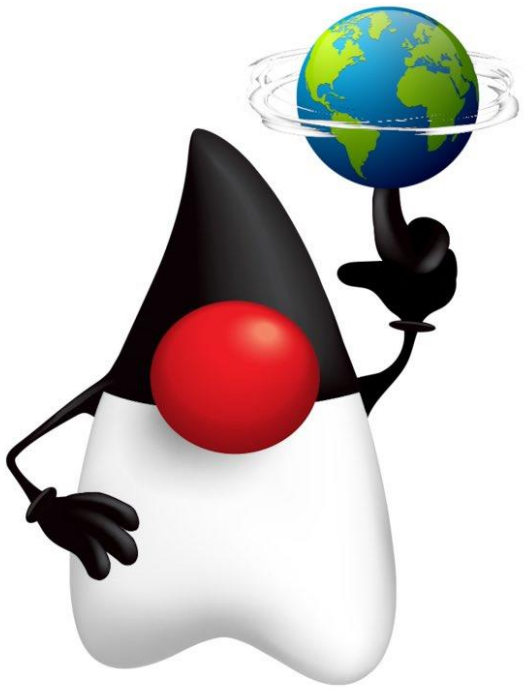


01.01. Entrada/Salida Almacenamiento de datos. Flujos

Flujos y ficheros – Pepe Calo

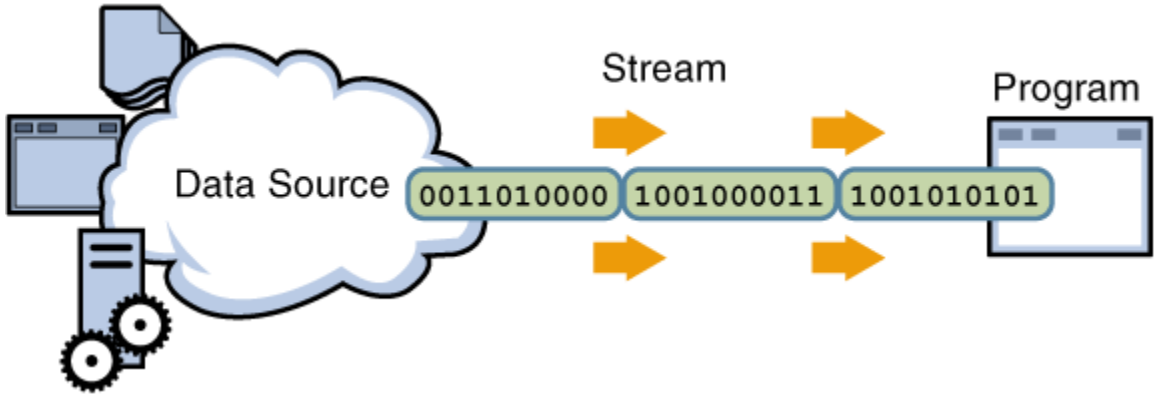
1. Flujos de entrada/salida

- Representan una fuente de entrada o un destino de salida.
- Puede representar diferentes **tipos de fuentes y destino**:
  - Archivos de disco.
  - Dispositivos.
  - Otros programas.
  - *Arrays* de memoria.
- Pueden representar diferentes **tipos de datos**:
  - Bytes simples.
  - Tipos de datos primitivos
  - Caracteres.
  - Objetos
- Algunos flujos simplemente pasan datos, otros manipulan y transforman los datos.

1. Flujos de entrada/salida
1. **Input Stream**

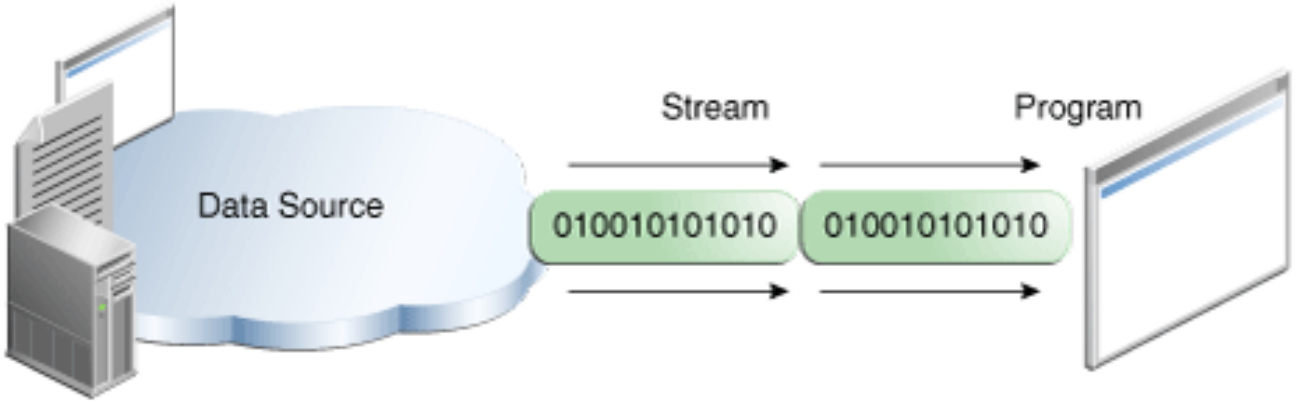
- No importa cómo trabajan internamente, representa el mismo sencillo modelo para que usen los programas:
  - Un FLUJO (***Stream***) es UNA SECUENCIA DE DATOS.

- Flujo de entrada (***input stream***):
  - El programa lee datos de una fuente, uno cada vez



