

# tree school

## Modulo 1

Introduzione a Java e all'ambiente di sviluppo

*Melvin Massotti*



# Menù del giorno

## Tipi di dato, Variabili e Costanti

- Introduzione alle Variabili ed alle Costanti
- Dichiarazione di Variabili e Costanti
- I Tipi di Dato in Java
- I tipi primitivi
- I tipi riferimento
- Il tipo String Boxing, unboxing e autoboxing

## Operatori

- Operatori Aritmetici
- Operatori Logici
- Operatori di Incremento
- Operatori Relazionali
- Operatori di Assegnazione
- Operatori Bitwise e BitShift
- Priorità degli Operatori

Orario: 9-13 & 14:00-16:00



# Menù del giorno

## Sintassi, Naming e Convenzioni

- Espressioni, Statements e Blocchi
- Commenti
- Dichiarazioni ed Assegnazioni
- Naming Conventions
- Keywords Riservate
- Organizzazione del Codice
- Spazi ed indentazione
- Commenti Buone Pratiche

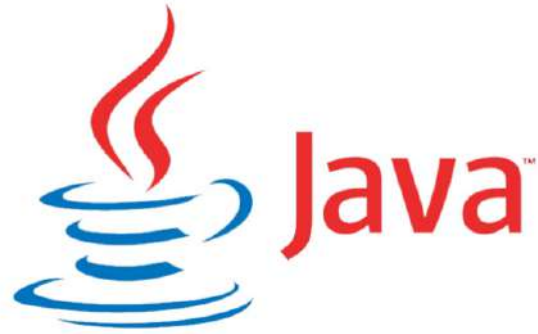
Orario: 9-13 & 14-16



# La vostra Bibbia

Java Official Documentation by Oracle

<https://docs.oracle.com/javase/tutorial/>



# Tipi di dato, Variabili e Costanti

## Cos'è una variabile?

- Una **variabile**, nell'informatica, è un contenitore di dati situato in una (o più) locazioni di memoria, destinata a contenere valori che possono essere modificati nel corso dell'esecuzione di un programma
- Una variabile è caratterizzata da un nome
- Nei linguaggi tipizzati, una variabile è caratterizzata anche da un **tipo** di dato che la descrive (restringendo i valori accettabili)
- Il formato standard per la dichiarazione di una variabile si divide in 3 parti:

```
NAME of the variable | an ASSIGNMENT OPERATOR | the VALUE of the variable
```

# Variabili in Java

- Java è un linguaggio staticamente **tipizzato**: significa che ogni variabile deve essere dichiarata prima di essere utilizzata

```
int gear = 1;
```

- In questo modo, stiamo dicendo al nostro programma di allocare una variabile di nome «gear», di tipo numerico e con valore iniziale 1.

## Costanti in Java

- La sintassi utilizzata per la definizione delle **costanti** è analoga a quella che abbiamo utilizzato per definire le variabili, a parte l'aggiunta della parola chiave `final` che precede la dichiarazione:

```
final int costante = 31;
```

- La keyword *final* indica al compilatore che il valore associato alla variabile di tipo intero chiamata `costante` non potrà più essere variato durante l'esecuzione del programma. Anche in questo caso è possibile eseguire le operazioni di dichiarazione ed inizializzazione in due passi, ma sempre con il vincolo che, una volta eseguita l'inizializzazione della costante, il valore di quest'ultima non venga più variato.



## Primitivi e oggetti

- Nei prossimi moduli vedremo nel dettaglio cos'è un **oggetto** e cosa significa **OOP** (Object-oriented programming)
- Per ora, ci limitiamo a capire le differenze nell'utilizzo dei tipi primitivi rispetto agli oggetti
  - Gli oggetti Java mantengono uno stato composto dalle sue variabili, anche dette attributi
  - Un tipo primitivo non è composto da altri tipi di dato, al contrario di un oggetto
  - I primitivi seguono la logica del **pass-by-value**, mentre gli oggetti la **pass-by-reference**
  - I tipi primitivi non hanno una gerarchia, sono indipendenti, mentre ogni oggetto è discendente della classe Object
  - Ogni oggetto ha dei metodi di default, che eredita appunto dalla classe Object, mentre i primitivi non hanno alcun metodo associato

