

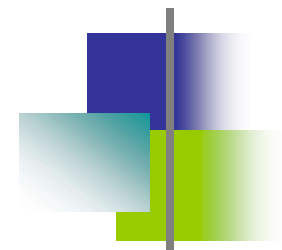


---

# Libreria java.io



**Dott. Romina Fiorenza**  
**[fiorenza.romina@gmail.com](mailto:fiorenza.romina@gmail.com)**



# Il pacchetto java.io

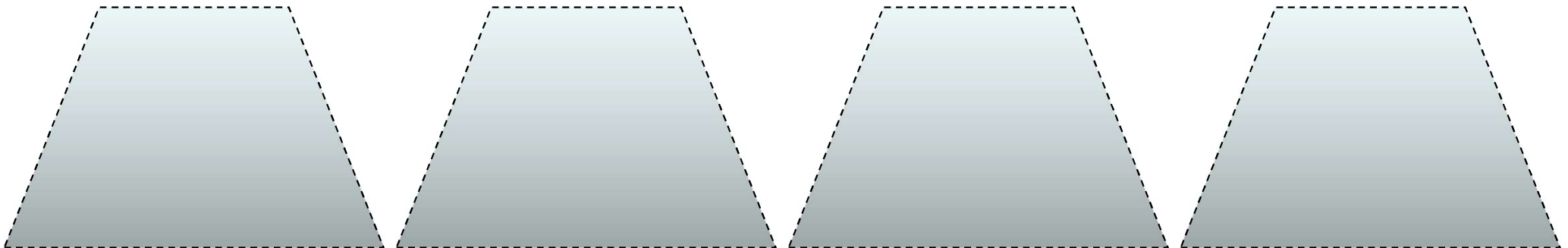
- Flussi: sequenze di byte che viaggiano da una origine a una destinazione lungo un percorso di comunicazione.
- Il pacchetto `java.io` comprende le seguenti principali gerarchie di classi Per gestire le operazioni di input/output
  - Gli input stream sono utilizzati per leggere dati
  - Gli output stream sono utilizzati per scrivere dati

InputStream

OutputStream

Reader

Writer



# Gli stream

- Uno stream può essere visto come un lato di un canale di comunicazione mono-direzionale



- Gli stream offrono un'interfaccia uniforme per le operazioni di I/O, indipendentemente dal tipo e dall'origine dei dati
- Il package `java.io` contiene due gerarchie di classi:
  - Orientate al byte → **Stream**
  - Orientate ai caratteri → **Reader & Writer**
- Gli stream sono FIFO (First-In-First-Out)
- Gli stream sono bloccanti
  - operazioni lettura/scrittura bloccano il thread del processo finchè l'operazione non è conclusa!



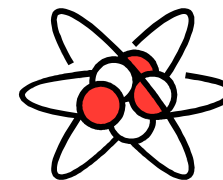
# Definizioni

---

- **Stream di input:** oggetto dal quale si possono leggere una sequenza di bytes
- **Stream di output:** oggetto nel quale è possibile inviare una sequenza di bytes
- Sono stati modellati con le classi astratte  
`InputStream` e `OutputStream`  
che hanno alberi gerarchici simmetrici e che esistono dalla JDK 1.0.

