Programmazione ad oggetti

di Roberta Molinari

Risoluzione di un problema

Prevede 4 passaggi

1. Analisi dei dati
2. Stesura dell'algoritmo
3. Codifica del programma
4. Test del programma

- La **programmazione strutturata** si concentra sulla strutturazione dell'algoritmo
- La **programmazione ad oggetti** si concentra sull'analisi dei dati

Programmazione strutturata

- Nella programmazione strutturata si utilizza la tecnica di top-down: si parte da un problema e lo si scompone in sottoproblemi.
- Il programma prevede la chiamata ad un main iniziale che richiama delle funzioni
- Il programma è suddiviso in moduli indipendenti

Svantaggi:

- Difficile descrivere sistemi complessi
- Difficile riutilizzare codice già scritto

OOP-Object Oriented Programming Introduzione

- Nel mondo reale esistono oggetti semplici, autonomi ciascuno con le sue caratteristiche e funzionalità specifiche, in grado di fare o su cui è possibile fare alcune operazioni. Gli oggetti semplici possono essere uniti per formare oggetti più complessi

Es. un PC è formato da CPU, RAM, Scheda madre, … Ogni componente è autonoma e può essere creata da produttori diversi; non è necessario conoscerne la struttura interna, ma solo le caratteristiche, la funzionalità e le specifiche di interfaccia per creare un PC funzionante

OOP-Object Oriented Programming Introduzione

- La **programmazione orientata agli oggetti OOP** si sviluppa intorno al 1970
- Nei sistemi complessi è difficile descrivere il problema come un'unica entità: nella programmazione ad oggetti si decompone il sistema in unità più piccole aventi comportamenti e caratteristiche più semplici: un sistema sw viene realizzato mediante un'insieme di **oggetti** che cooperano per la soluzione. Come nel mondo reale non è necessario conoscerne la struttura interna, ma solo le funzionalità e l'interfaccia di comunicazione

OOP-Object Oriented Programming Introduzione

- Nella OOP non si definiscono i singoli oggetti reali, ma si vanno a individuare le caratteristiche generiche che ogni singolo oggetto deve possedere per far parte di una specifica classe di oggetti

ES. la classe *Processore* è un modello astratto di ogni singolo processore reale. Ogni singolo processore reale è un istanza della classe *Processore*

OOP-Object Oriented Programming Introduzione

- La **programmazione orientata agli oggetti** usa un approccio **buttom-up**, si parte dalle definizione delle classi o dall'individuazioni di classi già esistenti, per creare il programma.
- Migliora:
  - lettura e **comprensione** del codice
  - **Manutenzione**
  - **Riusabilità** del codice in altri programmi
  - **Robustezza** in sistemi complessi o con grandi quantità di dati

OOP-Object Oriented Programming
Linguaggi orientati agli oggetti

I linguaggi che utilizzano gli oggetti si possono classificare in

Tipo	Permettono di trattare
object-oriented	–classi –oggetti –ereditarietà
class-based	–classi –oggetti
object-based	–oggetti

Quelli **object-oriented** si dicono **puri** se ogni cosa è un oggetto, **ibridi** se è possibile utilizzare tipi di dati "classici"

OOP-Object Oriented Programming
Classe

- Una **classe** è un modello astratto generico per una famiglia di oggetti con caratteristiche comuni. Ne definisce:
  - le caratteristiche (proprietà)
  - i comportamenti o funzionalità (metodi)
- Le classi sono organizzate in gerarchie e collegate tramite relazioni di varie tipologie
- L'implementazione di una classe in un linguaggio di programmazione conterrà il codice descrittivo degli oggetti della classe

OOP-Object Oriented Programming
Oggetto

- Ogni **oggetto** deriva da una classe (ne è l'istanza)
- Gli oggetti sono in grado di memorizzare uno stato e eseguire operazioni o interagire tra loro richiedendo operazioni ad altri oggetti tramite messaggi
- I metodi e le proprietà sono chiamati anche i **membri** dell'oggetto
- **Stato**: insieme dei valori assunti dalle proprietà in un determinato istante

Programmare ad oggetti

- Dato il problema si deve:
  - Individuare gli oggetti che realizzano un modello del problema
  - Definire le classi degli oggetti indicando attributi e metodi e relazioni fra classi
  - Stabilire come gli oggetti interagiscono (creando un flusso di messaggi)

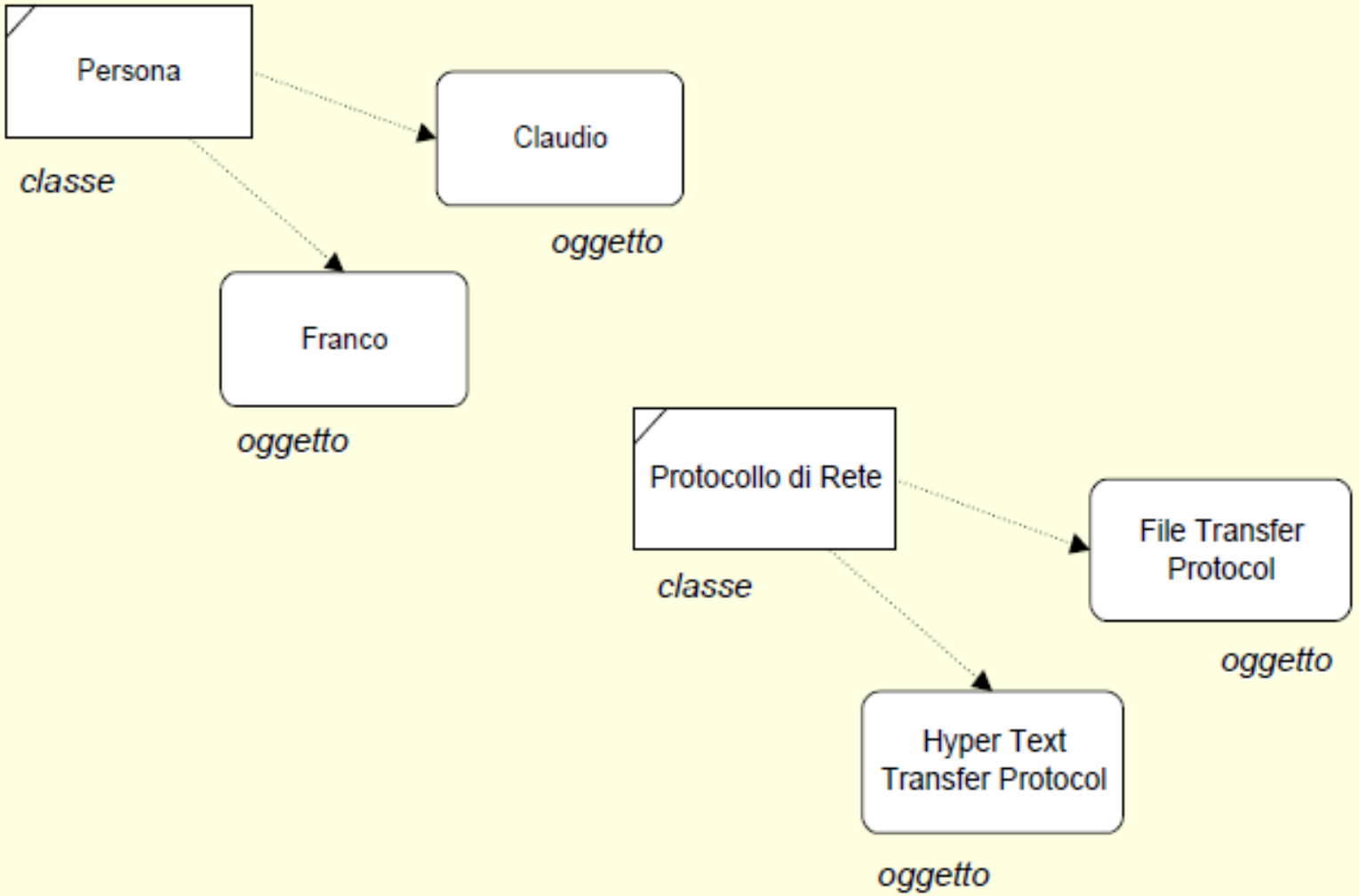
Gli oggetti e le classi

- Una **classe** descrive un insieme di oggetti tramite gli attributi e i metodi che li accomunano, ne rappresenta un modello. Non rappresentano un oggetto particolare. Non può esistere un oggetto se non viene creata la classe a cui appartiene. È la dichiarazione del "tipo", non occupa spazio in memoria
- Gli oggetti creati a partire da una classe sono le **istanze della classe**. Prima si dichiara a quale classe appartengono, poi si alloca lo spazio in memoria e quindi si possono modificare gli attributi o usare i metodi.
- Ogni oggetto della stessa classe si differenzia per il suo stato interno, mentre hanno lo stesso comportamento

Gli oggetti e le classi

- Gli oggetti sono scatole nere la cui unica parte visibile dall'esterno è la sua interfaccia pubblica
- Sono gli attori le entità del sistema che si vuole realizzare per risolvere un problema
- Le classi sono "fabbriche" di oggetti: se si ha bisogno di un oggetto si usa una classe come "stampo" per avere un nuovo oggetto con le caratteristiche definite dalla classe
- Si può vedere la classe come un tipo e l'oggetto come una variabile di quel tipo

Gli oggetti e le classi



Tipo di dato astratto

- Un **tipo di dato astratto ATD** è un tipo di dato completamente specificato, ma indipendente dalla sua implementazione. È caratterizzato da:
  - Definizione di un nuovo tipo
  - Definizione delle operazioni possibili sul tipo di dato (**l'interfaccia**)
  - L'unico modo di interagire con il dato è tramite le operazioni previste dall'interfaccia

  Es. tipo frazione algebrica

Tipo di dato astratto e classi

- Realizzano un tipo di dato astratto in quanto definiscono per gli oggetti
  - **Attributi o campi o proprietà**: descrivono l'oggetto tramite delle sue caratteristiche (*caratteristiche statiche*, definiscono lo **stato interno** dell'oggetto). Sono delle variabili
  - **Metodi**: operazioni che sono in grado di svolgere, che utilizzano i suoi attributi (*caratteristiche dinamiche*, definiscono il comportamento dell'oggetto). Sono delle funzioni o procedure accessibili solo attraverso l'oggetto. Per ciascuno deve essere definita l'*interfaccia o firma* (nome metodo, numero e tipo/ordine dei parametri) in modo da poter essere richiamato dall'esterno

I tipi di metodi

- **Modificatori**: modificano lo stato interno
- **Query**: restituiscono lo stato interno
- In particolare i metodi **accessor set/get** modificano/restituiscono le singole proprietà

  Es. setAltezza() getAltezza()
- **Costruttori**: deve avere lo stesso nome della classe e viene eseguito quando si crea un nuovo oggetto. Inizializzano i vari attributi e compiono tutte le operazioni necessarie alla creazione dell'oggetto. Se non definiti, ne esiste uno definito implicitamente che crea un'istanza della classe che inizializza gli attributi ai valori di default per i tipi corrispondenti senza fare controlli.

I messaggi

Formato messaggio:

oggetto1.metodoX(a,b,c)

- Gli oggetti interagiscono scambiandosi **messaggi** (per richiedere dei dati, per attivare un metodo, per cambiare lo stato)
- È costituito da
  - *Destinatario* (nome dell'oggetto)
  - *Selettore* (nome del metodo)
  - *Elenco argomenti* (insieme dei parametri per il metodo)
- Quando un oggetto riceve un messaggio verifica se al suo interno è definito il metodo richiesto, se lo è esegue il codice associato

Visibilità

Package: insieme di librerie di classe correlate, già compilate. Sono organizzati in modo gerarchico, la root è formata dal package *java*. Il package è come una cartella e le classi come i file.

Sono i **namespace** di .net

- La visibilità che una classe ha dei suoi metodi e delle sue proprietà è definita come segue

Visibilità modificatore di accesso	Public	Protected	Package (se non specificato)	Private
Stessa classe	Sì	Sì	Sì	Sì
Classe nello stesso package	Sì	Sì	Sì	No
Sottoclassi stesso package	Sì	Sì	Sì	No
Sottoclasse in package diverso	Sì	Sì *	No	No
Non sottoclasse in package diverso	Sì	No	No	No

* è consentito ad una istanza di una classe figlia far riferimento ai metodi e alle proprietà protected implementate nella classe padre ma non è permesso accedere ad essi attraverso una istanza della classe padre.

Incapsulamento

- Gli oggetti hanno la proprietà di contenere nel loro interno tutto ciò che si riferisce ad esso (caratteristiche e comportamenti), come in una capsula che racchiude tutto quello che li identifica: è l'**incapsulamento.**
- l'incapsulamento è il meccanismo su cui la classe basa la sua esistenza e robustezza. Grazie ad esso ogni classe risulta un'entità ben definita e distinta dal resto del codice: i metodi e i dati al suo interno sono separati da quelli delle altre classi, quindi non possono subire interferenze né possono a loro volta interferire in parti esterne alla classe non di loro competenza

Information hiding

- Quando tramite un messaggio si richiede un metodo, non si conosce e non interessa come questo sia implementato, che variabili utilizzi, questo perché l'oggetto è come una scatola nera (*blackbox*), che nasconde i dettagli delle sue caratteristiche (si parla anche di **information e data hiding**). Mantenendo l'interfaccia invariata, la manutenzione dei programmi è più efficiente e limitata all'interno del singolo oggetto.

Interfaccia di classe: elenco dei metodi pubblici di una classe, ovvero elenco delle funzioni utilizzabili dalle istanze della classe.

Incapsulamento e Information Hiding: regole

- Incapsulamento
  - Inserire nella stessa classe i dati e le operazioni sui dati
  - Progettare in base alle responsabilità
    - Lasciarsi guidare dalle operazioni: chiedersi quali sono le azioni di cui dovrebbe essere responsabile un oggetto della classe
- Information Hiding
  - Non esporre i dati
    - Dichiarare "privati" tutti gli attributi e usare metodi "get" e "set"
  - Non esporre la differenza tra attributi e dati derivati
    - Se velocità è un dato calcolato da altri attributi chiamare comunque il metodo getVelocità() piuttosto che calcolaVelocità()
- Non esporre la struttura interna della classe
  - Evitare metodi come getDataArray()

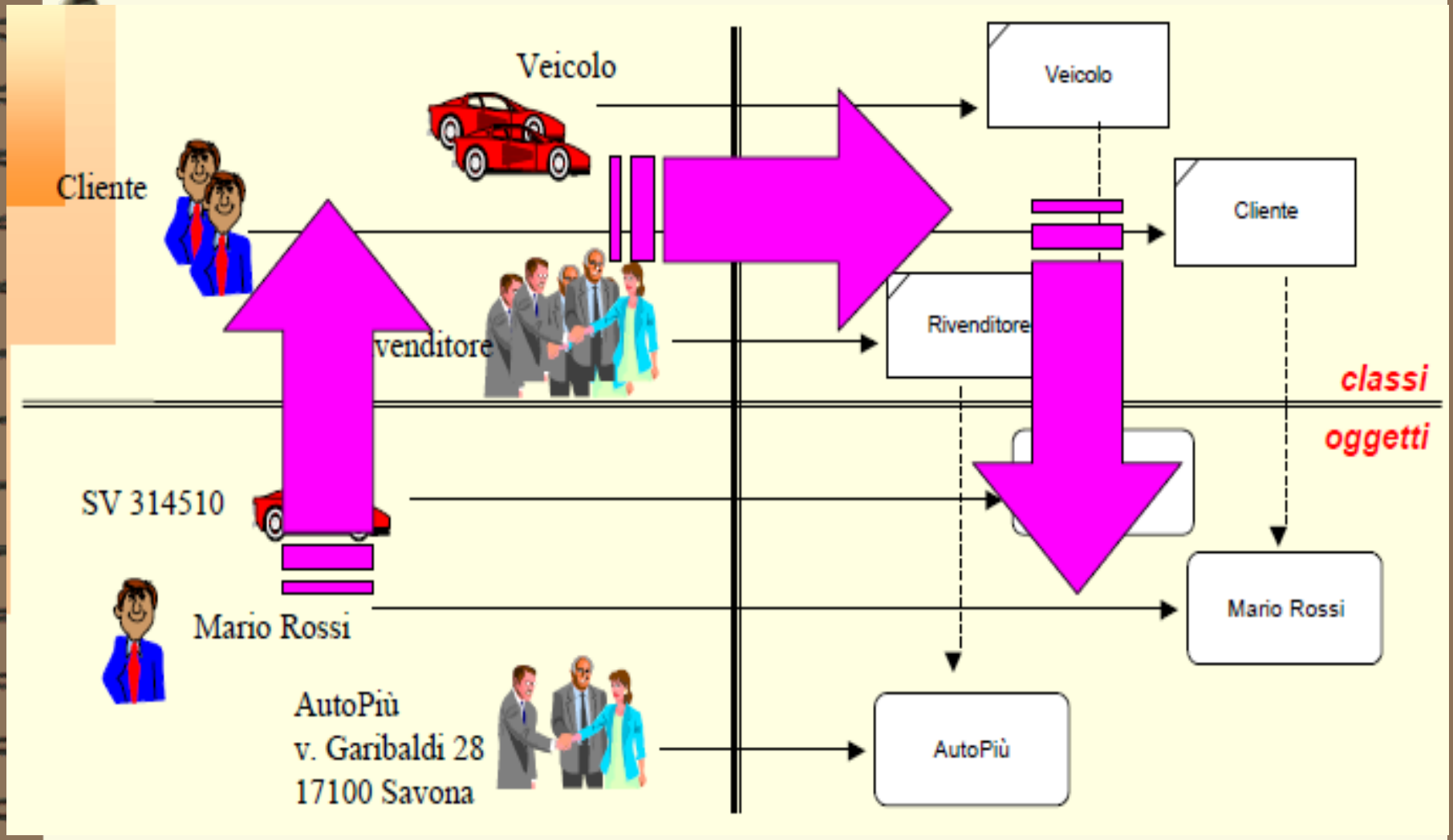
Sezione pubblica o privata

- I metodi o gli attributi direttamente visibili e utilizzabili da altri oggetti, saranno dichiarati PUBLIC. Normalmente lo sono solo i metodi per il principio del data hiding. La sezione pubblica rappresenta l'interfaccia della classe
- I metodi o gli attributi accessibili solo all'interno della propria classe, in quanto sono utilizzati per implementare il proprio comportamento, sono dichiarati PRIVATE. Normalmente lo sono gli attributi, per questi dovranno essere dichiarati dei metodi pubblici per leggerli **get** o scriverli **set**, in modo che gli oggetti li possano usare.

