Unidad 7: Entrada/Salida. Persistencia. Serialización

Metodología de la Programación

Curso 2021-2022

© *Candi Luengo Díez , Francisco Ortín Soler y José Manuel Redondo López, Alberto M. Fernandez Álvarez*

Bibliografía

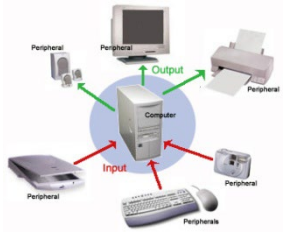
Lectura del Tutorial **Lesson: Basic I/O**
que se puede encontrar online en:

http://download.oracle.com/javase/tutorial/essential/io/index.html

En este tema veremos como…

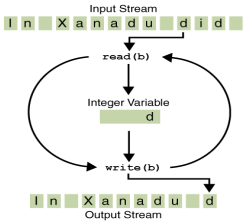
- Obtener información de la API de Java para **leer y escribir** información a/desde **fuentes externas**.
- Distinguir entre **fuentes de datos** de tipo **carácter** y **binario**.
- Saber cómo crear una **cadena de filtros** para la lectura/escritura de los datos.
- Aprender a utilizar la **persistencia de objetos.**

Entrada / Salida



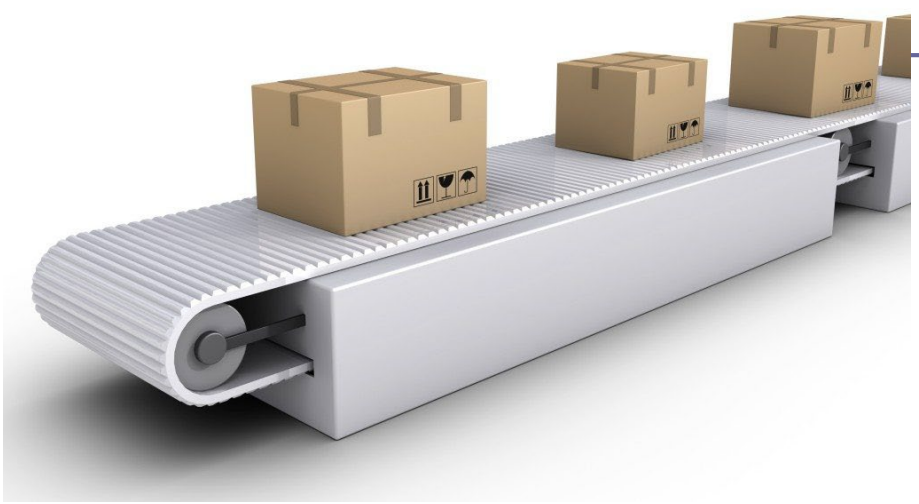
- Los programas normalmente procesan y generan información
- Esta información puede ser:
  - Leer de una fuente de entrada
  - Escribir en un destino de salida
- Un ejemplo es el compilador de Java:

Streams



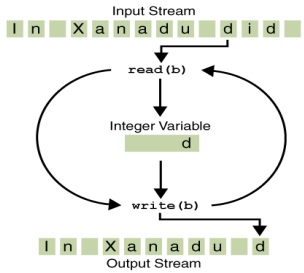
- Estas fuentes (de entrada) y destinos (de salida) son denominados **Streams** (flujos).
- Un stream es “algo” donde se puede leer o escribir la información .
- Ejemplos de streams son:
  - Ficheros en disco
  - La red
  - Otro programa
  - La memoria principal
  - Un dispositivo periférico
- Los Streams proporcionan transparencia.

Streams …….metáforas

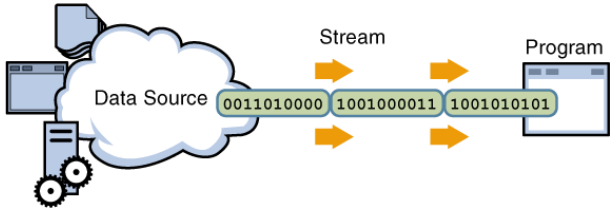


End of the file token (EOF)

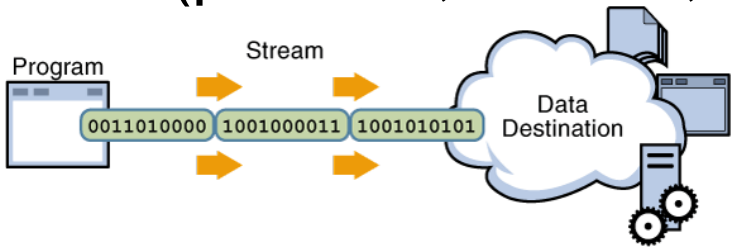
Input & Output Streams

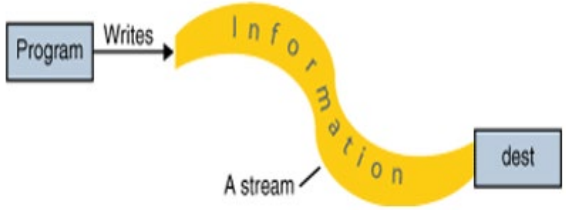


- Un **input stream** proporciona una secuencia de datos procedente de una fuente que una aplicación puede leer (teclado, fichero, memoria, red, …)



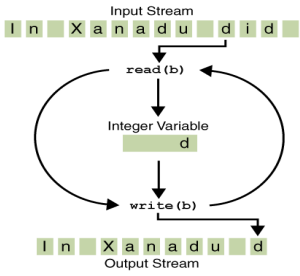
- Un **output stream** permite que una aplicación escriba una secuencia de datos dirigida a un destino (pantalla, fichero, …)





- Todas las clases **stream** están en `java.io`

Métodos en InputStream y OutputStream



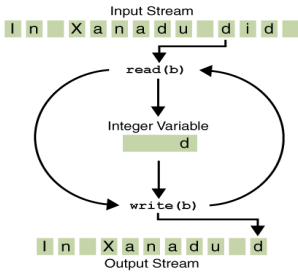
InputStream

- available() : int
- close() : void
- mark(int) : void
- markSupported() : boolean
- read() : int
- read(byte[]) : int
- read(byte[], int, int) : int
- reset() : void
- skip(long) : long

OutputStream

- close() : void
- flush() : void
- write(byte[]) : void
- write(byte[], int, int) : void
- write(int) : void

Leyendo de un InputStream



InputStream, el tipo general
FileInputStream, un tipo especializado

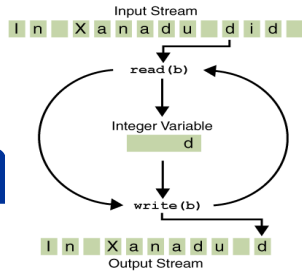
```
private void run() throws IOException {

    InputStream in = new FileInputStream("data.dat");
    try {
        int b;
        while ( (b = in.read()) != -1 ) {
            System.out.println(b);
        }
    } finally {
        in.close();
    }
}
```

Se leen bytes, y se devuelven como un int

Cuando se acaban los datos read() devuelve -1, esa es la marca para parar

Es un recurso y debe ser cerrado, finally obligatorio

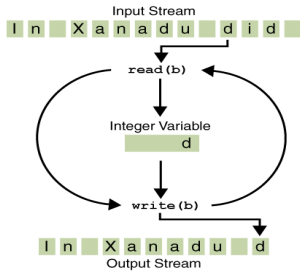
Escribiendo a un OutputStream



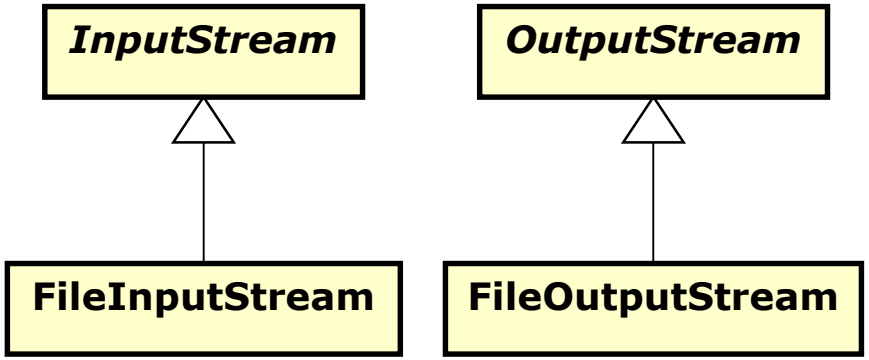

```
private void run() throws IOException {

    OutputStream out = new FileOutputStream("data2.dat");
    try {
        for(byte b = -128; b < 127; b++) {
            out.write(b);
        }
    } finally {
        out.close();
    }
}
```

