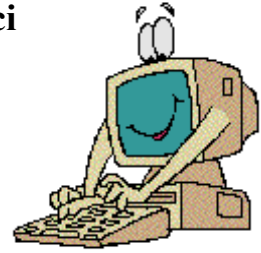
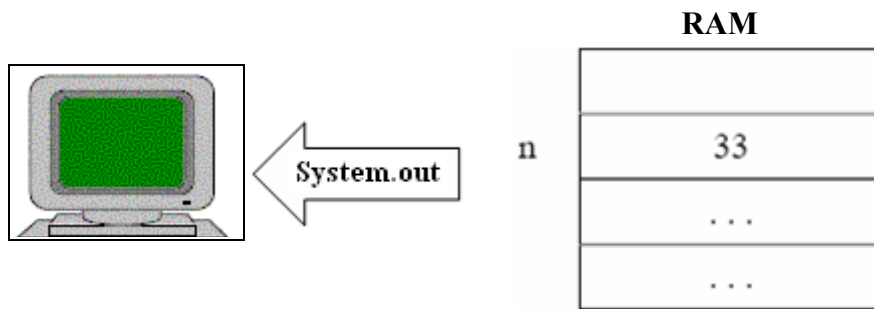


Operazioni di scrittura e lettura con periferici in linguaggio Java



Flussi logici (*stream*) di output

Operazione di scrittura a monitor con uso di *stream* (flusso logico) di output



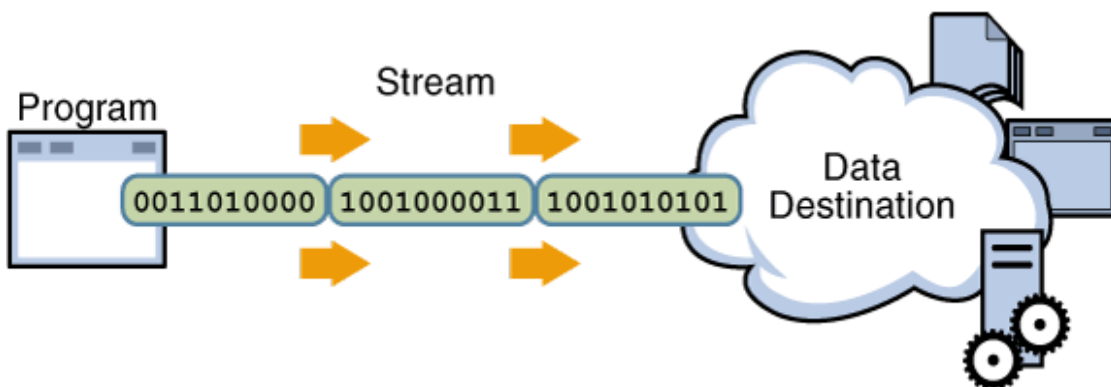
```
System.out.println (n);    /* scrive a monitor come stringa il valore 33  
                           e ritorna a capo */
```

```
System.out.print (n);     /* scrive a monitor come stringa il il valore 33  
                           senza andare a capo */
```

out è un **oggetto** della classe `System`, di tipo `PrintStream` cioè **canale** bufferizzato, adatto per inviare **sequenze di caratteri** al monitor: possiede infatti due metodi (*servizi* che possiamo richiedere):

```
println ("stringa")    // scrive a monitor la stringa ed inserisce un ritorno a capo  
print ("stringa")     // scrive a monitor la stringa senza andare a capo
```

è un canale più sofisticato del solo *stream* adatto per inviare un *flusso* (sequenze) di *byte*:

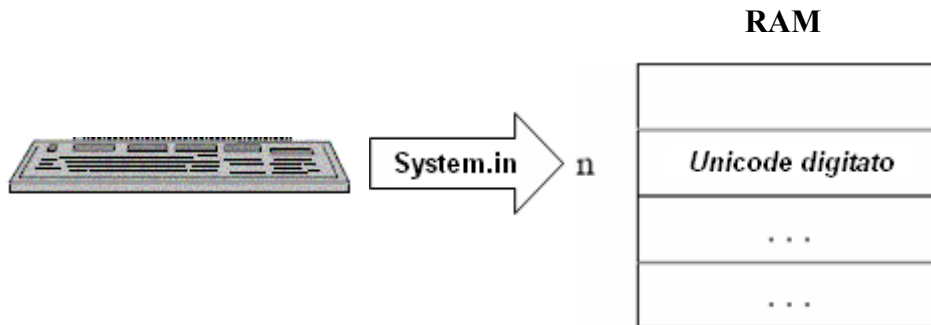


<http://java.sun.com/docs/books/tutorial/essential/io/index.html>

nb: dalla versione *Tiger* è stato introdotto il metodo `printf` con possibile formattazione.

