

UT8

Excepciones. IO en Java.

Módulo - Programación
Ciclo - Desarrollo de Aplicaciones Web
CI Maria Ana Sanz

Contenidos

- Programación defensiva
- Excepciones
 - lanzar una excepción
- Jerarquía de excepciones
- Excepciones verificadas y no verificadas
- Tratamiento de excepciones
 - verificadas
 - no verificadas
- try ... catch ... finally
- lanzar y capturar múltiples excepciones
- propagar una excepción
- Definiendo nuestras propias excepciones

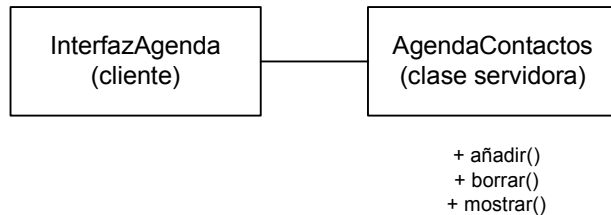
Contenidos

- I/O en Java
 - concepto de flujo
 - flujos de bytes y de caracteres
 - flujos de proceso
- El paquete java.io
- Readers / Writers
 - FileReader / BufferedReader
 - FileWriter / BufferedWriter / PrintWriter
- clase File
- clase Scanner
- Serialización

Excepciones en Java

Programación defensiva

- Errores sintácticos
 - los detecta el compilador
- Errores lógicos
 - el programa no hace lo que debe / resultados incorrectos
 - se detectan a través de tests/debug
- Errores de ejecución
 - el programa falla en ejecución y se para
- Nos centraremos en la programación defensiva desde el punto de vista de una clase servidora (la que ofrece servicios)



Programación defensiva

- Hasta ahora o hemos eludido errores o si los hemos tratado ha sido
 - a) con if comprobando los argumentos de un método y mostrando mensajes de error

```
if (n < 1)
    System.out.println("Argumento incorrecto");
```

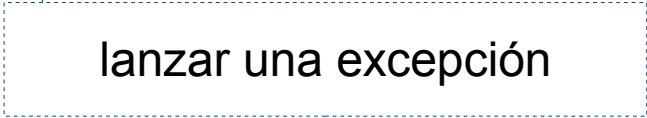
- b) devolviendo algún código retorno

```
public boolean borrarNota(int i)
{
    if (i < 0 || i >= numNotas)
        return false;
    agenda.remove(i);
    return true;
}
```

Programación defensiva

- c) algunas veces hemos lanzado una excepción (no verificadas)

```
if (i < 0 || i >= numNotas)
    throw new IllegalArgumentException("Argumento
                                   incorrecto");
```



lanzar una excepción

Programación defensiva

- Programa **correcto**
 - hace lo que debe (se consigue con los test)
- Programa **robusto**
 - ante situaciones de error se comporta bien, es capaz de responder adecuadamente - **excepciones**

Qué debemos saber sobre las excepciones?

■ Excepciones

- mecanismo utilizado por Java para manejar los errores que pueden ocurrir en un programa
- separa la gestión de errores de la lógica de nuestro programa

■ Qué son?

- objetos que se lanzan ante una situación de error
 - intento de división entre 0 (ArithmeticException)
 - uso de un índice incorrecto (ArrayIndexOutOfBoundsException)
 - no encontrar un fichero (FileNotFoundException)
 - intento de uso de una referencia a un objeto null (NullPointerException)

Qué debemos saber sobre las excepciones?

■ Una excepción

- se lanza – **throw** (explícitamente) / implícitamente
- se avisa – **throws**
- se captura o se propaga – se denomina **Tratar la excepción**

try ... catch

gestor de excepción

throws

se evita que el programa
falle y se acaba de forma
elegante

Qué debemos saber sobre las excepciones?

■ Tipos de excepciones

■ no verificadas

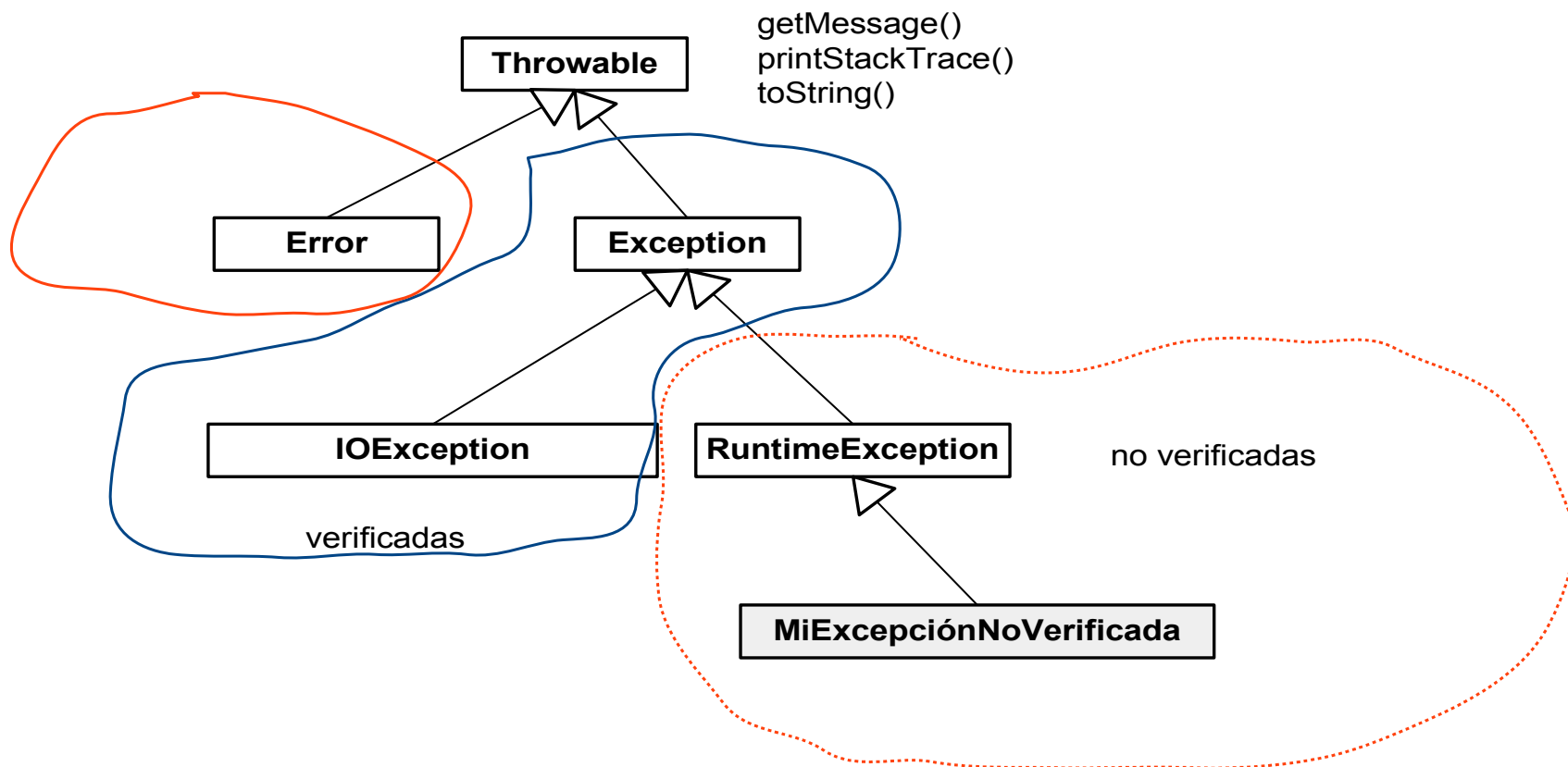
- deriva de RuntimeException o Error
- evitables, errores que el programador puede evitar
- **no obligatorio tratarlas**
- no hay que avisar (propagar)
- el coste de verificarlas puede ser mayor que el beneficio de capturarlas