# Unità fondamentale

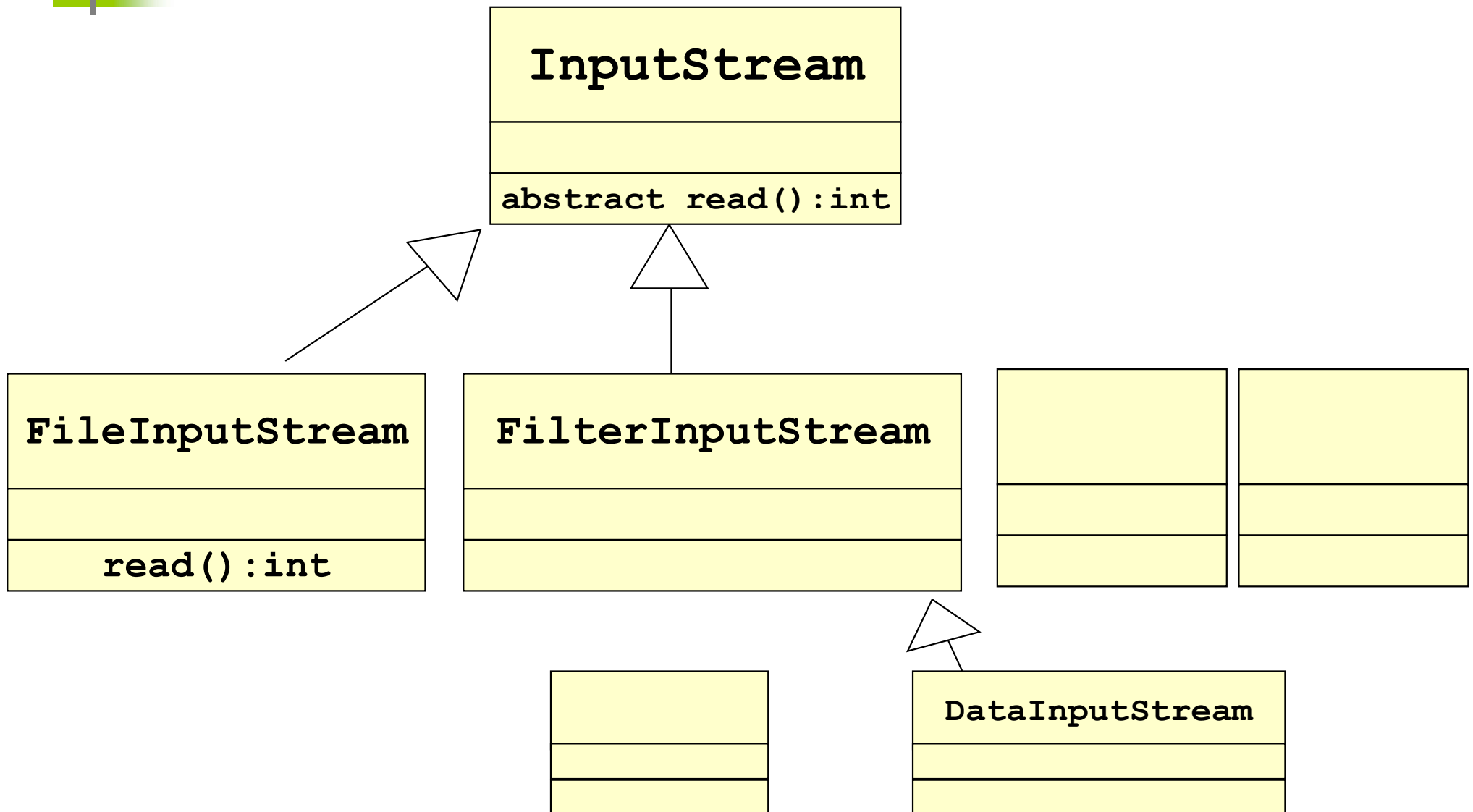
- Gli Stream sono orientati al byte
  - Leggono e scrivono singoli **bytes**.



- Successivamente, nella versione JDK 1.1, sono state create le classi astratte:
  - **Reader**
  - **Writer**

Queste invece sono orientate ai **caratteri** e sono ideali per la lettura/scrittura di caratteri e stringhe

# Gerarchia di InputStream



# InputStream



La classe `InputStream` ha un metodo astratto

```
public abstract int read() throws IOException
```

- legge sequenzialmente byte
- restituisce il byte letto su `int` oppure `-1` (se incontra la fine della sorgente di input)

## **Osservazione**

L'oggetto `System.in` è l'input stream standard.

