Universidad Complutense de Madrid
Facultad de Informática

Tecnología de la Programación

Entrada/Salida en Java

Simon Pickin

Entrada/Salida y streams

- Datos de entrada/salida
  - Datos que el programa lee/escribe de/en alguna fuente
- Fuentes y sumideros de datos:
  - Fuente o sumidero: ficheros, pipes, conexión de red, buffer en memoria
  - Solo fuente: entrada estándar (Java: `System.in`)
  - Solo sumidero: salida / error estándar (Java: `System.out` & `System.err`)
- Streams
  - Flujo potencialmente ilimitado de datos
  - Manipulados mediante la inserción y extracción de datos
  - Un programa puede leer de, o escribir en, un *stream*
    - Lectura: generalmente resulta en la retirada inmediata del dato del *stream*
    - Escritura: generalmente resulta en la introducción inmediata del dato en el *stream*

Streams en Java (paquete `java.io`)

- El tipo de stream se declara como una de las siguientes clases:
  - Streams de bytes
    - I/O: cualquier subclase de las clases abstractas `InputStream` / `OutputStream`
  - Streams de caracteres
    - I/O : cualquier subclase de las clases abstractas `Reader` / `Writer`
- Java tiene *stream* de caracteres que se usa para envolver *streams* de bytes
  - Convierte un stream de bytes en un stream de caracteres (caso entrada), o vice versa (caso salida), según un *charset* especificado explícitamente:
    - `InputStreamReader` (caso entrada) / `OutputStreamWriter` (caso salida)
- Clases utilizadas por los streams estándar de entrada/salida
  - En Java, `stdin`, `stdout` y `stderr` son streams de bytes (razones históricas)
  - `System.in`: `InputStream`
  - `System.out` & `System.err`: `PrintStream`, subclase de `OutputStream`

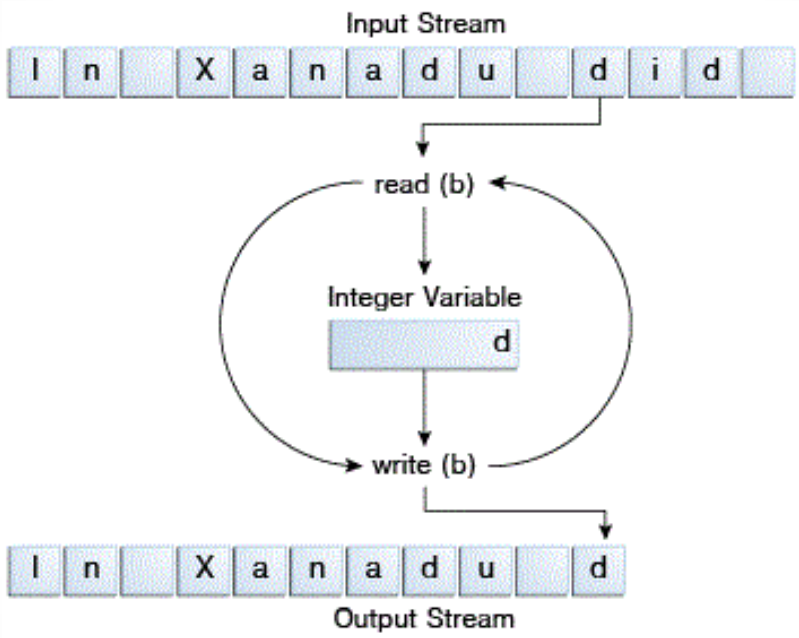
`InputStream`, `OutputStream`, `Reader`, `Writer`

- Clases abstractas que tienen varias subclases diferentes
  - Para cumplir los propósitos de los distintos *streams*
- Estos propósitos pueden resumirse en:
  - Acceso a ficheros
  - Accesos de red
  - Acceso a buffers internos de memoria
  - Pipes: comunicación inter-proceso (y a veces inter-thread)
  - Buffering
  - Filtrado
  - Parseo
  - Lectura y escritura de texto
  - Lectura y escritura de datos primitivos (`long`, `int` etc.)
  - Lectura y escritura de objetos

Streams de bytes en Java

- Subclase de `InputStream` o `OutputStream`
- Lleva a cabo operaciones E/S sobre bytes de 8-bit
- Tipo de *stream* básico
  - El sistema construye el resto de tipos de *stream* sobre *streams* de bytes
- Los *streams* de bytes solo deben utilizarse para la E/S más básica
- Deben cerrarse cuando ya no se vayan a utilizar
  - o bien en un bloque `finally`
  - o bien utilizando `try-with-resources` (desde Java 7)
  - También se aplica a los streams de caracteres

