PACKAGE DI I/O

Dott. PhD Denis Ferraretti

denis.ferraretti@unife.it

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.

☑ **Uno STREAM è quindi un'astrazione di alto livello che produce o consuma informazioni: rappresenta una connessione ad un canale di comunicazione.**

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 ci mette a disposizione anche un altro tipo di stream che hanno come obiettivo quello di fare una elaborare i dati in ingresso o in uscita.

Non si collegano direttamente ad una sorgente o destinazione d 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

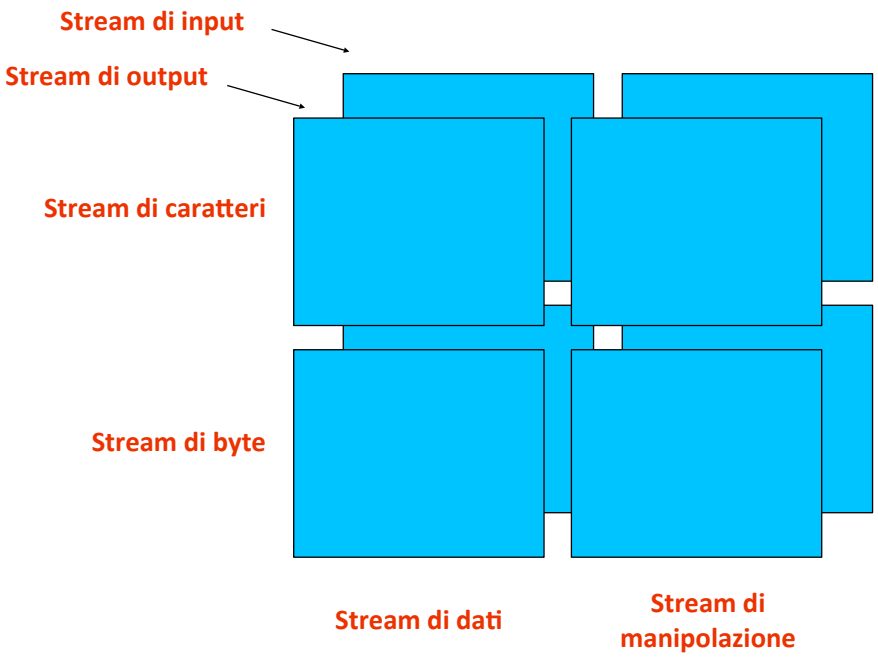
Possiamo quindi classificare gli stream sulla base di tre criteri:

1. **Direzione**: input o output
2. **Tipo di dati**: byte o caratteri
3. **Scopo**: collegamento con una dispositivo/file o manipolazione di un altro stream

Le tre classificazioni sono indipendenti (ortogonali) fra loro

Ogni stream ha 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"*

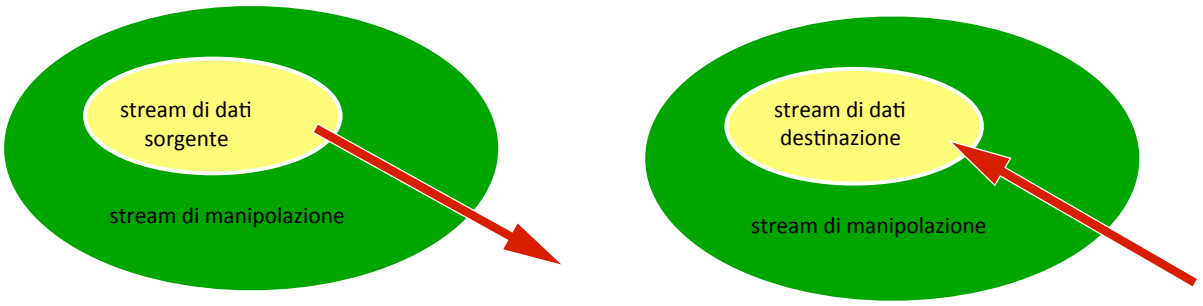
- **alcuni tipi di stream rappresentano sorgenti di dati o dispositivi di uscita [stream di dati]**
  - **file, connessioni di rete,**
  - **array di byte, ...**

