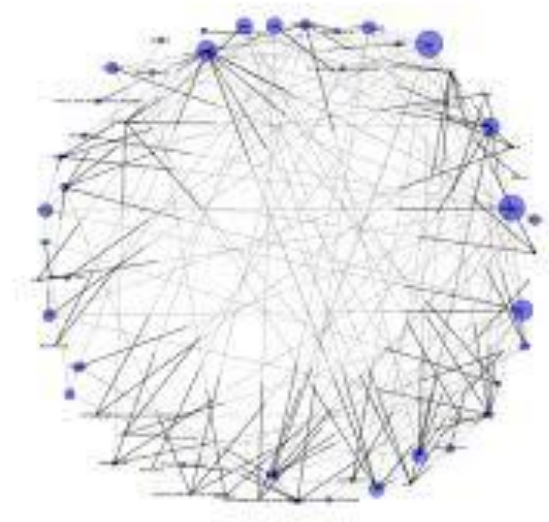


DAM

ACCESO A DATOS



UD1

MANEJO DE FICHEROS EN JAVA

2. FLUJO DE DATOS

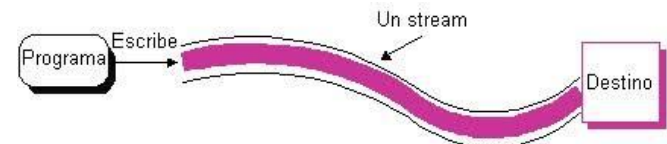


Introducción

- Hasta ahora todos los datos que se creaban durante la ejecución de los programas dejaban de existir al finalizar la misma.
- Si queremos que los datos tengan un vida más allá que la de los programas que los general, deberemos hacer que los datos sean **persistentes**, objetivo que se logrará haciendo uso de ficheros o bases de datos.
- *En este UD veremos el uso básico de los archivos en Java para conseguir persistencia de datos.*
- Las clases que usaremos para el tratamiento de ficheros están almacenadas en el paquete **java.io**, por lo que deberán ser importadas.
- Cuando trabajamos con ficheros tendremos que tener en cuenta que se pueden producir problemas debido a distintas causas: el archivo puede estar corrupto, el soporte donde está el fichero no es accesible, o intentamos guardar información donde no hay espacio libre suficiente, etc. Las excepciones será la herramienta que nos permita controlar en el programa estos errores, todas ellas derivadas de **IOException**.

Streams o flujos de datos

- Un **input file** es un fichero que es leído por un programa.
- Un **output file** es un fichero que es escrito por un programa.
- Las operaciones de entrada y salida a menudo se denominan de **E/S** o **I/O** (Input/Output)
- Un **flujo de datos** es una secuencia ordenada de datos que tiene una fuente (flujos de entrada) o un destino (flujos de salida).
- Las clases que manejan los flujos aíslan a los programadores de los detalles específicos del sistema de funcionamiento del dispositivo y del sistema de entrada y salida.
- Un **stream** es un **objeto** que representa el **flujo de datos** entre nuestra aplicación y una fuente de datos externa, y viceversa.
- Todo programa que necesite tener acceso a información de e/s necesita abrir un **stream**, así podemos tener dos tipos de flujos:
 - Flujo de datos de entrada: **input stream**.
 - Flujo de datos de salida: **output stream**.
- Los flujos o *streams* actúan como interfaz con el dispositivo o clase asociada haciendo la e/s independiente del tipo de datos y del dispositivo.



Tipos de flujos

- **Flujos de bytes (8 bits):**

- Se utiliza para leer ficheros binarios.
- El flujo de datos se realiza byte a byte.
- Las clases que trabajan con estos flujos descenden de las clases abstractas **InputStream** y **OutputStream**.
- Las clases más comunes son **FileInputStream** y **FileOutputStream**.

```
1  ||java||Murach's Beginning Java@H|||jps||Murach's Java Servlets ar
```

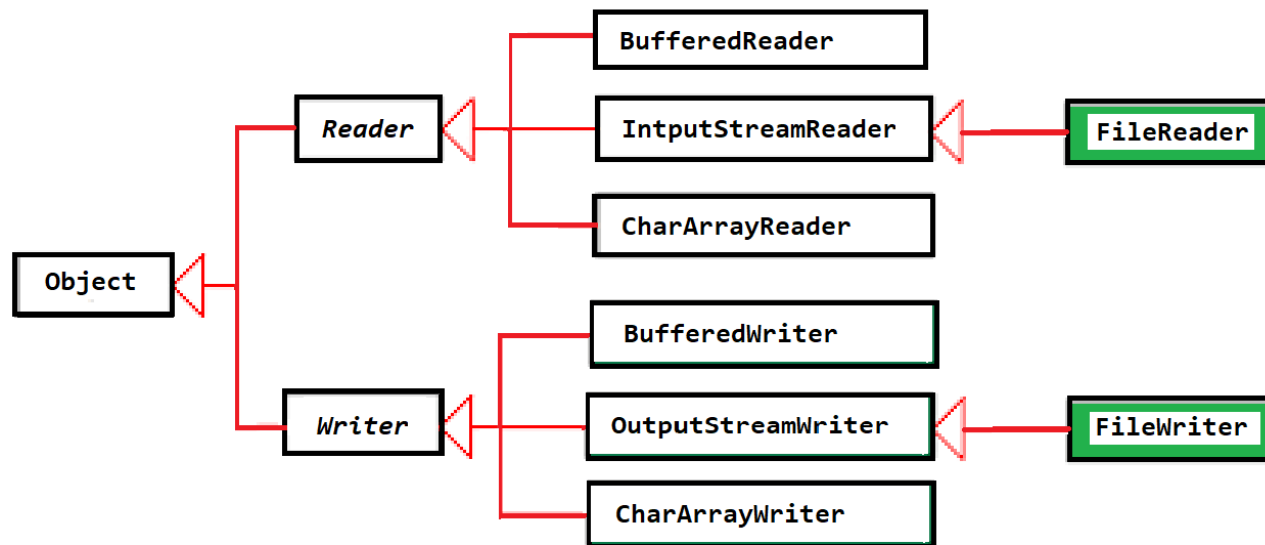
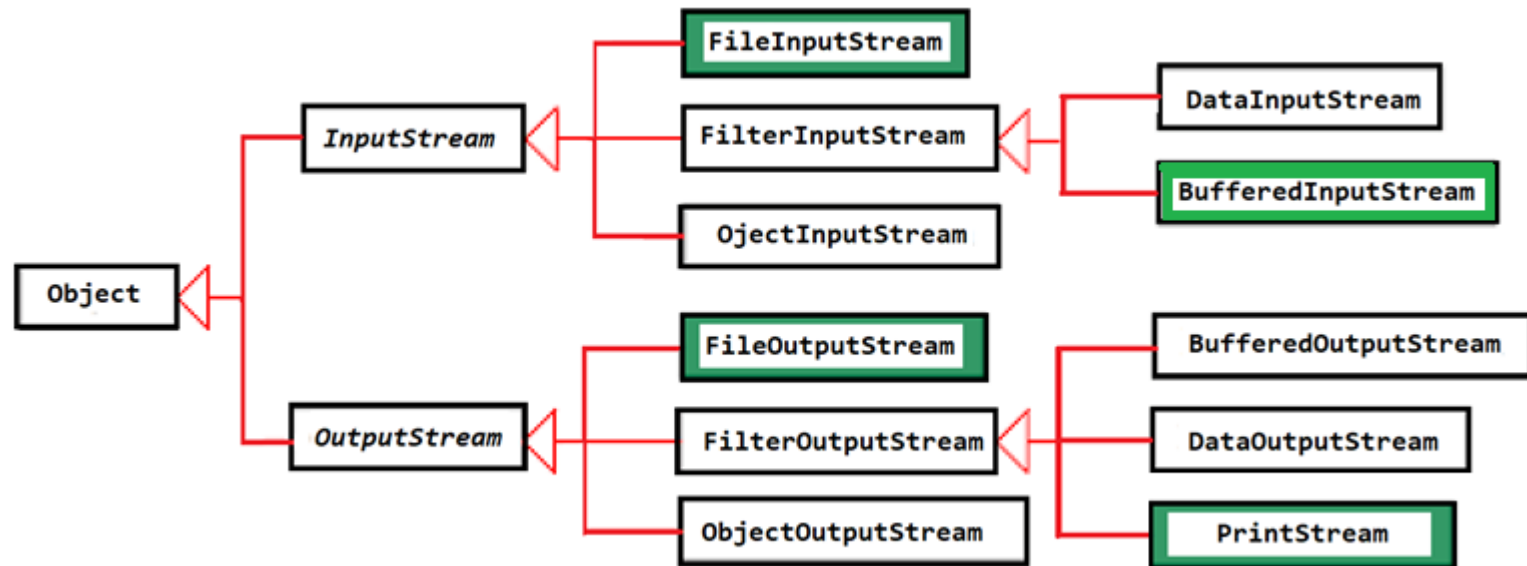
Fichero binario en un editor de texto. Puede contener caracteres así como otros tipos de datos que no pueden ser leídos por un editor de texto.

