SETTIMANA 2

Rappresentazione dei numeri (appendice H)

Numeri binari

- I numeri che siamo abituati ad utilizzare sono espressi con la notazione ***posizionale*** in ***base decimale***
  - ***base decimale*** perché usiamo ***dieci cifre diverse*** (da 0 a 9)
  - notazione ***posizionale*** perché ***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

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

Numeri binari

- 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***

- La parte frazionaria, a destra del simbolo separatore, si valuta con potenze ***negative***

$$4,34 = 4 \cdot 10^{0} + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}$$

Numeri binari

- 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

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

$$(1,101)_2 = (1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3})_{10} = (1,625)_{10}$$

- I numeri binari sono più facili da manipolare per i computer, perché è meno complicato costruire circuiti logici che distinguono tra “acceso” e “spento”, piuttosto che fra ***dieci livelli diversi*** di voltaggio

Numeri binari

- 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 binari

- 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*** de divisioni, ***iniziando dall'ultim*** 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
```

$(100)_{10} = (1100100)_2$

- **ESERCIZIO:** Come si dimostra?

Numeri binari

- 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

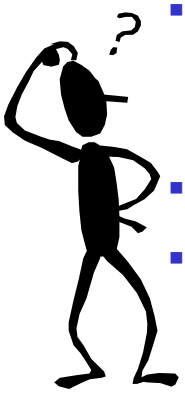
- Il numero binario si ottiene scrivendo ***la serie delle parti intere dei prodotti*** ottenuti, ***iniziando dal primo***
- Se si ottiene un risultato già ottenuto in precedenza, il numero sarà ***periodico***, anche se ***non lo era*** in base decimale

$0,35 \cdot 2 = 0,7$
$0,7 \cdot 2 = 1,4$
$0,4 \cdot 2 = 0,8$
$0,8 \cdot 2 = 1,6$
$0,6 \cdot 2 = 1,2$
$0,2 \cdot 2 = 0,4$

$(0,35)_{10} = (0,0\overline{10110})_2$

Numeri binari

- Per programmare non c'è bisogno di conoscere i numeri binari, ma tale conoscenza fa parte della cultura di base dell'informatica
- Perché il massimo numero rappresentabile in Java con una variabile di tipo **int** è, come vedremo, **2147483647**?



- una variabile di tipo **int** occupa uno spazio in memoria pari a **32 *bit***, cioè il suo valore viene rappresentato con 32 cifre binarie
- il primo bit rappresenta il segno (0 ⇒ positivo, 1 ⇒ negativo)
- il massimo numero è

  01111111111111111111111111111111 che equivale a $2^{31}-1 = 2147483647$

Rappresentazione dei numeri

- Abbiamo visto che per i numeri interi positivi si usa la ***rappresentazione binaria posizionale***

  $(101100)_2 = (44)_{10}$

- Se si usa una rappresentazione a **n** bit, si possono rappresentare i 2^n numeri interi che sono compresi nell'intervallo

  $[0 , 2^n - 1] \cap Z$

n è la dimensione (in bit) della cella di memoria che contiene il numero

Numeri interi relativi

- Come possiamo rappresentare i numeri negativi?
  - la rappresentazione più naturale è quella detta ***rappresentazione con modulo e segno***
  - si rappresenta il segno positivo o negativo del numero con il primo bit della sequenza (quello più a sinistra), quindi si rappresenta il modulo o valore assoluto del numero, che ovviamente è un numero non negativo

    $(101100)_2 = (-12)_{10}$

    $(001100)_2 = (+12)_{10}$

Numeri interi relativi

- Se si usa una rappresentazione a **n** bit, si possono rappresentare i 2^n-1 numeri interi che sono compresi nell'intervallo

$$[-(2^{n-1} - 1) , 2^{n-1} - 1] \cap Z$$

- Problema: c'è una doppia rappresentazione per lo zero (+0 e -0), per cui si "spreca" una configurazione
- Problema: l'algoritmo per l'addizione di numeri così rappresentati è complesso

Numeri interi relativi

- Addizione **S = A + B** eseguita con numeri rappresentati in modulo e segno
- Se segno(A) = segno(B)

  segno(S) = segno(A), |S| = (|A| + |B|)

  altrimenti

  se |A| ≥ |B|

  segno(S) = segno(A), |S| =(|A| - |B|)

  altrimenti

  segno(S) = segno(B), |S| =(|B| - |A|)

Complemento a due

- Una rappresentazione più efficiente è quella denominata ***complemento a due***, così definita
  - dato un numero intero relativo

$$a \in [-2^{n-1} , 2^{n-1} - 1] \cap Z$$

la sua rappresentazione in complemento a due è

  - rappresentazione binaria senza segno a n bit di a
    - se a≥0
  - rappresentazione binaria senza segno a n bit di $(a+2^n)$
    - se a<0

Complemento a due

- Esempio: il numero -13 in complemento a due **a 8 bit**
  - Ha la rappresentazione senza segno a 8 bit di
    - $-13 + 2^8 = -13 + 256 = 243$
  - Allora $(243)_{10} = (11110011)_2$ (verificare per esercizio)

- Fortunatamente non dobbiamo fare questa operazione, esiste un procedimento più semplice
  - Scrivere la rappresentazione di +13
    - *00001101*
  - Scambiare gli 0 con gli 1
    - *11110010*
  - Aggiungere 1 al risultato
    - *11110011*

Complemento a due

- Proprietà (dimostrabili)
  - il segno di un numero rappresentato in complemento a due è ancora il bit più a sinistra della rappresentazione (0 se positivo o nullo, 1 se negativo)
  - la parte restante della rappresentazione NON è il valore assoluto del numero
    - lo è soltanto per i numeri positivi
  - non ci sono più configurazioni "sprecate"
    - con **n** bit si rappresentano 2^n numeri diversi

Complemento a due

- Proprietà (dimostrabili)
  - per eseguire l'addizione di numeri rappresentati in complemento a due si esegue semplicemente l'addizione binaria delle rappresentazioni, senza pensare al fatto che il bit più a sinistra rappresenta il segno
  - al termine dell'addizione, NON bisogna considerare un eventuale riporto che si venisse a trovare nella posizione **n**, cioè un'eventuale (**n**+1)-esima cifra del risultato non ne fa parte

Overflow in complemento a due

- Come per tutte le rappresentazioni numeriche, anche il complemento a due può dar luogo a fenomeni di ***trabocco*** (***overflow***) quando il risultato di un'operazione non rientra nell'intervallo dei numeri rappresentabili con il numero di bit usati dalla rappresentazione

- Ad esempio, nell'addizione **a+b** si ha overflow se

$$a+b \notin [-2^{n-1} , 2^{n-1} - 1] \cap Z$$

- Eseguendo l'addizione, come ci si accorge di una situazione di overflow?

Overflow in complemento a due

- Nell'addizione di due numeri in complemento a due si ha una situazione di overflow se e solo se
  - si ha un riporto tra la colonna (n-1)-esima e la colonna n-esima e non si ha un riporto tra la colonna n-esima e la colonna (n+1)-esima
    - **Esercizio**: eseguire la somma ***6+2*** a 4 bit (qual è il massimo intero positivo rappresentabile con 4 bit?)
- oppure
  - non si ha un riporto tra la colonna (n-1)-esima e la colonna n-esima e si ha un riporto tra la colonna n-esima e la colonna (n+1)-esima
    - **Esercizio**: eseguire la somma ***(-3) + (-7)*** a 4 bit (qual è il minimo intero negativo rappresentabile con 4 bit?)
