Input/Output in Java

Comunicare con il mondo

- Praticamente ogni programma ha la necessità di comunicare con il mondo esterno
  - Con l'utente attraverso tastiera e video
  - Con il file system per leggere e salvare dati
  - Con altre applicazioni sullo stesso computer
  - Con altre applicazioni su altri computer collegati in rete
  - Con dispositivi esterni attraverso porte seriali o USB
- Java gestisce tutti questi tipi di comunicazione in **modo uniforme** usando un unico strumento: lo **stream**

Input e Output

- Uno **stream** (in italiano **flusso**) è un canale di comunicazione attraverso cui passano dati **in una sola direzione**
- E' un "tubo" attraverso cui passano informazioni.
- **Gli stream sono un'insieme di classi contenute nel package java.io**
- Dal momento che gli stream sono monodirezionali avremo bisogno di:
  - Flussi di ingresso: **input stream**
  - Flussi di uscita: **output stream**

Sorgenti e destinazioni

- I dispositivi esterni possono essere
  - **Sorgenti** – per esempio la tastiera – a cui possiamo collegare solo stream di input
  - **Destinazioni** – per esempio il video – a cui possiamo collegare solo stream di output
  - **Sia sorgenti che destinazioni** – come i file o le connessioni di rete – a cui possiamo collegare –sia input stream (per leggere) che output stream (per scrivere).

- **Attenzione:** anche se un dispositivo è bidirezionale uno stream è sempre monodirezionale e quindi per comunicare contemporaneamente sia in scrittura che in lettura dobbiamo collegare **due stream allo stesso dispositivo**.

Byte e caratteri

- Esistono due misure di “tubi”:
  - stream di byte
  - stream di caratteri
- Java adotta infatti la codifica UNICODE che utilizza due byte (16 bit) per rappresentare un carattere
- Per operare quindi correttamente con i dispositivi o i file che trattano testo dovremo utilizzare **stream di caratteri**
- Per i dispositivi che trattano invece flussi di informazioni binarie utilizzeremo **stream di byte**

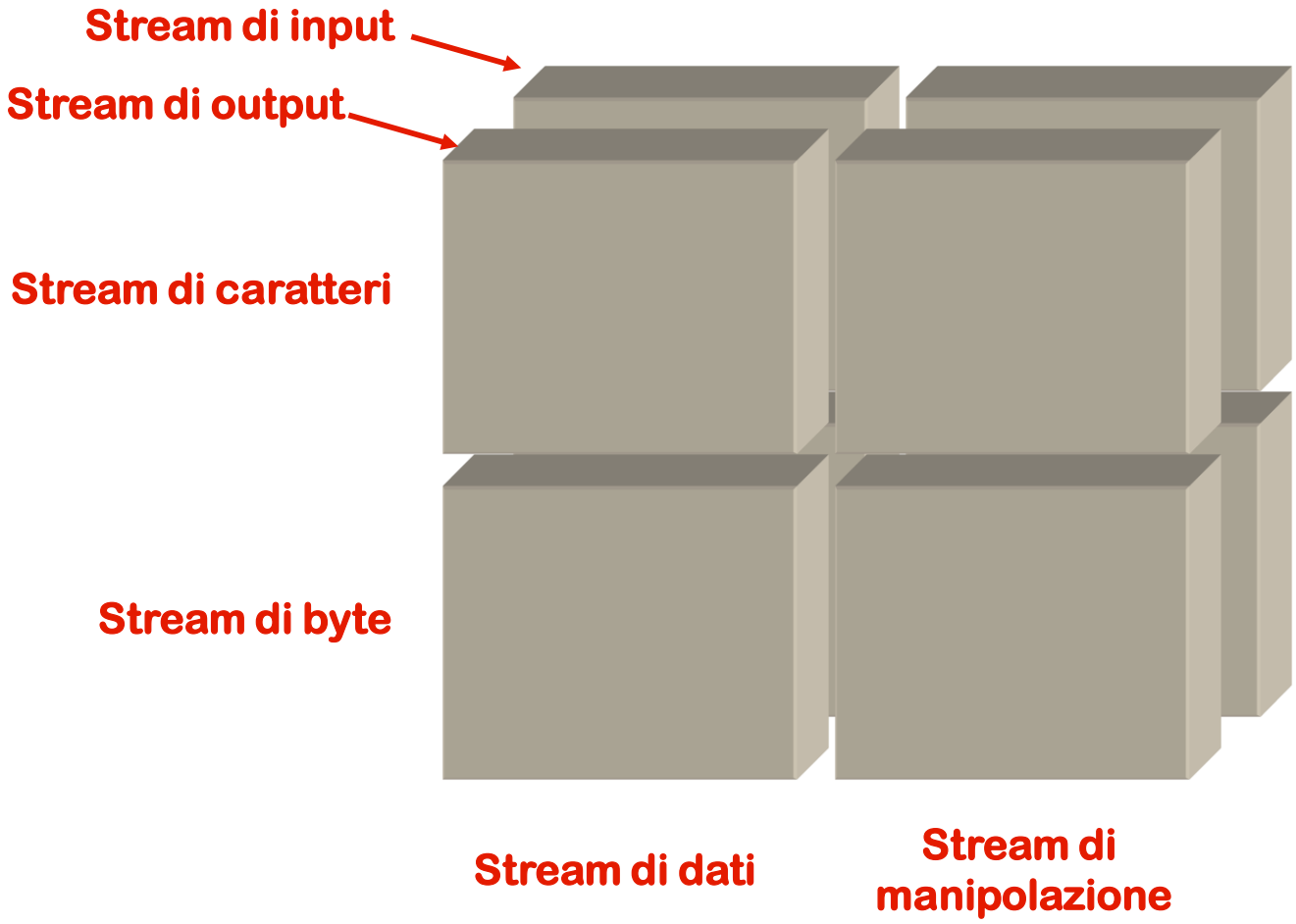
Stream di dati e stream di manipolazione

- Finora abbiamo parlato di **stream di dati** che, come abbiamo visto, hanno lo scopo di collegare un programma con una sorgente o una destinazione di dati
- Java però mette a disposizione anche un altro tipo di stream che ha come obiettivo quello di **elaborare** i dati in ingresso o in uscita
- Non si collegano direttamente ad una sorgente o destinazione di dati ma ad un altro stream e forniscono in uscita un contenuto informativo elaborato
- Anche gli **stream di manipolazione** sono di input o di output e possono trattare byte oppure caratteri

Criteri di classificazione

- Possiamo quindi classificare gli stream sulla base di tre criteri:
  - **Direzione**: input o output
  - **Tipo di dati**: byte o caratteri
  - **Scopo**: collegamento con una dispositivo/file o manipolazione di un altro stream
- Le tre classificazioni sono indipendenti (ortogonali) fra loro
- Ogni stream ho quindi una direzione, un tipo di dati trasportati e uno scopo

Schema di classificazione



Un gioco di incastri

- Le classi stream sono state realizzate in modo da potersi incastrare una con l'altra
- Si può quindi partire con uno stream di dati e incastrare uno dopo l'altro un numero qualsiasi di stream di manipolazione in modo da ottenere il risultato desiderato
- E' un meccanismo molto flessibile e potente
- Inoltre, utilizzando l'ereditarietà, il sistema può essere anche esteso a piacimento

Concetto base: l'approccio

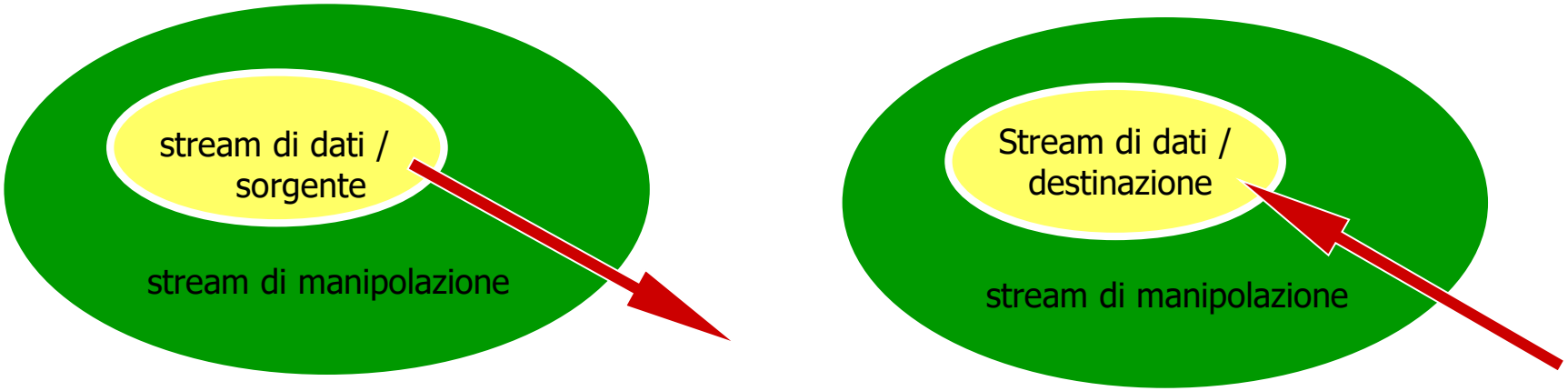
- L'approccio *"a cipolla"*
- alcuni tipi di stream rappresentano sorgenti di dati o dispositivi di uscita **[stream di dati]**
  - file, connessioni di rete,
  - array di byte, ...