Rappresentazione classi e oggetti

- **Diagramma delle classi**
  - Struttura logica del sistema
  - Componenti e relazioni tra esse
  - Descrizione generiche di sistema (casi generali)
- **Diagramma degli oggetti**
  - Particolari istanze di sistemi (casi particolari)

Rappresentazione classi e oggetti



Formalismo UML

NomeClasse

+attributoPubblico

-attributoPrivato

#attributoProtetto

attributoPackage

±attributoStatico

+metodoPubblico(par1:tipo;par2:tipo2)

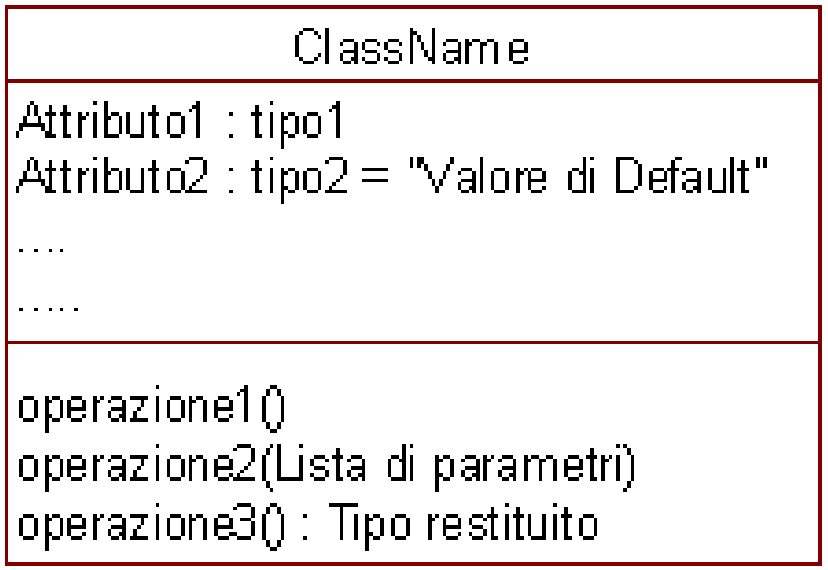
-metodoPrivato(par1:tipo;par2:tipo2)

#metodoProtetto(par1:tipo;par2:tipo2)

metodoPackage(par1:tipo;par2:tipo2)

+metodoVirtuale(par1:tipo;par2:tipo2)

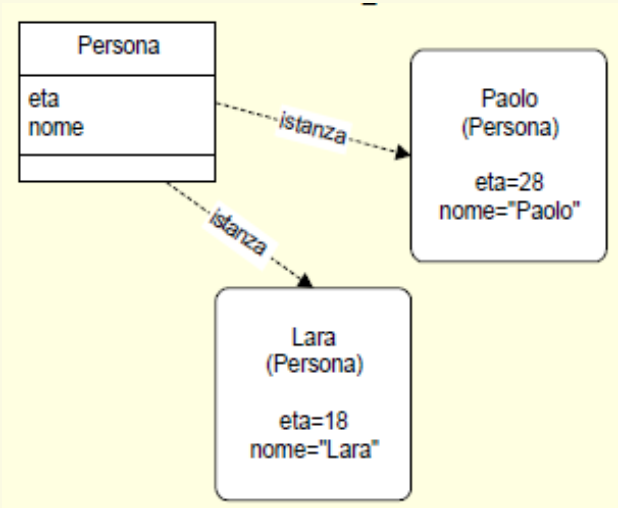
±metodoStatico (par1:tipo;par2:tipo2)



statico o **di classe**, esiste ed è visibile e utilizzabile anche se non ci sono istanze, è condiviso da tutte le eventuali istanze

Rappresentazione classi e oggetti

- Ad ogni **attributo** in una classe corrisponde un valore in un oggetto



- I **metodi** sono uguali per tutti gli oggetti della stessa classe: il comportamento di un oggetto dipende dalla sua classe

Occupazione di memoria

In teoria, la generazione di un oggetto per istanzazione di una classe comporta:

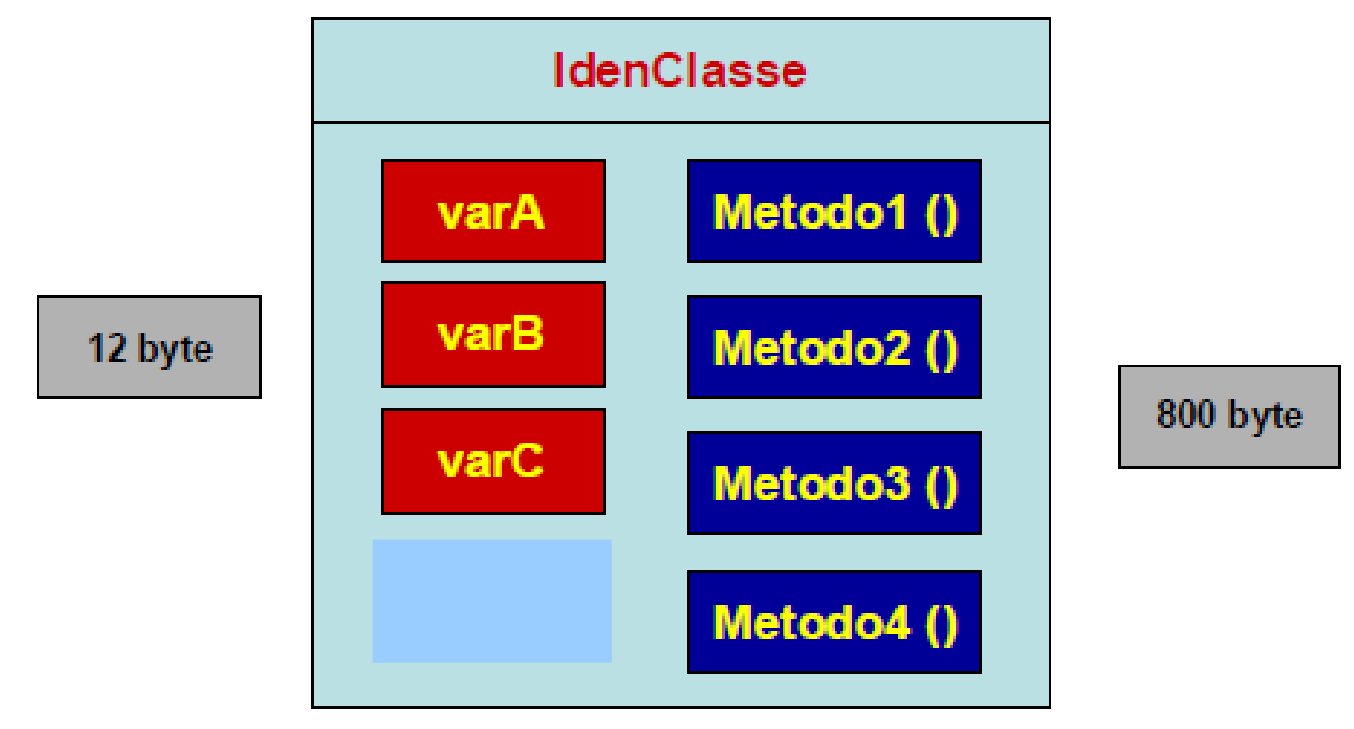
- la creazione di una struttura dati analoga a quella definita dalla classe
- la duplicazione nell'oggetto dei metodi specificati nella classe

In realtà, è necessario distinguere tra:

- attributi e metodi di istanza (propri dell'oggetto) e di classe (comuni a tutti gli oggetti della classe)
- modello teorico e realizzazione effettiva di classi e oggetti

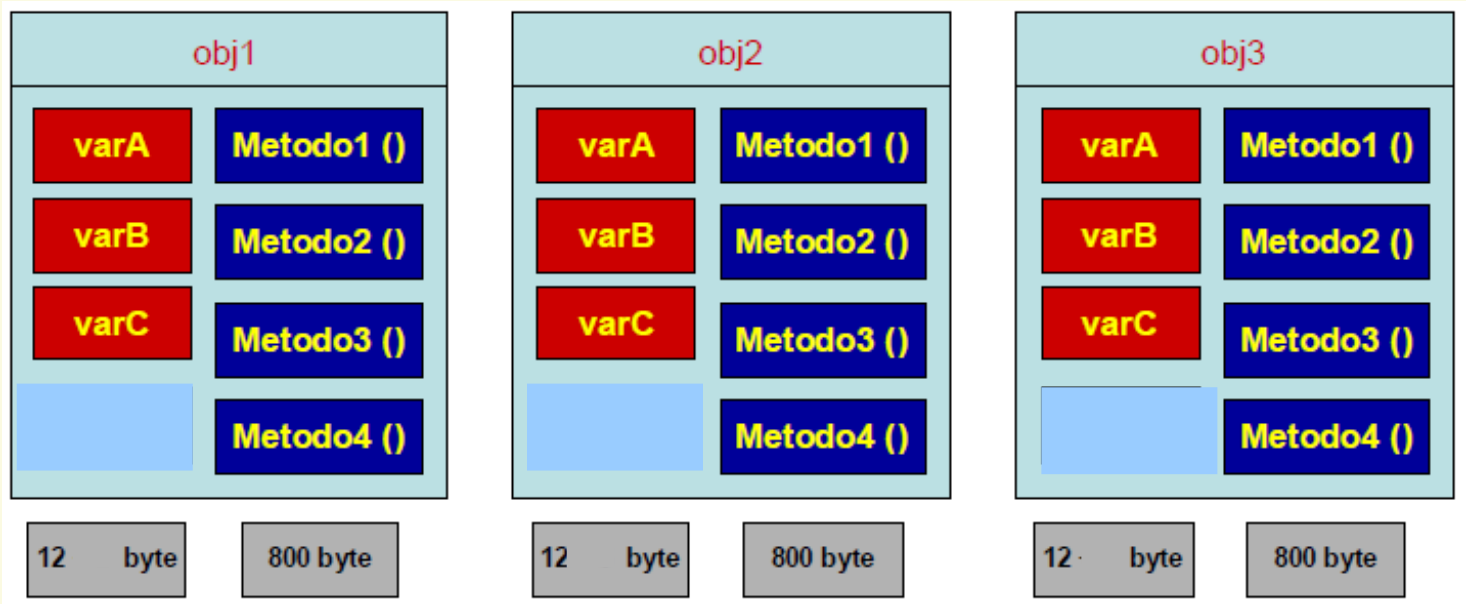
Occupazione di memoria

Supponiamo che questa sia la dichiarazione di una classe **IdenClasse**



Occupazione di memoria Teoricamente

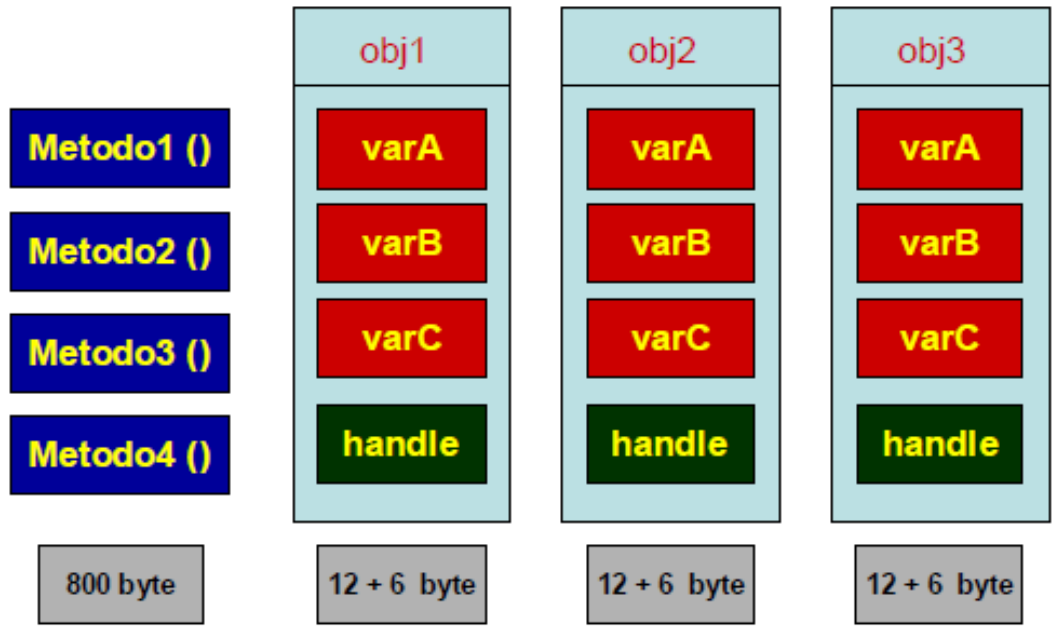
Da IdenClasse sono istanziati gli oggetti **obj1, obj2, obj3**



Memoria teorica richiesta: (12+ 800) * 3 = 2436 byte

Occupazione di memoria Realizzazione

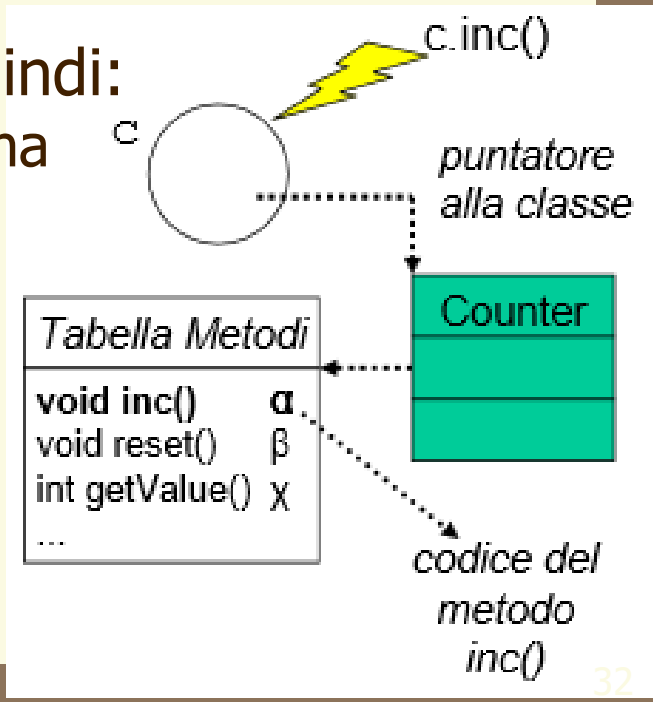
Da IdenClasse sono istanziati gli oggetti **obj1, obj2, obj3:** duplicando gli attributi e condividendo i metodi (logicamente distinti)



Memoria effettiva richiesta: 800 + 18 * 3 = 854 byte

Occupazione di memoria Realizzazione

- Ogni istanza contiene un riferimento alla propria classe che include una tabella VMT (Virtual Method Table) che mette in corrispondenza i nomi dei metodi da essa definiti con il codice compilato relativo a ogni metodo
- Chiamare un metodo comporta quindi:
  - accedere alla tabella VMT opportuna in base alla classe dell'istanza
  - in base alla signature del metodo invocato, accedere alla entry della tabella corrispondente e ricavare il riferimento al codice del metodo
  - invocare il corpo del metodo così identificato



Attributi e metodi di classe

- Gli **attributi statici** o **di classe** permettono di definire attributi comuni a tutte le istanze di una classe a cui ci si può riferire anche senza aver istanziato nessun oggetto (viene allocato al primo utilizzo)
- I **metodi di classe o statici:** si possono richiamare indipendentemente dall'esistenza di una istanza della relativa classe (sono metodi di utilità). Lavorano solo su attributi statici
- Per riferirsi all'altro tipo di metodi/attributi si parla di **metodi /attributi di istanza**

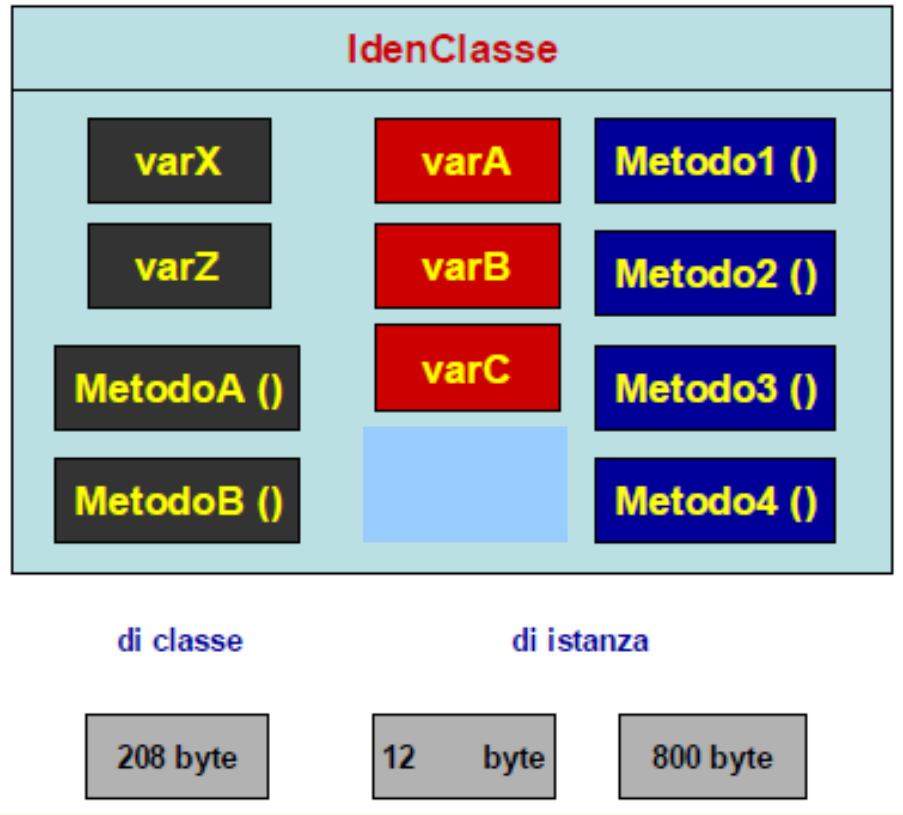
Ci si può riferire ad essi sia tramite il nome della classe che tramite l'eventuale istanza di essa

