INICIACIÓN A LA PROGRAMACIÓN
LENGUAJE JAVA con BlueJ

Tema 3 *Clases y Objetos*
Tema 4 *Comunicación entre objetos. Algoritmos*
Tema 5 *Herencia y abstracción de datos*
Tema 6 *Diseño de clases*

TEMA 6 : Diseño de clases

Diseño de clases

- Acoplamiento
- Cohesión
- Diseño dirigido por responsabilidades
- Refactorización

Conceptos sobre ficheros

- Ficheros en Java
- Ficheros binarios
- Ficheros de texto

TEMA 6 : Diseño de clases

Necesidad de un buen diseño

- Objetivo de un buen diseño: minimizar tiempo de programación y errores
  - Mantenimiento del código
  - Modificaciones futuras
    - Añadir nuevas funcionalidades
    - Adapatación a nuevas situaciones
  - Reusabilidad
- Un buen diseño
  - Requiere esfuerzo
  - Recompensa a largo plazo

Síntomas indicadores de un mal diseño

- Rigidez – las clases son difíciles de cambiar
- Fragilidad – al introducir cambios las clases dejan de funcionar
- Viscosidad – las clases son difíciles de utilizar correctamente
- Inmovilidad – las clases son difíciles de reutilizar
- Complejidad innecesaria
- Repetición innecesaria – abuso de copiar y pegar
- Opacidad – aparente desorganizacion

Principios para eliminar estos síntomas

- Acoplamiento bajo
- Cohesión alta
- Diseño dirigido a responsabilidades
- Refactorización

TEMA 6 : Diseño de clases
Acoplamiento

- Medida de la conectividad entre clases
- Determina el grado de dificultad en la realización de modificaciones
  - Clases fuertemente acopladas
    - Un cambio en una clase se ha de propagar al resto
    - Al introducir cambios en una clase las clases acopladas dejan de funcionar
    - La introducción de cambios consume tiempo
    - Aumenta el peligro de cometer errores
  - Clases débilmente acopladas (deseable)
    - Cada clase es altamente independiente del resto
    - Un cambio en una clase no se ha de propagar al resto
    - Al introducir cambios en una clase el resto de clases siguen funcionando
    - La introducción de cambios consume poco tiempo
- La comunicación entre clases débilmente acopladas se realiza mediante una interfaz

TEMA 6 : Diseño de clases
Ejemplo de acoplamiento (I)

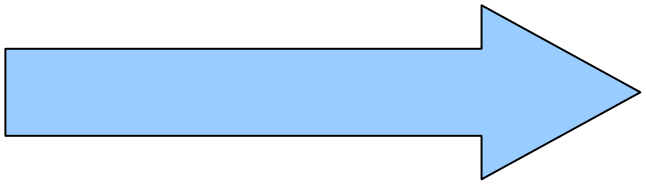
- Solución acoplada al problema de rotación de puntos en el plano

```
public class Punto {
  private double x;
  private double y;
    . . . .

  public Punto rotar( double[][] R ) // R, matriz de rotación
  {
     Punto pr = new Punto(); // Punto rotado

     pr.x = R[0][0] * x + R[0][1] * y;
     pr.y = R[1][0] * x + R[1][1] * y;

     return pr;
  }
 }
```

TEMA 6 : Diseño de clases
Ejemplo de acoplamiento (II)

- Si modificamos la representación de los campos en la clase Punto tenemos que modificar las instrucciones del método Rotar.

```
public class Punto {
  private double x;
  private double y;
    . . . .
 }
```



```
public class Punto {
  private double[] x;
    . . . .
 }
```

```
public class Punto {

  private double x;
  private double y;
  . . . .

  public Punto Rotar()
  {
     Punto pr = new Punto(); // Punto rotado

     pr.x = R[0][0] * x + R[0][1] * y;
     pr.y = R[1][0] * x + R[1][1] * y;

     return pr;
  }
 }
```

```
private double[]x;
```

```
pr.x[0] = R[0][0] * x[0] + R[0][1] * x[1];
pr.x[1] = R[1][0] * x[0] + R[1][1] * x[1];
```

TEMA 6 : Diseño de clases
Ejemplo de acoplamiento (III)

- Solución desacoplada al problema de rotación de puntos en el plano mediante **encapsulación**. Aunque modifiquemos la representación de los campos de la clase Punto el método Rotar sigue siendo correcto.

```
public class Punto {
  private double x;
  private double y;
  public double getX()
  {    return x;     }

  public void setX( double nx
)
  {    x = nx;     }
    . . . .
 }
```

```
public class Punto {
  private double[] x;

  public double getX()
  {  return x[0];    }

  public void setX( double nx )
  {  x[0] = nx;    }
    . . . .
 }
```

```
public class Punto {

  . . . .

  public Punto rotar( double[][] R )
  {
     Punto pr = new Punto(); // Punto rotado

     pr.setX( R[0][0] * getX() + R[0][1] * getY() );
     pr.setY( R[1][0] * getX() + R[1][1] * getY() );

     return pr;
  }
 }
```

TEMA 6 : Diseño de clases
La encapsulación reduce el acoplamiento

- Encapsulación de datos (concepto introducido en Tema 5)
- Objetivos de la encapsulación
  - Mostrar información de la **interfaz** – qué hace una clase
  - Ocultar información de la **implementación** – cómo hace una clase
- Cada clase ha de ser responsable de manejar sus propios datos
  - La definición de los campos es **privada**
  - El acceso y la modificación de los campos desde otras clases se realiza mediante métodos get y set

TEMA 6 : Diseño de clases
Acoplamiento implícito

- Una clase depende de la información interna de otra pero la dependencia no es obvia
- Difícil de detectar

TEMA 6 : Diseño de clases
Cohesión

- Medida de la coherencia de las clases y su contenido
- Determina el número y diversidad de tareas de las que son responsables
  - Clases
  - Métodos
- Cohesión alta deseable
  - Clases responsables de tareas coherentes entre sí
  - Métodos responsables de una sóla tarea lógica
- Reusabilidad
  - Clases y métodos coherentes pueden ser reutilizados en contextos diferentes

TEMA 6 : Diseño de clases
Ejemplo de cohesión (I)

- Solución con mala cohesión al problema de rotación de puntos en el plano

```
public class RotacionPunto {
  private double x;       // Coordenada x del punto a
  private double y;       // Coordenada y del punto a
  private double[][] R;  // Matriz de rotación

  . . . .

  public void rotar()
  {
     double xr; // Coordenada x del punto rotado
     double yr; // Coordenada y del punto rotado

     xr = R[0][0] * x + R[0][1] * y;
     yr = R[1][0] * x + R[1][1] * y;

     System.out.println( “El punto rotado es (“, x , “,” , y , “)”);
  }
 }
```