- gli altri tipi di stream sono pensati per *"avvolgere" i precedenti* per aggiungere ulteriori funzionalità **[stream di manipolazione]**

Concetto base: l'approccio

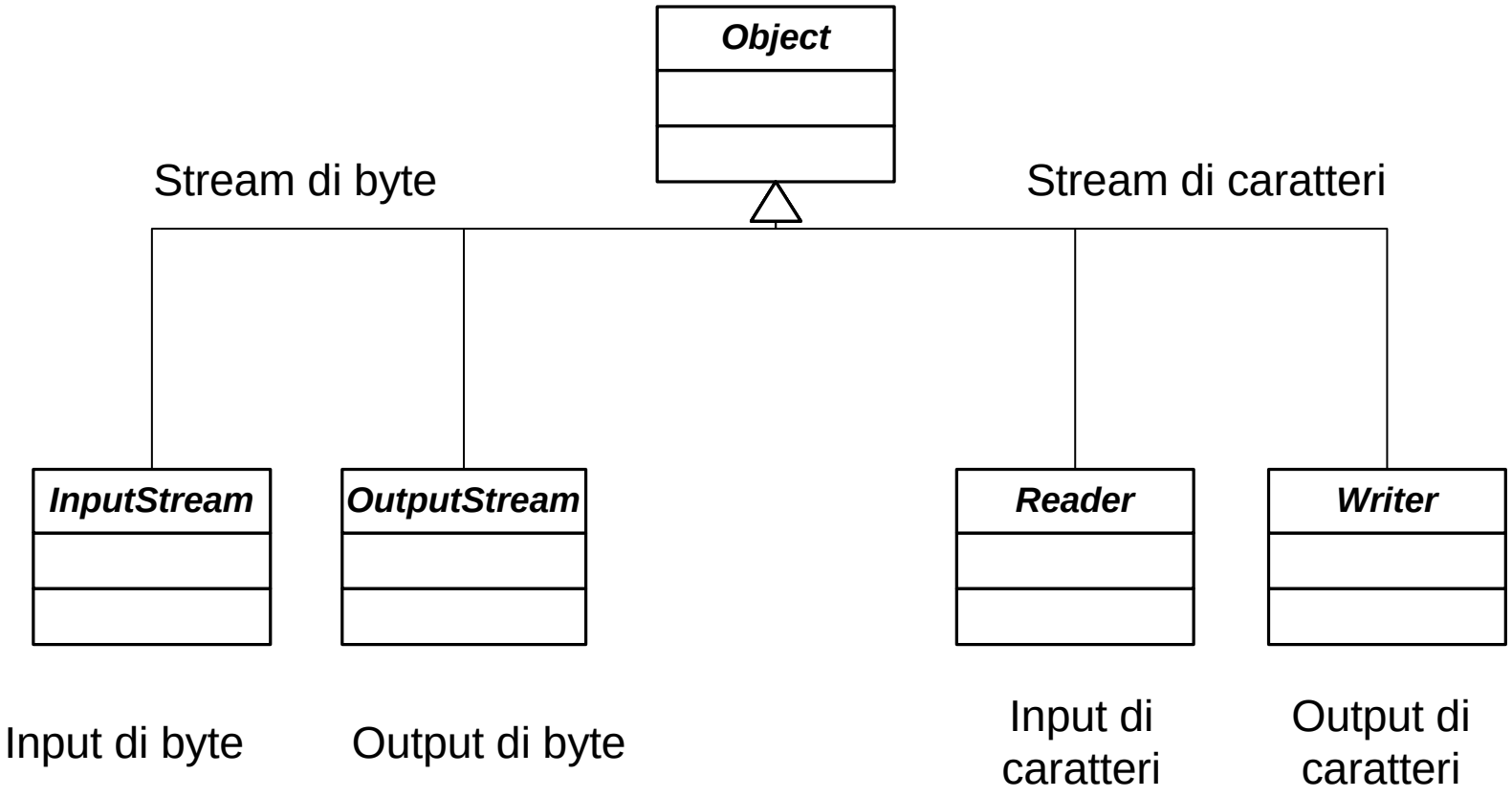
- Così, è possibile configurare il canale di comunicazione ***con tutte e sole le funzionalità che servono…***
- **senza doverle replicare e reimplementare più volte.**

Massima flessibilità, minimo sforzo.

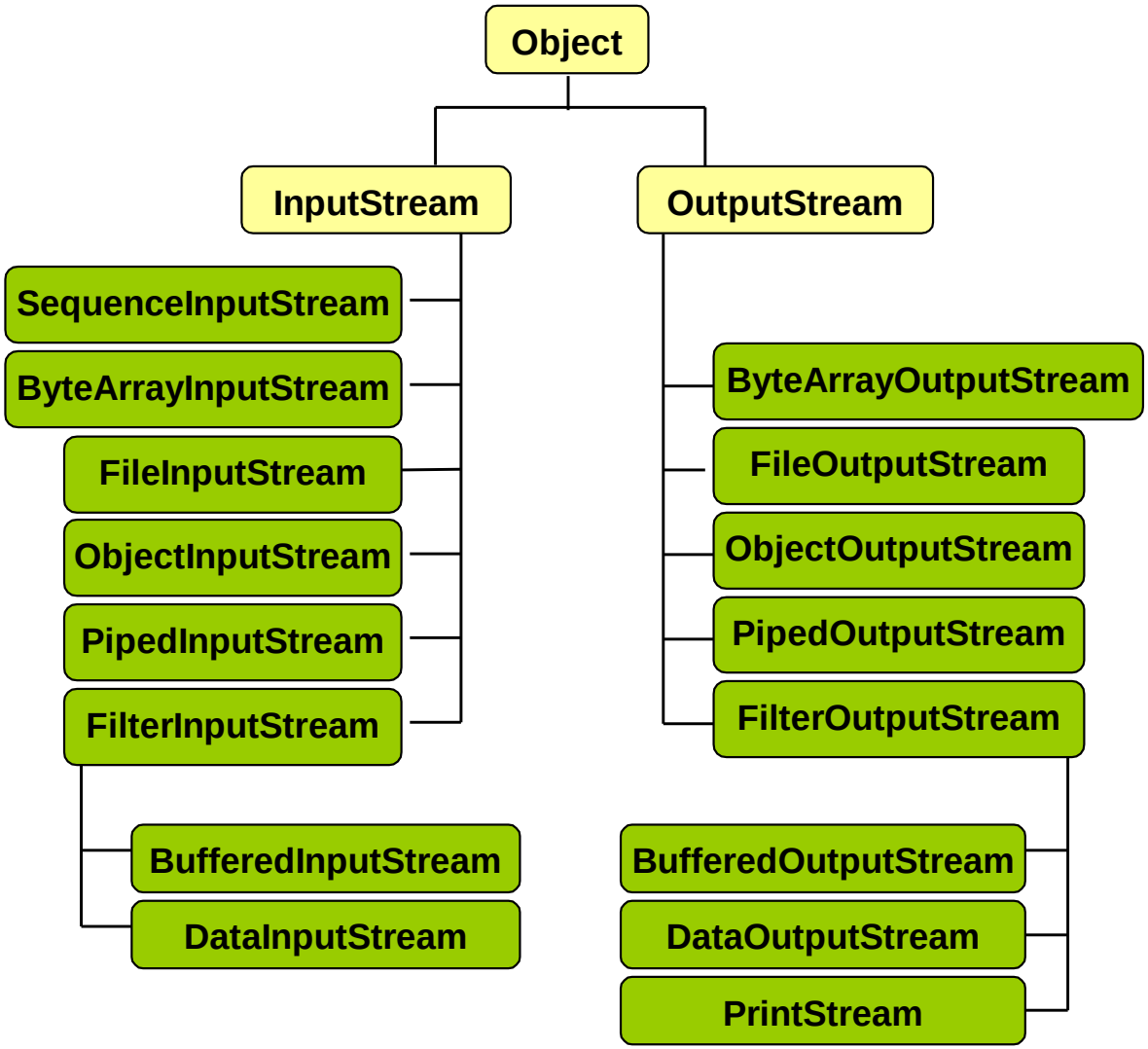


L'albero genealogico

- La gerarchia delle classi stream (contenute nel package **java.io**) rispecchia la classificazione appena esposta
- Abbiamo una prima suddivisione fra **stream di caratteri** e **stream di byte** e poi all'interno di ogni ramo tra **stream di input** e **stream di output** (sono tutte classi astratte)



La gerarchia degli stream di byte



InputStream

- E' il capostipite degli stream di input per i byte
- E' una classe astratta e definisce pochi metodi
- La sua definizione (semplificata) è:

```
package java.io
public abstract class InputStream
{
  public abstract int read()
    throws IOException;
  public int available()
    throws IOException
  { return 0; }
  public void close()
    throws IOException {}
}
```

Lettura di un byte

Numero di byte disponibili

Chiusura del canale

- read() è astratto e deve essere implementato in modo specifico dalla classi concrete
- **N.B. Tutti i metodi possono generare eccezioni**

OutputStream

- E' il capostipite degli stream di output per i byte
- E' una classe astratta e definisce pochi metodi
- La sua definizione (semplificata) è:

```
package java.io;
public abstract class OutputStream
{
  public abstract void write(int b)
    throws IOException;
  public void flush()
   throws IOException {}
  public void close()
   throws IOException {}
}
```

Scrittura di un byte

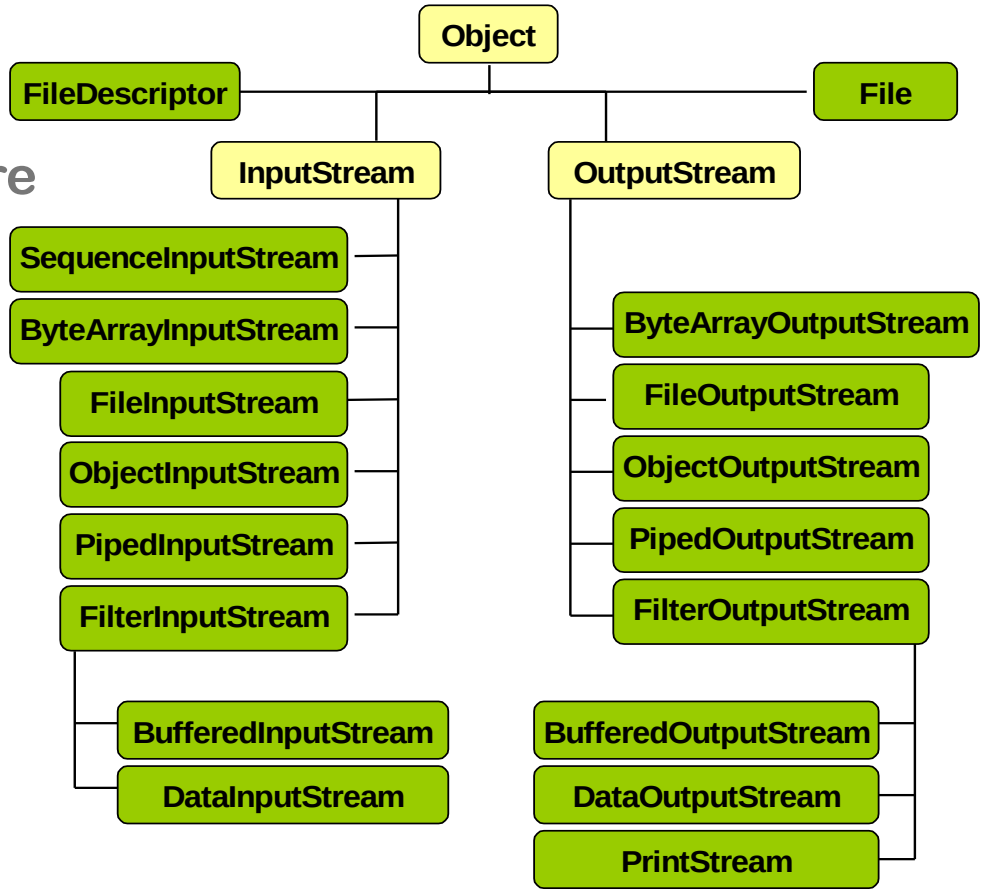
Forza l'emissione dei byte trasmessi

Chiusura del canale

- write() è astratto e deve essere implementato in modo specifico dalla classi concrete
- N.B. Tutti i metodi possono generare eccezioni

Stream di byte

- Dalle classi base astratte si derivano varie classi concrete, specializzate per fungere da:
  - sorgenti per input da file
  - dispositivi di output su file
  - stream di incapsulamento, cioè pensati per aggiungere a un altro stream nuove funzionalità.



