

2

I metodi statici

IN QUESTA LEZIONE IMPAREREMO A...

- ➔ Definire un metodo statico
- ➔ Definire la modalità del passaggio dei parametri
- ➔ Utilizzare metodi predefiniti nei programmi

Linguaggi del paradigma a oggetti

Abbiamo già detto che **Java** è un linguaggio classificabile nel paradigma dei **linguaggi di programmazione a oggetti** (**OOP**, **Object-Oriented Programming**), che consiste nel definire “cose”, chiamate “oggetti”, in grado di interagire tra loro attraverso lo **scambio di messaggi**.

I vantaggi di questo paradigma si possono sintetizzare in:

- una modellazione naturale della realtà;
- la stratificazione e specializzazione di dati e comportamenti;
- la modularità del software;
- una gestione e manutenzione di grandi progetti più semplice;
- la realizzazione estremamente facile di interfacce grafiche.

I **linguaggi OOP** possono essere ulteriormente classificati in alcune categorie:

- **puri**: ogni entità è un oggetto, come in **Smalltalk** e **Eiffel**;
- **ibridi**: esistono entità che non sono oggetti, per esempio i tipi primitivi **int**, **char**, **float** ecc., come in **Java**, **C++**, **Visual Basic** ecc.

Come vedremo, programmare a oggetti vuol dire solo applicare sistematicamente e consapevolmente tre aspetti chiave, che sono l'**incapsulamento**, l'**ereditarietà** e il **polimorfismo**, sfruttando sintassi adeguate, che offrono costrutti adatti a implementarli in modo semplice ed efficiente.

|| **Java** è un **linguaggio ibrido**, che mantiene una certa compatibilità, permettendo di mischiare i due paradigmi, imperativo e a oggetti, consentendo un passaggio graduale da un paradigma all'altro.

L'utilizzo che abbiamo fatto finora di **Java** è stato possibile proprio per la parte “imperativa” presente in esso, che permette la definizione di variabili di tipo primitivo: tutto il codice scritto sino a oggi non ha “quasi” nessun riferimento alla programmazione **OOP**, se non per alcune intestazioni e per l'utilizzo di particolari librerie, come la classe **Scanner**.

In particolare, abbiamo scritto i nostri algoritmi all'interno del **main()**, del quale ricordiamo la struttura.

```
public static void main(String [ ] args){...}
```

Abbiamo detto che il **main()** è un **metodo** particolare che ha la funzione di **entry point**, cioè di “colui che avvia il programma” e i tre “termini **qualificatori** che lo precedono” hanno il seguente significato:

```
public          // è pubblico, cioè può essere invocato da chiunque
static         // è un metodo statico
void           // non ritorna nessun dato/risultato a chi lo richiama
```

Il termine **static** merita un'osservazione particolare.

Ambiente statico e metodi statici

Finora abbiamo scritto i nostri algoritmi all'interno del metodo `main()`, anche richiamando comandi (funzioni) presenti in librerie esterne, includendole con la direttiva `import`, per esempio:

```
import java.util.Scanner           // libreria per la gestione dell'input
import java.lang.Math             // libreria con le funzioni matematiche
import java.text.DecimalFormat;   // libreria per la formattazione dell'output
```

Queste librerie contengono funzioni (**funzioni membro statiche/metodi statici**) che vengono utilizzate richiamandole su dati elementari, cioè anche su **NON** oggetti, e quindi all'interno di ambienti statici, dove l'allocazione di memoria delle variabili è effettuata in **compiled time**.

Le **funzioni membro statiche** sono metodi particolari, che possono essere invocati senza la necessità di **istanziare un oggetto** della classe di appartenenza: per questo motivo, spesso si dice che i metodi statici appartengono alla classe e non alle sue istanze.

Vedremo che quest'ultima affermazione trova riscontro nell'effetto che il qualificatore `static` ha sul ciclo di vita anche di eventuali **costanti** e/o **variabili** dichiarate come statiche, istanziate all'inizio del programma, condivise da tutti gli oggetti della classe e persistenti fino alla sua terminazione.