La clase RotacionPunto desempeña dos tareas:
1. Representar las coordenadas de un punto del plano
2. Calcular la rotación de un punto del plano

El método Rotar desempeña dos tareas:
1. Calcular el punto rotado
2. Mostrar el mensaje por pantalla

Además no podemos devolver el punto rotado para que pueda ser utilizado en otros métodos

TEMA 6 : Diseño de clases
Ejemplo de cohesión (II)

- Solución con buena cohesión al problema de rotación de puntos en el plano

```
public class Punto {
  private double x;
  private double y;
    . . . .

  public void mostrarPunto()
  {
    System.out.println("(", x , "," , y , ")");
  }
}
```

```
public class RotacionPunto {
  private Punto p;        // Punto a rotar
  private double[][] R;   // Matriz de rotación

  . . . .

  public Punto rotar()
  {
     Punto pr= new Punto();
     pr.setX( R[0][0] * p.getX() + R[0][1] * p.getY() );
     pr.setY( R[1][0] * p.getX() + R[1][1] * p.getY() );
     return pr;
  }
 }
```

La clase Punto desempeña tareas relacionadas con la representación de puntos:
1. Representar las coordenadas de un punto del plano
2. Mostrar las coordenadas

La clase RotacionPunto desempeña tareas relacionadas con la rotación de puntos:
Calcular la rotación de un punto del plano

TEMA 6 : Diseño de clases
Ejemplo de cohesión (III)

- Solución con buena cohesión al problema de rotación de puntos en el plano

```
public class Punto {
  private double x;
  private double y;
    . . . .

  public void mostrarPunto()
  {
    System.out.println("(", x , "," , y , ")");
  }
}
```

El método Rotar desempeña la tarea de calcular el punto rotado

La clase Punto permite devolver el punto rotado para que pueda ser utilizado por otros métodos

La clase Punto puede ser reutilizada por otras clases, por ejemplo, TraslacionPunto

```
public class RotacionPunto {
  private Punto p;        // Punto a rotar
  private double[][] R;  // Matriz de rotación

  . . . .

  public Punto rotar()
  {
     Punto pr= new Punto();
     pr.setX( R[0][0] * p.getX() + R[0][1] * p.getY() );
      pr.setY( R[1][0] * p.getX() + R[1][1] * p.getY() );
     return pr;
  }


 }
```

TEMA 6 : Diseño de clases
Duplicación de código

- Indicador de mala cohesión
- Diseño de mala calidad
- Incrementa el trabajo a la hora de realizar cambios
- Aumenta el riesgo de errores
  - Cambiar el código duplicado en un sitio si y otro no

TEMA 6 : Diseño de clases

Diseño dirigido por responsabilidades

- Sigue el principio de responsabilidad única

  Tom DeMarco 1979 "Structured Analysis and System Specification"

  "Cada clase ha de responsabilizarse de una sola tarea lógica"

- Seguir el principio de responsabilidad única reduce el acoplamiento

TEMA 6 : Diseño de clases

Ejemplo de diseño orientado a responsabilidades

- En el problema de rotación de puntos en el plano la clase punto debería ser responsable de su rotación

```
public class Punto {
  private double x;
  private double y;
    . . . .

  public void mostrarPunto()
  {
    System.out.println("(", x , "," , y , ")");
  }

  public Punto rotar( double[][] R )
  {
     Punto pr= new Punto();
     pr.setX( R[0][0] * p.getX() + R[0][1] * p.getY() );
      pr.setY( R[1][0] * p.getX() + R[1][1] * p.getY() );
     return pr;
  }

}
```

TEMA 6 : Diseño de clases
Refactorización

- Reestructuración de un código existente que altera su estructura interna sin cambiar su comportamiento externo
  - Persigue mejorar el código en algún aspecto (eficiencia, reusabilidad, extensión...)
- Supone repensar y rediseñar la estructura de clases y métodos
  - División o unión de clases
  - División o unión de métodos

TEMA 6 : Diseño de clases
Ejemplos de refactorización (I)

- Si el mismo fragmento de código está en todas las ramas de una expresión condicional se puede mover fuera de la expresión

```
. . . .

if( x % 2 == 0 )
{
   y = y + 1;
   System.out.println( y );
}
else
{
   y = y – 1;
   System.out.println( y );
}

. . . .
```

Código sin refactorizar

```
. . . .

if( x % 2 == 0 )
{
   y = y + 1;
}
else
{
   y = y – 1;
}

System.out.println( y );

. . . .
```

Código refactorizado

TEMA 6 : Diseño de clases
Ejemplos de refactorización (II)

- Si tienes una expresión condicional complicada, convierte la condición en un método booleano

```
. . . .

if(
(anio % 4 == 0 &&
 anio % 100 != 0) ||
 anio mod 400 == 0 )
  System.out.println( “Bisiesto” );
else
  System.out.println( “No bisiesto” );

. . . .
```

Código sin refactorizar

```
. . . .

boolean esBisiesto( int anio )
{
  return anio % 4 == 0 &&
         anio % 100 != 0) ||
         anio mod 400 == 0;
}

. . . .

if( esBisiesto( anio ) )
  System.out.println( “Bisiesto” );
else
  System.out.println( “No bisiesto” );

. . . .
```

Código refactorizado

TEMA 6 : Diseño de clases
Ejemplos de refactorización (III)

- Si tienes un bucle que realiza dos tareas, dividelo en dos

```
. . . .

double mediaEdad = 0.0;
double totalSalario = 0.0;

for( int i = 0; i < empleados.longitud; i++ )
{
  mediaEdad += empleados[i].edad;
  totalSalario += empleados[i].salario;
}

mediaEdad /= empleados.longitud;

System.out.println( mediaEdad );
System.out.println( totalSalario );
. . . .
```

Código sin refactorizar

```
. . . .

double mediaEdad = 0.0;
double totalSalario = 0.0;

for( int i = 0; i < empleados.longitud; i++ )
    mediaEdad += empleados[i].edad;

for( int i = 0; i < empleados.longitud; i++ )
    totalSalario += empleados[i].salario;

mediaEdad /= empleados.longitud;

System.out.println( mediaEdad );
System.out.println( totalSalario );
. . . .
```

Código refactorizado

Ejemplos de refactorización (IV)

- Si tienes una variable local con un alcance mayor a dónde se utiliza, reduce su alcance para que sea sólo visible donde es realmente utilizada

```
. . . .

int x = 0;

// x no se utiliza aqui

if( condicion )
{
  // x se utiliza aqui
}

// x no se utiliza aqui
. . . .
```

Código sin refactorizar

```
. . . .

if( condicion )
{
  int x = 0;

  // x se utiliza aqui
}

. . . .
```

Código refactorizado

TEMA 6 : Diseño de clases
Ejemplos de refactorización (V)