- **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.**

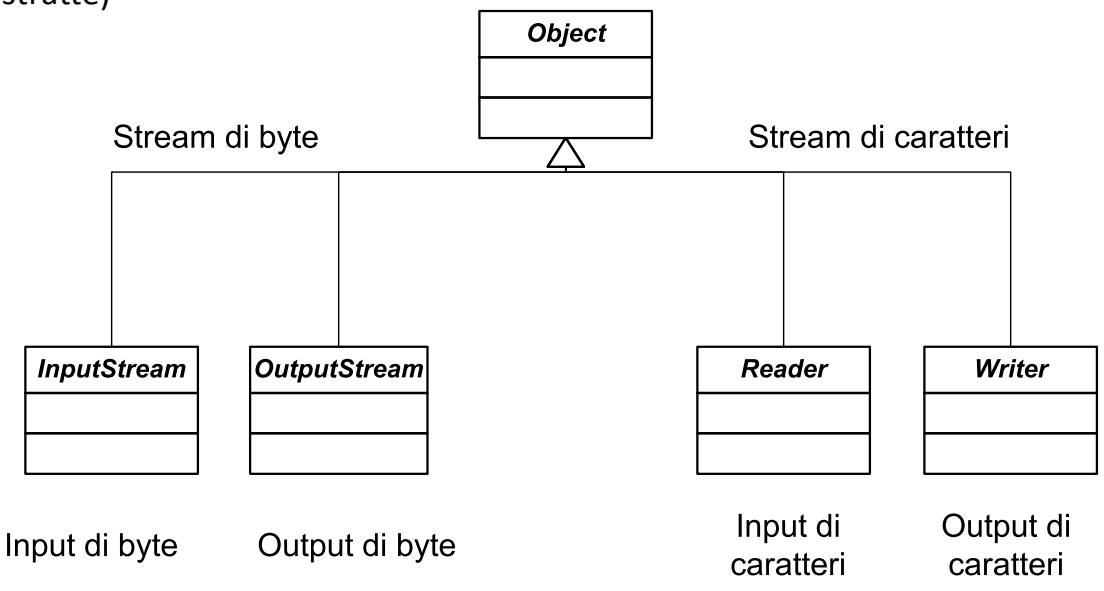
Massima flessibilità, minimo sforzo.



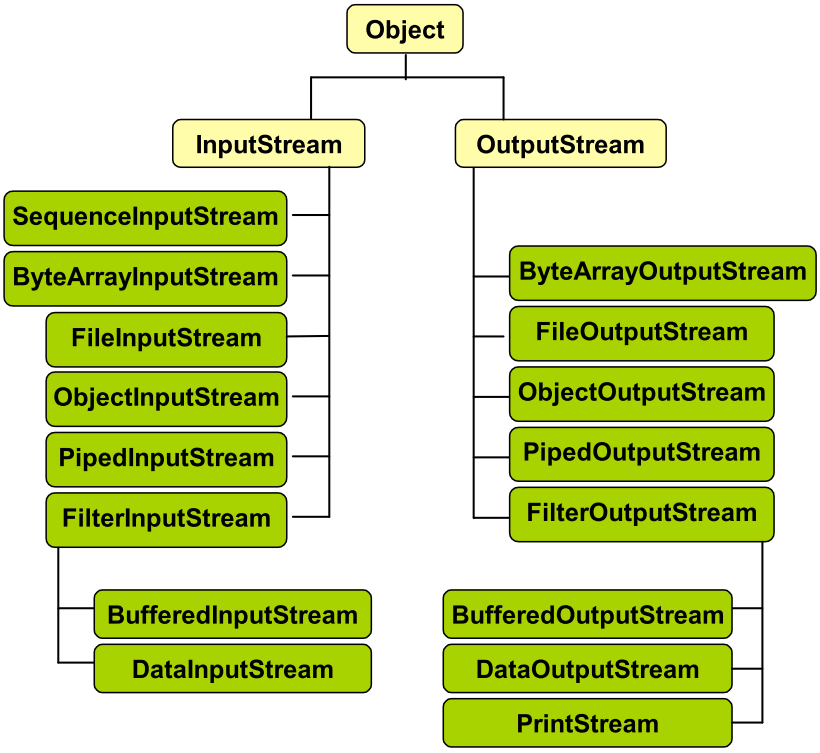
L'albero genealogico

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

`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

forza l'emissione

chiusura

`write()` è astratto e deve essere implementato in modo specifico dalla classi concrete