`NomeClasse.attr` oppure `objClasse.attr`

`NomeClasse.met()` oppure `objClasse.met()`

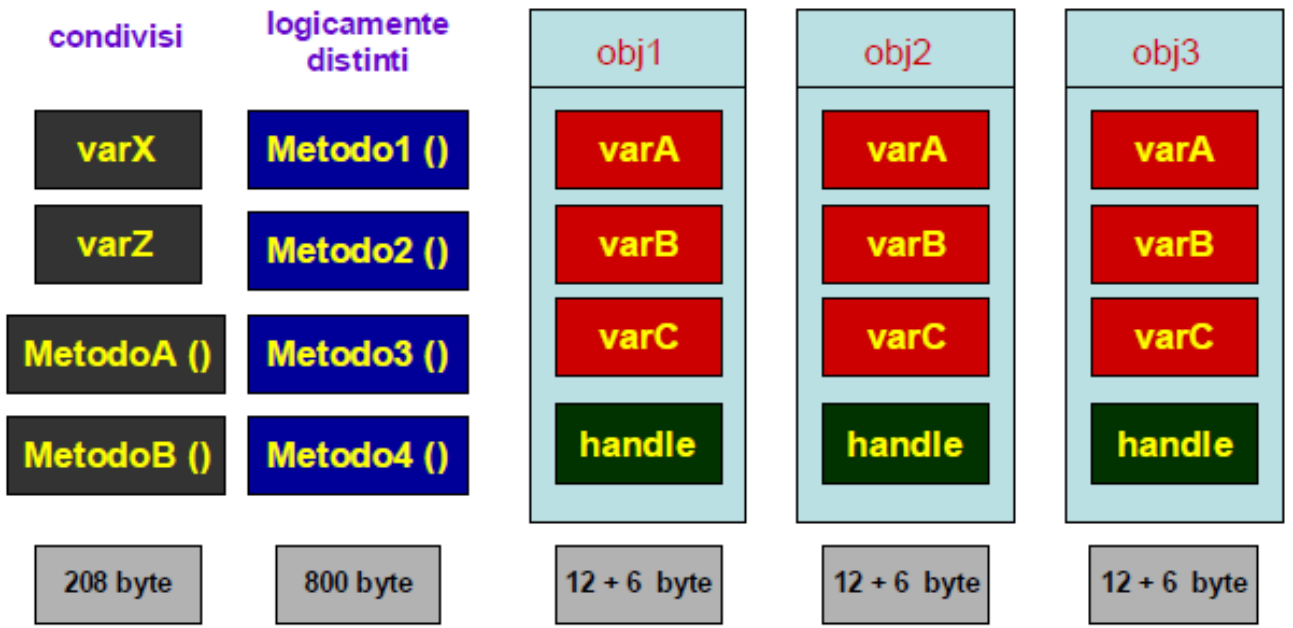
Occupazione di memoria Attributi e metodi di classe

Attributi e metodi (di classe): esiste una copia comune per tutti gli oggetti creati



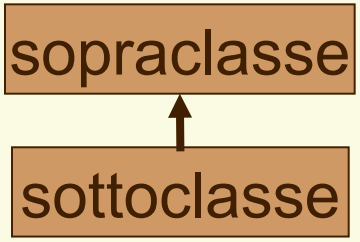
Occupazione di memoria Attributi e metodi di classe

Da IdenClasse sono istanziati gli oggetti **obj1, obj2, obj3** duplicando gli attributi di istanza, condividendo i metodi (logicamente distinti) di istanza e condividendo gli attributi e i metodi di classe



Memoria richiesta: 208 + 800 + 18 * 3 = 1062 byte

Ereditarietà



- È possibile, tramite il meccanismo dell'**ereditarietà**, costruire nuove classi (**sottoclassi o derivate**) a partire da classi già esistenti (**sopraclasse o base**). Migliora il riutilizzo del software.
- La sottoclasse **eredita** tutti i metodi e gli attributi della sopraclasse anche privati e può fare riferimento nel codice direttamente a tutti quelli non privati.
- Le classi vengono così organizzate in una **gerarchia**.
- La sottoclasse specializza la sopraclasse per:
  - **Estensione**: vengono aggiunti nuovi attributi o metodi
  - **Ridefinizione**: ridefinisce attributi (**hiding**) o metodi della sopraclasse (**overriding**)

```
public class SottoClasse extends SuperClasse {
<nuove_variabili_istanza>    <nuovi_metodi>  }
```

Ereditarietà

- La classe base definisce **attributi e interfaccia comuni (ereditati)**
- Le classi derivate possono **invocare i metodi delle classi base**, purché definiti **pubblici o protetti**,ma non ereditano i costruttori
- Le classi derivate **non possono eliminare** dati o metodi delle sopraclassi
- Seguono il **Principio di Sostituibilità di Liskov LSP**
  - *Oggetti di una classe derivata (subclass) sono sostituibili ad oggetti della classe più generale (base class or super-class) ovvero dovunque si usa un oggetto della classe madre deve essere possibile sostituirlo con un oggetto di una qualunque delle classi figlie*

Ereditarietà
Costruttore di una sottoclasse

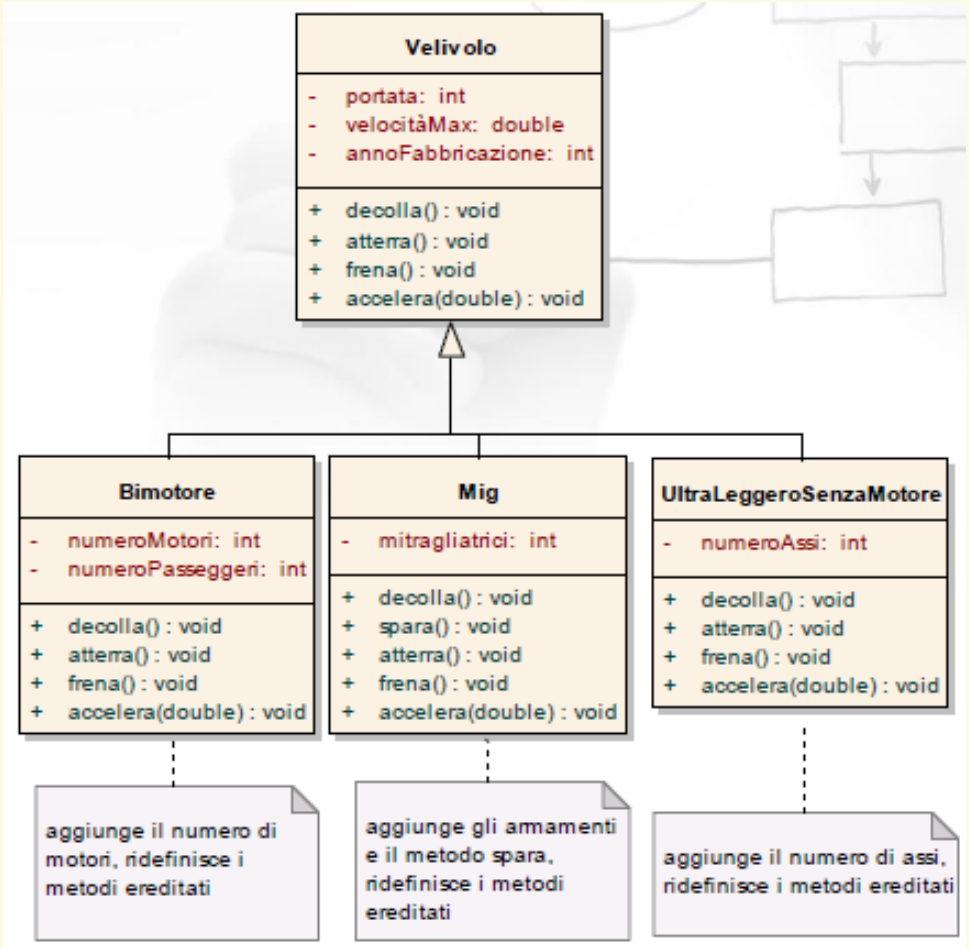
- I costruttori non vengono ereditati
- Ogni istanza della classe derivata comprende in sé, indirettamente, un oggetto della classe base, quindi quando istanzio un oggetto di una classe derivata, viene automaticamente richiamato il costruttore della classe base e poi quello della classe derivata. È possibile richiamare esplicitamente il costruttore della superclasse tramite l'istruzione `super(listaParametriAttuali)` posta come prima istruzione del costruttore
- Il costruttore è passibile di overloading

ATTENZIONE: se non c'è un costruttore nella sopraclasse senza parametri, devo definire nella classe derivata un costruttore che richiami in modo esplicito super() con i parametri richiesti (il compilatore dà un errore)

Ereditarietà UML

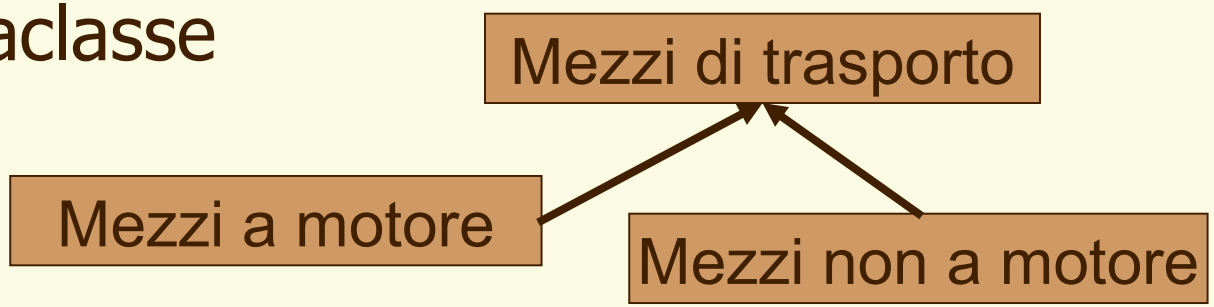
In UML la relazione che esprime l'ereditarietà è la **generalizzazione**

Si usa una freccia continua che indica una relazione **IS-A**



Ereditarietà Singola

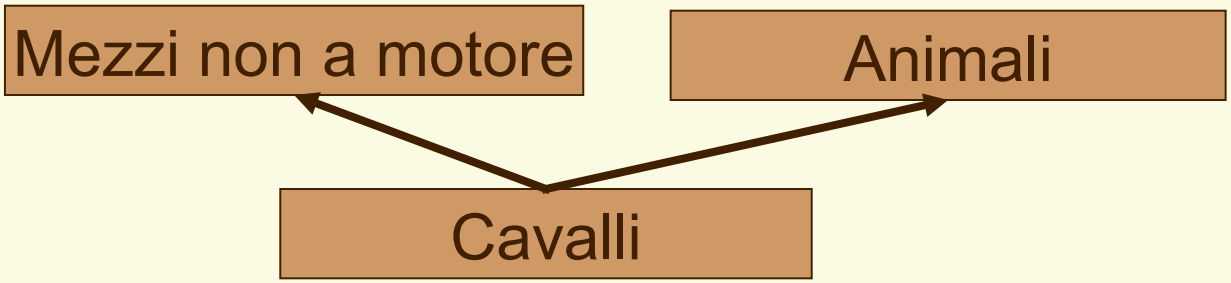
- **Singola**: se una sottoclasse deriva da una sola sopraclasse



- Java supporta solo questo tipo di ereditarietà, e ogni classe non specificata esplicitamente quale sottoclasse di altra classe viene vista comunque come sottoclasse della classe **Object**, che costituisce la radice comune dell'albero di ereditarietà delle classi

Ereditarietà Multipla

- **Multipla**: se una sottoclasse deriva da più sopraclassi (non tutti i linguaggi di programmazione la prevedono)



Ereditarietà Polimorfismo

Si ha **polimorfismo** se un metodo è in grado di adattare il suo comportamento allo specifico oggetto su cui deve operare. Ha 2 aspetti:

- **Prima forma**: lo stesso metodo di una classe base è applicato ai diversi oggetti delle classi derivate
- **Seconda forma**: lo stesso metodo genera l'esecuzione di codice diverso. Si può distinguere:
  - **overloading:** quando lo stesso metodo, ha tipo o numero di parametri diversi, (**firma** diversa) e quindi codice diverso (non basta cambiare il tipo restituito) (all'interno della stessa classe o in classi derivate per estensione)
  - **overriding:** quando lo stesso metodo ha la stessa firma, ma codice diverso, implica l'ereditarietà (sottoclasse ridefinisce metodo della sopraclasse)

Si possono avere 2 metodi **statici** con lo stesso nome, ma restano distinti e quello della sottoclasse non sovrascrive (overriding) quello della superclasse, ma lo nasconde (hiding). Entrambi restano raggiungibili attraverso il nome della classe (o con super). Se si cerca di ridefinirlo non statico si segnala errore

overriding e overloading

- Per fare l'**overriding** di un metodo, nella classe base questo deve essere visibile (public o protected) e overridable (né finale, né statico perché si ha un hiding)

```
class Base
   [{Protected|Public}] NomeMetodo([par])

class Derivata extends Base
   [{Protected|Public}] NomeMetodo([par])
```

- Per fare l'**overloading** di un metodo (sono tutti overloadable per default) (ATTENZIONE non basta cambiare il tipo restituito)

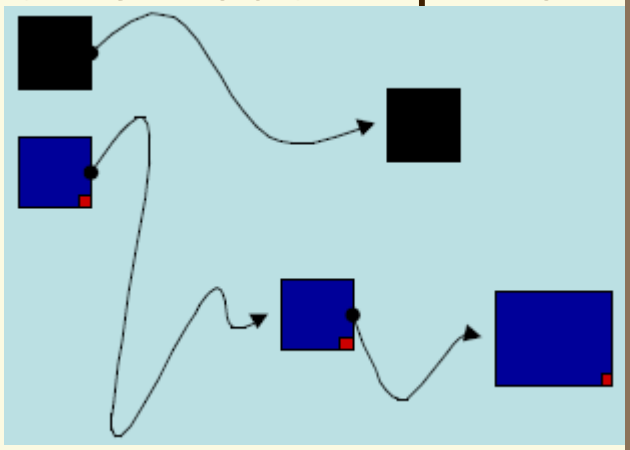
```
   modifAccesso NomeMetodo([par])

   modifAccesso NomeMetodo([parDiversi])
```

Ereditarietà
Esempio

Generare quadrati da sottoporre a due tipi di operazioni:

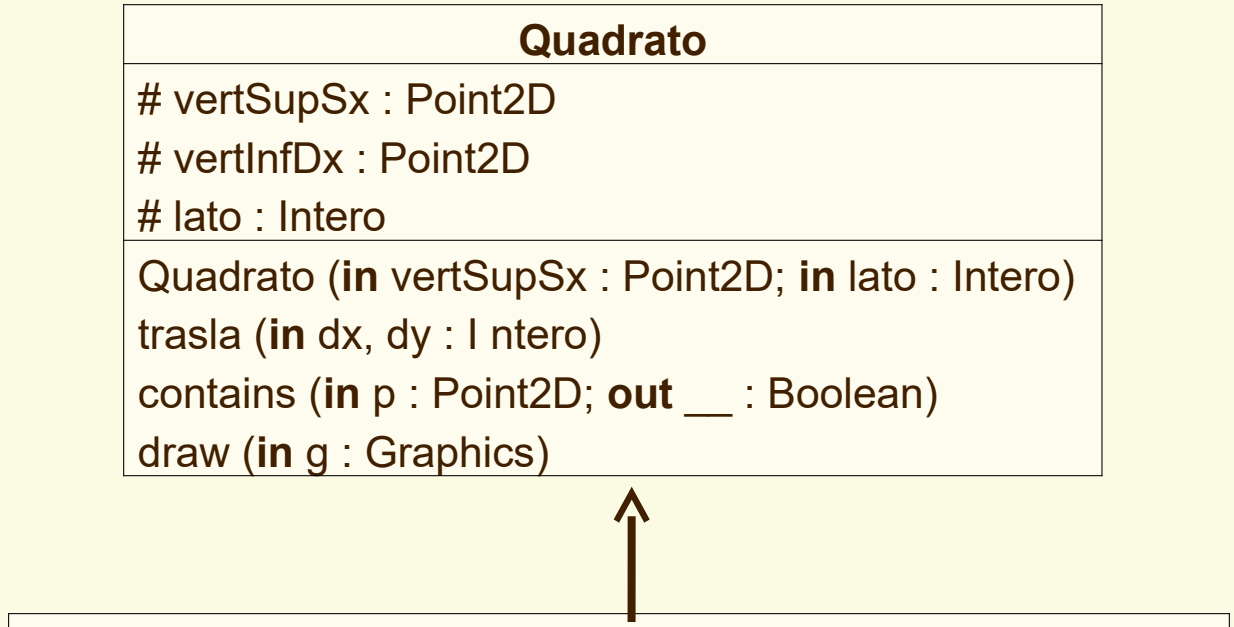
- traslazione
- traslazione e deformazione



Operazioni disponibili:

- istanzazione di oggetti di tipo Quadrato
- verifica di punto interno (ovvero esterno) a un quadrato
- tracciatura di un quadrato
- istanzazione di oggetti di tipo QuadratoDeformabile
- registrazione della deformazione di un quadrato deformabile traslazione di un quadrato
- verifica di punto interno (ovvero esterno) alla regione di deformazione

Ereditarietà
Esempio



Ereditarietà Esempio

Istanze di Quadrato

- *vertSupSx*
- *vertInfDx*
- *lato*
- *trasla*()
- *contains*()
- *draw*()

Istanze di QuadratoDeformabile

- *vertSupSx* ereditato
- *vertInfDx* ereditato
- *lato* ereditato
- *trasla*() ereditato
- *contains*() ereditato
- *draw*() ereditato e sovrascritto
- *setDeformazione*() aggiunto
- *containsDeformazione*() aggiunto

Un oggetto che è istanza di QuadratoDeformabile è anche un oggetto di tipo Quadrato (possiede attributi e metodi ereditati)

Ereditarietà
NOTA BENE per overriding

- Un metodo ridefinente può cambiare il tipo restituito solo se il tipo di ritorno è a sua volta un sottotipo del tipo restituito dal metodo della superclasse. Se il tipo è primitivo, deve restare lo stesso (compilatore dà errore). Esempio in cui Rettangolo è una sottoclasse di Shape:

```
class Madre { …
      Shape metodo1() { … } }
class Figlia extends Madre { …
      Rettangolo metodo1() { …
  } }
```