## Piccola digressione: pass-by-value vs pass-by-reference

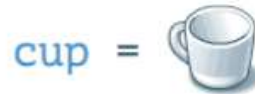
- **Pass-by-value:** viene passata una copia del valore. Quindi chiamante e chiamato hanno due variabili indipendenti
- **Pass-by-reference:** viene passata la reference diretta all'oggetto (ossia una copia diretta dell'indirizzo dell'oggetto in memoria)

*pass by reference*



*fillCup(        )*

*pass by value*



*fillCup(        )*

## Tipi primitivi di dato

- Java ha 8 tipi di dato detti **primitivi**, ossia predefiniti dal linguaggio e nominati con parole riservate (non utilizzabili dal programmatore):
- **byte**: Il tipo byte contiene 8 bit (uno è utilizzato per il segno), ha un valore minimo di -128 ed un valore massimo di 127 (compresi). E' utile per risparmiare memoria in caso di manipolazione di grandi quantità di dati, dove il risparmio di memoria può diventare cruciale, ma anche quando ad esempio dobbiamo leggere un file.
- **short**: Il tipo short contiene 16 bit (uno è utilizzato per il segno), ha un valore minimo di -32,768 ed un valore massimo di 32,767 (compresi). Come il tipo byte, è utile per risparmiare memoria in caso di manipolazione di grandi quantità di dati.

## Tipi primitivi di dato

- **int**: Il tipo int contiene 32 bit (uno è utilizzato per il segno), ha un valore minimo di  $-2^{31}$  ed un valore massimo di  $2^{31}-1$  (compresi). Da Java SE 8 si può anche utilizzare come unsigned tramite la classe Integer per ottenere un range di valori da 0 a  $2^{32}-1$  (compresi).
- **long**: Il tipo long contiene 64 bit (uno è utilizzato per il segno), ha un valore minimo di  $-2^{63}$  ed un valore massimo di  $2^{63}-1$  (compresi). Come per il tipo int, Da Java SE 8 si può anche utilizzare come unsigned tramite la classe Long per ottenere un range di valori da 0 a  $2^{64}-1$  (compresi). Utilizzate questo tipo di dato quando avete bisogno di un range di dati superiore a quello fornito dal tipo int.



## Tipi primitivi di dato

- **float:** Il tipo float contiene 32 bit (uno è utilizzato per il segno) ed è utilizzato per rappresentare decimali.
- **double:** Il tipo double contiene 64 bit (uno è utilizzato per il segno) ed è utilizzato come il tipo float per rappresentare decimali. E' la scelta standard solitamente per questo tipo di dato.
- **boolean:** Il tipo boolean ha solo due possibili valori: true e false. Si utilizza per tracciare semplici «flags» che stabiliscono condizioni per il programma.
- **char:** Il tipo char contiene un carattere Unicode da 16 bit, ha un valore minimo di '\u0000' (0) e un valore massimo di '\uffff' (65,535).

## Tipi primitivi di dato

- In aggiunta ai tipi primitivi di dato che abbiamo appena analizzato, il linguaggio Java fornisce anche un supporto speciale per le sequenze di caratteri tramite la classe `java.lang.String`: se racchiudiamo una sequenza di char con le virgolette, automaticamente Java creerà un oggetto String
- Esempio:

**`String s = "this is a string";`**

- La classe String non è un tipo primitivo, ma dato il supporto di primo livello fornito dal linguaggio, potete pensare come se lo fosse effettivamente.

## Valori di default

- Non sempre è necessario assegnare un valore quando una variabile viene dichiarata.
- Alle variabili dichiarate ma non inizializzate vengono assegnati valori di default in base al tipo:

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

- In generale, non è una buona pratica di programmazione:

## Stranezze...

- `Int a = 3;`
- `Int b = 2;`
- `Int c = a/b;`
- Quanto vale c? E perché?



## Stranezze...

- `int a = 3;`
- `int b = 2;`
- `double c = a/b;`
- Quanto vale c? E perché?

## Stranezze...

- `int a = 3;`
- `int b = 2;`
- `double c = a/b;`
- Quanto vale c? E perché?
- Convertire l'output non risolve il problema, perché ormai è troppo tardi: Java ha già effettuato una divisione tra interi!

