

Programmazione ad'Oggetti

Leonardo Mengozzi

Titoletti indice link a rispettive sezioni, in alto a sinistra "[←Indice](#)" link a pagina Indice.

Indice

1 Fasi sviluppo Software	2
1.1 Problem space vs Solution space	2
1.2 Programmazione ad oggetti (OOP)	2
2 Teoria degli oggetti	3
2.1 Com'è fatto un buon oggetto	3
3 Perché Java?	3
3.1 Elementi fondamentali	4
3.2 Differenze con C	4
3.3 Gestione Oggetti in memoria	4
4 Struttura Programma Java	4
4.1 Esecuzione Programma	5
4.2 I package	5
4.3 Convenzioni	5
4.3.1 I Commenti	6
5 (Almost) Everything is an object	6
5.1 Stack e Heap	6
5.2 Tipi Primitivi	6
5.2.1 Boolean	7
5.2.2 Caratteri	7
5.2.3 Interi	7
5.2.4 Virgola mobile	8
5.3 Conversioni	8
5.4 Array	8
5.4.1 Foreach	8
6 Classi	9
6.1 Campi	9
6.2 final e costanti	10
6.2.1 Oggetti immutabili	10

6.3	Metodi	10
6.3.1	return this	10
6.4	Costruttore	11
6.4.1	Overloading	11
6.5	this	11
6.6	static	11
6.7	Livelli d'accesso	11
6.7.1	Incapsulamento	12
6.8	toString()	12
6.9	Precisazioni	12
6.10	Fasi Implementazione	13
7	Oggetti lato Utente	13

1 Fasi sviluppo Software

Un Programma (algoritmo) risolve una classe di problemi.

Un Sistema software fornisce varie funzionalità grazie alla cooperazione di componenti di diversa natura.

Fasi processo sviluppo: **1 Analisi** che fare?, **2 Design** come farlo?, **3 Implementazione/codifica** Quale algoritmo?, **4 Post-coficia**. Fasi 1-2 fatte dai senior e fase 3 junior (2-3 oggi unificate). La fase 4 più impiegare fino 70% se fase precedenti fatte male/sbrigativamente (Software crisis). Tutte fasi fattibili dalla stessa persona.

Un analisi è corretta se persone diverse giungono alla stessa soluzione.

1.1 Problem space vs Solution space

Problem space sono le entità/relazioni/processi del mondo reale che formano il problema. Solution space sono le entità/relazioni/processi nel mondo artificiale (esprese nel linguaggio di programmazione).

Per passare dal Problem space al Solution solution si esegue un "mapping" che più semplice è meglio ho fatto le **astrazioni**¹.

I linguaggi di programmazione attuano l'astrazione coi loro costrutti, più o meno performanti, che rendono il mapping più o meno facile.

I linguaggi moderni hanno un livello d'astrazione lontano dall'HardWare, i suoi problemi e la gestione della memoria.

1.2 Programmazione ad oggetti (OOP)

Vantaggi: poche astrazioni chiave, mapping ottimo e semplice, estensibilità e riutilizzo, librerie auto costruite, C-like, esecuzione efficiente. **Critiche:** necessaria disciplina.

¹Strumento che semplifica sistemi informatici ma anche del mondo reale evidenziando "la parte importante". Si possono fare più livelli di astrazione

2 Teoria degli oggetti

Classe: Descrizione comportamento e forma oggetti. Indica come comunicare con i suoi oggetti, con messaggi che modificano stato e comportamento.

Oggetto: Entità (istanza di classe) manipolabile, con memoria, che comunicano tramite le loro operazioni descritte dalla classe di appartenenza.

Oggetti della stessa classe hanno comportamento e forma indentica, sono detti simili. Un oggetto non cambia mai classe, semmai si elimina e sene crea il sostituto.

Nota: L'approccio OOP è usato anche in UML.

