

Terminologia di base per la programmazione

Prof. Luca Trevisan

Corso di Fondamenti di Informatica
A.A. 2025/2026





Abbiamo visto...

```
public class NomeClasse
{ public static void main(String[] args)
    { enunciati
    }
}
```

- classe:
 - contenitore di **metodi** o **fabbrica di oggetti** (elementi manipolabili in Java)
- metodo:
 - sequenza di istruzioni (=enunciati) per eseguire un dato compito
 - main
 - System.out.println()
- parole chiave (riservate dal linguaggio)



Obiettivi

- Attraverso l'analisi di un esempio di programma di elaborazione di dati introdurremo i concetti di:
 - Variabili
 - Commenti
 - Assegnazione
 - Costanti
 - Operatori aritmetici



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Variabili



Le variabili

- Un programma (spesso) elabora ***dati***
- È quindi necessario che esista un meccanismo che permetta di memorizzare i dati da elaborare e gli eventuali risultati ottenuti
- **Variabili**: spazi di memoria, identificati da un ***nome***, che possono conservare ***valori*** di un ***determinato tipo***



Le variabili

□ In Java ogni **variabile** ha:

□ **Nome**

- identificativo che viene utilizzato per “usare” la variabile

□ **Tipo**

- determina l’insieme dei valori ammissibili (assegnabili)
- determina l’occupazione di memoria

□ **Contenuto**

- valore associato

□ **Attributi**

- determinano la visibilità (public, protected, private) o specificano tipi particolari di variabili (static, final)

□ **Scope**

- determina la regione del programma in cui la variabile è accessibile e utilizzabile



Un programma che elabora numeri

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

      // 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:
int
- ***numeri frazionari*** ("in virgola mobile") per gli euro, che prevedono l'uso di decimi e centesimi e assumono valori con il separatore decimale:
double



Perché usare due tipi di numeri?

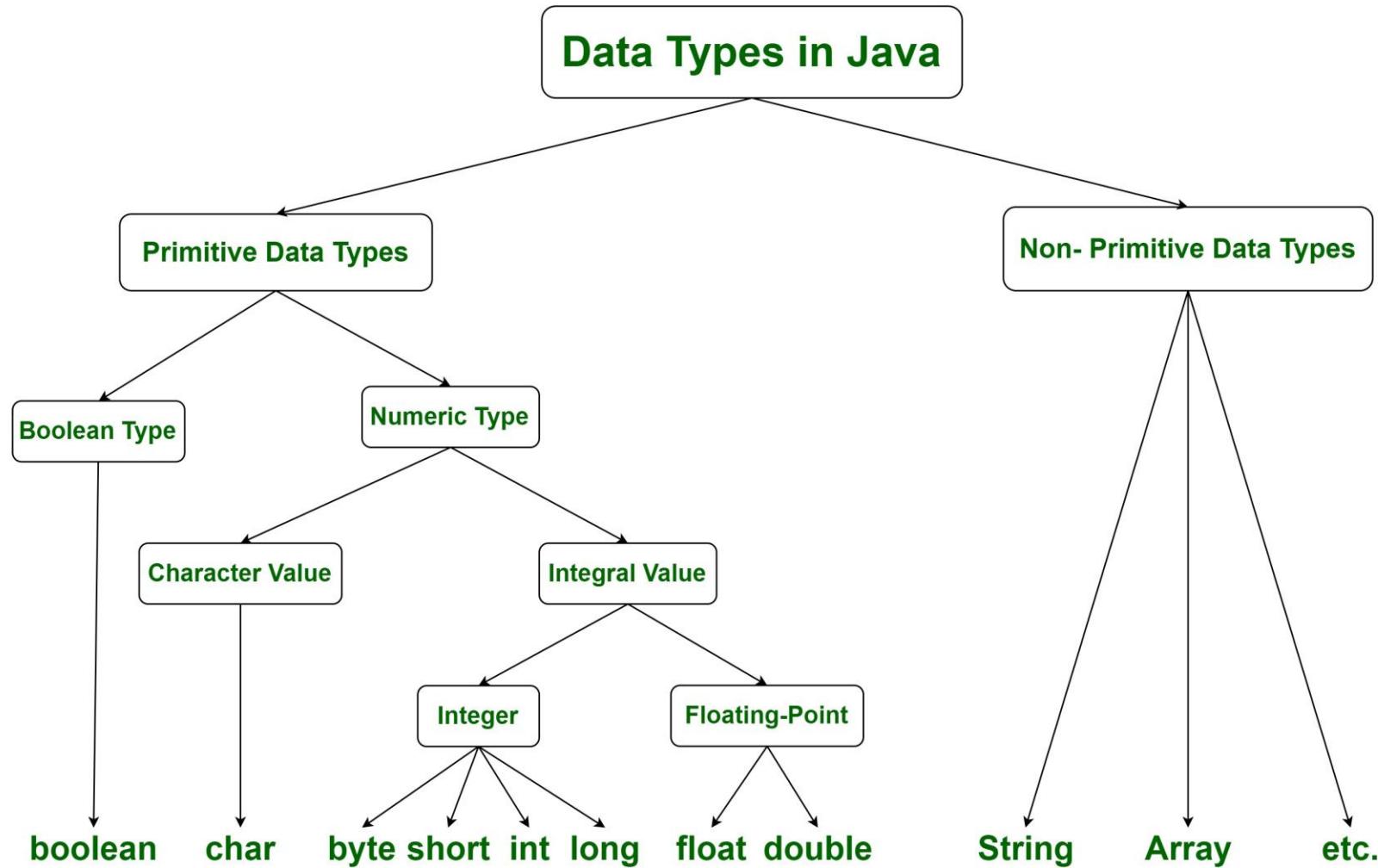
- In realtà sarebbe possibile usare numeri in virgola mobile anche per rappresentare i numeri interi, ma ecco due buoni motivi per non farlo:
 - “**filosofia**”: indicando esplicitamente che per le lire italiane usiamo un numero intero, rendiamo **evidente** il fatto che non esistono i decimali per le 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**



