

# Ereditarietà

Dal materiale di  
Stefano Ferretti e Angelo Di Iorio  
s.ferretti@unibo.it  
Angelo.diiorio@unibo.it

# Programmazione ad oggetti

- Abbiamo finora introdotto in modo approssimativo il concetto di classi e oggetti
  - Inevitabile per il linguaggio Java
- Tuttavia, il **paradigma** utilizzato principalmente è stato quello della programmazione **imperativa**
  - O al più **procedurale** attraverso l'utilizzo di **metodi**
  - Paradigma = “stile di programmazione”
- Cominciamo ora a definire meglio il paradigma della **programmazione ad oggetti**
  - *OOP, Object Oriented Programming*

# Programmazione ad oggetti

- *"Object-oriented programming is an exceptionally bad idea which could only have originated in California."*  
(attributed to) Edsger Dijkstra
- *"OOP to me means only messaging, local retention, and protection and hiding of state-process, and extreme late-binding of all things."*  
Alan Kay
- Nonostante le critiche -non infondate- di Dijkstra, la OOP si è rivelata molto importante nello sviluppo software
  - Uno dei suoi concetti fondamentali è l'**ereditarietà**

# Ereditarietà

- Spesso è utile definire una certa classe **in relazione** ad un'altra classe
  - Non una relazione qualsiasi...
- Spesso vale la relazione **IS-A** o di **sotto-tipo** tra classi (*concetti*), es.
  - uno studente *è una* persona
  - un felino *è un* animale
  - un'auto *è un* veicolo
  - ...
  - *Altri esempi?*

# Ereditarietà

- Es. se abbiamo già definito la classe **Persona** è utile/comodo **riusare** il codice già scritto in una nuova classe **Studente**:
  - *Studente ha tutti* gli **attributi** di *Persona*
  - *Studente ha tutti* i **metodi** di *Persona*
    - Eventualmente **specializzati**
  - *Studente può* avere **altri** metodi e attributi
- Come *ogni* persona, uno studente ha nome, cognome, telefono, etc... ma **in più** anche altre caratteristiche come matricola, lista di esami dati, media voti etc...

# Ereditarietà

- L'**ereditarietà** è una **relazione IS-A** tra classi:
  - Una è **superclasse** (*classe base, classe genitore*)
  - L'altra è **sottoclasse** (*classe derivata, classe figlia*)
- Ogni classe può avere a sua volta sottoclassi, in modo da creare una **gerarchia**
  - Lo abbiamo visto per le eccezioni
- In Java, per definire una sottoclasse si usa la keyword **extends**

# Ereditarietà

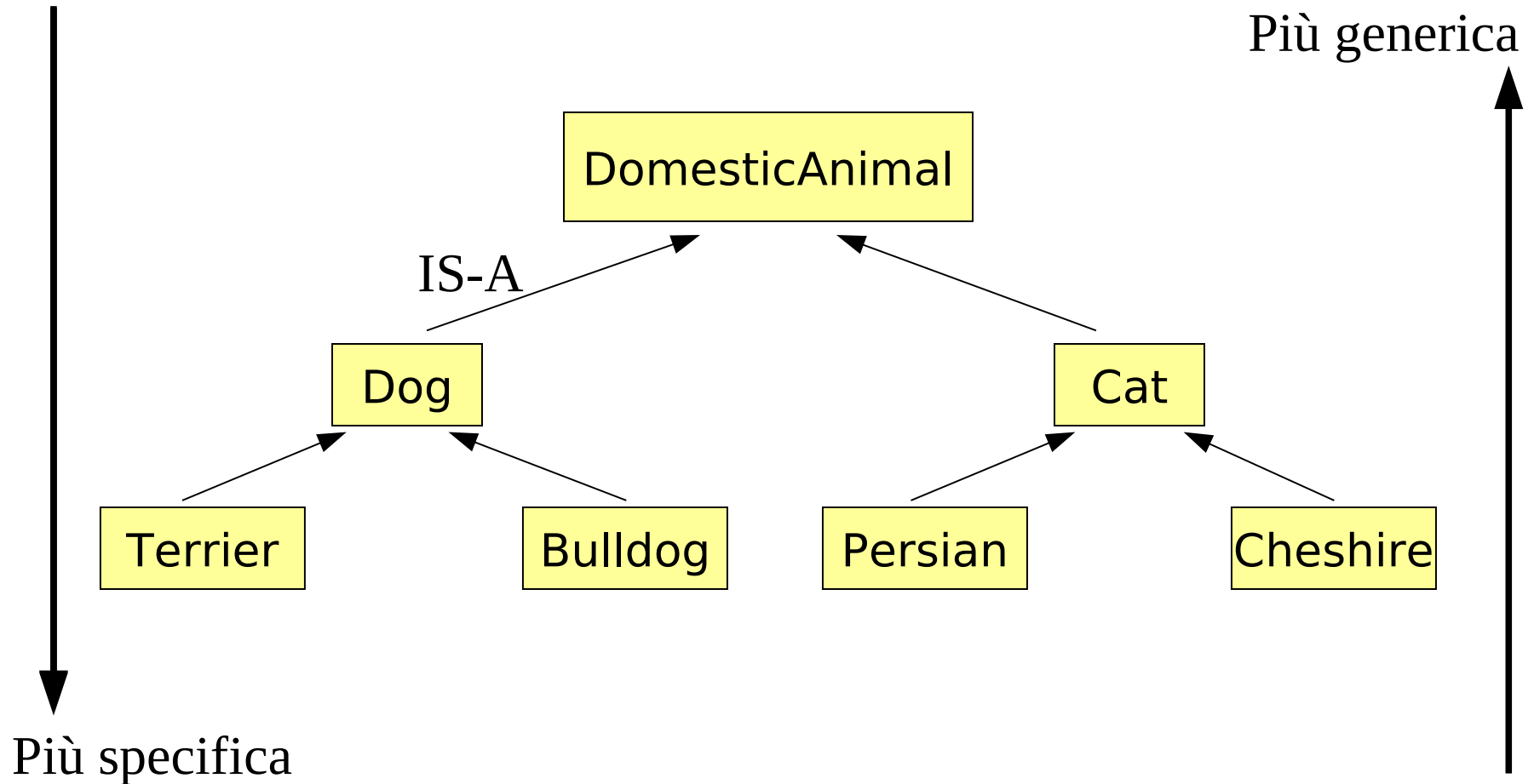
- Ogni sottoclasse può essere definita in modo **incrementale** a partire dalla superclasse
- Questo promuove il **riuso** di codice
- Nella sottoclasse specifichiamo solo il codice che **differisce** dalla classe genitore

