Università degli Studi della Campania *Luigi Vanvitelli*

Università di degli Studi della Campania Luigi Vanvitelli Dipartimento di Ingegneria

Programmazione ad Oggetti

a.a. 2020-2021

Java I/O

Docente: Prof. Massimo Ficco
E-mail: massimo.ficco@unicampania.it

Obiettivi

Acquisire familiarità con i concetti dell'I/O in JAVA
Capire la differenza tra file binario e file testo
Imparare a salvare dati in un file
Imparare a leggere dati da un file

I/O Overview

I/O = Input/Output

Si intende per Input una trasmissione di dati verso il programma

Si intende per Output una trasmissione di dati dal programma verso un'altra destinazione

Input può avvenire da tastiera o da file

Output può avvenire verso il video o un file

Vantaggio dell'I/O da/verso file:

- Copia permanente
- Scambio di informazioni da un programma all'altro
- Inout automatizzato (non inserito manualmente)

Stream

In tutti i linguaggi moderni l'I/O è basato sul concetto di stream.

CONCETTO BASE: LO *STREAM*

Uno stream è *un canale di comunicazione monodirezionale* *(o di input, o di output)*
di uso generale
adatto a trasferire byte (o anche caratteri)

Streams

Lo Stream: è un oggetto che trasmette i dati verso la destinazione (video, file, atro pc, etc.) o che prende dati da una sorgente (tastiera, file, etc.)

- it acts as a buffer between the data source and destination

Input stream: è uno stream che fornisce informazioni in ingresso ad un programma

- **`System.in`** è un input stream (collegato alla tastiera).

Output stream: è uno stream che accetta informazioni in uscita da un programma

- **`System.out`** è un output stream (collegato al video)

Uno stream connette un programma ad una destinazione di I/O

- `System.out` connette un programma al video.
- `System.in` connette un programma alla tastiera.

Il package java.io

`import java.io.*;`

Mette a disposizione una libreria di classi che consentono:

Apertura di uno stream

Utilizzo di uno stream (read, write)

Chiusura di uno stream

Perchè l'I/O Java è complesso?

V:

Java I/O è molto potente, mette a disposizone un numero impressionante di opzioni

Non è difficile utilizzare la generica classe di I/O

Il problema è capire quale classe occorre utilizzare

Binary Versus Text Files

V:

Tutti i dati e programmi sono alla fine 0 e 1

- Ogni digit può assumere due valori: tipo *binary*
- *bit* è un tipo binario
- *byte* è un insieme di 8 bit

Text files: i bits rappresentano caratteri stampabili

- Il codice ASCII utilizza un byte per carattere
- I sorgenti Java sono file di testo
- Anche un qualunque altro file creato con un "text editor"

Binary files: i bits rappresentano altri tipi di informazione codificata, come istruzioni eseguibili o un dato numerico

- Questi files sono facilmente interpretati da computers non da uomini
- Non sono files "stampabili"
  - "stampabili" significa che possono essere facilmente essere interpretati da un utente umano quando vengono stampati

Java: Text Versus Binary Files

I file di testo sono più facilmente coprensibili da un uomo

I file binari sono più efficienti

- I computers trattano i file binary meglio che i files di testo

Text Files vs. Binary Files

Number: 127 (decimal)

- Text file
  - Tre bytes: “1”, “2”, “7”
  - ASCII (decimal): 49, 50, 55
  - ASCII (octal): 61, 62, 67
  - ASCII (binary): 00110001, 00110010, 00110111
- Binary file:
  - One byte (`byte`)**:** 01111110
  - Two bytes (`short`): 00000000 01111110
  - Four bytes (`int`): 00000000 00000000 00000000 01111110

Text file: un esempio

```
127 smiley
faces

0000000 061 062 067 011 163 155 151 154
          1   2   7  \t   s   m   i   l
0000010 145 171 012 146 141 143 145 163
          e   y  \n   f   a   c   e   s
0000020 012
         \n
```