- E' un oggetto di tipo `InputStream` e consente un carattere alla volta dalla tastiera
- E' un oggetto statico, quindi è sempre disponibile, e non bisogna istanziarlo

# OutputStream



La classe `OutputStream` definisce il metodo astratto

```
public abstract void write(int b) throws  
IOException
```

- invia il byte parametro nello stream

## ***Osservazione:***

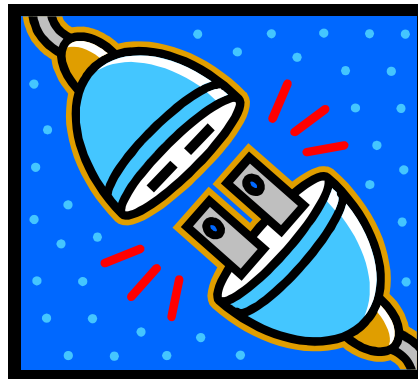
L'oggetto `System.out` è l'output stream standard.

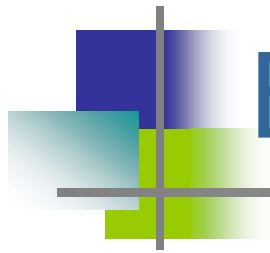
- E' un oggetto di tipo `OutputStream` e consente di visualizzare dati sul dispositivo di default (di solito video)
- E' un oggetto statico, quindi è **sempre** disponibile, e non bisogna istanziarlo



# Chiusura del canale

Conclusa la lettura o scrittura lo stream deve essere chiuso con il metodo `close()` per rilasciare le risorse ad esso dedicate.





# Base e filtrati

---

- Java possiede più di 60 stream diversi
- Sono suddivisi nelle 4 principali gerarchie.
- Esiste poi un'ulteriore suddivisione:
  - Stream **base**,  
istanziabili direttamente
  - Stream **filtrati**,  
che derivano dalle classi  
`FilterInputStream` / `FilterOutputStream`  
che necessitano di uno stream base per essere costruiti e operare (tecnica di “composizione”, detta filtraggio)

# Tecnica del Filtraggio

- Gli **Stream base** indicano solitamente la *sorgente/destinazione* della lettura/scrittura,
- Gli **Stream filtrati** invece specificano il *tipoDati/modalitàTrasporto*.
- **Componendo** le 2 tipologie di Stream si ottengono tutte le possibili ***combinazioni di sorgenti/destinazioni & dati.*** Questa tecnica si chiama **Filtraggio**.





# Stream Base da/per File

**FileInputStream** e **FileOutputStream** permettono di leggere e scrivere dati da file

```
FileInputStream fin = new FileInputStream("prova.txt");
```

oppure

```
File f = new File("prova.txt");
```

```
FileInputStream fin = new FileInputStream(f);
```

...e infine

```
int b = fin.read();
```

Legge in modo sequenziale i byte del file prova.txt

**Nota:** il file **prova.txt** deve essere già esistente per lo Stream di input, ma non per lo Stream di output

# Classe File

- La classe **File** serve a manipolare file e directory in modo indipendente dalla piattaforma.
- Oggetti di tipo **File** rappresentano i nomi dei file e non i file stessi:
  - un oggetto di tipo File può esistere anche se il file che esso rappresenta non esiste affatto!
- Per creare fisicamente un file → costruire un stream di output (come FileOutputStream) con un oggetto di tipo **File** che ne rappresenta il nome.
- La classe **File** può rappresentare sia file che directory
- Quando il file esiste allora è possibile invocare i metodi di File per effettuare operazioni come: *rinominare, cancellare, cambiare permessi, ecc*





# Path dei File

---

- Se il file è posizionato sotto la directory di progetto allora si usa

```
File f = new File("prova.txt");
```

- Se il file è posizionato sotto la directory di pacchetto **mypackage** sotto **src**, allora si usa

```
File f = new File("src/mypackage/prova.txt");
```

- Se il file è posizionato sotto la directory **Users/PC-50** sotto la root, allora si usa

```
File f = new File("/Users/PC-50/prova.txt");
```

oppure

```
File f = new File("C:/Users/PC-50/prova.txt");
```

# DataInputStream



- E' uno Stream filtrato
- Permette di leggere primitivi da una sorgente
- Dobbiamo comporlo con Stream di tipo sorgente

