v 1.8

Tema 7
Entrada salida en Java

Programación Orientada a Objetos

José Francisco Vélez Serrano

Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Índice

- Streams de entrada y salida
- Reader y Writer
- Scanner
- Serialización
- Otras clases para la gestión de ficheros

Streams de entrada y salida

En esta sección se introducen los streams que son un medio que ofrece Java para gestionar la entrada y la salida de datos de un programa.

Concepto de stream

Un stream de Java es un flujo de datos y permiten gestionar elementos tan diversos como los ficheros, las comunicaciones con periféricos, la transmisiones a redes de ordenadores…

Los streams de Java son objetos que permiten enviar, recibir y transformar bytes.

Se puede abrir un stream para enviar bytes a un fichero, para recibir bytes de un servidor de Internet, para enviar bytes a una impresora…

También se puede abrir un stream para cifrar o descifrar un flujo de bytes, o para comprimirlo o descomprimirlo...

FF 01 0B AB 7E 64 23 42 0A 10 F

InputStream

Un InputStream es una clase que permite leer datos de un stream.

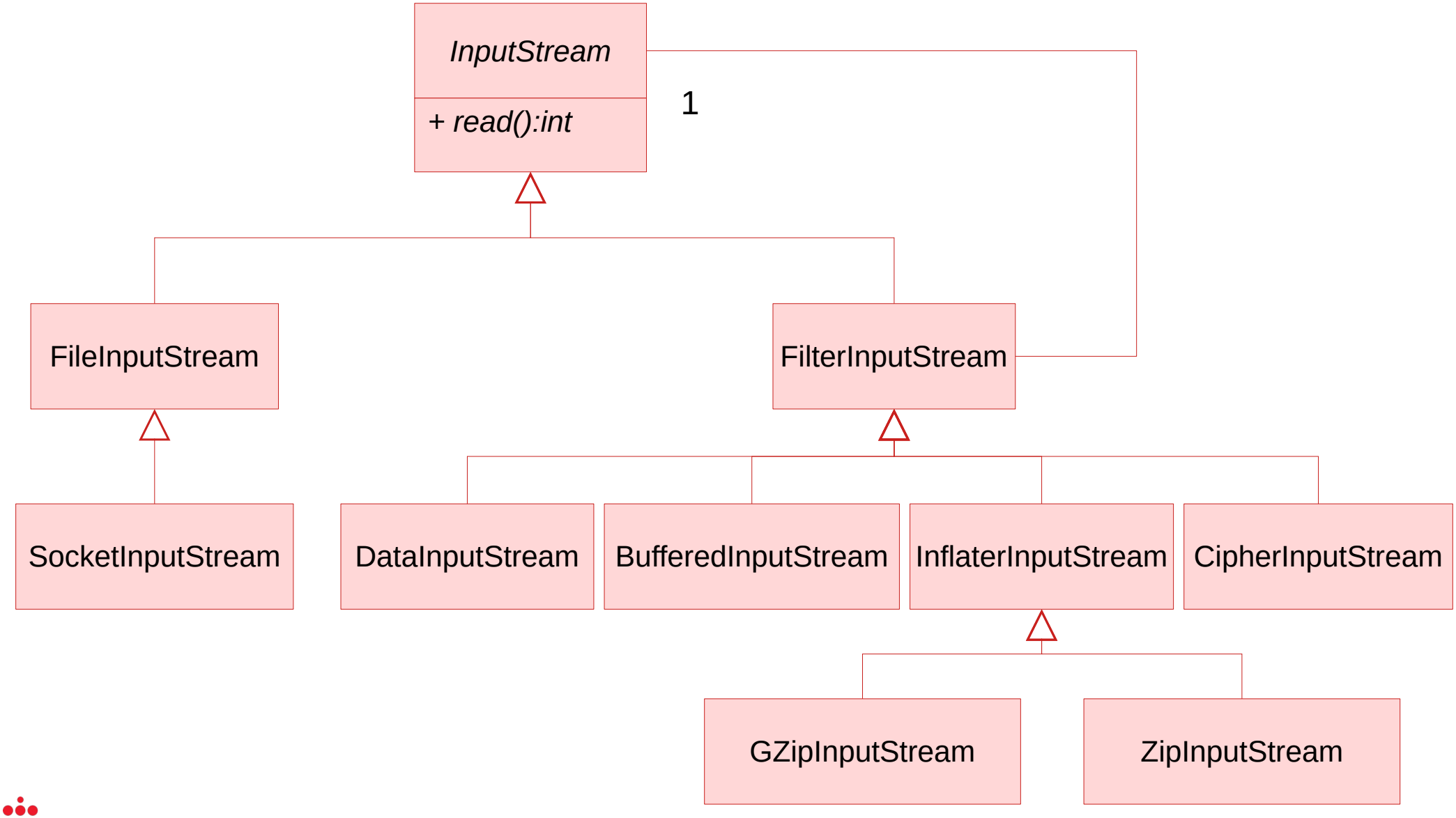
La clase InputStream es abstracta. Hay que disponer de una clase derivada de InputStream adecuada al tipo de flujo del que se desea leer.

Para leer de un InputStream basta con utilizar el método `read`. Dicho método devuelve el siguiente byte a leer del flujo.

InputStream devuelve un tipo de dato `int`, pero el dato contenido nunca supera `255`. Además, el valor siempre es mayor o igual a cero, excepto para indicar que ya no quedan datos devolviendo `-1`.

Streams de entrada y salida

Algunas clases derivadas de InputStream



Ejemplo de uso de InputStream

El siguiente fragmento de código abre un `InputStream` para leer bytes de un fichero y guardarlos en un array.

```
InputStream in = new FileInputStream("C:/home/juan/NombreFichero");
ArrayList <Byte> l = new ArrayList<>();
int dato = in.read();

while (dato != -1) {
    l.add((byte)dato);
    dato = in.read();
}
```

OutputStream

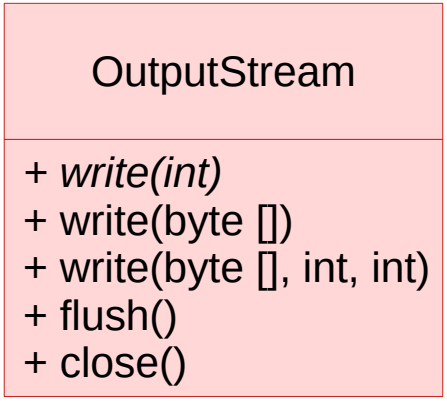
Un OutputStream es una clase que permite escribir datos en un stream.

La clase OutputStream también es abstracta. Hay que usar la clase derivada adecuada al tipo de flujo.

Para escribir en un `OutputStream` basta con utilizar el método `write`. Dicho método escribe un byte en el flujo.

