

## Preparazione Prova Orale

### Domande e risposte

#### 1) Fornire le definizioni di PC, Hardware, Programma e Software

**Computer:** supporto programmabile per la rappresentazione ed elaborazione dell'informazione. Viene generalmente considerato come l'insieme di componenti hardware e software.

**Hardware:** rappresenta l'insieme dei componenti fisici della macchina.

**Programma:** insieme di istruzioni che un computer esegue.

**Software:** indica generalmente tutti i possibili tipi di programmi utilizzati per fornire istruzioni a un computer.



#### 2) Cosa si intende per CPU?

Componente hardware interno che esegue le istruzioni di un programma. Il **processore (CPU)** è in grado di eseguire di istruzioni molto semplici, come effettuare operazioni aritmetiche di base (somme e/o sottrazioni) o spostare numeri o altri dati presenti in memoria.



#### 3) Indicare la differenza tra memoria principale e memoria ausiliaria

La **memoria principale** conserva il programma che attualmente è in esecuzione e gran parte dei dati che il programma sta utilizzando. Le informazioni immagazzinate sono volatili, cioè sono cancellate quando il computer viene spento (esempio **RAM**).

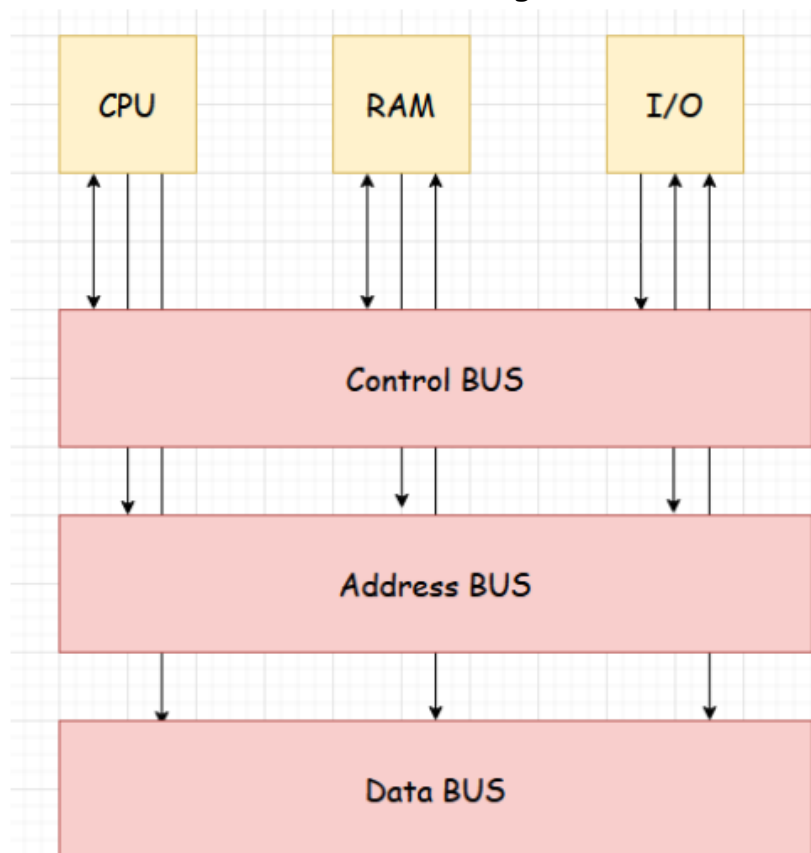


La **memoria ausiliaria** conserva i programmi anche a PC spento. I componenti della **memoria ausiliaria** sono **HDD, Flash Drive, CD, DVD**.

#### 4) Cosa rappresenta la macchina di Von Neumann

Fu la prima proposta di architettura hardware per un calcolatore. Questa architettura vede il calcolatore composto da più elementi:

- **CPU (Central Processing Unit)**: in grado di acquisire, interpretare ed eseguire programmi/istruzioni;
- **Memoria**, per il salvataggio delle informazioni;
- **Periferiche**, come la memoria di massa e le periferiche input/output (chiavette e stampanti), permettono lo scambio di informazioni con l'esterno;
- **Bus di sistema**, ovvero i collegamenti tra i vari elementi;



#### 5) Cosa si intende per bus di sistema?

Il **bus di sistema** è il canale di comunicazione, che permette la comunicazione tra la CPU e le periferiche I/O del sistema attraverso il trasferimento dei dati da una parte all'altra dell'elaboratore.

Ci sono tre tipologie di bus:

- **Bus dati**: è il bus sul quale transitano le informazioni ed è usufruibile da tutti i componenti del sistema, sia in lettura che in scrittura ed è bidirezionale, nel senso che permette il passaggio dei dati in più direzioni contemporaneamente.
- **Bus degli indirizzi**: è il bus unidirezionale attraverso il quale la CPU decide in quale indirizzo andare a scrivere o a leggere informazioni; sia le celle di

memoria (**RAM**) sia le **periferiche di I/O (Input/Output)** sono infatti divise in zone e porte, ognuna delle quali ha un dato indirizzo.

→ **Bus di controllo**: è un insieme di collegamenti il cui scopo è coordinare le attività del sistema; tramite esso, la **CPU** può decidere quale componente deve scrivere sul bus dati in un determinato momento, quale indirizzo leggere sul bus indirizzi, quali celle di memoria devono essere scritte e quali invece lette.

#### 6) Cosa si intende per calcolatore?

Il **calcolatore** è la macchina per l'elaborazione di dati in grado di comprendere la sola codifica binaria. Il calcolatore è un dispositivo attivo infatti è in grado di elaborare i dati prendendo decisioni in base a particolari input.

**Esempio**: una pagina web mostra all'utente solo i dati che gli competono in base alle sue autorizzazioni. A differenza delle altre macchine, il calcolatore può essere programmato e quindi specializzato in base al compito che dovrà eseguire. Per tale motivo è detto dispositivo di **General Purpose** (di scopo generale);

#### 7) Cosa si intende per linguaggio macchina? E per linguaggio assembly?

Per **linguaggio macchina** si intende il linguaggio con cui vengono scritti i programmi eseguibili da un computer. La grammatica dei **linguaggi macchina** dipende fortemente dalla tipologia di processore, il quale traduce le istruzioni presenti nel programma e le esegue. I **linguaggi macchina** sono anche denominati linguaggi di basso livello. Il **linguaggio assembly** è il linguaggio di programmazione le cui istruzioni sono composte da stringhe alfanumeriche corrispondenti in modo biunivoco alle istruzioni elementari dell'unità di elaborazione centrale (**CPU - Central Processing Unit**) di un calcolatore elettronico.

#### 8) Cosa si intende per informazione? Come può essere rappresentata?

L'**informazione** è tutto ciò che viene manipolato da un calcolatore e può essere rappresentata mediante suoni, numeri ed etc. La più piccola unità di informazione viene denominata **bit (Binary Digit, 0/1)** e tutte le informazioni sono rappresentate in **bit**. Un **bit** può assumere soltanto due valori (0 e 1). La rappresentazione dell'informazione è la modalità con la quale l'informazione viene descritta. Ad esempio, il valore "**16**" può essere rappresentato in diverse modalità:

→ **sistema romano**: XVI;

→ **sistema decimale**: 16;

- **sistema binario**: 10000;
- **sistema esadecimale**: 0x10.

### 9) Quale è la differenza tra sistema di numerazione posizionale e non posizionale?

Il sistema di numerazione posizionale è il sistema di numerazione in cui i simboli usati per scrivere i numeri assumono valori diversi a seconda della posizione che occupano nella notazione. Il sistema di numerazione non posizionale è il sistema di numerazione in cui ogni cifra assume sempre lo stesso valore.

Esempio:

**Posizionale**: sistema decimale.

**Non posizionale**: sistema romano.

Nei sistemi numerici posizionali, un valore numerico  $N$  è caratterizzato dalla seguente rappresentazione:

$$N = d_{n-1}d_{n-2} \dots d_1d_0, d_{-1} \dots d_{-m}$$

$$N = d_{n-1} \cdot r^{n-1} + \dots + d_0 \cdot r^0 + d_{-1} \cdot r^{-1} + \dots + d_{-m} \cdot r^{-m}$$

$$N = \sum_{i=-m}^{n-1} d_i \cdot r^i$$

### 10) Cosa si intende per algoritmo e quali sono le sue caratteristiche?

Un **algoritmo** è una sequenza ordinata di passi o istruzioni finiti necessari per la risoluzione del problema. Le caratteristiche dell'algoritmo sono:

- Ogni istruzione deve concretamente realizzabile dall'esecutore, a cui è affidato l'algoritmo.
- Le istruzioni che compongono l'algoritmo devono essere precise e non ambigue in modo che non lasciano dubbi nell'interpretazione da parte dell'esecutore.
- Ogni istruzione, se pur concretamente eseguibile e non ambigua, deve avere una durata limitata nel tempo.
- Ogni istruzione deve produrre un risultato osservabile.
- Ogni istruzione deve produrre sempre il medesimo effetto se eseguita a partire dalle stesse condizioni iniziali (carattere deterministico).
- Le istruzioni devono essere elementari, cioè non ulteriormente scomponibili rispetto alle capacità dell'esecutore.

