



Fondamenti di Informatica

(L-Z)

Corso di Laurea in Ingegneria Gestionale

Java: Fondamenti

Prof. Stefano Mariani
Dott. Alket Cecaj



- Introduzione
- Strumenti di sviluppo
- Variabili e costanti
- Tipi di dato e operatori
- Istruzioni per il controllo di flusso
- Il concetto di “scope”
- Classi e librerie
- Gestione I/O
- Varie ed eventuali =)



INTRODUZIONE



Flashback: Perché Java?

- Attualmente è:
 - ▶ standard per applicazioni **Web** (mobile computing)
 - ▶ uno dei linguaggi di **referimento** per l'ICT
 - ▶ il linguaggio di riferimento per la piattaforma **Android**
- Inoltre:
 - ▶ è il linguaggio più “pulito”, **didatticamente** più adatto all'introduzione dei concetti base di programmazione

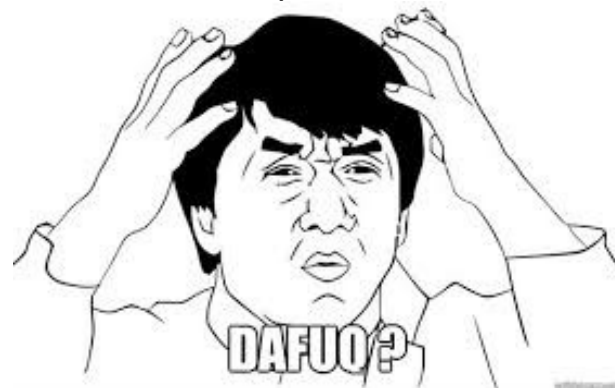
Feb 2018	Feb 2017	Change	Programming Language	Ratings	Change
1	1		Java	14.988%	-1.69%
2	2		C	11.857%	+3.41%
3	3		C++	5.726%	+0.30%
4	5	▲	Python	5.168%	+1.12%
5	4	▼	C#	4.453%	-0.45%
6	8	▲	Visual Basic .NET	4.072%	+1.25%

<http://www.tiobe.com/tiobe-index/> (febbraio 2018)



Cos'è Java?

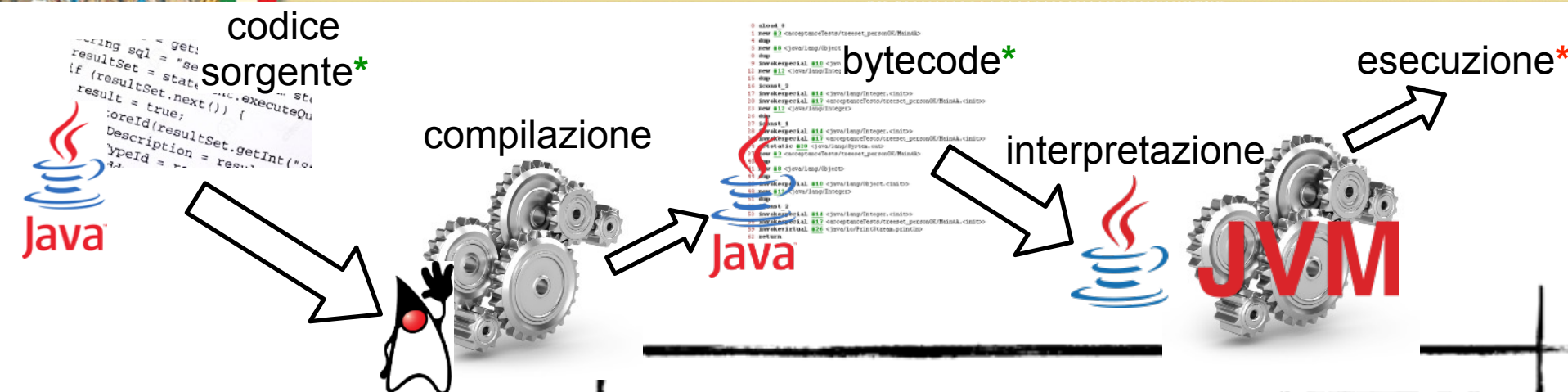
- Java è un **linguaggio imperativo**
 - orientato al *comando*
 - un programma è composto da una *sequenza di comandi*
- *Managed* programming language
 - non esegue direttamente sull'hardware (o meglio, sul S.O.) del computer, ma su una *piattaforma* (**Java Virtual Machine - JVM**) che fornisce un livello di astrazione più elevato
- Il codice Java viene *compilato* in un formato intermedio detto **bytecode**
 - che viene poi *interpretato* dalla JVM



La JVM permette di eseguire
lo stesso codice compilato su più S.O.



Flashback: Linguaggi di programmazione



La JVM è un'applicazione che esegue tra il livello applicativo e il S.O.

*Indipendente dal computer (S.O.)

*Dipendente dal computer (S.O.)



Quali comandi?

- I *comandi* Java sono composti da:
 - **parole riservate**, e.g.: public, private, for, if, ...
 - **costanti**, e.g.: 3, 5.8, "hello", ...
 - **simboli speciali e operatori**, e.g.: +, -, %, {}, [], =, ..., ;*
 - **separatori**, e.g.: lo spazio bianco
 - **identificatori**, e.g.: x, var, pippo, my_var, ... (qualunque altra cosa in pratica)
- Tutti gli elementi sono *case-sensitive*
 - var è diverso da VAR e da VaR
- Oltre ai comandi ci possono essere dei **commenti**
 - *non vengono considerati dal compilatore* (non sono "programma")
 - qualunque cosa scritta tra /* è un commento anche multi-linea */
 - usare // per commenti su singola linea

*Il ";" termina un'istruzione, va messo alla fine di ogni riga (a meno che non ci sia "{")



Dove li metto?

- Ogni sequenza di comandi in Java deve stare dentro un **metodo**

▸ da Wikipedia:

Un **metodo** (o anche **funzione membro**), in **informatica**, è un termine che viene usato principalmente nel contesto della **programmazione orientata agli oggetti** per indicare un **sottoprogramma** associato in modo esclusivo ad una **classe** e che rappresenta (in genere) un'operazione eseguibile sugli **oggetti** e istanze di quella classe.



- *Per ora**

- una **classe** è l'astrazione di un *concetto*, un *tipo di dato* che rappresenta uno **stato**, descritto da una serie di **attributi**, e un comportamento, descritto da una serie di **metodi**
 - e.g. la classe “docente”
- un **oggetto** è una specifica *istanza* di una classe, con un preciso stato e un preciso comportamento
 - e.g. lo specifico docente “Stefano Mariani”

*Le definizioni che vi do vogliono essere un buon compresso tra *correttezza* e *semplicità*



In pratica?