■ verificadas

- derivan de Exception / IOException
- **obligatorio tratarlas** (o se capturan o se propagan)
 - el compilador genera un error si no se tratan
- errores de E/S (leer de un fichero, escribir en disco, conectar a una BD)
 - el programador no puede evitarlas

Jerarquía de excepciones

- en el paquete *java.lang*



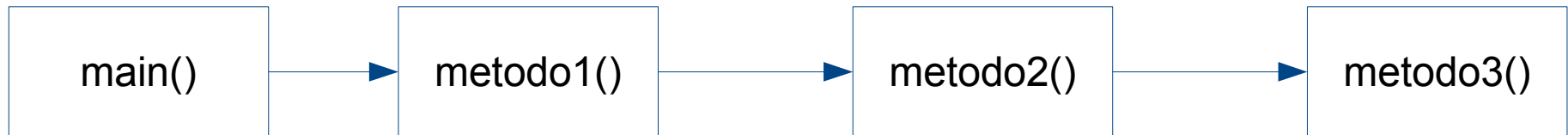
Qué es una excepción

- evento que ocurre durante la ejecución de un programa y que interrumpe su flujo normal de ejecución
- las excepciones ocurren dentro de los métodos (código)
- Cuando ocurre un error el método crea un objeto excepción que contiene
 - información sobre el error que se ha producido
- crear un objeto excepción y enviarlo al sistema (JVM) para que lo trate es lo que se llama **lanzar una excepción**

Qué es una excepción

- si la JVM no encuentra un gestor de excepción adecuado para la excepción generada el programa termina
- si se encuentra un gestor apropiado la JVM pasa la excepción al gestor para que la maneje
 - un gestor de excepciones es apropiado si el tipo del objeto excepción lanzado es compatible con el tipo que puede manejar el gestor
 - se dice que se ha capturado la excepción

Qué es una excepción



error
(throw
ímplicito /
explícito)

Pila de llamadas - stack

```
java.lang.IllegalArgumentException: Error
  at Ejemplo01.metodo3(Ejemplo01.java:248)
  at Ejemplo01.metodo2(Ejemplo01.java:240)
  at Ejemplo01.metodo1(Ejemplo01.java:232)
  at Ejemplo01.main(Ejemplo01.java:224)
```

:excepcion

Lanzar una excepción

- Para lanzar una excepción:
 - a) se crea un objeto excepción (objeto de tipo `IllegalArgumentException`, `NullPointerException`,)
 - b) se lanza la excepción a través de **throw**

```
throw new tipo_excepción("Mensaje descriptivo");
```

- el mensaje está disponible después al receptor de la excepción a través del accesor `getMessage()` del objeto excepción

Excepciones no verificadas y verificadas

■ No verificadas

- son todas las subclases de la clase `RuntimeException`
- se lanzan cuando se dan situaciones de error evitables que conducen a errores lógicos de programación
 - `IllegalArgumentException`, `NumberFormatException`

■ Verificadas

- derivan de `Exception`
- se lanzan cuando se produce un error fuera del control del programador (operaciones de E/S) y para aquellos en los que una recuperación es posible

Excepciones no verificadas y verificadas

Las excepciones no verificadas pueden ignorarse, las verificadas no.

Si un objeto cliente llama a un método que puede lanzar una excepción verificada ha de gestionar la excepción obligatoriamente (no puede ignorarse, o se capturan o se propagan).

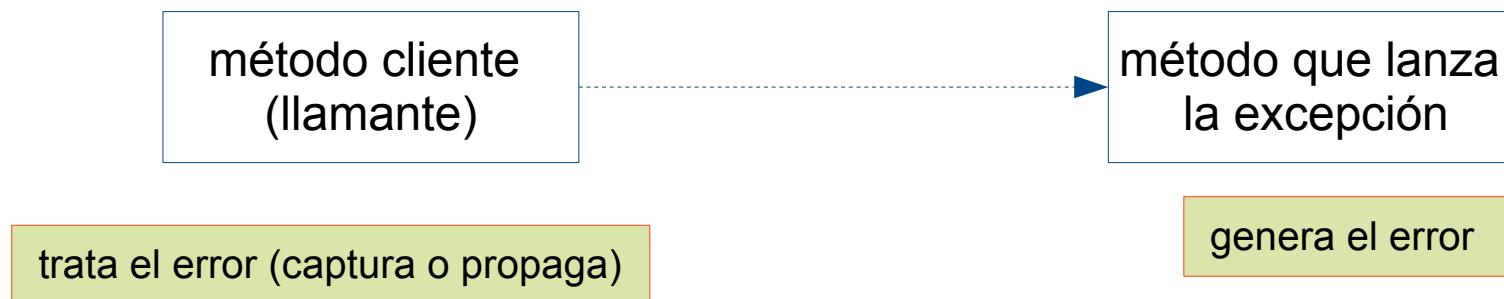
Capturar la excepción significa que se proporciona un gestor de excepciones para ella (try ... catch).

