Programmazione ad'Oggetti

Leonardo Mengozzi

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.

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

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

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

[3] Set di oggetti che si comunicano cosa fare.

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.

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].

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

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 il bytecode contenente 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):

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

- `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.

4.2.1 Compilazione ed'esecuzione Avanzata

Per tenere sorgenti e eseguibili separati, in progetti grandi si dispone la cartella *src* per i sorgenti (.java) e la cartella *bin* per i bytecode (.class).

```
javac -d "<dirExe>"[-cp "<lib1:|;...>"] <.../elencoSorgenti>

java -cp "<dirExe[lib1:|;...]>"<fullyClasseMain>
```

- `-d` specifica al compilatore la cartella dove creare i .class. Se non già presente la crea.
- `-cp` linka le classi dei percorsi specificati (divisi da : o ;) al classpath[8].
- La JVM si aspetta il FQCN (Fully-Qualified Class Name).

4.2.2 Gradle

Build system ibrido composto da:

- **Progetto** Cartella contenente i file *build.gradle.kts* e/o *settings.gradle.kts* (build files).
  1. Il build contiene la logica di costruzione del software.

     `plugins{java}`

     Io creo *src/main/java* per i .java e sarà creato *build/classes/java/-main* per i .class.
  2. Il settings contine la personalizzazione, come il nome del progetto.

     `rootProject.name="..."`
- **Plugin** Sofware contenente *task* pronti all'uso, come per java.
- **Task** Singola operazione atomica del processo di costrutzione del software. Una esecuzione gradle richiede una o più task di cui sono automaticamente dedotte le dipendenze.

  Comandi: gradle tasks, gradle tasks –all, gradle compileJava, gradle clean.

[8] Sottopercorso (insieme ordinato) di cartelle che corrisponde al percorso del package dichiarato per la classe corrispondente. Il classpath di default è il JRE (Java Runtime Environment).

L'esecuzione è delegata al compilatore del linguaggio specifico.

Wrapper: sofware scarica gradle e lo usa per creare il sofware. File del wrapper: gradle/wrapper/gradle-wrapper.jar (wrapper in s'è), gradle/wrapper/gradle-wrapper.properties (versione), ./gradlew oppure gradlew.bat (eseguono il wrapper).

Per compilare: `./gradlew compileJava`

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.
- 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 "SNAKE_CASE". Nomi di package in minuscolo con numeri e senza "_".
- 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

Introdotto come tipo risultato delle espressioni e condizioni.

- **Valori** true, false.
- **Operatori Unari** `! (not)`. Peratori binari: `& (and)`, `| (or)`, `^ (xor)`, `&& (and-C)`, `|| (or-c)`[9].
- **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.

[9] `& 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 tipo statico e a rum-time

- **Tipo Statico**: Tipo di un espressione desumibile dal compilatore.
- **Tipo run-time**: Tipo effettivamente presente è tipo statico o sottotipo.

Per ispezionare il tipo a run-time:

```
<espressione> instanceof <tipo>
```

Restituisce `boolean` e usato nella condizione di un `if` permette di specializzare esecuzione.

Downcast: Creare nuovo riferimento di tipo statico all'espressione (da sovraTipo a sottoTipo):

```
... (<tipoEffettio>)<espressione> ...
```

5.5 Autoboxing

Ogni tipo primitivo può essere "boxed" in un oggetto.

- **boxing** azione di inscatolamento di un tipo primitivo in un oggetto. `<Tipo> <variabile> = <Tipo>.valueOf(<tipoPrimitivo>)`
- **de-boxing** azione di estrarre un tipo primitivo da un ogetto. `... <varOggetto>.<tipoPrimitivo>Value()...`

Boxing e De-boxing sono effettuati in automatico.

5.6 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`.

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

```
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.

Per variabili di tipo classe non si può cambiare il riferimento all'oggetto, ma i campi dell'oggetto si.