El manejo de excepciones omitido para simplificar

Bajo nivel de abstracción



- Los métodos para leer y escribir datos binarios con `InputStream` y `OutputStream` ofrecen un **¡nivel muy bajo de abstracción!**
  - `int read()` o `int read(byte[] data)` →`Lee data.length bytes y los almacena en el array`
  - `void write(int)` o `void write(byte[] data)` → `Escribe data.length bytes del array al flujo de salida`
- ¿Como se traducen datos binaros (p.e. integers, longs) a `byte[]`?
- También se necesita leer y escribir otros datos como… (`String`, `int`, `float`, `double`…)



Ejemplo de bajo nivel

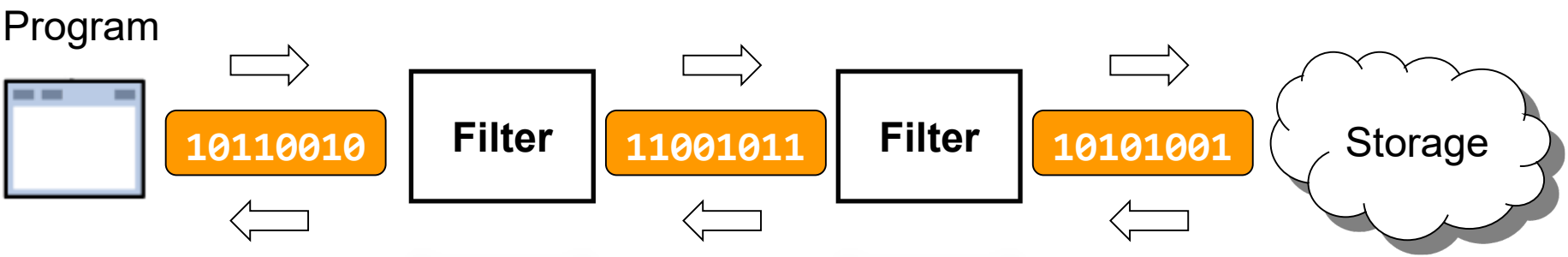
```
private void run() throws IOException {
    InputStream in = new FileInputStream( "data.dat" );
    try {
        i = readInteger( in );
        d = readDouble( in );
        b = readByte( in );
        c = readChar( in );
    }
    finally {
        in.close();
    }
```

```
  private double readDouble(InputStream in) throws IOException {
      long res = 0;
      res = (long)(in.read() << 56);
      res |= (long)(in.read() << 48);
      res |= (long)(in.read() << 40);
      res |= (long)(in.read() << 32);
      res |= (long)(in.read() << 24);
      res |= (long)(in.read() << 16);
      res |= (long)(in.read() << 8);
      res |= (long)(in.read());
      return Double.longBitsToDouble( res );
  }
```

Es una pesadilla leer los tipos básicos de java

Flujos filtrados (Filter Streams)

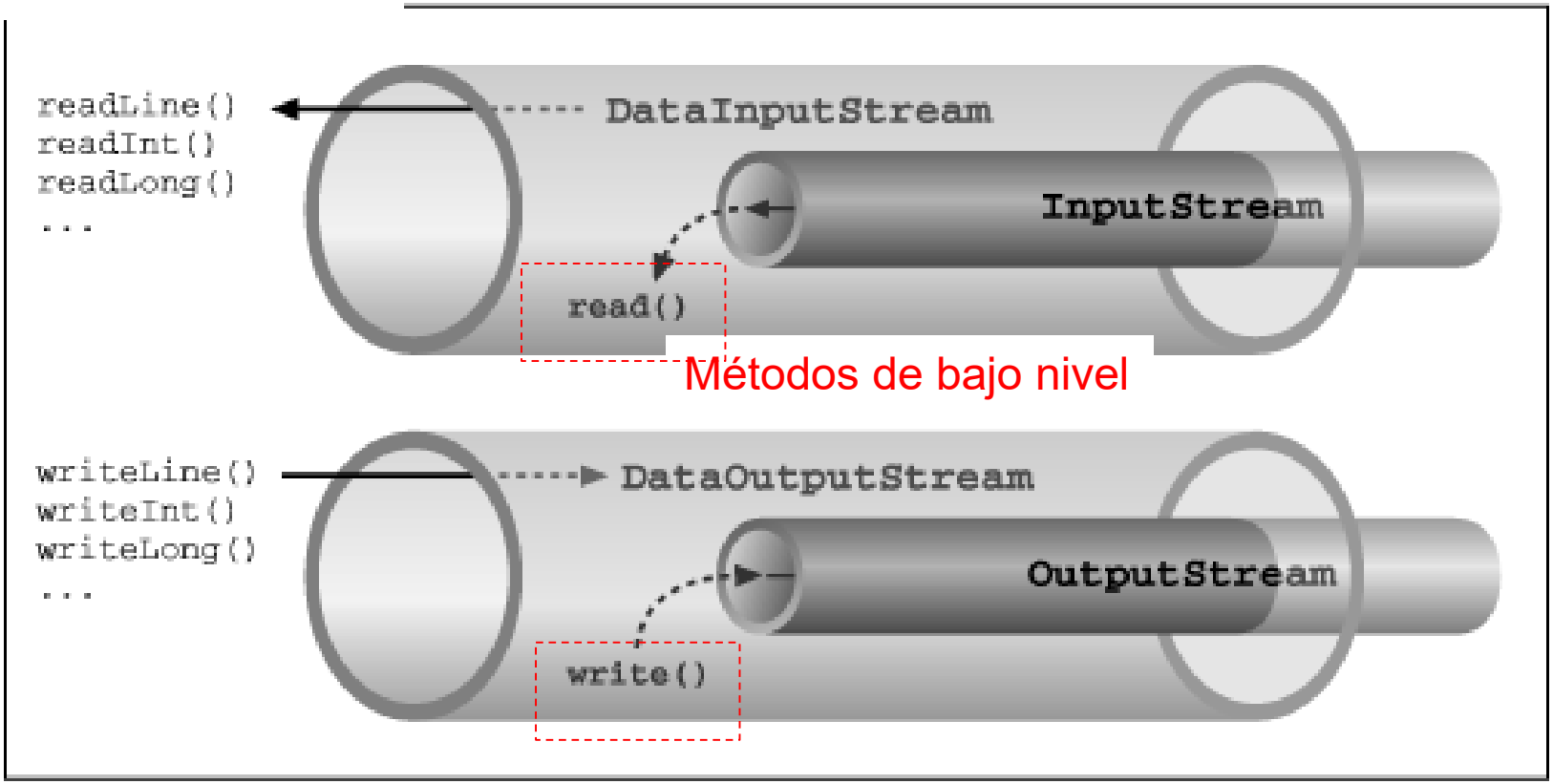
- Para ofrecer un nivel de abstracción más alto, Java ofrece **Filter Streams** (también llamados flujos de procesamiento *Processing Streams*)
- Un Filter Stream contiene otro stream, que lo utiliza como su principal fuente de datos, y transforma los datos por el camino o proporciona funcionalidad adicional
- **Son conectados por su constructor**



Flujos filtrados (uno de ellos)

- **Clases DataInputStream y DataOutputStream**

Métodos de alto nivel



Ejemplo leyendo con DataInputStream

```
private void run() throws IOException {
    String file = "data.dat";
    DataInputStream in = new DataInputStream( new FileInputStream( file ));

    try {
        i = in.readInt();
        d = in.readDouble();
        b = in.readByte();
        c = in.readChar();
    }
    finally {
        in.close();
    }
```

Los filtros son conectados a través de los constructores

Los filtros ofrecen métodos extra con un alto nivel de funcionalidad

Leyendo texto con InputStream

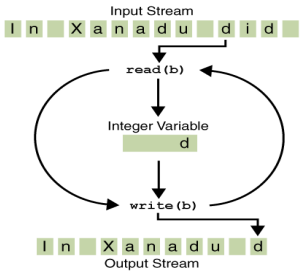
```
public static void main(String[] args) throws IOException {

    InputStream in = new FileInputStream( "sample.txt" );
    try {
        int ch;
        while( (ch = in.read()) != -1 ) {
            System.out.print( (char) ch );
        }
    } finally {
        in.close();
    }
}
```

Leer texto no es tan sencillo.

