



# **UML 2.0: CLASS DIAGRAM (+ OBJECT DIAGRAM)**

**Ingegneria del Software a.a. 2024-25**

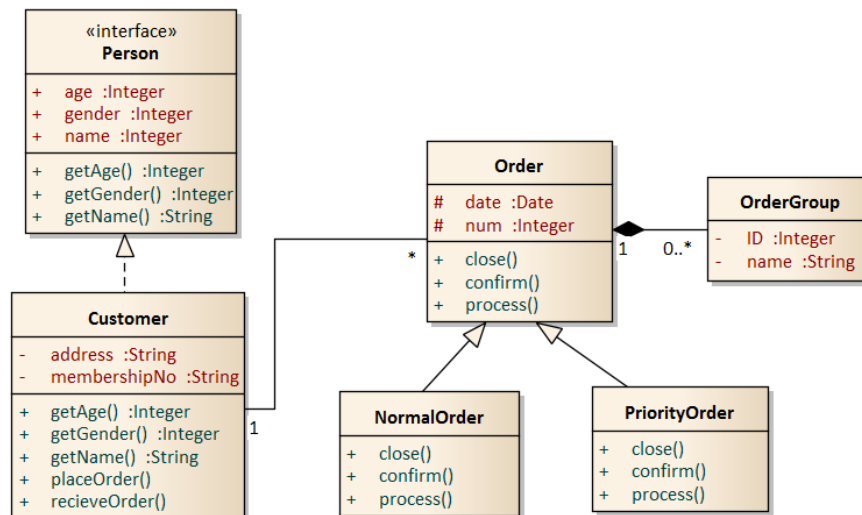
- Se qualcuno vi avvicinasse in un **vicolo buio** dicendo ...



Psst, vuoi  
vedere un  
diagramma  
UML?



- Molto probabilmente ...

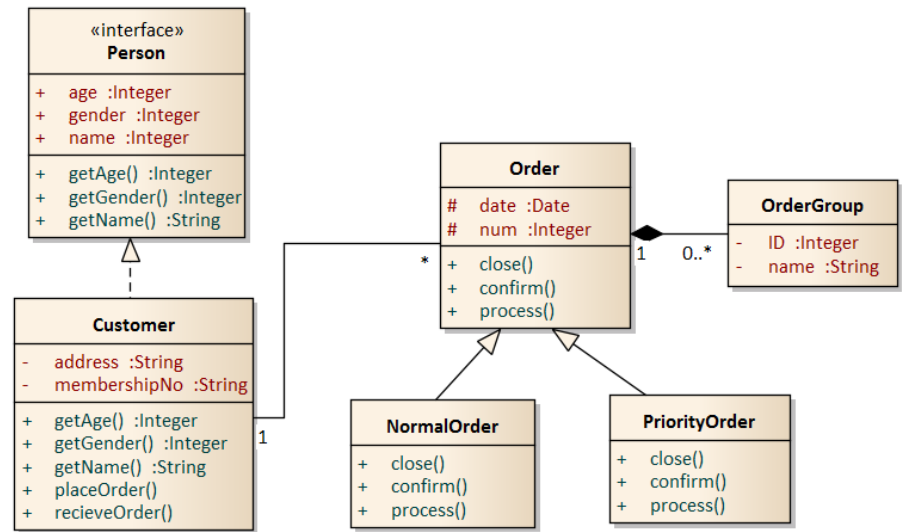


*È il diagramma UML più diffuso e conosciuto ...*

# CLASS DIAGRAM

- Definisce:

- le classi
- le loro **feature** (terminologia UML)
  - attributi
  - operazioni** (in UML esistono anche i metodi, ma sono altro)
- le relazioni tra classi
  - associazioni
  - aggregazione/composizione
  - specializzazione/generalizzazione
  - dipendenze



# CLASS DIAGRAM

- Definisce:

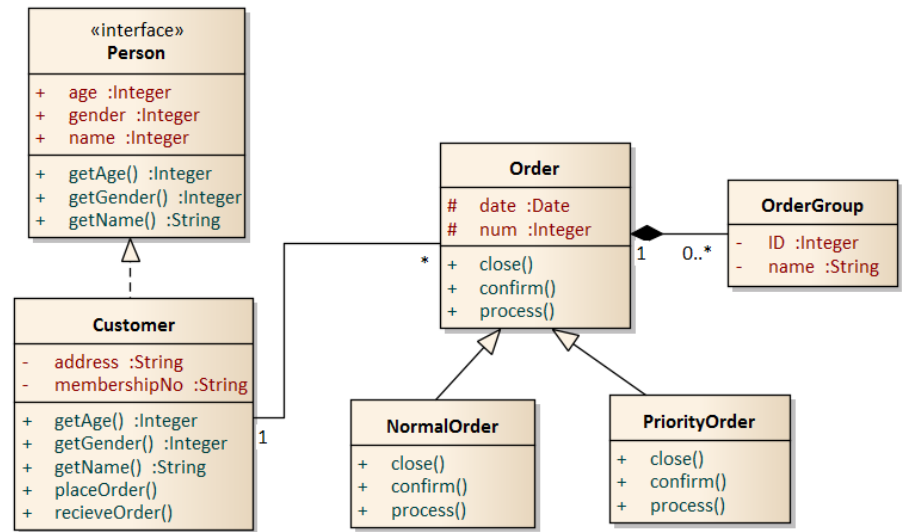
- le classi
- le loro **feature** (terminologia UML)

- attributi

**Ovvero la parte statica di quello che intendiamo modellare...**

Es. La struttura di un sistema software

- le relazioni tra classi
  - associazioni
  - aggregazione/composizione
  - specializzazione/generalizzazione
  - dipendenze



# PROSPETTIVE

- Il significato di un class diagram e dei suoi elementi dipende dalla *prospettiva*:

- **Prospettiva concettuale** *Analista*
  - Descriviamo gli elementi del “pezzo di mondo” che ci interessa modellare
  - Classe UML → concetto proprio del dominio
    - Oggetto di una classe → entità del mondo reale
  - Operazione UML → azione/responsabilità
    - Es. **Dipendente preleva lo stipendio**
- **Prospettiva software** *Sviluppatore*
  - Descriviamo il **design di un software**, ovvero i moduli software che costituiranno l'implementazione vera e propria del sistema
  - Classe UML → classe in un linguaggio OO
  - Operazione UML → implementata da un metodo
    - Es. **Catalog.AddElement (...)**

# PROSPETTIVE

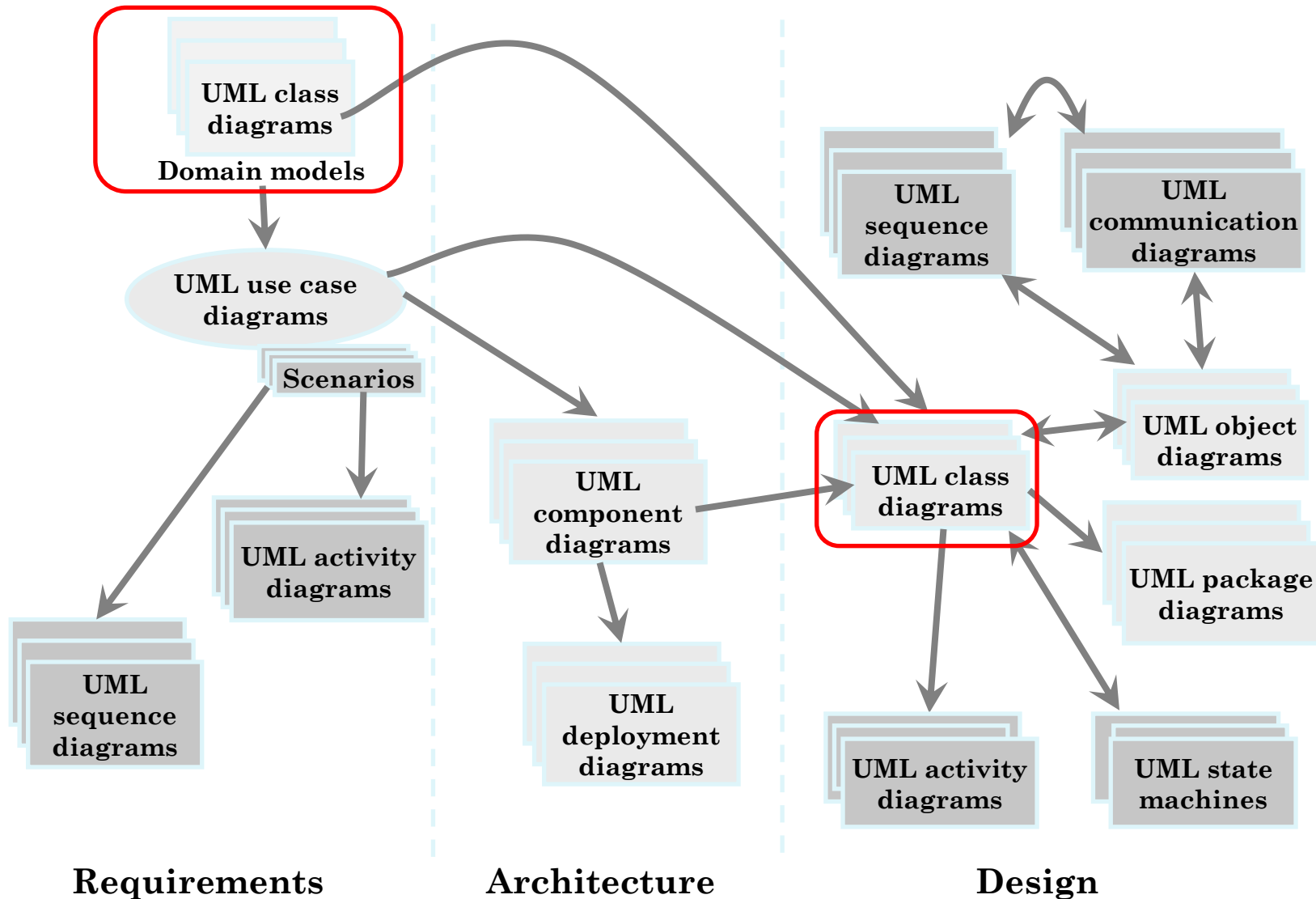
- Il significato di un class diagram e dei suoi elementi dipende dalla *prospettiva*:

- Prospettiva concettuale *Analista*
  - Descriviamo gli elementi del “pezzo di mondo” che ci interessa modellare
  - Classe UML → concetto proprio del dominio

