

# Módulo profesional Programación

UD 10 – Entrada y salida de información. Flujos de datos

## Contenido

- Introducción a la lectura y escritura de ficheros
  - Concepto fichero
  - Tipos de ficheros
  - Clase File
- Flujos
  - Concepto de flujo
  - Flujos de caracteres
  - Flujos de bytes
  - Flujos con memorias intermedias
  - Cerrando recursos
  - Adaptadores de flujos
- E/S estándar
- Flujos de objetos

# Introducción a la lectura y escritura de ficheros

¿Cómo definirías lo que es un **fichero** de ordenador?



# Introducción a la lectura y escritura de ficheros

Un **archivo** o **fichero informático** es un conjunto de bits que son almacenados en un dispositivo.

- Los ficheros tienen un **nombre** y se ubican en **directorios o carpetas**.
- A los archivos informáticos se les llama así porque son los equivalentes digitales de los archivos escritos en expedientes, tarjetas, libretas, papel del entorno de oficina tradicional.

# Introducción a la lectura y escritura de ficheros

¿Qué es un fichero de **texto**? ¿y uno **binario**?



# Introducción a la lectura y escritura de ficheros

Tipos de ficheros según el **tipo de contenido**:

- Ficheros de **texto** (o de caracteres): son aquellos creados exclusivamente con caracteres, por lo que pueden ser creados y visualizados utilizando cualquier editor de texto que ofrezca el sistema operativo (por ejemplo: notepad, gedit, etc.)
- Ficheros **binarios** (o de bytes): son aquellos que no contienen caracteres reconocibles sino que los bytes que representan otra información: imágenes, música, vídeo, etc. Estos ficheros solo pueden ser abiertos por programas concretos que entiendan cómo están dispuestos los bytes dentro del fichero, y así poder reconocer la información que contiene.

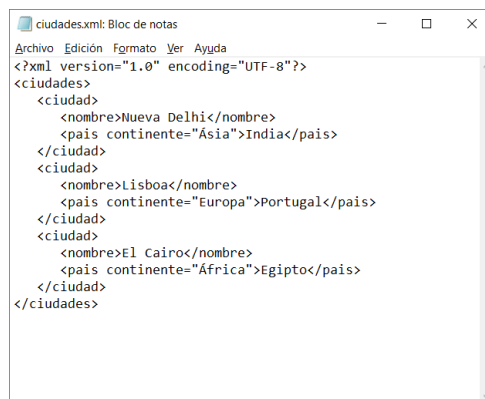
# Introducción a la lectura y escritura de ficheros

¿Qué pasa si abro un **fichero binario** como una imagen jpg en un procesador de textos como el bloc de notas?



# Introducción a la lectura y escritura de ficheros

¿y un **fichero de texto**?



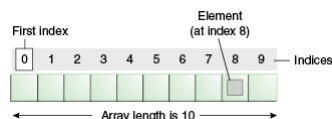
# Introducción a la lectura y escritura de ficheros

¿Qué es un fichero de **acceso secuencial**? ¿y uno **acceso aleatorio**?

# Introducción a la lectura y escritura de ficheros

Tipos de archivos según el **modo de acceso**:

- Ficheros de **acceso secuencial**: en este tipo de ficheros la información es almacenada como una secuencia de bytes (o caracteres), de manera que para acceder al byte (o carácter)  $i$ -ésimo, es necesario pasar antes por todos los anteriores ( $i-1$ ).
- Ficheros de **acceso aleatorio**: a diferencia de los anteriores el acceso puede ser directamente a una posición concreta del fichero, sin necesidad de recorrer los datos anteriores. Un ejemplo de acceso aleatorio en programación es el uso de la estructura de datos arrays.



## Clase File

Java define una clase dentro del **paquete java.io** que representa un archivo o un directorio dentro de un sistema de ficheros. Esta clase es **File**.

Un objeto de la clase File **representa** fichero o de un directorio **que existe en el sistema de ficheros**.

## Clase file

Los métodos de File permiten obtener toda la información sobre las características del fichero o directorio.

Para crear un objeto File se pueden crear cualquiera de los cuatro constructores que posee. Uno de ellos es:

```
File f = new File (String pathname);
```

Ejemplo:

- Linux: new File("/directorio/fichero.txt")
- Windows: new File ("c:\\directorio\\fichero.txt")

## Clase file

Un ejemplo de código para la creación de un objeto File que corresponde a un fichero de texto llamado libros.txt es el siguiente:

```
File f = new File (String pathname);
```

## Clase file

```
File f = new File (String pathname);
```

Si se ejecuta el siguiente código

```
System.out.println("Nombre: " + f.getName());  
System.out.println("Directorio padre: " + f.getParent());  
System.out.println("Ruta relativa: " + f.getPath());  
System.out.println("Ruta absoluta: " + f.getAbsolutePath());
```

Se obtendrá la siguiente salida:

```
Nombre: libros.txt  
Directorio padre: proyectos  
Ruta relativa: proyectos\libros.txt  
Ruta absoluta: C:\dev\prg\ad\proyectos\libros.txt
```

# Actividad

1. Copia el código anterior en una clase **InformacionFichero** y ejecútalo

## File: Métodos más importantes

- |                     |  |
|---------------------|--|
| ▪ getName()         | devuelve el nombre del fichero o directorio  |
| ▪ getPath()         | devuelve el camino relativo  |
| ▪ getAbsolutePath() | devuelve el camino absoluto del fichero/directorio   |
| ▪ canRead()         | devuelve true si el fichero se puede leer  |
| ▪ canWrite()        | devuelve true si el fichero se puede escribir  |
| ▪ length()          | devuelve el tamaño del fichero en bytes  |
| ▪ createNewFile()   | crea un nuevo fichero, vacío, asociado a File si y solo si no existe un fichero con dicho nombre |
| ▪ delete()          | borra el fichero o directorio asociado a File  |
| ▪ exists()          | devuelve true si el fichero/directorio existe  |
| ▪ getParent()       | devuelve el nombre el directorio padre, o null si no existe                                      |



# File: Métodos más importantes

- `isDirectory()` devuelve true si el objeto File corresponde a un directorio
- `isFile()` devuelve true si el objeto File corresponde a un fichero
- `mkdir()` crea un directorio con el nombre indicado en la creación del objeto File
- `renameTo(File nuevoNombre)` renombra el fichero

