



Pattern Recognition  
and Applications Lab

# Le basi della Programmazione Orientata agli Oggetti

**Docente:** Ambra Demontis

**Anno Accademico:** 2022 - 2023

Corso di Laurea in Ingegneria Elettrica, Elettronica e Informatica



University of Cagliari,  
Italy

Department of Electrical and  
Electronic Engineering



# Il Paradigma di Programmazione Orientata agli Oggetti

Questo **paradigma** è **ispirato alla realtà**.

Problema: vogliamo accendere la televisione.

Soluzione: utilizziamo il telecomando per accendere il televisore.

Diversi problemi “pratici” si risolvono utilizzando oggetti, ognuno con le proprie caratteristiche, e facendoli interagire tra loro.

I linguaggi di programmazione ad oggetti modellano un programma come un insieme di oggetti che interagiscono tra loro.

# Oggetto

Entità con degli **attributi** che può, eventualmente, compiere azioni (**metodi**).  
E' una specifica **istanza** di classe e.g., della classe “telecomando auto”.



**attributi** {  
Altezza: 5 cm  
Larghezza: 3cm  
Spessore: 1 cm

**metodi** {  
-Inserisci antifurto  
-Rimuovi antifurto

# Classe

Definisce gli attributi e i metodi condivisi da un insieme di oggetti.

**Tipo di oggetto:** telecomando auto

## **Attributi:**

Altezza

Larghezza

Spessore

## **Metodi:**

Inserisci antifurto

Rimuovi antifurto

# Classe vs Oggetto

Diversi oggetti appartengono alla stessa classe se hanno gli stessi attributi e metodi.

Esempio di oggetti che appartengono alla stessa classe:



Hanno gli stessi attributi e gli stessi metodi.

Appartengono alla stessa classe ma sono oggetti differenti e lo sarebbero anche se fossero all'apparenza identici. Provate ad aprire una macchina con il telecomando di un'altra macchina dello stesso modello...

# Le Fasi di Sviluppo di un Programma

- 1. Object-Oriented Analysis:** si ragiona sul problema da risolvere e si *identificano gli oggetti*, le loro proprietà e le interazioni tra gli stessi
- 2. Object-Oriented Design:** si crea un *prototipo*, cioè si schematizza l'idea derivante dall'analisi al passo precedente
- 3. Object-Oriented Programming:** si implementa il *prototipo* nel linguaggio di programmazione orientato agli oggetti scelto

# Descrivere gli Oggetti: I Diagrammi di Classe

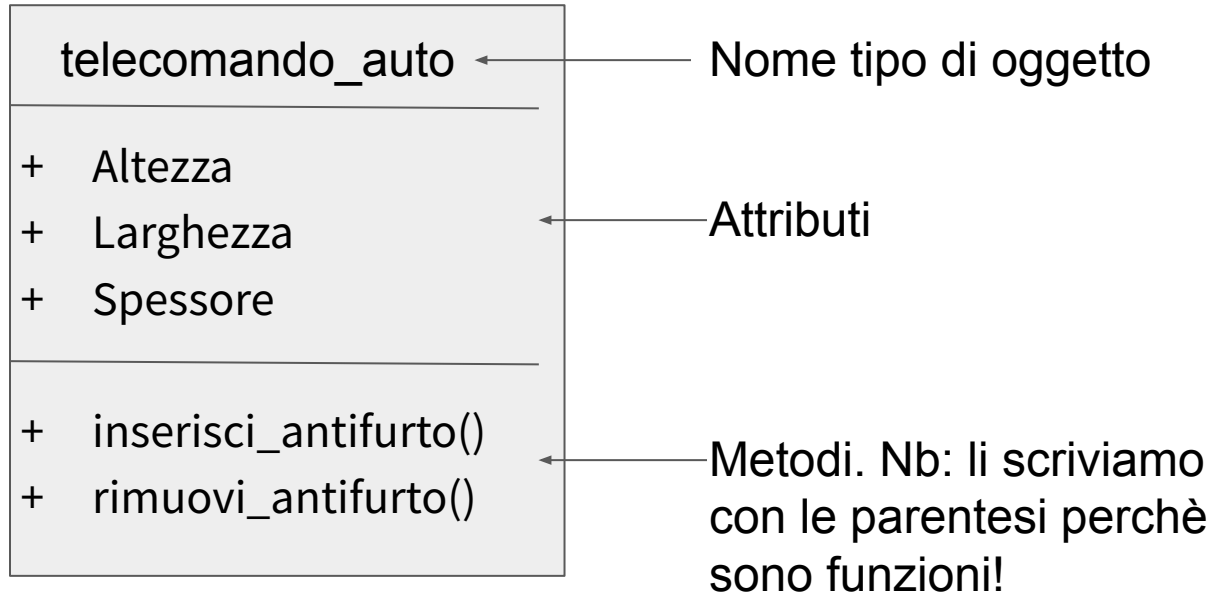
Per schematizzare il prototipo di un programma utilizzeremo i **diagrammi di classe** nel linguaggio “Unified Modeling Language” (UML).

I diagrammi di classe **descrivono le classi degli oggetti che compongono il sistema e le relazioni tra essi.**

Nb: in queste slide vedremo la notazione mano a mano che ci servirà e verrà poi riepilogata alla fine.

# Rappresentare una Classe

Rappresentiamo la classe *telecomando auto* con un [diagramma di classe](#).