Gli **algoritmi** devono eseguire delle determinate regole e deve essere:

→ **Finito**: Un algoritmo deve essere composto da un numero finito di istruzioni e deve rappresentare un punto di inizio, dove incomincia la parte risolutiva, e un punto di fine, dove finisce l'algoritmo.

→ **Esaustivo**: deve essere completo ed esaustivo, ovvero che per tutti i casi che si possono verificare durante l'esecuzione deve essere indicata la soluzione da seguire.

→ **Riproducibile**: Ogni successiva esecuzione dello stesso algoritmo con i medesimi dati iniziali deve produrre sempre i medesimi risultati finali.

### 11) Cosa si intende per programma? Per primitive? Per sintassi e semantica?

Un **programma** è una sequenza di istruzioni scritte in un linguaggio comprensibile al calcolatore.

**Attenzione:**

→ Il computer esegue algoritmi descritti tramite un **programma**. La differenza tra algoritmo e programma è che l'algoritmo è la sequenza ordinata di istruzioni finite necessarie per risolvere il problema che può essere descritta mediante lo pseudocodice, il linguaggio naturale o una formula matematica, mentre il programma è la rappresentazione di un algoritmo mediante il linguaggio di programmazione.

Le primitive identificano le componenti base, i mattoni del linguaggio di programmazione. La sintassi descrive come è strutturata la primitiva, mentre la semantica fornisce il significato della primitiva.

### 12) Quali sono le caratteristiche dei linguaggi di alto livello?

Le caratteristiche dei linguaggi di alto livello sono:

→ Primitive indipendenti dalla macchina;

→ primitive di alto livello (1 primitiva = più istruzioni macchina);

→ un programma può essere utilizzato su computer diversi traducendolo con l'apposito compilatore;

→ specificazione chiara e universale (indipendente da hardware e sistema operativo) del linguaggio;

→ estensioni del linguaggio: aggiunte allo standard per aumentarne le potenzialità o adattarlo ad una specifica macchina.

### 13) Cosa si intende per compilatore?

Il **compilatore** è un programma software che traduce le istruzioni di un linguaggio di programmazione ad alto livello in linguaggio macchina. Esso salva le

nuove istruzioni in memoria (crea file.exe). Il compilatore traduce prima il codice poi lo esegue; per tale motivo offre una traduzione più veloce ma l'eseguibile creato (esempio.exe) sarà utilizzabile solo sul calcolatore su cui il codice è stato compilato e quindi non sarà portatile.

#### 14) Cosa si intende per interprete?

Un **interprete** è un programma software che traduce un'istruzione alla volta e, senza memorizzare il risultato, la esegue. L'interprete, anche se impiega più tempo, è in grado di eseguire lo stesso codice su qualunque macchina perché la traduzione verrà fatta ogni volta, senza salvarne poi il risultato.

#### 15) Cosa si intende per byte-code?

Il **byte-code** è un linguaggio intermedio più astratto tra il linguaggio macchina e il linguaggio di programmazione, usato per descrivere le operazioni che costituiscono un programma. Non è il linguaggio macchina di un computer, ma un linguaggio macchina di una macchina virtuale (un computer virtuale) simile a tutti quelli più diffusi. Il programma che effettua questa traduzione è una specie di interprete che combina la compilazione con l'interpretazione. Questo viene chiamato **JVM (Java Virtual Machine)**.

#### 16) Descrivere le caratteristiche del linguaggio Java

Il linguaggio **Java** è un linguaggio ibrido che sfrutta i vantaggi sia del compilatore, in fatto di velocità di traduzione, che dell'interprete, in fatto di portabilità. Per fare ciò **Java** sfrutta il byte code, un particolare file creato dal compilatore partendo dal programma e scritto in uno pseudo linguaggio macchina; questo linguaggio comunque non è ancora riconoscibile dal calcolatore, infatti, il byte - code viene successivamente interpretato dall'interprete che lo tramuta in linguaggio macchina e lo sottopone al calcolatore che ora è in grado di eseguirlo. La **JVM (Java Virtual Machine)** è una macchina/calcolatore virtuale in grado di comprendere il byte code ed è indipendente dall'hardware su cui gira. Ciò rende la portabilità del programma.

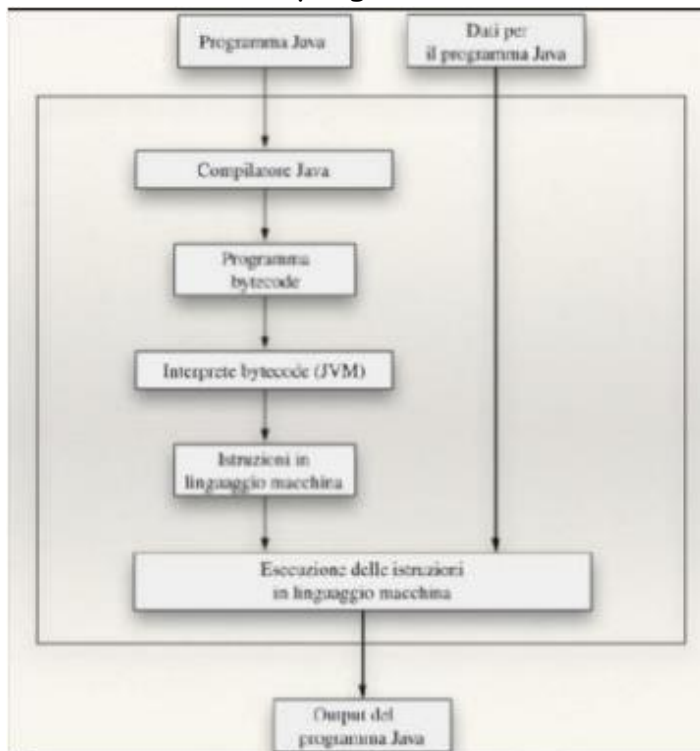
#### 17) Quali sono i procedimenti di esecuzione del codice Java?

Il procedimento è il seguente:

- 1) Compilazione del programma e creazione del byte code;
- 2) Interpretazione del byte code da parte della JVM;



### 3) Esecuzione del programma dal calcolatore;



#### 11) Cosa si intende per classe?

Una **classe** è la definizione di un tipo di oggetto. Una **classe** specifica il nome e il tipo delle variabili di istanza degli **oggetti**, ma non specifica il loro valore. Essa specifica i metodi dei suoi **oggetti**. Un **oggetto** di una istanza della classe.

#### 12) Cosa si intende per Class Loader?

La **Java Class Loader** è una parte di **Java Runtime Environment (JRE)** che carica dinamicamente le classi **Java** nella **Java Virtual Machine (JVM)**. Il sistema runtime **Java** non ha la necessità di conoscere file e file system poiché questo è delegato al caricatore di classi.

#### 13) Quali sono le tipologie di errori di codifica?

Gli errori di codifica che possono impedire l'esecuzione del programma o l'ottenimento di un risultato non desiderato sono:

- 1) **Errori sintattici**: i più facili da risolvere, legati a un non rispetto delle regole sintattiche/grammaticali precisate dal linguaggio. Questi errori sono identificati dal compilatore;
- 2) **Errori di runtime**: errori sollevati in fase di interpretazione ed esecuzione del byte code, non sono identificati dal compilatore. Errori dovuti a operazione illecite e non permesse (divisione per zero, OutOfBound etc.);
- 3) **Errori logici**: errori che potrebbero non intaccare l'esecuzione del

programma ma che portano all'ottenimento di un risultato non voluto. Per risolverli bisogna testare il programma e procedere a tastoni;

#### 14) Differenza tra Applicazioni e Applet

Le **applicazioni** sono programmi regolari pensati per essere eseguiti sul computer locale, mentre **applet** sono piccole applicazioni pensate per essere inviate via Internet ed essere eseguite su un host remoto. Esistono anche le **servlet** (mondo dei **Sistemi Distribuiti**), che sono degli oggetti creati in Java che operano all'interno del **server Web** oppure in un **server** per applicazioni che permettendo la creazione di **applicazioni web**.

