Politecnico
di Milano

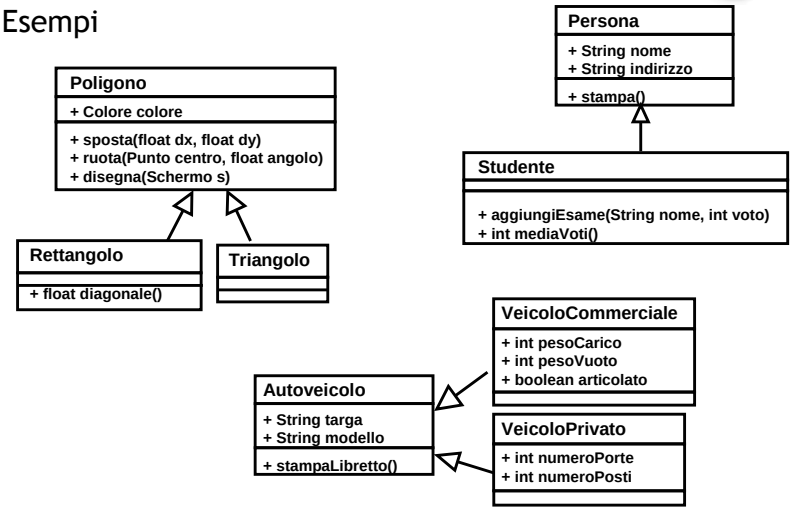
Il meccanismo dell'ereditarietà nei linguaggi orientati agli oggetti

Ereditarietà (1)

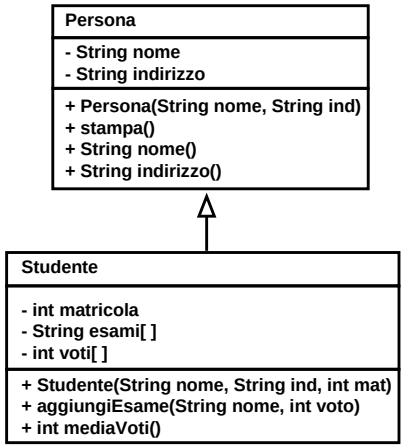
- Permette di definire una nuova classe (*classe erede*, *sottoclasse*) in funzione di una o più altre classi (*classi padre*, *super-classi*) specificandone solo le differenze
- La classe erede è un sottotipo, una specializzazione, della (delle) classe(i) padre
- L'ereditarietà genera gerarchie di classi

Ereditarietà (2)

- Esempi



Ereditarietà (3)

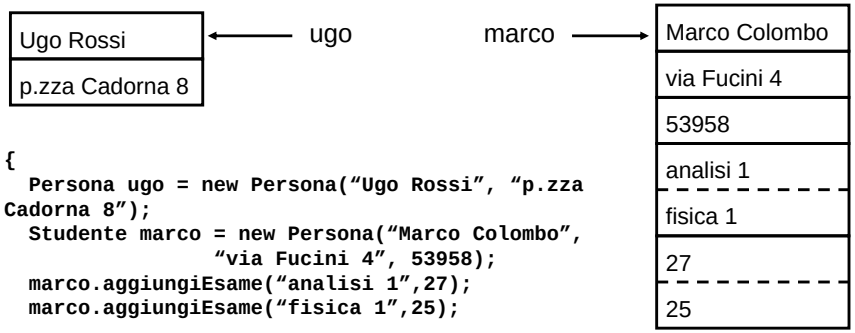


la classe Studente eredita dal padre:
- attributi
- metodi

Un oggetto Studente può essere trattato esattamente come un oggetto Persona

In cosa solitamente può differenziarsi la classe erede
- aggiunta di attributi e metodi
- i metodi possono essere ridefiniti

Ereditarietà (4)