Un carácter no es (siempre) un byte

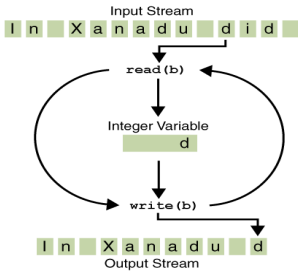
```
QuÃ© es Unicode

Unicode es un estÃ¡ndar para la codificaciÃ³n de caracteres.

El problema de las representaciones de caracteres existentes
```

Mismo texto, diferentes codificaciones



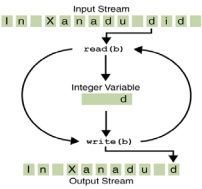
- "año" codificado en
  - ASCII 61 F1 6F
  - Windows-1252/latin1 61 F1 6F
  - GB18030 61 81 30 8A 39 6F
  - UTF8 61 C3 B1 6F
  - UTF16 00 61 00 F1 00 6F
  - UTF32 00 00 00 61 00 00 00 F1 00 00 00 6F
- Unicode es un conjunto de caracteres codificados en binario de diferentes formas (encodings)
  - utf8, utf16…

Texto en Unicode

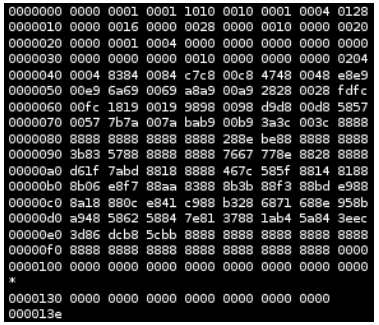


- Unicode es una tabla grande en la que cada carácter tiene 32 bits de ancho (4 bytes)
- Para ahorrar espacio de almacenamiento hay diferentes codificaciones para cada carácter
- Por ejemplo, en UTF-8 algunos caracteres ocupan 1 byte, otros 2 bytes, 3 y 4.
- Leer texto de bytes también es una pesadilla.
- Hay un filtro para eso : InputStreamReader

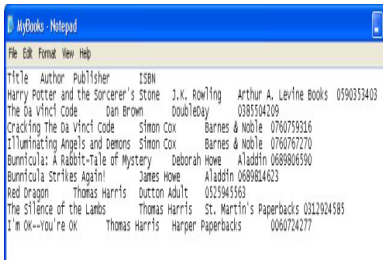
Streams Byte y Character



- Los **streams de byte** los usan los programas para realizar entradas y salidas de bytes (8-bit)
  - Todas las clases de stream byte descienden de las clases abstractas `InputStream` y `OutputStream`
  - Almacenan información binaria



- Los **streams de characteres** almacenan caracteres usando las convenciones Unicode
  - Todas las clases de stream character descienden de las clases abstractas `Reader` y `Writer`
  - El conjunto de caracteres local suele ser un super conjunto del ASCII de 8-bits
  - También se usan otros formatos (p.e, UTF-8, UTF-16, …)



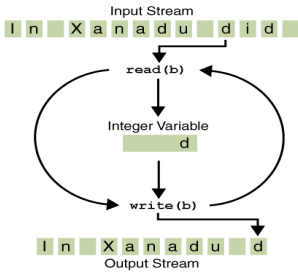
Leyendo texto con InputStreamReader

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

    Reader in = new InputStreamReader(
                    new FileInputStream( "sample.txt" ),
                    "UTF-8"  // codification scheme
                );
    try {
        int ch;
        while( (ch = in.read()) != -1 ) {
            System.out.print( (char) ch );
        }
    } finally {
        in.close();
    }
}
```

InputStreamReader lee bytes de InputStream pero devuelve caracteres

```
Qué es Unicode

Unicode es un estándar para la codificación de caracteres.

El problema de las representaciones de caracteres existente
```

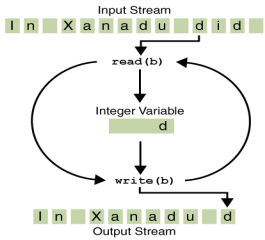
Detectando el End of File




```
int ch = 0;
while ((ch = in.read()) != -1) {
   System.out.print((char) ch);
}
```

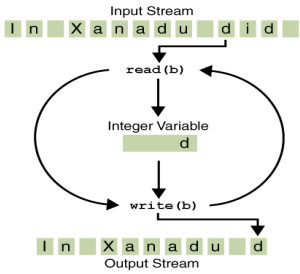
- read() devuelve un **int** en vez de un **character**, ¿por qué?
  - Un carácter es un subtipo de int
  - Con un rango 0..65535,
    - Cualquier valor en ese rango es un carácter válido, de esta forma ….
    - no es posible un “valor especial” para indicar el EoF (final de fichero)
  - Al promocionar el carácter a int hay más valores posibles
    - Entonces -1 es un valor int posible, y un valor de carácter imposible.
- El valor int devuelto
  - en el rango 0..65535 o -1
  - Se puede hacer el cast con seguridad

Tipos de Streams



- Las clases de flujos se dividen en dos grandes grupos:
  - **Flujos de bytes** (su nombre acaba en "Stream"). Por ellos "viajan" bytes.
  - **Flujos de caracteres** (su nombre acaba en "Reader" o en "Writer"). Por ellos viajan caracteres.
- **Funcionalmente**, los dividimos en:
  - **Flujos de origen o destino de datos**. Están conectados directamente con la fuente o destino de los datos.
  - **Flujos de procesamiento**. Son pasos intermedios en el proceso de la información (filtrar, comprimir, formatear...)

Métodos en Reader y Writer



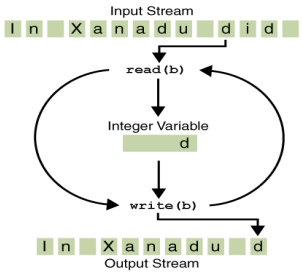
Reader

- close() : void
- mark(int) : void
- markSupported() : boolean
- read() : int
- read(char[]) : int
- read(char[], int, int) : int
- read(CharBuffer) : int
- ready() : boolean
- reset() : void
- skip(long) : long

Writer

- close() : void
- flush() : void
- write(char[]) : void
- write(char[], int, int) : void
- write(int) : void
- write(String) : void
- write(String, int, int) : void

Métodos en InputStream y OutputStream



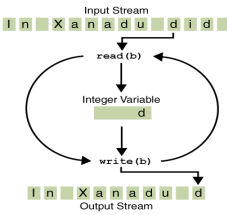
InputStream

- available() : int
- close() : void
- mark(int) : void
- markSupported() : boolean
- read() : int
- read(byte[]) : int
- read(byte[], int, int) : int
- reset() : void
- skip(long) : long

OutputStream

- close() : void
- flush() : void
- write(byte[]) : void
- write(byte[], int, int) : void
- write(int) : void

Ejemplo: Input (Flujo de caracteres)



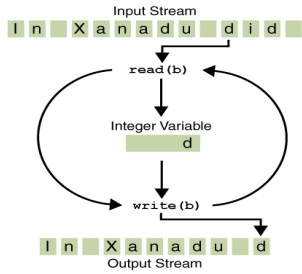

```
Reader in = new FileReader( "sample.txt" );
try {
    int ch;
    while( (ch = in.read()) != -1 ) {
        System.out.print( (char) ch );
    }
} finally {
    in.close();
}
```

Equivalente a esto

```
Reader in = new InputStreamReader(
                new FileInputStream( "sample.txt" )
            );
```

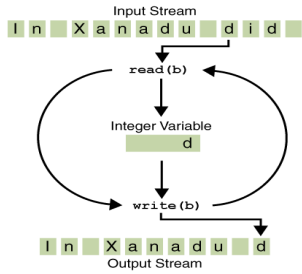
- Excepciones omitidas por simplicidad

Ejemplo: Input (Flujo de caracteres)



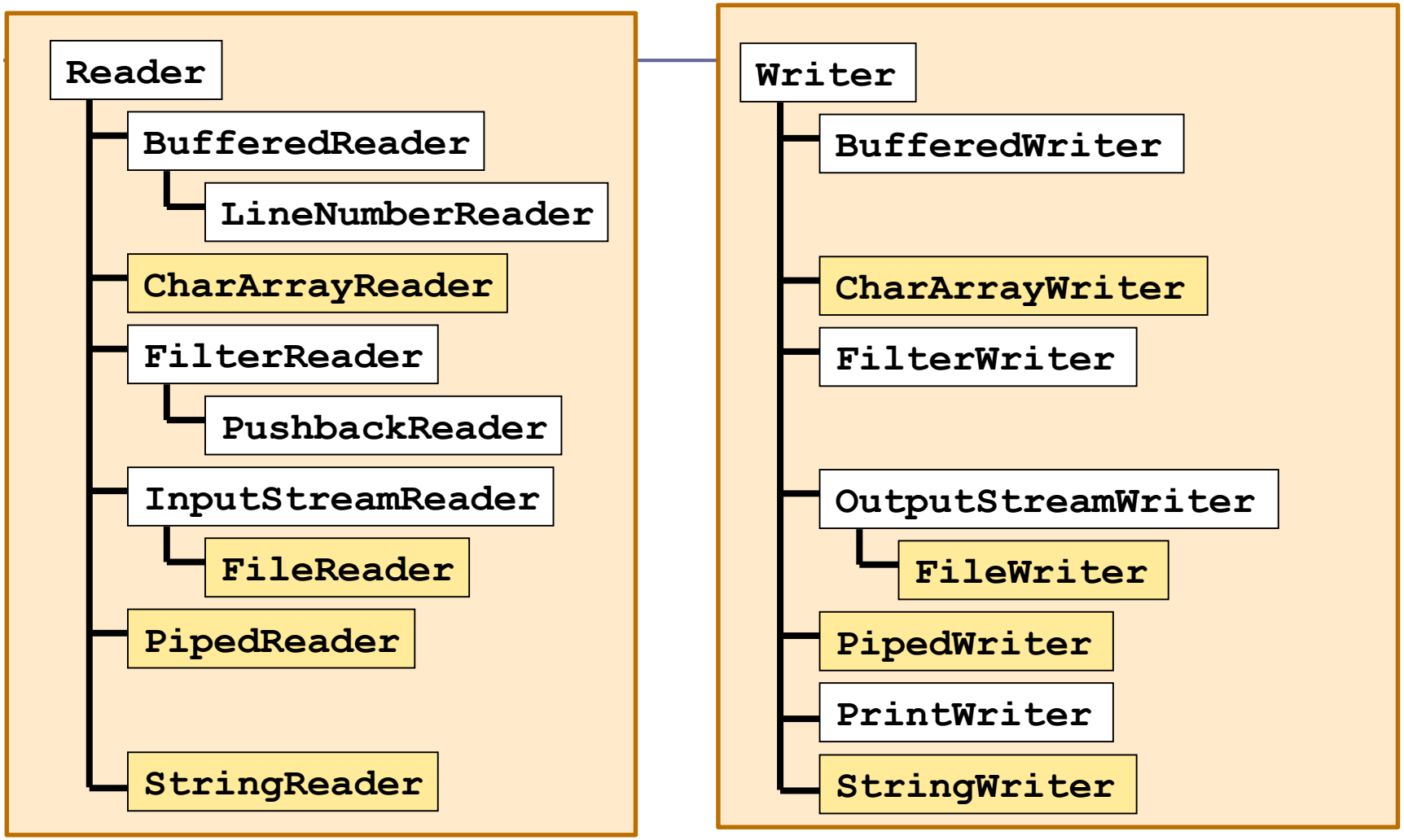

