12.12.2025

1. Metodi e Composizione del Software

Il concetto che "un programma Java è fatto da metodi che si usano a vicenda" riflette un principio fondamentale della programmazione: la **decomposizione funzionale** e il **riuso del codice**.

Ogni metodo rappresenta un'unità logica che svolge un compito specifico. I programmi complessi emergono dalla composizione di questi metodi più semplici. Questo approccio:

- Migliora la **leggibilità**: ogni metodo ha un nome che descrive cosa fa
- Facilita il **debugging**: i problemi sono isolati in unità più piccole
- Permette il **riuso**: lo stesso metodo può essere chiamato da contesti diversi
- Supporta l'**astrazione**: chi usa un metodo non deve sapere come è implementato internamente

La metafora filosofica che hai citato ("siamo vivi per usarci") cattura bene l'idea di **interdipendenza** e **collaborazione** tra componenti software.

2. Private: Il Controllo dell'Accesso

La parola chiave `private` è uno dei modificatori di **visibilità** (o accesso) in Java. Rappresenta il livello più restrittivo di incapsulamento.

Funzionamento teorico:

- Un membro `private` è accessibile solo all'interno della stessa classe in cui è definito
- Altre classi non possono vedere né accedere a questi membri
- Questo crea un confine netto tra l'interfaccia pubblica di una classe e i suoi dettagli implementativi

Perché è importante:

- **Incapsulamento**: nasconde i dettagli implementativi, esponendo solo ciò che è necessario
- **Manutenibilità**: puoi modificare l'implementazione interna senza rompere il codice che usa la classe
- **Protezione dell'invariante**: garantisce che lo stato interno della classe rimanga coerente

I livelli di visibilità in Java formano una gerarchia:

- `private`: solo la classe stessa
- (default/package-private): solo il package
- `protected`: package + sottoclassi
- `public`: ovunque

3. Static: Appartenenza alla Classe

`static` indica che un membro appartiene alla **classe** piuttosto che alle sue **istanze** (oggetti).

Differenza concettuale:

- **Membri non-static (di istanza)**: ogni oggetto ha la propria copia. Se hai 100 oggetti, hai 100 copie di quella variabile
- **Membri static (di classe)**: esiste una sola copia condivisa da tutti gli oggetti di quella classe

Implicazioni pratiche:

- I metodi `static` possono essere chiamati senza creare un oggetto della classe
- I metodi `static` non possono accedere a membri non-static (perché non esiste un "questo oggetto" specifico)
- Le variabili `static` sono utili per contatori condivisi, configurazioni globali, costanti

Il metodo main: `public static void main(String[] args)` è `static` perché la JVM deve poterlo chiamare prima che esistano oggetti della classe.

4. Void: L'Assenza di Valore di Ritorno

`void` indica che un metodo **non restituisce alcun valore** al chiamante.

Significato teorico:

- Il metodo viene eseguito per i suoi **effetti collaterali** (side effects) piuttosto che per produrre un valore
- Esempi di effetti collaterali: stampare output, modificare variabili, scrivere su file, aggiornare lo stato di un oggetto

Contrasto con metodi che restituiscono valori:

- Metodo con `void`: esegue un'azione
- Metodo con tipo di ritorno: calcola e restituisce un risultato

Nota per C:
Poiché scrivi in C, noterai che il concetto è identico: `void` in C ha lo stesso significato. Java ha ereditato questa sintassi da C.

Questi quattro elementi insieme definiscono la **firma** e le **caratteristiche** di un metodo:

```
private static void printPerson(String name, String surname, int age) {
    // Questo metodo:
    // - è privato: usabile solo in questa classe
    // - è statico: appartiene alla classe, non agli oggetti
    // - è void: non restituisce valori, agisce per effetti collaterali
}
```

1. Evitare la Ripetizione del Codice (DRY Principle)

Il principio **DRY** (Don't Repeat Yourself) è fondamentale nella programmazione.

2. Centralizzazione del Codice

La centralizzazione significa che la **logica risiede in un unico posto**.

Concetto teorico:
Quando centralizzi il codice in un sottoprogramma, crei un **punto unico di verità** (single source of truth). Qualsiasi cambiamento nella logica si propaga automaticamente a tutti i punti dove il sottoprogramma viene chiamato.