**Attenzione:** nella presentazione odierna passeremo spesso da una prospettiva all'altra ....

- Prospettiva software *Sviluppatore*
  - Descriviamo il **design di un software**, ovvero i moduli software che costituiranno l'implementazione vera e propria del sistema
  - Classe UML → classe in un linguaggio OO
  - Operazione UML → implementata da un metodo
    - Es. **Catalog.AddElement (...)**

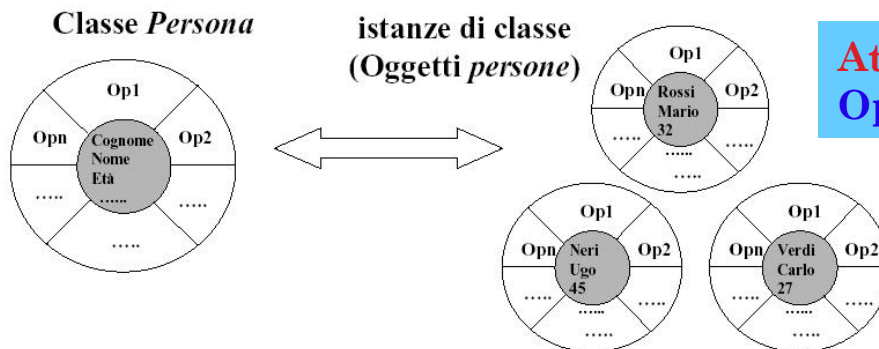
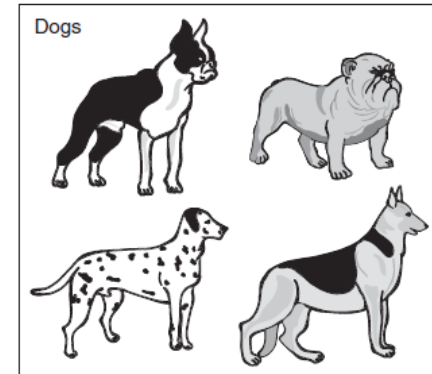
# UML E PROCESSO





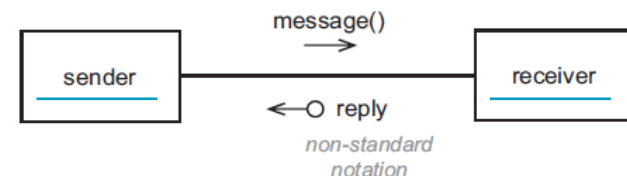
# CONCETTO DI CLASSE IN UML

- Il concetto di **classe** è lo stesso dell'OO
- Una classe “incapsula” caratteristiche comuni ad un gruppo di oggetti
- Una classe genera oggetti → create()
  - new() nel linguaggio Java
- Un **oggetto** è un istanza di una classe



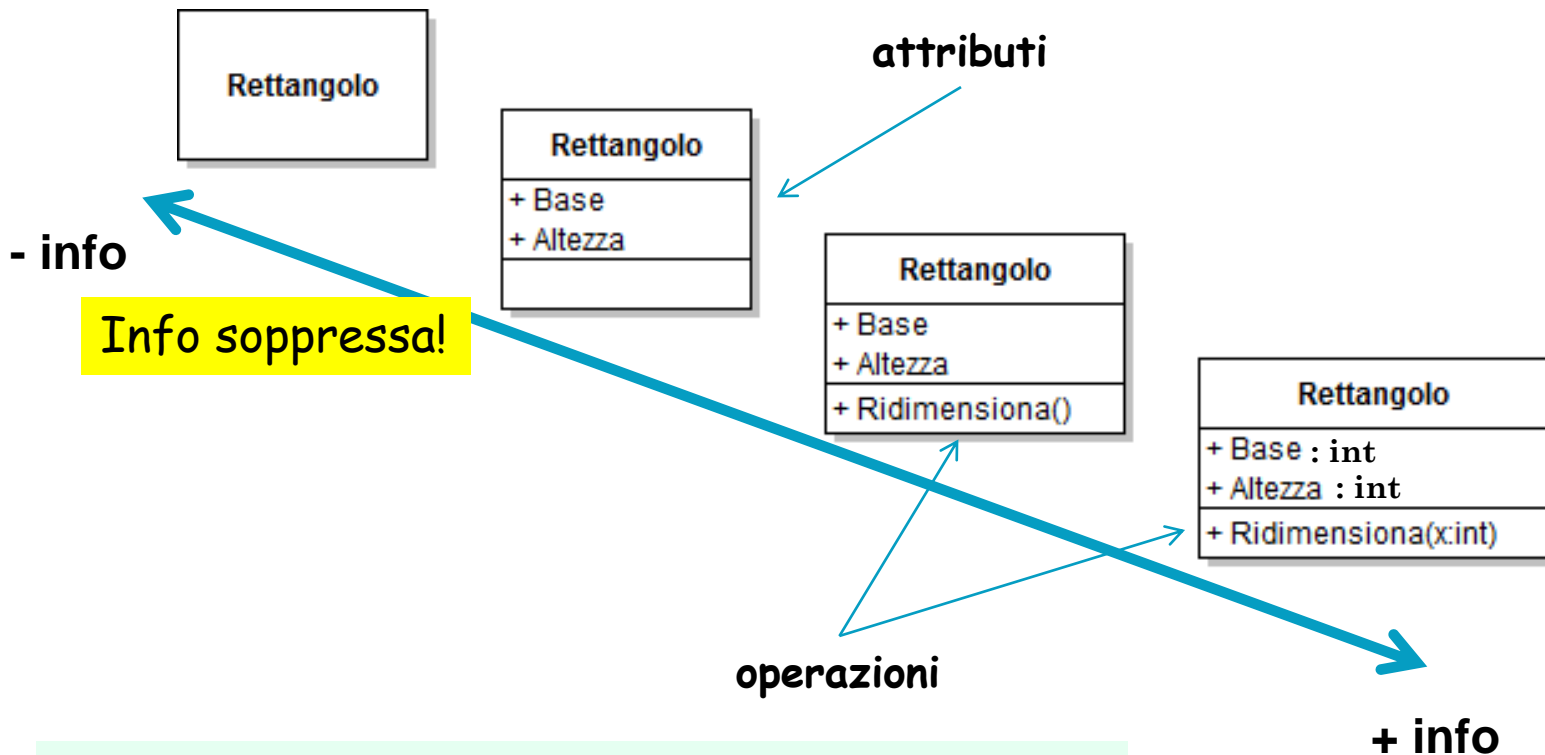
**Attributi:** determinano lo stato degli oggetti  
**Operazioni:** descrivono il comportamento

- Oggetti connessi tra loro possono collaborare per compiere **task + complessi**
  - Mediante **scambio di messaggi**
  - Ovvero chiamando le loro operazioni



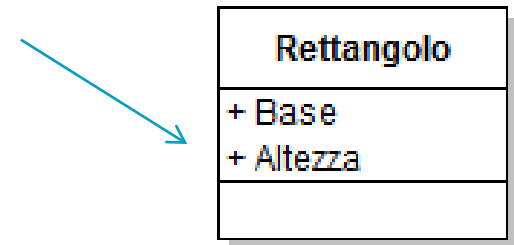
# COME SI RAPPRESENTA UNA CLASSE?

- Una classe in UML è semplicemente rappresentata da un rettangolo con il **nome** della classe all'interno
- Possiamo aggiungere gli **attributi** e le **operazioni**



Es. decido di non mostrare gli attributi in tutto il CD

# ATTRIBUTI

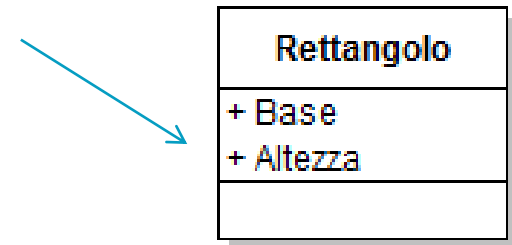


**visibilità nome: tipo [molteplicità] = default {proprietà}**

- Sono consentiti 4 livelli di visibilità (uguale per le op):
  - **+ pubblico**: l'utilizzo viene esteso a tutte le classi
  - **- privato**: solo la classe originale può utilizzare gli attributi
  - **# protetto**: l'utilizzo è consentito soltanto alle sottoclassi e alla classe stessa
  - **~ package**: l'utilizzo è consentito alle classi del package
- Il **nome** dell'attributo è *l'unica parte obbligatoria*
- Il **tipo** dell'attributo può essere:
  - un tipo primitivo (int, double, char, etc...) oppure
  - il nome di una classe definita nello stesso modello
- **Default** rappresenta il valore di default dell'attributo di un oggetto appena creato (se non specificato diversamente durante la creazione)
- **Proprietà** aggiuntive: es. **readOnly**

- titolo: String[1] = "UML distilled" {read only}

# ATTRIBUTI



**visibilità nome: tipo [molteplicità] = default {proprietà}**

- Sono consentiti 4 livelli di visibilità (uguale per le op):

## Warning:

- le regole di visibilità tra i vari linguaggi e UML sono spesso differenti

M. Fowler consiglia:

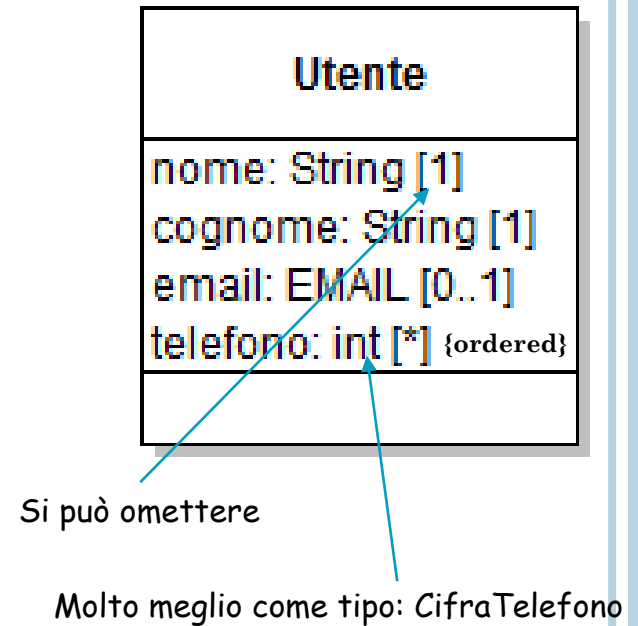
- Usare quelle del linguaggio usato per l'implementazione
- Usare solo "+" e "-"

- un tipo primitivo (int, double, char, etc...) oppure
- il nome di una classe definita nello stesso modello
- **Default** rappresenta il valore di default dell'attributo di un oggetto appena creato (se non specificato diversamente durante la creazione)
- **Proprietà** aggiuntive: es. **readOnly**