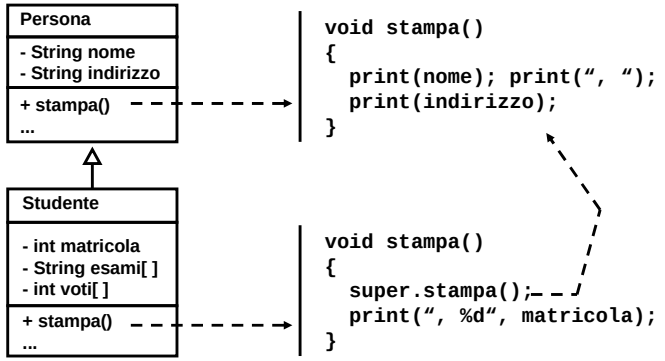

```
{
  Persona ugo = new Persona("Ugo Rossi", "p.zza
Cadorna 8");
  Studente marco = new Persona("Marco Colombo",
                "via Fucini 4", 53958);
  marco.aggiungiEsame("analisi 1",27);
  marco.aggiungiEsame("fisica 1",25);

  ugo.stampa(); print("\n");
  marco.stampa();
}
```

OUTPUT:

```
Ugo Rossi, p.zza Cadorna 8
Marco Colombo, via Fucini 4
```

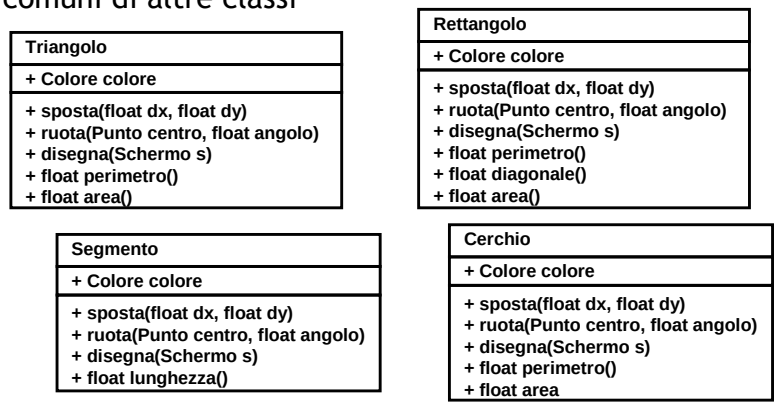
Ereditarietà (5)

- ridefinizione del metodo "stampa"
- L'implementazione dei metodi caratteristici di una classe erede (nuovi metodi o ridefinizione di metodi ereditati) può richiamare esplicitamente i metodi dei padri



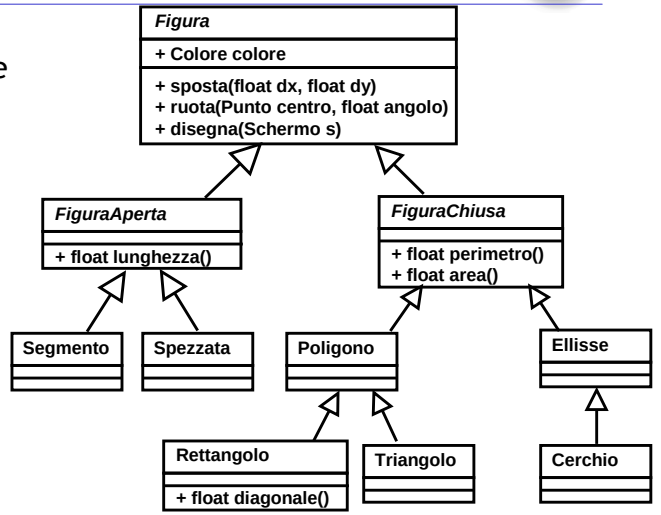
Vantaggi dell'ereditarietà (1)

- l'ereditarietà promuove l'astrazione: il programmatore è indotto a specificare classi che catturano gli aspetti comuni di altre classi



Gerarchia delle figure

- Il programmatore può scrivere parti di codice che trattano tutte le figure allo stesso modo.