- **Flujos de caracteres (16 bytes):**

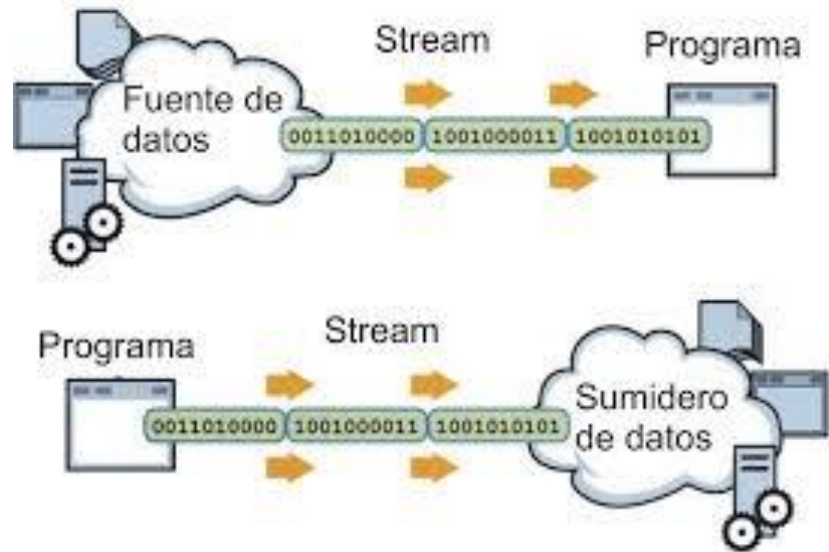
- Se utilizan para leer ficheros de texto.
- El flujo de datos se realiza de 16 en 16 bytes, facilitando el trabajo con los caracteres *Unicode* que tienen una codificación de 16 bytes.
- Las clase que trabajan con estos flujos descenden de las clases abstractas **Reader** y **Writer**.
- Las clases más comunes son **FileReader** y **FileWriter**

```
1  java    Murach's Beginning Java 49.5
2  jps     Murach's Java Servlets and JSP  49.5
3  txtp    TextPad 4.0      20.0
4
```

Fichero de texto en un editor de texto. Contiene caracteres. Los campos y registros pueden estar delimitados por caracteres especiales como tabulador y saltos de línea



FLUJOS DE BYTES



Archivos Binarios



InputStream: flujo de bytes de entrada

- La clase abstracta **InputStream** es la superclase de distintas clases que representan distintos orígenes de datos: un array de bytes, un objetos String, un fichero, una tubería, etc.
- En la clase **InputStream** se declaran los datos desde una fuente concreta y es la clase base de la mayor parte de los flujos de entrada de **java.io**. Soporta los métodos siguientes:
 - **int read()** → lee un solo byte de datos y los devuelve en el rango [0..255]. Devuelve -1 cuando alcanza el final del flujo y no puede seguir leyendo bytes.
 - **int read (byte[] bufer)** → lee un array de bytes hasta *bufer.length*. Devuelve el número de bytes leídos o -1 cuando llega al final del flujo.
 - **int read (byte[] bufer, int desde, int numBytes)** → lee *numBytes* bytes de flujo (o los que pueda) y los coloca a partir de la posición *desde* del búfer.
 - **void close()** → cierra el flujo de entrada que abrió el constructor sin argumentos, liberando los recursos asociados a ese flujo. No es necesario invocar a esta método ya que el flujo se cierra cuando se destruye el objeto aunque es conveniente ya que vuelca el buffer sobre el disco.

FileInputStream

- La clase **FileInputStream** tiene como superclase a **InputStream**.
- Se utiliza para determinar **un fichero como origen** de un flujo de entrada de bytes.
- **Constructores:**
 - **FileInputStream (File obj)**

```
InputStream miFIS = new FileInputStream(Paths.get("fondo1.jpg").toFile())
```

```
File miFile = new File("original.txt");  
InputStream miFIS = new FileInputStream(miFile);
```

- **FileInputStream ("nombreFichero")**

```
InputStream miFIS = new FileInputStream("entrada.txt");
```


Sentencia try con recursos

- Java 7 incorporó la sentencia **try con recursos** con el objetivo de **cerrar** los recursos de forma automática en la sentencia *try-catch-finally* y hacer más simple el código.
- Para **try con recursos**, el recurso utilizado debe implementar la interfaz **AutoCloseable**..
- CUANDO SE ABRE UN RECURSO EN LA SENTENCIA **TRY**, ÉSTE SE CIERRA AUTOMÁTICAMENTE CUANDO EL BLOQUE **TRY** SE EJECUTA O CUANDO SE PRODUCE UNA EXCEPCIÓN, SIN NECESIDAD DE CERRAR DE FORMA EXPRESA EL RECURSO EN EL BLOQUE **FINALLY**.

```
public void tryConRecursos() {  
    File file = new File("./tmp.txt");  
    try (InputStream elFileInputStream = new FileInputStream(file);) {  
        // Uso de InputStream para leer un archivo  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

• Ejemplo de flujos de entrada (FILEINPUTSTREAM):







- Vamos a leer un fichero de tipo imagen (*.jpg) que tiene contenido binario.
- Utilizaremos un flujo de la clase *InputStream* que permite *leer byte a byte* cualquier tipo de fichero.