2.1 Com'è fatto un buon oggetto

Un oggetto ha un interfaccia², deve fornire un servizio, deve nascondere le implementazioni (riutilizzabili) e l'intero oggetto deve essere riutilizzabile tramite ereditarietà. Precisazioni:

- Un oggetto fornisce un **sotto-servizio** dell'intero programma³ (principio decomposizione). Linne guida: 1 oggetto senza servizio si elimina, 2 oggetto con più servizi si divide.
- L'implementazione di un oggetto (logiche interne) devono essere note solo al creatore della classe (Information hiding), così facendo l'utilizzatore è tutelato, da modifiche interne, avendo una piccola visione del tutto. *less is more*.
- Il creatore e l'utilizzatore riutilizzano le classi con gli approcci:
 1. **has-a** (composizione), classe costituita da altre classi (oggetto ha come campi altri oggetti). Approccio dinamico, occultabile.
 2. **is-a** (ereditarietà), classe estende servizi di un'altra classe (oggetto ha campi/metodi di altri oggetti).

3 Perché Java?

- **Write once run everywhere**, eseguibile ovunque senza ricompilazione grazie JVM (HardWare virtuale, a stack) che processa un codice specifico "byte code", creando il corrispettivo eseguibile per ogni pc/os. Meno prestante.
- **Keep it simple, stupid**, in teoria non in pratica.

²Insieme dei metodi definiti dall'interfaccia con cui l'oggetto riceve messaggi

³Set di oggetti che si comunicano cosa fare.

3.1 Elementi fondamentali

Espressione: fornisce un risultato, riusabile ove si attende un valore.

Comando: istruzione da terminare con ";", non componibile con altre

Alcune parole chiave del linguaggio: `for`, `while`, `do`, `switch`, `if`, `break`, `continue`, `return`, `var`, ecc.

3.2 Differenze con C

- Condizioni di `if`, `for`, `while`, `do` restituiscono `boolean`.
- Nel `for` è possibile dichiarare le variabili contatore (visibili solo internamente).
`for (<tipo> <c1>, ...; <condizioneBoolean>, <modificaC1>, ...){...}`
- Java da `unreachable statement`, `variable may not have been initialised`, `missing return statement`, ecc come errori, C solo come warning. Approccio più "rigido" ma anche più "corretto".

3.3 Gestione Oggetti in memoria

Alla creazione di un oggetto, la `new` chiama il gestore della memoria, si calcola la dimensione della memoria da allocare, si alloca, si inizializza e si restituisce il riferimento alla variabile.

La memoria occupata da un oggetto, non si sa com'è organizzata, ma contiene i campi non statici e il riferimento alla classe con la tabella dei metodi. Gli elementi statici sono allocati in una sezione dedicata.

Il **garbage collector** (componente della JVM) dealloca automaticamente memoria heap non più utilizzata direttamente o indirettamente. Un'oggetto continua a esistere dopo la fine esecuzione dello scope di una variabile che gli fa riferimento.

4 Struttura Programma Java

Un programma Java è composta da librerie di classi del JDK, Package⁴ e Moduli⁵, librerie di esterne e un insieme di classi fondamentali.

La classe cardine di un programma è la `class main`⁶.

```
1 class NomeProgetto {
2     ...
3     public static void main(String[] args) {
4         ...
5     }
```

⁴Contenitori, gerarchici tra loro, di una decina di classi di alto livello con scopo comune

⁵Insieme di Package costituente un frammento di codice autonomo.

⁶Un `main` è il punto d'accesso di un programma.

```
6     ...  
7 }
```

Il parametro del main è un array di stringhe che sono i parametri che l'utente può immettere da tastiera quando il programma è lanciato da CLI. Poco usato.

4.1 Esecuzione Programma

1. Salvare la classe in un file **"NomeClasse.java"**.
2. Compilare con *javac NomeFileClasse.java*. Genererà il **bytecode NomeFileClasse.class** per la JVM.
3. Eseguire con *java NomeFileClasse*. La JVM cercherà il main da cui partire a eseguire.

Lavorando con più file: si compila tutto con *javac *.java* poi si esegue solo la classe main.