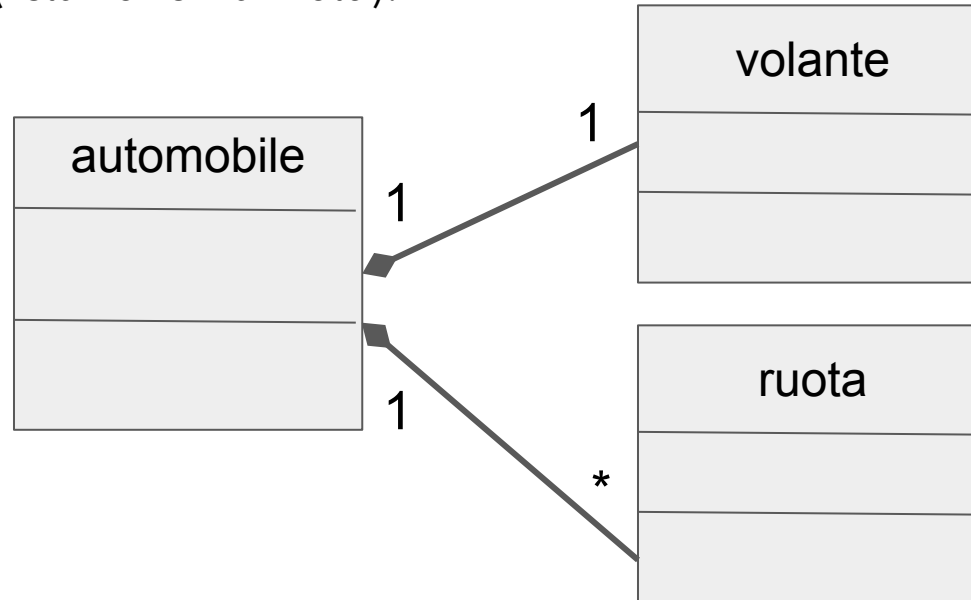




# Le Relazioni tra Oggetti: Composizione

Alcuni oggetti possono essere composti da altri oggetti.

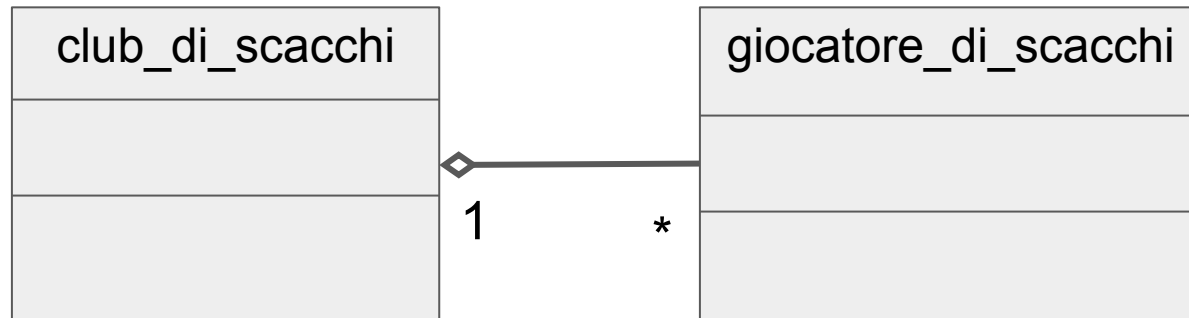
Ad esempio, un'*automobile* è composta da un *volante* (relazione 1 a 1) e da diverse *ruote* (relazione 1 a molti).



Nb: \* = molti

# Le Relazioni tra Oggetti: Aggregazione

Esistono degli oggetti che aggregano altri oggetti che hanno una vita propria.



# Composizione vs Aggregazione

**Composizione:** Se distruggo l'oggetto composto (e.g., la *macchina*) allora **distruggo anche gli oggetti che lo compongono** (*volante e ruote*).

Si dice che è una relazione “forte”.



**Aggregazione:** Se distruggo l'oggetto che aggrega altri oggetti (e.g., il *club\_di\_scacchi*) **non distruggo gli oggetti che sono aggregati** da quell'oggetto (gli oggetti di classe *giocatore\_di\_scacchi*).

Si dice che è una relazione “debole”.



# Le Relazioni tra Oggetti: Ereditarietà

Gli esseri umani ereditano le loro caratteristiche dai loro antenati.  
Anche tra oggetti esiste l'ereditarietà!

