

# Introduzione a Java

"line:"# Comandi e Nozioni di Base" line:"#### operazioni matematiche" line:"####  
Q Richiamare un metodo" line:"# Error handling 101" line:"#### Variabili" line:"#### Strings"  
line:"#### Array" line:"#### Control flow" line:"#### Java Method Overloading""

## ▼ Introduzione a Java

9

# Comandi e Nozioni di Base

#### Operazioni matematiche

#### Richiamare un metodo

# Error handling 101

#### Variabili

#### Strings

#### Array

#### Control flow

#### Java Method Overloading

## Comandi e Nozioni di Base

```
public class Main{                                // Inizializza la classe Main
    public static void main(String[] args){ // Inizializza il metodo main, il punto di
partenza
    // Something something                        // del codice
    }
}
```

```
public static void NomeDelMetodo(){ // E' il modo standard per inizializzare un
metodo, in particolare
...                                // solo NomeDelMetodo può essere qualsiasi. Si
dice firma del metodo
...
}
```

```
System.out.println("Hello World!"); // Stampa e "Hello World!" e va a capo
System.out.print("Hello World!"); // Stampa senza andare a capo
```

I blocchi di codice sono separati da graffe {} (basta contare quante se ne aprono)

L'indentazione (la spaziatura delle righe) e' **importante** per questioni di chiarezza e leggibilità.

Le **variabili** vanno *inizializzate* prima di poter essere usate. Per esempio:

```
...  
int IlMioPrimoPreferito = 2;  
...
```

L'operatore = è l'operatore di assegnazione, e *assegna* alla variabile IlMioPrimoPreferito il valore 2.

l'assegnazione **NON** è opzionale, prima di usare una variabile va sempre inizializzata.

E' buona norma dare dei nomi **sensati** alle variabili, Rende più semplice sia la fase di scrittura che la fase di revisione.

Pena: *suicidio professionale*.

Inoltre devono essere **unici** per tutto il codice, indipendentemente dal loro tipo.

I ; vanno inseriti alla fine di **ogni** istruzione, in caso contrario, il programma va in crash.

Il linguaggio Java è *case sensitive*, per esempio *void* **NON** è *VOID*.

---

## Operazioni matematiche

```
public class Main{  
    public static void main(String[] args){  
        int a=5;  
        int b=2;  
        System.out.println(a+b); //Somma  
        System.out.println(a-b); //Differenza  
        System.out.println(a*b); //Prodotto  
        System.out.println(a/b); //Divisione  
        System.out.println(a%b); //Modulo (resto della divisione tra a e b)  
        System.out.println(++a); //Incremento per 1  
        System.out.println(--a); //Decremento per 1  
    }  
}
```

OUTPUT:

```

7 //5 + 2 = 7
3 //5 - 2 = 3
10 //5 * 2 =10
2 //5 / 2 = 2 (il compilatore arrotonda all'intero più piccolo, a meno che a e b
siano di tipo float)
1 //5 % 2 = 1 (Perché 5= 2*2 + 1)
6 // 5 + 1 = 6
4 // 5 - 1 = 4

```

Per tutte le varie operazioni si può aggiungere un "=", producendo per esempio:

```

...
x+=5; //è la stessa cosa di x= x + 5;
...

```

Gli operatori di comparazione sono invece:

1. == uguale a,
  2. != diverso da,
  3. | maggiore di,
  4. < minore di,
  5. | = maggiore o uguale a,
  6. <= minore o uguale a.
- Mentre quelli logici:
7. && AND
  8. || OR
  9. ! NOT

Si usano tra le condizioni per legarle tra loro, ad esempio:

```

...
if ((x<5) && (x>0)){
    //something
    //something
}
...

```

## Richiamare un metodo

```
public class Main{  
    public static void main(String[] args){  
        somma(5,8);  
    }  
    public static void somma(int a, int b){  
        System.out.println(a+b);  
    }  
}
```

OUTPUT:

13

il metodo `somma` accetta due *argomenti* di tipo *intero*. Se al momento della chiamata, i due valori sono:

A) non interi, o

B) non sono due,

avremo un *errore* e il programma non partirà.