Ejemplo 1: un stream de bytes en Java

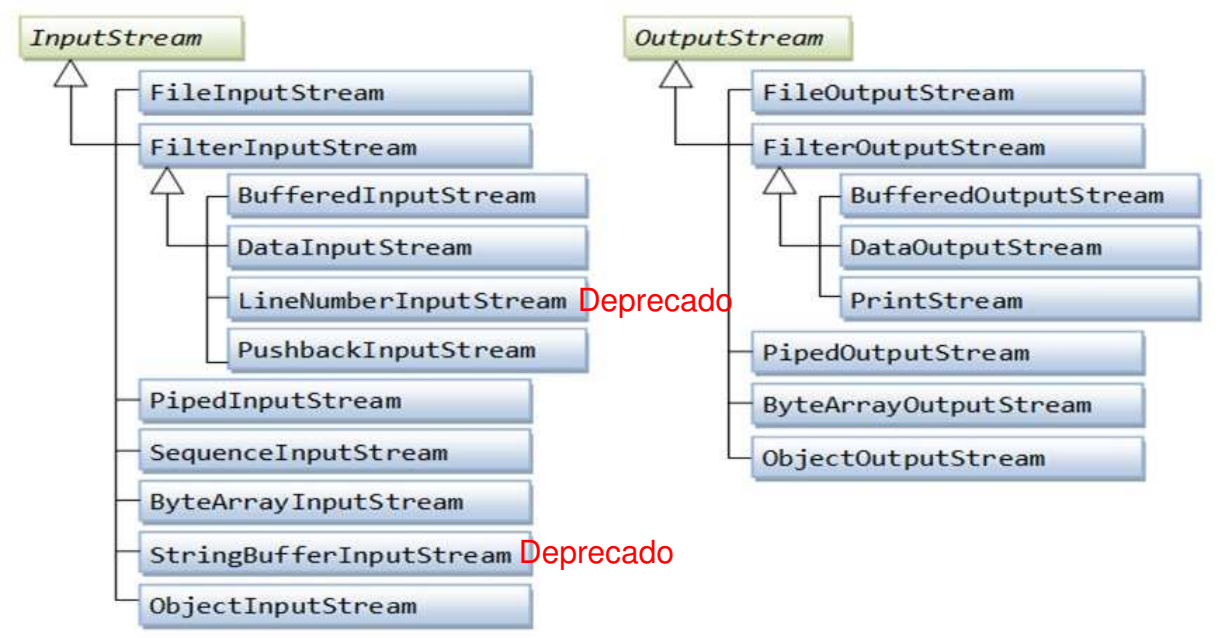


Fuente:
`https://docs.oracle.com/javase/tutorial/essential/io/bytestreams.html`

Ejemplo 1: un stream de bytes en Java

```
// FileInputStream read() devuelve -1 si se llega a fin de fichero
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream inBytes = null;
        FileOutputStream outBytes = null;
        try {
            inBytes = new FileInputStream("bytes_in.txt");
            outBytes = new FileOutputStream("bytes_out.txt");
            int c; // almacena el valor del byte en los últimos 8 bits
            while ((c = inBytes.read()) != -1) { outBytes.write(c); }
        } finally {
            if (in != null) { inBytes.close(); }
            if (out != null) { outBytes.close(); }
        }
    }
}
```


Clases principales de streams de bytes en Java



Fuente: `http://www3.ntu.edu.sg/home/ehchua/programming/java/j5b_io.html`
Para más detalles: `http://www.studytrails.com/java-io/classDiagram.jsp`
Ver también: `https://docs.oracle.com/javase/8/docs/api/java/io/package-tree.html`