## Esempio:

- Componendo con FileInputStream possiamo leggere dati primitivi da file

```
FileInputStream fin = new FileInputStream("prova.dat");  
DataInputStream din = new DataInputStream(fin);  
double s = din.readDouble();
```



# Lettura e Scrittura primitivi

- Metodi a confronto:

	<b>DataOutputStream</b>	<b>DataInputStream</b>
una riga		<code>readLine</code>
stringa	<code>writeChars</code>	
<b>int</b>	<b><code>writeInt</code></b>	<b><code>readInt</code></b>
<b>short</b>	<b><code>writeShort</code></b>	<b><code>readShort</code></b>
<b>long</b>	<b><code>writeLong</code></b>	<b><code>readLong</code></b>
<b>float</b>	<b><code>writeFloat</code></b>	<b><code>readFloat</code></b>
<b>double</b>	<b><code>writeDouble</code></b>	<b><code>readDouble</code></b>
<b>boolean</b>	<b><code>writeBoolean</code></b>	<b><code>readBoolean</code></b>
char	<code>writeChar</code>	<code>readChar</code>
testo Unicode	<code>writeUTF</code>	<code>readUTF</code>



# BufferedInputStream



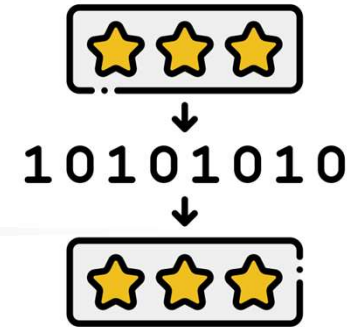
- E' uno Stream filtrato
- Serve per migliorare le prestazioni, in quanto inserisce e gestisce un buffer nello stream
- Maggiormente usato è il BufferedOutputStream → è chi scrive decide come usare il buffer

## *Esempio:*

- Lo componiamo con FileInputStream e leggiamo primitivi da file componendo ulteriormente con DataInputStream

```
FileInputStream fis = new FileInputStream("prova.dat");  
BufferedInputStream bis = new BufferedInputStream(fis);  
DataInputStream dis = new DataInputStream(bis);  
  
double d = dis.readDouble();
```

# Stream per gli Oggetti



- Sono Stream filtrati
- Consentono di scrivere e leggere lo stato di un oggetto.
- Le classi sono `ObjectOutputStream` e `ObjectInputStream`

## *Esempio:*

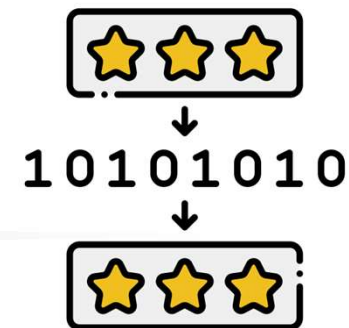
- Per scrivere, filtriamo `ObjectOutputStream` con `FileOutputStream`



```
FileOutputStream fos = new FileOutputStream("Oggetti.txt");  
ObjectOutputStream out = new ObjectOutputStream(fos);  
Impiegato imp = new Impiegato("Harry Hacker", 35000,  
                                new Date());  
out.writeObject(imp);
```

- Per leggere, filtriamo `ObjectInputStream` con `FileInputStream`

```
FileInputStream fis = new FileInputStream("Oggetti.txt");  
ObjectInputStream in = new ObjectInputStream(fis);  
Impiegato imp = (Impiegato)in.readObject();
```

# Serializzazione



- Gli ObjectOutputStream trasportano solo istanze di classi che implementano l'interfaccia **Serializable**
- Serializable è un'interfaccia di markup → Non include metodi
- Una classe **Serializable**, deve avere attributi primitivi o serializzabili, se non lo fossero verrà generata a runtime una **java.io.NotSerializableException**
  - **non** ci sono mai errori in compilazione.
- Un oggetto è serializzabile se si può convertire in una sequenza di byte, in altre parole se si può ricondurre ad un tipo primitivo:  
 *“Smontare e rimontare in pezzi primitivi”* 
- Questo meccanismo funziona anche in ambiente di rete → serializzazione compensa differenze fra S.O.