# Ereditarietà

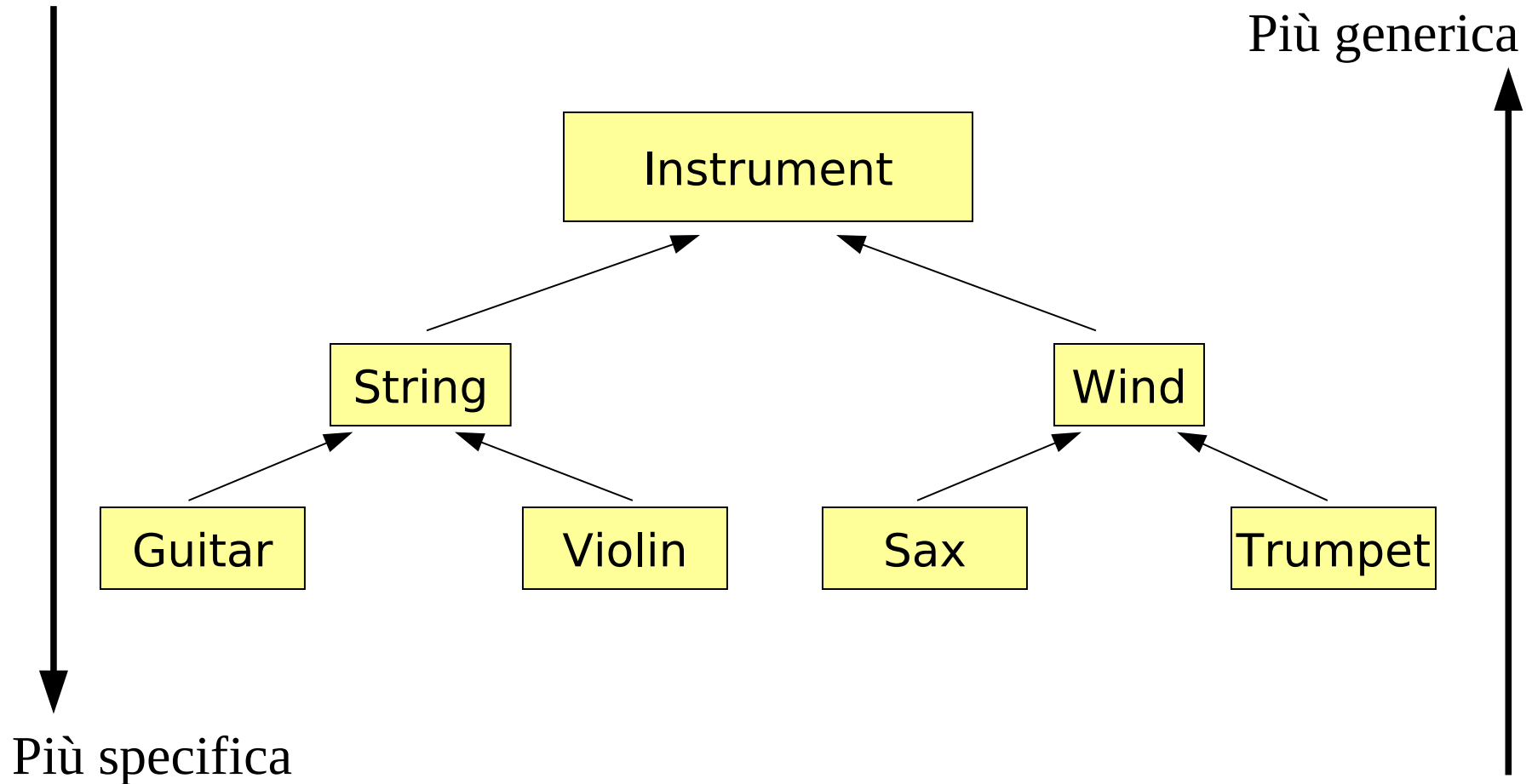
- Il riuso del codice aiuta a scrivere codice di **modulare** e a **risparmiare** tempo di:
  - Scrittura
  - Debugging
  - Modifica
- Si può **specializzare** la superclasse con:
  - Nuovi **attributi** (aggiuntivi)
  - Nuovi **metodi** (aggiuntivi)
  - Nuove **varianti** di metodi della superclasse
    - *Overriding*



# Gerarchia DomesticAnimal



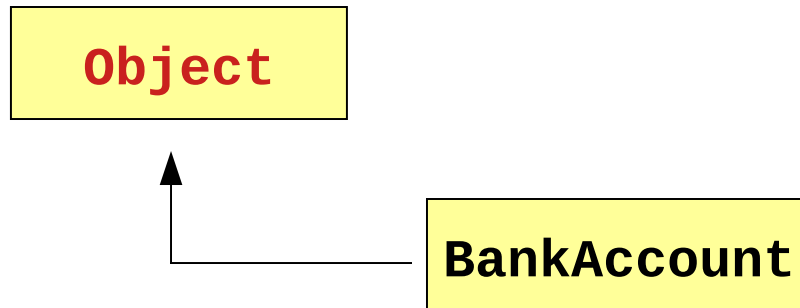
# Gerarchia strumenti musicali



# Richiamo: BankAccount

```
public class BankAccount {  
    private int number;  
    private String owner;  
    private double balance;  
  
    public BankAccount(int accountNumber,  
                        String ownerName,  
                        double initialBalance) {...}  
  
    public void deposit(double amount) {...}  
    public void withdraw(double amount) {...}  
    public int getNumber() {...}  
    public String getName() {...}  
    public String getBalance() {...}  
    public String toString() {...}  
}
```