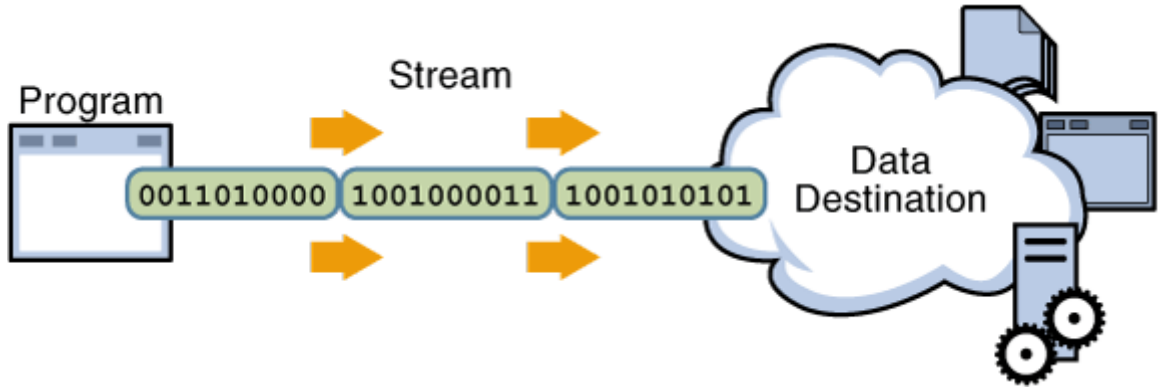
1. Flujos de entrada/salida

1. **Input Stream**



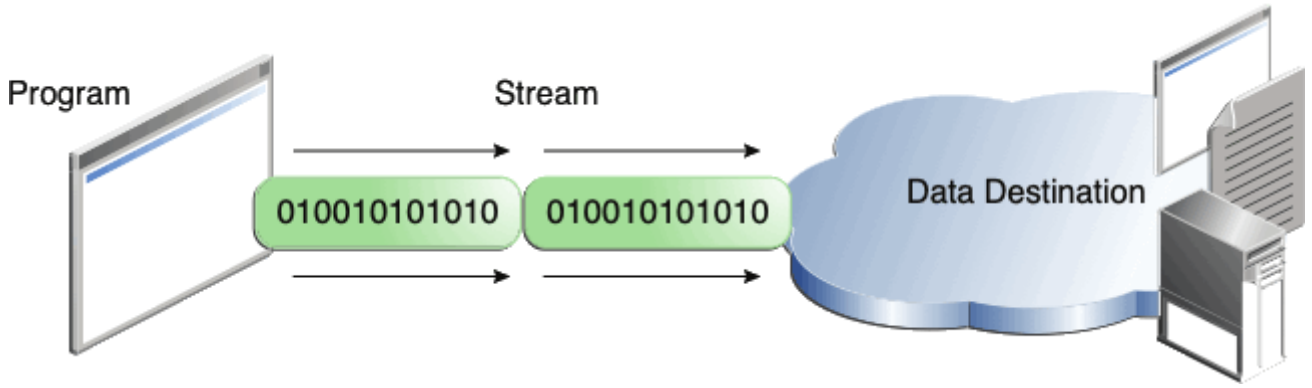
1. Flujos de entrada/salida
2. **Output Stream**

- Flujo de salida(***output stream***):
  - El programa escribe datos en una destino, uno cada vez:

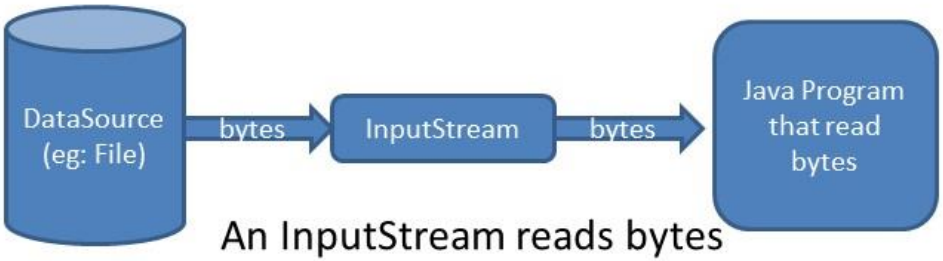


- Los flujos pueden manejar cualquier tipo de datos: primitivos u objetos.
- Fuente o destino: ***archivos*** de disco, ***programas***, ***periféricos***, ***sockets*** de red o un ***array***.

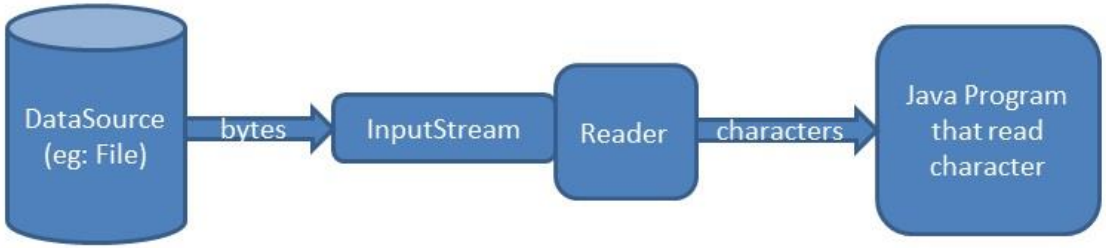
1. Flujos de entrada/salida
2. **Output Stream**



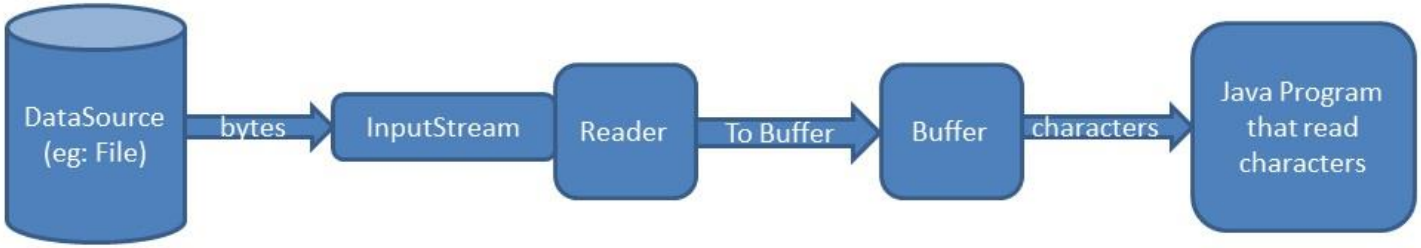
Esquema de flujos



An InputStream reads bytes

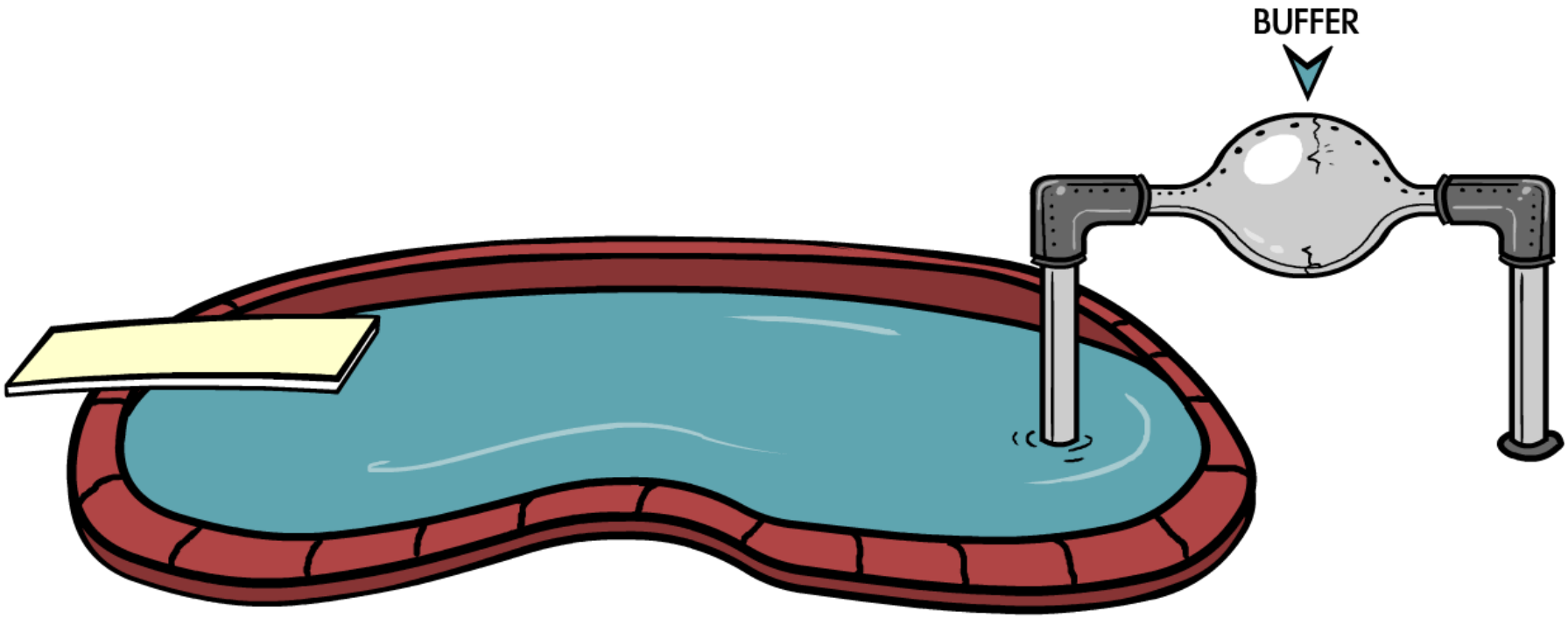


A Reader works with an InputStream and reads characters



A BufferedReader pre-fetches characters in to a buffer for performance

Un buffer o memoria intermedia



1. ***Byte Streams (flujos de bytes)***

Proporcionan entrada/salida de bytes.