```
<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 }
```

Nota: Classi POJO sono classi con solo campi privati, costruttori e metodi get e set.

6.3.1 Variable arguments

Per passare un numero indefinito di parametri di un certo tipo:

`<tipo> <nomeMetodo>([<parametri>], <tipo>... <nomeParametro>){...}`

- Va definito come ultimo o unico parametro.
- Nel corpo del metodo sarà trattato come un array di `<tipo>`.
- Il chiamante passa al metodo una lista di parametri di `<tipo>`.

6.3.2 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.

Nota: per fare una classe non istanziabile basta fare i costruttori `private`/`protected`.

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 accedento 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: `... 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.

6.7 Livelli d'accesso

Si antepongono a classi, metodi, campi, costruttori per definire il grado di utilizzo, in ordine crescente di libertà:

1. `private` Visibile e richiamabile solo nella classe di definizione.
2. `package` default (no keyword). Visibile e richiamabili dentro il package e invisibili fuori.
3. `protected` Visibile alla classe corrente e dalle sotto classi ricorsivamente. (in java anche al package).
4. `public` Visibile e richiamabile da qualunque classe.

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 contratto[10] di un oggetto.

6.7.1 Incapsulamento

Scopo: Avere controllo su come vengono usati i dati.

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**.

Cosi modifiche implementative non influenzano uso dell'utente. Controllo dati e riuduco possibilità errori lato utente. Maggior distacco tra dato e implementazione. (*Information hidding* = nascondere implementazione informazioni).

[10] Insieme degli scienari d'utilizzo e quindi aspettative di un utente al suo utilizzo.

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 dipendenze[11] 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 pacakge, classi e metodi.

Nota: Per rispettare l'incapsulamento quando si fa un get di un tipo non primitivo, bisogna restituire una copia del campo e non il campo stesso.

6.8 Precisazioni

Inizializzazioni particolari degli oggetti Stringa:

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

6.9 Fasi Implementazione

1. Progettazione della parte pubblica della classe (nome classe, signatura metodi e costruttori necesari).
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.

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

- `var`[12] fa infierire[13] 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.

8 Dipendenze

Il minimo numero di dipendenze fra classi sono manifestazione di "riuso" di codice e ne fanno un sistema e non un mero gruppo.

I tipi di dipendenze sono:

- **Associazione** (uses), un oggetto ne usa un altro.
- **Composizione** (has-a), un oggetto ne aggrega altri.

  Un oggetto di una classe *si compone di* (campi) un insieme di altri oggetti, di altre classi.

  Tale composizione può essere permanente (campo final) o opzionale (campo possibilmente `null`), multipla nota (più campi) o sconosciuta (campo array).
- **Specializzazione** (is-a), una classe ne specializza un'altra.

Nota: Nella composizione alla cancellazione dell'oggetto utilizzatore gli oggetti usati vengono cancellati a loro volta. Diversamente nell'aggregazione questi persistono.

8.1 Poliformismo Inclusivo (subtyping)

Fornire sovratipi che raccolgono classi uniformi tra loro. Gode del principio di sostituibilità utile a creare collezioni omogene.

Upcast: Per il principio di sostituibilità quando ci si aspetta A si può usare B ma solo con le cose di A e non quelle esclusive di B. Ciò perchè B risponde almeno a tutti i messaggi di A (hanno i medesimi contratti).

Con Interfacce Sia B classe sottotipo di A interfaccia, con medesimi metodi (contratto) e altro possibilmente. Facile far aderire una classe a un iterfaccia.

Con Classi Sia B sottotipo di A, con medesimi metodi e campi (contratto+comportamento) e altro possibilmente. B e A hanno un comportamente compatibile.

In memoria: Intestazione (16byte) con indicazioni a run-time sulla classe dell'oggetto, tabella puntatori per vtable, campi privati di Object. Poi campi classe partendo da quelli della super classe.

Poliformismo Parametrico (genericità): ???.

[12] Local variable type inference.

[13] Far dedurre al compilatore il tipo della variabile locale dall'espressione assegnata.

8.2 Interfacce

Sopra tipo con caratteristiche comuni. Le classi implementative sono sotto tipi dell'interfaccia.

Separa esplicitamente l'interfaccia (contratto, fisso) dalla realizzazione (implementazione, variabile) della classe.

Consente diverse realizzazioni di un contratto, usabili omogeneamente. (più classi implementano medesime funzioni e si vuole usarele come fossero la stessa).

