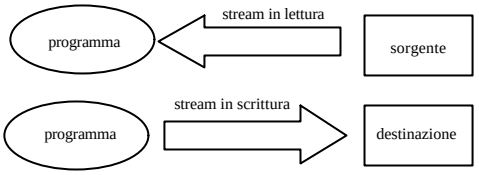
INPUT OUTPUT

a.a. 2001-02

A. Martelli

Le operazioni di I/O avvengono attraverso **stream**.



Sorgente e destinazione possono essere file, memoria, socket, ...

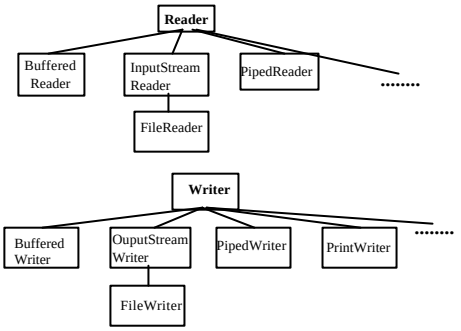
Java fornisce molte classi per I/O.

Divise in due gerarchie
- basate su caratteri (2 byte)
- basate su byte

Le classi possono essere classificate in base allo scopo:

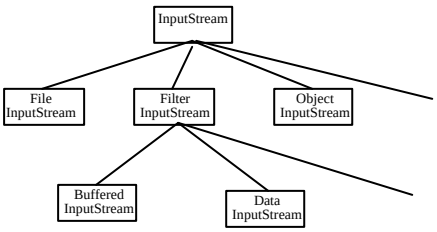
- tipo di sorgente/destinazione
- tipo di elaborazione sui dati

I/O a caratteri: 2 classi astratte alla radice.



Analogamente le classi per gestire input output a byte sono sottoclassi delle due classi astratte:

InputStream **OutputStream**



Sorgente o destinazione possono essere di vario tipo

- memoria (array o stringhe),
- *file*
- *pipe* (per collegare due programmi - thread),
- connessioni via socket o internet.

Input/output su file.

```
FileReader in = new FileReader("esempio.txt");
FileWriter out = new FileWriter("copia.txt");

int c = in.read();
out.write(c);
```

Esiste anche la classe **File**:

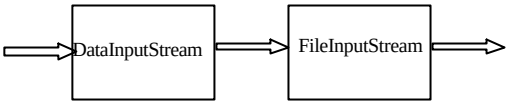
```
File inputFile = new File("esempio.txt");
FileReader in = new FileReader(inputFile);
```


Gli *stream* che specificano il tipo di sorgente, come ad es.**FileInputStream**, non hanno metodi per leggere tipi primitivi (numeri, ...) - leggono solo byte o caratteri.

Altri *stream*, derivati da **FilterInputStream**, possono assemblare i byte in tipi di dati.

Analogamente per l'output.

Gli *stream* possono essere composti:



Alcuni tipi di stream che elaborano dati

Buffering: BufferedReader, ...

usano un buffer per leggere e scrivere
sono più efficienti di quelli senza buffer

Data conversion: DataInputStream, ...

input/output di tipi primitivi

Printing: PrintWriter, PrintStream

I/O di oggetti: ObjectInputStream,.....

Composizione di stream:

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

La classe `InputStreamReader` trasforma uno stream di input che contiene byte in un *reader* che emette caratteri Unicode.

Componendo `BufferedReader` con `InputStreamReader` si ottiene uno stream di input che usa un buffer.

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


I/O di testi

BufferedReader fornisce un metodo **readLine**

```
BufferedReader in=new BufferedReader(
        new FileReader("esempio.txt"));
String s;
while ((s=in.readLine())!=null) .....
```

PrintWriter fornisce i metodi **print** e **println**

```
PrintWriter out=new PrintWriter(
        new FileWriter("esempio.txt"));
out.println(......)
```


Vedere esempio su *Core Java* di lettura e scrittura di array di impiegati in un file di testo.

Per estrarre informazione da una riga di testo può essere utile la classe **StringTokenizer**, che consente di estrarre porzioni di testo (**token**) separate da delimitatori

```
StringTokenizer t = new StringTokenizer(line, |)
```

con il metodo `nextToken` si può ottenere il prossimo token.

C'è anche una classe **StreamTokenizer** che riconosce identificatori, numeri, ...

I/O di tipi di dati primitivi

Le classi **DataInputStream** e **DataOutputStream** forniscono metodi per leggere e scrivere dati primitivi

I dati vengono codificati in formato binario e non sono leggibili come testi.

```
DataOutputStream dout = new DataOutputStream(
      new FileOutputStream("prova.dat"));
dout.writeInt(250);
dout.writeDouble(3.14);
dout.writeChar('a');
dout.close();
DataInputStream din = new DataInputStream(
      new FileInputStream("prova.dat"));
System.out.println(din.readInt());
System.out.println(din.readDouble());
System.out.println(din.readChar());
```


I/O di oggetti

In Java è possibile leggere o scrivere qualunque oggetto con **ObjectInputStream** e **ObjectOutputStream**.

```
ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("impiegati.dat"));
Impiegato rossi = new Impiegato(......);
out.writeObject(rossi);

ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("impiegati.dat"));
Impiegato imp = (Impiegato)in.readObject();
```


E' possibile leggere o scrivere un oggetto solo se questo appartiene ad una classe che implementa l'interfaccia **Serializable**.

```
class Impiegato implements Serializable(...)
```

L'interfaccia **Serializable** non contiene nessun metodo.

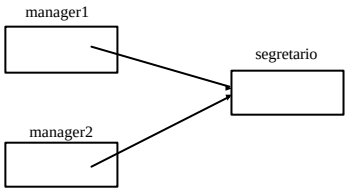
Deve essere specificata solo per motivi di sicurezza.

Quando si legge un oggetto, Java controlla che la definizione della classe non sia cambiata da quando l'oggetto era stato scritto.

Cosa succede se si scrive un oggetto che contiene riferimenti ad altri oggetti?

Vengono scritti anche tutti gli altri oggetti raggiungibili.

Se un oggetto è condiviso, ne viene scritta solo una copia.



NOTA Per fare una copia di un oggetto (*clonarlo*) si può usare la serializzazione: è sufficiente scriverlo e leggerlo.

```
Impiegato rossi = new Impiegato(......);
out.writeObject(rossi);
Impiegato copiaDiRossi = (Impiegato)in.readObject();
```

Esiste anche un metodo **clone** di Object e un'interfaccia **Cloneable**.

Altri tipi di stream disponibili:

- *random access file*
- *ZIP file*

La classe **File** può anche essere usata per gestire directory.