#### 15) Sintassi di Java

Il linguaggio **Java** è composto da una serie di elementi lessicali, cioè parole che identificano le più piccole unità a cui attribuire un significato. Questi elementi sono divisi in:

- parole e caratteri riservati del linguaggio (public, for, while, =, etc.);
  - numeri;
  - stringhe, cioè sequenza di caratteri;
  - identificatori, definiti dal programmatore e utilizzati per identificare elementi del programma quali classi, variabili, metodi etc.
- Tutti gli **identificatori** (classi, variabili etc.) devono rispettare le seguenti regole in modo da non sollevare errori sintattici in fase di compilazione:
- non devono iniziare con numeri;
  - non devono essere presenti spazi;
  - sono ammesse solo lettere, numeri e i soli caratteri **\_** e **\$** e nessun altro carattere è ammesso;
  - non usare parole riservate;
  - due identificatori con le stesse scope non possono avere lo stesso nome;

#### 16) Significato di scope

Con il termine **scope** di un identificatore si fa riferimento alla sua visibilità. Ogni identificatore ha un determinato **scope** e quindi è visibile solo per certi metodi/classi.

#### 17) Cosa si intende per variabile?

Le **variabili** sono locazioni di memoria (una riga della ipotetica tabella) utilizzate per salvare dei dati.

Ogni **variabile** deve essere definita con:



- un identificatore, il suo nome, necessario per farvi riferimento;
- un tipo, per definire la tipologia di variabile che andremo a memorizzare (intero, stringa etc.);
- uno scope, che mi dice in che area del programma è utilizzabile.

Il **tipo di una variabile** determina il valore che può contenere e le operazioni che sono consentite su di essa. I tipi in **Java** si dividono in:

- **primitivi**, più semplici e non scomponibili (int, float etc.);
- **non primitivi**, più complessi e composti a loro volta da altri tipi (Scanner, String, Array etc.).

### 18) Cosa si intende per casting implicito ed esplicito? Per type cast?

Considerando il seguente ordine:

**byte** → **short** → **int** → **long** → **float** → **double**

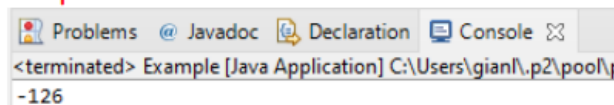
alle variabili bisogna assegnare valori dello stesso tipo, ma, tramite l'operazione di **casting** (modifica) si è in grado di assegnare a variabili di un certo tipo, valori o altre variabili di un altro tipo. Il casting può essere:

- **implicito**: eseguito direttamente dall'esecutore senza particolari specifiche ed è ammesso solo se si va ad assegnare a una variabile di **grandezza M**, una **variabile di grandezza minore di M**, seguendo quindi lo schema sopra riportato;
- **esplicito**: specificato dal programmatore e che permette di assegnare a una variabile di grandezza M un valore di qualunque grandezza M, anche maggiore. In quel caso il valore assegnato verrà troncato per difetto, se necessario.

**Esempio:**

```
int x = 130;  
byte y = (byte)x;  
System.out.println(y);
```

**Output**



**Attenzione**

- se si divide tra loro due numeri interi sto eseguendo una divisione intera e infatti mi viene ritornato come risultato la sola parte intera della divisione. Per ottenere una divisione esatta devo utilizzare almeno un double o un float;
- l'operatore %, utilizzato per sapere il solo resto della divisione, può essere usato anche tra valori a virgola mobile e agisce in questo modo:

$$N \% D = N - (D * Q) \text{ dove } Q \text{ è la parte intera di } N/D$$

Per **type cast** si intende l'operazione con cui si converte una variabile da un tipo

di dato a un altro.

**Esempio:**

```
double distance;  
distance = 9.0;  
int points;  
points = (int)distance;
```

**19) Cosa sono le espressioni e quali sono le sue caratteristiche?**

Le espressioni sono un misto di variabile, detti operandi, e operatori che producono come risultato un valore che può essere un valore intero, in virgola mobile, booleano etc. Gli operatori Java si dividono in:

- **unari**: agiscono su un solo valore (incremento, decremento, opposto, negazione etc.) e tali operatori ammettono una notazione prefissa, prima dell'operatore, dando così all'espressione precedenza massima, oppure notazione postfissa, dopo l'operatore, dando all'espressione precedenza minima;
- **binari**: agiscono su due valori (somma, differenza, prodotto, divisione etc.) e tali operatori, come i ternari, ammettono la sola notazione infissa, tra i valori;
- **ternari**: richiedono tre operatori. L'unico è la condizione: Vero? Falso;

Ogni **operatore** ha una **precedenza ben definita** che determina poi l'ordine con cui vengono eseguite le varie **sotto operazioni** (metodologia **top - down**) che compongono l'**operazione**.

Gli operatori vengono classificati in:

- aritmetici;
- aritmetici di incremento e decremento;
- assegnamento;
- relazionali;
- condizionali;
- bit a bit.

Gli operatori aritmetici (validi per interi e floating-point)

Operatore	Descrizione	Uso	Significato
+	Somma	op1+op2	Somma il valore di op1 a quello di op2
-	Sottrazione	op1-op2	Sottrae al valore di op1 quello di op2
*	Moltiplicazione	op1*op2	Moltiplica il valore di op1 con quello di op2
/	Divisione	op1/op2	Divide il valore di op1 con quello di op2
%	Modulo	op1%op2	Calcola il resto della divisione tra il valore di op1 e quello di op2
-	Negazione aritmetica	-op	Trasforma il valore di op in positivo o negativo

Gli operatori hanno una precedenza ben definita che determina l'ordine con

cui vengono valutati. Gli operatori che hanno la stessa precedenza sono valutati da sinistra verso destra e mediante le parentesi è possibile alterare l'ordine di precedenza.

### Esempio:

<code>a + b + c + d + e</code>	<code>a / (b+c) - d % e</code>
<code>a + b * c + d / e</code>	<code>a / (b * (c + (d-e)))</code>

### Precedenza:

<code>a + b + c + d + e</code> 1    2    3    4	<code>a / (b+c) - d % e</code> 2    1    4    3
<code>a + b * c + d / e</code> 3    1    4    2	<code>a / (b * (c + (d-e)))</code> 4    3    2    1

**Attenzione:** l'operatore `=` è quello con precedenza minore perché sempre eseguito alla fine per assegnare il valore ottenuto dalla computazione dell'espressione alla variabile;

## 20) Descrivere la differenza tra gli operatori pre incremento e post incremento

L'operatore **pre/incremento** incrementa la variabile prima di essere utilizzata, mentre l'operatore **post incremento** incrementa la variabile dopo essere stata utilizzata.

### Esempio 1:

```
int x = 130;
int risultato0 = ++x * 3;
```

```
System.out.println("R0 = " + risultato0);
System.out.println("x = " + x);
```

### Output

```
R0 = 393
x = 131
```

### Esempio 2:

```
int x = 130;
int risultato0 = x++ * 3;
```

```
System.out.println("R0 = " + risultato0);
System.out.println("x = " + x);
```

### Output

```
R0 = 390
x = 131
```

## 21) Descrivere il flusso di controllo e il branching statement (selezione)

→ Il flusso di controllo è l'ordine con cui il programma svolge le azioni.

→ Un'istruzione di selezione (Branching Statement) sceglie tra due o più azioni possibili.

→ Un **ciclo** (loop) ripete un'azione fino a che non si verifica una condizione di stop.

### Istruzione if - else semplice

→ È un'istruzione che, in base alla valutazione della condizione, sceglie tra 2 statement possibili.

### Come comportarsi se uno Statement fosse composto da più istruzioni?

Se occorre più di un'istruzione in ciascuno dei due rami definiti dall'istruzione **if - else**, è sufficiente racchiudere le diverse istruzioni tra parentesi graffe {}.

