.



Entrada Orientada a Bytes (III)

- Clase **InputStream** (Abstracta) → **FileInputStream**
- Ejemplo: Leer un fichero en bytes y mostrarlo por pantalla

```
import java.io.*;
public class InputDemo {
    public static void main ( String [] args ) {
        int dato;
        try{
            InputStream entrada= new FileInputStream("prueba.dat");
            while ((dato=entrada.read())!=-1)
                System.out.write(dato);
            System.out.println("");
            entrada.close();
        }catch(IOException e){
            System.out.println("Error en lectura del fichero");
        }
    }
}
```

Entrada Orientada a Bytes (IV)

➤ Clase **InputStream** (Abstracta) → **FileInputStream**

➤ Problema de **FileInputStream**:

❖ Trabajar a tan bajo nivel, bytes te limita en cantidad, las operaciones que se pueden hacer. ¿Qué ocurre si deseamos leer una palabra? ¿o un entero? ¿o un carácter? ¿o un float?

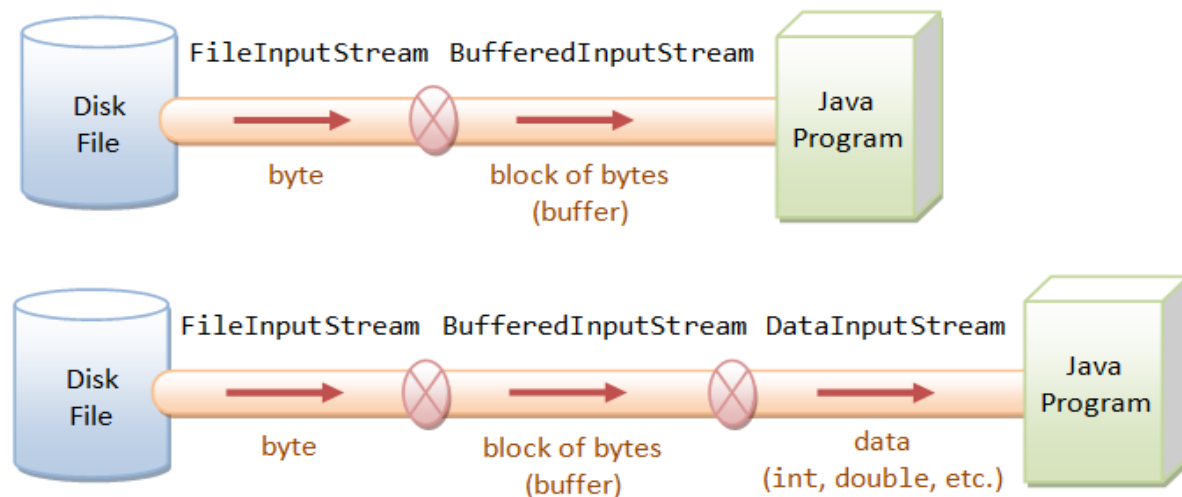
❖ **Solución:**

- ✓ Los **flujos filtrados** añaden funcionalidad a un flujo existente que permitan acceder a los datos de otros modos (entero, float,...).
- ✓ Los flujos filtrados de entrada extienden a la clase **FilterInputStream**

InputStream (Abstracta)

– **FilterInputStream** (Clase)

- **BufferedInputStream**:
- **DataInputStream**: per
líneas de texto, etc.
- **LineNumberInputStre**
- **PushBackInputSteam**



Entrada Orientada a Bytes (V)

➤ Ejemplo: **DataInputStream**

- ❖ El constructor de esta clase admite otro flujo de entrada, que es utilizado como fuente de datos (la lectura se realiza a través de éste) proporcionando una funcionalidad adicional que no posee el flujo original.

```
public class DataInputDemo {  
    //Formato de archivo: líneas de dos bytes separados por tabulador  
    public static void main(String[] args) {  
        DataInputStream dis = null;  
        try {  
            double total = 0;  
            dis = new DataInputStream(new  
                FileInputStream("DatosInputDemo.txt"));  
            while (true) {  
                double price = dis.readDouble();  
                dis.readChar(); // Elimina el tab  
                int unit = dis.readInt();  
                dis.readChar(); // Elimina el tab  
                total = total + unit * price;  
                System.out.println("Para un TOTAL de: $" + total);  
            }  
            dis.close();  
        } catch (Exception ex) {  
        }  
    }  
}
```