- Si tienes un conjunto de datos que van juntos de manera natural, reemplázalos por un objeto

```
 . . . .

private int numerador;
private int denominador;

 . . . .
```

Código sin refactorizar

```
public class Fraccion
 {
   private int numerador;
   private int denominador;
 . . . .
 }
```

Código refactorizado

TEMA 6 : Diseño de clases
Ejemplos de refactorización (VI)

- Si tienes dos clases con características similares, crea una superclase y mueve los campos comunes a ella

```
public class SemaforoAutomoviles
{
  private char color;
  private double tVerde;
  private double tAmbar;
  private double tRojo;
. . . .
  public void cambiarColor( char nc )
  . . . .
}
public class SemaforoPeatones
{
  private char color;
  private double tVerde;
  private double tRojo;
. . . .
  public void cambiarColor( char nc )
  . . . .
}
```

Código sin refactorizar

```
public class Semaforo
{
  private char color;
  private double tVerde;
  private double tRojo;
. . . .
  public void cambiarColor( char nc )
  . . . .
}
public class SemaforoAutomoviles extends Semaforo
{
  private double tAmbar;
. . . .
}
public class SemaforoPeatones extends Semaforo
{
. . . .
}
```

Código refactorizado

TEMA 6 : Diseño de clases
Ejemplos de refactorización (VII)

- Si tienes constructores en subclases con cuerpos casi idénticos, crea un constructor de superclase y llámalo desde las subclases

```
public class Empleado
{
  private String nombre;
  private String id;

  public Empleado() {}

  public Empleado( String nnombre, String nid )
  {
     nombre = nnombre;
     id = nid;
  }
}
```

```
public class Manager extends Empleado
{
  private int grado;

  public Manager( String nnombre, String nid, int ngrado
)
  {
     setNombre(nnombre);
     setId(nid);
     grado = ngrado;
  }
}
```

Código sin refactorizar

```
public class Manager extends Empleado
{
  private int grado;

  public Manager( String nnombre, String nid, int ngrado
)
  {
     super( nnombre, nid );
     grado = ngrado;
  }
}
```

Código refactorizado

TEMA 6 : Diseño de clases
Mas ejemplos de refactorización

- http://www.refactoring.com/catalog/index.html

TEMA 6 : Diseño de clases

Diseño de clases

- Acoplamiento
- Cohesión
- Diseño dirigido por responsabilidades
- Refactorización

Conceptos sobre ficheros

- Ficheros en Java
- Ficheros binarios
- Ficheros de texto

Índice

Conceptos Básicos de Ficheros

Estructuras de Datos estudiadas:

- Tipos: vectores, matrices, etc.
- Almacenadas en **memoria principal** (RAM)
  - Rápida
  - Volátil
  - Tamaño Limitado
- Para tratar grandes volúmenes de información es necesario almacenarla en **memoria secundaria** (Disco Duro, CD-ROM, Disquete, Disco USB, etc.)
  - Los datos agrupados en estructuras denominadas **archivos o ficheros** (File en inglés)

Conceptos Básicos de Ficheros

- Un **archivo o fichero** es una colección de datos homogéneos almacenados en un soporte físico del computador que puede ser permanente o volátil.
  - Datos **homogéneos**: Almacena colecciones de datos del mismo tipo.
  - Puede ser almacenado en **diversos soportes** (Disco duro, CD, DVD, ...)

Operaciones sobre Ficheros

- Tipos de operaciones
  - Operación de **Creación**
  - Operación de **Apertura**. Varios modos:
    - Sólo lectura
    - Sólo escritura
    - Lectura y Escritura
  - Operaciones de **lectura** / **escritura**
  - Operaciones de **inserción** / **borrado**
  - Operaciones de **renombrado** / **eliminación**
  - Operación de **cierre**

Operaciones sobre Ficheros

- Operaciones para el manejo habitual de un fichero:

1.- **Crearlo** (sólo si no existía previamente)

2.- **Abrirlo**

3.- **Operar** sobre él (lectura/escritura, inserción, borrado, etc.)

4.- **Cerrarlo**

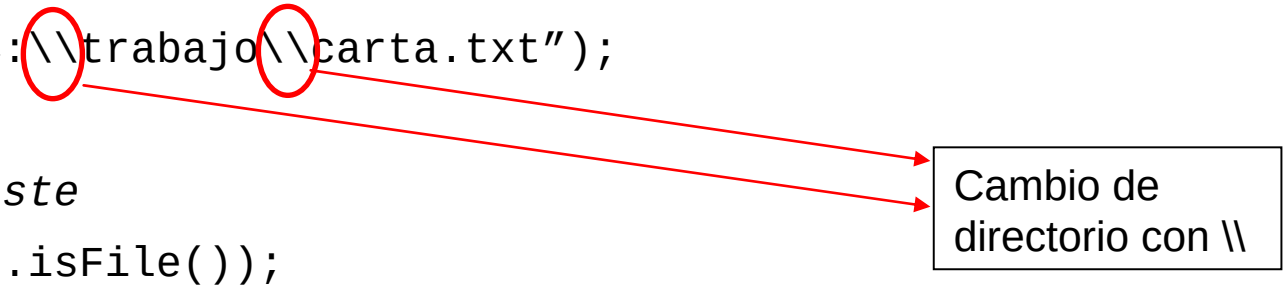
Tipos de Ficheros

- La clase File (del paquete java.io) puede usarse para representar el nombre de un **archivo concreto** o **directorio** .
- Para gestionar el sistema de ficheros se utiliza la clase *File*. El constructor más habitual de esta clase es *File(String path)*.
- Métodos más usados de *File*
  - *boolean isDirectory()*: retorna si es un directorio
  - *boolean isFile()*: retorna si es un fichero
  - *boolean mkdir()*: crea un directorio. Si se puede realizar la operación, devuelve true. Si no, retorna false (ya existe el directorio, no se tienen permisos,...)
  - *String[] list()*: en caso de ser un directorio, retorna un array con el contenido. Si no, retorna null

Tipos de Ficheros

Ejemplos:

```
//f representa un File en C:\trabajo\carta.txt
File f = new File("C:\\trabajo\\carta.txt");

//comprobamos si existe
System.out.println(f.isFile());

File d1 = new File("fundamentos");

d1.mkdir(); //en caso de no existir crea el directorio fundamentos
//en el directorio desde donde se ejecuta el programa

File d2 = new File ("C:\\trabajo");
String [] listado = d2.list(); //guarda contenido de d2 en listado
```



Tipos de Ficheros

- Clasificación de los ficheros según el tipo de la información almacenada:
  - Ficheros **Binarios**: Almacenan secuencias de dígitos binarios (ej: ficheros que almacenan enteros, floats,...)
  - Ficheros de **Texto**: Almacenan caracteres alfanuméricos en un formato estándar (ASCII, Unicode, ...).
    - Pueden ser leídos y/o modificados por aplicaciones denominadas editores de texto (Ej: Notepad, Editplus, etc.).

