IL PACKAGE `java.io`

Il package *`java.io`* definisce i concetti base per gestire l'I/O *da qualsiasi sorgente e verso qualsiasi destinazione.*

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)***

IL PACKAGE `java.io`

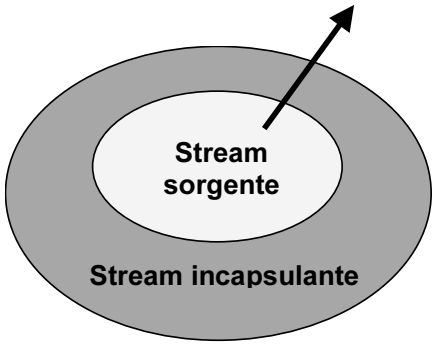
L'obiettivo

- **fornire un'astrazione che incapsuli tutti i dettagli di una *sorgente dati* o di un *dispositivo di output***
- **fornire un modo semplice e flessibile per *aggiungere ulteriori funzionalità* a quelle fornite dal canale "base"**
- **basandosi sul concetto di stream**

IL PACKAGE `java.io`

L'approccio *"a livelli"*

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



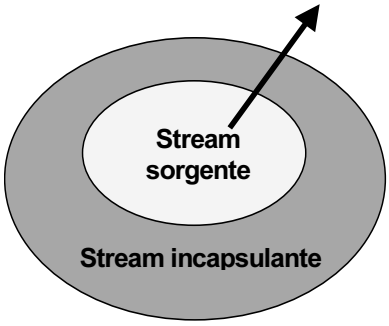
- **gli altri tipi di stream sono pensati per *"avvolgere" i precedenti* per aggiungere ulteriori funzionalità.**

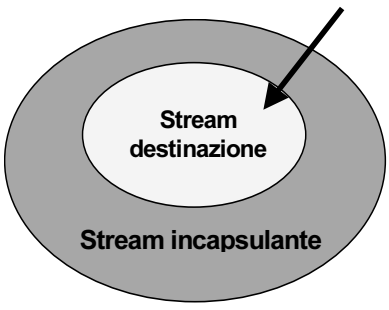
IL PACKAGE `java.io`

L'approccio *"a livelli"*

- **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





IL PACKAGE `java.io`

Il package *`java.io`* distingue fra:

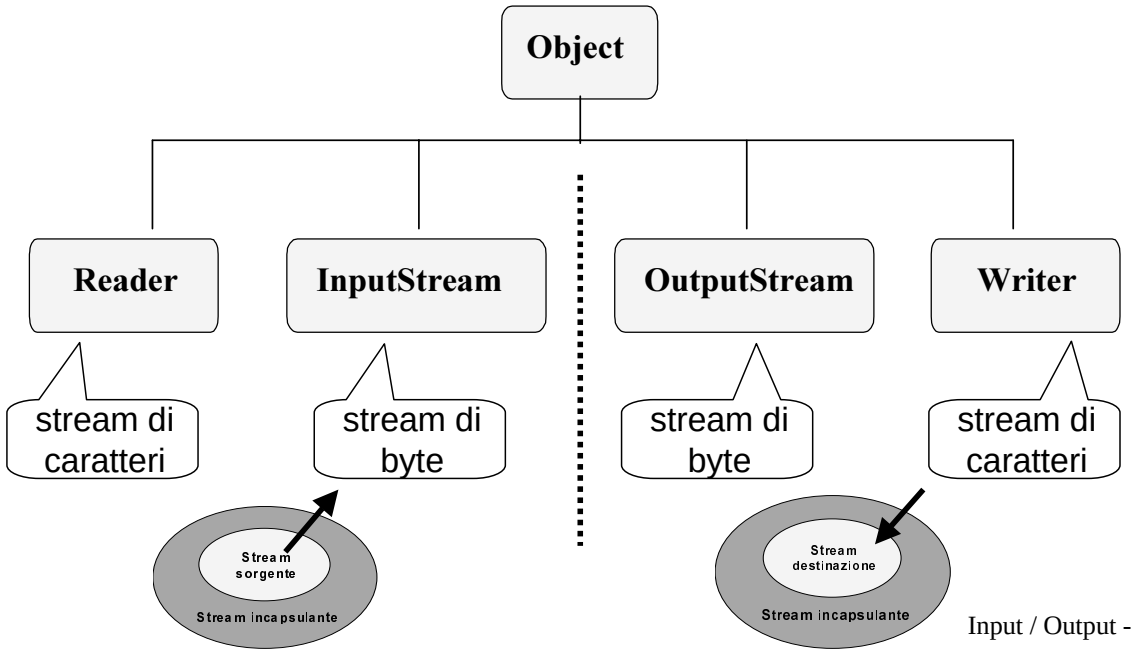
- ***stream di byte*** (analoghi ai *file binari* del C)
- ***stream di caratteri*** (analoghi ai *file di testo* del C) (solo da Java 1.1 in poi)