- titolo: String[1] = "UML distilled" {read only}

# MOLTEPLICITÀ

- La **molteplicità** indica il **quantitativo** degli attributi
  - ad esempio le dimensioni per una lista
- Alcuni valori possibili sono:
  - 1 (uno e uno solo). È il valore di default
  - 0..1 (al più uno)
  - \* (un numero imprecisato, eventualmente nessuno)
    - equivalente a 0..\*
  - 1..\* (almeno uno)
  - anche **n .. m** ed **m** ( $m \geq 1$ )
- Gli elementi di un attributo con molteplicità  $> 1$  sono considerati come **un insieme**
  - Se essi sono dotati anche di ordine si aggiunge la proprietà {ordered}



# OPERAZIONI



Rettangolo
+ Base
+ Altezza
+ Ridimensiona(x:int)

**visibilità nome (lista parametri) : tipo-ritornato {proprietà}**

- **Visibilità e nome** stesse regole degli attributi
- **Lista parametri** contiene nome e tipo dei parametri, secondo la forma:

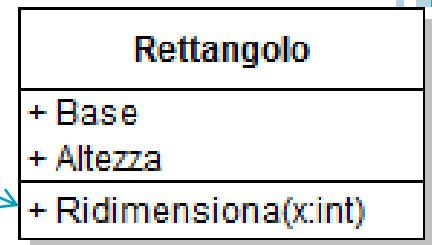
**direzione nome: tipo = default**

- **direzione:** input (*in*), output (*out*) o entrambi (*inout*). Il valore di default è *in*
- **nome, tipo e valore di default** sono analoghi a quelli degli attributi
- **tipo-ritornato** è il tipo del valore di ritorno



**+ saldo(data: Data): Euro**

# OPERAZIONI



**visibilità nome (lista parametri) : tipo-ritornato {proprietà}**

- **Visibilità e nome** stesse regole degli attributi

- **Warning:**

Le operazioni get/set di solito non si indicano  
(E' scontato che esistono!)

**direzione nome: tipo = default**

- In UML esistono due tipi di operazioni:

- **query**: che ottengono un valore da un oggetto senza side effect
- **modificatori**: che modificano gli attributi (lo stato) dell'oggetto su cui sono chiamate

```
classDiagram
    class Data {
        + saldo(data: Data): Euro
    }
```

A UML class diagram for the class **Data**. It has one operation: **+ saldo(data: Data): Euro**. A black arrow points from the text 'In UML esistono due tipi di operazioni:' to this operation.

**+ saldo(data: Data): Euro**

# DATATYPE

- In UML esiste il concetto di **datatype**
  - $\neq$  concetto di oggetto
- Un oggetto per sua natura ha un **identità**



- Mentre un datatype no ...
  - **le istanze sono valori e non oggetti!**
- Esempi: **Date**, **Euro**, ...

10€ = 10€

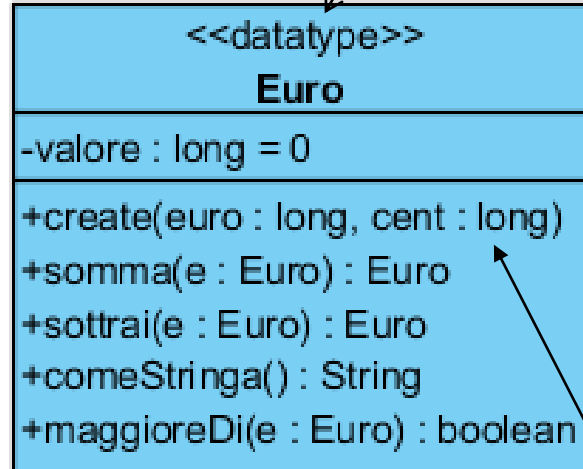
12/10/2019 = 12/10/2019

- Si rappresentano come classi con lo stereotipo **<<datatype>>**



# DATATYPE ESEMPI

## Stereotipo



## Caso speciale di datatype

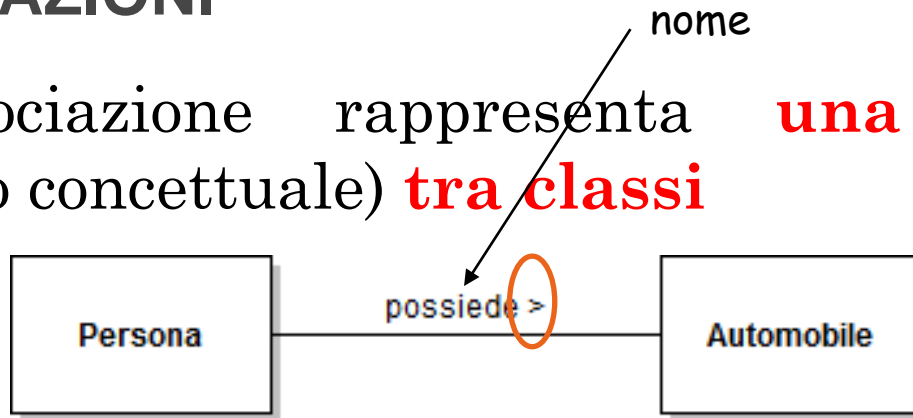


## Costruttore

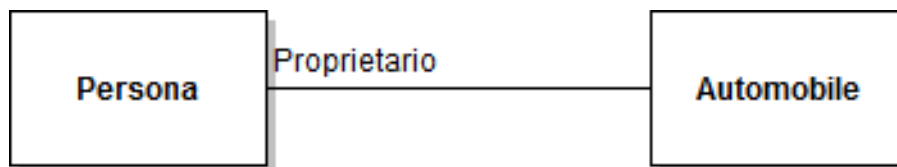
Alle volte si usa anche **make()** o nome della classe

# ASSOCIAZIONI

- Un'associazione rappresenta **una relazione** (fisica o concettuale) **tra classi**

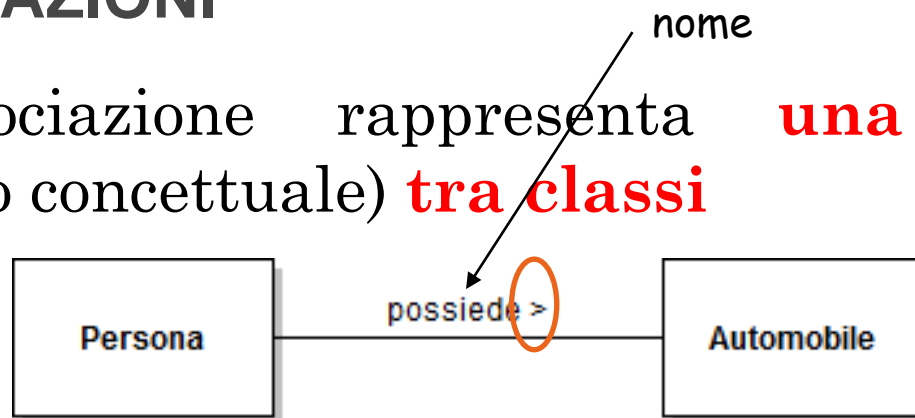


- Il ► indica in che direzione deve essere *letta* l'associazione
  - In questo caso indica che è la Persona a possedere l'Automobile e non l'Automobile a possedere la Persona!
- In alternativa, si può indicare il **ruolo** di uno dei due estremi dell'associazione



# ASSOCIAZIONI

- Un'associazione rappresenta **una relazione** (fisica o concettuale) **tra classi**

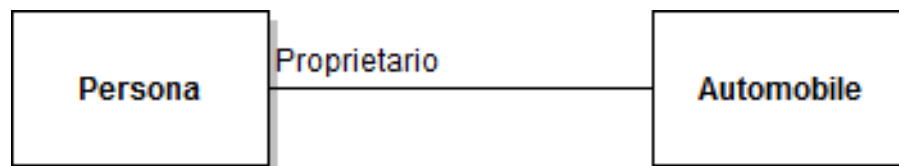


## Warning:

- In un diagramma delle classi non è consigliato aggiungere nomi delle associazioni e ruoli

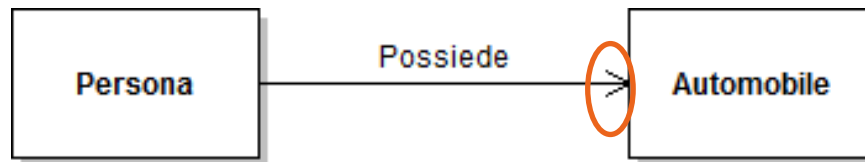
Persona!

- In alternativa, si può indicare il **ruolo** di uno dei due estremi dell'associazione



## VERSO DI NAVIGAZIONE

- Il verso di navigazione indica in quale direzione è possibile reperire le informazioni

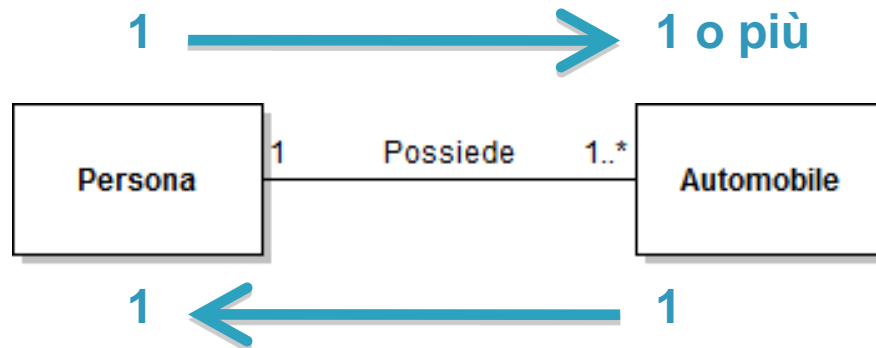


- Esempio: nota una persona è possibile sapere quali sono le automobili che possiede
  - se ne possiede
- Viceversa, non è possibile conoscere il possessore di una data automobile
  - Attenzione: In questo diagramma non ci sono indicazioni sul **quantitativo** di automobili possedute, né sul numero di proprietari di un automobile ...