`somma(5,8)` e' detta *chiamata* del metodo *somma* con argomenti 5 e 8.

Si può chiamare un metodo dentro un altro.

Inoltre, un metodo potrebbe non accettare alcun argomento come ad esempio un ipotetico metodo `Hello()`:

```
...  
public static void Hello(){  
    System.out.println("No args required!");  
}  
...
```

OUTPUT:

No args required!

Nota:

Un metodo inizializzato *non* va chiamato per forza. Tendenzialmente ogni pezzo di codice deve avere senso di esistere ma mentre si *debuggando* il programma, commentare le parti che non danno problemi è una buona idea.

la keyword `void` indica che il metodo dichiarato *non* ritornerà alcun dato alla fine dell'esecuzione.

In caso non sia presente la keyword `void` è necessario ad un certo punto del

metodo (ragionevolmente la fine) usare la keyword *return* seguita dal un dato dello stesso tipo indicato nella *firma* del metodo. Per esempio:

```
public class Main{  
    public static void main(String[] args){  
        int z= Somma5(3);  
        System.out.println(z);  
    }  
  
    static int Somma5(int x){  
        return x+5;  
    }  
}
```

OUTPUT:

8

---

## Error handling 101

Ci sono principalmente 2 tipi di errore:

1. **Compiletime error** (errore a tempo di compilazione), indica che il programma in sè, contiene errori sintattici che vengono trovati subito.
2. **Runtime error** (errore a tempo di esecuzione), indica che dopo la compilazione, durante l'esecuzione del programma (magari per colpa dell'input di un utente, o per l'interazione con altra roba) il programma riscontra un errore.

In entrambi i casi il programma va in *crash*.

Alternativamente il compilatore potrebbe lamentarsi con un *Warning* dicendo che ci sono metodi, variabili, classi, etc che non sono state usate. Non impedisce il funzionamento del programma, ma una revisione è fortemente suggerita.

---

## Variabili

I principali tipi di variabili sono:

1. String, contiene del testo, come "Hello!";
2. int, contiene dei numeri interi, anche negativi;
3. long, contiene numeri interi fino a 19 cifre

4. float, contiene numeri con la virgola come 19.99f, anche negativi (i float terminano con la f);
  5. double, contiene numeri con la virgola a doppia precisione (i double, opzionalmente, terminano con la d);
  6. char, contiene caratteri singoli, per esempio 'a';
  7. boolean, contiene un valore binario che può essere **solo** vero o falso;
- 

## Strings

Le variabili di tipo String sono trattate come degli oggetti in java.

Se all'interno di una stringa vogliamo usare un carattere speciale va preceduto da un \. Ad esempio:

```
String fattiELogica: "Fatti \"e\" logica";  
System.out.println("fattiELogica");
```

OUTPUT:

```
fatti "e" logica
```

### Metodi comuni delle stringhe

Il metodo `length()` dà la lunghezza della stringa.

Il metodo `toLowerCase()` crea una nuova stringa da quella di partenza, di tutti caratteri minuscoli.

Il metodo `toUpperCase()` crea una nuova stringa da quella di partenza, di tutti caratteri maiuscoli.

il metodo `indexOf("something")` dà la posizione della stringa cercata.