Flussi logici (*stream*) di input ed eccezioni

Operazione di lettura da tastiera con uso di *stream* (flusso logico) di input

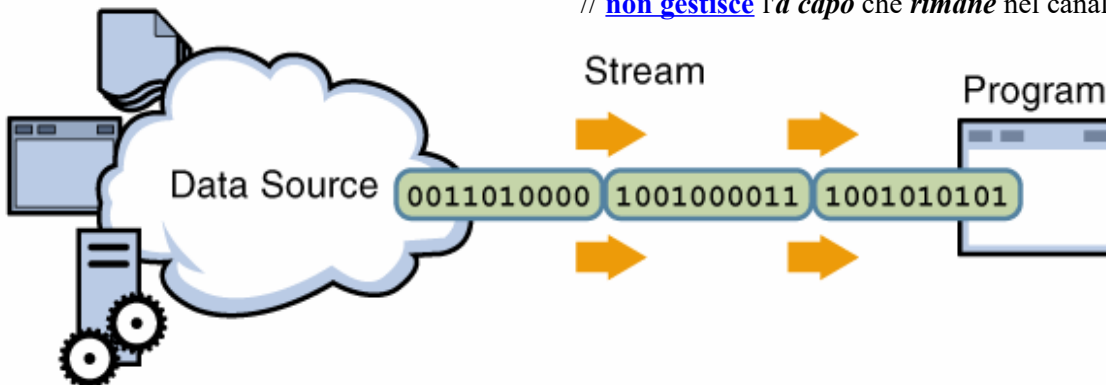


`in` è un **oggetto** della classe `System`, di tipo [InputStream](#) cioè un **canale** non adatto per leggere sequenze di caratteri da tastiera ma solo **singoli byte**: possiede infatti solo un metodo (*servizio* che possiamo richiedere):

```
int read()
```

che legge il successivo **byte**, lo converte in un carattere e ne restituisce il valore sotto forma di un intero maggiore o uguale a zero in codice [UNICODE](#)

```
int n = System.in.read(); // lettura di unico carattere da tastiera
                          // salva nella variabile n il codice intero UNICODE
                          // non gestisce l'a capo che rimane nel canale
```



Ogni lettura può generare un evento inusuale (*eccezione*) ed, in Java, è necessario prevederne la **cattura** e gestione: si pone il codice che può generare *eccezioni* nel costrutto **try-catch** cioè, a livello di compilazione, si **prova ad eseguire** e nel caso vengano generate *eccezioni*, si **cattura** l'eccezione di un determinato tipo (`Exception` nel caso generale) eseguendo in conseguenza un opportuno segmento di programma:

```
try {
    System.out.println(System.in.read());
} catch (Exception e) { }

try {
    // operazione critica che può sollevare eccez.
} catch (Exception e) {
    // gestione dell'eccezione
}
```

In alternativa si deve segnalare **quali eccezioni può generare** un metodo, non gestite al suo interno, e **rilanciarle** con la clausola **throws**.

nb: per realizzare una lettura con cattura di eccezioni di tipo `IOException` è necessario importare la classe contenuta nel package `java.io` con istruzione **import java.io.IOException;**