## Cos'è un metodo?

- Nelle lezioni successive, approfondiremo il concetto di **metodo** di una classe, per ora vi basta sapere che, dato un insieme di **input** (anche vuoto), può restituire un **output** oppure fare soltanto delle operazioni, senza restituire nulla.
- Nel secondo caso, si dice che un metodo è **void**
- Per adesso, pensate ai metodi come a piccole «scatole nere» che prendono in input qualcosa e, in alcuni casi, ne restituiscono un'altra.



## Metodi della classe String

Vediamo ora alcuni metodi utilizzabili con gli oggetti di tipo **String** molto utili per lo sviluppo

## Metodo length

Il metodo **length** restituisce il numero dei caratteri all'interno della stringa.

```
String s1 = "Ciao a tutti!";  
s1.length();
```

## Metodo substring

Dati due numeri interi in ingresso (inizio e fine) dove inizio è minore di fine, il metodo **substring** "taglia" la stringa e ci restituisce solo la parte della stringa compresa tra i due intervalli

In questo caso, quale stringa otterremo nell'oggetto «s2»?

```
String s1 = "wikitolearn";  
String s2;  
s2 = s1.substring(0,4);
```

## Metodo equals

Il metodo **equals** ci restituisce un valore booleano (o false o true), ci restituisce true, se i caratteri delle due stringhe sono identiche, false altrimenti

In questo caso, quale valore otterremo nell'oggetto «confronta»?

```
String s1 = "wikitolearn";  
String s2 = "wikitolearn";  
boolean confronta;  
confronta = s1.equals(s2);
```

## Metodo charAt

Dato un numero intero (indice), il metodo **charAt** ci restituisce il carattere (un char) della posizione voluta

```
String s1 = "wikitolearn";  
Char car1;  
car1 = s1.charAt(0);
```



## Metodo startsWith

- Data una Stringa prefisso in input, il metodo **startsWith** restituisce true se la Stringa attuale inizia con quel prefisso

```
boolean startsWith(String prefix)
```

- Qual è l'output del seguente programma?

```
String prefisso1 = "cia";  
String prefisso2 = "Cia";  
String stringa = "ciao come va";  
System.out.println(stringa.startsWith(prefisso1));  
System.out.println(stringa.startsWith(prefisso2));
```

## Metodi toLowerCase e toUpperCase

- **toLowerCase**: produce una nuova stringa convertendo la stringa attuale in minuscolo

```
String toLowerCase()
```

- **toUpperCase**: produce una nuova stringa convertendo la stringa attuale in maiuscolo

```
String toUpperCase()
```

## Metodo indexOf

Data una stringa `s` in input, il metodo **indexOf** ci restituisce l'indice della prima occorrenza di `s` nella stringa attuale

```
indexOf( @NotNull String str)
```

Esempio:

Qual è l'output di questo codice?

```
String s = "cia";  
String stringa = "ehi |ciao come va";  
System.out.println(stringa.indexOf(s));
```

## Metodi replace e replaceAll

- **replace**: date due CharSequence old e new in input, sostituisce ogni occorrenza di old con new nella stringa attuale

```
String replace(CharSequence target, CharSequence replacement)
```

- **replaceAll**: date due Stringhe old e new in input, sostituisce ogni occorrenza di old con new nella stringa attuale

```
String replaceAll(String regex, String replacement)
```

## Metodi replace e replaceAll: esempio

In questo caso, quale stringa otterremo nell'oggetto a?

```
String target = "ciao come va comecome";  
String a = target.replaceAll(regex: "come", replacement: "how");
```

## Metodo split

Data una stringa *s* in input, il metodo **split** ci restituisce un array (vedremo in seguito) di stringhe contenente tutte le parti in cui la stringa attuale viene suddivisa «tagliando» esattamente ad ogni occorrenza di *s*

```
String[] split( @NotNull String regex)
```

Esempio:

Qual è l'output di questo codice?

```
String a = "ciao come";  
String[] split = a.split( regex: " ");
```

# Classi Wrapper

- In Java, per ogni tipo primitivo esiste una corrispondente classe **wrapper**.
- Anche se ad un primo sguardo può sembrare che ci sia poca differenza tra un tipo primitivo e la sua controparte 'wrapped' (spesso detta '**boxed**') tra le due c'è una fondamentale distinzione: i tipi primitivi non sono oggetti, mentre i wrapper lo sono a tutti gli effetti.
- I tipi primitivi non hanno associata alcuna classe e quindi devono essere trattati in modo diverso rispetto agli altri tipi, ad esempio non è possibile utilizzarli nelle collezioni, che saranno argomento di future lezioni) e non possono avere metodi.