In questo caso l'insieme di istruzioni racchiuse tra parentesi graffe è considerato come un'unica istruzione "più ampia".

```
if (count<3) {
    totale = 0;
    count = 0;
}
```

### Omissione di else

```
if (Espressione_booleana)
    Statement_1;
```

### Cosa si intende per espressione booleana?

È un misto di variabili e operatori il cui risultato può assumere soltanto due valori: **true** o **false**. La forma più elementare è il confronto tra due variabili, come ad esempio:

```
tempo < tempolimito
peso <= 50
```

Gli operatori di confronto in Java sono:

Math Notation	Name	Java Notation	Java Example
=	Equivalente	== (no =)	valore == 0
!=	Diverso	!=	valore != 0
>	Maggiore di	>	valore > 0
<	Minore di	<	valore < 0
>=	Maggiore o uguale	>=	valore>=0
<=	Minore o uguale	<=	valore<=0

A volte però, è necessario costruire espressioni booleane un po' più complesse e, in base alle specifiche, è possibile costruire espressioni booleane con diverse modalità:

1) con **operatore AND (&&)**: utilizzato ad esempio per verificare se il valore è compreso tra 0 e 100 inclusi:

```
if ((value>=0) && (value<=100))
```

**Sintassi**

```
if ((condizione1) && (condizione2))
```

**Importante:** l'espressione è vera solamente se entrambe le condizioni (o sotto espressioni) sono vere! (prodotto logico - **AND** logico).

2) Con **operatore OR (||)**: utilizzato, ad esempio, per verificare se un valore è minore di 0 o maggiore di 100:

```
if ((valore<0) || (valore>100))
```

**Sintassi**

```
if ((condizione1) || (condizione2))
```

**Importante:** l'espressione è vera se una delle due condizioni (o sotto espressioni) è vera! (somma logica - **OR** logico)

3) Con **operatore NOT (!)**: utilizzato per negare un'espressione booleana:

```
if ((a || b) && !(a && b))
```

**Sintassi:**

```
!(Espressione_booleana)
```

**Operatori Logici**

!(A Op1 B)	(A Op2 B)
<	>=
<=	>
>	<=
>=	<
==	!=
!=	==

Operatore Logico	Simbolo	Esempi Java
AND	&&	(n>=0) && (n<=100)
OR		(n<0)    (n>100)
NOT	!	(n<0) && !(n>100)

**Tableaux (Tavola di Verità)**

0 = **false**

1 = **true**

A	B	A && B	A    B	!A	!B
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	1
1	1	1	1	0	0

### Cosa si intende per if - else annidato (o innestato)?

È un'istruzione **if - else** che può essere contenuta (annidata o innestata) nel ramo **if**, nel ramo **else** o in entrambi rami.

#### Sintassi:

```
if (Espressione_booleana_1) {  
    if (Espressione_booleana_2) {  
        Statement_1;  
    }  
    else {  
        Statement_2;  
    }  
}
```

#### Attenzione:

→ Ogni **else** è associato all'istruzione **if** più vicino e l'indentazione rende chiara l'associazione del ramo **else** al ramo **if**.

→ Le **parentesi graffe** ({} ) possono essere utilizzate per raggruppare le istruzioni.

**Esempio:** notare bene la differenza!

```
if (a > b) {  
    if (c > d)  
        e = f;  
}  
else  
    g = h;
```

Il ramo **else** è associato al ramo **if** con condizione (a>b). Se questa è vera allora si entra nel ramo e viene valutata la seconda condizione (c>d).

#### Infatti:

```
int a = 2;  
int b = 1;  
int c = 4;  
int d = 3;  
int e = 5;  
int f = 6;  
int g = 7;  
int h = 8;  
  
if (a > b) {  
    if (c > d)  
        e = f;  
}  
else  
    g = h;  
  
System.out.println("a = " + a);  
System.out.println("b = " + b);  
System.out.println("c = " + c);  
System.out.println("d = " + d);  
System.out.println("e = " + e);  
System.out.println("f = " + f);  
System.out.println("g = " + g);  
System.out.println("h = " + h);
```



Discutere dell'utilizzo dell'operatore di confronto ==

→ L'operatore == può essere utilizzato per determinare se due interi o due caratteri hanno lo stesso valore.

```
int a;  
a = ...;  
if (a == 3)
```

→ Questo operatore, però, non è indicato per determinare se due valori in virgola mobile sono uguali. Per determinare ciò, si utilizza < e una determinata tolleranza sul valore assoluto.

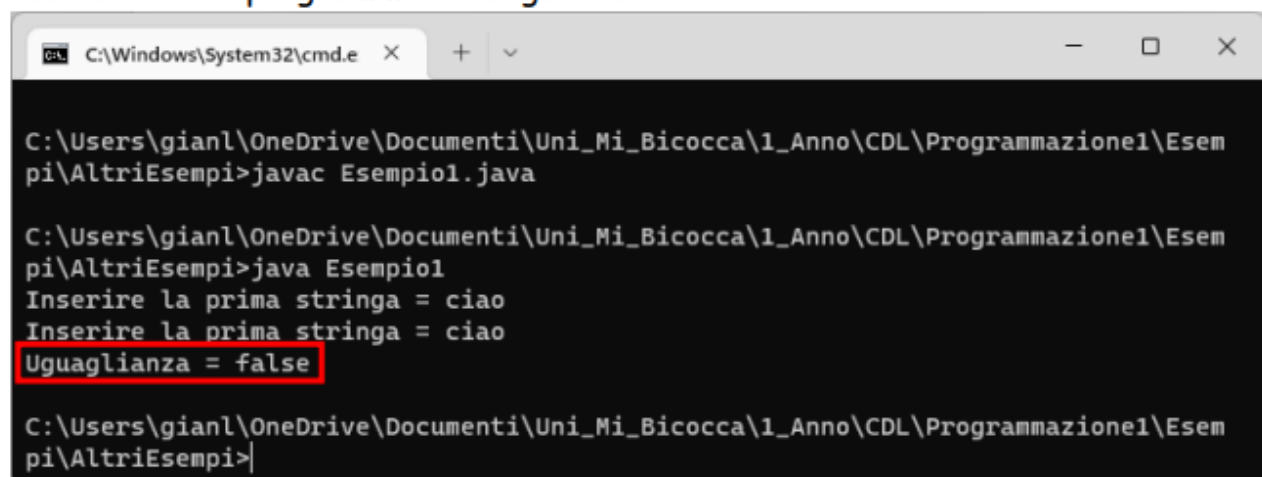
```
if (Math.abs(b-c) < epsilon)  
  
b, c ed epsilon sono floating point.
```

→ Inoltre l'utilizzo di == non è appropriato per determinare l'uguaglianza fra due **oggetti!!**

**Esempio:**

```
String s1;  
String s2;  
Scanner input = new Scanner(System.in);  
boolean uguaglianza;  
  
System.out.print(s: "Inserire la prima stringa = ");  
s1 = input.next();  
System.out.print(s: "Inserire la prima stringa = ");  
s2 = input.next();  
if (s1 == s2)  
|   uguaglianza = true;  
else  
|   uguaglianza = false;  
System.out.println("Uguaglianza = " + uguaglianza);
```

Ci si aspetta che, inserendo due stringhe equivalenti, venga stampato true, ma l'esecuzione del programma è il seguente:



```
C:\Windows\System32\cmd.e  X  +  v  -  □  X  
  
C:\Users\gian\OneDrive\Documenti\Uni_Mi_Bicocca\1_Anno\CDL\Programmazione1\Esem  
pi\AltriEsempi>javac Esempio1.java  
  
C:\Users\gian\OneDrive\Documenti\Uni_Mi_Bicocca\1_Anno\CDL\Programmazione1\Esem  
pi\AltriEsempi>java Esempio1  
Inserire la prima stringa = ciao  
Inserire la prima stringa = ciao  
Uguaglianza = false  
  
C:\Users\gian\OneDrive\Documenti\Uni_Mi_Bicocca\1_Anno\CDL\Programmazione1\Esem  
pi\AltriEsempi>
```

### Perché??

Il motivo, per cui viene stampato false, è che l'operatore == non confronta l'uguaglianza dei contenuti dei due oggetti, ma i loro indirizzi di memoria. L'operatore == stabilisce se i due oggetti sono allocati nella stessa area di memoria!!

### Come risolvere questo problema?

Per testare l'uguaglianza tra due oggetti della classe String, occorre utilizzare il metodo `equals`.