- In pratica un **programma** Java
 - ha per forza *almeno una* classe
 - che ha per forza *almeno un* metodo
- E gli oggetti? No?
 - *non necessariamente*
 - per ora non ci interessa il perché, lo vedremo a tempo debito =)
- Java si definisce comunque *orientato agli oggetti* (**Object Oriented**)

```
public class HelloWorld {  
    public static void main (String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

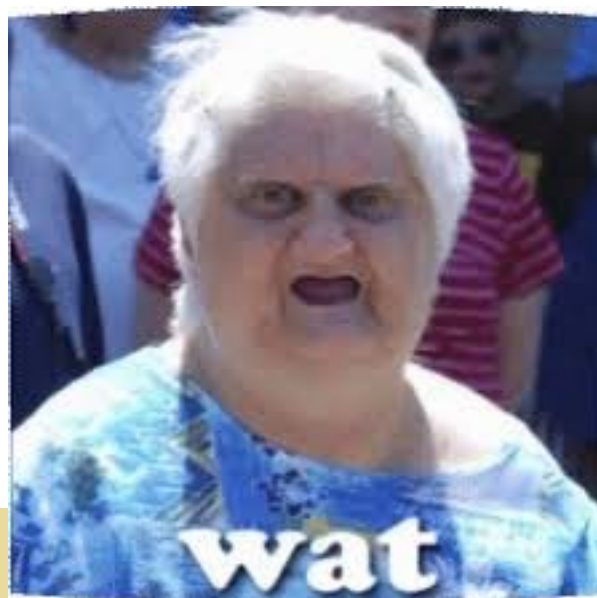
Classe

Metodo

“Comando” (o meglio, “statement”)



STRUMENTI: JDK, JRE, IDE





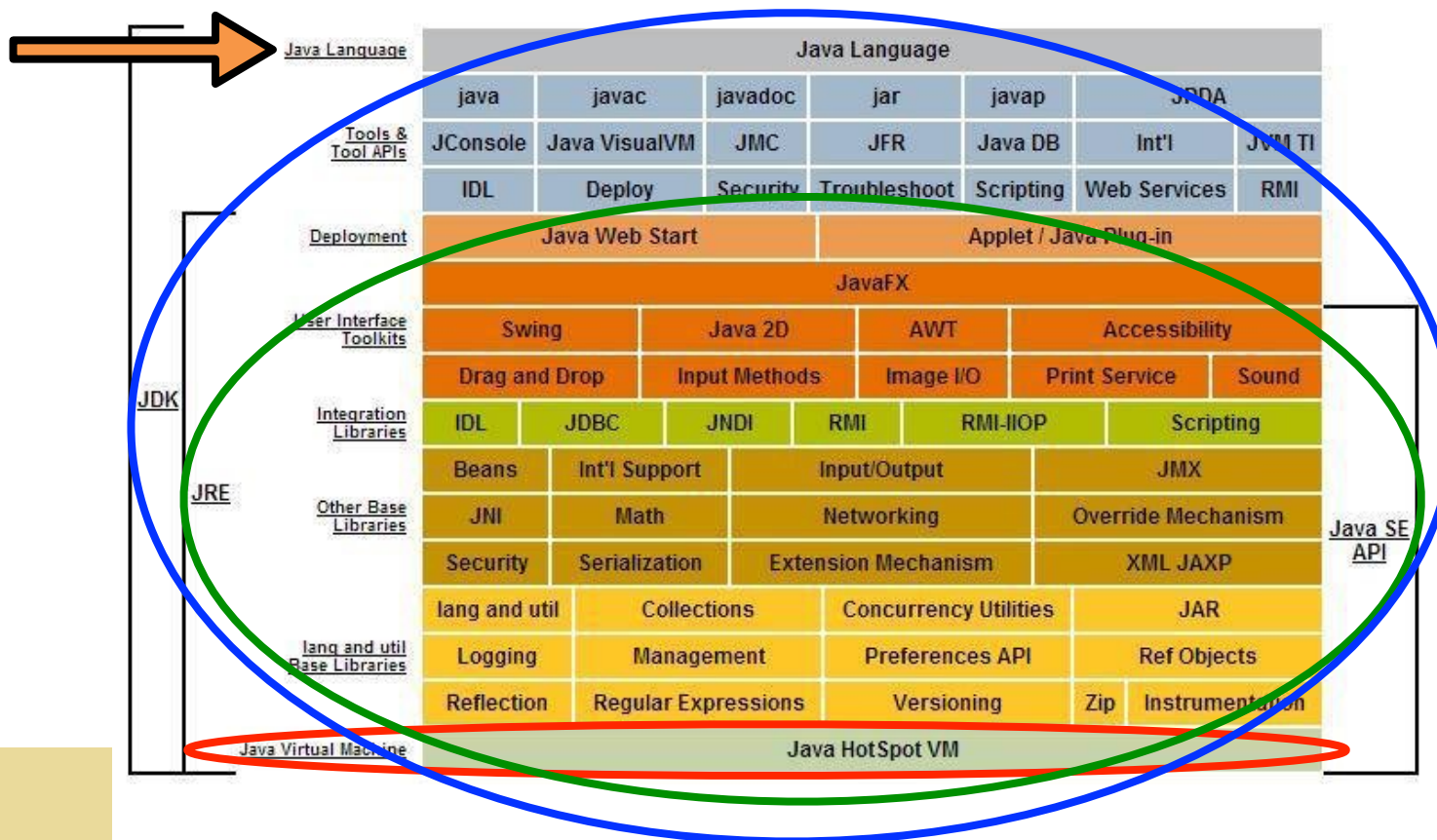
- *Managed* programming language
 - non esegue direttamente sull'hardware (o meglio, sul S.O.) del computer, ma su una *piattaforma* (**Java Virtual Machine - JVM**) che fornisce un livello di astrazione più elevato
- Per scrivere programmi Java vi serve il **Java Development Kit (JDK)**
 - <http://www.oracle.com/technetwork/java/javase/downloads/jdk9-downloads-3848520.html>
- Per eseguire programmi Java vi serve il **Java Runtime Environment (JRE)**
 - <http://www.oracle.com/technetwork/java/javase/downloads/jre9-downloads-3848532.html>
 - è compreso nel JDK :)

I link sono anche sul sito del corso, nella sezione “Strumenti software”



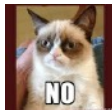
Riassumendo

- **JDK** = JRE + tutto ciò che serve per *sviluppare* (scrivere codice per) applicazioni Java
- **JRE** = JVM + tutto ciò che serve per *eseguire* applicazioni Java
- **JVM** = tutto ciò che serve al JRE per “parlare” coi vari S.O. usando *uno stesso linguaggio*





Scriviamo codice in Word?

- Il **codice sorgente** di un qualunque programma non è altro che una serie di parole, potreste usare anche *Microsoft Word* per scriverlo... 
- Scrivere codice è un'attività *complessa*, che richiede strumenti adeguati
 - un meccanico non vi aggiusta la macchina a mani nude...
- Uno dei *tanti strumenti* a disposizione di ogni programmatore è **l'IDE**
 - **Integrated Development Environment** (Ambiente Integrato di Sviluppo)
 - un software che aiuta nello sviluppo del codice *segnalando errori* di sintassi, supportando il *debugging* (correzione degli errori) e il *deployment* (messa in esecuzione), etc.
- Io userò Eclipse (free, open source, ampia community)
 - <https://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/oxygen3rc2>
 - tante altre alternative possibili*

**Sul sito del corso, nella sezione "Strumenti software"*



VARIABILI E COSTANTI: dichiarazione e assegnamento



Costante vs. Variabile

- Una **costante** è un dato che *non cambia* valore durante l'esecuzione del programma
 - ▶ e.g. 3, "ciao", 3.14, ...
- Una **variabile** è un dato che *cambia* valore durante l'esecuzione del programma
 - ▶ e.g. `int x`, `MyType obj`, `boolean flag`, ...
- La **dichiarazione** di una variabile associa un *identificatore* (nome logico) ad un **tipo** di dato
 - ▶ e.g. `int x` indica che la variabile `x` è di tipo `int`
- L'**assegnamento** associa alla variabile un **valore**
 - ▶ e.g. `int x = 10`, `String greet = "hello"`, ...

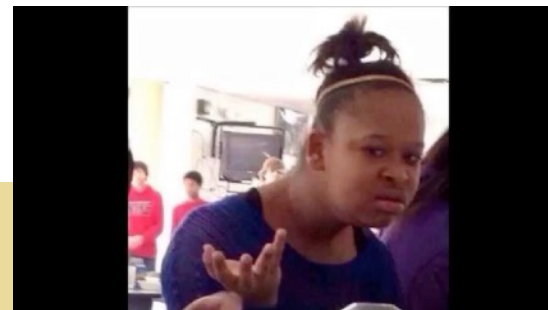




- La dichiarazione può essere seguita dall'**inizializzazione** (ovvero, il primo assegnamento)
 - e.g. `String greet = "hello"`
- L'operatore di assegnamento in Java è l'operatore `=`