```
static void lecturalImagenFlujoBinario(){
    int dato;
    try (InputStream miFIS = new FileInputStream(Paths.get("fondo1.jpg").toFile()))
    {
        //leo byte a byte; asigna y compara a la vez
        while ((dato = miFIS.read()) != -1) {
            //Imprime el código binario convertido en caracteres
            System.out.print((char) dato);
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

int read() → lee un solo byte de datos y los devuelve en el rango [0..255]. Devuelve -1 cuando alcanza el final del flujo.



OutputStream: flujo de bytes de salida

- 
- 
- 
- 
- 
- 
- La clase abstracta **OutputStream** es la superclase que se utilizará para manejar el flujo de salida que podría introducir bytes en un fichero del disco.
 - Soporta los métodos siguientes:
 - **void write(int i)** → escribe *i* como *byte*, que, aunque sea declarado como *int* es transformado a *byte*. Esto es porque habitualmente es el resultado de una operación previa.
 - **void write(byte [] bufer)** → escribe un array de *bytes* al final del fichero.
 - **void write(byte [] bufer, int start int count)** → escribe un *array bufer* de *bytes*, empezando en la posición *start* y escribiendo *count* de ellos, deteniéndose antes si encuentra el final del *array*.
 - **void flush()** → vacía el flujo de modo que los *bytes* que quedaran por escribir son escritos.
 - **void close()** → cierra el flujo de salida liberando los recursos asociados a ese flujo.

FileOutputStream

- La clase **FileOutputStream** tiene como superclase a **OutputStream**.
- Se utiliza para determinar **un fichero como destino** de un flujo de salida.
- **Constructores:**
 - **FileOutputStream (File obj)**

```
OutputStream miFOS = new FileOutputStream(Paths.get("fondo1.jpg").toFile());
```

```
File miFile = new File("original.txt");  
OutputStream miFOS = new FileOutputStream(miFile);
```

- **FileOutputStream (String nombreFichero, boolean)**

```
OutputStream miFOS = new FileOutputStream("entrada.txt");
```

- Cualquiera de los constructores puede tener un segundo parámetro *boolean* que en el caso de ser *true*, añade el flujo al final de fichero.

• Ejemplo de flujos de salida (FILEOUTPUTSTREAM):

- Vamos a leer un fichero de tipo imagen (*.jpg) que tiene contenido binario y lo vamos a escribir en otro fichero.
- Para la lectura utilizaremos un búfer formado por un *array* de *bytes*.
- Al final de la operación tendremos una copia de la imagen en otro fichero.

```
static void lecturaEscrituraImagenFlujoBinario(){
```

```
//Crea un flujo de entrada con el fichero origen y un flujo de salida con el fichero destino
```

```
//Se utiliza try con recursos para omitir el bloque finally para cerrar los recursos en caso de excepción
```

```
try (InputStream miFIS =
```

```
    new FileInputStream(Paths.get("fondo.jpg").toFile());
```

```
    OutputStream miFOS =
```

```
        new FileOutputStream(Paths.get("fondo1.jpg").toFile());)
```

```
{
```

```
    //Se crea un array de bytes de 1KB
```

```
    byte[] buf = new byte[1024];
```

```
    //Lectura del flujo de entrada y almacenamiento en buf
```

```
    while ( miFIS.read(buf) != -1) {
```

```
        //Escritura del búfer de entrada en el flujo de salida
```

```
        miFOS.write(buf);
```

```
    }
```

```
} catch (FileNotFoundException e)
```

```
    e.printStackTrace();
```

```
} catch (IOException e) {
```

```
    e.printStackTrace();
```

```
}
```

```
}
```

int read (byte[] bufer)

lee un array de bytes del flujo de entrada y devuelve el número de bytes leídos o -1 cuando alcanza el final de fichero

void write(byte[] bufer) escribe un array bufer de bytes al final del fichero

Fujos standar

- **Flujos estándar** son flujos de bytes:
 - **System.in**
 - Instancia de la clase **BufferedInputStream** que a su vez tiene como superclase a la clase abstracta **InputStream**: **flujo de bytes de entrada**
 - Métodos:
 - **read()** → permite leer un byte de la entrada como un entero.
 - **skip(n)** → ignora n bytes de la entrada.
 - **available()** → número de bytes disponibles para leer en la entrada.
 - **System.out**
 - Instancia de la clase **PrintStream**: **flujo de bytes de salida**.
 - Métodos:
 - **print(String), println(String)**, muestra por pantalla el String.
 - **flush()**, vacía el buffer de salida escribiendo su contenido.
 - **System.err**
 - Instancia de la clase **PrintStream**: **flujo de bytes de salida**.
 - Funcionamiento similar a **System.out**.
 - Se utiliza para enviar mensajes de error a un fichero *log* o a la pantalla.

• Ejemplo1 de flujos estándar:

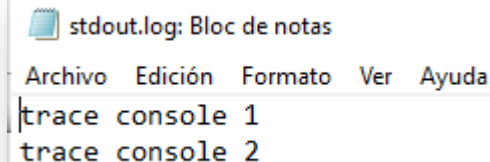
- Se lee una línea por teclado carácter a carácter utilizando el flujo estándar de entrada que es producido por el teclado.
- Se muestra cada carácter leído en una línea.
- Finaliza cuando introducimos el carácter de fin de línea ('\n')

```
static void lecturaTecladoFlujoBinario(){
    int c;
    int contador=0;
    try {
        while ((c=System.in.read())!='\n'){
            contador++;
            System.out.println((char)c);
        }
        System.out.println("Se han introducido "+contador+" caracteres");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

• Ejemplo2 de flujos estándar:

- Se redirige el flujo de salida de **System.err** y de **System.out** hacia dos ficheros y se envían mensajes e información de excepciones.

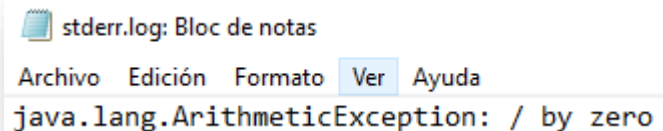
```
static void dirigirErroresAFichero() {  
    try {  
        //Crear dos flujos de salida hacia sendos ficheros  
        OutputStream fout = new FileOutputStream("stdout.log");  
        OutputStream ferr = new FileOutputStream("stderr.log");  
        //Crar dos objetos PrintStream con los flujos de salida creados  
        PrintStream miOut = new PrintStream(fout);  
        PrintStream miErr = new PrintStream(ferr);  
        //Dar valor a las propiedades out y err de System que definen los flujos de salida estándar y de error  
        System.setOut(miOut);  
        System.setErr(miErr);  
        //Envío un string al flujo de salida estándar mediante el objeto creado  
        miOut.println("trace console 1");  
        //Envío un string al flujo de salida estándar que está modificado  
        //ya que out tiene el valor de miOut (System.setOut(miOut);)  
        System.out.println("trace console 2");  
        //Provoco una excepción que será capturada por catch  
        int a = 100 / 0;  
    } catch (Exception ex) {  
        //Envío el valor de la excepción a System.err cuyo destino del flujo está modificado  
        System.err.println(ex.toString());  
    }  
}
```



stdout.log: Bloc de notas

Archivo Edición Formato Ver Ayuda

trace console 1
trace console 2



stderr.log: Bloc de notas

Archivo Edición Formato Ver Ayuda

java.lang.ArithmeticException: / by zero

- **Ejercicio: modificar el Ejemplo2 de flujos estándar para que:**
 - El flujo de salida **System.out** será dirigido al fichero **salida.txt**.
 - El flujo de salida **System.err** será dirigido al flujo de salida **System.out**.

- **Solución *casi* correcta del Ejercicio.**
- Indica qué hay que modificar para que la solución sea correcta.