In questo caso:

```
import java.util.Scanner;

public class Esempio1 {
    Run | Debug
    public static void main(String[] args) {
        String s1;
        String s2;
        Scanner input = new Scanner(System.in);
        boolean uguaglianza;

        System.out.print(s: "Inserire la prima stringa = ");
        s1 = input.next();
        System.out.print(s: "Inserire la prima stringa = ");
        s2 = input.next();
        if (s1.equals(s2))
            uguaglianza = true;
        else
            uguaglianza = false;
        System.out.println("Uguaglianza = " + uguaglianza);
    }
}
```

### Output

```
C:\Users\gianl\OneDrive\Documenti\Uni_Mi_Bicocca\1_Anno\CDL\Programmazione1\Esem
pi\AltriEsempi>javac Esempio1.java

C:\Users\gianl\OneDrive\Documenti\Uni_Mi_Bicocca\1_Anno\CDL\Programmazione1\Esem
pi\AltriEsempi>java Esempio1
Inserire la prima stringa = ciao
Inserire la prima stringa = ciao
Uguaglianza = true
```

Per testare l'uguaglianza ignorando maiuscole e minuscole, utilizzare il metodo `equalsIgnoreCase`.

```
if (s1.equalsIgnoreCase(s2))
    uguaglianza = true;
else
    uguaglianza = false;
```

### Operatore ternario (? : )

È possibile riscrivere l'istruzione **if - else** in un modo più compatto mediante l'**operatore ternario (o condizionale)**.

```
if (n1>n2)
    max = n1;
else
    max = n2;
```

Può essere riscritto come:

```
max = (n1>n2) ? n1 : n2;
```

### Che cosa si intende per valutazione short - circuit? Quali sono le sue caratteristiche?

La valutazione short - circuit è un meccanismo di valutazione di una condizione che permette solamente di valutare una parte di una espressione booleana per determinare il valore dell'espressione intera.

→ Se il primo operando associato con **||** è **true**, l'espressione completa sarà **true**.

→ Se il primo operando associato con **&&** è **false**, l'espressione completa sarà **false**.

Essa ha le seguenti caratteristiche:

- la valutazione short-circuit non solo è efficiente, talvolta è anche essenziale;
- può prevenire un errore a run-time, ad esempio, a causa di un tentativo di dividere per zero;
- viene effettuata solo con operatori **&&** o **||**.
- può essere disabilitata sostituendo **&** (**AND Bitwise**) al posto di **&&** oppure **|** (**OR Bitwise**) al posto di **||**.

### Descrivere l'istruzione switch

→ L'istruzione **switch** è una selezione multi ramo che effettua una scelta sulla base di un intero, di un carattere o di una stringa.

→ L'istruzione **switch** inizia con la keyword **switch** seguita da un'espressione di controllo riportata tra parentesi tonde.

→ A seguire, una lista di casi, racchiusi tra parentesi graffe.

→ Ogni caso è formato dalla keyword **case** seguito da:

- 1) una costante chiamata etichetta **case** (case label);
- 2) due punti (:);
- 3) una sequenza di espressioni.

→ All'interno della lista viene cercata l'etichetta che è uguale all'espressione di controllo. L'azione associata al caso corrispondente viene eseguita.

→ Se l'espressione di controllo non è uguale a nessun caso, viene eseguito il caso etichettato **default** (opzionale, ma raccomandato).

→ Non è permesso ripetere le case labels: ognuna può e deve apparire solamente una volta.

### Sintassi

```
switch (espressione_di_controllo) {  
    case etichettacaso1: azione(i);  
                        break;  
    case etichettacaso2: azione(i+1);  
                        break;  
    ...  
    default:  
    ...  
}
```

→ Le azioni da intraprendere, corrispondenti ad ogni caso, tipicamente terminano con la keyword **break**. Essa è opzionale e previene che vengano considerati altri casi.

### Cosa si intende per visibilità delle variabili?

→ La **visibilità di una variabile** è la porzione di programma in cui tale variabile può essere utilizzata.

→ La **visibilità di una variabile locale** (cioè definita in un **blocco**) è la porzione di programma che va dalla dichiarazione della variabile stessa fino alla fine del blocco che la contiene.

→ Limitare la visibilità di una variabile al blocco che la contiene permette di riutilizzare nomi di variabili in parti diverse del programma, anche con tipi diversi.

→ Ovviamente le variabili dichiarate in un blocco non possono avere il nome di variabili esterne al blocco stesso. Il compilatore darebbe errore!

```
int a=0;  
  
{  
    int a=0; // ERRORE: variabile già dichiarata  
    int b=10;  
    System.out.println(a+b);  
}
```

### Cosa è un ciclo?

→ Un **ciclo (loop)** è la porzione di programma che ripete un'istruzione o un gruppo di istruzioni.

→ Il gruppo di istruzioni che vengono ripetuti sono chiamati corpo del ciclo.

→ Ogni ripetizione è chiamata iterazione del ciclo.

→ È necessario, inoltre, imporre una condizione che imponga la terminazione del ciclo.



### Descrivere le caratteristiche del ciclo while

→ L'istruzione **while** (detto anche **ciclo while**) ripete più volte l'azione definita nel corpo del ciclo finché un'espressione booleana di controllo rimane vera. Per questo motivo che viene nominato ciclo **while** (ciclo **mentre**).

→ Quando l'espressione diviene **falsa**, il ciclo termina.

→ Il corpo del ciclo tipicamente contiene un'azione che fa diventare falsa l'espressione di controllo.

→ Prima viene valutata la condizione e se questa risulta vera, allora viene eseguito il corpo del ciclo.

#### Sintassi

```
while (espressione_booleana)
    corpo
```

#### oppure

```
while (espressione_booleana) {
    istruzione1;
    istruzione2;
    ....
    istruzioneen;
}
```

### Descrivere le caratteristiche del ciclo do - while

→ È simile all'istruzione **while**, eccetto che il corpo del ciclo viene eseguito almeno una volta.

#### Sintassi

```
do {
    istruzione1;
    istruzione2;
    ....
    istruzioneen;
} while (espressione_booleana);
```

→ Per prima cosa, viene eseguito il corpo del ciclo.

→ Poi viene valutata l'espressione booleana di controllo:

1) Finché è vera, il ciclo viene eseguito di nuovo.

2) Se è falsa, si esce dal ciclo.

#### Sintassi

```
istruzione(i)_S1
while (Espressione_Booleana)
    istruzione(i)_S1
```

#### Semantica

```
istruzione(i)_S1
while (Espressione_Booleana)
    istruzione(i)_S1
```

### Cosa è un ciclo infinito?

→ Un ciclo che ripete il proprio corpo senza mai terminare è chiamato ciclo infinito (infinite loop).

→ Normalmente le istruzioni presenti nel corpo di un ciclo **while** o **do-while** alterano in qualche modo una o più variabili così che prima o poi l'espressione booleana di controllo diventi falsa.

→ Se tuttavia la/le variabili non cambiano in modo corretto, si potrebbe generare un ciclo infinito.

### Cosa è un ciclo annidato?

È un ciclo che include un altro ciclo.

### Descrivere le caratteristiche del ciclo for

→ L'istruzione **for**, a differenza del **while** esegue il corpo del ciclo un numero prefissato di volte (il numero di iterazioni è noto a priori).

#### Sintassi

```
for (Inizializzazione; Condizione; Aggiornamento)
    Corpo
```

**Corpo** → può essere sia un'istruzione semplice che un blocco {}.

#### Istruzione while corrispondente

```
Inizializzazione

while (Condizione)
    Corpo_con_Update
```

#### Semantica

```
for (Inizializzazione; Condizione; Aggiornamento)
    Corpo
```

Le caratteristiche del **ciclo for** sono:

→ È possibile dichiarare variabili all'interno dell'istruzione **for**.

#### Esempio

```
int somma = 0;
for (int n = 1 ; n <= 10 ; n++)
    somma = somma + n * n;
```

→ Notare però che **n** è **locale al ciclo** (la sua visibilità, o scope, è solamente all'interno del ciclo).



→ È possibile fare inizializzazioni multiple separandole con una virgola.

#### Esempio

```
for (n = 1, prodotto = 1; n <= 10; n++)  
    prodotto = prodotto * n;
```

→ Nella condizione è possibile inserire una sola espressione booleana (ma ovviamente può contenere **&&**, **||**).

→ È possibile anche inserire aggiornamenti multipli:

```
for (n = 1, prodotto = 1; n <= 10; prodotto = prodotto * n, n++);
```

#### Cosa si intende per empty statement

Questa istruzione viene rappresentata mediante un punto virgola al di fuori dei cicli (for, **while**, **do - while**) ed è un ciclo vuoto che viene ripetuto per n volte.