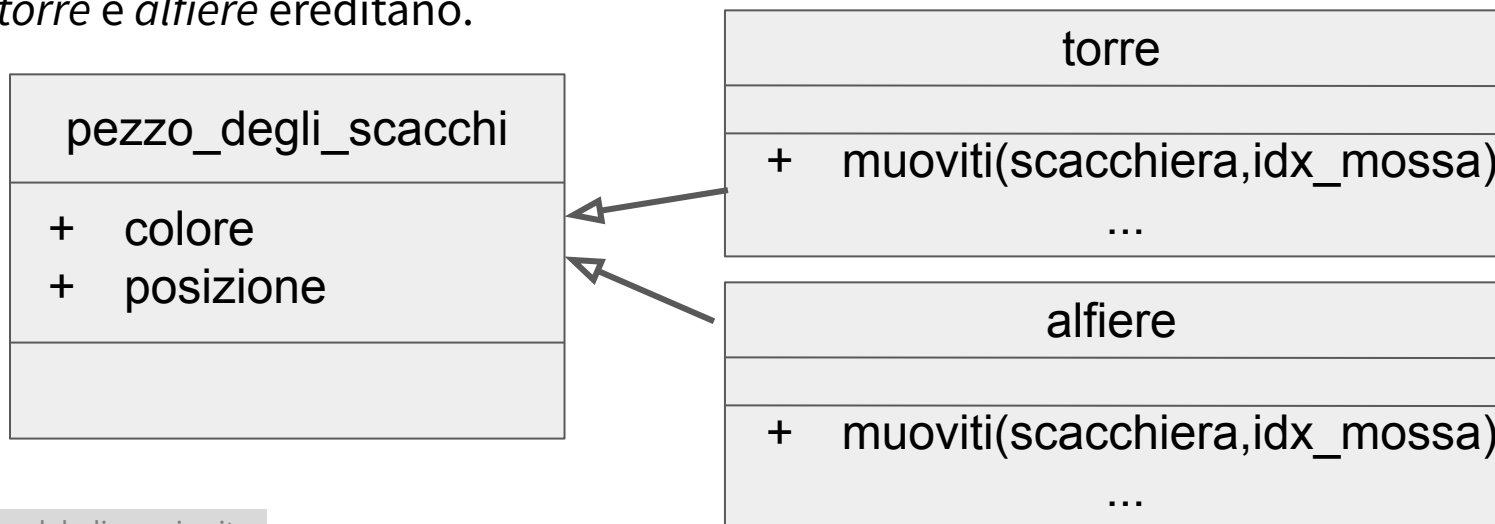
Questo ci **permette di evitare la duplicazione di codice** nelle classi coinvolte in questa relazione.

# Le Relazioni tra Oggetti: Ereditarietà



Un esempio di ereditarietà. I pezzi degli scacchi *torre* e *alfiere* hanno entrambi la caratteristica *colore* (bianco o nero) e entrambi possono effettuare mosse, ma le mosse che possono effettuare sono differenti.

Avremo quindi una classe generica *pezzo\_degli\_scacchi* dalla quale le classi *torre* e *alfiere* ereditano.



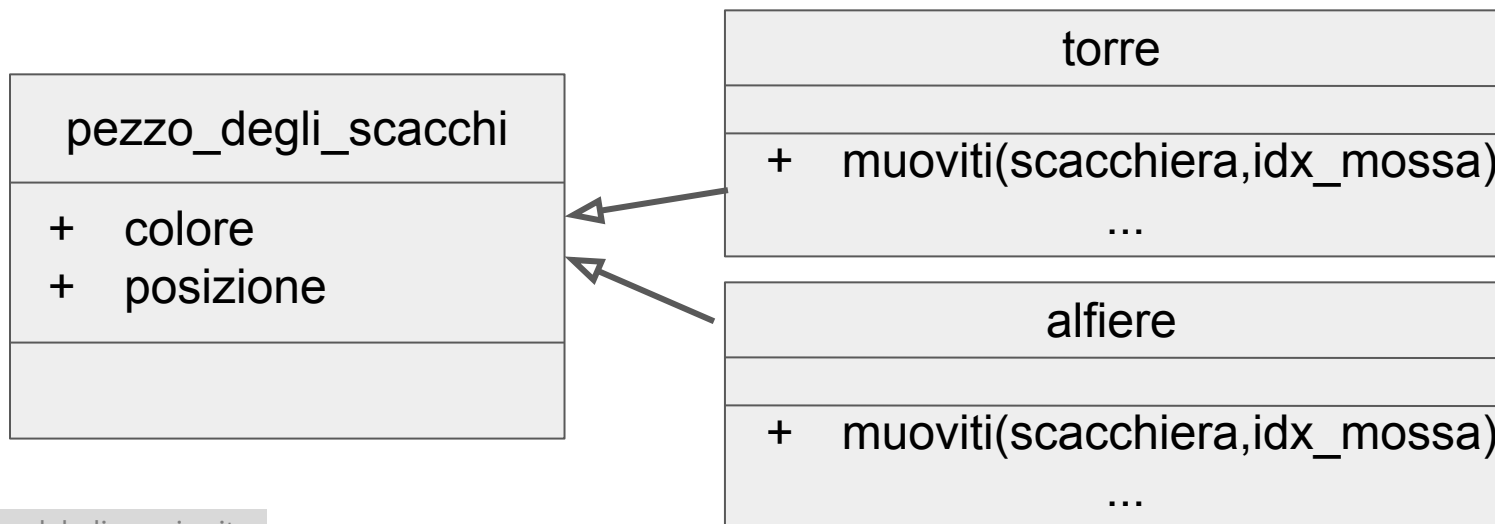
# Le Relazioni tra Oggetti: Ereditarietà



La classe pezzo degli scacchi si dice classe **padre** e torre e alfiere sono i **figli**.

I figli ereditano tutto dal padre.

Es: negli oggetti *torre* e *alfiere* non specifichiamo gli attributi *colore* e *posizione* in quanto li ereditano dalla classe padre.