- La clausola throws può differire, purché le eccezioni intercettate siano le stesse o sottotipi di quelle intercettate nel metodo della superclasse

Ereditarietà
NOTA BENE per overriding

- L'overriding è possibile solo se il metodo è accessibile, visibile. Altrimenti se nella sottoclasse viene creato un metodo con la stessa signature di quello non accessibile della superclasse, i due metodi saranno completamente scorrelati
- I metodi che operano l'overriding possono avere i propri modificatori di accesso, che possono tuttavia solo ampliare (rilassare) la protezione. Ad esempio un metodo protected nella superclasse può essere ridefinito protected o public, ma non private
- Il metodo ridefinente può essere reso final, se ritenuto opportuno e può essere reso abstract anche se non lo è quello della superclasse

Ereditarietà
NOTA BENE per overriding

- Attributi e metodi statici NON possono essere sovrascritti, ma solo adombrati (hiding); questo comunque non ha alcuna rilevanza perché si utilizza comunque il nome della classe di appartenenza per accedervi
- Un attributo non viene ridefinito ma adombrato (hiding): il nuovo campo rende inaccessibile quello definito nella superclasse (si deve usare super)

```
class Madre {int x;  public x() {…}
public class Figlia extends Madre {
  int x;
public Figlia(int k){
  if k==super.x this.x=k;
  super.x();}}
```

Ereditarietà
Livelli di protezione

Analizziamo i vari livelli di protezione nel contesto dell'ereditarietà

- **private** impedisce a chiunque di accedere al dato, anche a una classe derivata. Se deve usare per dati "veramente privati", nella maggioranza dei casi è troppo restrittivo
- **public** non implementa l'information hiding
- **package** è quello di default, ma il concetto di package non c'entra niente con l'ereditarietà
- **protected** è come package per chiunque non sia una classe derivata, ma consente libero accesso a una classe derivata, presente o futura, indipendentemente dal package in cui essa è definita. È quello da preferire, applicabile ad attributi e metodi

Ereditarietà
Up-casting, down-casting

- In una gerarchia di classi derivate è possibile assegnare a un riferimento (var) un oggetto istanza di una qualsiasi sottoclasse, ma non viceversa (principio di sostituibilità) perché non è possibile la corretta invocazione dei metodi: il riferimento vede i metodi e gli attributi della classe di dichiarazione a cui deve corrispondere del codice. Quindi ad un riferimento di tipo Object può essere assegnato qualsiasi istanza di oggetto perché sopraclasse di tutti
- I riferimenti agli oggetti si dicono **polimorfi**, ovvero possono riferirsi ad oggetti di tipo diverso all'interno della gerarchia
- Un oggetto non può essere assegnato ad un riferimento di una sottoclasse (errore di compilazione)

Ereditarietà
Up-casting, down-casting

- Si possono fare operazioni di casting per trasformare il riferimento ad un oggetto in un riferimento a:
  - una sopraclasse **up-casting** (sempre possibile e sottinteso)
  - una sottoclasse **down-casting** (possibile solo se il riferimento è in realtà ad un oggetto della stessa classe. Altrimenti verrà sollevata un'eccezione di tipo *ClassCastException* in esecuzione)

```
Persona p1,p2;
Docente doc1,doc2;      //Docente extends Persona
doc1= new Docente("Rossi Ugo","Mate");
p2= new Persona("Verdi Lia");
//errore doc2= new Persona("tizio");
p1=doc1;    //up-casting sottinteso (Persona)doc1
doc2=(Docente)p2; //down-casting non lecito
doc2=(Docente)p1;  //down-casting lecito perché dip1
  in realtà si riferisce ad un Docente
```

Ereditarietà instanceof

- Facendo il casting è possibile in fase di compilazione riferirsi ai metodi e agli attributi della nuova classe, ma posso sorgere errori in fase di esecuzione. Bisognerebbe verificare la classe di appartenenza
- **`instanceof`** è un operatore che consente di determinare a run-time il tipo di un oggetto. Restituisce true o false a seconda che l'oggetto sia o no un'istanza della classe di confronto o di una delle sue superclassi. L'operatore si applica anche alle interfacce.

```
if (p1 instanceof Docente)
  ((Docente)p1).setMateria("info");
```

Ereditarietà
Overloading e overriding

- L'**overloading** consente di definire in una stessa classe più metodi aventi lo stesso nome, ma che differiscano nella ***firma***, cioè nella sequenza dei tipi dei parametri formali. È il **compilatore** che può determinare quale dei metodi verrà invocato, in base al numero e al tipo dei parametri attuali
- L'**overriding** consente di ridefinire un metodo in una sottoclasse: il metodo originale e quello che lo ridefinisce hanno necessariamente la stessa firma. A tempo di esecuzione verrà invocato il metodo corrispondente alla classe dell'istanza a cui si fa riferimento. Solo l'**interprete** può quindi determinare quale deve essere eseguito

Ereditarietà
Binding

Il meccanismo che determina quale metodo deve essere invocato in base alla classe di appartenenza dell'oggetto si chiama **binding (legame).**

Si distingue in:

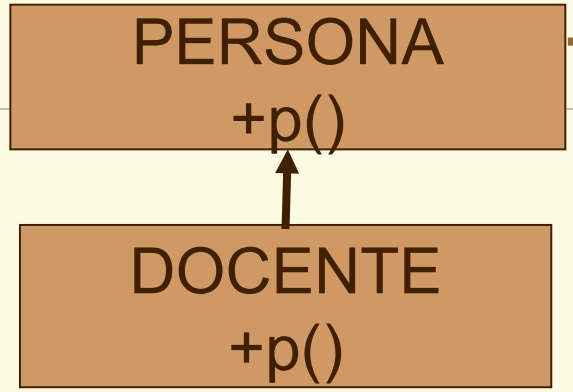
- **binding statico** o **early binding**: avviene a tempo di compilazione (standard in C, default in C++ e VB, per i metodi statici o final in Java)
- **binding dinamico** o **late binding**: avviene a tempo di esecuzione (non presente in C, possibile in C++ e VB (virtual), default in Java)

Ereditarietà
Binding

- **binding statico** o **early binding**: le chiamate ai metodi sono collegate alla versione del metodo a tempo di compilazione. Per sapere il codice associato il compilatore percorrerà a ritroso la gerarchia delle classi fino a trovare la sua implementazione. Tale collegamento non può essere indefinito (il compilatore dà errore) e non può cambiare in fase di esecuzione
- **binding dinamico** o **late binding**: le chiamate ai metodi sono collegate alla versione del metodo determinata a tempo di esecuzione, basandosi sul tipo dinamico dell'oggetto referenziato in quel momento. Si demanda la scelta del metodo da invocare all'interprete. Il compilatore si preoccuperà solo di assicurare che la chiamata sia collegabile a "qualche" metodo di quella classe o di una sua derivata, senza interessarsi di "quale" sarà.

Ereditarietà

tatico

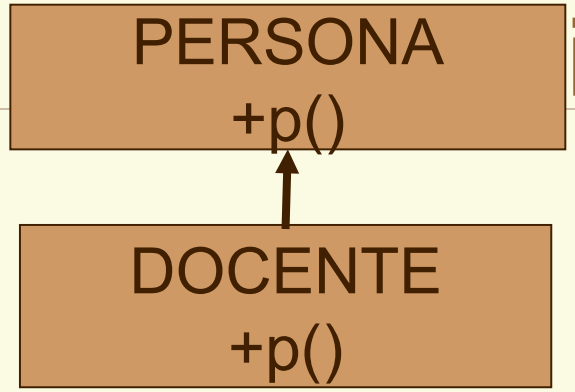



```
Persona p1;
Docente doc1;
p1= new Persona("Verdi Lia");
doc1= new Docente("Rossi Ugo","Mate");
p1.p()   //della classe Persona
   doc1.p()    //della classe Docente
```

Si sa quale usare in fase di compilazione.
In Java però, anche in questo caso, il binding è fatto a tempo di esecuzione. Solo se il metodo è non overridable (static o final), per cui non è possibile un overriding, Java assegna il codice in fase di compilazione

Ereditarietà

Binding dinamico




```
int cond = Input.readInt();
Persona tmp;
if (cond > 0)
      tmp = new Persona("Mario Rossi");
else

tmp = new Docente("Mario Rossi", "mate");tmp.p();
```

L'oggetto si saprà solo in fase di esecuzione, il metodo da lanciare non è conosciuto in fase di compilazione.
Il binding dinamico è quello usato di default in Java

Ereditarietà
Binding statico vs dinamico

STATICO

PRO il programma eseguibile finale ha un numero di istruzioni macchina minore e quindi è più efficiente

CONTRO se modifico un metodo devo ricompilare tutto il progetto

DINAMICO

PRO riusabilità del codice eseguibile (si ricompila solo il modulo modificato)

CONTRO meno efficienza in fase di esecuzione perché, per eseguire un metodo, devono anche eseguirsi le istruzioni per riconoscere l'oggetto a cui appartiene

Ereditarietà esempio

```
public class Animale {
public void verso () {} //metodo vuoto
}

public class Scoiattolo extends Animale {
public void verso() {
        System.out.println("SQUIT SQUIT"); }
}

public class Cane extends Animale {
public void verso() {
        System.out.println("BAU BAU");}
public void ringhiare() {
        System.out.println("GRRRR"); }
}
```

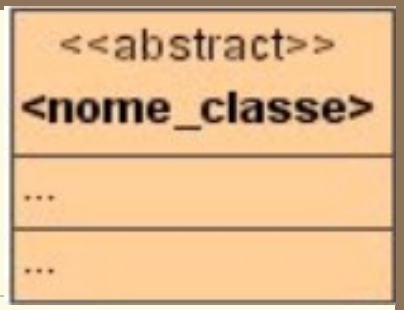
Ereditarietà esempio

```
public class Zoo {
 public static void main (String[] argv) {
  Animale[] gabbie= new Animale[2];
  gabbie[0] = new Scoiattolo();
  gabbie[1] = new Cane();
  for (int k=0; k< gabbie.length; k++){
    gabbie[k].verso();
    if (gabbie[k] instanceof Cane)
((Cane)gabbie[k]).ringhiare(); //downcasting
  }
 }
}
```

output

```
SQUIT SQUIT
BAU BAU
GRRRR
```

Classi astratte



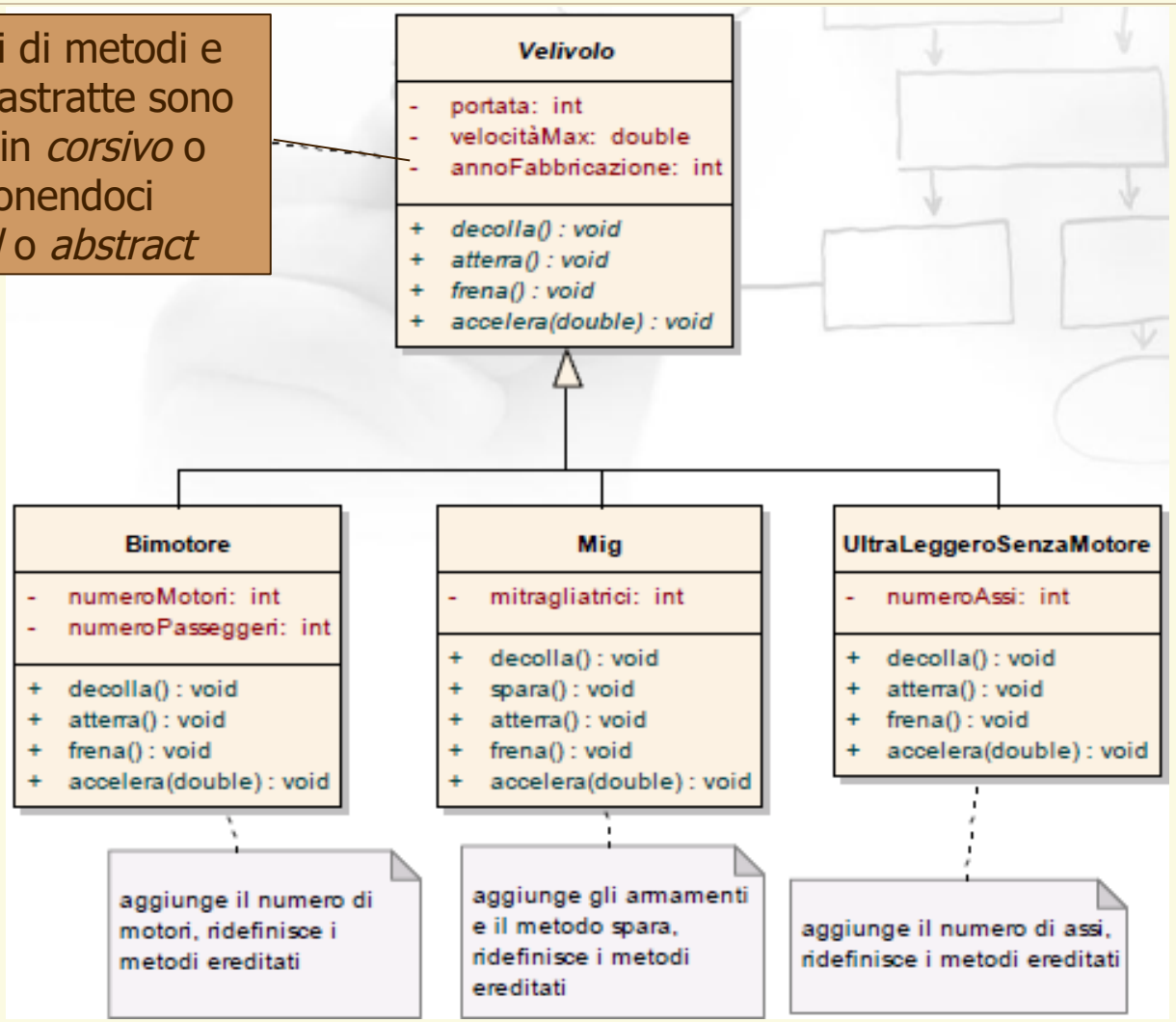
- Una **classe astratta o virtuale** definisce
  - un tipo di dato
  - un insieme di operazioni sul tipo di dato
    - alcune operazioni sono implementate: metodi
    - alcune operazioni possono non essere implementate: metodi astratti o virtuali (abstract)
- Una classe astratta non può essere istanziata, ma è possibile derivarne sottoclassi.
- Viene usata come classe base le cui sottoclassi vanno a specificare i comportamenti o le proprietà.
- In genere possiede almeno un metodo astratto non implementato. Una classe che ha un metodo abstract deve essere definita abstract.

Classi astratte

- Potrebbe avere anche definiti dei costruttori. Sarà il compilatore a segnalare un errore se si cerca di istanziarne un oggetto.
- Le sottoclassi della classe astratta implementeranno i metodi virtuali. Quando tramite la gerarchia delle classi, tutti i metodi virtuali saranno implementati, la classe finale può non essere più astratta e quindi si potranno istanziare gli oggetti di questa sottoclasse
- Una classe astratta può contenere chiamate di metodi astratti prescindendo dalla loro implementazione, una classe concreta non può usare metodi astratti

Classi astratte UML

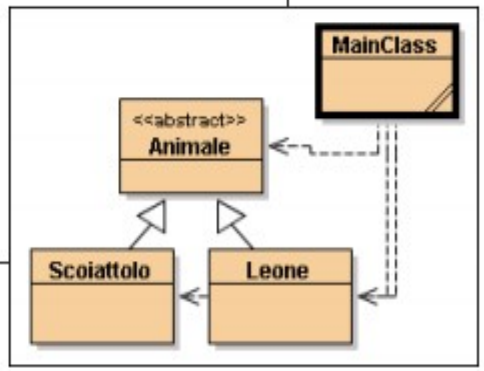
I nomi di metodi e classi astratte sono scritti in *corsivo* o anteponendoci *virtual* o *abstract*



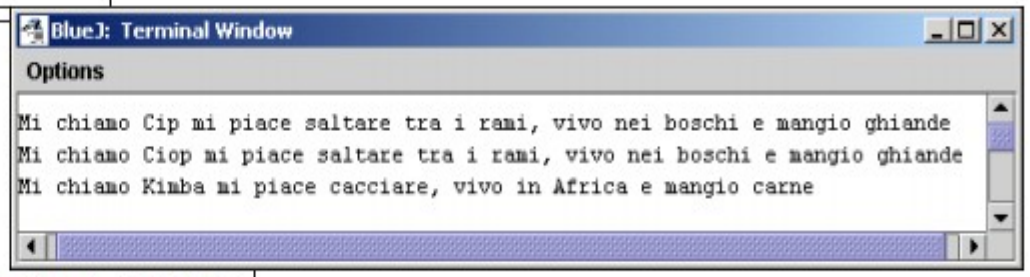
Classi astratte

```
1  public abstract class Animale
2  {
3  public String nome;
4  Animale(String nome)
5  {
6  this.nome=nome;
7  }
8  public abstract String attivita();
9  public abstract String vive();
10 public abstract String mangia();
11 public void presentati()
12 {
13 System.out.println("Mi chiamo "+nome+
14                    " mi piace "+attivita()+", "+
15                     "vivo "+vive()+
16                     " e mangio "+mangia());
17 }
18 }
```

```
1  public class MainClass
2  {
3  public static void main(String arg[])
4  {
5  Animale[] a=new Animale[3];
6  a[0]=new Scoiattolo("Cip");
7  a[1]=new Scoiattolo("Ciop");
8  a[2]=new Leone("Kimba");
9
10 for(int i=0;i<3;i++)
11 {
12 a[i].presentati();
13 }
14 }
15 }
```




```
1  public class Scoiattolo extends Animale
2  {
3  public Scoiattolo(String s)
4  {
5      super(s);
6  }
7  public String attivita(){ return "saltare tra i rami";}
8  public String vive(){ return "nei boschi";}
9  public String mangia(){ return "ghiande";}
10 }
```



Interfacce

- Ogni oggetto possiede implicitamente un'interfaccia: l'insieme degli attributi e dei metodi pubblici, con relative firme della classe di appartenenza
- È possibile definire esplicitamente un'**interface**, ovvero un'evoluzione più restrittiva del concetto di classe astratta, contenente solo le firme di metodi pubblici e la dichiarazione di attributi pubblici, final e statici (i modificatori se non dichiarati sono sottintesi). Non possono contenere codice