In generale, seguono i vari metodi:

Method	Description	Return Type
<a href="#"><u>charAt()</u></a>	Returns the character at the specified index (position)	char
<a href="#"><u>codePointAt()</u></a>	Returns the Unicode of the character at the specified index	int
<a href="#"><u>codePointBefore()</u></a>	Returns the Unicode of the character before the specified index	int
<a href="#"><u>codePointCount()</u></a>	Returns the number of Unicode values found in a string.	int
<a href="#"><u>compareTo()</u></a>	Compares two strings lexicographically	int
<a href="#"><u>compareToIgnoreCase()</u></a>	Compares two strings lexicographically, ignoring case differences	int
<a href="#"><u>concat()</u></a>	Appends a string to the end of another string	String
<a href="#"><u>contains()</u></a>	Checks whether a string contains a sequence of characters	boolean
<a href="#"><u>contentEquals()</u></a>	Checks whether a string contains the exact same sequence of characters of the specified CharSequence or StringBuffer	boolean
<a href="#"><u>copyValueOf()</u></a>	Returns a String that represents the characters of the character array	String
<a href="#"><u>endsWith()</u></a>	Checks whether a string ends with the specified character(s)	boolean
<a href="#"><u>equals()</u></a>	Compares two strings. Returns true if the strings are equal, and false if not	boolean
<a href="#"><u>equalsIgnoreCase()</u></a>	Compares two strings, ignoring case considerations	boolean
<a href="#"><u>format()</u></a>	Returns a formatted string using the specified locale, format string, and arguments	String
<a href="#"><u>getBytes()</u></a>	Converts a string into an array of bytes	byte[]
<a href="#"><u>getChars()</u></a>	Copies characters from a string to an array of chars	void
<a href="#"><u>hashCode()</u></a>	Returns the hash code of a string	int
<a href="#"><u>indexOf()</u></a>	Returns the position of the first found occurrence of specified characters in a string	int
<a href="#"><u>intern()</u></a>	Returns the canonical representation for the string object	String
<a href="#"><u>isEmpty()</u></a>	Checks whether a string is empty or not	boolean
<a href="#"><u>join()</u></a>	Joins one or more strings with a specified separator	String
<a href="#"><u>lastIndexOf()</u></a>	Returns the position of the last found occurrence of specified characters in a string	int
<a href="#"><u>length()</u></a>	Returns the length of a specified string	int
<a href="#"><u>matches()</u></a>	Searches a string for a match against a regular expression, and returns the matches	boolean
<a href="#"><u>offsetByCodePoints()</u></a>	Returns the index within this String that is offset from the given index by codePointOffset code points	int
<a href="#"><u>regionMatches()</u></a>	Tests if two string regions are equal	boolean
<a href="#"><u>replace()</u></a>	Searches a string for a specified value, and returns a new string where the specified values are replaced	String
<a href="#"><u>replaceAll()</u></a>	Replaces each substring of this string that matches the given regular expression with the given replacement	String
<a href="#"><u>replaceFirst()</u></a>	Replaces the first occurrence of a substring that matches the given regular expression with the given replacement	String
<a href="#"><u>split()</u></a>	Splits a string into an array of substrings	String[]
<a href="#"><u>startsWith()</u></a>	Checks whether a string starts with specified characters	boolean

<code>subSequence()</code>	Returns a new character sequence that is a subsequence of this sequence	CharSequence
<code>substring()</code>	Returns a new string which is the substring of a specified string	String
<code>toCharArray()</code>	Converts this string to a new character array	char[]
<code>toLowerCase()</code>	Converts a string to lower case letters	String
<code>toString()</code>	Returns the value of a String object	String
<code>toUpperCase()</code>	Converts a string to upper case letters	String
<code>trim()</code>	Removes whitespace from both ends of a string	String
<code>valueOf()</code>	Returns the string representation of the specified value	String

## Array

Gli array sono variabili che contengono più valori diversi. Per definire un array si scrive:

```
String[] cars;
```

Per inserirci dei valori (*necessariamente* del tipo specificato) scriviamo:

```
String[] cars {"Volvo","BMW","Ford","Mazda"};
```

Gli array possono essere del tipo di ogni altra variabile, come int, float, double etc.

Per accedere agli elementi di un array si gioca con l'indice in parentesi quadre.

```
...
String[] cars {"Volvo","BMW","Ford","Mazda"};
System.out.println(cars[0]);
...
```

OUTPUT:

Volvo

Nota: gli indici per gli array partono da 0 e arrivano a n-1, dove n è il numero di elementi dell'array.