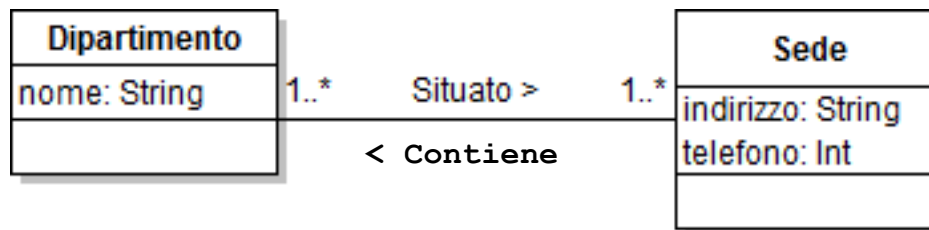
***Mancano le molteplicità ...***

# MOLTEPLICITÀ DELLE ASSOCIAZIONI

- La molteplicità delle associazioni indica il numero di **link** (istanza dell'associazione) tra gli oggetti delle classi

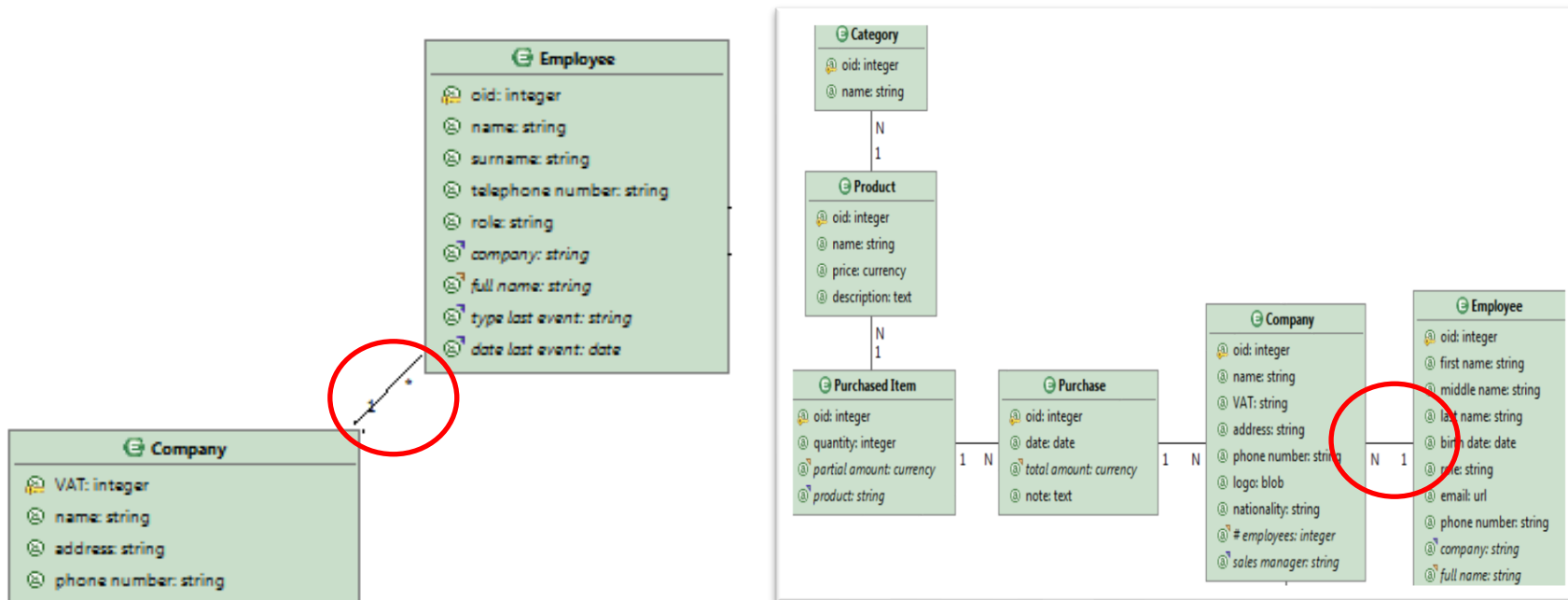


- Una Persona possiede **almeno** un'Automobile
- Un'Automobile è posseduta da **una e una sola** Persona



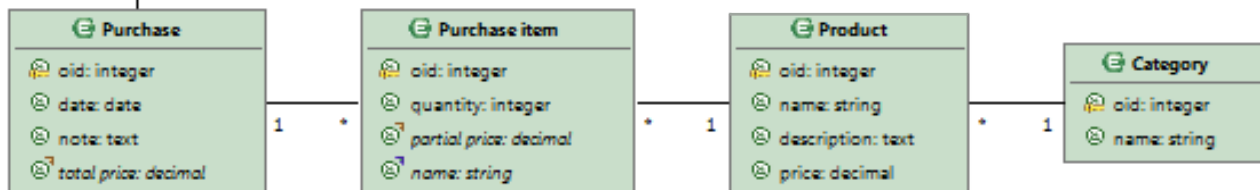
- Un Dipartimento è situato in una o più sedi
- Una Sede contiene uno o più dipartimenti

# MOLTEPLICITÀ NEI DIAGRAMMI E-R



**Warning: fare attenzione perché le molteplicità sono invertite!**

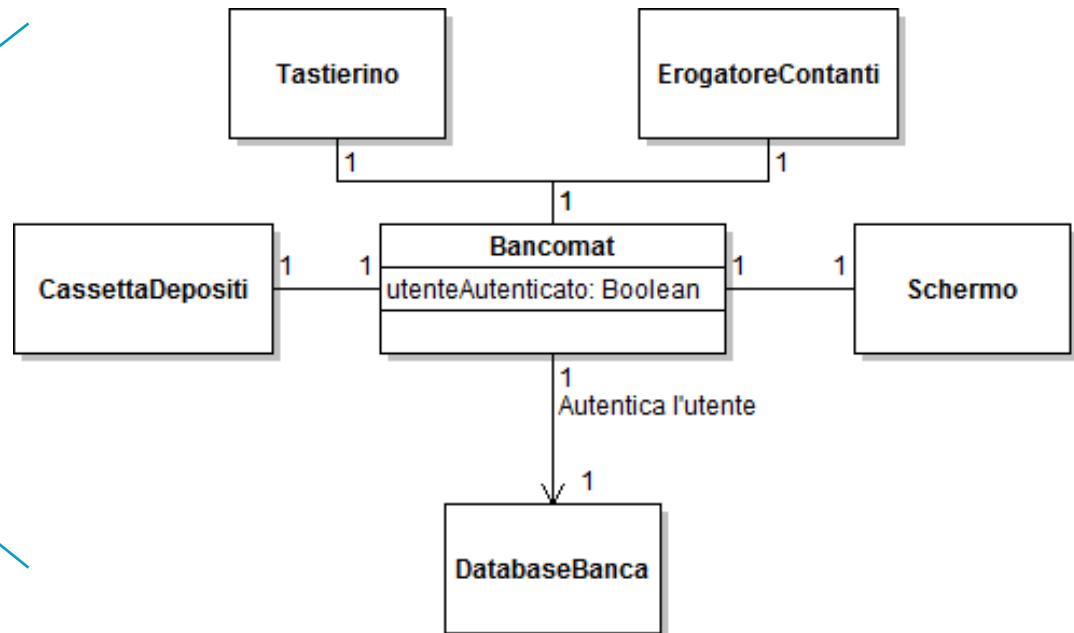
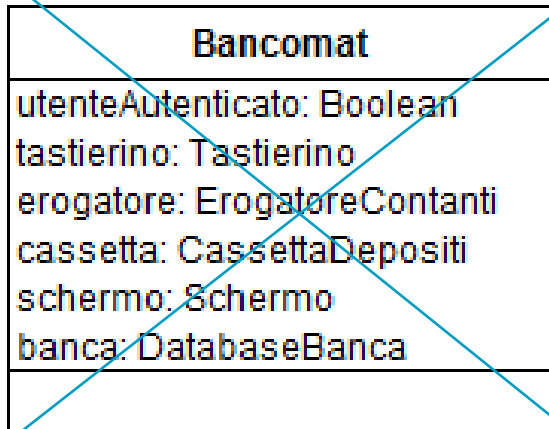
**Differenze rispetto alla notazione E-R**



# ATTRIBUTI E ASSOCIAZIONI

- Due modi diversi di rappresentare 'la stessa cosa' ...
- Di solito si usano: **Convenzione!**
  - Attributi per **tipi primitivi** (es. boolean) e **datatype**
    - es: date, stringhe, booleani
  - Associazioni se tipati da **classi**

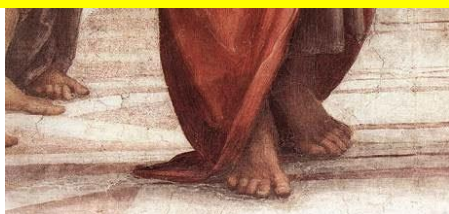
*Scelta migliore!*



*Equivalenti*



**Dalle "forme platoniche" alle "istanze aristoteliche"**



**Platone**



**Aristotele**

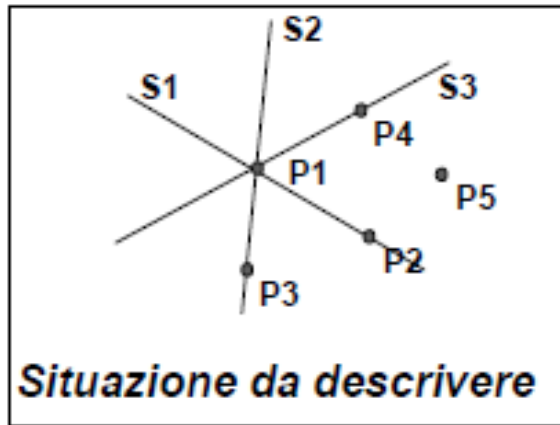
**Scuola di Atene di Raffaello**

*Platone solleva il dito verso l'alto a indicare l'iperuranio e sottintendere la sua filosofia basata sul mondo **delle idee trascendenti** ...*

*Aristotele distende il braccio destro tenendolo sospeso a mezz'aria mostrando che l'idea non ha esistenza propria, ma **s'incarna negli individui che la realizzano** ...*