Binary file: un esempio [a .class file]

```
0000000 312 376 272 276 000 000 000 061
        312 376 272 276  \0  \0  \0   1
0000010 000 164 012 000 051 000 062 007
         \0   t  \n  \0   )  \0   2  \a
0000020 000 063 007 000 064 010 000 065
         \0   3  \a  \0   4  \b  \0   5
0000030 012 000 003 000 066 012 000 002
         \n  \0 003  \0   6  \n  \0 002

...
0000630 000 145 000 146 001 000 027 152
         \0   e  \0   f 001  \0 027   j
0000640 141 166 141 057 154 141 156 147
          a   v   a   /   l   a   n   g
0000650 057 123 164 162 151 156 147 102
          /   S   t   r   i   n   g   B
0000660 165 151 154 144 145 162 014 000
          u   i   l   d   e   r  \f  \0
```

Buffering

Not buffered: ogni byte è letto/scritto su disco appena possibile

- "piccolo" ritardo per ogni byte
- Una operazione su disco per ogni byte --- alto overhead

Buffered: lettura/scrittura per gruppi byte (chunks)

- Del ritardo per alcuni bytes
  - Si assuma un buffer di 16-byte
  - Lettura: si accede ai primi 4 bytes, occorre aspettare che si leggano tutti e 16 bytes dal buffer in memoria prima di accedere nuovamente a disco
  - Scrittura: si salvano i primi 4 bytes, occorre aspettare per tutti e 16 prima di scrivere su disco
- Una operazione su disco per buffer -- basso overhead

IL PACKAGE `java.io`

Il package `java.io` distingue fra:

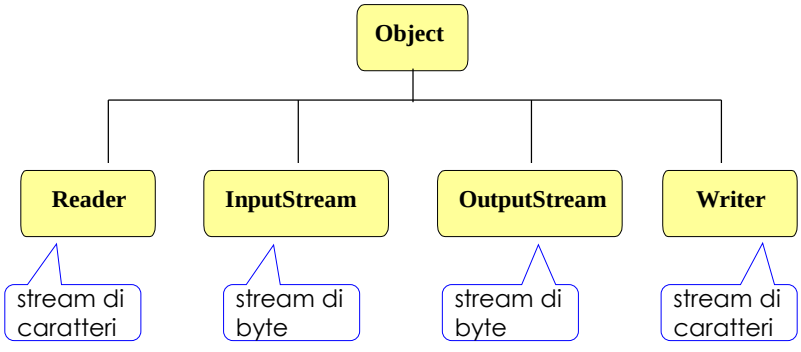
- ***stream di byte*** (analoghi ai *file binari* del C)
- ***stream di caratteri*** (analoghi ai *file di testo* del C) (solo da Java 1.1 in poi)

Questi concetti si traducono in *altrettante famiglie di classi:*

- ***stream di byte*****:** `InputStream` **e** `OutputStream`
- ***stream di caratteri*****:** `Reader` **e** `Writer`

IL PACKAGE `java.io`

Le quattro classi base <u>astratte</u> di `java.io`



IL PACKAGE `java.io`

Tratteremo separatamente

***prima* gli stream di byte**

- **`InputStream` e `OutputStream`**

***poi* gli stream di caratteri**

- **`Reader` e `Writer`**

N.B.:

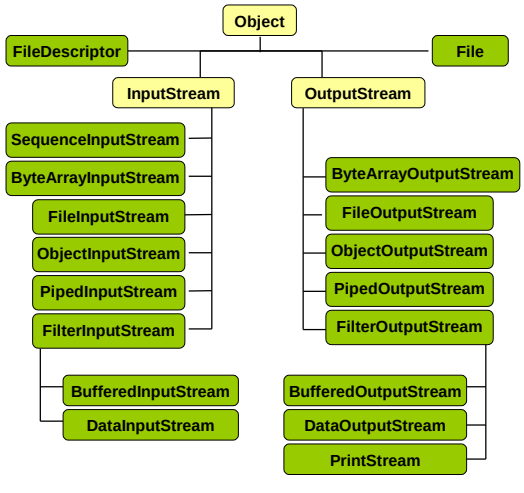
- Ogni Classe che tratta byte avrà un nome che contiene InputStream o OutputStream
- Ogni Classe che tratta caratteri avrà un nome che contiene Reader o Writer

Classi Astratte

OutputStream ed **InputStream** sono le classi capostipiti dell'I/O basato su byte.