```
public static void main(String[] args) {
    try {

        Reader in = new FileReader("sample.txt");
        try {
            int ch;
            while ((ch = in.read()) != -1) {
                System.out.print((char) ch);
            }
        } finally {
            in.close();
        }

    } catch (FileNotFoundException nfe) {
        System.out.println("The file does not exist");
    } catch (IOException ex) {
        System.err.println("System error rerading the file");
    }
}
```

Ejemplo: Output (flujo de caracteres)



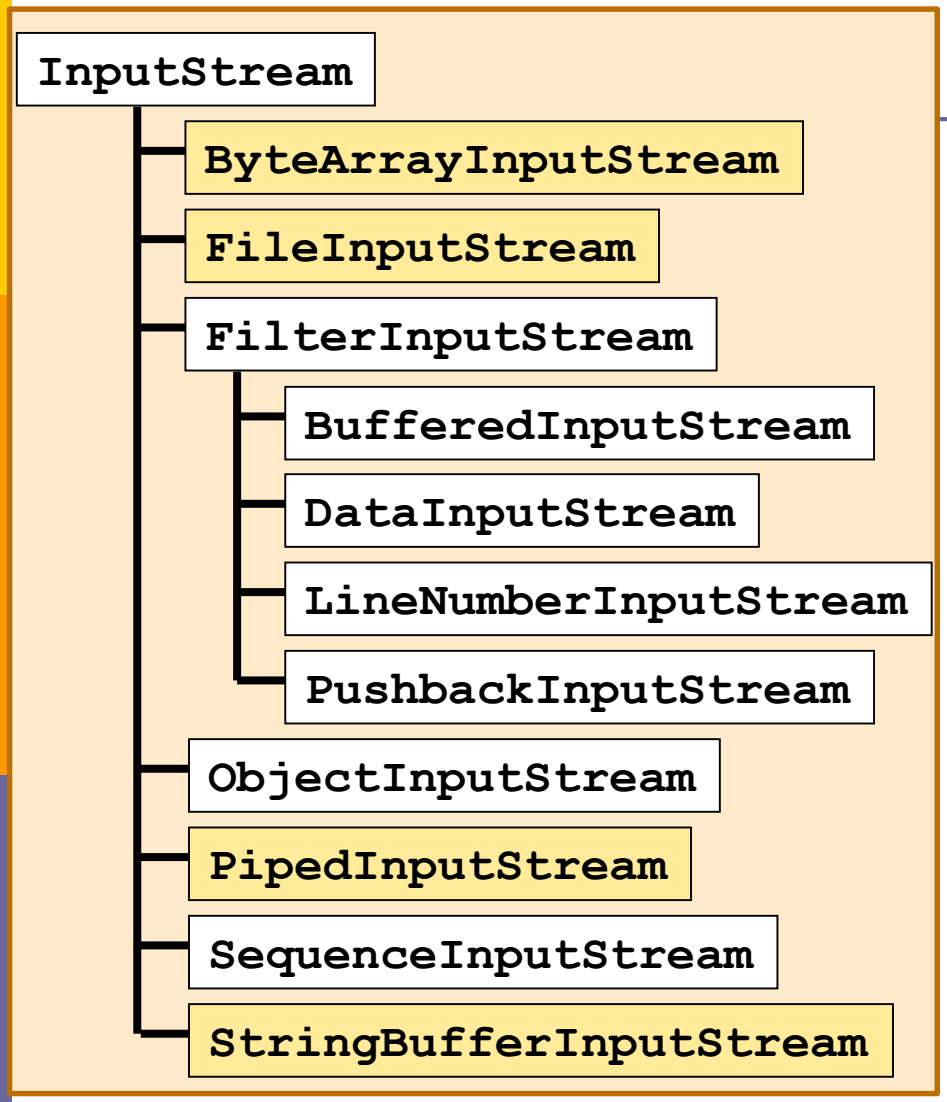

