Lezione V
Lu 8 Ott. 2007

Tipi di dati fondamentali

Un programma che elabora numeri

```
public class Coins1
{
   public static void main(String[] args)
   {
      int lit = 15000;     // lire italiane
      double euro = 2.35; // euro

      // calcola il valore totale
      double totalEuro = euro + lit / 1936.27;

      // stampa il valore totale
      System.out.print("Valore totale in euro ");
      System.out.println(totalEuro);
   }
}
```

Un programma che elabora numeri

- Questo programma elabora ***due tipi di numeri***
  - ***numeri interi*** per le lire italiane, che non prevedono l'uso di decimi e centesimi e quindi non hanno bisogno di una parte frazionaria
  - ***numeri frazionari*** ("in virgola mobile") per gli euro, che prevedono l'uso di decimi e centesimi e assumono valori con il separatore decimale
- I numeri interi (positivi e negativi) si rappresentano in Java con il tipo di dati **int**
- I numeri in virgola mobile (positivi e negativi, ***a precisione doppia***) si rappresentano in Java con il tipo di dati **double** **(IEEE 754 doppia precisione)**

Perché usare due tipi di numeri?

- In realtà sarebbe possibile usare numeri in virgola mobile anche per rappresentare i numeri interi, ma ecco due buoni motivi per non farlo
  - **"pratica"**: i numeri interi rappresentati come tipo di dati **int** sono più ***efficienti***, perché ***occupano meno spazio in memoria*** e sono ***elaborati più velocemente***
  - **"filosofia"**: indicando esplicitamente che per le lire italiane usiamo un numero intero, rendiamo ***evidente*** il fatto che non esistono i decimali per le lire italiane
    - ***è importante rendere comprensibili i programmi!***

I commenti

- Nel programma sono presenti anche dei ***commenti***, che vengono ***ignorati*** dal compilatore, ma che rendono il programma molto più comprensibile

```
// lire italiane
```

- Un commento ***inizia con una doppia barra*** // e ***termina alla fine della riga***
- Nel commento si può scrivere ***qualsiasi cosa***
- Se il commento si deve estendere ***per più righe***, è molto scomodo usare tante volte la sequenza //
- Si può iniziare un commento con /* e terminarlo con */

```
// questo e' un commento
// lungo,inutile...
// ... e anche scomodo
```

```
/*
 questo e' un commento
 lungo, inutile...
 ma piu' comodo
*/
```

Alcune note sintattiche

- L'operatore che indica la divisione è /, quello che indica la moltiplicazione è *

```
lit / 1936.27
```

- Quando si scrivono numeri in virgola mobile, bisogna usare il ***punto*** come separatore decimale, invece della virgola (uso anglosassone)

```
1936.27
```

- Quando si scrivono numeri, non bisogna indicare il punto separatore delle migliaia

```
15000
```

- I numeri in virgola mobile si possono anche esprimere in ***notazione esponenziale***

```
1.93627E3 // vale 1.93627 × 10^3
```

System.out.println ()

- Il metodo **`System.out.print()`** invia una stringa all'output standard come **`System.out.println()`** ma non va a capo alla fine della stringa stampata
- **`System.out.println(parametro)`**
  - Come parametro possiamo avere una stringa
    - **`System.out.println("Hello, World!");`**
  - Il parametro puo' essere un numero o una variabile numerica
    - **`System.out.println(7+5);`**
    - **`System.out.println(totalEuro);`**
- out è un oggetto di classe PrintStream definito nella classe System della libreria standard. Consultare la documentazione della classe System e della classe PrintStream (javadoc)

Uso delle variabili

L'uso delle variabili

- Il programma fa uso di ***variabili*** di tipo numerico
  - **lit** di tipo **int**, **euro** e **totalEuro** di tipo **double**
- Le ***variabili*** sono spazi di memoria, identificati da un ***identificatore (nome)***, che possono conservare ***valori*** di un ***determinato tipo***
- Ciascuna variabile deve essere ***definita***, indicandone il ***tipo*** ed il ***nome***

```
int lit;
```

- Una variabile può contenere soltanto valori del suo ***stesso tipo***
- Nella ***definizione di una variabile***, è possibile ***assegnare*** un ***valore iniziale***

```
int lit = 15000;
```

L'uso delle variabili

- Il programma poteva risolvere lo stesso problema anche senza fare uso di variabili

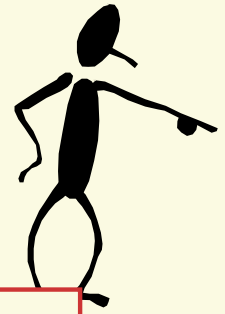
```
public class Coins2
{ public static void main(String[] args)
   { System.out.print("Valore totale in euro ");
     System.out.println(2.35 + 15000 / 1936.27);
    }
}
```

ma sarebbe stato ***molto meno comprensibile*** e ***modificabile con difficoltà***

I nomi delle variabili

- La scelta dei nomi per le variabili è molto importante, ed è bene ***scegliere nomi che descrivano adeguatamente la funzione della variabile***
- In Java, un nome (di variabile, di metodo, di classe…) può essere composto da ***lettere***, da ***numeri*** e dal ***carattere di sottolineatura***, ma
  - deve iniziare con una lettera (anche _ va bene)
  - non può essere una parola chiave del linguaggio
  - non può contenere spazi
- Le lettere ***maiuscole*** sono diverse dalle ***minuscole***! Ma è buona norma non usare nello stesso programma nomi di variabili che differiscano soltanto per una maiuscola

Definizione di variabili



- Sintassi:

```
nomeTipo nomeVariabile;
```

```
nomeTipo nomeVariabile = espressione;
```

- Scopo: definire la nuova variabile ***nomeVariabile***, di tipo ***nomeTipo***, ed eventualmente assegnarle il valore iniziale ***espressione***
- Di solito in Java si usano le seguenti ***convenzioni***
  - ***i nomi di variabili e di metodi iniziano con una lettera minuscola***

```
lit
```

```
main
```

```
println
```

  - ***i nomi di classi iniziano con una lettera maiuscola***

```
Coins1
```

  - i nomi composti, in entrambi i casi, si ottengono attaccando le parole successive alla prima con la maiuscola

```
totalEuro
```

```
MoveRectangle
```

L'uso delle variabili

- Abbiamo visto come i programmi usino le variabili per memorizzare i valori da elaborare e i risultati dell'elaborazione
- Le ***variabili*** sono posizioni in memoria che possono conservare ***valori*** di un ***determinato tipo***
- Il valore memorizzato in una variabile può essere ***modificato***, non soltanto ***inizializzato***…
- Il cambiamento del valore di una variabile si ottiene con un ***enunciato di assegnazione***

L'uso delle variabili

```
public class Coins2
{  public static void main(String[] args)
   { int lit = 15000;          // lire italiane
     double euro = 2.35;       // euro
     double dollars = 3.05; // dollari
     // calcola il valore totale
     // sommando successivamente i contributi
     double totalEuro = lit / 1936.27;
     totalEuro = totalEuro + euro;
     totalEuro = totalEuro + dollars * 0.93;
     System.out.print("Valore totale in euro  ");
     System.out.println(totalEuro);
   }
}
```

L'uso delle variabili

- In questo caso il valore della variabile **totalEuro** ***cambia*** durante l'esecuzione del programma
  - per prima cosa la variabile viene ***inizializzata*** contestualmente alla sua ***definizione***

```
double totalEuro = lit / 1936.27;
```

  - poi la variabile viene ***incrementata***, due volte

```
totalEuro = totalEuro + euro;
totalEuro = totalEuro + dollars * 0.93;
```

mediante ***enunciati di assegnazione***

L'assegnazione

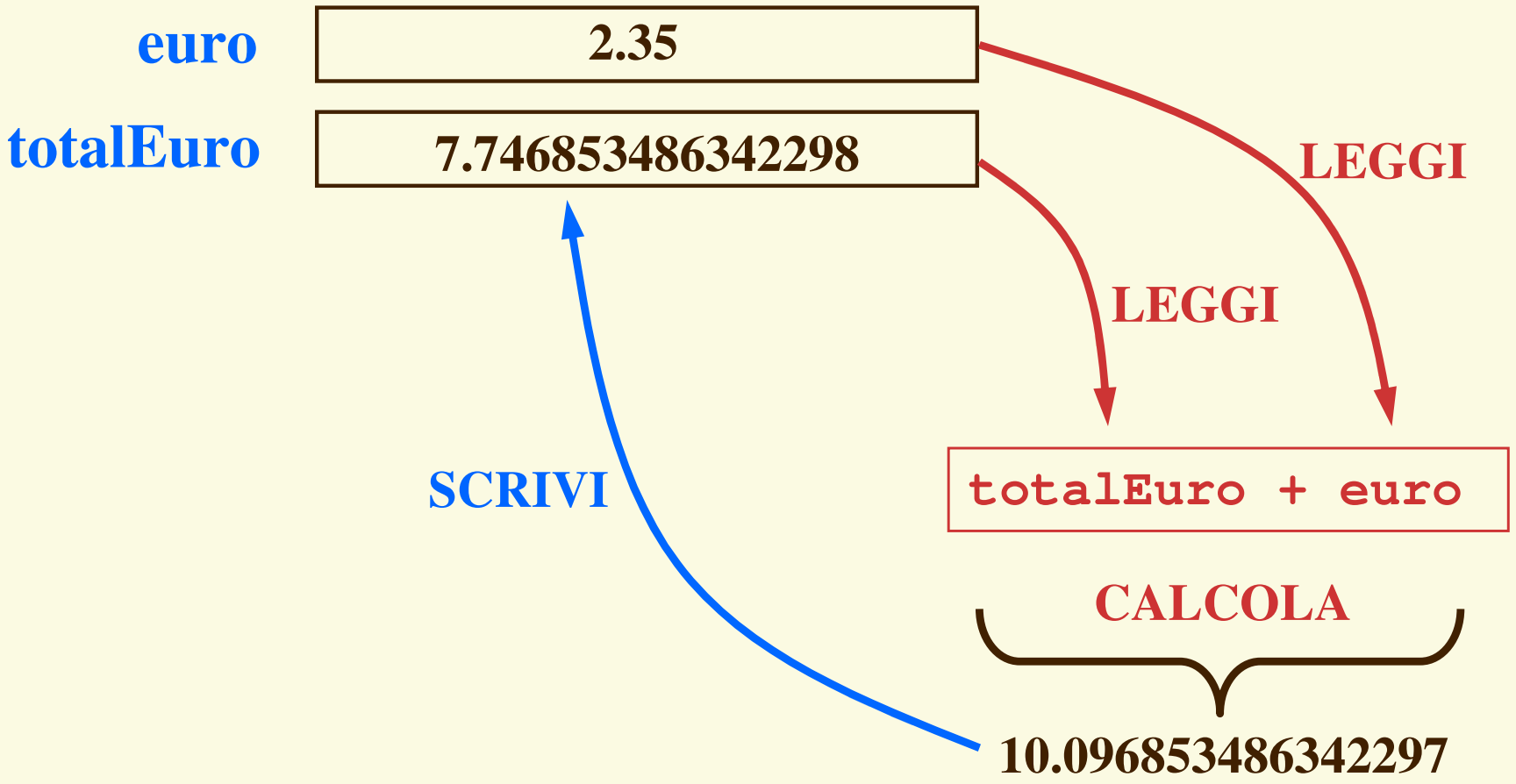
- Analizziamo l'enunciato di assegnazione

```
totalEuro = totalEuro + euro;
```

- Cosa significa? ***Non*** certo che **totalEuro *è uguale*** a se stessa più qualcos'altro…
- L'enunciato di assegnazione significa

  ***Calcola* il valore dell'espressione a *destra* del segno = e *scrivi* il risultato nella posizione di memoria assegnata alla variabile indicata a *sinistra* del segno =**

L'assegnazione



Assegnazione o definizione?

- Attenzione a non confondere la ***definizione*** di una variabile con un enunciato di ***assegnazione***!

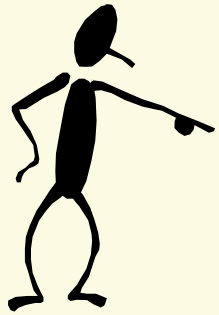
```
double totalEuro = lit / 1936.27;
totalEuro = totalEuro + euro;
```

- La definizione di una variabile inizia specificando il ***tipo*** della variabile, l'assegnazione no
- ***Una variabile può essere definita una volta sola in una sezione di codice***, mentre le si può assegnare un valore molte volte
- Il compilatore segnala come errore il tentativo di definire una variabile una seconda volta

```
double euro = 2;
double euro = euro + 3;
```

euro is already defined

Assegnazione



- Sintassi:

```
nomeVariabile = espressione;
```

- Scopo: assegnare il nuovo valore ***espressione*** alla variabile ***nomeVariabile***
- Nota: purtroppo Java (come C e C++) utilizza il segno **=** per indicare l'assegnazione, creando confusione con l'operatore di uguaglianza (che vedremo essere un doppio segno **=**, cioè **==**); altri linguaggi usano simboli diversi per l'assegnazione (ad esempio, il linguaggio Pascal usa **:=**)

Costanti

L'uso delle costanti

- Un programma per il cambio di valuta

```
public class Convert1
{
   public static void main(String[] args)
   {
      double dollars = 2.35;
      double euro = dollars * 0.84;
   }
}
```

- Chi legge il programma potrebbe legittimamente chiedersi quale sia il significato del "***numero magico***" **0.84** usato nel programma per convertire i dollari in euro...

L'uso delle costanti

- Così come si usano nomi simbolici descrittivi per le variabili, è opportuno assegnare ***nomi simbolici*** anche alle ***costanti*** utilizzate nei programmi

```
public class Convert2
{  public static void main(String[] args)
   {
      final double EURO_PER_DOLLAR = 0.84;
      double dollars = 2.35;
      double euro = dollars * EURO_PER_DOLLAR;
   }
}
```

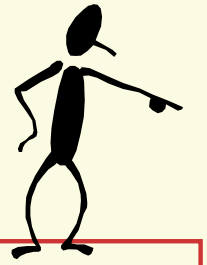
- Un primo ***vantaggio*** molto importante
  aumenta la leggibilità

L'uso delle costanti

- Un ***altro vantaggio***: se il valore della costante deve cambiare (nel nostro caso, perché varia il tasso di cambio euro/dollaro), la modifica va fatta ***in un solo punto*** del codice!

```
public class Convert3
{
   public static void main(String[] args)
   {  final double EURO_PER_DOLLAR = 0.84;
      double dollars1 = 2.35;
      double euro1 = dollars1 * EURO_PER_DOLLAR;
      double dollars2 = 3.45;
      double euro2 = dollars2 * EURO_PER_DOLLAR;
   }
}
```

Definizione di costante



- Sintassi:

```
final nomeTipo NOME_COSTANTE = espressione;
```

- Scopo: definire la costante ***NOME_COSTANTE*** di tipo ***nomeTipo***, assegnandole il valore ***espressione***, che non potrà più essere modificato
- Nota: il compilatore segnala come ***errore semantico*** il tentativo di assegnare un ***nuovo valore*** a una costante, dopo la sua inizializzazione
- Di solito in Java si usa la seguente convenzione
  - ***i nomi di costanti sono formati da lettere maiuscole***
    - i nomi composti si ottengono attaccando le parole successive alla prima con un ***carattere di sottolineatura***

I tipi fondamentali di dati numerici

- Numeri interi
- **byte**
  - intero a 8 bit
- **short**
  - intero a 16 bit
- **int**
  - intero a 32 bit
- **long**
  - intero a 64 bit

- Numeri in virgola mobile
- **float**
  - virgola mobile a singola precisione (IEEE 754 – 32 bit)
- **double**
  - virgola mobile doppia precisione (IEEE 754 – 64 bit)

Il linguaggio Java, a differenza di C, definisce il numero di bit usati per la rappresentazione dei tipi fondamentali di dati.