 - `int a;`
`int b;` // dichiarazione
 - `int c = 0;` // dichiarazione + assegnamento = inizializzazione
 - `b = 1;` // assegnamento
 - `a = b;` // assegnamento (a assume il valore di b *in quell'istante*)
 - `b = c;` // assegnamento (b diventa 0, a resta 1*)

*Perchè, doveva venirci un dubbio??

Ne riparlamo tra qualche slide =)





TIPI DI DATO E OPERATORI



Il concetto di tipo

- Java è un linguaggio *imperativo*, *orientato agli oggetti*, e a **tipizzazione statica forte**
- Tipizzazione
 - ogni dato è di un certo tipo
 - i **tipi primitivi** descrivono strutture dati *atomiche*
 - non ulteriormente scomponibili
 - i **tipi strutturati** descrivono strutture dati *composte*
 - scomponibili in tipi strutturati e/o primitivi
- Statica
 - il tipo di un dato è stabilito nel *codice sorgente* (non a run-time)
- Forte (*type-safety*)
 - o in fase di *compilazione*, o in fase di *esecuzione*, la JVM esegue controlli di tipo per stabilire l'*ammissibilità* dei comandi





- Numeri *interi*
 - **int** : un numero intero codificabile con 32 bit (2^{32} numeri, negativi inclusi)
 - **long** : un numero intero codificabile con 64 bit (2^{64} numeri, negativi inclusi)
- Numeri *decimali*
 - **float** : decimale in *virgola mobile** a 32 bit
 - **double** : decimale in virgola mobile a 64 bit
- Numeri *booleani*
 - **boolean** : true o false
- *Caratteri*
 - **char** : intero che rappresenta un carattere in *codice ASCII*
 - singoli caratteri racchiusi tra *singoli apici* (e.g. '1', 'x', '?', ...)
 - *sequenze di escape*, ossia caratteri speciali (e.g. '\n' va a capo)

E.g. **int** x = 2.5

- è errato
- viene segnalato dal *compilatore*

*Notazione simile alla notazione scientifica decimale: Segno + mantissa + esponente



- Qualunque tipo Java che non sia un tipo primitivo
 - ▶ e.g. **String** : una qualunque stringa di testo
 - rappresentata internamente come *array di caratteri*
 - vanno racchiuse tra *doppi apici* (e.g. `String greet = "Hello \n world!"`)
 - ▶ e.g. gli **array** : un qualunque tipo seguito da "**[]**"
 - e.g. `int[] array;`
 - e.g. `String[N] array = new String{"a", "b", "c", "d", "e"};`
 - ▶ in generale, qualunque **classe** Java
 - ogni classe Java definisce infatti *un tipo* di dato
 - gli oggetti di una *stessa classe* sono dunque tutti dello *stesso tipo*



- **Aritmetici**

- $+$, $-$, $*$, $/$, $\%$
- $/$ funziona diversamente per interi e decimali
 - $5 / 2 = 2$ vs. $5.0 / 2 = 2.5$

- **Incremento/Decremento**

- $++$, $--$ (di una unità)

- **Assegnamento**

- $=$, $+=$, $-=$, $*=$, $/=$
- assegnamento e assegnamento + operazione (e.g. $x += y$ equivale a $x = x + y$)

- **Relazionali** (di confronto)

- $>$, $>=$, $==$, $<$, $<=$, $!=$
- maggiore, maggiore o uguale, uguale, minore, minore o uguale, diverso

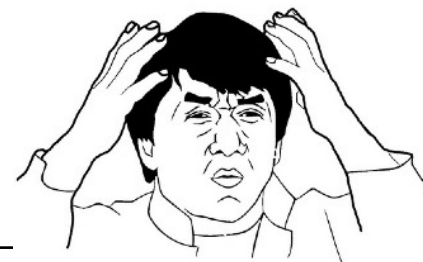
- **Logici** (composizione di confronti)

- $\&\&$, $||$, $!$
- AND logico, OR logico, NOT logico





- Alcuni operatori assumono un *diverso significato* se usati su oggetti (dati) di tipo String
 - ▶ **+** diventa la **concatenazione**
 - e.g. `String s = "One is " + 1` *equivale* a `String s = "One is 1"`
 - ▶ **==** viene *sostituito* da un'invocazione al **metodo** `equals()`
 - `==` non si comporta come ci aspetteremmo