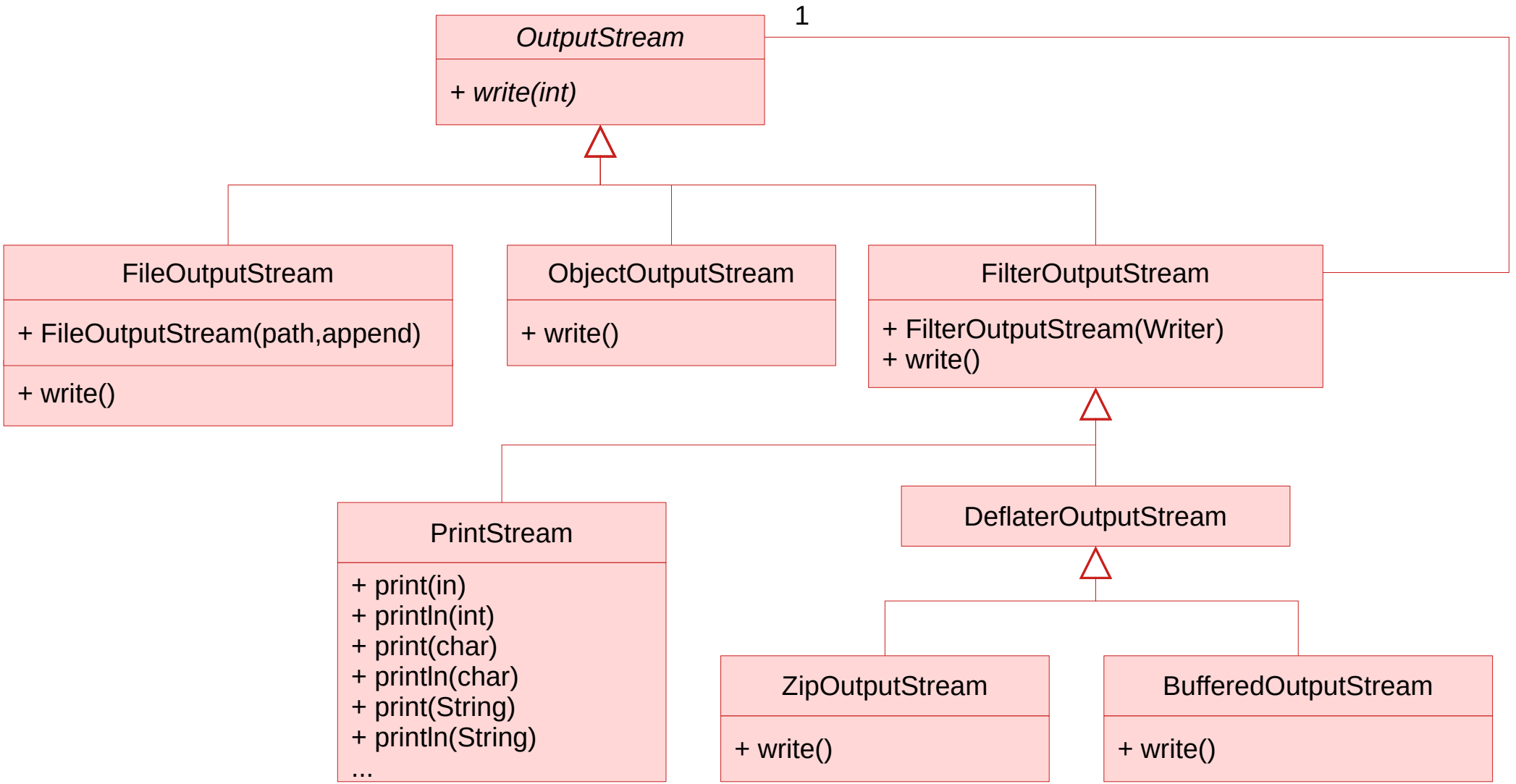
OutputStream usa `int`, pero el dato proporcionado siempre será del rango `[0,255]`.

Para asegurar la escritura a fichero hay que hacer flush o cerrar el stream con close.