# Classi Wrapper

Nella seguente tabella troviamo, per ogni tipo primitivo, la corrispondente classe wrapper:

Tipo primitivo	Classe Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean



## Boxing e unboxing

Tipi primitivi e classi wrapper portano al problema dell'interoperabilità tra i due sistemi, ossia l'assegnamento di un valore primitivo ad un oggetto wrapper e viceversa. In particolare, diamo le definizioni di boxing e unboxing:

- **boxing**: conversione da un valore primitivo ad un oggetto

Esempio:

```
Character ch = 'a';
```

- **unboxing**: conversione da un oggetto ad un valore primitivo

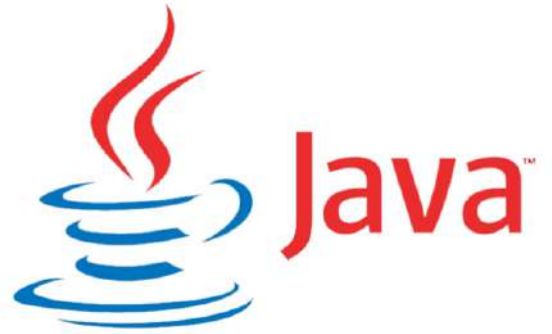
Esempio:

```
Integer n = new Integer(7);  
int n = i;
```

# Autoboxing

L'autoboxing è una caratteristica del linguaggio che, a partire dalla versione 1.5, ci consente di lavorare con tipi primitivi e tipi wrapper in maniera intercambiabile, senza preoccuparci di effettuare boxing o unboxing. Ad esempio, sono possibili i seguenti assegnamenti:

```
Integer x = 10;  
Double y = 5.5f;  
Boolean z = true;  
Number n = 0.0f;
```



# Operatori



# Operatori

- Ora che sappiamo come dichiarare ed inizializzare variabili, è ora di imparare a farci qualcosa!
- Gli **operatori** sono simboli speciali che elaborano specifiche operazioni su uno, due o tre operandi e ritornano un risultato
- Come nella matematica, gli operatori hanno diversa priorità (precedenza): operatori con maggiore precedenza vengono valutati prima di operatori con precedenza minori
- Tutti gli operatori binari, eccetto gli operatori di assegnamento, sono valutati da sinistra a destra; gli operatori di assegnamento, invece sono valutati da destra a sinistra.

## Priorità degli operatori

Gli operatori nella tabella a destra sono ordinati per precedenza: più un operatore è in alto, maggiore è la sua precedenza.

Operator Precedence	
Operators	Precedence
postfix	<i>expr++ expr--</i>
unary	<i>++expr --expr +expr -expr ~ !</i>
multiplicative	<i>* / %</i>
additive	<i>+ -</i>
shift	<i>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</i>
relational	<i>&lt; &gt; &lt;= &gt;= instanceof</i>
equality	<i>== !=</i>
bitwise AND	<i>&amp;</i>
bitwise exclusive OR	<i>^</i>
bitwise inclusive OR	<i> </i>
logical AND	<i>&amp;&amp;</i>
logical OR	<i>  </i>
ternary	<i>? :</i>
assignment	<i>= += -= *= /= %= &amp;= ^=  = &lt;&lt;= &gt;&gt;= &gt;&gt;&gt;=</i>

## Operatori aritmetici

- Operatori per effettuare addizioni, sottrazioni, moltiplicazioni e divisioni

+	Additive operator (also used for String concatenation)
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

# Operatori logici

- Gli operatori logici (anche detti «booleani») sono operatori che ritornano un valore booleano in base al risultato di una o più espressioni

Operator	Description	Example
<code>  </code>	conditional-OR: <code>true</code> if either of the boolean expression is <code>true</code>	<code>false    true</code> is evaluated to <code>true</code>
<code>&amp;&amp;</code>	conditional-AND: true if all boolean expressions are <code>true</code>	<code>false &amp;&amp; true</code> is evaluated to <code>false</code>