- Anche questa proprietà è dimostrabile

I numeri frazionari

- I numeri frazionari vengono convertiti in binario effettuando separatamente la conversione della parte intera e della parte frazionaria propria, con le regole già viste
  - le due rappresentazioni binarie così ottenute vengono giustapposte con il separatore decimale (virgola o punto, secondo il sistema adottato)
  - questa rappresentazione dei numeri frazionari viene detta "**in virgola fissa**", ma è poco usata nei calcolatori

I numeri in virgola mobile

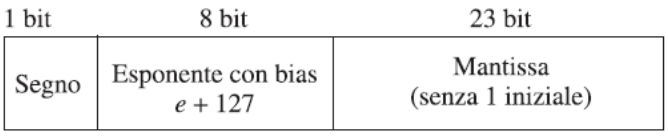
- I numeri frazionari vengono di solito rappresentati nei computer come **numeri in virgola mobile**
  - si rappresentano con la sequenza delle loro ***cifre significative*** e con l'indicazione della ***posizione del separatore decimale***
    - **250** e **2,5** hanno le stesse cifre significative (**25**), ma diverse posizioni del separatore decimale
  - equivale alla rappresentazione esponenziale
    - **$2{,}5 \times 10^2$**

I numeri in virgola mobile

- I numeri frazionari vengono di solito rappresentati nei computer come numeri in virgola mobile
  - la sequenza delle cifre significative si chiama **mantissa**
    - il numero in virgola mobile si dice **normalizzato** se la mantissa inizia con una cifra diversa da 0
  - è molto facile moltiplicare o dividere per 10, in quanto cambia solo la posizione della virgola (che è, in questo senso, ***mobile***)

I numeri in virgola mobile

- In realtà si usa la rappresentazione in **base 2** anziché 10, ma il concetto non cambia
  - ovviamente, cambiando la base, cambia la mantissa
- Esiste uno standard internazionale (**IEEE 754**) che definisce esattamente il formato e la disposizione dei bit di mantissa ed esponente (con i relativi segni)
  - due formati, 32 bit e 64 bit



Singola precisione

Doppia precisione

Virgola mobile a 32 bit (IEEE754)

- Il numero più piccolo (positivo) rappresentabile è

  $\mathbf{1.00000000000000000000000_2 \times 2^{-126} \sim 1.8 \times 10^{-38}}$

- Il numero più grande rappresentabile è

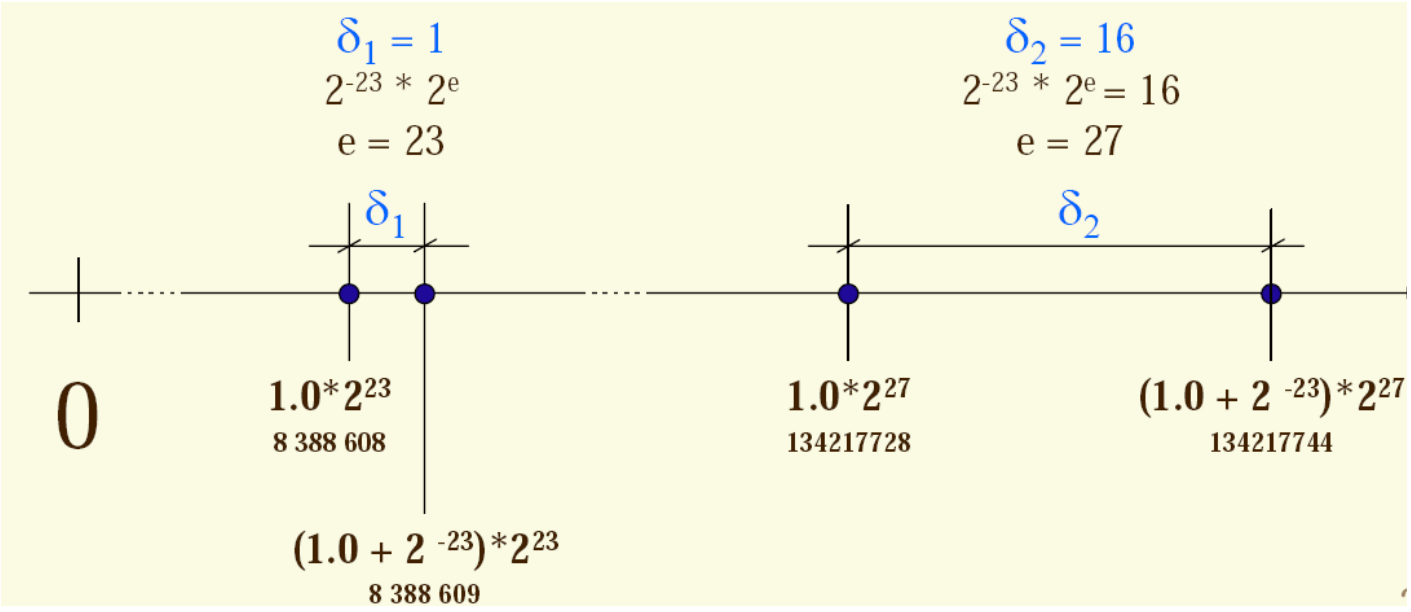
  $\mathbf{1.11111111111111111111111_2 \times 2^{+127} \sim 3.4 \times 10^{+38}}$

- La distanza fra due numeri reali successivi rappresentabili **dipende dal valore dell'esponente**
  - i numeri più vicini differiscono per il valore del bit meno significativo della mantissa e perciò la loro distanza **δ** è

    $\mathbf{\delta = 2^{-23} \times 2^{E}}$ (E e' il valore dell'esponente)

Densità numeri in virgola mobile

- Esempio in formato IEEE754: singola precisione (mantissa a 23 bit)



Arrotondamento in virgola mobile

- Il numero 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 inferiore a 4,35
- Se effettuiamo la moltiplicazione 435 x 100, il risultato è un numero appena inferiore a 435
  - L'errore di arrotondamento viene amplificato dalla moltiplicazione

$$4.35 \times 100 = 434.9999999999999994 \neq 435$$

Errori nelle somme in virgola mobile

- Si consideri la somma **10.5 + 0.125**
  - $10.5_{10} = 1010.1_2 = 1.0101_2 \times 2^3$
  - $0.125_{10} = 0.001_2 = 1.0_2 \times 2^{-3}$
- Per eseguire la somma bisogna **riportare entrambi i termini allo stesso esponente**:

$$10.5+0.125 = 1.0101_2 \times 2^3 + 0.000001 \times 2^3 = 1.010101 \times 2^3$$

- 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 !!!!!

Numeri “speciali” in virgola mobile

- La rappresentazione in virgola mobile dello standard **IEEE 754** include alcuni valori speciali
  - Zero
    - Esponente = -127 (esponente con bias = 00000000)
    - Mantissa = 0
  - Infinito
    - Esponente = +128 (esponente con bias = 11111111)
    - Mantissa = 0
  - NaN (Not a Number, “non è un numero”)
    - Esponente = +128 (esponente con bias = 11111111)
    - Mantissa diversa da 0

Rappresentazione esadecimale

- Spesso si usano anche numeri in base **sedici** (sistema di numerazione **esadecimale**)
- Sedici è una potenza di due ($16 = 2^4$)
  - si può facilmente passare dalla base binaria alla base esadecimale, raggruppando i bit **quattro a quattro**
- Il problema è la rappresentazione delle **sedici diverse cifre** della base esadecimale!

Rappresentazione esadecimale

- Per le prime dieci cifre si usano le cifre decimali, poi le lettere da A (=10) a F (=15)

  $(\mathbf{111}\mathbf{1}\mathbf{1}\mathbf{000}\mathbf{110})_2 = (\mathbf{7C6})_{16} = \mathbf{0x}7C6$

  notazione alternativa

- Per la conversione inversa, si sostituisce ciascuna cifra esadecimale con le corrispondenti quattro cifre binarie, eliminando eventuali zeri a sinistra

Rappresentazione ottale

- Numeri binari di valore elevato sono rappresentati da lunghe sequenze di cifre binarie, difficili da leggere
- Spesso si usano numeri in base **otto** (sistema di numerazione **ottale**)
- Dato che otto è una potenza di due ($8 = 2^3$), si può facilmente passare dalla base binaria alla base ottale, raggruppando i bit **tre a tre**

Rappresentazione ottale

$(100010)_2 = (42)_8$ = 042

notazione alternativa

- Attenzione: si raggruppano i bit tre a tre **a partire da destra**!

  $(11100010)_2 = (342)_8$

- Per la conversione inversa
  - si sostituisce ciascuna cifra ottale con le corrispondenti tre cifre binarie, eliminando eventuali zeri a sinistra

Rappresentazione dei caratteri (appendice G)

I caratteri

- I caratteri appartenenti ad un alfabeto vengono **codificati** (cioè "rappresentati") mediante sequenze di bit
  - una diversa sequenza per ciascun diverso carattere
- L'esempio più famoso è il codice **ASCII**
  - American Standard Code for Information Interchange
  - usa una sequenza di **7 bit** per ciascun carattere dell'alfabeto inglese
    - ci sono 128 ($=2^7$) sequenze diverse, utilizzate anche per segni di punteggiatura, cifre decimali, ecc.

I caratteri

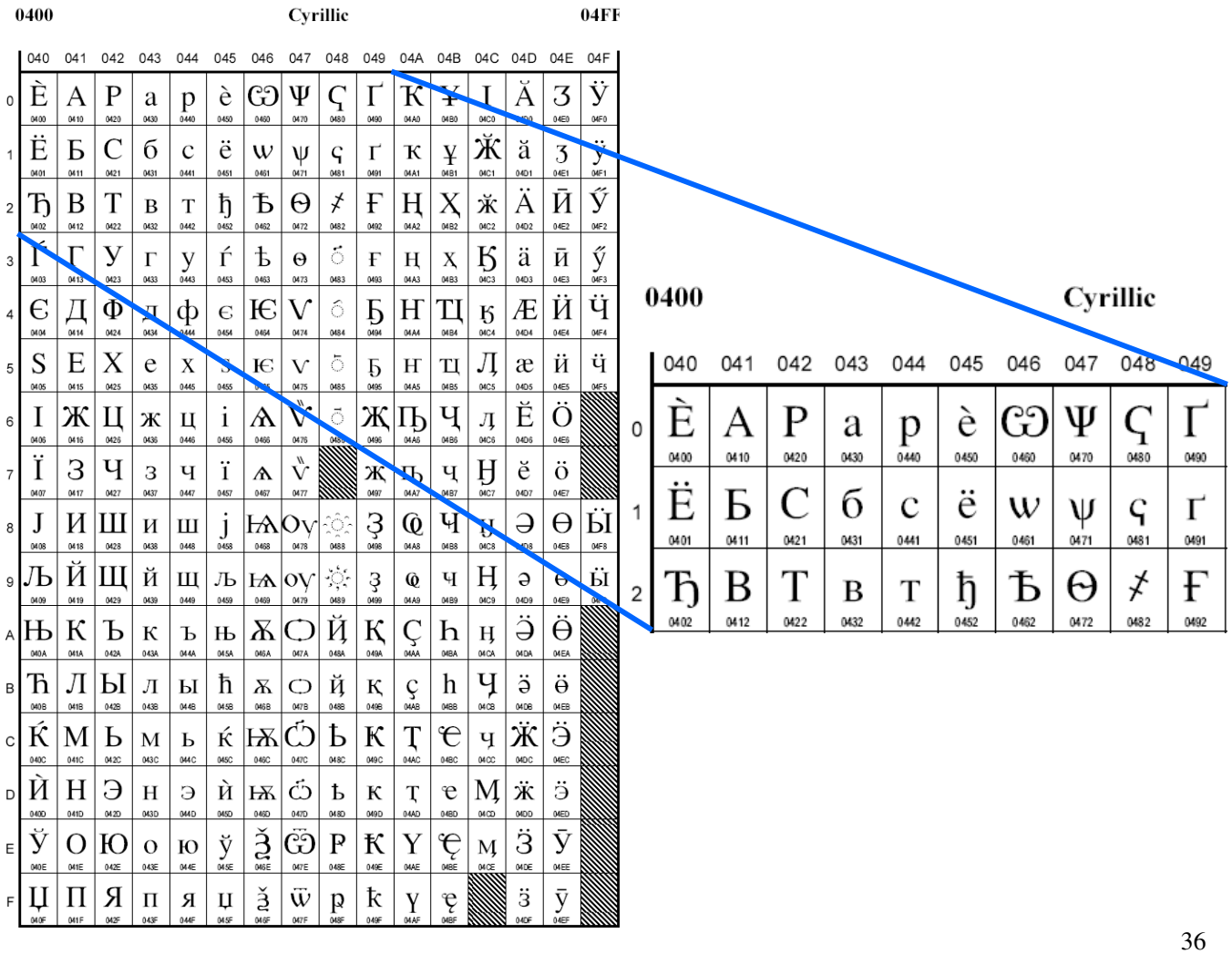
- Dato che l'unità elementare di informazione nei calcolatori è il **byte** (= 8 bit), si usa quasi sempre il **codice ASCII esteso**
  - usa una sequenza di **8 bit** per ciascun carattere degli alfabeti occidentali
    - ci sono **256** ($=2^8$) sequenze diverse, utilizzate anche per vocali accentate e altre lettere speciali (es. ß tedesca, ç francese)
    - le sequenze con la prima cifra uguale a zero coincidono con il codice ASCII

      ➔**compatibilità**

I caratteri (codice Unicode)

- Per rappresentare i segni grafici utilizzati da tutti gli alfabeti del mondo servono molti più simboli diversi
  - codifica **Unicode**, `http://www.unicode.org`
  - usa una sequenza di **16 bit** per ciascun segno grafico
    - ci sono **65536** (=2^{16}) sequenze diverse
    - le sequenze con le prime otto cifre uguali a zero coincidono con il codice ASCII esteso

      ➔**compatibilità**



Utilizzare oggetti (capitolo 2)

Tipi e Variabili

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
      String outMessage = "Valore totale in euro ";
      System.out.print(outMessage);
      System.out.println(totalEuro);
   }
}
```