## Actividad

2. Al programa anterior agrega las líneas de código necesarias para mostrar:
  - Si se puede leer
  - Si se puede escribir
  - Tamaño
  - Si es un directorio
  - Si es un fichero
3. Desarrolla un programa que cree un nuevo directorio con nombre “**nuevudir**” en el directorio actual, a continuación crea dos ficheros cuyos nombres son “**fichero1.txt**” y “**fichero2.txt**” vacíos en dicho directorio y uno de ellos se debe renombrar a “**fichero1nuevo.txt**”. A continuación borra el **fichero2.txt**

```

import java.io.File;
import java.io.IOException;
public class CreaDirectorio {

    public static void main(String[] args){

        File dir = new File("NuevoDirectorio");
        File f1 = new File(dir, "fichero1.txt");
        File f2 = new File(dir, "fichero2.txt");

        dir.mkdir();
        try {
            if (f1.createNewFile())
                System.out.println("Fichero1 creado");
            else
                System.out.println("No se ha podido crear al fichero 1...");

            if (f2.createNewFile())
                System.out.println("Fichero2 creado");
            else
                System.out.println("No se ha podido crear al fichero 2...");

            if (f1.renameTo(new File(dir,"fichero1nuevo.txt")))
                System.out.println("Fichero1 renombrado");
            else
                System.out.println("No se ha podido renombrar el fichero1...");

            if (f2.delete())
                System.out.println("Fichero2 borrado");
            else
                System.out.println("No se ha podido borrar el fichero2...");
        } catch (IOException e){
            System.out.println("Error de E/S");
        }
    }
}

```

## Flujos o streams

El sistema de entrada/salida de Java presenta una gran cantidad de clases que se implementan en el paquete `java.io`.

En Java el acceso a ficheros es tratado como un flujo (stream) entre el programa y el fichero.

# Flujos o streams

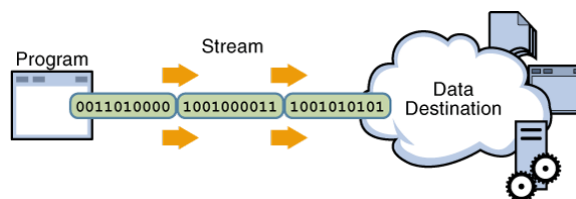
La lectura y escritura de **ficheros** se basa en el concepto de **flujo de datos**.

Un **flujo** de datos es un **canal de comunicación** entre el programa java y el fichero almacenado en el soporte de almacenamiento.

# Flujos o streams

Un **stream** no es más que un objeto que hace de intermediario entre el programa y el origen o el destino de la información.

Es un canal de comunicación que un programa tiene con el mundo exterior.



Este es un ejemplo de un programa usa un flujo de salida para escribir datos en un destino, un elemento a la vez.

## Flujos o streams

Cualquier programa que tenga que **obtener** información de cualquier fuente necesita abrir un **flujo de entrada**, igualmente si necesita **enviar** información abrirá un **flujo de salida** y se escribirá la información en serie.

## Flujos o streams

El concepto de flujo (*stream*) no es exclusivo de los **ficheros**, sino que se trata de una abstracción relacionada con cualquier proceso de transmisión de información entre un contenedor de datos y una zona de la memoria primaria controlada por la aplicación.

# Flujos o streams

Las clases para el manejo de flujos, pueden trabajar con **bytes** o con **caracteres**.

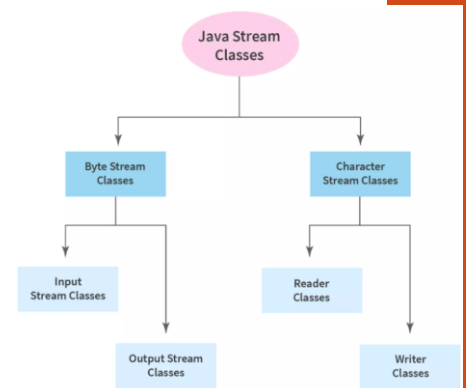
Por tanto, en un **flujo de entrada** se leerán o bytes o caracteres. Igualmente en un **flujo de salida** se escribirán bytes o caracteres

También existen otro tipo de flujos que permiten hacer **conversiones de caracteres a bytes y viceversa**, filtros que mejoran el comportamiento de los flujos...

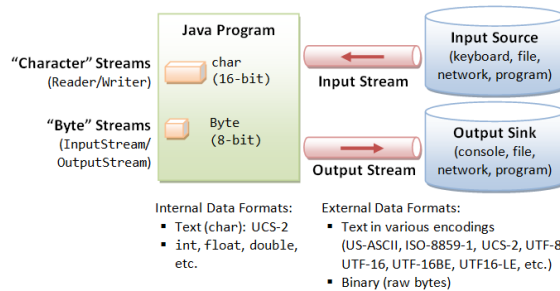
# Flujos o streams

La vinculación de este flujo al dispositivo físico la hace el sistema de entrada/salida de Java. Se definen dos tipos de flujos:

- **Flujos de bytes (8 bits):** realizan operaciones de E/S de bytes y su uso está orientado a la lectura/escritura de datos **binarios**. Todas las clases de flujos de bytes descenden de las clases **InputStream** y **OutputStream**, cada una de estas clases tienen varias subclases que controlan los distintos dispositivos de E/S que se pueden utilizar
- **Flujos de caracteres (16 bits – Unicode-):** realizan operaciones de entrada/salida de **caracteres**. El flujo de caracteres viene gobernado por las clases **Reader** y **Writer**



# Flujos de entrada/salida



Los programas **leen entradas** de fuentes de datos (por ejemplo, teclado, archivo, red, búfer de memoria u otro programa) y **escriben salidas** en los destinos de datos (por ejemplo, consola, archivo, red, búfer de memoria u otro programa).

En la E/S estándar de Java, las entradas y salidas son manejadas por los llamados flujos. **Un flujo es un canal de datos unidireccional secuencial y contiguo** (al igual que el agua o el aceite fluye a través de la tubería).

# Flujos de entrada/salida

Es importante mencionar que Java no distingue entre los distintos tipos de orígenes de datos o destinos (por ejemplo, archivos o redes) en los flujos de E/S. Todos ellos son tratados como un flujo secuencial de datos.

Un **programa Java** recibe datos de una fuente abriendo un **flujo de entrada** y **envía datos a un destino** abriendo un **flujo de salida**. Todos los flujos de E/S de Java son unidireccionales (excepto `RandomAccessFile`, que se explicará más adelante).