```
static void dirigirErroresAFichero_Modificado() {  
    try {  
        //Crear un flujos de salida hacia el fichero salida.txt  
        OutputStream fout = new FileOutputStream("salida.txt");  
        //Crar un objeto PrintStrem a partir del flujo de salida creado  
        PrintStream miOut = new PrintStream(fout);  
        //Dar valor a las propiedades out y err según los requisitos deseado  
        System.setOut(miOut);  
        System.setErr(System.out);  
        //Envío un string al flujo de salida estándar mediante el objeto creado  
        miOut.println("trace console 1");  
        //Envío un string al flujo de salida estándar que está modificado  
        //ya que out tiene el valor de miOut (System.setOut(miOut);)  
        System.out.println("trace console 2");  
        //Provoco una excepción que será capturada por catch  
        int a = 100 / 0;  
    } catch (Exception ex) {  
        //Envío la excepción a System.err cuyo destino del flujo está modificado  
        System.err.println(ex.toString());  
    }  
}
```

Excepciones: throws

- A la hora de tratar las excepciones se puede optar por no hacerlo en el método donde que se producen sino en el método *padre*.
- Para enviar una excepción al método que realizó la llamada se utilizar en la cabecera el método la clausula siguiente:

throws ClaseDerivadaException

- **Ejemplo:**

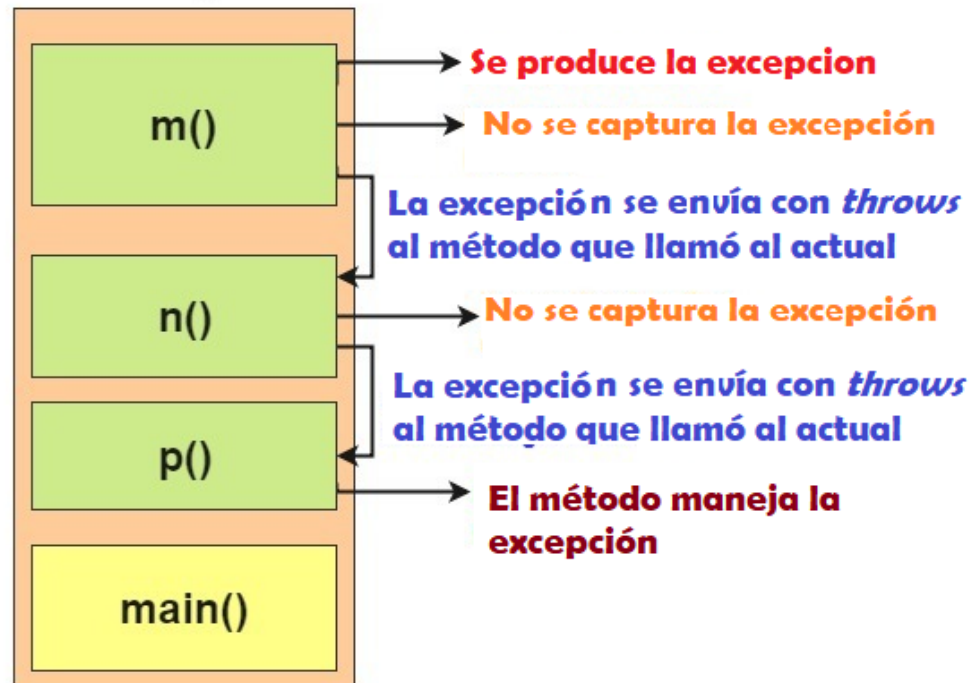
```
public String leerFichero (String nombreFichero) throws IOException{  
    /*código del método*/  
}
```

- El método del ejemplo lanza la excepción *IOException* para que sea tratada por el método que llama a *leerFichero()*.

Excepciones: throws

- Al implementar un método, hay que decidir si las excepciones se propagarán hacia arriba, en la pila de llamadas, (**throws**) o se capturarán en el propio método (**catch**).
- Si todos los métodos de una aplicación lanzan sus excepciones con **throws**, tendría que existir, como mínimo, un **try/catch** en el método *main*.

Memory Stack



Excepciones: throws. Ejemplos

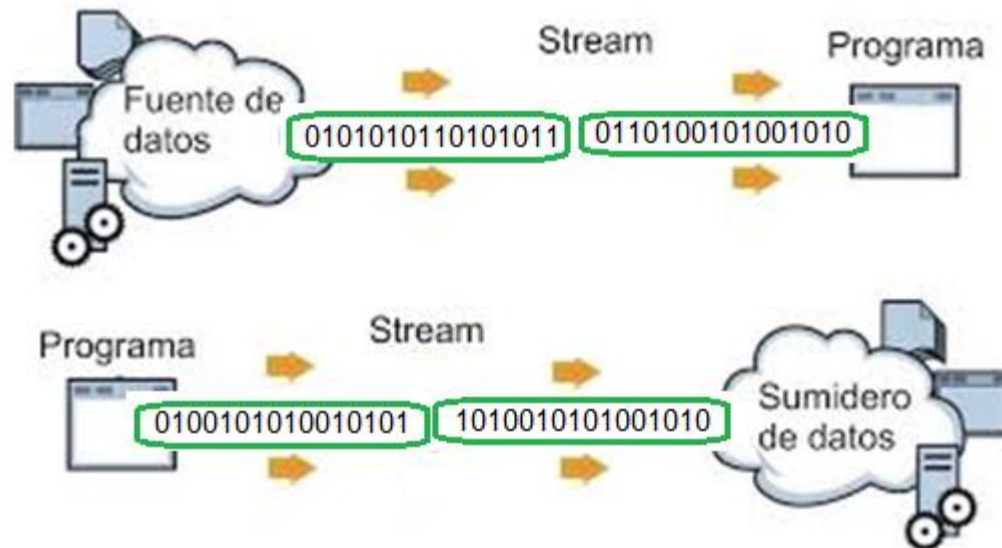
```
public void f() throws IOException{  
    //Fragmento de código que puede  
    //lanzar una excepción de tipo IOException  
}
```

Métodos equivalentes con tratamiento de excepciones diferentes

```
public void f(){  
    //Fragmento de código que no provoca excepciones  
    try{  
        //Fragmento de código que puede  
        //una excepción de tipo IOException  
    }catch (IOException e){  
        //Tratamiento de la excepción  
    }finally{  
        //Siempre se ejecuta  
    }  
}
```

- **Ejercicio de uso de THROWS:** Realizar modificaciones sobre alguna de los programas realizados para que envíen a *main* el tratamiento de las excepciones mediante la clausula *throws*.

FLUJOS DE CARACTERES



Archivos de Texto



Ficheros de texto

- Los ficheros de texto son aquellos que almacenan caracteres alfanuméricos en formato estándar (ASCII, UNICODE, UTF-8) y normalmente son generados por un editor.
- Para trabajar con ellos usaremos as clases **FileReader** para leer caracteres y **FileWriter** para escribir los caracteres en un fichero.



Reader: flujos de entrada de caracteres

- **Reader** es una clase abstracta que define las funcionalidades básicas de lectura de **caracteres Unicode**.
- Soporta los métodos siguientes:
 - **int read()** → lee un carácter y devuelve el carácter leído pasado a entero devuelve -1 si el fichero se ha acabado.
 - **int read (char[] bufer)** → lee un array de caracteres hasta *bufer.length*. Devuelve el número de caracteres leídos o -1 cuando llega al final del flujo.
 - **int read (char[] bufer, int desde, int numChar)** → lee *numChar* caracteres de *flujo* (o los que pueda) y los coloca a partir de la posición *desde* del búfer.
 - **void close()** → cierra el fichero.

FileReader

- La clase **FileReader** tiene como superclase a **InputStreamReader** que a su vez es hija de **Reader**.
- Se utiliza para determinar **un fichero como origen** de un flujo de entrada de caracteres.
- **Constructores:**
 - **FileReader (File obj)**