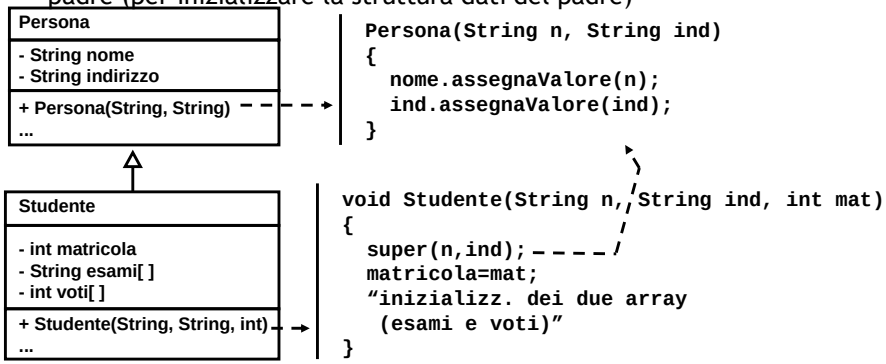


Vantaggi dell'ereditarietà (2)

- Promuove il riuso di classi esistenti
  - il programmatore ha la possibilità di trasformare le caratteristiche di una classe SENZA alterarne il codice originale, ma definendo un'opportuna sottoclasse

Inizializzazione

- Il costruttore non viene ereditato: la sottoclasse deve sempre definirlo
- L'erede deve sempre richiamare come prima cosa il costruttore del padre (per inizializzare la struttura dati del padre)



Ereditarietà ed information hiding

- Il codice che implementa i metodi della classe erede vede i segreti dei padri?
  - in caso affermativo sarebbe violata l'information hiding
  - in caso negativo i metodi nuovi (o ridefiniti) avrebbero possibilità limitate
- La soluzione adottata in genere dai linguaggi è quella di introdurre un altro livello di visibilità e le seguenti regole:
  - public - accessibile a tutti (client, classe stessa, classi erede)
  - protected - accessibile solo alla classe stessa ed agli eredi
  - private - accessibile solo alla classe stessa

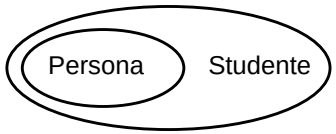
Le due interfacce di una classe

- grazie all'ereditarietà gli utilizzatori di una classe sono due:
  - i moduli "client esterni"
  - le classi erede
- a ciascun utilizzatore corrisponde una interfaccia diversa:
  - "client esterni": l'insieme dei metodi ed attributi pubblici
  - eredi: campi pubblici e protetti
- affinché un modulo sia realmente riutilizzabile è importante che entrambe le interfacce siano studiate accuratamente
- regola generale: nascondere il più possibile

Ereditarietà ed interfacce (1)

Legame tra interfaccia di una classe e quella dei propri eredi

- Le regole enunciate in precedenza prevedono che l'interfaccia del padre sia "contenuta" in quella del figlio
  - l'erede può essere trattato come il padre (viene ereditato il contratto tra classe e clienti esterni)

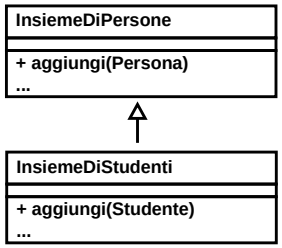


- Alcuni linguaggi seguono una filosofia differente

Ereditarietà ed interfacce (2)

Ridefinizione dei metodi, esistono diverse politiche:

- il metodo ha la medesima interfaccia (tipo parametri e tipo del valore di ritorno) di quello che ridefinisce
- i parametri possono essere sottoclasse di quelli del metodo originale (covarianza)
- ...