4.2 I package

Per prassi nella cartella di un progetto, si fa corrispondere ai package auto-costruiti delle directory del file system.

Di default il package di una unità di compilazione⁷, ".java", è la root di gerarchia. Per definire il package di appartenenza si specifica a inizio sorgente: `package pname;`

Per poi usare una classe di un package bisogna importarla (non aggiunge codice nell'eseguibile):

- `import java...;` importa una singola classe.
- `import java...*;` importa l'intero Package.
- `import java.lang.*;` importazione di default.

Il nome completo di una classe dipende dal Package in cui si trova. Se non si importa bisogna specificare all'uso il nome completo.

4.3 Convenzioni

- Linee max 90 caratteri e indenzazione 2-4 caratteri.
- Una sola istruzione per riga e quindi si definisce una sola variabile per riga, quando necessario, sempre inizializzata.
- Apertura "{" a fine riga della dichiarazione e chiusura "}" in linea a dove inizia la riga di apertura.
- Condice senza righe vuote di separazione a meno di metodi/costruttori.

⁷File.java compilabile atomicamente contenente classi una dopo l'altra.

- Non usare assegnamenti dentro espressioni e usare parentesi solo in espressioni non banali.
- Nomi classi e interfacce in "PascalCase". Nomi metodi, campi, variabili in "camelCase" e costanti in "BIG_CASE".
- I metodi che returnano, senza parametri, si chiamano `get...` oppure `is/has...` se return boolean.
- I metodi che returnano void, e accettano parametri, si chiamano `set...`

4.3.1 I Commenti

Oltre a `//...` e `/*...*/` in java abbiamo `/** ... */` un commento multi linea usato per generare documentazione.

5 (Almost) Everything is an object

Le variabili, contenitori con nomi, ora non denotano solo valori numerici (come in C), ma anche veri e propri oggetti irriducibili.

Non ci sono meccanismi per controllo diretto memoria. Le variabili sono nomi "locali" con riferimenti ad *oggetti* e non maschere di indirizzi in memoria a cui accedere direttamente.

Le variabili posso essere di tipo *Java Types* quindi classi predefinite e autoimplementate oppure *tipi primitivi*.

Visibilità legata al blocco di definizione. Variabili non inizializzate sono inutilizzabili.

5.1 Stack e Heap

Gli oggetti sono memorizzati nell'**heap**. Tutte le variabili sono memorizzate nello **stack**.

Le variabili di tipo primitivo contengono direttamente il valore. Le variabili tipo classe contengono il riferimento dell'oggetto oppure null.

Nota: Uno stesso oggetto può essere puntato da variabili che si riferiscono alla stessa identità.

5.2 Tipi Primitivi

Ripasso: Un tipo classifica valori/oggetti tramite un nome, set valori, operatori.

I tipi atomici **signed** del C si sono mantenuti (non conveniva trattarli come oggetti) definendo un'unica interpretazione:

Tipi primitivi	Dimensione	Minimo	Massimo
boolean	—	—	—
char	16bits	Unicode 0	Unicode $2^{16} - 1$
byte	8bits	-128	+127
short	16bits	-2^{15}	$-2^{15} - 1$
int	32bits	-2^{31}	$+2^{31} - 1$
long	64bits	-2^{63}	$-2^{63} - 1$
float	32bits	IEEE754	IEEE754
double	64bits	IEEE754	IEEE754

Le librerie *BigDecimal*, *BigInteger* gestiscono numeri di dimensione/precisione arbitraria.

typing statico: espressioni tipo noto al compilatore, vantaggio intercettazione errori.

Nota: Uso memoria non dato a sapere al programmatore.

5.2.1 Boolean

Introdotta come tipo risultato delle espressioni e condizioni.