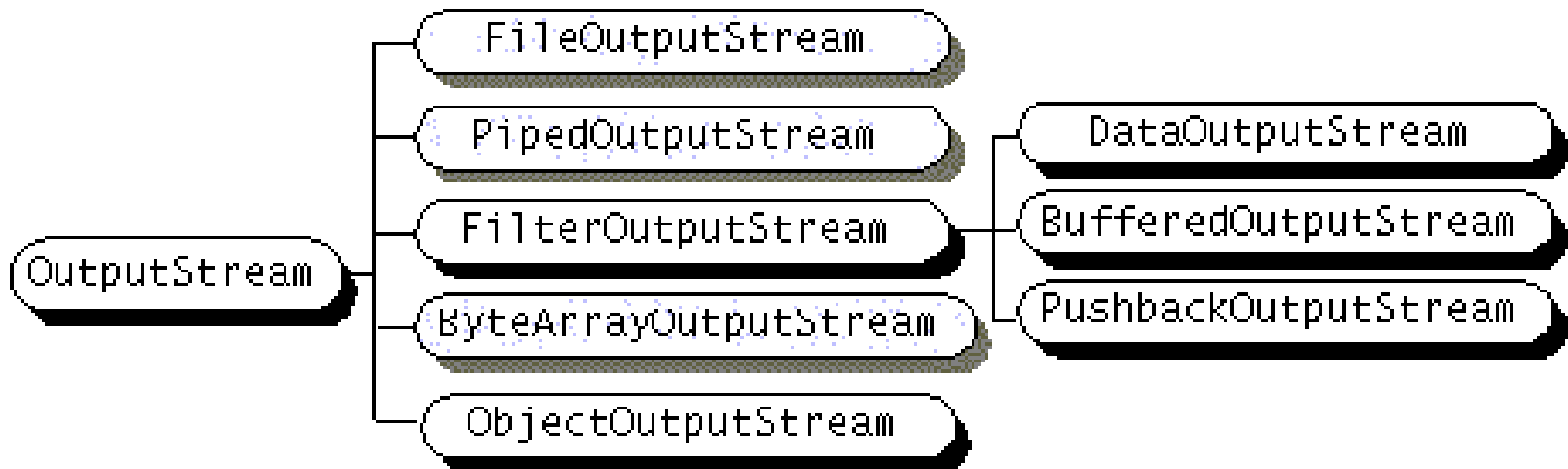

➤ Clase **OutputStream** (abstracta)

Método	Descripción
write(int b)	Escribe b en un flujo de datos de salida.
write(byte b[])	Escribe la matriz b en un flujo de datos de salida.
write(byte b[], int off, int long)	Escribe len bytes de la matriz de bytes en el flujo de datos de salida, empezando en la posición dada por el desplazamiento off
flush()	Vacía el flujo de datos y fuerza la salida de cualquier dato almacenado en el búfer.
close()	Cierra el flujo de datos de salida y libera cualquier recurso del sistema asociado con él.

Salida Orientada a Bytes (II)

➤ Clase **OutputStream** (Abstracta)

- ❖ Cada subclase debe proporcionar una implementación de estos métodos
- ❖ Algunas Subclases:
 - ✓ `ByteArrayOutputStream` escribe bytes de datos a un array que esté en memoria.
 - ✓ **`FileOutputStream`**: escribe bytes de datos a un fichero del sistema local de ficheros.
 - ✓ `PipedOutputStream`: escribe bytes de datos a un pipe creado por un hilo.



Salida Orientada a Bytes (III)

- Clase **OutputStream (Abstracta)** → **FileOutputStream**
- Ejemplo: Escribir en un fichero una serie de bytes (cadena convertida a bytes)

```
public class OutputDemo {  
    public static void main(String[] args) {  
        int dato;  
        try {  
            OutputStream salida = new FileOutputStream("prueba.dat");  
            String s = "ProgramacionII";  
            byte[] b = s.getBytes();  
            for (int i = 0; i < b.length; i++) {  
                salida.write(b[i]);  
            }  
            salida.close();  
        } catch (IOException e) {  
            System.out.println("Error de lectura en fichero prueba");  
        }  
    }  
}
```

Salida Orientada a Bytes (IV)

- Clase **OutputStream (Abstracta)** → **FileOutputStream**
- Problema de **FileOutputStream**:
 - ❖ Trabajar a tan bajo nivel, bytes te limita en cantidad, las operaciones que se pueden hacer. ¿Qué ocurre si deseamos escribir una palabra? ¿o un entero? ¿o un carácter? ¿o un float?
 - ❖ Solución: Clases Filtradas

OutputStream (Abstracta)

– **FileOutputStream**

- **BufferedOutputStream**: Permite almacenar bloques de bytes en lugar de byte a bytes.
- **DataOutputStream**: Permite escribir tipos Java primitivos.
- **PrintStream**

Entrada/Salida Orientada a Bytes

	Input	Output	Objetos Manip.
Abstractas	Input Stream	Output Stream	bytes
Interfaces	DataInput	DataOutput	Tipos Fundam
	ObjectInput	ObjectOutput	Objetos
Concretas	DataInputStream	DataOutputStream	Tipos Fundam.
	FileInputStream	FileOutputStream	Bytes
	ObjectInputStream	ObjectOutputStream	Objetos

➤ Ejercicio 1

Implementa un programa que solicite por teclado:

- ❖ Nombre (string)
- ❖ Año de nacimiento (int)

y los guarde al final de un fichero binario “info.dat”: una fila por persona.

➤ Ejercicio 2

Implementa un programa que lea el fichero “info.dat” del ejercicio anterior y lo muestre por pantalla.

➤ Notas:

- ❖ Es importante ver los métodos de cada clase, para ver cómo se puede escribir y recuperar información de forma adecuada.
- ❖ Se puede añadir información adicional al fichero (separadores de campos) si creéis que os sirve de ayuda.