Dato che i membri statici di una classe esistono nella memoria del nostro programma, anche se non è stato istanziato nessun oggetto, all'interno dei metodi statici si può fare riferimento a soli membri statici di classe: essi potranno interagire con le variabili di istanza, ma solamente con quelle statiche.

Il metodo `main()` è **statico** perché, quando è invocato, non esiste ancora alcun oggetto!

È anche possibile effettuare quella che prende il nome di **importazione statica**: basta aggiungere il qualificatore `static` alla libreria, dopo la clausola `import`, in modo che non sia necessario specificare la classe di appartenenza per utilizzare i suoi attributi campi e metodi nel programma.

Vediamo un esempio con la funzione che calcola la radice quadrata.

SENZA IMPORTAZIONE STATICA	CON IMPORTAZIONE STATICA
<pre> EsempioRadice1 1 import java.lang.Math; 2 public class EsempioRadice1 { 3 public static void main(String[] arg) { 4 double num = 36.0; 5 System.out.println("Il numero e' : " + num); 6 System.out.println("La sua radice: " + Math.sqrt(num)); 7 } 8 } </pre>	<pre> EsempioRadice2 1 import static java.lang.Math.sqrt; 2 public class EsempioRadice2 { 3 public static void main(String[] arg) { 4 double num = 36.0; 5 System.out.println("Il numero e' : " + num); 6 System.out.println("La sua radice: " + sqrt(num)); 7 } 8 } </pre>

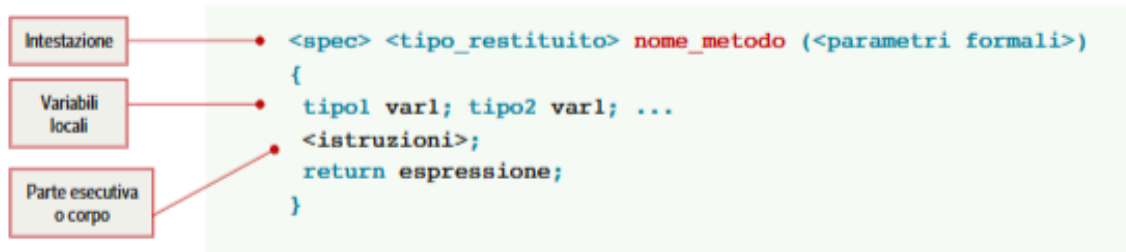
Se usiamo troppi **metodi statici**, si utilizza poco la programmazione orientata agli oggetti: ciò significa che le classi che usiamo non modellano adeguatamente le entità su cui vogliamo operare.

Metodi statici definiti dal programmatore

Anche l'utente può scrivere **metodi statici**, inserendoli direttamente nel codice della classe che contiene il `main()`. In generale, la codifica di un metodo utilizza la seguente struttura:

- **testata (header) o intestazione o scrittura;**
- **istruzioni**, racchiuse tra le parentesi `{}`.

In particolare possiamo distinguere:



L'**intestazione** è la parte più delicata ed è formata dai seguenti elementi:

- **spec**: specificatore di visibilità, che se viene omissso equivale a **public**; nel nostro caso indichiamo con **static** il fatto che stiamo definendo un **metodo statico**;
- **nome_metodo** (o identificatore di metodo): a ogni metodo dev'essere associato un nome, come per le variabili, per poterlo "richiamare"; per convenzione, l'identificatore deve iniziare con la lettera **minuscola**;
- **<lista_parametri_formali>**: è un elenco di coppie costituite da **<tipo_variabile> <identificatore_variabile>** separate da una virgola, che permettono di interfacciare la funzione al programma chiamante, fornendo le variabili che la funzione dovrà elaborare (indicano che cosa la funzione "si aspetta" da chi la utilizzerà);
- **tipo_restituito** (o tipo del risultato): il risultato delle elaborazioni si chiama **parametro di ritorno di una funzione** e può essere di uno dei tipi elementari definiti dal linguaggio: **int**, **double**, **bool** ecc.; il tipo del parametro di ritorno dev'essere indicato come primo elemento nella testata della funzione, in analogia con quanto è necessario fare per definire le variabili.

|| I parametri vengono passati esclusivamente per valore.