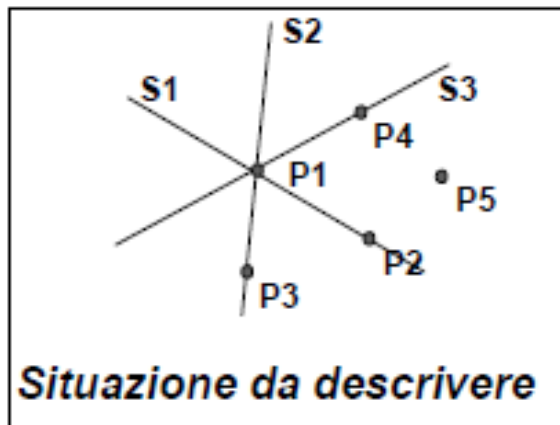


# OBJECT DIAGRAM: ESEMPIO

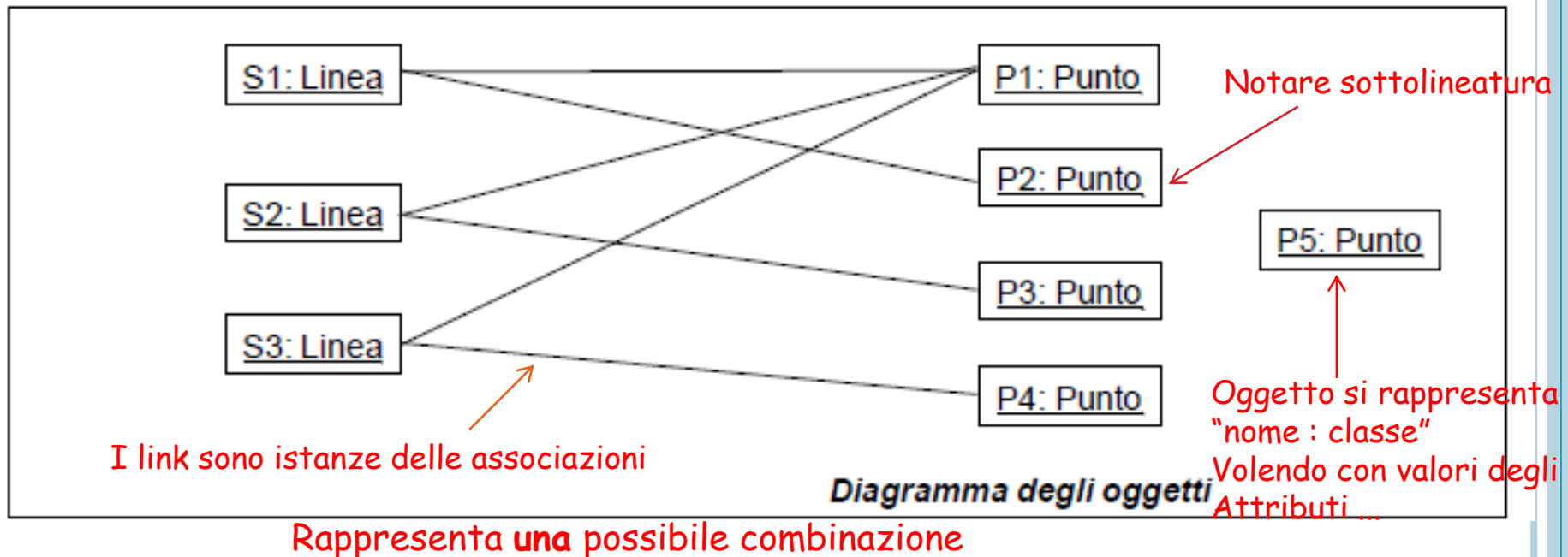
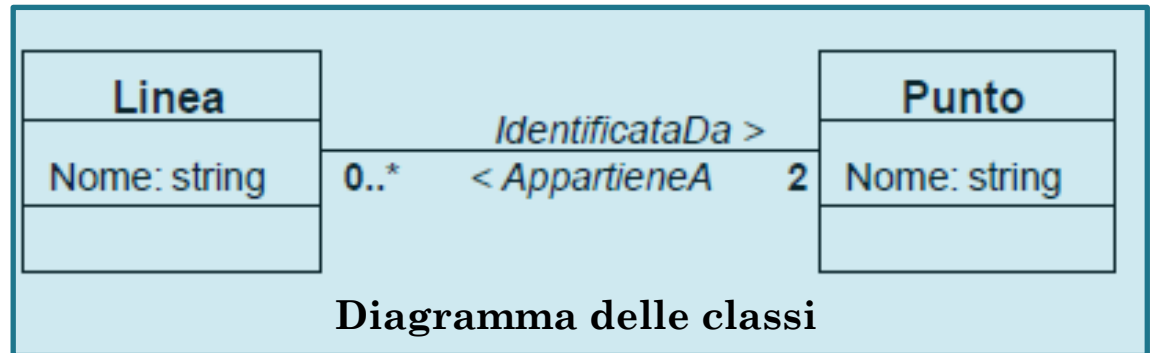


# OBJECT DIAGRAM: ESEMPIO

Gli object diagram servono a chiarire i Class diagram "complessi" ...

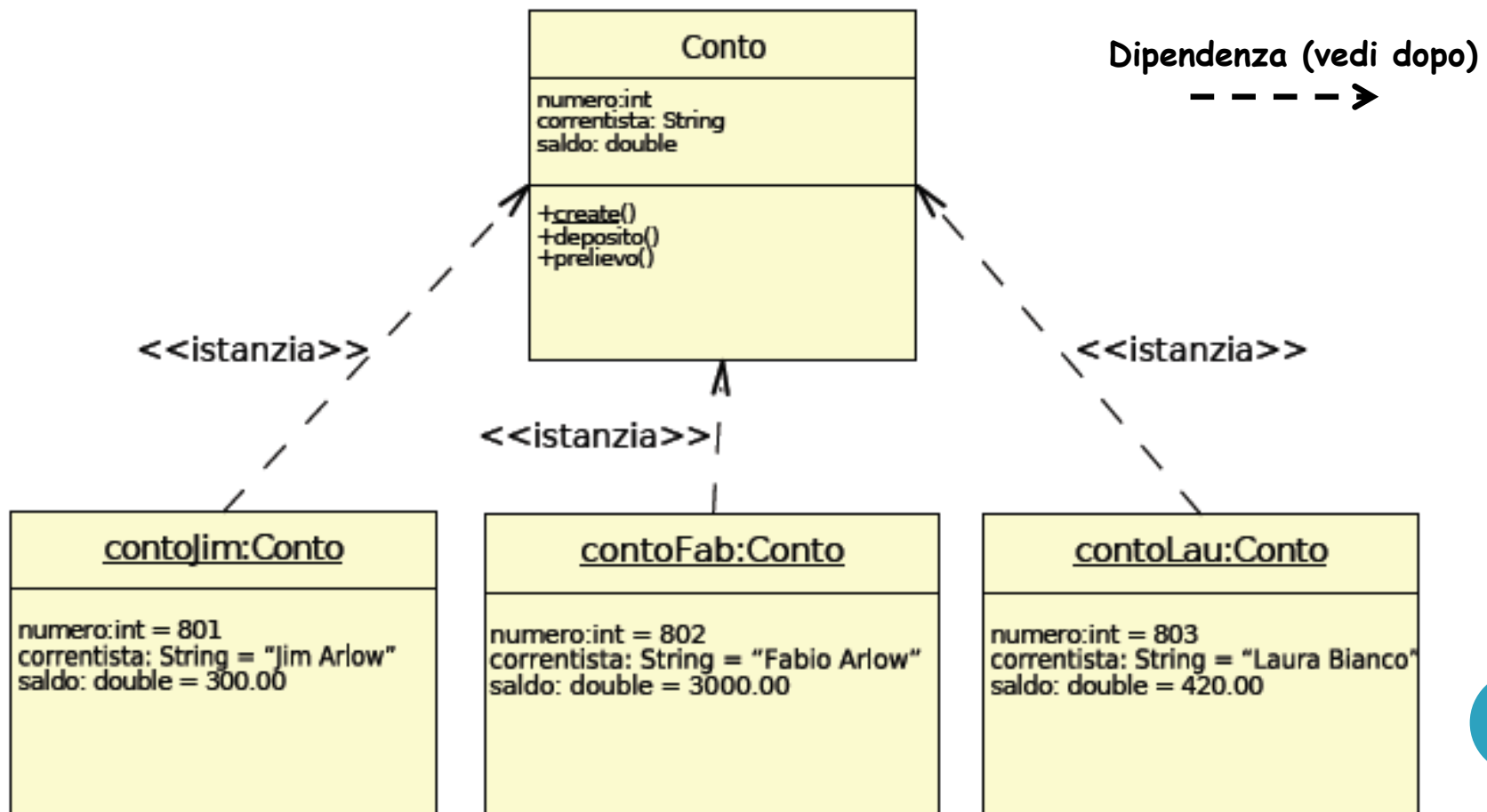


Rappresenta **tutte** le possibili combinazioni valide



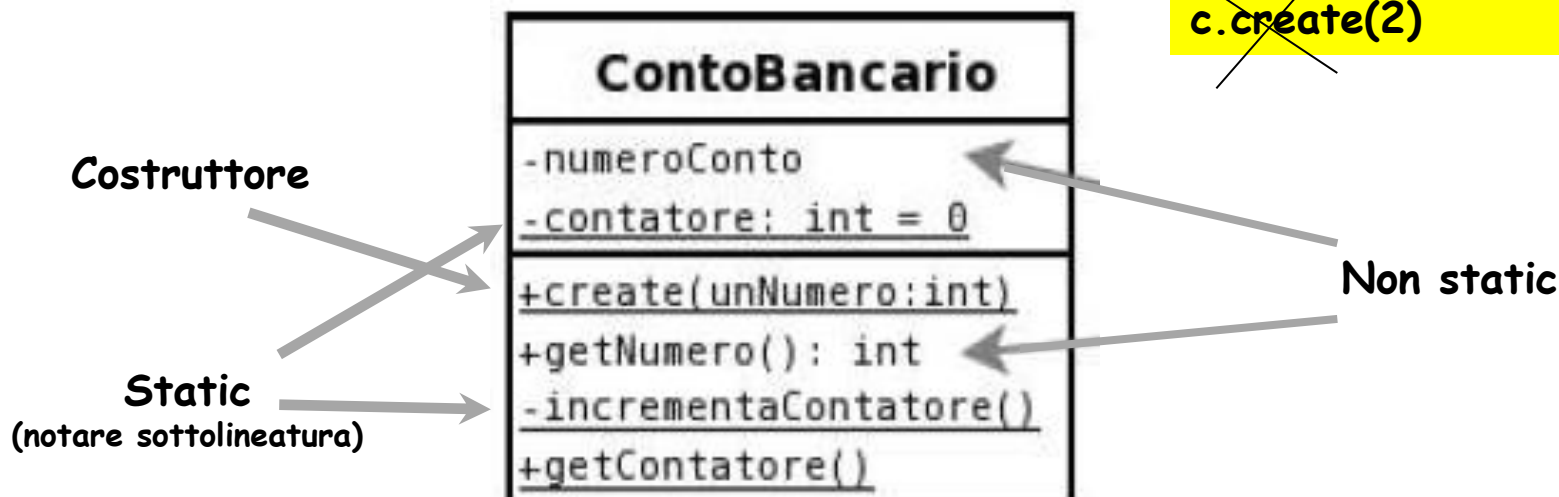
# CLASSI E OGGETTI

- La relazione esistente tra una classe e gli oggetti di quella classe è la relazione **<<istanzia>>**



# OPERAZIONI ED ATTRIBUTI STATICI

- Corrispondono a **metodi e campi static** del linguaggio Java
  - Attributi**: gli oggetti di una stessa classe condividono lo stesso valore per un attributo
  - Operazioni**: le operazioni non operano solo su una particolare istanza della classe, ma sulla classe stessa
    - Ad esempio: costruttori



