

# LINGUAGGI

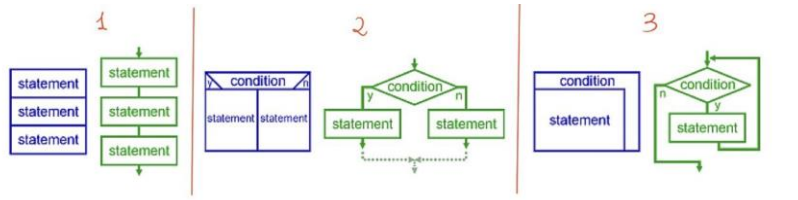
## - Introduzione

L'unico meccanismo per gestire il flusso decisionale di un algoritmo era il GOTO, cioè un meccanismo che permette di passare da un blocco di codice ad un altro.

### ▪ Programmazione Strutturata

Un programma strutturato è composto di tre combinazioni principali, applicate a blocchi primitivi di operazioni (statement):

- 1) **Sequenza**: una esecuzione ordinata di statement
- 2) **Selezione**: a seconda dello 'stato' del programma, uno statement viene 'selezionato' da un insieme di statement per essere eseguito.
- 3) **Ciclo**, o **iterazione**: uno statement è eseguito fino a che il programma non raggiunge un determinato stato.



I seguenti requisiti devono essere rispettati da un linguaggio di programmazione strutturato:

- **Completezza**: le tre strutture della programmazione strutturata devono avere almeno una rappresentanza sintattica nel linguaggio.
- **Singolo punto di Ingresso e Uscita**: in ogni struttura di controllo si devono poter identificare un singolo punto di ingresso e un singolo punto di uscita. Ciò è necessario per la successiva:
- **Componibilità**: ogni struttura di controllo deve poter essere considerata come un macro-statement, di modo da poter essere usata, ricorsivamente, come istruzione in altre strutture di controllo.

### ▪ Programmazione Procedurale

Consiste nella possibilità di agevolare la leggibilità di un programma tramite la definizione di blocchi di codice, racchiusi da delimitatori e identificati da un nome.

Tali blocchi prendono il nome di:

- Subroutine
- Procedure
- Funzioni
- Metodi

- Programmazione Orientata agli Oggetti

Esistono 4 principi fondamentali:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

# | JAVA

- **Come nomino un file Java?**

Per **NOMINARE** un file Java bisogna avere:

- Il nome del file deve corrispondere al nome della classe
- Dopo il nome mettiamo *.java*

Quindi in breve un file Java viene nominato come **nomeclasse.java**

- **Come eseguire un codice Java?**

Un file java non può essere eseguito direttamente da un computer. Per

**ESEGUIRE** un file Java bisogna usare un compilatore chiamato compilatore *javac* che trasforma il nostro codice in un tipo di file chiamato *.class*

Questo file non contiene un file scritto in linguaggio macchina ma contiene bytecodes, cioè il linguaggio macchina della Java Virtual Machine ( Java VM ).

Una volta compilato il file lo eseguiamo con il launcher Java.

- **Che cos'è la programmazione ad oggetti?**

In Java useremo la programmazione ad oggetti, ciò significa che andremo a ricreare delle entità che esistono nella vita reale in programmazione.

- **Che cos'è un oggetto?**

Gli **OGGETTI** di Java si comportano in modo simile agli oggetti che ci circondano nella realtà. Infatti in Java contengono i propri stati in *campi* e indicano i loro comportamenti tramite i *metodi*. I metodi vanno a modificare lo stato interno dell'oggetto e sono il meccanismo che consente di comunicare con un oggetto.

- **Cos'è una classe?**

Definisco la **CLASSE** come un modello da cui vengono creati i singoli oggetti.

In una classe troviamo:

- I campi: gli attributi o caratteristiche dell'oggetto
- I metodi: i comportamenti che fa l'oggetto (N.B: in ogni metodo posso trovare un altro metodo)

Ogni classe si indica con la lettera maiuscola.