Vediamo un primo esempio.

ESEMPIO

MAGGIORE TRA DUE NUMERI

Realizziamo un metodo che determina e restituisce il valore maggiore tra due numeri; tale metodo avrà:

- in ingresso due numeri interi;
- in uscita un valore intero;
- un nome significativo che ne permette l'utilizzo (per es., **max**).

Il codice completo del metodo è molto semplice ed è riportato di seguito:

```
13  
14 static int max(int num1, int num2){  
15     if (num1 > num2)  
16         return num1;  
17     else  
18         return num2;  
19 }  
20 }
```

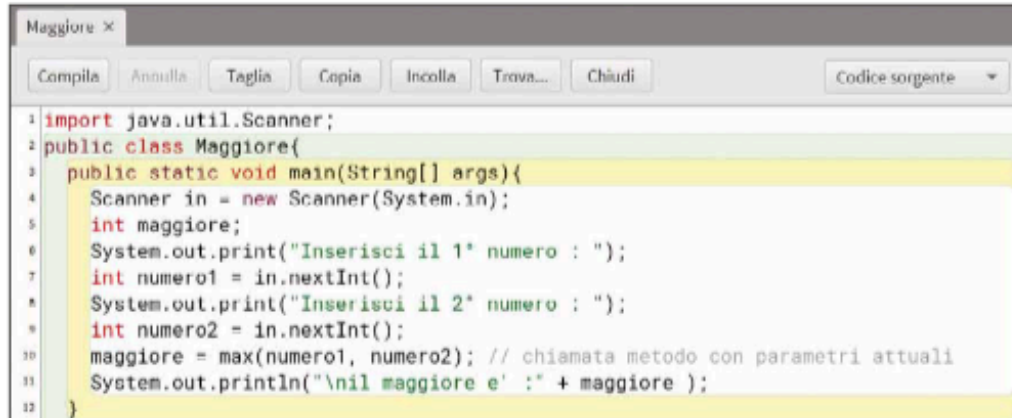
|| Il **prototipo** prende anche il nome di **firma** o **segnatura** di **prototipo** ed è la "carta d'identità" del metodo: in esso individuiamo il nome (**max**) e sappiamo che restituisce un valore intero (**int**), ricevendo dal programma che lo chiama (**programma chiamante**) due dati di tipo intero (due **parametri in ingresso**). Essi saranno posti nelle variabili interne al metodo (**variabili locali**), rispettivamente il primo nella variabile **num1** e il secondo nella variabile **num2**.

PER SAPERNE DI PIÙ

OVERLOADING DEI METODI

La firma o segnatura indica l'insieme del nome e dei parametri di un **metodo** e dev'essere unica: in **Java**, però, possono esistere più metodi con lo stesso nome, a patto che abbiano parametri formali di numero e/o tipo differenti; in questo caso si parla di **overloading** o **sovraccarico** (questo meccanismo sarà dettagliatamente discusso nella prossima Unità dedicata alla **OOP**).