Índice

Conceptos básicos de E/S

- **Streams**: Canales, flujos de datos o “tuberías”.
- Agrupados en el paquete **java.io**

- Dos jerarquías de clases independientes:
  - una para lectura/escritura binaria (**bytes**)
  - otra para lectura/escritura de caracteres de texto (**char**)

Conceptos básicos de E/S

- Los ***stream*** son objetos que nos permitirán comunicar nuestro programa con otra fuente o destino de datos.
- Un stream *"es un hilo conductor por donde fluyen datos."*
- Tendremos dos clases de streams:
  - stream de entrada: fuente → programa
  - stream de salida: programa → destino

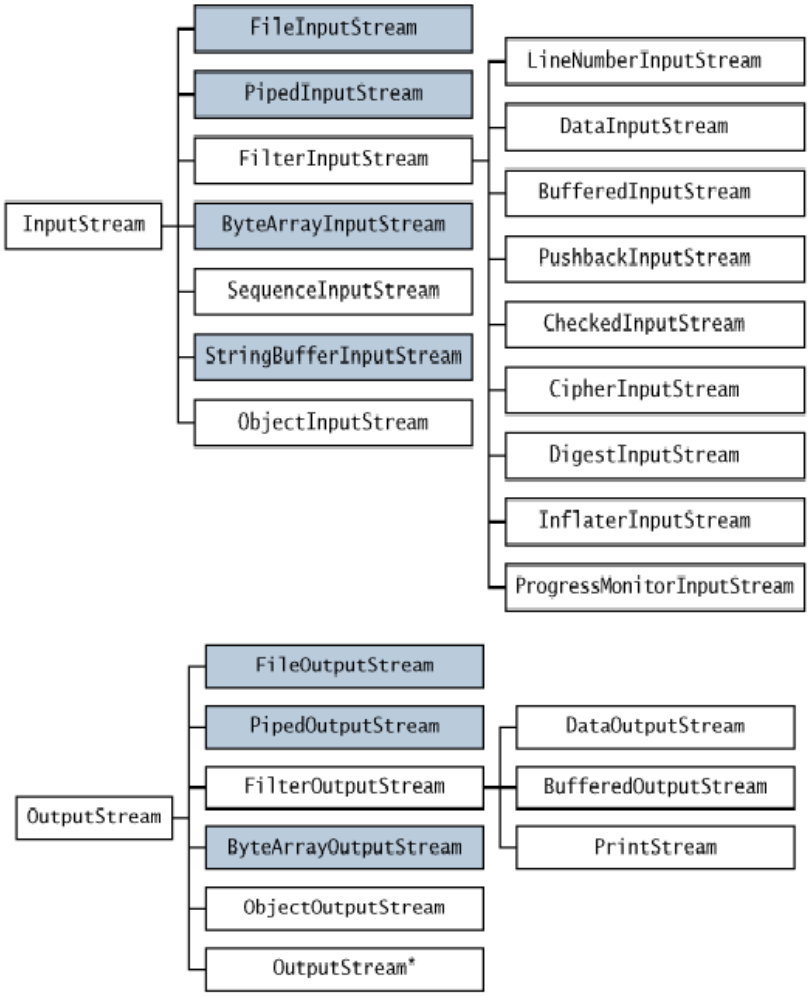
Conceptos básicos de E/S

- Para usar streams:
  - Crear el stream con *new*.
    - En esta operación se abre el stream para las comunicaciones.
  - Leer o escribir, dependiendo del stream.
    - En la lectura, para indicar que en el stream no hay más datos para leer, se recibirá un –1 o null, dependiendo del tipo de método que se utilice para leer.
  - Cerrar el stream.
    - Si no se cierra, no se garantiza que las operaciones de escritura se hayan realizado correctamente.

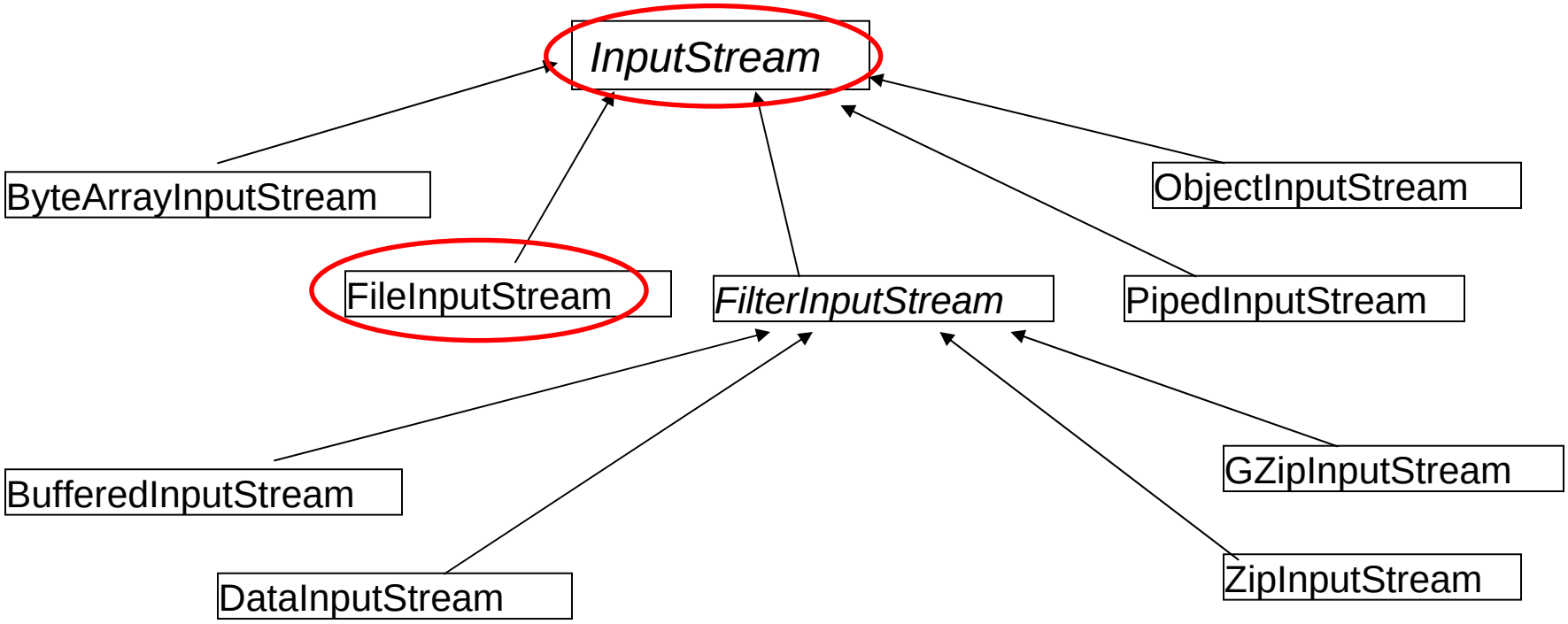
Streams de bytes

- Streams orientados a leer bytes: subclases de la clase ***InputStream.***

- Streams orientados a escribir bytes: subclases de la clase ***OutputStream.***



Streams de bytes



System.in, es un objeto de la clase BufferedInputStream

Streams de bytes

Las distintas subclases de *InputStream* se clasifican según de la fuente para leer bytes:

FUENTE	CLASE
array de bytes	ByteArrayInputStream
fichero	FileInputStream
varios streams	SequenceInputStream
PipedOutputStream de otro hilo	PipedInputStream
objetos escritos por ObjectOutputStream	ObjectInputStream
funciones especiales	*FilterInputStream*

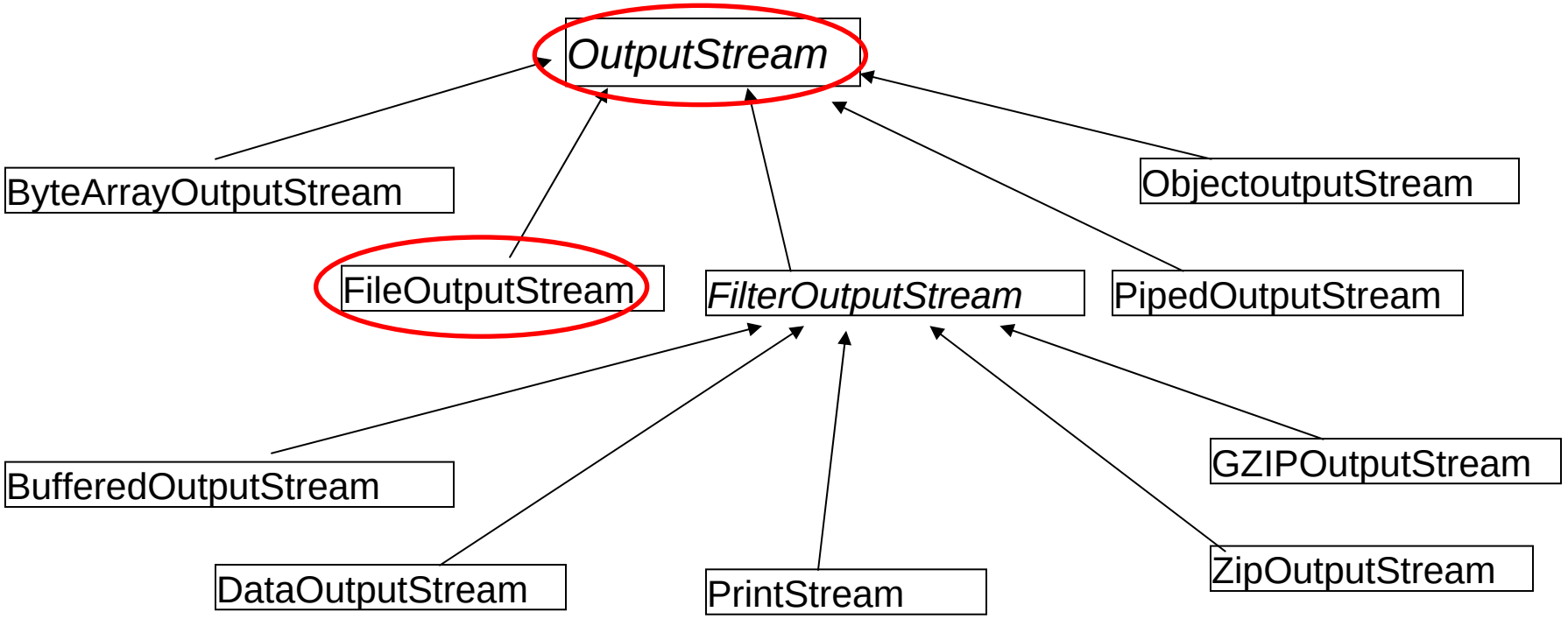
Streams de bytes

Métodos básicos de **InputStream**:

- int read(): lee el siguiente byte disponible en el stream. Si no hay más, retorna –1
- int read(byte v[]): intenta leer tantos bytes como tamaño del array v. Devuelve el número de bytes leídos, si no hay bytes disponibles en el stream, -1
- void close (): cierra el stream

Todos estos métodos lanzan la excepción IOException en caso de problemas E/S.

Streams de bytes



System.out es objeto de la clase PrintStream

Streams de bytes

Se clasifican las distintas subclases de *OutputStream* según el destino de los bytes escritos:

DESTINO	CLASE
Array de bytes	ByteArrayOutputStream
Fichero	FileOutputStream
Objetos escritos por ObjectOutputStream	ObjectOutputStream
funciones especiales	*FilterOutputStream (abstracta)*

Streams de bytes

Métodos básicos de **OutputStream**:

- void write(int c): escribe en el destino el byte menos significativo del entero *b* (un *int* son 4 bytes)
- void write(byte[] v): escribe en destino todos los bytes de *v*.
- void write(byte[] v, int pos, int long): escribe *long* bytes del vector *v* empezando desde la posición *pos*
- void close (): cerrar stream

Todos estos métodos lanzan la excepción IOException en caso de problemas E/S.

Streams de bytes: ficheros

Acceso a ficheros:

- Acceso binario: FileInputStream / FileOutputStream

`FileInputStream`:similar a `InputStream`, para leer archivos.

`FileInputStream(String name)`: constructor con argumento un String, nombre de fichero

`FileInputStream(File name)`: constructor con argumento de tipo File, fichero a leer.

Estos constructores lanzan la excepción **FileNotFoundException** en caso de que el fichero no exista

`FileOutputStream`: similar a `OutputStream`, para escribir en archivos.

`FileOutputStream(String name)`: constructor con argumento un String, nombre de fichero

`FileOutputStream(File name)`: constructor con argumento de tipo File, fichero a leer.