```
String s1 = "babbo";  
String s2 = "babbo";  
boolean isSame = s1 == s2;  
// isSame è false!
```

```
String s1 = "babbo";  
String s2 = "babbo";  
boolean isSame = s1.equals(s2);  
// isSame è true!
```



Perchè?? (pt.1)

- **Casting**

- ▶ operazione con cui si *converte* una variabile di un certo tipo a diventare di un altro tipo
- ▶ la *concatenazione* fa casting* *automaticamente*

```
double d = 5.5;  
int i = (int) d; // i = 5!
```

```
double d = 5.5;  
String s = "Val is " + d; // s = "Val is 5.5"
```

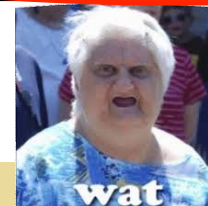
- Non tutti i casting sono ammessi

- ▶ in alcuni casi, Java ci aiuta

```
String num = "3.14";  
double d = (double) num;
```

```
String num = "3.14";  
double d = Double.parseDouble(num);
```

Il compilatore segnala l'errore
già in fase di scrittura



*Non è proprio così, ma per ora fingiamo sia così

A tempo debito ne parleremo =)



Perchè?? (pt.2)

- I tipi di Java possono essere divisi in due categorie
 - ▶ tipi **per valore** (e.g. i tipi *primitivi*)
 - rappresentano valori *senza stato*, dunque gli operatori creano *sempre nuovi* valori
 - ▶ tipi **per riferimento** (*qualunque* altra cosa)
 - rappresentano valori *con stato*, *modificabile* attraverso operatori e metodi
- Motivazione
 - ▶ dato un array di N elementi, cambiarne *un* elemento comporterebbe la creazione di un *nuovo* array!
 - *estremamente inefficiente*



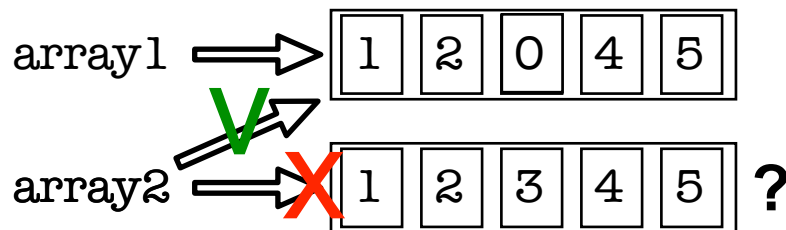
Perchè?? (pt.3)

- Quando si (ri)assegna il contenuto di una variabile
 - se il contenuto è di tipo **per valore** (primitivo)
 - il valore viene *copiato* e sovrascritto / creato ex-novo
 - se invece è di tipo **per riferimento** (oggetto)
 - il valore viene *riferito*
- Ricordate?

`b = c; // assegnamento (b diventa 0, a resta 1)`

 - in quel caso a, b e c erano tutte variabili primitive, per valore, dunque è ovvio che a resti 1
- Ma qua invece?

```
int[] array1 = {1,2,3,4,5};  
int[] array2 = array1;  
array1[2] = 0;  
boolean flag array2[2] == 0;  
// quanto vale flag?
```





CONTROLLO DI FLUSSO



- Le **istruzioni di controllo** permettono di modificare l'*esecuzione sequenziale* del programma

- ▶ **if-then-else**

- test booleano (true/false)

- ▶ **switch-case**

- test “multi-caso” (case1, case2, ..., caseN)

- ▶ **while**

- ciclo condizionale (while <condition> do <action>)

- ▶ **do-while**

- variante “at least once” del while (do <action> while <condition>)

- ▶ **for**

- ciclo con indice/contatore (for <index/counter> in <range> do ...)





If-then-else

- **Sintassi:**

```
if (<condition>) {  
    <action1>;  
    <action2>;  
    ...  
} else {  
    <another-action1>;  
    <another-action2>;  
    ...  
}
```

- **E.g.:**

```
if (x == y) {  
    z = true;  
    w = false;  
} else {  
    z = false;  
    w = true;  
}
```

- Le istruzioni “if-then-else” possono essere *innestate* arbitrariamente

- ▶ come praticamente *qualunque altra* istruzione di controllo (switch, while, for, etc.)

```
if (x == y) {  
    if (x == z) {  
        if (x == w) {  
            ...  
        }  
    }  
} else {  
    if (...) {  
        ...  
    }  
}
```



If-then-else

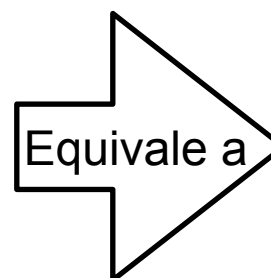
- La parte else dell'if è *opzionale* (si usa se serve)

```
if (<condition>) {  
    <action1>;  
    <action2>;  
    ...  
}
```

```
if (x == y) {  
    z = true;  
    w = false;  
}
```

- ▶ **<action>** può essere una qualunque istruzione Java
- ▶ **<condition>** una qualunque composizione di *operatori logici e relazionali*

```
if (x == y) {  
    z = true;  
    w = false;  
} else {  
    z = false;  
    w = true;  
}
```

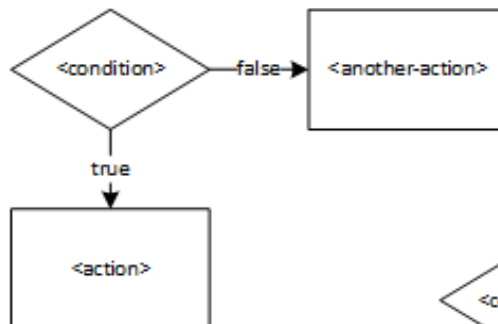