Scriviamo ora il **programma principale** che, dopo aver letto due numeri, richiama il metodo, passandogli proprio come variabili (**parametri attuali**) i due numeri letti:



```

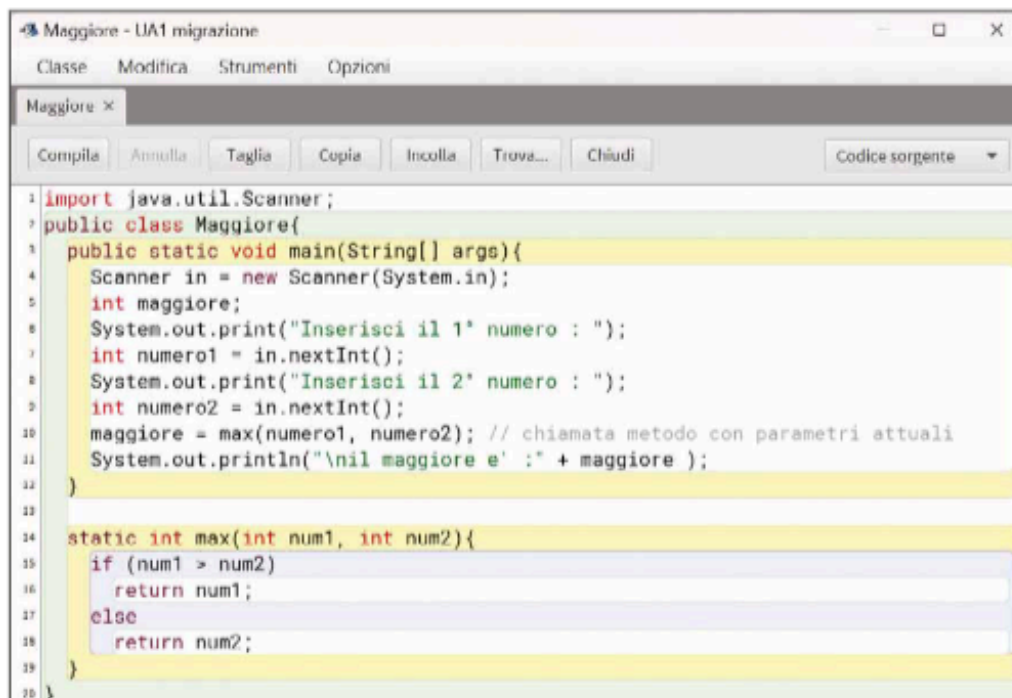
1 import java.util.Scanner;
2 public class Maggiore{
3     public static void main(String[] args){
4         Scanner in = new Scanner(System.in);
5         int maggiore;
6         System.out.print("Inserisci il 1° numero : ");
7         int numero1 = in.nextInt();
8         System.out.print("Inserisci il 2° numero : ");
9         int numero2 = in.nextInt();
10        maggiore = max(numero1, numero2); // chiamata metodo con parametri attuali
11        System.out.println("\nil maggiore e' : " + maggiore );
12    }

```

I **parametri attuali** devono corrispondere ai **parametri formali** in **numero**, **posizione** e **tipo**.

Il linguaggio **Java** ci permette di scrivere i **metodi** nello stesso file del programma principale e di collocarli nella posizione preferita dal programmatore: è possibile, cioè, collocarli prima o dopo la scrittura del codice del **main()**.

Il programma completo che definisce e utilizza il metodo **max()** è il seguente:



```

1 import java.util.Scanner;
2 public class Maggiore{
3     public static void main(String[] args){
4         Scanner in = new Scanner(System.in);
5         int maggiore;
6         System.out.print("Inserisci il 1° numero : ");
7         int numero1 = in.nextInt();
8         System.out.print("Inserisci il 2° numero : ");
9         int numero2 = in.nextInt();
10        maggiore = max(numero1, numero2); // chiamata metodo con parametri attuali
11        System.out.println("\nil maggiore e' : " + maggiore );
12    }
13
14    static int max(int num1, int num2){
15        if (num1 > num2)
16            return num1;
17        else
18            return num2;
19    }
20 }

```

Il codice sorgente di questo programma lo trovi nel file **Maggiore.java**.