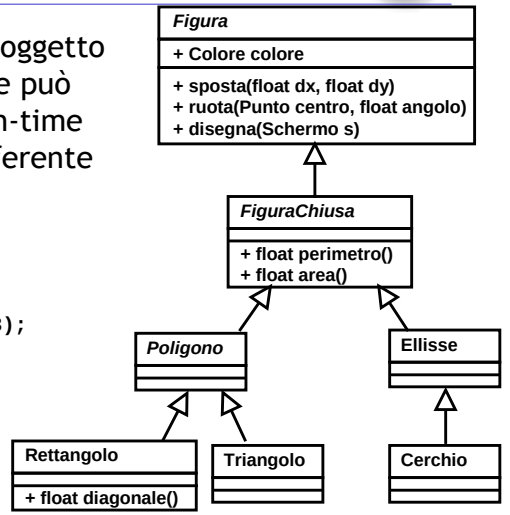
Classi astratte

- una classe è astratta quando non fornisce una completa implementazione della propria interfaccia
- una classe astratta non può essere istanziata
- le classi astratte servono come punto di partenza per la costruzione di altre classi
- una classe astratta può avere tutti i metodi non implementati
  - serve a specificare l'interfaccia comune ad un insieme di classi

Polimorfismo (1)

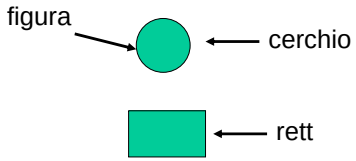
- un riferimento ad un oggetto è detto polimorfico se può essere associato a run-time ad oggetti di tipo differente
- esempio:

```
...
Figura figura;
Rettangolo rett(12,2,5,6,8,33);
Cerchio cerchio(8,9,12);

figura=cerchio;
figura.disegna(schermo);
figura=rett;
figura.disegna(schermo);
```



Polimorfismo (2)

```
...
Figura figura;
Rettangolo rett(12,2,5,6,8,33);
Cerchio cerchio(8,9,12);

figura=cerchio;
figura.disegna(schermo);

figura=rett;
figura.disegna(schermo);

float f=figura.diagonale();
```



- un riferimento polimorfico ha
  - *un tipo statico*
  - *un tipo dinamico* - è il tipo dell'oggetto associato
- controllo statico dei tipi e polimorfismo
  - idea:
    - attraverso un riferimento polimorfico permettere l'invocazione di tutti i metodi definiti dal suo tipo statico
    - garantire che il tipo dinamico sia sempre coincidente con quello statico oppure una sua specializzazione (sottotipo)
  - come: controllo degli assegnamenti

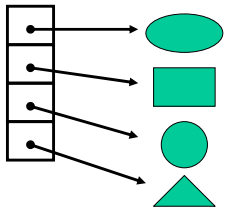
Esempio polimorfismo (1)

Immagine
- Figure figure[ ] - int numFigure
+ inserisci(Figura f) + inserisci(Figura f, int posizione) + rimuovi(Figura f) + rimuoviTutto() + disegna(Schermo s) + int numeroElementi() + Figura elemento(int posizione)

```
Cerchio c(...);
Ellisse e(...);
Rettangolo r(...);
Triangolo t(...);

Immagine imm(...);
imm.inserisci(e); imm.inserisci(r);
imm.inserisci(c); imm.inserisci(t);
```

modulo client

- Immagine è una sequenza ordinata di figure
- Immagine è una struttura dati polimorfica: le sue istanze possono contenere oggetti di tipo diverso



Esempio polimorfismo (2)

- "Immagine" non conosce i dettagli dei diversi tipi di figure (rettangoli, cerchi, triangoli…)
- La classe "Immagine" sa che TUTTE le figure sono in grado di rispondere all'invocazione dei metodi specificati nella classe "Figura"

Immagine
- Figure figure[ ] - int numFigure
+ inserisci(Figura f) + inserisci(Figura f, int posizione) + rimuovi(Figura f) + rimuoviTutto() + disegna(Schermo s) + int numeroElementi() + Figura elemento(int posizione)

```
void disegna(Schermo s)
{
  for(int i=0; i<numFigure; i++)
    figure[i].disegna(s);
}
```