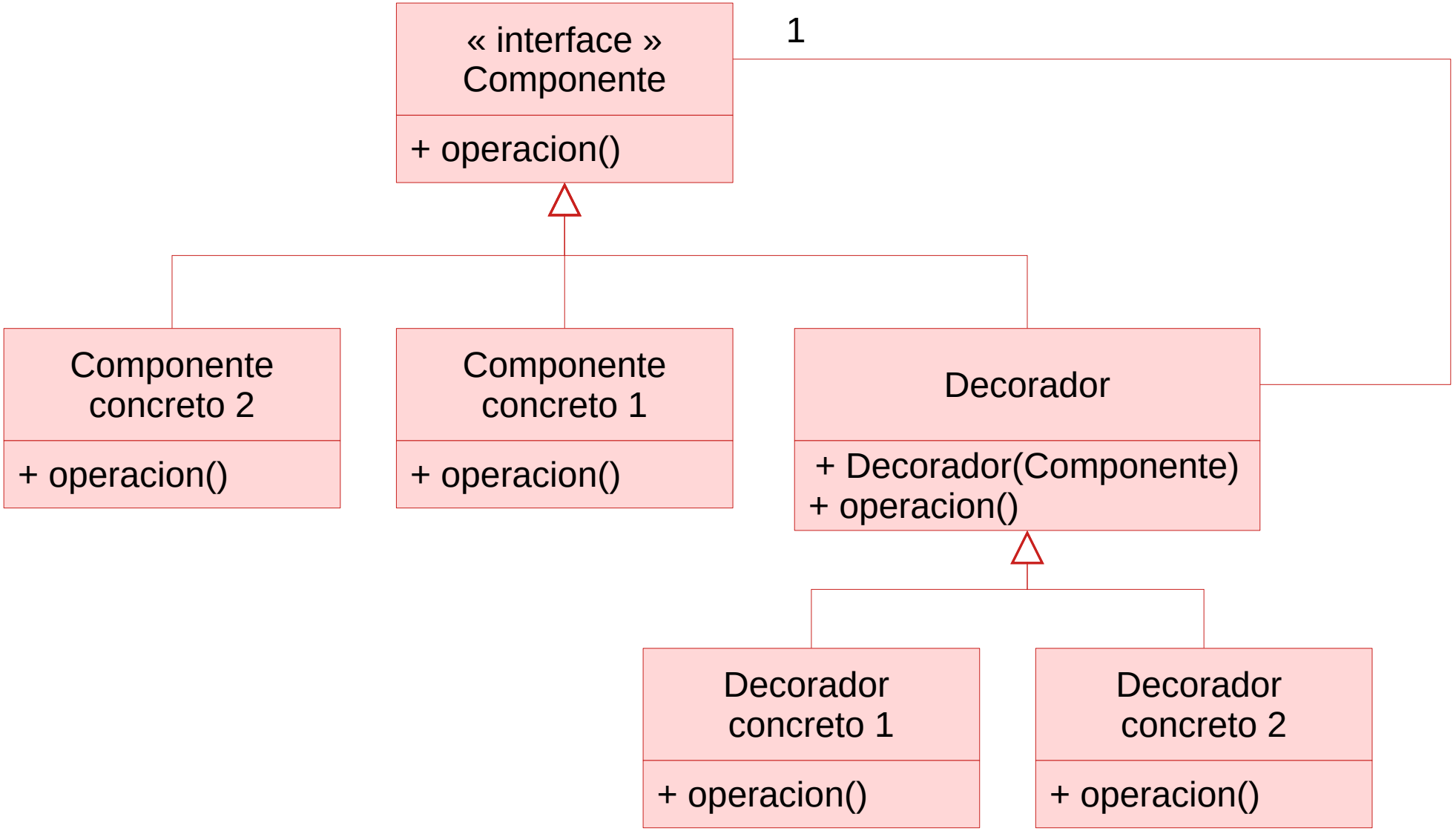
Streams de entrada y salida

Algunas clases derivdas de OutputStream



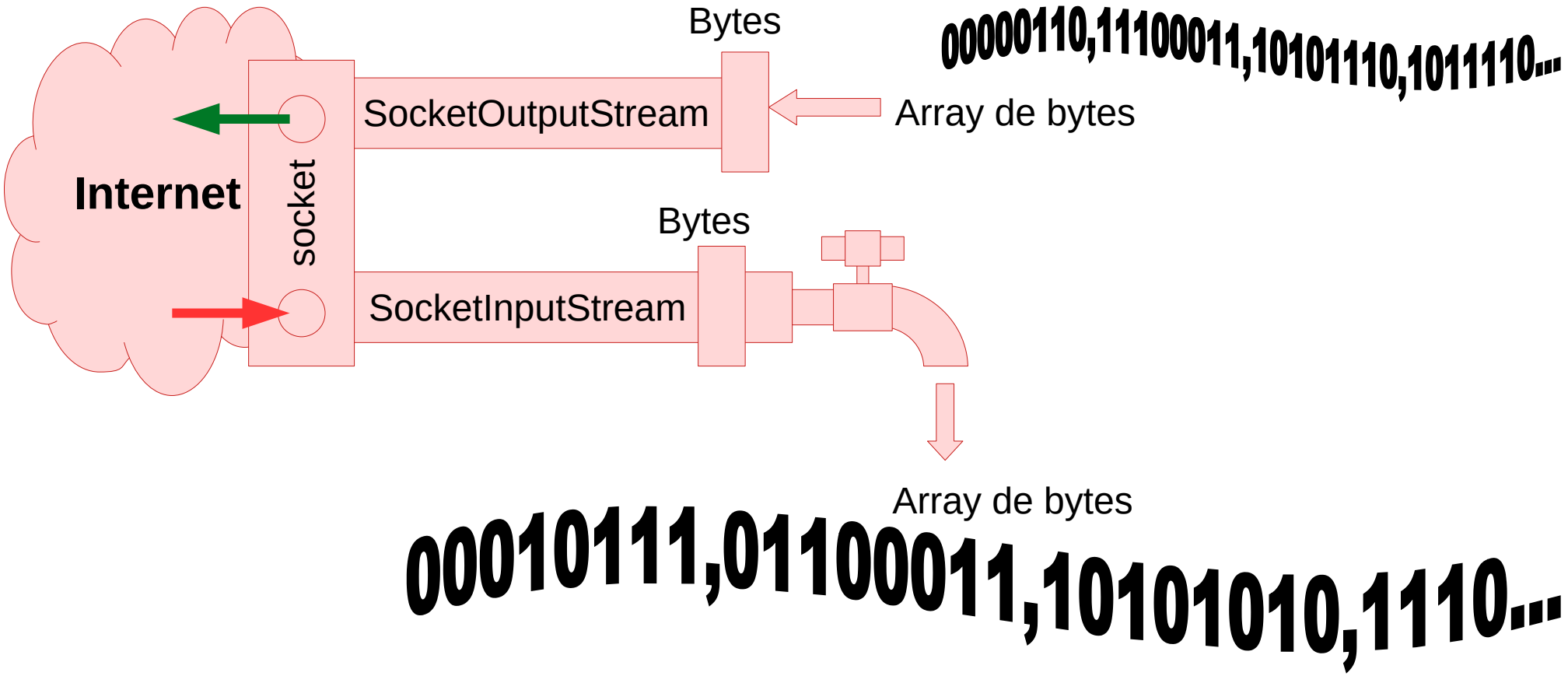
Streams de entrada y salida

Los streams siguen el Patrón Decorator



Streams de entrada y salida

Streams para comunicar dos ordenadores



Reader y Writer

En esta sección se presentan los `Reader` y los `Writer` que gestionan los flujos suponiendo que los bytes corresponden a caracteres unicode 16.

Reader

Un `Reader` es un flujo de entrada que, a diferencia de los `InputStream`, presupone que los bytes corresponden a caracteres unicode 16 (UTF-16).

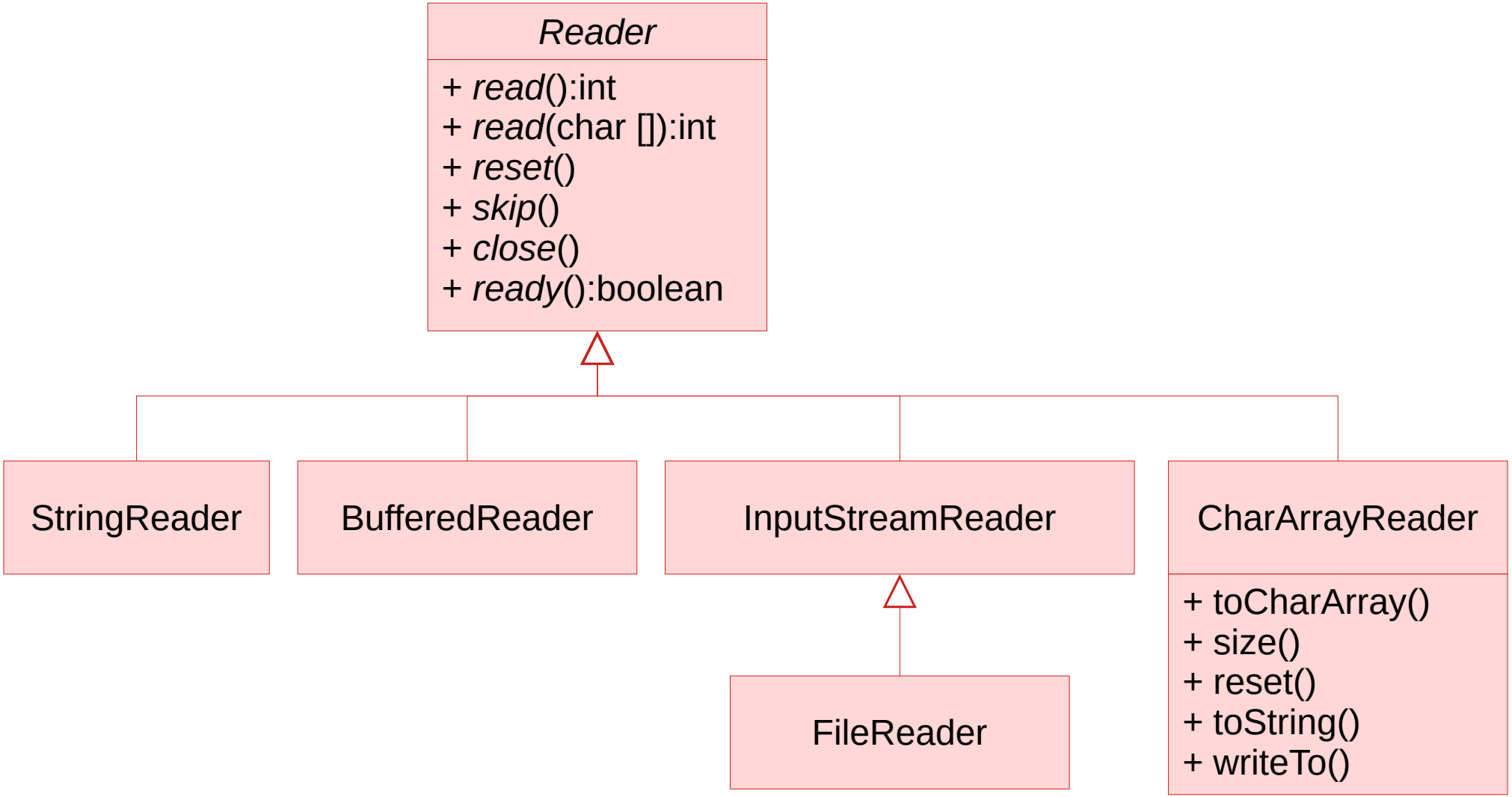
Así, cuando se invoca al método `read()` devuelve un entero que contiene 2 bytes correspondientes a un carácter UTF-16.