# Esempio

---

```
class Impiegato implements Serializable{  
    private String nome;  
    private MyDate nascita;  
}
```

```
class MyDate {  
    private int giorno;  
    private String mese;  
    private int anno;  
}
```

La classe Impiegato è **Serializable**, ma l'attributo nascita non lo è!  
Il compilatore NON segnala errore, ma se si prova a serializzare  
un Impiegato si avrà **NotSerializableException** per **MyDate**



# Attributi transienti

---

- Una variabile d'istanza marcata **transient** non parteciperà alla serializzazione → non è significativa per lo stato dell'oggetto
- Questo significa che
  - quando l'oggetto viene serializzato, il valore **non** viene scritto/inviato
  - quando l'oggetto viene deserializzato , il valore è **posto al default**

# Stream per le stringhe

- **Reader** e **Writer** manipolano caratteri e sono stream filtrati → devono, in generale, essere composti con stream di base.
- Realizzano le conversioni del testo in modo trasparente, indipendentemente dalla piattaforma
- **Reader** e **Writer** sono classi astratte e una implementazione di base è data
  - **InputStreamReader**
  - **OutputStreamWriter**usate per convertire da i flussi di **byte** in flussi di **caratteri** e viceversa





# Adattatori di Stream

---

- **InputStreamReader** e **OutputStreamWriter** sono anche detti “**adattatori**” perché vengono usati per convertire Stream → Reader e Stream → Writer
- **InputStreamReader**
  - converte da stream di **byte** a stream di **caratteri**
  - legge byte e li trasforma in caratteri (secondo codifica specificata)
  - **NB**: si usa spesso abbinato al **BufferedReader**
- **OutputStreamWriter**
  - converte da stream di **caratteri** a stream di **byte**
  - legge caratteri (secondo codifica specificata) e li trasforma in byte
  - **NB**: non si usa spesso, poiché esiste il **PrintWriter** che costituisce un flusso bufferizzato già pronto

La codifica di default è la Unicode a 16 bits.



# Stream bufferizzati

---

- **BufferedReader/BufferedWriter** si usano per leggere e scrivere caratteri e usano un **buffer** di appoggio per ottenere una comunicazione più performante
- Sono simili ai relativi oggetti orientati ai byte
- Si utilizzano attraverso la tecnica del filtraggio
  - Componendoli con altri stream per caratteri





# Stream verso i file

---

- **FileReader** e **FileWriter** sono classi che mascherano la “decorazione” degli stream → si possono usare direttamente per leggere e scrivere da file di testo
- Sono stream **bufferizzati**, non necessitano di essere filtrati con stream bufferizzati, anche se spesso si fa lo stesso per disporre di metodi più “comodi”
  - es. `BufferedReader.readLine()`
- **NB**: Il buffer è impostato per default con **flush automatico**
  - dopo le scritture NON si deve invocare `flush()`
- Sono stream **orientati ai caratteri**, quindi **non** è possibile leggere dati diversi (per es. byte, primitivi, ecc...)
  - per queste situazioni si usa **FileInputStream**



# BufferedReader/BufferedWriter

---

- Lettura nel file

```
File file = new File("myDir/prova.txt");
try {
    FileReader fr = new FileReader(file);
    BufferedReader br = new BufferedReader(fr);
    String data;
    while((data=br.readLine())!=null)
        System.out.println(data);
    br.close();
} catch (Exception e) {}
```

- Scrittura dal file

```
File file = new File("myDir/prova.txt");
try {
    FileWriter fr = new FileWriter(file);
    BufferedWriter bw = new BufferedWriter(fr);
    bw.write("ciao\n");
    bw.write("come stai?\n");
    bw.close();
} catch (Exception e) {}
```