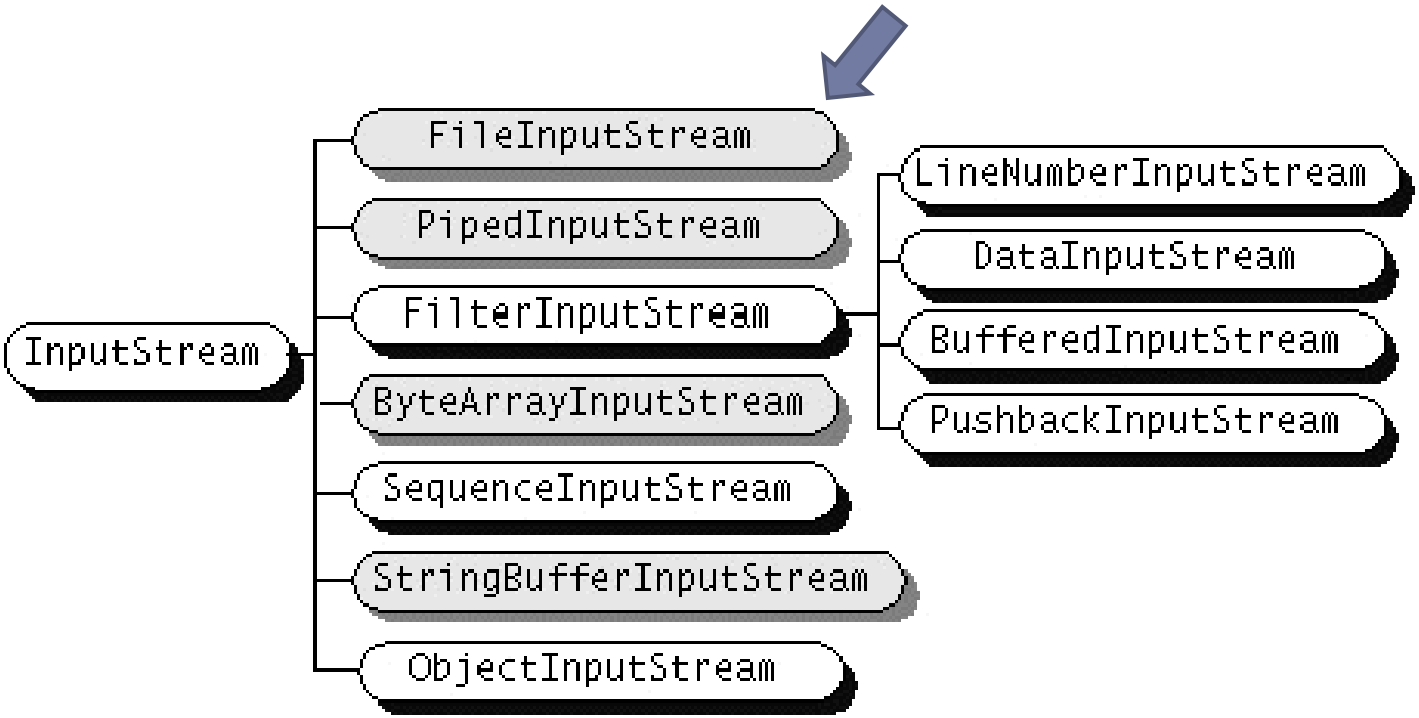
Pepe Calo

1. **Byte Streams (flujos de bytes)** *(I)*

- Flujos de bytes de 8 bits (para datos binarios)
- Todas las clases descienden (*heredan*) de ***InputStream*** y ***OutputStream***.
- Hay muchas clases de flujos de bytes. Por ejemplo, para archivos:
  - ***FileInputStream***
  - ***FileOutputStream***
- Todos los restantes flujos funcionan de igual modo, sólo difieren en la forma de construirlos.

1. **Byte Streams (flujos de bytes)** *(II)*

Ejemplo: clases principales que heredan ***InputStream***

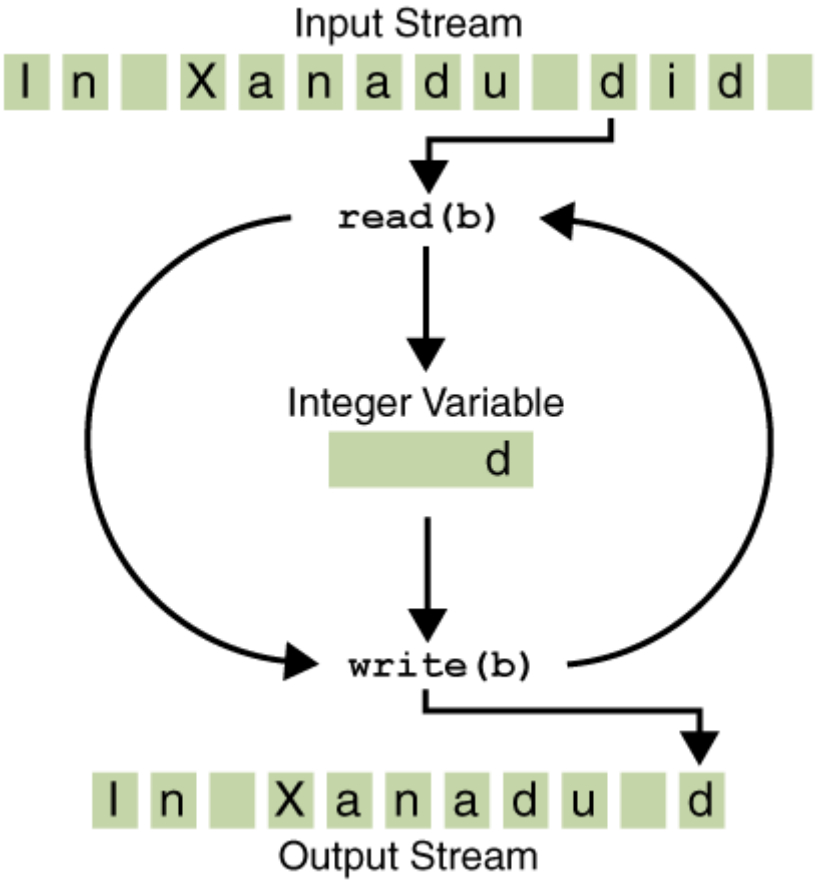


1. Byte Streams (flujos de bytes) (III)

- Ejemplo:

```
import java.io.*;
public class CopiarBytes {
    public static void main (String[] args) throws IOException {
            FileInputStream in = null;
            FileOutputStream out = null;
            try {
                    in = new FileInputStream("ficheiro.txt");
                    out = new FileOutputStream("novo.txt");
                    int c;
                    while ((c = in.read()) != -1) out.write(c);
            } finally {
                    if (in != null) in.close(); if (out != null) out.close();
            }
    }
}
```

1. **Byte Streams (flujos de bytes)** *(IV)*



1. ***Byte Streams (flujos de bytes)*** *(V)*

- El método ***read()*** devuelve un valor ***int***.
- El método ***read()*** no devuelve ***byte*** para poder **emplear el -1 como fin del flujo**.
- Hay que **cerrar siempre el flujo** con el método ***close()***. Recomendable hacerlo en el bloque ***finally*** para garantizar que también se cierra si ocurre un error.
- Un posible error es que no pueda abrir un archivo, en ese caso el flujo vale ***null***.