Questi concetti si traducono in *altrettante classi base astratte:*

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

IL PACKAGE `java.io`

Le quattro classi base astratte 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`**

Molte delle considerazioni che faremo per i primi si traspongono facilmente ai secondi in modo analogo

STREAM DI BYTE

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

- ***il costruttore*** **apre lo stream (vedi open)**
- **`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 adatto alla specifica sorgente dati

STREAM DI BYTE

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

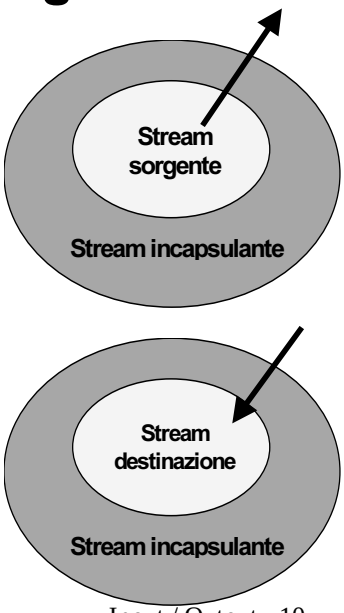
- ***il costruttore* apre lo stream (vedi open)**
- **`write()` scrive uno o più byte**
- **`flush()` svuota il buffer di uscita**
- **`close()` chiude lo stream**

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

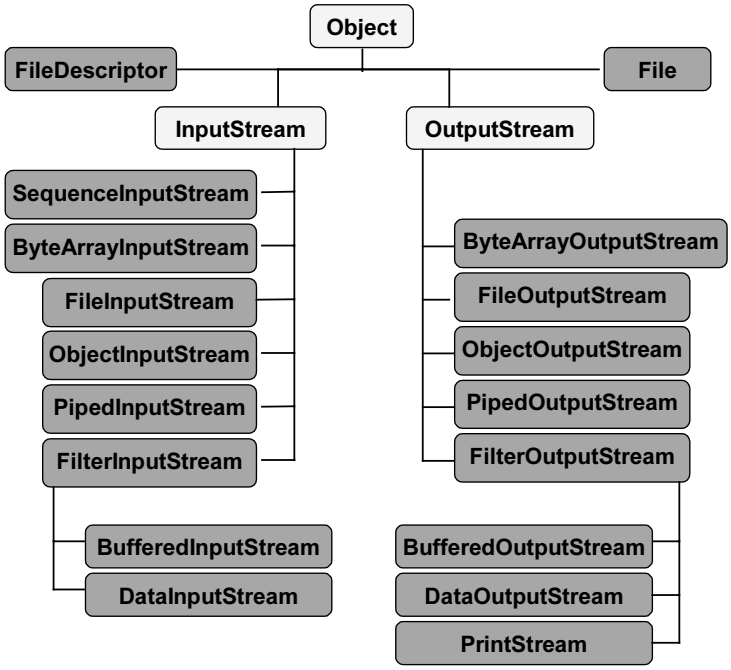
STREAM DI BYTE

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 *nuove funzionalità* a un altro stream**
  - **I/O bufferizzato, filtrato,…**
  - **I/O di numeri, di oggetti,...**



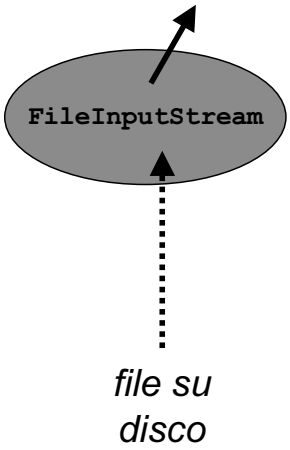
STREAM DI BYTE



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 (apertura invocata in modo implicito)**
- **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 le operazioni su stream possono fallire 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(FileNotFoundException e){
     System.out.println("File non trovato");
     System.exit(1);
   }
   // ... lettura ...
 }
}
```

INPUT DA FILE - ESEMPIO

La fase di lettura:

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

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

SCHEMA per INPUT DA FILE

La lettura viene sempre fatta fino alla fine del file; il ciclo lavora sul carat