Esempio:

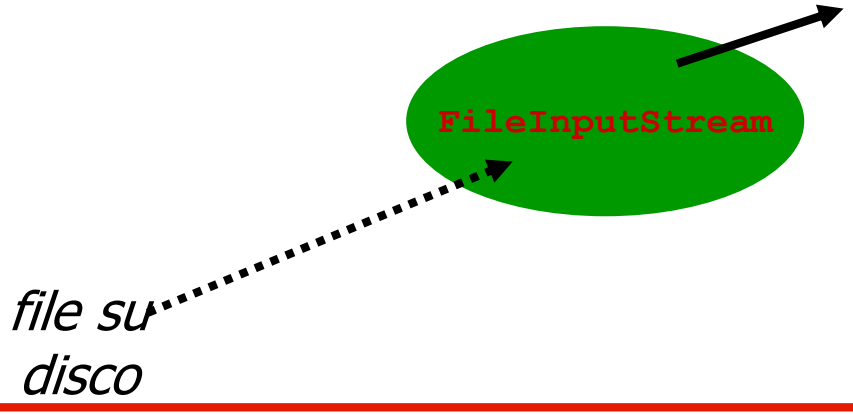
I/O bufferizzato, filtrato,…

I/O di numeri, di oggetti,…

Stream di byte – input da file

- `FileInputStream` è la classe derivata che rappresenta il concetto di sorgente di byte agganciata a un file

- Il nome del file da aprire è passato come parametro al costruttore di `FileInputStream`
- In alternativa si può passare al costruttore un oggetto **File** (o un FileDescriptor) costruito in precedenza



Stream di byte – input da file

- Per aprire un file binario in lettura si crea un oggetto di classe `FileInputStream`, specificando il nome del file all'atto della creazione.
- Per leggere dal file si usa poi il metodo `read()` che permette di leggere uno o più byte
  - restituisce il byte letto come intero fra 0 e 255
  - se lo stream è finito, restituisce -1
  - se non ci sono byte, ma lo stream non è finito, rimane in attesa dell'arrivo di un byte.

Poiché è possibile che le operazioni su stream falliscano per varie cause, tutte le operazioni possono sollevare eccezioni.

Necessità di utilizzare blocchi `try / catch`

Input da file – esempio

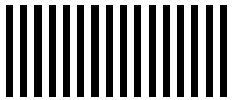
```
import java.io.*;
public class EsLeggiFile1 {
   public static void leggiFile(String filename) throws
                     FileNotFoundException, IOException {

    FileInputStream is = new FileInputStream(filename);
    int x = is.read();
    int n = 0;
    while (x >= 0) {
        System.out.print(" " + x);
        n++;
        x = is.read();
    }
    System.out.println("\nTotale byte: " + n);
    is.close();
   }
...
```

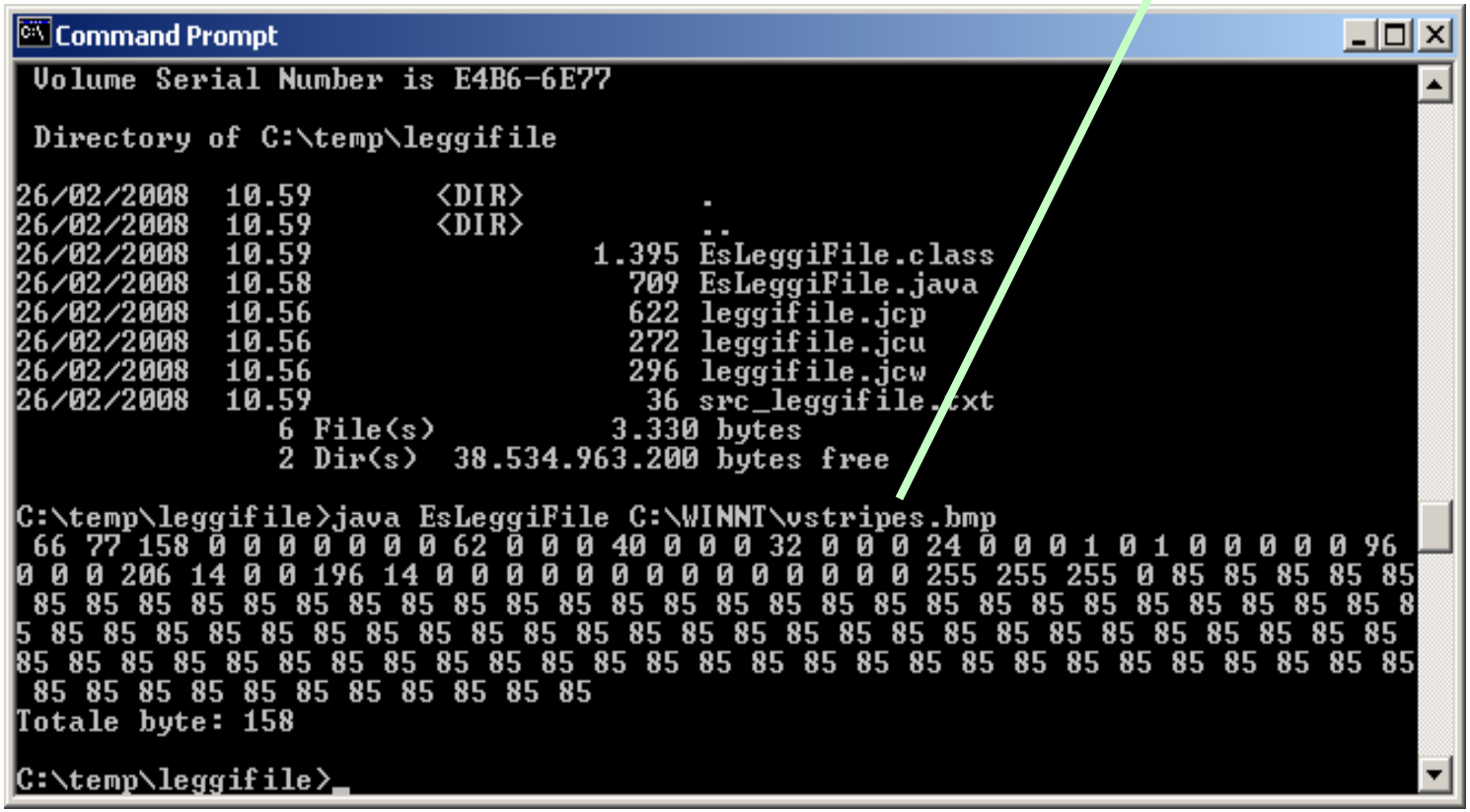
quando lo stream termina, `read()` restituisce -1

Input da file – esempio

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

    try {
      leggiFile("immagine.bmp");
    }
    catch (FileNotFoundException ex) {
      System.out.println("File non trovato");
    }
    catch(IOException ex) {
      System.out.println("Errore di input");
    }
  }
}
```

input da file – esempio

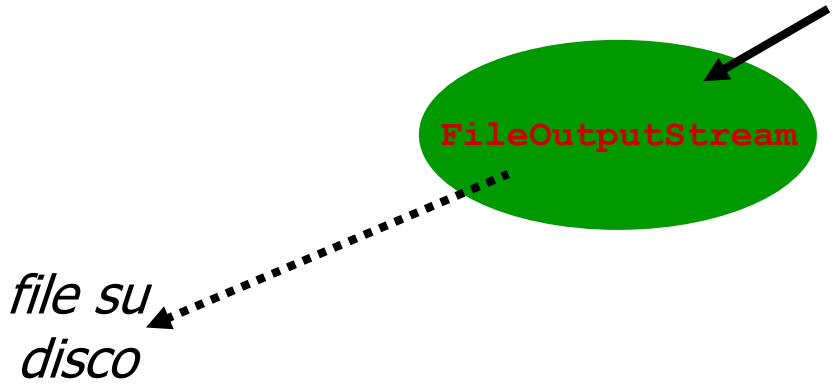




Stream di byte – output su file

- `FileOutputStream` è la classe derivata che rappresenta il concetto di dispositivo di uscita agganciato a un file

- Il nome del file da aprire è passato come parametro al costruttore di `FileOutputStream`
- In alternativa si può passare al costruttore un oggetto File (o un FileDescriptor) costruito in precedenza



stream di byte – output su file

- Per aprire un file binario in scrittura si crea un oggetto di classe `FileOutputStream`, specificando il nome del file all'atto della creazione
- Un secondo parametro opzionale, di tipo `boolean`, permette di chiedere l'apertura in modo append
- Per scrivere sul file si usa il metodo `write()` che permette di scrivere uno o più byte
  - scrive l'intero [0, 255] passatogli come parametro
  - non restituisce nulla

Poiché è possibile che le operazioni su stream falliscano per varie cause, tutte le operazioni possono sollevare eccezioni.

Necessità di utilizzare blocchi `try / catch`

stream di byte – output su file

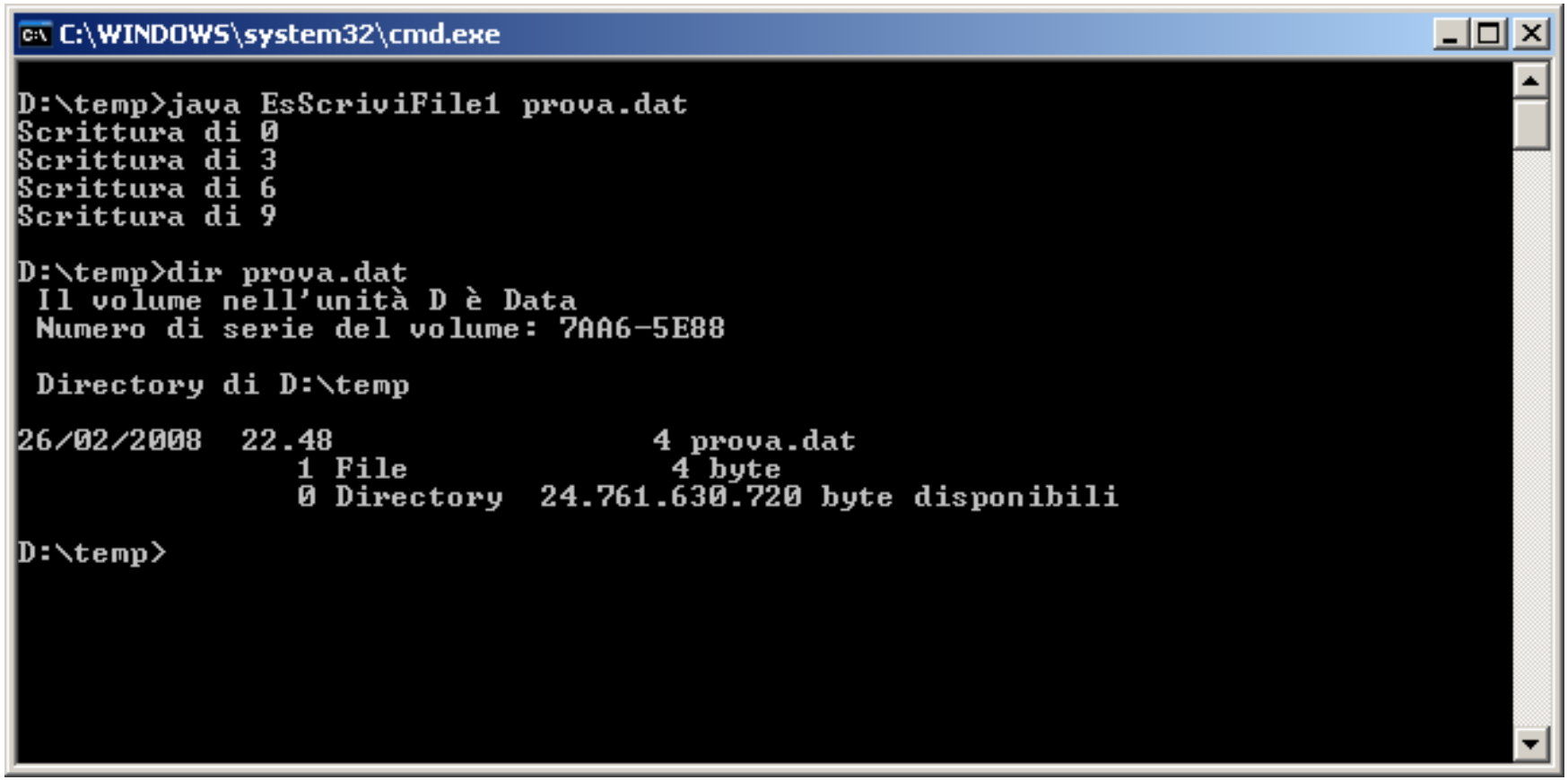
```
import java.io.*;