## Operatori di incremento

- Gli operatori ++ e -- possono essere usati sia come prefisso che come suffisso
- L'operatore ++ aumenta il valore di 1 mentre l'operatore -- decrementa il valore di 1
- Qual è l'output del seguente programma?

```
public static void main(String[] args) {  
  
    double number = 5.2;  
  
    System.out.println(number++);  
    System.out.println(number);  
  
    System.out.println(++number);  
    System.out.println(number);  
}
```



## Operatori di incremento

- Gli operatori ++ e -- possono essere usati sia come prefisso che come suffisso
- L'operatore ++ aumenta il valore di 1 mentre l'operatore -- decrementa il valore di 1
- Qual è l'output del seguente programma?

```
public static void main(String[] args) {  
  
    double number = 5.2;  
  
    System.out.println(number++);  
    System.out.println(number);  
  
    System.out.println(++number);  
    System.out.println(number);  
}
```

5.2

6.2

7.2

7.2

## Operatori relazionali

- Come gli operatori logici, anche gli operatori relazionali restituiscono un valore booleano (true o false), che in questo caso è il risultato di un confronto secondo le seguenti regole:

Operatore	Descrizione	Esempio	Risultato?
==	Uguale a	5 == 3	
!=	Diverso da	5 != 3	
>	Maggiore di	5 > 3	
<	Minore di	5 < 3	
>=	Maggiore o uguale a	5 >= 5	
<=	Minore o uguale a	5 <= 5	

## Operatori relazionali

- Come gli operatori logici, anche gli operatori relazionali restituiscono un valore booleano (true o false), che in questo caso è il risultato di un confronto secondo le seguenti regole:

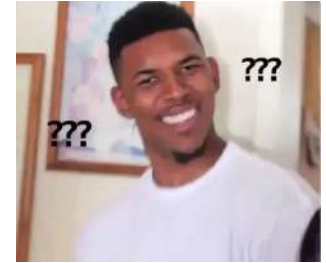
Operatore	Descrizione	Esempio	Risultato?
==	Uguale a	5 == 3	false
!=	Diverso da	5 != 3	true
>	Maggiore di	5 > 3	true
<	Minore di	5 < 3	false
>=	Maggiore o uguale a	5 >= 5	true
<=	Minore o uguale a	5 <= 5	true

## Operatori Bitwise e Shift


- Java fornisce anche degli operatori, poco utilizzati eccetto in specifiche occasioni (solitamente quando le bisogna garantire alte performance computazionali) per effettuare operazioni bitwise e bit shift

Operator	Description
<code>~</code>	Bitwise Complement
<code>&lt;&lt;</code>	Left Shift
<code>&gt;&gt;</code>	Right Shift
<code>&gt;&gt;&gt;</code>	Unsigned Right Shift
<code>&amp;</code>	Bitwise AND
<code>^</code>	Bitwise exclusive OR
<code> </code>	Bitwise inclusive OR


# Operatori Bitwise e Shift



Esempio: 60 & 13 fa 12, ma 60 | 13 fa 61!

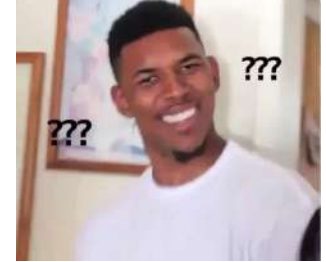
						
a	0	0	1	1	1	1 0 0 (60)
b	0	0	0	0	1	1 0 1 (13)
<hr/>						
a&b	0	0	0	0	1	1 0 0 (12)

a	b	a&b
0	0	0
1	0	0
0	1	0
1	1	1

						
a	0	0	1	1	1	1 0 0 (60)
b	0	0	0	0	1	1 0 1 (13)
<hr/>						
a b	0	0	1	1	1	1 0 1 (61)