N.B. Tutti i metodi possono generare eccezioni

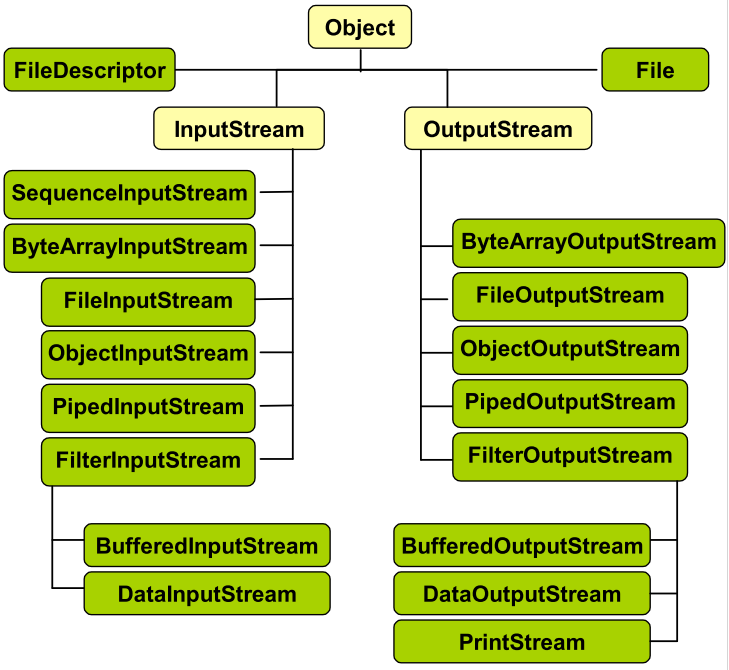
stream di byte

- 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 manipolazione, cioè pensati per aggiungere a un altro stream nuove funzionalità.

Esempio:

I/O bufferizzato, filtrato,…

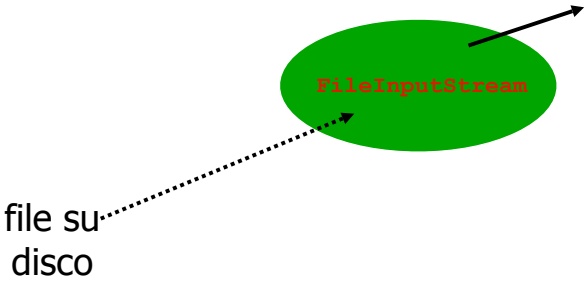
I/O di numeri, di oggetti,...



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

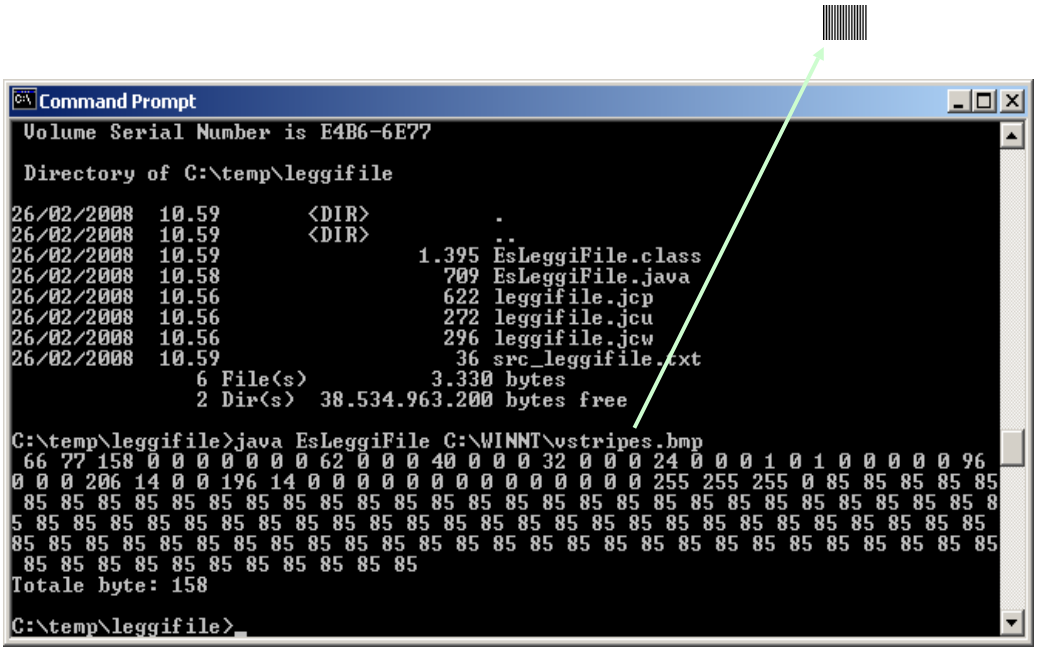
```
import java.io.*;
public class EsLeggiFile1 {
   public EsLeggiFile1(String filename) throws
                  FileNotFoundException, IOException {

   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 {
      EsLeggiFile1 oggetto = new EsLeggiFile1(args[0]);
   }
   catch (FileNotFoundException ex) {
    System.out.println("File non trovato");
           System.exit(1);
       }
       catch(IOException ex) {
           System.out.println("Errore di input");
           System.exit(2);
   }
   }
}
```