# Gerarchia BankAccount



**Ogni** classe Java eredita dalla classe **Object**  
La classe **Object** è **antenata** di ogni classe Java

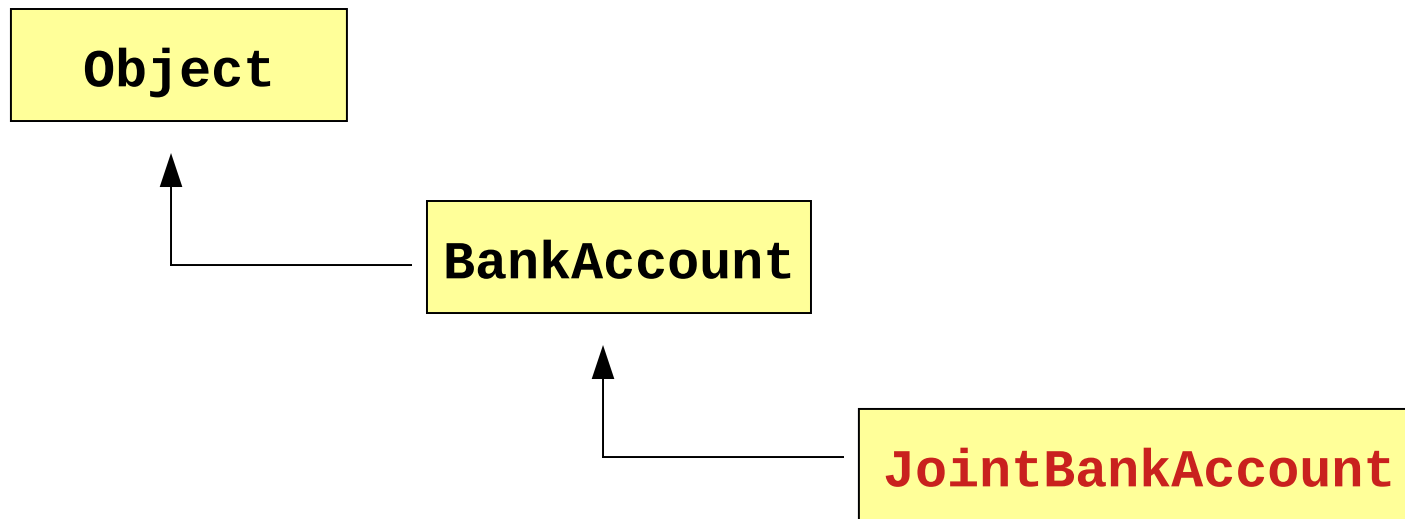
# Template di sottoclasse

```
public class SubclassName extends SuperclassName {  
    dichiarazione nuovi attributi, se esistono  
  
    dichiarazione costruttori (mai ereditati)  
  
    dichiarazione nuovi metodi, se esistono  
  
    dichiarazione metodi overridden (sovrascritti), se esistono  
}
```

# JointBankAccount

```
public class JointBankAccount extends BankAccount {  
    // nuovo attributo per il possessore congiunto  
  
    // nuovi costruttori  
  
    // nuovo metodo getJointName  
    // versione overridden di toString  
  
    // ...  
}
```

# Gerarchia BankAccount



**JointBankAccount** eredita da **BankAccount**  
La classe **Object** è **antenata** di ogni classe Java  
(*quindi anche di JointBankAccount*)

# Regole attributi

- Ogni attributo della *superclasse* diventa **sempre** un attributo della *sottoclasse*
  - *Attributo = campo = var. istanza*
  - *Non sempre accessibili!*
- Gli attributi della superclasse **non** vanno ridichiarati nella sottoclasse
- **Esempio: JointBankAccount** eredita **number, owner, balance** da **BankAccount**



# Visibilità attributi

- Regole di visibilità (**scope**):
  - Un attributo **public** è accessibile da **ogni classe**
  - Un attributo **private** è accessibile **solo** dalla classe che lo contiene (*no sottoclasse*)
  - Un attributo **protected** è accessibile solo da classi dello stesso **package** e da **ogni sottoclasse**
  - **Default:** se non si specifica la visibilità, l'attributo è visibile solo dalle classi nello *stesso package*
- Es.: gli attributi di **BankAccount** non sono visibili da **JointBankAccount**

# Richiamo: BankAccount

```
public class BankAccount {  
    private int number;  
    private String owner;  
    private double balance;  
  
    public BankAccount(int accountNumber,  
                        String ownerName,  
                        double initialBalance) {...}  
  
    public void deposit(double amount) {...}  
    public void withdraw(double amount) {...}  
    public int getNumber() {...}  
    public String getName() {...}  
    public String getBalance() {...}  
    public String toString() {...}  
}
```

# Dichiarazione attributi

- Dichiarare attributi in una sottoclasse non richiede nulla di che: si fa **esattamente** come in ogni classe vista finora
  - ...Anche perchè ogni classe vista finora è sottoclasse di **Object**
- **Ex:** In **JointBankAccount** è necessario un nuovo attributo per il titolare congiunto del conto
  - O magari un *array* se ho >2 titolari

# JointBankAccount

```
public class JointBankAccount extends BankAccount {  
    // nuovo attributo per il possessore congiunto  
  
    // nuovi costruttori  
  
    // nuovo metodo getJointName  
    // versione overridden di toString  
}
```