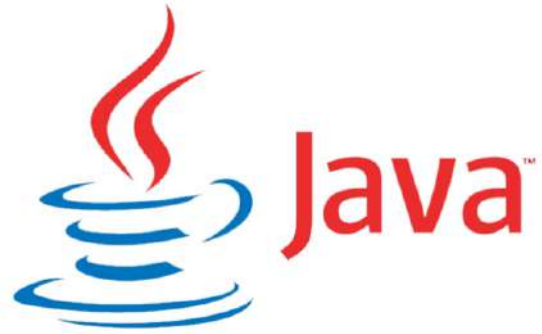
a	b	a b
0	0	0
1	0	1
0	1	1
1	1	1

# Operatori Bitwise e Shift



Esempio di left shift

<u>SYNTAX</u>	<u>BINARY FORM</u>	<u>VALUE</u>
<code>x = 7;</code>	00000111	7
<code>x=x&lt;&lt;1;</code>	00001110	14
<code>x=x&lt;&lt;3;</code>	01110000	112
<code>x=x&lt;&lt;2;</code>	11000000	192



# Sintassi, Naming e Convenzioni

# Sintassi Java: Dichiarazioni ed assegnazioni

- Un paio di definizioni «formali»:
  - **Dichiarazione** di una variabile:
    - Quando viene definita una nuova variabile con il suo tipo annesso
  - **Assegnazione** di una variabile:
    - Quando viene cambiato il valore di una variabile assegnandole un nuovo valore
  - **Inizializzazione** di una variabile:
    - Quando viene effettuato un assegnamento insieme alla dichiarazione di una variabile

```
int i;
```

```
i = 42
```

```
int i = 42
```



# Sintassi Java: Commenti

- In java esistono 3 tipi di commento:
  - Singola linea, effettuato ponendo i caratteri «//» all'inizio della linea

```
//This is single line comment
```

- Linea multipla, utilizzando i caratteri «/\*» come inizio e «\*/» come fine dell'area di commento

```
/*  
This  
is  
multi line  
comment  
*/
```

## Sintassi Java: Commenti

- Commenti di documentazione, utilizzati per creare la documentazione del nostro programma tramite il tool Javadoc (che vedremo in dettaglio nelle prossime lezioni). Si realizza utilizzando i caratteri «/\*\*» come inizio e «\*/» come fine dell'area di commento

```
/**  
This  
is  
documentation  
comment  
*/
```

# Sintassi Java: Commenti e best-practices

Come in ogni cosa, ci sono due «scuole» di pensiero:

Scrivere codice senza commentarlo  
è come costruire un mobile Ikea  
senza fornire le istruzioni

VS

Un codice ben scritto e ben  
strutturato non ha bisogno di  
commenti per essere compreso

## Sintassi Java: Commenti e best-practices



E, come in ogni cosa, la verità sta nel mezzo

# Sintassi Java: Espressioni

- Gli operatori che abbiamo visto possono essere utilizzati per comporre espressioni e calcolare valori.
- Le **espressioni** compongono le **dichiarazioni (statements)**, i quali compongono i **blocchi (blocks)**
- Un'espressione è un costrutto composto da variabili, operatori e invocazioni a metodi, che vengono valutati ad un singolo valore
- Il tipo del valore ritornato da un'espressione dipende dagli elementi utilizzati nell'espressione

## Sintassi Java: Espressioni e ordine

- Fate attenzione l'ordine degli operatori utilizzati nelle espressioni, perché può cambiare sostanzialmente il risultato dell'espressione
- Esempio:

$x + y / 100$

$(x + y) / 100$

Siate sempre espliciti nella dichiarazione delle espressioni, facendo uso di parentesi dove necessario per rendere il codice facilmente comprensibile

## Sintassi Java: Statements

- Gli statement sono come le frasi nel linguaggio naturale
- Uno statement forma una completa unità di esecuzione
- I seguenti tipi di espressioni possono comporre uno statement:
  - Espressioni di assegnamento variabili
  - Ogni utilizzo degli operatori di incremento e decremento
  - Invocazioni a metodi
  - Espressioni di creazione di oggetti (vedremo in seguito cos'è un oggetto)
- Importante: ogni statement deve terminare con il ;

# Sintassi Java: Statements

Alcuni esempi di statement

- // statement di assegnamento
- `aValue = 8933.234;`
- // statement di incremento
- `aValue++;`
- // invocazione di un metodo
- `System.out.println("Hello World!");`
- // statement di creazione oggetto
- `Bicycle myBike = new Bicycle();`
- // statement di dichiarazione
- `double aValue = 8933.234;`