*1. **Byte Streams (flujos de bytes)** (VI)*

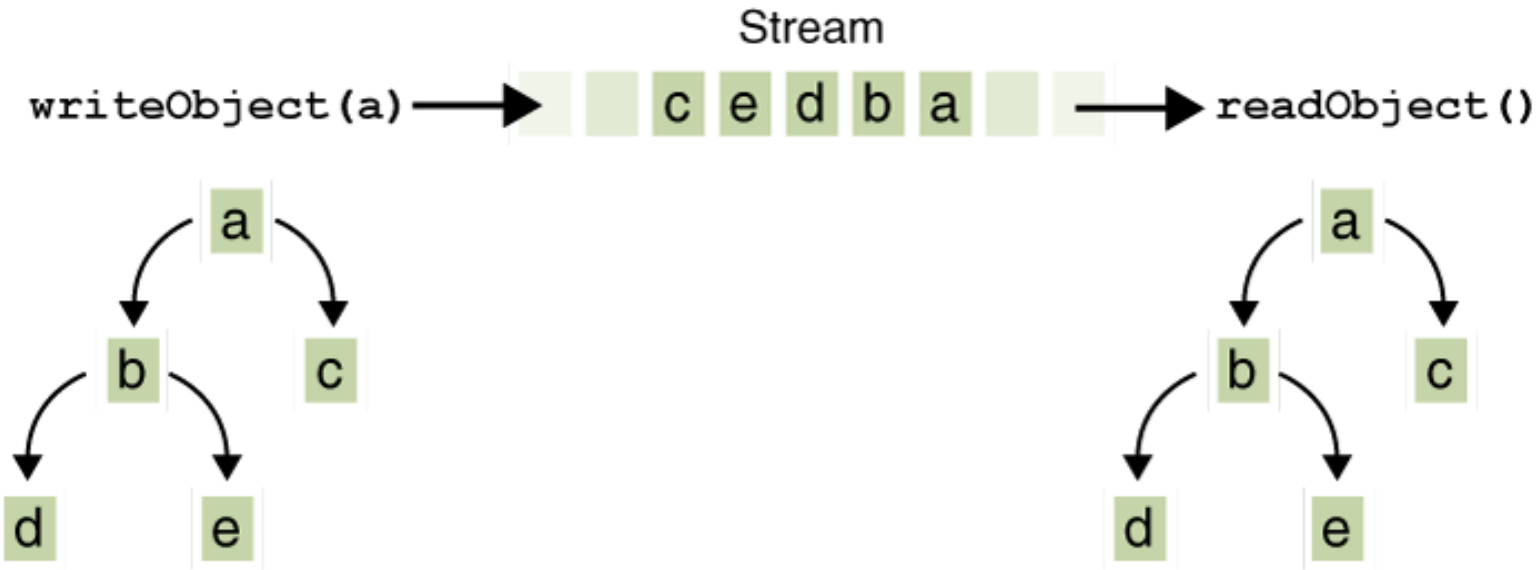
- Para ficheros de texto (con caracteres, como en el ejemplo) es mejor emplear **flujos de caracteres** (***character streams***).

- Los flujos de bytes deben **usarse sólo para E/S más primitiva** (binaria)

- Todos los otros **tipos de flujos** (incluso caracteres) se **construyen sobre los flujos de bytes**.

1. ***Byte Streams (flujos de bytes)*** *(V)*

- Otras clases de interés son: ***ObjectInputStream***, ***ObjectOutStream***, que permiten escribir y leer objetos en flujos (ficheros, sockets,…):



1. **Byte Streams (flujos de bytes)** *(VI)*

- Para emplear las clases ***ObjectInputStream***, ***ObjectOutStream*** los objetos a leer (escribir deben implantar la interface: ***Serializable*** (dicha interface no tiene métodos para implantar)

```
Object ob = new Object();
out.writeObject(ob);  //out es un flujo de tipo ObjectOutputStream
out.writeObject(ob);
```

- Para leer:

```
Object ob1 = in.readObject();
Object ob2 = in.readObject();
```

1. **Byte Streams (flujos de bytes)** *(VII)*

Clase de Stream	Descripción
BufferedInputStream	Se utiliza para el flujo de **entrada almacenado en búfer..**
DataInputStream	Contiene un método para **leer tipos de datos estándar de Java.**
FileInputStream	Empleado para **leer** desde un **archivo.**
InputStream	**Clase abstracta** que describe un **flujo de entrada (InputStream)**
PrintStream	Contiene los conocidos métodos **print() y println().**
BufferedOutputStream	Se usa para **salida con buffer** de tipo **OutputStream.**
DataOutputStream	Contiene métodos para **escribir tipos de datos estándar.**
FileOutputStream	Empleado para **escribir en un archivo.**
OutputStream	**Clase abstracta** que describe un **flujo de salida (OutputStream).**

1. ***Byte Streams (flujos de bytes)*** *(VIII)*

- Los flujos de bytes **sólo deben usarse para la E/S más primitiva**.

- Todos los **demás tipos de flujo se basan en flujos de bytes**.

- Existen flujos de caracteres que usan flujos de bytes, funcionando como clases “envolventes”:
  - ***FileReader***, por ejemplo, usa *FileInputStream*, y ***FileWriter*** usa *FileOutputStream*.

1. Byte Streams (flujos de bytes) (IX)

- Ejemplos:
  - **URL:**

```
new URL(FILE_URL).openStream();
Abreviatura de:
new URL(FILE_URL).openConnection()
                 .getInputStream();
```

  - **HttpURLConnection:**

```
URL url = new URL(FILE_URL);
HttpURLConnection httpConnection =
                 (HttpURLConnection) url.openConnection();
httpConnection.getInputStream();
httpConnection.setRequestMethod("HEAD");
long tamanho = httpConnection.getContentLengthLong();
```