ES: *Myclass* può essere una classe

*myclass* non è una classe

▪ **Quali sono i 4 principi fondamentali della programmazione ad oggetti?**

Questa tipologia di programmazione è caratterizzata da 4 principi fondamentali:

- **Ereditarietà:** permette ad una nuova classe di ereditare gli attributi della classe precedentemente creata

- ES: nel codice troveremo la seguente dicitura

```
public class Impiegato extends Person{  
  
}
```

In una classe ereditata da un'altra classe, quando assegno i valori dei parametri del costruttore agli attributi, userò SUPER che mi permette di accedere ai membri della super classe dall'interno di una sottoclasse e di "ereditarli".

```
public Impiegato(String Nome,String Cognome,String  
Codice_Fiscale){  
  
Super (Nome, Cognome);  
  
this.Codice_Fiscale = Codice_Fiscale;  
  
}
```

- **Incapsulamento:** rende protetti i dati all'interno di una classe privata, diventano accessibili solo attraverso i metodi GETTER e SETTER

- ES: nel codice troveremo la seguente dicitura

```
private String Nome;  
private String Cognome;  
  
//assegno i valori agli attributi  
  
public Persona(String Nome, String Cognome){  
  
    this.Nome = Nome;  
  
    this.Cognome = Cognome;  
  
}
```

```
//leggo un valore con get

    public String getNome(){
        return Nome;
    }

    public String getCognome(){
        return Cognome;
    }

//assegno un valore con set

    public void setNome(String Nome){
        this.Nome = Nome;
    }

    public void setCognome(String Cognome){
        this.Cognome = Cognome;
    }
}
```

- **Polimorfismo:** capacità di un oggetto di identificarsi con più tipi di dato, cioè consente ad un oggetto di assumere più forme

```
public class Animale {

    public void faiVerso() {

        System.out.println("L'animale fa rumore");

    }

}

public class Cane extends Animale {

    @Override
    public void faiVerso() {

        System.out.println("Il cane abbaia.");

    }

}
```

```

public class Gatto extends Animale {

    @Override
    public void faiVerso() {

        System.out.println("Il gatto miagola.");
    }

}

public class Main {

    public static void main(String[] args) {

        Animale animale1 = new Cane();
        Animale animale2 = new Gatto();

        animale1.faiVerso();
        animale2.faiVerso();

    }

}

```

- **Astrazione:** permette di modellare concetti con oggetti e interfacce semplici

```

public abstract class Veicolo {

    public abstract void muoviti();

}

public class Auto extends Veicolo {

    @Override
    public void muoviti() {

        System.out.println("L'auto si muove su strada.");
    }

}

public class Main {

```

```

        public static void main(String[] args) {

            Veicolo auto = new Auto();

            auto.muoviti();

        }
    }
}

```

- **Cos'è un costruttore di una classe?**

I costruttori sono metodi speciali chiamati automaticamente quando si crea un oggetto di una classe.

- **Cos'è un'interfaccia?**

Un'interfaccia è un gruppo di metodi correlati con i corpi vuoti. Per accedere ai metodi dell'interfaccia, questa deve essere "implementata" da un'altra classe con ***implements*** parola chiave.

```

// Interface
interface Animal {
    public void animalSound(); // interface method (does not have a body)
    public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
    public void sleep() {
        // The body of sleep() is provided here
        System.out.println("Zzz");
    }
}

class Main {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}

```

L'implementazione di un'interfaccia consente ad una classe di diventare più formale.

## ▪ Cos'è un pacchetto?

Un pacchetto è uno spazio dei nomi che organizza un insieme di classi e interfacce correlate. Java ci fornisce un'enorme libreria di classi conosciuta come API (Application Programming Interface).

## ▪ Altre Nozioni

Le parentesi graffe { } indicano l'inizio e la fine di un blocco di codice.

Alla fine di ogni riga metto ; (punto e virgola).