Non possono essere utilizzate direttamente, ma definiscono solo le caratteristiche che devono possedere le classi appartenenti a questa famiglia

STREAM DI BYTE



STREAM DI BYTE V:

La classe base `OutputStream` definisce il concetto generale di "*canale di output*" operante *a byte*

il costruttore apre lo stream

`write()` scrive uno o più byte

`flush()` svuota il buffer di uscita

`close()` chiude lo stream

Attenzione: `OutputStream` è una classe astratta, quindi *il metodo write() dovrà essere realmente definito dalle classi derivate*, in modo specifico allo specifico dispositivo di uscita.

STREAM DI BYTE V:

La classe base `InputStream` definisce il concetto generale di "*canale di input*" operante *a byte*

il costruttore apre lo stream

`read()` legge uno o più byte

`close()` chiude lo stream

Attenzione: `InputStream` è una classe astratta, quindi *il metodo read() dovrà essere realmente definito dalle classi derivate*, in modo specifico alla specifica sorgente dati.

*Esempio: System.out V:

System.out è un oggetto di tipo: **PrintStream**

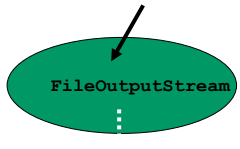
alcuni metodi di questo oggetto sono:

- print() con i suoi overload
- println() con i suoi overload
- write(int)
- write(byte a[], int off, int len)

STREAM DI BYTE - OUTPUT SU FILE V:

`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

OUTPUT SU FILE - ESEMPIO

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 `try` / `catch`

Cenni sull'Eccezioni

Handling `IOException` V:

`IOException` non può essere ignorata

- O si gestisce con un blocco catch
- Oppure si rimanda con una direttiva `throws`

Metteremo il codice per aprire un file e per scrivervi o leggervi in un blocco try-catch per gestire l'eccezione.

```
catch(IOException e)
{
  System.out.println("Problem with output...";
}
```

OUTPUT SU FILE - ESEMPIO V:

```
import java.io.*;

public class ScritturaSuFileBinario {
 public static void main(String args[]){
   FileOutputStream os = null;
   try {
       os = new FileOutputStream(args[0]);
       ….
       …..
   }
   catch(Exception e){
     System.out.println("Impo      prire file");
     System.exit(1);
   }
  // ... scrittura ...
 }
}
```

Per aprirlo in modalità append:
`FileOutputStream(args[0],true)`

OUTPUT SU FILE - ESEMPIO

Esempio: scrittura di alcuni byte a scelta

```
...
try {
 for (int x=0; x<10; x+=3) {
   System.out.println("Scrittura di " + x);
   os.write(x);
 }
} catch(Exception ex){
    System.out.println("Errore di output");
    System.exit(2);
}
...
```

OUTPUT SU FILE - ESEMPIO

Esempio d'uso:

```
C:\temp>java ScritturaSuFileBinario prova.dat
```

Il risultato:

```
Scrittura di 0
Scrittura di 3
Scrittura di 6
Scrittura di 9
```

Controllo:

```
C:\temp>dir prova.dat
16/01/01  prova.dat   4 byte
```

Esperimenti

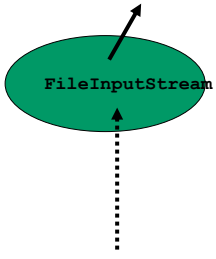
Aggiungere altri byte riaprendo il file in modo append

