Programmazione ad'Oggetti

Leonardo Mengozzi

Navigazione tra pagine: titoletti indice link a rispettive sezioni, in alto a sinistra "←Indice" link a pagina Indice.

Indice

1 Fasi sviluppo Sofware

Un Programma (algoritmo) risolve una classe di problemi.

Un Sistema sofware 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 (espresse 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 efficente. **Critiche:** necessaria disciplina.

[1] Strumento 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 interfaccia[2], 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 programma[3] (principio decomposizione). Linne giuda: 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 hidding), cosi 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 Perchè Java?

- **Write once run everywhere**, eseguibile uvunque 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.

[2] Insieme dei metodi definiti dall'interfaccia con cui l'oggetto riceve messaggi
[3] Set di oggetti che si comunicano cosa fare.

3.1 Elementi fondamentali

Espressione: fornisce un risulato, 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 dichiare le variabili contatore (visibili solo internamente).

  ```
  for (<tipo> <c1>,...; <condizoneBoolean>, <modificaC1>,...){...}
  ```

- Java da `unreachable statement, variable may not have been initialised , missing return statemet,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 dimensine 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, Package[4] e Moduli[5], 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   }
```

[4] Contenitori, gerarcici tra loro, di una decina di classi di alto livello con scopo comune
[5] Insieme di Package costituente un frammento di codice autonomo.
[6] Un 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. Esegiure 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 hai package autocostruiti delle directory del file system.

Di default il package di una unità di compilazione[7], ".java", è la root di gerarchia. Per definire il pakage 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.

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.

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

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

Introdotto come tipo risultato delle espressioni e condizioni.

- **Valori** true, false.
- **Operatori Unari** `! (not)`. Peratori binari: `& (and), | (or), ^ (xor), && (and-C), || (or-c)`[8].
- **Operatori confronto numerici** `<, >, <=, >=`.
- **Operatori di uguaglianza** `==, !=`. Con gli oggett 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.

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

5.2.2 Caratteri

Rappresentazione `'carattere'` o `'u<0-65535>'`. Formato ASCII o UTF16 (16bit). I caratteri d'escape si fanno `'\<specificatore>'`.

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 efficente). Se voglio un long va aggiunta una "l" dopo il numero.

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 efficente) 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 esperessioni e nell'assegnamento portando tutti 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 perdità di informazioni.

5.4 Array

Oggetti (variabili hanno riferimento nello heap) di lunghezza esplicita e acessibile, inacessibile 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(<tipo> <v>: <e>)`

- `v` è variabile del tipo indicato che vale via via ogni elemento della colezzione.
- `e` è un espressione che restituisce una collezione di elementi del tipo indicato.

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  [public | private | ""] class NomeDiUnaClasse {
2    // Campi
3    [public | private | ""] [static] [final] <tipo> <
         nomeCampo>;
4    ...
5    // Costruttore/i
6    [public | private | ""] NomeClasse([<parametri>]) {...}
7    ...
8    // Metodi
9    [public | private | ""] [static] <tipo> <nomeMetodo>([<
         parametri>]) {...}
10   ...
11 }
```

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**.

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

6.1.1 final e costanti

Modificatore (opzionale) 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 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 }
```

6.2.1 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**.

6.3 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.

6.3.1 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 }
```

6.4 this

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

6.5 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: `... nomeClase.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.

6.6 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.

6.7 Precisazioni

Inizializzazioni particolari degli oggetti Stringa:

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

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`[9] fa infierire[10] il tipo della variabile locale per allegerire il codice. Se manca l'espressione non va, esempio `var i;`.
- `altraVariabile` deve essere della stessa classe della variabile che sto definendo.

[9] Local variable type inference
[10] Far dedurre al compilatore il tipo della variabile locale dall'espressione assegnata.