L'uso delle variabili

- Ogni programma fa uso di ***variabili***
- Le ***variabili*** sono spazi di memoria, identificati da un ***nome***, che possono conservare ***valori*** di un ***determinato tipo***
- Ciascuna variabile deve essere ***definita***, indicandone il ***tipo*** ed il ***nome*** `int lit;` `String outMessage;`
- Una variabile può contenere soltanto valori del suo ***stesso tipo***
- Nella ***definizione di una variabile***, è possibile ***assegnarle*** 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);
   }
}
```

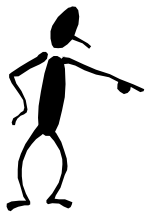
ma sarebbe stato ***molto meno comprensibile*** e ***modificabile con difficoltà***

Attenzione:
questo visualizza il risultato dell'espressione

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
  - non può essere una **parola riservata** o un **simbolo riservato** del linguaggio
  - non può contenere spazi
- Le lettere ***maiuscole*** sono diverse dalle ***minuscole***! Ma è buona norma non usare in un 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 metodi iniziano con una lettera minuscola***

```
lit    main
```

  - ***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 (nomi a "forma di cammello")

```
outMessage    totalEuro    MoveRectangle
```

È tutto chiaro? …

1. Di che tipo sono i valori **0** e **"0"**?
2. Quali dei seguenti identificatori sono validi?

