

EREDITARIETA' E DYNAMIC DISPATCH

Ereditarietà per estensione

In Java è possibile definire una classe come estensione di un'altra classe esistente usando la parola chiave **extends**

▶ `class B extends A { ... }`

In questo caso, A è **super-classe**, mentre B è **sotto-classe**

Osservazione:

Java si basa su **ereditarietà singola** (o **semplice**)

- ▶ Una classe può implementare più interfacce, ma può estendere **una sola super-classe**
- ▶ Altri linguaggi (es. C++) prevedono ereditarietà multipla

La classe Object

Una classe che non estende altre classi, implicitamente estende la classe di default **Object**

▶ `class A {...} // implicitamente extends Object`

La classe Object mette a disposizione alcuni **metodi assunti essere presenti in tutte le classi**

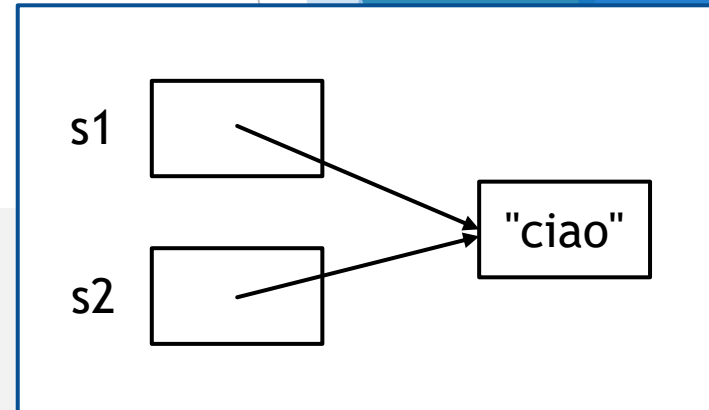
- ▶ `toString()` - restituisce una rappresentazione testuale dell'oggetto
- ▶ `equals(obj)` - confronta l'oggetto corrente con l'oggetto `obj`
- ▶ `clone()` - crea una copia dell'oggetto
- ▶ ... vedere <https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>

Nota importante: confronto con equals()

Tra i metodi di Object, è importante sottolineare `equals`

- ▶ Consente di **confrontare** due oggetti dal punto di vista dei **contenuti**!
- ▶ Ogni classe può ridefinire il metodo `equals` in modo da realizzare un confronto sensato per il tipo di dato rappresentato dalla classe
- ▶ Confrontare gli oggetti con `==` corrisponde invece a valutare se i riferimenti a quegli oggetti sono uguali (cioè, **alias**)

```
String s1 = "ciao";  
String s2 = s1;  
String s3 = "ciao a tutti"  
String s4 = s3.substring(0,4); // restituisce sottostringa "ciao"  
System.out.println(s1==s2);      // true (stesso riferimento)  
System.out.println(s2==s4);      // false (riferimenti diversi)  
System.out.println(s2.equals(s4)); // true (stesso contenuto)
```



Nota importante: confronto con equals()

Attenzione String è speciale...

- I valori letterali ("abcd") sono trattati nel linguaggio come degli oggetti, ma IMMUTABILI
- I metodi di String non modificano gli oggetti, ma ne restituiscono sempre dei nuovi come risultato
- Il compilatore riconosce valori letterali uguali nel codice e ottimizza il codice costruendo l'oggetto corrispondente una volta sola:

```
String s1 = "ciao";
```

```
String s2 = "ciao";
```

```
System.out.println(s1 == s2); // true (per l'ottimizzazione)
```

```
System.out.println(s2==s4); // false (riferimenti diversi)
```

```
System.out.println(s2.equals(s4)); // true (stesso contenuto)
```

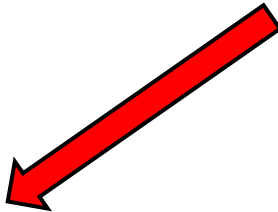
Modificatori di visibilità e sotto-classi

Le **regole di visibilità** legate ai modificatori public e private valgono anche tra **super-classe** e **sotto-classe**

```
class A {  
    private int a1;  
    private int a2;  
    ...  
}
```

```
class B extends A  
    private int a3;  
    public int somma() {  
        return a1 + a2 + a3;  
    }  
}
```

B non può accedere ad a1 e a2
perché sono membri privati di A



Modificatori di visibilità e sotto-classi

Le **regole di visibilità** legate ai modificatori public e private valgono anche tra **super-classe** e **sotto-classe**

```
class A {  
    protected int a1;  
    protected int a2;  
    ...  
}
```



Il modificatore **protected** rende i membri **visibili alle sotto-classi** senza renderli completamente pubblici

```
class B extends A  
    private int a3;  
    public int somma() {  
        return a1 + a2 + a3;  
    }  
}
```

Altrimenti B dovrebbe accedere ad `a1` e `a2` tramite opportuni **metodi pubblici** di A

- è un'altra soluzione che può aver senso...