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



INPUT DA FILE - ESEMPIO

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 `try` / `catch`

INPUT DA FILE - ESEMPIO

```
import java.io.*;
public class LetturaDaFileBinario {
 public static void main(String args[]){
  FileInputStream is = null;
  try {
    is = new FileInputStream(args[0]);
  }
  catch(Exception e){
    System.out.println("File non trovato");
    System.exit(1);
  }
  // ... lettura ...
 }
}
```

INPUT DA FILE - ESEMPIO

La fase di lettura:

```
...
try {
 int x = is.read();
 int n = 0;
 while (x>=0) {
   System.out.print(" " + x); n++;
   x = is.read();
 }
 System.out.println("\nTotale byte: " + n);
} catch(Exception ex){
  System.out.println("Errore di input");
  System.exit(2);
}
...
```

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

INPUT DA FILE - ESEMPIO

Esempio d'uso:

```
C:\temp>java LetturaDaFileBinario question.gif
```

Il risultato:

```
71 73 70 56 57 97 32 0 32 0 161 0 0 0 0 0 255 255 255 0 128 0
191 191 191 33 249 4 1 0 0 3 0 44 0 0 0 0 32 0 32 0 0 2 143 156
143 6 203 155 15 15 19 180 82 22 227 178 156 187 44 117 193 72
118 0 56 0 28 201 150 214 169 173 237 236 65 170 60 179 102 114
91 121 190 11 225 126 174 151 112 56 162 208 130 61 95 75 249
100 118 4 203 101 173 57 117 234 178 155 172 40 58 237 122 43
214 48 214 91 54 157 167 105 245 152 12 230 174 145 129 183 64
140 142 115 83 239 118 141 111 23 120 161 16 152 100 7 3 152
229 87 129 152 200 166 247 119 68 103 24 196 243 232 215 104
249 181 21 25 67 87 9 130 7 165 134 194 35 202 248 81 106 211
128 129 82 0 0 59
Totale byte: 190
```

Incapsulamento

È scomodo utilizzare alcuni tipi di classi base per l'I/O.

Ad esempio è scomodo leggere o scrivere uno o più byte per volta in un File.

Java mette a disposizione delle classi che offrono funzionalità di I/O avanzate utilizzando quelle offerte dalle classi base

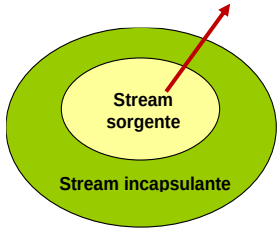
V: IL PACKAGE `java.io`

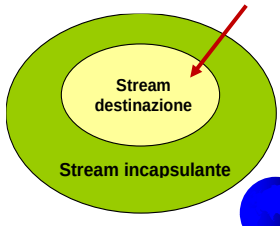
L'approccio *"a cipolla"*

Così, è possibile configurare il canale di comunicazione *con tutte e sole le funzionalità che servono…*

..senza peraltro doverle replicare e re-implementare più volte.

Massima flessibilità, minimo sforzo.

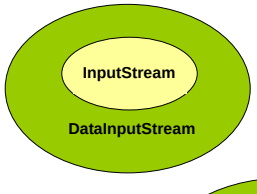


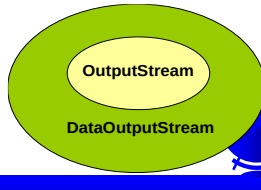


V: STREAM DI INCAPSULAMENTO

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.***





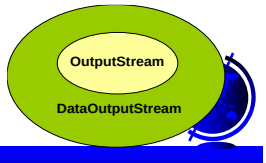
ESEMPIO 1

Scrittura di dati su file binario

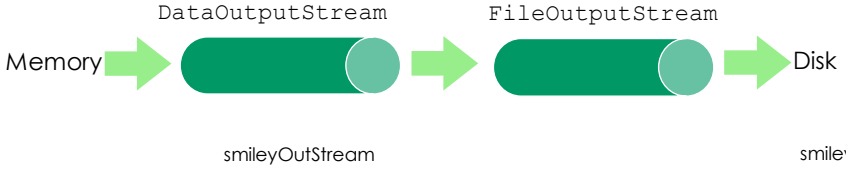
Per scrivere su un file binario occorre un `FileOutputStream`, che però consente solo di scrivere un *byte* o un *array di byte*