Tipi di dati, valori e variabili

- Java è un linguaggio **fortemente tipizzato** (**strongly typed**) e quindi ogni dato è di un ben preciso **tipo** noto al momento della **compilazione** del programma
- I **tipi** di dati in Java possono essere:
 - tipi primitivi
 - riferimenti ad un oggetto

Gerarchia di tipi di dato



Tipi di dati primitivi - valori

- Tipi numerici

- Interi

■ byte	1 byte
■ short	2 byte
■ int	4 byte
■ long	8 byte
■ char	2 byte

con segno

- Virgola mobile (numeri frazionari)

■ float	4 byte
■ double	8 byte

- Booleani

■ boolean	(tipicamente 1 byte)
------------------	----------------------



Uso delle variabili

- Il programma fa uso di **variabili** di **tipo numerico**
 - ▣ **lit** di tipo **int**, **euro** e **totalEuro** di tipo **double**

- *Dichiarazione* di una variabile
 - ▣ indico il **tipo** ed il **nome**

```
int lit;
```

- *Assegnazione* di un valore
 - ▣ una variabile può contenere soltanto valori del suo **stesso tipo**

```
lit = 15000;
```

- *Definizione* di una variabile
 - ▣ la **dichiariamo** e le **assegnamo** un **valore iniziale**

```
int lit = 15000;
```



Dichiarazione di variabili locali

□ La **dichiarazione**:

- Fissa nome, tipo e attributo (se presente)
- Determina l'**ambito di visibilità** (chiamato **scope**) che è dalla dichiarazione fino alla fine del blocco
 - (un blocco è una sezione di codice delimitata da parentesi graffe)

Sintassi: **nomeTipo nomeVariabile;**

□ La **definizione**:

- Riserva lo spazio in memoria necessario
- Inizializza il valore della variabile

Sintassi: **nomeTipo nomeVariabile = espressione;**



Variabili in Java

- **Variabili Statiche**: associate alla **classe** stessa, condivise tra tutte le istanze della classe
- **Variabili di Istanza**: associate a un **oggetto**, accessibili in tutta la classe
- **Variabili Locali**: definite all'interno di **metodi** o **blocchi**, limitate a quel contesto
 - Inizieremo da questa categoria di variabili
- **Parametri**: variabili definite nelle **firme** dei metodi, locali al metodo

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 ***meno comprensibile e modificabile con maggiore difficoltà!***

Ricordate:
questo visualizza il **risultato** dell'espressione, non ci sono le virgolette!!!!



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 (e concisi)**
- In Java, un nome o identificatore (di variabile, di metodo, di classe...) può essere composto da **lettere** (maiuscole e minuscole), da **numeri** e dal **carattere di sottolineatura** (*underscore*, `_`), ma:
 - deve iniziare con una lettera (o con `_` o il simbolo `$`)
 - non può essere una parola chiave del linguaggio (ma la può contenere, es. `publicCompany` o `republic`)
 - non può essere una parola riservata: `true`, `false`, `null`
 - non può contenere spazi
- Le lettere **maiuscole** sono diverse dalle **minuscole!**
E' buona norma non usare nello stesso programma nomi di variabili che differiscano soltanto per una maiuscola



I nomi delle variabili

- Parole/simboli **non** utilizzabili in nomi di variabili
- **Parole riservate** di Java: (slide seguente)
- **Separatori**: i seguenti 9 caratteri sono utilizzati come separatori (caratteri di interpunkzione)
 - () { } [] ; , .
- **Operatori**: i seguenti 37 caratteri o token sono utilizzati nelle espressioni come operatori
 - = > < ! ~ ? :
 - == <= >= != && || ++ --
 - + - * / & | ^ % << >> >>>
 - += -= *= /= &= |= ^= %= <<= >>= >>>=



Le parole chiave di Java

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Nota stilistica

- Di solito in Java si usano le seguenti **convenzioni**:

- *i nomi di variabili e di metodi iniziano con una lettera minuscola*

lit

main

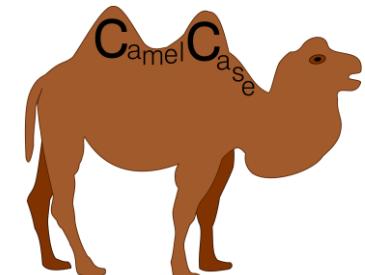
- *i nomi di classi iniziano con lettera maiuscola*