`Reader` es una clase abstracta, y tiene derivados para:

- Leer caracteres de un fichero de texto.
- Procesar uno a uno los caracteres de un `String`.
- Convertir un `InputStream` en un `Reader`
- Almacenar en un buffer los caracteres de otro reader y leerlos línea a línea.

Reader y Writer

Reader



Writer

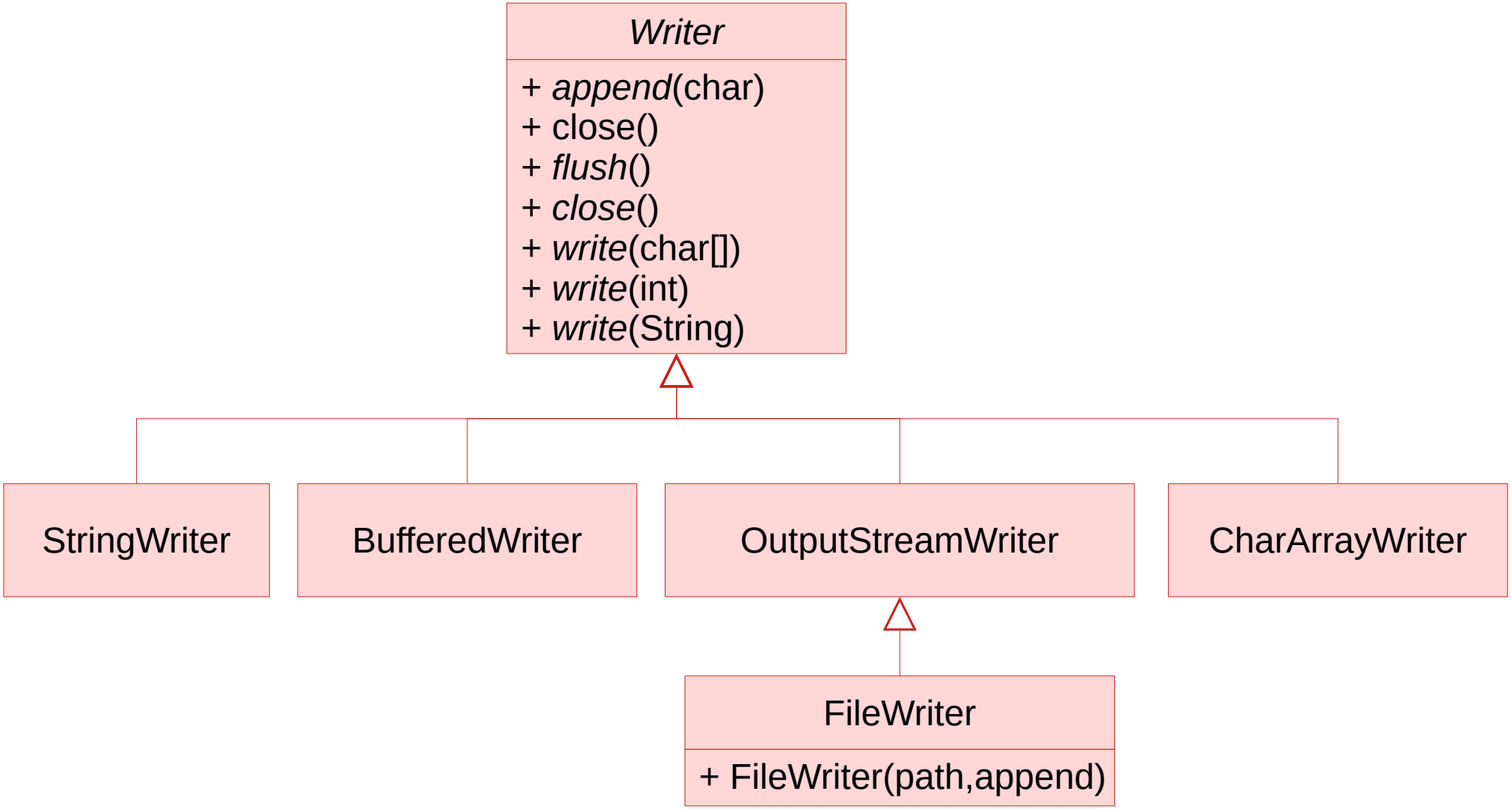
Un `Writer` es un flujo de salida que, a diferencia de un `OutputStream`, presupone que los bytes forman parte de un carácter unicode 16.

Así, cada vez que se invoca al método `write()` se envía un entero que contiene los dos bytes correspondientes a un carácter unicode 16.

Se puede abrir un `Writer` para escribir caracteres en un fichero de texto.

Reader y Writer

Writer



OutputStream sigue el Patrón Decorator

El siguiente fragmento de código abre un `FileReader` para leer caracteres y lo concatena a un `BufferedReader` para leer líneas completas.

```
Reader in = new FileReader("C:/prueba.txt");
BufferedReader buf = new BufferedReader(in,10);
String s;
do {
  s = buf.readLine();
  System.out.print(s);
} while (s != null);
```

Streams del sistema

Unix introdujo el concepto de entrada/salida/error estándar para definir los flujos de entrada, salida y error que por defecto utiliza un programa.

Java ha definido streams de bytes para esos mismos tres flujos:

- `System.in` – Para la recepción de datos.
- `System.out` – Para la salida de datos.
- `System.err` – Para los avisos de error.

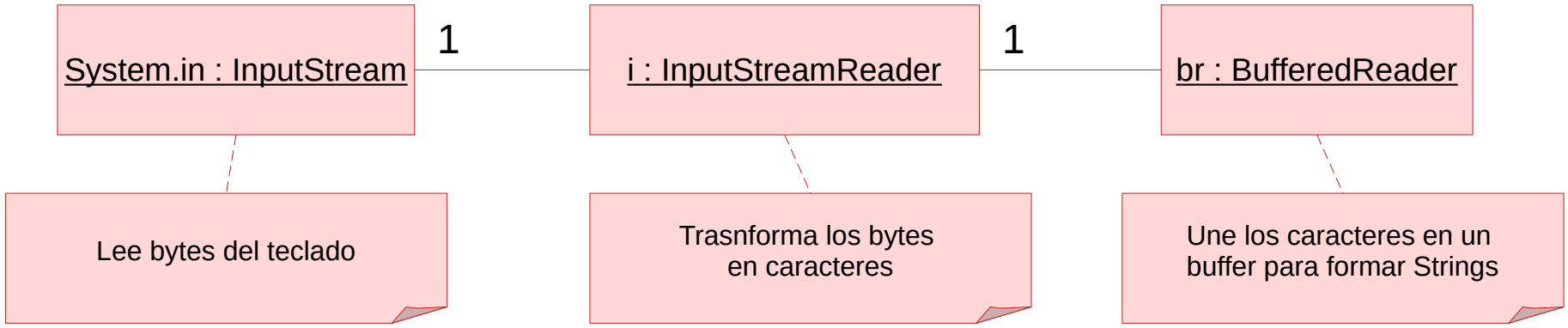
System
+ in:InputStream + out:PrintStream + err:PrintStream
+ setErr(PrintStream) + setIn(InputStream) + setOut(PrintStream) + console():Console + currentTimeMilis():long + exit() + getenv():Map<String,String> + getProperties():Properties + getSecurityManager():SecurityManager + lineSeparator():String + nanoTime():long + runFinalization()

Ejemplo de uso de los streams del sistema

El siguiente fragmento de código convierte el stream que viene de la entrada estándar en un flujo de caracteres, Luego, lo concatena a un `BufferedReader` para leer líneas completas.