# Flujos de entrada/salida

Si tu programa necesita realizar tanto la entrada como la salida, debe abrir dos flujos: un flujo de entrada y un flujo de salida.

Las operaciones de E/S de flujo implican tres pasos:

- **Abrir** un flujo de entrada o salida asociado con un dispositivo físico (por ejemplo, archivo, red, consola / teclado), construyendo una instancia de del flujo que corresponda (entrada o salida)
- **Leer** desde el flujo de entrada abierto hasta que se encuentre el "final del flujo", o **escribir** en el flujo de salida abierto (y, opcionalmente, vacíe la salida almacenada en búfer).
- **Cerrar** el flujo de entrada o salida.

# Tipos de flujos de datos

El paquete java.io es el que contiene las clases para el manejo de flujos

Tipo de Flujo	Descripción
FileReader	Lee caracteres de un fichero
FileWriter	Escribe caracteres en un fichero
FileInputStream	Lee Bytes en un fichero
FileOutputStream	Escribe Bytes en un fichero
InputStreamReader	Lee Bytes y los convierte en caracteres
OutputStreamWriter	Recibe caracteres y los escribe como Bytes
BufferedReader	Añade un buffer permitiendo leer líneas completas
BufferedWriter	Añade un buffer permitiendo escribir líneas completas

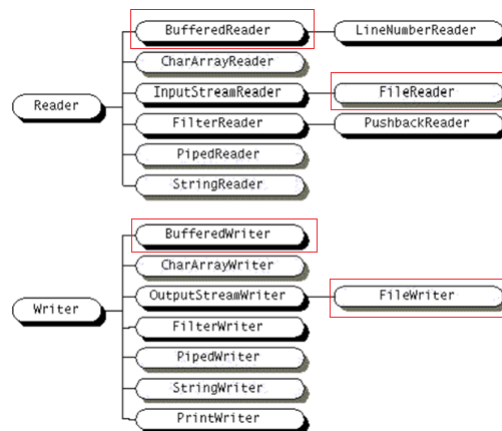
# Flujos de caracteres

Las clases **Reader** y **Writer** manejan flujos de caracteres Unicode.

Nota: **Unicode** es un estándar de codificación de caracteres diseñado para facilitar el tratamiento informático, transmisión y visualización de textos

# Flujos de caracteres

Jerarquía de clases para la lectura y escritura de flujo de caracteres





# Flujos de caracteres

Las clases de **flujos de caracteres** más importantes son:

- Para acceso a ficheros, lectura y escritura de caracteres en ficheros: **FileReader** y **FileWriter**.
- Para la bufferización de datos: **BufferedReader** y **BufferedWriter**.  
Permite la lectura/escritura de bloques de datos

buffer = espacio de memoria en el que se almacenan datos de manera temporal

## Actividad

```
import java.io.FileWriter;
import java.io.IOException;
public class EscribeFicheroTexto {

    public static void main(String[] args) {

        //Crea el String con la cadena XML
        String texto = "<Libros><Libro><Titulo>El Capote</Titulo></Libro></Libros>";

        //Guarda en la variable nombre el nombre del archivo que se creará.
        String nombre = "libros.xml";

        FileWriter os = null;

        try{
            //Se crea un nuevo objeto FileWriter
            os = new FileWriter(nombre);

            for (int i=0;i<texto.length();i++)
                os.write(texto.charAt(i));    //Se escribe en el flujo

        } catch (IOException ex){
            System.out.println("Error al acceder al fichero");
        } finally{
            try {
                os.close(); //Se cierra el flujo
            } catch (IOException ex){
                System.out.println("Error al acceder al cerrar el fichero");
            }
        }
    }
}
```

Si el archivo no existe, se creará. Si el fichero existe se sobrescribirá, a menos que le indiquemos que se desea añadir al final con el parámetro **true**.  
`FileWriter(nombre,true)`

# Actividad

4. Escribe un programa `CopyCharacters` que copie un archivo xml en otro.

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

public class CopyCharacters2 {
    public static void main(String[] args) {

        FileReader is = null;
        FileWriter os = null;

        try {
            is = new FileReader("biblioteca.xml");
            os = new FileWriter("biblioteca_copy.xml");

            int c;
            while ((c = is.read()) != -1)
                os.write(c);

        } catch (IOException e){
            System.out.println("Error de acceso al archivo");
        } finally {
            try{if (is!= null) is.close();
            } catch (IOException ex1){System.out.println("Error al cerrar el primer archivo");};
            try{if (os!= null) os.close();
            } catch (IOException ex1){System.out.println("Error al cerrar el segundo archivo");};
        }
    }
}
```

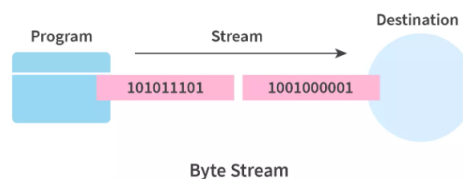
## Cerrando flujos

Hay que tener cuidado de cerrar siempre flujos al terminar su uso, a fin de no desperdiciar recursos del sistema.

Por eso es importante poner el cierre en la sentencia *finally*. De este modo, aseguramos que a pesar de que durante el acceso se produjera una excepción, antes de abandonar el método en la búsqueda de alguna sentencia *catch* que capture el error, se ejecutará obligatoriamente el cierre.

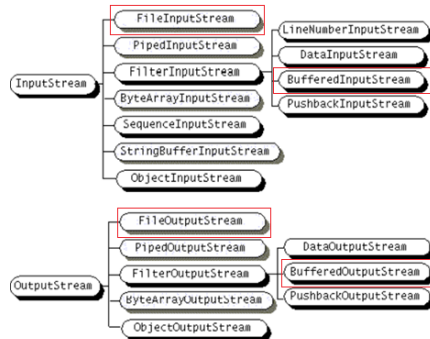
## Flujos de bytes

- La clase **InputStream** representa clases que producen entradas de distintas fuentes. Estas fuentes pueden ser: array de bytes, un objeto String, un fichero, una secuencia de otros flujos, otras fuentes como una conexión a internet.
- La clase **OutputStream** representa las clases que deciden donde irá la salida: un array de bytes, un fichero o una tubería.



# Flujos de bytes

Jerarquía de clases para la lectura y escritura de flujo de bytes



Dentro de los flujos de bytes están las clases **FileInputStream** y **FileOutputStream** que manipulan flujos de bytes provenientes o dirigidos hacia ficheros en disco. (secuencial)

# Flujos de bytes