```
Reader miFIS = new FileReader (Paths.get("poema.txt").toFile())
```

```
File miFile = new File("original.txt");  
Reader miFIS = new FileReader(miFile);
```

- **FileReader ("nombreFichero")**

```
Reader miFIS = new FileReader("entrada.txt");
```

• Ejemplo de uso de FileReader:

- Se lee un fichero almacenado en la misma carpeta del proyecto *poesía.txt*.
- El fichero se lee utilizando un búfer en forma de array de caracteres.
- Los caracteres leídos se almacenan en un *StringBuilder* y se muestran por pantalla.

```
static void leeTodosLosCaracteres() throws IOException {  
    //Objeto File con el fichero de texto  
    File miFiles= Paths.get("poesia.txt").toFile();  
    //Objeto para almacenar los caracteres leídos del fichero  
    StringBuilder miSB = new StringBuilder();  
    //Objeto Reader para leer el fichero de texto en UT-8  
    Reader miFileReader = new FileReader(miFiles,Charset.forName("UTF-8"));  
    //fuffer de char para ir leyendo del fichero  
    char[] bufer = new char[1024];  
    //Leo bufer a bufer hasta el final de fichero  
    while (miFileReader.read(bufer)!=-1) {  
        //Agrego a la cadena los caracteres leídos  
        miSB.append(bufer);  
    }  
    //Cierro el fichero  
    miFileReader.close();  
    //Muestro por pantalla la cadena, resultado de leer el fichero  
    System.out.println(miSB.toString());  
}
```

Writer: flujos de salida de caracteres

- **Writer** es una clase abstracta que define las funcionalidades básicas de escritura de caracteres *Unicode* en ficheros de texto.
- Soporta los métodos siguientes:
 - **void write(String s)** → escribe en el fichero la cadena *s*.
 - **void write(char [] bufer)** → escribe un array de *caracteres* al final del fichero.
 - **void write(char [] bufer, int start, int count)** → escribe un *array bufer* de *caracteres*, empezando en la posición *start* y escribiendo *count* de ellos, deteniéndose antes si encuentra el final del *array*.
 - **void write(String s, int start, int count)** →
 - **void flush()** → vacía el flujo de modo que los *caracteres* que quedaron por escribir son escritos.
 - **void close()** → cierra el flujo de salida liberando los recursos asociados a ese flujo.

FileWriter

- La clase **FileWriter** tiene como superclase a **InputStreamWriter** que a la vez es hija de **Writer**.
- Se utiliza para determinar **un fichero como origen** de un flujo de entrada.

- **Constructores:**

- **FileWriter (File *obj*)**

```
Writer miFIS = new FileWriter(Paths.get("fondo1.jpg").toFile())
```

```
File miFile = new File("original.txt");  
Writer miFIS = new FileWriter(miFile);
```

- **FileWriter ("*nombreFichero*")**

```
Writer miFIS = new FileWriter("entrada.txt");
```

- Cualquiera de los constructores puede tener un segundo parámetro *boolean* que en el caso de ser *true*, añade el flujo al final de fichero.

• Ejemplo de uso de Writer:

- Añade una cadena al final de un fichero existente.

```
static void escribeCaracteres()  
    throws IOException {  
  
    //Creo un objeto FileWriter a partir de un fichero existente  
    //para añadir texto al final  
    Writer miFW = new FileWriter(Paths.get("poesia.txt").toFile(), true);  
    //Determino el texto a añadir  
    String cadena = "\n Poema de Rosalía de Castro";  
    //Agrego la cadena al final del fichero  
    miFW.write(cadena);  
    //Cierro el fichero  
    miFW.close();  
}
```



Buffered....

- Si usamos sólo **FileInputStream**, **FileOutputStream**, **FileReader** o **FileWriter**, cada vez que hagamos una lectura o escritura, se hará físicamente en el disco duro. Si escribimos o leemos pocos caracteres cada vez, el proceso se hace costoso y lento, con muchos accesos a disco duro.
- Los **BufferedReader**, **BufferedInputStream**, **BufferedWriter** y **BufferedOutputStream** añaden un buffer intermedio. Cuando leamos o escribamos, esta clase controlará los accesos a disco.
- Si vamos escribiendo, se guardará los datos hasta que tenga bastantes datos como para hacer la escritura eficiente.
- Si queremos leer, la clase leerá muchos datos de golpe, aunque sólo nos dé los que hayamos pedido. En las siguientes lecturas nos dará lo que tiene almacenado, hasta que necesite leer otra vez.
- Esta forma de trabajar hace los accesos a disco más eficientes y el programa correrá más rápido. La diferencia se notará más cuanto mayor sea el fichero que queremos leer o escribir.

BufferedInputStream

Hacer un **cálculo del tiempo de ejecución** de la lectura del fichero utilizando ***BufferedInputStream* vs leer directamente de *FileInputStream***

```
/**
 * Lee un fichero binario utilizando BufferedInputStream
 */
static void leerBufferInputStream() {
    //Para controlar el tiempo de ejecución (nanosegundos)
    long startTime = System.nanoTime();
    File fichero=Paths.get("foto.jpg").toFile();