Entrada Orientada a Caracteres

- Los flujos anteriores se utilizan para leer y escribir información binaria (bytes) así como tipos primitivos de datos.
- Sin embargo, para información textual es recomendable utilizar clases específicas para esta labor: **Reader** y **Writer**.
 - ❖ Leen / escriben caracteres **unicode**
 - ❖ ¿Qué es Unicode?
 - ✓ Unicode es un conjunto de caracteres extendido.
 - ✓ El código ASCII representa los caracteres usando un byte (256 símbolos)
 - ✓ Los caracteres Unicode están representados mediante uno o más bytes (dependiendo de la codificación) lo que hace que varíe entre uno y cuatro bytes.
 - ✓ Ejemplos Codificación:
 - US-ASCII: código ASCII para Latin de 7 bits.
 - UTF-8: Unicode Transformation Format 8; es similar al US-ASCII
 - UTF-16 Unicode Transformation Format de 16 bits, en el que el orden de los bytes está especificado por una marca inicial que indica el orden.
 - Etc.

Entrada Orientada a Caracteres(II)

➤ Clase **Reader** (Abstracta)

- ❖ Tiene los mismos métodos que `InputStream` exceptuando que los métodos `read()` trabajan con 2 bytes (Unicode) en lugar de con byte.
- ❖ Cada subclase debe proporcionar una implementación de estos métodos
- ❖ Algunas Subclases:
 - ✓ `CharArrayReader`: Lee de un array de caracteres.
 - ✓ **`FileReader`**: Lee de un fichero en el sistema de ficheros local.
 - ✓ `StringReader`: Lee caracteres de un `String`.
 - ✓ **`InputStreamReader`**: Permite asociar un reader a un input stream leyendo desde este último.



Entrada Orientada a Caracteres (III)

➤ Clase **Reader** (Abstracta)--> **FileReader**

- ❖ Define un método `read()` que lee carácter a carácter hasta que llega al final (-1) y lo muestra por pantalla.

```
import java.io.*;
public class ReaderDemo {
    public static void main ( String [] args ) {
        int c;
        try{
            FileReader entrada= new FileReader("prueba.txt");
            while ((c=entrada.read())!=-1) //Leo carácter a carácter
                System.out.write(c);
            System.out.println(""); //Muestro por pantalla
            entrada.close();
        }catch(IOException e){
            System.out.println("Error de lectura en fichero prueba");
        }
    }
}
```

- ❖ **FileReader** permite especificar la codificación en el constructor

```
FileReader entrada= new FileReader("prueba.txt","UTF-8");
```

➤ Clase **Reader** (Abstracta) → **FileReader**

- ❖ Los **flujos filtrados** añaden funcionalidad a un flujo existente que permitan acceder a los datos de otros modos.

- Reader
 - **BufferedReader**: Lee un conjunto de caracteres de un fichero
 - CharArrayReader
 - FilterReader
 - PushBackReader
 - **InputStreamReader**
 - **FileReader**
 - PipedReader
 - StringReader

Entrada Orientada a Caracteres (V)