Altri tipi di dati numerici

- In generale useremo **int** per i numeri interi e **double** per i numeri frazionari, a meno di non avere qualche particolare motivo per fare diversamente
- Ad esempio useremo il tipo long in quelle applicazioni dove l'intervallo rappresentato dal tipo int non sia sufficiente
- La precisione del tipo **float** (circa ***sette cifre*** in base dieci) non e' generalmente sufficiente, per cui si usa il tipo comunemente **double** (circa ***quindici cifre in base*** dieci) per i numeri frazionari

Rappresentazione dell'Informazione (Horstmann app. H)

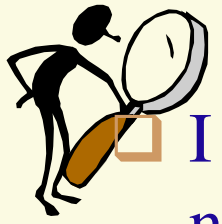
Rappresentazione dell'Informazione

- L'informazione che l'uomo elabora è varia:
  - testi, numeri, immagini, suoni
- Per memorizzare ed elaborare queste informazioni in un sistema informatico (computer) è necessario definire una codifica che consenta di esprimerle in formato adatto all'elaboratore
- Vediamo come si codifica l'informazione relativa a:
  - numeri
  - testi
  - immagini

Notazione posizionale in base decimale

- I numeri naturali sono definiti dal seguente insieme

$$N = \{0,1,2,...\}$$



- I numeri naturali che siamo abituati a utilizzare sono espressi nella ***notazione posizionale*** in ***base decimale***
  - ***base decimale:*** usiamo ***dieci cifre diverse*** (da 0 a 9)
  - ***notazione posizionale:*** ***cifre uguali in posizioni diverse hanno significato diverso*** (si dice anche che hanno ***peso*** diverso, cioè pesano diversamente nella determinazione del valore del numero espresso)
  - il ***peso*** di una cifra è uguale alla ***base del sistema di numerazione*** (**10**, in questo caso), elevata alla ***potenza uguale alla posizione*** della cifra nel numero, posizione che ***si incrementa da destra a sinistra a partire da 0***

$$434 = 4 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$$

Notazione posizionale in base decimale

- in generale con n cifre

$$a_{n-1}a_{n-2}\ldots a_0 = a_{n-1}10^{n-1} + a_{n-2}10^{n-2} + \ldots + a_0 10^0, a_k \in \{0,1,\ldots,9\}$$
$$k = 0,1,\ldots,n-1$$

- Anche i numeri razionali (frazioni) si possono esprimere con notazione posizionale in base decimale
- la parte frazionaria, a destra del simbolo separatore, si valuta con potenze ***negative***

$$4.34 = 4 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}$$

- Come separatore fra parte intera e decimale adopereremo nel corso il punto (convenzione anglosassone), e non la virgola (convenzione italiana)

Numeri *Naturali* in Notazione Binaria

- I computer usano invece ***numeri binari***, cioè numeri rappresentati con ***notazione posizionale in base binaria***
  - la base binaria usa solo ***due cifre diverse***, 0 e 1
  - la conversione da base binaria a decimale è semplice

base 2

base 10

$$1101_2 = (1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0)_{10} = 13_{10}$$

$$1.101_2 = (1\cdot 2^0 + 1\cdot 2^{-1} + 0\cdot 2^{-2} + 1\cdot 2^{-3})_{10} = 1.625_{10}$$

- I numeri binari sono più facili da manipolare per i sistemi elettronici, perché è meno complicato costruire circuiti logici che distinguono tra due stati ("acceso" e "spento"), piuttosto che fra ***dieci livelli diversi*** di tensione

Numeri naturali in notazione binaria

- La ***conversione*** di un numero ***da base decimale a base binaria*** è, invece, più complessa
- Innanzitutto, la parte intera del numero va elaborata indipendentemente dalla eventuale parte frazionaria
  - la parte intera del numero decimale viene convertita nella parte intera del numero binario
  - la parte frazionaria del numero decimale viene convertita nella parte frazionaria del numero binario
  - la posizione del punto separatore rimane invariata

Numeri naturali in notazione binaria

- Per convertire ***la sola parte intera***, si divide il numero per 2, eliminando l'eventuale resto e continuando a dividere per 2 il quoziente ottenuto fino a quando non si ottiene quoziente uguale a 0
- Il numero binario si ottiene scrivendo ***la serie dei resti*** delle divisioni, ***iniziando dall'ultimo*** resto ottenuto
- **Attenzione**: non fermarsi quando si ottiene **quoziente 1**, ma proseguire fino a **0**

```
100 / 2 = 50 resto 0
 50 / 2 = 25 resto 0
 25 / 2 = 12 resto 1
 12 / 2 =  6 resto 0
  6 / 2 =  3 resto 0
  3 / 2 =  1 resto 1
  1 / 2 =  0 resto 1
```

$100_{10} = 1100100_2$

Numeri razionali binari in notazione a virgola fissa

- Per convertire ***la sola parte frazionaria***, si moltiplica il numero per 2, sottraendo 1 dal prodotto se è maggiore di 1 e continuando a moltiplicare per 2 il risultato così ottenuto fino a quando non si ottiene un risultato uguale a 0 oppure un risultato già ottenuto in precedenza
- Il numero binario si ottiene scrivendo ***la serie delle parti intere dei prodotti*** ottenuti, ***iniziando dal primo***
- Se si ottiene un risultato già ottenuto in precedenza, il numero sarà ***periodico***, anche se ***non lo era*** in base decimale

```
0.35 · 2 = 0.7
0.7  · 2 = 1.4
0.4  · 2 = 0.8
0.8  · 2 = 1.6
0.6  · 2 = 1.2
0.2  · 2 = 0.4
```

$$0.35_{10} = 0,0\overline{10110}_2$$

Lezione VI
Ma 9 Ott. 2007

Rappresentazione dei numeri

Numeri Naturali

- Con 8 cifre binarie (o come si dice 8 bit) in binario si possono rappresentare 2^8 disposizioni, pari a 256 numeri diversi (disposizione con ripetizione di due simboli):

  - $0_{10} = 0000\ 0000_2$
  - $1_{10} = 0000\ 0001_2$
  - $2_{10} = 0000\ 0010_2$
  - $3_{10} = 0000\ 0011_2$
  - $4_{10} = 0000\ 0100_2$
  - ...
  - $254_{10} = 1111\ 1110_2$
  - $255_{10} = 1111\ 1111_2 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$

Con **n bit** si rappresentano **2^n** numeri interi

- 2^8 => 256 numeri
- 2^{16} => 65 536 numeri
- 2^{32} => 4 294 967 296 numeri

Numeri Interi Binari $Z = \{\ldots,-2,-1,0,+1,+2,\ldots\}$ in Formato Modulo e Segno

- Si può usare un bit per esprimere l'informazione del segno

- $+\ 127_{10} =\ 0|111\ 1111_2$
- $+\ 126_{10} =\ 0|111\ 1110_2$ → Bit di segno + (bit più significativo)
- ... = ...
- $+\ 1_{10} =\ 0000\ 0001_2$
- $+\ 0_{10} =\ 0000\ 0000_2$ DUE RAPPRESENTAZIONI
- $-\ 0_{10} =\ 1000\ 0000_2$ DELLO ZERO
- $-\ 1_{10} =\ 1000\ 0001_2$
- ... = ...
- $-\ 126_{10} =\ 1111\ 1110_2$ → Bit di segno - (bit più significativo)
- $-\ 127_{10} =\ 1111\ 1111_2$

In pratica questa rappresentazione non viene usata

Numeri Interibinari in Formato Complemento a due

- In complemento a due la sequenza di n cifre binarie
  - $a_{n-1}a_{n-2}\ldots a_1 a_0$ significa $-a_{n-1}\,2^{n-1} + a_{n-2}\,2^{n-2} + \ldots + a_1\,2^1 + a_0\,2^0$

- $+\ 127_{10} = \ 0111\ 1111_2$
- $+\ 126_{10} = \ 0111\ 1110_2$
- $\ldots \quad = \quad \ldots$
- $+\ 1_{10} = \ 0000\ 0001_2$
- $0_{10} = \ 0000\ 0000_2$
- $-\ 1_{10} = \ 1111\ 1111_2$
- $\ldots \quad = \quad .$
- $-\ 127_{10} = \ 1000\ 0001_2$
- $-\ 128_{10} = \ 1000\ 0000_2$

0 quando il numero è non negativo

Rappresentazione unica dello zero

Il bit piu' significativo indica ancora il segno

1 quando il numero è negativo

Con n = 8 bit si rappresentano numeri da –128 a +127

Con **n bit (cifre binarie)** si rappresentano numeri da -2^{n-1} a $+2^{n-1}-1$

Numeri Interi binari in Formato complemento a due

- Attenzione: data una sequenza di bit che rappresentano un numero in binario, per convertirlo in decimale dobbiamo conoscere il formato in cui il numero è espresso
  - **Complemento a 2 a 8 bit**
    - $1111\ 1111_2 = -1_{10}$
    - $\mathbf{1}111\ 1111_2 = \mathbf{-}2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0$
  - **Complemento a 2 a 16 bit**
    - $0000\ 0000\ 1111\ 1111_2 = +255_{10}$
    - $\mathbf{1}111\ 1111_2 = \mathbf{+}2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0$
    - **Attenzione: le cifre 0 a sinistra possono essere omesse**
    - $\mathbf{0000\ 0000}\ 1111\ 1111_2 = 1111\ 1111_2$
  - **Modulo e segno a 8 bit**
    - $\mathbf{1}111\ 1111_2 = -127_{10}$
    - $\mathbf{1}111\ 1111_2 = -(+2^6+2^5+2^4+2^3+2^2+2^1+2^0)$

Operazioni in Complemento a due
Somma algebrica

- Somma con le regole della somma in colonna
  - **0 + 0 = 0** **0 + 1 = 1 + 0 = 1** **1 + 1 = 0 con riporto di 1**
- Esempi con rappresentazione a 8 bit

0000 0101 +	+5 +	
0000 0010	+2	
0000 0111	+7	

	0000 0101 +	+5 +
	1111 1110	−2
1	**0000 0011**	+3

  - Un riporto a sinistra della cifra piu' significativa puo' essere ignorato a patto che si ottenga come risultato un numero che puo' essere rappresentato con il numero di bit a disposizione
- Con rappresentazione a 8 bit

0111 1111	+127+	
0000 0001	1	
1000 0000	**−128!!**	

	1000 0000	−128
	1111 1111	−1
1	**0111 1111**	**+127!!**

Esempi di Errore di Trabocco (Overflow)

Regola per l'errore di Overflow

- Errore di Overflow: quando si ha superamento dei limiti di rappresentazione con i bit a disposizione
- Analizzando la due cifre piu' significative del risultato si puo' determinare se e' avvenuta una condizione di overflow
- ASSENZA DI OVERFLOW
  - Nessun riporto nelle due cifre piu' significative
  - Riporto in entrambe le cifre piu' significative
- PRESENZA DI OVERFLOW
  - Riporto in una sola delle due cifre piu' significative
- In un numero si dicono ***cifre più significative*** quelle che occupano le posizioni ***più a sinistra***, ***meno significative*** quelle che occupano le posizioni ***più a destra***
  - Es. Nel numero **102**, **1** e 2 sono le cifre è più e meno significative, rispettivamente.

Inversione in complemento a due

- La rappresentazione in complemento a due dei numeri interi e' adottata in tutti i computer per la semplicità della realizzazione della procedura di somma algebrica.
- Inverso di un numero (Cambio segno)

	+7	-7
	00000111	11111001
Complemento a 1	11111000	00000110
Incremento di 1	00000001	00000001
–	11111001	00000111
	-7	+7

Complemento a 1: 0 diventa 1, 1 diventa 0

Rappresentazione Esadecimale

- E' la rappresentazione in base 16
  - Si usano 16 cifre: (0,1,..., 9, A, B, C, D, E, F)
  - **$A = 10_{10}$, $B = 11_{10}$, $C = 12_{10}$, $D = 13_{10}$, $E = 14_{10}$, $F = 15_{10}$**
- Viene usata dall'uomo per rappresentare numeri naturali (senza segno) binari o sequenze di bit in modo più compatto
- Per convertire un numero binario in esadecimale si raggruppano i bit a gruppi di quattro partendo da destra verso sinistra
- Esempio a 8 bit (byte)
  - 01111111_2 => $0111 | 1111_2$ => **7 F_{16}**
- Esempio a 32 bit: $11110001101011100010011110000101_2$
  - 1111|**0001**|1010|**1110**|0010|**0111**|1000|**0101**
  - F| **1**| A| E| 2| **7**| 8| **5**
  - **$F1AE\ 2785_{16}$**

Conversioni Esadecimali

- Per convertire un numero naturale da esadecimale a binario si convertono le sue singole cifre:

$3F_{16} = \underset{3}{0011}\ \underset{F}{1111}_2$

- Per convertire un numero naturale da esadecimale a decimale o viceversa si applica la definizione di notazione posizionale in base 16

$3F_{16} = 3 \times 16^1 + 15 \times 16^0 = 63_{10}$

$1000_{10} / 16 = 62_{10}$ resto 8
$62_{10} / 16 = 3_{10}$ resto E (14_{10})
$3_{10} / 16 = 0$ resto 3

$1000_{10} = 3E8_{16}$

  - Per convertire da decimale a esadecimale si puo' anche convertire in base 2 e poi da base 2 a base 16

$1000_{10} = 1111101000_2 = \mathbf{00}11\ 1110\ 1000_2 = 3E8_{16}$

Numeri Razionali e Reali

- I numeri reali devono essere rappresentati necessariamente in modo approssimato
- Infatti con n bit possiamo rappresentare 2^n numeri, mentre sappiamo che **in ogni intervallo reale** ci sono **infiniti numeri reali**
- Si utilizza una notazione a ***mantissa*** ed ***esponente***, come nel calcolo scientifico:
  - **1024.3** viene rappresentato come **1.0243***10^{3}
  - **1.0243** e' la mantissa
  - **3** e' l'esponente
- Nella rappresentazione binaria dei numeri reali la **mantissa** e l'**esponente** sono espressi **in binario**
- La base e' 2 anziche' 10

Numeri Reali

- Il motivo della base 2 e' che e' facile dividere e moltiplicare per 2 i numeri binari, semplicemente spostando a sinistra o a destra la posizione della virgola (virgola mobile)
  - $1101 = 1.101 * 2^3$
  - $0.0010 = 1.0 * 2^{-3}$
- In passato sono stati usati vari formati per rappresentare i numeri reali
- Una convenzione si e' imposta sulle altre
  - Standard IEEE 754

Standard IEEE 754

- Due formati: a 32 e 64 bit (**singola** e **doppia precisione**)
- Descriviamo il formato a 32 bit (singola precisione)
  - |**S**|**EEEEEEEE**|**MMMMMMMMMMMMMMMMMMMMMMM**|
  - |1| 8bit |23bit |
  - | 32 bit |

S un bit di segno della mantissa (bit piu' significativo)

E 8 bit per esprimere l'esponente

M 23 bit per esprimere la mantissa

- Il numero viene ***normalizzato*** in modo che il bit piu' significativo del numero si trovi immediatamente a sinistra della virgola
  - **1**011.11 viene normalizzato in **1.01111***2^{3}
  - Il bit a sinistra del punto non viene memorizzato, perche' dopo il processo di normalizzazione e' sempre 1
  - La mantissa da memorizzare e' **01111**

Standard IEEE754 singola precisione

- Effettuata la normalizzazione si memorizza
  - La **mantissa** in modulo e segno (S e M...M)
  - L'**esponente in eccesso 127 a 8 bit**
    - si rappresentano i numeri tra **–127** e **+ 128**
    - formato diverso dal complemento a due: il numero viene incrementato di 127 prima di essere convertito in binario come numero naturale
- Rappresentazione dello zero:
  - Il numero con esponente –127 e mantissa tutta a zero viene usato per rappresentare lo 0
  - zero: 0 00000000 00000000000000000000000
- L'esponente +128 viene usato per rappresentare ***NaN*** (Not a Number) simbolo speciale per indicare condizioni d'errore: ad esempio il risultato dell'operazione 0 / 0
  - NaN: 0 11111111 00000000000000000000000

Standard IEEE754 singola precisione

- Il numero piu' piccolo positivo rappresentabile e'
  - $\mathbf{1.00000000000000000000000_2 * 2^{-126} \sim 1.8 * 10^{-38}}$
- Il numero piu' grande rappresentabile e':
  - $\mathbf{1.11111111111111111111111 * 2^{+127} \sim 3.4 * 10^{+38}}$
- La distanza fra due numeri reali successivi rappresentabili in questo formato dipende dal valore dell'esponente, infatti i numeri piu' vicini differiranno per il valore del bit meno significativo della mantissa e percio' la loro distanza sara'
  - $\Delta = 2^{-23} * 2^{E}$ dove E e' il valore dell'esponente