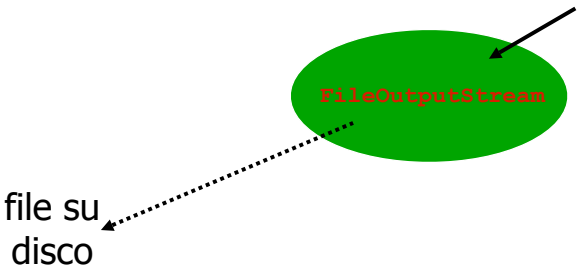
input da file – esempio



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

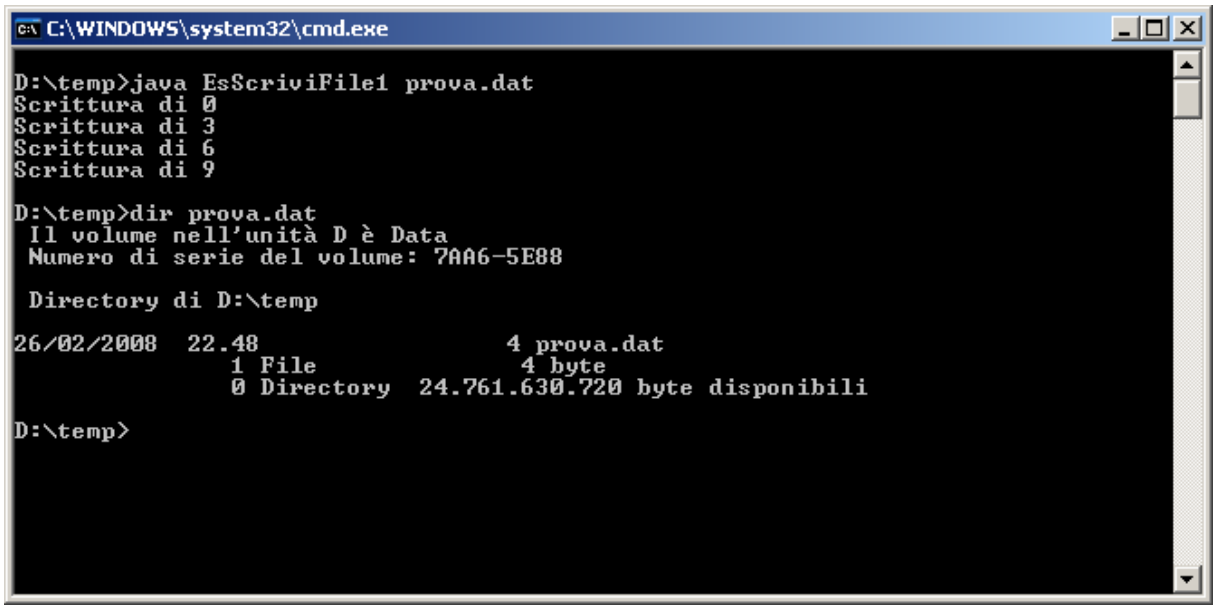
```
import java.io.*;

public class EsScriviFile1 {

    public EsScriviFile1(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 {
          EsScriviFile1 oggetto = new EsScriviFile1(args[0]);
      }
      catch(FileNotFoundException e) {
        System.out.println("Impossibile aprire file");
        System.exit(1);
      }
      catch(IOException ex) {
        System.out.println("Errore di output");
        System.exit(2);
      }
  }
}
```


stream di byte – output su file



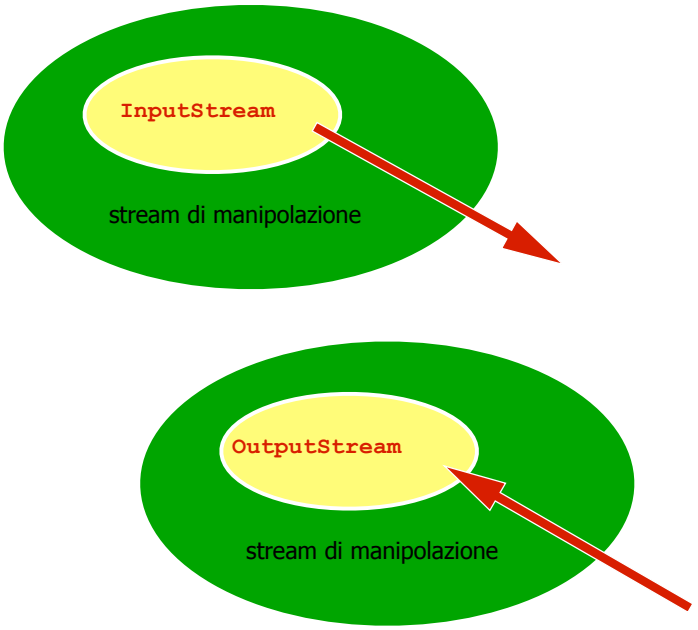
Esperimenti

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

stream di manipolazione

- Gli STREAM di manipolazione 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 manipolazione - 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 manipolazione - 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`**