Streams de caracteres en Java

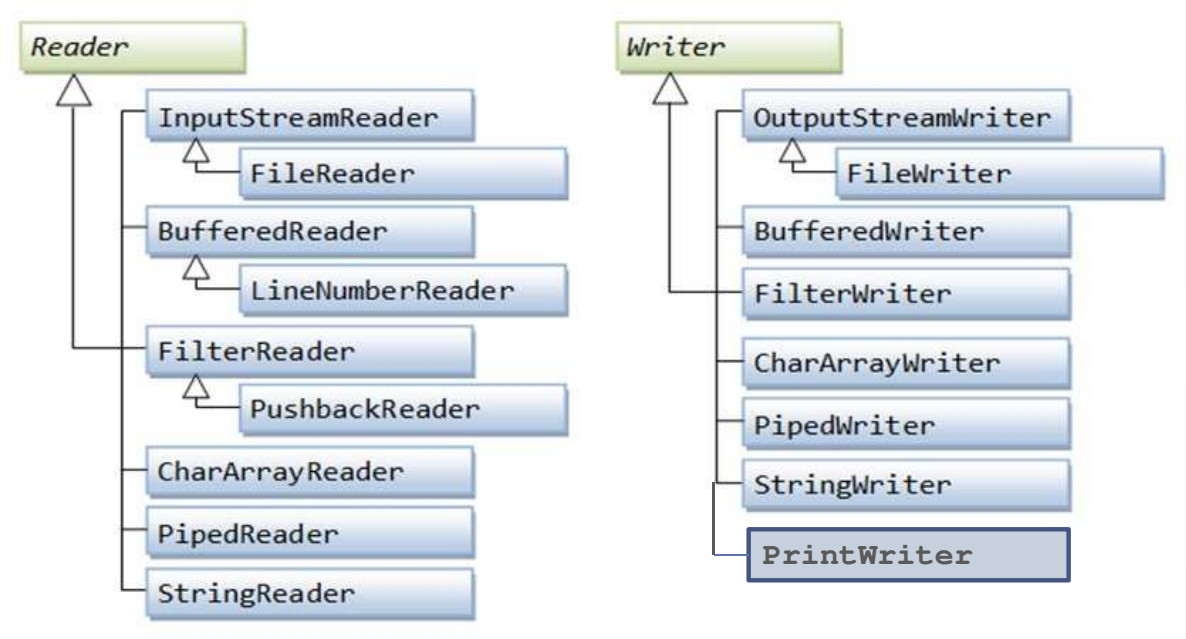
- Subclase de `Reader` o `Writer`
- *Stream* de caracteres
  - Traducción automática del *charset* local al *charset* interno
    - Java almacena internamente caracteres utilizando el charset UTF-16
    - En España, el *charset* local es UTF-8 (Unix) / CP-1252 (Windows) (*)
- Sistema implementa *streams* de caracteres mediante *streams* de bytes
  - `FileReader/FileWriter` usan `FileInputStream/FileOutputStream`
- Las operaciones se realizan generalmente en cantidades > 1 carácter.
  - Líneas: `String` de caracteres con un carácter de terminación al final.
  - Necesita E/S con buffers (ver siguiente transparencia)
- Los *streams* deben cerrarse cuando no sean requeridos

(*) CP-1252 es caso igual que ISO-Latin-1

Ejemplo 2: un stream de caracteres en Java

```
// FileReader read() devuelve -1 si se llega a fin de fichero
public class CopyCharacters {
    public static void main(String[] args) throws IOException {
        FileReader inChars = null;
        FileWriter outChars = null;
        try {
            inChars = new FileReader("default_chars_in.txt");
            outChars = new FileWriter("default_chars_out.txt");
            int c; // almacena el character en la variable de tipo int
            while ((c = inChars.read()) != -1) { outChars.write(c); }
        } finally {
            if (inChars != null) { inChars.close(); }
            if (outChars != null) { outChars.close(); }
        }
    }
}
```

Clases principales de streams de caracteres



Fuente: `http://www3.ntu.edu.sg/home/ehchua/programming/java/j5b_io.html`
Más detalles: `http://www.studytrails.com/java-io/classDiagram.jsp`
Ver también: `https://docs.oracle.com/javase/8/docs/api/java/io/package-tree.html`

Entrada/Salida con buffer

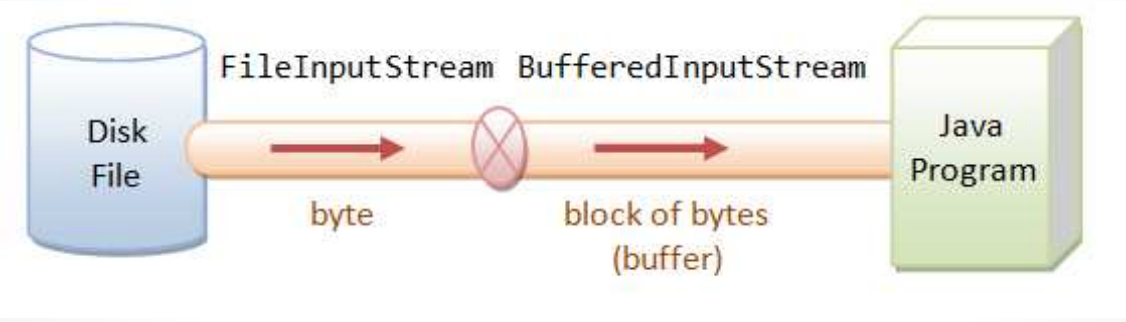
- E/S sin buffers: el programa lee/escribe desde/en el recurso
  - Cada lectura/escritura del programa provoca un lectura/escritura del SO
- E/S con buffers: el programa lee/escribe desde/en un buffer
  - Las operaciones del S.O. pueden esperar hasta que el buffer esté lleno
  - Sirve para desacoplar la operaciones del programa de las del SO
    - Aunque un programa lee/escribe por bytes, el SO lee/escribe por bloques
- E/S con buffers es más rápida que E/S sin buffers
  - Especialmente cuando las operaciones E/S son costosas y/o numerosas
    - En particular: accesos a disco y/o manipulación de grandes cantidades de datos
  - La velocidad incrementa con el tamaño del buffer (hasta cierto punto)
    - Tamaño por defecto en Java: 8 Kbytes
    - Buen valor para el tamaño del buffer: múltiplo del tamaño de la caché de disco
  - Incremento pequeño adicional si el programa también lee por bloques