```
if (x == y) {  
    z = true;  
    w = false;  
}  
if (x != y) {  
    z = false;  
    w = true;  
}
```

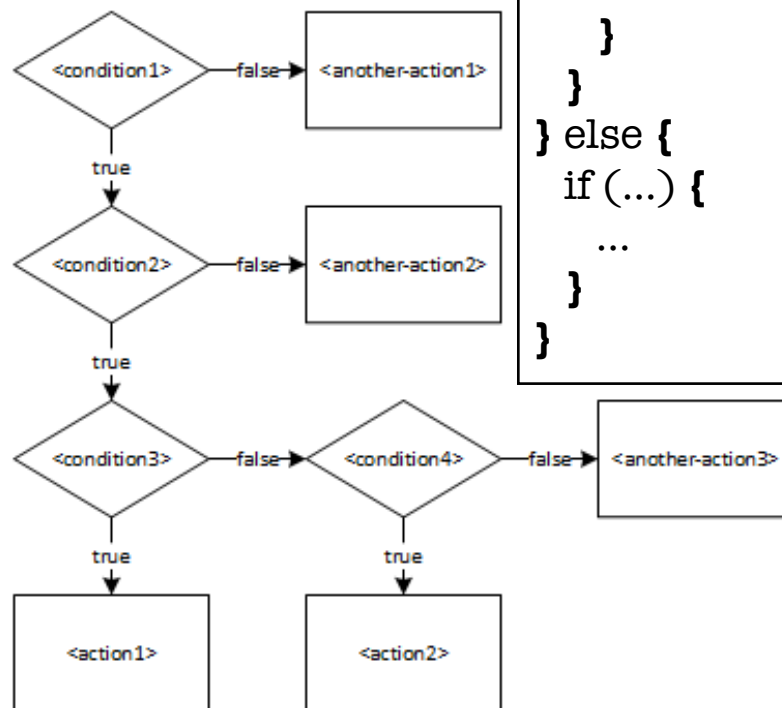
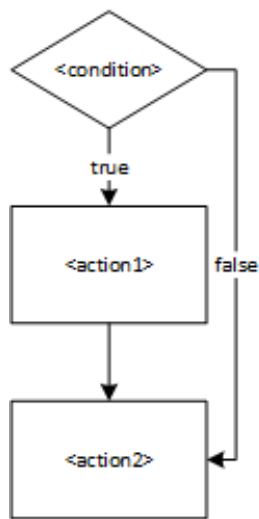


Codice vs. Flow charts

```
if (<condition>) {  
  <action1>;  
  <action2>;  
  ...  
} else {  
  <another-action1>;  
  <another-action2>;  
  ...  
}
```



```
if (<condition>) {  
  <action1>;  
  ...  
}  
<action2>;
```



```
if (x == y) {  
  if (x == z) {  
    if (x == w) {  
      ...  
    }  
  }  
} else {  
  if (...) {  
    ...  
  }  
}
```



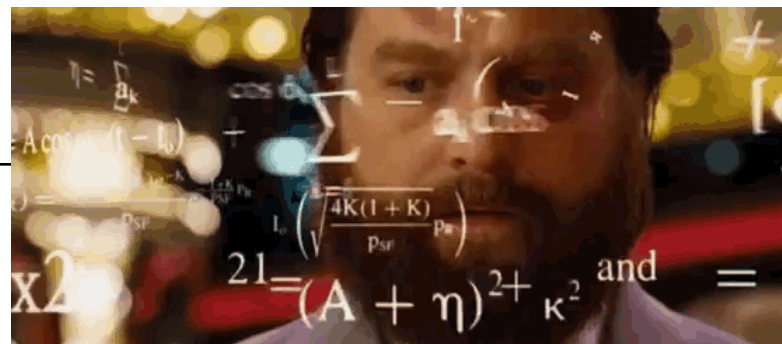
- **Relazionali** (di confronto)

- ▶ $>$, $>=$, $==$, $<$, $<=$, $!=$
- ▶ maggiore, maggiore o uguale, uguale, minore, minore o uguale, diverso

- **Logici** (composizione di confronti)

- ▶ $\&\&$, $||$, $!$
- ▶ AND logico, OR logico, NOT logico

```
if (x == y && x != z) {  
    ...  
} else {  
    ... // finisco qua se x != y oppure se x == z  
}
```



```
if (x == y || x != z) {  
    ...  
} else {  
    ... // finisco qua se x != y e se x == z  
}
```



Switch-case

- **Sintassi:**

```
switch (<variable>) {  
    case <value1>:  
        <action1>;  
    case <value2>:  
        <action2>;  
    ...  
    default:  
        <actionDef>;  
}
```

E.g.:

```
switch (num) {  
    case 0:  
        x += num;  
        break;  
    case 1:  
        x -= num;  
        break;  
    ...  
    default:  
        x = -1;  
}
```

- ▶ **default** viene eseguito in caso nessun case esegua
- ▶ **<variable>** può essere int, char, String*
- ▶ **break** evita il cosiddetto “*fall-through*”
 - interrompe lo switch (esce dallo switch)
 - senza, tutti i case vengono eseguiti in sequenza

*E altri tipi composti Java, ma noi non li useremo...



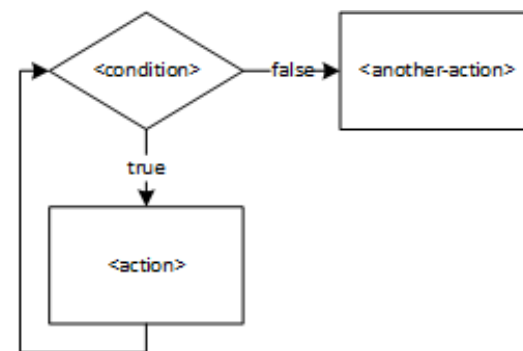
While

- **Sintassi:**

```
while (<condition>) {  
    <action1>;  
    <action2>;  
    ...  
}
```

E.g.:

```
while (x < N) {  
    x += x;  
}
```



- **Iterazione**

1. si valuta la condizione del ciclo (*invariante*):

- se *vera*, si esegue il corpo del ciclo
- se *falsa*, si passa all'istruzione successiva (dunque saltando il corpo)

2. al termine del corpo, si *torna* alla valutazione della condizione

- **Terminazione***

- affinché il ciclo termini è necessario che almeno una delle istruzioni nel corpo modifichi (opportunamente) almeno una variabile coinvolta nella condizione

**Condizione valida per tutti i cicli che vedremo*



Do-while

- **Sintassi:**

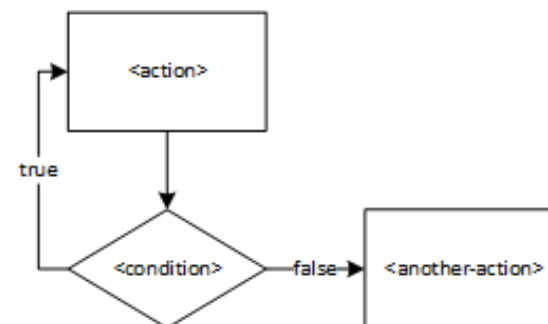
```
do {  
  <action1>;  
  <action2>;  
  ...  
} while (<condition>;
```

E.g.:

```
do {  
  x += x;  
} while (x < N);
```

- **Iterazione**

1. si esegue l'azione
2. si valuta la condizione del ciclo (*invariante*):
 - se vera, si torna alla valutazione della condizione
 - se falsa, si passa all'istruzione successiva





For

- **Sintassi:**