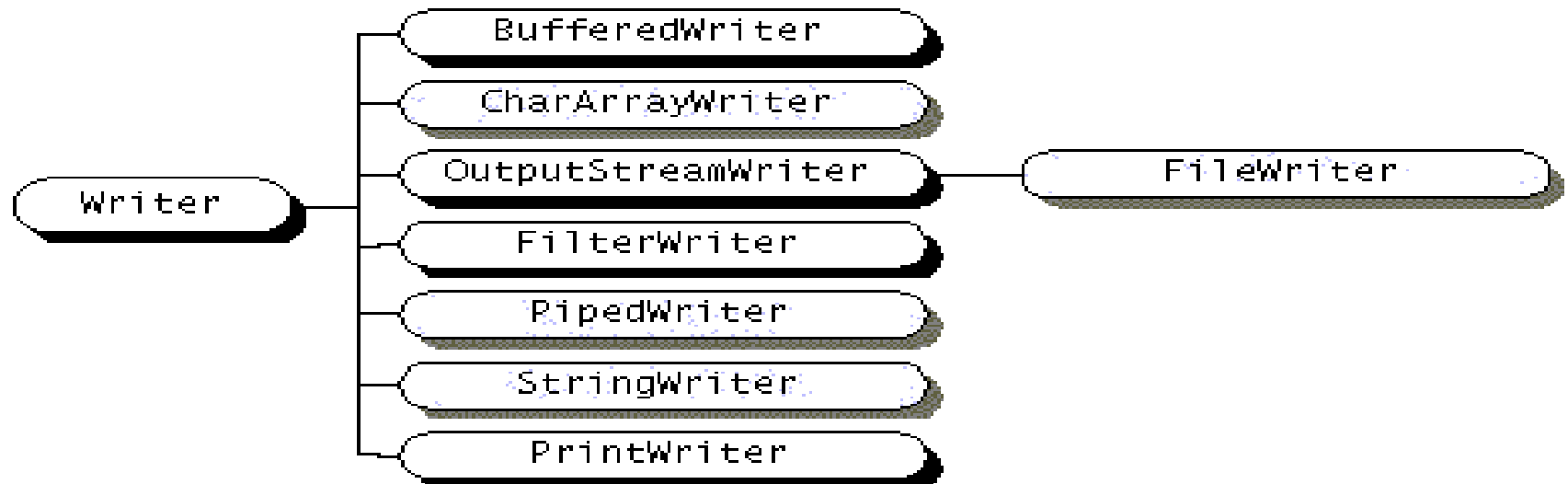
➤ Ejemplo: **BufferedReader**

- ❖ El constructor admite otro flujo de entrada, que es utilizado como fuente de datos (la lectura se realiza a través de éste).

```
public class BufferedReaderDemo {  
    //Formato de entrada: Cada línea tiene: NombrePersona:DNI:Edad  
    public static void main(String[] args) {  
        BufferedReader br = null;  
        try {  
            br = new BufferedReader(new InputStreamReader(new  
                FileInputStream("Persona.txt")));  
            String line=null;  
            int cont=0;  
            Persona[] p= new Persona[100]; //Vector de personas  
            while ((line=br.readLine())!=null) { //Leo línea a línea  
                //divido la línea por cada : que presenta un nombre:dni:edad  
                String[] split=line.split(":");  
                p[cont]=new Persona(split[0],split[1],Integer.parseInt(split[2]));  
                cont+=1;  
            }  
            br.close();  
        } catch (Exception ex) {  
        }  
    }  
}
```

➤ Clase **Writer** (abstracta)

- ❖ Una clase que la extienda debe implementar los métodos `write(char[], int, int)`, `flush()` y `close()`.
- ❖ Cada subclase debe proporcionar una implementación de estos métodos
- ❖ Algunas Subclases:
 - ✓ `CharArrayWriter`: Escribe en un array de caracteres.
 - ✓ `FileWriter`: Escribe en un fichero en el sistema de ficheros local.
 - ✓ `StringWriter`: Escribe caracteres en un `String`.
 - ✓ `OutputStreamWriter`: Permite asociar un writer a un output stream escribiendo en éste último.



Salida Orientada a Caracteres (II)

➤ Clase Writer (**Abstracta**) --> **FileWriter**

- ❖ Write para escribir caracteres Unicode
- ❖ Ejemplo: Escribo en un fichero una cadena carácter a carácter

```
import java.io.*;
public class WriterDemo {
    public static void main ( String [] args ) {
        int c;
        try{
            FileWriter entrada= new FileWriter("prueba.txt");

            String s="ProgramacionII";
            for (int i=0;i<s.length();i++)
                entrada.write(s.charAt(i));
            entrada.close();
        }catch(IOException e){
            System.out.println("Error de lectura en prueba");
        }
    }
}
```

Salida Orientada a Caracteres (III)

- Clase **Writer** (Abstracta)
- Solución: Clases Filtradas
 - **Writer**
 - **BufferedWriter**: Escribe caracteres de un fichero, almacenando los caracteres para que la lectura sea eficiente.
 - **CharArrayWriter**
 - **FilterWriter**
 - **PushBackWriter**
 - **OutputStreamWriter**
 - **PrintWriter**: Permite escribir tipos primitivos y objetos a un flujo de salida orientado a caracteres.
 - **PipedWriter**
 - **StringWriter**

Salida Orientada a Caracteres(IV)

➤ Clase **Writer** (Abstracta) → **BufferedWriter**

- ❖ El constructor admite otro flujo de entrada, que es utilizado como fuente de datos (la lectura se realiza a través de éste).
- ❖ Ejemplo: Escritura de una cadena en bloque.

```
import java.io.*;
public class BufferedWriterDemo {
    public static void main(String[] args) {
        int dato;
        BufferedWriter dis = null;
        try {
            dis = new BufferedWriter(new OutputStreamWriter(
                                   new FileOutputStream("Persona.txt")));
            String s="ProgramacionII";
            dis.write(s);
            dis.close();
        } catch (Exception ex) {
        }
    }
}
```

Salida Orientada a Caracteres(IV)

➤ Clase **Writer (Abstracta)** → **PrintWriter**

- ❖ Permite escribir bloques de información.
- ❖ Ejemplo: La escritura se realiza gracias al método `toString` de persona. El formato del fichero lo determina este método.

```
import java.io.*;
public class BufferedWriterDemo {
    public static void main(String[] args) {
        int dato;
        PrintWriter dis = null;
        try {
            dis = new PrintWriter(new BufferedOutputStream(
                new FileOutputStream("Persona.txt")));
            Persona p=new Persona("Luis","10",10);
            dis.println(p.toString());
            dis.close();
        } catch (Exception ex) {
        }
    }
}
```


Entrada/Salida estándar

➤ Stream de salida estándar: **System.out**

- ❖ Objeto `PrintStream` (es un tipo de `OutputStream`)
- ❖ Métodos de escritura `print(valor)` y `println(valor)` para los siguientes tipos:

<code>char</code>	<code>int</code>	<code>float</code>	<code>Object</code>	<code>boolean</code>
<code>char[]</code>	<code>long</code>	<code>double</code>	<code>String</code>	