Volendo scrivere dei `float`, `int`, `double`, `boolean`, ... è molto più pratico un `DataOutputStream`, che ha metodi idonei

Si incapsula `FileOutputStream` dentro un `DataOutputStream`



Output File Streams




```
DataOutputStream  OutStr = new DataOutputStream ( new FileOutputStream("smiley.txt") );
```

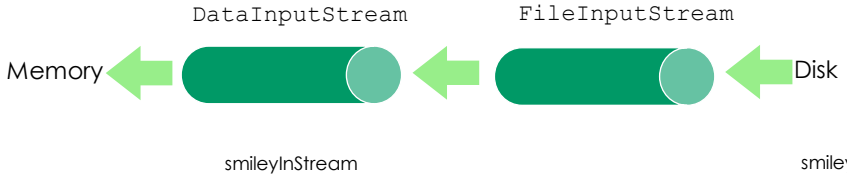
ESEMPIO 1

```
import java.io.*;
public class Esempio1 {
 public static void main(String args[]){
  FileOutputStream fs = null;
  try {
   fs = new FileOutputStream("Prova.dat");
  }
  catch(Exception e){
    System.out.println("Apertura fallita");
    System.exit(1);
  }
  // continua...
```

ESEMPIO 1 (segue)

```
  DataOutputStream os =
                  new DataOutputStream(fs);
  float   f1 = 3.1415F;  char   c1 = 'X';
  boolean b1 = true;     double d1 = 1.4142;
  try {
    os.writeFloat(f1);  os.writeBoolean(b1);
    os.writeDouble(d1); os.writeChar(c1);
    os.writeInt(12);    os.close();
  } catch (Exception e){
    System.out.println("Scrittura fallita");
    System.exit(2);
  }
 }
}
```

Input File Streams



DataInputStream smileyInStream = new DataInputStream(new DataInputStream("smiley.txt"));

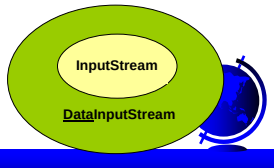
ESEMPIO 2

Rilettura 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 2

```
import java.io.*;
public class Esempio2 {
 public static void main(String args[]){
  FileInputStream  fin = null;
  try {
    fin = new FileInputStream("Prova.dat");
  }
  catch(Exception e){
    System.out.println("File non trovato");
    System.exit(3);
  }
  // continua...
```

ESEMPIO 2 (segue)

```
 DataInputStream is =
                new DataInputStream(fin);
 float f2; char c2; boolean b2; double d2;
 int i2;
 try {
  f2 = is.readFloat(); b2 = is.readBoolean();
  d2 = is.readDouble(); c2 = is.readChar();
  i2 = is.readInt();    is.close();
  System.out.println(f2 + ", " + b2 + ", "
  + d2 + ", " + c2 + ", " + i2);
 } catch (Exception e){
    System.out.println("Errore di input");
    System.exit(4);
 }
}
```

V: STREAM DI INCAPSULAMENTO - OUTPUT

`DataOutputStream`

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

`BufferedOutputStream`

- aggiunge un buffer e ridefinisce `write(`byte[] b, int off, int len`)` in modo da avere una scrittura bufferizzata

`PrintStream`

- definisce metodi per stampare come stringa valori primitivi (con `print()`) e classi standard (con `toString()`)