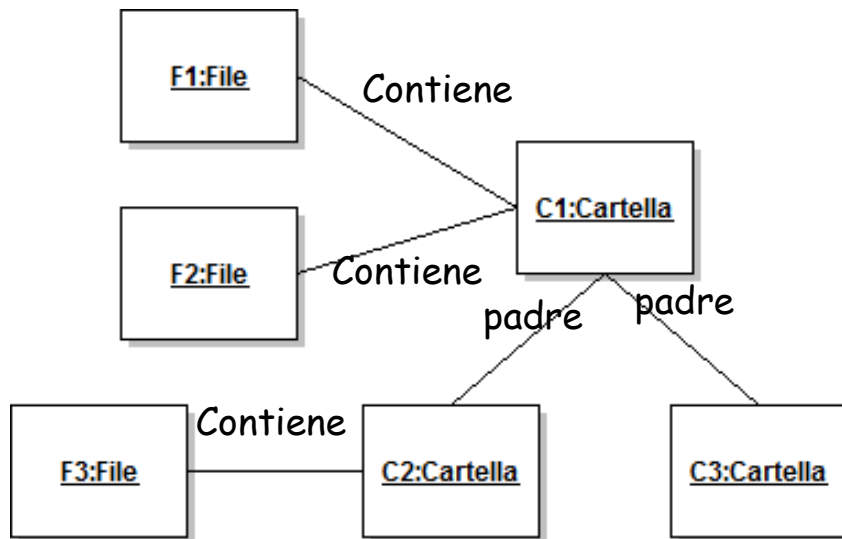
**ContoBancario.create(2)**  
~~v.s.  
c.create(2)~~

# ASSOCIAZIONI RIFLESSIVE

- Si ha un'associazione riflessiva se:
  - Una classe ha una associazione con se stessa ...



Class Diagram



Object Diagram

# IMPLEMENTAZIONE (JAVA)

```
import java.util.HashSet;
```

```
Public class Libro {  
    private int numMatricola;  
    private String titolo;  
    private String autore;  
    private Descrizione descrizione;  
    private HashSet<CopiaLibro> copie;  
    ...  
}
```

Tipare i generici!

Usare un nome plurale per associazioni 1..\* e 0..\*



# IMPLEMENTAZIONE (JAVA)

import

**In generale non esiste una ricetta unica!**

```
Public class Libro {
```

Ma per le associazioni orientate (attenti a quelle bidirezionali):

associazione molteplicità 1 → attributo

associazione molteplicità 0..\* → HashSet

associazione molteplicità {ordered} 0..\* → LinkedList o ArrayList

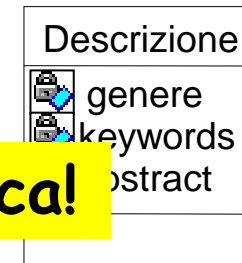
```
private HashSet<CopiaLibro> copie;
```

```
...
```

```
}
```

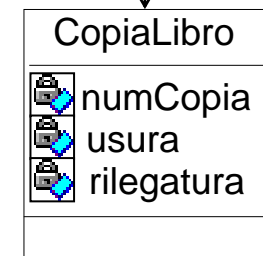
**Tipare i generici!**

**Usare un nome plurale per associazioni 1..\* e 0..\***



**dispone**

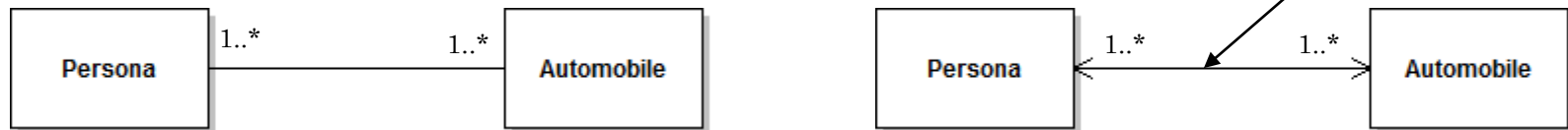
0..\*



# IMPLEMENTAZIONE ASSOCIAZIONI BIDIREZIONALI

E' possibile esplicitare le frecce

- Sono navigabili nelle due direzioni ....



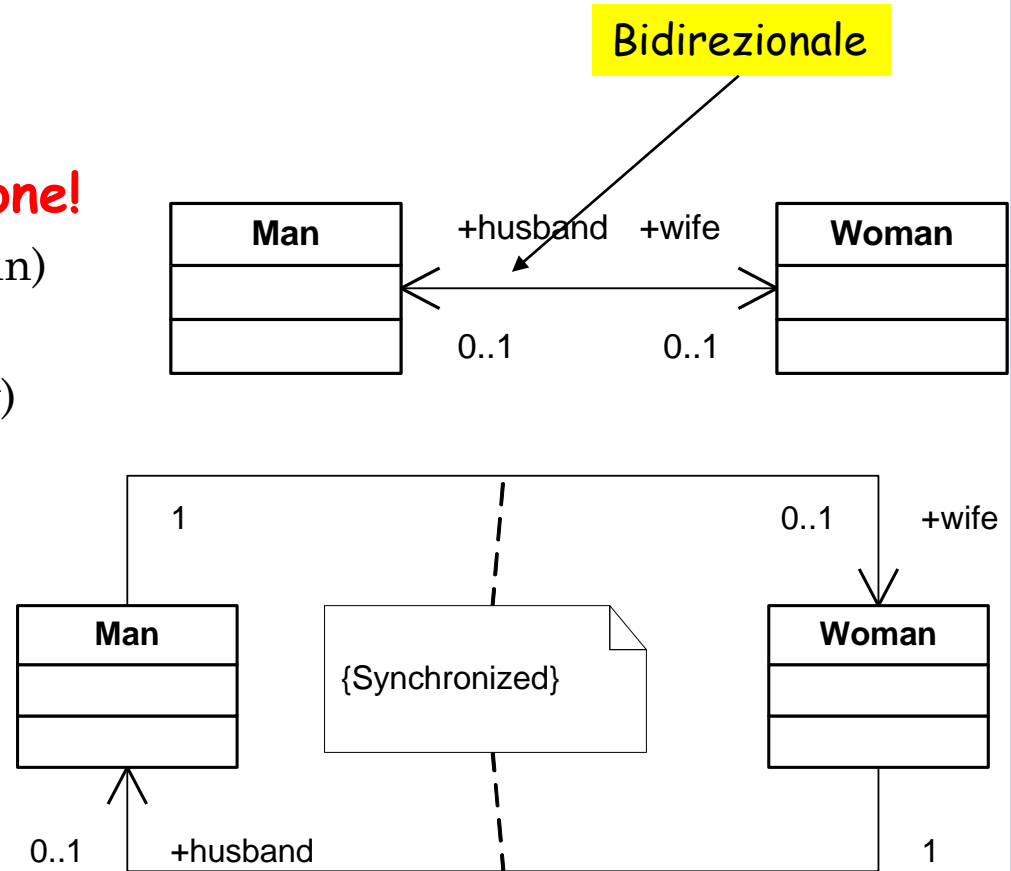
- Cioè: nota una persona è possibile sapere quali sono le automobili che possiede e nota un'automobile è possibile sapere chi sono i possessori
- **Attenzione:** non possono essere implementate con un singolo attributo! (come invece abbiamo fatto prima)
- Complesse da implementare:
  - Vedere:
    - <https://www.edc4it.com/blog/implementing-a-bi-directional-association-in-java>



# ASSOCIAZIONI BIDIREZIONALI: 1 A 1

- Problema della **sincronizzazione!**

- se “setto” il marito a Mary (John)  
devo contemporaneamente  
settare a John la moglie (Mary)

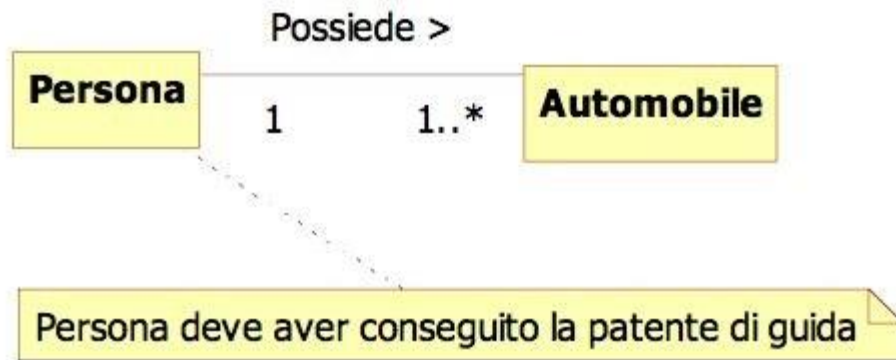


## Warning:

Più il design è “libero” e astratto più si complica la fase di codifica!

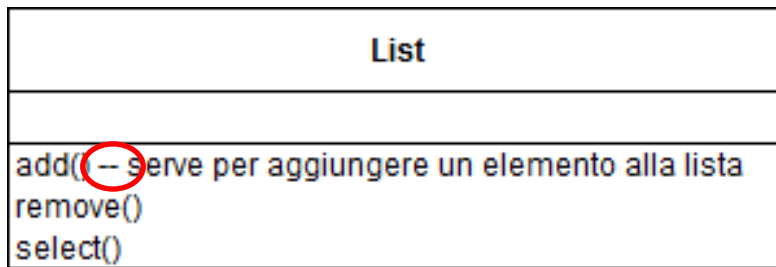
# NOTE

- È come un commento in un linguaggio di programmazione



**A livello di classe**

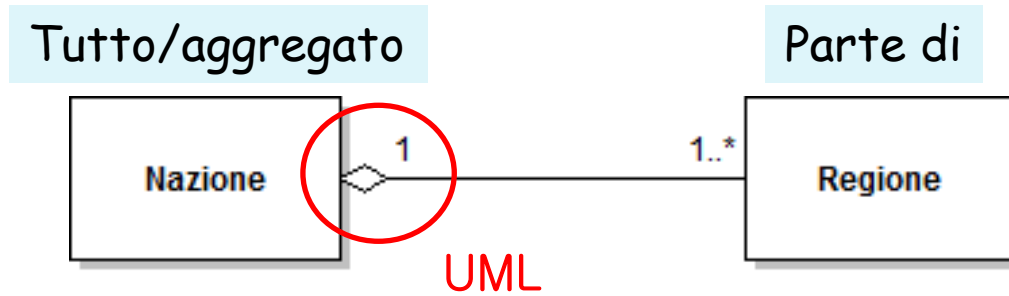
- Si usa anche per specificare il “behaviour” di un operazione