N.B: Se io ho un'**Espressione**, cioè un costrutto costituito da variabili operatori e invocazioni di metodi, e a fine di questa inserisco il punto e virgola questa diventerà un'**Istruzione**. Invece, se ho un gruppo di zero o più istruzioni tra parentesi graffe avrò un **Blocco**.

System è una classe Java incorporata che contiene membri utili, come out, che è l'abbreviazione di "output".

Il metodo **main ( )** lo troviamo sempre in un programma Java.

## ▪ Java Output

Per generare valori o **stampare** testo in Java usiamo:

- Il metodo **.println ( )** abbreviazione di "print line", viene utilizzato per stampare un valore e ci permette di andare a capo (va alla linea dopo).
- Anziché scrivere **.println ( )** possiamo scrivere **.print ( "quello che vogliamo stampare/n" )**
- Il metodo **.print ( )** ci permette solo di stampare

N.B: quando scrivi un testo lo devi racchiudere tra virgolette doppie " "

```
System.out.println("This sentence will work!");
```

Quando scriviamo dei numeri non serve che si usano le virgolette

```
System.out.println(3);  
System.out.println(358);  
System.out.println(50000);
```

Puoi anche eseguire calcoli matematici all'interno di **.println( )**



## ▪ Commenti

Se voglio inserire un **commento** posso avere due casi a seconda di che commento voglio fare:

- commento in una sola linea precedo questo da `//`
- commento in più linee compreso tra `/* (commento) */`

```
System.out.println("Hello World"); // This is a comment
```

```
/* The code below will print the words Hello World  
to the screen, and it is amazing */  
System.out.println("Hello World");
```

## ▪ Variabili

In Java esistono i seguenti tipi di variabili:

- *String* (stringa)
- *int* (numero intero)
- *float* (numero con la virgola)
- *boolean* ( True, False )
- *char* (singolo carattere)

```
int myNum = 5;  
float myFloatNum = 5.99f;  
char myLetter = 'D';  
boolean myBool = true;  
String myText = "Hello";
```

Per creare una variabile serve specificare un tipo ***type vaariablename = value***

ES: *string* name = "Giorgia"

Una volta che specifico il tipo della variabile in una linea del programma non serve che lo specifico sempre.

ES: *string* name  
name = "Giorgia"

Se assegno un nuovo valore a una variabile esistente, sovrascriverà il valore precedente.

Se inserisco *final* ad una variabile il suo valore non può essere cambiato.

All'interno di `println ( )` posso inserire sia un testo che una variabile utilizzando il `+`

```
String name = "John";  
System.out.println("Hello " + name);
```

Possiamo usare il `+` per aggiungere una variabile alla stessa variabile

```
String firstName = "John ";  
String lastName = "Doe";  
String fullName = firstName + lastName;  
System.out.println(fullName);
```

Per i valori numerici, il `+` funziona come un operatore matematico

```
int x = 5;  
int y = 6;  
System.out.println(x + y); // Print the value of x + y
```

Per dichiarare più di una variabile dello **stesso tipo**, puoi utilizzare un elenco separato da virgole

```
int x = 5, y = 6, z = 50;  
System.out.println(x + y + z);
```

Puoi anche assegnare lo stesso valore a più variabili in una riga

```
int x, y, z;  
x = y = z = 50;  
System.out.println(x + y + z);
```

Per la nomenclatura delle variabili possiamo:

- usare nomi significativi, cioè il nome della variabile dovrebbe riflettere ciò che rappresenta

ES: `int etàGiorgia = 20` (così sappiamo che quella variabile indica l'età di Giorgia)

- usare la convenzione CamelCase, cioè il nome con una lettera minuscola e usa maiuscole per ogni parola successiva.