#### 1) Descrivere le caratteristiche dei metodi

→ Alcune sequenze di istruzioni possono dover essere ripetute più volte all'interno di un programma.

→ Risulta quindi comodo poter scrivere tali sequenze una volta sola e poter far riferimento ad esse all'interno del programma tutte le volte che la loro esecuzione risulta necessaria.

→ I **metodi** costituiscono lo strumento di programmazione che realizza quanto descritto.

→ Il **metodo** è una porzione di codice riutilizzabile in diverse aree del programma. Raggruppa una sequenza di istruzioni che realizzano una funzionalità del programma e assegna loro un **nome**.

→ Ogni qualvolta è necessario eseguire quella funzionalità, è sufficiente richiamarla attraverso il **nome**.

→ Quando si utilizza un **metodo**, si dice che esso viene "**chiamato**" o "**invocato**".

→ In **Java** esistono due tipologie di metodi:

1) metodi che restituiscono un valore;

2) metodi che eseguono delle istruzioni ma non restituiscono alcun valore (**void**).

#### 2) Come si utilizzano i metodi?

→ In primo luogo occorre definire il metodo scrivendo la sequenza di istruzioni e assegnando alla sequenza un nome.

→ La definizione viene effettuata una sola volta e all'interno di una **classe**.

→ Una volta definito il **metodo**, è possibile invocare il metodo usando il nome del **metodo**. Quando viene invocato un **metodo**, vengono eseguite le istruzioni definite al suo interno.

→ Quando tutte le istruzioni del metodo sono state eseguite, l'esecuzione viene ripristinata nella posizione in cui era stata eseguita la chiamata al metodo.

### 3) Come vengono definiti i metodi?

- Un **metodo** è definito all'interno di una classe. Pertanto si dice che un metodo appartiene alla classe in cui è stato definito.
- Se un **metodo** viene definito **public** può essere invocato in ogni area del programma, anche in classi diverse da quelle in cui è stato definito.
- La parola chiave **static** è un altro modificatore che regola il modo con cui il metodo può essere invocato: **static** indica che è un metodo di classe.

### 3) Descrivere le caratteristiche dei metodi void

- Un **metodo** di tipo **void**, viene utilizzato quando non si ha la necessità di restituire alcun tipo di valore. Dopo il termine **void**, il nome del metodo è seguito da una coppia di parentesi tonde (), dove all'interno delle quali **possono** essere elencati gli argomenti di cui il metodo necessita per poter eseguire il **corpo (body)** in esso definite.
- La definizione del metodo viene detta intestazione (**heading**) del metodo.
- Dopo l'intestazione viene riportata la parte rimanente della definizione di un metodo: il **corpo (body) del metodo**. Le istruzioni contenute nel corpo del metodo sono racchiuse tra parentesi graffe {}.
- Le variabili dichiarate all'interno del metodo vengono dette **variabili locali**.

Esempio:

```
public static void saluta() {  
    System.out.println("Ciao");  
}
```

### 4) Come avviene l'invocazione del metodo void?

- L'invocazione di un metodo **void** avviene semplicemente scrivendo un'istruzione che include il nome del metodo seguito da una coppia di parentesi e da un punto e virgola.
- Tra le parentesi è indicato il valore degli argomenti di cui il metodo necessita per eseguire le istruzioni in esso eseguite. Se il metodo invocato non richiede argomenti, tra le due parentesi non viene riportato nulla.
- Un **metodo** appartiene alla classe in cui è definito. Esso può essere invocato all'interno di altri metodi definiti nella sua stessa classe. Se il modificatore del metodo è di tipo **public**, il metodo può essere invocato anche al di fuori della classe in cui è stato definito.

**Esempio:**

```
import java.util.Scanner;

public class Saluta {
    public static void saluta() {
        System.out.println("Ciao");
    }
    public static void main(String[] args) {
        System.out.println("Prima dell'esecuzione.");
        saluta();
        System.out.println("Dopo dell'esecuzione.");
    }
}
```

**5) Definizione di metodi che restituiscono un valore**

→ I metodi che restituiscono un valore vengono definiti in maniera simile ai metodi **void**, con l'aggiunta della specifica del tipo di valore che restituiscono. Il corpo della definizione di un metodo che restituisce un valore è come il corpo di un metodo **void**, con l'aggiunta dell'istruzione **return** al suo interno.

**Esempio:**

```
public class AreaQuadrato {
    public static double areaQuadrato(double lato) {
        return lato*lato;
    }
    public static void main(String[] args) {
        System.out.println("Area Quadrato = " + areaQuadrato(2.5));
    }
}
```

**Output**

```
C:\Users\gianl\OneDrive\Documenti\Uni_Mi_Bicocca\1_Anno\CDL\PrimoSemestre\Programmazio
nel\Teoria\Esempi\AltriEsempi>javac AreaQuadrato.java

C:\Users\gianl\OneDrive\Documenti\Uni_Mi_Bicocca\1_Anno\CDL\PrimoSemestre\Programmazio
nel\Teoria\Esempi\AltriEsempi>java AreaQuadrato
Area Quadrato = 6.25
```

→ Un **metodo** che restituisce un **valore**, lo si può invocare in qualsiasi punto del codice in cui si potrebbe usare un elemento dello stesso tipo di ritorno del metodo.



## 6) Definizione di variabili locali

→ Una **variabile locale** è una variabile dichiarata all'interno di un metodo e possono essere utilizzate solamente all'interno del metodo. Anche le variabili dichiarate all'interno del **main** sono locali al **main**.

→ Le **variabili locali** aventi lo stesso nome, ma dichiarate in metodi diversi sono variabili a tutti gli effetti differenti (**non sono la stessa variabile**).

## 7) Parametri di tipo primitivo

```
public class AreaQuadrato {  
    public static double areaQuadrato(double lato) {  
        double risultato = lato*lato;           //Variabile 1  
        return risultato;  
    }  
    public static void main(String[] args) {  
        double risultato = areaQuadrato(30.6);  
        System.out.println("Area Quadrato = " + risultato); //Variabile 2  
        System.out.println("Area Quadrato = " + areaQuadrato(2.5));  
    }  
}
```

→ Lato è un parametro (di tipo **double**), detto anche parametro formale. Questo parametro rappresenta una sorta di "sostituto temporaneo" di un valore effettivo che sarà disponibile solo al momento dell'invocazione del metodo. Il valore effettivo è chiamato **argomento**.

→ I **nomi dei parametri** sono locali al metodo, per cui anch'essi sono locali al metodo.

→ Quando il **metodo** viene invocato, ogni parametro viene inizializzato al valore dell'argomento corrispondente (**chiamata per valore**). Se come **argomento nell'invocazione di un metodo** si ha una variabile di un tipo primitivo, l'invocazione del metodo non può cambiare il valore dell'argomento stesso. Ad ogni parametro del metodo deve corrispondere un argomento dello stesso tipo.

→ Questa regola non è così restrittiva! Viene fatta una conversione di tipo automatica qualora nell'invocazione venga usato un argomento il cui tipo non corrisponde a quello del parametro. La conversione automatica, è la solita:

byte -> short -> int -> long -> float -> double

## 9) Cosa succede quando si invoca un metodo?

→ Quando si invoca un **metodo**, l'esecuzione passa al corpo del metodo creando in **memoria RAM**, una struttura dati chiamata **record di attivazione**, che contiene tutte le informazioni necessarie per gestire correttamente l'esecuzione del metodo invocato.

→ Le informazioni relative al metodo sono:

- 1) **parametri formali** del metodo;
- 2) **variabili locali** al metodo;
- 3) **indirizzo di rientro (Instruction Pointer)**: quando il metodo termina, il controllo torna al chiamante, che deve riprendere la sua esecuzione dall'istruzione successiva alla chiamata del metodo;
- 4) **risultato**: se il metodo chiamato restituisce un valore, tale valore viene copiato nel campo risultato del chiamante.

parametro 1: valore
parametro 2: valore
.
.
.
parametro n: valore
variabile locale 1: valore
variabile locale 2: valore
.
.
.
variabile locale n: valore
rientro: indirizzo di memoria
risultato: valore

#### 10) Descrivere il record di attivazione

→ Il **record di attivazione** viene creato dinamicamente nel momento in cui il metodo viene chiamato e viene posto in cima ad **un'area di memoria (statica)** denominata **stack**.