Streams con buffer

- Un *stream* con buffer en Java:
  - tiene la forma de un *stream* básico con "envoltorio" un *buffered stream*
    - El programador puede especificar explícitamente el tamaño del buffer si quiere
  - Sintaxis: pasar el objeto del *stream* básico al constructor del *buffered stream*
    - El mismo sintaxis puede usarse para encadenar más de 2 tipos de *stream*
- Específicamente, se envuelve
  - Un objeto `Reader` en un objeto `BufferedReader`
  - Un objeto `Writer` en un objeto `BufferedWriter`
  - Un objeto `InputStream` en un objeto `BufferedInputStream`
  - Un objeto `OutputStream` en un objeto `BufferedOutputStream`
- Evitar E/S sin buffers salvo si la latencia de las operaciones es crítica
  - Incluso en este caso, para la salida puede bastar usar `flush()` (o autoflush)

Ejemplo 3: streams con buffer en Java

```
BufferedInputStream in = new BufferedInputStream(
                             new FileInputStream("in.dat"));
```



Fuente: `http://www3.ntu.edu.sg/home/ehchua/programming/java/j5b_io.html`

Ejemplo 3: un stream de bytes con buffer en Java

```
public class CopyBytes {  // se ejecuta más rápido que el ejemplo 1
    public static void main(String[] args) throws IOException {
        BufferedInputStream inBytes = null;
        BufferedOutputStream outBytes = null;
        try {
            inBytes = new BufferedInputStream(
                        new FileInputStream("bytes_in.txt"));
            outBytes = new BufferedOutputStream
                        new FileOutputStream("bytes_out.txt"));
            int c; // almacena el valor del byte en los últimos 8 bits
            while ((c = inBytes.read()) != -1) { outBytes.write(c); }
        } finally {
            if (in != null) { inBytes.close(); }
            if (out != null) { outBytes.close(); }
        }
    }
}
```


Ejemplo 4: un stream de caracteres con buffer en Java

```
public class CopyLines { // se ejecuta más rápido que el ejemplo 2
    public static void main(String[] args) throws IOException {
        BufferedReader inChars = null;
        BufferedWriter outChars = null;
        try {
            inChars = new BufferedReader(
                        new FileReader("default_chars_in.txt") );
            outChars = new BufferedWriter(
                        new FileWriter("default_chars_out.txt") );
            String l; // con BufferedReader se puede leer líneas enteras
            while ((l = inChars.readLine()) != null){
                outChars.write(l); outChars.newline();
            }
        } finally {
            if (inChars != null) { inChars.close(); }
            if (outChars != null) { outChars.close(); }
        }
    }
}
```

Streams de caracteres que envuelven streams de bytes

- Utilizar el *charset* por defecto puede resultar problemático
  - Leer un fichero de texto escrito con una codificación diferente
  - Leer un fichero de texto escrito en una región que tiene una codificación diferente
- E/S con las clases `InputStreamReader` / `OutputStreamWriter`
  - *Streams* de caracteres que pueden ser utilizados únicamente para "envolver" streams de bytes
- Estas dos clases tienen varios constructores
  - 1 argumento: contiene el *stream* a "envolver" y utiliza el *charset* por defecto
  - 2 argumentos: el segundo argumento especifica el *charset*:
    - Como `String` que contenga el nombre del *charset*
      - Tal como estandarizado en el "Charset Registry" de IANA
    - Como objeto `java.nio.charset.Charset`
    - Como objeto `java.nio.charset.CharsetDecoder`