```
Writer file = new FileWriter( "table.txt" );
try {
    for (int i = 0; i <= 10; i++) {
        file.write(
                "2 ^ " + i + " = "
                + (int) Math.pow(2, i)
                + ".\r\n"
        );
    }
} finally {
    file.close();
}
```

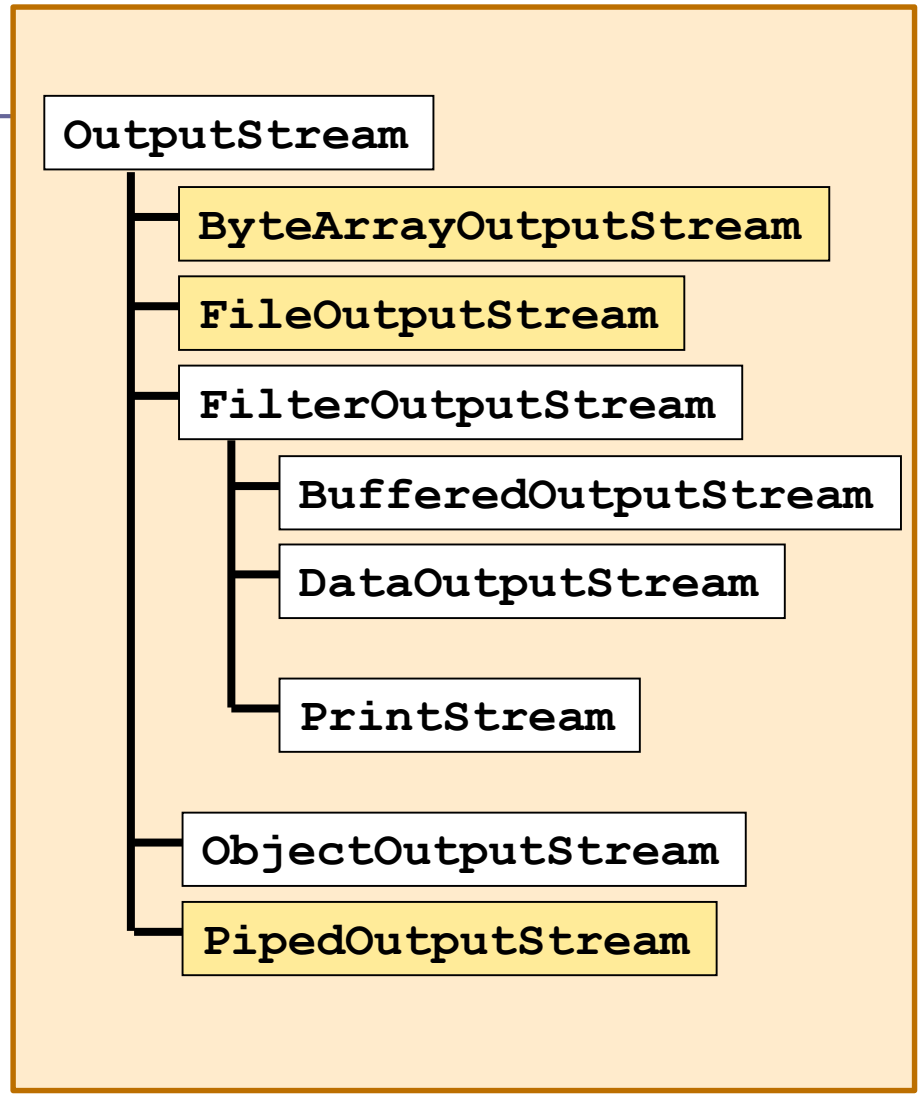
Clases de flujos de caracteres



- En amarillo son las clases que permiten **leer o escribir**
- En blanco son las clases que permiten algún tipo de **procesamiento**

Clases de flujos de bytes





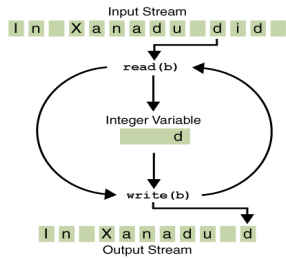
- En amarillo son las clases que permiten **leer o escribir**
- En blanco son las clases que permiten algún tipo de **procesamiento**

Typos de Streams

	Binary		Character	
	final	*processing*	*final*	*processing*
Input	***InputStream***		***Reader***	
	FileInputStream	BufferedInputStream	FileReader	BufferedReader
	StringBufferInputStream	DataInputStream	StringReader	Input**Stream**Reader
	ByteArrayInputStream	ObjectInputStream	CharArrayReader	*FilterReader*
	PipedInputStream	GZipInputStream	PipedReader	...
	...	*FilterInputStream*	...	
		...		
Output	***OutputStream***		***Writer***	
	FileOutputStream	BufferedOutputStream	FileWriter	BufferedWriter
	ByteArrayOutputStream	PrintStream	StringWriter	PrintWriter
	PipedOutputStream	DataOutputStream	CharArrayWriter	Output**Stream**Writer
	...	ObjectOutputStream	PipedWriter	*FilterWriter*
		GZipOutputStream	...	
		FilterOutputStream		