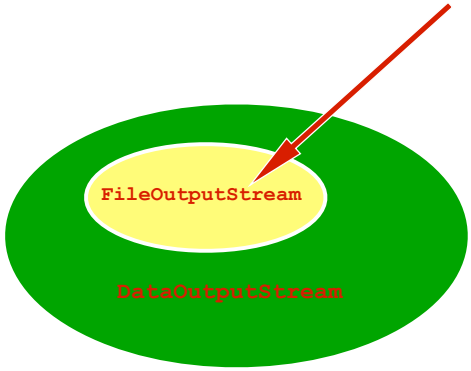
  definisce un metodo per scrivere oggetti **"serializzati"**; offre anche metodi per scrivere i tipi primitivi di Java

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 EsScriviFile2(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

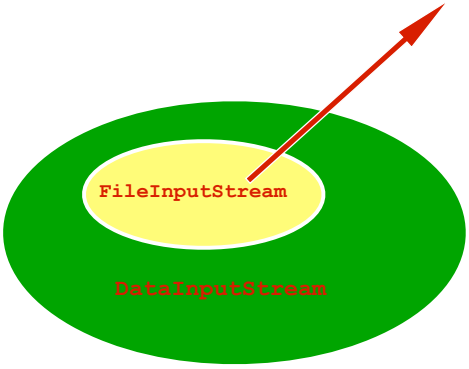
```
public static void main(String[] args) {

  try {
    EsScriviFile2 oggetto = new EsScriviFile2(args[0]);
  }
  catch(FileNotFoundException e) {
     System.out.println("Impossibile aprire file");
     System.exit(1);
  }
  catch(IOException ex) {
     System.out.println("Errore di output");
     System.exit(2);
   }
 }
}
```

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`**



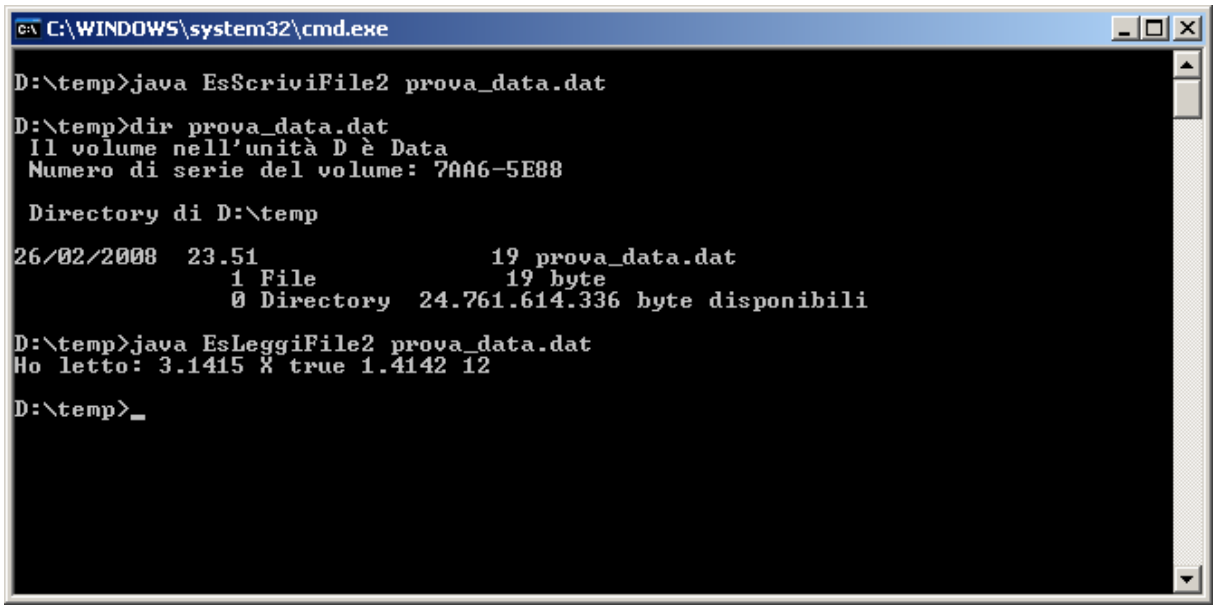
esempio - INPUT

```
import java.io.*;
public class EsLeggiFile2 {
  public EsLeggiFile2(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 {
              EsLeggiFile2 oggetto = new EsLeggiFile2(args[0]);
          }
          catch(FileNotFoundException e) {
             System.out.println("Impossibile aprire file");
             System.exit(1);
          }
          catch(IOException ex) {
             System.out.println("Errore di output");
             System.exit(2);
         }
      }
}
```


