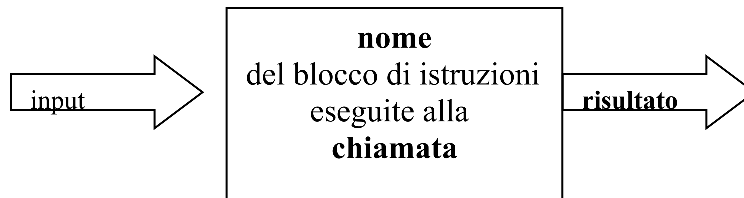
FUNZIONI

Ogni funzione è un insieme di istruzioni (**corpo**) che implementa un blocco di comandi visto come una "scatola nera" capace, dunque, di realizzare una certa elaborazione fornendo sempre un risultato (*output*) potendo ricevere come *input* i dati da elaborare.



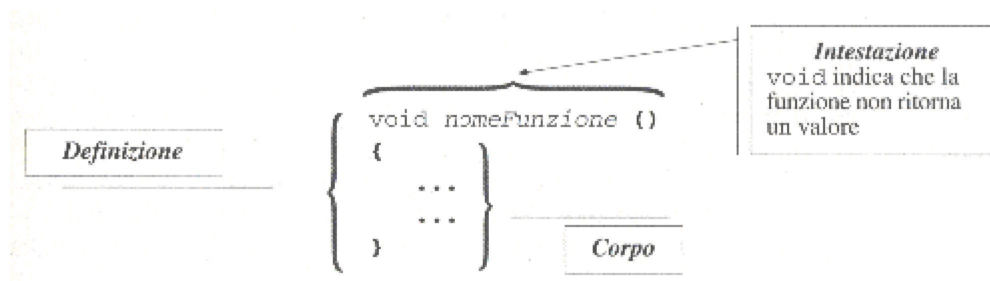
Tale "scatola" non può esser inserita in un'altra "scatola" ma può essere usata trattando il suo **nome** come una variabile.

Tale *modulo* di programma è dunque caratterizzato da un **nome** seguito da ()

All'interno delle *parentesi tonde* possono essere indicati tipo e nome dei parametri formali nel caso che la funzione riceva in ingresso dei valori da elaborare, che gli vengono passati *per copia*[1] dalla funzione *chiamante*.

Come per ogni variabile, anche per una funzione è necessario definirne il **tipo**: si deve informare il compilatore affinché riservi l'area di memoria necessaria a contenere il risultato della funzione.

Se la funzione *chiamata* non ***ritorna*** un valore alla funzione *chiamante*, in realtà il risultato risulta indefinito ed è di tipo *void*



La sintassi per la **definizione** di una funzione è pertanto:

```
tipoFunzione nomeFunzione (tipoVariabilePassata nome, ……)
                      {
                          tipo risultato ; // dichiarazione se non void

                          // dichiarazioni di variabili locali
                          // istruzioni (almeno una)

                          return (risultato);  // se non void
                       }
```

[1] In realtà, alla chiamata, gli argomenti passati dalla funzione chiamante sono copiati nell'area di **stack** (una porzione di RAM dove i dati sono salvati come in una catasta: l'ultimo inserito è il primo estratto LIFO) e poi ricopiati nel codice della funzione chiamata (si realizza quindi un flusso di "copie" personali)

Se il nome della funzione è scritto all'interno di un generico segmento di programma, provoca, durante il *runtime*, l'esecuzione di quell'insieme di istruzioni presenti nell'area di RAM dove è memorizzata la funzione stessa e, terminata l'esecuzione sequenziale di tale sottoprogramma, il controllo è restituito al programma **chiamante** che proseguirà, eseguendo l'istruzione successiva alla **chiamata** di funzione che ne aveva interrotto il flusso.

L'istruzione di **chiamata di funzione** ha la seguente sintassi:

```
nomeFunzione();
```

All'interno delle *parentesi tonde* si può elencare la lista degli **argomenti attuali** i cui valori saranno assegnati ai corrispondenti parametri formali della funzione **chiamata** al *runtime*.