Operazione di lettura da tastiera di sequenze di caratteri

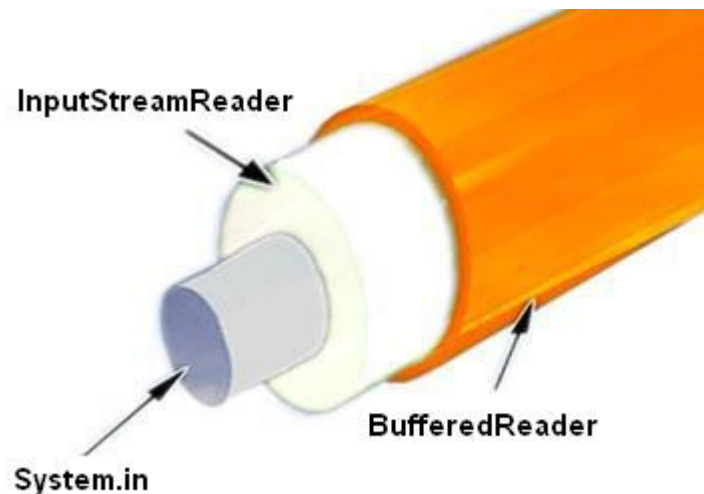
Per realizzare una lettura di una sequenza di caratteri con corretta gestione dei controlli come il *fine riga*, è necessario specializzare sempre più il canale, “*componendo*” col **metodo dell'incapsulamento** l'oggetto più evoluto che ci serve (dotato delle funzioni volute):

- innanzi tutto convertire il flusso standard di ingresso **System.in** (di norma associato alla tastiera, nato storicamente per leggere un flusso di byte) creando un oggetto di tipo *lettore* capace di leggere un *flusso di caratteri*
- poi convertire tale lettore **creando** un *canale bufferizzato* capace di gestire correttamente caratteri di controllo come l'*a capo* grazie al metodo

readLine()

che ritorna come oggetto String
la sequenza di caratteri letti

```
String s =  
oggettoBufferedReader.readLine();
```



Si ricorre allora a classi contenute nel package `java.io` che si importerà nell'applicazione con istruzione **`import java.io.*;`**

In particolare si useranno i metodi *costruttori* di tali classi (che hanno lo stesso nome della classe) per **creare** gli oggetti necessari con la seguente sintassi:

```
NomeClasse nomeOggetto = new NomeClasse (parametro);
```

- il costruttore **InputStreamReader** (*nome_stream_tipo_InputStream*) serve per convertire un flusso di ingresso in un *lettore* o *reader*. Viene normalmente utilizzato per convertire il flusso di ingresso standard **System.in** in un *lettore* cioè un **flusso di caratteri** di ingresso

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

- il costruttore **BufferedReader** (*nome_lettore_tipo_InputStreamReader*) per letture con salvataggio temporaneo in area *buffer* gestita opportunamente

```
BufferedReader stdin = new BufferedReader (lettore);
```

nb: dalla versione *Tiger* è stata introdotta la classe **Scanner**



// INPUT da tastiera : legge una stringa di caratteri *catturando eccezioni* legate ad I/O di dati

```
import java.io.*;
```

```
class Input {
```

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

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

```
        BufferedReader stdin = new BufferedReader(lettore);
```

```
        System.out.print("Digita una riga di testo : ");
```

```
        try { // try (prova) ad eseguire ...
```

```
            String s = stdin.readLine();
```

```
            System.out.println(s);
```

```
        } catch (IOException e) { // e ... catch (cattura) eccezioni
```

```
            // gestione di eventi inusuali
```

```
        }
```

```
    }
```

```
}
```

Effetto: Digita una riga di testo : posso leggere frasi
 posso leggere frasi

nb: si può usare la seguente *sintassi più compatta* per la creazione del canale opportuno con metodo dell'incapsulamento:

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

Alternativa con uso della clausola *throws*

// legge da tastiera una stringa di caratteri *rilanciando eccezioni* legate ad I/O di dati

```
import java.io.*;
```

```
class Leggi{
```

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

```
        throws IOException{ // segnala quali eccezioni può generare tale metodo
```

```
        // non gestite al suo interno e rilanciate
```

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

```
        System.out.print("Digita una riga di testo : ");
```

```
        String stringa = stdin.readLine();
```

```
        System.out.println(stringa);
```

```
    }
```

```
}
```