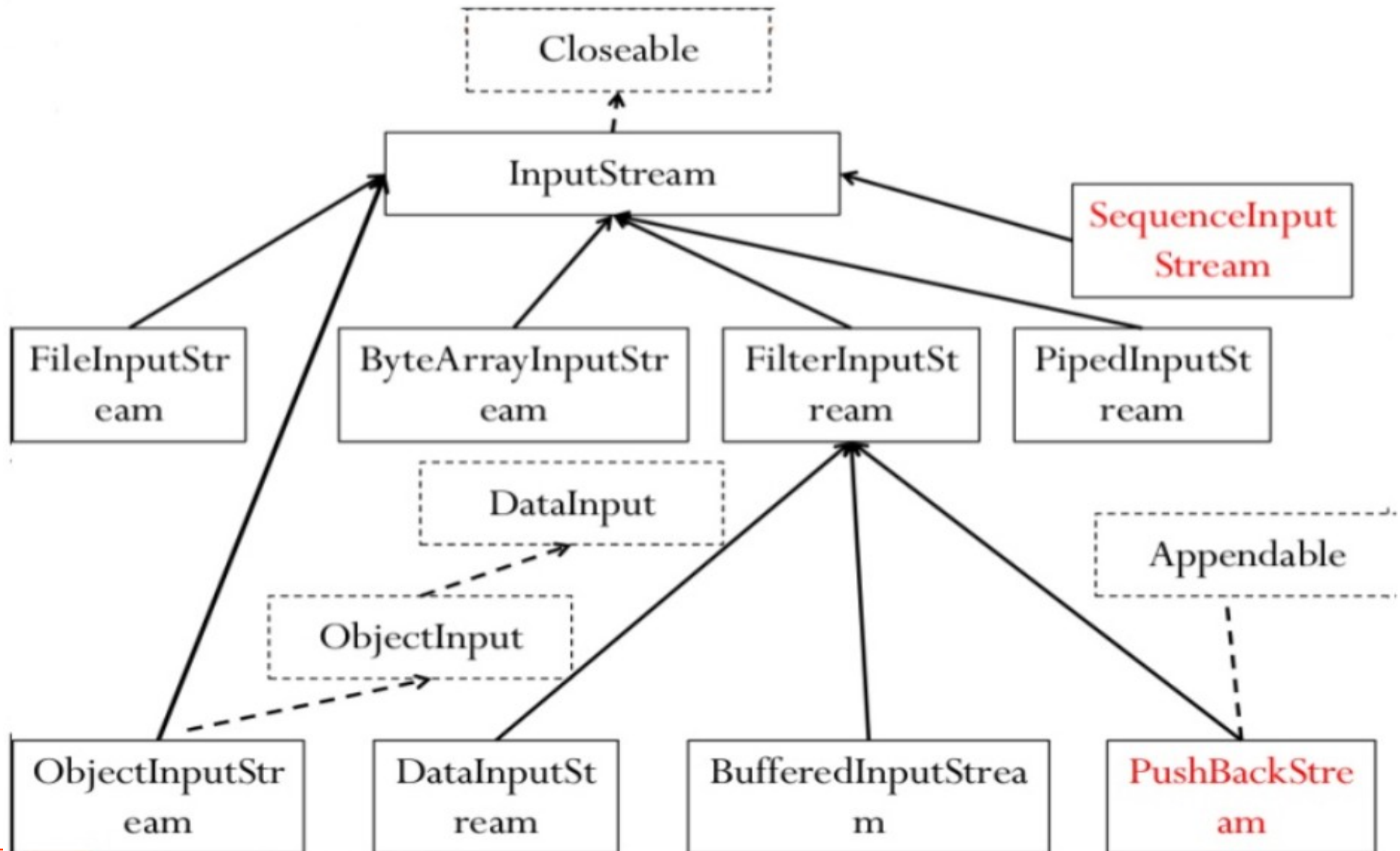
➤ Stream de entrada estándar: **System.in**