Standard IEEE754

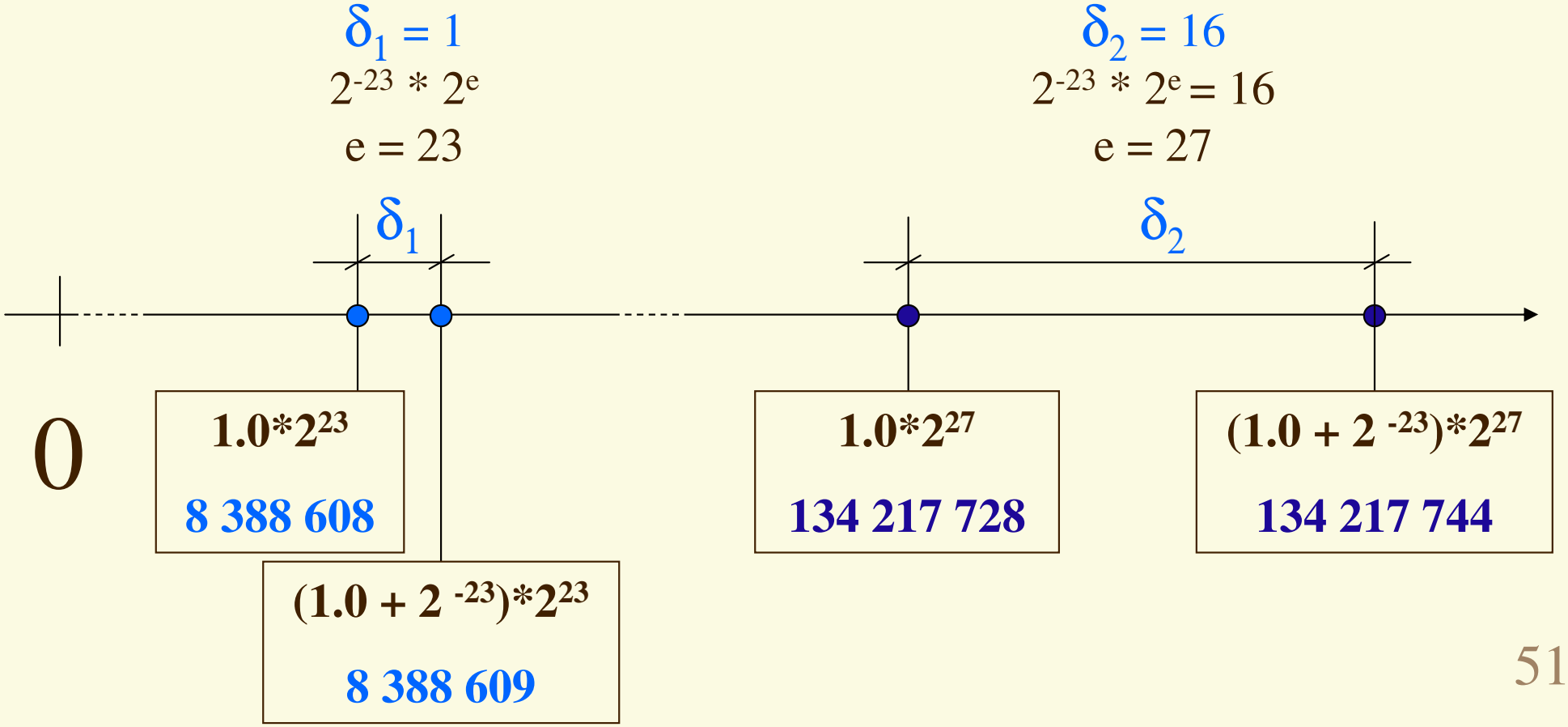
- Ad esempio si consideri il numero binario
  - **a = 1.00000000000000000000000*2^5** = $2^5 = 32_{10}$
  - Il numero successivo rappresentabile e':
  - **b = 1.00000000000000000000001*2^5** = $(1+ 2^{-23})*2^5$
  - La distanza fra i due numeri:

    $\delta = b - a = 2^{-23}\ 2^5 = 2^{-18}$

- La distanza dei numeri rappresentabili cresce al crescere del loro valore assoluto (ovvero la densita' dei numeri rappresentabili diminuisce)
- Di questi fatto si deve tener conto quando si sommano numeri reali in questo formato

Densita' dei numeri rappresentabili in virgola mobile

- **Esempio in formato IEEE754 Singola Precisione (mantissa a 23 bit)**



Standard IEEE754: approssimazione della rappresentazione

- Si consideri la somma **10.5 + 0.125**
- Traduciamola in binario e normalizziamola
  - $10.5_{10} = 1010.1_2 = 1.0101_2 * 2^3$
  - $0.125_{10} = 0.001_2 = 1.0 * 2^{-3}$
- Per eseguire la somma bisogna **riportare entrambi i termini allo stesso esponente**:
  - $10.5 + 0.125 = 1.0101 * 2^3 + 0.000001 * 2^3 = 1.010101 * 2^3$
- Si osservi che se il numero di bit destinati alla mantissa fosse stato inferiore a 6, l'operazione avrebbe dato per risultato
  - 10.5 + 0.125 = 10.5 !!!!!
- A causa della necessaria approssimazione introdotta dalla rappresentazione

Esercizi per Casa: conversioni

- decimale → binario complemento a due a 8 bit
  - 93, 25, 127, -34, -88, -125
- binario complemento a due a 8 bit → decimale
  - 0010 1111, 0111 0101, 1010 1111, 1100 1100
- binario → esadecimale
  - 1111 0101, 1001 0101, 1101 0001, 1000 0111
- esadecimale → binario
  - FAB8, 7CE0, 49B2, AF01
- esadecimale → decimale (numeri naturali)
  - 3F, 41, 2A, 5B
- decimale → esadecimale
  - 72, 27, 88, 116

Esercizi per Casa

- Inversione di numeri binari in complemento a 2 a 8 bit
  - 0011 1000, 1111 1111, 0101 0101, 1100 1111
- Somme algebriche binarie in complemento a due a 8 bit (segnalare i casi di overflow)
  - 0011 0111 + 0001 0000
  - 1000 0010 + 1000 1000
  - 0111 1110 + 1111 1101
  - 0111 1101 + 0001 1111

Ve 12-10-2007
Lezione ore 10:30
in aula P300
(via Luzzati)

Risposte del questionario nel sito didattico

Lezione VII
Me 10 Ott. 2007

Rappresentazione di testi

Rappresentazione di testi

- I caratteri sono rappresentati come numeri naturali (senza segno)
- A ciascun carattere viene associato un numero naturale. La tabella che associa i caratteri ai numeri si dice CODICE. L'uso di *Codici Standard* permette a computer di tipo diverso di scambiare testi
- Codice UNICODE
  - Usa 2 byte (16 bit) per ciascun carattere
  - Si possono rappresentare $2^{16} = 65{,}536$ caratteri
  - Praticamente tutti i caratteri degli alfabeti umani esistenti
- Codice ASCII
  - Sottoinsieme del codice UNICODE
  - ancora largamente usato
  - usa solo 7 bit
  - Si possono rappresentare solo $2^7 = 128$ caratteri

I codici *Unicode* e *ASCII*

- I programmi Java sono scritti usando l'insieme di caratteri *Unicode*
- *Unicode* utilizza, per rappresentare un carattere, un numero intero senza segno di 16 bit e definisce circa 39.000 caratteri in molti alfabeti diversi
- I primi 128 codici Unicode coincidono con l'insieme di caratteri *Basic Latin* noto anche come *ASCII* (American Standard Code for Information Interchange)
- In java, i token ***parole chiave***, ***caratteri di interpunzione***, **operatori** e ***costanti numeriche*** sono definiti usando i soli caratteri ASCII
- Noi utilizzeremo solo i caratteri ASCII anche per definire ***identificatori*** e quindi possiamo dimenticarci di Unicode

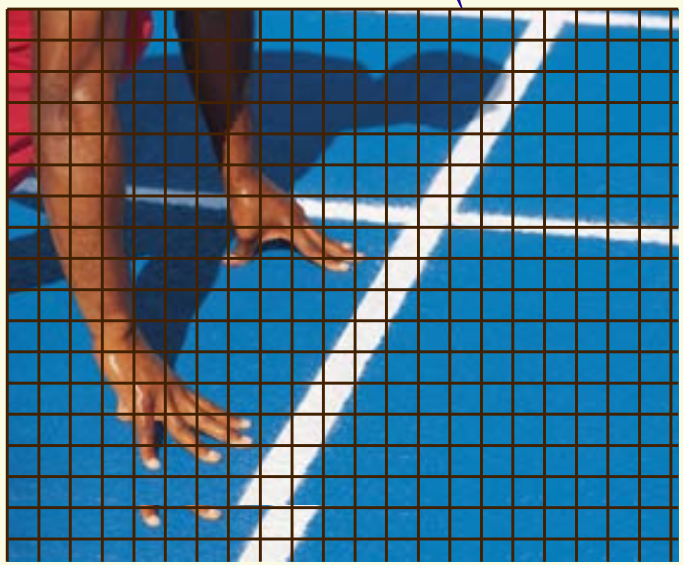
Il codice ASCII

- I primi 32 caratteri del codice ASCII (con codice da 0 a 31) sono *caratteri di controllo*, di nostro interesse sono solo i caratteri
  - 9 tabulatore `'\t'`
  - 10 nuova riga `'\n'`
  - 13 invio `'\r'`
- I caratteri da 32 a 127 sono caratteri stampabili
  - 32 spazio `' '`
  - da 48 a 57 caratteri numerici, le cifre decimali `'0'`, `'1'` …
  - da 65 a 90, da 97 a 122 caratteri alfabetici (maiuscoli e minuscoli)
  - da 33 a 47, da 58 a 64, da 91 a 96, da 123 a 127 caratteri di interpunzione

Il codice Unicode e i numeri in complemento a due

- Non bisogna confondere le rappresentazioni dei numeri e dei caratteri
- Il numero intero 1 in ***notazione binaria complemento a due a 16 bit*** ha la seguente rappresentazione
  - **$1_{10} = 0000\ 0000\ 0000\ 0001_2$**
- Il carattere '1' in ***Codice Unicode (16 bit)*** ha la seguente rappresentazione binaria
  - **'1' = 0031_{16} = 0000 0000 0011 0001**

Rappresentazione di Immagini

- Le immagini vengono *discretizzate* suddividendole in *Pixel (Picture Element)*
- 
- *Un Pixel* corrispondente a un singolo punto sullo schermo quando vengono visualizzate su video.
- Risoluzioni tipiche degli schermi (pixel per riga x pixel per colonna) sono
  - 640x480, 800x600, 1024x864, 1152x864, 1280x1024

Rappresentazione di Immagini

- A ciascun pixel e' necessario associare un colore
  - con 8 bit si rappresentano $2^8 = 256$ toni di colore
  - Con 16 bit fino a $2^{16} = 65\ 536$ colori
- Si comprende quindi come la memorizzazione di immagini richieda la memorizzazione di molti numeri
  - Un'immagine 1280 x 1024 con colori a 16 bit (2 byte) richiede 2.56 Mbyte ovvero
    - piu' di due milioni di byte (esattamenet 2 621 440)
    - 1 kByte = 1024 byte
    - 1 Mbyte = 1024 kByte

Rappresentazione di Immagini

- Per questo motivo le immagini non sono generalmente memorizzate come semplici sequenze di colori associati ai pixel, ma vengono utilizzati dei formati che permettono una *compressione,* ovvero una riduzione del numero di bytes usati per codificare l'immagine
- I due formati piu' comuni per memorizzare immagini statiche sono denominati ***gif*** e ***jpeg***
- Nel formato ***gif*** viene effettuata una riduzione dei byte senza perdita di informazione
- Nel formato **jpeg** si ha perdita di informazione. La qualita' dell'immagine risulta comunque perfettamente accettabile all'occhio umano

Rappresentazione di immagini

- Per memorizzare sequenze di immagini si usa il formato ***mpeg***, che e' un'estensione del *jpeg*. Nel formato ***mpeg*** vengono memorizzate non sequenze di immagini, ma solo le differenze tra l'immagine corrente (frame) e il frame precedente

Numeri interi in Java

- In Java tutti i tipi di dati fondamentali per numeri interi usano internamente la rappresentazione in complemento a due
- La JVM ***non segnala le condizioni di overflow*** nelle operazioni aritmetiche
  - si ottiene semplicemente un risultato errato
- L'unica operazione aritmetica tra numeri interi che genera un **errore** è la divisione con divisore zero
- Questo **errore** si chiama
  - **ArithmeticException**
  - E viene segnalato in esecuzione dall'interprete java, che arresta l'esecuzione del programma

Altri tipi di dati numerici

- In generale useremo **int** per i numeri interi e **double** per i numeri frazionari, a meno di non avere qualche particolare motivo per fare diversamente
- Ad esempio useremo il tipo long in quelle applicazioni dove l'intervallo rappresentato dal tipo int non sia sufficiente
- La precisione del tipo **float** (circa ***sette cifre*** in base dieci) non e' generalmente sufficiente, per cui si usa il tipo comunemente **double** (circa ***quindici cifre in base*** dieci) per i numeri frazionari

Tipi di dati fondamentali

tipo	Numero di byte	Intervallo di rappresentazione
byte	1 byte	$-2^7 \div 2^7-1$ -128 ÷ 127
short	2 byte	$-2^{15} \div 2^{15}-1$ -32768 ÷ 32767
int	4 byte	$-2^{31} \div 2^{31}-1$ ***circa -2e+9 ÷ + 2e+9***
long	8 byte	$-2^{63} \div 2^{63}-1$ circa -9e+18 ÷ 9e+18
float	4 byte	circa ± (1.8e-38 ÷ 3.4e+38)
double	8 byte	***circa ± (4.9e-324 ÷ 1.7e+308)***

Tipi di dati fondamentali

- Se servono i valori massimi e minimi dei numeri rappresentati con i vari tipi di dati non occorre ricordarli, perché nel ***pacchetto*** java.lang della libreria standard per ciascun tipo di dati fondamentali è presente una classe in cui sono definiti questi valori come costanti

tipo	Valore minimo	Valore massimo
byte	**Byte.MIN_VALUE**	**Byte.MAX_VALUE**
short	**Short.MIN_VALUE**	**Short.MAX_VALUE**
int	**Integer.MIN_VALUE**	**Integer.MAX_VALUE**
long	**Long.MIN_VALUE**	**Long.MAX_VALUE**
float	**Float.MIN_VALUE**	**Float.MAX_VALUE**
double	**Double.MIN_VALUE**	**Double.MAX_VALUE**

java.lang pacchetto di libreria speciale

Numeri in virgola mobile in Java

- Lo standard IEEE 754 prevede anche la rappresentazione di NaN, di +∞ e di -∞
- Sono definite le seguenti costanti
  - **Double.NaN**
  - **Double.NEGATIVE_INFINITY**
  - **Double.POSITIVE_INFINITY**
- e le corrispondenti costanti **Float**
  - **Float.NaN**
  - **Float.NEGATIVE_INFINITY**
  - **Float.POSITIVE_INFINITY**

Numeri in virgola mobile in Java

- In java la divisione con divisore zero non è un errore se effettuata tra numeri in virgola mobile (float o double)
  - se il dividendo è diverso da zero, il risultato è **infinito** (con il segno del dividendo)
  - se anche il dividendo è zero, il risultato non è un numero e viene usata la codifica speciale **NaN** (Not a Number)

Numeri in virgola mobile in Java

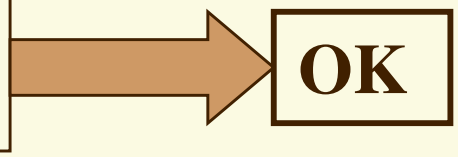
- Il tipo ***float*** (mantissa a 23 bit) esprime ***7 ÷ 8 cifre decimali significative***
  - Il tipo ***float*** rappresenta π con 8 cifre decimali significative
    - `float pi  = 3.1415927;`
  - Il numero 3.14159 26535 89793 23846 esprime π con 21 cifre decimali significative
- Il tipo ***double*** (mantissa a 52 bit) esprime ***15 ÷ 16 cifre decimali significative***
  - Il tipo ***double*** rappresenta π con 16 cifre significative
    - `double pi = 3.141592653589793;`

Conversioni fra diversi tipi fondamentali di dati

Assegnazioni con conversione

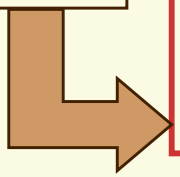
- In un'assegnazione, il ***tipo*** di dati dell'***espressione*** e della ***variabile*** a cui la si assegna devono essere ***compatibili***
  - se i tipi non sono compatibili, il compilatore segnala un ***errore*** (non sintattico ma ***semantico***)
- I tipi ***non*** sono compatibili se provocano una ***possibile perdita di informazione*** durante la conversione
- L'assegnazione di un valore di tipo numerico intero **int** a una variabile di tipo numerico in virgola mobile **double** non può provocare perdita di informazione, quindi è ammessa

```
int intVar = 2;
double doubleVar = intVar;
```



Tipi di dati numerici incompatibili

```
double doubleVar = 2.3;
int intVar = doubleVar;
```



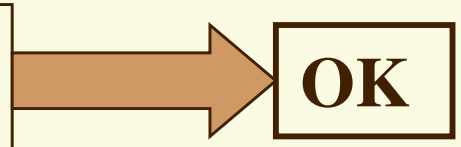
```
possible loss of precision
found    : double
required: int
```

- In questo caso si avrebbe una perdita di informazione, perché la (eventuale) ***parte frazionaria*** di un valore in virgola mobile non può essere memorizzata in una variabile di tipo intero
- Per questo motivo il compilatore non accetta un enunciato di questo tipo, segnalando l'**errore semantico** e interrompendo la compilazione

Conversioni forzate (*cast*)