```
Greeting1
g
void
101dalmatians
Hello, World
<greeting>
```

3. Definire una variabile adatta a memorizzare il vostro nome, usando un nome a forma di cammello

L'assegnazione

- 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'assegnazione

```
public class Coins3
{  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 * 0.72;
      System.out.print("Valore totale in euro ");
      System.out.println(totalEuro);
   }
}
```


L'assegnazione

- 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 * 0.79;
```

  mediante ***enunciati di assegnazione***

L'assegnazione

Molto importante!

- Analizziamo l'enunciato di assegnazione

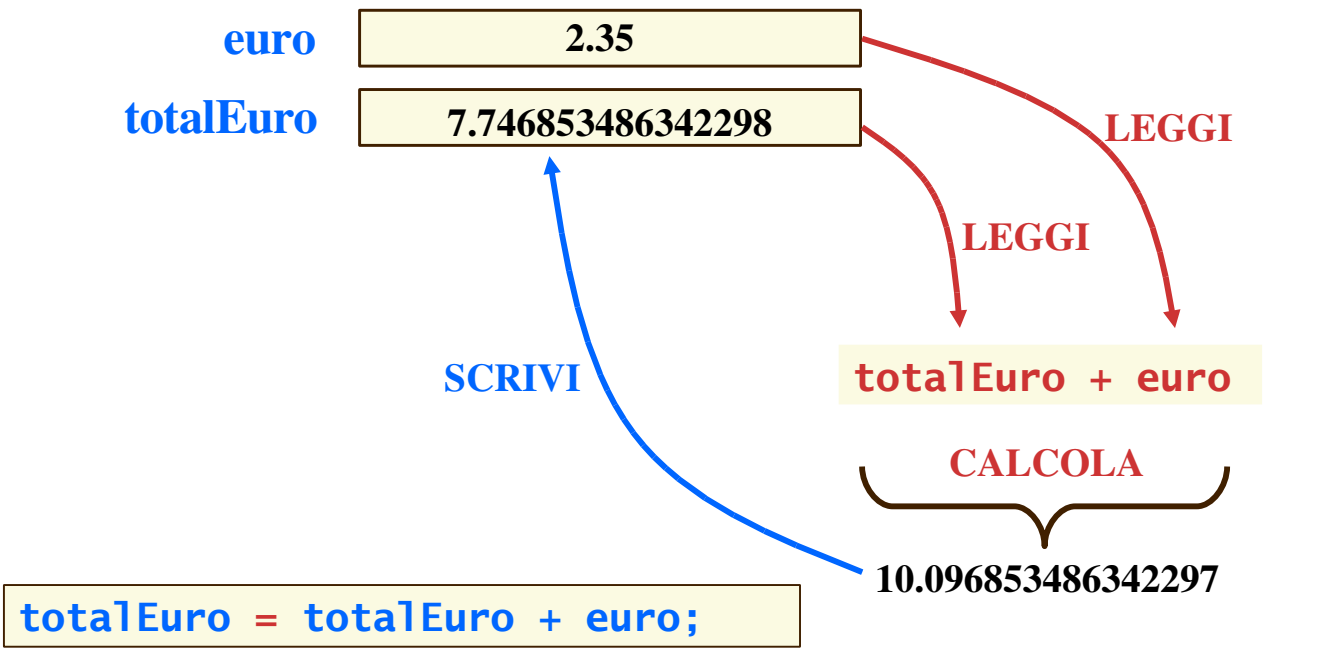
```
totalEuro = totalEuro + euro;
```

- Cosa significa? ***Non*** certo che la variabile **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

Molto importante!



Assegnazione o definizione?

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

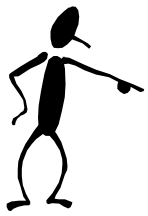
```
double totalEuro = lit / 1936.27;
totalEuro = totalEuro + euro;
```

- La definizione di una variabile inizia specificando il ***tipo*** della variabile, l'assegnazione no
- ***Una variabile può essere definita una volta sola***, 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 **:=**)

È tutto chiaro? …

1. L'espressione `12 = 12` è valida in Java?
2. Come si assegna il valore "Hello, Nina" alla variabile **`greeting`**, definita precedentemente nel codice?

Tipi numerici

Tipi numerici

```
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);
   }
}
```

Tipi numerici

- 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**

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
  - **"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!***
  - **"pratica"**: i numeri interi rappresentati come tipo di dati **int** sono più ***efficienti***, perché ***occupano meno spazio in memoria*** e sono ***elaborati più velocemente***

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***

```
1.93E3 // vale 1.93 × 10³
```

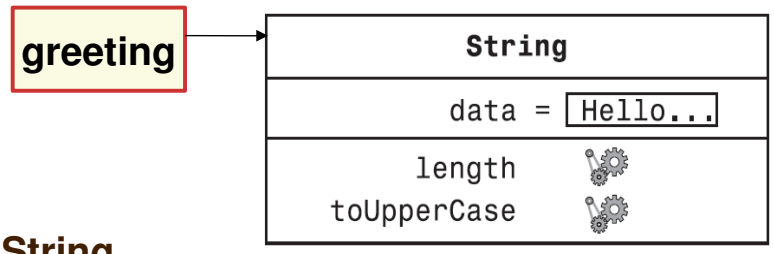
Oggetti, classi, metodi

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*** ed il loro ***comportamento*** vengono descritti mediante le ***classi*** ed i loro ***metodi***

Oggetti

- Un ***oggetto*** è un'entità che può essere manipolata in un programma mediante l'invocazione di ***metodi***



  - **greeting** è un oggetto
  - appartiene alla **classe String**
  - si può manipolare (usare) mediante i suoi **metodi**
    - Ad esempio **toUpperCase**
  - 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*** (il codice dei metodi ed 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 un ***contenitore*** di
    - metodi statici (**Hello** contiene **main**)
    - oggetti statici (**System** contiene **out**)

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 ad utilizzare
  - **usare oggetti di una classe**
  - **realizzare una classe**

Sono due attività *ben distinte*!