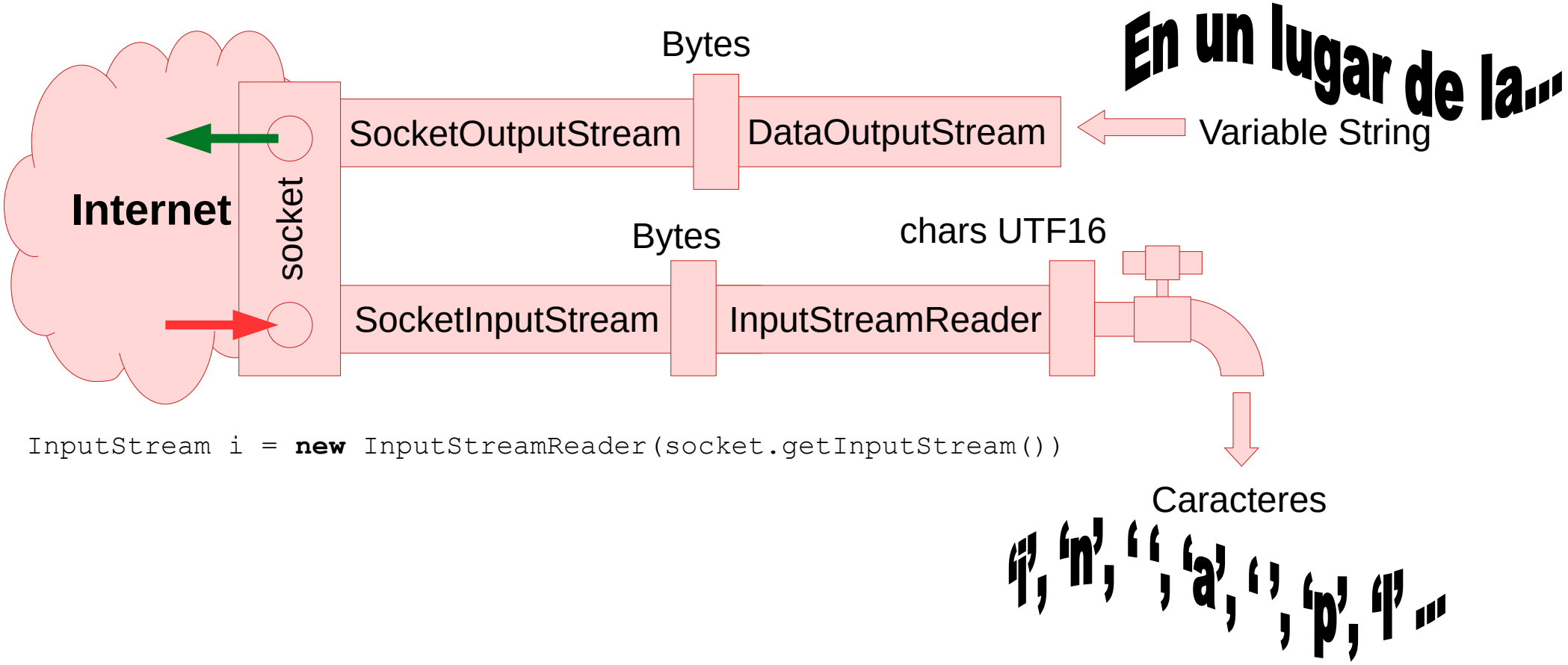
```
Reader i = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(i);
String s = br.readLine();

System.out.print("Ha escrito: ");
System.out.println(s);
```



Reader y Writer

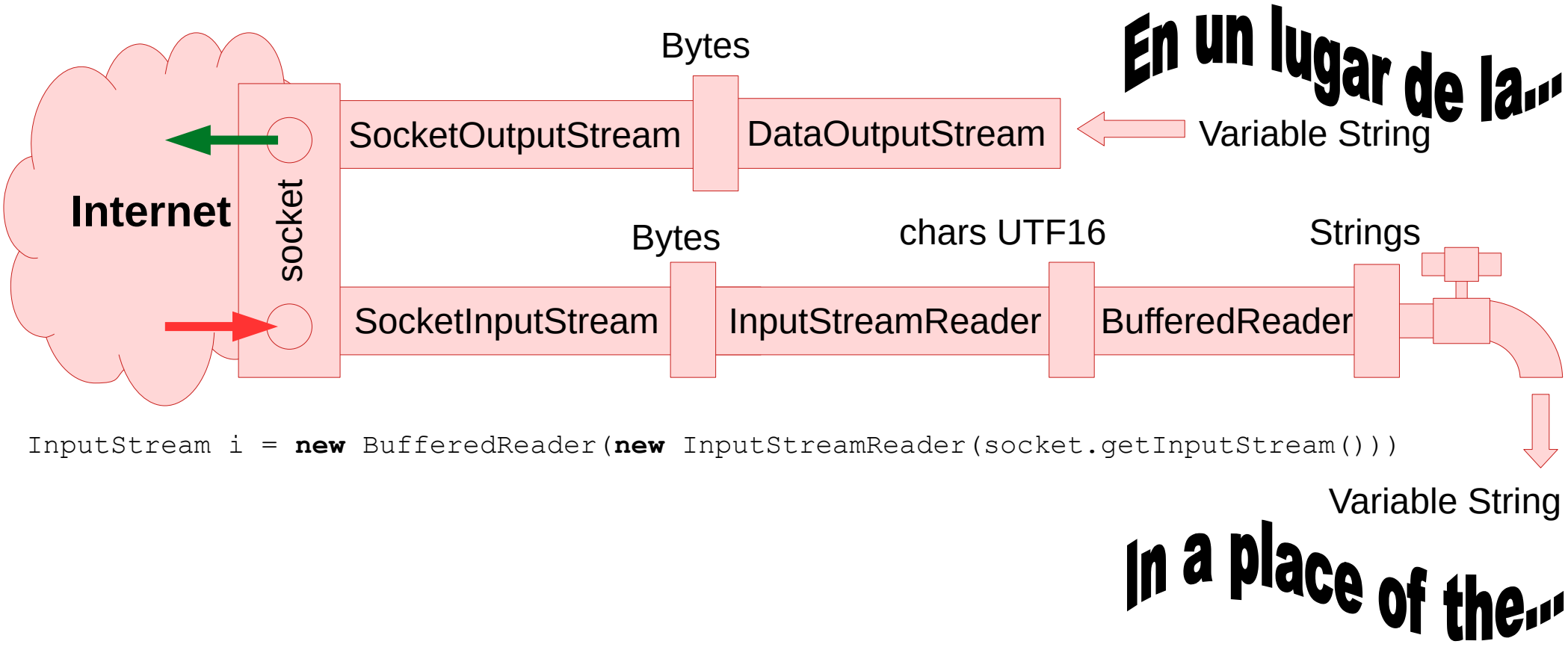
Envando caracteres de un ordenador a otro




```
InputStream i = new InputStreamReader(socket.getInputStream())
```

Reader y Writer

Enviando cadenas de un ordenador a otro



Ejemplo: obtención de página web por HTTP

```
public class HTTPGetExample {

  public static void main(String[] args) throws IOException {
    Socket sc = new Socket("128.59.105.24",80);

    OutputStream flujoSalida = sc.getOutputStream();
    String peticion = "GET /~fdc/sample.html HTTP/1.0\r\n\r\n";
    flujoSalida.write(peticion.getBytes());

    InputStream flujoEntradaBytes = sc.getInputStream();
    InputStreamReader flujoEntradaCars = new InputStreamReader(flujoEntradaBytes);
    BufferedReader flujoEntradaAcumulado = new BufferedReader(flujoEntradaCars);

    String s = flujoEntradaAcumulado.readLine();
    while (s != null) {
      System.out.println(s);
      s = flujoEntradaAcumulado.readLine();
    }
  }
}
```

Scanner

Esta sección se dedica a la clase `Scanner` y a las posibilidades que ofrece para extraer diferentes tipos de datos de los diferentes flujos de entrada.