- **Valori** true, false.
- **Operatori Unari** ! (not). Operatori binari: & (and), | (or), ^ (xor), && (and-C), || (or-c)⁸.
- **Operatori confronto numerici** <, >, <=, >=.
- **Operatori di uguaglianza** ==, !=. Con gli oggetti di base confronta i riferimenti.
- **Operatore ternario** <eb> ? e1 : e2. restituisce e1 se eb è true, altrimenti restituisce e2. e1 e e2 espressioni dello stesso tipo.

5.2.2 Caratteri

Rappresentazione 'carattere' o 'u<0-65535>'. Formato ASCII o UTF16 (16bit). I caratteri d'escape si fanno '\<specificatore>'.
8 & e — valutano sempre primo e secondo termine, essendo pensati per operazioni bit a bit mentre && e ——— valutano il secondo operatore solo se necessario.

5.2.3 Interi

- **Codificati** in complemento a 2.
- **Operatori** +, -, *, /, %, + e - unari, &, |, ~, <<, >>, >>>. Output stesso tipo Input.
- **Rappresentabili** in codifica decimale (1.000.000), ottale (0...), esadecimale (0x...).
- **Input tastiera** default int (più usato e efficiente). Se voglio un long va aggiunta una "l" dopo il numero.

⁸& e — valutano sempre primo e secondo termine, essendo pensati per operazioni bit a bit mentre && e ——— valutano il secondo operatore solo se necessario.

5.2.4 Virgola mobile

- **Operatori** +, -, *, /, %, + e - unari.
- **Codificati** in IEEE754.
- **Rappresentabili** in codifica decimale o scientifica.
- **Input tastiera** default double. Se voglio un float (più efficiente) va aggiunta una "f" dopo il numero.

Ricorda: IEEE754 ha errori di precisione che portano all'approssimazione dei risultati.

5.3 Conversioni

- **Implicita** automaticamente applicata nelle espressioni e nell'assegnamento portando tutti i tipi a quello più "generale" presente nell'espressione. Detta *coercizione*.
 $byte \rightarrow short \rightarrow int \rightarrow long \rightarrow float \rightarrow double$
- **Esplicito** fatto con operatore di casting: `...=<tipo><espressione>;`. Può causare perdita di informazioni.

5.4 Array

Oggetti (variabili hanno riferimento nello heap) di lunghezza esplicita e accessibile, inaccessibile fuori dai limiti (errore esecuzione), mutabili. Indirizzi elementi [0, lunghezza-1].

Sintassi:

- `<tipo>[] <nome> = new <tipo>[] {v1,...,vn};`
- `<tipo>[] <nome> = new <tipo>[<dim>];`. Elementi inizializzati a valore default tipo (false, 0, null).
- `<tipo>[] <nome> = {v1,...,vn};`

Note: Si possono fare array di array. Gli array di oggetti sono oggetti con puntatori ad'altri oggetti.

Accesso a variabile `... <nome>[<ind>]`

Per sapere la lunghezza dell'array `<nome>.length`.

5.4.1 Foreach

Astazione **for** in sola **lettura**, utile quando non importa ordine scorrimento elementi collezione e valore indice.

`for(final<tipo> <v>: <e>)`

- *v* è variabile del tipo indicato che vale via via ogni elemento della collezione.
- *e* è un'espressione che restituisce una collezione di elementi del tipo indicato.
- Si può esplicitare `final`, anche se non servirebbe, perchè il `foreach` crea una nuova variabile a ogni iterazione in realtà.

6 Classi

Sono template (tipo, struttura in memoria, comportamento) per generare oggetti (istanza).

Le classi hanno un nome (NomeClasse) che sarà anche il nome del tipo per le variabili e del file.

I membri fondamentali di una classe sono:

- **Campi**, descrivono la struttura/stato
- **Metodi**, descrivono i messaggi e il comportamento

Poi possiamo avere Costruttori, ecc.