`ObjectInputStream`

- definisce un metodo per scrivere oggetti "serializzati"
- offre anche metodi per scrivere i tipi primitivi e gli oggetti delle classi wrapper (`Integer`, etc.) di Java

V: STREAM DI INCAPSULAMENTO - INPUT

`DataInputStream`

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

`BufferedInputStream`

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

`ObjectInputStream`

- definisce un metodo per leggere oggetti "serializzati" (salvati) da uno stream
- offre anche metodi per leggere i tipi primitivi e gli oggetti delle classi wrapper (`Integer`, etc.) di Java

STREAM DI CARATTERI

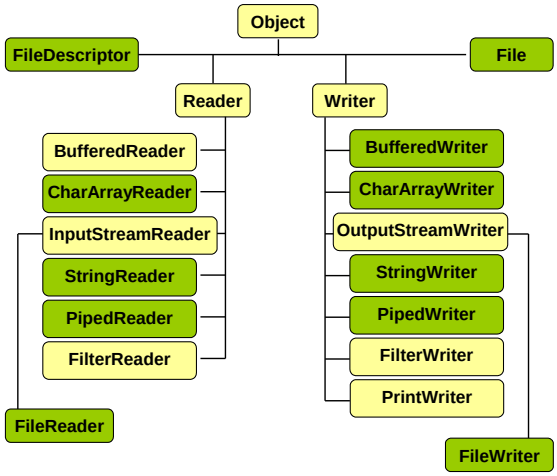
Le classi per l'I/O da stream di caratteri (`Reader` e `Writer`) sono *più efficienti* di quelle a byte

Hanno nomi analoghi e struttura analoga

Convertono correttamente la codifica UNICODE di Java in quella locale

- specifica della piattaforma in uso (tipicamente ASCII)...
- ...e della lingua in uso (essenziale per l'internazionalizzazione).

STREAM DI CARATTERI



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

- ricorda: un carattere UNICODE è lungo *due* byte
- read() restituisce -1 in caso di fine stream

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

INPUT DA FILE - ESEMPIO

```
import java.io.*;
public class LetturaDaFileDiTesto {
 public static void main(String args[]){
  FileReader r = null;
  try {
    r = new FileReader(args[0]);
  }
  catch(Exception e){
    System.out.println("File non trovato");
    System.exit(1);
  }
  // ... lettura ...
 }
}
```

INPUT DA FILE - ESEMPIO

La fase di lettura:

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

```
...
try {
 int n=0, x = r.read();
 while (x>=0) {
    char ch = (char) x;
    System.out.print(" " + ch); n++;
    x = r.read();
 }
System.out.println("\nTotale caratteri: " + n);
} catch(Exception ex){
  System.out.println("Errore di input");
  System.exit(2);
}
...
```

INPUT DA FILE - ESEMPIO

Esempio d'uso:

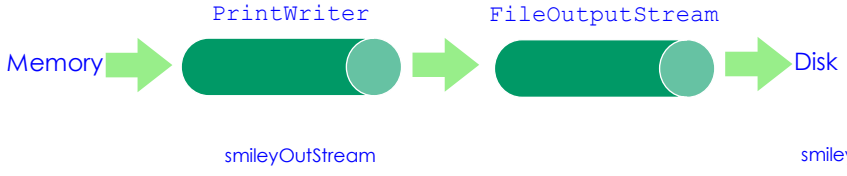
```
C:\temp>java LetturaDaFileDiTesto prova.txt
```

Il risultato:

```
N e l   m e z z o   d e l   c a m m i n   d i
n o s t r a   v i t a
Totale caratteri: 35
```

Analogo esercizio può essere svolto per la scrittura su file di testo.