		...		

Italics means abstract class

Fuentes y destinos (final)



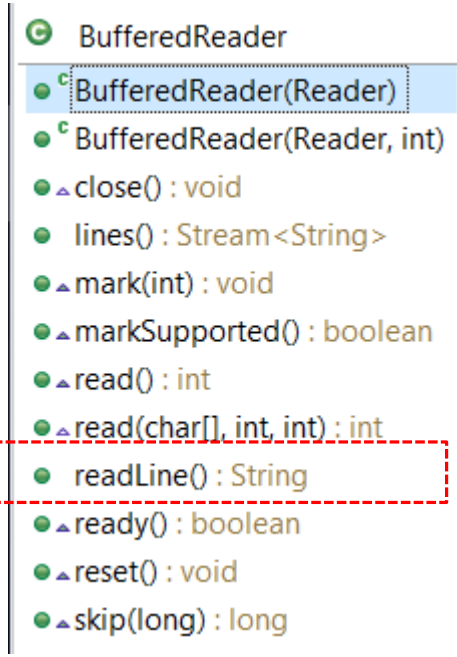
- Los streams pueden tener más **fuentes y destinos** en Java.
- Estos son los más comunes:
  - Files
  - Array de bytes (binario) o caracteres (texto)
  - String objetos
  - Console (`System.in`, `System.out` y `System.err`)
  - Pipes: Consiste en poner “cosas” por un extremo y salen por el otro.
  - Internet conexiones (`java.net`)

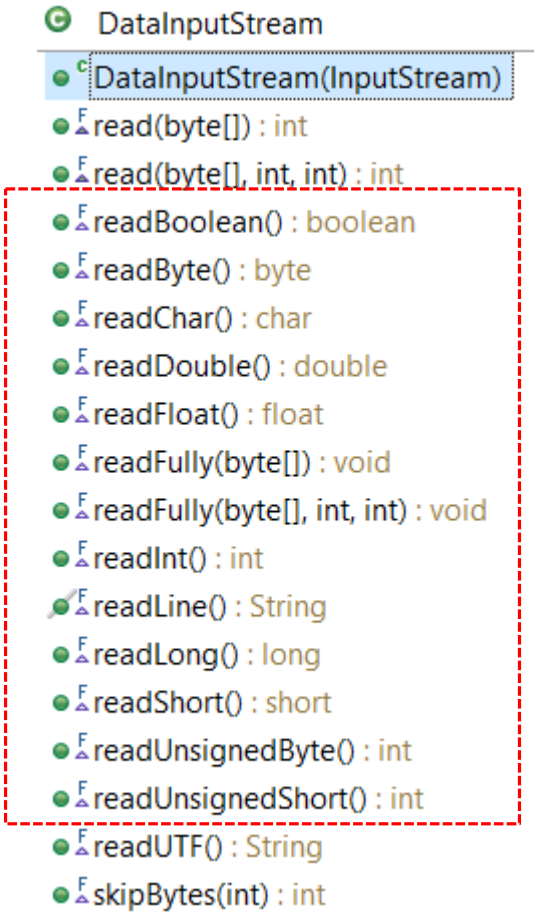
Flujos filtrados (Filter Streams)

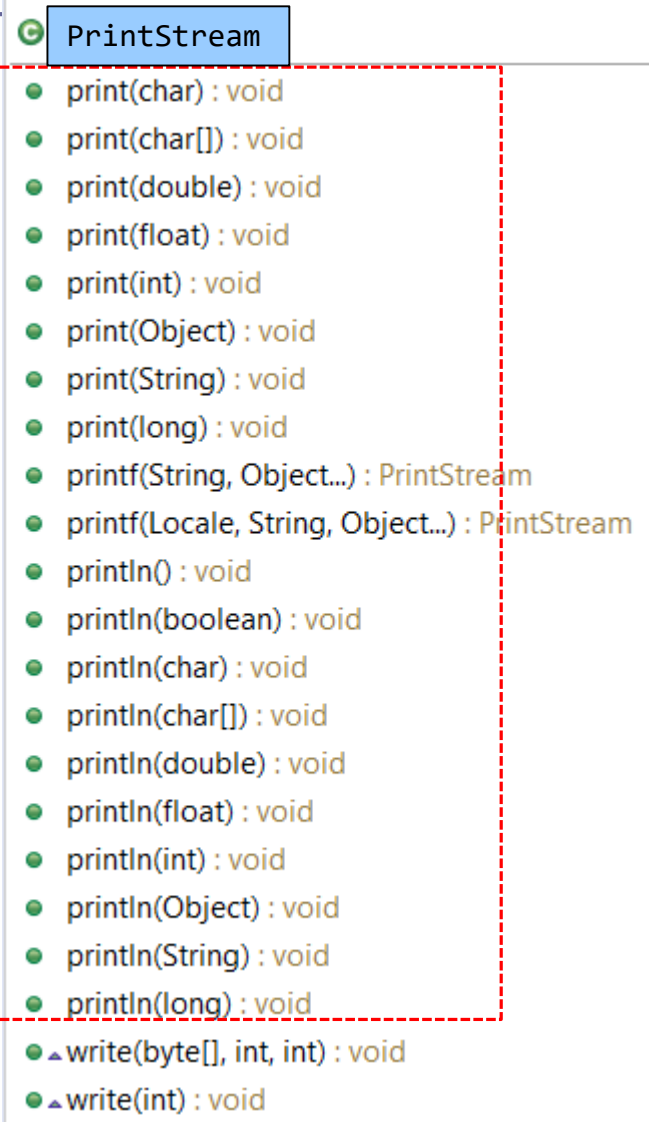
- Estos son los filtros más comunes
  - **Buffered**: Se leen o escriben datos a través de un buffer, ofreciendo un mejor rendimiento en tiempo de ejecución.
  - **Data**: Permite a una aplicación leer y escribir tipos de datos (p.e., `int`, `double`...).
  - **LineNumber**: Realiza un seguimiento del número de línea actual.
  - **Pushback**: Añade la funcionalidad "push back" o "unread" un dato.
  - **Print**: Añade la capacidad de imprimir las representaciones de los valores de los distintos datos, utilizando diferentes formatos (p.e., `System.out` y `System.err`).
  - **Zip**: Lee y escribe ficheros en formato ZIP.

Métodos añadidos por algunos filtros útiles

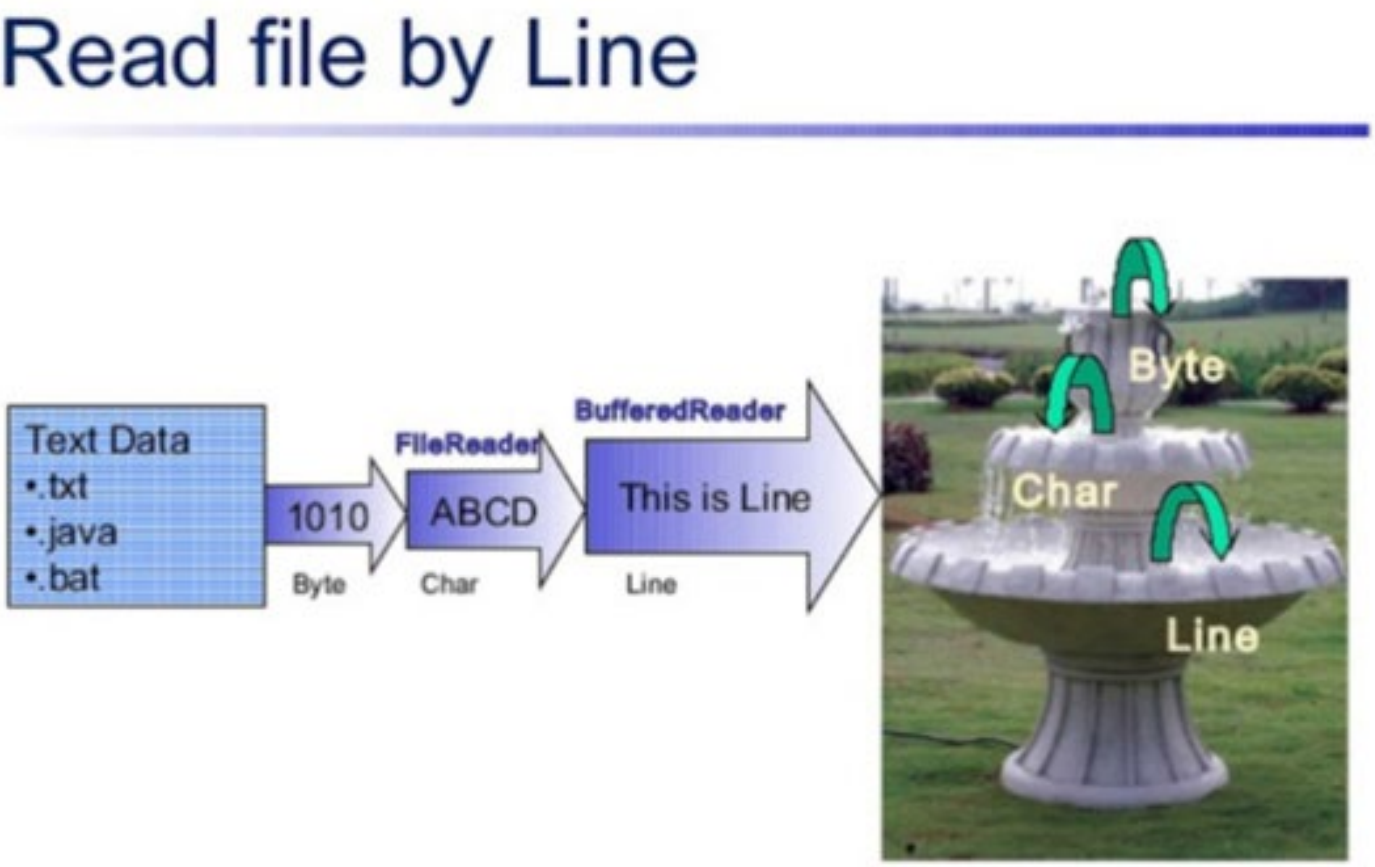
- BufferedReader
- DataInputStream
- PrintStream







Filtros: BufferedReader



Ejemplo: BufferedReader

BufferedReader acelera la lectura