Le classi sono tipi di dato in un linguaggio a oggetti tutto è un oggetto fino a un certo punto.

```
1  [modAccesso] class NomeDiUnaClasse {
2      // Costanti
3      static final <tipo> <NOME_COSTANTE>;
4      ...
5      // Campi
6      private [final] <tipo> <nomeCampo>;
7      ...
8      // Costruttore/i
9      public NomeClasse([<parametri>]) {...}
10     ...
11     // Metodi
12     [modAccesso] [static] <tipo> <nomeMetodo>([<parametri>])
13         {...}
14     ...
15     // Campi e Metodi statici
16     static ...
17 }
```

6.1 Campi

Sono lo stato attuale dell'oggetto. Nome con la minuscola. Simili hai membri di una struttura C, con la differenza che possono essere 0,1,diversi (5-7max). Simili a variabili (tipo+nome), ma non si può usare *var*. Possono essere valori primitivi o altri oggetti (anche della classe stessa). L'ordine dei non conta.

I campi sono iniziabili alla dichiarazione dell'oggetto (coi parametri), sennò sono inizializzati in base al tipo a **0**, **false**, **null**. Mai alla dichiarazione si inializzano (...=...;).

```
public|private|"[final] <tipo> <nomeCampo>;
```

Uso dei campi lato utente: Assegnamento ... `obj.campo = ...`, Lettura ...
= `obj.campo` ...

6.2 final e costanti

Modificatore (da usare quando si può) per campi, variabili e parametri. Dopo inizializzazione non più modificabile il valore.

```
<tipo> final <nomeVar [= new ... | valore;]
```

Per fare le **costanti** (NOME.COSTANTE) usare `static final` ...;. Soluzione a *Magic Number*.

6.2.1 Oggetti immutabili

Oggetti i cui campi non si modificheranno mai, dopo la prima modifica del costruttore.

Si creano dichiarando `final private` tutti i campi della classe di tipo: tipi primitivi o oggetti immutabili.

6.3 Metodi

Definiscono il comportamento dell'oggetto. Nome con la minuscola. Simili a funzioni C. Hanno un intestazione (tipo di ritorno—void, nome, argomenti) e un corpo. I metodi di una classe possono essere 0, 1, diversi.

i metodi possono leggere/scrivere i campi.

Uso dei metodi lato utente ... `obj.metodo()`.... L'invocazione del metodo, corrisponde a inviare un messaggio al receiver (obj nell'esempio) azionando l'esecuzione del corpo del metodo.

```
1 tipoDiRitorno nomeMetodo([tipo1 arg1, ...]) {
2     ...
3     [return ...;]
4 }
```

6.3.1 return this

Un metodo può ritorna `this`, l'oggetto corrente. Deve avere come tipo di ritorno la classe stessa.

Usato per concatenare più metodi in una sola espressione, combinando i risultati parziali, e restituendo con l'ultimo metodo l'oggetto processato:

```
... obj.m1().m2()...mThis();
```

Questo schema è detto **fluente**.

6.4 Costruttore

Simile a un metodo con nome il nome della classe, senza tipo di ritorno e opzionalmente dei parametri formali.

`NomeClasse([parametri])...`

Se non si definisce il costruttore viene implicitamente inserito il costruttore di default (0 parametri) che inizializza i campi ai tipi di default.

6.4.1 Overloading

Si possono definire più costruttori e metodi, ma non è una buona pratica. Nel caso devono essere **distinguibili dal numero e/o tipo** dei parametri.

I costruttori si possono **invocare a vicenda** (riuso codice). `this(...)` deve essere la prima riga del corpo del costruttore.

```
1 NomeClasse(p1) {  
2     this(p1,...);  
3     ...  
4 }
```

6.5 this

Variabile contenente il riferimento all'oggetto che sta gestendo il messaggio corrente. Si usa per rendere meno ambiguo il codice accedendo tramite *this* a campi o metodi. (Usare sempre). ... `this.cmapo` ... `this.metodo()`...

6.6 static

Scollega metodi e campi dagli oggetti di una stessa classe, rendendoli condivisi tra essi. I metodi diventano funzioni pure e i campi variabili globali alla classe.