Parametri: Il Passaggio dei Valori

I parametri sono il **meccanismo di comunicazione** tra chiamante e sottoprogramma

```
Definizione del metodo (parametri formali):
static int somma(int a, int b) {
   //           ↑     ↑
   //       parametri formali
   //       "segnaposto" che riceveranno valori
   return a + b;
}
```

```
Chiamata del metodo (argomenti/parametri attuali):

int risultato = somma(5, 3);
//                ↑  ↑
//           argomenti/valori concreti
```

Il Meccanismo di Binding (Associazione)

Quando chiami un metodo, avviene un **binding** tra argomenti e parametri:

1. **Valutazione degli argomenti**: i valori 5 e 3 vengono calcolati
2. **Copia dei valori**: 5 viene copiato in `a`, 3 viene copiato in `b`
3. **Esecuzione**: il metodo lavora con le copie locali
4. **Ritorno**: il risultato viene restituito al chiamante

💡 **Aspetto critico in Java:**

- **Tipi primitivi** (int, double, boolean, char, ecc.): vengono passati **per valore** (copia)
- **Oggetti**: viene passato il **riferimento per valore** (copia del riferimento, ma l'oggetto è lo stesso)

```
static void modifica(int x) {
    x = x + 10;  // modifica solo la copia locale
    System.out.println("Dentro: " + x);  // 15
}

public static void main(String[] args) {
    int numero = 5;
    modifica(numero);
    System.out.println("Fuori: " + numero);  // ancora 5!
}
```

Spiegazione teorica:

- `numero` nel `main` e `x` nel metodo sono **variabili separate**
- Il valore 5 viene **copiato** da `numero` a `x`
- Modificare `x` non influenza `numero`

Scope (Ambito) delle Variabili

Le variabili parametro hanno **scope locale** al metodo:

```
static void metodo1(int x) {
    // x esiste solo qui dentro
    System.out.println(x);
}  // x "muore" qui

static void metodo2() {
```

```
    // x non esiste qui
    // System.out.println(x);  // ERRORE!
}
```

Principio di località:
Ogni metodo ha il suo **spazio dei nomi** privato.
Questo previene conflitti e rende il codice più prevedibile.

4. Analogia con le Funzioni Matematiche

I sottoprogrammi sono simili alle funzioni matematiche:

Funzione matematica:
```
f(x, y) = x² + y²
f(3, 4) = 9 + 16 = 25

static int f(int x, int y) {
    return x*x + y*y;
}
// f(3, 4) restituisce 25
```

Naming Conflict (Conflitto di Nomi)

1. Cos'è un Naming Conflict
Un **naming conflict** si verifica quando due o più entità (variabili, metodi, classi) hanno lo **stesso nome** nello stesso contesto, creando ambiguità su quale entità si stia riferendo.

Conflitto tra Variabile Locale e Variabile di Istanza

```
class Esempio {
    private int numero = 5;  // variabile di istanza
```

```
    void metodo() {
        int numero = 10;  // variabile locale (NASCONDE quella di istanza)
        System.out.println(numero);  // stampa 10, non 5
    }
}
```

Questo è permesso, ma crea shadowing (mascheramento):

La variabile locale numero "nasconde" la variabile di istanza
Per accedere alla variabile di istanza, devi usare this.numero

Conflitto tra Parametro e Variabile di Istanza

```
class Persona {
    private String nome;

    // Pattern comune: parametro con stesso nome della variabile di istanza
    void setNome(String nome) {
        this.nome = nome;  // this.nome è la variabile di istanza
                           // nome è il parametro
    }
}
```

Risoluzione con this:

this.nome si riferisce alla variabile di istanza
nome (senza this) si riferisce al parametro
Questo è un pattern molto comune in Java (constructor e setter)

Conflitto tra Metodi (Overloading)

```
class Calcoli {
    static int somma(int a, int b) {
```

```
        return a + b;
    }

    static double somma(double a, double b) {  // OK! Firma diversa
        return a + b;
    }

    static int somma(int a, int b) {  // ERRORE! Firma identica
        return a + b + 1;
    }
}
```

Overloading (sovraccarico):

Più metodi possono avere lo stesso nome
SE hanno parametri diversi (numero o tipo)
La firma del metodo include: nome + tipi dei parametri (NON il tipo di ritorno)
Java sceglie quale metodo chiamare in base agli argomenti passati

CONVENZIONI BEST PRACTICE

```
// Costanti: MAIUSCOLE_CON_UNDERSCORE
static final int MAX_TENTATIVI = 3;

// Variabili e metodi: camelCase
int numeroStudenti;
void calcolaMedia() { }

// Classi: PascalCase
class GestoreStudenti { }

// Package: minuscole
package com.azienda.progetto;
```

Scomposizione in Metodi (Method Decomposition)

1. Cos'è la Scomposizione in Metodi

La **scomposizione in metodi** (o decomposizione funzionale) è il processo di dividere un problema complesso in **sottoproblemi più semplici**, ciascuno risolto da un metodo separato.