1. *Ejercicios*

- Realiza los ejercicios 1.a, 1.b y 1.c del boletín 01.01

2. *Character Streams (flujos de carácter)*

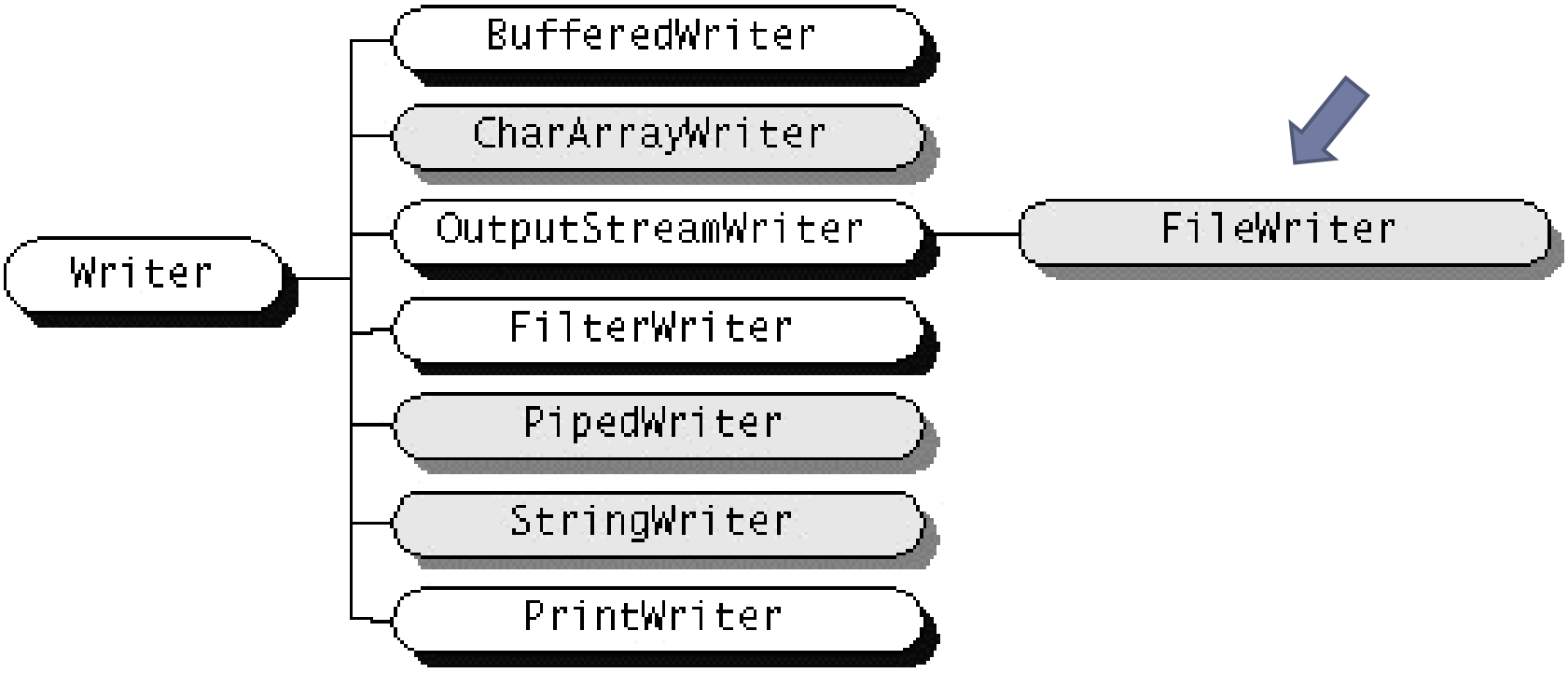
Pepinho.com 2008-2009

4. *Character Streams (flujos de carácter)*

- Automáticamente transforma caracteres Unicode al conjunto de caracteres local.
- Todas las clases descienden de ***Reader*** y ***Writer***.
- Hay muchas clases de flujos de carácter, para archivos, por ejemplo:
  - ***FileReader*** (usa internamente *FileInputStream*)
  - ***FileWriter*** (usa internamente *FileOutpuStream*)
- Todos los restantes flujos funcionan de igual modo, sólo difieren en la forma de construirlos.

4. *Character Streams (f. de carácter) (II)*

Ejemplo: jerarquía de clases de ***Writer***



4. **Character Streams (f. de carácter)** (III)

- Ejemplo:

```
import java.io.*;
public class CopiarCaracteres {
    public static void main (String[] args) throws IOException {
            FileReader in = null;
            FileWriter out = null;
            try {
                    in = new FileReader("ficheiro.txt");
                    out = new FileWriter("novo.txt");
                    int c;
                    while ((c = in.read()) != -1) out.write(c);
            } finally {
                    if (in != null) in.close(); if (out != null) out.close();
            }
    }
}
```

4. **Character Streams (f. de carácter)** (IV)

- El método ***read()*** devuelve un valor ***int***, pero en este caso contiene un valor carácter de los últimos 16 bits (2 bytes)

- El método ***read()*** no devuelve ***byte*** (ni ***char***) para poder **emplear el -1 como fin del flujo**.

- Hay que cerrar siempre el flujo con el método ***close()*** y recomendable hacerlo dentro del bloque ***finally***.

- Un posible error es que no pueda abrir un archivo, en ese caso el flujo vale ***null***.

4. **Character Streams (f. de carácter)** (V)

- Muchas operaciones de I/O trabajan con bytes (Sockets, teclado,…), por lo que existen clases para **pasar flujos de byte a carácter.**

- Existen **dos flujos "puente"** de propósito general entre de byte-a-carácter:
  - ***InputStreamReader***: sus constructores reciben un *InputStream*.
  - ***OutputStreamWriter***: sus constructores reciben *OutputStream*

- *Ejemplo*: crear flujos de caracteres desde flujos de bytes proporcionados por (clases de) sockets o leer de teclado (***System.in***)

1. Flujos de entrada/salida
4. **Character Streams (f. de carácter)** (VI)

- E/S orientada a línea
  - Permite leer más de un carácter, generalmente una línea.
  - El delimitador de línea puede ser: "\r\n", "\r" ó "\n" (depende del SO) *(ver **%n** en vez de \n)*

- Se emplean dos clases con buffer:
  - ***BufferedReader***
  - ***PrintWriter/BufferedWriter***

- Son más **eficientes** pues no leen/escriben directamente al flujo (disco, red…) y **lo hacen a través de un buffer o memoria intermedia**.

1. Flujos de entrada/salida
4. **Character Streams (f. de carácter)** (VII)

- Ejemplo con BUFFER:

```
import java.io.*;
public class CopiarLineas{
    public static void main (String[] args) throws IOException {
            BufferedReader in = null;
            PrintWriter out = null;
            try {
                    in = new BufferedReader (new FileReader("ficheiro.txt"));
                    out = new PrintWriter (new FileWriter("novo.txt"));
                    String s;
                    while ((s = in.readLine()) != -1) out.println(s);
            } finally {
                    if (in != null) in.close(); if (out != null) out.close();
            }
    }
}
```

1. Flujos de entrada/salida
4. **Character Streams (f. de carácter)** (VIII)

- El método ***readLine()*** devuelve una línea de texto.

- El método ***println()*** añade un terminado de línea del sistema operativo que lo ejecuta (no tiene porque ser el mismo que en el flujo de entrada).

- Existen otras formas de estructurar el texto de entrada y salida, más allá de caracteres y líneas: ***java.util.Scanner*** y ***System.out.format(…)***.

3. ***Buffered Streams (flujos con buffer)***

pepinho.com

3. **Buffered Streams**

- Los ejemplos anteriores (**sin buffer**) cada petición de lectura/escritura se **envía directamente al sistema E/S**: puede ser **ineficiente** (acceso a disco, actividad de red,…)

- Java implementa flujos con buffer:
  - Los **buffers de entrada** leen de memoria intermedia (buffers), el API de **entrada nativa sólo** es llamada cuando el **buffer está vacío**.
  - Los flujos de buffers de salida escriben en el **buffer de salida**, que **escribe datos sólo cuando el buffer está lleno**.

3. **Buffered Streams** (II)

- Existen cuatro clases de flujo con buffer:
  - **BufferedInputStream**
  - **BufferedOutputStream**
  - **BufferedReader**
  - **BufferedWriter**

- Ejemplo:

```
inputStream = new BufferedReader (new FileReader("entrada.txt"));
outputStream = new BufferedWriter (new FileWriter("saida.txt"));
```

3. **Buffered Streams** (III): *flushing*

- A veces interesa liberar el buffer en puntos críticos, sin esperar a que se llene: ***flushing*** (liberar) el buffer.