```
try {
int carcorrente, int fine
while ((carcorrente = is.read ))!= finefile)
   { System.out.print(" " + carcorrente);
     // lavora sul carattere
    }
    System.out.println("\nFine File");
} catch(IOException ex){
  System.out.println("Errore di input");
  System.exit(2);
}
```

quando lo stream termina, **read()** restituisce indicatore di fine file

INPUT DA FILE - ESEMPIO

Esempio d'uso:

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

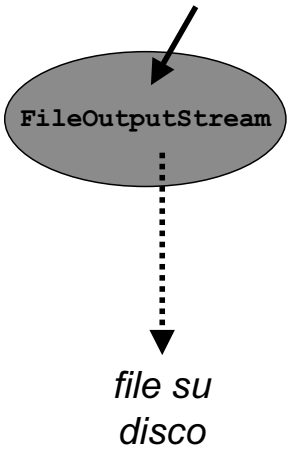
Il risultato:

```
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
```


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**



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`***

OUTPUT SU FILE - ESEMPIO

```
import java.io.*;
public class ScritturaSuFileBinario {
 public static void main(String args[]){
  FileOutputStream os = null;
  try {
    os = new FileOutputStream(args[0]);
  }
  catch(FileNotFoundException e){
    System.out.println("Imposs. apr     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(IOException 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
```

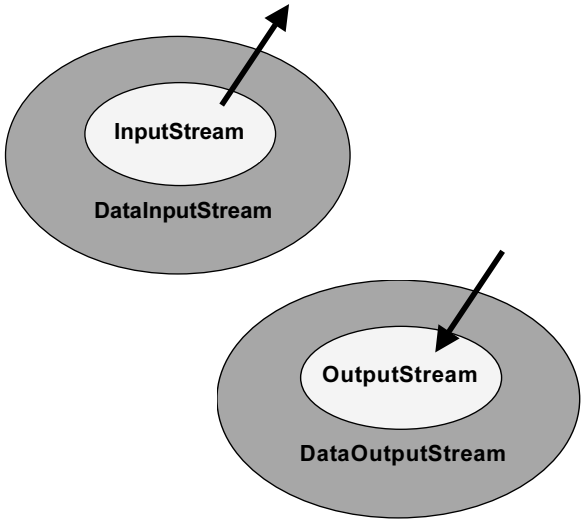
Esperimenti

- ***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` 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: `readInteger()`, `readFloat()`, ...**
- **`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**

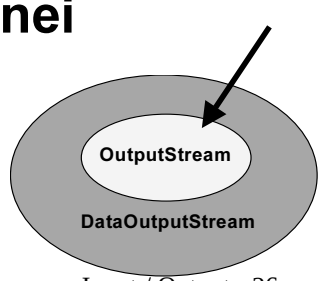
STREAM DI INCAPSULAMENTO - OUTPUT

- `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: `writeInteger()`, ...
- `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

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`



ESEMPIO 1

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

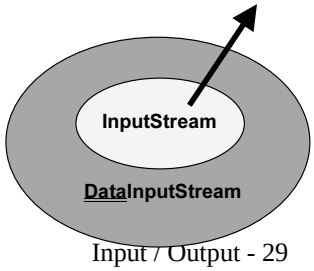

ESEMPIO 1 (segue)

```
  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 (IOException e){
    System.out.println("Scrittura fallita");
    System.exit(2);
  }
 }
}
```

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(FileNotFoundException e){
    System.out.println("File non trovato");
    System.exit(3);
  }
  // continua...
```

ESEMPIO 2 (segue)