- Ci sono però casi in cui si vuole effettivamente ottenere la ***conversione di un numero in virgola mobile in un numero intero***
- Lo si fa segnalando al compilatore l'intenzione ***esplicita*** di accettare l'eventuale perdita di informazione, mediante un ***cast*** ("forzatura")

```
double doubleVar = 2.3;
int intVar = (int)doubleVar;
```



- Alla variabile **intVar** viene così assegnato il valore 2, la ***parte intera*** dell'espressione (troncamento)

Operazioni aritmetiche
+ - * / %

Operazioni aritmetiche

- L'operatore di ***moltiplicazione*** * va sempre indicato ***esplicitamente***, non può essere ***sottinteso***
- Le operazioni di ***moltiplicazione*** e ***divisione*** ***hanno la precedenza*** sulle operazioni di ***addizione*** e ***sottrazione***, cioè vengono eseguite prima
- È possibile usare ***coppie di parentesi tonde*** per indicare in quale ordine valutare sotto-espressioni

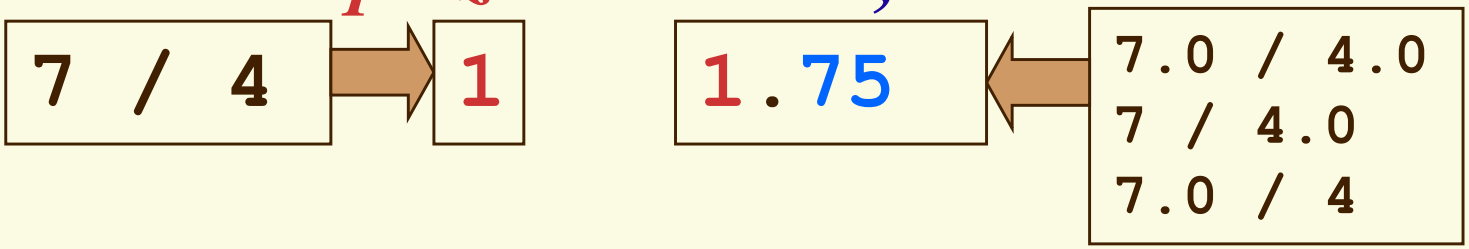
```
a + b / 2
```
$\neq$
```
(a + b) / 2
```

- In Java non esiste il ***simbolo di frazione,*** le frazioni vanno espresse "***in linea***", usando l'operatore di divisione

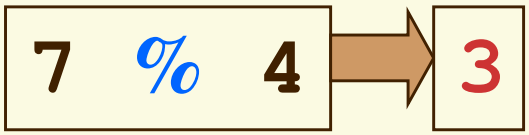
$$\frac{a+b}{2}$$ → `(a + b) / 2`

Operazioni aritmetiche

- Quando ***entrambi*** gli operandi sono numeri ***interi***, la ***divisione*** ha una caratteristica particolare, che può essere utile ma che va usata con attenzione
  - ***calcola il quoziente intero, scartando il resto!***

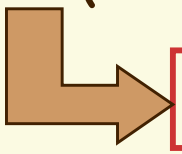


- Il resto della divisione tra numeri interi può essere calcolato usando l'operatore **% (modulo)** e il cui simbolo è stato scelto perché è simile all'operatore di divisione



Divisione fra interi

```
public class Coins5
{  public static void main(String[] args)
   {
      double euro = 2.35;
      final int CENT_PER_EURO = 100;
      int centEuro = (int)(euro * CENT_PER_EURO);
      int intEuro = centEuro / CENT_PER_EURO;
      centEuro = centEuro % CENT_PER_EURO;
      System.out.print(intEuro);
      System.out.print(" euro e ");
      System.out.print(centEuro);
      System.out.println(" centesimi");
   }
}
```

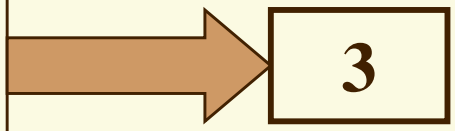


```
2 euro e 35 centesimi
```

Conversioni con arrotondamento

- La ***conversione forzata*** di un valore in virgola mobile in un valore intero avviene con ***troncamento***, ***trascurando la parte frazionaria***
- Spesso si vuole invece effettuare tale conversione con ***arrotondamento***, ***convertendo all'intero più vicino***
- Ad esempio, possiamo ***sommare 0.5 prima*** di fare la conversione

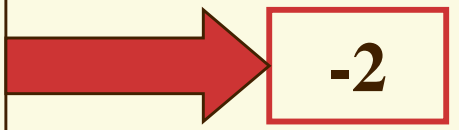
```
double rate = 2.95;
int intRate = (int)(rate + 0.5);
System.out.println(intRate);
```



Conversioni con arrotondamento

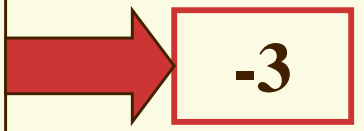
- Questo semplice algoritmo per arrotondare i numeri in virgola mobile funziona però soltanto per numeri positivi, quindi non è molto valido…

```
double rate = -2.95;
int intRate = (int)(rate + 0.5);
System.out.println(intRate);
```

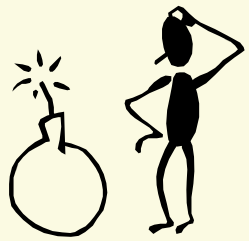


- Un'ottima soluzione è messa a disposizione dal metodo **round()** della classe **Math** della **java platform API**, che funziona bene per tutti i numeri

```
double rate = -2.95;
int intRate = (int)Math.round(rate);
System.out.println(intRate);
```

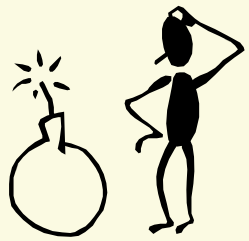


Errori di arrotondamento



- Gli errori di arrotondamento sono un fenomeno naturale nel calcolo in virgola mobile eseguito con un numero ***finito*** di cifre significative
  - calcolando 1/3 con due cifre significative, si ottiene 0,33
  - moltiplicando 0,33 per 3, si ottiene 0,99 e non 1
- Siamo abituati a valutare questi errori pensando alla rappresentazione dei numeri in base ***decimale***, ma gli elaboratori rappresentano i numeri in virgola mobile in base ***binaria*** e a volte si ottengono dei risultati inattesi!

Errori di arrotondamento



```
double f = 4.35;
int n = (int)(100 * f);
System.out.println(n);
```

434 ≠ 435

- Qui l'errore ***inatteso*** è dovuto al fatto che 4,35 non ha una ***rappresentazione esatta*** nel sistema binario, proprio come 1/3 non ha una rappresentazione esatta nel sistema decimale
  - 4,35 viene rappresentato con un numero appena un po' inferiore a 4,35, che, quando viene moltiplicato per 100, fornisce un numero appena un po' inferiore a 435, quanto basta però per essere troncato a 434
- È sempre meglio usare **Math.round()**

Funzioni più complesse

- Non esistono ***operatori*** per calcolare funzioni più complesse, come l'elevamento a potenza
- La classe **Math** della **java platform API** mette a disposizione ***metodi statici*** per il calcolo di tutte le funzioni algebriche e trigonometriche, richiedendo parametri **double** e restituendo risultati **double**
  - **Math.pow(x, y)** restituisce $\mathbf{x^y}$

    (il nome **pow** deriva da ***power***, potenza)
  - **Math.sqrt(x)** restituisce la radice quadrata di **x**

    (il nome **sqrt** deriva da ***square root***, radice quadrata)
  - Math.log(x) restituisce il logaritmo naturale di x
  - Math.sin(x) restituisce il seno di x espresso in radianti

Visitate la documentazione della classe **Math**

nella **java platform API**

Lezione VIII
Gi 11 Ott. 2007

Metodi statici

Il metodo Math.round()

```
double rate = -2.95;
int intRate = (int)Math.round(rate);
System.out.println(intRate);
```

- C'è una differenza sostanziale tra il metodo **round()** e, ad esempio, il metodo **println()** già visto
  - **println()** agisce su un oggetto (ad esempio, **System.out**)
  - **round()** ***non agisce su un oggetto*** (**Math** ***è una classe!***)
- Il metodo **Math.round()** è un ***metodo statico***

Il metodo Math.round( )

- Come si fa a capire che **System.out.println()** è un metodo applicato a un oggetto, mentre **Math.round()** no?
- ***La sintassi è identica…*****Math** sembra un oggetto!
- Tutte le classi, gli oggetti e i metodi della libreria standard seguono una rigida ***convenzione***
  - i nomi delle classi (**Math**, **System**) iniziano con una lettera ***maiuscola***
  - i nomi di oggetti (**out**) e metodi (**println()**, **round()**) iniziano con una lettera ***minuscola***
    - oggetti e metodi si distinguono perché ***solo i metodi sono sempre seguiti dalle parentesi tonde***

Invocazione di un metodo statico

- Nella documentazione della classe **java.lang.Math** troviamo la seguente ***firma*** del metodo round:

```
public static long round(double a)
```

firma del metodo round

- **`public`**: il metodo può essere invocato in qualsiasi classe
- **`static`**: il metodo è statico (vedremo che ci sono anche metodi non-statici)
  - un ***metodo statico*** si invoca usando il nome della classe in cui è definito, con la sintassi ***NomeClasse.nomeMetodo***
  - ***es.: Math.round(...)***
- **`long`**: tipo di dato restituito; il metodo restituisce un dato di tipo long, ovvero l'elaborazione delle istruzioni del metodo produce un dato di tipo long
  - è possibile che un metodo non restituisca dati, in questo caso il tipo del dato restituito è **`void`**

Invocazione di un metodo statico

- `round:` nome o identificatore del metodo
- `double a`: parametro esplicito del metodo.
  - un metodo può non avere parametri espliciti, averne uno o più di uno.
  - se non ha parametri nella sua firma non viene indicato nulla all'interno delle parentesi tonde
  - per ciascun parametro esplicito nella firma del metodo viene indicato un nome (identificatore) ***preceduto dal tipo***
  - Nei parametri vengono "passati" valori al metodo, valori che sono elaborati internamente
-
```
int intRate = (int)Math.round(rate);
```

Invocazione di un metodo statico

- Il valore restituito viene, generalmente, memorizzato in una variabile o usato in una espressione

```
int intRate = (int)Math.round(rate);
long longRate = Math.round(2.5) + Math.round(3.5);
```

- `(int)`: è una conversione forzata (già incontrata!) per convertire il dato restituito (di tipo long) in un dato di tipo int (tipo di dato della variabile intRate).
- La conversione da long a int non è automatica, perchè potrebbe generare perdita di precisione
  - infatti l'insieme dei numeri rappresentati con il tipo long è molto più vasto dell'insieme dei numeri rappresentati con il tipo int.

Invocazione di metodo statico

- Sintassi:

```
NomeClasse.nomeMetodo(parametri)
```

- Scopo: invocare il metodo statico ***nomeMetodo*** definito nella classe ***NomeClasse***, fornendo gli eventuali ***parametri*** richiesti

- La classe **java.lang.Math** contiene solo metodi statici:
  - Math.sin(), Math.exp(), Math.round() …

Combinare assegnazioni e aritmetica

- Abbiamo già visto come in Java sia possibile combinare in un unico enunciato ***un'assegnazione*** ed ***un'espressione aritmetica che coinvolge la variabile a cui si assegnerà il risultato***

```
totalEuro = totalEuro + dollars * 0.84;
```

- Questa operazione è talmente comune nella programmazione, che il linguaggio Java fornisce una ***scorciatoia***

```
totalEuro += dollars * 0.84;
```

che esiste per tutti gli operatori aritmetici

```
x = x * 2;
```



```
x *= 2;
```

Incremento di una variabile

- L'*incremento* di una variabile è l'operazione che consiste nell'***aumentarne il valore di uno***

```
int counter = 0;
counter = counter + 1;
```

- Questa operazione è talmente comune nella programmazione, che il linguaggio Java fornisce un ***operatore apposito per l'incremento***

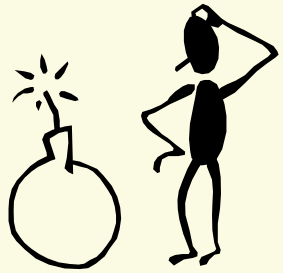
```
counter++;
```

e per il decremento

```
counter--;
```

Variabili non inizializzate

Variabili non inizializzate

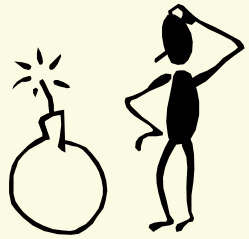


- È buona regola fornire ***sempre*** un valore di inizializzazione nella definizione di variabili
- Cosa succede altrimenti?

```
int lit;
```

  - la definizione di una variabile "***crea***" la variabile, cioè le ***riserva uno spazio*** nella memoria primaria (la quantità di spazio dipende dal tipo della variabile)
  - tale spazio di memoria non è "***vuoto***", una condizione che non si può verificare in un circuito elettronico, ma contiene un valore "***casuale***" (in realtà contiene l'ultimo valore attribuito a quello spazio da un precedente programma… valore che a noi non è noto)

Variabili non inizializzate

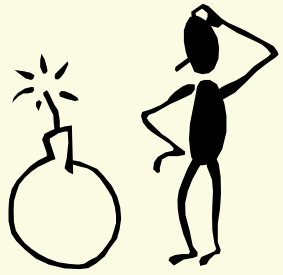


- Se si usasse il valore di una variabile ***prima*** di averle assegnato un qualsiasi valore, il programma si troverebbe a elaborare quel valore che "***casualmente***" si trova nello spazio di memoria riservato alla variabile

ERRORE

```
public class Coins4 // NON FUNZIONA!
{  public static void main(String[] args)
   {  int lit;
      double euro = 2.35;
      double totalEuro = euro + lit / 1936.27;
      System.out.print("Valore totale in euro ");
      System.out.println(totalEuro);
   }
}
```

Variabili non inizializzate



- Questo problema provoca insidiosi errori di esecuzione in molti linguaggi di programmazione
  - il ***compilatore*** Java, invece, segnala come ***errore*** l'***utilizzo*** di variabili a cui non sia mai stato assegnato un valore (mentre non è un errore la sola definizione...)

Coins4.java:5: variable lit might not have been initialized

  - questi errori non sono sintattici, bensì ***logici***, ma vengono comunque individuati dal compilatore, perché si tratta di ***errori semantici*** (cioè di ***comportamento*** del programma) ***individuabili in modo automatico***

Stringhe

Il tipo di dati “stringa”

- I tipi di dati più importanti nella maggior parte dei programmi sono i ***numeri*** e le ***stringhe***
- Una ***stringa*** è una ***sequenza di caratteri***, che in Java (come in molti altri linguaggi) vanno ***racchiusi tra virgolette*** `"Hello"`
  - ***le virgolette non fanno parte della stringa***
- Possiamo ***dichiarare*** e inizializzare ***variabili di tipo stringa***

```
String name = "John";
```

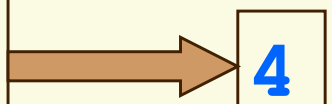
- Possiamo ***assegnare un valore*** a una variabile di tipo stringa

```
name = "Michael"
```

Il tipo di dati "stringa"

- I numeri sono **tipi di dati fondamentali**
- Diversamente dai numeri, le ***stringhe*** sono ***oggetti***
  - infatti, il tipo di dati **String** inizia con la ***maiuscola***!
  - invece, **int** e **double** iniziano con la minuscola…
  - I numeri sono dati fondamentali
- Una ***variabile*** di tipo stringa può quindi essere utilizzata per ***invocare metodi*** definiti nella classe **String (metodi della classe String)**
  - ad esempio, il metodo **length( )** restituisce la ***lunghezza*** di una stringa, cioè ***il numero di caratteri*** presenti in essa (senza contare le virgolette)

```
String name = "John";
int n = name.length();
```



Il tipo di dati "stringa"

- Il metodo **length( )** della classe **String** ***non è un metodo statico***
  - infatti ***per invocarlo usiamo un oggetto della classe String, e non il nome della classe stessa***

```
// NON FUNZIONA!
String s = "John";
int n = String.length(s);
```

```
// FUNZIONA
String s = "John";
int n = s.length();
```

- Una ***stringa di lunghezza zero***, che non contiene caratteri, si chiama ***stringa vuota*** e si indica con due caratteri virgolette ***consecutivi***, senza spazi interposti

```
String empty = "";
System.out.println(empty.length());
```

→ 0

Estrazione di sottostringhe

Attenzione alla minuscola!