Streams de bytes: Ejemplos

CUIDADO, CÓDIGO INCOMPLETO

Muestra por pantalla los bytes contenidos en el fichero c:\misfotos\galicia.jpg

Lee byte a byte del fichero.

```
//Crear un stream asociado al fichero
FileInputStream in = new FileInputStream("c:\\misfotos\\galicia.jpg");

int leido;
while ((leido = in.read()) != -1) { //lee hasta que no tengas más
   System.out.println(leido);
}
in.close();
```

Gestión de excepciones

- Al leer de un fichero hay que tener en cuenta que se pueden *lanzar* excepciones:
  - al crear el stream con el nombre del fichero: FileNotFoundException (no existe el fichero a leer)
  - al leer o cerrar el fichero: IOException (problemas de entrada/salida)
- Las excepciones de este tipo es necesario *capturarlas*
- Se capturan mediante bloques *try{…} catch{…}*

```
try {
    FileInputStream in = new FileInputStream("c:\\misfotos\\galicia.jpg");
    int leido;
    while ((leido = in.read()) != -1) { //lee hasta que no tengas más
          System.out.println(leido);
    }
    in.close();
} catch (FileNotFoundException e) {
     System.out.println("No existe el fichero");
} catch (IOException e) {
     System.out.println("Error E/S");
}
```

Streams de bytes: Ejemplos

- Lee un grupo de bytes y los saca por pantalla
- Al igual que en el ejemplo anterior, es necesario capturar las posibles excepciones que se lancen.

```
try {
   //Crear streams asociados a los ficheros: entrada y salida
   FileInputStream in = new FileInputStream("c:\\misfotos\\galicia.jpg");
   FileOutputStream out = new FileOutputStream("c:\\misfotos\\copia.jpg");

   byte[] buffer = new byte[1024];
   int totalLeidos;
   while ((totalLeidos = in.read(buffer)) != -1) {
      for (int i = 0; i < totalLeidos; i++) {
          System.out.println(buffer[i]);
      }
   }
   in.close();
   out.close();
} catch (FileNotFoundException e) {
     System.out.println("No existe el fichero");
} catch (IOException e) {
     System.out.println("Error E/S");
}
```

Streams de bytes: Ejemplos

Copia el fichero c:\misfotos\galicia.jpg en c:\misfotos\copia.jpg

Copia byte a byte

```
try {
   //Crear streams asociados a los ficheros: entrada y salida
   FileInputStream in = new FileInputStream("c:\\misfotos\\galicia.jpg");
   FileOutputStream out = new FileOutputStream("c:\\misfotos\\copia.jpg");

   int leido;
   while ((leido = in.read()) != -1) {
      out.write(leido);
   }
   in.close();
   out.close();
} catch (FileNotFoundException e) {
     System.out.println("No existe el fichero");
} catch (IOException e) {
     System.out.println("Error E/S");
}
```

Streams de bytes: Ejemplos

Copia el fichero c:\misfotos\galicia.jpg en c:\misfotos\copia.jpg

Copia en grupos de bytes

```
try {
   //Crear streams asociados a los ficheros: entrada y salida
   FileInputStream in = new FileInputStream("c:\\misfotos\\galicia.jpg");
   FileOutputStream out = new FileOutputStream("c:\\misfotos\\copia.jpg");

   byte[] buffer = new byte[1024];
   int totalLeidos;
   while ((totalLeidos = in.read(buffer)) != -1) {
      for (int i = 0; i < totalLeidos; i++) {
         out.write(buffer, 0, totalLeidos);
      }
   }
   in.close();
   out.close();
} catch (FileNotFoundException e) {
     System.out.println("No existe el fichero");
} catch (IOException e) {
     System.out.println("Error E/S");
}
```

Índice

Introducción

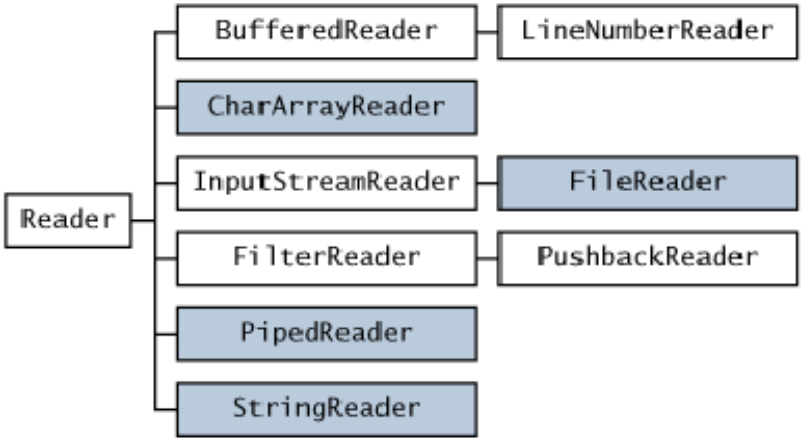
- Conceptos Básicos de Ficheros
- Operaciones sobre ficheros
- Tipos de ficheros

Ficheros en Java

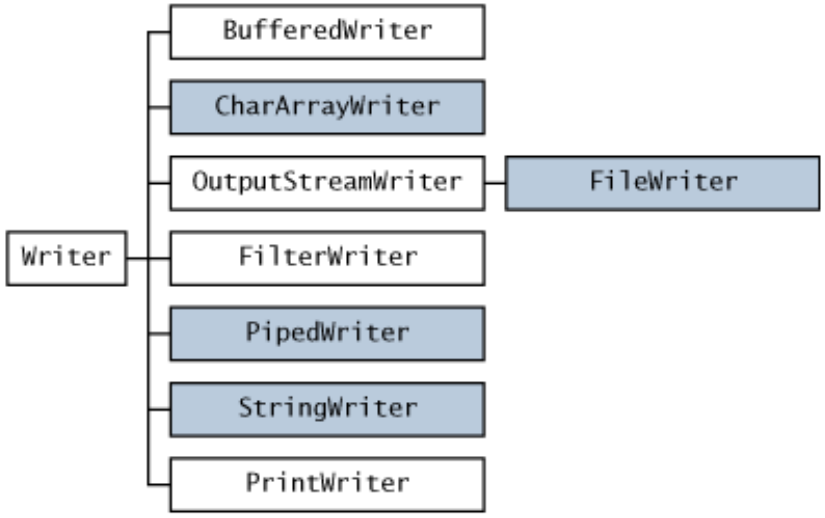
- Conceptos Básicos de Entrada/Salida
- Ficheros Binarios
  - Leer bytes: InputStream, FileInputStream
  - Escribir bytes:OutputStream, FileOutputStream
  - Gestión de excepciones: IOException, FileNotFoundException
- Ficheros de Texto
  - Leer carácter: Reader, FileReader
  - Escribir caracteres: Writer, FileWriter
  - Escribir texto formateado: PrintWriter
  - Leer texto formateado: Scanner

Streams de caracteres

- Streams orientados a leer caracteres: subclases de la clase abstracta ***Reader.***