**A livello di feature**

# AGGREGAZIONE

- Rappresenta la relazione “**tutto-parti**”



- Una Regione è **parte di** una Nazione
  - Oppure una Nazione è un aggregato di Regioni
- Differenza tra aggregazione e associazione?
  - Spesso difficile da capire
- Distinzione + importante a livello concettuale
  - A livello software si implementa come un'associazione

**M. Fowler consiglia di usarle solo a livello concettuale**

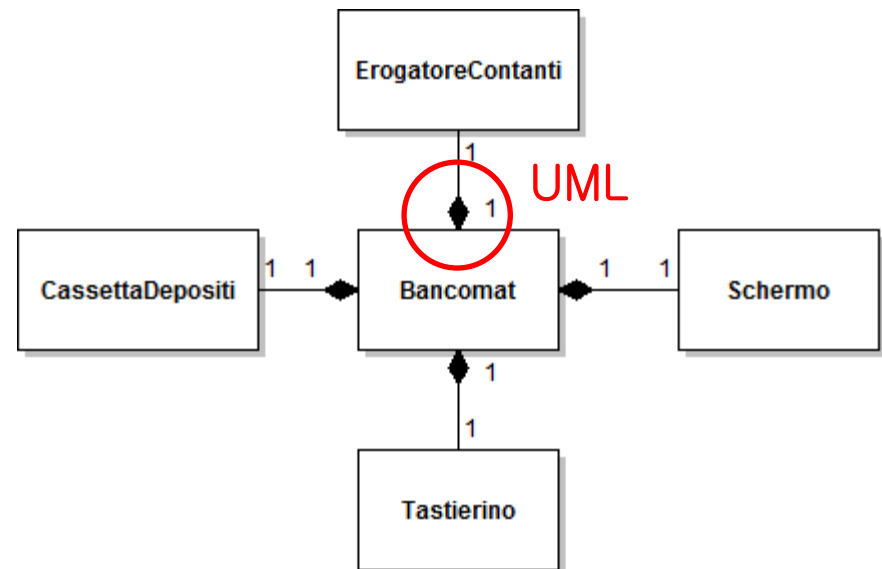
# COMPOSIZIONE

- **Forma forte** di aggregazione

- Esprime la relazione “**ha-un**” / “**è composto di**”

- **Proprietà:**

- Se il composto/intero viene distrutto, anche le sue parti saranno distrutte
  - Le parti non esistono senza il tutto
- Una parte può appartenere ad un solo oggetto intero alla volta
  - Regola di non condivisione



# GENERALIZZAZIONE E EREDITARIETÀ (INHERITANCE)

## ○ Generalizzazione = relazione “è un” + concettuale

- Ogni istanza di una classe è anche istanza della superclasse

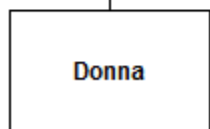
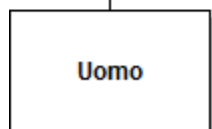
## ○ Ereditarietà + implementativo

- **Meccanismo** attraverso il quale elementi specializzati incorporano la struttura ed il comportamento di elementi più generali

superclasse



sottoclasse

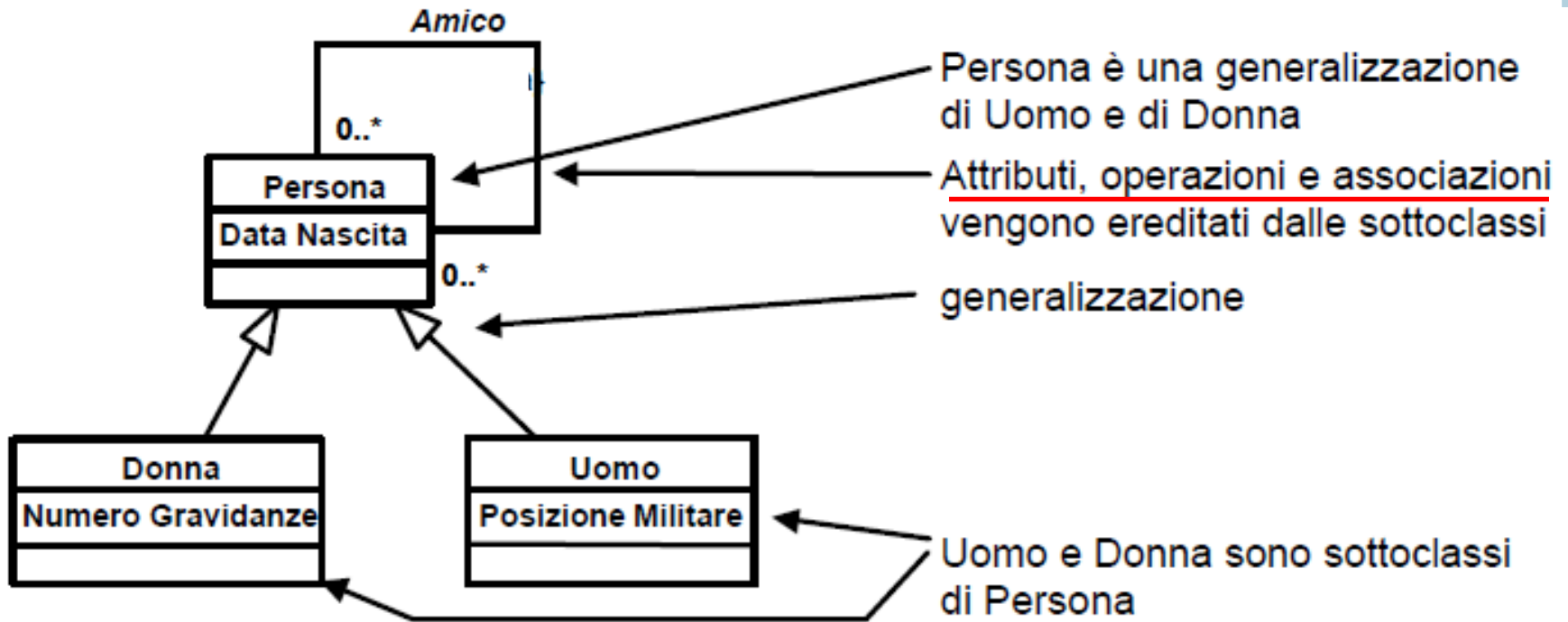


- **Persona** è una generalizzazione di Uomo e di Donna

- Un Uomo è una Persona

Generalizzazione in UML

# GENERALIZZAZIONE: ESEMPIO

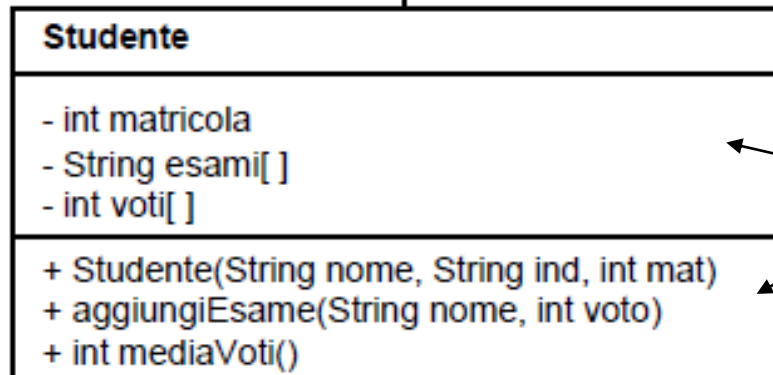
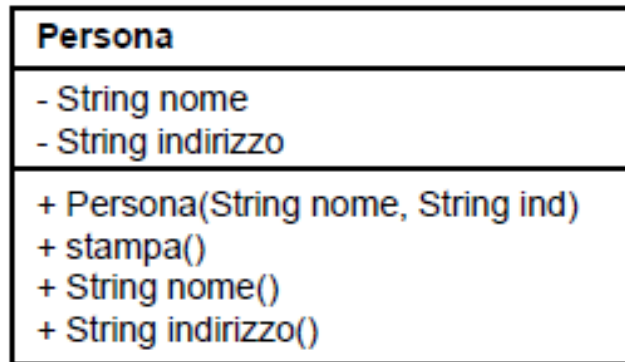


Uomo e Donna **specializzano** Persona

Uomo e Donna ereditano:

- l'associazione "amico" da persona
- la data di nascita

# EREDITARIETÀ: ESEMPIO



- La classe **Studente** eredita da **Persona**:
  - Attributi e operazioni
- Un oggetto **Studente** può essere trattato esattamente come un oggetto **Persona**
- Nella classe erede è possibile:
  - aggiungere nuovi attributi e operazioni
    - Es. matricola / mediaVoti()
  - ridefinire le operazioni
    - **Overriding**
      - Es. stampa()