```
interface <NomeInterfaccia> {...}
```

- Nuovo tipo solo dichiarabile (no `new NomeInterfaccia()`) con valore oggetti delle classi che implementano l'interfaccia.

  Principio sostituibilità di Liskov: A sottotipo di B, ogni oggetto A deve essere usabile dove ci si attende un oggetto B.

- Un oggetto tipo interfaccia consente solo chiamate ai metodi definiti dall'interfaccia e esegue l'implementazione specifica classe implementativa dell'interfaccia.

  ```
  <NomeInterfaccia> <nomeVar> = new <ClasseImplementativa>();
  ```

  Tipo statico è interfaccia e tipo run-time è tipo classe implementativa. **Late binding**: Codice da eseguire scelto dinamicamente, dipende da classe implementativa dell'interfaccia tipo della variabile.

- Include solo intestazioni di metodi. Sono automaticamente `public` e non si specifica.

- Compilato come una classe (produce un .class).

- Un interfaccia può estendere altre interfacce.

  ```
  interface <NomeInterfaccia> extends int1[,...] {...}
  ```

  Nel corpo dell'interfaccia si specificano solo i metodi che aggiunge.

```
class <NomeClasse> implements <NomeInterfaccia[,...]> {...}
```

Una classe può implementare più interfacce.

- Deve implementare il corpo di tutte le intetazioni di metodo delle interfacce (metodi in comune implementati una volta). La classe potrà avere altri metodi suoi.

- Le istanze avranno tipo la classe e le interfacce.

Nota: L'interfaccia è il sopra tipo/insieme delle classi, è più generale, ma fornisce meno funzionalità.

Nota: La classe che rispeccia l'interfaccia si chiama *impNnomeInterfaccia*.

8.3 Composizione/delegazione

Data una classe (di cui possono non avere i sorgenti) ne realizzo un'altra con caratteristiche solo in parte diverse, ovvero una specializzazione. Questa nuova classe potrebbe sostituire la precedente (gerarchia *is-a*).

Senzia violare **DRY**: Don't Repeat Yoursealf.

Nella nuova classe:

1. **Incapsulare** un oggetto della classe da modificare/estendere mediante campo `private final`[14]. Sarà inizializzato nel costruttore della nuova classe.
2. **Metodi** possono delegare all'oggetto incapsulato e essere di più o meno, ampliando cos'ì le funzionalità. Viola poco DRY.
3. **Combinare** con l'uso delle interfacce per ottenere la sostituibilità.

Quando si può è meglio usare i metodi "ridefiniti" che "originali" per rispettare l'incapsulamento.

8.4 Ereditarietà

Definisce una classse specializzandone una esistente **ereditando**. Rispetta DRY.

```
class <ClasseFiglio> extends <ClassePare> {...}
```

Ereditarietà singola (tree), una classe può estenderne solo un'altra.

- Eredita (usabili) campi, metodi, costruttori non `private`.
- Costruttori non privati del padre richiamti solo da costruttori figlio (c.Figlio¿=c.Padre), per inizializzare campi Padre.