Scanner

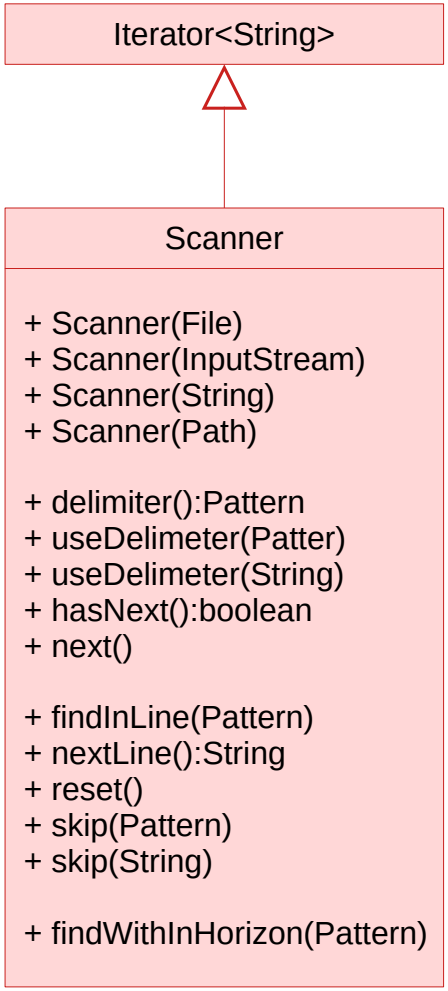
La clase `Scanner` de Java, de `java.util`, permite extraer diferentes tipos de datos de manera secuencial mientras se recorre un `InputStream`.

Para facilitar su uso sobre `InputStream` comunes, `Scanner` tiene constructores directos sobre ficheros, rutas a ficheros o `String`.

Básicamente, `Scanner` se usa de dos formas:

- Para extraer cadenas delimitadas por un elemento delimitador.
- Para extraer cadenas delimitadas por fin de línea.

Así, `Scanner` siempre tiene una cadena en curso que está delimitada por algún elemento.



Extraer datos usando un delimitador

Por defecto, `Scanner` usa el espacio como delimitador. Se puede cambiar el delimitador en cualquier momento, lo que permite leer ficheros de estructura conocida.

```
String s = "En un lugar de la Mancha, de cuyo nombre " +
           "no quiero acordarme, hace no mucho...";

Scanner scanner = new Scanner(s);
scanner.useDelimiter(",");

while (scanner.hasNext())
  System.out.println("Scanner String:" + scanner.next());

scanner.close();
```

Además, el delimitador se define mediante una expresión regular, lo cual es muy versátil.

Expresiones regulares

La siguiente tabla recoge un brevísimo resumen de los símbolos utilizados para crear expresiones regulares.

Expresión	Explicación
a	La letra “a”
.	Comodín para cualquier carácter
[abc] [a-c]	Un carácter a elegir entre la “a”, la “b” y la “c”
[^abc] [^a-c]	Un carácter distinto a la “a”, la “b” y la “c”
^abc$	Entre el principio y el fin de la línea solo está la cadena “abc”
\b \B	Borde de palabra y no borde de palabra
a* a+ a?	Cero o más, una o más, cero o una apariciones de la “a”
a{3}, a{2,} a{1,5}	3 apariciones de la “a”, más de dos apariciones o entre 1 y 5
ab|cd	“ab” o “cd”
* \\ \{ \[...	Caracteres escapados
(abc)	Grupo de captura de la cadena “abc”
\1 \2 ...	Referencia al grupo de captura anterior 1, 2...
(?:abc)	Agrupamos “abc” pero no es grupo de captura
a(?=bc) (?!) (?<=) (?<!)	Encaja “a” si le sigue “bc”, pero “bc” no se procesa (no sigue/precede/no).
\t \n \r	Tabulado, retorno de carro e inicio de párrafo
\w \d \s	Palabra dígito o espacio
\W \D \S	No palabra, no dígito o no espacio

Extraer datos predefinidos y delimitados

`Scanner` incorpora métodos para extraer fácilmente los tipos predefinidos en Java, separados por el delimitador que se defina.

```
public static void main(String[] args) {

   Scanner sc = new Scanner("12.3 2.34 3.14 1.12 0.15");

   while (sc.hasNextDouble())
         System.out.println(sc.nextDouble());

}
```

```
12.3
2.34
3.14
1.12
0.15
```

Además, permite ajustar parámetros como el separador usado para los decimales (`useLocale`) o la base del número (`radix`).

Scanner
+ delimiter():Pattern + radix() + useDelimeter(Patter) + useDelimeter(String) + useRadix(int) + hasNext():boolean + hasNext(Pattern):boolean + hasNext(String):boolean + hasNextLine():boolean + hasNextBigDecimal():boolean + hasNextBigInteger():boolean + hasNextBoolean():boolean + hasNextByte():boolean + hasNextDouble():boolean + hasNextLong():boolean + hasNextFloat():boolean + hasNextInt():boolean + hasNextShort():boolean + next(String) + next(Patter) + nextBigDecimal():BigDecimal + nextBigInteger():BigInteger + nextBoolean():Boolean + nextDouble():double + nextFloat():float + nextInt():int + nextLine():long + nextLong():long + nextShort():short

Scanner

Cadenas que cumplen expresiones regulares

`Scanner` permite definir una expresión regular para comprobar si el siguiente elemento sigue un patrón. En caso de no encontrar encaje en la cadena en curso hay que pasar a la siguiente cadena (con `next`).

- `hasNext(Pattern)`, que comprueba si la cadena actual entre delimitadores cumple el patrón.
- `next(Pattern)`, que hace al Scanner pasar al siguiente elemento si cumple el patrón. Si no lo cumple lanza una excepción.

```
String test = "Esto es un ejemplo de
   búsqueda.\nEsto es otro ejemplo de
   búsqueda.";

Scanner scanner = new Scanner(test);
while (scanner.hasNext()) {
  while (scanner.hasNext("e.*o")) {
    String a = scanner.next("e.*o");
    System.out.println(a);
  }
  scanner.next();
}
```

```
ejemplo
ejemplo
```

Extraer cadenas delimitadas por fin de línea

Para facilitar el manejo de líneas, delimitadas por el carácter de fin de línea, `Scanner` define los siguientes métodos:

- `nextLine()`, que hace al Scanner devolver la siguiente línea.
- `hasNextLine()`, que consulta si hay siguiente línea.

Estos métodos son bloqueantes mientras se reciben datos de un `InputStream` (por ejemplo de consola o de Internet).

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

System.out.print("Ha escrito: ");
System.out.println(s);
```

Scanner

Buscar en líneas expresiones regulares

`Scanner` permite definir una expresión regular para extraer de la línea en curso patrones definido. En caso de no encontrar encaje en la línea en curso hay que pasar a la siguiente (con `nextLine`).

`findInLine(Pattern)` busca el patrón hasta el siguiente carácter de fin de línea. Si lo encuentra los devuelve y sino devuelve `null`.

Hay otros métodos que usan expresiones regulares cómo `skip` o `findWithInHorizon`.

Scanner se detiene en cada encaje de la expresión regular en la cadena en curso. Para buscar de una vez todos los encajes de una expresión regular en una cadena se debería usar la clase `Matcher` en vez de `Scanner`.