    try (BufferedInputStream miBIS = new BufferedInputStream(new FileInputStream(fichero)))
    {
        int data=0;
        while ((data = miBIS.read()) != -1) {
            System.out.print((char) data);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    //Para controlar el tiempo de ejecución
    long time = System.nanoTime() - startTime;
    System.out.println("El tiempo transcurrido es "+time );
}
```

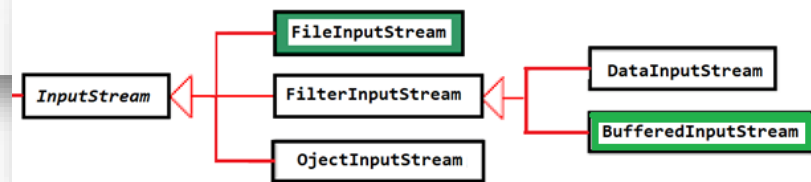
Utiliza un buffer para leer el fichero

`BufferedInputStream miBIS = new BufferedInputStream(new FileInputStream(fichero))`

Objeto que accederá al búfer para leer los datos almacenados en él

Crear un buffer donde guardará bytes del fichero en memoria y evitará el nº de accesos físicos al disco.

Lee los bytes del fichero



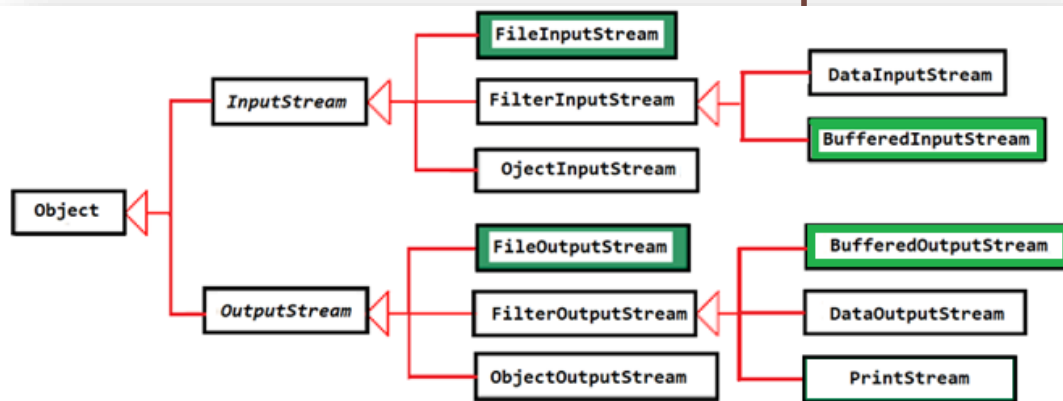
BufferedOutputStream

Hacer un **cálculo del tiempo de ejecución** de la lectura/escritura comparando la ejecución sin usar búferes y la ejecución utilizándolos

```
/**
 * Lee y escribe un fichero binario utilizando
 * BufferedInputStream y BufferedOutputStream
 */
static void lecturaEscrituraBufferFlujoBinario_Imagen() {

    int dato;
    try (BufferedInputStream miBIS = new BufferedInputStream(
        new FileInputStream(Paths.get("fondo.jpg").toFile()));
        BufferedOutputStream miBOS = new BufferedOutputStream(
            new FileOutputStream(Paths.get("fondoCopia.jpg").toFile()));)
    {
        while ((dato = miBIS.read()) != -1) {
            miBOS.write(dato);
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Utiliza un buffer para leer el fichero y otro para escribir

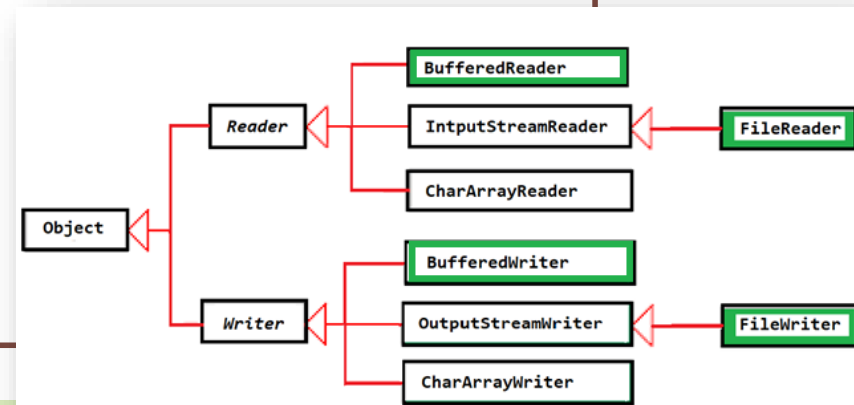


BufferedReader/BufferedWriter

- Los **búferes** correspondientes a **Reader** y **Writer** funcionan de una forma equivalente a los que hemos visto con flujos de bytes.

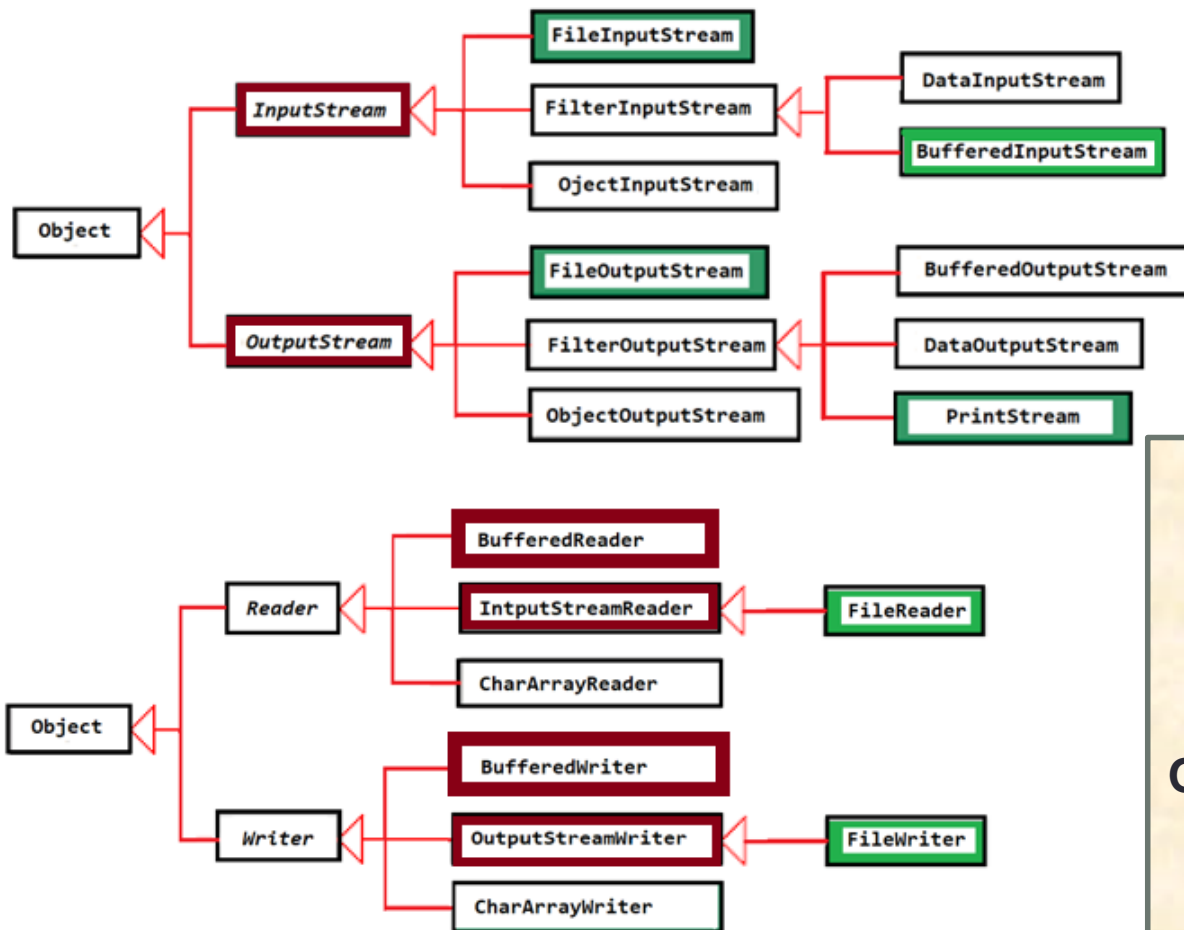
```
/**
 * Lee y escribe un fichero de texto utilizando BufferedReader y Buffer
 */
static void lecturaEscrituraBufferFlujoTexto_Texto() {
    int dato;
    //Objeto File con el fichero de texto
    File miFilesOrigen= Paths.get("poesia.txt").toFile();
    File miFilesDestino= Paths.get("poesiaCopia.txt").toFile();

    //Objeto Reader para leer el fichero de texto en UT-8
    try(BufferedReader mBR = new BufferedReader(
        new FileReader(miFilesOrigen,Charset.forName("UTF-8")));
        BufferedWriter mBW= new BufferedWriter(
            new FileWriter(miFilesDestino, Charset.forName("UTF-8")))){
        //Leo dato a dato del bufer hasta el final de fichero
        while ((dato=mBR.read())!=-1) {
            //Escribo en el búfer de salida
            mBW.write(dato);
        }
    }catch (FileNotFoundException e){
        //Tratar la excepción
    }catch (IOException e){
        //Tratar la excepción
    }
}
```