**Il BufferedWriter fa il flush  
quando chiude.  
Quindi evito le singole  
chiamate a flush()**

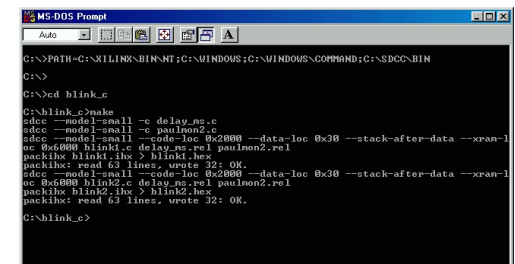
# Standard IN/OUT

- Lettura da tastiera: si utilizza **System.in**
- **System.in** è un **InputStream** orientato al byte
  - è agganciato alla tastiera
  - consente la lettura di un solo carattere per volta



```
InputStreamReader isr = new InputStreamReader(System.in);  
BufferedReader buff = new BufferedReader(isr);  
String l = buff.readLine();
```

- Stampa a video: si utilizza **System.out**
- **System.out** è un **OutputStream** di base
  - è agganciato al video
  - consente la scrittura/stampa di primitivi, String, oggetti

A screenshot of an MS-DOS command prompt window. The title bar says "MS-DOS Prompt". The command history shows:   
C:\>PATH=C:\XILINK\BIN;MT:C:\WINDOWS;C:\WINDOWS\COMMAND;C:\SDCC\BIN  
C:\>  
C:\>cd blink\_c  
C:\>blink\_c>make  
sdcc -mmodel=small -c delay\_ms.c  
sdcc -mmodel=small -c paulnon2.c  
sdcc -mmodel=small -c delay\_ms.c -data-loc 0x2000 -stack-after-data --xran-1  
oc 0x6000 blink1.c delay\_ms.rel paulnon2.rel  
packihx blink1.hex > blink1.hex  
packihx: read 63 lines, wrote 32: OK.  
sdcc -mmodel=small -c delay\_ms.c -data-loc 0x2000 -stack-after-data --xran-1  
oc 0x6000 blink2.c delay\_ms.rel paulnon2.rel  
packihx blink2.c > blink2.hex  
packihx: read 63 lines, wrote 32: OK.  
C:\>blink\_c>



# Lettura stringhe

---

Per **leggere** una stringa da una sorgente, bisogna:

1. ottenere/creare lo stream base agganciato alla sorgente
2. filtrarlo con **InputStreamReader** (ed eventualmente bufferizzarlo con **BufferedReader**)

**Esempio:** leggo una stringa da file

```
InputStream stream = new FileInputStream("nomeFile.txt");  
InputStreamReader isr = new InputStreamReader(stream);  
BufferedReader buff = new BufferedReader(isr);  
String l = buff.readLine();
```

**Osservazioni:**

- Per leggere da file potevamo usare *semplicemente* un **FileReader** filtrandolo con un **BufferedReader**



# Scrittura stringhe

Per **scrivere** una stringa verso una destinazione, bisogna:

- ottenere/creare lo stream base agganciato alla destinazione
- filtrarlo con **PrintWriter** (stream già bufferizzato)

Esempio: scrittura su file

```
OutputStream stream = new FileOutputStream("nomeFile.txt");  
PrintWriter pw = new PrintWriter(stream, true);  
pw.println("Salve!");
```

**Note:** Il parametro booleano serve ad indicare il comportamento del buffer:

- **true** = auto-flush.

Ogni dato inviato è subito disponibile al destinatario

- **false** = flush manuale.

I dati sono nel buffer e non sono disponibili al destinatario fino a quando non viene effettuata l'operazione di flush (metodo `flush()` sull'oggetto **PrintWriter**).

**NB:** il costruttore `PrintWriter(OutputStream stream)`  
usa il boolean a **false**

# Lo Scanner

- Un oggetto di tipo `java.util.Scanner` (disponibile da JDK 1.5) non è uno stream, ma permette di scorrere un testo ed estrarre tipi primitivi e stringhe
- Lo Scanner spezza l'input in token usando un delimitatore: per default è lo spazio bianco.
- E' possibile usare uno Scanner per leggere da file di testo con varie modalità:
  - Singole parole
  - Singole righe
  - Intero file