Nuovo attributo:

```
private String jointName;
```

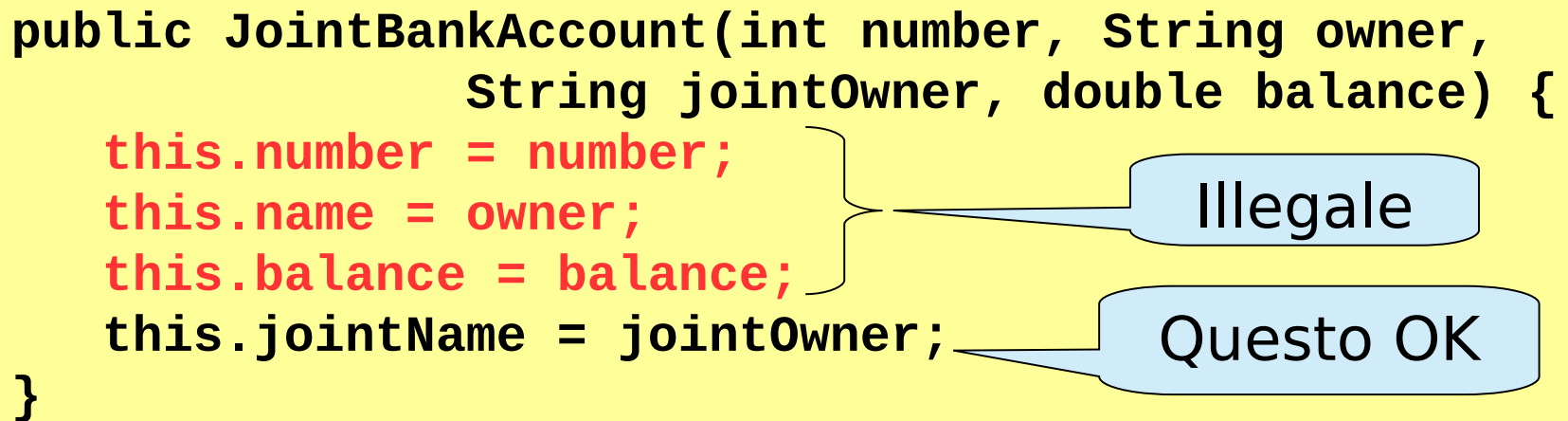
# Regole sottoclassi

- Ogni sottoclasse **deve** dichiarare i propri **costruttori**
- Il costruttore della *sottoclasse* può comunque chiamare il costruttore della superclasse con l'istruzione **super(...parametri...)**
- **super** deve essere la **prima istruzione** del costruttore della sottoclasse
- **Ex:** Se in **JointBankAccount** abbiamo un nuovo attributo, allora serve un nuovo costruttore
  - Per il resto, posso costruire l'oggetto come viene fatto nella superclasse

# JointBankAccount

Costruttore JointBankAccount (tentativo)

```
public JointBankAccount(int number, String owner,  
                        String jointOwner, double balance) {  
    this.number = number;  
    this.name = owner;  
    this.balance = balance;  
    this.jointName = jointOwner;  
}
```



The diagram illustrates a code review of the JointBankAccount constructor. A yellow background contains the code. A blue callout bubble labeled 'Illegale' points to a bracket grouping the first three assignment lines: `this.number = number;`, `this.name = owner;`, and `this.balance = balance;`. Another blue callout bubble labeled 'Questo OK' points to the line `this.jointName = jointOwner;`.

# JointBankAccount

Costruttore JointBankAccount (tentativo)

```
public JointBankAccount(int number, String owner,  
                        String jointOwner, double balance) {  
    this.number = number;  
    this.name = owner;  
    this.balance = balance;  
    this.jointName = jointOwner;  
}
```

Illegale

Questo OK

Gli **attributi** private **non** sono direttamente accessibili nelle classi derivate

I **metodi** privati **non** sono direttamente accessibili nelle classi derivate

# JointBankAccount

Costruttore JointBankAccount (versione corretta)

```
public JointBankAccount(int number, String owner,  
                        String jointOwner, double balance) {  
    super(number, owner, balance);  
    this.jointName = jointOwner;  
}
```

Chiama  
costruttore della  
superclasse



# Regole sottoclassi

- Tutti i metodi **public** e **protected** della *superclasse* sono *automaticamente* ereditati dalla sottoclasse
  - Esattamente come per gli attributi
- **Ex: JointBankAccount** eredita i metodi **getNumber**, **getOwner**, **getBalance**, **withdraw**, **deposit**, **toString**

# Overriding

- I metodi public o protected della superclasse possono essere **ridefiniti** (**overridden**) dalla sottoclasse per **specializzare funzionalità**
- La sottoclasse che ridefinisce un metodo **m** può comunque invocare la versione della superclasse con **super.m(..parametri..)**
- Se non c'è bisogno di ridefinire un metodo in una sottoclasse, semplicemente non lo si dichiara

# BankAccount class

```
public class BankAccount {  
    private int number;  
    private String name;  
    private double balance;  
  
    public BankAccount(int accountNumber,  
        String ownerName, double initialBalance) {...}  
    public void deposit(double amount) {...}  
    public void withdraw(double amount) {...}  
    public int getNumber() {...}  
    public String getName() {...}  
    public String getBalance() {...}  
    public String toString() {...}  
}
```

Vanno bene

Da sovrascrivere in  
JointBankAccount

# Regole sottoclassi

- Oltre a fare overriding, una sottoclasse può **aggiungere** nuovi metodi
- Ex: In **JointBankAccount** possiamo fornire un **nuovo metodo** **getJointOwner** che restituisca il nome del possessore congiunto

# JointBankAccount

```
public String getJointName()  
{  
    return jointName;  
}  
  
public String toString()  
{  
    return "JointBankAccount[" +  
        super.toString() + ", " +  
        jointName + "];  
}  
}
```