Metodi

- Costituiscono l'***interfaccia pubblica*** di una classe
  - Istruzioni valide:

```
String greeting = “Hello, World!”;
int n = greeting.length();
String river = “Mississipi”;
String BigRiver = river.toUpperCase();
```

  - Istruzione non valida (il metodo non appartiene alla classe)

```
System.out.length();
```

Metodi, parametri espliciti/impliciti

- Alcuni metodi necessitano di valori in ingresso che specifichino l'operazione da svolgere

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

  - greeting è un *parametro esplicito*

- Altri metodi no: tutte le informazioni necessarie sono memorizzate nell'oggetto corrispondente, il parametro implicito

```
int n = greeting.length();
```

  - greeting è il *parametro implicito*

Metodi, valori restituiti

- Alcuni metodi restituiscono un valore

```
System.out.println(greeting.length());
```

  - Il valore restituito da length viene usato come parametro esplicito di println
- Esempio più complesso

```
river.replace("issipp", "our");
```

  - Due parametri espliciti (le stringhe "issipp", "our")
  - Un parametro implicito (l'oggetto river)
  - Un valore restituito (la stringa "Missouri")
  - ATTENZIONE: river contiene ancora "Mississippi"

Definizioni di metodi

```
public void println(String output)
public String replace(String target, String replac)
```

- La ***definizione di un metodo*** inizia sempre con la sua ***intestazione*** (***firma***, ***signature***), composta da
  - uno specificatore di accesso
    - in questo caso **public**, altre volte vedremo **private**
  - il tipo di dati restituito dal metodo (**String**, **void...**)
  - il nome del metodo (**println**, **replace**, **length**)
  - un **elenco di parametri**, eventualmente vuoto, racchiuso tra **parentesi tonde**
    - di ogni parametro si indica il tipo ed il nome
    - più parametri sono separati da una virgola

Lo specificatore di accesso

- Lo ***specificatore di accesso*** di un metodo indica ***quali altri metodi possono invocare il metodo***
- Dichiarando un metodo **public** si consente l'accesso a ***qualsiasi altro metodo di qualsiasi altra classe***
  - è comodo per programmi semplici e ***faremo sempre così***, salvo casi eccezionali

Il tipo di dati restituito

- La dichiarazione di un metodo specifica quale sia il tipo di dati restituito dal metodo al termine della sua invocazione
  - ad esempio, il metodo **length()** restituisce un valore di tipo **int**
- Se un metodo ***non restituisce alcun valore***, si dichiara che restituisce il tipo speciale **void** (assente, non valido…)

```
int n = river.length();
```

```
double b = System.out.println(river); // ERRORE
System.out.println(river);            // OK
```


È tutto chiaro? …

1. Identificate i parametri impliciti ed espliciti, e il valore restituito dall'invocazione del metodo
   `river.length()`
2. Identificate il risultato dell'invocazione
   `river.replace("p", "s")`
   (immaginando che **`river`** contenga la stringa **`"Mississippi"`**)
3. Identificate il risultato dell'invocazione
   `greeting.replace("World", "Dave").length()`
   (immaginando che **`greeting`** contenga la stringa **`"Hello, World!"`**)

Variabili oggetto

- ***Una variabile oggetto*** conserva non l'oggetto stesso, ma informazioni sulla sua posizione nella memoria del computer
  - è un ***riferimento*** o ***puntatore***
- Per definire una variabile oggetto si indica il nome della classe ai cui oggetti farà riferimento la variabile, seguito dal nome della variabile stessa

```
NomeClasse nomeOggetto;
```

- La definizione di una variabile oggetto crea un riferimento ***non inizializzato***, cioè la variabile non fa riferimento ad alcun oggetto

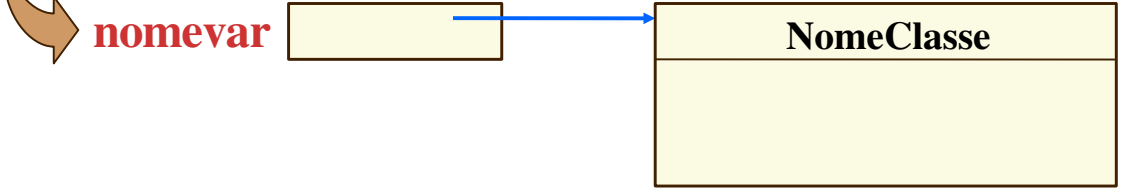
Costruire oggetti: l'operatore new

- Per ***creare un nuovo oggetto*** di una classe si usa l'***operatore*** **new** seguito dal ***nome della classe*** e da una coppia di parentesi tonde

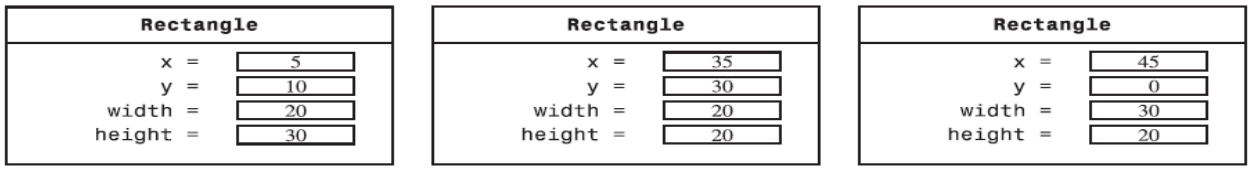
```
new NomeClasse(parametri);
```

- L'operatore **new** ***crea un nuovo oggetto e ne restituisce un riferimento***, che può essere assegnato ad una variabile oggetto del tipo appropriato

```
NomeClasse nomevar = new NomeClasse(parametri);
```



Esempio: la classe Rectangle



- Un rettangolo è descritto dalle coordinate (x,y) del suo vertice in alto a sinistra, e da larghezza e altezza.
- Per creare un rettangolo bisogna
  - Specificare x, y, width, height
  - Invocare l'operatore **new**
  - **Assegnare il rettangolo appena creato ad una variabile oggetto**

```
Rectangle box = new Rectangle(5, 10, 20, 20);
```


È tutto chiaro? …

1. Come si costruisce un quadrato centrato nel punto di coordinate 100, 100 con lato di lunghezza 20?

Copiare i riferimenti agli oggetti

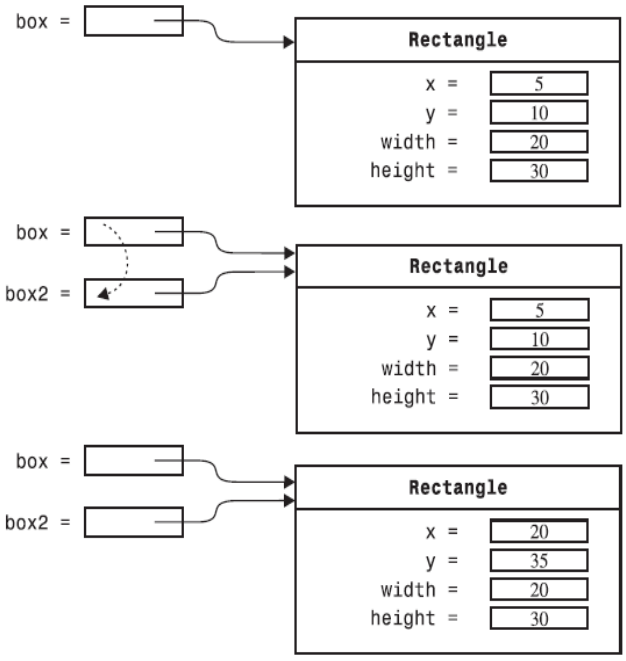
Riferimenti ad oggetti

```
Rectangle box = new Rectangle(5, 10, 20, 20);
Rectangle box2 = box;
box2.translate(15,25);
```