Tratamiento de excepciones

- Los métodos que llaman a métodos que pueden lanzar excepciones tienen que:
 - capturar la excepción y tratarla o
 - volverla a lanzar, es decir, propagarla
- Obligatorio para las verificadas

Tratamiento de excepciones

- Cuando se lanza una excepción hay que considerar dos efectos:
 - en el método que lanza la excepción
 - la ejecución del programa termina , no se devuelve ningún valor y el control del programa no vuelve al punto del cliente a no ser que el cliente en el que se llamó al método “capture” la excepción
 - el efecto en el método *llamante*
 - dependerá de si se ha escrito o no código para capturar (catch) la excepción.



Qué significa tratar una excepción?

- **tratar una excepción**
 - **capturarla**
 - proporcionar código para manejarla con **try ... catch** o
 - **propagarla**
 - se “lanza” la pelota al método que llamó al que provocó la excepción

Situación habitual en las excepciones verificadas

Método llamante – cliente

```
public void metodo1()  
{  
    try  
    {  
        metodo2();  
    }  
    catch (IOException e)  
    {  
        ....  
    }  
}
```

Captura la excepción o propaga.
Si hace esto último

public void metodo1() throws IOException

Método llamado – servidor

```
public void metodo2() throws IOException  
{  
    .....  
    throw new IOException();  
    .....  
}
```

avisa

aquí acaba metodo2() – no hay valor de retorno si lo tuviese y se lanza la excepción

Capturar una excepción try ... catch

Proporcionar un gestor de excepción (**exception handler**)

```
try
{
    //código a proteger
}
catch (Exception e)
{
    ....
}
```

El bloque catch() es el que captura la excepción y en él se indica el tipo de excepción que captura y la variable que referencia al objeto excepción que se crea

```
try
{
    //código a proteger
}
catch (Exception e)
{
    ....
}
finally
{
    //código a ejecutar siempre
    // haya o no excepción
}
```

opcional, cerrar ficheros, liberar recursos, ...

Capturar una excepción try ... catch

```
try
{
    int[] array = new int[1];
    array[0] = 10;
    array[1] = 20; // error
    System.out.println("Esto no se imprimirá si hay error");
}
catch (ArrayIndexOutOfBoundsException e)
{
    System.out.println("Error");
}
System.out.println("Fin");
```

Resultado

Error
Fin

Capturar una excepción try ... catch

```
try
{
    int[] array = new int[1];
    array[0] = 10;
    array[1] = 20; // error
    System.out.println("Esto no se imprimirá si hay error");
}
catch (ArrayIndexOutOfBoundsException e)
{
    System.out.println("Error");
}
finally
{
    System.out.println("en finally");
}
System.out.println("Fin");
```

Resultado

Error
en finally
Fin

Capturar una excepción try ... catch

```
try
{
    int[] array = new int[1];
    throw new IllegalArgumentException();

}
catch (ArrayIndexOutOfBoundsException e)
{
    System.out.println("Error");
}

System.out.println("Fin");
```

Resultado

java.lang.IllegalArgumentException

Capturar una excepción try ... catch

```
try
{
    int[] array = new int[1];
    throw new IllegalArgumentException();

}
catch (ArrayIndexOutOfBoundsException e)
{
    System.out.println("Error");
}
finally
{
    System.out.println("en finally");
}
System.out.println("Fin");
```

Resultado
en finally
java.lang.IllegalArgumentException

Lanzar y capturar múltiples excepciones

- Un método puede lanzar muchas excepciones
 - `public void procesar() throws EOFException, FileNotFoundException`
- Si son verificadas hay que indicar todas en la cabecera del método que las puede lanzar con **throws** y separadas por ,
- El gestor de excepciones debe proporcionar código para manejar todas las posibles excepciones

Capturar múltiples excepciones

```
try
{
    String str = "2m";
    int n = Integer.parseInt(str);
    System.out.println("Resultado " + n / 0);
}
catch (NumberFormatException e)
{
    System.out.println("Error, formato incorrecto");
}
catch (ArithmeticException e)
{
    System.out.println("Error, división entre 0");
}
System.out.println("Fin");
```

Lanzar y capturar múltiples excepciones

```
try
{
    .....
    obj.procesar();
    .....
}

catch (EOFException e)
{
}

catch (FileNotFoundException e)
{
}
```

- Los bloques catch() se analizan en el orden en que se han escrito hasta encontrar uno que concuerde con la excepción lanzada
- varios bloques catch() importa el orden en que se escriben en una sentencia try. Un bloque catch() para un tipo concreto de excepción no puede seguir a un bloque catch() para una excepción de su supertipo

Lanzar y capturar múltiples excepciones

```
try
{
    .....
    Persona p = bd.buscar(nombrePersona);
    .....
}

catch (Exception e)
{

}

catch (RuntimeException e)
{

}
```

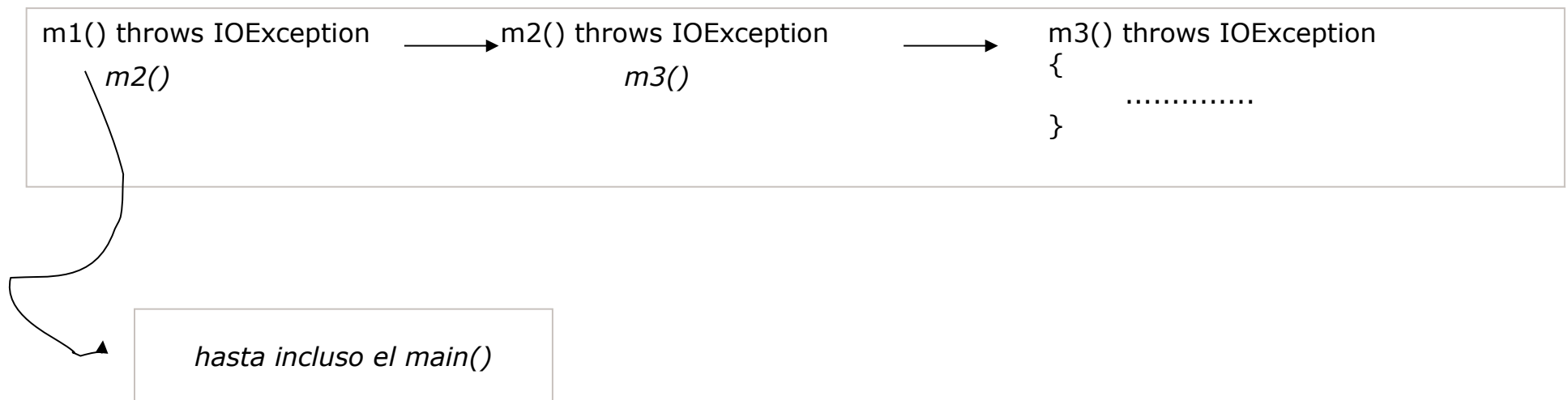
- Incorrecto porque Exception es un supertipo de RuntimeException.
- Hay que comenzar siempre los bloques catch() por las excepciones más específicas.