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

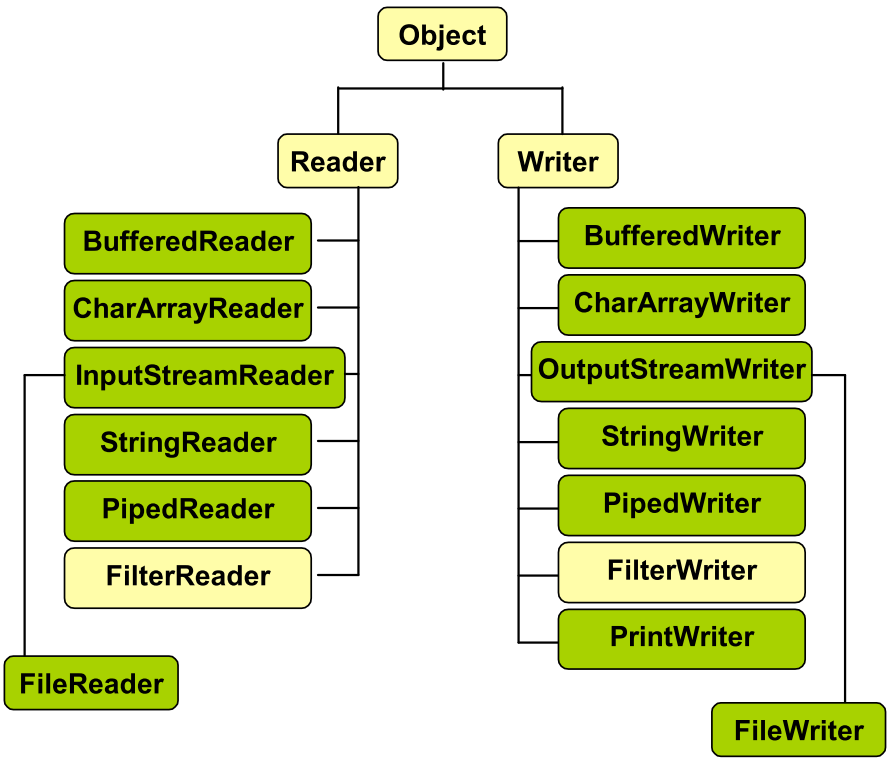
Hanno nomi analoghi e struttura analoga

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 Reader
{
  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

chiusura

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

chiusura

Esistono più versioni di write() (overloading)

stream di caratteri

Cosa cambia rispetto agli stream binari ?

- 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.

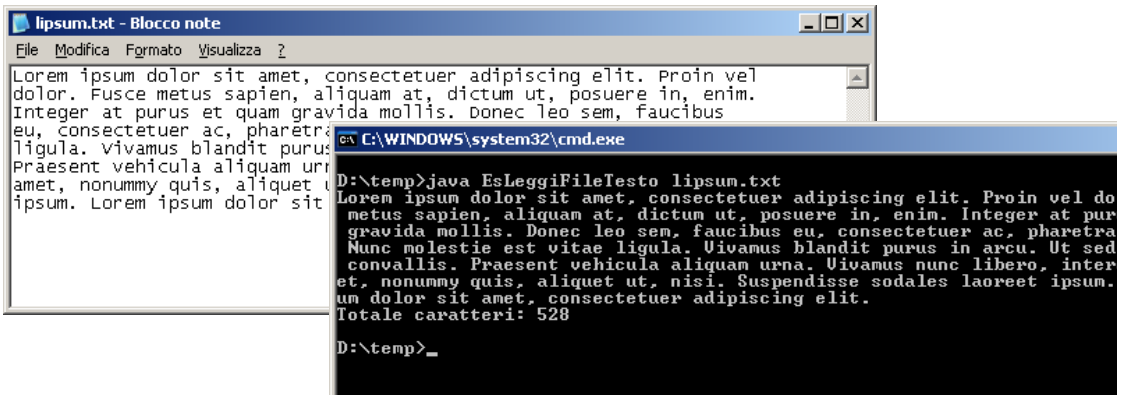
esempio - input da file

```
import java.io.*;
public class EsLeggiFileTesto {
    public EsLeggiFileTesto(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 {
      EsLeggiFileTesto oggetto =
       new EsLeggiFileTesto(args[0]);
    }
    catch(IOException ex) {
      System.out.println("Errore di I/O.");
      System.exit(1);
    }
  }
}
```