- **box** e **box2** contengono un riferimento allo stesso oggetto
  - Usando **box** o **box2** **modifico lo stesso oggetto!**
- Comportamento diverso dalle variabili numeriche:

```
int luckyNumber = 13;
int luckyNumber2 = luckyNumber;
luckyNumber2 = 12;
```

  - In questo caso il valore di **luckyNumber** **non cambia!**



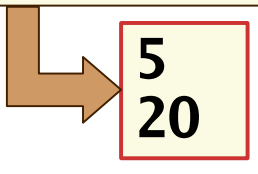
Copiare variabili

- Le variabili di tipi numerici fondamentali indirizzano una posizione in memoria che contiene un ***valore***, che viene copiato con l'assegnazione
  - cambiando il valore di una variabile, non viene cambiato il valore dell'altra
- Le variabili oggetto indirizzano una posizione in memoria che, invece, contiene un ***riferimento ad un oggetto***, e solo tale riferimento viene copiato con l'assegnazione
  - modificando lo stato dell'oggetto, tale modifica è visibile da entrambi i riferimenti
  - ***non viene creata una copia dell'oggetto!***

Copiare variabili

- Se si vuole ottenere anche con variabili oggetto lo stesso effetto dell'assegnazione di variabili di tipi numerici fondamentali, è necessario ***creare esplicitamente una copia dell'oggetto con lo stesso stato***, inizializzarlo adeguatamente e assegnarlo alla seconda variabile oggetto

```
Rectangle box = new Rectangle(5, 10, 20, 20);
Rectangle box2 = new Rectangle(box.getX(),
   box.getY(),box.getWidth(),box.getHeight());
box2.translate(15,25);
System.out.println(box.getX());
System.out.println(box2.getX());
```



Collaudare una classe

Programmi di collaudo

- Usati per “collaudare” il funzionamento di una classe
- Passi per costruire un programma di collaudo
  1. Definire una nuova classe
  2. Definire in essa il metodo main
  3. Costruire oggetti all'interno di main
  4. Applicare metodi agli oggetti
  5. Visualizzare risultati delle invocazioni dei metodi

- ATTENZIONE: bisogna *importare* le classi utilizzate

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
  - *Esempio*: la classe **Rectangle** appartiene al pacchetto **java.awt** (Abstract Window Toolkit)
- 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***

Esempio: MoveTester.java

```
import java.awt.Rectangle;
public class MoveTester
{
   public static void main(String[] args)
   {
      Rectangle box = new Rectangle(5, 10, 20, 30);

      // sposta il rettangolo
      box.translate(15, 25);

      // visualizza informaz. su rettangolo traslato
      System.out.println("After moving,
               the top-left corner is:");
      System.out.println(box.getX());
      System.out.println(box.getY());
   }
}
```

È tutto chiaro? …

1. La classe Random è definita nel pacchetto java.util. Cosa bisogna fare per utilizzarla in un programma?
2. Perché il programma MoveTester non visualizza altezza e larghezza del rettangolo dopo l'invocazione del metodo translate?

La documentazione della libreria standard

Guida in linea

- Le classi della libreria Java sono migliaia
  - Non usare la memoria, usare la documentazione!
- L'ambiente JDK fornisce la ***documentazione API (Application Programming Interface)*** per ***l'utilizzo delle classi della libreria standard***
  - Si può **scaricare** seguendo le indicazioni sul sito del corso
  - È disponibile **online** sul sito ***java.sun.com/***
  - È disponibile **online** sul sito dell'aula Taliercio
- Vengono inoltre forniti alcuni documenti in formato "***tutorial***" per la descrizione delle funzionalità di interi pacchetti ed esempi di programmi ("***demo***")
- È anche disponibile ***il codice sorgente di tutte le classi della libreria standard***, la cui lettura è interessante e utile, anche se spesso complessa

Guida in linea

- La ***descrizione di una classe*** comprende l'elenco di tutti i suoi metodi pubblici (la sua ***interfaccia***) ed una sintetica descrizione testuale delle motivazioni alla base del progetto della classe e delle modalità del suo utilizzo
- Per ogni metodo, vengono indicati
  - il nome e la funzionalità svolta
  - il tipo ed il significato dei parametri richiesti, se ci sono
  - il tipo ed il significato del valore restituito, se c'è
  - le eventuali ***eccezioni*** lanciate in caso di errore

Tipi di dati fondamentali (Capitolo 4)

Tipi primitivi

- In java ci sono 8 tipi primitivi
  - Di questi, sei sono tipi numerici, quattro per numeri interi e due per numeri in virgola mobile

Tipo	Descrizione	Dimensione
`int`	Tipo intero con intervallo –2147483648...2147483647 (circa 2 miliardi)	4 byte
`byte`	Tipo che descrive un singolo byte, con intervallo –128...127	1 byte
`short`	Tipo intero "corto", con intervallo –32768...32767	2 byte
`long`	Tipo intero "lungo", con intervallo –9223372036854775808... 9223372036854775807	8 byte
`double`	Tipo in virgola mobile a doppia precisione, con intervallo circa $\pm 10^{308}$ e circa 15 cifre decimali significative	8 byte
`float`	Tipo in virgola mobile a singola precisione, con intervallo circa $\pm 10^{38}$ e circa 7 cifre decimali significative	4 byte
`char`	Tipo che rappresenta caratteri codificati secondo lo schema Unicode (Argomenti avanzati 4.5)	2 byte
`boolean`	Tipo per i due valori logici `true` e `false` (Capitolo 6)	1 bit

Rappresentazione dei numeri in Java

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 una **eccezione** è la divisione con divisore zero
  - **ArithmeticException**

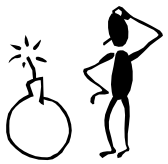
Numeri in virgola mobile in Java

- In Java tutti i tipi di dati fondamentali per numeri in virgola mobile usano internamente una rappresentazione binaria codificata dallo standard internazionale IEEE 754
  - **float** (32bit), **double** (64 bit)
- La divisione con divisore zero non è un errore se effettuata tra numeri in virgola mobile
  - se il dividendo è diverso da zero, il risultato è **infinito** (con il segno del dividendo)
  - se anche il dividendo è zero, il risultato è indeterminato, cioè non è un numero, e viene usata la codifica speciale **NaN** (**Not a Number**)

Numeri in virgola mobile in Java

- Lo standard IEEE 754 prevede quindi 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**

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 i computer rappresentano i numeri in virgola mobile in base ***binaria*** e a volte si ottengono dei risultati inattesi!

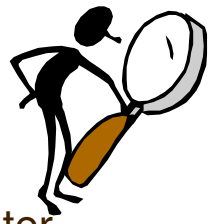
Arrotondamento

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

Stampa 434.9999999999999994 ≠ 435

- Il numero 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 inferiore a 4,35, che, quando viene moltiplicato per 100, fornisce un numero appena un inferiore a 435

Intervalli numerici e precisione



- Come molte rappresentazioni di dati in un computer, anche ***la rappresentazione di numeri*** in Java ***soffre di alcune limitazioni***, dovute a scelte progettuali di compromesso tra la precisione della rappresentazione e la velocità di elaborazione delle operazioni aritmetiche in Java
- I valori di tipo **int** sono compresi tra

  -2147483648 (costante **Integer.MIN_VALUE**)

  +2147483647 (costante **Integer.MAX_VALUE**)

  quindi, ad esempio, la popolazione mondiale non può essere rappresentata con una variabile **int**

Intervalli numerici e precisione

- I numeri in ***virgola mobile*** hanno invece un ***intervallo di variabilità molto più ampio***
  - i **double**, ad esempio, hanno un valore assoluto massimo di circa 10^{308} (**Double.MAX_VALUE**)

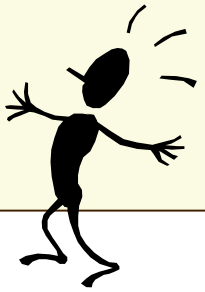
  ma soffrono di un altro importante problema, la ***mancanza di precisione***, perché possono rappresentare "soltanto" 15 ***cifre significative***

- Cosa significa "mancanza di precisione"?
  - significa che a volte le operazioni aritmetiche con numeri in virgola mobile ***danno risultati inattesi***...