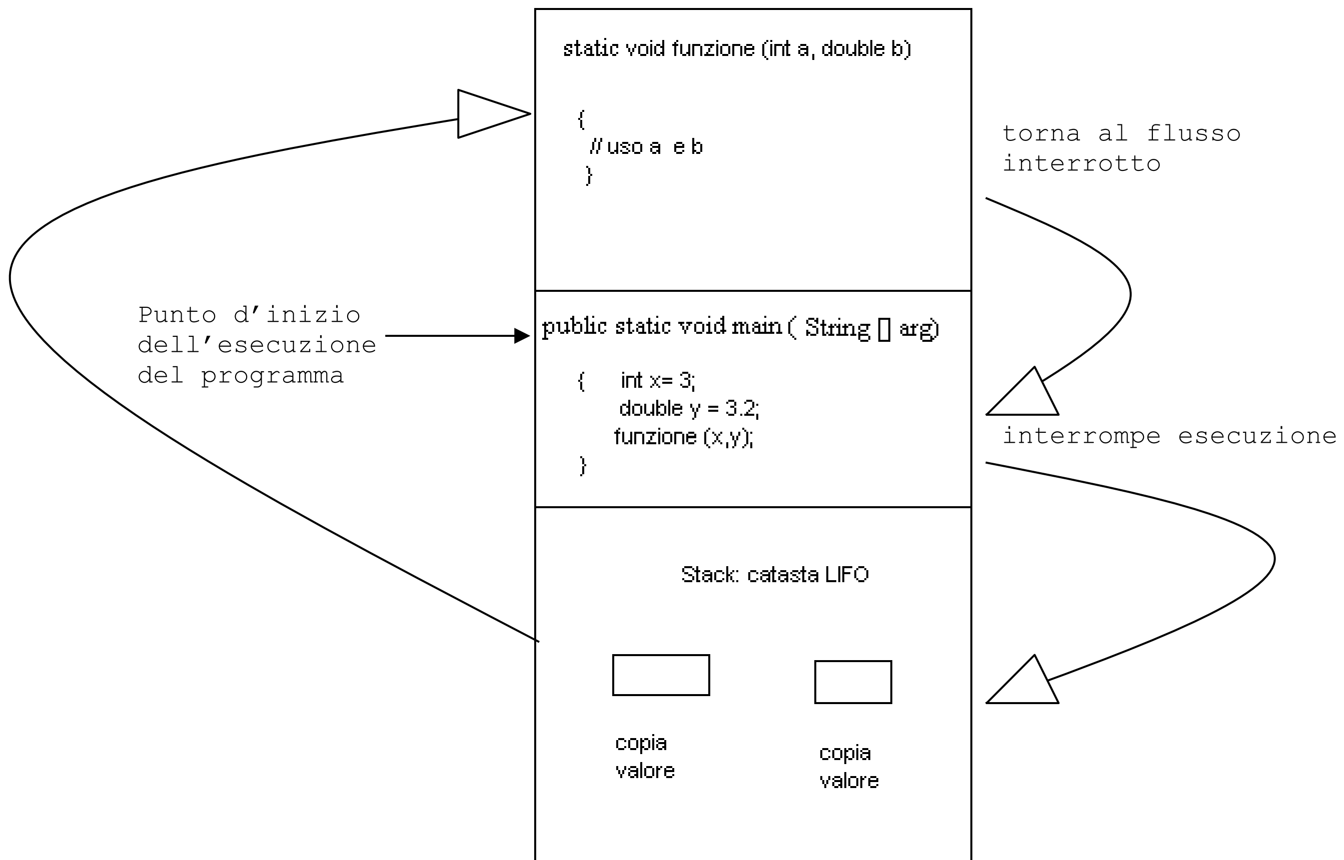
Il passaggio è ***posizionale***

il primo valore è assegnato al primo parametro etc....

```
...
int massimo (int a, int b)
{
   ...
}
...
int max = massimo (-10,2);
...
```