public class EsScriviFile1 {

    public static void scriviFile(String filename)
       throws FileNotFoundException, IOException {

        FileOutputStream os = new FileOutputStream(filename);

        for(int x=0; x<10; x+=3) {
          System.out.println("Scrittura di "+x);
          os.write(x);
        }
        os.close();
    }
...
```

Per aprirlo in modalità append:
`FileOutputStream(filename,true);`

stream di byte – output su file

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

      try {
          scriviFile("prova.dat");
      }
      catch(FileNotFoundException e) {
        System.out.println("Impossibile aprire file");
      }
      catch(IOException ex) {
        System.out.println("Errore di output;
      }
  }
}
```

Stream di byte – output su file



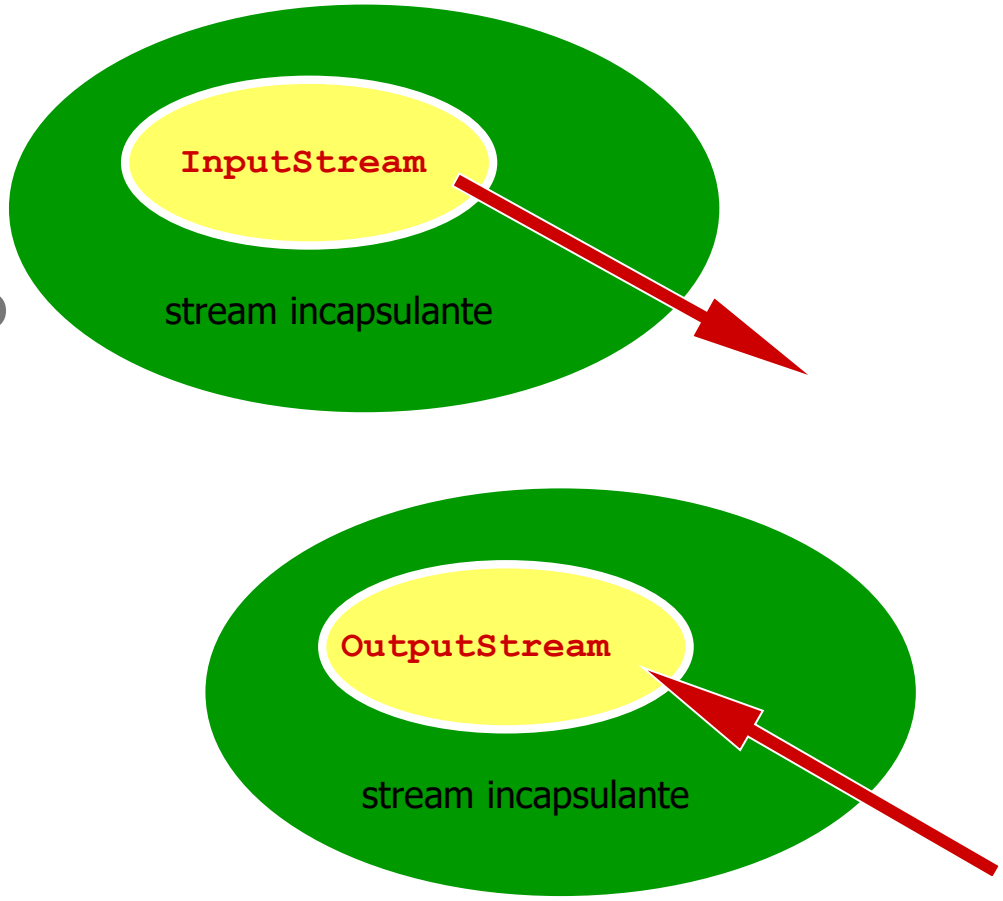
Esperimenti

- *Provare a rileggere il file con il programma precedente*
- *Aggiungere altri byte riaprendo il file in modo append*

Stream di incapsulamento

- Gli STREAM di incapsulamento hanno come scopo quello di "avvolgere" un altro STREAM per **creare un'entità con funzionalità più evolute.**

Il loro costruttore ha quindi come parametro un `InputStream` o un `OutputStream` già esistente.



Stream di incapsulamento - INPUT

- **`BufferedInputStream`**

  aggiunge un buffer e ridefinisce `read()` in modo da avere una lettura bufferizzata

- **`DataInputStream`**

  definisce metodi per leggere i tipi di dati standard in forma binaria: `readInt()`, `readFloat()`, ...

- **`ObjectInputStream`**

  definisce un metodo per leggere oggetti **serializzati** (salvati) da uno stream, offre anche metodi per leggere i tipi primitivi di Java

Stream di incapsulamento - OUTPUT

- **`BufferedOutputStream`**

  aggiunge un buffer e ridefinisce `write()` in modo da avere una scrittura bufferizzata

- **`DataOutputStream`**

  definisce metodi per scrivere i tipi di dati standard in forma binaria: `writeInt()`, `writeFloat()` ...

- **`PrintStream`**

  definisce metodi per stampare come stringa valori primitivi (es. `print(int)`) e classi standard (es. `print(Object)`)

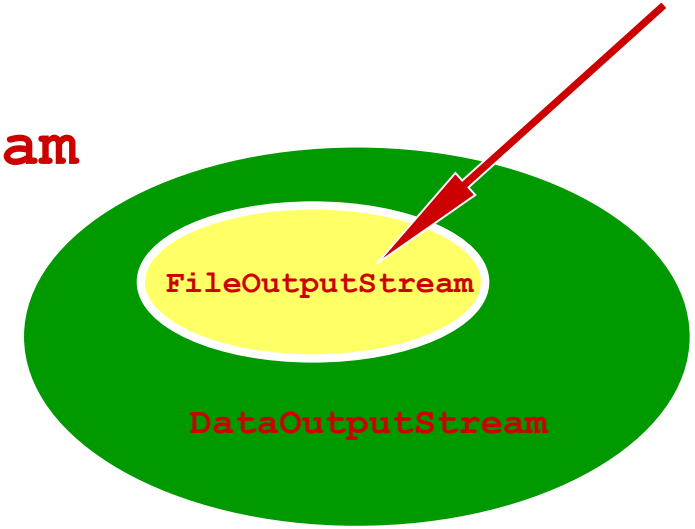
- **`ObjectOutputStream`**

  definisce un metodo per scrivere oggetti “serializzati”; offre anche metodi per scrivere i tipi primitivi di Java

Esempio - OUTPUT

- **Scrittura di dati su file binario**
- Per scrivere su un file binario occorre un `FileOutputStream`, che però consente solo di scrivere un byte o un array di byte
- Volendo scrivere dei `float`, `int`, `double`, `boolean`, … è molto più pratico un `DataOutputStream`, che ha metodi idonei

- Si incapsula `FileOutputStream` dentro un `DataOutputStream`



Esempio - OUTPUT

```
import java.io.*;
public class EsScriviFile2 {
  public static void scriviFile(String filename)
      throws FileNotFoundException, IOException {

    FileOutputStream fos=new FileOutputStream(filename);
    DataOutputStream dos=new DataOutputStream(fos);
    float   f = 3.1415F;   char    c = 'X';
    boolean b = true;      double  d = 1.4142;
    int     i = 12;
    dos.writeFloat(f);     dos.writeChar(c);
    dos.writeBoolean(b);   dos.writeDouble(d);
    dos.writeInt(i);    dos.close();
  }
```

Esempio - OUTPUT

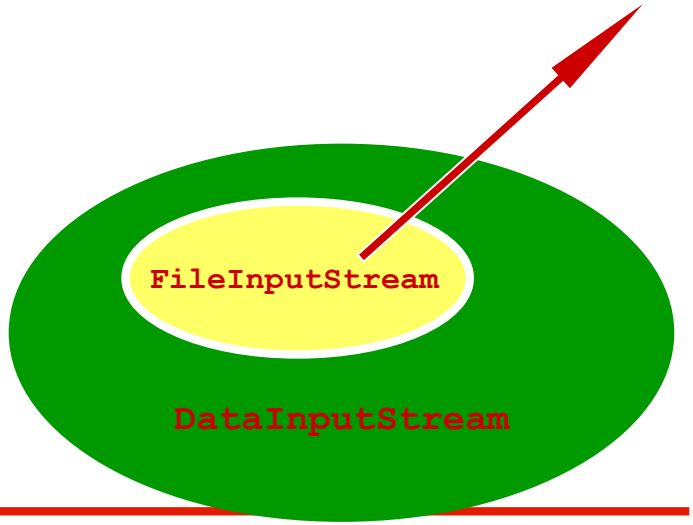
```
public static void main(String[] args) {

   try {
        scriviFile("fileOut.bin");
   }
   catch(FileNotFoundException e) {
      System.out.println("Impossibile aprire file");
   }
   catch(IOException ex) {
      System.out.println("Errore di output;
    }
  }
}
```

Esempio - INPUT

- **Lettura di dati da file binario**

- Per leggere da un file binario occorre un `FileInputStream`, che però consente solo di leggere un byte o un array di byte
- Volendo leggere dei `float`, `int`, `double`, `boolean`, … è molto più pratico un `DataInputStream`, che ha metodi idonei