```
for (<variable>; <condition>; <inc/dec>) {  
    <action1>;  
    <action2>;  
    ...  
}
```

- E.g.:

```
for (int i = 0; i < N; i++) {  
    array[i] = i;  
}
```

- **Iterazione**

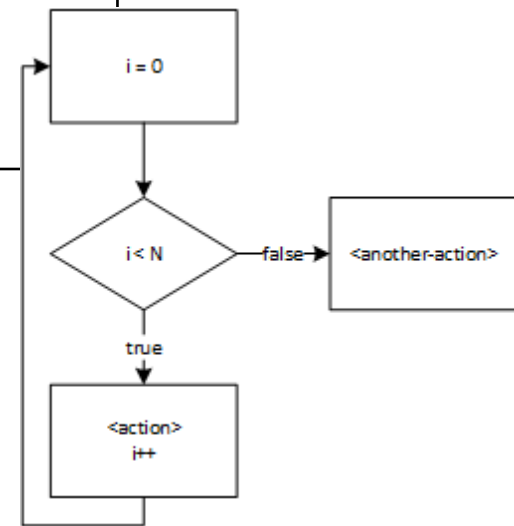
1. si valuta la condizione del ciclo (*invariante*):

- se *vera*, si esegue il corpo del ciclo
- se *falsa*, si passa all'istruzione successiva (dunque saltando il corpo)

2. al termine del corpo, si *torna* alla valutazione della condizione

- **Terminazione**

- già inclusa nella dichiarazione del for (coppia <condition> & <inc/dec>)





For: osservazioni

- La variabile d'iterazione non deve essere esplicitamente incrementata/decrementata

- ▶ a meno che non sia necessario per il programma

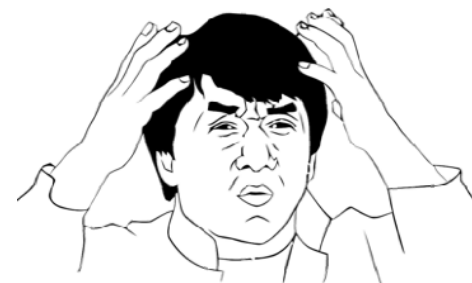
```
for (int i = 0; i < N; i++) {  
    array[i] = i;  
}
```

- <variable>, <condition> e <inc/dec> possono specificare *più di una espressione*

```
for (int i = 0, int j = N; i < N && j >= 0; i++, j--) {  
    array[i] = j;  
}
```

- Se <variable> è una dichiarazione, la variabile non “*sopravvive*” alla fine del ciclo (a breve vedremo perché)

```
int i;  
for (i = 0, int j = N; i < N && j >= 0; i++, j--) {  
    array[i] = j;  
}  
  
array[i] = ... // errore: i vale N!  
array[j] = ... // errore: quì j “non esiste”
```





Break e Continue

- Istruzioni che *modificano l'esecuzione di un ciclo*

- ▶ while, do-while, for

- **break**

- ▶ esce *immediatamente* dal ciclo corrente

- se ciclo innestato, esce solamente da quello più interno

- **continue**

- ▶ passa *immediatamente* alla prossima iterazione del ciclo corrente

- se ciclo innestato, passa alla prossima iterazione del ciclo più interno

```
for (int i = 0; i < N; i++) {  
    array[i] = i;  
    if (i > N/2) {  
        break;  
    }  
}
```

```
for (int i = 0; i < N; i++) {  
    if (i < N/2) {  
        continue;  
    } else {  
        array[i] = i;  
    }  
}
```



IL CONCETTO DI “SCOPE”



- **break**

- ▶ ...

- se ciclo innestato, esce solamente da quello più interno

- **continue**

- ▶ ...

- se ciclo innestato, passa alla prossima iterazione del ciclo più interno

```
for (int i = 0; i < N; i++) {  
    for (int j = N; j >= 0; j--) {  
        array[i] = j;  
        if (j < N/2) {  
            break;  
        }  
    }  
}
```

```
for (int i = 0; i < N; i++) {  
    if (i < N/2) {  
        continue;  
    } else {  
        for (int j = N; j >= 0; j--) {  
            array[j] = i;  
        }  
    }  
}
```



Lo “scope” delle istruzioni

```
for (int i = 0; i < N; i++) {  
    for (int j = N; j >= 0; j--) {  
        array[i] = j;  
        if (j < N/2) {  
            break;  
        }  
    }  
}
```

- Lo **scope** (*visibilità*) del **break** è il ciclo più *interno* (innestato)
 - ▶ in particolare, il blocco (delimitato da “{...}”) di istruzioni in cui si trova
 - ▶ dunque è in quello scope che il suo effetto si manifesta

```
for (int i = 0; i < N; i++) {  
    if (i < N/2) {  
        continue;  
    } else {  
        for (int j = N; j >= 0; j--) {  
            array[j] = i;  
        }  
    }  
}
```

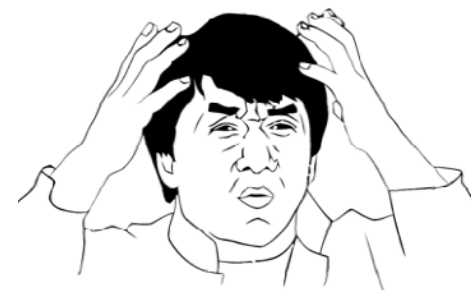
- Lo scope del **continue** è il ciclo più *esterno*
 - ▶ in particolare, il blocco di istruzioni in cui si trova (“{...}”)



Ricordate?

- Se <variable> è una dichiarazione, la variabile non “sopravvive” alla fine del ciclo (a breve vedremo perché)

```
int i;  
for (i = 0, int j = N; i < N && j >= 0; i++, j--) {  
    array[i] = j;  
}  
array[i] = ... // errore: i vale N!  
array[j] = ... // errore: quì j “non esiste”
```



- Lo **scope** di una *variabile* è definito allo stesso modo
 - ovvero, il blocco di istruzioni in cui si trova
 - dunque è in quello scope che la variabile “*esiste*”, è “accessibile”



- Una variabile è visibile solo dopo la sua *definizione*
- Le variabili definite in un blocco sono visibili in tutti i blocchi in esso *contenuti*
- All'interno di uno stesso blocco non possono esserci due variabili con lo *stesso nome*
 - ▶ in blocchi distinti, sì

```
...  
i = 10; // errore: i mai definita
```

```
while (i < N) {  
    while (j >= 0) {  
        j += i; // ok: i visibile  
    }  
}
```