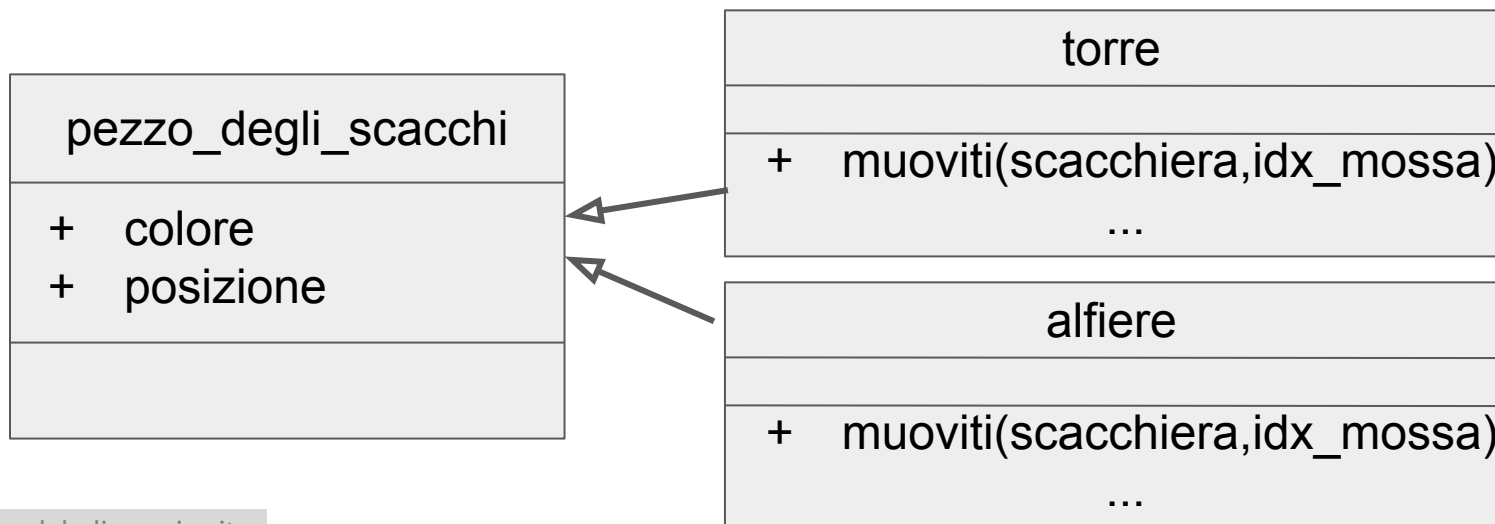


# Le Relazioni tra Oggetti: Ereditarietà



I figli ereditano tutto dal padre.

Questo ci permette di evitare di duplicare codice nelle classi coinvolte nell'ereditarietà.

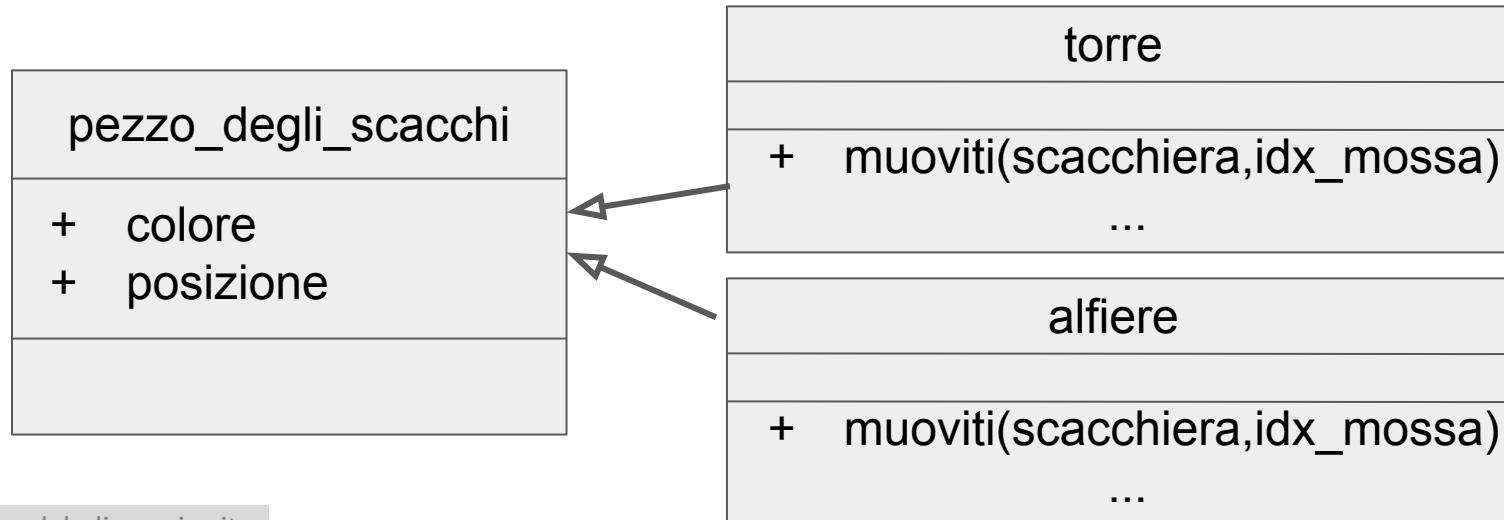


# Interfacce Comuni per Oggetti Differenti



Il metodo *muoviti* la definiamo nelle classi *torre* e *alfiere* in quanto non tutti i pezzi degli scacchi si muovono nello stesso modo.

Tuttavia per ridurre il codice che le classi utilizzatrici dovranno scrivere, le definiamo con la stessa **interfaccia** (nome del metodo e dei parametri).