`[public] interface Nome{…}//`senza public è package

Interfacce

- Contengono solo
  - metodi `public abstract`
  - attributi `public static final`
- Una classe le può implementare (realizzare) definendo tutti i "corpi" dei metodi

```
[public] class C implements int [,int2]
```

  - Una interfaccia non può essere istanziata, ma definisce un protocollo del comportamento che deve essere fornito. Una qualsiasi classe che implementa una data interfaccia è quindi obbligata a fornire l'implementazione di tutti i metodi elencati nell'interfaccia

Interfacce

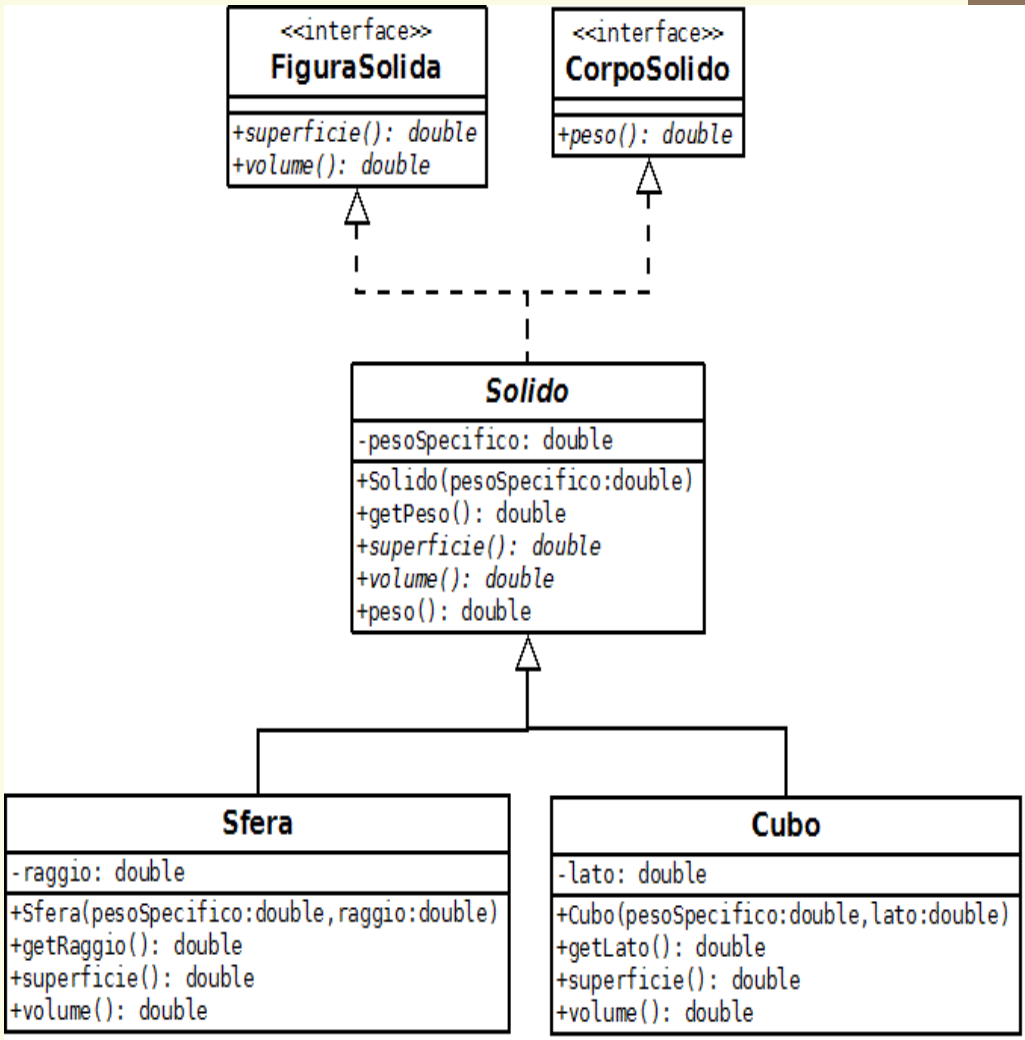
- Si dice che un'interfaccia è un contratto tra chi la implementa e chi la usa
- La classe che implementa un’interfaccia, essendo obbligata ad implementarne tutti i metodi garantisce la fornitura di un servizio
- Chi usa un interfaccia ha la garanzia che il contratto di servizio è effettivamente realizzato: non può accadere che un metodo non possa essere chiamato.

Interfacce

- Una **interfaccia** (**interface)** ha una struttura simile a una classe, ma può contenere SOLO **metodi d'istanza astratti** e **costanti statiche** (è sottinteso che lo siano, quindi non può contenere *costruttori*, *variabili statiche*, *variabili di istanza* e *metodi statici*).
- Si può dichiarare che una classe **implementa** (**implements**) una data interfaccia: in questo caso deve **realizzare** tutti i suoi metodi astratti. Se non li implementa tutti, la classe dovrà essere astratta. La realizzazione di un metodo deve rispettare la specifica firma del corrispondente metodo astratto.
- Un'interfaccia può estendere per ereditarietà altre interfacce (si crea una gerarchia di interfacce)

Interfacce UML

- La freccia tratteggiata indica la "**realizzazione", ossia l'implementazione di un'interfaccia**
- Una classe interfaccia può essere "realizzata" (implementata) da più classi concrete
- Una classe può realizzare (implementare) più interfacce



Interface raggruppamento di costanti

- Poichè qualsiasi campo in un'interfaccia è automaticamente static e final, l'interfaccia è stata anche usata per creare gruppi di valori costanti (come con enum in C o in C++) (in Java c'è Enum o Enumerated Type)

Esempio

```
public interface Months {
int JANUARY = 1, FEBRUARY = 2, MARCH = 3,
  APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
  AUGUST = 8, SEPTEMBER = 9, OCTOBER =
  10, NOVEMBER = 11, DECEMBER = 12; }
```

Interfacce e classi astratte

Si noti che:

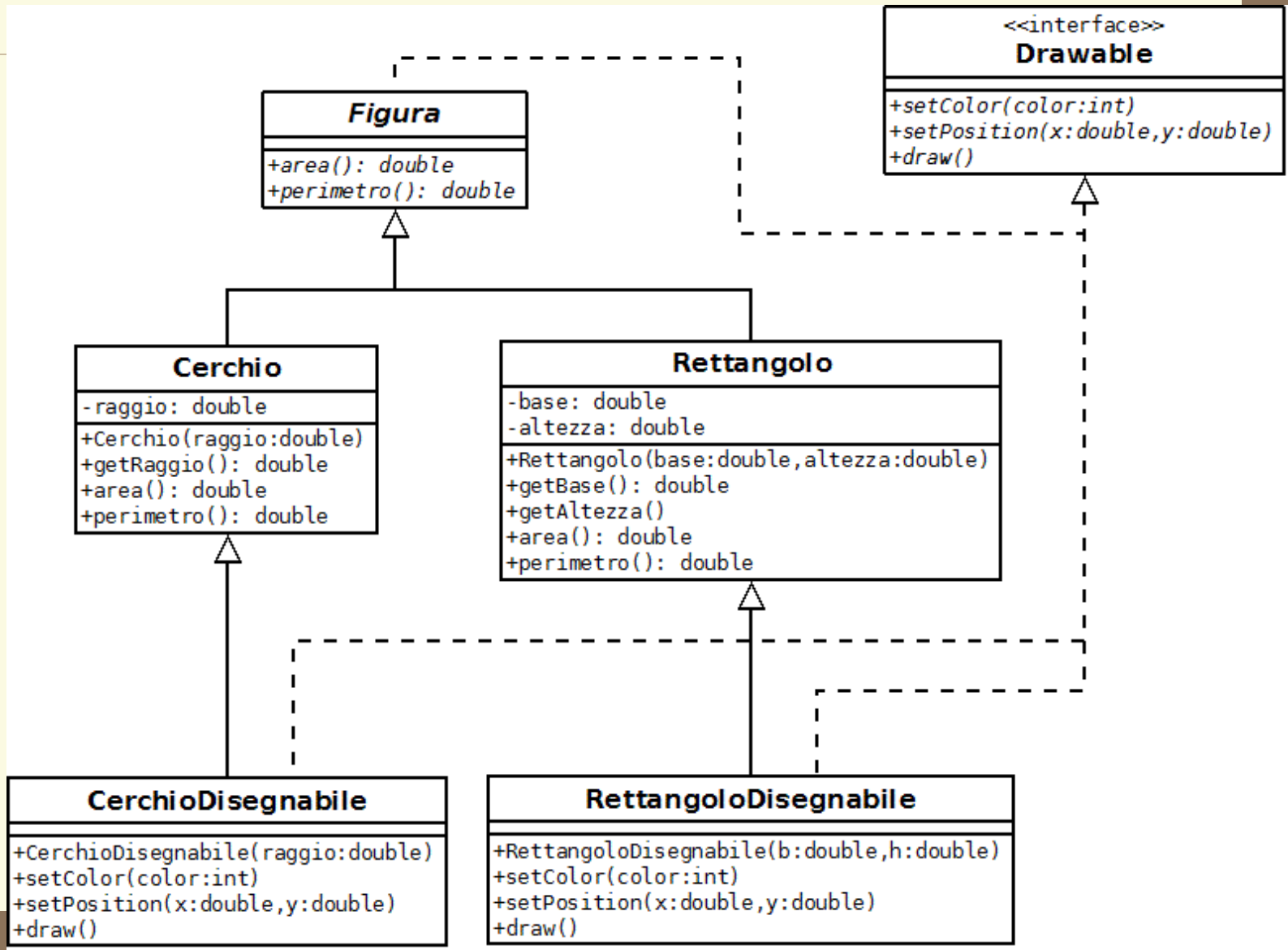
- Una classe (astratta o concreta) può implementare più interfacce (simula ereditarietà multipla), ma estendere solo una classe (astratta o concreta)
- Una classe (astratta o concreta) può essere usata per "fattorizzare" codice comune alle sue sottoclassi, una interfaccia non può contenere codice
- Una classe astratta può contenere chiamate di metodi astratti prescindendo dalla loro implementazione, una classe concreta non può usare metodi astratti, una interfaccia non può contenere codice

Interfacce come tipi

- Analogamente alle classi, ogni interfaccia definisce un tipo di dati in Java
- Un oggetto che implementa una data interfaccia ha come tipo anche il tipo dell'interfaccia, un oggetto può implementare molte interfacce di conseguenza può avere molti tipi
- Possiamo dichiarare una variabile indicando come tipo un'interfaccia. Ad una variabile di tipo interfaccia possiamo assegnare solo istanze di classi che implementano l'interfaccia.
- Su di una variabile di tipo interfaccia possiamo invocare solo metodi dichiarati nell'interfaccia (o nelle sue "super-interfacce").

Interfacce come tipi

Es.



Interfacce come tipi

```
public class UsaDrawable {
   public static void main(String args[]) {
    Drawable[] drawables = new Drawable[3];
    drawables[0]= new DrawableCircle(2.5);
    drawables[1]= new DrawableRectangle(1.2, 3.0);
    for (int i = 0; i < drawables.length; i++) {
       drawables[i].setPosition(i*10.0, i* 20.0);
       drawables[i].draw();
    }
…}
```

Grazie all'uso dell'interfaccia abbiamo potuto costruire un array che contiene indifferentemente istanze di classi diverse che implementano l'interfaccia Drawable e disegnarle tutte insieme con un solo ciclo for

Interfacce come tipi

- Le interfacce permettono una forma di compatibilità e di sostituibilità tra classi indipendente dalla catena di ereditarietà

  Per esempio è possibile definire una classe che implementa Drawable, ma non discende da Shape: un testo che può essere disegnato in una data posizione.

```
public class DrawableText implements Drawable {
    protected int c; protected double x, y;
    protected String s;
    public DrawableText (String s) { this.s = s; }
    public void setColor(int c) { this.c = c; }
    public void setPosition(double x, double y) {
      this.x = x; this.y = y; }
    public void draw() {…}
}
```

Quando utilizzare l'interfaccia e quando la classe astratta?

- Si usa una **classe astratta** per condividere codice fra più classi, se più classi hanno in comune metodi e campi o se si vogliono dichiarare metodi comuni che non siano necessariamente campi static e final.
- Si decide di utilizzare una **interfaccia** se ci si trova nella situazione in cui alcune classi (assolutamente non legate fra di loro) si trovano a condividere i metodi di una interfaccia, se si vuole *specificare il comportamento di un certo tipo di dato* (ma non implementarne il comportamento) o se si vuole avere la possibilità di sfruttare la "multiple inheritance".

L'interfaccia di ogni classe è definita al passo 2 (Individua attributi e metodi) e al passo 4 (individua il livello di protezione)

Approccio per la stesura di programmi OOP

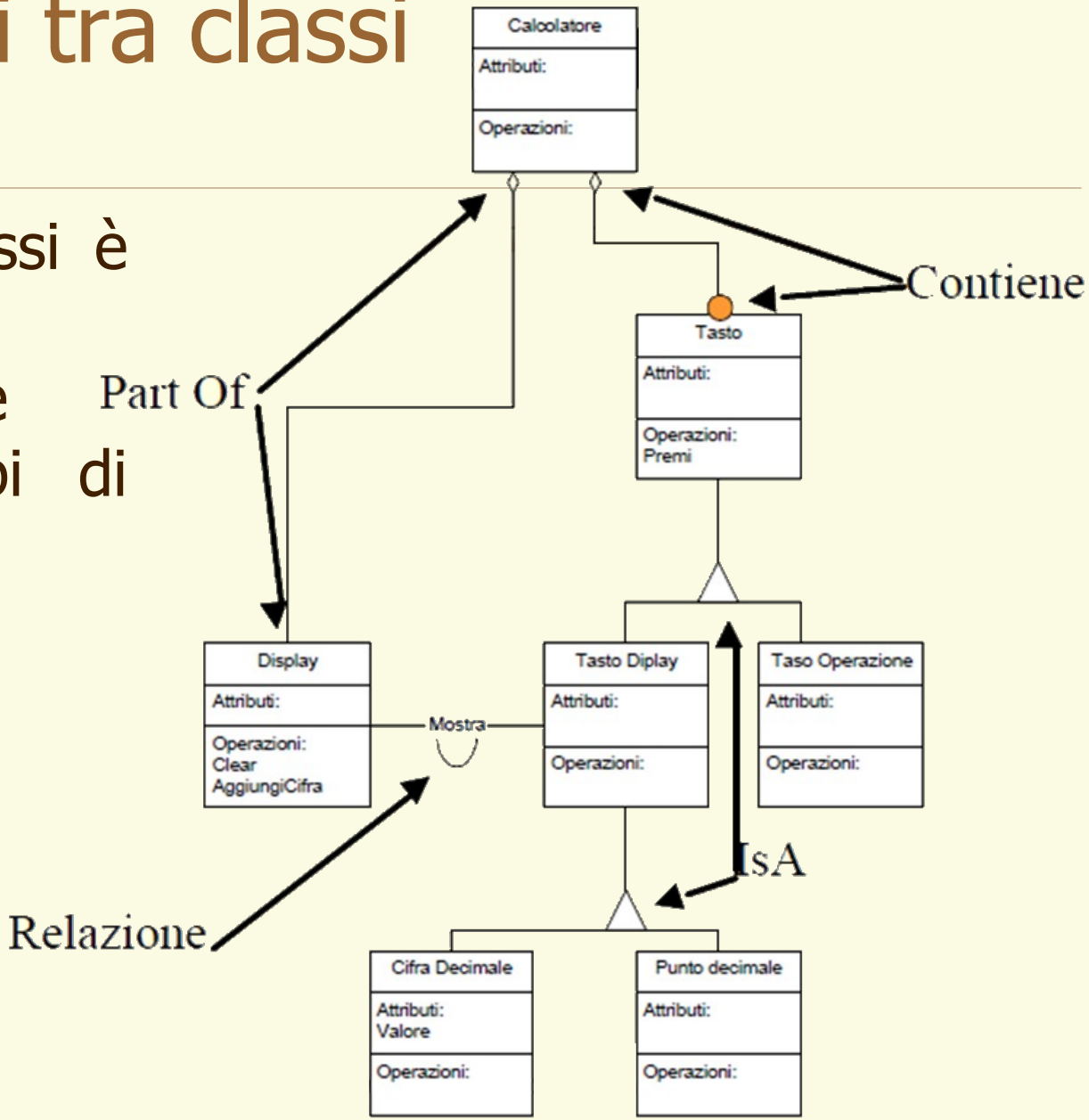
1. Esaminare la realtà, ed individuare gli elementi essenziali (protagonisti) della stessa
2. Differenziare i ruoli: gli elementi importanti diventeranno classi, quelli meno diventeranno attributi; le azioni che gli oggetti possono fare o subire diventano invece metodi. Inizialmente, si devono solo individuare classi, attributi e metodi
3. Se qualche classe dovrà possedere attributi e/o metodi già posseduti da altre, sfruttare il meccanismo di ereditarietà
4. Stabilire il livello di protezione degli attributi e gli eventuali metodi per la gestione degli stessi (metodi probabilmente necessari in caso di attributi privati); occorre anche stabilire il livello di protezione dei metodi.
5. Stesura dei costruttori delle classi, per decidere qual è lo stato iniziale degli oggetti
6. Implementazione dei metodi

Relazioni tra classi

- **Associazioni**
- **Generalizzazioni o specializzazione**
- **Aggregazioni**
- **Composizioni**
- **Dipendenza**

Relazioni tra classi

- Tra le classi è possibile individuare diversi tipi di relazione



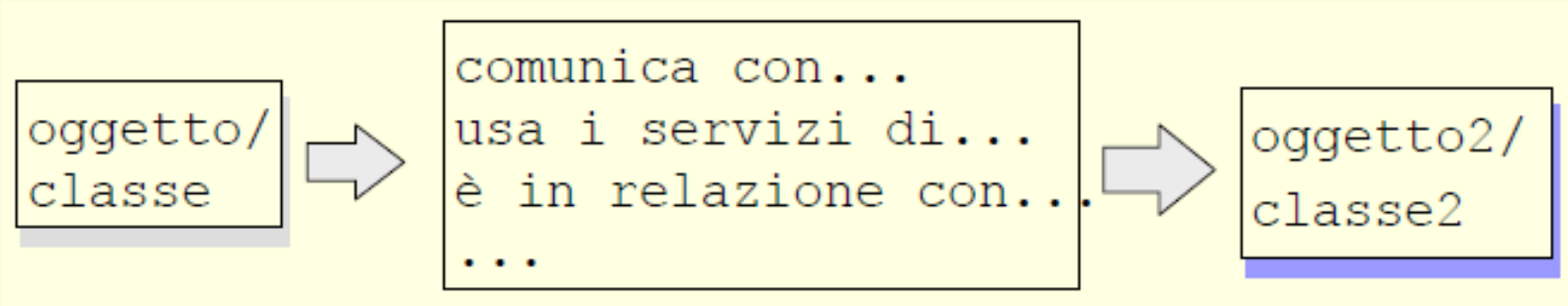
Relazioni tra classi
Associazioni

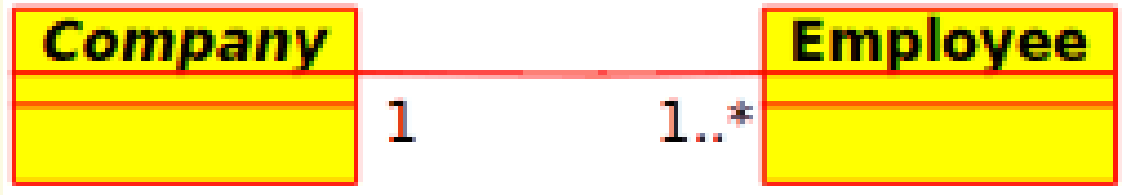
- Relazione tra due o più classi che specifica la presenza di connessioni tra le corrispondenti istanze (oggetti)
- Implica un'iterazione tra gli oggetti o classi, che *"riconoscono"* gli oggetti all'altro estremo: diciamo che una classe A *utilizza* una classe B se un oggetto della classe A è in grado di inviare dei messaggi ad un oggetto di classe B oppure se un oggetto di classe A può *creare, ricevere o restituire* oggetti di classe B.
- Una classe è associata ad un'altra se è possibile "navigare" da oggetti della prima classe ad oggetti della seconda classe seguendo semplicemente un riferimento ad un oggetto.