Coins1

- i nomi composti, in tutti i casi, si ottengono attaccando le parole, dopo aver messo l'iniziale maiuscola alle parole successive alla prima: **Camel Case**

totalEuro

MoveRectangle





I commenti

- Nel programma sono presenti anche dei **commenti**, che vengono **ignorati** dal compilatore, ma che rendono il programma **molto più comprensibile**
- Un commento **inizia con una doppia barra // e termina alla fine della riga**

// lire italiane
- 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 ma inutile...
*/
```

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



Utilità dei commenti

- Commentare il codice è **IMPORTANTISSIMO!!!!**
- **Chiarezza:** i commenti aiutano a **spiegare** cosa fa ogni parte del codice. Per chi non è familiare con la logica, i commenti possono rendere chiaro il funzionamento del programma.
- **Manutenibilità:** quando si **torna** al codice in un secondo momento, i commenti possono fornire contesto, rendendo più facile modificare o fare debug.
- **Collaborazione:** in ambienti di team, i commenti aiutano i membri a capire il codice degli altri, facilitando una migliore **collaborazione**.
- **Apprendimento:** per i nuovi sviluppatori, i commenti possono fungere da **guida** per comprendere lo scopo e la funzione dei segmenti di codice, aiutandoli a imparare migliori pratiche.



Utilità dei commenti

```
public class Factorial
{
    public static void main(String[] args)
    {
        int number = 5;
        int result = 1;

        for (int i = 1; i <= number; i++)
        {
            result *= i;
        }

        System.out.println("Factorial of " + number + " is " + result);
    }
}
```



Utilità dei commenti

```
// Calculates factorial of two numbers
public class Factorial
{
    public static void main(String[] args)
    {
        // Declare a variable for the number to calculate the factorial
        int number = 5;

        // Initialize result variable to store the factorial result
        int result = 1;

        // Loop from 1 to the number (inclusive)
        for (int i = 1; i <= number; i++)
        {
            // Multiply result by the current value of i
            result *= i; // This accumulates the product
        }

        // Print the final result to the console
        System.out.println("Factorial of " + number + " is " + result);
    }
}
```



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Assegnazioni



Uso delle variabili

- Il valore memorizzato in una variabile può essere **modificato**, non soltanto **inizializzato**...
 - **È per questo che si chiama variabile!!**
- Il cambiamento del valore di una variabile si ottiene con un **enunciato di assegnazione**



L'assegnazione

□ Sintassi:

```
nomeVariabile = espressione;
```

□ Scopo: assegnare il nuovo valore **espressione** alla variabile **nomeVariabile**

Nota:

- Java (come C e C++) utilizza il segno `=` per indicare l'assegnazione, creando una potenziale confusione con l'operatore di uguaglianza (che vedremo essere un doppio segno `=`, cioè `==`).
- Altri linguaggi usano simboli diversi per l'assegnazione (ad esempio, in linguaggio Pascal si usa `:=`)

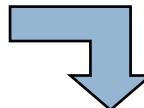
Assegnazione o definizione?

- 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



Un terzo esempio

```
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;
       double convertedDollars = dollars * 0.93;
       totalEuro = totalEuro + convertedDollars;
       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 mediante **enunciati di assegnazione**