→ Esso rimane nello **stack** per tutto il tempo in cui il metodo è in esecuzione e viene rimosso al termine dell'esecuzione. Metodi che invocano altri metodi danno luogo ad una sequenza di record di attivazione. La sequenza viene gestita secondo la politica **LIFO (Last In First Out)**.

### 11) Istruzione return

→ L'istruzione **return** può essere utilizzato più volte all'interno del corpo di un metodo che restituisce un valore, ma è preferibile utilizzarne una volta sola.

Inserire un'unica istruzione **return** vicino alla fine del corpo del metodo lo rende più semplice da leggere

Esempio:

```
public static int maggiore(int primo, int secondo) {  
    int risultato;  
    if (primo >= secondo)  
        risultato = primo;  
    else  
        risultato = secondo;  
    return risultato;  
}
```

```
public static int maggiore(int primo, int secondo) {  
    if (primo >= secondo)  
        return primo;  
    else  
        return secondo;  
}
```

### 12) Come possono essere collaudati i metodi?

→ Per verificare la correttezza di un metodo si può usare un programma **driver**. Sono molto semplici da utilizzare e devono solo fornire al metodo da collaudare una serie di argomenti e invocare il metodo stesso. Ogni **metodo** definito in una classe dovrebbe essere collaudato individualmente.

→ Un primo modo per collaudare ciascun metodo separatamente è detto **testing bottom-up** (letteralmente "collaudo dal basso verso l'alto").

Se un metodo A invoca il metodo B, questo approccio prevede che:

- 1) il metodo B viene collaudato per prima;
- 2) il metodo A viene collaudato dopo.

### 13) Cosa si intende per stub?

Il metodo **stub** è una porzione di codice utilizzata per simulare il comportamento di funzionalità software e può fungere anche da temporaneo sostituto di codice ancora da sviluppare. Sono utili durante il porting del software e durante lo sviluppo di un software o software testing.



#### 14) Descrivere le caratteristiche degli array

→ Un **array** è una **sequenza di variabili di tipo omogeneo** (cioè dello stesso tipo) e distinguibili l'una dall'altra (indirizzabili) in base alla loro posizione all'interno della sequenza (indici interi).

→ L'**array** è un particolare tipo di oggetto, ma è possibile considerarlo come una collezione di variabili dello stesso tipo.

##### Esempio:

```
double[] temperatura = new double[7];
```

→ Per accedere ad un elemento dell'array si usa il nome dell'array e un indice riportato tra parentesi quadre (ricorda che in qualsiasi **linguaggio di programmazione** (Java, C++, C), gli indici degli array partono da zero).

##### → Rappresentazione di un array

32	30	25.7	26	34	31.5	29
----	----	------	----	----	------	----

##### → Dichiarazione e creazione di un array

La dichiarazione e creazione di un array avviene mediante l'operazione **new**.

##### Sintassi

```
tipo_base[] nome_array = new tipo_base[dimensione];
```

##### Oppure

```
tipo_base nome_array[] = new tipo_base[dimensione];
```

##### Esempio

```
char[] simbolo = new char[80];  
double[] valore = new double[100];  
char simbolo[] = new char[80];  
double valore[] = new double[100];
```

#### Variabile di istanza length

→ Un **array** è un particolare tipo di oggetto che può avere delle proprietà, dette **variabili di istanza**.

→ È possibile far riferimento alle proprietà accessibili degli oggetti usando:

```
nome_oggetto.proprietà;
```

→ Un **array** ha una sola proprietà accessibile: **length**

Questa proprietà indica il numero di elementi dell'array. Il suo valore non può essere cambiato.

#### Indici degli array

→ L'indice del primo elemento dell'array è **0**;

→ L'ultimo indice valido è quindi **nomeArray.length-1**.

→ Gli indici dell'array devono essere nei limiti affinché siano validi. Quando il programma cerca di accedere ad un elemento con indice al di fuori dei limiti dell'array, viene generato errore a **run time**, o meglio eccezione (**array index out of bound**).

### 15) Come possono essere inizializzati gli array?

→ È possibile inizializzare un array in fase di dichiarazione:

```
double[] valore = {3.3, 15.8, 9.7};
```

→ In questo caso la sua lunghezza viene fissata al minimo necessario per contenere i valori elencati. L'istruzione precedente è quindi equivalente a:

```
double [] valore = new double[3];  
valore[0]=3.3;  
valore[1]=15.8;  
valore[2]=9.7;
```

### 16) Cosa succede se gli elementi di un array non vengono inizializzati esplicitamente?

→ Vengono inizializzati con il valore di default del loro tipo base (ad esempio con **int** vengono inizializzati a 0, **double** a 0.0).

→ È preferibile inizializzare il vettore in maniera esplicita in tre tipologie:

- 1) con valori tra parentesi graffe;
- 2) leggendo i valori da tastiera;
- 3) tramite un ciclo, ad esempio:

```
int [] valore = new int[3];  
  
for (int i=0;i<valore.length;i++)  
    valore[i] = 0;
```

### 17) È possibile riempire parzialmente un array?

→ È possibile che non tutti gli elementi di un array siano effettivamente riempiti con un valore. In questo caso si parla di array riempiti parzialmente.

→ La capacità dell'array `a` è `a.length` e questo corrisponde al numero di elementi che sono stati creati quando si è definito l'array. È possibile non utilizzare tutti questi elementi.

Esempio:

24.5	23.7	26.9	0.0	0.0
------	------	------	-----	-----

### 18) Come avviene la creazione di un array?

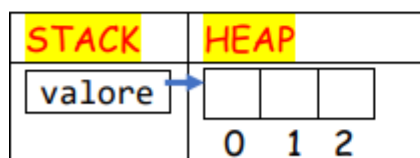
→ Come già detto, l'array è un tipo di dato non primitivo utilizzato per correlare valori dello stesso tipo.

→ Dal punto di vista della memoria RAM, l'array è una sequenza di valori, tutti dello stesso tipo, di lunghezza finita da specificare in fase della dichiarazione dello stesso.

Esempio

```
double [] valore = new double[3];
```

Con questa dichiarazione si va riservare 3 celle nell'area di memoria dinamica, chiamata **HEAP**. La variabile `valore` viene chiamato **referimento** (o **puntatore**!!!) dell'array creato ed è allocato nell'area di memoria dinamica denominata **STACK**. Il **referimento** non contiene il valore della variabile ma l'indirizzo di memoria, in cui viene allocato l'array nello **HEAP**.



```
double [] valore = new double[3];  
System.out.println("Indirizzo valore: " + valore);
```

Output

```
Indirizzo valore: [D@3fee733d]
```

### 19) Come è possibile utilizzare gli array nei metodi?

→ Una variabile indicizzata può essere usata come argomento di un metodo in tutte le situazioni in cui può essere utilizzato il tipo base dell'array.

**Esempio:**

```
double var = 0.1;
double[] a = new double[10];
int indice = 4;

metodo1(a[3]);
metodo1(a[indice]);
metodo1(var);
}
public static void metodo1(double p1) {
    //corpo_del_metodo
}
```

→ La modalità con cui si specifica che l'argomento di un metodo è un **array** è simile al modo con cui si dichiara un **array**.

**Esempi**

```
public static int getAnElement(char[] anArray, int index);
public static void readAnArray(int[] anArray);

public static int getAnElement(char anArray[], int index);
public static void readAnArray(int anArray[]);
```

**Nota**

- Nell'intestazione del metodo si deve specificare il tipo base dell'array, ma non la lunghezza. Può essere passato al metodo un array di qualunque lunghezza (sempre che il tipo base dell'array sia lo stesso).
- Quando si passa un intero array come argomento ad un metodo, non devono essere utilizzate le parentesi quadre.
- Un metodo può modificare il valore degli elementi di un array passato come argomento.



### 23) Quali sono le problematiche dell'assegnamento e dell'uguaglianza di un array?

- Gli operatori di assegnamento e uguaglianza si comportano con gli array come si comportano con ogni altro oggetto.
- L'intero array è memorizzato in un'unica area di memoria e la posizione può essere specificata con un unico indirizzo di memoria.
- L'operatore di assegnamento = assegna ad una variabile l'indirizzo di memoria in cui è allocato l'array.
- L'operatore di uguaglianza == verifica se due array sono memorizzati nella stessa area di memoria del computer.
- Una variabile di tipo array contiene l'indirizzo di memoria (chiamato riferimento) in cui l'array è memorizzato nello HEAP.
- Le variabili di tipo array sono dette di tipo riferimento (reference type o puntatori). Un tipo riferimento è un qualsiasi tipo le cui variabili contengono indirizzi di memoria al posto dei valori degli elementi.
- Gli array e classi sono tipi riferimento.