Práctica: Input/Output Stream

- Crear un fichero de texto llamado **líneas.txt** que escriba 100 líneas, cada línea debe ser: Línea número*numero*...
- Copiar el fichero de texto creado a otro fichero de texto (**lineasCopia.txt**) tomando los datos a partir de un flujo **FileInputStream** que es el que accede al fichero y después convertirlo en un **InputStreamReader** y mediante un **BufferedReader** copiarlo a un fichero de texto.



Tendremos que crear un *stream* de texto a partir de un *stream* binario

A partir de un **InputStream** se podrá crear un **InputStreamReader**.
A partir de un **OutputStream** se podrá crear un **OutputStreamWriter**



```
/**
```

```
* Crea un flujo de lectura de texto a partir de un flujo binario  
*/
```

```
static void crearReaderAPartirDeFileInputStream() {
```

```
    String linea=null;
```

```
    //Se toma un flujo de bytes como parámetro para crear un  
    InputStreamReader
```

```
        try (InputStream miFIS = new  
FileInputStream(Paths.get("poema.txt").toFile())) {  
            InputStreamReader miISR = new InputStreamReader(miFIS);
```

```
                //Se crea un BufferedReader a partir del  
                InputStreamReader
```

```
                BufferedReader miBR=new BufferedReader(miISR);
```

```
                //Se lee el buffer línea a línea y se muestra por  
pantalla
```

```
                while ((linea=miBR.readLine())!=null){  
                    System.out.println(linea);  
                }
```

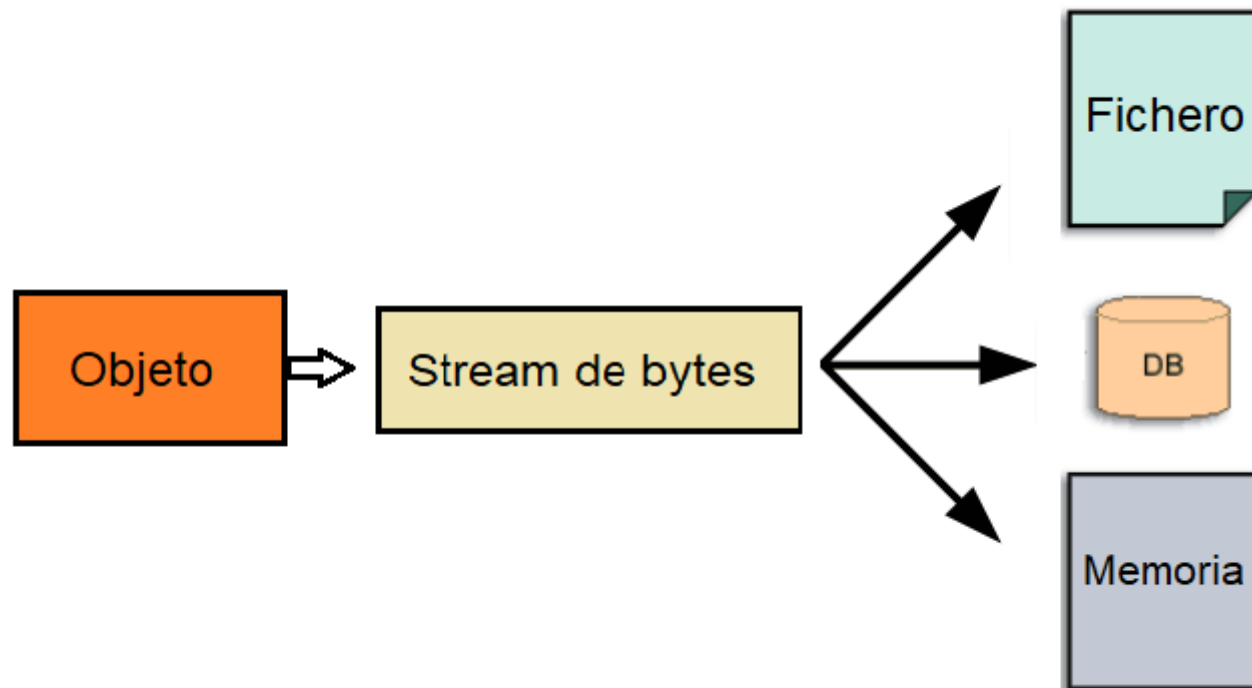
```
            } catch (Exception e) {  
                System.out.println(e.toString());  
            }
```

```
}
```