Possiamo estrarre la lunghezza di un array con la proprietà length.

```
...
String[] cars {"Volvo","BMW","Ford","Mazda"};
System.out.println(cars.length);
```



```
...
```

```
OUTPUT:
```

```
4
```

Un loop specifico per gli array è il *For-Each*:

```
...
```

```
String[] cars {"Volvo","BMW","Ford","Mazda"};
```

```
for (String i: cars){           //letteralmente: per ogni entrata i dell'array cars
    System.out.println(cars[i]); //esegui il che segue. i e le entrate di cars
    devono avere lo stesso
}
```

```
...
```

```
OUTPUT:
```

```
Volvo
```

```
BMW
```

```
Ford
```

```
Mazda
```

Gli array possono anche essere dichiarati vuoti, di una certa lunghezza, con la sintassi:

```
...
```

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

```
...
```

### Array multidimensionali

Questi arrays, o matrici, sono array di array e permettono di gestire dati più vari, tipo tabelle.

```
...
```

```
int [][] numbers= {{1,2,3,4},{5,6,7}};
```

```
...
```

Questo array, essendo bidimensionale si può immaginare come tabella:

Index		0	1	2	3
i	0	1	2	3	4
j	1	5	6	7	

Quindi l'elemento in posizione `[0][1]` è 5, dato che *i* indica l'indice di riga e *j* quello di colonna.

si possono inserire tutti i sotto array che uno vuole, basta aggiungere altre parentesi tonde all'interno delle graffe.

Nota: si mettono sempre 2 parentesi quadre `[] []` per indicare un array bidimensionale qualsiasi.

#### Side note:

Anche gli array multidimensionali più complicati si possono visualizzare graficamente, con la speranza di non perdere completamente il senno, faccio l'esempio con un 3-array:

```
int [][][] numbers= {  
    {  
        {1,2},  
        {3,4}}  
    },  
    {  
        {5,6},  
        {7,8}}  
};
```

In questo caso gli indici sono, ad esempio, *i*, *j*, *k*.

Per accedere gli elementi bisogna specificare la posizione in entrambe le entrate:

```
...  
System.out.println(numbers[0][0][1]);  
...  
OUTPUT:  
2
```

Perché:

TABELLA *i*=0

Index		0	1
j	0	1	2
k	1	3	4

TABELLA *i*=1

Index		0	1
j	0	5	6
k	1	7	8

---

## Control flow

Il metodo per verificare una condizione binaria è l'utilizzo degli *if* ed *else* statement. La sintassi è come segue:

```
if (la tua condizione preferita) {  
    //Allora succede tutto quello scritto qui  
}  
else { //altrimenti  
    //tutto quello scritto qua  
}
```

Ad esempio:

```
...  
x=6;  
  
if (x>0){  
    System.out.println("x e' positivo");  
}  
else {  
    System.out.println("x e' negativo");  
}  
....  
  
OUTPUT:  
x e' potitivo
```

Nota: gli else sono opzionali.

Nel caso si vogliano esplorare più possibili scelte si usa *else if* statement. Riprendendo il caso di prima:

```
x=6;  
  
if (x>0){
```

```

    System.out.println("x e' positivo");
}
else if(x==0) {
    System.out.println("x e' 0");
}
else {
    System.out.println("x e' negativo");
}
....

OUTPUT:
x e' potitivo

```

Lo *switch* statement permette di scegliere tra tanti casi diversi senza usare tanti if/else statements.

```

...
switch(d){ //Dove "d" impone la condizione
    case 1:
        System.out.println("uno");
        break;
    case 2:
        //and so on
        break;
    case 3:
        // and so forth
        break;
    default: //se viene inserito qualcosa di non specificato, viene invocato il caso
default
        System.out.println("Dato non valido");
}
...