# Interfacce Comuni per Oggetti Differenti



Vediamo lo pseudocodice (la descrizione a parole del codice) del metodo *muoviti* della classe *torre*:

funzione *muoviti*(scacchiera, idx\_mossa):

    se idx\_mossa è 1

*muoviti\_a\_destra*(scacchiera)

    se idx\_mossa è 2

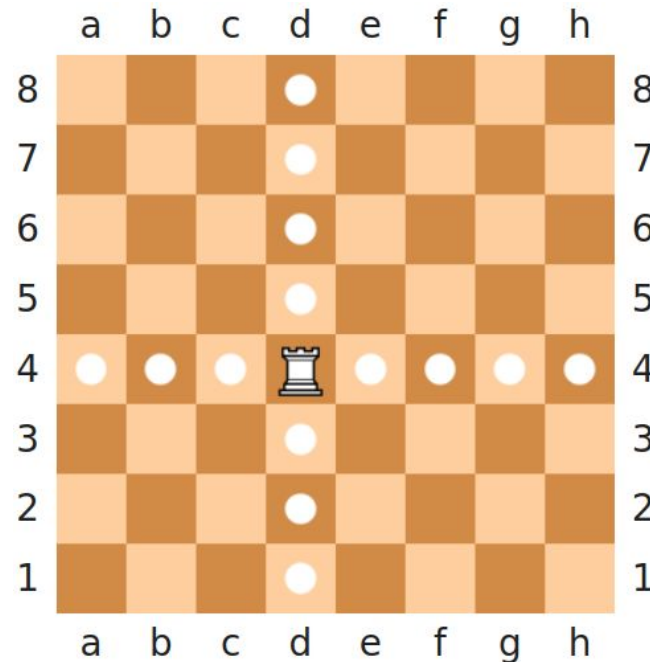
*muoviti\_a\_sinistra*(scacchiera)

    se idx\_mossa è 3

*muoviti\_in\_alto*(scacchiera)

    se idx\_mossa è 4

*muoviti\_in\_basso*(scacchiera)

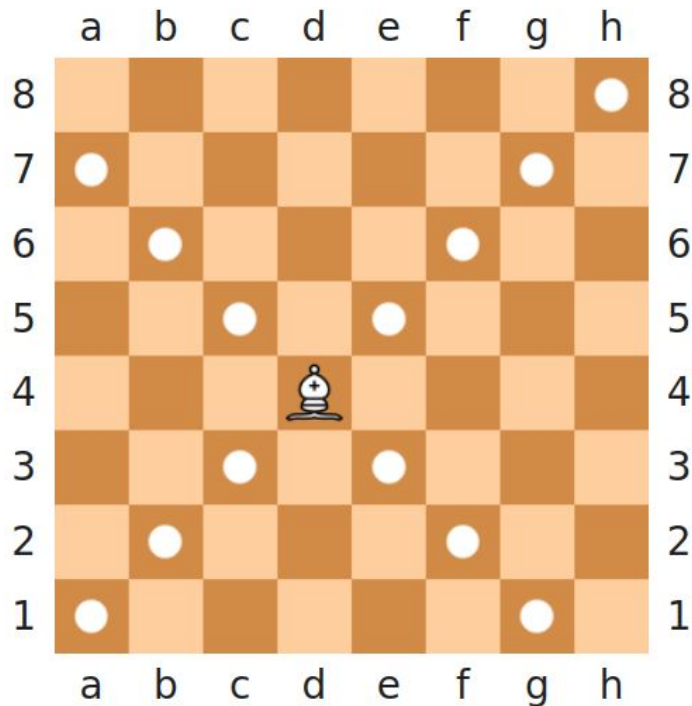


# Interfacce Comuni per Oggetti Differenti



Vediamo lo pseudocodice del metodo *muoviti* della classe *alfiere*:

```
funzione muoviti(scacchiera, idx_mossa):  
    se idx_mossa è 1  
        muoviti_a_nord-est(scacchiera)  
    se idx_mossa è 2  
        muoviti_a_nord-ovest(scacchiera)  
    se idx_mossa è 3  
        muoviti_a_sud-est(scacchiera)  
    se idx_mossa è 4  
        muoviti_a_sud-ovest(scacchiera)
```



# Interfacce Comuni per Oggetti Differenti



Questo ci permette di semplificare il codice nelle classi che utilizzano gli oggetti figli *torre* e *alfiere*.

Proviamo a modellare la classe *giocatore\_di\_scacchi* che usa i pezzi degli scacchi.

giocatore_di_scacchi
+ effettua_mossa(scacchiera)

# Interfacce Comuni per Oggetti Differenti



funzione `effettua_mossa(scacchiera)`:

`pezzo = scegli_pezzo_da_muovere(scacchiera)`

`idx_mossa = scegli_mossa_da_effettuare(scacchiera)`

`pezzo.muoviti(scacchiera, idx_mossa)`

...

Chiamata la funzione `muoviti` di un oggetto appartenente alla classe `pezzo` degli scacchi.

Il fatto che l'interfaccia del metodo che muove il pezzo sia uguale per tutte le classi figlie di *pezzo\_degli\_scacchi* semplifica il codice del metodo *effettua\_mossa* della classe *giocatore\_di\_scacchi*!

Non deve prevedere diversi casi a seconda del pezzo che si vuole muovere!

# Attributi e Funzioni Pubblici e Privati

Attributi e funzioni possono essere pubblici o privati.

Se sono **pubblici** sono visibili dagli altri oggetti.

Se sono **privati** non sono visibili dagli altri oggetti.

# Attributi e Funzioni Pubblici

Vediamo le classi *torre* e *alfiere* con tutte le loro funzioni.