```

import java.io.FileInputStream;
import java.io.IOException;

public class FicherosBinariosApp {

    public static void main(String[] args) {

        FileInputStream fis = null;
        try {

            fis = new FileInputStream("C:\\dev\\prg\\ad\\proyectos\\donald.jpg");
            int valor = -1;
            while((valor = fis.read()) != -1) // usando el método read(), cuando llega al final del fichero devuelve -1
                System.out.print((char)valor);

        } catch (IOException e) {
            System.out.println("error al acceder al fichero");
        } finally {
            try {
                if (fis != null) fis.close();
            } catch (IOException e) {
                System.out.println("error al cerrar el fichero");
            }
        }
    }
}
  
```

# Actividad

5. Escribe un programa **CopyBytes** que copie un archivo jpg en otro.

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

public class CopyBytes {

    public static void main(String[] args) {

        FileInputStream fis = null;
        FileOutputStream fos = null;

        try {

            fis = new FileInputStream("C:\\dev\\prg\\ad\\proyectos\\donald.jpg");
            fos = new FileOutputStream("C:\\dev\\prg\\ad\\proyectos\\donald-copia.jpg");

            int bytesRead = -1;

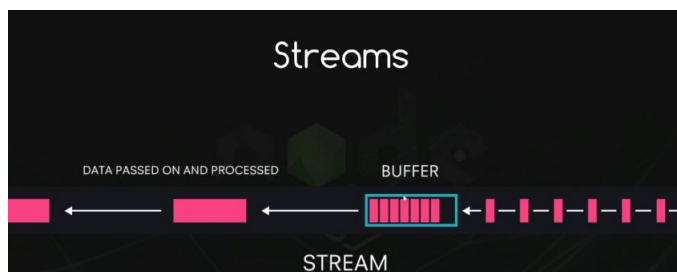
            while((bytesRead=fis.read())!=-1)
                fos.write(bytesRead);

        } catch (IOException e) {
            System.out.println("error al acceder al fichero");
        } finally{
            try{if (fis!= null) fis.close();
            } catch (IOException ex1){System.out.println("Error al cerrar el primer archivo");}
            try{if (fos!= null) fos.close();
            } catch (IOException ex1){System.out.println("Error al cerrar el segundo archivo");}
        }
    }
}
```

# Flujos con memorias intermedias (buffered streams)

Si tenemos un fichero pequeño el código del último ejemplo estaría bien. Pero ahora imaginemos un archivo muy grande. En este caso el sistema utilizado anteriormente se vuelve poco eficiente.

Una solución sería utilizar un **búfer**. Un búfer es como una memoria interna



## Buffered streams –lectura-

`FileReader` no contiene métodos que nos permitan leer líneas completas, pero `BufferedReader` sí; dispone del método `readLine()` que lee una línea del fichero y la devuelve; o devuelve null si no hay nada que leer o llegamos al final del fichero. También dispone de un método `read()` para leer un carácter.

Para construir un `BufferedReader` necesitamos un `FileReader`:

```
BufferedReader br = new BufferedReader(new FileReader(nombrefichero));
```

Wrapper class = Clase envolvente

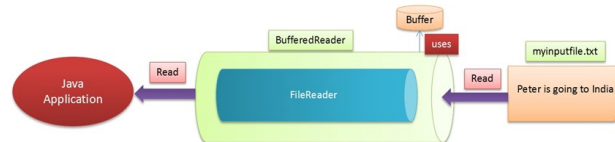


## Buffered streams –lectura-

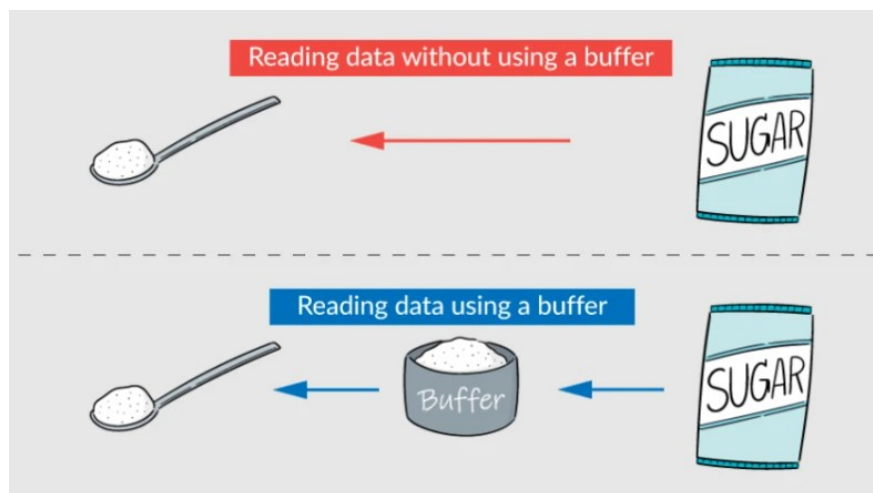
La clase `BufferedReader` provee memoria intermedia a las instancias de `Reader`. Los buffers aceleran la E/S. En lugar de leer un carácter por vez de una red o disco, la clase `BufferedReader` lee bloques más grandes. Esto es típicamente más rápido, especialmente para accesos a disco y grandes cantidades de datos.

La clase `BufferedReader` es similar a la clase `BufferedInputStream` pero no son iguales. La principal diferencia es que `BufferedReader` lee texto y `BufferedInputStream` lee bytes.

Hay que tener en cuenta que el tamaño del búfer puede especificarse o se puede usar el tamaño por defecto



## Buffered streams –lectura-



```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
public class LeeFicheroTextoBuff {

    public static void main(String[] args) {

        BufferedReader br = null;

        try {

            br = new BufferedReader(new FileReader("HolaMundo.java"));

            String linea;
            while ((linea=br.readLine()) != null)
                System.out.println(linea);

        } catch (IOException e) {
            System.out.println("Error de E/S");
        } finally{
            try {
                if (br != null) br.close();
            } catch (IOException e){
                System.out.println("Error al acerrar el ficherp");
            }
        }
    }
}
```

## Clase envoltente (Wrapper class)

Esta solución elegida se conoce técnicamente también con el nombre “decorador”

Representación gráfica que simboliza el concepto "decorador"



Consiste en **envolver dos o más objetos**, uno dentro del otro, como si se trataran de muñecas rusas.

Cada envoltorio aporta una cierta funcionalidad extra ( "decora" el objeto interno de forma diferente al original). De esta manera es posible crear instancias "a gusto del consumidor", seleccionando sólo la funcionalidad que sea necesaria.