```
String test = "Esto es un ejemplo de
   búsqueda.\nEsto es otro ejemplo de
 búsqueda.";

Scanner scanner = new Scanner(test);
while (scanner.hasNextLine()) {
  String a = scanner.findInLine("e.");
  System.out.println(a);
  while (a != null)
    a = scanner.findInLine("e.");
    System.out.println(a);
    scanner.nextLine();
}
```

```
es
ej
em
e
ed
null
es
ej
em
e
ed
null
```

Ejemplo de scanner con expresión regular

El siguiente fragmento de código busca en cada línea el nombre de los métodos de una página web de la documentación de Oracle.

```
//Conectamos un Reader a una página web
URL url = new URL("https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html");
InputStreamReader reader = new InputStreamReader(url.openStream());

//Conectamos un scanner al reader
Scanner scanner = new Scanner(reader);

//Definimos una expresión regular para extraer el nombre de los métodos
Pattern pattern = Pattern.compile("memberNameLink\"><a href=\".*-\">([a-zA-Z]{1,15})</a>");

//Vamos leyendo línea a línea y en cada una de ellas buscamos la expresión regular
while (scanner.hasNextLine()) {
  String matchString = scanner.findInLine(pattern);

  //Si encontramos encaje de la expresión regular extraemos el grupo de captura definido
  if (matchString != null) {
    MatchResult match = scanner.match();
    String indexEntry = match.group(1);
    Sytem.out.println(indexEntry);
    } else {
      scanner.nextLine();
    }
}
```

Serialización

Esta sección se dedica a explicar los mecanismos que tiene Java para facilitar la persistencia de sus objetos.

Idea intuitiva

Los lenguajes de programación suelen ofrecer mecanismos de persistencia que permiten almacenar y recuperar de memoria secundaria objetos y estructuras de datos.

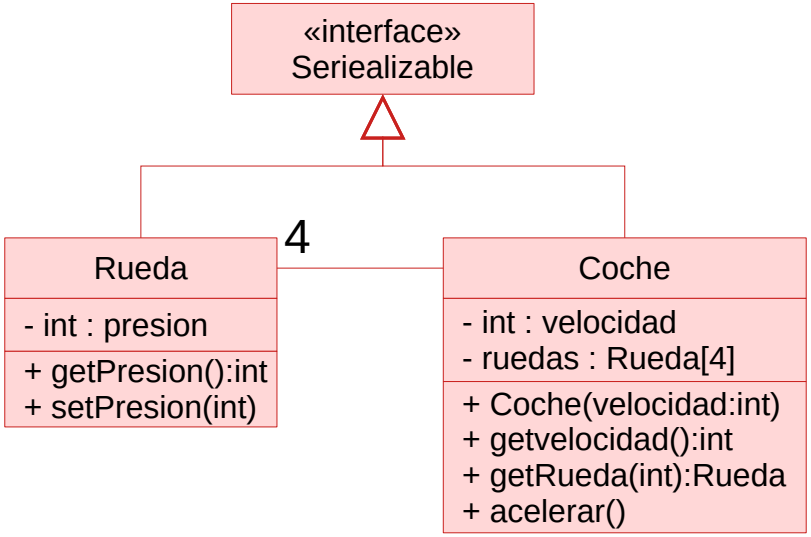
Las operaciones de guardar un objeto (por ejemplo una lista de objetos) en un fichero se denomina serialización.

Los lenguajes suelen proporcionar diferentes formas de serialización:

- A un formato binario propio
- A un formato legible (como XML o JSON).
- A una base de datos (hibernate).

Persistencia binaria: clases persistentes

El mecanismo de serialización por defecto de Java consiste en declarar que las clases implementan la interfaz Serializable.




```
public class Rueda implements Serializable {
  private int presion = 1;

  public int getPresion() {
    return presion;
  }

  public void setPresion(int presion) {
    this.presion = presion;
  }
}
```

```
public class Coche implements Serializable {

  private int velocidad;
  public Rueda [] ruedas = new Rueda [4];

  public Coche(int velocidad) {
    this.velocidad = velocidad;
    for (int cont = 0; cont < 4; cont++)
      ruedas[cont] = new Rueda();
  }

  public int acelerar() {
    velocidad++;
  }

  public int getVelocidad() {
    return velocidad;

  }

  public Rueda getRueda(int i) {
    return ruedas[i];
  }
}
```

Persistencia binaria: escritura a disco

Para serializar un objeto Serializable basta con utilizar `ObjectOutputStream`, para convertirlo en un flujo, y luego dirigirlo a donde se desee (un fichero, una cadena, Internet…).

```
public class Ejemplo {
  public static void main(String[] args) {
    try {
      Coche c = new Coche();
      c.getRueda(1).setPresion(4);
      c.acelerar();
      String fic = "c:/MiCoche.CocheObj";
      ObjectOutputStream salida = new ObjectOutputStream(new FileOutputStream(fic));
      salida.writeObject(c);
      salida.close();
      c.getRueda(1).setPresion(3);
      System.out.println(c.getRueda(1).getPresion());
    }
    catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

Persistencia binaria: lectura de disco

La serialización por defecto de Java da lugar a ficheros binario.

Para deserializar un objeto basta con utilizar `ObjectInputStream`, para convertir el flujo en un objeto y luego hacerle un casting.

```
public class Ejemplo {
  public static void main(String[] args) {
    try {
      String fic = "c:/MiCoche.CocheObj";
      ObjectInputStream entrada = new ObjectInputStream(new FileInputStream(fic));
      Coche c_copia = (Coche) entrada.readObject();
      System.out.println(c_copia.getVelocidad());
      System.out.println(c_copia.getRueda(1).getPresion());
    }
    catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

```
2
4
```

Persistencia en XML: clase persistente

Java también permite serializar a archivos de tipo XML. En este caso es importante que la clase sea pública, que haya un constructor sin parámetros y que cada propiedad tenga getter y setter.

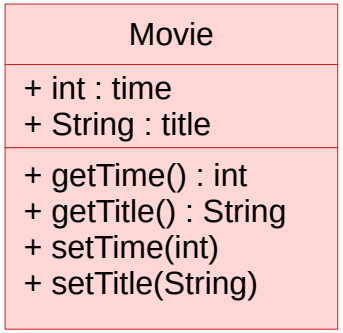
```
public class Movie {
    private int time;
    private String title;

    public void setTime(int time) {
      this.time = time;
    }

    public void setTitle(String title) {
      this.title = title;
    }

    public int getTime() {
      return time;
    }

    public String getTitle() {
      return title;
    }
}
```



Persistencia en XML: escritura a disco

Para la serialización de un objeto a XML se usa la clase XMLEnoceder.