nuovo metodo

Metodo  
overridden

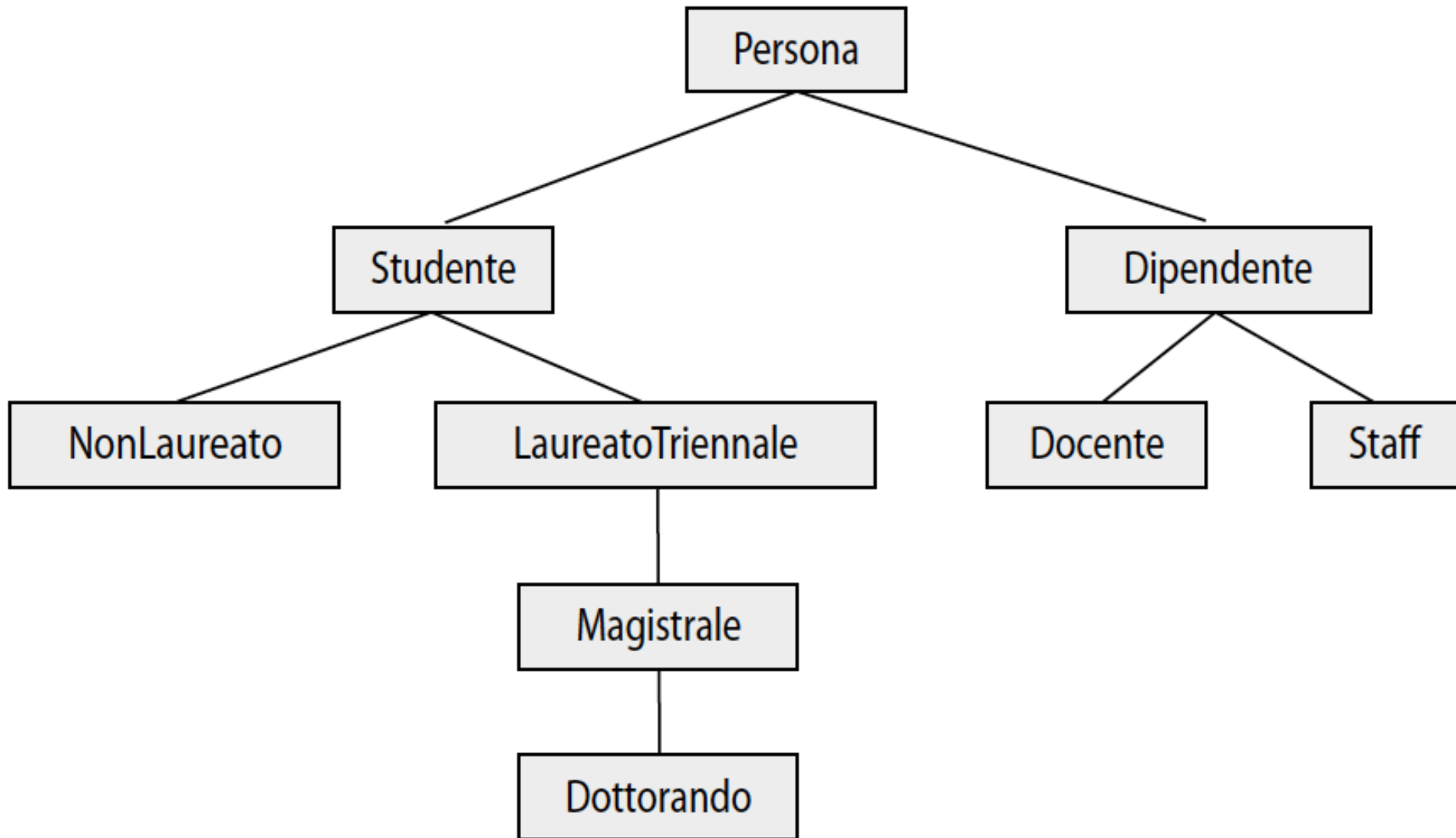
superclass  
toString  
value

```
public class Persona {  
    private String nome;  
  
    public Persona() {  
        nome = "Ancora nessun nome";  
    }  
  
    public Persona(String nomeIniziale) {  
        nome = nomeIniziale;  
    }  
  
    public void setName(String nuovoNome) {  
        nome = nuovoNome;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void scriviOutput() {  
        System.out.println("Nome: " + nome);  
    }  
  
    public boolean haLoStessoNome(Persona altraPersona) {  
        return this.nome.equalsIgnoreCase(altraPersona.nome);  
    }  
}
```

---

# Esempio: classe Persona

# Gerarchia di classi



```
public class Studente extends Persona {  
    private int matricola;
```

```
    public Studente() {  
        super();  
        matricola = 0;    //Ancora nessuna matricola  
    }
```

```
    public Studente(String nomeIniziale, int matricolaIniziale) {  
        super(nomeIniziale);  
        matricola = matricolaIniziale;  
    }
```

```
    public void reimposta(String nuovoNome, int nuovaMatricola) {  
        setName(nuovoNome);  
        matricola = nuovaMatricola;  
    }
```

```
    public int getMatricola() {  
        return matricola;  
    }
```

```
    public void setMatricola(int nuovaMatricola) {  
        matricola = nuovaMatricola;  
    }
```

`super` viene spiegato più avanti. Per il momento non occorre preoccuparsene.

Esempio:  
classe  
Studente



# Classe Studente

```
public void scriviOutput() {  
    System.out.println("Nome: " + getNome());  
    System.out.println("Matricola: " + matricola);  
}  
  
public boolean equals(Studente altroStudente) {  
    return this.haLoStessoNome(altroStudente) &&  
        (this.matricola == altroStudente.matricola);  
}  
}
```