Relazioni tra classi
Associazioni

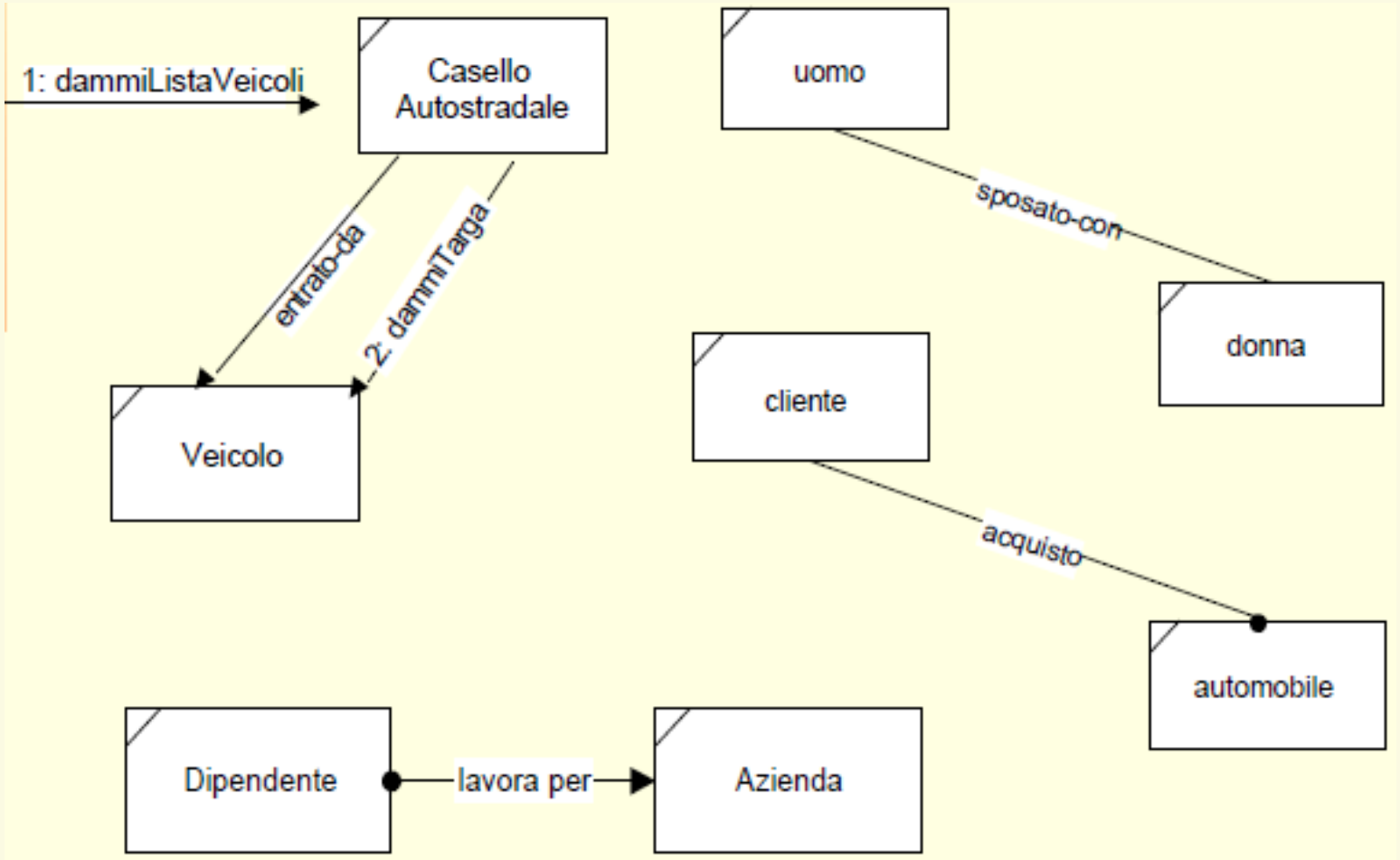
- Un'associazione può essere indicata con un nome o semplicemente con i nomi dei ruoli. Sia nome che ruoli sono facoltativi
- La freccetta accanto al nome indica la direzione di lettura del nome: il termine "impiega" ha senso se si legge l'associazione da destra a sinistra. Si potrebbe invertire la direzione di lettura cambiando anche il nome dell'associazione in "è impiegato"
- La molteplicità è un vincolo la cui funzione è quella di limitare il numero di oggetti di una classe che possono partecipare ad un'associazione in un dato istante

Relazioni tra classi
Associazioni





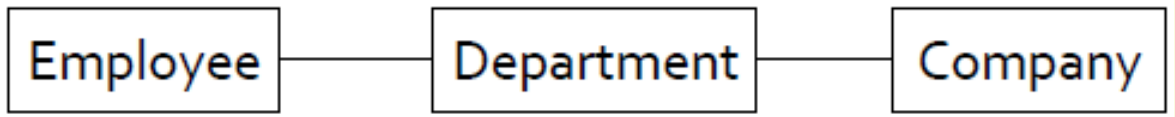
Relazioni tra classi
Associazioni



Relazioni tra classi
Associazione - UML

- L'associazione è rappresentata da una linea continua che può essere
  - Monodirezionale (la classe di partenza è detta *origine, mentre quella di arrivo è detta destinazione)* con freccia
  - Bidirezionale (composta da due associazioni monodirezionali)
- Il numero di istanze legate dall'associazione è specificato dalle *molteplicità agli estremi dell'associazione*

Esempio: "An Employee works in a department of a Company"



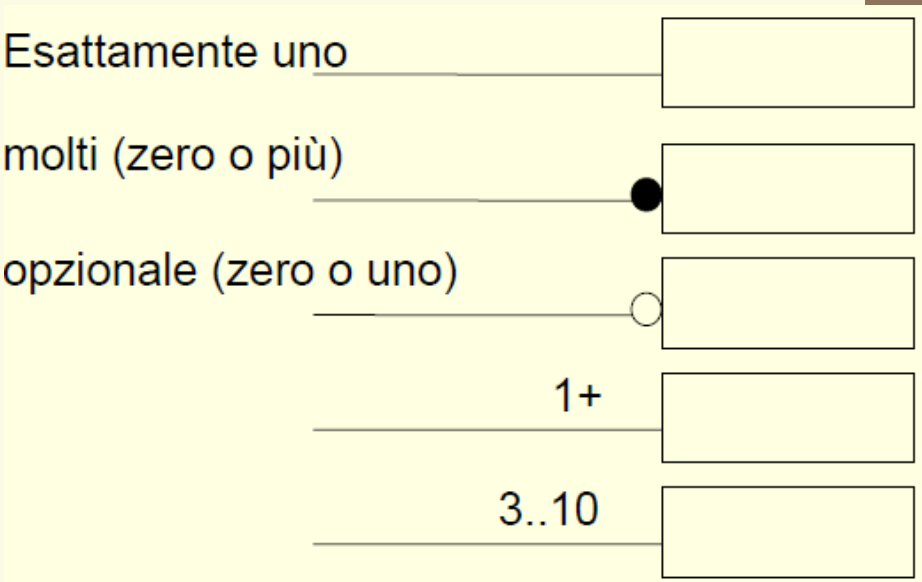
Relazioni tra classi
Associazione - UML

- Il **nome dell'associazione** è posta al centro dell'associazione stessa (solitamente un verbo)
- Un **ruolo** è un'etichetta posta ad (almeno) uno degli estremi di un'associazione
  - Può indicare il ruolo svolto dalla classe a cui si riferisce nell'ambito dell'associazione
  - Solitamente è un nome (talvolta visto come attributo stesso della classe collegata)
  - Obbligatorio solo per relazioni di associazione riflessive

Relazioni tra classi
Cardinalità o molteplicità

- Definisce il numero di oggetti che per ogni classe partecipano alla relazione stessa

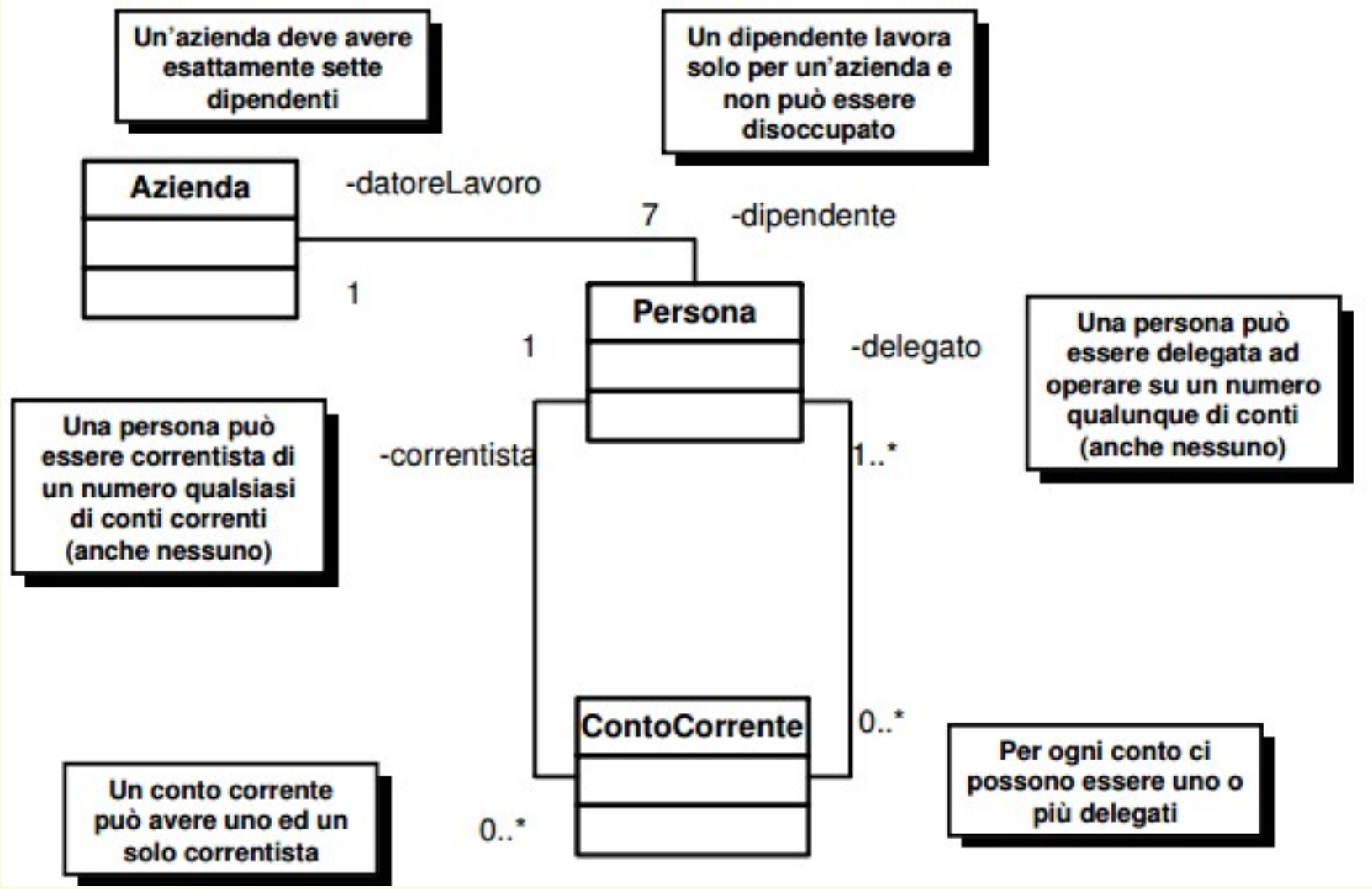
  - Uno (1)
  - Zero o uno (0..1)
  - Zero o più * o (0..*)
  - Da zero a N (0..N)
  - Uno o più (1..*)
  - Da 1 a N (1..N)
  - Da N a M (N..M)



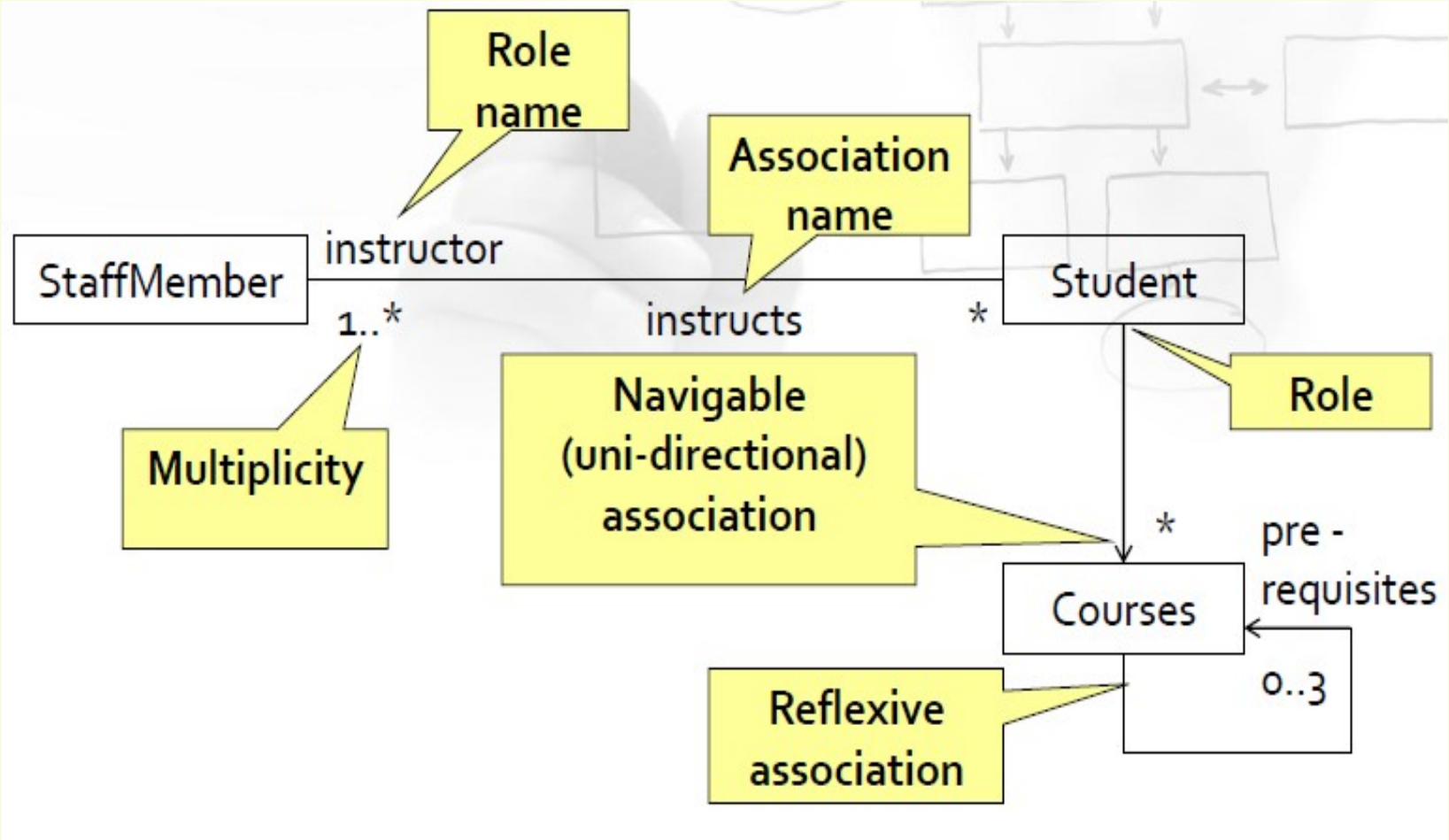
Relazioni tra classi
Cardinalità o molteplicità

- Indica il *numero di istanze della classe più vicina che possono esistere a run-time in una configurazione valida del sistema …*
- … che sono riferite da una *singola istanza della classe che sta all'estremo opposto della relazione*
- Specifica se un'associazione è obbligatoria o meno
- Fornisce un *intervallo di validità (estremo inferiore ed estremo superiore) relativamente al numero di istanze contemporaneamente presenti e legate a ciascun ruolo dell'associazione in un dato momento*