Argomenti attuali

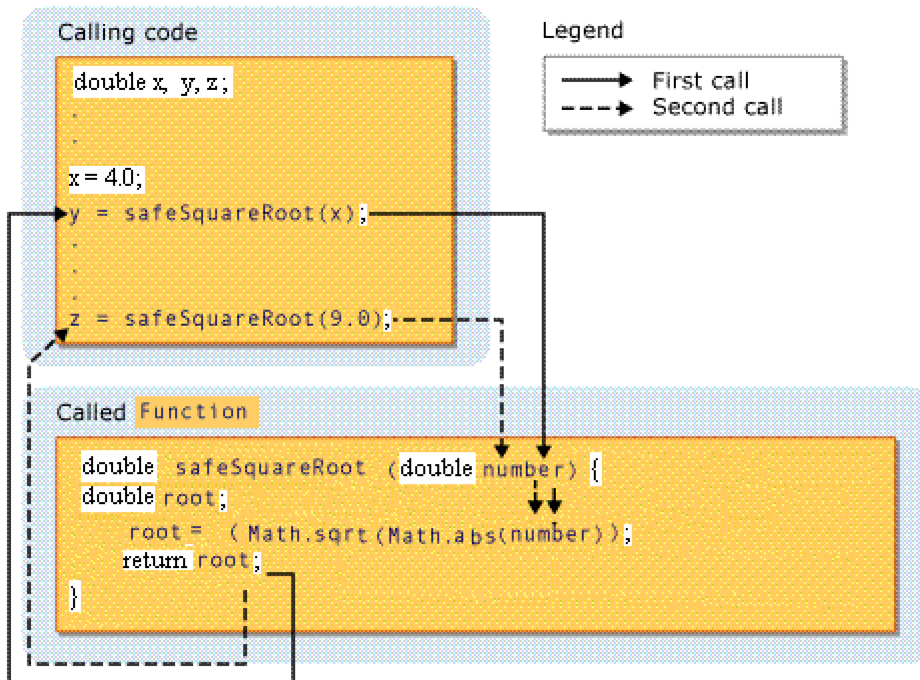
Con l'istruzione **return** si potrà **ritornare** il **valore** del risultato che si sostituirà al nome della funzione **chiamata** per essere utilizzato nel generico segmento di programma **chiamante**.



In figura è illustrato il caso di funzione che non ritorna un valore, riceve in ingresso due valori ed al suo interno referenzia il primo valore intero (il numero 3) con a ed il secondo valore double (numero 3.2) con b.
Nb: in Java con **static** si intende che la funzione può essere *chiamata direttamente* (di default è **public** cioè *visibile a tutti*)e la sintassi completa è la seguente:

<**modificatori**> tipoDiRitornoDelMetodo nomeDelMetodo(<parametr>) { corpoDelMetodo }

Nella figura seguente è illustrato il caso di funzione che ritorna un valore reale a doppia precisione:



Il codice della funzione è scritto un'unica volta ma è eseguibile, in teoria, un numero illimitato di volte, utilizzabile da altri programmi o, in modo **ricorsivo**, dalla funzione stessa.

Una funzione è un modo di *arricchire il linguaggio*.

Le funzioni possono essere raggruppate in librerie e ***riutilizzate*** da altri programmi; rendono un programma più immune da errori imprevisti grazie alla possibilità di *nascondere* i dati

prevedendo meccanismi di passaggio controllato delle informazioni;

consentono una scrittura più semplice e aumentano la *leggibilità* del programma: una funzione, infatti, serve per *ridurre la complessità* di un problema consentendo facile implementazione di soluzioni con approccio top-down che prevedono un lavoro parallelo in equipe.

Progettare bene una funzione significa pensarla con le seguenti caratteristiche: efficienza, generalizzazione e portabilità.