Prima abbiamo evidenziato il metodo *muoviti* che viene utilizzato dagli oggetti di classe *giocatore\_di\_scacchi* che utilizzano queste classi.

torre
+ muoviti(scacchiera,idx_mossa)
...

alfiere
+ muoviti(scacchiera,idx_mossa)
...

Questo metodo deve essere quindi pubblico.

# Attributi e Funzioni Privati

Come abbiamo visto dallo pseudocodice però, la classe *torre* e *alfiere* hanno anche dei metodi che utilizzano per effettuare il metodo *muoviti*.

Queste **funzioni** sono **differenti a seconda della classe** (torre o alfiere) **e non vengono utilizzate dall'oggetto** *giocatore\_di\_scacchi*.

# Attributi e Funzioni Pubblici e Privati

torre
+ muoviti(scacchiera,idx_mossa) - muoviti_a_destra(scacchiera) - muoviti_a_sinistra(scacchiera) - muoviti_in_alto(scacchiera) - muoviti_in_basso(scacchiera)

alfiere
+ muoviti(scacchiera,idx_mossa) - muoviti_a_nord-est(scacchiera) - muoviti_a_nord-ovest(scacchiera) - muoviti_a_sud-est(scacchiera) - muoviti_a_sud-ovest(scacchiera)

Gli attributi e i metodi:

- **Pubblici**, vengono indicati con (+)
- **Privati**, vengono indicati con (-)

Il fatto che siano pubblici o private viene chiamato **visibilità**.



# Incapsulamento

Il fatto di “nascondere” attributi e funzioni utili alla classe stessa ma che non sono necessari ad altre classi per utilizzare l’oggetto viene detto **incapsulamento**.

Perchè è utile?

- Per chi vuole utilizzare la classe è subito chiaro quali funzioni è previsto utilizzi e quindi quali possono essere di suo interesse.
- Si evita che gli altri oggetti vadano a modificare il valore di quegli attributi compromettendo il funzionamento del programma

## Esempi

# Il Programma per la Gestione di una Libreria

Supponiamo di essere un programmatore al quale il gestore di una libreria ha chiesto di creare un programma gestionale.

Il programma deve:

- permettere alla libreria di memorizzare i dati di tutti i libri che possiede
- mostrare il prezzo di vendita di un libro considerando che la libreria ha deciso di applicare, a tutti i libri pubblicati da più di 5 anni uno sconto del 5%.

# Object-Oriented Analysis

Abbiamo un problema da risolvere con tecniche di OOP.

Come abbiamo detto precedentemente il primo passo è capire quali classi ci servono e quali devono essere le loro proprietà e i loro metodi.

# Quali Classi ci servono?

# Quali Classi ci servono?

- 1) dobbiamo memorizzare i dati di ogni libro, memorizzare il prezzo iniziale e calcolare il prezzo scontato, quindi ci servirà una classe *libro*
- 1) Dobbiamo memorizzare e cercare i dati di più libri quindi ci servirà una classe che raccolga gli oggetti di classe libro es. la classe *lista\_libri*

# La Classe Libro

Deve permetterci di memorizzare i dati del libro...

libro
+ titolo
+ autore
+ Editore
+ prezzo
+ anno di pubblicazione

# La Classe Libro

Deve anche permetterci di calcolare il prezzo scontato del libro.

libro
+ titolo
+ autore
+ editore
+ prezzo
+ anno di pubblicazione
+ calcola_prezzo_scontato()

Per poter calcolare il prezzo scontato, dobbiamo calcolare l'età del libro e calcolare lo sconto. E' quindi comodo creare due funzioni private...



# La Classe Libro

Deve anche permetterci di calcolare il prezzo scontato del libro.

libro
+ titolo
+ autore
+ editore
+ prezzo
+ anno di pubblicazione
+ calcola_prezzo_scontato()
- calcola_eta_libro()
- calcola_sconto()

# Lista Libri

Deve permetterci di memorizzare e cercare i libri.

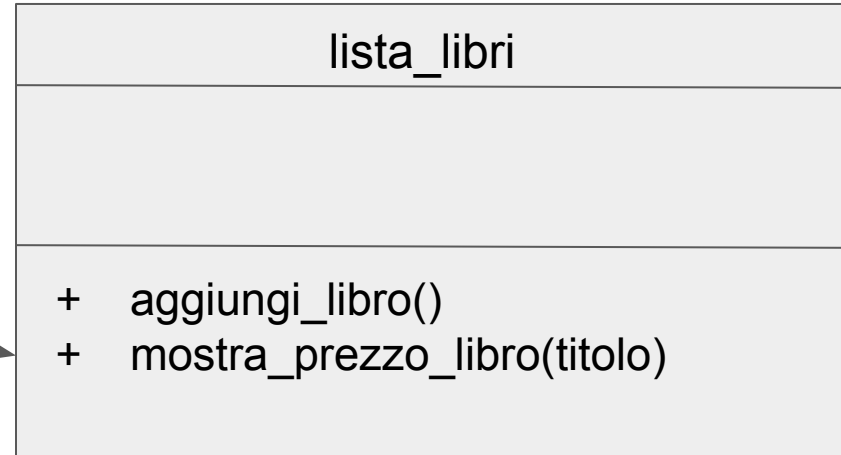
Crea un oggetto di tipo libro  
E lo memorizza



# Lista Libri

Deve permetterci di memorizzare e cercare i libri.