Propagar una excepción

- Java permite que una excepción sea propagada desde el método receptor de la excepción (en principio el que debería manejarla) hacia atrás, hacia el método que lo llamó e incluso más atrás
- Un método propaga una excepción no incluyendo un gestor (`try ... catch()`) que proteja las sentencias que pueden ocasionar la excepción
- La cadena de propagación puede llegar incluso hasta el método `main()`
- La propagación es habitual cuando el método llamante no sabe cómo tratar la excepción

Propagar una excepción

- Para las excepciones verificadas
 - Java obliga a que el método que las propaga incluya una cláusula throws en su cabecera, incluso si él mismo lanza la excepción
- Para las excepciones no verificadas
 - la cláusula throws es opcional (nosotros la omitiremos)



Algunos métodos de Throwable

- **getMessage()**
 - obtiene el String de mensaje asociado a la excepción
- **printStackTrace()**
 - muestra la secuencia de llamadas a métodos que han conducido a la excepción (en orden inverso)
 - es lo que vemos en la pantalla cuando la JVM captura la excepción (útil para tareas de depuración)

```
java.lang.NumberFormatException: For input string: "rr"  
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)  
    at java.lang.Integer.parseInt(Integer.java:580)  
    at java.lang.Integer.parseInt(Integer.java:615)  
    at Ejemplo01.prueba2(Ejemplo01.java:34)
```

Definiendo nuevas clases excepción

- Las nuevas clases para excepciones verificadas pueden ser definidas como subclases de cualquier excepción verificada, por ejemplo, `Exception`
- Las nuevas excepciones no verificadas pueden extender de `RuntimeException`

```
public class FormatoExcepcion extends IOException
{
    private String valorErroneo;
    public FormatoExcepcion(String valorErroneo)
    {
        this.valorErroneo = valorErroneo;
    }
    public String toString()
    {
        return "Se ha introducido un valor incorrecto "
            + valorErroneo;
    }
}
```

Mejoras a partir de Java 7

- Podemos capturar varias excepciones en un gestor con una única cláusula **catch**

```
try
{
    if (strNum.length() > 5)
    {
        throw new IllegalArgumentException();
    }
    int n = Integer.parseInt(strNum);
    .....
}
catch (NumberFormatException | IllegalArgumentException e)
{
    e.printStackTrace();
}
```

Proyecto Hora

- **Clase Hora -**
 - constructor con tres parámetros (h, m, s)
 - El constructor indica que en la creación se puede lanzar una excepción de tipo `IllegalArgumentException`.
 - Incluye un `toString()`
 - Si h no correcto (< 0 o > 59) lanzar la excepción
 - Idem para m y s.
- En una clase **DemoHora**
 - incluye un método **demo()** que capture la posible excepción al crear una o varias horas
 - No captures y comprueba que no hay error del compilador.
 - Cambia a `IOException` en `Hora` sin capturar y mira el mensaje del compilador.
 - Crea una excepción propia `HoraExcepcion` que hereda de `RuntimeException` con constructor, atributo y `toString()`.
 - Ahora que herede de `Exception`

IO en Java

I/O en Java

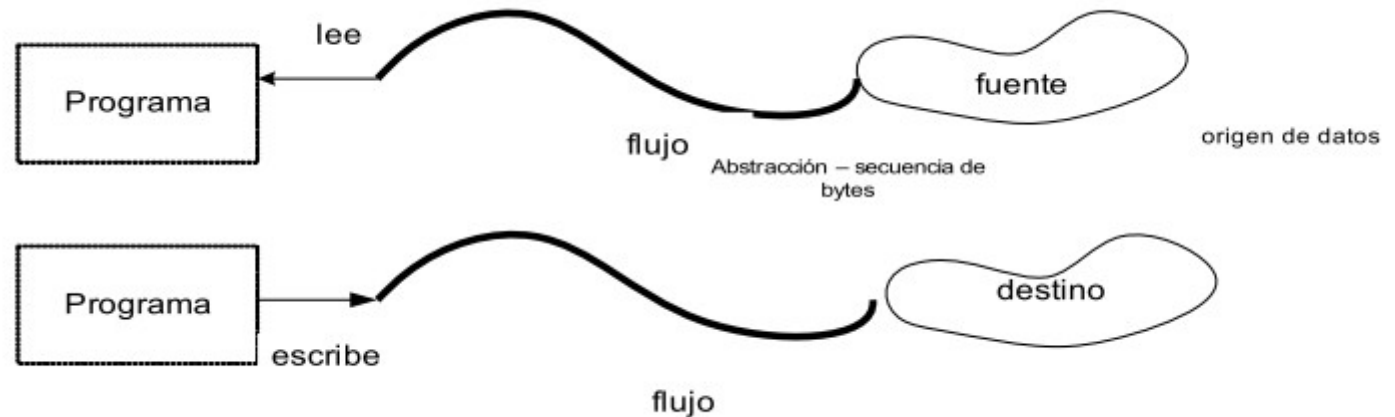
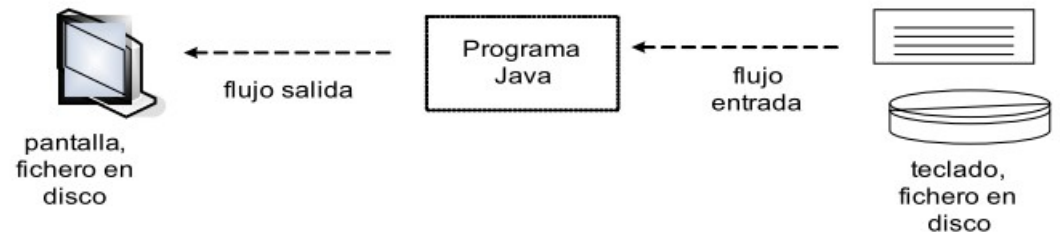
- Todos los programas necesitan realizar operaciones de E/S (Input/Output)
- Salida de datos
 - envío texto a la consola (`System.out.println()`)
 - salida gráfica (`JOptionPane` y otros)
 - **envío de datos a un fichero de disco**
 - envío de datos a través de la red, ...
- Entrada de datos
 - lectura de texto desde teclado (`Scanner`)
 - **lectura de datos de un fichero de disco**
 - obtención de datos desde otros programas, a través de la red, ...
 - entrada a través de una GUI (click en un botón, selección de un menú, ...)

I/O en Java. Concepto de flujo o stream.