```
BufferedReader reader = new BufferedReader(new FileReader("lorem-ipsum.txt"));
try {
    String line;
    while ( (line = reader.readLine()) != null) {
        System.out.println(line);
    }
} finally {
    reader.close();
}
```

BufferedReader también ofrece un método de nivel superior para leer líneas (strings)

Ejemplo: BufferedWriter

```
BufferedWriter writer = new BufferedWriter(new FileWriter("table.txt"));
try {

    for(int i = 0; i <= 10; i++) {
        String line = "2 * " + i + " = " + i*2;

        writer.write( line );
        writer.newLine();
    }

} finally {
    writer.close();
}
```

BufferedWriter acelera la escritura

BufferedWriter también ofrece un método de nivel superior para marcar el final de una línea

Ejemplo: Data Stream

```
int number = 123456;
DataOutputStream output = new DataOutputStream(
      new BufferedOutputStream(
            new FileOutputStream(args[0])));
output.writeInt(number);
output.close();

DataInputStream input = new DataInputStream(
      new BufferedInputStream(
                  new FileInputStream(args[0])));
number = input.readInt();
input.close();
```

Escribiendo tipos Java directamente

Usando un buffer (rendimiento)

A un fichero binario

Leyendo tipos Java directamente

Usando un buffer (rendimiento)

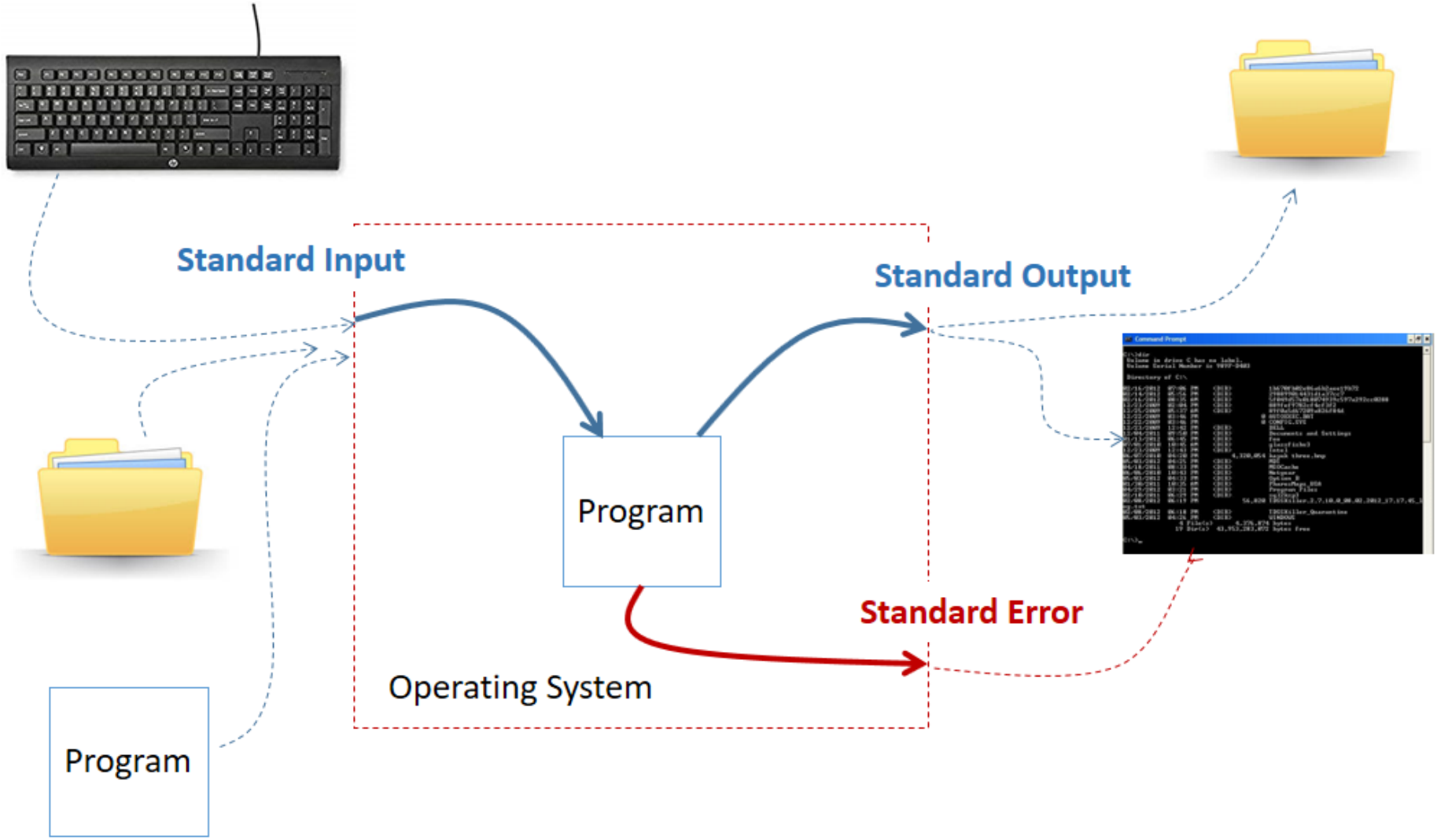
De un fichero binario

Existe un método diferente para cada tipo básico

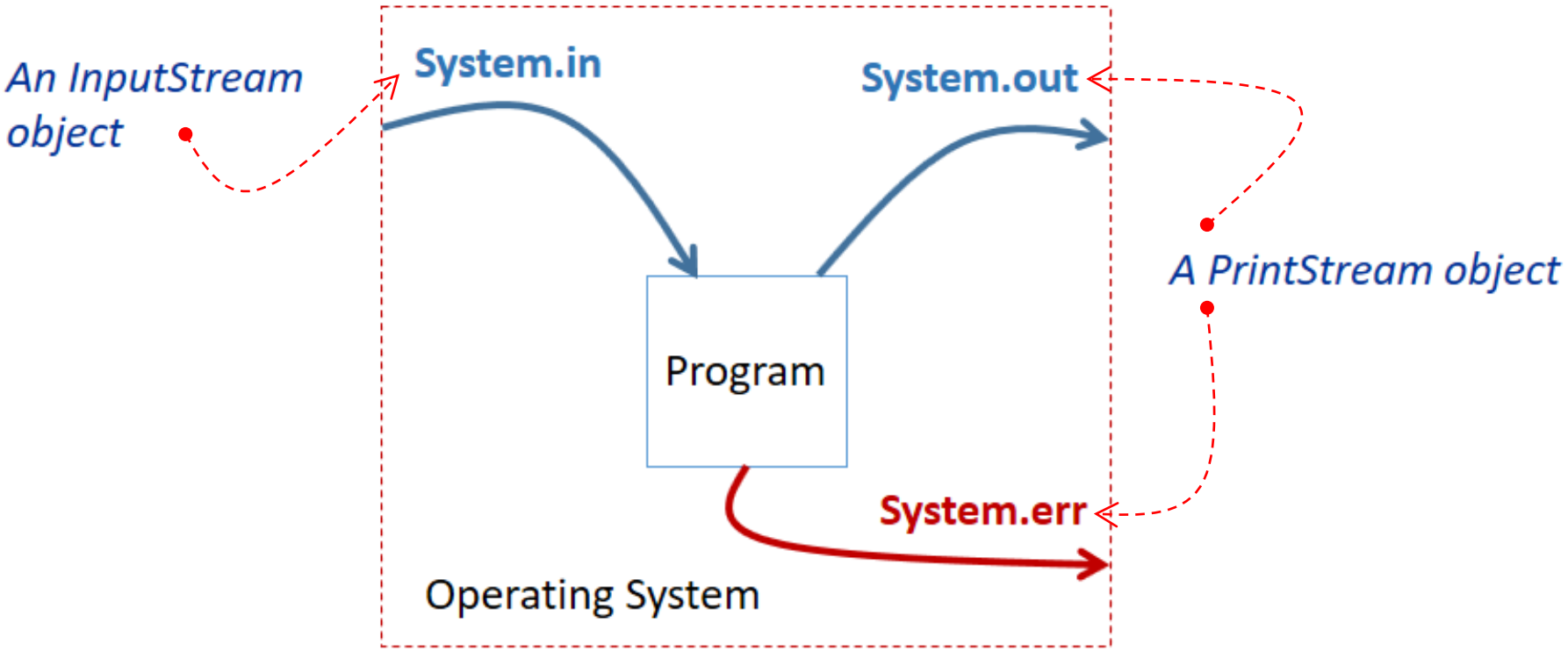
Flujos estándar

- Todos los lenguajes de programación proporcionan soporte para la E/S estándar donde un programa puede obtener datos por teclado y luego mostrar la salida en la pantalla del ordenador.
- **Entrada estándar**− normalmente conectada al teclado, representada como **System.in**.
- **Salida estándar**− normalmente la pantalla, representada como **System.out**.
- **Salida de error**− usada para mostrar los errores, normalmente la pantalla, representada como **System.err**.

Standard streams

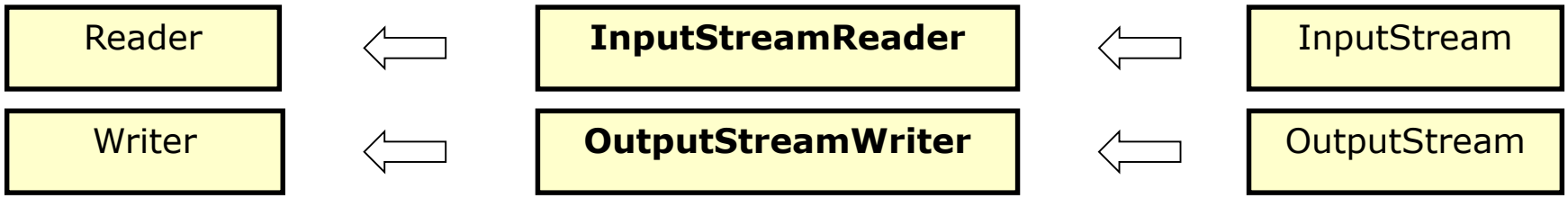


Standard streams



Leyendo desde la consola

- **¿ Como se pueden leer enteros desde la consola?**
- `System.in` es una instancia de `InputStream` (`System.out` y `System.err` son instancias de `PrintStream`)
- Pero, **`InputStream` es binario**, y la entrada estándar es texto
- Los flujos de caracteres (`Reader` y `Writer`) **fueron introducidos en Java 1.1**
  - No existían en Java 1.0
- Así que añadieron **dos clases de conversión** de binario a texto



Leyendo desde la consola

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

String line = keyboard.readLine();
```

Ofrece el método readLine()

Convierte Binario a Texto

Alternativa (usar scanner):
http://www.mkyong.com/java/how-to-read-input-from-console-java/

Uso del filtro GZipInputStream

```
BufferedReader in =
    new BufferedReader(
        new InputStreamReader(
            new GZIPInputStream(
                new FileInputStream( "data.dat.gz" )
            )
        )
    );

try {
    String line;
    while( (line = in.readLine()) != null) {
        // ...
    }
} finally {
    in.close();
}
```

Buffer para aumentar el rendimiento

Convierte Binario a Texto

Filtro que descomprime

Un fichero binario comprimido

Fin de la detección de datos

Líneas como strings

Creando nuevos filtros

- El programador puede crear sus propios filtros
- El único requerimiento es que sea un subtipo de
  - `Reader`
  - `Writer`
  - `InputStream`
  - `OutputStream`
- Sin embargo.....

Creando nuevos filtros

- Heredando de
  - `FilterReader`
  - `FilterWriter`
  - `FilterInputStream`
  - `FilterOutputStream`

es más sencillo

- La herencia es un mecanismo para la reutilización de código.

- Ejemplo: Encriptar / Desencriptar de cualquier stream binario

Filtro de encriptación

```
public class EncryptionOutputStream extends FilterOutputStream {

    public EncryptionOutputStream(OutputStream out) {
        super(out);
    }

    @Override
    public void write(int aByte) throws IOException {
        aByte = aByte == 255 ?  0 : aByte + 1; // Caesar cipher
        super.write( aByte );
    }

}
```

Encryptando Streams

```
String fileName = "lorem-ipsum.txt";

InputStream input =
        new BufferedInputStream(
            new FileInputStream( fileName )
        );

OutputStream output =
        new EncryptionOutputStream(
            new BufferedOutputStream(
                new FileOutputStream( fileName + ".crypt")
            )
        );

int aByte = 0;
while ( (aByte = input.read()) != -1) {
    output.write(aByte);
}

input.close();
output.close();
```

Filtro de desencriptación

```
public class DecryptionInputStream extends FilterInputStream {

    public DecryptionInputStream(InputStream in) {
        super(in);
    }

    @Override
    public int read() throws IOException {
        int theByte = super.read();

        if (theByte == -1) return -1; // End of file

        theByte = theByte == 0 ? 255 : theByte - 1; // Caesar decipher
        return theByte;
    }
}
```

Desencriptando Streams

```
String fileName = "lorem-ipsum.txt.crypt";

InputStream input =
    new DecryptionInputStream(
        new BufferedInputStream(
            new FileInputStream( fileName )
        )
    );

OutputStream output =
    new BufferedOutputStream(
        new FileOutputStream( fileName + ".decrypt")
    );

int aByte = 0;
while ( (aByte = input.read()) != -1) {
    output.write(aByte);
}

input.close();
output.close();
```

Persistencia de objetos

- **Persistencia de objetos** es la característica de un lenguaje o plataforma que hace que los estados de los objetos sobrevivan al proceso que los creó.
- Permite guardar los objetos específicos de una aplicación, por lo que en las siguientes ejecuciones de un programa pueden recuperarlos
- Los objetos pueden ser almacenados en:
  1. Ficheros
  2. Bases de datos
  3. Repositorios en la red

Serialización de objetos

- **Serialización de objetos** es el proceso de convertir un objeto en una serie de bits
- Esta serie de bits permitirá la recuperación del estado del objeto más adelante
- La serialización de objetos es una técnica utilizada para implementar **la persistencia de objetos**