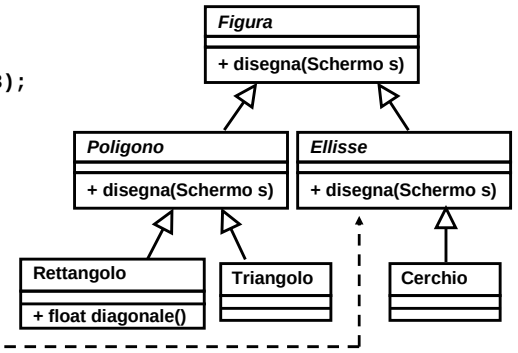
- Il metodo "disegna" funziona correttamente?

Binding dinamico (1)

- Quando viene chiamato un metodo attraverso un riferimento polimorfico, il metodo effettivamente chiamato è quello definito dal tipo dinamico

```
...
Figura figura;
Rettangolo rett(12,2,5,6,8,33);
Ellisse ellisse(8,9,12,4);

figura=ellisse;
figura.disegna(schermo);
...
```

viene chiamato il metodo "stampa" definito nella classe "Ellisse"

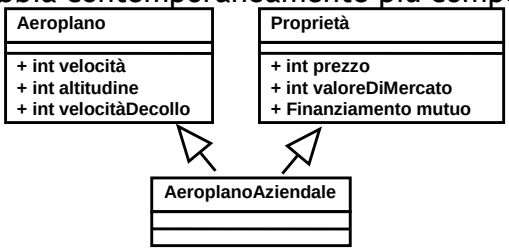


Binding dinamico (2)

- il binding è dinamico nel senso che non viene risolto dal compilatore ma a run-time dal'ambiente che supporta l'esecuzione dei programmi
- ereditarietà+polimorfismo+binding dinamico = codice generico, "facilmente" comprensibile e modificabile
  - si può interagire con gli oggetti ai diversi livelli di astrazione

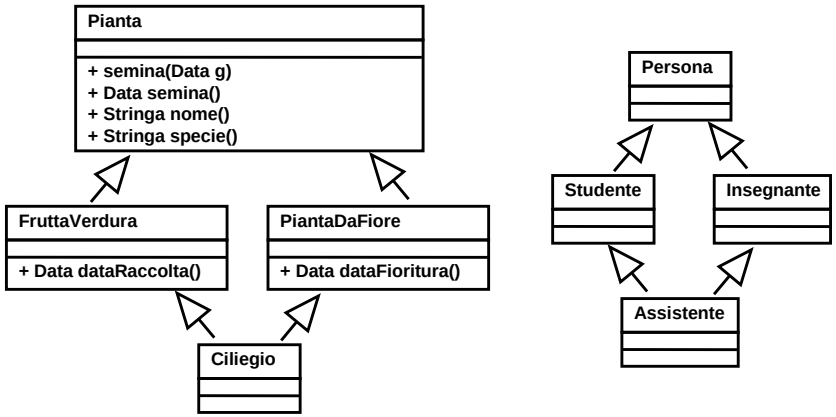
Ereditarietà multipla

- attraverso l'ereditarietà una sottoclasse eredita dalla sua superclasse:
  - contratto
  - implementazione
- l'ereditarietà multipla è utile quando si vuole che una classe abbia contemporaneamente più comportamenti



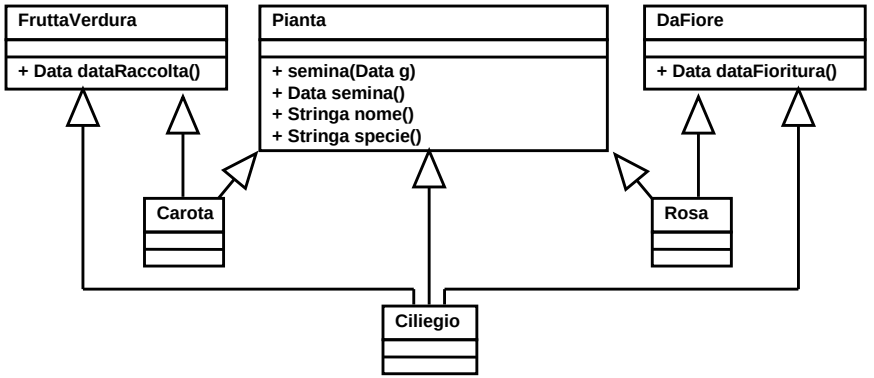
Problemi dell'ereditarietà multipla

- Le implementazioni possono entrare in conflitto



Una possibile soluzione

classi "Mix-in" (Grady Booch)