Ejemplo 5: un stream de caracteres, charset explícito

```
public class CopyUTF8Characters {  // compara con el ejemplo 2
    public static void main(String[] args) throws IOException {
        InputStreamReader inChars = null;
        OutputStreamWriter outChars = null;
        try {           // el flujo de char "envuelve" el flujo de byte
            inChars = new InputStreamReader(
                    new FileInputStream("UTF8chars_in.txt"), "UTF-8");
            outChars = new OutputStreamWriter(
                    new FileOutputStream("UTF8chars_out.txt"), "UTF-8");
            int c;      // almacena el caracter en los últimos 8-32 bits
            while ((c = inChars.read()) != -1) { outChars.write(c); }
        } finally {
            if (inChars != null) { inChars.close(); }
            if (outChars != null) { outChars.close(); }
        }
    }
}
```

Ejemplo 6: un stream de caracteres con buffer, charset explícito

```
public class CopyUTF8Lines { // compara con el ejemplo 4
    public static void main(String[] args) throws IOException {
        BufferedReader inChars = null;
        PrintWriter outChars = null;
        try {
            inChars = new BufferedReader(
                        new InputStreamReader(
                         new FileInputStream("UTF8chars_in.txt"), "UTF-8"));
            outChars = new PrintWriter(
                         new OutputStreamWriter(
                          new FileOutputStream("UTF8chars_out.txt"), "UTF-8"));
            String l;  // PrintWriter tiene el método println (incluye salto)
            while ((l = inChars.readLine()) != null){ outChars.println(l); }
        } finally {
            if (inChars != null) { inChars.close(); }
            if (outChars != null) { outChars.close(); }
        }
    }
}
```


Streams de ficheros en Java

- Lee/escribe el contenido de un fichero como *stream* de bytes/chars
  - `FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter`
- Tienen 3 constructores, cada uno especifica el fichero de una manera:
  - Como un `String`: contiene la ruta (absoluta o relativa) al fichero
    - Sintaxis Unix (separados: "/") también debe funcionar en Windows, si no "\\"
  - Como un objeto `java.io.File`
  - Como un objeto `java.io.FileDescriptor`

  y 2 constructores más en el caso de *streams* de ficheros de salida
  - Especifican con un parámetro tipo `boolean` modo "añadido" (append)
- Características dependientes de la plataforma
  - Escribir en un fichero no existente implica su creación, o no
  - Más de un stream puede estar asociado a un fichero, o no

Streams con acceso a una posición arbitraria de un fichero

- Lectura/escritura de una posición arbitraria (¡no aleatoria!) en el fichero
  - `RandomAccessFile` no es subclase de las 4 clases de stream principales
- Tipo de *stream* de ficheros con puntero (o *cursor*)
  - Operaciones de lectura/escritura desplazan automáticamente el puntero
  - `seek()` mueve el puntero a una posición dada (número de bytes)
  - `getFilePointer()` devuelve la posición del puntero (como `long`)
- 2 constructores, cada uno especifica el fichero de una manera distinta:
  - Como un `String`: ruta (absoluta o relativa) al fichero
  - Como un objeto `java.io.File`

  Cada constructor tiene un parámetro `String` para especificar el modo:
  - Modos posibles: `"r"`, `"rw"`, `"rws"`, `"rwd"`

Ejemplo 7: ficheros de acceso arbitrario en Java

```
public class RandomAccessFileExample {// Utiliza try-with-resources de Java 7
    public static void main(String[] args) throws IOException {
        try (RandomAccessFile raf = new RandomAccessFile("books.dat", "rw")){
            String books[] = new String[2];
            books[1] = "Java Security";
            books[2] = "Java Security Handbook";
            for (int i = 0; i < books.length; i++) {
                raf.writeUTF(books[i]);
            }
            raf.seek(raf.length());
            raf.writeUTF("Servlet Programming");
            raf.seek(0);
            while (raf.getFilePointer() < raf.length()) {
                System.out.println(raf.readUTF());
            }
        }  // SUSTITUIR ESTE EJEMPLO POR UNO QUE SALTA HASTA MEDIO DEL FICHERO
    }      // SALTAR HASTA EL FINAL NO NECESITA RANDOMACCESSFILE
}
```