- Per estrarre una sottostringa da una stringa si usa il metodo **substring( )**

```
String greeting = "Hello, World!";
String sub = greeting.substring(0, 4);
// sub contiene "Hell"
```

  - il ***primo*** parametro di **substring**( )è la ***posizione del primo carattere*** che si vuole estrarre
  - il ***secondo*** parametro è la ***posizione successiva all'ultimo carattere*** che si vuole estrarre

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

Estrazione di sottostringhe

- La ***posizione*** dei caratteri nelle stringhe viene ***stranamente numerata a partire da 0*** anziché da 1

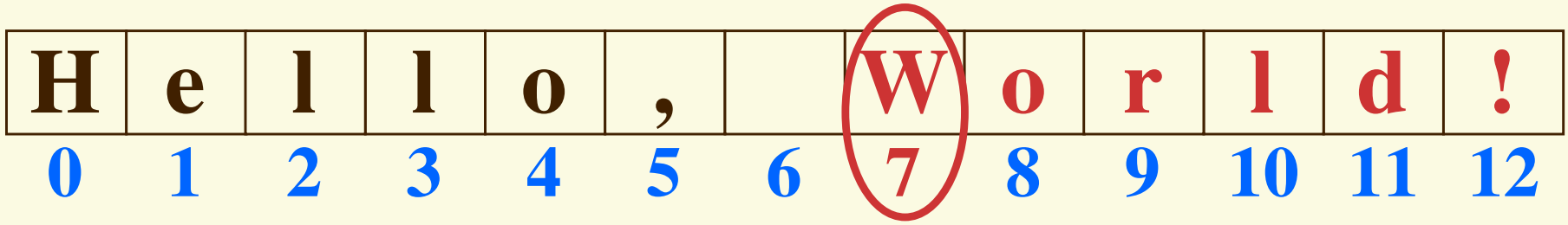
  - in linguaggi precedenti, come il C e il C++, questa era un'esigenza ***tecnica***, mentre in Java non lo è più e si è mantenuta questa strana caratteristica soltanto per ***uniformità*** con tali linguaggi molto diffusi
- Alcune cose da ricordare
  - la posizione dell'ultimo carattere corrisponde alla lunghezza della stringa meno 1
  - la differenza tra i due parametri di **substring()** corrisponde alla lunghezza della sottostringa estratta

Estrazione di sottostringhe

- Il metodo **substring()** può essere anche invocato con ***un solo*** parametro

```
String greeting = "Hello, World!";
String sub = greeting.substring(7);
// sub contiene "World!"
```

- In questo caso il parametro fornito indica la posizione del primo carattere che si vuole estrarre, e l'estrazione continua fino al termine della stringa



Estrazione di sottostringhe

- Cosa succede se si fornisce un ***parametro errato*** a **substring()**?

```
// NON FUNZIONA!
String greeting = "Hello, World!";
String sub = greeting.substring(0, 14);
```

- Il programma viene compilato correttamente, ma viene generato un **errore** in esecuzione

Exception in thread "main"
java.lang.StringIndexOutOfBoundsException
String index out of range: 14

Concatenazione di stringhe

- Per concatenare due stringhe si usa l'***operatore*** **+**

```
String s1 = "li";
String s2 = "re";
String s3 = s1 + s2; // s3 contiene lire
int lit = 15000;
String s = lit + s3; // s contiene "15000lire"
```

- L'operatore di concatenazione è identico all'operatore di addizione
  - se una delle espressioni a sinistra o a destra dell'operatore **+** è una stringa, l'altra espressione viene ***convertita*** in stringa e si effettua la concatenazione

Concatenazione di stringhe

```
int lit = 15000;
String litName = "lire";
String s = lit + litName;
// s contiene "15000lire"
```

- Osserviamo che la concatenazione prodotta non è proprio quella che avremmo voluto, perché ***manca uno spazio*** tra **15000** e **lire**
  - l'operatore di concatenazione ***non aggiunge spazi!*** (***meno male***, diremo la maggior parte delle volte…)
- L'effetto voluto si ottiene così

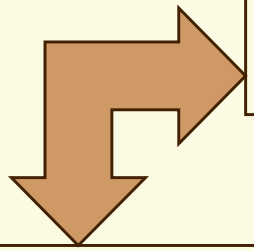
```
String s = lit + " " + litName;
```

Non è una stringa vuota, ma una stringa con un solo carattere, uno spazio (*blank*)

Concatenazione di stringhe

- La concatenazione è molto utile per ridurre il numero di enunciati usati per stampare i risultati dei programmi

```
int total = 10;
System.out.print("Il totale e' ");
System.out.println(total);
```



```
int total = 10;
System.out.println("Il totale e' " + total);
```

- Bisogna fare attenzione a come viene gestito il concetto di "***andare a capo***", cioè alla differenza tra **print()** e **println()**

Alcuni metodi utili di String

- Un problema che capita spesso di affrontare è quello della conversione di una stringa per ottenerne un'altra tutta in maiuscolo o tutta in minuscolo
- La classe **String** mette a disposizione due metodi
  - **toUpperCase()** converte tutto in maiuscolo
  - **toLowerCase**() converte tutto in minuscolo

```
String s = "Hello";
String ss = s.toUpperCase() + s.toLowerCase();
// ss vale "HELLOhello"
```

Alcuni metodi utili di String

```
String s = "Hello";
String ss = s.toUpperCase() + s.toLowerCase();
// s vale ancora "Hello" !
```

- Si noti che l'applicazione di uno di questi metodi alla stringa s ***non altera il contenuto*** della stringa s, ma ***restituisce una nuova stringa***
- In particolare, ***nessun metodo della classe* String *modifica l'oggetto con cui viene invocato!***
  - si dice perciò che gli oggetti della classe **String** sono ***oggetti immutabili***

Esempio di uso di stringhe

- Scriviamo un programma che genera la login per un utente, con la regola seguente
  - si prendono le iniziali del nome e del cognome dell'utente, si rendono minuscole e si concatena il numero di matricola dell'utente espresso numericamente

```
Utente: Marco Dante
Matricola: 565136
⇒ Login: md565136
```

Esempio

```
public class MakeLogin
{
   public static void main(String[] args)
   {
      String firstName = "Marco"; // nome
      String lastName = "Dante";  // cognome
      int matr = 565136;          // matricola
      // estrai le iniziali
      String initials = firstName.substring(0, 1)
         + lastName.substring(0, 1);
      // converti in minuscolo e concatena matr
      String lg = initials.toLowerCase() + matr;
      // stampa la login
      System.out.println("La login e' " + lg);
   }
}
```

Sequenze di “escape”

- Proviamo a stampare una stringa che ***contiene*** delle virgolette

```
Hello, "World"!
```

```
// NON FUNZIONA!
System.out.println("Hello, "World"!");
```

- Il compilatore identifica le seconde virgolette come la fine della prima stringa `"Hello, "`, ma poi non capisce il significato della parola `World`
- Basta inserire una barra rovesciata `\` (*backslash*) ***prima*** delle virgolette ***all’interno*** della stringa

```
System.out.println("Hello, \"World\"!");
```

Sequenze di “escape”

```
// FUNZIONA!
System.out.println("Hello, \"World\"!");
```

- Il carattere *backslash* ‘\’ all’interno di una stringa non rappresenta se stesso, ma si usa per codificare altri caratteri che sarebbe ***difficile*** inserire in una stringa, per vari motivi (***sequenza di escape***)
- Allora, come si fa ad inserire veramente un carattere *backslash* in una stringa?
  - si usa la sequenza di escape `\\`

```
System.out.println("File C:\\autoexec.bat");
```

```
File C:\autoexec.bat
```

Sequenze di “escape”

- Un’altra sequenza di escape che si usa è **\n**, che rappresenta il carattere di “nuova riga” o “andare a capo”

```
System.out.println("*\n**\n***\n");
```

```
System.out.println("*");
System.out.println("**");
System.out.println("***");
```

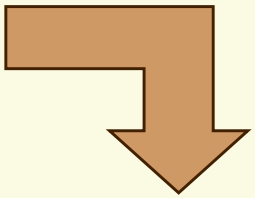
```
*
**
***
```

- Le sequenze di escape si usano anche per inserire caratteri di lingue straniere o simboli che non si trovano sulla tastiera

Sequenze di “escape”

- Ad esempio, per scrivere parole italiane con lettere accentate senza avere a disposizione una tastiera italiana

```
System.out.println("Perch\u00E9?");
```



```
Perché?
```

- Queste sequenze di escape utilizzano la codifica standard **Unicode**

  http://www.unicode.org

Il tipo fondamentale di dati char

Caratteri in una stringa

- Sappiamo già come estrarre sottostringhe da una stringa con il metodo **substring()**
- A volte è necessario estrarre ed elaborare ***sottostringhe*** di dimensioni minime cioè ***di lunghezza unitaria***
  - una stringa di lunghezza unitaria contiene ***un solo carattere*** che ***può essere memorizzato in una variabile di tipo*** **char** anziché in una stringa
  - il tipo **char** in Java è ***un tipo di dato fondamentale*** come i tipi di dati numerici ed il tipo **boolean** cioè ***non è una classe***

Caratteri in una stringa

- La presenza del tipo di dati **char** non è strettamente necessaria in Java (ed è anche per questo motivo che non l'avevamo ancora studiato)
  - infatti ***ogni elaborazione che può essere fatta su variabili di tipo* char *potrebbe essere fatta su stringhe di lunghezza unitaria***
- L'uso del tipo **char** per memorizzare stringhe di lunghezza unitaria è però importante perché
  - una variabile di tipo **char** ***occupa meno spazio in memoria*** di una stringa di lunghezza unitaria
  - le ***elaborazioni*** su variabili di tipo **char** sono ***più veloci***

Caratteri in una stringa

- Il metodo **charAt()** della classe **String** restituisce il singolo carattere che si trova nella posizione indicata dal parametro ricevuto
  - la convenzione sulla numerazione delle posizioni in una stringa è la stessa usata dal metodo **substring**

```
String s = "John";
char c = s.charAt(2); // c contiene 'h'
```

Caratteri in una stringa

- Come si può elaborare un variabile di tipo **char**?
  - la si può ***stampare*** passandola a **System.out.print()**
  - la si può ***concatenare a una stringa*** con l'operatore di concatenazione **+** (verrà convertita in stringa con le stesse regole della conversione dei tipi numerici)
- Una variabile di tipo **char** può anche essere inizializzata con una ***costante di tipo carattere***
  - una costante di tipo carattere è ***un singolo carattere*** racchiuso tra ***singoli*** apici ("apostrofo")

```
char ch = 'x';
```

- Il singolo carattere può anche essere una "sequenza di escape"

```
char ch = '\u00E9'; // carattere 'é'
char nl = '\n'; // carattere per andare a capo
```

Caratteri in una stringa

- Java gestisce correttamente i caratteri dello standard Unicode
- La maggior parte dei sistemi operativi non li gestisce correttamente. Questo si riflette nel seguente fenomeno: se un programma Java visualizza una stringa che contiene un carattere che non fa parte del codice ASCII (sottoinsieme dei primi 128 caratteri dello standard Unicode), l'interazione dello standard output di Java con il sistema operativo provoca la visualizzazione di caratteri strani e non del carattere corretto.
- Il fenomeno è presente in alcune versioni (anche la più recente) del JDK, anche se non in tutti i sistemi operativi.
- Verificare tale fenomeno sul proprio sistema, provando, ad esempio, ad eseguire il seguente programma:

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

   { System.out.println(" èèèèèèè ");
   }
}
```

- Per evitare questo problema, si consiglia di non usare lettere accentate nei messaggi visualizzati dai programmi (usare, in alternativa, l'apostrofo).

Lezione IX
Ve 12 Ott. 2007

Introduzione a Classi e Oggetti

Motivazioni

- Elaborando numeri e stringhe si possono scrivere programmi interessanti ma ***programmi più utili*** hanno bisogno di manipolare ***dati più complessi***
  - conti bancari, dati anagrafici, forme grafiche…
- Il linguaggio Java gestisce questi dati complessi sotto forma di ***oggetti***
- Gli ***oggetti*** e il loro ***comportamento*** vengono descritti mediante le ***classi*** e i loro ***metodi***

Oggetti

- Un ***oggetto*** è un'entità che può essere manipolata in un programma mediante l'invocazione di ***metodi***
  - **System.out** è un oggetto che si può manipolare (usare) mediante il suo metodo **println()**
- Per il momento consideriamo che un oggetto sia una "scatola nera" (***black box***) dotata di
  - un'***interfaccia pubblica*** (i metodi che si possono usare) che definisce il comportamento dell'oggetto
  - una ***realizzazione*** (*implementazione*) ***nascosta (privata)*** (il codice dei metodi e i loro dati)

Classi

Finora abbiamo visto solo questo aspetto che è forse quello meno importante

- Una ***classe***
  - è una ***fabbrica di oggetti***
    - gli oggetti che si creano sono ***esemplari*** ("*istanze*" *instance*) di una classe che ne è il prototipo
    - specifica i metodi che si possono invocare per gli oggetti che sono esemplari di tale classe (l'interfaccia pubblica)
  - definisce i particolari della realizzazione dei metodi (codice e dati)
  - è anche un ***contenitore*** di
    - metodi statici
      - **Hello** contiene il metodo **main()**
      - **Java.lang.Math** contiene i metodi **pow(), exp(), sin()…**
    - oggetti statici (**System** contiene **out** e **in**)

Usare una classe

- Iniziamo lo studio delle classi analizzando come si ***usano*** oggetti di una classe che si suppone già definita da altri
  - vedremo quindi che è possibile ***usare oggetti di cui non si conoscono i dettagli realizzativi*** un concetto molto importante della programmazione orientata agli oggetti come abbiamo già visto con oggetti di tipo **String**
- In seguito analizzeremo i ***dettagli realizzativi*** della classe che abbiamo imparato a utilizzare
  - **usare oggetti di una classe**
  - **realizzare una classe**

Sono due attività *ben distinte*!

Metodi di esemplare

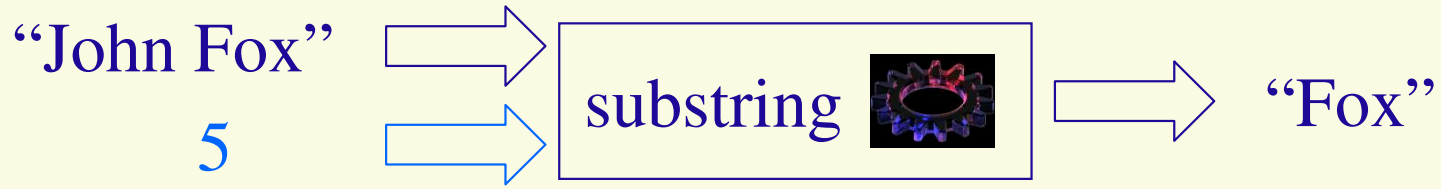
Invocazione di metodi di esemplare

- I metodi si suddividono in metodi ***statici*** e ***di esemplare*** (non-statici)

parametro implicito

```
String name = "John Fox";
String surname  = name.substring(5);
```

- nell'esempio, la variabile ***name*** si dice ***parametro implicito*** dell'invocazione al metodo di esemplare substring()



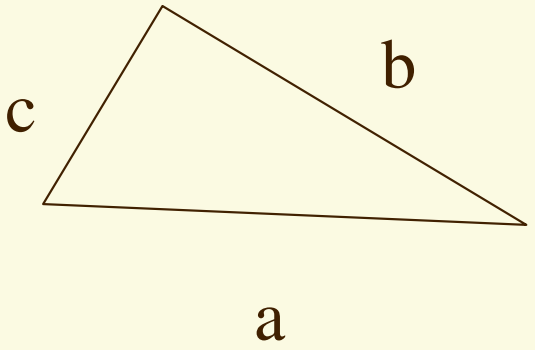
- Il numero **5** e' invece il ***parametro esplicito***

Sovraccarico del nome di un metodo

- Il ***nome di un metodo*** si dice ***sovraccarico*** (***overloaded***) se nella classe sono definiti più metodi con lo stesso nome
- Ad esempio nella classe ***java.io.PrintStream*** sono definiti molti metodi di nome println(); ad esempio
  - **`println(int n)`**
  - **`println(double a)`**
- Le firme dei metodi sovraccarichi di una classe devono differire per il numero e/o il tipo dei ***parametri espliciti***
  - Nell'esempio precedente le firme differiscono per il tipo del parametro esplicito (**`int n`** – **`double a`**).
- Il nome del metodo substring della classe java.lang.String è sovraccarico
  - Le firme dei due metodi differiscono per il numero di parametri espliciti

Costruire e usare oggetti

- I dati eleborati da un programma possono essere tipi fondamentali, stringhe od oggetti più o meno complessi.
- Consideriamo un triangolo nel piano. Un triangolo è caratterizzato da tre dati:
  - ad esempio la lunghezza dei tre lati
- Supponiamo di avere a disposizione una ***classe Triangolo*** (ad esempio scritta dal vostro docente).



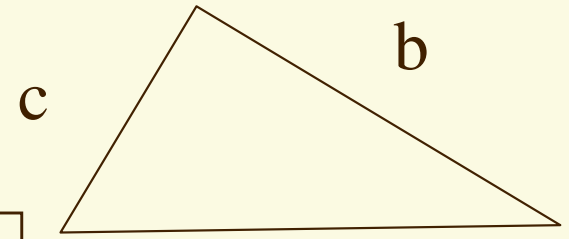
- Per usare una classe non è necessario avere a disposizione il codice sorgente, ma è sufficiente avere il ***bytecode*** e la ***documentazione dell'interfaccia pubblica.***
- L'interfaccia pubblica fornisce informazioni sulla classe
  - es.: documentazione dei metodi pubblici della classe

Costruire e usare oggetti

- Scriviamo un frammento di codice che operi con oggetti di tipo ***Triangolo***, ad esempio che ***definisca*** e ***inizializzi*** un oggetto di tipo triangolo e ne calcoli alcuni parametri, quali perimetro e area.