# Esempio

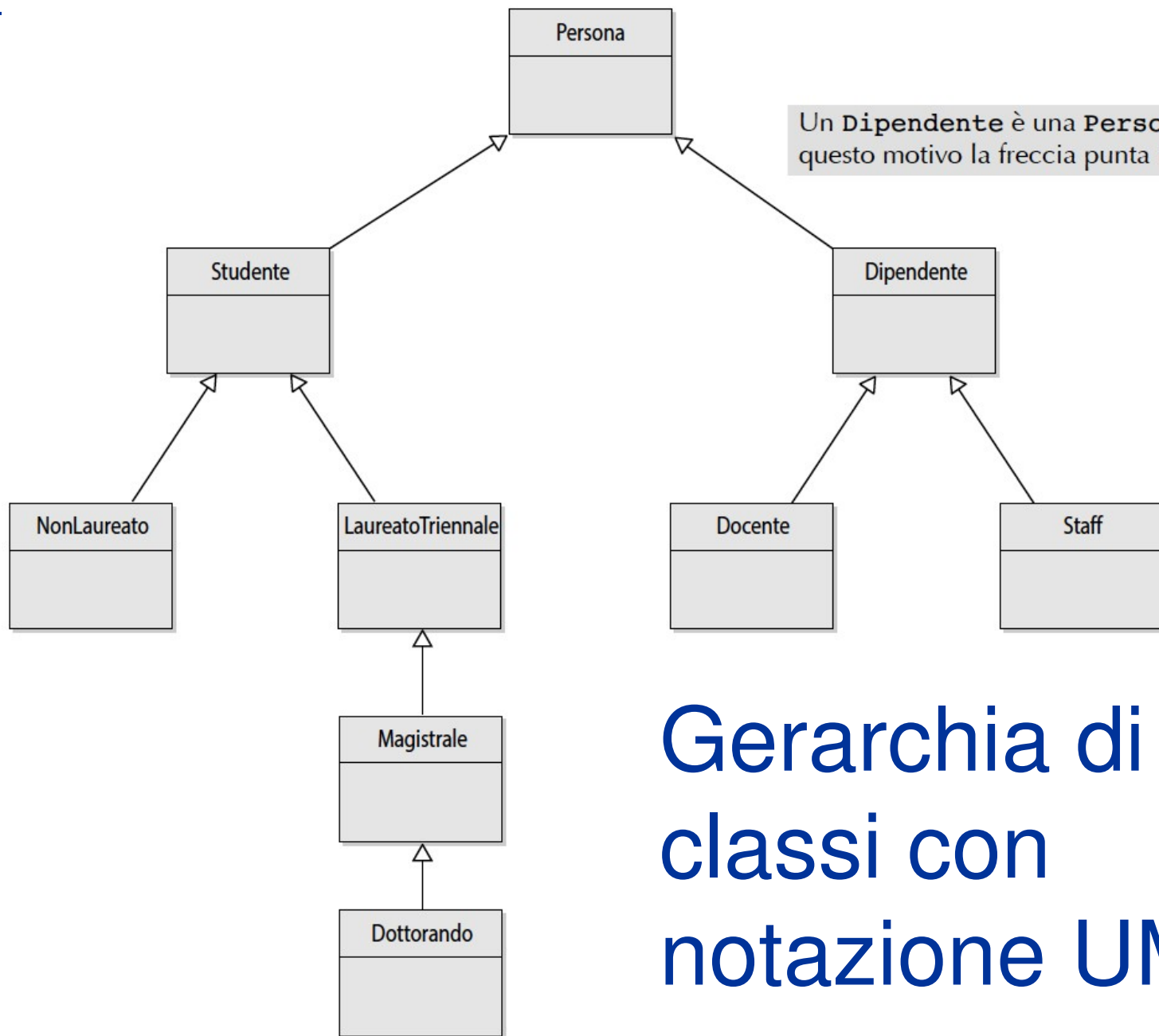
```
public class EreditarietaDemo {  
    public static void main(String[] args) {  
        Studente s = new Studente();  
        s.setNome("Stefano Rampoldi");  
        s.setMatricola(1234);  
        s.scriviOutput();  
    }  
}
```

← setNome è ereditato  
dalla classe **Persona**.