```
while (i < N) {  
    int i = 0; // errore: I già definita  
}
```

```
for (int i=0; i > N; i++) {  
    ...  
}  
for (int i=0; i > N; i++) { // ok, blocco diverso  
    ...  
}
```



CLASSI E LIBRERIE



Ricordate?

- *Per ora*

- ▶ una **classe** è l'astrazione di un *concetto*, un *tipo di dato* che rappresenta uno **stato**, descritto da una serie di **attributi**, e un comportamento, descritto da una serie di **metodi**
 - e.g. la classe “docente”
- ▶ un **oggetto** è una specifica *istanza* di una classe, con un preciso stato e un preciso comportamento
 - e.g. lo specifico docente “Stefano Mariani”

- [e, parlando dei tipi composti]

- ▶ in generale, *qualunque classe Java*
 - *ogni classe Java* definisce infatti *un tipo* di dato
 - gli oggetti di una *stessa classe* sono dunque tutti dello *stesso tipo*



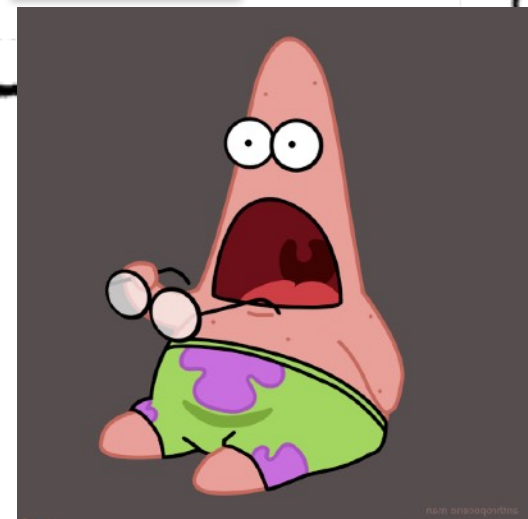
Raffiniamo la definizione

Nella programmazione orientata agli oggetti una **classe** è un costrutto di un linguaggio di programmazione usato come modello per creare oggetti. Il modello comprende attributi e metodi che saranno condivisi da tutti gli oggetti creati (istanze) a partire dalla classe. Un "oggetto" è, di fatto, l'istanza di una classe.

Una **classe** è identificabile come un *tipo di dato astratto* che può rappresentare una persona, un luogo, oppure una cosa, ed è quindi l'**astrazione di un concetto**, implementata in un software. Fondamentalmente, essa definisce al proprio interno lo stato, i cui dati sono memorizzati nelle cosiddette variabili membro o attributi, e il comportamento dell'entità di cui è rappresentazione, descritto da blocchi di codice riutilizzabili chiamati metodi.

riutilizzabili

- Riassumendo
 - ▶ **classe** = *modello* per definire oggetti
 - *stato* = insieme di **attributi**
 - *comportamento* = insieme di **metodi**
 - ▶ **oggetto** = particolare *istanza* di una certa classe

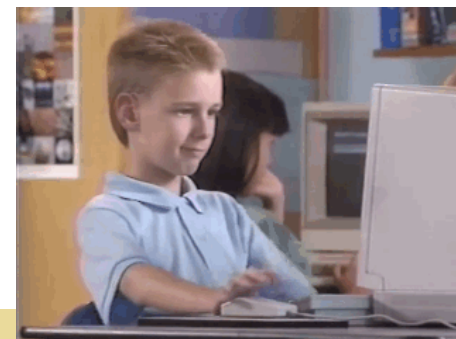




- Il concetto di **riusabilità** è *fondamentale* in informatica
 - ▶ evita di *ri-programmare* di continuo le stesse strutture dati e gli stessi comportamenti
 - il cosiddetto “*reinventing the wheel*”
 - ▶ è ciò che abbiamo sperimentato coi flow chart quando “copia-incollavamo” pezzi di flow chart
 - e.g. i 2-3 blocchi necessari per realizzare un ciclo
- In Java
 - ▶ i **metodi** sono il *più piccolo blocco* software riutilizzabile
 - ▶ poi le **classi**
 - ▶ poi interi **programmi**

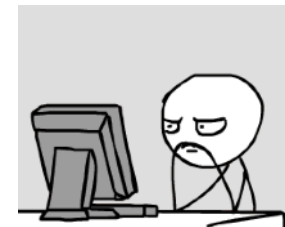


- In informatica, una **libreria** è una *collezione* di *strutture dati* e/o *funzionalità* correlate predisposte al *riutilizzo* da parte di altri programmi tramite opportuno **collegamento**
- In Java (per capirci meglio)
 - ▶ *strutture dati* = **classi**
 - ▶ *funzionalità* = **metodi**
 - ▶ *collegamento* = istruzione **import**
- Un qualunque linguaggio di programmazione offre tutta una serie di *librerie fondamentali*
 - ▶ strutture dati note (e.g. liste, mappe, alberi, grafi, etc.)
 - ▶ algoritmi noti (e.g. ordinamento, ricerca, manipolazione stringhe, etc.)
 - ▶ gestione I/O (e.g. lettura da shell/file, scrittura su shell/file, etc.)
 - ▶ ...
- Molte altre librerie sono fornite da altri programmatori
 - ▶ anche voi le potete costruire e fornire ad altri =)





- Le classi Java sono organizzate in **package**, *contenitori logici* che
 - raggruppano classi che offrono *funzionalità correlate*
 - definiscono un *namespace**
- Un package non è altro che una *lista di nomi separati da punto*
 - e.g. `java.lang`
- Attraverso tali package, la *libreria standard* (= inclusa nel JDK) di Java offre un grande numero di strutture dati e funzionalità
 - `java.io` per la gestione dell'I/O
 - `java.lang` per le classi *fondamentali* del linguaggio
 - `java.math` per strutture dati e funzioni *matematiche*
 - `java.util` per svariate funzioni di *utilità*
 - in particolare, le `Collections`, ovvero *strutture dati complesse*



**Argomento tecnico, solo per chi vuole approfondire*



Ok, ma quindi??

- Nei vostri programmi, se/quando volete usare classi di libreria (standard o di altri programmatori), dovete *prima importarle*
 - ▶ keyword Java **import**
- Ad esempio, nei nostri esempi leggeremo spesso l'input da *console* (ovvero dalla shell*)
 - ▶ per farlo, Java ci fornisce la classe Scanner nel package java.util della libreria standard
 - ▶ dunque `import java.util.Scanner;`



**È un'approssimazione, ma per ora non approfondiamo*



GESTIONE I/O



- Abbiamo parlato molte volte di *acquisire input* e *produrre output*
 - definizione di algoritmo
 - definizione di programma
 - nei flow charts
- La gestione dell'I/O (input/output) in Java è supportata da opportune librerie (e.g., java.lang, java.util, java.io)
- Ad esempio:
 - I/O tramite *console* (leggo da tastiera e scrivo a video)

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