- Si incapsula `FileInputStream` dentro un `DataInputStream`



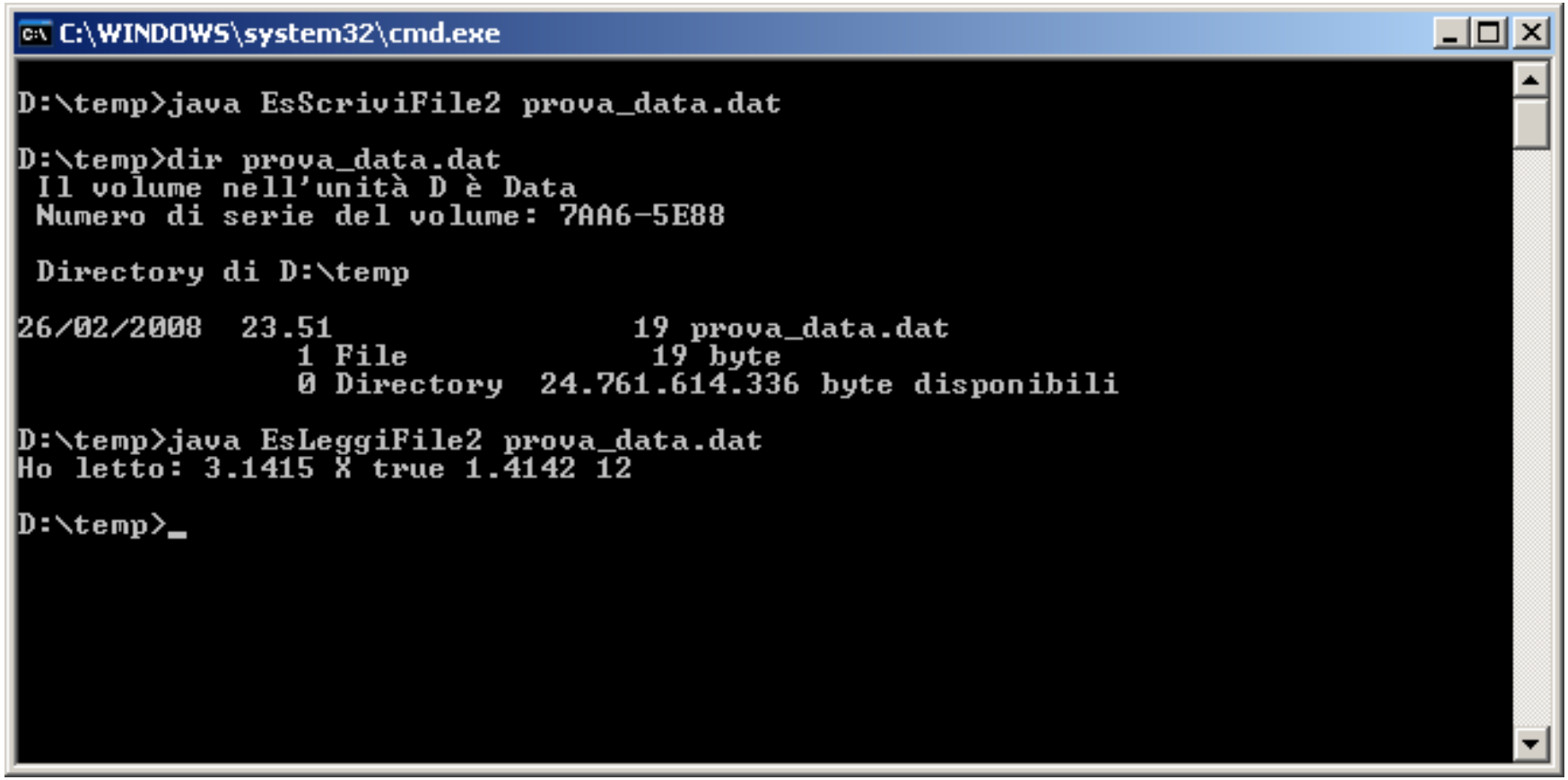
esempio - INPUT

```
import java.io.*;
public class EsLeggiFile2 {
  public static void leggiFile(String filename)
       throws FileNotFoundException, IOException {

     FileInputStream fis = new FileInputStream(filename);
     DataInputStream dis = new DataInputStream(fis);
     float   f = dis.readFloat();
     char    c = dis.readChar();
     boolean b = dis.readBoolean();
     double  d = dis.readDouble();
     int     i = dis.readInt();
     System.out.println("Ho letto:"+f+" "+c+" "+b+" "+d+" "+i);
     dis.close();
  }
. . .
```

Esempio - INPUT

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

         try {
             leggiFile("fileOut.bin");
         }
         catch(FileNotFoundException e) {
            System.out.println("Impossibile aprire file");
         }
         catch(IOException ex) {
            System.out.println("Errore di output;
       }
    }
}
```

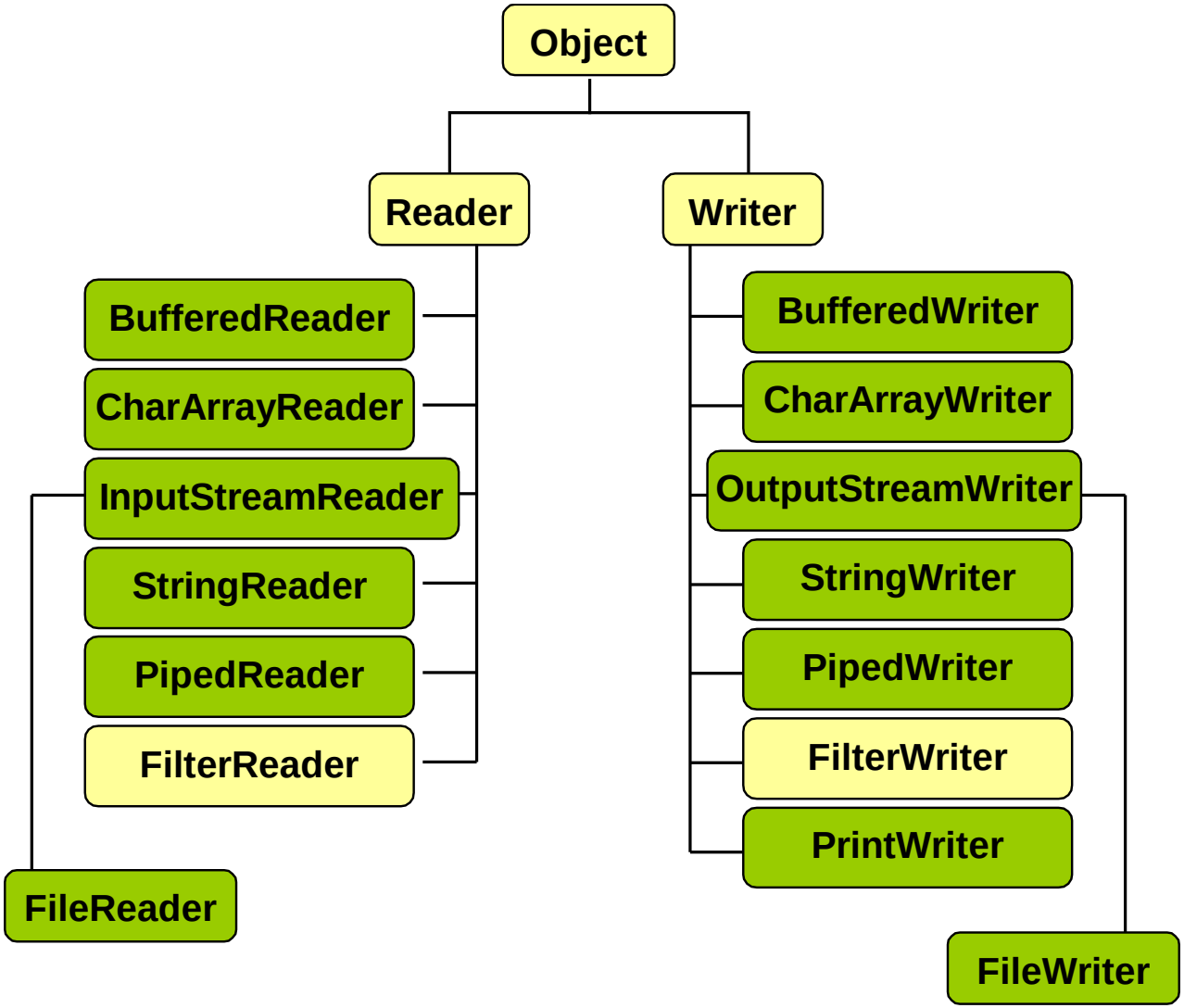
Esempio - risultato



Gli stream di caratteri

- Le classi per l'I/O da stream di caratteri (Reader e Writer) sono più efficienti di quelle a byte
- Hanno nomi analoghi e struttura analoga
- Convertono correttamente la codifica UNICODE di Java in quella locale:
  - specifica del **sistema operativo**: Windows, Mac OS-X, Linux… (tipicamente ASCII)
  - e della **lingua** in uso
    (essenziale per l'internazionalizzazione)
- Per esempio gestiscono correttamente le lettere accentate e gli altri segni diacritici delle lingue europee

La gerarchia degli stream di caratteri



Reader

- E' il capostipite degli stream di input per i caratteri
- E' una classe astratta e definisce pochi metodi
- Una sua definizione (semplificata) è:

```
package java.io
public abstract class InputStream
{
  public abstract int read()
    throws IOException;
  public boolean ready()
    throws IOException
  { return 0; }
  public void close()
    throws IOException  { }
}
```

Lettura di un carattere

Dice se c'è qualcosa da leggere

Chiusura del canale

- read() restituisce un intero e quindi bisogna ricorrere ad un cast esplicito

Writer

- E' il capostipite degli stream di output per i caratteri
- E' una classe astratta e definisce pochi metodi
- Una sua definizione (semplificata) è:

```
package java.io;
public abstract class Writer
{
  public abstract void write(int c)
    throws IOException;
  public abstract void write(String str)
    throws IOException;
  public void flush()
   throws IOException {}
  public void close()
   throws IOException {}
}
```

Scrittura di un carattere

Scrittura di una stringa

Forza l'emissione dei byte trasmessi

Chiusura del canale

- Esistono più versioni di write() (overloading)

I/O Standard

- Esistono due stream standard definiti nella classe System: **System.in** e **System.out**
- Sono attributi statici e quindi sono sempre disponibili
- Gestiscono l'input da tastiera e l'output su video

Attenzione: purtroppo per ragioni storiche (in Java 1.0 non c'erano gli stream di caratteri), sono stream di byte e non di caratteri

- In particolare:
  - **System.in** è di tipo **InputStream** (punta effettivamente ad un'istanza di una sottoclasse concreta) e quindi fornisce solo i servizi base
  - **System.out** è di tipo **PrintStream** e mette a disposizione i metodi print() e println() che consentono di scrivere a video qualunque tipo di dato

Gestione della tastiera: problemi

- **System.in** è molto rudimentale e non consente di trattare in modo semplice è corretto l'input da tastiera
- Infatti:
  - Essendo uno stream di byte non gestisce correttamente le lettere accentate
  - Non possiede metodi per leggere comodamente un'intera stringa
- Fortunatamente il meccanismo degli "incastri" di Java ci permette di risolvere in maniera efficace questi problemi.
- Per farlo useremo due classi che discendono da Reader: **InputStreamReader** e **BufferedReader**
- Sono entrambe **stream di manipolazione**

Gestione tastiera: da byte a caratteri

- Innanzitutto risolviamo i problemi legati al fatto che System.in è uno stream di byte
- **InputStreamReader** è una sorta di adattatore: converte uno stream di byte in uno stream di caratteri:
- Il suo costruttore è definito in questo modo:

```
public InputStreamReader(InputStream in)
```

- Grazie al subtyping può quindi "agganciarsi" ad un qualunque discendente di InputStream, quindi a tutti gli stream di input a byte,
- Per eseguire l'adattamento scriveremo:

```
InputStreamReader isr =
 new InputStreamReader(System.in);
```

- In questo modo possiamo utilizzare `isr` per leggere singoli caratteri da tastiera con un gestione corretta dei caratteri speciali (lettere accentate, ecc.)

Gestione tastiera: leggere le stringhe

- Vediamo ora come fare per leggere le stringhe
- BufferedReader è un discendente di Reader che aggiunge un metodo che ci consente di leggere una stringa dalla tastiera:
- `public String readLine()`
- E' quindi uno **stream di manipolazione** a caratteri
- Il costruttore è definito in questo modo:
- `public BufferedReader(Reader in)`
- Possiamo quindi agganciarlo a qualunque stream di caratteri.
- Per completare la nostra "conduttura" scriveremo quindi:

```
BufferedReader kbd =
 new BufferedReader(isr);
```

Gestione tastiera: soluzione completa

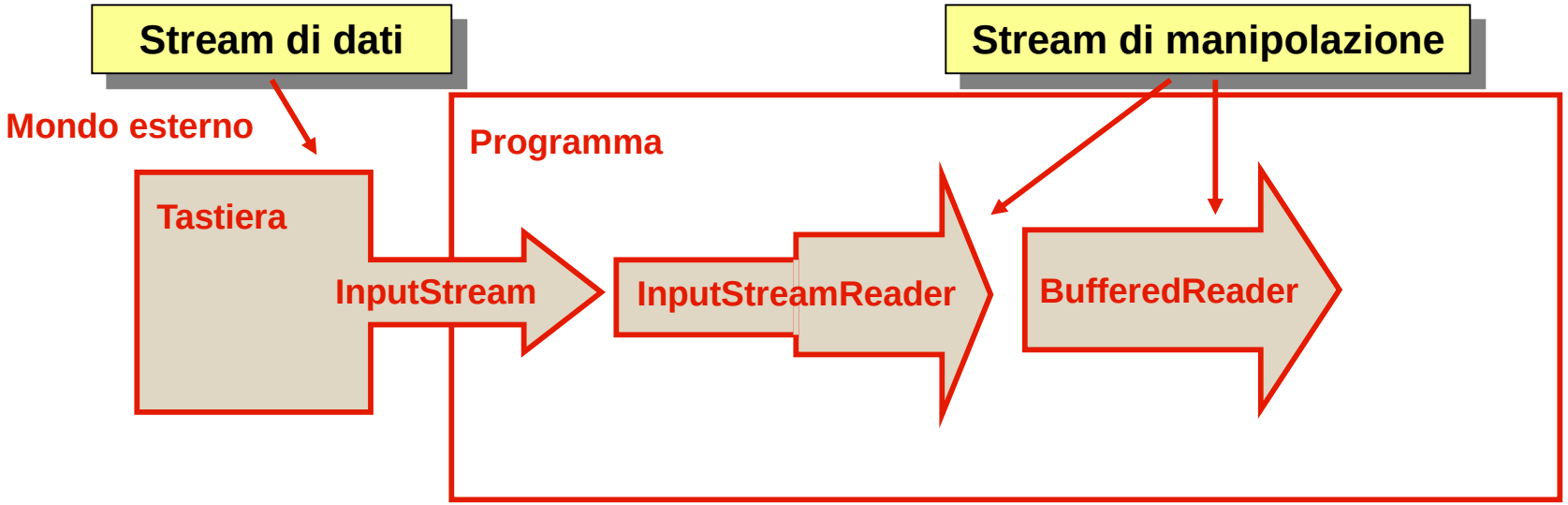
- Per gestire correttamente la tastiera useremo quindi una sequenza di istruzioni di questo tipo:

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader kbd = new BufferedReader(isr);
```

- Oppure in forma sintetica:

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

- Abbiamo quindi realizzato la nostra "conduttura":



Gestione del video

- **System.out** è già sufficiente per gestire un output di tipo semplice: print() e println() forniscono i servizi necessari
- E' uno stream di byte ma non crea particolari problemi.
- Tuttavia volendo possiamo utilizzare una tecnica simile a quella utilizzata per la tastiera
- E' sufficiente usare un solo stream di manipolazione - **PrintWriter** - che svolge anche la funzione di adattamento.
- Definisce infatti un costruttore di questo tipo:

```
public PrintWriter(OutputStream out)
```

- E mette a disposizione i metodi print() e println() per i vari tipi di dati da stampare

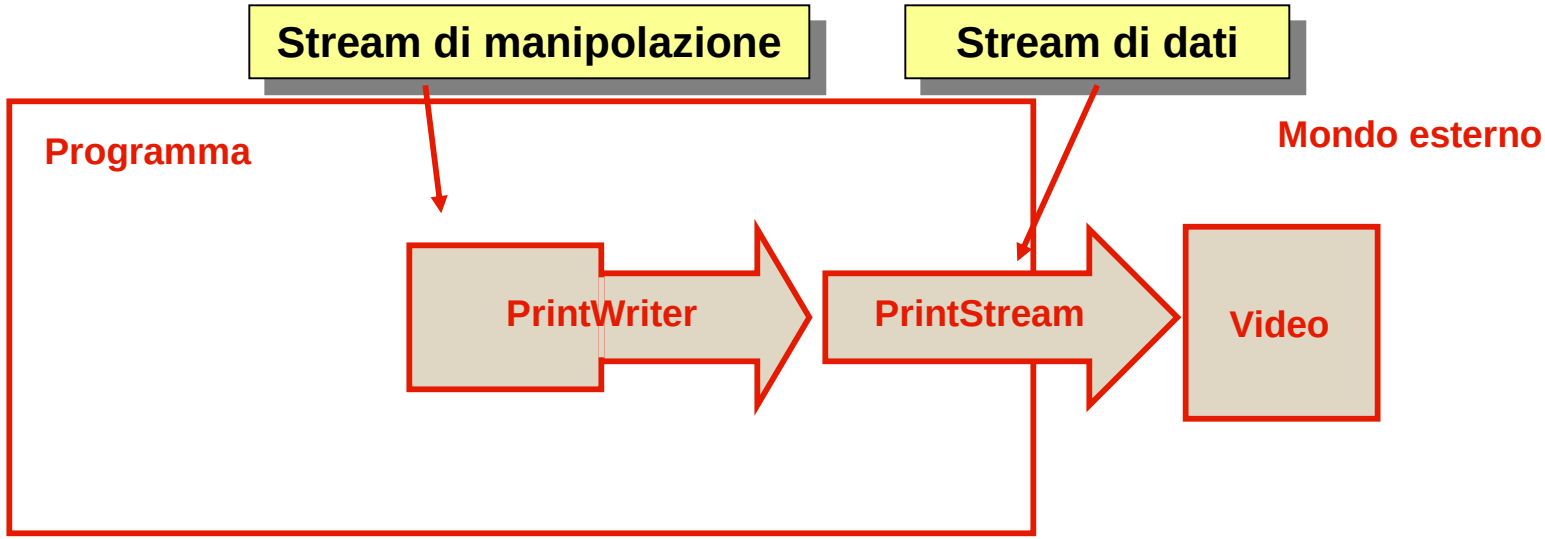
Gestione del video: soluzione completa

- Potremo quindi scrivere:

```
PrintWriter video = new PrintWriter(System.out);
```

- E utilizzarlo nello stesso modo con cui useremmo System.out

```
video.println(12);
video.println("Ciao");
video.println(13,56);
```



File di testo

- Possiamo leggere e scrivere file di testo utilizzando stream di caratteri
- In particolare:
  - La classe **FileReader** (derivata da Reader) ci permette di leggere un file di testo
  - La classe **FileWriter** (derivata da Writer) ci permette di scrivere in un file di testo
- Sono entrambi stream di dati: il loro scopo è quindi quello di creare un collegamento con un dispositivo esterno.

Stream di caratteri

- Il file di testo si apre costruendo un oggetto `FileReader` o `FileWriter`, rispettivamente
- `read()` e `write()` leggono/scrivono un `int` che rappresenta un carattere UNICODE

Un carattere UNICODE è lungo due byte.

- `read()` restituisce un valore tra 0 e 65535
- oppure -1 in caso di fine stream

Occorre dunque un cast esplicito per convertire il carattere UNICODE in `int` e viceversa.

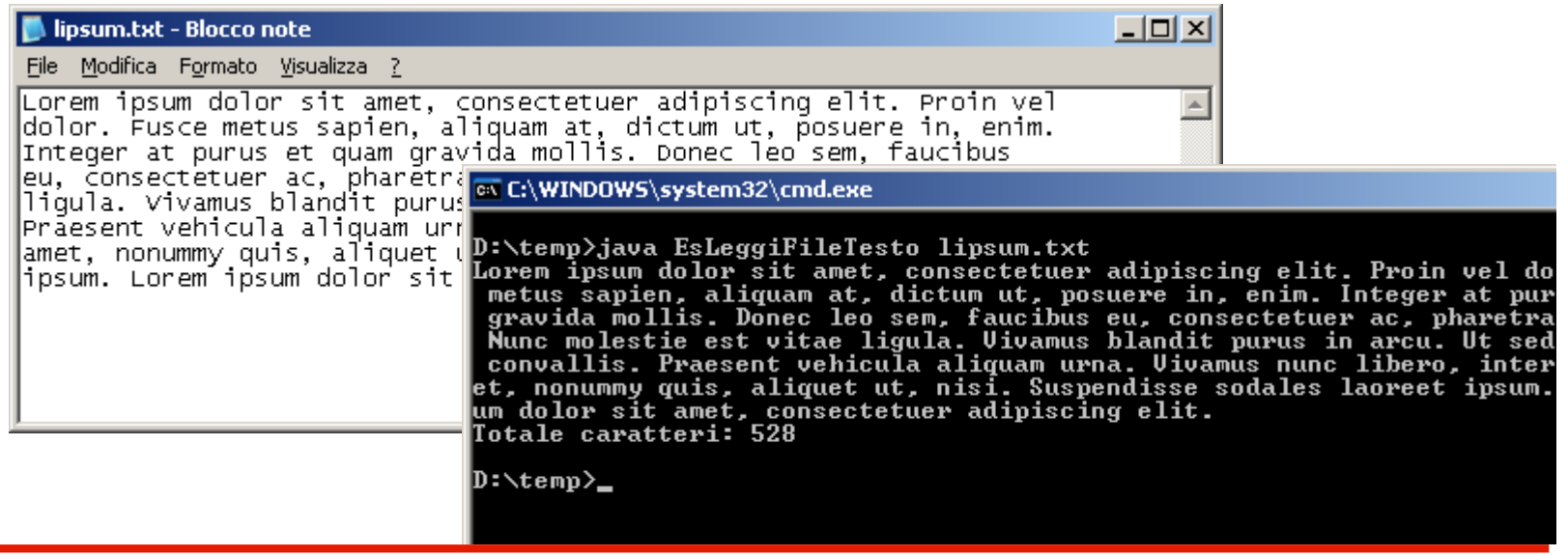
Esempio - input da file

```
import java.io.*;
public class EsLeggiFileTesto {
    public static void leggiFileTesto(String filename)
                    throws IOException {
        FileReader fr = new FileReader(filename);
        int n = 0;
        int x = fr.read();
        while (x >= 0) {
            char ch = (char) x;
            System.out.print(ch);
            n++;
            x = fr.read();
        }
       System.out.println("\nTotale caratteri: " + n);
        fr.close();
    }
. . .
```