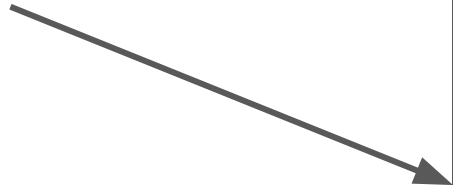
Metodo pubblico che cerca il libro, calcola il prezzo scontato (richiamando l'apposito metodo della classe libro) e mostra a schermo il prezzo del libro



# Lista Libri

Deve permetterci di memorizzare e cercare i libri.

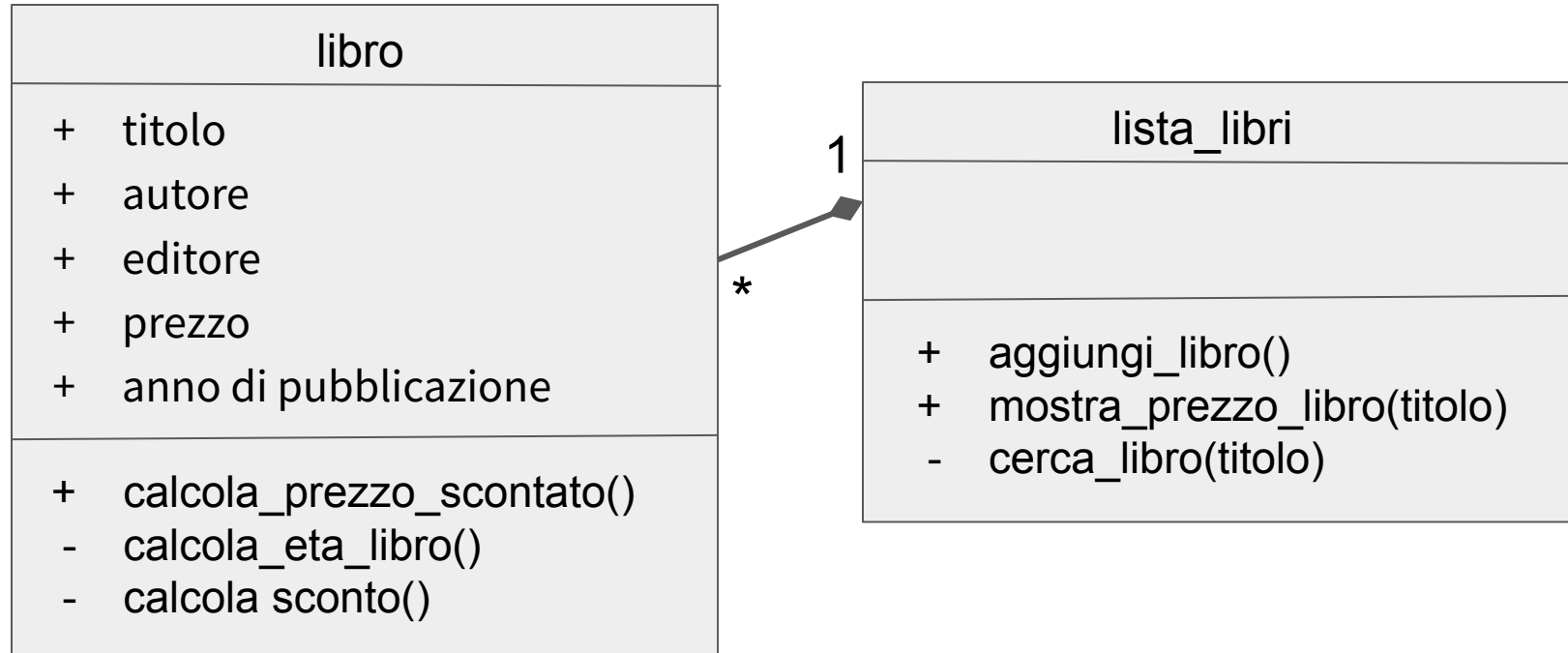
Metodo privato che dato un titolo cerca un libro e lo restituisce (utilizzata dal metodo pubblico *mostra\_prezzo\_libro*)



lista_libri
<ul style="list-style-type: none"><li>+ aggiungi_libro()</li><li>+ mostra_prezzo_libro(titolo)</li><li>- cerca_libro(titolo)</li></ul>




# Lista Libri

La lista libri è composta da oggetti di classe libro.



# Riepilogo Notazione

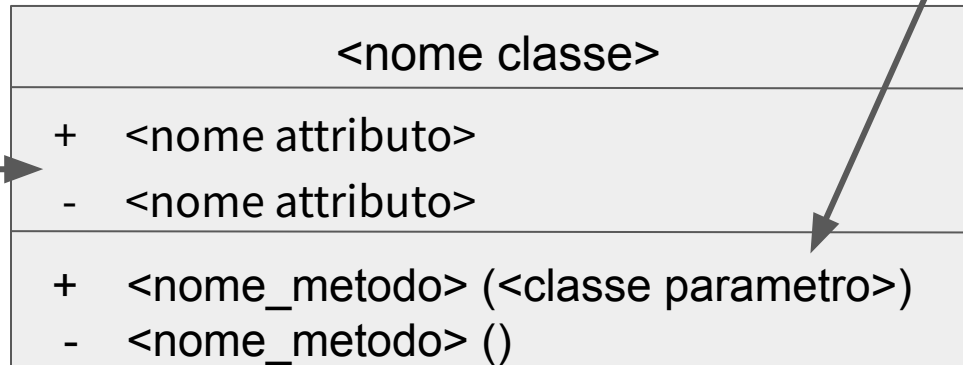
## Relazioni:

- Composizione 
- Aggregazione 
- Ereditarietà 

## Rappresentazione oggetti:

Visibilità

- + pubblico
- privato



Metodo che riceve  
un parametro

# Riepilogo Notazione

Volendo è possibile specificare anche i tipi di dato.