Approcci all'ereditarietà multipla

opinioni contrastanti sulla ereditarietà multipla

- complessa, misteriosa, error-prone, il goto della progr. OO
- raramente utilizzata ma in alcuni casi indispensabile
- utile e necessaria sebbene complessa

i linguaggi offrono diversi approcci:

- fornire solo l'ereditarietà semplice
  - Smalltalk
- fornire l'ereditarietà multipla e i costrutti per risolvere i conflitti
  - C++, Eiffel
- supporto all'ereditarietà multipla dei contratti ma non dell'implementazione
  - Java (interfacce)

Vantaggi derivanti dai concetti OO

- Modularità
- Information Hiding
- Estendibilità
- Integrabilità
- Riutilizzo del codice

Vantaggi derivanti dai concetti OO

Modularità

- supportata sia dalla metodologia di design che dai costrutti del linguaggio
- OOP: classe = modulo
- OOP altre proposte:
  - cluster di classi legate logicamente
  - sottosistemi di classi

Vantaggi derivanti dai concetti OO

Information Hiding

- OO: netta distinzione tra l'interfaccia di una classe e la sua implementazione
- possibili diverse implementazioni dell'interfaccia

Vantaggi derivanti dai concetti OO

Estendibilità

- ereditarietà:
  - il riutilizzo di classi esistenti facilita la definizione di nuove classi
  - tanto più l'albero di ereditarietà è ricco , tanto meno sforzo deve essere speso per sviluppare una nuova classe
- ereditarietà + polimorfismo
  - esempio

Esempio ereditarietà + polimorfismo

Soluzione tradizionale

- 3 tipi di figure, array di figure geometriche, per ogni figura calcolare l'area
- codice tradizionale:

```
while elemento
begin
  if elemento isa "cerchio" then write p X raggio X
  raggio
  elseif elemento isa "quadrato" then write lato X lato
  elseif elemento isa "triangolo" then write base X
                                          altezza X 0.5
  end
  elemento++
end
```

Esempio ereditarietà + polimorfismo

Soluzione OOP

- definizioni:
  - tre classi (cerchi, quadrati, triangoli) più una super classe (elemento)
- codice OO:

```
while elemento
begin
  write elemento.area
  elemento++
end
```

- notare:
  - il codice globalmente da scrivere non si riduce nè aumenta, vanno scritte 3 implementazioni del metodo area

Esempio ereditarietà + polimorfismo

Vantaggio riguardo all'estendibilità

- ... arrivano nuove specifiche
  - aggiungere il pentagono alle figure geometriche
- programma tradizionale
  - correggere per inserire il nuovo case
    - il rischio di correggere male è alto (devo trovare tutti i punti del codice dove un oggetto è trattato in modo diverso a seconda della forma geometrica)
    - le correzioni sono sparpagliate in tutto il sistema (vengono fatte da programmatori diversi: necessità di coordinamento o duplicazioni del lavoro)
- programma OOP:
  - si aggiunge una nuova classe e l'algoritmo funziona come prima

Integrabilità

- O-O: interfacce ben definite e di limitate dimensioni
  - facilità di comprensione
  - facilità di integrazione

Vantaggi derivanti dai concetti O-O

Riutilizzo

- ogni volta che viene creata una istanza di una classe si ha riutilizzo del codice
- ereditarietà
  - riuso di una classe esistente per definire nuove classi