Il legame tra **parametri attuali** e **parametri formali** viene effettuato **al momento della chiamata**, in modo dinamico e:

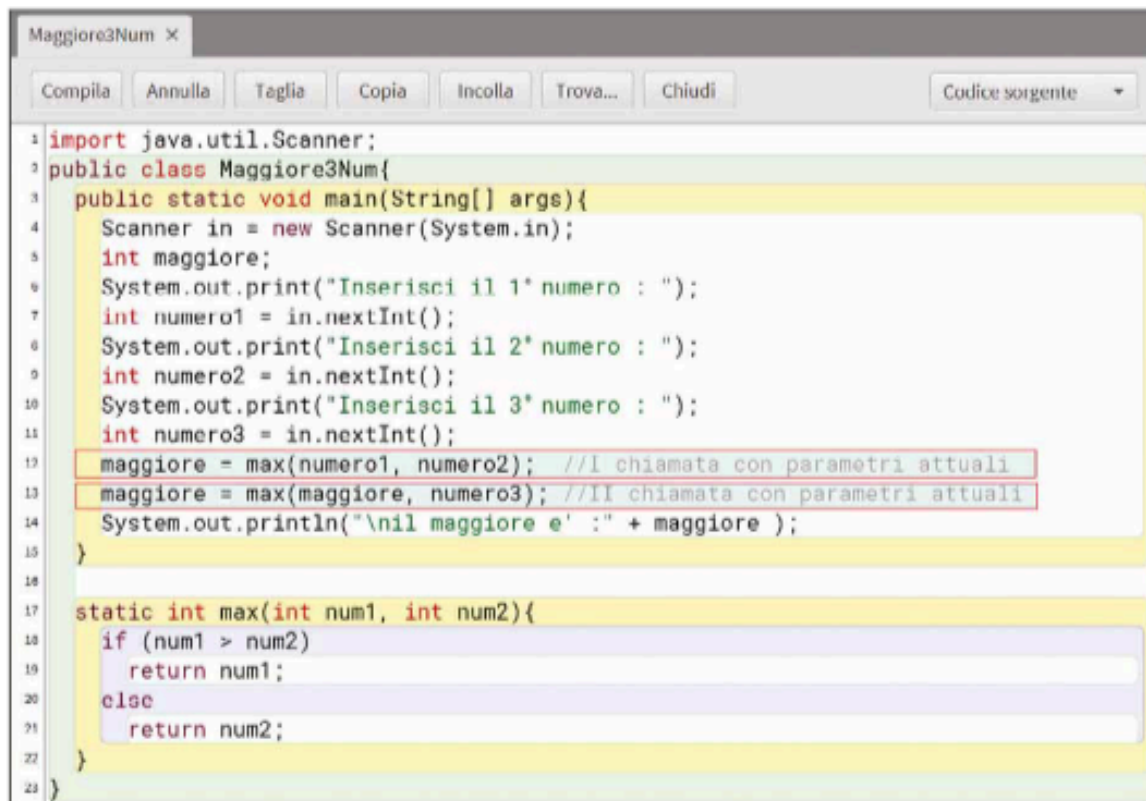
- **vale solo** per l'**invocazione corrente**;
- **vale solo** per la **durata della funzione**.

Vediamo ora un secondo esempio, in cui la stessa funzione è richiamata due volte.

ESEMPIO

MAGGIORE TRA TRE NUMERI

Leggiamo ora tre numeri e individuiamo il maggiore effettuando una doppia chiamata alla funzione, in modo da confrontare con la prima operazione due dei tre numeri e, successivamente, prendendo il maggiore tra i due per confrontarlo con il terzo numero, sempre utilizzando la funzione **max()** prima descritta.



```
1 import java.util.Scanner;
2 public class Maggiore3Num{
3     public static void main(String[] args){
4         Scanner in = new Scanner(System.in);
5         int maggiore;
6         System.out.print("Inserisci il 1° numero : ");
7         int numero1 = in.nextInt();
8         System.out.print("Inserisci il 2° numero : ");
9         int numero2 = in.nextInt();
10        System.out.print("Inserisci il 3° numero : ");
11        int numero3 = in.nextInt();
12        maggiore = max(numero1, numero2); //I chiamata con parametri attuali
13        maggiore = max(maggiore, numero3); //II chiamata con parametri attuali
14        System.out.println("\nIl maggiore e' : " + maggiore );
15    }
16
17    static int max(int num1, int num2){
18        if (num1 > num2)
19            return num1;
20        else
21            return num2;
22    }
23 }
```