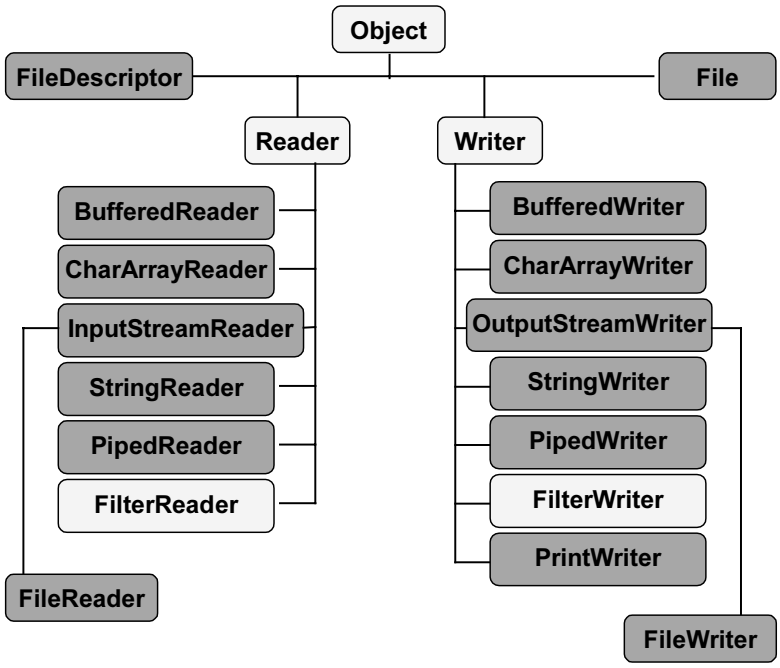
```
DataInputStream is = new DataInputStream(fin)
  float f2; char c2; boolean b2;double d2;
  int i2;
  try {
   f2 = is.readFloat(); b2 = is.readBoolean();
   d2 = is.readDouble(); c2 = is.readChar();
   i2 = is.readInt();    is.close();
   System.out.println(f2 + ", " + b2 + ", "
   + d2 + ", " +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

E se non sappiamo la struttura delle informazioni?

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(FileNotFoundException e){
     System.out.println("File non trovato");
     System.exit(1);
   }
   // ... lettura ...
 }
}
```


INPUT DA FILE - ESEMPIO

La fase di lettura:

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

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

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.

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

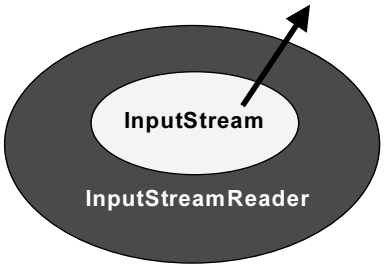
- **Occorre dunque poter *reinterpretare* uno stream di byte come reader / writer *quando opportuno*** *(cioè quando trasmette caratteri)*
- **Esistono due classi "incapsulanti" progettate proprio per questo scopo:**

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

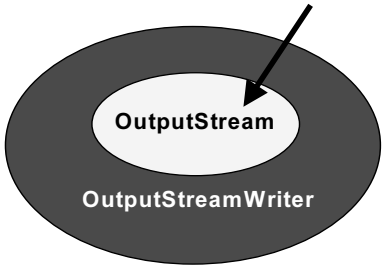
  `OutputStreamWriter`* che *reinterpreta un `OutputStream` come un `Writer`

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`***



da fuori, si vede un ***`Reader`***



da fuori, si vede un ***`Writer`***

IL CASO DELL' I/O DA CONSOLE

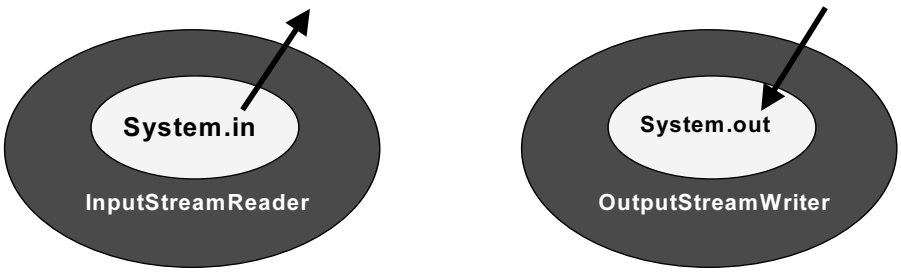
- **Video e tastiera sono rappresentati dai due oggetti statici `System.in` e `System.out`**
- **Poiché esistono fin da Java 1.0** (quando Reader e Writer non esistevano), **essi sono *formalmente* degli *stream di byte*...**
  - **`System.in` è *formalmente* un *`InputStream`***
  - **`System.out` è *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