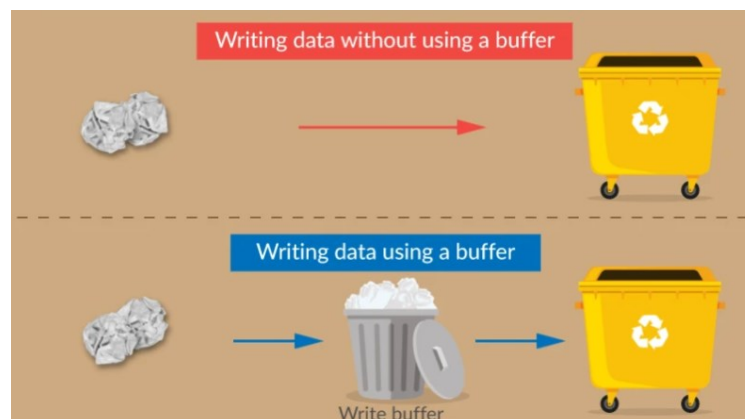
## Buffered streams –escritura-

La clase `BufferedWriter` también deriva de la clase `Writer`. Esta clase **añade un buffer** para realizar una escritura eficiente de caracteres.

Para construir un `BufferedWriter` necesitamos la clase `FileWriter` (o cualquier otra subclase de `Writer`)

```
BufferedWriter bw = new BufferedWriter(new FileWriter(nombrefichero));
```

## Buffered streams –escritura-



## Buffered streams –escritura-

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.IOException;
public class EscribeFicheroTextoBuff2 {

    public static void main(String[] args) {

        BufferedWriter bw = null;

        try {
            bw = new BufferedWriter(new FileWriter("fichero_texto.txt"));

            for (int i=1; i<11;i++)
                bw.write("Fila número: " + i + "\n");
        } catch (IOException e) {
            System.out.println("Error de E/S");
        } finally{
            try{if (bw!= null) bw.close();
            } catch (IOException e){System.out.println("Error al cerrar el primer archivo");};
        }
    }
}
```

## Lectura con Scanner

En Java, otro mecanismo para leer texto es utilizar la clase Scanner. Ya sabes cómo utilizar un Scanner para leer la entrada de la consola.

Para leer la entrada de un archivo de disco, la clase Scanner se apoya en otra clase, File, que describe archivos y directorios.

El siguiente ejemplo lee de un archivo con la clase Scanner y escribe en otro con la clase PrintWriter

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.io.PrintWriter;
4 import java.util.Scanner;
5
6 /**
7  * This program reads a file with numbers, and writes the numbers to another
8  * file, lined up in a column and followed by their total.
9  */
10 public class Total
11 {
12     public static void main(String[] args) throws FileNotFoundException
13     {
14         // Prompt for the input and output file names
15
16         Scanner console = new Scanner(System.in);
17         System.out.print("Input file: ");
18         String inputFile = console.next();
19         System.out.print("Output file: ");
20         String outputFile = console.next();
21
22         // Construct the Scanner and PrintWriter objects for reading and writing
23
24         File inputFile = new File(inputFile);
25         Scanner in = new Scanner(inputFile);
26         PrintWriter out = new PrintWriter(outputFile);
27
28         // Read the input and write the output
29
30         double total = 0;
31
32         while (in.hasNextDouble())
33         {
34             double value = in.nextDouble();
35             out.printf("%15.2f\n", value);
36             total = total + value;
37         }
38
39         out.printf("Total: %8.2f\n", total);
40
41         in.close();
42         out.close();
43     }
44 }
```

## Actividad

6. Crea un programa que se llame **EscribeFicheroTexto** que escriba un archivo **agenda.txt** pidiendo nombre y teléfono por pantalla hasta que el usuario no quiera ingresar más datos. Cada línea en el archivo debe ser

nombre, telefono

Utiliza flujos con memorias intermedias (buffered streams)

7. Crea un programa que se llame **LeeFicheroTexto** que lea el archivo generado en la actividad anterior y muestre los datos por pantalla. Utiliza flujos con memorias intermedias (buffered streams). Los datos por pantalla se imprimirán como muestra el siguiente ejemplo:

Nombre: Javier Teléfono: 646115712

Nombre: Mariví Teléfono: 912704803

## Buffered Streams –bytes-

Los flujos con memorias intermedias para **flujos de bytes**: **BufferedInputStream** y **BufferedOutputStream** tienen un tratamiento similar a los flujos de caracteres con memorias intermedias.

No hay ejemplos de código FALTA

# Cerrando flujos

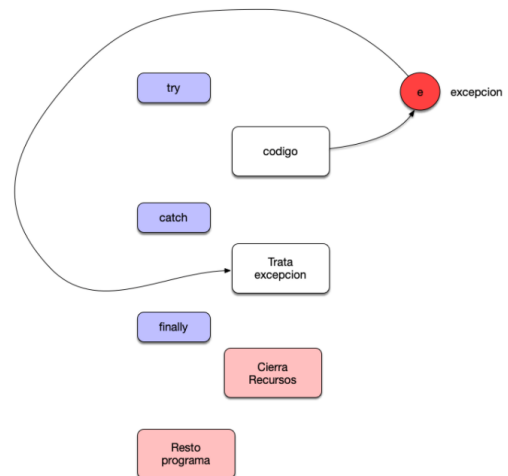
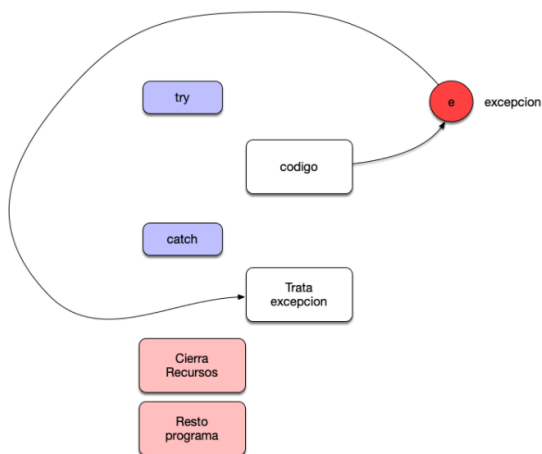
Un ejemplo de lectura de un archivo de texto con un flujo con memoria intermedia (BufferedReader) se puede ver en la siguiente figura:

¿Qué **error** tiene el código?

```
import java.io.*;
public class LeeFicheroTextoBuff {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new FileReader("HolaMundo.java"));
            String linea;
            while ((linea=br.readLine()) != null)
                System.out.println(linea);

            br.close();
        } catch (FileNotFoundException e) {
            System.out.println("No se encuentra el fichero");
        }
    }
}
```

# Cerrando flujos



## Cerrando flujos

Veamos este código que lee de un flujo de bytes de entrada y escribe en un flujo de bytes de salida y cierra ambos flujos en la cláusula finally

¿Qué **error** tiene el código?

```

FileInputStream fis = null;
FileOutputStream fos = null;
try {
    fis = new FileInputStream("../input/fxrates.txt");
    fos = new FileOutputStream("../output/fxrates.txt");
    // código que lee del flujo de entrada y escribe en el flujo de salida
} catch (...) {...}
} finally {
    try { // El programador tuvo cuidado en cerrar los flujos en la cláusula
        // finally, pero no está del todo resuelto el problema
        // ¿Puedes encontrar el error?
        if(fis != null) fis.close();
        if(fos != null) fos.close();
    } catch(IOException e) {
        System.out.println("Error en el cierre de flujos");
    }
}

```

## Cerrando flujos

La forma correcta de cerrar los flujos es cerrándolos en su propio bloque try catch, de manera que un problema en cerrar un flujo no impida llamar a close() en el otro flujo. Esta es la forma correcta de cerrar ambos flujos en Java:

```

InputStream is = null;
OutputStream os = null;
try {
    is = new FileInputStream("../input/fxrates.txt");
    os = new FileOutputStream("../output/fxrates.txt");
    // código que lee del flujo de entrada y escribe en el flujo de salida
} catch (...) {...}
} finally {
    try { if (is != null) is.close(); } catch(IOException e) { //closing quietly}
    try { if (os != null) os.close(); } catch(IOException e) { //closing quietly}
}

```

## Cerrando flujos: try-with-resources

Por cierto, tienes una alternativa **mucho mejor** si utilizas Java 7.

A partir de esta versión de proporcionan declaraciones **try-with-resources** para la gestión automática de recursos en Java.

## Cerrando flujos: try-with-resources

**Todos** los recursos abiertos en el **bloque try** serán **cerrados automáticamente** por Java, siempre que implementen **Closable** y **AutoClosable**.

Dado que todos los InputStream y OutputStream son elegibles para ser utilizados dentro de las sentencias try-with-resources, debes tomar ventaja de esto.

Esto es muy bueno para **los programadores de Java**, ya que **no son tan cuidadosos como sus homólogos de C++**, especialmente cuando liberan recursos.

## Cerrando flujos: try-with-resources

Este es el aspecto del código anterior con la sentencia try-with-resources

```
try (FileInputStream fis = new FileInputStream("../input/fxrates.txt");
    FileOutputStream fos = new FileOutputStream("../output/fxrates.tx")) {

    // código de lectura/escritura .....

} catch (IOException ioex) {
    System.out.println("Error en la copia de archivos : " + ioex.getMessage());
    ioex.printStackTrace();
}
```

Veremos un ejemplo de su uso en la conexión a bases de datos relacionales que sigue en este apartado.

## Adaptadores de bytes a caracteres

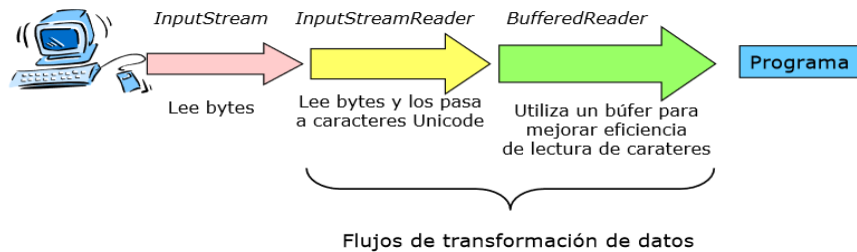
CLASES DE FLUJOS DE BYTES	CLASE CORRESPONDIENTE DE FLUJO DE CARÁCTERES
InputStream	Reader, convertidor InputStreamReader
OutputStream	Writer, convertidor OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
(sin clase correspondiente)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

En programación orientada a objetos, las clases que consiguen hacer compatibles dos jerarquías independientes llaman **adaptadores**.

Las clases **InputStreamReader** y **OutputStreamWriter** (de la jerarquía de Reader y Writer) son de este tipo y consiguen adaptar cualquier flujo orientado a bytes en un flujo orientado a caracteres.

- **InputStreamReader** convierte un **InputStream** en un **Reader**
- **OutputStreamWriter** convierte un **OutputStream** en un **Writer**

# Adaptadores de bytes a caracteres



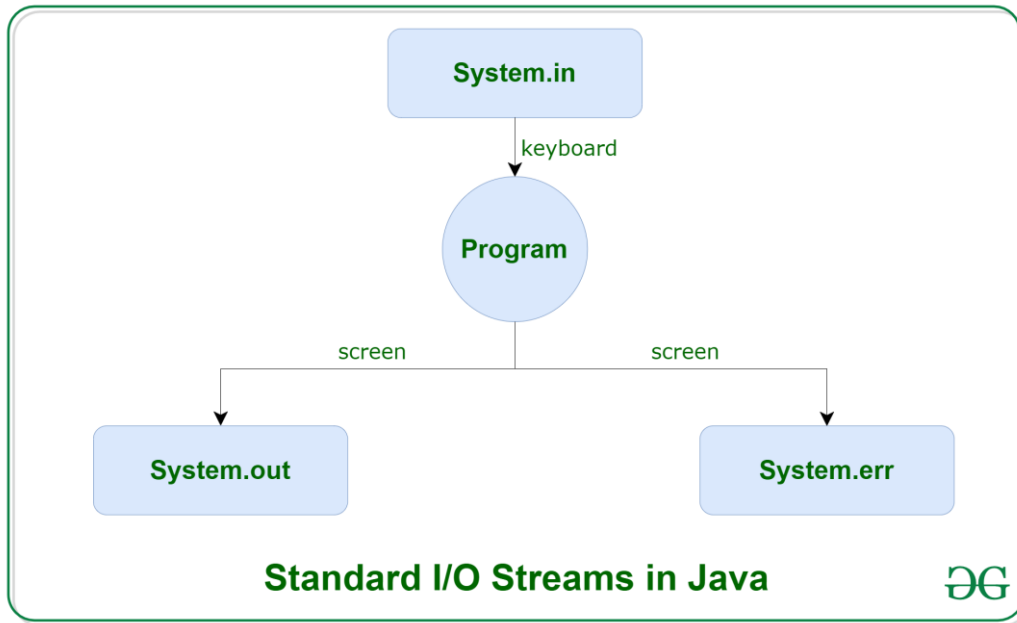
## Entrada y salida estándar

Java tiene 3 flujos llamados **System.in**, **System.out**, y **System.err** que se utilizan comúnmente para proporcionar la entrada y la salida de las aplicaciones Java. El más utilizado es probablemente **System.out** para escribir la salida a la consola.