Relazioni tra classi
Associazione - UML

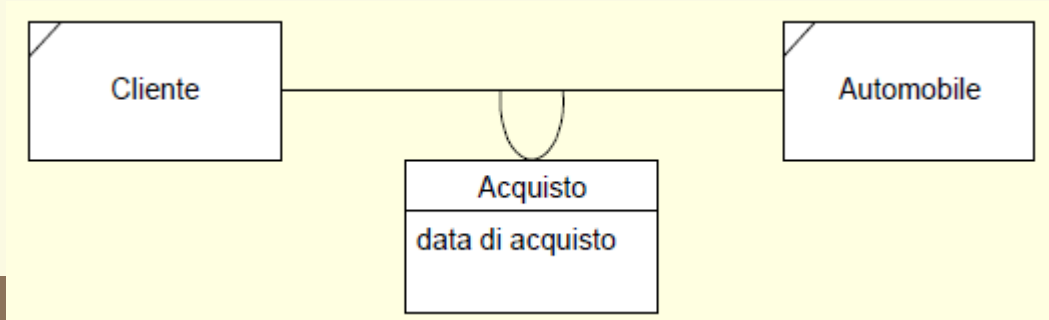


Relazioni tra classi
Associazione - UML



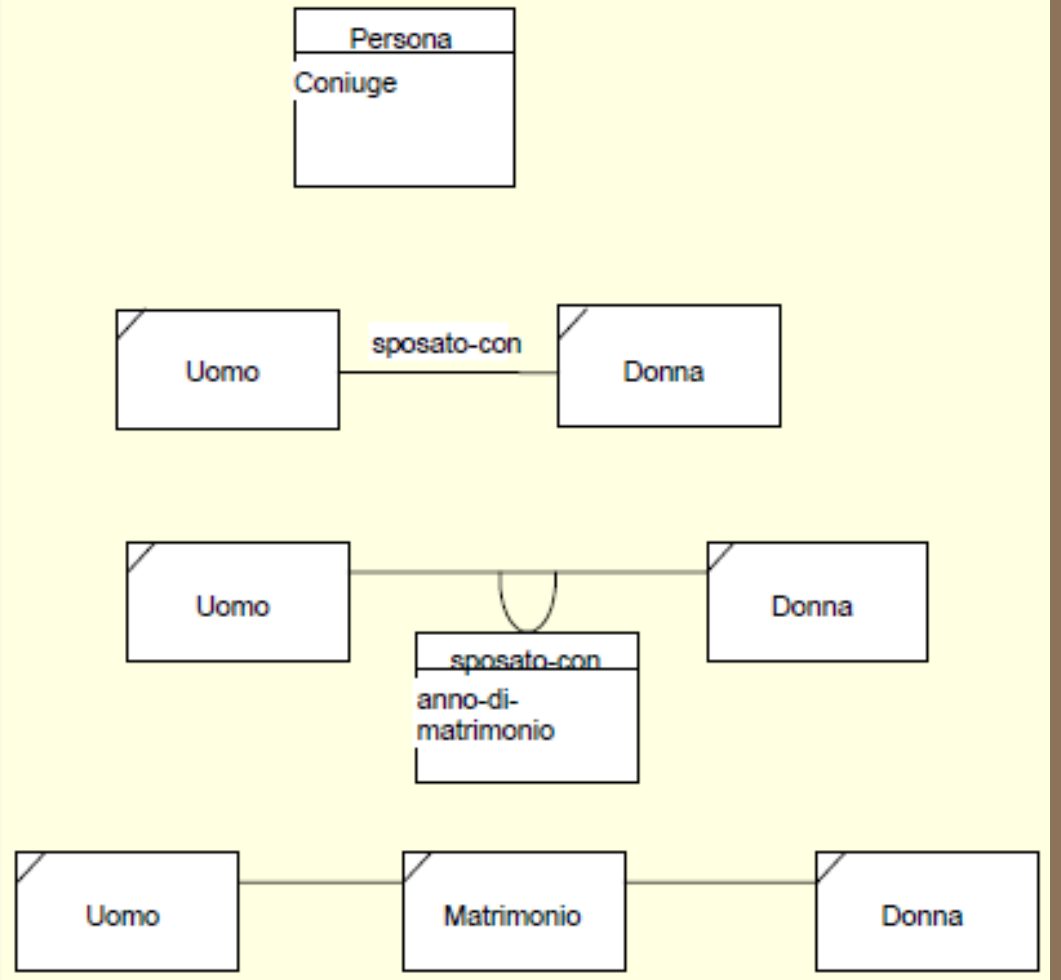
Relazioni tra classi
Classe associativa: attributi

- Per poter aggiungere attributi ad una associazione piuttosto che alle classi coinvoltesi usa la classe associativa, ossia una classe derivata da una associazione.
- Le classi associative sottintendono un vincolo aggiuntivo, ossia il fatto che ci può essere solo un'istanza della classe di associazione fra ogni coppia di oggetti associati.



Relazioni tra classi
Associazione, attributo o classe?

- Quale fare?

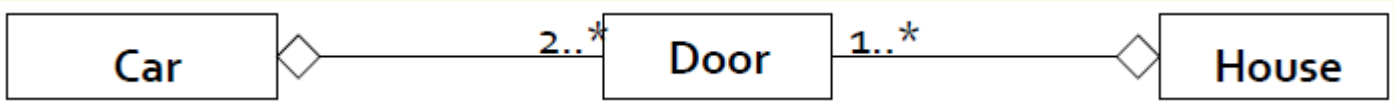


Relazioni tra classi Generalizzazione-Specializzazione

- È una associazione di tipo **IS-A** (in UML si usa una freccia continua)
- Si basa sul concetto di **ereditarietà**: un oggetto di classe A deriva da un oggetto di classe B se A è in grado di compiere tutte le azioni che l'oggetto B è in grado di compiere.
- In più A è una specializzazione e B è una generalizzazione se l'oggetto di classe A è in grado di eseguire anche azioni che l'oggetto B non può compiere.

Relazioni tra classi
Aggregazione (lasca)

- È un associazione più forte, di tipo “intero-parte”. Indica "contiene", **"è parte di"**, “è un insieme di”.
- È anche sinonimo di (possibile) condivisione; semantica del contenimento “by reference” (puntatori) (vettori).
  - Un oggetto di classe A contiene un oggetto di classe B se B è una proprietà (attributo) di A (c'è un riferimento a B)
- Gli oggetti aggreganti (parti) possono appartenere a più di un oggetto aggregato il quale a sua volta può esistere indipendentemente dalle parti
- La classe che fa da intero ha molteplicità 1 o 0.



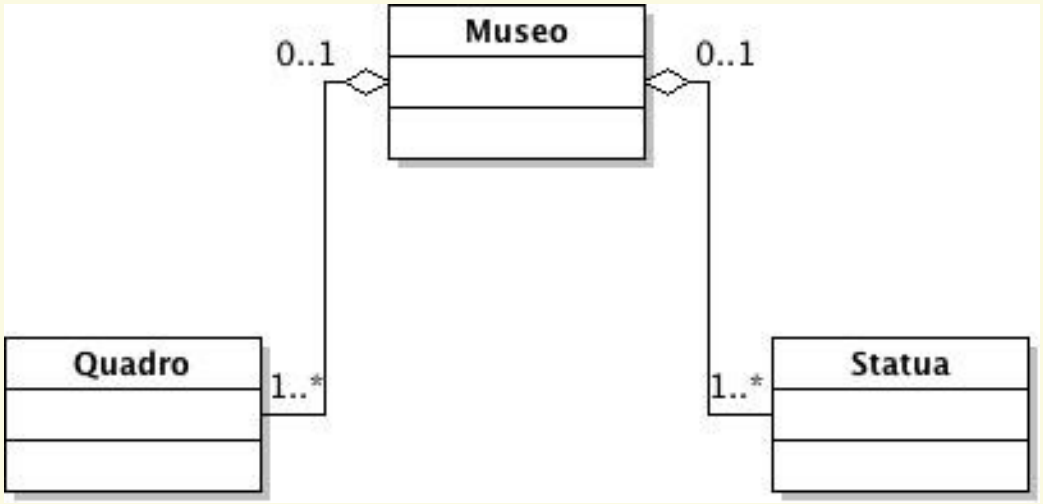
Relazioni tra classi
Aggregazione (lasca)

Domande per capire se è un'aggregazione lasca:

- L'espressione "è-parte-di" ("part of") viene usata per descrivere questa relazione?
  - A door is "part of" a car
- Esistono alcune operazioni del "tutto" che possono essere applicate alle singole "parti"?
  - Move the car, move the door.
- Esistono particolari valori di attributi che possono essere propagati dal "tutto" alle "parti"?
  - The car is blue, therefore the door is blue.
- Esiste un'intrinseca asimmetria nella relazione per cui una classe è subordinata ad un'altra?
  - A door is part of a car. A car is not part of a door.

Relazioni tra classi
Aggregazione (lasca)

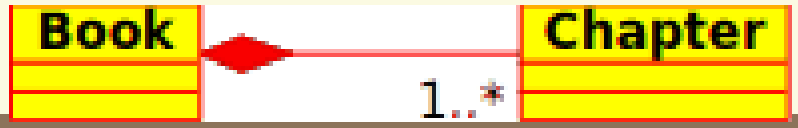
Esempi:



Un'automobile ha un particolare tipo di motore e quattro ruote di un certo tipo. Sia il tipo di motore che il tipo di ruota continuano ad avere dignità ed esistenza propria anche al di là dell'oggetto automobile. La distruzione dell'automobile, non comporta automaticamente quella delle sue parti …

Relazioni tra classi
Composizione (aggregazione stretta)

- È una forma di aggregazione ancora più forte (**HAS–A**) che indica che una "parte" può appartenere ad un solo "intero" in un certo istante di tempo, la parte non può esistere di per sé. (record)
- La composizione associa composto e componente per tutta la vita dei due oggetti
- La composizione è esclusiva: uno specifico oggetto componente non può appartenere a due composti contemporaneamente



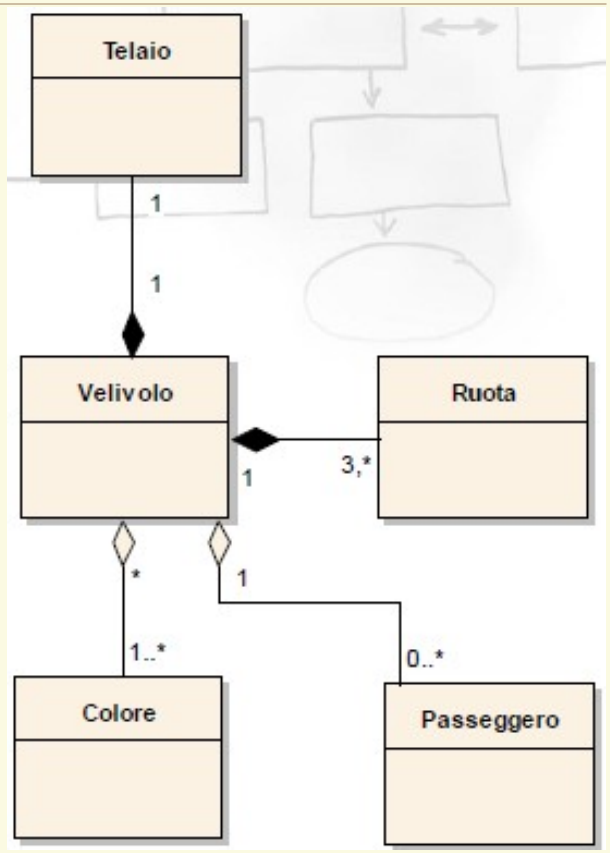
Relazioni tra classi
Composizione (aggregazione stretta)

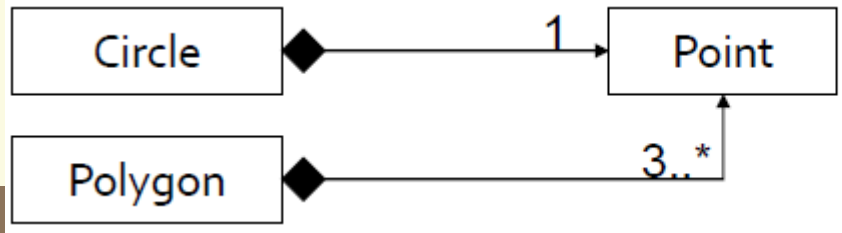
- Il "tutto", detto "composto", è il solo proprietario (owner) delle proprie parti, dette "componenti"
  - L'oggetto "parte" appartiene al più ad un unico "tutto".
  - La molteplicità dalla parte del "tutto" può essere solo 0 oppure 1
- Il tempo di vita (life-time) della "parte" è subordinato (minore o uguale) a quello del "tutto"
  - L'oggetto "composto" gestisce la creazione e la distruzione delle sue parti (ownership completa)
  - Le parti esistono solo all'interno dell'intero, e se l'intero è distrutto anche le parti muoiono. La composizione governa il periodo di vita delle classi parti.

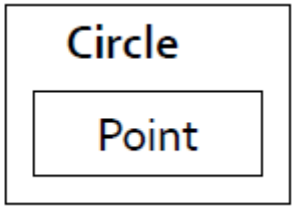
Relazioni tra classi

Composizione (aggregazione stretta)

- Ad esempio, un capitolo può far parte di un solo libro, mentre, al contrario, una persona potrebbe lavorare contemporaneamente per due ditte o ancora un modello di pneumatico potrebbe essere utilizzato su più modelli di auto, ma quel veicolo ha montate solo quelle particolari ruote fisiche



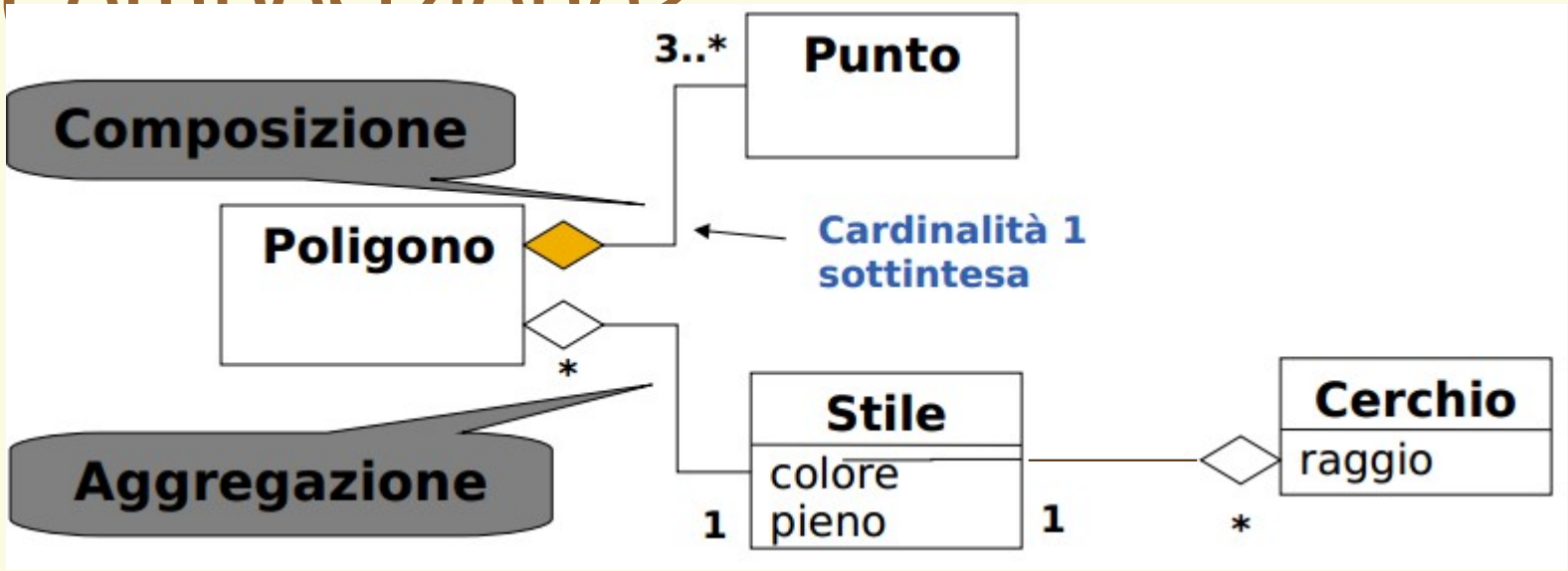




Relazioni tra classi
Aggregazione o Composizione?

- L'oggetto aggregato/composto ha senso di esistere, ha uno scopo, se l'oggetto aggregante/componente non esiste?
  - Sì: aggregazione, non necessariamente l'oggetto aggregato deve ricevere l'oggetto aggregante dall'esterno, può anche crearne uno di default che poi sia successivamente sostituibile
  - No: composizione
- Se si elimina oggetto composto si eliminano anche le parti?
  - Sì: composizione
  - No: aggregazione

Relazioni tra classi
Aggregazione o Composizione?



- La cancellazione di una istanza della classe Poligono viene estesa ad ogni suo Punto, ma non allo Stile ad esso associato
- Un'istanza della classe Stile può, invece, essere condivisa tra Poligono e Cerchio

Relazioni tra classi
Derivazione o Composizione?

- È un'associazione 1 a 1?
  - No: non può essere una derivazione

Relazioni tra classi
Dipendenza

- Rappresenta una relazione tra due classi, nonostante non ci sia un'esplicita associazione tra esse: una classe **dipende** da un'altra quando esiste un riferimento della seconda nella prima (non è possibile compilare la prima senza la seconda). Per esempio:
  - Una classe "nomina" al suo interno una classe esterna
  - Passaggio di parametri all'interno di un metodo
  - Tipo di ritorno di un metodo
  - Creazione o distruzione
  - Eccezioni
  - Il tipo più comune è la relazione d'uso che esprime un rapporto client-server fra due classi o fra una classe e un'interfaccia

Relazioni tra classi
Dipendenza

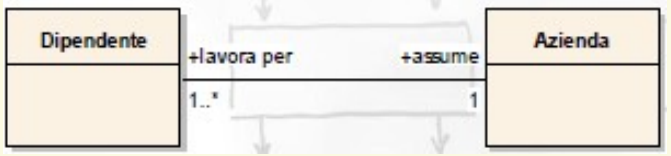
- Una relazione d'uso fra due classi si ha se:

1. Un metodo di Class1 ha un parametro di tipo Class2
2. Un metodo di Class1 restituisce un valore di tipo Class2
3. Un metodo di Class1 usa un oggetto di tipo Class2 ma non come attributo. (per es. quando in un metodo di Class1 si dichiara una variabile locale di tipo Class2)

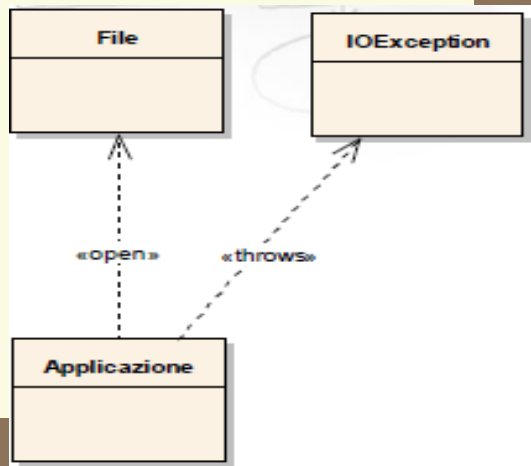
- Nel caso di una classe e di un'interfaccia la relazione d'uso ha un significato più generico: indica che la classe invoca uno o più metodi definiti nell'interfaccia. A volte si parla di relazione *client-of*

Relazioni tra classi
Dipendenza o associazione?