```

Ci sono anche i così detti cicli, o loop. Questi permettono di eseguire più volte uno stesso blocco di codice. Tra di loro ci sono i:

1. While loop
2. Do While loop
3. For loop

Nel While loop la sintassi è:

```

...
int = 0;
while (i<5){
//Blocco di istruzioni
System.out.println(i);

++i; //incremento

//è FONDAMENTALE avere un istruzione che eventualmente permetterà al loop di
terminare. In caso non ci fosse, il blocco di codice verrà ripetuto all'infinito.
}
System.out.println("il ciclo è terminato");
...

OUTPUT:
0
1
2
3
4
il ciclo è terminato

```

Nel Do While loop la sintassi è:

```

...
int = 0;
do{
//Blocco di istruzioni
System.out.println(i);
++i; //incremento

} while (i<5);
System.out.println("il ciclo è terminato");
...

OUTPUT:
0
1
2
3
4
il ciclo è terminato

```

La differenza principale è che nel Do While il blocco di codice interno viene eseguito sempre *ALMENO* una volta.

Il For loop è un ciclo che viene usato quando sappiamo quante volte un blocco di codice fa iterato. La sintassi è:

```
...
for(int i=0; i<5, i++){ //Questi 3 statement sono rispettivamente, inizializzazione,
controllo condizione e
//Incremento. L'incremento avviene solo all'inizio del primo
ciclo, la condizione
//funziona come quella di un while. E l'incremento avviene
alla fine di ogni
//iterazione
System.out.println(i);
}
...
```

OUTPUT:

```
0
1
2
3
4
```

## OOP (Object Oriented Programming)

In java, tutti i vari costrutti etc, sono *metodi*. I metodi sono blocchi di codice che partono solo quando chiamati, posso accettare argomenti o parametri.

I metodi devono essere dichiarati all'interno di una classe, la sintassi, per esempio è:

```
public class Main(){
    static void myMethod(){
        // il codice del metodo
    }
}
```

"myMethod" è il nome del metodo.

static e public si chiamano modificatori di accesso.

void indica che il metodo non ha un valore di ritorno.

Ad esempio il codice:

```

public class Main(){
    static void myMethod(){
        System.out.println("I just got executed! \n Now get off my lawn");
    }
    public static void main(String[] args){
        myMethod();
    }
}

```

OUTPUT:

```

I just got executed!
Now get off my lawn

```

Possiamo inserire all'interno di un metodo un qualunque numero di parametri separati da virgole.

```

public class Main{
    static void myMethod(String fname) //parametro del metodo
    System.out.println(fname + "Refsnes");
}

```

```

public static void main(String[] args){
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anya");
}

```

OUTPUT:

```

Liam Refsnes
Jenny Refsnes
Anya Refsnes

```

L'ordine e il tipo dei parametri è importante. Parametri in numero, tipo e ordine diverso da ciò che è dichiarato nella *firma* del metodo risulterà in un errore.

Nota: Quando un parametro viene passato ad un metodo, viene chiamato *argomento*.

Un metodo ha bisogno di avere nella firma o void o un tipo di variabile. Nel secondo caso, il metodo deve contenere l'istruzione **return**.

```

public class Main{
    static int myMethod(int x) //parametro del metodo
    return x+5;
}

```

```
}
```

```
public static void main(String[] args){  
    System.out.println(myMethod(3));  
}
```

OUTPUT:

```
8
```

Si possono avere un qualunque numero di return in un metodo che ritorna un valore, a patto che siano in flussi diversi, ovvero non esiste il caso in cui ne viene eseguito più di uno.

Ma comunque, qualunque sia il flusso considerato un return **deve** essere raggiunto.

## Java Method Overloading

con il method overloading, più metodi possono avere lo stesso nome ma parametri diversi:

```
int myMethod(int x)  
float myMethod(float x)  
double myMethod(double x, double y)
```

non darà errore perché anche se i metodi hanno lo stesso nome, la loro firma è diversa.

Il compilatore deciderà automaticamente quale metodo utilizzare in base al tipo di parametro fornito.