- paquete *java.io*

- **Flujo – stream**

- corriente de datos (bytes que van a un destino o vienen de una fuente)
- la conexión entre un programa y una fuente de datos o un destino



los flujos se conectan a múltiples fuentes y destinos

I/O en Java. Concepto de flujo o stream.

En escritura:

```
abrir el flujo (puede asociarse a un fichero)
while (haya información)
{
    escribir información en el flujo
}
cerrar el flujo
```

En lectura:

```
abrir el flujo (puede asociarse a un fichero)
while (haya información)
{
    leer información del flujo
}
cerrar el flujo
```

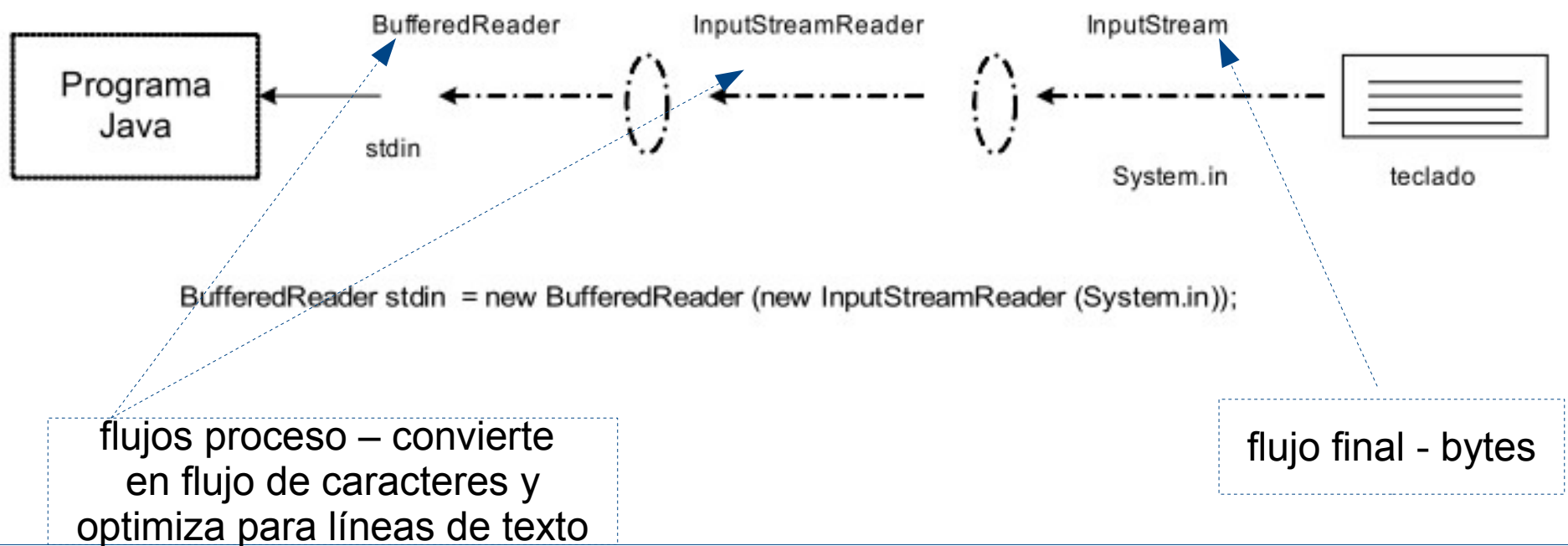
tratamiento habitual de
los flujos

Tipos de flujos

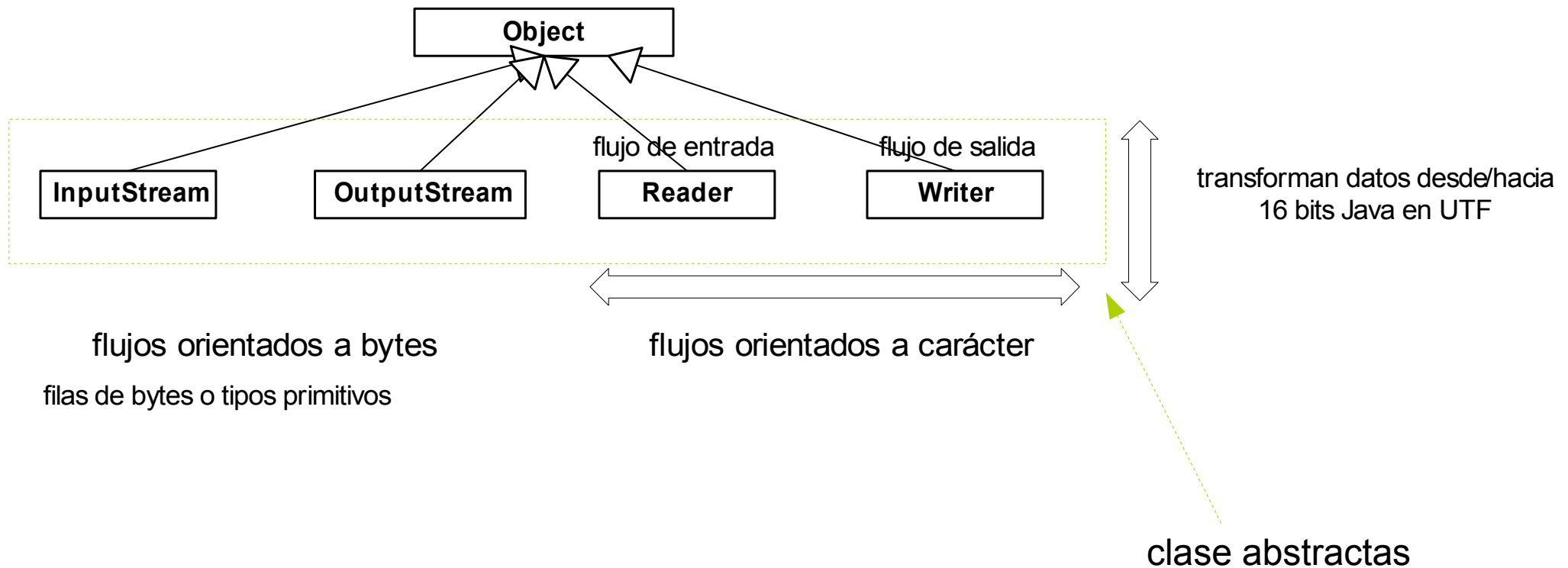
- **Flujos de entrada / de salida**
 - de **entrada** – se lee de ellos (InputStream,)
 - de **salida** – se escribe en ellos (OutputStream,)
- **Flujos de bytes / de caracteres**
 - de **bytes** – flujos binarios (stream)
 - leer / escribir bytes
 - se utiliza en serialización (escritura de objetos en disco)
 - datos binarios (.exe, .jpg, .wav)
 - de **caracteres** – readers / writers (los que veremos)
 - leer / escribir caracteres
 - texto en formato legible
 - posteriores a los binarios