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



L'assegnazione

- Analizziamo l'enunciato di assegnazione

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

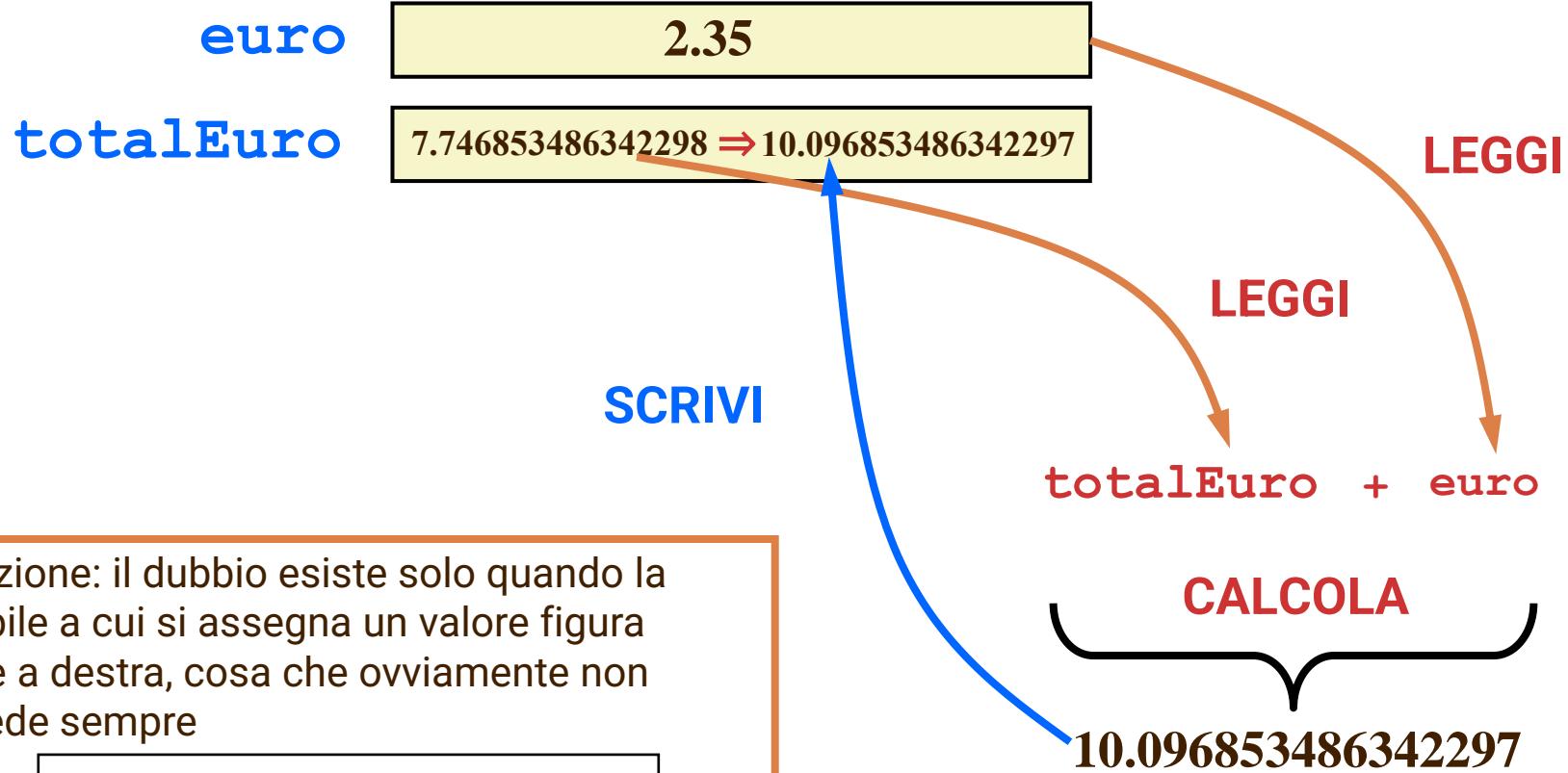
- Cosa significa? **Non** certo che la variabile **totalEuro** è **uguale** a se stessa più qualcos'altro...
- L'enunciato di assegnazione significa: Molto importante!

**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
=**
- Può essere utile **immaginare** la presenza di un simbolo diverso, distinto da quello matematico (che è fuorviante...)

```
totalEuro ← totalEuro + euro;
```

L'assegnazione

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





"Usare" una variabile

- Quando si dice "usare" una variabile (o, meglio, usare il **valore** di una variabile) ci si riferisce all'azione di "*lettura*" compiuta dal programma nella zona di memoria destinata alla variabile
- Questo avviene quando si scrive il nome della variabile all'interno di una **espressione**:
 - Nella parte destra di un enunciato di assegnazione o di inizializzazione
 - In un parametro (detto anche argomento) nell'invocazione di un metodo (vedremo più avanti)



Semantica del trasferimento di dati

- In tutta l'informatica (fatte salve eccezioni esplicite), il “**trasferimento di dati**” è una **COPIATURA** di dati:
- Al termine di un trasferimento di dati, le informazioni contenute nell'entità sorgente del trasferimento sono identiche a quelle che vi erano memorizzate prima del trasferimento stesso
- Detto in altri termini:
 - **Le operazioni di lettura non sono distruttive**



Semantica del trasferimento di dati

- Ciò è vero sia in hardware sia in software
 - Hardware: Il trasferimento di dati da una cella di memoria a un registro della CPU non altera il contenuto originario della cella
 - Software: L'esecuzione di un'operazione di lettura non altera il valore della variabile che viene letta



Semantica del trasferimento di dati

- Le **operazioni di scrittura**, che riguardano la **destinazione** di un trasferimento di dati, sono invece **completamente distruttive**
- Quindi, al termine di un trasferimento di dati, le informazioni contenute nell'entità **destinazione** del trasferimento non dipendono in alcun modo da quelle che vi erano (eventualmente) memorizzate prima del trasferimento stesso
- Si dice anche che l'operazione di scrittura “sovrascrive” l'eventuale informazione precedentemente contenuta nell'entità di destinazione



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Variabili non inizializzate



Variabili non inizializzate

- È buona regola fornire **sempre** un valore di inizializzazione nella **dichiarazione** di **variabili locali** (e quindi **definirle**)
- Cosa succede altrimenti? `int lit;`
- La dichiarazione di una variabile **crea** la variabile ma **non alloca spazio** in memoria e **non la inizializza** con un valore
- La variabile quindi **non** è utilizzabile in **lettura!**



Variabili non inizializzate

- In altri linguaggi di programmazione, come in **C** e **C++**, alle variabili locali al momento della dichiarazione viene riservato spazio in memoria **non inizializzato**
 - Questo spazio **non è vuoto**, bensì contiene un valore **randomico**
 - Se si utilizza in **lettura** una variabile locale non inizializzata, si legge quindi un valore “spazzatura”
 - Questo può portare ad comportamenti **imprevedibili** e ad errori (anche segmentation fault)
 - Questi errori (semantici) possono essere molto difficili da individuare a posteriori!