- L'**associazione** esprime un legame di ruoli, tipicamente di natura strutturale (quindi non transiente ovvero non variabile nel tempo)
  - Il dipendente, nel suo ruolo, è perennemente legato all'azienda per la quale lavora.



- La **dipendenza** esprime tipicamente un generico legame di natura spesso transiente, senza ulteriore semantica più forte
  - L'eccezione viene collegata alla classe *Applicazione* solo qualora "scatti".
  - Lo «stereotipo» nella dipendenza definisce la semantica intuitiva della dipendenza

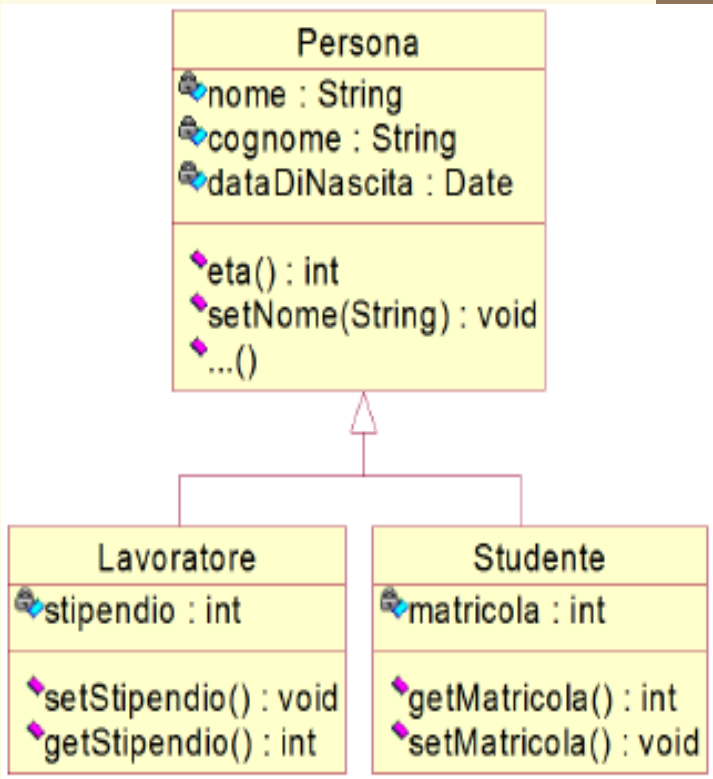


Relazioni tra classi
Esempi di analisi

Si vuole gestire un'anagrafica degli studenti e dei lavoratori di una città

Primo approccio: gerarchia Persona←Lavoratore e Persona←Studente

- Cosa succede quando una persona smette di studiare ed inizia a lavorare?
- … devo creare una nuova istanza di Lavoratore, copiare lo stato dell'istanza di Studente, "valorizzare" la parte di stato specifica di Lavoratore (e.g. stipendio), cancellare la vecchia istanza di Studente …
- … ma adesso ho un'istanza diversa dalla precedente, però la persona è sempre la stessa!

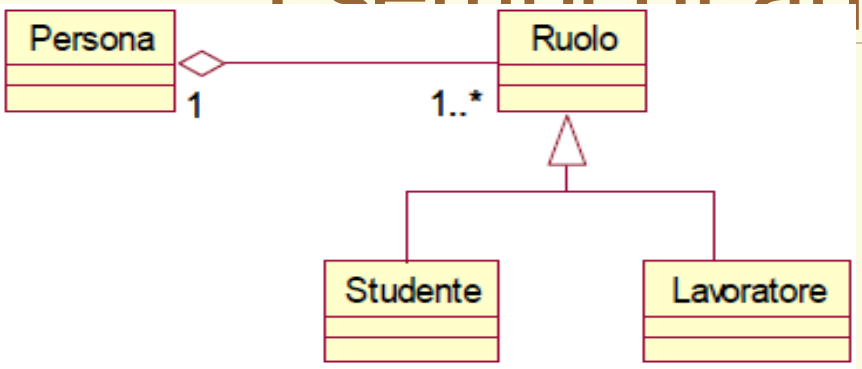


Relazioni tra classi
Esempi di analisi

- Quali problemi emergono attraverso questo approccio? Se Andrea è uno studente lavoratore?
  - Due oggetti diversi e "vivi" (andrea1 e andrea2) per un unico esemplare di Persona (identità) (andrea)
- Possibili incoerenze (anche in fase di manutenzione):
  - Ho copiato correttamente lo stato di andrea1 nello stato di andrea2?
  - Ho inizializzato completamente andrea2?
  - Una volta diventato solo più Lavoratore, il numero di matricola di andrea1 non dovrebbe essere più valido. Chi lo garantisce? Ossia chi mi garantisce che il programmatore si ricordi sempre anche di cancellare la vecchia istanza?

L'ereditarietà è una relazione statica!

Relazioni tra classi
Esempi di analisi



- Come risolvere il problema? Capendo che ciò che sto modellando non è l'identità di una Persona, bensì il suo Ruolo!
- Linea guida: preferire sempre il contenimento all'ereditarietà.
- Vantaggi:
  - Una persona può svolgere più ruoli contemporaneamente (studente e lavoratore)
  - Una persona può cambiare ruolo dinamicamente

Traduzione in Java di UML aggregazione lasca

- Comporta l'indipendenza del ciclo di vita dell'oggetto contenuto dall'oggetto contenitore. L'oggetto contenuto potrà quindi esistere anche indipendentemente dal contenitore. Il contenitore **non** ha responsabilità per la creazione e distruzione dell'oggetto contenuto
- Il contenitore dovrà definire un costruttore che riceva in input il puntatore (la maniglia) all'oggetto contenuto e ha un attributo privato di tipo riferimento ad un oggetto di tipo contenuto.
- Il client della classe contenitore deve:
  - Istanziare l'oggetto contenuto e ottenere la maniglia
  - Istanziare l'oggetto contenitore passando la maniglia al contenuto

Traduzione in Java di UML aggregazione lasca

```
class Contenitore {
private Contenuto contenuto; .....
     public Contenitore(Contenuto q) {
     contenuto=q;
};
public met(){//richiama metodi di Contenuto
contenuto.metContenuto();…}
.......}
public static void main (...) {
Contenuto x= new  Contenuto(3);
Contenitore c=new  Contenitore(x);
//usa classe C
...
}
```

Traduzione in Java di UML aggregazione stretta

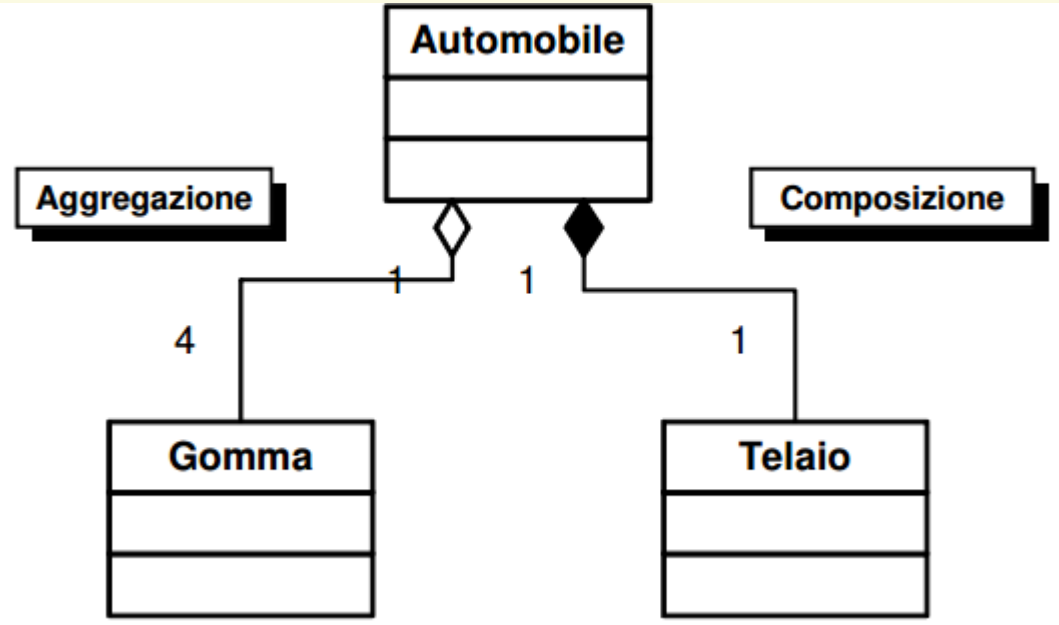
- L'oggetto "contenitore" è **responsabile** della **costruzione** e **distruzione** dell'oggetto contenuto
- Si deve aggiungere un attributo privato della classe "contenuto" che viene istanziato:
  - nel costruttore del "contenitore" oppure
  - prima del costruttore, nella classe "contenitore", cioè nel momento in cui è definito

Traduzione in Java di UML aggregazione stretta

```
public class Contenitore {
   private  Contenuto contenuto;
   .......
   public  Contenitore(int val) {
      contenuto = new  Contenuto(val);…}
}
//se non è possibile fornire un argomento al
  costruttore
public class  Contenitore  {
   private Contenuto contenuto = new Contenuto();
   .......
   public  Contenitore() {…};
}
```

Traduzione in Java di UML aggregazione e composizione

Un esempio classico per comprendere la differenza è quello dell'automobile: sia il telaio che le gomme sono parti dell'auto, ma la relazione con il telaio è più stretta: il telaio può appartenere ad una sola auto e nasce e muore assieme all'auto



Traduzione in Java di UML aggregazione e composizione

```
class Automobile {
    private Gomma gomme[];
    private Telaio telaio;
    public Automobile() {
        telaio = new Telaio(); ... }
    public Gomma getGomma(int n) {
       return Gomme[n]; }
    public void setGomma(Gomma g; int n) {
       gomme[n] = g; }
}
```

Gestione tempo di vita

Non c'è setTelaio(): non è possibile cambiare il telaio di un auto

Traduzione in Java di UML associazione

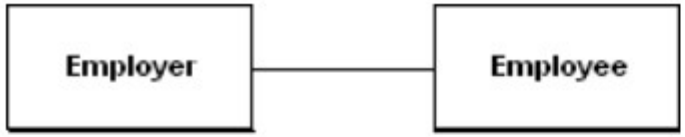
È caratterizzata da:

- un nome (opzionale) che esprime il legame semantico tra le classi associate;
- il ruolo giocato dalle parti associate (opzionale);
- la molteplicità dell'associazione che esprime la cardinalità delle connessioni tra gli oggetti:
  - uno a uno
  - uno a molti
  - molti a molti
- la direzionalità (bidirezionale di default) indica chi ha la responsabilità di tenere traccia dell'associazione:
  - bidirezionale responsabilità multipla
  - unidirezionale responsabilità singola

Traduzione in Java di UML associazione

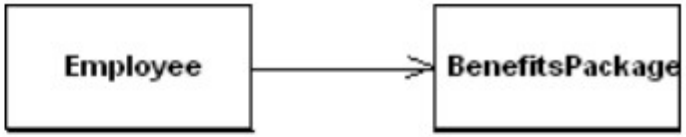
Esempio di:

- associazione bidirezionale: ogni classe ha un riferimento all'altra
- associazione unidirezionale, solo la classe da cui parte la freccia ha il riferimento all'altra




```
class Employer
{
  // ...
  private Employee itsWorker;
}

class Employee
{
  // ...
  private Employer itsBoss;
}
```

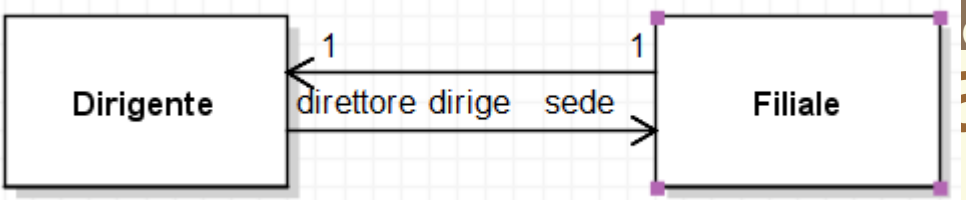



```
class Employee
{
  // ...
  private BenefitsPackage itsBenefits;
}

class BenefitsPackage
{
  // ... Non c'è un riferimento
    all'immpiegato
}
```

Traduzione in Java di UML associazione

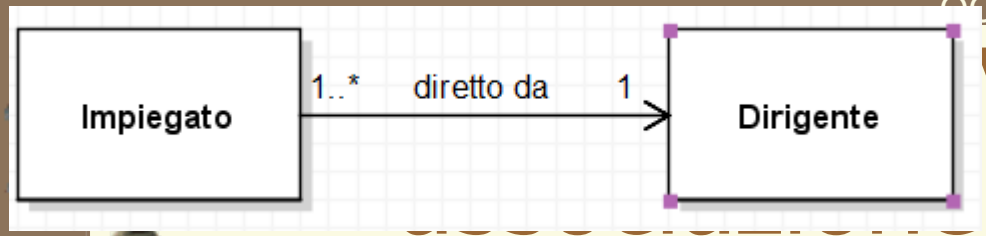
- Se è specificato solo il nome dell'associazione la variabile membro di tipo puntatore prende il nome dell'associazione stessa.
- Se è specificato il ruolo della classe associata il nome della variabile membro sarà quello del ruolo
- Il programma client avrà la responsabilità della creazione e del collegamento dei due oggetti



ava di UML es. associazione 1:1 bidirezionale

```
class Filiale{
  private Dirigente direttore; …
  public void direttaDa(Dirigente d){direttore=d;}
…}
class Dirigente{
  private Filiale sede; …
  public voiddirige(Filiale f){sede=f;}
…}

//nella classe client si avrà
Dirigente d=new Dirigente(…);
Filiale f=new Filiale(…);
//collega i due oggetti.
d.dirige(f);
f.direttaDa(d);
```

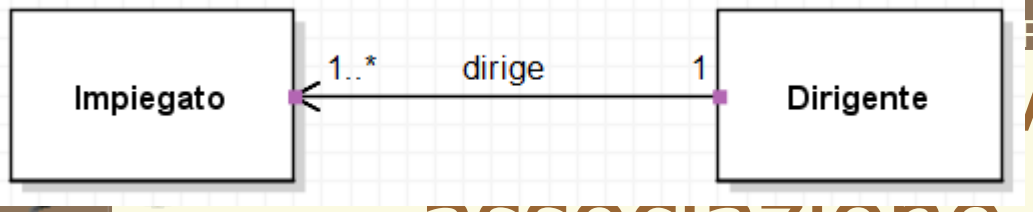


va di UML
1:N
unidirezionale

- Se a responsabilità singola VERSO 1 senza attributi

```
public class Impiegato{
    private String nome;
    private Dirigente direttoDa;
    public Impiegato(String n, Dirigente d) {
      nome = n; direttoDa = d;}
    public String getNome() { return nome; }
    public void associaDir(Dirigente d) {
      if (d!= null) direttoDa = d; }
}
```

- Se l'impiegato potesse non avere nessun dirigente, nel costruttore non si passerebbe il dirigente, resterebbe solo `associaDir()`

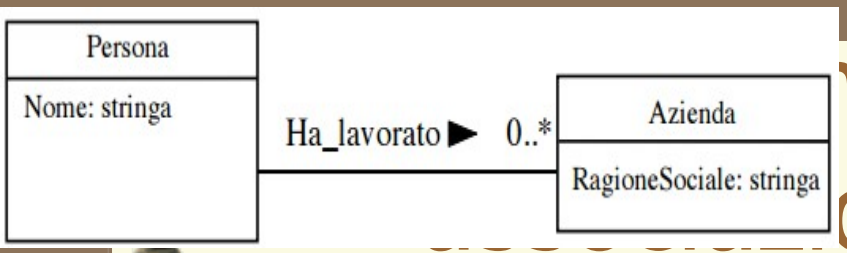


va di UML

associazione 1:N unidirezionale

- Se a responsabilità singola VERSO N senza attributi

```
public class Dirigente{
    private String nome;
    private Vector <Impiegato> sottoposti;
    public Dirigente(String n) {
      nome = n; sottoposti = new Vector<Impiegato>(); }
    public String getNome() { return nome; }
    public void addSottoposto(Impiegato i) {
      if (i != null) sottoposti.add(i); }
    public void eliminaSottoposto(Impiegato i) {
      if (i != null) {
      int k= sottoposti.indexOf(i);
      if (k!=-1) sottoposti.removeElementAt(k);}
     }
}
```



n Java di UML
one N:N unidirezionale

- Se a responsabilità singola senza attributi

```
public class Persona {
    private String nome;
    private Vector <Azienda> haLavorato;
    public Persona(String n) {
      nome = n; haLavorato = new Vector<Azienda>(); }
    public String getNome() { return nome; }
    public void inserisciLinkHaLavorato(Azienda az) {
      if (az != null) haLavorato.add(az); }
    public void eliminaLinkHaLavorato(Azienda az) {
      if (az != null) {
      int i=haLavorato.indexOf(az);
      if (i!=-1) haLavorato.removeElementAt(i);}
     }
}
```

Traduzione in Java di UML

- Per rappresentare l'associazione A con attributi fra le classi UML C e D si introduce un'ulteriore classe Java TipoLinkA. Ci sarà un oggetto di classe TipoLinkA per ogni link fra un oggetto di classe C ed uno di classe D.
- La classe Java TipoLinkA conterrà:
  - gli attributi dell'associazione;
  - i riferimenti agli oggetti delle classi C e D che costituiscono le componenti della tupla che il link rappresenta

Per approfondimenti vedi dispense "da UML a Java.pdf"