Tipos de flujos

- **Flujos de proceso / finales**
 - de **proceso** – filtros, clase envolventes, decoradores
 - operan sobre datos proporcionados por otros flujos
 - **finales** – sink – ej. FileReader / FileWriter
 - más cercanos a la fuente física de datos



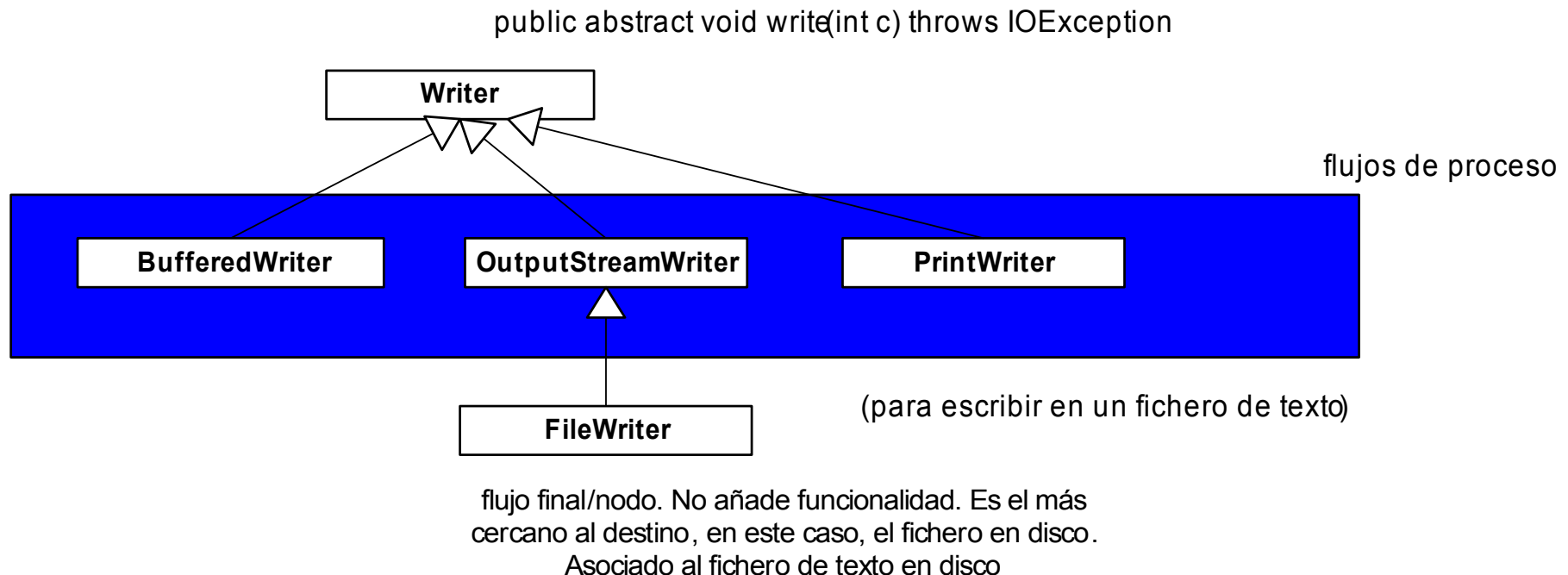
Jerarquía de clases



Writers

■ **Writer**

- clase abstracta
- de ella derivan todos los flujos de salida orientados a carácter.
- esos flujos reciben datos de 16 bits de un programa y los envían a su destino como puede ser un fichero de texto en formato UTF8



La clase `FileWriter`

■ **FileWriter**

- flujo final asociado a un fichero en disco
 - para escribir caracteres en un fichero de texto en disco
- `FileWriter (String nombre)`
 - crea un nuevo fichero en el disco con el nombre indicado
 - si existe ya un fichero con ese nombre será reemplazado con el nuevo contenido
- `FileWriter (String nombre, boolean append)`
 - si `append` es `true` se abre un fichero que ya existe sin destruir su contenido
 - si el fichero no existe se crea

La clase `FileWriter`

■ **FileWriter**

- `public void write(String str)` throws `IOException`
- `close()` cierra el fichero
 - si un fichero no se cierra pueden perderse sus datos
- La mayoría de los métodos de IO lanzan una `IOException` si ocurre un error que hay que tratar en el bloque `try ... catch`

```
try
{
    FileWriter f = new FileWriter(nombre);
    f.write("Creando una linea de texto\r\n");
    f.write("otra linea mas\n");
    f.close();
}
catch (IOException e)
{
    System.out.println("Error al crear " + nombre);
}
```

La clase `BufferedWriter` (envolvente)

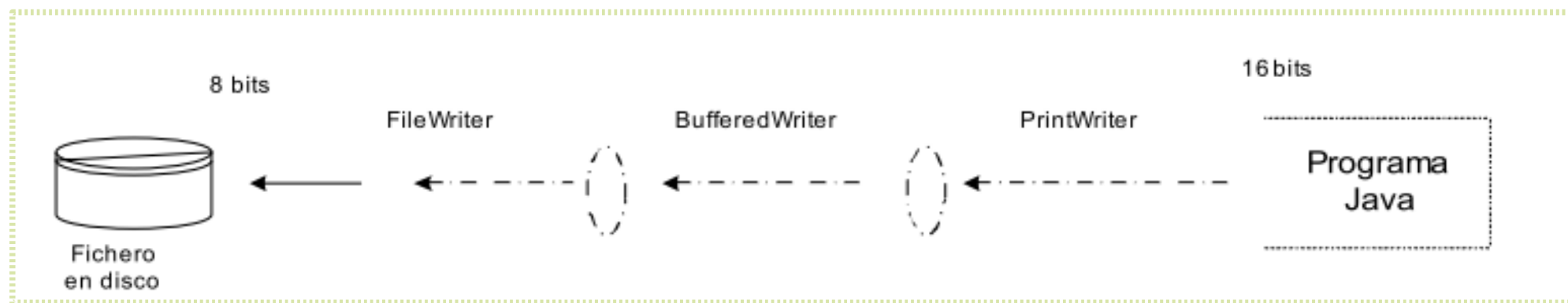
- **BufferedWriter**

- mejora la eficiencia en escritura utilizando un buffer para reducir las escrituras a disco
- `public BufferedWriter(Writer salida)`
 - `BufferedWriter salida = new BufferedWriter(new FileWriter("fichero.txt"));`
- `newLine()`
 - para escribir un salto de línea