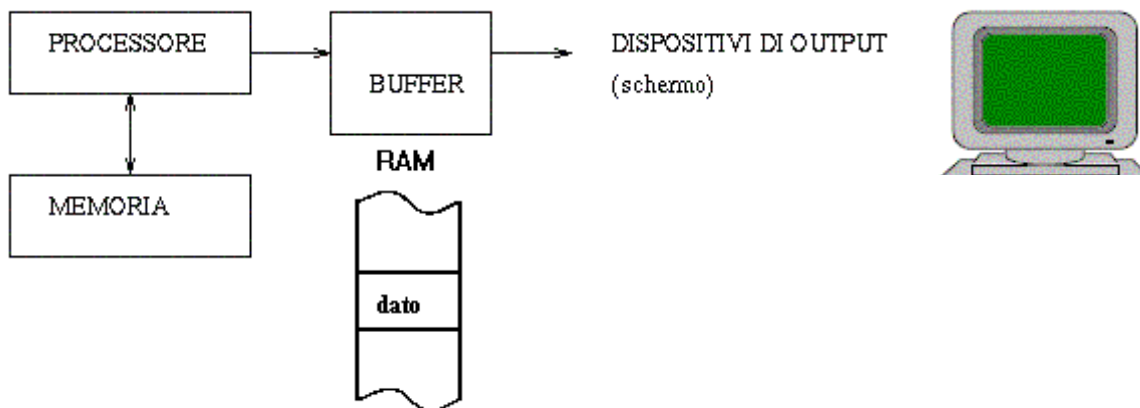
Effetto: Digita una riga di testo : posso leggere frasi
 posso leggere frasi

Canali bufferizzati

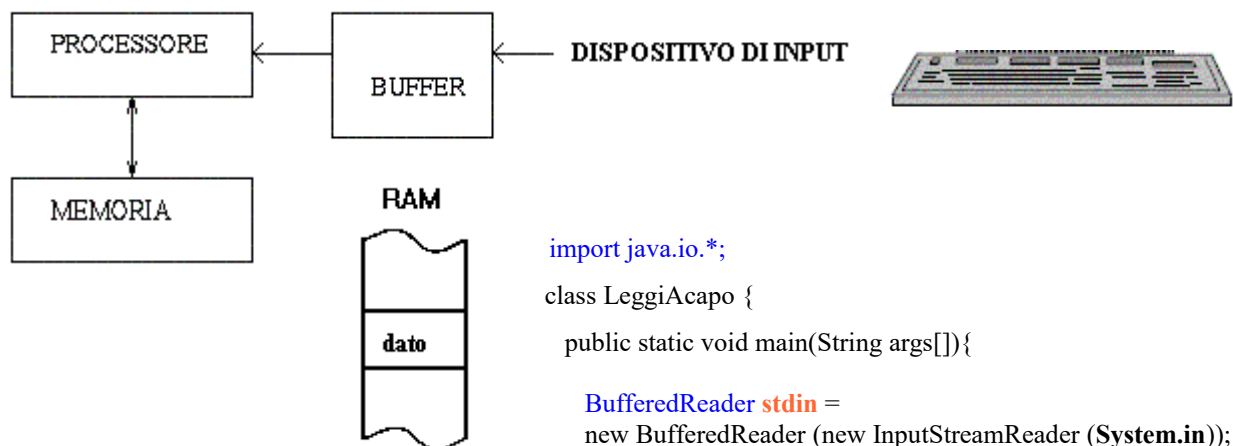


Se i canali (flussi logici o stream) - che permettono di scambiare dati con dispositivi esterni (periferici quali tastiera o monitor) o file (memorizzati su supporti magnetici o magneto-ottici) - vengono implementati con dei buffer (aree di Ram) le letture e/o scritture non avvengono direttamente infatti tali buffer hanno la funzione di **memorizzare temporaneamente** (come su un foglietto di appunti) i dati risolvendo problemi di *sincronizzazione* tra l'elevata velocità di esecuzione della CPU e la relativa lentezza degli apparati su cui scrivere o dai quali leggere.

Ad esempio, un' **operazione di scrittura a monitor** con uso di **flusso logico di output di tipo PrintStream**, corrisponde a salvare nel **buffer di scrittura** il dato; il buffer viene *svuotato* e avviene l'effettiva scrittura *solamente in certe condizioni*.



Analogamente, un' **operazione di lettura da tastiera** con uso di **flusso logico di input di tipo BufferedReader**, corrisponde a salvare nel **buffer di lettura** il dato; il buffer viene *svuotato* e avviene l'effettiva scrittura *solamente in certe condizioni*.



```
import java.io.*;
class LeggiAcapo {
    public static void main(String args[]){

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

        System.out.print("Digita un carattere e premi ENTER : ");
        try {
            System.out.println(System.in.read());
            stdin.readLine(); // per gestire "a capo"
        } catch (IOException e) { // gestione di eventi inusuali }
        }
    } // fine class
```

Leggere l'input con la classe Scanner

La classe Scanner

Sfortunatamente, la classe **InputStream** non possiede metodi comodi per la ricezione di **dati numerici e stringhe**

Per ovviare a questo inconveniente, **Java 5.0** ha introdotto la classe [Scanner](#) nel package **util**.