stream di manipolazione

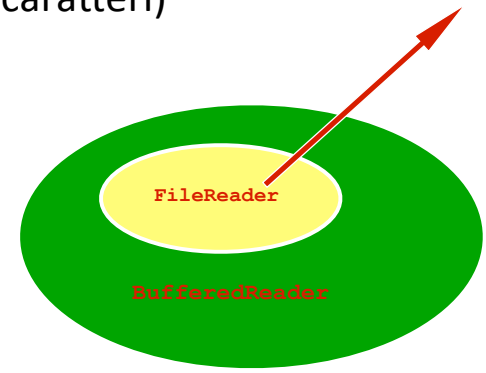
- Le operazioni di I/O con caratteri in genere coinvolgono tanti caratteri alla volta.
- In genere l'unità di base per lavorare con i caratteri è la **LINEA:** una stringa di caratteri con un terminatore di linea alla fine (in win "\r\n" carriage-return/line-feed).
- Anche nel caso degli stream di caratteri, possiamo utilizzare gli **stream di manipolazione**, che estendono le funzionalità dei **`Reader`** e **`Writer`** fornendo metodi idonei al trattamento delle linee di testo.

stream di manipolazione - INPUT

`BufferedReader` lettura bufferizzata

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

- **Si incapsula** **`FileReader`**
  dentro un **`BufferedReader`**

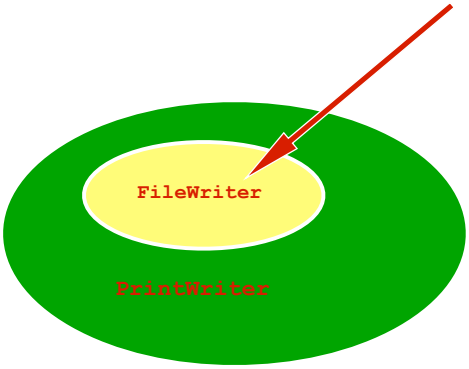


stream di manipolazione - OUTPUT

`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:
  `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 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 cp = new CopyLines(args[0]);
        }
        catch(IOException ex) {
            System.out.println("Errore di I/O.");
            System.exit(1);
        }
    }
}
```

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 strem 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, umlaut 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

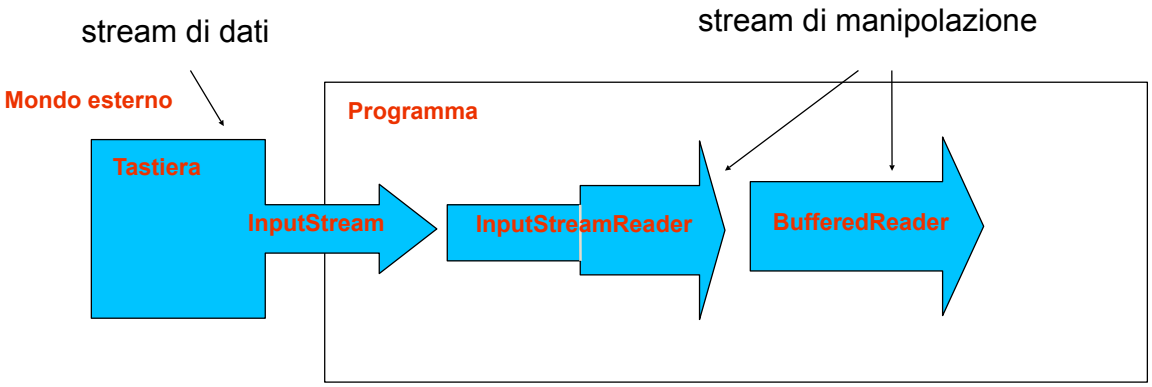
Per gestire correttamente la tastiera useremo quindi una sequenza di istruzioni di questo tipo:

```
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

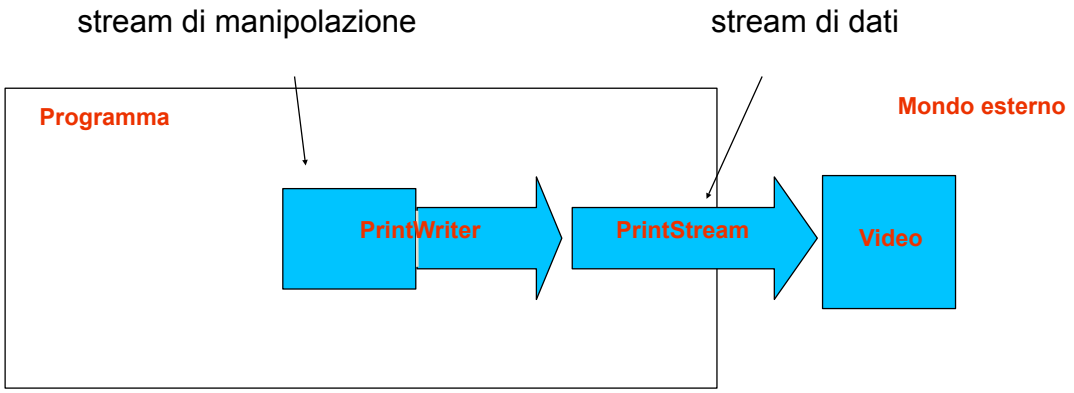
Gestione del video: soluzione completa

Potremo quindi scrivere:

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

E utilizzarlo nello stesso modo con cui useremmo System.out

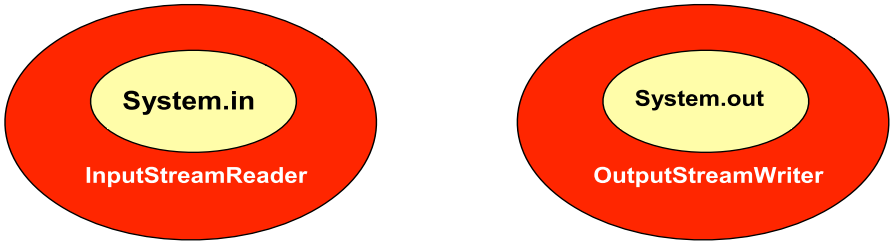
```
video.println(12);
video.println("Ciao");
video.println(13,56);
```



riassumendo...

- **`System.in`** può essere **"interpretato come un `Reader`"** incapsulandolo dentro a un **`InputStreamReader`**

- **`System.out`** può essere **"interpretato come un `Writer`"** incapsulandolo dentro a un **`OutputStreamWriter`**



il caso dell'I/O da console

IN PRATICA

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

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

MA! Per quanto detto precedentemente, è meglio scrivere:

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

Equivalente a:
`System.out.println(…)`

OPPURE

```
BufferedReader tastiera =
   new BufferedReader(new InputStreamReader(System.in));
PrintWriter video =
   new PrintWriter(new OutputStreamWriter(System.out));
```


esempio – scrittura su file

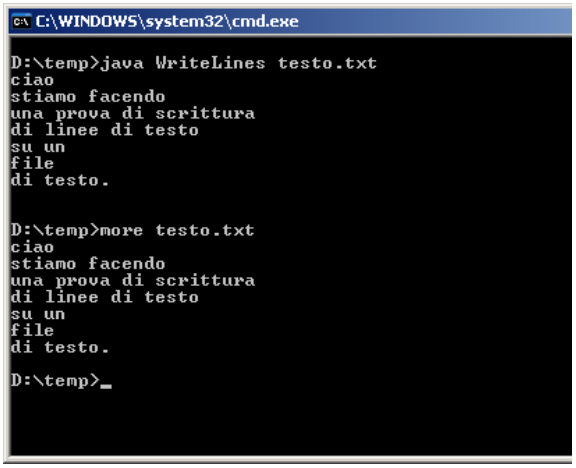
- Scrivere su un file di testo le linee inserite da console, fino a quando non viene inserita la linea vuota.
- Bisogna incapsulare `System.in` in un `InputStreamReader` che a sua volta è incapsulato da un `BufferedReader`
- Si può inoltre incapsulare `FileWriter` in un `PrintWriter`
- **Leggere linee di testo e inserirle nel file fino a quando non si incontra una linea vuota.**

esempio – scrittura su file

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


    public WriteLines(String filename) throws IOException {

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

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


esempio – scrittura su file

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

        try {
            WriteLines cp = new WriteLines(args[0]);
        }
        catch(IOException ex) {
            System.out.println("Errore di I/O.");
            System.exit(1);
        }
    }
}
```



Lettura di un file di testo - Esempio

Vediamo come si viene gestita la lettura di un file di testo con un 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

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'oggeto File è una direcotry
- **`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)

serializzazione di oggetti

- Se un oggetto **contiene riferimenti ad altri oggetti**, si invoca 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 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

...SCRITTURA SU FILE

```
…
public static void main(String[] args) {
        try {
        ScriviPunto sp = new ScriviPunto(args[0]);
        }
        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 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

...LETTURA DA FILE

```
   ……
   public static void main(String[] args) {
      try {
    LeggiPunto lp = new LeggiPunto(args[0]);
      }
      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);
      }

  }
}
```