```
Triangolo t1 = new Triangolo(3, 4, 5);
System.out.println("perimetro = " + t1.perimetro());
System.out.println("area = " + t1.area());
```

- Analizziamo l'enunciato di definizione e inizializzazione



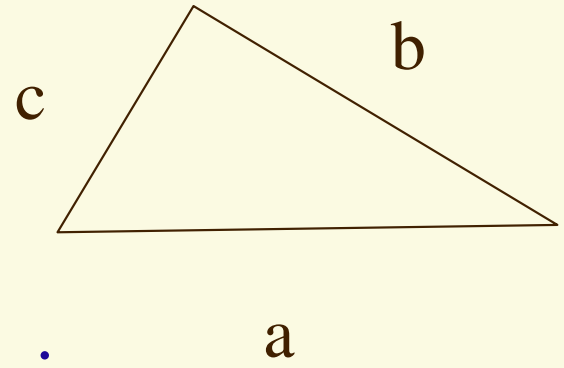

```
Triangolo t1 = new Triangolo(3, 4, 5);
```

definizione e inizializzazione

Costruire e usare oggetti

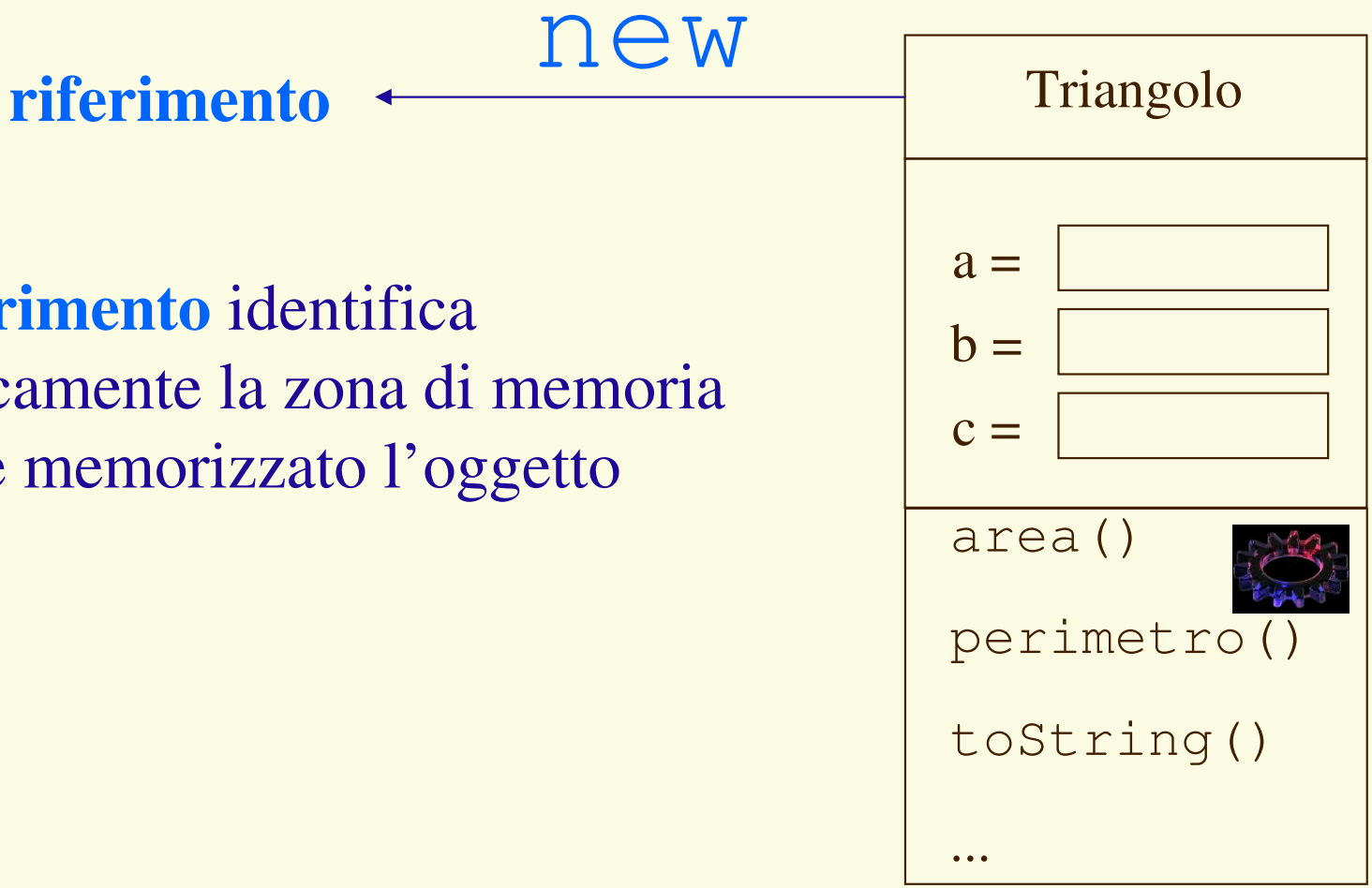
- La parte a destra del simbolo di assegnazione "crea" e inizializza l'oggetto.

```
... = new Triangolo(3, 4, 5);
```



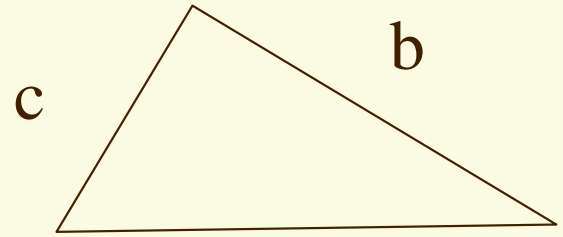
- **new** è una ***parola chiave*** del linguaggio.
- L'operatore **new** crea l'oggetto, ovvero
  - riserva nella memoria una zona di dimensione adeguata per memorizzare le informazioni relative all'oggetto, ad esempio
    - i dati associati all'oggetto, come la lunghezza dei lati nel caso del Triangolo
    - le informazione relative ai metodi definiti per l'oggetto
  - restituisce ***il riferimento*** all'oggetto creato

Costruire e usare oggetti



Il **riferimento** identifica univocamente la zona di memoria dove è memorizzato l'oggetto

Costruire e usare oggetti

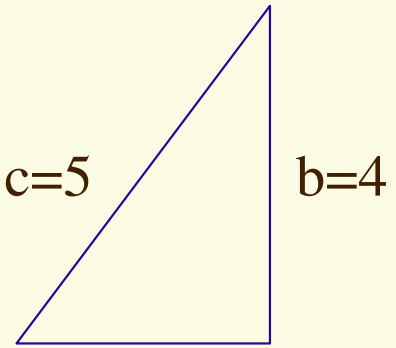


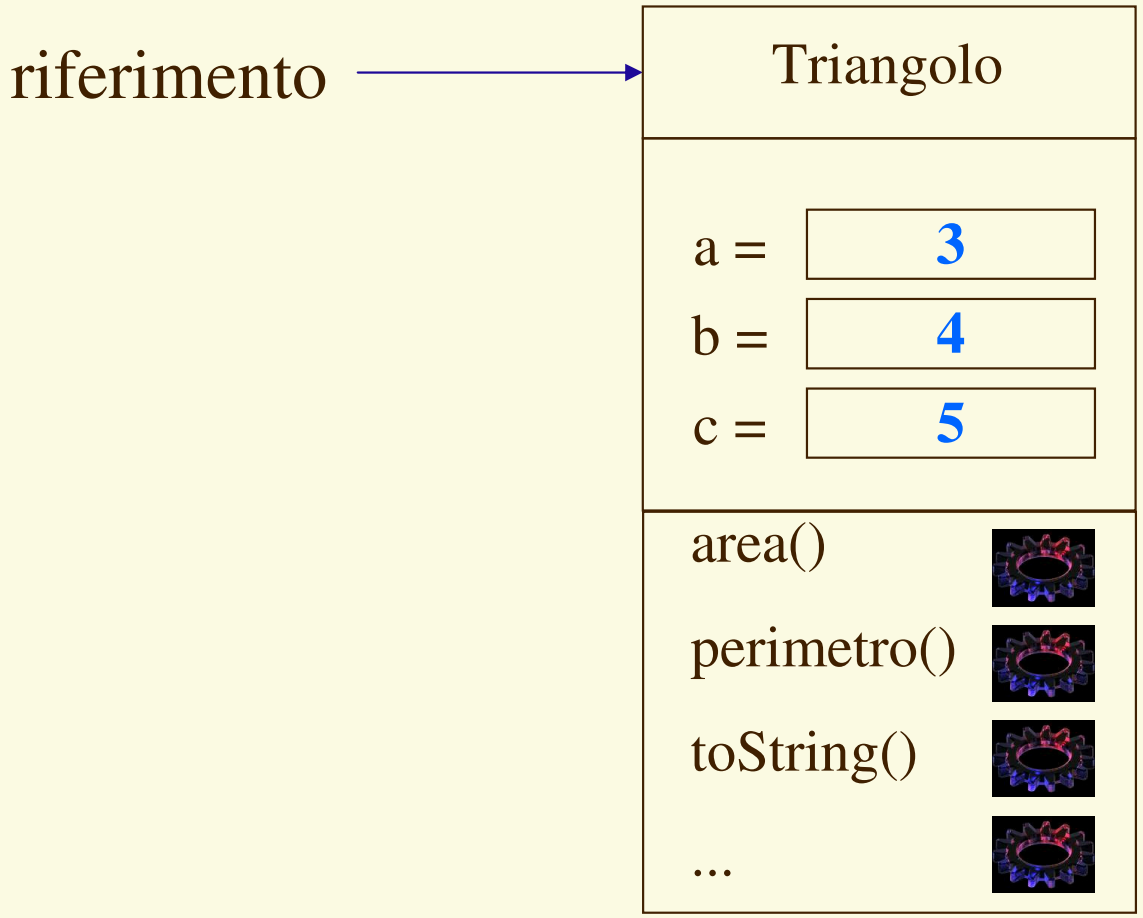

```
... = new Triangolo(3, 4, 5);
```

- `Triangolo(3, 4, 5)` rappresenta l'invocazione del ***costruttore*** della classe Triangolo
- Lo scopo del costruttore è di ***inizializzare*** quei campi che contengono i dati associati all'oggetto, ad esempio nel caso dei triangoli la lunghezze dei lati
- Il costruttore è un metodo ***speciale***
- Ha sempre lo ***stesso nome*** della classe, quindi il suo nome inizia con la ***maiuscola***, nella convenzione della libreria standard
- Una classe può avere uno o più costruttori; se ha molteplici costruttori, il nome del costruttore è sovraccarico

Costruire e usare oggetti

```
... = new Triangolo(3, 4, 5);
```

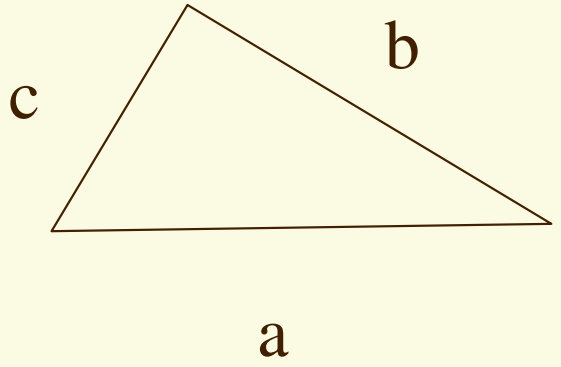




Costruire e usare oggetti

- La parte a sinistra del simbolo di assegnazione definisce una ***variabile riferimento*** a un ***oggetto*** di tipo Triangolo

```
Triangolo t1 = ...
```

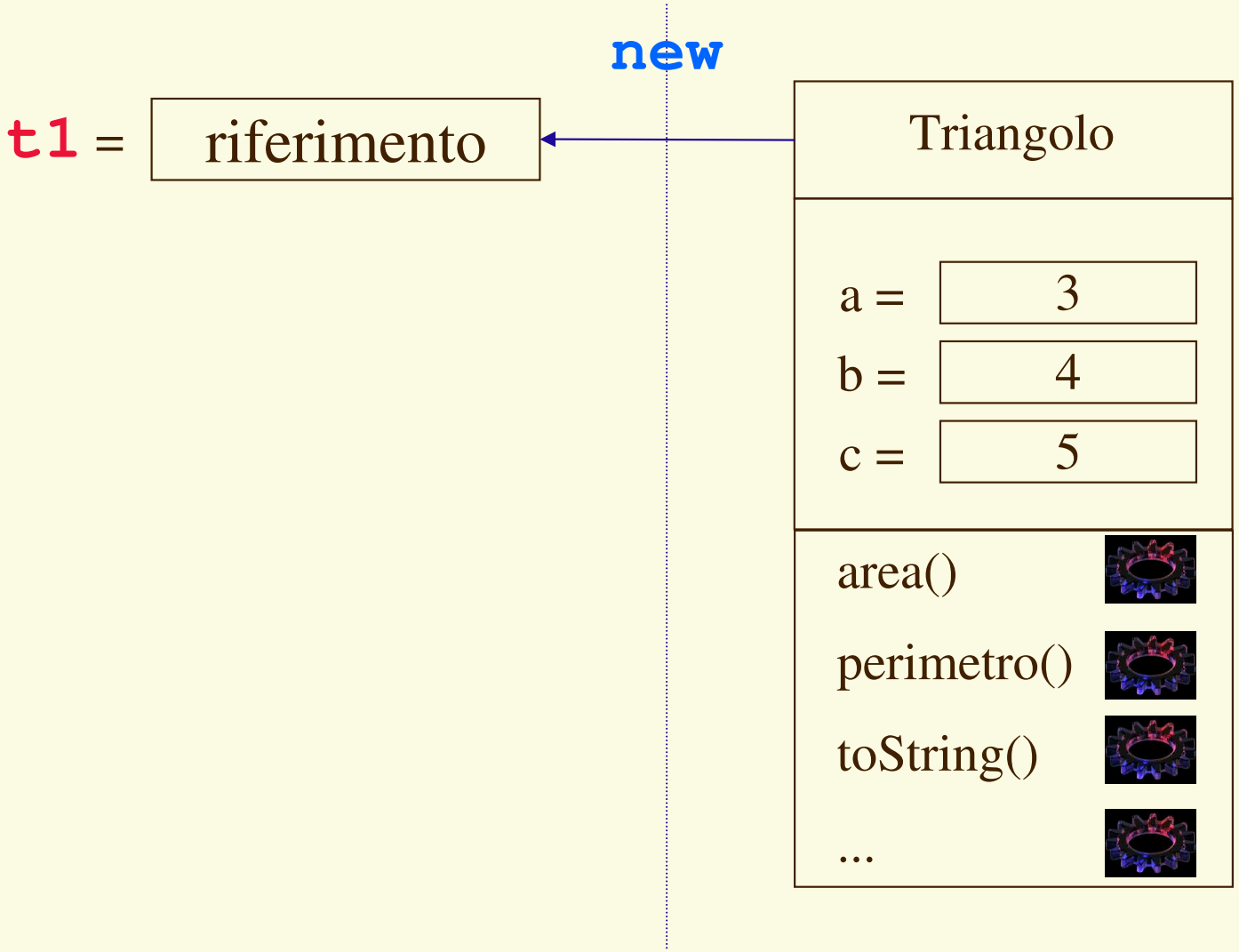


- Il ***tipo*** della variabile è definito con il nome della classe: **`Triangolo`**

- La ***variabile riferimento*** memorizza il riferimento all'oggetto

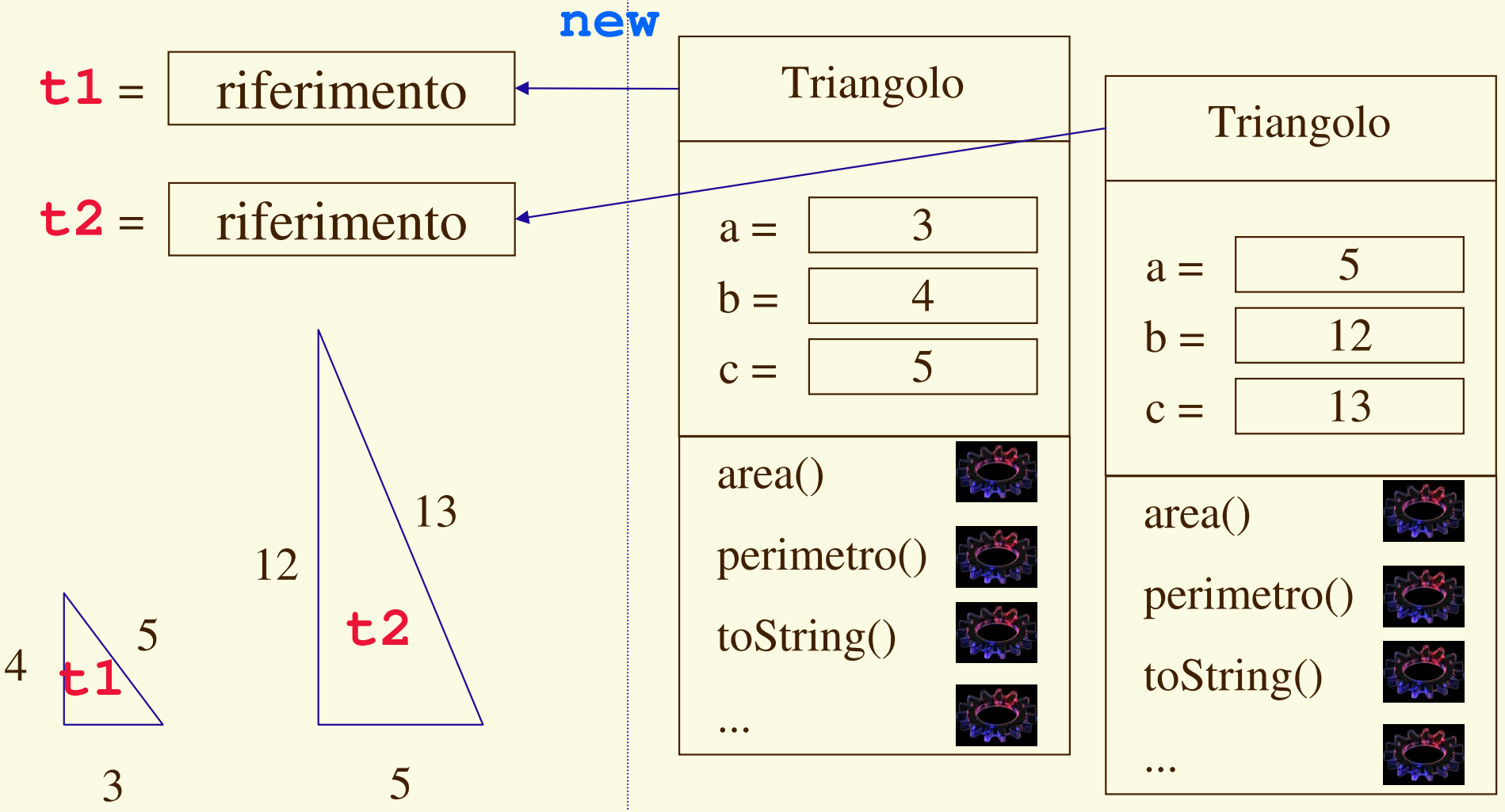
Costruire e usare oggetti