Cast esplicito da **`int`** a **`char`**
- Ma solo se è stato *davvero* letto un carattere (cioè se non è stato letto -1)

Esempio - input da file

```
  public static void main(String[] args) {
    try {
       leggiFileTesto("testo.txt");
    }
    catch(IOException ex) {
      System.out.println("Errore di I/O");
    }
  }
}
```



Stream di incapsulamento

- Le operazioni di I/O con caratteri in genere coinvolgono tanti caratteri alla volta.

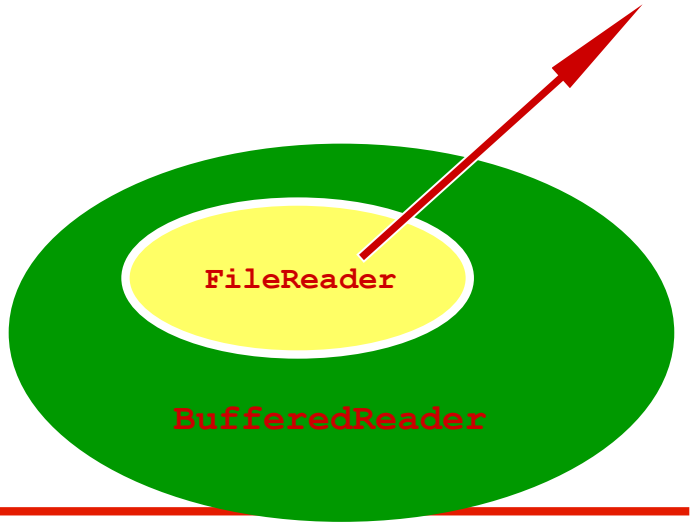
- In genere l'unità di base per lavorare con i caratteri è la LINEA: una stringa di caratteri con un terminatore di linea alla fine (in Windows "`\r\n`" carriage-return/line-feed, in Linux/UNIX "`\n`").

- Anche nel caso degli stream di caratteri, possiamo utilizzare gli stream di incapsulamento, che estendono le funzionalità dei `Reader` e `Writer` fornendo metodi idonei al trattamento delle linee di testo.

Stream di incapsulamento - INPUT

- **`BufferedReader`** lettura bufferizzata
- Fornisce i metodi:
  - `readLine()`
    - restituisce una stringa contenente il testo presente nella linea escluso il terminatore di linea
    - `null` se si raggiunge la fine dello stream
  - `read()` (singolo carattere e array di caratteri)

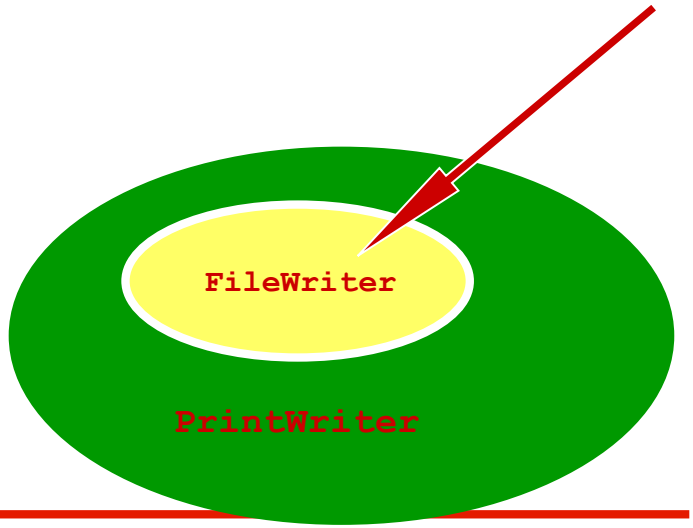
- Si incapsula `FileReader` dentro un `BufferedReader`



Stream di incapsulamento - OUTPUT

- **`PrintWriter`** scrittura "formattata"
- Fornisce i metodi per scrivere i tipi primitivi e non, su uno stream di testo.
- Ad esempio: `print(boolean b)`, `print(double d)`, `print(String s)`…
- O la loro versione con il terminatore di linea: `println(boolean b)`, `println(double d)`, `println(String s)`…

- Si incapsula `FileWriter` dentro un `PrintWriter`



Esempio – INPUT/OUTPUT

```
import java.io.*;
public class CopyLines {

    public static void copyLines(String filename)
                        throws IOException {

        FileReader fr = new FileReader(filename);
        BufferedReader input = new BufferedReader(fr);
        FileWriter fw = new FileWriter("Copia_di_" + filename);
        PrintWriter output = new PrintWriter(fw);

        String linea = input.readLine();
        while (linea != null) {
            output.println(linea);
            linea = input.readLine();
        }
        input.close();
        output.close();
    }
```

. . .

Esempio – INPUT/OUTPUT

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

        try {
           copyLines("testo.txt");
        }
        catch(IOException ex) {
            System.out.println("Errore di I/O.");
        }
    }
}
```

Esempio – INPUT/OUTPUT

```
import java.io.*;
public class CopyLines {

    public static void copyLines(String filename)
                        throws IOException {

        BufferedReader input =
             new BufferedReader(new InputStreamReader(System.in));
        FileWriter fw = new FileWriter(filename);
        PrintWriter output = new PrintWriter(fw);

        String linea = input.readLine();
        while (linea != null) {
            output.println(linea);
            linea = input.readLine();
        }
        input.close();
        output.close();
    }
```

Lettura da tastiera
e scrittura su un file

. . .

Lettura di un file di testo - Esempio

- Vediamo come viene gestita la lettura di un file di testo con un ulteriore esempio
- Supponiamo di voler leggere un file di testo (inventory.dat) che contiene l'inventario di una cartoleria.
- Ogni riga del file è un prodotto e per ogni prodotto abbiamo nome, quantità e prezzo unitario, separati da spazi:

```
Quaderno 14 1.35
Matita 132 0.32
Penna 58 0.92
Gomma 28 1.17
Temperino 25 1.75
Colla 409 3.12
Astuccio 142 5.08
```

La classe InventoryItem

- I dati letti vengono messi in un array di oggetti di classe InventoryItem definita così:

```
public class InventoryItem
{
  private String name;
  private int units;
  private float price;

  public InventoryItem(String nm, int num, float pr)
  {
    name = nm; units = num; price = pr;
  }
  public String toString()
  {
    return name + ": " + units + " a euro " + price;
  }
}
```

Accesso e lettura del file

- Per accedere al file possiamo utilizzare la classe **FileReader**
- Il costruttore di questa classe prende come parametro il nome del file da leggere e lo apre in lettura
- FileReader è però uno stream di dati e offre solo le funzionalità base: lettura a caratteri singoli
- Sappiamo però come risolvere questo problema: ricorriamo allo stream di manipolazione **BufferedReader** e lo agganciamo al FileReader:

```
FileReader fr = new FileReader("inventory.dat");
BufferedReader inFile = new BufferedReader(fr);
```

- Possiamo quindi utilizzare `inFile.readLine()` per leggere le righe del file
- Quando il file termina `readLine()` restituisce `null`

StringTokenizer

- Ci rimane però un problema: all'interno di ogni riga abbiamo più informazioni separate da spazi e quindi dobbiamo scomporla
- La classe **StringTokenizer**, inclusa nel package **java.util** svolge proprio questo compito
- Il costruttore prende come parametro la stringa da scomporre e con il metodo nextToken() possiamo estrarre le singole sottostringhe e convertirle:

```
...
tokenizer = new StringTokenizer (line);
name = tokenizer.nextToken();
units = Integer.parseInt (tokenizer.nextToken());
price = Float.parseFloat (tokenizer.nextToken());
...
```

L'esempio completo - 1

- **N.B.** La lettura del file è sotto gestione di eccezioni

```
import java.io.*;
import java.util.StringTokenizer;

public class CheckInventory
{
  public static void main (String[] args)
  {
    String line, name;
    int units, count = 0;
    float price;

    InventoryItem[] items = new  InventoryItem[100];
    StringTokenizer tokenizer;
```

L'esempio completo - 2

```
try
{
  FileReader fr = new FileReader("inventory.dat");
  BufferedReader inFile = new BufferedReader(fr);
  line = inFile.readLine();
  while (line != null)
  {
    tokenizer = new StringTokenizer(line);
    name = tokenizer.nextToken();
    try
    {
      units = Integer.parseInt(tokenizer.nextToken());
      price = Float.parseFloat (tokenizer.nextToken());
      items[count++] = new InventoryItem(name,units,price);
    }
    catch (NumberFormatException e)
    { System.out.println("Error in input. Line:"+line);}
    line = inFile.readLine();
  }
```

L'esempio completo - 3

```
      inFile.close();

      // Scrive a video i dati letti
      for (int scan=0; scan<count; scan++)
        System.out.println(items[scan]);
    }

    // Gestione delle eccezioni in cascata

    catch (FileNotFoundException e)
    {
      System.out.println("File " + file + " not found.");
    }
    catch (IOException e)
    {
      System.out.println(e);
    }
  }
}
```

Scrittura un file di testo

- Vediamo con un esempio come si scrive in un file di testo
- Il programma scrive su un file la tavola pitagorica
- Usiamo un oggetto di classe FileWriter
- File Writer però è uno stream di dati e fornisce solo le funzionalità base
- Procediamo quindi come al solito agganciando uno stream di manipolazione – PrintWriter – che consente di scrivere agevolmente righe di testo
- In questo esempio non gestiamo le eccezioni e quindi siamo obbligati a dichiarare che main() può emettere eccezioni di tipo IOException (meglio evitarlo!!)

Esempio completo