Intervalli numerici e precisione

```
public class DiscountTester
{
   public static void main(String[] args)
   {
      final double AMOUNT = 1.0e+17;
      final int DISCOUNT = 50;

      double doubleResult = AMOUNT - DISCOUNT;
      long longResult = ((long) AMOUNT) - DISCOUNT;
      System.out.println(doubleResult);
      // mancano due unita` !!!!
      System.out.println(longResult);
      // questa volta e` giusto
   }
}
```



Altri tipi di dati numerici

- Quando il tipo **int** non rappresenta in modo soddisfacente le esigenze numeriche del problema, si possono usare ***altri tipi di dati interi*** in Java
- Se l'intervallo di variabilità è insufficiente, si può usare il tipo **long**
  - il massimo valore assoluto esprimibile con una variabile di tipo **long** è circa ***9 miliardi di miliardi*** (**Long.MAX_VALUE** e **Long.MIN_VALUE**)
  - per assegnare un valore numerico ad una variabile di tipo **long** bisogna aggiungere un carattere **L** alla fine

```
long x = 3000000000L;
```


Altri tipi di dati numerici

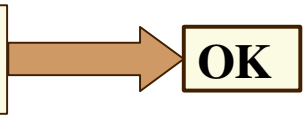
- Esistono altri due tipi di dati ***per numeri interi***
  - **byte**, con valori tra **-128** e **+127**
  - **short**, con valori tra **-32768** e **+32767**
- Esiste anche un altro tipo di dati ***per numeri in virgola mobile***, **float**, le cui variabili occupano meno spazio in memoria e (a volte...) vengono elaborate più velocemente, ma hanno una precisione molto limitata (7 cifre significative)
- Questi tipi di dati ***si usano molto raramente***, ma alcuni metodi di classi della libreria standard richiedono l'uso di parametri di tipo **byte** o **float**

Conversioni fra tipi numerici

Assegnazioni con conversione

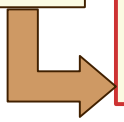
- In un'assegnazione, il ***tipo*** di dati dell'***espressione*** e della ***variabile*** a cui la si assegna devono essere ***compatibili***
  - se i tipi non sono compatibili, il compilatore segnala un ***errore*** (non sintattico ma ***semantico***)
- I tipi ***non*** sono compatibili se provocano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



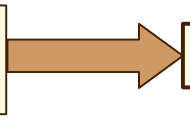
```
possible loss of precision
found    : double
required: int
```

- 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 ed 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")

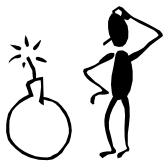
```
double doubleVar = 2.3;
int intVar = (int)doubleVar;
```



OK

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

Errori di arrotondamento



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

434 ≠ 435

- Come abbiamo visto, il numero 4,35 non ha una ***rappresentazione esatta*** nel sistema binario, proprio come 1/3 non ha una rappresentazione esatta nel sistema decimale
- Il cast fornisce quindi un risultato inatteso
  - 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** (che vediamo tra poco)

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

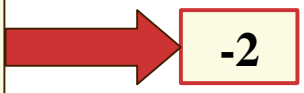
```
double rate = 2.95;
int intRate = (int)(rate + 0.5);
System.out.println(intRate);
```

3

Conversioni con arrotondamento

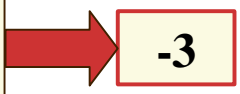
- Questo semplice algoritmo per arrotondare i numeri in virgola mobile funziona però soltanto per numeri positivi, quindi non è molto valido…

```
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 libreria standard, che funziona bene per tutti i numeri

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



È tutto chiaro? …

1. In quali situazioni il cast

   `(long) x`

   produce un risultato diverso dall'invocazione

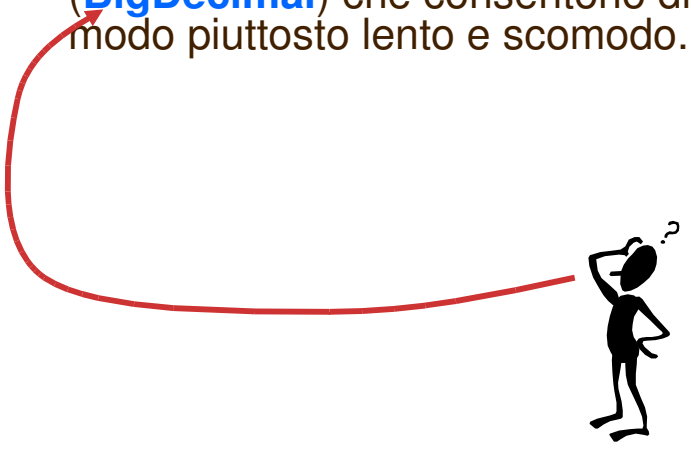
   `Math.round(x)` ?

2. In che modo possiamo arrotondare al più vicino valore di tipo int il valore x di tipo double, sapendo che è minore di 2×10^9 ?

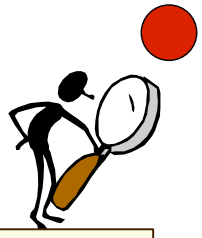
Tipi di dati numerici: materiale di complemento

Altri tipi di dati numerici

- Come possiamo elaborare numeri interi o numeri in virgola mobile che non rientrino nel campo di variabilità di **long** o **double**? Come possiamo elaborare numeri in virgola mobile ***con precisione arbitraria*** (cioè con tutta la precisione necessaria per il problema in esame)?
- Il ***pacchetto*** **java.math** della libreria standard mette a disposizione due classi per rappresentare rispettivamente numeri interi (**BigInteger**) e numeri in virgola mobile (**BigDecimal**) che consentono di fare ciò, anche se in modo piuttosto lento e scomodo...



Altri tipi di dati numerici



```
import java.math.BigInteger;

public class BigNumbers
{  public static void main(String[] args)
   {  BigInteger a = new BigInteger("123456789");
      BigInteger b = new BigInteger("987654321");
      BigInteger c = a.multiply(b);
      System.out.println(c);
   }
}
```

```
121932631112635269
```


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 * 0.72;
   }
}
```