```
Triangolo t1 = new Triangolo(3, 4, 5);
```



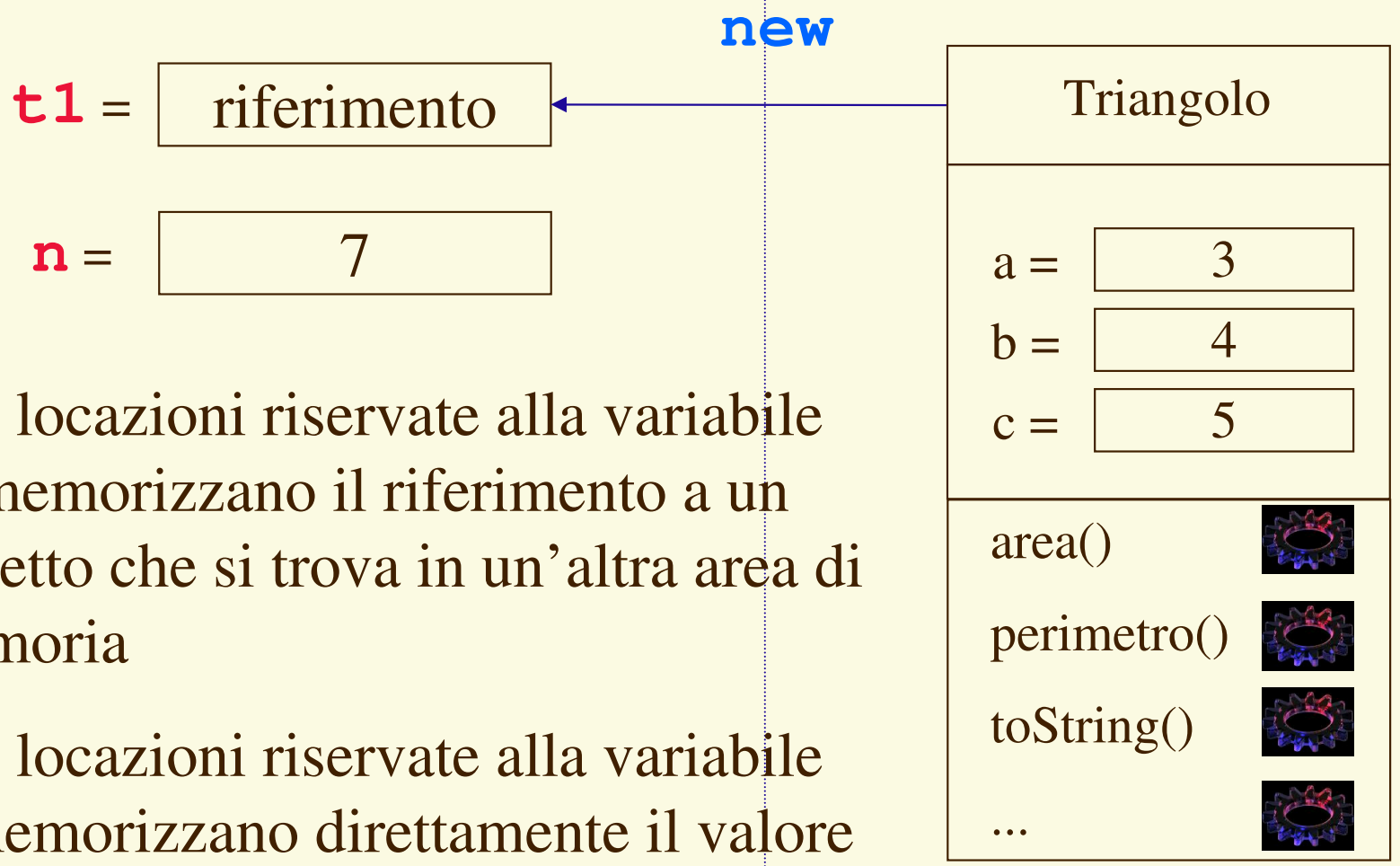
Costruire e usare oggetti

```
Triangolo t1 = new Triangolo(3, 4, 5);
Triangolo t2 = new Triangolo(5, 12, 13);
```



Costruire e usare oggetti

```
Triangolo t1 = new Triangolo(3, 4, 5);
int n = 7;
```



- Le locazioni riservate alla variabile **t1** memorizzano il riferimento a un oggetto che si trova in un'altra area di memoria
- Le locazioni riservate alla variabile **n** memorizzano direttamente il valore assegnato alla variabile

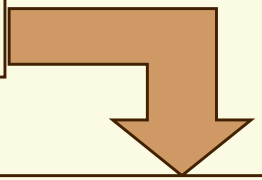
Costruire e usare oggetti

- Possiamo ora invocare i metodi della classe triangolo su un oggetto della classe.

```
Triangolo t1 = new Triangolo(3, 4, 5);
System.out.println("perimetro = " + t1.perimetro());
System.out.println("area = " + t1.area());
```

- Per usare i metodi della classe si scrive, ad esempio

```
int p = t1.perimetro();
double a = t1.area();
String s = t1.toString();
System.out.println(t1.toString());
```



```
triangolo di lati 3, 4, 5
```

Metodi di accesso e modificatori

- I **metodi di esemplare** possono essere divisi in due categorie
  - metodi ***di accesso*** e metodi ***modificatori***
- Metodi di accesso: non modificano i dati dell'oggetto su cui sono applicati
  - esempio

```
String s = "John";
int n = s.length();
```

  - Il metodo length() restituisce un'informazione relativa all'oggetto riferito dalla variabile riferimento s, ma non ne modifica i dati
  - Gli oggetti della classe String sono immutabili perchè non ha metodi modificatori!
- Metodi modificatori: modificano i dati relativi all'oggetto su cui vengono applicati

Costruttore della classe java.io.String

- La definizione di variabili riferimento a stringhe e la loro inizializzazione con stringhe costanti è un'operazione molto comune
- Il linguaggio supporta il costrutto speciale:

```
String s = "John";
```

- È equivalente al seguente, che può essere usato in alternativa:

```
String s = new String("John");
```

Descrizione testuale di un oggetto

- L'invio a standard output della descrizione testuale di un oggetto, ***se è definito il metodo toString()***, può essere programmata come nel seguente esempio:

```
Triangolo abc = new Triangolo(3, 4, 5);
System.out.println(abc.toString());
```

```
triangolo di lati 3, 4, 5
```

- Ma anche, in alternativa, nel seguente modo:

```
Triangolo abc = new Triangolo(3, 4, 5);
System.out.println(abc);
```

```
triangolo di lati 3, 4, 5
```

- Comodo, ma misterioso! Più avanti la spiegazione.
- Funziona solo se è stato definito il metodo toString() nella classe che definisce l'oggetto!

Ricevere Dati in Ingresso

I dati in ingresso ai programmi

- I programmi visti finora non sono molto utili, visto che eseguono ***sempre la stessa elaborazione a ogni esecuzione***
- La classe **MakeLogin** genera sempre la login **md525136**
  - se si vuole che generi la login per un altro utente è necessario modificare il codice sorgente (in particolare, le inizializzazioni delle variabili) e compilarlo di nuovo!
- I programmi utili hanno bisogno di ***ricevere dati in ingresso*** dall'utente

javax.swing.JOptionPane

- La classe JOptionPane contenuta nel pacchetto javax.swing fornisce un comodo strumento: il metodo statico showInputDialog( )
- Visualizza un ***dialog box*** in cui e' contenuta una stringa e un campo in cui può essere inserita una stringa
- Restituisce una stringa
- Alla fine bisogna usare il metodo ***System.exit()*** per terminare il programma

javax.swing.JOptionPane

```
import javax.swing.JOptionPane;
public class Echo
{  public static void main(String[] args)
   {
      // acquisisce una stringa da dialog box
      String s = JOptionPane.showInputDialog
               ("inserisci una stringa");

      // invia la stringa a message box
      JOptionPane.showMessageDialog(null, s);

      // invia la stringa a standard output
      System.out.println(s);

      // termina l'esecuzione della JVM
      System.exit(0);
   }
}
```

javax.swing.JOptionPane

- ***import***: parola chiave del linguaggio java. Serve a importare una o più classi da un pacchetto di libreria
- ***null***: costante nulla.
- ***JOptionPane.showInputDialog()***: visualizza, in una finestra (dialog box), una stringa e un campo di ingresso in cui è possibile inserire una stringa. Restituisce la stringa introdotta nel campo apposito della finestra.

```
JOptionPane.showInputDialog ("inserisci una stringa");
```

javax.swing.JOptionPane

- **showMessageDialog():** visualizza una stringa in una finestra (***message box***).

```
JOptionPane.showMessageDialog(null, "Hello, World!");
```

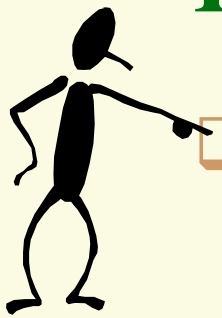
- **System.exit(0):** metodo statico della classe System che termina l'esecuzione della JVM
  - il parametro serve come codice di stato: per convenzione un valore non nullo indica una terminazione anomala

I pacchetti di classi

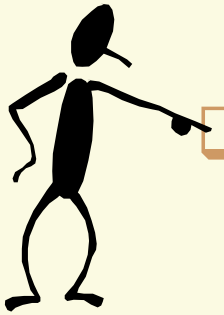
I pacchetti di classi (*package*)

- Tutte le classi della libreria standard sono raccolte in ***pacchetti*** (***package***) e sono organizzate per argomento e/o per finalità
  - la classe **JOptionPane** appartiene al pacchetto **javax.swing**
- Per ***usare*** una classe di una libreria bisogna ***importarla*** nel programma usando l'enunciato
  - **import** ***nomePacchetto.NomeClasse;***
- Le classi **System** e **String** appartengono al pacchetto **java.lang**
  - il pacchetto **java.lang** viene ***importato automaticamente***

Importare classi da un pacchetto



- Sintassi: `import nomePacchetto.NomeClasse;`
  - Scopo: importare una classe da un pacchetto per poterla utilizzare in un programma



- Sintassi: `import nomePacchetto.*;`
  - Scopo: importare tutte le classi di un pacchetto per poterle utilizzare in un programma

- Nota: le classi del pacchetto **java.lang** non hanno bisogno di essere importate
- Attenzione: non si possono importare ***più pacchetti*** con un solo enunciato

```
import java.*.*; // ERRORE
```

- Gli enunciati di importazione vanno inseriti prima della definizione della classe

Stili per l'importazione di classi

- Usare un enunciato **import** per ogni classe importata

```
import javax.swing.JOptionPane;
import javax.swing.JRadioButton;
```

- Usare un enunciato **import** che importa ***tutte le classi di un pacchetto***

```
import javax.swing.*;
```

  - non è un errore importare classi che non si usano!
  - se si usano più enunciati di questo tipo non è più chiaro il pacchetto di appartenenza di ciascuna classe

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

Se adesso usiamo la classe **File** a **quale pacchetto** appartiene?

  - sapere a quale pacchetto appartiene una classe è utile ad esempio per reperire la documentazione della sua interfaccia pubblica

Stili per l'importazione di classi

- Si può non usare per nulla gli enunciati **import** ed indicare sempre il ***nome completo*** delle classi utilizzate nel codice

```
javax.swing.JOptionPane.showMessageDialog(null, s);
```

- Questo stile è ***assai poco usato*** perché è molto noioso aumenta la probabilità di errori di battitura e aumenta la lunghezza delle linee del codice (diminuendo così la leggibilità del programma)

Standard input dei programmi

- Un modo semplice e immediato per fornire dati in ingresso a un programma consiste nell'***utilizzo della tastiera***
  - altri metodi fanno uso del mouse, del microfono…

- Abbiamo visto che tutti i programmi Java hanno accesso al proprio ***Standard Output***, tramite l'oggetto ***System.out*** di tipo ***java.io.PrintStream*** **definito nella classe java.lang.System**

- Analogamente, l'interprete Java mette a disposizione dei programmi in esecuzione il proprio ***Standard Input*** (***flusso di ingresso***), tramite l'oggetto ***System.in*** di tipo ***java.io.InputStream*** **definito nella classe java.lang.System**

Classe java.io.InputStream

- Lo Standard Input è tipicamente la tastiera
- La classe InputStream mette a disposizione il metodo
  - read ( ) che legge un byte alla volta
  - non molto comodo!
- Leggere da input standard richiede nozioni di Java che ancora non abbiamo

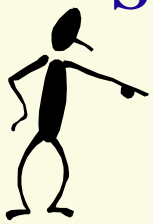
La classe Scanner (jdk5.0!)

- Sfortunatamente, la classe **InputStream** ***non possiede metodi comodi*** per la ricezione di dati numerici e stringhe
  - **PrintStream** ha invece i ***comodissimi*** metodi **print( ) e println()**
- La libreria standard di ***JDK5.0*** mette a disposizione la comoda classe ***java.util.Scanner***
  - ***Attenzione: non c'è in JDK 2.0: Horstmann ed. seconda***
- Lo scopo della classe **Scanner** è quello di fornire una comoda interfaccia all'oggetto **System.in**

Acquisire dati da Standard Input con la classe java.util.Scanner

Costruire un oggetto di classe Scanner

- Prima di tutto bisogna ***creare*** un nuovo ***oggetto*** di classe **Scanner**



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

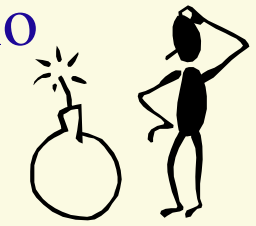
- Analizziamo l'espressione a destra dell'operatore di assegnazione nell'enunciato precedente
- Il parametro del costruttore è un oggetto di classe java.io.InputStream
- ***System.in*** è un oggetto di classe ***java.io.InputStream***
- Scanner definisce 8 costruttori
  - In futuro useremo anche altri costruttori

Acquisire Numeri Interi da Standard Input

- Ora che abbiamo definito un oggetto di classe Scanner possiamo acquisire dati dallo Standard Input
- Come si fa se si vogliono acquisire dei valori numerici?
  - numero intero:si usa il **`int nextInt()`**