Output File Streams




```
PrintWriter smileyOutStream = new PrintWriter( new FileOutputStream("smiley.txt") );
```


PrintWriter

È molto comodo eseguire la stampa su on stream di uscita utilizzando la classe PrintWriter

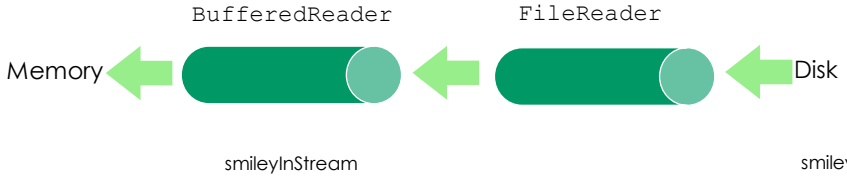
Il costruttore di PrintWriter vuole come parametro uno stream Writer.

Es:

```
PrintWriter p=new PrintWriter(new FileWriter("nomefile"));
p.println("ciao");
p.close();
```

// Il metodo print di PrintWriter quando riceve in ingresso un oggetto ne richiama il metodo toString() e stampa la String ritornata.

Input File Streams



BufferedReader smileyInStream = new BufferedReader(new FileReader("smiley.txt"));

BufferedReader

Il costruttore di BufferedReader vuole come parametro uno stream Reader.
Es:

```
BufferedReader br=new BufferedReader(new FileReader("nomefile"));
String s=br.readLine();
while(s!=null){
        System.out.println(s);
        s=br.readLine()}
```

// Il metodo readLine() restituisce null quando lo stream è terminato.

UN PROBLEMA

Gli *stream di byte* sono *più antichi* e di *livello più basso* rispetto agli stream *di caratteri*

- Un carattere UNICODE viene espresso a livello macchina come sequenza di *due* byte
- Gli *stream di byte* esistono da Java 1.0, quelli *di caratteri* esistono invece da Java 1.1

Varie classi esistenti fin da Java 1.0 usano quindi stream di byte anche quando dovrebbero usare in realtà *stream di caratteri*

Conseguenza: *i caratteri rischiano di non essere sempre trattati in modo coerente*

PROBLEMA - SOLUZIONE

Occorre dunque poter *reinterpretare* uno stream di byte come reader / writer *quando opportuno* (cioè quando trasmette caratteri)

Esistono due classi "incapsulanti" progettate proprio per questo scopo:

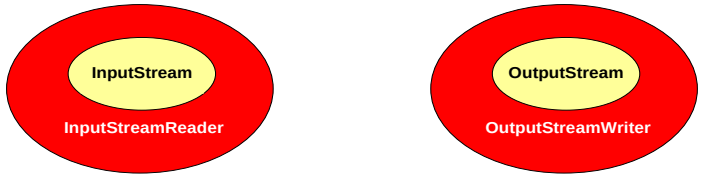
`InputStreamReader` che *reinterpreta un* `InputStream` *come un* `Reader`

`OutputStreamWriter` che *reinterpreta un* `OutputStream` *come un* `Writer`

STREAM DI CARATTERI

InputStreamReader **ingloba un** ***InputStream*** **e lo fa apparire all'esterno come un** ***Reader***

OutputStreamWriter **ingloba un** ***Output-Stream*** **e lo fa apparire fuori come un** ***Writer***



IL CASO DELL' I/O DA CONSOLE

Video e tastiera sono rappresentati dai due oggetti statici `System.in` **e** `System.out`

Poiché esistono fin da Java 1.0 (quando Reader e Writer non esistevano), **essi sono *formalmente* degli *stream di byte*...**

- `System.in` **è *formalmente* un** ***InputStream***
- `System.out` **è *formalmente* un** ***OutputStream***

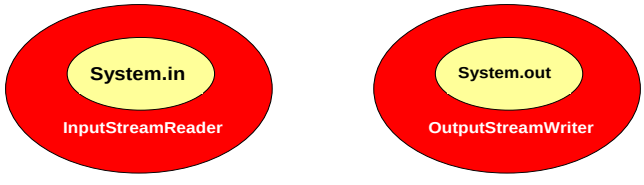
...ma in realtà sono stream di caratteri!