Variabili non inizializzate

- Il **compilatore** Java, invece, segnala come **errore** l'**utilizzo in lettura** di variabili locali a cui non sia mai stato assegnato un valore (mentre non è un errore la sola dichiarazione...)

```
public class Coins6 // 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);
    }
}
```

ERRORE

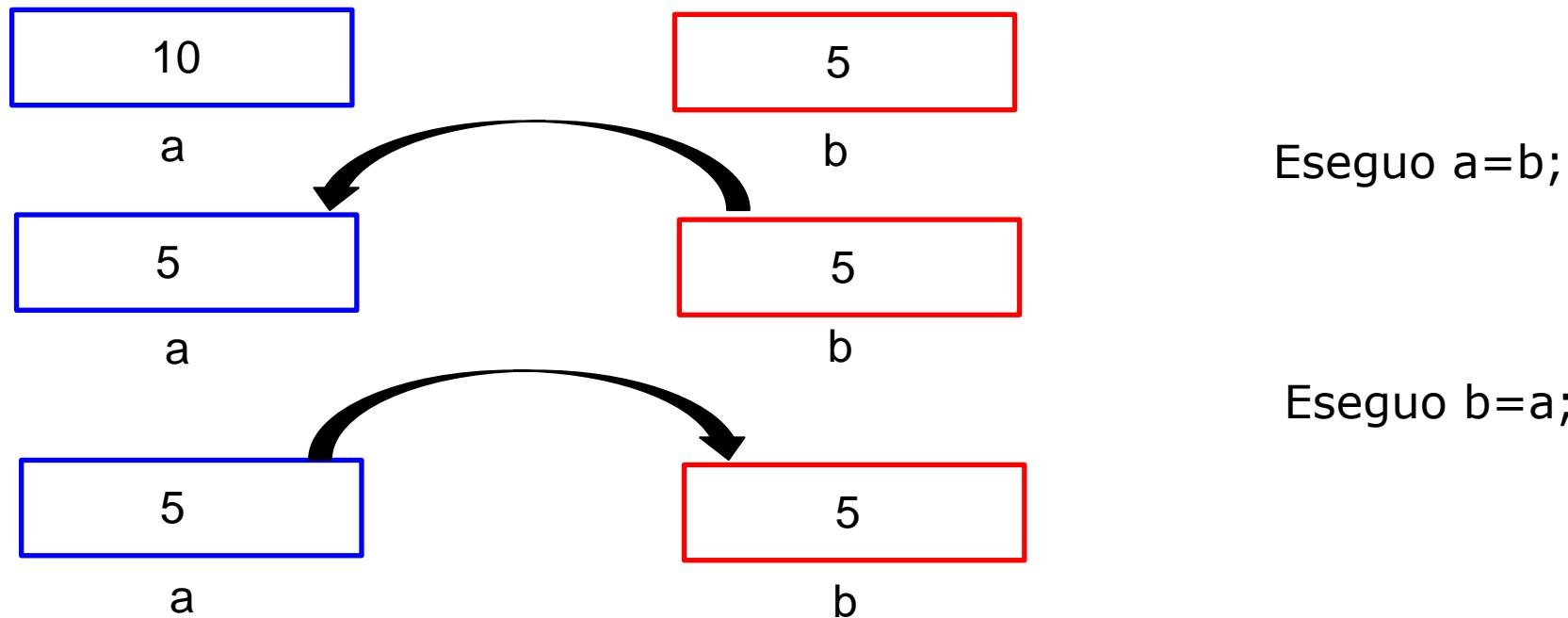
Coins6.java:5: variable lit might not have been initialized

Scambio di due variabili

- Se volessi scambiare il valore di due variabili... perché non posso fare:

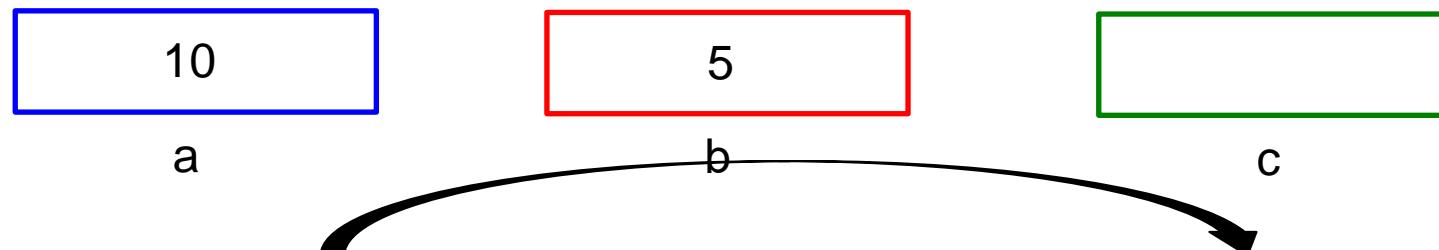
a=b; ?
b=a;

- L'operazione di scrittura è distruttiva, ovvero quando assegno un valore ad una variabile il precedente valore viene cancellato.