```
Scanner in = new Scanner(System.in);
System.out.println("Introduci un numero intero :")
int n = in.nextInt();
System.out.print("Introdotto : " + n);
```

  - Quandi viene invocato il metodo **`nextInt()`** l'esecuzione del programma si ferma attendendo l'introduzione di un dato da tastiera
  - Per questo prima di invocarlo, si manda un messaggio all'utente per avvisarlo di introdurre un numero
  - E se il dato introdotto non e' un numero intero?



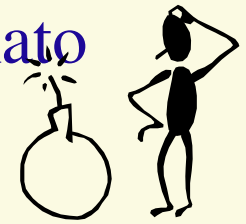
Errore in esecuzione: **java.util.InputMismathException**

Acquisire Numeri in Virgola Mobile da Standard Input

- Con il metodo **`double nextDouble()`** si possono acquisire numeri in virgola mobile

```
…
System.out.println("Introduci un numero frazionario :")
double x = in.nextDouble();
System.out.print("Introdotto : " + x);
```

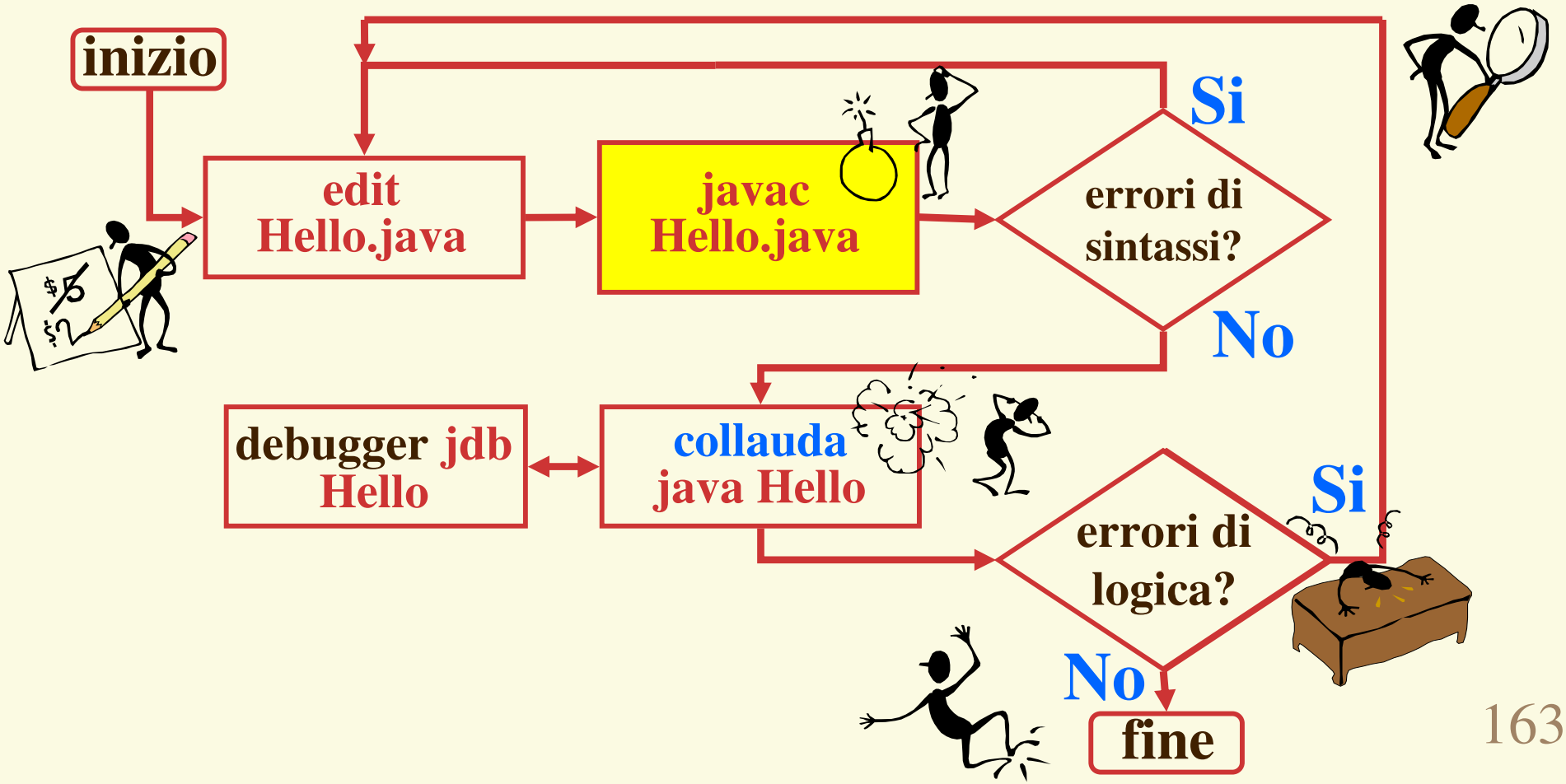
- Anche in questo caso, se il dato non e' un numero in formato virgola mobile l'interprete java genera l'errore



java.util.InputMismathException

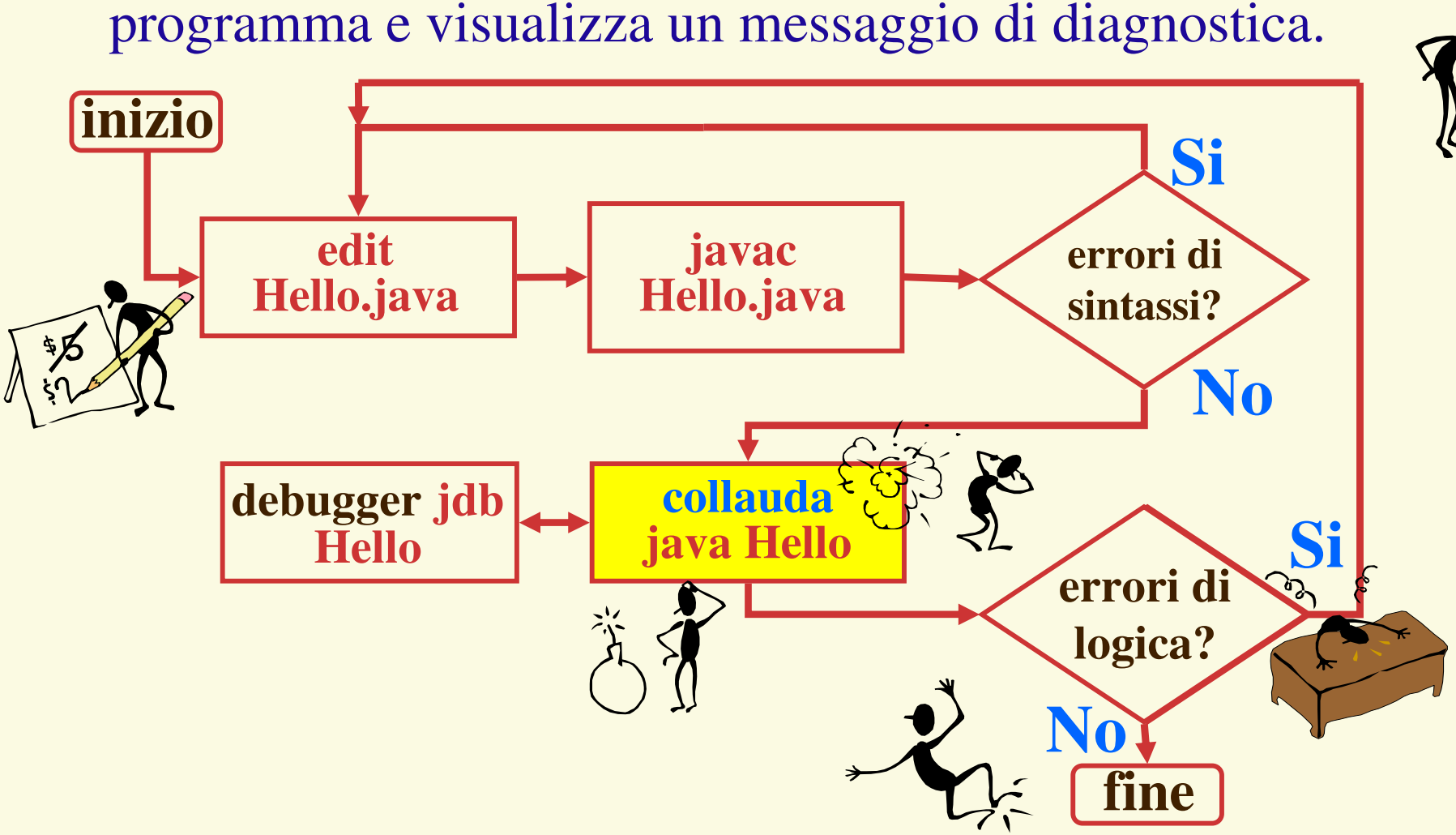
Errori di compilazione

- Errori di compilazione: vengono rilevati dal compilatore (comando ***javac***). In presenza di un errore, il compilatore interrompe il processo di compilazione e visualizza uno o più messaggi d'errore



Errori run-time (Eccezioni)

- Eccezioni: vengono generate dall'interprete (comando ***java***) durante l'esecuzione del programma. In presenza di una eccezione errore, l'interprete interrompe l'esecuzione del programma e visualizza un messaggio di diagnostica.



Acquisire Stringhe da Standard Input

- Come si fa se si vogliono acquisire delle stringhe?
  - parola (stringa delimitata dai caratteri **SP**, **'\t'**, **'\n'**, **'\r'**):
    - metodo **`String next()`**

```
…
System.out.print("Introduci una parola: ");
String s = in.next();
System.out.println("Introdotta la parola: " + s);
```

  - riga (stringa delimitata dai caratteri '\n' o '\r'):
    - metodo String nextLine()

```
…
System.out.print("Introduci una o piu' parole: ");
String s = in.nextLine();
System.out.println("Introdotta la riga: " + s);
```

La classe Scanner

- Dato che la classe **Scanner** non fa parte del pacchetto java.lang, ma del pacchetto **java.util**, è necessario importare la classe prima di usarla

- Quando non si usa piu' l'oggetto di classe Scanner e' bene chiuderlo
  - **in.close();**

Esempio

```
/* produce la login a partire dai dati nome,
   cognome e numero di matricola di uno studente
*/
import java.util.Scanner;
public class MakeLogin2
{ public static void main(String[] args)
  { Scanner in = new Scanner(System.in);
    System.out.print("Inserire il nome: ");
    String nome = in.next();
    System.out.print("Inserire il cognome: ");
    String cognome = in.next();
    System.out.print("Inserire la matricola: ");
    int matr = in.nextInt();
    String inits = nome.substring(0,1) +
                   cognome.substring(0,1);
    String lg = inits.toLowerCase() + matr;
    System.out.println("La login e' " + lg);
    in.close();
  }
}
```

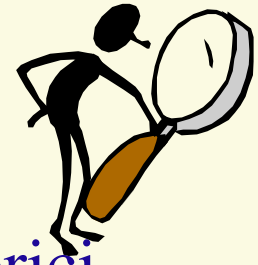
Esempio

```
/* produce la login a partire dai dati nome,
   cognome e numero di matricola di uno studente
*/
import java.util.Scanner;
public class MakeLogin3
{ public static void main(String[] args)
  { Scanner in = new Scanner(System.in);
    System.out.print("Inserire nella stessa riga ");
    System.out.print("nome cognome e matr");
    String nome = in.next();
    String cognome = in.next();
    int matr = in.nextInt();
    String inits = nome.substring(0,1) +
                   cognome.substring(0,1);
    String lg = inits.toLowerCase() + matr;
    System.out.println("La login e' " + lg);
    in.close();
  }
}
```

Esempio

```
/* produce la login a partire dai dati nome,
   cognome e numero di matricola di uno studente
*/
import javax.swing.JOptionPane;
import java.util.Scanner;
public class MakeLogin4
{ public static void main(String[] args)
  { String s = JOptionPane.showInputDialog
                  ("Inserire nome cognome e matricola");
    Scanner in = new Scanner(s);
    String nome = in.next();
    String cognome = in.next();
    int matr = in.nextInt();
    String inits = nome.substring(0,1) +
              cognome.substring(0,1);
    String lg = inits.toLowerCase() + matr;
    JOptionPane.showMessageDialog
      (null, "La login e' " + lg);
    in.close();
    System.exit(0);
  }
}
```

Convertire stringhe in numeri



- Convertire una stringa che contiene solo caratteri numerici decimali (e' lecito anche il segno – prefisso) in un numero intero

```
String matr = "543210";
int n = Integer.parseInt(matr);
n++;
System.out.println(n);
```

→ `543211`

- e se la stringa non rappresenta un numero intero?

```
String matr = "543b10";
int n = Integer.parseInt(matr);
```

→ `NumberFormatException`

in esecuzione l'interprete genera un'eccezione

Convertire stringhe in numeri

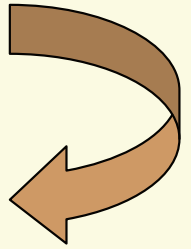
- Convertire una stringa che rappresenta un numero a virgola mobile in un numero in virgola mobile

```
String euroStr = "2.15";
double  euro = Double.parseDouble(euroStr);
euro += 10.0;
System.out.println(euro);
```

```
12.15
```

- e se la stringa non rappresenta un numero?

```
String euroStr = "2,15"; // la virgola!
double euro = Double.parseDouble(euroStr);
```



```
NumberFormatException
```

E Convertire numeri in stringhe?

- Si puo' usare l'operatore di concatenazione; funziona anche con i numeri:

```
double euro = 2.15;
String euroStr = euro + "";  // "" = stringa vuota!
```

```
int matr = 543210;
String matrStr = matr + "";  // stringa vuota!
```

- In alternativa

```
double euro = 2.15;
String euroStr = Double.toString(euro);
```

```
int matr = 543210;
String matrStr = Integer.toString(matr);
```

Formattazione di Numeri

- Talvolta si desidera stampare numeri con formati particolari, ad esempio
  - intero decimale (standard)
  - esadecimale
  - virgola mobile con un numero di cifre e precisione prestabilite
  - virgola mobile con notazione esponenziale
- Il metodo ***printf()*** della classe ***java.io.PrintStream*** ci permette di fare questo
- Il primo parametro del metodo printf() e' una stringa di formato che contiene caratteri da stampare e caratteri specificatori del formato

```
double total = 1.1234567;
System.out.printf("Totale = %5.2f", total);
```

Formattazione di Numeri

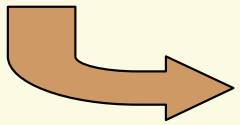
```
double total = 12.345678;
System.out.printf("Totale = %5.2f", total);
```

```
12.35
```

- **%5.2f** e' uno specificatore di formato che significa: numero in virgola mobile (%f) formato da 5 caratteri (compreso il punto!) con due cifre dopo la virgola
- Questo formato e' applicato alla variabile total, che e' il secondo parametro del metodo

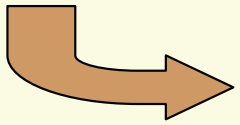
Formattazione di Numeri

```
int address = 0x0000000f; //numero in formato esadecimale
double value = 10.123456789;
System.out.println(address + " -> " + value);
```



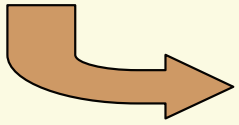
```
15 -> 10.123456789
```

```
System.out.printf("%x -> %7.4f %n", address, value);
```



```
f -> 10.1235
```

```
System.out.printf("%x -> %e %n", address, value);
```



```
f -> 1.012346e+01
```

- `%x` e' uno specificatore di formato che significa: numero intero in notazione esadecimale
- `%e` e' uno specificatore di formato che significa: numero in virgola mobile (float) con notazione esponenziale
- `%n` e' uno specificatore di formato che significa a capo

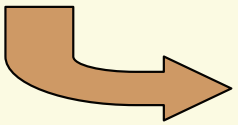
Formattazione di Numeri

- Nella classe **java.lang.String** esiste anche il metodo statico **format**() che restituisce una stringa formattata;
- Nel metodo si usano gli specificatori di formato gia' visti

```
int address = 0x0000000f; //numero in formato esadecimale
double value = 10.123456789;

String s = String.format("%x -> %7.4f %n", address,
                value);

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



```
f -> 10.1235
```

Programma proposto

- Leggere da input standard un numero intero positivo avente al massimo 5 cifre e visualizzare sull'output standard le singole cifre del numero separandole con uno spazio
- Se il numero ha meno di 5 cifre, si introducano zeri a sinistra; ad esempio:
  - il numero 12345 viene visualizzato come 1 2 3 4 5
  - il numero 123 come 0 0 1 2 3