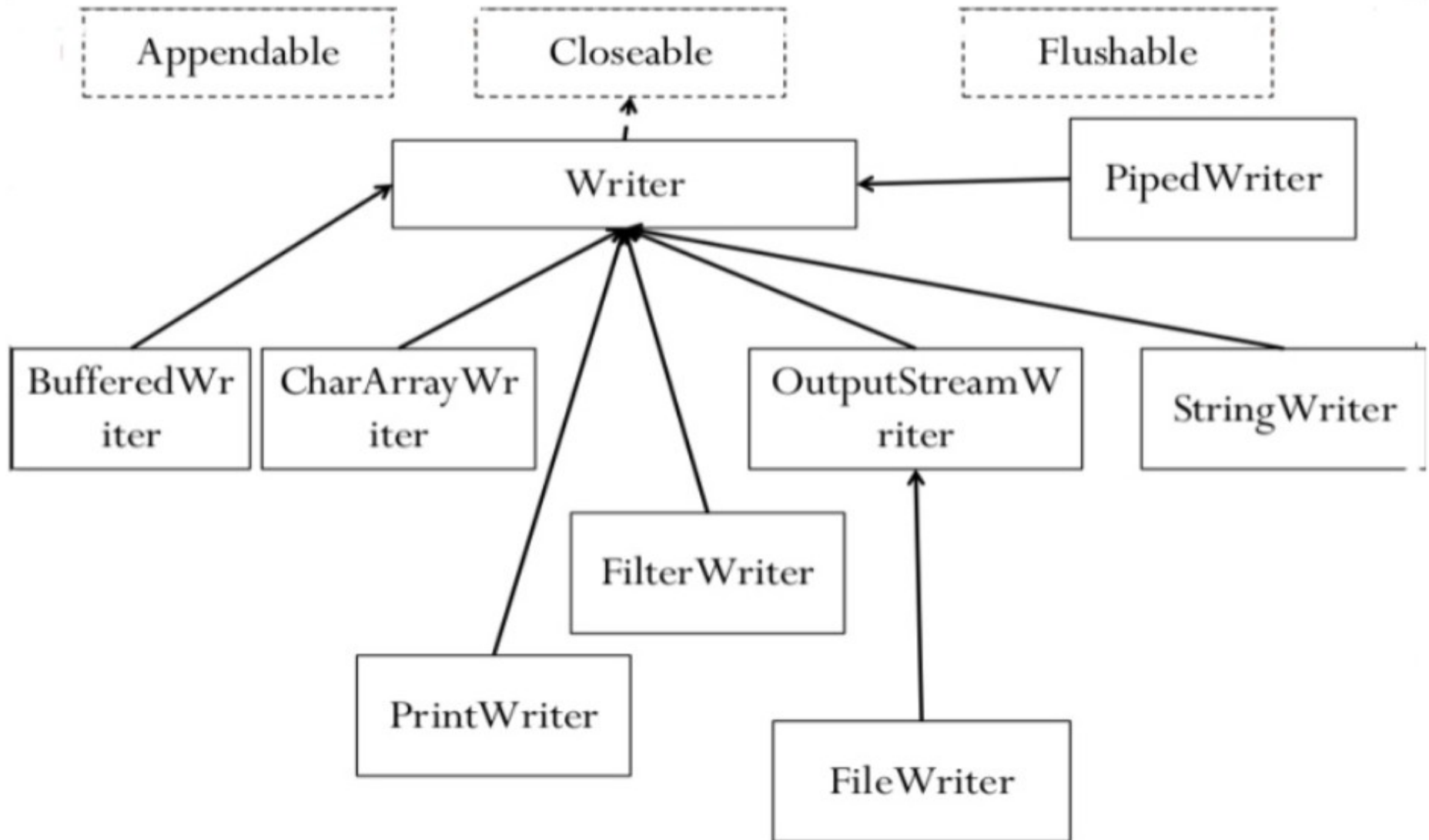
- ❖ Objeto `InputStream`

`BufferedReader br = new BufferedReader(new InputStreamReader(System.in));`

➤ Salida de error “estándar”: **System.err**

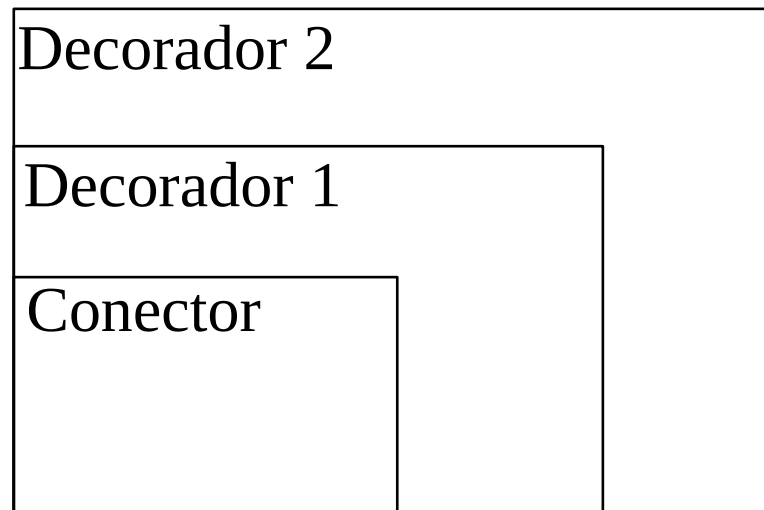
- ❖ Objeto `PrintStream`
- ❖ Mostrar mensajes de error o cualquier otra información que requiera la atención inmediata del usuario