Streams de datos en Java

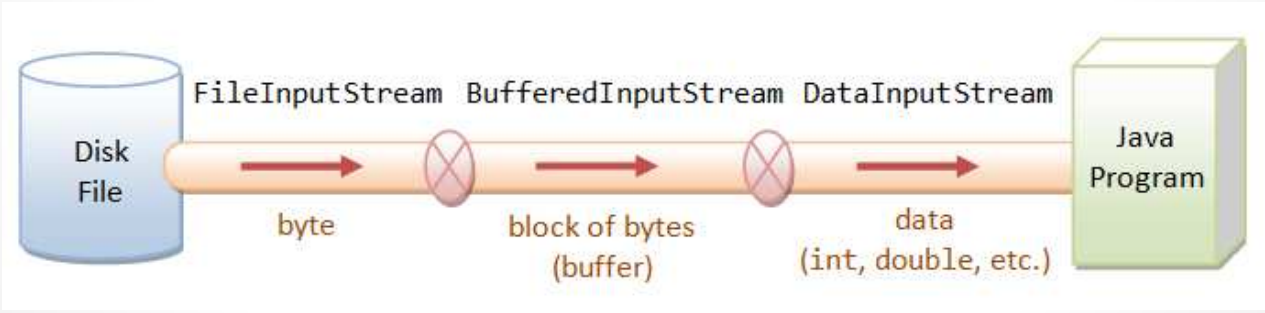
- Los *data stream* son *byte stream* compuestos de tipos de datos primitivos
  - `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`

  así como valores de tipo `String`
  - Clases: `DataInputStream` / `DataOutputStream`
- Un *data stream* debe envolver otro *byte stream*
- Con los *data stream* se detecta la condición de "fin de fichero"
  - capturando `EOFException` en lugar de comprobar el valor de retorno
- Streams de datos están formados por datos binarios
  - sin ninguna indicación del tipo de cada dato individual,
  - ni de dónde comienza cada data individual en el *stream*

  → No puede indicarse si se leen datos utilizando un tipo erróneo

Ejemplo 8: streams de datos en Java

```
DataInputStream in = new DataInputStream(
                         new BufferedInputStream(
                             new FileInputStream("in.dat")));
```



Fuente: `http://www3.ntu.edu.sg/home/ehchua/programming/java/j5b_io.html`

Ejemplo 8: streams de datos en Java

```
// Utiliza try-with-resources de Java 7
public void dataStreamInOut() throws IOException {
     try(DataOutputStream data_out = new DataOutputStream(
                                     new BufferedOutputStream(
                                     new FileOutputStream("data.bin")))){
          data_out.writeInt(123);
          data_out.writeFloat(123.45F);
          data_out.writeLong(789);
     }
     try(DataInputStream data_in = new DataInputStream(
                                     new BufferedInputStream(
                                     new FileInputStream("data.bin")))){
          int anInt = data_in.readInt();       // anInt == 123
          float aFloat = data_in.readFloat(); // aFloat == 123.45
          long aLong = data_in.readLong();     // aLong == 789
          int anotherInt = data_in.readInt(); // lanza EOFException
     }
}
```


E/S mixta: caracteres y otros tipos de datos

- ¿Qué tipo de stream utilizar para E/S mixta?
  - Stream de bytes:
    - *Data streams* son básicos y poco amenos
    - No se recomienda `PrintStream`, aunque es lo que utiliza la salida estándar
  - Stream de caracteres básico (sin buffer):
    - Al escribir, debe convertir explícitamente otros tipos de datos en `String`
      - P.ej. `Integer.toString(x)` (o `"" + x`) para escribir el valor de la variable entera `x`
      - Preparar la cadena previamente con el método `String.format` puede facilitar la tarea
    - Al leer, debe convertir explícitamente `String` en otros tipos de datos
      - P.ej. `x = Integer.parseInt()` para leer un valor en la variable entera `x`
      - La conversión explícita de cadena a número puede lanzar `NumberFormatException`
    - Para escribir el "fin de línea" independiente de plataforma debe usar
      - `System.getProperty("line.separator")` o `System.lineSeparator()`

E/S mixta: caracteres y otros tipos de datos

- ¿Qué tipo de stream utilizar para E/S mixta?
  - Stream de caracteres con buffer
    - Manera más sencilla de escribir el “fin de línea” independiente de plataforma
      - `BufferedWriter.newLine()`
    - Sigue siendo necesaria la conversión explícita desde o hacia `String`
  - Stream de caracteres con formato. Ventajas:
    - Conversión implícita hacia (salida) y desde (entrada) `String`
      - Conversión implícita depende del “locale” del sistema; el programador lo puede cambiar.
      - e.g. ¿qué carácter utilizamos para la separación de decimales?
      - Conversión implícita con `Scanner` puede lanzar `InputMismatchException`
    - Métodos que implícitamente utilizan “line.separator” para escribir fin de línea.
      - Independiente de plataforma sin trabajo adicional!