**System.in**, **System.out** y **System.err** son inicializados por el tiempo de ejecución de Java cuando una VM Java se inicia, por lo que no tienes que instanciar ningún flujo tú mismo (aunque puedes intercambiarlos en tiempo de ejecución).

Tipo	Descripción	Valor por Defecto
System.in	Representa la entrada estándar	Teclado
System.out	Representa la salida estándar	Pantalla
System.err	Representa la salida de errores	Pantalla



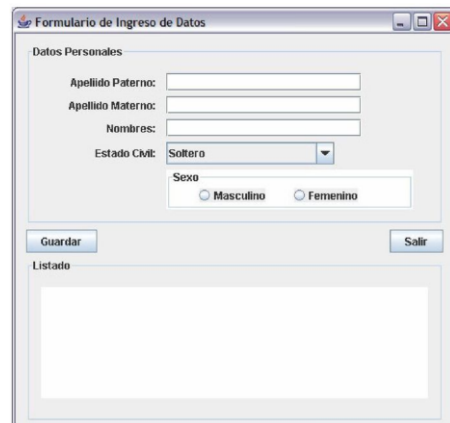
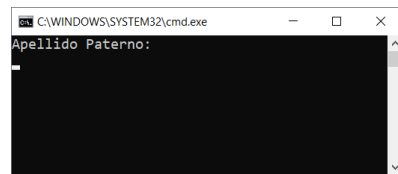


## System.in

**System.in** es un **InputStream** que suele estar conectado a la entrada del teclado de los programas de consola.

System.in no se utiliza tan a menudo ya que los datos se pasan comúnmente a una aplicación Java a través de **argumentos de línea de comandos**, **archivos**, o posiblemente a través de **conexiones de red** si la aplicación está diseñada para ello.

En aplicaciones con GUI la entrada a la aplicación se da a través de la GUI. Este es un mecanismo de entrada separado de System.in.

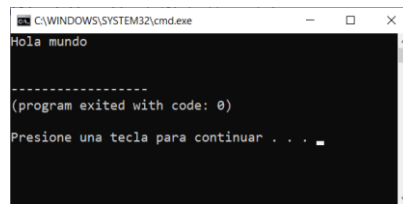


# System.out

**System.out es un `PrintStream`** en el que se pueden escribir caracteres. System.out normalmente da salida a los datos que se escriben en él a la consola.

System.out se utiliza a menudo desde los programas de consola como una forma de mostrar el resultado de su ejecución al usuario.

También se utiliza a menudo para imprimir declaraciones de depuración de un programa (aunque podría decirse que no es la mejor manera de obtener información de depuración de un programa).



# System.err

**System.err es un `PrintStream`.** System.err funciona como System.out, excepto que normalmente sólo se utiliza para dar salida a textos de error.

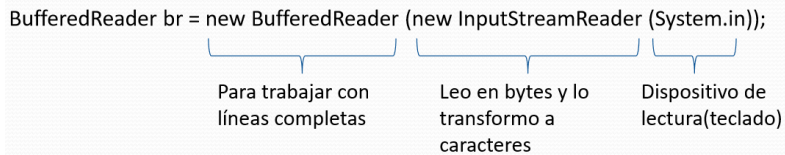
Algunos programas (como Eclipse) mostrarán la salida a System.err en **texto rojo**, para hacer más obvio que es un texto de error.

```
try {
    InputStream input = new FileInputStream("c:\\data\\...");
    System.out.println("File opened...");
} catch (IOException e) {
    System.err.println("File opening failed:");
    e.printStackTrace();
}
```

# Lectura de teclado

Para leer de teclado lo primero que debemos hacer es abrir el flujo de datos de entrada.

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



Para trabajar con líneas completas      Leo en bytes y lo transformo a caracteres      Dispositivo de lectura(teclado)

Una vez abierto el flujo podemos comenzar a leer datos. Para esto tenemos la función `read` para leer un carácter: `int read()`. Como hemos utilizado un `BufferedReader`, también podemos leer una línea completa: `String readLine()`.

# Lectura de teclado

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
public class LecturaTeclado {

    public static void main(String[] args) {

        try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in));) {
            String linea = br.readLine();
            System.out.println("La línea leída es: " + linea);
        } catch (IOException e){
            System.out.println("Error en la lectura");
        }
    }
}
```

# Lectura de teclado

Al leer desde teclado con `readline()` lo que se obtiene es una cadena de texto (`String`). Si queremos convertirlo a otro tipo de dato podemos usar las siguientes conversiones explícitas:

- `float Float.parseFloat(String)`
- `int Integer.parseInt(String)`
- `double Double.parseDouble(String)`
- `short Short.parseShort(String)`

# Lectura de teclado

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
public class LecturaTeclado2 {

    public static void main(String[] args) {

        try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in));) {
            String linea = br.readLine();
            int num = Integer.parseInt(linea);
            num *= 2;
            System.out.println("El doble del número introducido es: " + num);
        } catch (IOException e){
            System.out.println("Error en la lectura");
        }
    }
}
```

# Lectura de teclado

## Scanner vs BufferedReader

### BufferedReader:

```
InputStreamReader isr=new InputStreamReader(System.in);  
BufferedReader br= new BufferedReader(isr);  
String st= br.readLine();
```

### Scanner:

```
Scanner sc= new Scanner(System.in);  
String st= sc.nextLine();
```

- Scanner tiene un **tamaño de buffer** más pequeño (1024 chars) que BufferedReader (8192 chars), pero es más que suficiente.
- BufferedReader está **sincronizado (es thread-safe)** mientras que Scanner no lo está. BufferedReader debe ser utilizado si estamos trabajando con múltiples hilos de ejecución.
- BufferedReader es un poco **más rápido** comparado con Scanner que parsea los datos de entrada mientras que BufferedReader simplemente lee secuencias de caracteres.