```
void main(String [] args]) {
     Movie p1 = new Movie();
     p1.setTitle("Los Goonies");
     p1.setTime(95);

     Movie p2 = new Movie();
     p2.setTitle("StarWars");
     p2.setTime(120);

     List <Movie> moviesList=new LinkedList<>();
     moviesList.add(p1);
     moviesList.add(p2);

     try{
          XMLEncoder encoder = new XMLEncoder(
                                  new BufferedOutputStream(
                                    new FileOutputStream("pelis.xml")));

          encoder.writeObject(moviesList);
          encoder.close();

     }catch(FileNotFoundException fileNotFound){

          System.out.println(fileNotFound.getMessage());

   }
}
```

Persistencia en XML: fichero generado

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_141" class="java.beans.XMLDecoder">
 <object class="java.util.LinkedList">
  <void method="add">
   <object class="ejemplostema7.Movie">
    <void property="time">
     <int>95</int>
    </void>
    <void property="title">
     <string>Los Goonies</string>
    </void>
   </object>
  </void>
  <void method="add">
   <object class="ejemplostema7.Movie">
    <void property="time">
     <int>120</int>
    </void>
    <void property="title">
     <string>StarWars</string>
    </void>
   </object>
  </void>
 </object>
</java>
```

película 1: Los Goonies

película 2: Starwars

lista con dos películas

Persistencia en XML: lectura de disco

Para la deserialización de un objeto desde XML se usa la clase XMLDecoder.

```
void main(String [] args]) {

     try {

       XMLDecoder decoder = new XMLDecoder(
                                  new FileInputStream("pelis.xml"));

       List moviesList = decoder.readObject();
       decoder.close();
       System.out.print(moviesList);

     } catch(FileNotFoundException fileNotFound){
         System.out.println(fileNotFound.getMessage());
   }
}
```

```
Los Goonies
StarWars
```

Persistencia en XML y encapsulación

La serialización en XML rompe la encapsulación al obligar a poner setter a todas las propiedades.

Para evitar este problema, Java introduce la clase PersistenceDelegate que ayuda a especificar estos casos especiales.

En particular, la clase DefaultPersistenceDelegate permite especificar los parámetros del constructor. Así, se pueden omitir los setter si el constructor permite fijar las propiedades.

```
public class Movie {
  final private int time;
  final private String title;

  public Movie(int time, String title) {
    this.time = time;
    this.title = title;
  }

  public int getTime() {
    return time;
  }

  public String getTitle() {
    return title;
  }
}

...

Movie m = new Movie(23,"Dune");

XMLEncoder encoder = new XMLEncoder(
  new BufferedOutputStream(
    new FileOutputStream("pelis.xml")));

String [] constParams = {"time","title"};
encoder.setPersistenceDelegate(Movie.class,
  new DefaultPersistenceDelegate(constParams));

encoder.writeObject(movie);
encoder.close();

...
```

Otras clases para la gestión de ficheros

Esta sección se dedica a otras clases que permiten la gestión de los ficheros.

Otras clases para la gestión de ficheros

Idea intuitiva de RandomAccessFile

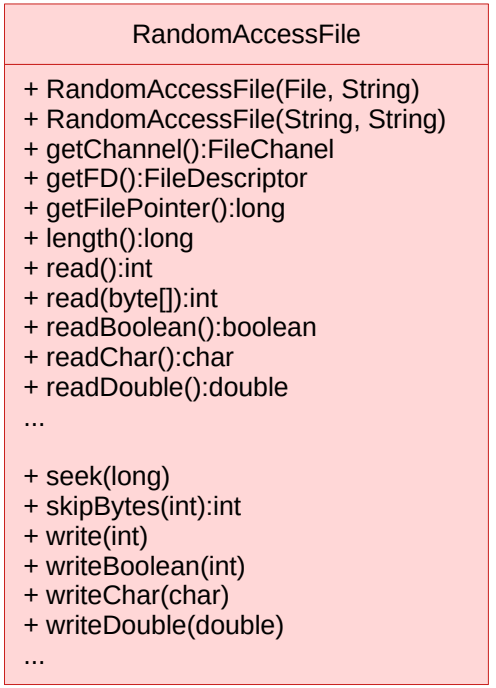
La clase `RandomAccessFile` facilita la lectura y escritura no secuencial de ficheros.

`RandomAccessFile` trata al fichero cómo un array en el que puede escribir o leer usando un puntero a una posición (`FilePointer`).

Permite leer o escribir bytes, caracteres Unicode (8 o 16) o tipos primitivos de Java.

Si se intenta leer más allá de la posición de fin de fichero se lanza una `EOFException`.

Si se produce cualquier otro error de lectura o escritura se lanza una `IOException`.

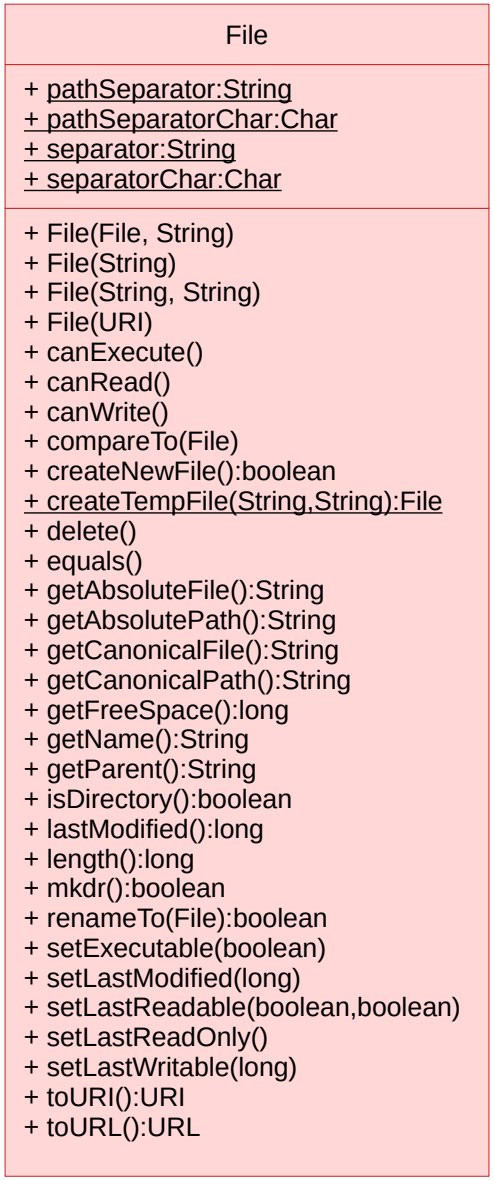


Otras clases para la gestión de ficheros

Idea intuitiva de File

Un objeto File es una representación de un archivo o directorio y permite:

- Consultar y cambiar propiedades:
  - Nombre
  - Tipo (archivo o directorio)
  - Permisos (lectura, escritura, ejecución)
  - Fecha de modificación
  - Tamaño
  - Ruta
- Crearlo, borrarlo.



Otras clases para la gestión de ficheros

Idea intuitiva de Files

La clase Files contiene muchos métodos estáticos para la gestión de archivos. Con Files se puede:

- Copiar, mover y borrar archivos.
- Crear directorios, ficheros, enlaces.
- Crear ficheros temporales.
- Consultar propiedades y permisos.
- Leer todo el fichero en un array.
- Escribir un array a un fichero.

Files
+ copy(Path, Path):Path + createDirectory(Path):Path + createFile(Path):Path + createLink(Path):Path + createSymbolicLink(Path):Path + createTemDirectory(Path):Path + createTemFile(Path):Path + delete(Path) + getAttribute(Path,String) + getFileStore(Path):FileStore + getOwner(Path):UserPrincipal + isDirectory(Path):boolean + isExecutable(Path):boolean + isHidden(Path):boolean + isReadable(Path):boolean + isRegularFile(Path):boolean + isSameFIle(Path,Path):boolean + isSymbolicLink(Path):boolean + isWritable(Path):boolean + move(Path, Path):Path + newBufferedReader(Path):BufferedReader + newBufferedWriter(Path):BufferedWriter ... + readAllBytes(Path):byte[] + readAllLines(Path):List<String> ... + setLastModifiedTime(Path path):Path + setOwner(Path path, UserPrincipal owner):Path + size(Path):long + walkFileTree(Path):Path + write(Path,bytes[])

Referencias

- Scanner [web]
- Expresiones regulares [tutorial][herramienta][juego]