## Sintassi Java: Blocks

- Un blocco è un gruppo di zero o più statement compreso tra parentesi graffe
- Può essere utilizzato ovunque un singolo statement sia permesso
- Esempio:

```
public static void main(String[] args) {  
    boolean condition = true;  
    if (condition) { // begin block 1  
        System.out.println("Condition is true.");  
    } // end block one  
    else { // begin block 2  
        System.out.println("Condition is false.");  
    } // end block 2  
}
```

## Sintassi Java: Naming conventions

- E' molto importante rispettare le convenzioni sul codice (non solo in Java).
- Le più importanti sono le seguenti:
  - **Nomi di pacchetto:** sempre con lettere minuscole e preferibilmente costituiti da un'unica parole (es. package veicoli.auto)
  - **Nomi delle classi:** sempre al plurale e con la prima lettera maiuscola, le altre a seguire minuscole (es. Saluti). Se si hanno più parole si seguono le stesse regole (TestSaluti). Seguono le stesse regole le interfacce (le vedremo in seguito)
  - **Nomi dei metodi:** sempre con lettere minuscole (es. inizializza). Se il nome del metodo è costituito da più parole, solo l'iniziale delle altre parole sarà maiuscola (es. inizializzaNome). Questa pratica in particolare è detta **CamelCase**
  - **Nomi di variabili:** seguono le stesse regole dei metodi (es. nomeAnimale).

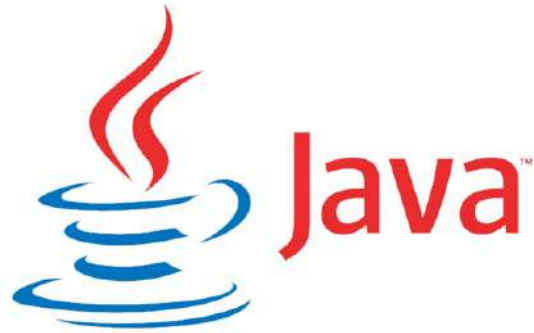
## Sintassi Java: Naming conventions

- **Costanti:** sempre con lettere maiuscole (COSTANTE), se il nome è costituito da più parole vengono separate con l'underscore (es. PI\_GRECO).
- **Spaziatura:** per rendere il codice più leggibile si utilizzano sempre due spazi per l'indentazione.
- **Parentesi:**

```
if ( condition ) {  
    // Fai qualcosa  
} else {  
    // Fai qualcosa  
}
```

# Sintassi Java: Keywords riservate

- Alcune parole (keyword) in Java sono riservate, ossia non possono essere utilizzate da sole come nomi di variabili od oggetti
- Alcuni esempi di keyword:
  - abstract
  - new
  - int
  - if
  - else
  - long
  - double
  - ...

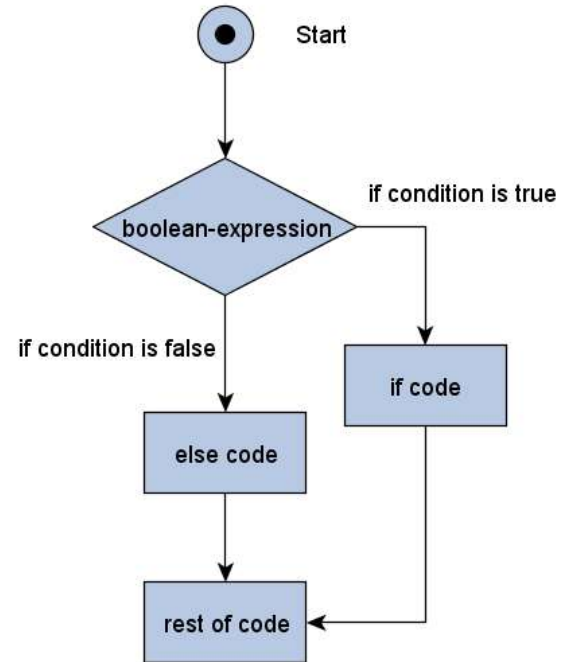


# Condizioni e cicli

Breve introduzione

# Introduzione ai costrutti condizionali

- Il costrutto condizionale permette di eseguire una certa sezione del codice solo se (**if**) una condizione booleana è soddisfatta
- E' possibile specificare anche un «percorso alternativo» (**else**) che viene seguito se la condizione di partenza è falsa



if-else statement flow chart

## If-then-else in Java

- In Java si utilizza il costrutto **if-then-else** (anche se la parola «then» non esiste)

```
if(condizione1) {  
    // ...  
} else if (condizione2) {  
    // ...  
} else {  
    //...  
}
```