Tipo di dato della  
proprietà es:  
+ nome: stringa

<nome classe>	
+ <nome proprietà>: tipo_dato	
- <nome proprietà>: tipo_dato	
+ <nome_metodo> ():tipo_dato	
- <nome_metodo> (<parametro>: tipo_parametro): tipo_dato	

Tipo di dato restituito  
dal metodo.  
**void** se non  
restituisce nulla

# Esercizi



# Voto Finale Esami (versione 1)

Progettiamo un programma che permetta ad uno studente di memorizzare, per ogni esame, i voti presi in due esami parziali e che metta a disposizione un metodo che permetta di mostrare, per ognuno degli esami, il voto finale, che è dato dalla media aritmetica dei voti conseguiti nei due parziali.

# Di quali Oggetti Abbiamo Bisogno?

# Di quali Oggetti Abbiamo Bisogno?

- 1) Un oggetto che contenga i voti di un esame
- 2) Un oggetto che ci permetta di memorizzare i voti di tanti esami

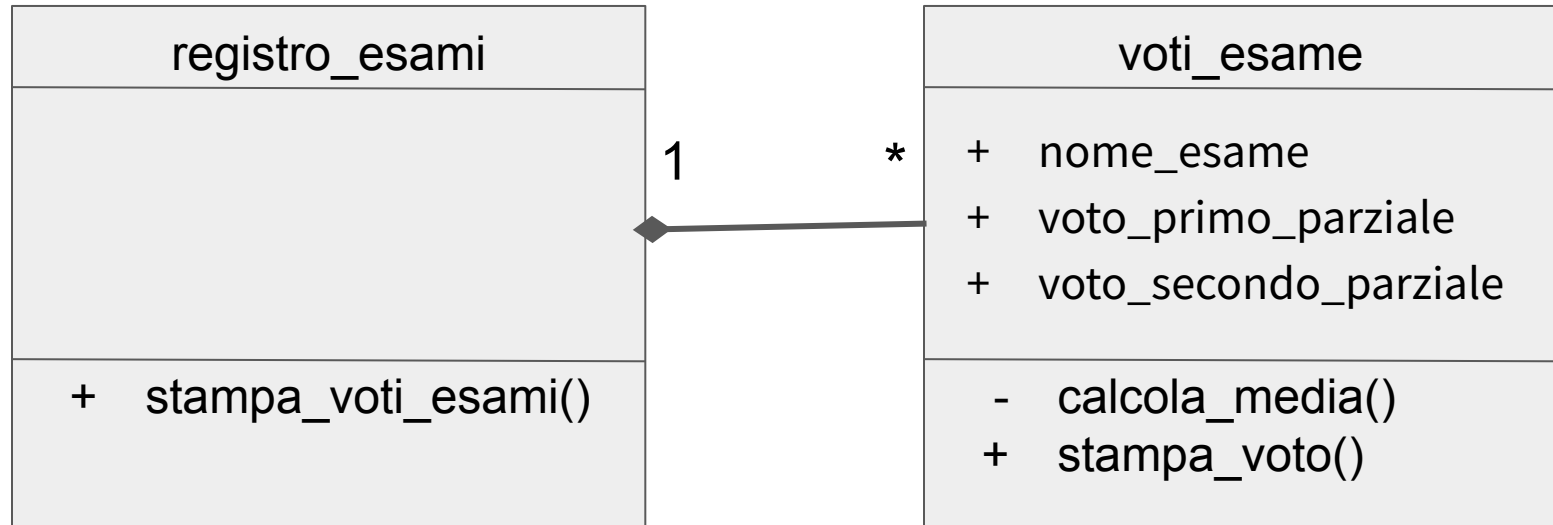
# La classe `Voti_esame`

<code>voti_esame</code>
<ul style="list-style-type: none"><li>+ <code>nome_esame</code></li><li>+ <code>voto_primo_parziale</code></li><li>+ <code>voto_secondo_parziale</code></li></ul>
<ul style="list-style-type: none"><li>- <code>calcola_media()</code></li><li>+ <code>stampa_voto()</code></li></ul>

## La classe Lista\_voti

registro_esami
+ stampa_voti_esami()

# La Relazione tra le classi



## Voto Finale Esami (versione 2)

Progettiamo un programma che permetta ad uno studente di memorizzare, per ogni esame, i voti presi in due esami parziali. Il programma deve mettere a disposizione inoltre un metodo che permetta di mostrare, la media aritmetica (o pesata) ottenuta in tutti gli esami. Il voto finale di un singolo esame supponiamo sia dato dalla media aritmetica dei voti conseguiti nei due parziali.

# La classe `Voti_esame`

<code>voti_esame</code>
<ul style="list-style-type: none"><li>+ <code>nome_esame</code></li><li>+ <code>voto_primo_parziale</code></li><li>+ <code>voto_secondo_parziale</code></li></ul>
<ul style="list-style-type: none"><li>+ <code>calcola_media()</code></li></ul>



## La classe Lista\_voti

registro_esami
+ stampa_media_voti_ esami()

# La Relazione tra le classi