  ```
  super(<parametri>)
  ```

  Se precesente deve essere la prima riga dei costruttori del figlio che devono avere almeno i parametri dei costruttori del padre.

  Costruttori default padre chiamato se `super()` assente/posizione errata.
- Figlio può aggiungere campi e metodi propri a quelli del pare (m.Figlio¿=m.Padre) essendone una specializzazione.
- Figlio acede a campi Padre (`private`) con get/set `protected` per rispettare incapsulamento.
- Classe Figlio vede la "catena" di ereditarietà come un unica sovra classe da cui eredità.

[14] Cosi che l'oggetto una volta inizializzato non possa essere cambiato ma si possa allo stesso tempo adoperare metodi e campi di conseguenza.

- Una classe col modificatore `final` non è ereditabile.

Nota: Bastano i binari per estendre una clsse.
Come le interfacce anche l'ereditarietà gode della sostituibilità. Posso fare variabili di tipo Padre ma che referenziano oggetti tipo Figlio. Questo perchè i Figli essondo delle specializzazioni "sono anche padre".

8.4.1 Overriding

Riscrizione di uno o più metodi `public` del Padre nel Figlio.

- Per sovrascrivere un metodo bisogna usare la stessa firma, a meno che del modificatore d'accesso che può solo aprirsi verso uno meno ristrettivo (fino a public).
- Un metodo definito col modificatore `final` non è Overridabile.
- Il metodo riscritto può invocare la versione Padre con `super.<metodoPadre()>`. Usabile non solo in caso di overdding.
- Un metodo ereditato o sovrascritto che usa `this` chiamerà la versione del metodo della classe chiamante (come ogni metodo).
- I metodi override hanno opzionalmente la notazione Override@ utile al programmatore e al compilatore per controlli di sintassi.

8.4.2 Vtable

Nomi: vtable, call table, dispatch table.
Riporta la classe con il body da associare a ogni metodo definito o ereditato di una classe.
Nella Memoria: Nello stack abbiamo le variabili che fanno riferimento a oggetti di una certa classe, memorizzati nella heap. Dentro la memoria di questi oggetti nella heap si hanno i riferimenti alle vtable definite per ogni classe. Queste tabelle per ogni metodo della classe dicono se usare la versione della classe corrente (metodo ex-novo o sovrascritto) oppure di una sovraclasse.

8.4.3 Classe Object

Una classe estende sempre indirettamente la classe `Object`. `Object` è la radice di ogni gerarchia d'ereditarietà.
Fornisce metodi di uso generico:

- **toString()** Restituisce una rappresentazione in stringa dell'oggetto (utile per il debug).Automaticamente chiamato dall'operatore "+" nelle concatenazioni tra stringhe e oggetti. Ogni classe deve sovrascriverlo.

Vantaggio? Fattorizza comportamento comune e crea funzionalità che lavorano su qualunque oggetto.

8.5 Classi Astratte

Classi dal comportamento parziale, fatte per raggruppare contratti+comportamenti comuni e opzionalmente contratti successivamente specializzati.

`abstract class ...`

- Fornisce codice comune a classi figlie che la specializzano.
- Metodi già implementati, spesso `final`, definiscono il comportamento comune.

  Metodi *astratti*, opzionali, hanno solo il contratto. Le classi figlio ne definiranno il comportamento specifico. Vanno definiti `abstract`: `abstract <tipo> <nomeMetodo>([<parametri>]);` Metodi astratti si definiscono solo in classi astratte.
- Possono contenere campi, costruttori.
- Classi non istanziabili (no `new`).
- Una classe astratta può specializzare un'altra classe astratta o un interfaccia senza l'obbligo di specializzarne i metodi.
- Una classe astratta può estendere una classe non astratta.
- Una classe non astratta è tenuta a specializzare i metodi astratti della classe astratta da cui eredità.

Intermedio tra ereditarietà tra classi e interfacce.

9 UML (Class Diagram)

Linguaggio grafico, OO-based, di modellazione di software.

- **Rettangoli**: un box rettangolare per classe o interfaccia diviso in:
  1. Nome classe. Nel caso delle interfacce aggiunge `<<interface>>` a sinistra del nome.
  2. Campi (solo per classi). Signatura: `nome : tipo`.
  3. Metodi e costruttori. Signatura: `nome(arg1:tipo,...):tipoRitorno`. Se `final` si indica l'attributo {leaf}

  \- indica `private`, + indica `public`, # indica `protected` e sottolineato indica `static`.

  Metodi e classi astratte si scrivono in corsivo.
- Dipendenze tra classi:

  - Composizione è arco con rombo.
  - Associazione è arco con freccia.
  - Generalizzazione è arco tratteggiato con triangolo vuoto, raggruppabile.
  - Estensione è arco con triangolo pieno, raggruppabile.
    Nella classe figlia si specificano solo le aggiunte e ovveride.

  L'arco può essere indicato con le molteplicità: 1,0, n, 0...1, 1...n.

- Spesso accompagnato da descrizione testuale.

Spesso si omette tutto ciò che non è "design" e le signature ed'alcune relazioni.