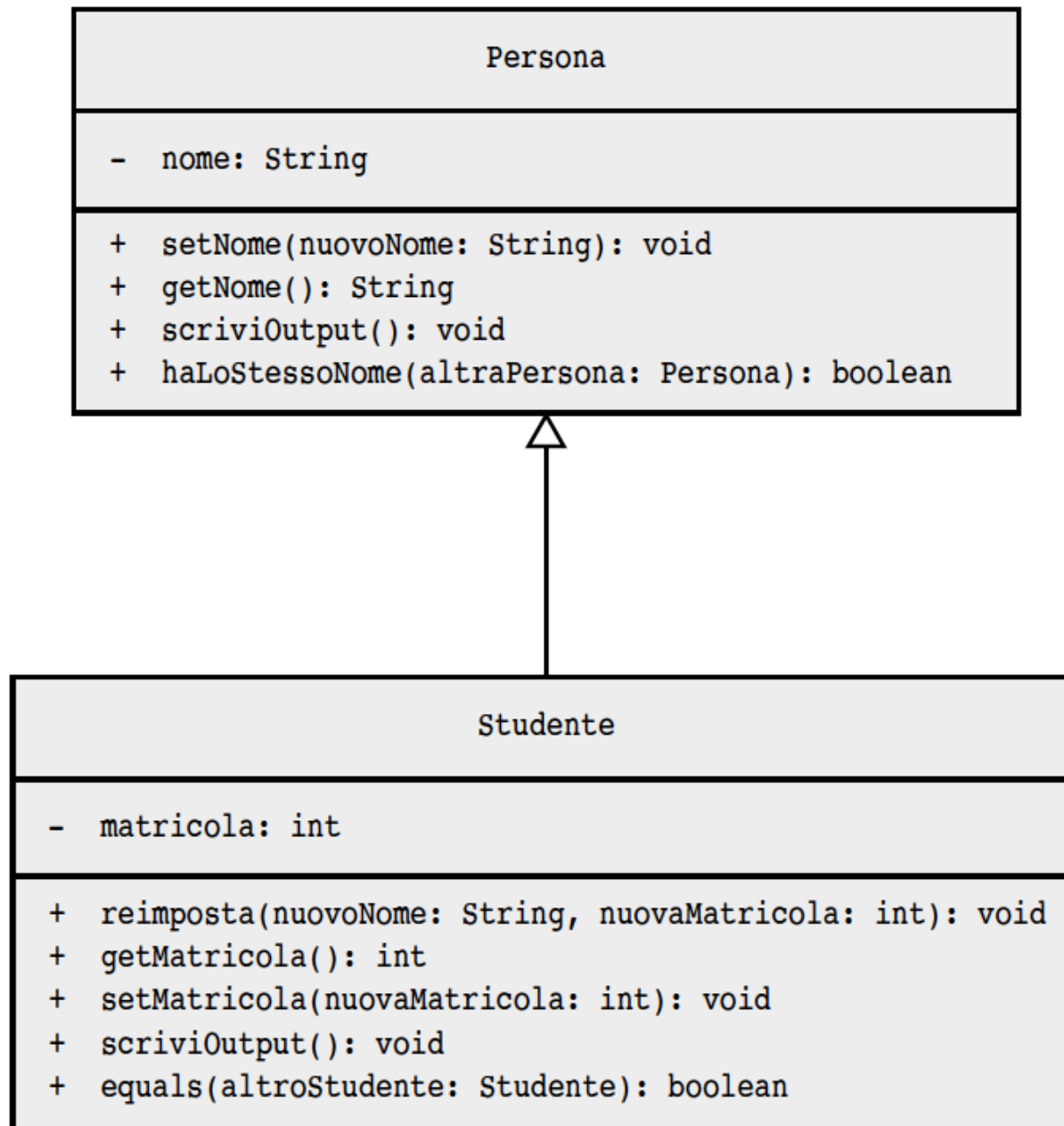
## Esempio di output

Nome: Stefano Rampoldi

Matricola: 1234



## Gerarchia di classi con notazione UML



```
public class NonLaureato extends Studente {
    private int annoDiCorso;    //1 per primo anno, 2 per secondo anno,
                                //3 per terzo anno, o 4 per fuori corso.

    public NonLaureato() {
        super();
        annoDiCorso = 1;
    }

    public NonLaureato(String nomeIniziale,
                        int matricolaIniziale, int annoDiCorsoIniziale) {
        super(nomeIniziale, matricolaIniziale);
        //Verifica 1 <= annoDiCorsoIniziale <= 4
        setAnnoDiCorso(annoDiCorsoIniziale);
    }

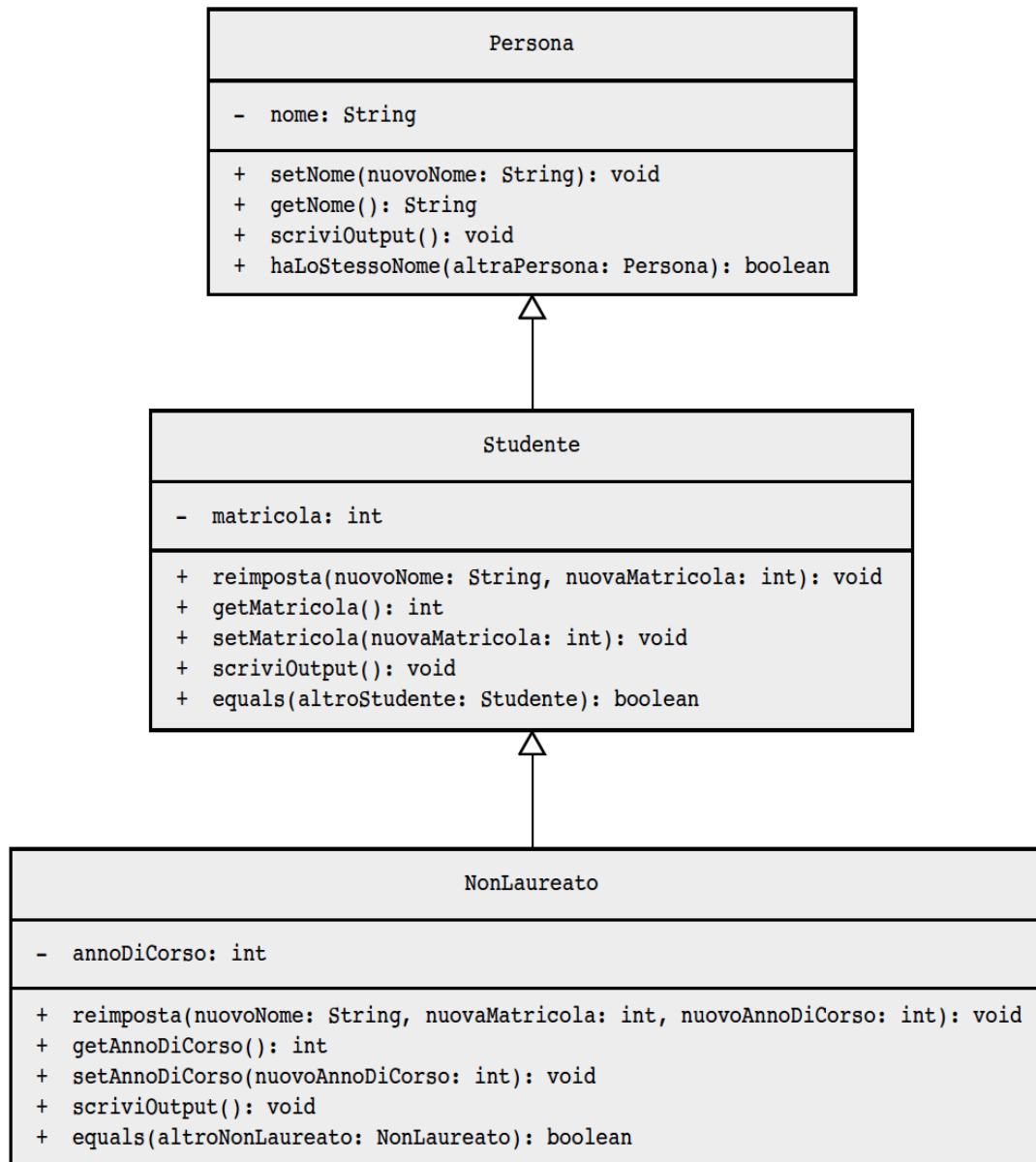
    public void reimposta(String nuovoNome, int nuovaMatricola,
                           int nuovoAnnoDiCorso) {
        reimposta(nuovoNome, nuovaMatricola);    //reimposta di Studente
        //Verifica 1 <= nuovoAnnoDiCorso <= 4
        setAnnoDiCorso(nuovoAnnoDiCorso);
    }

    public int getAnnoDiCorso() {
        return annoDiCorso;
    }
}
```

```
public void setAnnoDiCorso(int nuovoAnnoDiCorso) {
    if ((1 <= nuovoAnnoDiCorso) && (nuovoAnnoDiCorso <= 4))
        annoDiCorso = nuovoAnnoDiCorso;
    else {
        System.out.println("Anno di corso illegale!");
        System.exit(0);
    }
}


public void scriviOutput() {
    super.scriviOutput();
    System.out.println("Anno di corso: " + annoDiCorso);
}

public boolean equals(NonLaureato altroNonLaureato) {
    return equals((Studiante)altroNonLaureato) &&
        (this.annoDiCorso == altroNonLaureato.annoDiCorso);
}
}
```