- Chi legge il programma potrebbe legittimamente chiedersi quale sia il significato del "***numero magico***" **0.72** 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 = 0.72;
      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 dollaro/euro), la modifica va fatta ***in un solo punto*** del codice!

```
public class Convert3
{  public static void main(String[] args)
   {  final double EURO_PER_DOLLAR = 0.72;
      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*** ad 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***

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

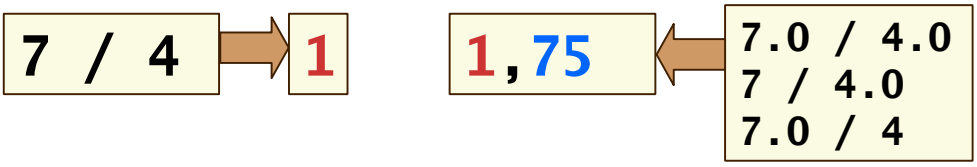
- In Java non esiste il ***simbolo di frazione,*** le frazioni vanno espresse "***in linea***", usando l'operatore di divisione

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

$\frac{a+b}{2}$ → `(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 %, (questo simbolo non esiste in algebra, 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) Math.round(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
```

Funzioni più complesse

- Non esistono ***operatori*** per calcolare funzioni più complesse, come l'elevamento a potenza
- La classe **Math** della libreria standard 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 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

Costanti della classe Math

- Nella classe **Math** sono definite alcune utili ***costanti***

```
public final class Math
{  ...
   public static final double PI =
                3.14159265358979323846;
   public static final double E =
                2.7182818284590452354;

}
```

- Sono ***costanti*** statiche, ovvero appartengono alla classe (approfondiremo in seguito)
- Tali costanti sono di norma **public** e per ottenere il loro valore si usa il nome della classe seguito dal punto e dal nome della costante, **Math.E**, oppure **Math.PI**

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***

```
totalEuro = totalEuro + dollars * 0.72;
```

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

```
totalEuro += dollars * 0.72;
```

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--;
```

È tutto chiaro? …

1. Qual è il valore dell'espressione **1729/100** ? E di **1729%100** ?
2. Perché questo enunciato non calcola la media tra s1, s2, ed s3?

```
double average = s1 + s2 + s3 / 3;
```

3. Come si esprime in notazione matematica la seguente espressione ?

```
Math.sqrt(Math.pow(x, 2) +
  Math.pow(y, 2))
```

Metodi statici

Invocare metodi statici

```
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***

Invocare metodi statici

Seguitela anche voi!

- Come si fa a capire che **System.out.println** è un metodo applicato ad 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 metodo statico

- Sintassi:

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

- Scopo: invocare il metodo statico ***nomeMetodo*** definito nella classe ***NomeClasse***, fornendo gli eventuali ***parametri*** richiesti
- Nota: un metodo statico non viene invocato con un oggetto, ma con un nome di classe

È tutto chiaro? …

1. Perché non si può invocare **`x.pow(y)`** per calcolare x^y ?
2. L'invocazione **`System.out.println(4)`** è l'invocazione di un metodo statico?

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";
```

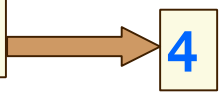
- Possiamo ***assegnare un valore*** ad una variabile di tipo stringa

```
name = "Michael";
```

Il tipo di dati "stringa"

- 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…
- Una ***variabile*** di tipo stringa può quindi essere utilizzata per ***invocare 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"

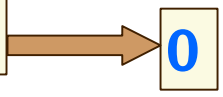
- Il metodo **length** della classe **String** ***non è un metodo statico***
  - infatti ***per invocarlo usiamo un oggetto della classe String, e non il nome della classe stessa***

```
// 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());
```



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 ***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 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

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

- 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
at java.lang.String.substring(String.java:1444)
at NomeClasse.main(NomeClasse.java:16)
```

Concatenazione di stringhe

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

```
String s1 = "eu";
String s2 = "ro";
String s3 = s1 + s2; // s3 contiene euro
int euro = 15;
String s = euro + s3; // s contiene "15euro"
```

- Il simbolo dell'operatore di concatenazione è identico a quello dell'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 euro = 15;
String euroName = "euro";
String s = euro + euroName;
// s contiene "15euro"
```

- 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

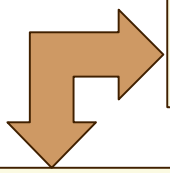
```
String s = euro + " " + euroName;
```

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 è ");
System.out.println(total);
```



```
int total = 10;
System.out.println("Il totale è " + 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

- Scriviamo un programma che genera la password per un utente, con la regola seguente
  - si prendono le iniziali dell'utente, le si rendono minuscole e si concatena l'età dell'utente espressa numericamente

```
Utente: Federico Avanzini
Età: 35
⇒ Password: fa35
```

(in realtà questa regola non è assolutamente da usare, perché è prevedibile e quindi poco sicura!)

Esempio

```
public class MakePassword
{  public static void main(String[] args)
   {  String firstName = "Federico";
      String lastName = "Avanzini";
      int age = 35; //ma non li dimostra

      // estrai le iniziali
      String initials = firstName.substring(0, 1)
         + lastName.substring(0, 1);
      // converti in minuscolo e concatena l'età
      String pw = initials.toLowerCase() + age;
      // stampa la password
      System.out.println("La password è " + pw);
   }
}
```


È tutto chiaro? ...

1. Se la variabile s di tipo String contiene il valore "Agent", che effetto produce il seguente enunciato?

   ```
   s = s + s.length();
   ```

2. Se la variabile river di tipo String contiene il valore "Mississippi", che valori hanno le seguenti espressioni?

   ```
   river.substring(1, 2)
   river.substring(2, river.length() - 3)
   ```

Conversione di stringhe in numeri

- A volte si ha una stringa che contiene un valore numerico e si vuole assegnare tale valore ad una variabile di tipo numerico, per poi elaborarlo

```
String password = "fa35";
String ageString = password.substring(2);
// ageString contiene "35"
// NON FUNZIONA!
int age = ageString;
```

```
incompatible types
found    : java.lang.String
required: int
```

- Il compilatore segnala l'errore semantico perché non si può convertire automaticamente una stringa in un numero, dato che ***non vi è certezza che il suo contenuto rappresenti un valore numerico***

Conversione di stringhe in numeri

- La conversione corretta si ottiene invocando il metodo statico **parseInt** della classe **Integer**

```
int age = Integer.parseInt(ageString);
// age contiene il numero 35
```

- La conversione di un ***numero in virgola mobile*** si ottiene, analogamente, invocando il metodo statico **parseDouble** della classe **Double**

```
String numberString = "35.3";
double number = Double.parseDouble(numberString);
// number contiene il numero 35.3
```

- **Integer e Double sono "classi involucro" dei tipi primitivi int e double**

Conversione di stringhe in numeri

- ***Cosa succede se la stringa passata come argomento non contiene un numero***?
  - i metodi **Integer.parseInt** e **Double.parseDouble** lanciano un'**eccezione** di tipo **NumberFormatException** ed il programma termina segnalando l'errore
- Abbiamo già visto casi in cui il verificarsi di una eccezione arresta il programma
  - **StringIndexOutOfBoundsException** in **substring**
- Il meccanismo generale di segnalazione di errori in Java consiste nel "lanciare" (***throw***) un'***eccezione***
  - si parla anche di ***sollevare*** o ***generare*** un'eccezione
  - Vedremo più avanti il meccanismo di gestione delle eccezioni

Conversione di numeri in stringhe

- Per ***convertire*** un numero in stringa si può ***concatenare il numero con la stringa vuota***

```
int ageNumber = 10;
String ageString = "" + ageNumber;
// ageString contiene "10"
```

- È però più elegante (e più comprensibile) utilizzare il metodo **toString** delle classi **Integer** e **Double**, rispettivamente per numeri interi e numeri in virgola mobile

```
int ageNumber = 10;
String ageString = Integer.toString(ageNumber);
```