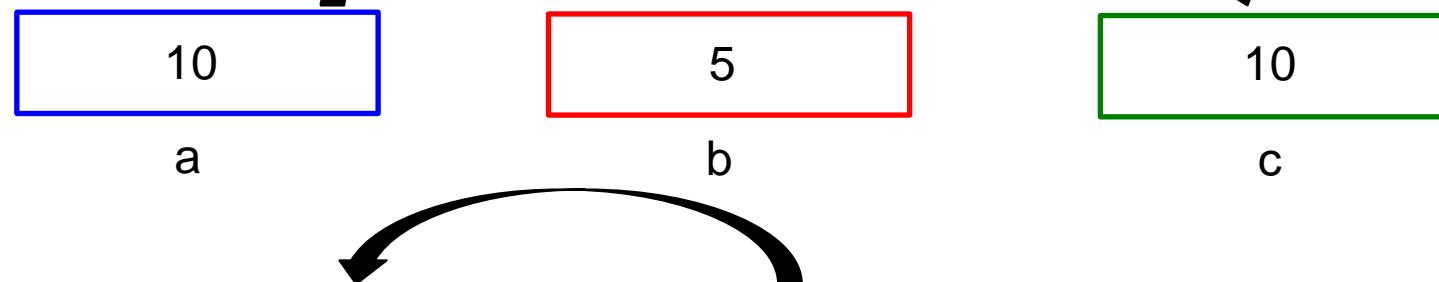


Scambio di due variabili

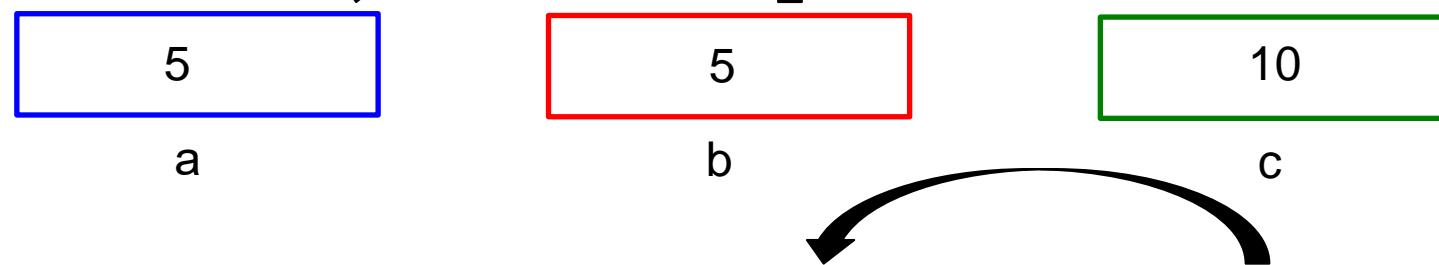
- Utilizzo una terza variabile, che chiamo "c" (oppure "help"):



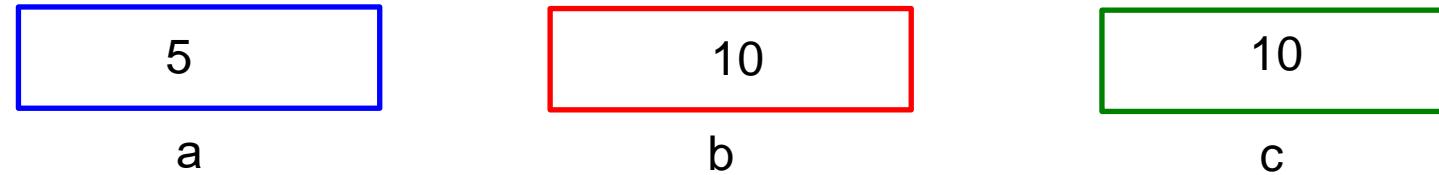
Eseguo $c=a$;



Eseguo $a=b$;



Eseguo $b=c$;





Scambio di due variabili

```
public class Exchange
{ public static void main(String[] args)
    { int a = 114;
        int b = 235;
        System.out.print("a = ");
        System.out.println(a);
        System.out.print("b = ");
        System.out.println(b);

        int c = a;
        a = b;
        b = c;
        System.out.print("a = ");
        System.out.println(a);
        System.out.print("b = ");
        System.out.println(b);
    }
}
```



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Costanti



Costanti

- Un programma per il cambio di valuta

```
public class Convert1
{ public static void main(String[] args)
    { double dollars = 2.35;
        double euro = dollars * 1.14;
        ...
    }
}
```

- Chi legge il programma potrebbe legittimamente chiedersi quale sia il significato del "**numero magico**" **1.14** 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 = 1.14;
        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 = 1.14;
        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



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Literals



Literals

- I valori costanti (=**literals**) per i numeri sono interpretati da Java come tipi **int** e **double**, a seconda che abbiano o meno la parte frazionaria
- Interi
 - Se il valore ricade nell'intervallo di un tipo di dato meno capace (byte o short) allora l'assegnamento avviene senza problemi, altrimenti errore in compilazione

```
byte value1 = 10;      //OK
byte value2 = 128;     //errore in compilazione
                      //max consentito 127

short value3 = 200;    //OK
short value4 = 33000;  // errore in compilazione
                      // max consentito 32767
```