Salida formateada con `java.io.PrintWriter`

- Tiene métodos para convertir datos internos en texto con formato
  - `print()` y `println()` formatean valores individuales de manera estándar
    - Imprime valores primitivos como texto; convierte objetos utilizando `toString`
  - `format()` formatea cualquier número de valores (usa clase `Formatter`)
    - Existen muchas opciones para el formato (método `printf()` es equivalente)
- Tiene varios constructores
  - para conectarlo con `Writer`, `OutputStream`, `File` o fichero nombrado
    - Instancias de muchas subclases de `Writer` únicamente pueden conectarse con instancias de otras subclases de `Writer`
- Los métodos de la clase `PrintWriter` nunca lanzan `IOException`
  - Usa flag de error & métodos `checkError()`, `setError()`, `clearError()`
- Tipo de la salida/error estándar es `PrintStream`, no `PrintWriter`

Ejemplo 9: escribir fichero de texto con `PrintWriter`

```
public void writeArrayToFile(float[] anArray) {

    PrintWriter out = null;

    try (out = new PrintWriter("OutFile.txt")){
        for (int i=0; i < anArray.length; i++)

            // escribe "fin de línea" independiente de plataforma con println
            out.println("Value at: " + i + " = " + anArray[i]);

    // puede ser lanzada por el constructor
    } catch (FileNotFoundException fnfe) {
        // …
    } finally
        // …
    }
}
```


Ejemplo 10: escribir fichero de texto con `PrintWriter`

```
public void writeArrayToFile(float[] anArray) {

    PrintWriter out = null;

    try (out = new PrintWriter("OutFile.txt")){
        for (int i=0; i < anArray.length; i++)

            // escribe "fin de línea" independiente de plataforma con %n
            out.format("Value at: %d = %f.%n", i, anArray[i]);

    // puede ser lanzada por el constructor
    } catch (FileNotFoundException fnfe) {
        // …
    } finally
        // …
    }
}
```

Entrada formateada con `java.util.Scanner`

- Tiene métodos para convertir texto con formato en datos internos
  - Eso es, para parsearlo (dividirlo en tokens y traducirlos)
  - Pueden leerse `float`s, `int`s, `boolean`s, `String`s, …
  - Utiliza su propio buffer interno
- Sus métodos pueden lanzar `InputMismatchException`
  - Pero no `IOException`: `ioException()` devuelve la última `IOException`
- Tiene varios constructores
  - para conectarlo con un `InputStream`, con un `Reader` o con un *channel*
  - para conectarlo directamente con un objeto `File` o que implemente `Path`
- Puede usarse con la entrada estándar, cuyo tipo es `InputStream`
  - Recuerde que `stdin` es un stream de bytes, no de caracteres

Ejemplo 11: lectura de fichero de texto con Scanner

```
public class ScanSum {
    public static void main(String[] args) throws IOException {
        Scanner s = null; double sum = 0;
        try(s = new Scanner(new BufferedReader(
                                new FileReader("usnumbers.txt")))) {
            s.useLocale(Locale.US);
            while (s.hasNext()) {
                if (s.hasNextDouble()) { sum += s.nextDouble(); }
                else { s.next(); }
            }
        } catch (InputMismatchException ime) { … }
    System.out.println(sum);
    }
}
```

Ejemplo de fichero de entrada: `usnumbers.txt`

```
8.5
32,767
3.14159
1,000,000.1
```

¿Cuál es la salida?

Streams de objetos

- ¿Cómo se puede escribir un objeto en un stream?
  - Serializar: conversión de objetos en streams de bytes
- ¿Qué ocurre con objetos que tienen otros objetos como atributo?
  - El "grafo" completo de objetos debe convertirse en un stream de bytes

  → La serialización se lleva a cabo de forma recursiva
- ¿Deben serializarse todos los atributos?
  - No, algunos no son parte del estado persistente del objeto
  - No, algunos son derivados de valores de otros atributos
- ¿Qué ocurre si al leer/escribir se utiliza una versión de clase diferente?
  - por ejemplo, una versión diferente de la clase `Person` (ver Ejemplo 12)
  - Es necesario utilizar versiones numeradas para evitar problemas
    - Gran parte de esto puede automatizarse

Streams de objetos en Java

- Los *object stream* de Java soportan E/S binaria de objetos
  - Clases Stream: `ObjectInputStream` / `ObjectOutputStream`
  - Las clases de estos objetos deben implementar `Serializable`
    - `Serializable` es un interfaz "de marcado": es decir, una interfaz sin métodos
    - Si no lo hacen, al escribir el objeto se lanza `NotSerializableException`
  - Los tipos primitivos también pueden escribirse en los objetos stream
- Debe realizarse un "casting" después de leer un objeto
  - El método `readObject()` devuelve un objeto de la clase `Object`
- La serialización de objetos en Java está optimizada
  - Ocurrencias posteriores de un mismo objeto son referencias a la primera
- Para indicar que un valor no debe ser serializado
  - Utilizar la palabra clave `transient`