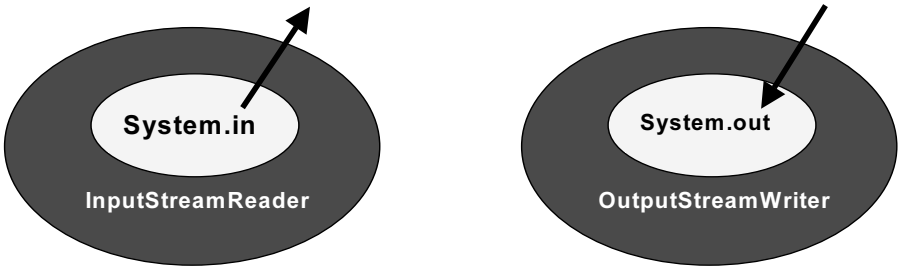
- **`System.in` può essere "interpretato come un `Reader`" incapsulandolo dentro a un *`InputStreamReader`***
- **`System.out` può essere "interpretato come un `Writer`" incapsulandolo dentro a un *`OutputStreamWriter`***



IL CASO DELL' I/O DA CONSOLE

Tipicamente:

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



ESEMPIO 3

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(IOException 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 = new Boolean(b1).toString();
    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());
    fout.close();
  } catch (IOException e){...}
 }
}
```

```
try {
  buffer = Float.toStrin...);
  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.close();...
} catch (IOException e){...}
}
}
```

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

La 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()**

ESEMPIO 3 - note

- **Il nostro esempio ha stampato sul file le *rappresentazioni sotto forma di stringa* di svariati valori...**
- **... *ma non ha inserito spazi intermedi !***
- **Ha perciò scritto:**

```
3.1415true1.4142X12
```

ESEMPIO 4

Rilettura di dati da file di testo

- **Per leggere da un file di testo occorre un `FileReader`, che però consente solo di leggere un *carattere* o una *stringa***
- **Per leggere fino alla fine del file:**
  - **si può usare il metodo `read()` come già fatto...**
  - **...oppure, in alternativa, sfruttare il metodo `ready()`, che restituisce `true` finché ci sono altri caratteri da leggere**

ESEMPIO 4

```
import java.io.*;
public class Esempio4 {
 public static void main(String args[]){
   FileReader fin = null;
   try {
     fin = new FileReader("Prova.txt");
   }
   catch(FileNotFoundException e){
     System.out.println("File non trovato");
     System.exit(3);
   }
   // continua...
```

ESEMPIO 4 (segue)

```
  try {
   while(fin.ready()){
    char ch = (char) fin.read();
    System.out.print(ch);  // echo
   }
   System.out.println("");
  }
  catch (IOException e){
    System.out.println("Errore di input");
    System.exit(4);
  }
}
```


ESEMPIO 3 - UNA VARIANTE

- **La versione precedente ha stampato sul file le *rappresentazioni sotto forma di stringa* di svariati valori, ma non ha inserito spazi intermedi**
- **Come esercizio ulteriore, *aggiungiamo uno spazio fra i valori*, in modo da stampare**
- ***non più*** `3.1415true1.4142X12`
- ***ma*** `3.1415 true 1.4142 X 12`

ESEMPIO 3 - VARIANTE

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


ESEMPIO 4 - UNA VARIANTE

- **La versione precedente ha letto dal file *una unica stringa* ininterrotta**
  - **non poteva far altro, mancando gli spazi**
- ***Ora però gli spazi fra i valori ci sono:*** **possiamo definire una funzione `readField()` che legga una "parola"** (*fino al primo spazio*)
  - **non può essere un metodo, perché esso dovrebbe far parte della classe `FileReader`, che non possiamo modificare**
  - **dev'essere perciò *una funzione statica* (di classe)**

ESEMPIO 4 - `readField()`

```
static public String readField(Reader in){
  StringBuffer buf = new StringBuffer();
  boolean nospace = true;
  try { while(in.ready() && nospace){
            char ch = (char)in.read();
            nospace = (ch!=' ');
            if (nospace) buf.append(ch);
        }
  } catch (IOException e){
    System.out.println("Errore di input");
    System.exit(4);
  }
  return buf.toString();
}
```


L'ESEMPIO 4 RIFORMULATO