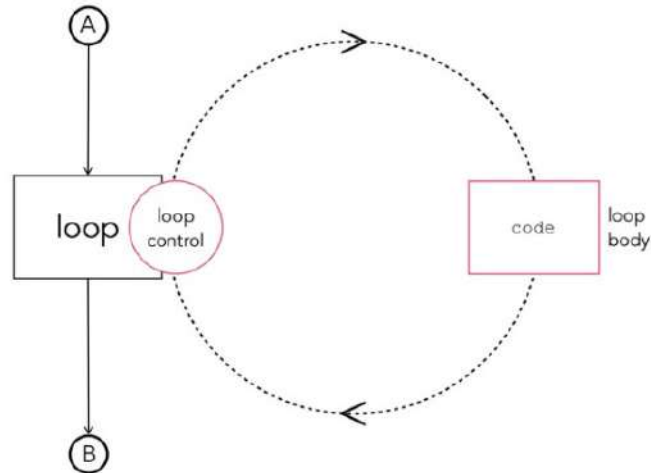
## If-then-else in Java

- `If(condizione1) {`
  - codice da eseguire se condizione1 è soddisfatta
  - `}`
  
  - `Else if(condizione2) {`
  - codice da eseguire se condizione2 è soddisfatta
  - `}`
  - ...
  - `Else {`
  - codice da eseguire se tutte le precedenti condizioni non sono soddisfatte
  - `}`
- SE condizione1
  - codice da eseguire se condizione1 è soddisfatta
  
  - SE INVECE condizione2
  - codice da eseguire se condizione2 è soddisfatta
  - ...
  
  - ALTRIMENTI
  - codice da eseguire se tutte le precedenti condizioni non sono soddisfatte



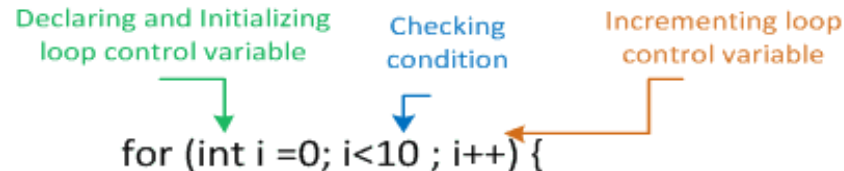
# Introduzione ai cicli

- Il **ciclo** è uno statement di controllo del flusso che itera una parte del programma più volte



## Ciclo classico in Java

- Il **for** in Java si utilizza nel caso di un numero finito di iterazioni e deve rispettare la seguente sintassi:



```
for (int i =0; i<10 ; i++) {  
  
    // Loop statements to be executed  
  
}
```

## For-each in Java

A partire da Java 1.5, è possibile utilizzare il costrutto **for-each** per iterare direttamente sugli elementi di un array (o anche collections, come vedremo) invece di accedervi attraverso gli indici

```
for (type var : array)
{
    statements using var;
}
```



```
for (int i=0; i<arr.length; i++)
{
    type var = arr[i];
    statements using var;
}
```



## While in Java

Il ciclo **while** è simile al **for**, ma risulta più comodo da utilizzare nel caso in cui si debba eseguire del codice finché una certa condizione è valida

```
while (condition) {  
    // code block to be executed  
}
```

## Cicli in Java: for vs while

- **For:** quando abbiamo a che fare con array o collections
- **While:** quando l'iterazione del ciclo dipende esclusivamente da una condizione booleana



# Interrompere un ciclo

- **break**: interrompe il ciclo, passando il controllo dell'esecuzione allo statement successivo al for

```
for (i = 0; i < arrayOfInts.length; i++) {  
    if (arrayOfInts[i] == searchfor) {  
        foundIt = true;  
        break;  
    }  
}
```

## Interrompere un ciclo

- **continue**: salta alla prossima esecuzione del ciclo

```
for (int i = 0; i < max; i++) {  
    // interested only in p's  
    if (searchMe.charAt(i) != 'p')  
        continue;  
  
    // process p's  
    numPs++;  
}
```

# Grazie per l'attenzione!

Domande?