```
import java.io.*;

public class Tabelline
{
  public static void main (String[] args) throws IOException
  {
    FileWriter fw = new FileWriter("tabelline.txt");
    PrintWriter outFile = new PrintWriter(fw);
    for (int i=1; i<=10; i++)
    {
      for (int j=1; j<=10; j++)
      {
       outFile.print((i*j)+" ");
      }
      outFile.println();
    }
    outFile.close();
  }
}
```

Classe `File`

- La classe **`File`** fornisce l'accesso a file e directory in modo indipendente dal sistema operativo.
- Mette a disposizione una serie di metodi per ottenere informazioni su un certo file o directory, e per visualizzare e modificarne gli attributi.

- A proposito di indipendenza…

Ogni sistema operativo utilizza convenzioni diverse per separare le varie directory in un *path.* Esempio: in Linux " / ", in Windows " \ ";

- Costruttore:

```
public File(String path)
```

Classe `File` – metodi utili

- `public String getName()` restituisce il nome dell'oggetto
- `public String getAbsolutePath()` restituisce il percorso assoluto dell'oggetto
- `public boolean exist()` restituisce vero se l'oggetto File esiste
- `public boolean isDirectory()` restituisce vero se l'oggetto File è una directory
- `public long length()` restituisce la lunghezza in byte dell'oggetto
- `public boolean renameTo(File dest)` rinomina l'oggetto
- `public boolean delete()` cancella l'oggetto File
- `public boolean mkdir()` crea una directory che corrisponde all'oggetto File
- `public String[] list()` restituisce un vettore contenente il nome di tutti file della directory associata all'oggetto File

classe `RandomAccessFile`

- Consente l'accesso in modo RANDOM (cioè non sequenziale), ai file.
- Consente l'accesso ad un file contemporaneamente in scrittura e il lettura.
- Implementa le interfacce DataInput e DataOutput, rendendo possibile la scrittura in file di tutti gli oggetti e i tipi primitivi.

- Costruttori:

```
public RandomAccessFile(String file, String mode)
public RandomAccessFile(File file, String mode)
```

- Il parametro `mode` è di tipo `String` e specifica la modalità di accesso al file: `"r"` oppure `"rw"`

- Oltre ai metodi `read/write`, offre i metodi:

```
long getFilePointer(), seek(long pos), skipBytes(long pos )
```

serializzazione di oggetti

- Serializzare un oggetto significa salvare un oggetto scrivendo una sua rappresentazione binaria su uno stream di byte

- Analogamente, deserializzare un oggetto significa ricostruire un oggetto a partire dalla sua rappresentazione binaria letta da uno stream di byte

- Le classi `ObjectOutputStream` e `ObjectInputStream` offrono questa funzionalità per qualunque tipo di oggetto.

serializzazione di oggetti

- Le due classi principali sono:
- `ObjectInputStream`
  - **legge oggetti** serializzati salvati su stream, tramite il metodo `readObject()`
  - offre anche metodi per leggere i tipi primitivi di Java
- `ObjectOutputStream`
  - **scrive un oggetto** serializzato su stream, tramite il metodo `writeObject()`
  - offre anche metodi per scrivere i tipi primitivi di Java

serializzazione di oggetti

- Una classe che voglia essere "serializzabile" deve implementare l'interfaccia `Serializable`

- È una interfaccia vuota, che serve come marcatore

  (il compilatore rifiuta di compilare una classe che usa la serializzazione senza implementare tale interfaccia)

- Vengono scritti / letti tutti i dati non static e non transient dell'oggetto, inclusi quelli ereditati (anche se privati o protetti)
  - Le variabili transient sono variabili d'istanza che non si vuole serializzare.

serializzazione di oggetti

- Se un oggetto contiene riferimenti ad altri oggetti, viene invocata ricorsivamente `writeObject()` su ognuno di essi
  - si serializza quindi, in generale, un intero grafo di oggetti
  - l'opposto accade quando si deserializza

- Se uno stesso oggetto è referenziato più volte nel grafo, viene serializzato una sola volta, affinché `writeObject()` non cada in una ricorsione infinita.

Esempio - serializzazione

ESEMPIO di classe serializzabile

```
public class PuntoCartesiano
   implements java.io.Serializable {
  private int x;
  private int y;

  public PuntoCartesiano(int x, int y) {
      this.x = x;
      this.y = y;
   }
  public PuntoCartesiano() { x = y = 0; }

  public float getAscissa()  { return x; }
  public float getOrdinata() { return y; }
}
```

Esempio - serializzazione

SCRITTURA SU FILE...

```
import java.io.*;

public class ScriviPunto {

   public static void scriviPunto(String filename)
                                throws IOException {

     PuntoCartesiano point = new PuntoCartesiano(3, 10);
     FileOutputStream fos =
               new FileOutputStream(filename);
     ObjectOutputStream oos =
               new ObjectOutputStream(fos);
     oos.writeObject(point);
     oos.flush();
     oos.close();
   }
.....
```

Esempio - serializzazione

```
…
public static void main(String[] args) {
        try {
            scriviPunto("punti.bin");
        }
        catch(IOException ex) {
            System.out.println("Errore di I/O.");
             System.exit(1);
        }
    }
}
```

Esempio - serializzazione

LETTURA DA FILE...

```
import java.io.*;
public class LeggiPunto {
  public static void leggiPunto(String filename)
              throws IOException, ClassNotFoundException {

    FileInputStream fis = new FileInputStream(filename);
    ObjectInputStream ois = new ObjectInputStream(fis);

    PuntoCartesiano point = (PuntoCartesiano)ois.readObject();
    ois.close();

    System.out.println("Punto (" +
       point.getAscissa() +
       ", " + point.getOrdinata() + ")");
    }
```

Il cast è necessario, perché `readObject()` restituisce un `Object`

.....

Esempio - serializzazione

```
   ……
   public static void main(String[] args) {
      try {
         leggiPunto("punti.bin");
      }
      catch(IOException ex) {
         System.out.println("Errore I/O: " + ex.getMessage());
         System.exit(1);
      }
       catch(ClassNotFoundException ex) {
         System.out.println("Errore: " + ex.getMessage());
         System.exit(2);
      }

   }
}
```

Esercizio
Uso di stream Java nel web

Comunicare con il mondo

- Praticamente ogni programma ha la necessità di comunicare con il mondo esterno
  - Con l'utente attraverso tastiera e video
  - Con il file system per leggere e salvare dati
  - Con altre applicazioni sullo stesso computer
  - Con altre applicazioni su altri computer collegati in rete
  - Con dispositivi esterni attraverso porte seriali o USB
- Java gestisce tutti questi tipi di comunicazione in **modo uniforme** usando un unico strumento: lo **stream**
  - Anche la lettura di documenti online viene gestita in questo modo

Breve introduzione ai concetti di rete

- **Gli indirizzi internet identificano in modo univoco ogni computer sulla rete. Utilizzano 4 byte (dot IP notation) come per esempio 220.210.34.7**
- **Il Domain Name System ci permette un facile modo per ricordare questo schema. I server DNS traducono il nome del dominio es "amazon.com" nella sua lista di IP.**
- **I Servers sono quei computer con delle risorse (come ad esempio stampanti e dischi) che devono essere condivisi. Clients sono quelle entità che vogliono usare queste risorse.**
  - I Servers "ascoltano" le loro porte sulle **socket** (connessioni) aspettando che un client si connetta con una richiesta. Sono **multithreaded** per permettere a molte richieste di essere gestite simultaneamente.

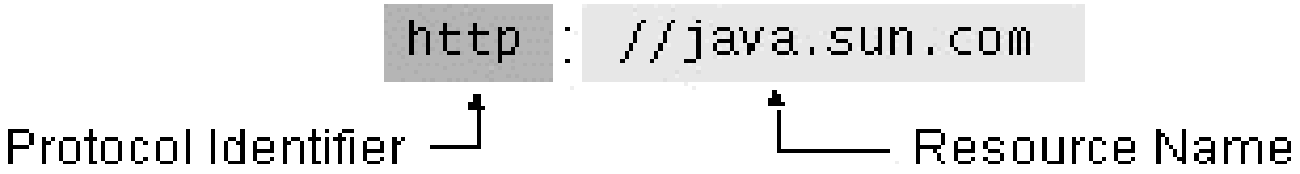
Java Networking

- L'architettura Java è network-ready e comprende package che permettono all'utente di gestire file, richiedere risorse, e anche richiamare metodi tramite la rete.

- Package `java.net`
  Concetti (classi) fondamentali per TCP (connessione):
  - `Socket, ServerSocket, URL, URLConnection`

Java URL e connessioni

- La classe **URL** (Uniform Resource Locator) cattura il concetto di puntatore a risorsa sul World Wide Web.
- Una risorsa può essere qualcosa di semplice come un file o una directory, o un riferimento a qualche oggetto più complicato, come una query a un database o a un motore di ricerca:
  - `http://localhost/index.html`
  - `file:///autoexec.bat`
  - `...`
- Come si crea un URL per identificare una risorsa:



  - L'identificativo del protocollo (Protocol Identifier)
  - Il nome della risorsa (Resource Name)

```
new URL("http://www.unife.it/ing/informazione");
```

Java URL e connessioni

- Per aprire una connessione, si invoca sull'oggetto `URL` il metodo `openConnection()`:

```
URLConnection c = url.openConnection();
```

- Il risultato è un oggetto `URLConnection`, che rappresenta una "connessione aperta"
  - in pratica, così facendo si è stabilito un canale di comunicazione verso l'indirizzo richiesto

- Per connettersi tramite tale connessione:

```
c.connect();
```

Java URL e connessioni

- Per comunicare si recuperano dalla connessione i due stream (di ingresso e di uscita) a essa associati, tramite i metodi:

- `public InputStream getInputStream()`
  - restituisce lo stream di input da cui leggere i dati (byte) che giungono dall'altra parte

- `public OutputStream getOutputStream()`
  - restituisce lo stream di output su cui scrivere i dati (byte) da inviare all'altra parte

Poi, su questi stream si legge/scrive come su qualunque altro stream di byte.

Java URL e connessioni

Programma per connettersi all'URL dato, e visualizzare il contenuto dello stream, supponendo che esso contenga testo.

```
import java.io.*;
import java.net.*;
public class EsempioURL {
 public static void main(String args[]) {
    URL u = null;
    try { u = new URL(args[0]); }
    catch (MalformedURLException e) {
    System.err.println("URL errato: " + u);
    }
    URLConnection c = null;
    try {
     c = u.openConnection(); c.connect();
     InputStreamReader is = new InputStreamReader(c.getInputStream());
     BufferedReader r = new BufferedReader(is);
     String line = r.readLine();
      while(line != null) {
        System.out.println(line);
        line = r.readLine();
      }
    } catch (IOException e) { System.err.println(e);}
 }
}
```

Possiamo provarlo sia coi file che con gli indirizzi internet.
NOTA: Ricordarsi il protocollo!!