- Algunas clases de salida con buffer admiten ***autoflush***, especificado como argumento opcional del constructor.

- Cuando el ***autoflush*** está habilitado **ciertos tipos de eventos provocan que se libere el buffer.**

3. **Buffered Streams** (III): *flushing (2)*

- Ejemplo:
  - Con ***PrintWriter*** se libera el buffer con cada invocación a los métodos ***println*** o ***format***.
- Para invocarlo **manualmente** debe llamarse al método ***flush()***, que es válido en cualquier flujo de salida pero **sólo tiene efecto en los flujos de salida con buffer**.

4. Scanning & Formatting

pepinho.com

4. **Scanner y formateo**

- El API de Java proporciona dos modos para facilitar el trabajo con datos:
  - ***Scanner***: clase trocea la **entrada** en *tokens* individuales asociados con los bits de datos.
  - Método ***format(..)***: da formato amigable a los datos.

- java.util.**Scanner**:
  - **Trocea la entrada en tokens** y los recoge como elementos individuales de acuerdo con el tipo de dato especificado.
  - Por defecto utiliza **espacio como separador** (espacio en blanco, tabulador y terminador de línea, ver: *Character.isWhitespace()*)

4. **Scanner (II)**

- *Ejemplo:*

```
import java.io.*;
import java.util.Scanner;
public class Scaneo {
        public static void main (String[] args) throws IOException {
                Scanner s = null;
                try {
                        s = new Scanner(new BufferedReader(
                                new FileReader("datos.txt")));
                        while (s.hasNext())
                                System.out.println(s.next());
                } finally {
                        if (s != null) s.close();
                }
        }
}
```

4. **Scanner (III)**

- Aunque no es un flujo, **necesita cerrarse** con el método ***close()***, que lo hace con el flujo subyacente.

- Para usar un separador diferente debe invocarse al método: ***useDelimiter(…)***, especificando una expresión regular.

- Por ejemplo:
  - *s.useDelimiter(",\\s*");* separa por una coma seguido opcionalmente por un espacio en blanco. (\s indica espacio y * indica 0 ó 1)

4. **Scanner (IV)**

- En el ejemplo separaba por tokens leídos como *String*.
- Scanner **admite tokens de todos los datos primitivos** (excepto *char*), así como *BigDecimal o BigInteger*.
- Puede emplearse distintos tipos de separadores para valores numéricos (, .). En US "33,543" es leído como un entero, por ejemplo. (ojo con los separadores decimales por defecto en cada localidad)
- Puede especificarse la localidad (US, ES,…), ¿US por defecto?.

4. Scanner (V)

- Ejemplo:

```
//…
Scanner s = null;
double sum = 0;
try {
        s = new Scanner( new BufferedReader(
                                new FileReader("numeros.txt")));
        s.useLocale (new Locale("es")); // s.useLocale (Locale.US);
        while (s.hasNext()) {
                if (s.hasNextDouble())
                        sum += s.nextDouble();
                else s.next();
        }
} finally { s.close(); }
```

4. **Dando formato**

- Los objetos que implantan formato son o instancias de:
  - ***PrintWriter*** (para clases de flujo de carácter).
  - ***PrintStream*** (para clases de flujo de bytes) (*System.out* y *System.err*).

- Ejemplos de ***PrintStream*** son ***System.out*** o ***System.err***
- Métodos de formato:
  - ***print*** / ***println***: formatea valores individuales de modo estándar.
  - ***format/printf***: da formato a números basados en formato de cadenas con la precisión deseada (a modo *printf* de Java/C)

4. Dando formato (II)

- ***print / println*: no hay control sobre la precisión del resultado.**

```
public class Proba {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);
        System.out.print("Raíz cuadrada de ");
        System.out.print(i); // método sobrecargado
        System.out.print(" es ");
        System.out.print(r); // método sobrecargado
        System.out.println(".");
        System.out.println("Raíz cuadrada de " + i + " es " + r + "."); // toString
    }
}
```

4. Dando formato (III)

- ***format*:**
  - *Se basa en una **cadena de formato** (como printf)*
  - *Es una cadena con **especificadores de formato**.*

Ejemplo:

```
public class ProbaFormato {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);
        System.out.format("La raíz cuadrada de %d es %f.%n", i, r);
    }
}
```

4. **Dando formato (IV)**

- Los especificadores de formato empiezan por % y acaban con uno o dos **caracteres de conversión**, que especifica el tipo de salida que será formateada:
  - **d**: formatea un valor **entero** como decimal
  - **f**: formatea un valor en **punto flotante** como un valor decimal.
  - **n**: proporciona un **salto de línea** (para la plataforma que sea). Mejor que un *\n*.
  - **x**: formatea un entero como valor **hexadecimal**.
  - **s**: formatea cualquier valor como **cadena**.
  - **tB**: formatea un entero como un **nombre de mes** propio de la localidad.
  - …

(\u000A)

4. **Dando formato (IV)**

- **%n**: en Java, el carácter \n siempre genera un salto de línea (\u000A). No debe usarse a menos que deseemos un salto de línea (ved System.lineSeparator()).

4. **Dando formato (V)**

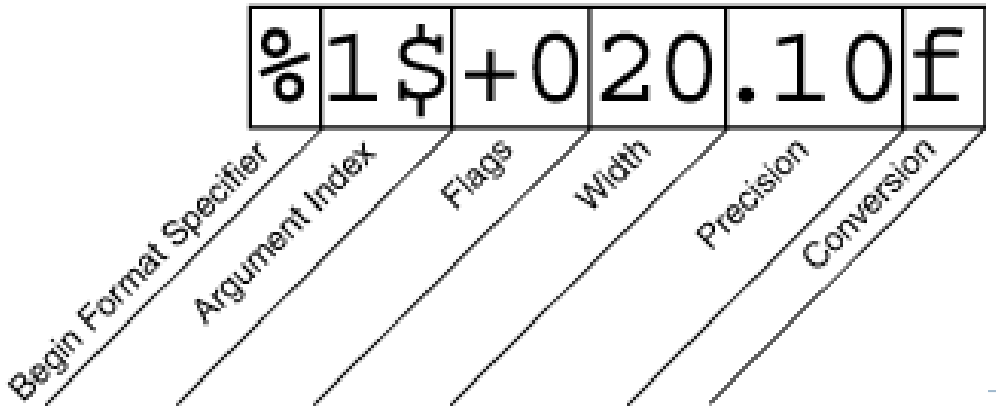
- Excepto %% y %n, **todos los especificadores de formato deben tener un argumento asignado**.