Possiamo osservare che il risultato della prima chiamata della funzione (**parametro in uscita**) è dapprima memorizzato nella variabile **maggiore**, che successivamente passa come parametro attuale (**parametro in ingresso**) alla seconda chiamata, sempre della stessa funzione.

Il codice sorgente di questo programma lo trovi nel file [Maggiore3Num.java](#).

Un esempio completo

Vediamo adesso un semplice esempio che ci permette di comprendere il concetto di moduli funzionali: scomponiamo il problema in sottoproblemi e, per ciascuno di essi, scriviamo un sottoprogramma che li risolve, riducendo il **main()** a un semplice insieme di chiamate di funzioni.

Nel caso di problemi complessi, lo schema rimane lo stesso e la difficoltà è “spostata” all’interno delle singole funzioni che, a loro volta, possono essere scomposte in funzioni più semplici, e via di seguito, come un insieme di “scatole cinesi”.

PROBLEMA SVOLTO PASSO PASSO

Il problema

Leggi il raggio di una circonferenza e calcola il perimetro e l'area del cerchio che essa delimita.

L'analisi e la strategia risolutiva

Dalla matematica, conosciamo le formule che ci permettono di calcolare l'area e il perimetro del cerchio:

perimetro = $2 \times \text{raggio} \times \pi$

area = $\text{raggio} \times \text{raggio} \times \pi$

Durante la fase di input verifichiamo che l'utente inserisca un numero positivo.

La pseudocodifica e l'algoritmo risolutivo

La **pseudocodifica** e il **flow chart** sono riportati di seguito.

I affinamento	II affinamento
• leggi il raggio	• chiama leggiDati ()
• calcola il perimetro	• chiama calcolaPerimetro()
• calcola l'area	• chiama calcolaArea()
• visualizza i risultati	• chiama stampaRisultati()

III affinamento – pseudocodifica

```

int leggiDati()
  mentre(raggio <= 0) fai
    leggi(raggio)
  fineMentre
  ritorna raggio

int calcolaPerimetro(int raggio)
  perimetro <- 2 * raggio * pigreco
  ritorna perimetro

int calcolaArea(int raggio)
  area <- raggio * raggio * pigreco
  ritorna area

void stampaRisultati(int perimetro, int area)
  scrivi("la misura del perimetro e'", perimetro)
  scrivi("la misura dell'area e'", area)
  ritorna void

inizio                                     // programma principale
  raggio <- leggiDati()
  perimetro <- calcolaPerimetro(raggio)
  area <- calcolaArea(raggio)
  stampaRisultati(perimetro, area)
fine
  
```

Codifica in Java ed esecuzione del programma

```

1 import java.util.Scanner;
2 public class Circonferenza{
3     final static double PIGRECO = 3.1415; // costante comune
4     public static void main(String[] args){
5         int raggio;
6         double perimetro, area;
7         raggio = leggiDati();
8         perimetro = calcolaPerimetro(raggio);
9         area = calcolaArea(raggio);
10        stampaRisultati(perimetro, area);
11    }
12
13    static int leggiDati(){
14        Scanner in = new Scanner(System.in);
15        int dato;
16        do{
17            System.out.print("\nInserisci il raggio : ");
18            dato = in.nextInt();
19        }while (dato <= 0);
20        return dato;
21    }
22
23    static double calcolaPerimetro(int raggio){
24        double numero1;
25        numero1 = 2 * raggio * PIGRECO;
26        return numero1;
27    }
28
29    static double calcolaArea(int raggio){
30        double numero1;
31        numero1 = raggio * raggio * PIGRECO;
32        return numero1;
33    }
34
35    static void stampaRisultati(double dato1, double dato2){
36        System.out.printf("la misura del perimetro e': %.2f%n", dato1);
37        System.out.printf("la misura dell'area e': %.2f%n ", dato2);
38    }
39 }
40
41 // | System.out.format("la misura del perimetro e': %.2f%n", dato1);
42 // System.out.format("la misura dell'area e': %.2f%n ", dato2);