Un oggetto di **tipo Scanner** consente di leggere da qualsiasi flusso di ingresso (ad es. un file)

Lo useremo per leggere dati in ingresso da tastiera ricevuti tramite l'oggetto **System.in** (di tipo **InputStream** adatto per la lettura di *flussi di byte*)

Il metodo `nextInt` di Scanner

Prima di tutto si deve importare la classe **Scanner** all'interno del file `.java` che ne fa uso.

```
import java.util.Scanner;
```

All'interno del codice si deve creare un nuovo **oggetto** della classe **Scanner**

```
Scanner in = new Scanner(System.in);
```

Poi si possono invocare i metodi. Ad esempio per leggere un intero:

```
int number = in.nextInt();
```

Durante l'esecuzione del metodo `nextInt` il programma si ferma ed attende l'introduzione dell'input da tastiera, che termina quando l'utente batte il tasto **Invio**

Il metodo `nextInt` restituisce un valore numerico di tipo **int**

Cosa succede se l'utente non digita un numero intero sulla tastiera (ad esempio, se scrive **55KZ**) ?

Effetto senza cattura delle eccezioni:

```
Dammi un numero intero: 55KZ
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:819)
    at java.util.Scanner.next(Scanner.java:1431)
    at java.util.Scanner.nextInt(Scanner.java:2040)
    at java.util.Scanner.nextInt(Scanner.java:2000)
    at LeggiScanner.main(LeggiScanner.java:13)
```

Effetto con cattura delle eccezioni:

```
int intero = 0;
try { intero= s.nextInt(); } catch (Exception e) {
    System.out.println("Errore: "+ e +" in input");
    System.exit(0);
}
```

```
Dammi un numero intero: 55KZ
Errore: java.util.InputMismatchException in input
```

Altri metodi di Scanner

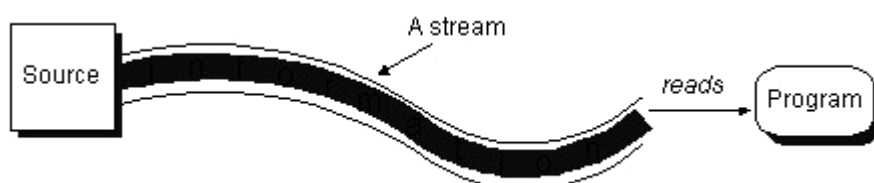
ATTENZIONE: bisogna sempre importare la classe **Scanner** e creare un oggetto di tipo **Scanner**

```
import java.util.Scanner;
...
Scanner in = new Scanner(System.in);
```

- Leggere un **numero in virgola mobile** con **separatore virgola**
`double price = in.nextDouble();`
- Leggere una intera **riga** fino alla pressione di **Enter**
`String city = in.nextLine();`
- Leggere una **parola** fino al primo **carattere di spaziatura**: spazio, fine riga, tabulazione
`String state = in.next();`

Scanner a confronto con BufferedReader: uso del tasto ENTER nella chiusura dell'applicazione

<pre>import java.util.Scanner; public class ScannerFine { /** * Legge da tastiera Java ver.5 * senza obbligatorietà della cattura/gestione eccezioni */ public static void main(String[] args) { Scanner s= new Scanner (System.in); System.out.print("\n\nPremi ENTER per terminare .. "); s.nextLine(); // lettura fino a ENTER s.close(); } }</pre>	<pre>import java.io.*; public class Input { /** * Legge da tastiera * con necessaria gestione eccezioni */ public static void main(String[] args) { BufferedReader s = new BufferedReader (new InputStreamReader(System.in)); System.out.print("\n\nPremi ENTER per terminare .. "); try { s.readLine(); // lettura fino a ENTER } catch(Exception e) { // gestione eccezioni } try { s.close(); } catch (Exception e) { // gestione eccezioni } } }</pre>
--	---



Modifica formato: precisione

Classe Formatter

sintassi per la **stringa di formattazione**

`%[argument_index$][flags][width][.precision] conversion`

³⁵₁₇ l'**indice dell'argomento** serve per individuare l'argomento nella lista. Ad esempio, `%2$` indica il secondo argomento della lista