- Ejemplo:

System.out.format("%f, %1$+020.10f %n", Math.PI);

genera:

3.141593, +00000003.1415926536



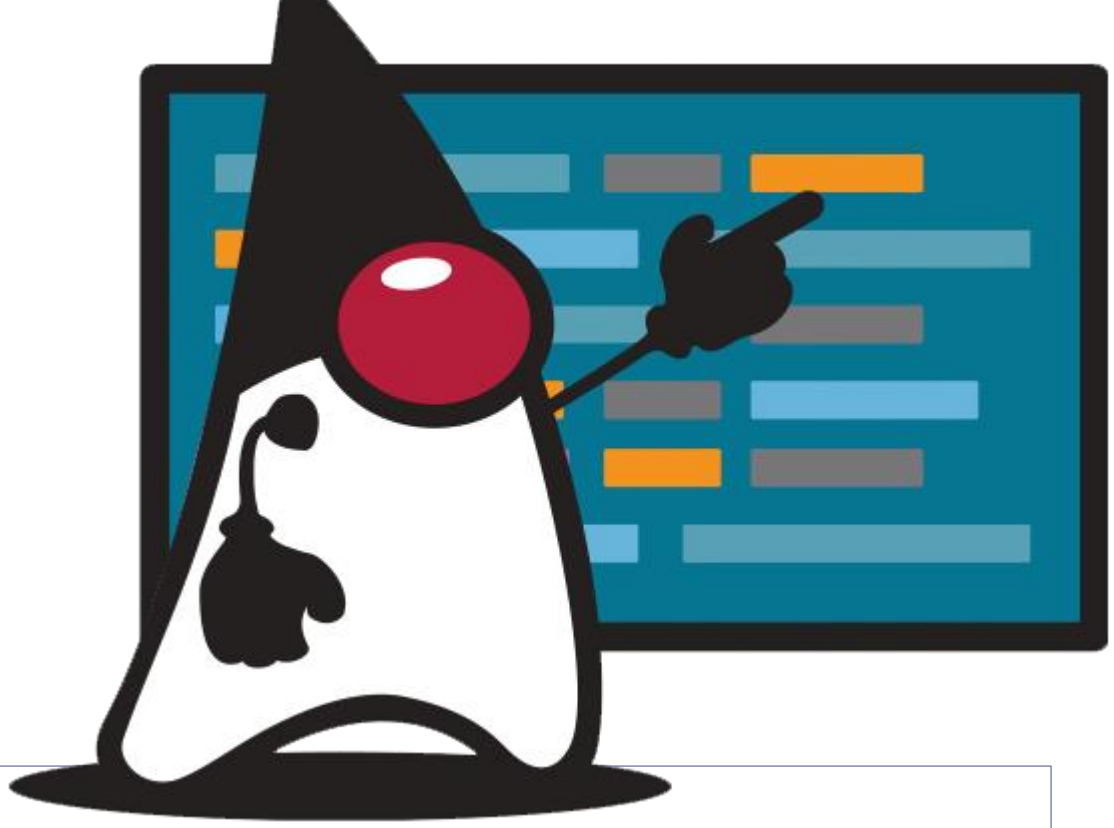
4. Dando formato (VI)

- **Precisión**:
  - Para números reales es la **precisión del valor formateado**.
  - Para cadenas (s) y otros es el la **anchura máxima**, truncando a la derecha si fuese necesario.
- **Ancho: mínima anchura** del valor formateado, rellenando si fuese necesario (con espacios por defecto).
- **Flags:** opciones adicionales de formateo.
  - **+** debe representar el signo.
  - **0** el "cero" será el carácter de relleno.
  - **-** rellenando a la derecha.
  - **,** formatea el número con el separador local de miles.
- **Índice de argumento:** permite especificar con qué argumento está asociado. < indica el igual al anterior.

4. **Dando formato (VII)**

- ***%[indiceDeArg$][flags][ancho][.precision]conversion***

- Para más información (formato de fechas, etc.) consúltese en la API la clase *Formatter*.



5. E/S desde consola

pepinho.com

5. E/S desde línea de comandos

- Puede hacerse a través de ***flujos estándar*** (System.in y System.out) o a través de la clase ***Console***.

A) Flujos estándar

- Java proporciona 3 flujos estándar:
  - Entrada estándar: ***System.in*** (normalmente teclado)
  - Salida estándar: ***System.out*** (normalmente monitor)
  - Error estándar: ***System.err*** (depende del intérprete de comandos usado)
- Son creados automáticamente y no necesitan ser abiertos.
- Por razones históricas **son FLUJOS de tipo BYTE!!**

5. E/S desde línea de comandos (II)

- ***System.out*** y ***System.err*** se definen como objetos de tipo **PrintStream** (de tipo byte stream).

- ***PrintStream*** utiliza internamente un objeto de tipo flujo de carácter para **emular muchos de los comportamientos de los "*character streams*".**

- **System.in** es un **flujo de byte** sin características de flujos de caracteres, por lo que hay que transformarlo en un flujo de carácter:

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

8. E/S desde línea de comandos (III)

B) Console

- Es un tipo definido ***java.io.Console***
- Útil para entrada con *password*, por ejemplo.
- Proporciona verdaderos **flujos** de entrada y salida tipo **carácter** a través de los métodos: **reader**, **writer**

5. E/S desde línea de comandos (IV)

- Antes de debe intentar **obtener un objeto de tipo *Console*** por medio del método:
  - ***System.console()***
  - ***Si la consola no está disponible devuelve null (p.e. desde Netbeans u otro entorno no interactivo)***

- Proporciona lectura segura de claves a través del método ***readPassword***:

  - No produce eco local de los caracteres escritos.
  - Devuelve un ***array* de caracteres**, no un String, por lo que puede sobrescribirse y eliminarse de memoria cuando sea necesario.

5. E/S desde línea de comandos (V)

```
import java.io.*;
import java.util.Arrays;
public class Passwd {
    public static void main (String args[]) throws IOException {
        Console c = System.console();
        if (c == null) {
            System.err.println("No existe consola.");
            System.exit(1);
        }

        String login = c.readLine("Introduce usuario: ");
        char [] vPassword = c.readPassword("Introduce clave antigüa: ");

        if (verifica(login, vPassword)) {
            boolean noCoinciden;
            do {
                char [] nuevoPassword1 =
                    c.readPassword("Nueva clave: ");
                char [] nuevoPassword2 =
                    c.readPassword("Reescribe la clave: ");
                noCoinciden = ! Arrays.equals(nuevoPassword1, nuevoPassword2);
                if (noCoinciden) {
                    c.format("No coinciden, inténtelo de nuevo. %n");
                } else {
                    cambia(login, nuevoPassword1);
                    c.format("El password %s ha sido cambiado.%n", login);
                }
                Arrays.fill(newPassword1, ' '); //limpio de memoria
                Arrays.fill(newPassword2, ' '); //limpio de memoria
            } while (noCoinciden);
        }
        Arrays.fill(vPassword, ' ');
    }
    static boolean verifica(String login, char[] password) {
        // … rellena
        return true;
    }
    static void cambia(String login, char[] password) {}
}
```

6. Stream de Datos: *DataInput y DataOutput*

pepinho.com

Data Streams

- Los flujos de datos admiten **E/S binaria de valores de tipos de datos primitivos: boolean, char, byte, short, int, long, floaty double,** así como **valores de String**.

- Todos los flujos de datos implementan las interfaces ***DataInput*** y ***DataOutput***.

- Existen varias implementaciones de dichas interfaces.