## Persistencia: objetos serializables

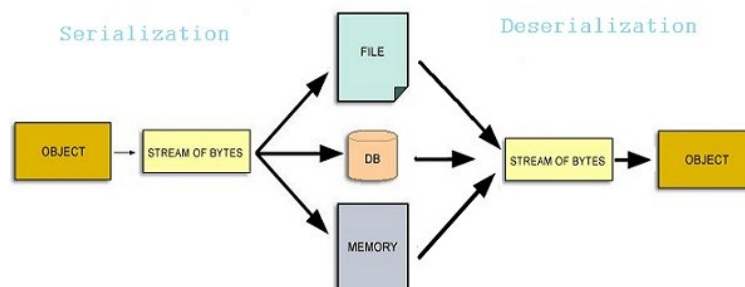
Si tenemos un objeto de tipo **Empleado** con varios atributos (nombre, dirección, salario, departamento, etc.) y queremos guardarlo en un fichero, tendríamos que guardar cada atributo que forma parte del objeto por separado, esto se vuelve engorroso si tenemos gran cantidad de objetos.

## Persistencia: objetos serializables

Por ello, Java nos permite guardar objetos en ficheros binarios; para poder hacerlo, el objeto tiene que implementar la interfaz **Serializable** que dispone una serie de métodos con los que podemos **guardar y leer objetos en ficheros binarios**.

## Persistencia: objetos serializables

La **serialización** es un mecanismo ampliamente usado para transportar objetos a través de una red, para hacer persistente un objeto en un archivo o base de datos, o para distribuir objetos idénticos a varias aplicaciones.



## Persistencia: objetos serializables

En java la capacidad de serialización, es decir, la de guardar información sobre un objeto para luego recuperarla, se llama **persistencia**.

## Persistencia: objetos serializables

La palabra **persistencia** viene del latin “**persistere**” que significa **durar por largo tiempo**.

En el mundo de los objetos todo se maneja en memoria volátil, esto quiere decir que los objetos creados en un programa siempre son temporales, porque son almacenados en la memoria RAM.

Lo que busca la persistencia es guardar un objeto permanentemente en un recurso de almacenamiento, base de datos, archivo, etc. para recuperarlo mas adelante.

# Persistencia: objetos serializables

La [serialización](#) de objetos en Java permite tomar cualquier objeto que implemente la interfaz `Serializable` y [convertirlo en una secuencia de bits](#), que puede ser posteriormente [restaurada](#) para [regenerar el objeto original](#).

Para leer y escribir objetos serializables a un stream se utilizan las clases `ObjectInputStream` y `ObjectOutputStream` respectivamente.

# Persistencia: objetos serializables

A continuación se muestra la clase `Persona` que implementa la interfaz `Serializable` y, que utilizaremos para escribir y leer objetos en un fichero binario.

```
import java.io.Serializable;
public class Persona implements Serializable {

    private String nombre;
    private int edad;

    public Persona(String nombre, int edad){
        this.nombre = nombre;
        this.edad = edad;
    }

    public Persona(){
        this.nombre=null;
    }

    public void setNombre (String nombre) {this.nombre=nombre;}
    public void setEdad (int edad) {this.edad=edad;}

    public String getNombre(){return this.nombre;}
    public int getEdad(){return this.edad;}
}
```



# Persistencia: objetos serializables

El siguiente ejemplo escribe objetos Persona en un fichero. Necesitamos crear un flujo de salida a disco con `FileOutputStream` y luego se crea el flujo de salida `ObjectOutputStream` que es el que procesa objetos.

```
File fichero = new File ("fichero_personas.dat");
FileOutputStream fos = new FileOutputStream (fichero);
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

# Persistencia: objetos serializables

El método `writeObject()` escribe los objetos al flujo de salida y los guarda en un fichero en disco:

```
oos.writeObject(Persona);
```

Cambiar para usar declaración **try-with-resources**

```
import java.io.IOException;
import java.io.File;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;

public class Marshall {

    public static void main(String[] args) {

        Persona persona;

        File fichero = new File ("fichero_personas.dat");
        ObjectOutputStream oos=null;

        try {
            FileOutputStream fos = new FileOutputStream (fichero);
            oos = new ObjectOutputStream(fos);

            String nombres[]={"Ana", "Sergio", "Maria", "Pepe", "Sandra",
                              "Manuel de Puerto Real", "Sofia", "Miguel",
                              "Paula", "Pablo", "Teresa", "Juan"};

            int edades[] = {14,15,13,15,16,12,8,5,12,15,16,17};

            for (int i=0; i<edades.length;i++){
                persona = new Persona(nombres[i],edades[i]);
                oos.writeObject(persona);
            }
        } catch (IOException ex){
            System.err.println("Error a abrir o procesar el archivo");
        } finally {
            try { if (oos != null) oos.close(); } catch (IOException e){System.err.println("Error al cerrar el flujo");};
        }
    }
}
```

# Persistencia: objetos serializables

Para leer los objetos Persona del fichero necesitamos el flujo de entrada de disco `FileInputStream` y luego crear el flujo de entrada `ObjectInputStream` que es el que procesa objetos.

```
File fichero = new File ("fichero_personas.dat");
FileInputStream fis = new FileInputStream (fichero);
ObjectInputStream ois = new ObjectInputStream(fis);
```

# Persistencia: objetos serializables

El método `readObject()` lee los objetos del flujo de entrada

```
Persona = (Persona)
ois.readObject();
```

Puede lanzar  
`ClassNotFoundException`

Cambiar para usar  
declaración `try-with-resources`

```
import java.io.IOException;
import java.io.File;
import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class Unmarshall {

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

        Persona persona;

        File fichero = new File ("fichero_personas.dat");
        FileInputStream fis = new FileInputStream (fichero);
        ObjectInputStream ois = new ObjectInputStream(fis);

        try {
            while (true){
                persona = (Persona) ois.readObject();
                System.out.println("Nombre: " + persona.getNombre()
                    + ", edad: " + persona.getEdad());
            }
        } catch (IOException ex){
            System.err.println("Error a abrir o procesar el archivo");
        } finally {
            try { if (ois != null) ois.close(); } catch (IOException e){System.err.println("Error al cerrar el flujo");}
        }
    }
}
```

# Recursos

- [Curso youtube : Java desde 0](#)
- [Libro Java 9. Manual imprescindible](#). F. Javier Moldes Teo.  
Editorial Anaya
- [App SoloLearn: Aprende a Programar. Curso Java](#)