# OPERAZIONI E METODI UML

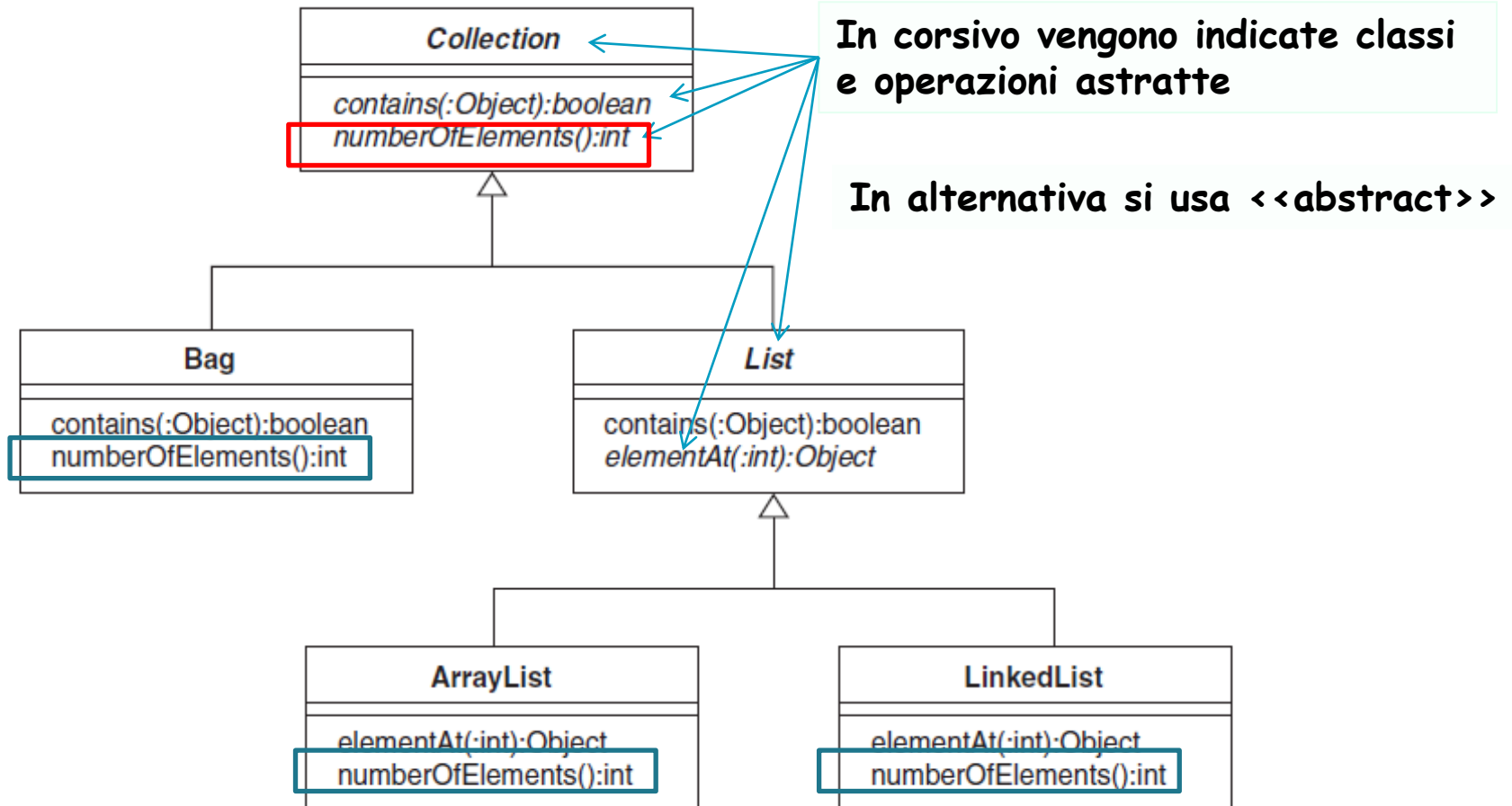
- **Non sono la stessa cosa!**
- Un operazione viene invocata su un oggetto e corrisponde alla **dichiarazione di una procedura/funzione** (è qualcosa di + astratto)
- Un metodo è **il corpo** di tale procedura/funzione
  - **E' l'implementazione di un operazione!**
  - Specifica il behaviour
  - Spesso viene scritto nel diagramma delle classi in una nota a fianco dell'operazione ...

metodo

```
public void productDD_processValueChange(ValueChangeEvent event) {  
    Object productId = productDD.getSelected();  
    try {  
        productDataProvider.setCursorRow(productDataProvider.findFirst("PRODUCT_ID", productId));  
        String prezzo = productDataProvider.getValue("PURCHASE_COST").toString();  
        textField3.setText(prezzo);  
    } catch (Exception e) {  
        error("Impossibile aggiornare il prezzo del prodotto");  
    }  
}
```



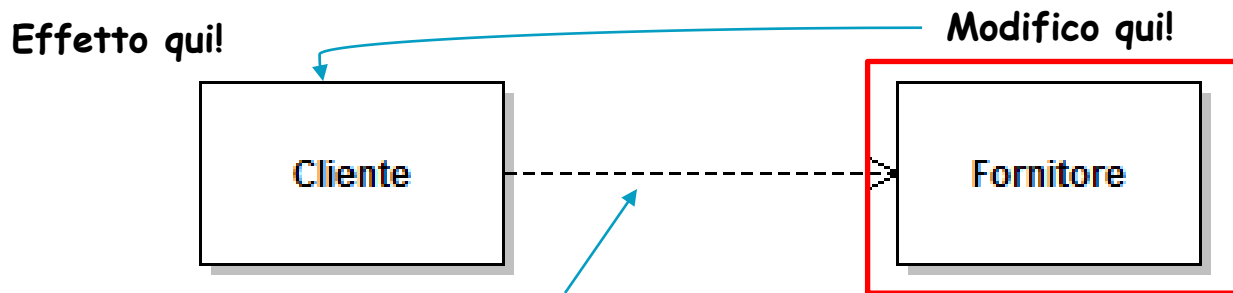
# OPERAZIONI E METODI (COLLECTION JAVA)



- Abbiamo un'operazione **numberOfElements()** e tre metodi diversi che la implementano!

# DIPENDENZE

- Si ha dipendenza tra due classi se la modifica del **Fornitore** può avere un effetto sul **Cliente**



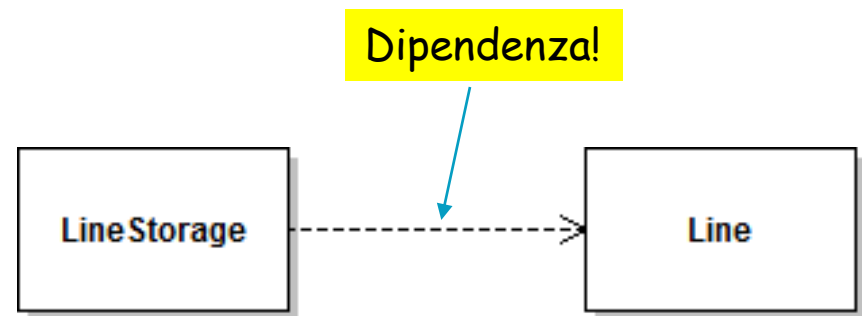
*Notare linea tratteggiata che indica dipendenza in UML*

- Se l'interfaccia di fornitore cambia, qualsiasi messaggio inviato ad esso potrebbe non esser più valido
- La dipendenza può essere causata da molti fattori:
  - Cliente** chiama le operazioni di **Fornitore**
  - Cliente** crea un oggetto di tipi **Fornitore**
  - Cliente** usa **Fornitore** come tipo di una variabile locale
  - Cliente** usa **Fornitore** come tipo di parametro di una sua operazione

# ESEMPIO DI DIPENDENZA

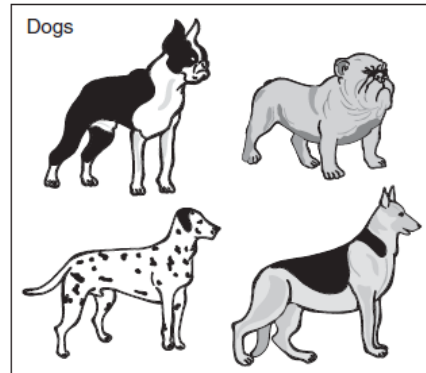
```
public class LineStorage {  
    ....  
    public void addLine(Line line) {  
        lines.add(line);  
    }  
    public int size() {  
        return lines.size();  
    }  
}
```

Crea dipendenza tra LineStorage e Line

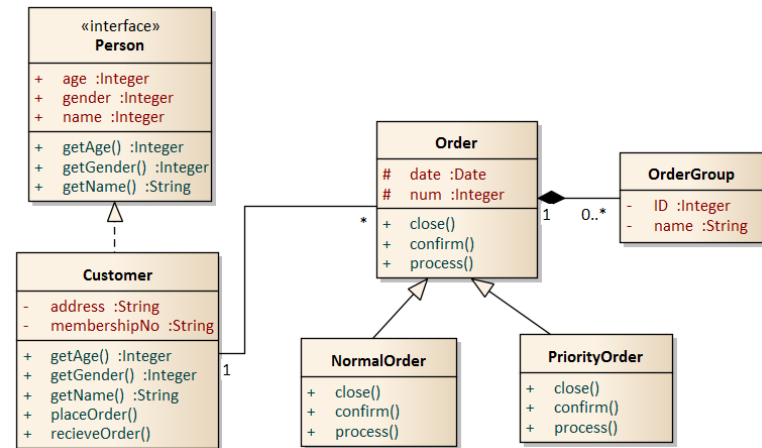


**Class Diagram**

# RIASSUMENDO



*Ripasso concetti chiave OO*



*Ingredienti di un diagramma delle classi UML*



alle "forme platoniche" alle "istanze aristoteliche"

```
import java.util.HashSet;
```

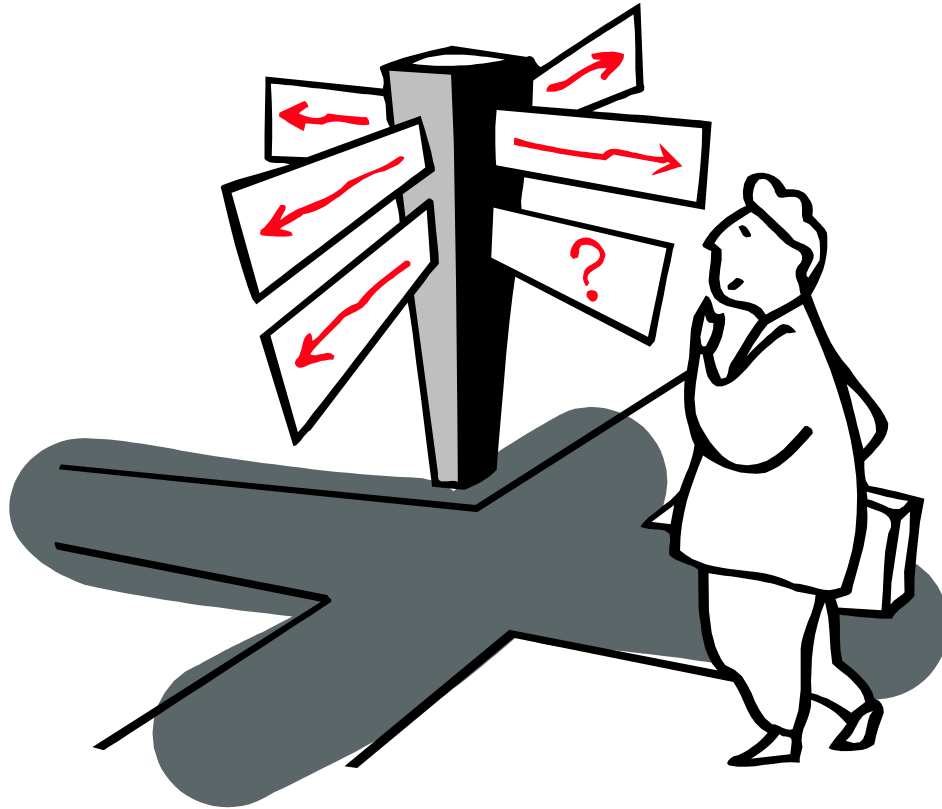
```
Public class Libro {
    private int numMatricola;
    private String titolo;
    private String autore;
    private String editore;
    private Descrizione descrizione;
    private HashSet<CopiaLibro> copie;
    ...
}
```



Tipare i generici!

Usare il plurale per associazioni 1..\* e 0..\*

THE END ...



Domande?