Data Streams: ejemplos. Escritura

	Tipo de dato	Descripción	Método de salida	Método de entrada	Valor de ejemplo
1	double	Precio	*DataOutputStream.**writeDouble***	*DataInputStream.**readDouble***	19.99
2	int	Cantidad	*DataOutputStream.**writeInt***	*DataInputStream.**readInt***	12
3	String	Descripción	*DataOutputStream.**writeUTF***	*DataInputStream.**readUTF***	"Camiseta chula"

```
static final double[] precios = { 19.99, 25.03, 15.99};
static final int[] cantidad = { 9, 7, 13};
static final String[] descripciones = {"Camiseta chula", "Pelota Basket"};

DataOutputStream out = new DataOutputStream(new
            BufferedOutputStream(
                        new FileOutputStream(archivoDatos)));
for (int i = 0; i < precios.length; i ++) {
    out.writeDouble(precios[i]);
    out.writeInt(cantidad[i]);
    out.writeUTF(descripciones[i]);
}
```

- ***writeUTF*** escribe valores ***String*** en un formato modificado de UTF-8. *Esta es una codificación de caracteres de ancho variable que sólo necesita un byte para los caracteres occidentales comunes.*

Data Streams: ejemplos (II). Lectura

```
DataInputStream in = new DataInputStream(new
        BufferedInputStream(
                new FileInputStream(archivo)));
double precio; int cantidad; String descripcion; double total = 0.0;

try {
    while (true) {
        precio = in.readDouble();
        cantidad = in.readInt();
        descripcion = in.readUTF();
        System.out.format("Pedido %d" + " unidades de %s a $%.2f%n",
                cantidad, descripcion, price);
        total += cantidad * precio;
    }
} catch (EOFException e) { }
```

6. E/S desde archivos

pepinho.com

6. E/S desde archivos

- Los **flujos** trabajan con gran variedad de fuentes de datos, incluyendo **archivos**, sin embargo, no proporcionan todas las operaciones comunes a los archivos de disco.

- Existen **clases de E/S para trabajar con ficheros que no son orientas a flujos**. Estas son:

  - ***File***: ayuda a escribir código independiente de plataforma para examinar y manipular archivos y directorios.
  - ***Archivos de acceso aleatorio***: proporciona soporte para acceso no secuencia a archivos de datos. (***RandomAccessFile***)

6. E/S desde archivos: ***File***

- ***File*** permite escribir código para examinar y manipular archivos **independiente de plataforma**.

- OJO!: la clase **File representa nombres de archivo**, no archivos, el fichero podría no existir.

- Ello permite usarlo para convertir el ***nombre de un fichero*** o emplearlo como **parámetro de algunas clases** (como *FileWriter*) para crearlo.

- Si el archivo existe puede **obtener sus atributos** y realizar varias operaciones como **renombrar**, **borrar** o **cambiar los permisos**.

6. E/S desde archivos: ***File*** (II)

- Un objeto de tipo ***File*** contiene la **cadena con el nombre del archivo** usado para construirlo.
- Esa cadena **nunca cambia a los largo del ciclo de vida** del objeto.
- Un programa puede usar el objeto ***File*** para obtener **otras versiones del nombre**, algunas de las cuales pueden no ser iguales al nombre original pasado al constructor.
- Ejemplo:

```
File f = new File("pepe.txt");
```

6. E/S desde archivos: ***File*** (III)

File *f* = ***new File***(".." + *File.separator* + *"pepe"* + *File.separator* + *"proba.txt"*);

Método	MS Windows	UNIX
f.**toString**()	..\pepe\proba.txt	../pepe/proba.txt
f.**getName**()	proba.txt	proba.txt
f.**getParent**()	..\pepe	../pepe
f.**getAbsolutePath**()	c:\users\pepe\..\pepe\proba.txt	/home/pepe/../pepe/proba.txt
f.**getCanonicalPath**()	c:\users\pepe\proba.txt	/home/pepe/pepe.txt

6. E/S desde archivos: ***File*** (IV)

- Ejemplo (edita, compila y prueba):

```
import java.io.*;
import static java.lang.System.out;

public class InfoFile {
    public static void main(String args[]) throws
      IOException {
        out.print("Raices del sistema de ficheros");
        for (File raiz: File.listRoots()) {
            out.format("%s ", raiz);
        }
        out.println();
        for (String nome : args) {
            out.format("%n-----%nnew File(%s)%n", nome);
            File f = new File(nome);
            out.format("toString(): %s%n", f);
            out.format("exists(): %b%n", f.exists());
            out.format("lastModified(): %tc%n",
          f.lastModified());
            out.format("isFile(): %b%n", f.isFile());
            out.format("isDirectory(): %b%n", f.isDirectory());
            out.format("isHidden(): %b%n", f.isHidden());
            out.format("canRead(): %b%n", f.canRead());
            out.format("canWrite(): %b%n", f.canWrite());
            out.format("canExecute(): %b%n", f.canExecute());
            out.format("isAbsolute(): %b%n", f.isAbsolute());
            out.format("length(): %d%n", f.length());
            out.format("getName(): %s%n", f.getName());
            out.format("getPath(): %s%n", f.getPath());
            out.format("getAbsolutePath(): %s%n",
          f.getAbsolutePath());
            out.format("getCanonicalPath(): %s%n",
          f.getCanonicalPath());
            out.format("getParent(): %s%n", f.getParent());
            out.format("toURI: %s%n", f.toURI());
        }
    }
}
```

6. E/S desde archivos: ***File*** (V)

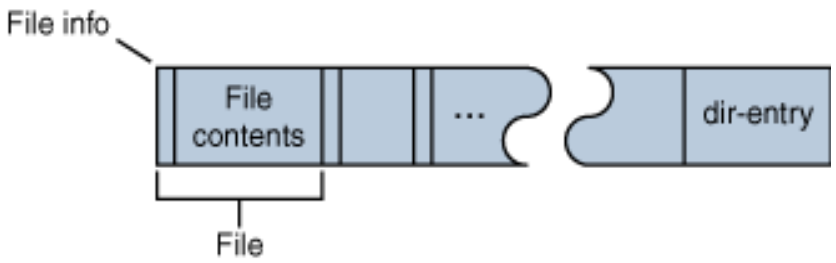
- Dispone de **métodos de manipulación**: ***delete***, ***deleteOnExit***, ***setLastModified***, ***renameTo***.
  - *new File("proba.txt").**setLastModified**(new Date().getTime());*

- Métodos para **trabajar con directorios**: ***mkdir*** (crea directorio), ***mkdirs*** (crea toda la jerarquía de directorios), ***list*** (lista el contenido como un *array* de String) o ***listFiles*** (listan el contenido de un directorio como una *array* de File).

- Métodos **estáticos**: ***createTempFile*** (crea un archivo de nombre único), ***listRoots*** (devuelve la lista de las raíces del sistema de ficheros, c:\ d:\...)

6. E/S desde archivos: *java.io.RandomAccessFile*

- Permite **acceso** no secuencial, o **aleatorio**, al contenido del archivo.



- Permite **leer y escribir** (implementa las interfaces ***DataInput*** y ***DataOutput***) en archivos de acceso aleatorio.
- En el constructor se especifica el modo de apertura, ***lectura o escritura***:
  - new RandomAccessFile("proba.txt", "r");
  - new RandomAccessFile("proba.txt", "rw");

6. E/S desde archivos: ***RandomAccessFile***

- Emplea la notación de **puntero a archivo** para especificar la posición actual en el archivo.
- Al **crearlo apunta al principio del archivo**, la posición 0.
- Las sucesivas llamadas a ***read*** o ***write*** **modifican la posición del puntero** el número de bytes leídos o escritos, respectivamente.
- Dispone de 3 métodos para modificar la posición del puntero:
  - ***int skipBytes(int n)*** — Mueve el puntero hacia delante n bytes.
  - ***void seek(long)*** — Sitúa el puntero justo antes del byte especificado.
  - ***long getFilePointer()*** — Devuelve la posición actual del puntero a archico.