Per assicurare che i caratteri UNICODE siano correttamente interpretati occorre quindi incapsularli rispettivamente in un Reader e in un Writer.

IL CASO DELL' I/O DA CONSOLE V:

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

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

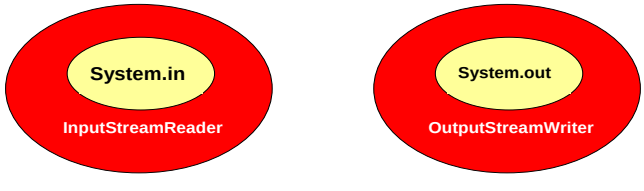


IL CASO DELL' I/O DA CONSOLE V:

Tipicamente:

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

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



Lettura da console

V:

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

ESEMPIO 3

V:

Scrittura di dati su file di testo

Per scrivere su un file di testo occorre un `FileWriter`, che però consente solo di scrivere un *carattere* o una *stringa*

Per scrivere `float`, `int`, `double`, `boolean`, … occorre *convertirli in stringhe a priori* con il metodo `toString()` della classe wrapper corrispondente, e poi scriverli sullo stream

- **Non esiste qualcosa di simile allo stream `DataOutputStream`**

ESEMPIO 3

```
import java.io.*;
public class Esempio3 {
 public static void main(String args[]){
  FileWriter fout = null;
  try {
   fout = new FileWriter("Prova.txt");
  }
  catch(Exception e){
    System.out.println("Apertura fallita");
    System.exit(1);
  }
  float   f1 = 3.1415F;  char   c1 = 'X';
  boolean b1 = true;     double d1 = 1.4142;
```

ESEMPIO 3 (segue)

```
  try {  String buffer = null;
    buffer = Float.toString(f1);
    fout.write(buffer,0,buffer.length());
    buffer = Double.toString(d1);
    fout.write(buffer,0,buffer.length());
    fout.write(c1); // singolo carattere
    buffer = Integer.toString(12);
    fout.write(buffer,0,buffer.length());
     buffer = new Boolean(b1).toString();
    fout.write(buffer,0,buffer.length());
    fout.close();
  } catch (Exception e){...}
 }
}
```

ESEMPIO 2 (segue)

Versione di write() che scrive un array di caratteri (dalla posizione data e per il numero di caratteri indicato)

```
try {
  buffer = Float.toStri...);
  fout.write(buffer,0,buffer.length());
  buffer = new Boolean(b1).toString();
  fout.write(buffer,0,buffer.length());
  buffer = Double...ng(d1);
  fout.writ...
  fout.writ...
  buffer = ...
  fout.writ...
  fout.clos...();
} catch (Exception e){...}
}
}
```

za per ogni tipo primitivo esiste un corrispondente **Simple Data Object** o, come si suol dire, una **Classe Wrapper**

Classe Boolean è l'unica a *non avere* una **funzione statica toString().** Quindi bisogna creare un oggetto Boolean e poi invocare su di esso il ***metodo toString()***

Per ogni tipo primitivo esiste un corrispondente **Simple Data Object** o, come si suol dire, una **Classe Wrapper**

Tipo primitivo	Classe Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Ricapitolando: lettura

Azione	Origine	Filtro	Oggetto
Lettura da console	System.in	InputStreamReader	BufferedReader
Lettura da File	FileReader	/	BufferedReader
Lettura oggetto	FileInputStream	/	ObjectInputStream
Lettura dati da file bin	FileInputStream	/	DataInputStream

Ricapitolando: scrittura

Azione	Origine	Filtro	Oggetto
Scrittura a video	System.out	/	PrintStream
Scrittura su File	FileWriter	/	PritnWriter
Scrittura oggetto	FileOutputStream	/	ObjectOutputStream
Scrittura dati su file bin	FileOutputStream	/	DataOutputStream

V:

A cura del

Prof. Massimo Ficco

e del

Prof. Salvatore Venticinque