Ejemplo12: streams de objetos en Java

```
public class ObjectStreamExample { // Utiliza try-with-resources
    public static class Person implements Serializable {
        public String name = null;  public int age = 0;
    }
    public static void main(String[] args)
                                throws IOException, ClassNotFoundException {
        try(ObjectOutputStream objectsOut = new ObjectOutputStream(
             new BufferedOutputStream(new FileOutputStream("person.bin")))){
            Person outPerson = new Person();
            outPerson.name = "Noam Chomsky"; outPerson.age = 89;
            out.writeObject(outPerson);
        }
        try(ObjectInputStream in = new ObjectInputStream(
             new BufferedInputStream(new FileInputStream("person.bin")))){
            Person inPerson = (Person) in.readObject();
        }
    }
}
```


Serialización de objetos en Java

- Los valores primitivos de Java son “serializables”
- Definición: decimos que un tipo no-primitivo de Java es serializable
  - Si la clase que define dicho tipo implementa el interface `Serializable`
- Entonces, un objeto es serializable, si y sólo si:
  1. Su tipo es serializable
  2. Cada uno de sus atributos
     a. Tiene un tipo que es serializable
     b. o se ha declarado como `transient`

Control de versiones durante la serialización en Java

- Puede que el escritor y el lector de un stream de objetos tengan versiones distintas de la clase de un objeto del stream
  - No será posible leer objetos de la clase del stream
- Durante la deserialización se utiliza `serialVersionUID`
  - Para verificar que la clase de un objeto del stream que tiene el lector es compatible respecto a la serialización con la clase del mismo nombre que tenía el escritor
    - Este puede no ser el caso si lector y escritor están usando distintas versiones de una clase
  - Una clase serializable puede declarar explícitamente `serialVersionUID`
    - Si no lo hace, como parte de la serialización, se calculará de forma implícita
    - Este cálculo dependerá de la versión del compilador de Java utilizado
- Si queremos garantizar consistencia en los valores `serialVersionUID`
  - entre diferentes compiladores,
  - la clase serializable debe declarar de forma explícita el valor `serialVersionUID`

Otras clases stream de `java.io`

- `ByteArrayInputStream`, `CharArrayReader`
  - Lee de un array de bytes o caracteres como si fuera un stream
- `ByteArrayOutputStream` / `CharArrayWriter`
  - Escribe en un array de bytes o caracteres como si fuera un stream
  - Utilizar `toByteArray()` / `toCharArray()` para obtener el resultado en forma de array
- `StringReader` / `StringWriter`
  - Lee/escribe utilizando un `String` como si fuera un stream de bytes
- `SequenceInputStream`
  - Se utiliza para combinar múltiples streams de bytes de entrada (leídos de manera consecutiva, no hace una fusión de streams)

Otras clases stream de `java.io`

- `PipedInputStream`, `PipedOutputStream`, `PipedReader`, `PipedWriter`
  - Pasa streams de datos entre hilos (*threads*) del mismo proceso (JVM)
  - No puede utilizarse para pasar streams de datos entre procesos diferentes
- `PusbackInputStream`, `PushbackReader`, `LineNumberReader`, `StreamTokenizer`
  - Utilizado para parsear
  - Mejor utilizar `Scanner` o `split()` que `StreamTokenizer`
- `FilterInputStream`, `FilterOutputStream`, `FilterReader`, `FilterWriter`
  - Utilizado para filtrar, esto es, añadir funcionalidad a un stream existente
  - El filtrado es una aplicación del patrón de diseño llamado Decorador

Resumen de cada clase stream de Java & su propósito

	Basados en bytes		Basados en caracteres	
	Entrada	**Salida**	**Entrada**	**Salida**
Básicos	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Ficheros	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes (tuberías)	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtrado	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parseo	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Datos	DataInputStream	DataOutputStream		
Datos con formato		PrintStream		PrintWriter
Objetos	ObjectInputStream	ObjectOutputStream		
Utilidades	SequenceInputStream			

Fuente:
`http://tutorials.jenkov.com/java-io/overview.html`

Separador de Líneas independiente de plataforma

- Character specification '`\n`' in output character streams
  - generates the linefeed character (\u000A)
  - so is NOT a platform-independent specification of a line separator
- For platform-independence, unbuffered character streams must use:
  - `System.getProperty("line.separator")` *very ugly!*
  - or, since Java 7,: `System.lineSeparator()` *still pretty ugly!*

  Must also be used when building a string, e.g. `StringBuilder`
- Buffered character streams can use:
  - method: `newline()` *better!*
- Formatted character streams can use:
  - method `println("…")`, prints text followed by a newline *much better!*
  - character specification `%n` with method `format()`, *much better!*