Literals

□ Interi

- Se il valore è più grande del range del tipo **int**, ma rientra nel range del tipo **long**, devo esplicitare che si tratta di un valore long, aggiungendo **L** alla fine del numero

```
long l = 1345845486748064820;      ERRORE
long l = 1345845486748064820L;     OK
```



Literals

- Numeri in virgola mobile
 - Anche se il valore ricade nell'intervallo del tipo float ho errore in compilazione
 - Devo esplicitamente dire che il valore va interpretato come float per poterlo assegnare: aggiungo f alla fine del numero
 - Posso anche inizializzare un numero in virgola mobile con un valore intero: in questo caso semplicemente la parte frazionaria corrisponderà a 0.

```
float f1 = 2.35;      // ERRORE
float f2 = 2.35f;     // OK
float f3 = 2;         // OK
System.out.println(f3); // stampa: 2.0
```



Literals

- Nota: non ci sono solo valori costanti numerici. Anche le **stringhe** o i **caratteri** hanno “literals”;

```
char c = 'A' ;  
  
String name = "Alberto";
```



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Operatori aritmetici



Alcune note sintattiche

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

`lit / 1936.27`

- Il **punto** è il separatore decimale, invece della virgola (uso anglosassone)

`1936.27`

- **Non** bisogna indicare il punto separatore delle migliaia

`15000`

- I numeri in virgola mobile si possono anche esprimere in **notazione esponenziale** (la sintassi è un po' strana...)

`1.93E3 // vale 1.93 × 103`

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 **copie di parentesi TONDE** per indicare in quale ordine valutare sotto-espressioni

$$\boxed{a + b / 2} \neq \boxed{(a + b) / 2}$$

- In Java non esiste il **simbolo di frazione**, le frazioni vanno espresse "**in linea**", usando l'operatore di divisione (attenzione alle parentesi...)

$$\frac{a+b}{2} \longrightarrow \boxed{(a + b) / 2}$$

Operatore di divisione: Attenzione

- Quando **entrambi** gli operandi sono numeri **intei**, 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 **%** (il cui simbolo è stato scelto perché è simile all'operatore di divisione)





Funzioni più complesse

- La classe **Math** della libreria standard mette a disposizione **metodi statici** per il calcolo di tutte le funzioni algebriche e trigonometriche
 - **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

Invocazione di un 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**
 - Un metodo statico elabora o modifica solo i propri **parametri espliciti**



Invocazione di un metodo statico

Differenza tra metodi statici e non statici

Concetto

Appartengono a...

Richiedono un oggetto?

Agiscono su...

Analogia

Esempio Java

Uso tipico

Metodi Statici

Alla **classe**

No, si usano tramite il nome della classe

Dati **generali**, non legati a una singola istanza

Funzioni del **progetto di una macchina** (es. calcoli teorici)

Auto.kmToMiles(100)
→ conversione generica

Operazioni di utilità, costanti, factory methods

Metodi Non Statici

All'**oggetto (istanza)**

Sì, serve un'istanza

Dati **specifici** di quell'istanza

Azioni che una **macchina reale** può compiere

miaAuto.accelera(20)
→ cambia la velocità di *miaAuto*

Comportamenti dell'oggetto, logica interna



Esempio di firma di un metodo statico

```
public static double pow(double a, double b)
```

- *public*: il metodo può essere invocato in qualsiasi classe
- *static*: il metodo è statico
- *double*: tipo di dato **restituito**
 - è possibile che un metodo non restituisca dati, in questo caso il tipo del dato restituito è **void**
- *pow*: nome o identificatore del metodo
- *double a, double b*: **parametri esplicativi** del metodo

NB: nella classe Math c'è il codice che calcola la potenza, ma noi non lo vediamo, e non ci interessa vederlo! Ci basta sapere cosa calcola il metodo, non come

```
public final class Math{  
    ...  
    public static double pow(double a, double b){  
        ... // codice per calcolare e restituire a^b  
    }  
    ...  
}
```



Esempio di chiamata di metodo statico

- Se il metodo nella libreria Math è descritto da:

```
public static double pow(double a, double b)
```

- dove il parametro *a* è la base e *b* è l'esponente
- Quando lo utilizzo nel mio codice posso
 - passare direttamente i **valori**, ad es:

```
System.out.println(Math.pow(2, 3)); // Stampa 8.0
```

- oppure passare delle **variabili** double

```
double valore1 = 2;  
double valore2 = 3;  
System.out.println(Math.pow(valore1, valore2)); // Stampa 8.0
```



□ Metodi di **Math** vs **println**:

- **println** agisce su un oggetto (ad esempio, `System.out`)
 - Ricordiamo che per il momento, consideriamo gli oggetti come elementi da manipolare in un programma Java
- **pow** non agisce su un oggetto (Math è una classe)
 - Il metodo `Math.pow` è un **metodo statico**