**Approfondimento:** I riferimenti sarebbero i puntatori, che sono un tipo di variabile che contengono l'indirizzo di memoria. In alcuni linguaggi di programmazione, come C, C++, i puntatori vengono gestiti dal programmatore, perché considerati linguaggi di basso livello. Java e altri linguaggi di alto livello, i puntatori vengono, invece gestiti dal sistema.

Infatti, in Java, quando si creano gli oggetti (array compresi) questi vengono allocati nell'area di memoria dinamica (HEAP), come dimostrato prima.

La differenza sostanziale tra STACK e HEAP è che lo STACK comporta un'allocazione lineare e sequenziale della memoria statica, mentre lo HEAP funge da pool di aree di archiviazione che allocano la memoria in modo casuale (allocazione dinamica della memoria). La dimensione dello HEAP è pari a tutta la dimensione della memoria RAM libera sommato (grazie allo swapping) allo spazio libero su disco (HDD).

**Come creiamo oggetti nello heap, è possibile cancellarli?**

La garbage collection ("raccolta della spazzatura") è una tecnica che permette di liberare l'area di memoria dinamica. In alcuni linguaggi, come Java o C#, la garbage collection viene effettuata in automatico dal sistema (perché linguaggi di alto livello), mentre in altri (come C, C++) questa viene effettuata dal programmatore con l'istruzione delete.

## 24) Può un metodo ritornare un array?

- Un metodo **Java** può restituire un **array**.
- Si specifica il tipo restituito dal metodo allo stesso modo con cui si specifica un parametro di tipo array.
- Per ritornare l'**array** si utilizza il suo identificatore all'interno dell'istruzione **return**.

### Esempio

```
public static int[] getNumbers() {  
    int[] newArray = {1,2,3,4,5,6};  
    return newArray;  
}
```

## 25) Descrivere le caratteristiche degli array multidimensionali (matrici)

- Gli **array bidimensionali** sono **array** che hanno esattamente due indici e possono essere utilizzati come tabelle bidimensionali.
- Per convinzione si attribuisce il **primo indice alla numerazione delle righe** e il **secondo indice alla numerazione delle colonne**.
- Gli **array** con più indici vengono generalmente chiamati **array multidimensionali**.
- Un **array** con più n indici vengono generalmente chiamati **array n dimensionali**
- Un **array** con un solo indice viene definito **array monodimensionale**.

### Esempio:

1	1	1	0	0	0
1	0	0	0	0	0
0	0	0	0	0	0
0	0	1	0	1	0
0	0	0	0	0	1
0	0	0	0	0	1

### Caratteristiche:

#### Dichiarazione e creazione array bidimensionali

```
int[][] matrice = new int[10][6];  
//oppure  
int matrice[][] = new int[10][6];
```

#### Scansione della matrice

```
for (int i=0;i<10;i++) {  
    for (int j=0;j<6;j++)  
        matrice[i][j] = 0;  
}
```



### Passaggio array multidimensionali come parametri e valori restituiti

Un parametro o un tipo di ritorno di un metodo può essere un array multidimensionale. La sintassi per l'intestazione del metodo è simile al caso in cui i parametri o tipi restituiti siano array monodimensionale, ma è necessario utilizzare più parentesi quadre [].

#### Esempio:

```
public static int getAnElement(char[][] anMatrix, int riga, int colonna);  
public static void readArray(int[][] anMatrix);  
public static char[][] copy(char[][] anMatrix);  
public static int[][] getArray();
```

#### Attenzione!!

Gli array multidimensionali sono rappresentati come più array monodimensionali.

Dato un array bidimensionale

```
int[][] matrice = new int[10][10];
```

La matrice è un array monodimensionale di lunghezza 10 con tipo `int[]`.

Bisogna prestare attenzione all'utilizzo della proprietà `length`:

→ `tabella.length` restituirà quindi 10

→ Mentre per accedere al numero di colonne è necessario scrivere

`tabella[indice_riga].length` (restituisce 6).

#### Array irregolari (ragged arrays)

→ All'interno di un array bidimensionale non è obbligatorio che tutte le righe abbiano la stessa lunghezza (vale anche per array n-dimensionali).

#### Esempio:

```
int[][] b;  
b = new int[3][];  
b[0] = new int[5];  
b[1] = new int[4];  
b[2] = new int[6];
```

## 26) Metodi ricorsivi

→ I **metodi ricorsivi** sono **metodi** che contengono al loro interno chiamate a sé stessi. Per risolvere un problema, il **metodo ricorsivo** scompone i problemi in sotto problemi (**divide et impera**) e richiama sé stesso su di essi per poi, alla fine, mettere insieme i vari risultati e giungere alla soluzione.

→ Ogni **metodo ricorsivo** è riconducibile ad una soluzione iterativa.

→ A volte però una soluzione ricorsiva risulta decisamente più adeguata e per questo motivo che **Wirth** definì quando **algoritmo** risulta essere

**intrinsecamente ricorsivo**: un algoritmo si dice **intrinsecamente ricorsivo** quando la sua **versione iterativa** (sempre esistente) necessita della manipolazione esplicita di uno **stack** di chiamate, operazione che spesso rende l'essenza del programma estremamente difficile da capire.

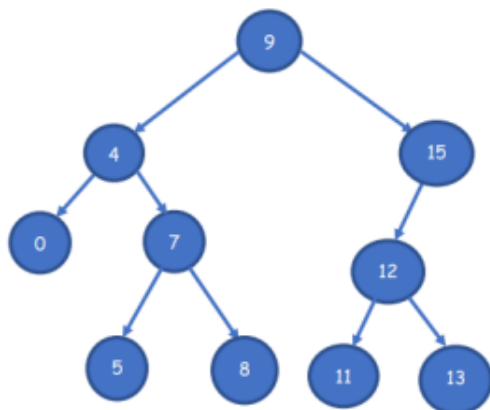
**Esempio 1:**

1) l'**algoritmo di ricerca dicotomica** è un **algoritmo non intrinsecamente ricorsivo**;

2) l'**algoritmo della stampa in order** è un **algoritmo intrinsecamente ricorsivo**;

**Esempio 2:**

Considerato il seguente **albero binario**:



Si ha l'obiettivo di effettuare una **ricerca dicotomica** e una **stampa in order**.

La versione ricorsiva di entrambi gli **algoritmi** è:

→ **ricerca dicotomica**: si confronta l'elemento  $x$  da cercare con la radice: se l'albero è vuoto, la ricerca termina, altrimenti se  $x < \text{radice}$  si effettua un richiamo ricorsivo sul sotto albero sinistro, altrimenti sul sotto albero destro;

→ **stampa in order**: controlla, partendo dalla radice, se esiste su un **sotto albero sinistro**: si chiama ricorsivamente a sinistra, si stampa la radice e, se esiste, si richiama il **sotto albero destro**. Esso stampa tutti gli elementi in ordine crescente e, per questo motivo, che la stampa viene detta **in order**.

La versione iterativa di entrambi gli algoritmi è:

→ **ricerca dicotomica**: ogni discesa lungo un sotto albero l'altro può essere trascurato;

→ **stampa in order**: ogni discesa nel sottoalbero sinistro si deve memorizzare il fatto che il sottoalbero destro non è ancora stato scandito. Occorre quindi utilizzare una struttura **LIFO (Last In First Out)**, ovvero lo **STACK**.

### 27) Quali sono gli elementi base della ricorsione?

→ Per scrivere un metodo ricorsivo bisogna distinguere due casi: il caso base, caso oltre il quale non si va e segna la fine della **ricorsione**, e il caso passo, caso che per essere risolto prevede la riapplicazione nel metodo.

→ Per permettere la scrittura di un **metodo ricorsivo** funzionante e non infinito, oltre a distinguere i due casi, è necessario pensare ad un qualcosa che permette l'applicazione del metodo stesso di agire su un qualcosa di più piccolo, in modo da arrivare ad operare (prima o poi) il **caso base**.

**Esempi di ricorsione**: il fattoriale ( $n!$ ) di un numero:

Codice

```
public static int fattoriale(int n) {  
    if (n==0)  
        return 1;  
    else  
        return n*fattoriale(n-1);  
}
```

**Caso base:**

```
if (n==0)  
    return 1;
```

**Caso passo:**

```
else  
    return n*fattoriale(n-1);
```