```

Classe compilata - nessun errore sintattico salvato

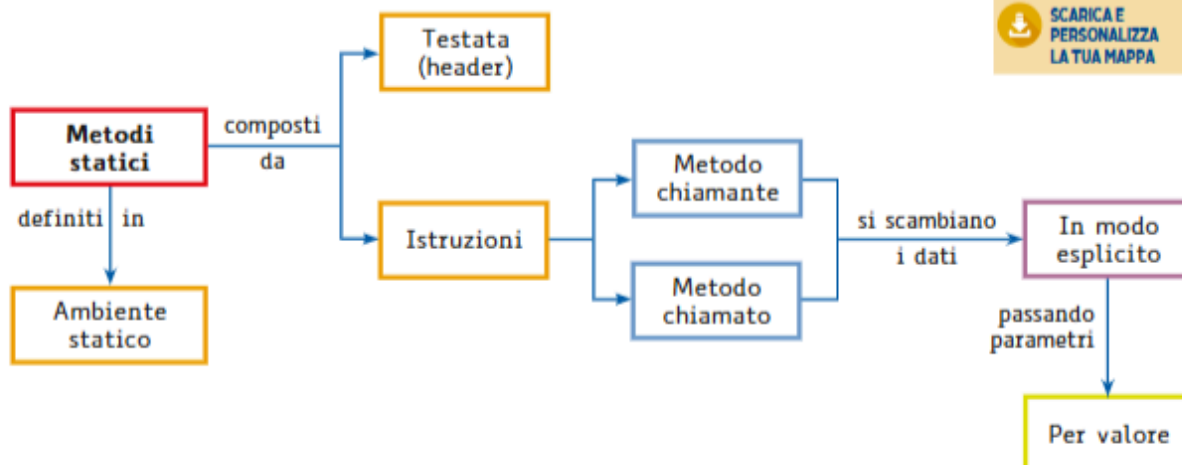
MAPPA CONCETTUALE



AREA DIGITALE



SCARICA E
PERSONALIZZA
LA TUA MAPPA



Che cosa abbiamo imparato?

- ➔ Spesso all'interno di un problema sono presenti due sottoproblemi identici: in questo caso, lo stesso codice viene riscritto due (o più) volte.
- ➔ Con l'aumentare della complessità di un problema aumenta la dimensione del programma e un segmento di codice lungo risulta spesso di difficile lettura e comprensione e di difficile manutenzione: per ovviare a queste situazioni, è necessario ricorrere alla scomposizione in sottoprogrammi.
- ➔ Più un codice è parametrico, più facilmente può essere riutilizzato (l'ingegneria del software indica questo processo con il termine *reengineering*).
- ➔ I principali vantaggi e benefici offerti dai sottoprogrammi sono:
 - riusabilità: consentono di utilizzare lo stesso codice come "mattoncino" per la soluzione di problemi diversi;
 - astrazione: permettono di esprimere operazioni complesse in modo sintetico;
 - risparmio: consentono di scrivere una sola volta il codice usato più volte.

Ora prova tu a rispondere

- ➔ Qual è lo schema di funzionamento di un metodo statico?
- ➔ Che cosa si intende per header di un metodo statico?
- ➔ Che cosa sono i parametri di ingresso di un metodo statico?
- ➔ Che cosa sono i parametri in uscita di un metodo statico?
- ➔ Che cosa viene chiamato prototipo di un metodo statico?