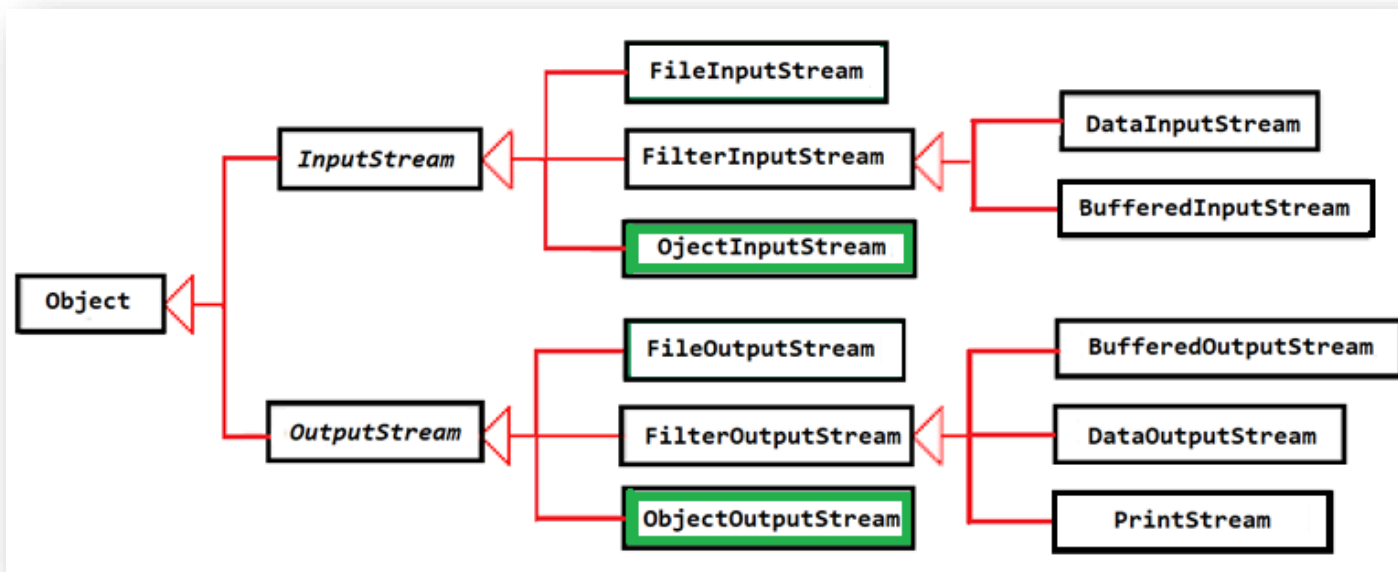



FLUJOS DE OBJETOS



Leer y escribir objetos

- Hasta ahora hemos visto como se guardan tipos de datos primitivos en ficheros, pero también podremos guardar objetos en ficheros binarios.
- Para guardar objetos en ficheros binarios la clase del objeto tiene que implementar la interfaz **Serializable** que dispone de una serie de métodos para esta operación.
- La serialización de objetos 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 poder leer y escribir objetos *serializables* a un *stream* se utilizan las clases **ObjectInputStream** y **ObjectOutputStream**.



ObjectOutputStream

```
public class Pessoa implements Serializable {  
  
    private String nome;  
    private int idade;  
  
    public Pessoa(String nome, int idade) {  
        this.nome = nome;  
        this.idade = idade;  
    }  
  
    public Pessoa() {  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String name) {  
        nome = name;  
    }  
  
    public int getIdade() {  
        return idade;  
    }  
  
    public void setIdade(int age) {  
        idade = age;  
    }  
}
```

- Utilizaremos los objetos creados a partir de la clase **Pessoa** como fuente de información de un fichero binario.
- La clase implementará la interfaz **Serializable** para poder guardar sus objetos en ficheros binarios.
- Necesitamos crear un flujo de salida a disco con **FileOutputStream** y a continuación enlazarlo con el flujo de salida **ObjectOutputStream** que será quien procese los datos.

```
File miFile = new File("FichPessoa.dat");  
FileOutputStream miFOS = new FileOutputStream (miFile);  
ObjectOutputStream miOOS = new ObjectOutputStream(miFOS);  
miOOS.writeObject(objPessoa); //Escribe el objeto en el fichero
```

```
static void crearFicheroSerializable() {
    //Definimos variable Persoa
    Persoa amigo;
    //Declaramos o ficheiro. Se non existe, créao. Se existe sobreescribe.
    File miFile = new File("ficheiropersoas.dat");
    //Creamos un fluxo de saída: aplicación -> ficheiro
    try (FileOutputStream miFOS = new
        FileOutputStream(miFile);
        ObjectOutputStream miOOS = new
            ObjectOutputStream(miFOS)) {
        //Definimos un array de nomes que queremos escribir no ficheiro
        String nomes[] = {"Antonio", "Rocio", "Cris", "José Luis"};
        //Definimos un array de idades que acompañarán aos nomes
        int idades[] = {31, 28, 35, 33};
        //Bucle para crear os obxectos e escribir no ficheiro
        for (int i = 0; i < nomes.length; i++) {
            //Creo un obxecto Persoa
            amigo = new Persoa(nomes[i], idades[i]);
            //Almaceno os obxectos Persoa no ficheiro
            miOOS.writeObject(amigo);
        }
    } catch (IOException e) {
        //Tratar excepciones
    }
}
```

Crea un fichero de
objetos

ObjectInputStream

- Para leer objetos que han sido almacenados en un fichero en las condiciones antes expuestas debemos utilizar la clase **ObjectInputStream**.

```
File miFile = new File("FichPersoa.dat");  
FileInputStream miFIS = new FileInputStream (miFile);  
ObjectInputStream miOIS = new ObjectInputStream(miFIS);  
//Lee el objeto, es necesario castearlo para acceder a él de forma legible  
objPersoa=(Persoa)miOIS.readObject();
```

Lee un fichero de
objetos

```
static void leerFicheroSerializable() {  
    //Definimos un objeto de tipo Pessoa  
    Pessoa amigo;  
    //Declaramos o ficheiro.Se non existe, créao.  
    File ficheiro = new File("ficheiropessoas.dat");  
    //Creamos un fluxo de entrada: ficheiro -> aplicación  
    try (FileInputStream miFIS = new  
        FileInputStream(ficheiro);  
        ObjectInputStream miOIS = new  
            ObjectInputStream(miFIS);) {  
        //Bucle para ler do ficheiro  
        //Cando chega ao fin de arquivo amigo é null!!!  
        while ((amigo = (Pessoa) miOIS.readObject()) != null) {  
            //Visualizamos contido do ficheiro  
            System.out.println("Nome:" + amigo.getNome() +  
                "Idade:" + amigo.getIdade());  
        }  
    } catch (ClassNotFoundException | IOException e) {  
        //Tratar excepcion  
    }  
}
```



ObjectOutputStream puede darnos problemas, por ejemplo si escribimos datos en un fichero y lo cerramos para después volverlo a abrir y agregar datos (**FileOutputStream**(fichero, true)). Esto es debido a que escribe una nueva cabecera al final de los objetos introducidos anteriormente y después se van añadiendo el resto de datos. Esto origina que al leer el fichero se produzca la excepción

StreamCorruptedException.

```
• /**
• * Redefinición de la clase ObjectOuputStream para que no escriba una cabecera
• * al inicio del Stream. Para no tener problemas con las cabeceras de los
• * objetos y evitar el error StreamCorruptedException, creamos una clase con
• * nuestro propio ObjectOutputStream, heredando del original y redefiniendo el
• * método writeStreamHeader() vacío, para que no haga nada.
• */
public class MiObjectOutputStream extends ObjectOutputStream {

    /**
    * Constructor que recibe OutputStream
    */
    public MiObjectOutputStream(OutputStream out) throws IOException {
        super(out);
    }

    /**
    * Constructor sin parámetros
    */
    protected MiObjectOutputStream() throws IOException, SecurityException {
        super();
    }

    /**
    * Redefinición del método de escribir la cabecera para que no haga nada.
    */
    protected void writeStreamHeader() throws IOException {

    }
}
```

Práctica: Flujos de objetos

- Crear un proyecto que:
 - Contenga una clase Empleado con:
 - Propiedades privadas para nombre, apellidos, salario y puesto de trabajo
 - Métodos setter y getter para todas las propiedades.
 - Método público `plus(double sueldoPlus)` que permita incrementar el salario en el valor que se pasa por parámetro.
 - Redefinición del método `toString()` para que, aplicado a un objeto, muestre un mensaje equivalente a “Ana Pérez es administrativa y tiene como salario 2000€”
 - Redefinición del método `equals()` para que determine que dos empleados son iguales si tienen el mismo nombre, el mismo apellido y el mismo puesto de trabajo.
 - Crear un menú que permita
 1. Introducir nuevos empleados y guardarlos en un fichero llamado `EmpleadosEmpresa.dat`. Este fichero puede crearse de nuevo o, si ya existe, debe agregar los datos al fichero existente.
 2. Mostrar los datos de un empleado concreto.
 3. Mostrar datos de todos los empleados.
 4. Mostrar los puestos de trabajo de la empresa.
 5. Salir del menú.
 - Podéis añadir una opción de menú que sea “Incrementar sueldo”.