# Scanner su String



Esempio: Vogliamo contare il numero di parole che compone la stringa assegnata

```
String s = "Guarda:oggi è una bella giornata! Meno male...";
Scanner sc = new Scanner(s);
sc.useDelimiter("[ .:!]");
int k = 0;
while(sc.hasNext()){
    String ss = sc.next();
    k++;
    System.out.println(k+ " parola: " +ss);
}
```

- Creo lo scanner da stringa
- Imposto i delimitatori con [xxx]+

- Ciclo sui token
- Recupero il token

- Visualizzo il token

Il metodo `useDelimiter()` consente di inserire un'espressione regolare come pattern di delimitatori

# Scanner su File



Esempio: Vogliamo contare il numero di parole che compongono il file di testo assegnato

```
FileReader file = new FileReader("nomeFile.txt");
Scanner sc = new Scanner(file);
sc.useDelimiter("[ .,:;!?\\" () - ]+");
int k = 0;
while(sc.hasNext()){
    String ss = sc.next();
    k++;
    System.out.println(k+ " parola: " +ss);
}
```

- Creo lo scanner da file
- Imposto i delimitatori con [xxx]+

- Ciclo sui token
- Recupero il token

- Visualizzo il token

Note: i doppi apici devono usare la sequenza di escape → \"

Il trattino (-) serve per definire un range di valori (es. [A-Z] sono le lettere da A a Z) , allora si può mettere nella regex solo all'inizio o alla fine della sequenza





# Scanner su tastiera



- E' possibile leggere da tastiera con un oggetto **Scanner** agganciato a **System.in**:
- Interi e primitivi si leggono con metodi nextXXX().

Esempio:

```
Scanner sc = new Scanner(System.in);  
int i = sc.nextInt();
```

Nota: il metodo solleva un'unchecked Exception `java.util.InputMismatchException` se i tipi letti non corrispondono al formato o al range

- Le stringhe si leggono invece con:

```
Scanner sc = new Scanner(System.in);  
String line = sc.nextLine();
```



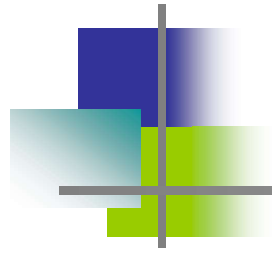
# Esercizi

---

- 1. Copia/Incolla di un file (esempio con .jpg)
- 2. Scrivere un double su un file VUOTO e rileggerlo
- 3. Scrivere oggetti Impiegato sul disco (in un file) e rileggerli
- 4. Stampare il contenuto di un file di testo
- 5. Contare il numero di parole diverse di un file di testo

## AVANZATI:

- 6. Contare il numero di occorrenze di tutte le parole di un file di testo
- 7. Simulare il funzionamento di una tabella di Impiegati con chiave primaria data dalla matricola e gestire semplici operazioni di scrittura e lettura



# Appendice

---

---



# Formattare la stampa

---

- Gli oggetti che implementano un qualche tipo di formattazione sono quelli di tipo **PrintWriter** (per stream di caratteri) o **PrintStream** (per stream di byte).
- `System.out` e `System.err` → **PrintStream**
- Per stream formattato di caratteri → **PrintWriter**
- Gli oggetti di tipo `PrintStream` e `PrintWriter` implementano
  - metodi **write** basilari
  - un insieme di metodi per convertire i dati in output formattato:
    - **print** e **println** formattano singoli argomenti in modo standard
    - **format** formatta argomenti multipli tramite stringa di formato con varie opzioni



# Specificatori

---

Tutti gli specificatori di formato obbligatoriamente:

- *iniziano con %*
- *terminano con il fattore di conversione (1-2 caratteri).*

Dettaglio:

- **d** formatta un intero
- **f** formatta numero in virgola mobile
- **x** formatta un intero come valore esadecimale
- **s** formatta qualsiasi valore come stringa

*Altri elementi (precisione, larghezza,ecc) sono opzionali*



# Esempi

---

```
// stampa il double formattato
```

```
double num = 5.33362;
```

```
System.out.format("%f", num);           // cioè 5,333620
```

```
// stampa l'intero in esadecimale
```

```
int num = 20;
```

```
System.out.format("%x", num);           // cioè 14
```