³⁵₁₇ un **flag** indica un'opzione del formato. Ad esempio '+' richiede che sia incluso il segno, oppure '0' richiede di riempire con zero le posizioni non occupate (padding)

³⁵₁₇ **width** indica il numero minimo di caratteri

³⁵₁₇ **precision** il numero di cifre decimali

Tale **stringa di formattazione** è utilizzata anche da **printf** metodo introdotto dalla versione **Tiger** (come venne chiamata la J2SE 5.0)

`printf(String format, Object... args)`



Ad es:

```
System.out.printf("%3d %8.2f %n", tempo, valore); // la formattazione è con la virgola
```

Il primo specificatore % riguarda l'argomento (o lista di argomenti):

³⁵₁₇ riserva **tre** posizioni per il numero **intero** con identificatore tempo

³⁵₁₇ riserva 8 posizioni con 2 cifre decimali per **numero reale** con identificatore valore

Il secondo specificatore % non corrisponde ad un argomento: **%n** indica un "a capo" con portabilità interpiattaforma (*correct platform-specific line separator*) che non si ottiene usando la sequenza di escape `\n`

Effetto:

Tempo	Valore
3	5,88

Consultabile all'indirizzo <http://java.sun.com/javase/6/docs/api/java/util/Formatter.html#syntax>

Conversion	Argument Category	Description
'b', 'B'	general	If the argument <i>arg</i> is null, then the result is "false". If <i>arg</i> is a boolean or Boolean , then the result is the string returned by String.valueOf() . Otherwise, the result is "true".
'h', 'H'	general	If the argument <i>arg</i> is null, then the result is "null". Otherwise, the result is obtained by invoking <code>Integer.toHexString(arg.hashCode())</code> .
's', 'S'	general	If the argument <i>arg</i> is null, then the result is "null". If <i>arg</i> implements Formattable , then <code>arg.formatTo</code> is invoked. Otherwise, the result is obtained by invoking <code>arg.toString()</code> .
'c', 'C'	character	The result is a Unicode character
'd'	integral	The result is formatted as a decimal integer
'o'	integral	The result is formatted as an octal integer
'x', 'X'	integral	The result is formatted as a hexadecimal integer
'e', 'E'	floating point	The result is formatted as a decimal number in computerized scientific notation
'f'	floating point	The result is formatted as a decimal number
'g', 'G'	floating point	The result is formatted using computerized scientific notation or decimal format , depending on the precision and the value after rounding.
'a', 'A'	floating point	The result is formatted as a hexadecimal floating-point number with a significand and an exponent
't', 'T'	date/time	Prefix for date and time conversion characters. See Date/Time Conversions .
'%'	percent	The result is a literal '%' ('\u0025')
'n'	line separator	The result is the platform-specific line separator



Altra modalità per modificare formato: precisione

Classe DecimalFormat

```
import java.text.*;          // per DecimalFormat
class <NomeClasse> {
    public <nomeMetodo>() {
        DecimalFormat f = new DecimalFormat ("0.00");    // per due cifre decimali
        double x1=0;
        System.out.println ("....." + f.format (x1));
    }
}
```

Ad esempio:

```
import java.text.*;

public class Precisione {
    public static void main(String[] args) {
        float valore1 = 0.933F;
        int tempo = 2 ;
        DecimalFormat f = new DecimalFormat ("0.00");    // per due cifre decimali
        System.out.println("Tempo\tValore1\n");
        System.out.println( tempo + "\t\t" + f.format(valore1));    // si noti la formattazione con virgola
    }
}
```

Effetto:

Tempo	Valore1
2	0,93

Altra modalità per modificare formato: uso BigDecimal

```
import java.math.BigDecimal; // per BigDecimal

public class Preciso {

    public static void main(String[] args) {

        float valore;
        int tempo = 3;

        valore = 35.879345F;

        System.out.println("\nTempo\tValore\n");

        BigDecimal bgv = new BigDecimal(valore);

        bgv = bgv.setScale(2, BigDecimal.ROUND_HALF_UP);

        System.out.println( tempo + "\t\t" + bgv.floatValue());

    }
}
```

Effetto:

Tempo	Valore
3	35.88