# Studente.equals generico

```
public boolean equals(Object altroOggetto) {  
    boolean uguale = false;  
    if ((altroOggetto != null) &&  
        (altroOggetto instanceof Studente)) {  
        Studente altroStudente = (Studente)altroOggetto;  
        uguale = this.haLoStessoNome(altroStudente) &&  
            (this.matricola == altroStudente.matricola);  
    }  
    return uguale;  
}
```



NonLaureato **instanceof** Persona ?



# Esempio: Football

- Usiamo l'ereditarietà per descrivere l'entità **calciatore**:
  - *Ogni* calciatore è una persona
  - *Ogni* calciatore ha segnato 0+ goal in carriera
  - *Alcuni* calciatori ricoprono il ruolo di portiere
  - *Ogni* portiere subisce 0+ goal in carriera
- Le statistiche **differenziano** tra calciatori e portieri: per un portiere ci interessa mostrare, oltre ai goal segnati (*rarissimo ma possibile!*) anche quelli subìti

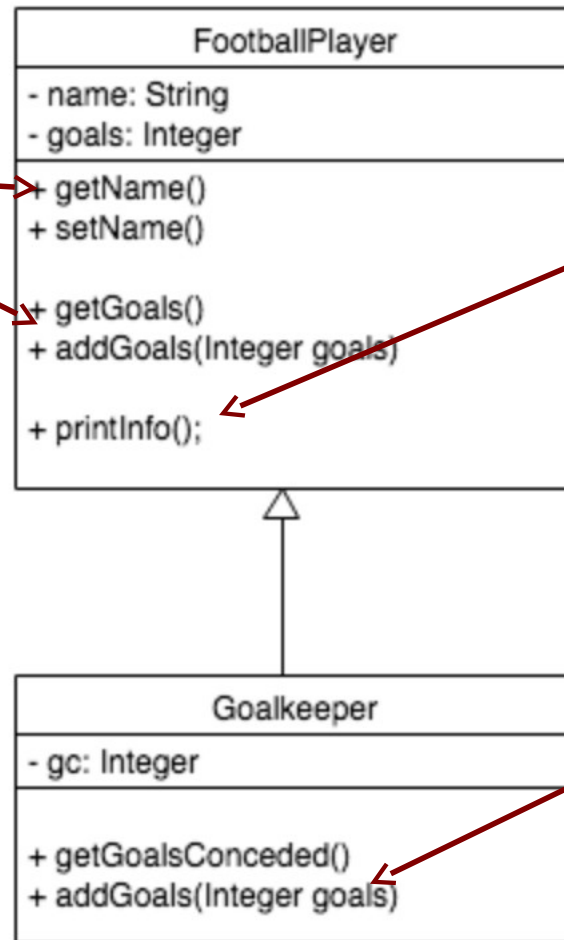
# Esempio: Football



# Esempio: Football

Stesso comportamento  
nella sottoclasse

Comportamento  
**diverso** nella  
sottoclasse



Nuovi metodi

# Classe FootballPlayer

```
public class FootballPlayer extends Person {  
  
    private String name;  
    private Integer goals;  
  
    public FootballPlayer(String name) {  
        this.name = name;  
        this.goals = 0;  
    }  
  
    public String getName() { return name; }  
  
    public void setName(String name) { this.name = name; }  
  
    public Integer getGoals() { return goals; }  
  
    public void addGoals(Integer goals) { this.goals += goals; }  
  
    public void printInfo(){  
        System.out.println(this.name + " scored " + this.goals + " goals");  
    }  
}
```

# Classe Goalkeeper

```
public class Goalkeeper extends FootballPlayer {  
    private Integer gc; // Goals Conceded (Goal subiti)  
  
    public Goalkeeper(String name) {  
        super(name);  
        this.gc = 0;  
    }  
  
    public Integer getGoalsConceded() {  
        return gc;  
    }  
  
    public void addGoalsConceded(Integer gc) {  
        this.gc = this.gc + gc;  
    }  
  
    public void printInfo() {  
        super.printInfo();  
        System.out.print("...and conceded " + this.gc + " goals.");  
    }  
}
```

# Esercizio (1/2)

- Scrivere una classe **ContoCorrente** avente le seguenti operazioni:
  - Apertura conto vuoto
  - Apertura conto con x euro
  - Deposito x euro sul conto
  - Prelievo x euro sul conto
  - Saldo del conto (ritornato da `toString()`)
- Definire una **sottoclasse ContoInternazionale** per gestire valute diverse aggiungendo un **campo immutabile valuta**
  - Può essere **solo** tra: EUR, USD, GBP, CAD, AUD, NZD, CHF, JPY
- Definire **2 nuovi costruttori** che tengano conto della valuta
- **Sovrascrivere** il metodo **`toString()`** per stampare il saldo del conto nella valuta corrente

# Esercizio (2/2)

- Definire una classe **ContoCompleto** che estenda **ContoInternazionale** tracciando i **singoli movimenti**
  - Prelievi/Depositi
- Definire nuovi metodi per il prelievo/deposito che oltre alla quantità prendano in input la **causale** (una stringa)
  - Sovrascrivere quelli esistenti: se la causale non è specificata si assume la stringa “**N/A**”
- Definire un metodo **stampaMovimenti(int n)** che stampi in *ordine cronologico inverso* gli **ultimi n** movimenti uno sotto l'altro
  - Se **n = 0** li stampa tutti