La clase `PrintWriter` (envolvente)

■ `PrintWriter`

- para escribir texto formateado en disco
- proporciona métodos convenientes para visualizar los diferentes tipos de datos Java en un formato legible
- gestiona directamente el fin de línea
- podemos escribir con `print()`, `println()` y `printf()` tal como hacíamos hasta ahora en el terminal de texto
- ninguno de sus métodos lanza excepciones



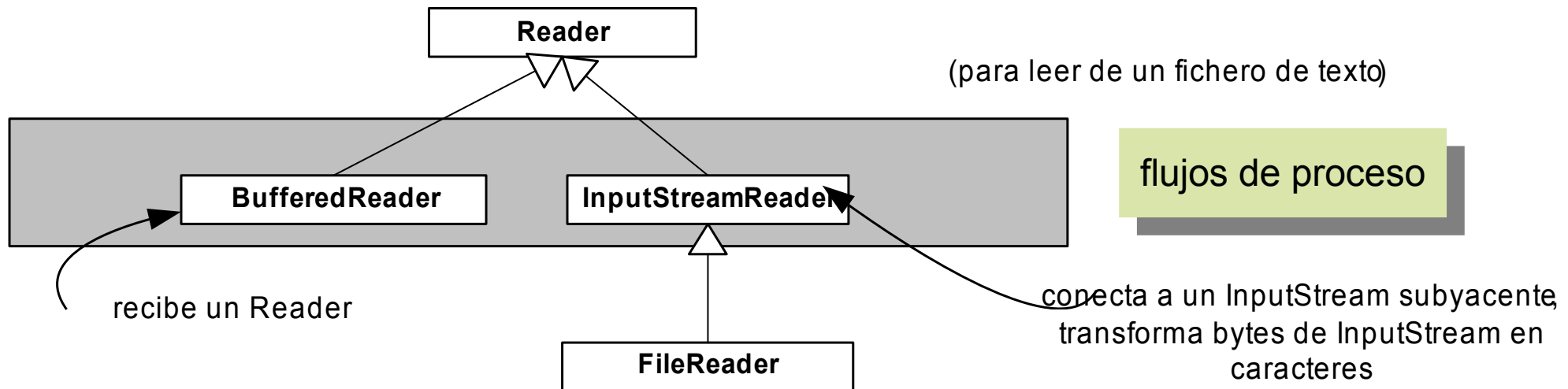
La clase PrintWriter - ejemplo

```
public void ejemploPrintWriter(String nombre)
{
    PrintWriter fsalida = null;
    try
    {
        fsalida = new PrintWriter(new BufferedWriter(new FileWriter(nombre)));
        fsalida.println("Tabla del 4");
        for (int i = 1; i <= 10; i++)
            fsalida.println(i + "*" + 4 + "\t" + i * 4);
        fsalida.close();
    }
    catch (IOException e)
    {
        System.out.println("Error al crear " + nombre);
    }
}
```

Readers

■ Reader

- clase abstracta de la que derivan todos los flujos de entrada orientados a carácter
- envía datos de 16 bits a un programa aunque la fuente tenga formato diferente

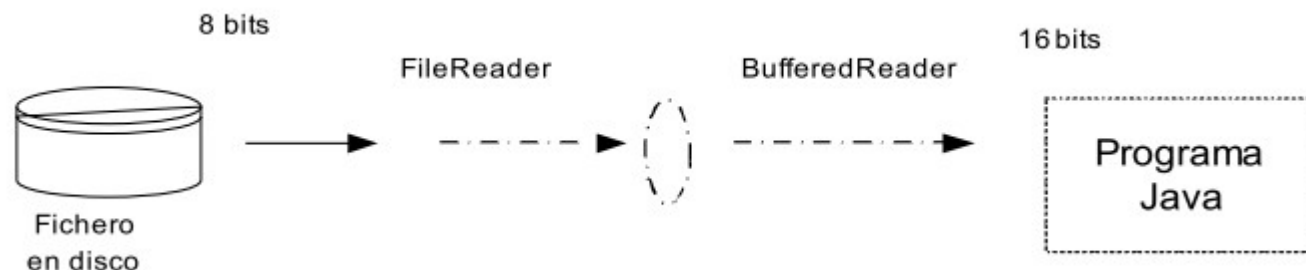


La clase FileReader

■ FileReader

- para leer caracteres desde un fichero de disco
- no permite leer una línea de caracteres por eso se utiliza como clase envolvente BufferedReader

```
FileReader fr = new FileReader();  
int car ;  
while (car = fr.read() != -1)  
    if ((char) car == c)  
        contador++,  
fr.close();
```



La clase `BufferedReader`

- **`BufferedReader`**

- clase envolvente
- mejora la eficiencia en la lectura utilizando un buffer
- método `readLine()`
 - lee una línea de texto del flujo de caracteres y devuelve un `String`
 - si no hay más datos en el fichero devuelve `null`.

La clase `BufferedReader`

```
public void ejemploReader(String nombre)
{
    String linea;
    try
    {
        BufferedReader entrada = new BufferedReader(new
                                                    FileReader(nombre));

        linea = entrada.readLine();
        while (linea != null)
        {
            System.out.println(linea);
            linea = entrada.readLine();
        }
        entrada.close();
    }
    catch (IOException e)
    {
        System.out.println("Error al leer " + nombre);
    }
}
```

La clase File

- Encapsula (abstrae) las propiedades de un fichero o directorio independientemente del sistema
- Proporciona acceso a información acerca de ficheros y directorios
- Para trabajar con el sistema de ficheros de una máquina local
- Muchos constructores reciben un objeto File
 - `FileReader(File fichero)`
 - `FileWriter(File fichero)`
- Detecta si un fichero existe o no
- Borra un fichero cuyo nombre se proporciona
- No vale para abrir/cerrar/leer/escribir en un fichero

La clase File

- `File f = new File("c:\\windows\\system\\imagen.gif");`
 - en Windows, hay que poner `\\` como carácter de escape
 - si no se pone ruta absoluta se asume directorio actual
- `File f = new File("c:/windows/system/imagen.gif");`
 - Java permite esta forma también para Windows (en realidad cualquier tipo de `/`) y lo traduce adecuadamente según la plataforma
- `File f = new File("/tmp/ejemplos");` // en Linux