Principio fondamentale:

> Un problema grande e complicato diventa gestibile quando lo dividi in problemi più piccoli e indipendenti.

2. Perché Scomporre

A. Gestione della Complessità

Il cervello umano può gestire circa **7±2 elementi** contemporaneamente (limite della memoria di lavoro). Un metodo con 200 righe è incomprensibile, ma 10 metodi da 20 righe ciascuno sono gestibili.

B. Riusabilità

Metodi piccoli e specifici possono essere riutilizzati in contesti diversi.

C. Manutenibilità

È più facile:

- Trovare e correggere bug
- Modificare una funzionalità
- Capire cosa fa il codice

D. Testing

Ogni metodo può essere testato **isolatamente**, semplificando la verifica della correttezza.

E. Leggibilità

Il codice scomposto si legge come un libro ben strutturato, con capitoli e sezioni.

CODICE MONOLITICO

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Input
    System.out.print("Nome studente: ");
    String nome = input.nextLine();
    System.out.print("Voto 1: ");
    int voto1 = input.nextInt();
    System.out.print("Voto 2: ");
    int voto2 = input.nextInt();
    System.out.print("Voto 3: ");
    int voto3 = input.nextInt();

    // Validazione
    if (voto1 < 0 || voto1 > 30 || voto2 < 0 ||
        voto2 > 30 || voto3 < 0 || voto3 > 30)
      {
        System.out.println("Voti non validi!");
        return;
    }

    // Calcolo media
    double media = (voto1 + voto2 + voto3) / 3.0;

    // Determinazione esito
    String esito;
    if (media >= 18) {
        esito = "PROMOSSO";
    } else {
        esito = "BOCCIATO";
```

```
    }

    // Output
    System.out.println("\n=== RISULTATI ===");
    System.out.println("Studente: " + nome);
    System.out.println("Media: " + media);
    System.out.println("Esito: " + esito);
    System.out.println("==================");
}
```

Problemi:

Tutto in un unico blocco → difficile da capire
Logica mescolata → input, validazione, calcolo, output tutto insieme
Non riutilizzabile → per calcolare un'altra media devi riscrivere tutto
Difficile da testare → non puoi testare solo il calcolo della media
Difficile da modificare → cambiare una parte rischia di rompere il resto

CODICE SCOMPOSTO:

```
public static void main(String[] args)
{
    Scanner input = new Scanner(System.in);

    String nome = leggiNome(input);
    int[] voti = leggiVoti(input, 3);

    if (!votiValidi(voti)) {
        System.out.println("Voti non validi!");
        return;
    }

    double media = calcolaMedia(voti);
    String esito = determinaEsito(media);
```

```
    mostraRisultati(nome, media, esito);
}

static String leggiNome(Scanner input)
{
    System.out.print("Nome studente: ");
    return input.nextLine();
}

static int[] leggiVoti(Scanner input, int numeroVoti)
{
    int[] voti = new int[numeroVoti];
    for (int i = 0; i < numeroVoti; i++) {
        System.out.print("Voto " + (i + 1) + ": ");
        voti[i] = input.nextInt();
    }
    return voti;
}

static boolean votiValidi(int[] voti)
{
    for (int voto : voti) {
        if (voto < 0 || voto > 30) {
            return false;
        }
    }
    return true;
}

static double calcolaMedia(int[] voti)
{
    int somma = 0;
    for (int voto : voti) {
        somma += voto;
    }
    return (double) somma / voti.length;
```

```
}

static String determinaEsito(double media)
{
   return media >= 18 ? "PROMOSSO" : "BOCCIATO";
}

static void mostraRisultati(String nome, double media, String esito)
{
   System.out.println("\n=== RISULTATI ===");
   System.out.println("Studente: " + nome);
   System.out.println("Media: " + media);
   System.out.println("Esito: " + esito);
   System.out.println("==================");
}
```

Vantaggi:

Il main è leggibile come un libro:
leggi nome → leggi voti → valida → calcola → mostra
Ogni metodo ha una singola responsabilità
Riutilizzabile: calcolaMedia() può essere usato ovunque
Testabile: puoi testare calcolaMedia() separatamente
Manutenibile: vuoi cambiare il formato dell'output?
Modifichi solo mostraRisultati()

Principi della Scomposizione

- Single Responsibility Principle (SRP) → **Ogni metodo deve fare UNA SOLA COSA.**
- High Cohesion → il codice all'interno di un metodo deve essere strettamente correlatoo
- Low Coupling → I metodi devono essere il piu indipendenti possibile

- Livelli di astrazione → metodi allo stesso livello di astrazione dovrebbero essere chiamati insieme