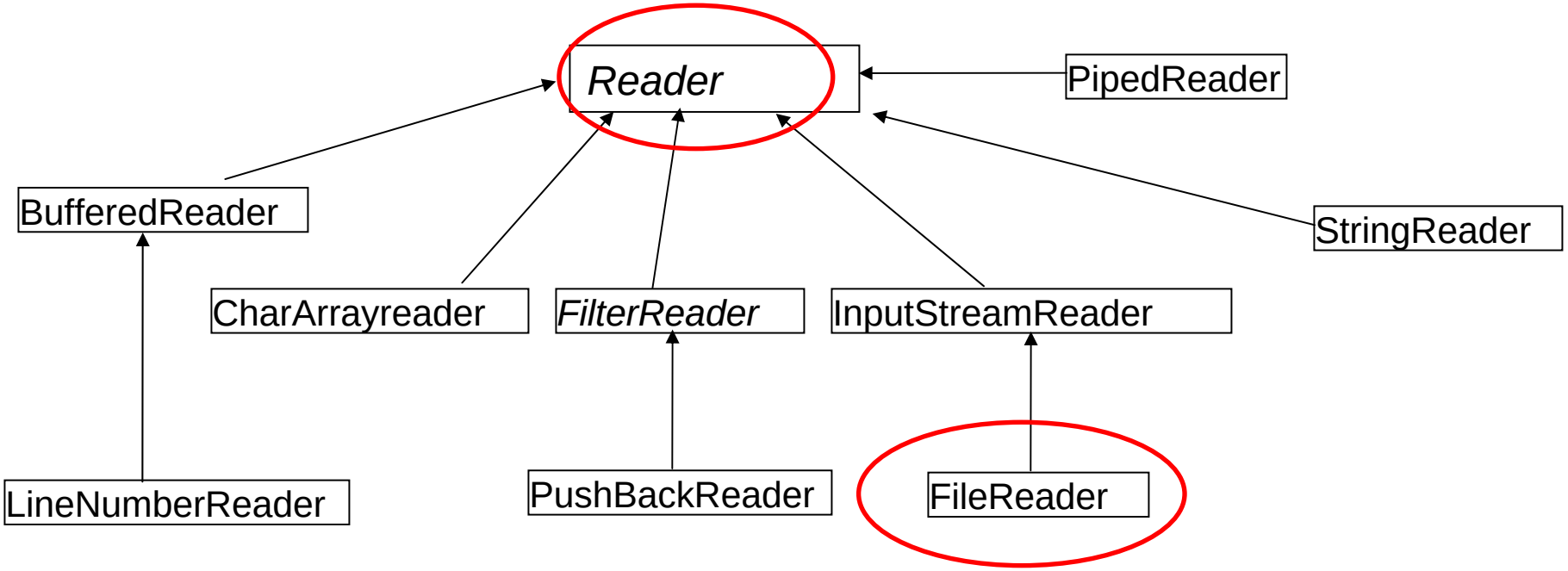


- Streams orientados a escribir caracteres: subclases de la clase abstracta ***Writer.***



Streams de caracteres

Las clases para leer caracteres derivan de la clase abstracta *Reader.*



Streams de caracteres

Las distintas clases de *Reader* se usan dependiendo de la fuente de los datos.

FUENTE	CLASE
un buffer	BufferedReader
fichero	FileReader
array de caracteres	CharArrayReader
stream al que se le pueden devolver caracteres una vez leídos	PushbackReader
buffer para contar líneas	LineNumberReader
un String	StringWriter
un stream de bytes	InputputStreamWriter

Streams de caracteres

Métodos básicos de **Reader**:

- int read (): lee el siguiente caracter disponible en el stream. Si no hay más, retorna –1.

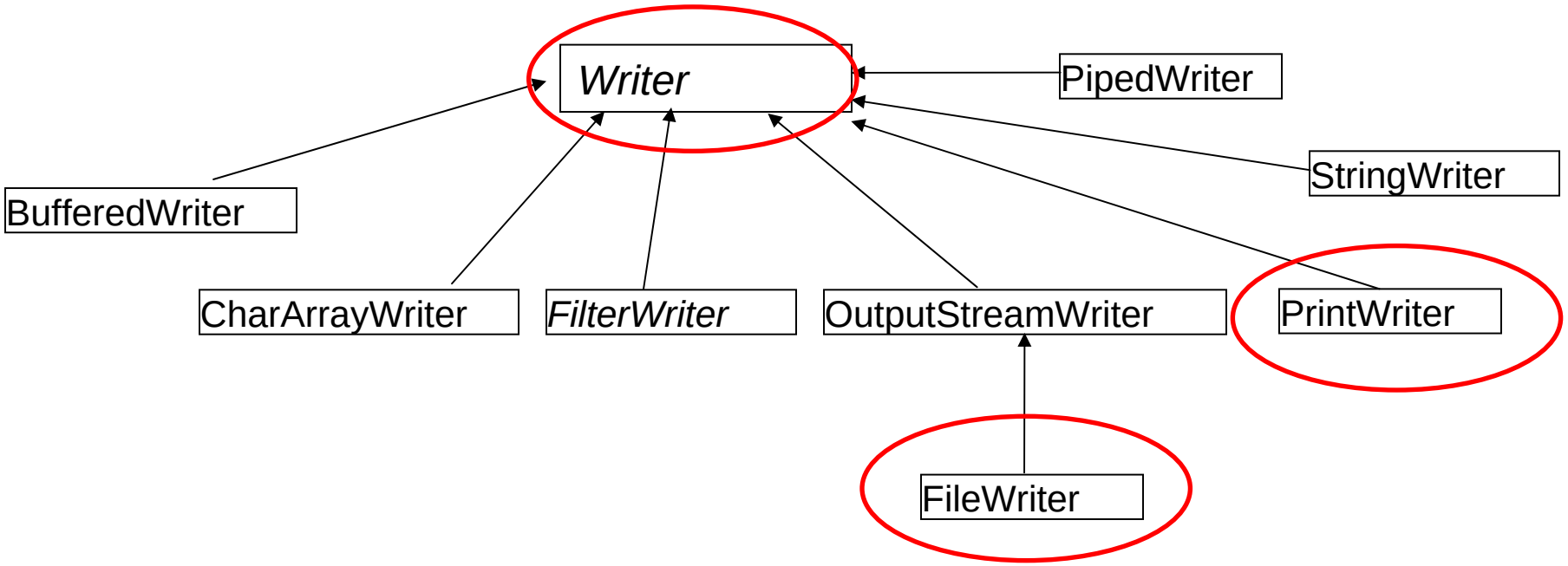
- int read (char [] v): intenta leer tantos caracteres como tamaño del array v. Devuelve el número de caracteres leídos. Si no hay más disponibles en el stream, -1.

- void close (): cierra el stream.

Todos los métodos lanzan la excepción IOException en caso de problemas E/S

Streams de caracteres

Las clases para escribir caracteres derivan de la clase abstracta *Writer.*



Streams de caracteres

Las distintas clases de Writer se usan dependiendo del destino de los datos.