Si possono fare classi solo statiche (utility class, `NomeClasse+s`) più non statiche (approccio migliore) o miste (campi e metodi static in fondo).

Richiamo fuori dalla classe: ... `nomeClasse.campo|metodo()`.... Richiamo nella classe come normale campo o metodo. Richiamabile anche da un oggetto.

`static` `campo|metodo|classe`

Quando `static`? Si fanno statici i metodi o campi che sono indipendenti dallo stato di un oggetto della classe. Fanno un servizio indipendente dal singolo oggetto.

6.7 Livelli d'accesso

Si antepongono a classi, metodi, campi, costruttori, ecc per definire il grado di utilizzo:

- `public` Visibile e richiamabile da qualunque classe.

- `package` default (no keyword). Visibile e richiamabili dentro il package e invisibili fuori.
- `private` Visibile e richiamabile solo nella classe di definizione.

In una unità di compilazione una sola classe è `public` e con lo stesso nome del file.

Vantaggio: Si può far rispettare al meglio il contratto⁹ di un oggetto.

6.7.1 Incapsulamento

Dichiaro `public` solo metodi/costruttori, di "design", necessari all'utente per interagire/creare l'istanza della classe.

Dichiaro `private` tutti i metodi/costruttori di "implementazione", e **tutti i campi**.

Così modifiche implementative non influenzano uso dell'utente. Controllo dati e riudo possibilità errori lato utente. Maggior distacco tra dato e implementazione. (*Information hiding*).

Principio di decomposizione (*divide et impera*): Soluzione di un problema complesso è la somma di due o più sotto problemi più semplici, tra loro indipendenti.

Tali sotto problemi devono avere il minor numero di dipendenze¹⁰ reciproche. Permettendo maggior autonomia, più modifiche senza danni collaterali e meno interazioni.

Una buona divisione da moduli con basse dipendenze esterne e alte dipendenze interne.

Nella OO le suddivisioni base sono package, classi e metodi.

6.8 toString()

Ogni classe deve definire tale metodo che restituisce una rappresentazione in stringa dell'oggetto, utile in fase di debug.

Automaticamente chiamato dall'operatore "+" nelle concatenazioni tra stringhe e oggetti.

6.9 Precisazioni

Inizializzazioni particolari degli oggetti Stringa:

1. ... = `new String()`; , stringa vuota (è diverso da null).
2. ... = `"..."`, come in C, comportamento speciale degli oggetti Stringa.

⁹Insieme degli scenari d'utilizzo e quindi aspettative di un utente al suo utilizzo.

¹⁰Riferito alle classi è quando una classe usa al suo interno un'altra. Ciò può comportare modifiche a cascata rischiando la *sindrome dell'"intoccabilità"*.

6.10 Fasi Implementazione

1. Progettazione della parte pubblica della classe (nome classe, signature metodi e costruttori necessari).
2. Costruzione dello stato (campi privati, con nome diverso dai metodi relativi).
3. Completamento implementazione. (test).
4. Miglioramento codice finale (commenti, eliminare *Magic Number*, fattorizzare sotto-funzioni). (test).
5. Test del risultato con test-case. (test-driven development con JUnit).

7 Oggetti lato Utente

Dichiarazione, creazione, inizializzazione:

```
<Tipo | var> <nome> = <new Tipo([argomenti])| altraVariabile | null>;
```

- `<Tipo> <nome>;` Si può solo dichiarare una variabile oggetto per poi crearla e inizializzarla successivamente.
- solo quando si scrive `new` (Keyword di linguaggio di chiamata al costruttore) si crea e inizializza un oggetto dalla classe indicata. `new` da il riferimento (`this`) dell'oggetto alla variabile.
- `var`¹¹ fa inferire¹² il tipo della variabile locale per alleggerire il codice. Se manca l'espressione non va, esempio `var i;`
- `altraVariabile` deve essere della stessa classe della variabile che sto definendo.

¹¹Local variable type inference

¹²Far dedurre al compilatore il tipo della variabile locale dall'espressione assegnata.