Problemas de serialización

- Se debe hacer frente a algunos problemas cuando se serializan los objetos
  1. ¿Son serializables todos los atributos?
  2. ¿Son serializables las clases?
  3. ¿Qué sucede con los objetos que apuntan (referencia) a otros objetos. Son serializables también?
  4. ¿Qué formato es usado en la serialización? ¿Es compatible entre diferentes plataformas?
- Veremos cómo responde Java a estas preguntas

Serialización de objetos en Java

- En java los objetos se pueden serializar implementando la interface `java.io.Serializable`
  - Las clases más comunes en Java son `Serializables`
- Esta interfaz no tiene métodos
  - La serialización de objetos se maneja de forma transparente por la máquina virtual de Java
  - La interfaz `Externizable` permite definir como se realiza
- Si se intenta serializar un objeto no serializable se lanza la excepción `NotSerializableException`
- Solamente se pueden serializar los objetos
  - Las clases no son porque están ya en sus ficheros .class

Serialización de objetos en Java

- La serialización necesita un número de versión
  - Si no lo tenemos se muestra un aviso

```
8  public class Database implements Serializable  {
9
```

- Como no se serializa la clase (sólo el objeto), ¿qué pasaría si modificamos la clase y luego se desserializa el objeto?
  - serialVersionUID se utiliza para comprobar el .class

```
7
8  public class Database implements Serializable  {
9      private static final long serialVersionUID = 1L;
10
```

Atributos y Transient

- Por defecto, cada atributo es serializable
- Si se quiere que un atributo no sea serializable, se debe usar **transient**

```
public abstract class Item
                implements Serializable {
  // Serializable no tiene métodos para redefinir
  private String title;
  private boolean gotIt;
  private transient String comment;
  // ...
}
```

Objetos con referencias

- *¿Qué sucede con los objetos que apuntan (referencia) a otros objetos. Son serializables también?*
- Sí, lo son
- Son serializables de forma automática por la máquina virtual
  - Se serializa todo el *grafo* de objetos
- Se debe usar `transient`, cuando no queremos que esto suceda

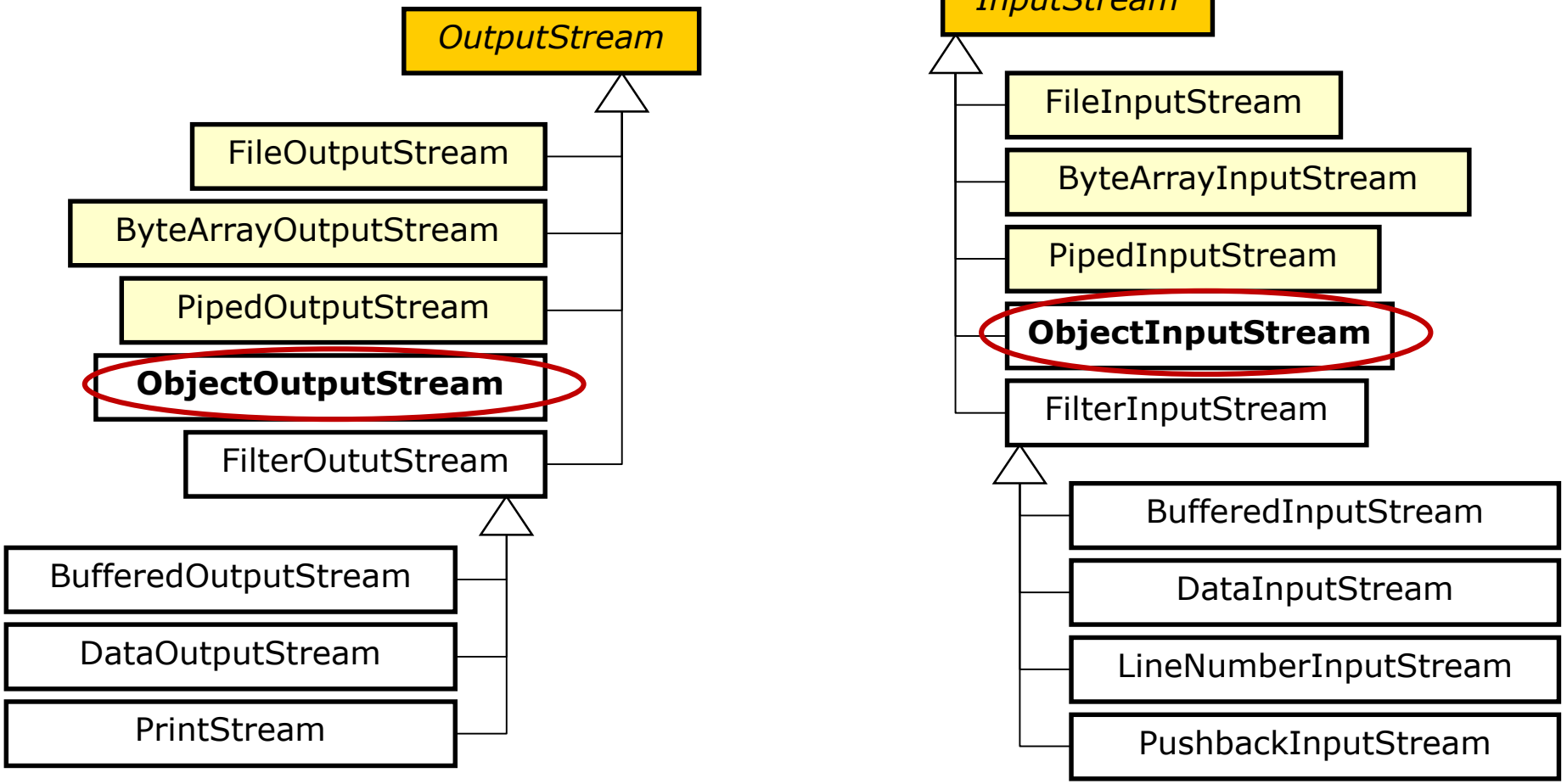
Objetos con referencias

```
public class Database implements Serializable {

  private ArrayList<Item> items;
  private ArrayList<Borrowable> borrowableItems;

  // Las dos colecciones y sus objetos son
  // serializadas porque ArrayList e Item
  // son Serializables

  // ...

}
```

```
public abstract class Item implements Serializable {
```

Haciendo objetos Persistentes

- Para conseguir que los objetos persistan en un fichero, Java ofrece dos Filtros: **`ObjectInputStream`** y **`ObjectOutputStream`**



Haciendo objetos Persistentes

- Los filtros de datos permiten leer y escribir tipos de Java, además de objetos
- Los dos métodos más importantes que proporcionan son
  - `void writeObject (Object obj)`
  - `Object readObject ()`
- Los objetos se pasan y se devuelve como instancias de `Object`
  - Por lo tanto, el objeto devuelto se debe convertir al tipo esperado

DoMe persistente (I)

```
public class PersistentDoME {

  private static final String FILE_NAME = " dome.dat";

  private Database database = new Database();

  public void updateDatabase() throws IOException,
        FileNotFoundException, ClassNotFoundException {

    ObjectOutputStream file = new ObjectOutputStream(
            new BufferedOutputStream(
                  new FileOutputStream(FILE_NAME)));
    file.writeObject(database);
    file.close();
  }
```

DoMe persistente (II)

```
public Database retrieveDatabase()
                throws FileNotFoundException,
                 IOException, ClassNotFoundException {

ObjectInputStream file = new ObjectInputStream(
              new BufferedInputStream(
              new FileInputStream(FILE_NAME)));
    database = (Database)file.readObject();
    file.close();
    return database;
  }
```

DoMe persistente (III)

```
public static void main(String... args)
                                throws Exception {
 PersistentDoME application =
                                    new PersistentDoME ();
 Database database;
    try {
      database = application.retrieveDatabase();
    } catch(FileNotFoundException e) {
      // ... Primera ejecución
    }
    // ... (Se ejecuta la aplicación)
    application.updateDatabase();
  }
}
```

Última pregunta

- *¿Qué formato es usado en la serialización? ¿Es compatible entre diferentes plataformas?*
- Un formato binario de la máquina virtual
- Por lo tanto es independiente de:
  - Hardware
  - Sistema Operativo
- Se puede recuperar en cualquier aplicación de *Java Standard Edition* (JSE)

Revisión

- Con input / output, se debe especificar
  - El **origen** (`File`, `String`, `CharArray`, `ByteArray`, `Pipe`)
  - Si la información es **binaria** (`[Input,Output]Stream`) o si es **character** (`[Reader,Writer]`)
  - Si se va a **leer** (`[InputStream,Reader]`), si se va a **escribir** (`[OutputStream,Writer]`)
  - **Tratamiento adicional** de la información (`Buffered`, `Data`, `LineNumber`, `Pushback`, `Print`, `Object`)
- **La serialización de objetos** es una técnica eficaz
  - Permite recuperar y guardar grafos de objetos complejos
  - Es independiente de la plataforma
  - No se debe usar para un almacemiento masivo (usar mejor bases de datos)