ES: `int numeroStudenti;`

- evitare nomi troppo generici, cioè evita di chiamare le variabili con x, y, z, ecc..
- scrivi tutte le variabili nello stesso modo

## ▪ Tipi di dato

Una variabile in Java deve essere un tipo di dato specificato, come ad esempio:

I tipi di dati sono divisi in 2 gruppi:

- **Tipi di dati primitivi** → byte, short, int, long, float, double, Boolean, char

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

- **Tipi di dati non primitivi** → String (hanno una serie di attributi che possiamo usare - UpperCase), Array e Classi...e altro

Le differenze tra i due sono:

- I tipi primitivi sono predefiniti in Java. I tipi non primitivi vengono creati dal programmatore e non sono definiti da Java (ad eccezione di String).
- I tipi non primitivi possono essere utilizzati per chiamare metodi per eseguire determinate operazioni, mentre i tipi primitivi no.

- Un tipo primitivo ha sempre un valore, mentre i tipi non primitivi possono essere null.
- Un tipo primitivo inizia con una lettera minuscola, mentre i tipi non primitivi iniziano con una lettera maiuscola.

#### ▪ Wrapper class

Le classi Wrapper forniscono un modo per utilizzare i tipi di dati primitivi (*int*, *boolean*, *ecc.*) come oggetti.

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

```

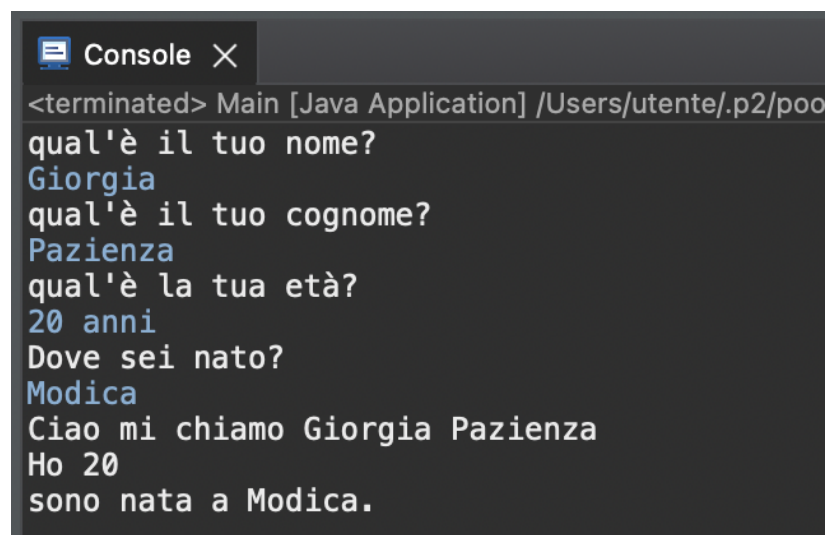
1
2 public class Main {
3     public static void main(String[] args) {
4
5         Boolean vero = true;
6         Character carattere = 'a';
7         Double virgola = 2.5;
8         Integer numero = 2;
9         String stringa = "io sono Giorgia";
10
11         boolean a = true;
12         char b = 'a';
13         double c = 2.5;
14         int d = 2;
15         String e = "io sono Giorgia";
16
17         if (vero == a) {
18             System.out.println ("corretto");
19         }
20     }
21 }
22
23
24 }
```

- **Input dell'utente del terminale**

Per ottenere input dall'utente possiamo fare uso della classe **Scanner**. Questa appartiene al pacchetto **java.util**. Questa classe consente di leggere in input della console

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         Scanner scanner = new Scanner (System.in);
7
8         System.out.println("qual'è il tuo nome? ");
9         String nome = scanner.nextLine();
10
11        System.out.println("qual'è il tuo cognome? ");
12        String cognome = scanner.nextLine();
13
14        System.out.println("qual'è la tua età? ");
15        int eta = scanner.nextInt();
16        scanner.nextLine();
17
18        System.out.println ("Dove sei nato? ");
19        String luogo = scanner.nextLine();
20
21        System.out.println ("Ciao mi chiamo " + nome + " " + cognome);
22        System.out.println("Ho " + eta);
23        System.out.println("sono nata a " + luogo + ".");
24
25
26    }
27 }
```

La console del seguente codice è la seguente:



```
<terminated> Main [Java Application] /Users/utente/.p2/poo
qual'è il tuo nome?
Giorgia
qual'è il tuo cognome?
Pazienza
qual'è la tua età?
20 anni
Dove sei nato?
Modica
Ciao mi chiamo Giorgia Pazienza
Ho 20
sono nata a Modica.
```

- **Array**

Gli array sono una raccolta ordinata di elementi dello stesso tipo. Vengono utilizzati per memorizzare più valori in una singola variabile, invece di dichiarare variabili separate per ciascun valore. Per dichiarare un array, definiamo il tipo di variabile con parentesi quadre.

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

```
int[] myNum = {10, 20, 30, 40};
```

(Il tipo può essere preceduto da *new type [array] = new int [10]* )

È possibile accedere a un elemento dell'array facendo riferimento al numero di indice.

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

N.B: gli indici degli array iniziano con 0 (che è infatti il primo elemento)...poi con 1 e così via.

Per modificare un elemento facciamo la seguente cosa

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
// Now outputs Opel instead of Volvo
```

Per scoprire la lunghezza usiamo **length**

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);
// Outputs 4
```

Possiamo scorrere gli elementi dell'array con il ciclo for e con length per specificare quante volte deve essere eseguito il ciclo. Questo viene chiamato iterazione.

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
    System.out.println(cars[i]);
}
```

Un array può essere multidimensionale, ciò significa che ogni elemento dell'array è a sua volta un array. Per accedere agli elementi di questo array avremo due indici, uno per l'array e uno per l'elemento all'interno di tale array.

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
System.out.println(myNumbers[1][2]); // Outputs 7
```

Possiamo iterare anche gli elementi di un array multidimensionale (sempre con for e length come prima).

```
for (int riga = 0; riga < matrice.length; riga++) {
    for (int colonna = 0; colonna < matrice[riga].length; colonna++) {
        System.out.print(matrice[riga][colonna] + " ");
    }
    System.out.println();
}
```

## ▪ Operatori

In Java abbiamo i seguenti operatori:

### - *Aritmetici*

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

### - *Unari*

Operator	Description
+	Unary plus operator; indicates positive value (numbers are positive without this, however)
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

### - *Uguaglianza e relazionali*

```

==      equal to
!=      not equal to
>       greater than
>=      greater than or equal to
<       less than
<=      less than or equal to

```

### - *Logical operators*

Operator	Name	Description
&&	Logical and	Returns true if both statements are true
	Logical or	Returns true if one of the statements is true
!	Logical not	Reverse the result, returns false if the result is true



- *Bitwise operators (Operatori bit a bit)*

Gli operatori bit a bit lavorano sui dati interi a livello di singolo bit. Infatti sono applicabili ai tipi di dati byte, short, int e long. Questi operatori sono:

- **OR Bitwise ( | )**: Restituisce 1 se almeno uno dei due operandi è 1, altrimenti restituisce 0.
- **AND Bitwise ( & )**: Restituisce 1 solo se entrambi gli operandi sono 1, altrimenti restituisce 0.
- **XOR Bitwise ( ^ )**: Restituisce 1 solo se uno dei due operandi è 1, ma non entrambi. Altrimenti restituisce 0.

- *Shift operators*

Gli operatori di shift spostano i bit di un valore a sinistra o a destra. Esistono due tipi di operatori di shift:

- **Shift a Sinistra (<<)**: Sposta i bit a sinistra di un certo numero di posizioni.
- **Shift a Destra (>>)**: Sposta i bit a destra di un certo numero di posizioni. Il bit più a sinistra viene replicato per mantenere il segno.

▪ **Math**

Il modulo Math ci permette di avere varie funzioni matematiche avanzate che consentono di eseguire calcoli complessi e precisi. Per usare però devo importare questo modulo nel seguente modo:

```
import java.lang.Math;
```

Ecco le seguenti funzioni:

```
double radiceQuadrata = Math.sqrt(25);
double potenza = Math.pow(2, 3);
int valoreAssoluto = Math.abs(-10);
double arrotondamentoSuperiore = Math.ceil(7.3);
double arrotondamentoInferiore = Math.floor(7.8);
long arrotondamentoVicino = Math.round(7.5);
int valoreMassimo = Math.max(5, 9);
int valoreMinimo = Math.min(3, 7);
double numeroCasuale = Math.random();
```

$\sin(x)$  ,  $\cos(x)$  ,  $\tan(x)$  : Funzioni trigonometriche.

$\arcsin(x)$  ,  $\arccos(x)$  ,  $\arctan(x)$  : Funzioni inverse trigonometriche.

$\log(x)$  : Logaritmo naturale (base e).

$\log_{10}(x)$  : Logaritmo in base 10.

- **If e Else**

**If** dice al programma di eseguire un determinato blocco di codice solo se una condizione specificata è vera

```
if (20 > 18) {  
    System.out.println("20 is greater than 18");  
}
```

**Else** si usa quando la condizione if precedente restituisce false

```
int time = 20;  
if (time < 18) {  
    System.out.println("Good day.");  
} else {  
    System.out.println("Good evening.");  
}  
// Outputs "Good evening."
```

- **Switch**

**Switch** si usa per specificare molti blocchi di codice alternativi da eseguire

```
int day = 4;  
switch (day) {  
    case 1:  
        System.out.println("Monday");  
        break;  
    case 2:  
        System.out.println("Tuesday");  
        break;  
    case 3:  
        System.out.println("Wednesday");  
        break;  
    case 4:  
        System.out.println("Thursday");  
        break;  
    case 5:  
        System.out.println("Friday");  
        break;  
    case 6:  
        System.out.println("Saturday");  
        break;  
    case 7:  
        System.out.println("Sunday");  
        break;  
}  
// Outputs "Thursday" (day 4)
```

- **Ciclo While e Do-while**

Il ciclo **while** scorre un blocco di codice purché una condizione specificata sia true

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

Il ciclo **do/while** è una variante del ciclo while. Questo ciclo eseguirà il blocco di codice una volta, prima di verificare se la condizione è vera, quindi ripeterà il ciclo finché la condizione è vera.

```
int i = 0;
do {
    System.out.println(i);
    i++;
}
while (i < 5);
```

- **Ciclo For**

Il ciclo **for** si usa quando sai già quante volte vuoi eseguire un blocco di codice.

L'istruzione 1 viene eseguita (una volta) prima dell'esecuzione del blocco di codice.

L'istruzione 2 definisce la condizione per l'esecuzione del blocco di codice.

L'istruzione 3 viene eseguita (ogni volta) dopo l'esecuzione del blocco di codice.

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

Un ciclo for può essere anche all'interno di un altro ciclo for e viene detto **Ciclo Annidato**.

- **Break**

Il **break** ha due forme:

- etichettata → esce dal blocco switch
- Non etichettata → uscire da un ciclo for, while o do-while

```
int i = 0;
while (i < 10) {
    System.out.println(i);
    i++;
    if (i == 4) {
        break;
    }
}
```

- **Continue**

**Continue** interrompe un'iterazione in un ciclo (for/while/do-while) e continua con l'iterazione successiva nel ciclo.

```
int i = 0;
while (i < 10) {
    if (i == 4) {
        i++;
        continue;
    }
    System.out.println(i);
    i++;
}
```

- **Return**

**Return** finisce l'esecuzione di un metodo e può ritornare un valore che deve essere compatibile con il tipo del metodo.

Un metodo che ritorna un valore:

```
public class Main {
    static int myMethod(int x) {
        return 5 + x;
    }

    public static void main(String[] args) {
        System.out.println(myMethod(3));
    }
}
// Outputs 8 (5 + 3)
```

Un metodo che NON ritorna un valore (ma stampa):

```
public class Main {  
    static void myMethod() {  
        System.out.println("I just got executed!");  
    }  
  
    public static void main(String[] args) {  
        myMethod();  
    }  
}
```