La clase File

```
public void demoFile(String nombre) throws IOException
{
    File fichero = new File(nombre);
    if (fichero.exists())
    {
        System.out.println("Nombre: " + fichero.getName());
        System.out.println("\nRuta completa: " + fichero.getAbsolutePath());
        System.out.println("\nTamaño: " + fichero.length() + " bytes");
        if (fichero.isFile())
            System.out.println("\nFichero normal");
        else
            if (fichero.isDirectory())
                mostrarContenidoDirectorio(fichero);
    }
    else
        throw new IOException("El fichero " + nombre +
                               " no existe");
}
```

La clase File

```
private void mostrarContenidoDirectorio(File directorio)
{
    String[] ficheros = directorio.list();
    for (int i = 0; i < ficheros.length; i++)
        System.out.println("\t" + ficheros[i]);
}
```

La clase Scanner

- en **java.util**
- a partir de Java 5
 - para hacer más fácil la lectura de los tipos de datos básicos desde una fuente de caracteres
- el constructor de la clase especifica la fuente de caracteres desde la que leerá Scanner
 - puede ser un Reader, un InputStream (flujo binario), un String o un File
- actúa como una clase envolvente (wrapper) para la entrada
- métodos para
 - leer enteros, cadenas, ... **nextInt()** **nextLine()** **next()**
- detecta si queda algo por leer en el flujo que recibe
 - **hasNextInt()** **hasNextLine()** **hasNext()**

La clase Scanner - ejemplos

- `Scanner teclado = new Scanner(System.in);`
`//System.in es un flujo InputStream`
- `Scanner fichero = new Scanner(new File("texto.txt"));`

```
Scanner cs = new Scanner(new File("numeros.txt"));
int suma = 0;
while (sc.hasNextInt())
    suma += sc.nextInt();
```

```
Scanner cs = new Scanner("12,34,13").useDelimiter(",");
int suma = 0;
while (sc.hasNextInt())
    suma += sc.nextInt();
```

System.in / System.out

- System.in System.out
 - flujos binarios standard definidos en System
 - se crean automáticamente

```
public class System
{
    .....
    public static final InputStream in;
    public static final PrintStream out;
    .....
}
```

- `BufferedReader entrada = new BufferedReader(new
InputStreamReader(System.in));`
 - así se hacía antes de Java 5 la lectura de datos desde teclado

System.in recibe lo que se introduce desde teclado, al pulsar Intro la secuencia de bytes (byte stream) tecleados se convierten en flujo de caracteres pasándolos al constructor de InputStreamReader. Este objeto a su vez es pasado al constructor de BufferedReader que con ayuda del método `readLine()` devuelve como String lo que se tecleó.

Resumen de flujos a utilizar

- Para **escribir** en un fichero de texto
 - `File f = new File(nombre);`
 - `PrintWriter salida =
new PrintWriter(new BufferedWriter(new FileWriter(f)));`
- Para **escribir** en un fichero de texto añadiendo
 - `File f = new File(nombre);`
 - `PrintWriter salida =
new PrintWriter(new BufferedWriter(new FileWriter(f, true)));`

Resumen de flujos a utilizar

- Para **leer** desde un fichero de texto con **BufferedReader**
 - `File f = new File(nombre);`
 - `BufferedReader entrada = new BufferedReader(new FileReader(f));`
- Para **leer** desde un fichero de texto con **Scanner**
 - `File f = new File(nombre);`
 - `Scanner entrada = new Scanner(f);`

Serialización

- **Serialización** (persistencia de los objetos)
 - almacenar (hacer persistente) un objeto completo y sus objetos dependientes en un flujo
 - en la serialización un objeto se transforma en una secuencia de bytes y
 - puede reconstruirse posteriormente a partir de la misma
- Con una simple llamada a un método
 - se guarda un objeto entero sin tener que partirlo en sus atributos
 - se recompone un objeto completo sin hacerlo a partir de sus atributos
- Usos
 - copia de objetos
 - persistencia
 - enviar objetos a través de la red

Serialización

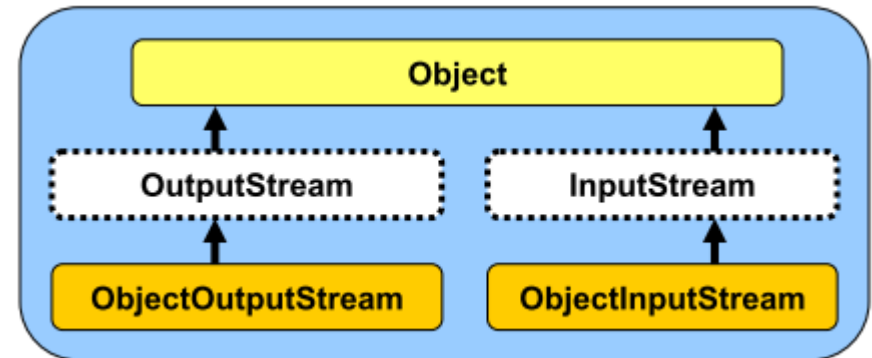
- Flujos que intervienen

- ObjectOutputStream

- Método – `writeObject()`
- Ejemplo: `flujoSalida.writeObject (objetoClase);`

- ObjectInputStream

- Método - `readObject()`
- Ejemplo: `objetoClase = (Clase) flujoEntrada.readObject();`



Interface Serializable

- La clase que desee serializar sus objetos debe implementar la interfaz **Serializable**
 - interfaz sin métodos (*tagging* interfaz)
 - `import java.io.Serializable;`
`public class Empleado implements Serializable`
- en esta implementación el objeto define cómo debe almacenarse o recuperarse de un fichero con los métodos
 - `writeObject`: responsable de escribir el estado del objeto en el flujo
 - `readObject`: responsable de recuperar el estado del objeto desde el flujo
- Si se intenta serializar un objeto que no lo implementa se obtiene la excepción `NotSerializableException`

Serializando objetos - Ejemplo

```
private Empleado[] empleados;
.....
public void salvar()
{
    try
    {
        ObjectOutputStream salida =
            new ObjectOutputStream(new
                FileOutputStream("personal.txt"));
        salida.writeObject(empleados);
        salida.close();
    }
    catch(IOException e)
    {
        e.printStackTrace(System.err);
    }
}
```

Deserializando objetos - Ejemplo

```
public void leer() throws IOException, ClassNotFoundException
{
    try
    {
        ObjectInputStream entrada =
            new ObjectInputStream(new
                FileInputStream("personal.txt"));
        empleados = ( Empleado[]) entrada.readObject();
        entrada.close();
    }
    catch( IOException e)
    {
        e.printStackTrace(System.err);
    }
}
```