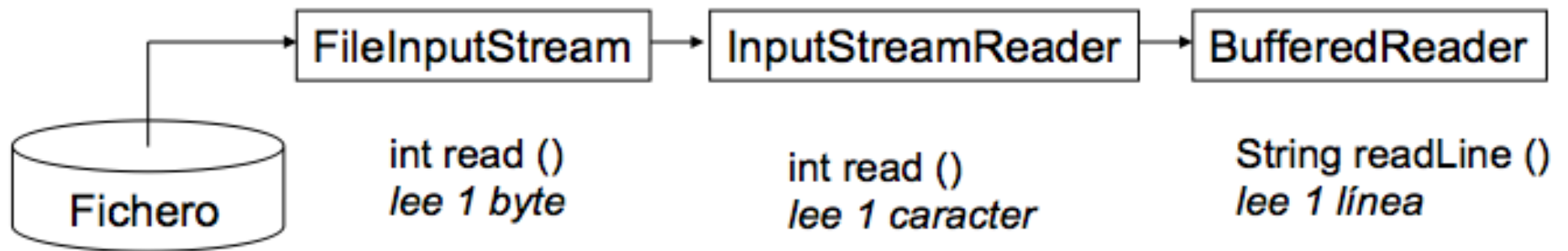


Arquitectura de clases en java.io

- Los diseñadores de Java han usado el patrón “Decorador” para estructurar la entrada/salida
 - ❖ Para usar un flujo de datos (e.g., fichero) necesitaremos “apilar” varios objetos hasta conseguir la funcionalidad deseada
 - ❖ Idea:
 - ✓ Hay una clase que conecta directamente con la fuente de información (e.g., el fichero)
 - ✓ Esta clase sólo permite leer/escribir, por ejemplo, bytes o caracteres
 - ✓ Si se quiere más funcionalidad, se usa un decorador (se denominan filtros dentro de la arquitectura de clases de Java), por ejemplo, para leer líneas o usar un buffer

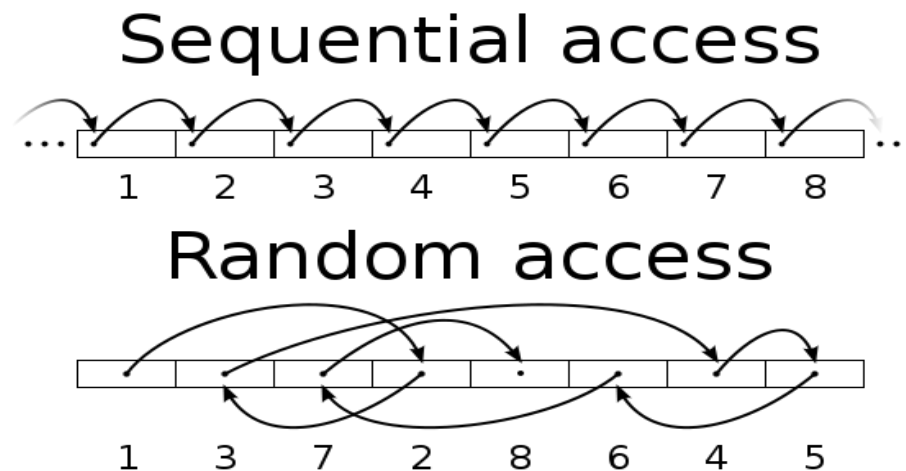


Arquitectura de clases en java.io



```
try {  
    FileInputStream fin = new FileInputStream("fichero.txt");  
    InputStreamReader isr = new InputStreamReader(fin);  
    BufferedReader miEntrada = new BufferedReader(isr);  
    String linea = miEntrada.readLine();  
    while( linea != null ) {  
        System.out.println(linea);  
        linea = miEntrada.readLine();  
    }  
}  
catch (IOException e){  
    System.out.println(e);  
}
```

- Se utiliza la clase `RandomAccessFile`
 - ❖ No está basada en el concepto de flujos o Streams.
 - ❖ No deriva de `InputStream/OutputStream` ni `Reader/Writer`
 - ❖ Permite leer y escribir sobre el fichero, no es necesario dos clases diferentes
 - ❖ Necesario especificar el modo de acceso al construir un objeto de esta clase: sólo lectura o lectura/escritura
 - ❖ Dispone de métodos específicos de desplazamiento como
 - ✓ `seek(long posicion)` o `skipBytes(int desplazamiento)` para poder moverse de un registro a otro del fichero, o posicionarse directamente en una posición concreta del fichero.



➤ Constructores:

- ❖ `RandomAccessFile(File f, String modoAcceso)`
- ❖ `RandomAccessFile(String nombreArchivo, String modoAcceso)`
- ❖ `modoAcceso` puede ser: “r” (sólo lectura) o “rw” (lectura y escritura).

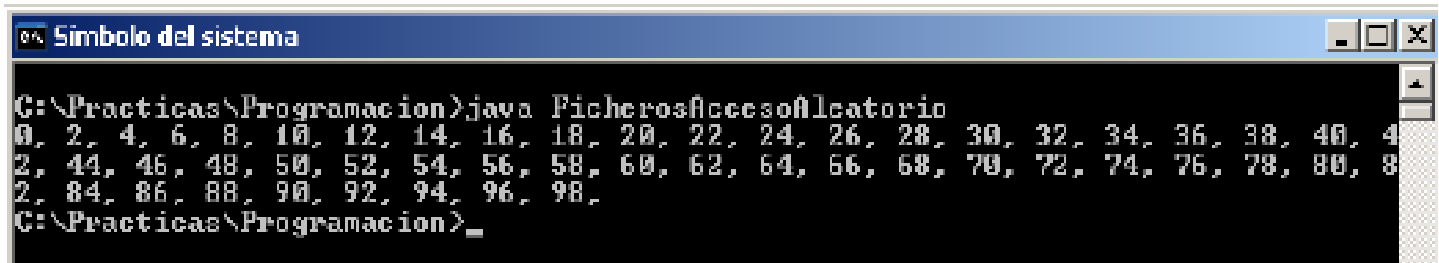
➤ Métodos:

- ❖ `void seek(long posicion)`: Sitúa el puntero de lectura/escritura en la posición indicada, desde el principio del fichero.
- ❖ `long getFilePointer()`: Devuelve la posición actual del puntero de lectura/escritura.
- ❖ `int skipBytes(int desplazamiento)`: Desplaza el puntero desde la posición actual, el número de bytes indicado por desplazamiento
- ❖ `long length()`: Devuelve la longitud o tamaño del fichero en bytes

Acceso Aleatorio

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class FicherosAccesoAleatorio {
    public static void main(String[] args) throws IOException {
        String nombre = "c:/practicas/programacion/datos4.dat";
        // Declara el fichero de acceso aleatorio
        RandomAccessFile f = new RandomAccessFile(nombre, "rw");
        // Escribe 100 bytes al fichero
        for(int i=0; i<100; i++) {
            f.write((byte) i);
        }
        // Vuelve al principio del fichero
        f.seek(0);
        // Lee uno de cada dos bytes
        for(int i=0; i<50; i++) {
            int b = f.read();
            System.out.print(b + ", ");
            f.skipBytes(1);
        }
        // Cierra el fichero
        f.close();
    }
}
```



```
Símbolo del sistema
C:\Practicas\Programacion>java FicherosAccesoAleatorio
0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98,
C:\Practicas\Programacion>
```


Serialización

- Serializar es almacenar objetos de forma permanente (p. e. en un archivo o en un flujo de comunicación)
- Consiste en convertir el objeto a bytes, para poder enviarlo por un flujo, y reconstruirlo luego a partir de esos bytes.
- Cualquier clase que desee poder serializar sus objetos debe **implementar el interfaz Serializable**.
 - ❖ Cuidado: no se serializan los miembros static.

```
public class Persona implements Serializable
{
    protected String nombre;
    protected int edad;
    protected String nif;
}
```

- Si dentro de la clase hay atributos que son otras clases, éstos a su vez también deben ser Serializable.

```
public class Contacto implements Serializable
{
    protected Persona p;
    protected int telefono;
}
```

- Las clases que necesiten algún tratamiento especial en la serialización deben implementar estos dos métodos
 - ❖ `private void writeObject (java. io. ObjectOutputStream salida) throws IOException`
 - ❖ `private void readObject (java. io. ObjectInputStream entrada) throws IOException, ClassNotFoundException;`

- Cuando un miembro dato de una clase contiene información sensible, hay disponibles varias técnicas para protegerla. Incluso cuando dicha información es privada (el miembro dato tiene el modificador `private`) una vez que se ha enviado al flujo de salida alguien puede leerla en el archivo en disco o interceptarla en la red.
 - ❖ El modo más simple de proteger la información sensible, como una contraseña (password) es la de poner el modificador **`transient`** delante del miembro dato que la guarda.

Ejemplo de Serialización

```
package Serializacion;
import java.io.*;
public class Cargar {
    public static void main(String args[]) throws IOException, ClassNotFoundException
    {
        Persona p;
        ObjectInputStream archivoObjetosEnt = new ObjectInputStream(
            new FileInputStream("datos"));
        p=(Persona) archivoObjetosEnt.readObject();
        archivoObjetosEnt.close();
    }
}
```

```
package Serializacion;
import java.io.*;
public class Salvar {
    public static void main(String args[]) throws IOException, ClassNotFoundException {
        Persona p = new Persona("Luis", "10", 10);
        ObjectOutputStream archivoObjetosSal = new ObjectOutputStream(new
            FileOutputStream("datos"));
        archivoObjetosSal.writeObject(p);
        System.out.println(p);
        archivoObjetosSal.close();
    }
}
```

Lectura de un Archivo desde un Servidor de Internet

- Java tiene un paquete `java.net` para representar direcciones en internet (URL).
- Una vez establecida una dirección, para recuperar su información sólo es necesario abrir un flujo entre el origen (nuestro ordenador) y el destino (el servidor de internet) y proceder a leer (`readLine()`) para leer un archivo desde un servidor.

```
package Socket_Basico;
import java.net.*;
import java.io.*;

public class ParseURL {
    public static void main(String[] args) throws MalformedURLException, IOException{

        URL cum = new URL("http://cum.unex.es/");
        BufferedReader in = new BufferedReader(new
                                                InputStreamReader(cum.openStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);

        in.close();
    }
}
```

Bibliografía Recomendada

➤ Libros:

- ❖ **Piensa en Java. 4ª Edición. Bruce Eckel. Pearson Prentice Hall.**
- ❖ Core Java 2. Autores Cay S. Horstmann Y Gary Cornell. Editorial Pearson Educación
- ❖ Java 2. Manual De Programación. Luis Joyanes Aguilar; Matilde Fernández Azuela. Editorial McGraw-Hill

➤ Web:

- ❖ Piensa en java como si estuviera en primero.
- ❖ <http://www.chuidiang.com>
- ❖ <http://ocw.uc3m.es/ingenieria-informatica/programacion/programa>