```
// continua...
try {
 while(fin.ready()){
  String s = readField(fin);
  System.out.println(s);       // echo
 }
}
catch (IOException e){
   System.out.println("Errore di input");
   System.exit(4);
}
}
```

L'ESEMPIO 4 RIFORMULATO

- In questo modo, siamo in grado di leggere *una stringa alla volta*
- Ogni stringa può quindi essere interpretata *nel modo che le è proprio...*
- ...applicando, se occorre, la conversione opportuna
  - nessuna conversione per le stringhe
  - conversione stringa / int per gli interi
  - conversione stringa / float per i reali
  - ...

`StreamTokenizer`

- La classe `StreamTokenizer` consente di leggere da input *una serie di "token"*
- Può estrarre da uno stream (reader) sia *numeri* sia *stringhe*, in modo configurabile
  - `whitespaceChars(int lo, int hi)` registra come separatori i caratteri da `lo` a `hi`
  - `nextToken()` estrae il token successivo, che viene posto nel campo pubblico `sval` (se è una stringa) o `nval` (se è un numero)
  - il valore restituito `nextToken()` da indica se si tratta di un numero o di una stringa

ESEMPIO 5

Leggere da input una serie di token

- **Avvolgiamo il reader che fornisce l'input con uno `StreamTokenizer`**

```
t = new StreamTokenizer(reader);
```

- **Configuriamo lo `StreamTokenizer` per assumere come separatori ("spazi") tutti i caratteri fra lo `'\0'` e lo spazio `' '`**

```
t.whitespaceChars('\0',' ');
```


ESEMPIO 5

Poi:

- **Predisponiamo un ciclo che ripeta**

```
res = t.nextToken();
```

- **e guardiamo cosa abbiamo letto:**

*Se **`nextToken()`** ha letto*	***`res`** vale la costante*
una stringa	**`StreamTokenizer.TT_WORD`**
un numero	**`StreamTokenizer.TT_NUMBER`**
un fine linea ("End of Line")	**`StreamTokenizer.TT_EOL`**
il fine file ("End of File")	**`StreamTokenizer.TT_EOF`**

ESEMPIO 5

```
import java.io.*;
public class Esempio5 {
  public static void main(String args[]){
   FileReader f = null;
   // ... apertura del file...
   StreamTokenizer t = new StreamTokenizer(f);
   t.whitespaceChars(0, (int)' ');
   int res = -1;
   do {
     try { res = t.nextToken(); }
     catch (IOException e) { ... }
     // ... continua ...
```


ESEMPIO 5

```
     // ... continua ...
     if (res == StreamTokenizer.TT_WORD) {
      String s = new String(t.sval);
      System.out.println("stringa: " + s);
     } else
     if (res == StreamTokenizer.TT_NUMBER) {
      double d = t.nval;
      System.out.println("double: " + d);
     }
   } while( res != StreamTokenizer.TT_EOL &&
            res != StreamTokenizer.TT_EOF);
 }
}
```

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 usi la serializzazione *senza* implementare tale interfaccia)
- **Vengono scritti / letti *tutti* i dati 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 6

Una piccola classe serializzabile...

```
public class Punto2D
 implements java.io.Serializable {
 float x, y;
 public Punto2D(float x, float y) {
     this.x = x; this.y = y; }
 public Punto2D() { x = y = 0; }
 public float ascissa(){ return x; }
 public float ordinata(){ return y;}
}
```


ESEMPIO 6

...e un main che ne fa uso

```
public class ScritturaPointSerializzato {
  public static void main(String[] args) {
   FileOutputStream f = null;
   try {
     f = new FileOutputStream("xy.bin");
   } catch(IOException e){
     System.exit(1);
   }
  // ... continua...
```

ESEMPIO 6

```
  // ... continua ...
   ObjectOutputStream os = null;
   Punto2D p = new Punto2D(3.2F, 1.5F);
   try {
     os = new ObjectOutputStream(f);
     os.writeObject(p);  os.flush();
     os.close();
   } catch (IOException e){
     System.exit(2);
   }
 }
}
```


ESEMPIO 7

Rilettura da stream di oggetti serializzati

```
public class LetturaPointSerializzato{
  public static void main(String[] args) {
   FileInputStream f = null;
   ObjectInputStream is = null;
   try {
     f = new FileInputStream("xy.bin");
     is = new ObjectInputStream(f);
   } catch(IOException e){
     System.exit(1);
   }
  // ... continua...
```

ESEMPIO 7

```
// ... continua ...
 Punto2D p = null;
 try {  p = (Punto2D) is.readObject();
        is.close();
 } catch (IOException e1){
        System.exit(2); }
   catch (ClassNotFoundException e2){
        System.exit(3); }
 System.out.println("x,y = " +
   p.ascissa() + ", " + p.ordinata());
 }
}
```

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