DESTINO	CLASE
un buffer	BufferedWriter
fichero	FileWriter
array de caracteres	CharArrayWriter
texto formateado	PrintWriter
un String	StringWriter
un stream de bytes	OutputStreamWriter

Streams de caracteres

Métodos básicos de **Writer**:

- void write (int c): escribe el “carácter” c
- void write (String s): escribe el String s
- void write(String s, in pos, int len):escribe *len* caracteres de *s* desde la posición *pos*
- void write (char [] b): escribe en el stream el array de caracteres b
- void write(char[] b, int pos, int len): escribe *len* caracters de *b* desde la posición *pos*
- void close(): cierra el stream

Todos los métodos lanzan la excepción IOException en caso de problemas E/S

Streams de caracteres: ficheros

Acceso a ficheros:

- Acceso ficheros de texto: FileReader / FileWriter

`FileReader`: similar a `Reader`, para leer archivos de texto.

`FileReader(String name)`: constructor con argumento un String, nombre de fichero

`FileReader(File name)`: constructor con argumento de tipo File, fichero a leer.

Estos constructores lanzan la excepción **FileNotFoundException** en caso de que el fichero no exista

`FileWriter`: similar a `Writer`, para escribir en archivos de texto.

`FileWriter(String name)`: constructor con argumento un String, nombre de fichero

`FileWriter(File name)`: constructor con argumento de tipo File, fichero a leer.

Streams de caracteres: Ejemplos

Muestra por pantalla el texto contenido en el fichero c:\miscosas\carta.txt

Lee carácter a carácter el fichero.

Como en los casos anteriores, hay que capturar las posibles excepciones

```
try {
   FileReader in = new FileReader("c:\\miscosas\\carta.txt");

   int leido;
   while ((leido = in.read()) != -1) { //lee hasta que no tengas más
     char c = (char) leido; //transformar a caracter
     System.out.print(c);
   }
   in.close();
} catch (FileNotFoundException e) {
     System.out.println("No existe el fichero");
} catch (IOException e) {
     System.out.println("Error E/S");
}
```

Streams de caracteres: Ejemplos

Muestra por pantalla el texto contenido en el fichero c:\miscosas\carta.txt

Lee un grupo de bytes del fichero

```
try {
   //Crear un stream asociado al fichero
   FileReader in = new FileReader("c:\\miscosas\\carta.txt");

    char[] buffer = new char[1024];
    int totalLeidos;
    while ((totalLeidos = in.read(buffer)) != -1) {
       for (int i = 0; i < totalLeidos; i++) {
          System.out.print(buffer[i]);
       }
    }
    in.close();
} catch (FileNotFoundException e) {
     System.out.println("No existe el fichero");
} catch (IOException e) {
     System.out.println("Error E/S");
}
```

Streams de caracteres: Ejemplos

Copia el fichero c:\miscosas\carta.txt en c:\miscosas\copia.txt

Copia carácter a carácter

```
try {
   //Crear streams asociados a los ficheros: entrada y salida
   FileReader in = new FileReader("c:\\miscosas\\carta.txt");
   FileWriter out = new FileWriter("c:\\miscosas\\copia.txt");

   int leido;
   while ((leido = in.read()) != -1) {
      out.write(leido);
   }
   in.close();
   out.close();
} catch (FileNotFoundException e) {
     System.out.println("No existe el fichero");
} catch (IOException e) {
     System.out.println("Error E/S");
}
```

Streams de caracteres: Ejemplos

Copia el fichero c:\miscosas\carta.txt en c:\miscosas\copia.txt

Copia en grupos de bytes

```
try {
   //Crear streams asociados a los ficheros: entrada y salida
   FileReader in = new FileReader("c:\\miscosas\\carta.txt");
   FileWriter out = new FileWriter("c:\\miscosas\\copia.txt");

   char[] buffer = new char[1024];
   int totalLeidos;
   while ((totalLeidos = in.read(buffer)) != -1) {
      out.write(buffer, 0, totalLeidos);
   }
  in.close();
  out.close();
} catch (FileNotFoundException e) {
     System.out.println("No existe el fichero");
} catch (IOException e) {
     System.out.println("Error E/S");
}
```

Streams de caracteres: escribir texto formateado

- Con la clases vistas hasta el momento es complicado escribir texto formateado en ficheros.
- ¿Cómo es escribir en un fichero de texto dos líneas de texto del tipo?

```
El valor de la variable n es 24
El valor de m es 36
```

- Existe una clase que soluciona fácilmente este problema: ***PrintWriter***

Streams de caracteres: texto formateado

- Constructores de *PrintWriter*
  - PrintWriter(File f): para escribir texto en el fichero *f*
  - PrintWriter(String name): para escribir texto en el fichero de nombre *name*

- Métodos más usados:
  - println(String s)
  - print(String s);

Streams de caracteres: texto formateado

Ejemplo:

```
int n = 24;
int m = 36;

try {
   PrintWriter out = new PrintWriter(“texto.txt”);
   out.println(“El valor de la variable n es “ + n);
   out.println(“El valor de m es “ + m);
   out.close();
} catch (IOException e) {
    System.out.println(“Error E/S”);
}
```

Leer texto formateado: clase Scanner

- A partir de jdk 1.5 se dispone de la clase *Scanner* para facilitar la lectura. Esta clase no es un stream. Es una utilidad. Pertenece al paquete **java.util**
- Con esta clase podemos leer variables de los tipos primitivos desde un fichero, desde la entrada estándar, de un string.
- *Scanner* divide la entrada en tokens según un delimitador que por defecto es el espacio en blanco.

  Ej. *'23 hola 67.8'* se divide en tres tokens:

23	*hola*	*67.8*

- Con los métodos de *Scanner* podemos transformar cada uno de los tokens en un tipo de dato primitivo o String, leer una línea de texto del fichero.

Clase Scanner

Principales métodos Scanner:

- int nextInt(): devuelve el siguiente token como un valor de tipo int.
- boolean hasNextInt(): nos indica si el siguiente token es de tipo int.
- Hay métodos similares para todos los tipos primitivos **salvo char**.

- String next(): devuelve el siguiente token como String.
- boolean hasNext(): devuelve cierto si hay más tokens. Falso en caso contrario
- String nextLine(): devuelve un String con todos los caracteres hasta encontrar un salto de línea.

Clase Scanner

```
try {
   File f = new File(“texto.txt”);
   Scanner sc = new Scanner(f);

   //sacar por pantalla todos los números enteros que hay al
   //inicio del fichero texto.txt
   while (sc.hasNextInt())
      System.out.println(sc.nextInt());

   //leer línea a línea el resto del fichero
   while (sc.hasNext())
   System.out.println(sc.nextLine());
} catch (FileNotFoundException e) {
    System.out.println(“Fichero no existe”);
}
```