```
System.out.println("X is " + x);
```

- I/O tramite *file* (leggo da e scrivo su file)
- ...

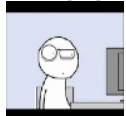
***new** è la keyword Java che *istanzia* un **oggetto** di una classe



- Uno **Scanner** spezzetta il suo input in *token* (“pezzi”) delimitati da un certo carattere (lo spazio, di default)
 - ▶ i token consumati possono essere *convertiti* in vari tipi di dato (se ammissibile) usando opportuni metodi (prossima slide)
- **Costruttori:**
 - ▶ `Scanner in = new Scanner(File source);`
 - costruisce un oggetto Scanner il cui input è alimentato dal file specificato
 - ▶ `new Scanner(InputStream source)`
 - costruisce un oggetto Scanner il cui input è alimentato dallo stream (flusso) specificato (e.g., **System.in**)
 - ▶ `new Scanner(String source)`
 - costruisce un oggetto Scanner il cui input è alimentato dall'oggetto stringa specificato



- boolean **hasNext()**



ritorna true se lo Scanner ha un token ancora da consumare

- String **next()**
 - ritorna in formato stringa il prossimo token disponibile
- Boolean **nextBoolean()**
 - ritorna in formato Boolean il prossimo token disponibile
- double **nextDouble()**
 - ritorna in formato double il prossimo token disponibile
- int **nextInt()**
 - ritorna in formato int il prossimo token disponibile
- String **nextLine()**
 - ritorna in formato stringa tutti i token trovati fino a un *carattere di fine linea*
 - solitamente, il carattere speciale `\n`



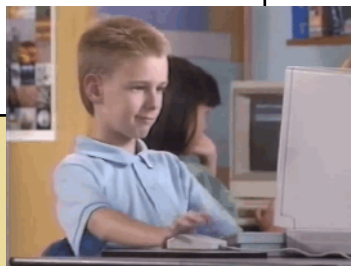
L'istruzione return

- **return** è un'altra *keyword* Java che useremo spesso (come `new`, già vista)
 - ▶ permette a un **metodo** di *restituire* un valore (primitivo o oggetto) all'istruzione *chiamante*
 - ▶ provoca l'*uscita immediata* dal metodo
- l'**invocazione** di (chiamata a) metodo non è altro che una delle tante possibili istruzioni Java (e non solo Java)
 - ▶ consente di continuare l'esecuzione del codice dal *corpo* del metodo invocato
 - ▶ è uno dei fondamenti della *programmazione strutturata*

Parleremo approfonditamente di metodi, oggetti e classi più avanti, non temete =)

- abilita il *riuso*
- abilita l'approccio "*dividi et impera*"
- ...

```
int sum(int op1, op2) {  
    return op1 + op2;  
}
```



```
public class MyClass {  
    int acc = 0;  
    int accumul(int val) {  
        acc = this.sum(acc, val);  
        return acc;  
    }  
}
```

- **this** indica la classe corrente
- Il "." è l'operatore di invocazione di metodo



- La classe **System** fornisce accesso ad alcuni servizi e proprietà *specifiche* del sistema su cui la JVM esegue
 - ▶ e.g. gli stream *standard* di I/O
 - ▶ il tempo di sistema
 - ▶ il carattere di fine linea di default (ricordate?)
 - ▶ ...
- Per ora, a noi interessano due dei suoi *attributi*
 - ▶ InputStream **in** (accessibile con l'istruzione System.in)*
 - ▶ PrintStream **out** (accessibile come System.out)

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



*Per ora prendetelo come un atto di fede, vedremo poi il perché =)



- **System.out** è un oggetto di tipo `PrintStream`
 - ▶ consente di stampare a video le *rappresentazioni* di alcuni tipi di dato in maniera automatica
- Sostanzialmente, tre tipi di metodo (ognuno disponibile per ogni tipo di dato primitivo + `String`)
 - ▶ **print(...)** — e.g. `print("Ciao")` per stampare una stringa
 - ▶ **println(...)** — e.g. `println(4.5)` per stampare un `double` e *andare a capo*
 - ▶ **printf(...)** — e.g. `printf("res is %d", val)` per stampare una serie di dati *formattati*
 - <https://docs.oracle.com/javase/9/docs/api/java/util/Formatter.html#syntax> (per chi è interessato)

```
String msg= "risultato";  
int res = 13;  
System.out.println("Il tuo " + msg + "è " + res);
```





VARIE ED EVENTUALI



- *Indipendentemente* dal linguaggio di programmazione, *formattare* opportunamente il codice sorgente è buona prassi
 - indentazione consistente
 - un'istruzione per linea
 - uso consistente delle parentesi
 - ...
- Ne va della *comprensibilità* del codice, che è fondamentale per la sua **manutenibilità**
 - codice corretto “scritto male” è difficile da capire

```
int accumul(int val)
Xacc = sum(acc, val); retXrn acc;}
```

```
int accumul(int val){
    acc = sum(acc, val);
    return acc;
}V
```



- Formattazione, nomi di classi, metodi e variabili, e tanti altri aspetti *stilistici* della programmazione sono oggetto di **convenzioni** tra programmatori
 - il nome di una classe inizia con la lettera maiuscola
 - il nome di un metodo o variabile inizia con la minuscola
 - per tutti i nomi si usa la notazione “camelCase”

```
int Sum_and_acc(int val) {  
    Acc = Sum(acc, val);  
    return acc;  
}
```

```
int sumAndAcc(int val){  
    acc = sum(acc, val);  
    return acc;  
}
```

- La stessa Oracle (organizzazione dietro Java) e colossi come Google hanno le loro convenzioni
 - <http://web.archive.org/web/20140222045352/http://www.oracle.com/technetwork/java/codeconv-138413.html>
 - <https://google.github.io/styleguide/javaguide.html>



- Abbiamo visto che Java offre
 - *istruzioni* del linguaggio
 - classi e metodi di *libreria*
- Ma come conoscerli tutti??
 - le istruzioni si imparano studiando il linguaggio
 - classi e metodi “pronti all’uso” si cercano =)
- Dove?
 - Nella **Java API**
 - <https://docs.oracle.com/javase/9/docs/api/overview-summary.html>
- Elenca tutte le classi e i metodi disponibili nella *standard Java library*
 - dunque in ogni JDK =)





Fondamenti di Informatica

(L-Z)

Corso di Laurea in Ingegneria Gestionale

Java: Fondamenti

Prof. Stefano Mariani
Dott. Alket Cecaj