I metodi statici della classe Math

- 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**, **pow**) iniziano con una lettera **minuscola**
 - oggetti e metodi si distinguono perché **solo i metodi sono sempre seguiti dalle parentesi tonde**

oggetto

```
graph TD; oggetto --> System[System.out.println(...)]; classe --> System; metodo[metodo dell'oggetto out] --> println((...))
```

System.out.println(...)

classe

metodo dell'oggetto out

Math.pow(...)

```
graph TD; classe --> Math[Math.pow(...)]; static[Metodo statico della classe Math] --> pow((...))
```

classe

Metodo statico della classe Math



Costanti della classe Math

- Nella classe Math sono definite alcune 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 risultato**

```
totalEuro = totalEuro + dollars * 1.41;
```

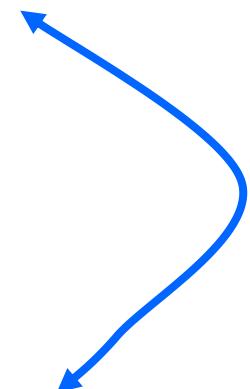
- Questa operazione è talmente comune nella programmazione, che il linguaggio Java fornisce una scorciatoia

```
totalEuro += dollars * 1.41;
```

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--;
```



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

La documentazione della libreria standard



Guida in linea

- Le classi della libreria Java sono migliaia
Non serve usare la memoria, usiamo la **documentazione!**
- L'ambiente **JDK** fornisce la **documentazione API (Application Programming Interface)** per l'utilizzo delle **classi della libreria standard**
 - ▣ È disponibile **online**:
 - <https://docs.oracle.com/en/java/javase/21/docs/api/index.html>
 - ▣ È disponibile **online** anche sul sito dell'aula Taliercio (NB: in aula taliercio c'è la versione 11 di Java)



Guida in linea

- JDK inoltre fornisce:
 - tutta la documentazione per l'**utilizzo delle classi della libreria standard**
 - alcuni documenti in formato “**tutorial**” per la descrizione delle funzionalità di interi pacchetti
 - esempi di programmi (“**demo**”)



SEARCH:



Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification

This document is divided into two sections:

Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.

JDK

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

All Modules | **Java SE** | **JDK** | Other Modules

Module	Description
<code>java.base</code>	Defines the foundational APIs of the Java SE Platform.
<code>java.compiler</code>	Defines the Language Model, Annotation Processing, and Java Compiler APIs.



Guida in linea

- La descrizione di una **classe** comprende
 - una sintetica descrizione testuale delle motivazioni alla base del progetto della classe e delle modalità del suo utilizzo
 - eventuali **costanti** e **campi static** accessibili
 - l'elenco di tutti i suoi **costruttori** e **metodi** pubblici
- 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



La classe Math

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#) [ALL CLASSES](#)
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3
java.lang

Class Math

java.lang.Object
java.lang.Math

```
public final class Math
extends Object
```

The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Unlike some of the numeric methods of class StrictMath, all implementations of the equivalent functions of class Math are not defined to return the bit-for-bit same results. This relaxation permits better-performing implementations where strict reproducibility is not required.

By default many of the Math methods simply call the equivalent method in StrictMath for their implementation. Code generators are encouraged to use platform-specific native libraries or microprocessor instructions, where available, to provide higher-performance implementations of Math methods. Such higher-performance implementations still must conform to the specification for Math.

Field Summary

Fields	Modifier and Type	Field and Description
	static double	E The double value that is closer than any other to <i>e</i> , the base of the natural logarithms.
	static double	PI The double value that is closer than any other to <i>pi</i> , the ratio of the circumference of a circle to its diameter.

Method Summary

All Methods	Static Methods	Concrete Methods
Modifier and Type	Method and Description	
static double	abs(double a) Returns the absolute value of a double value.	
static float	abs(float a) Returns the absolute value of a float value.	
static int	abs(int a) Returns the absolute value of an int value.	



La classe Math

Method Detail

sin

```
public static double sin(double a)
```

Returns the trigonometric sine of an angle. Special cases:

- If the argument is NaN or an infinity, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:

a - an angle, in radians.

Returns:

the sine of the argument.

cos

```
public static double cos(double a)
```

Returns the trigonometric cosine of an angle. Special cases:

- If the argument is NaN or an infinity, then the result is NaN.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:

a - an angle, in radians.



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

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
- Per usare una classe di una libreria, bisogna **importarla** nel programma, usando l'enunciato

```
import nomePacchetto.NomeClasse;
```

- Le classi **System**, **Math** e **String** appartengono al pacchetto `java.lang`
- Il pacchetto **java.lang** viene importato **automaticamente**



Importare una classe di 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
```



Stili per l'importazione delle classi

- Usare un enunciato import per ogni classe importata

```
import java.math.BigInteger;  
import java.math.BigDecimal;
```

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

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

- non è un errore importare classi che non si usano...
- MA... 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?



Stili per l'importazione delle classi

- È possibile non usare per nulla gli enunciati import, ma indicare sempre il nome completo delle classi utilizzate nel codice

```
java.math.BigInteger a =  
    new java.math.BigInteger("123456789");
```

- Questo stile è poco usato
 - + errori di battitura
 - + lunghezza del codice
 - + noioso
 - - leggibile