Classi astratte

In una gerarchia di classi, può aver senso avere delle **classi parzialmente definite** (**classi astratte**)

Una **classe astratta** è una classe che **contiene almeno un metodo astratto**, ossia presente nella classe ma **non implementato** (come nelle interfacce)

- ▶ Una classe astratta **non può essere istanziata**
- ▶ Una classe astratta **può essere estesa** da una classe che si occupi di implementare i metodi astratti

Ha lo **scopo** di **definire un tipo di oggetto** comune a più classi (come le interfacce), fornendo anche **l'implementazione di alcuni metodi comuni**

Classi astratte

```
public abstract class Solido {  
    // variabile d'istanza  
    private double pesoSpecifico ;  
  
    // costruttore  
    public Solido ( double ps ){  
        pesoSpecifico = ps;  
    }  
  
    // metodo implementato  
    public double peso (){  
        return volume () * pesoSpecifico ;  
    }  
  
    // metodi astratti  
    public abstract double volume ();  
    public abstract double superficie ();  
}
```

Classi astratte

```
public abstract class Solido {  
    // variabile d'istanza  
    private double pesoSpecifico ;  
  
    // costruttore  
    public Solido ( double ps ){  
        pesoSpecifico = ps;  
    }  
  
    // metodo implementato  
    public double peso (){  
        return volume () * pesoSpecifico ;  
    }  
  
    // metodi astratti  
    public abstract double volume ();  
    public abstract double superficie ();  
}
```

Classi astratte

```
public abstract class Solido {  
    // variabile d'istanza  
    private double pesoSpecifico ;  
  
    // costruttore  
    public Solido ( double ps ){  
        pesoSpecifico = ps;  
    }  
  
    // metodo implementato  
    public double peso (){  
        return volume () * pesoSpecifico ;  
    }  
  
    // metodi astratti  
    public abstract double volume ();  
    public abstract double superficie ();  
}
```

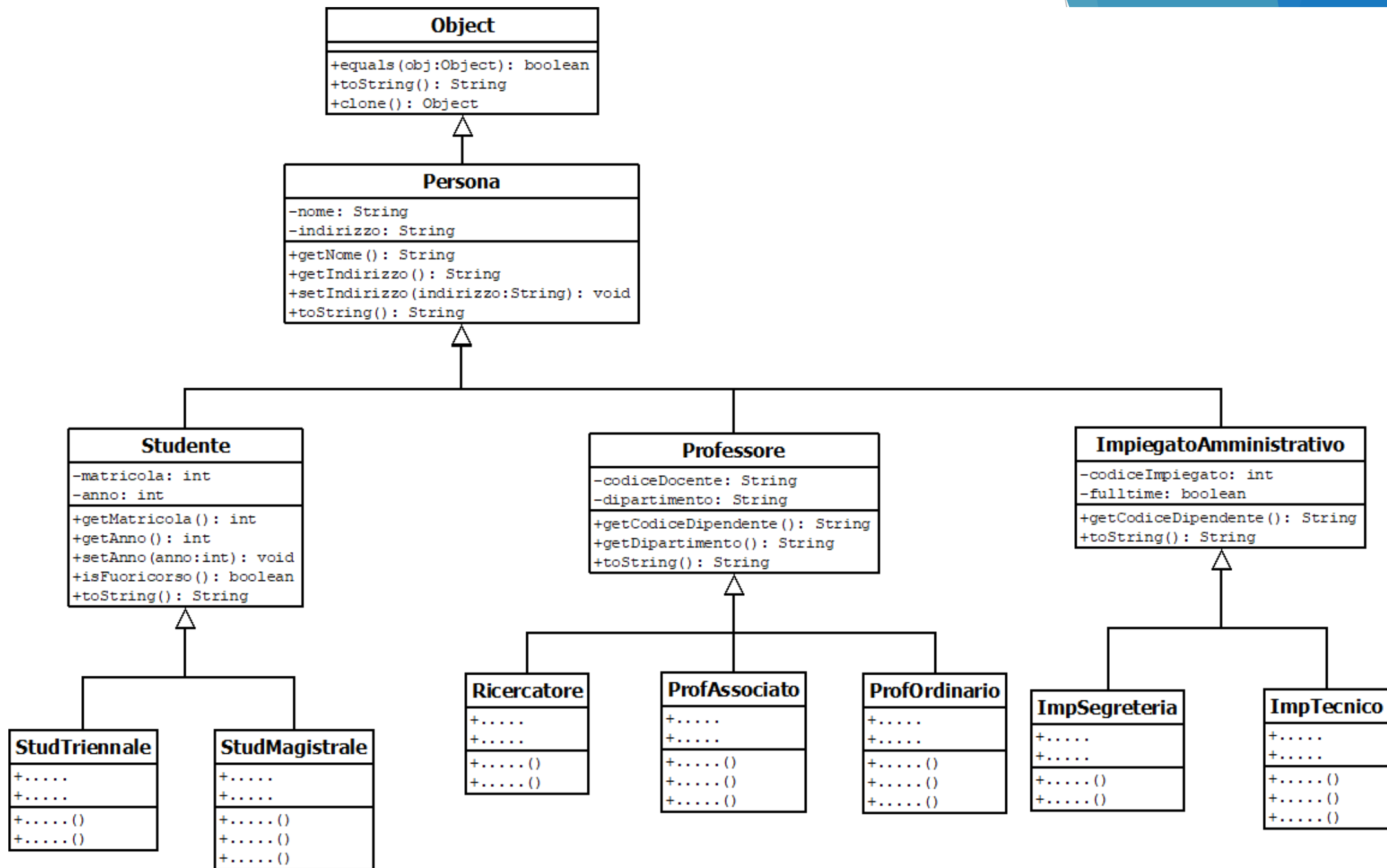
```
public class Sfera extends Solido {  
    // variabili d'istanza (oltre a pesoSpecifico)  
    private double raggio;  
    // costruttore  
    public Sfera(double raggio, double ps) {  
        super(ps); // chiama il costruttore di Solido  
        this.raggio = raggio; // usa this per disambiguare  
    }  
    // implementazione dei metodi astratti di Solido  
    public double volume() {  
        return 4/3 * Math.PI * Math.pow(raggio, 3);  
    }  
    public double superficie() {  
        return 4 * Math.PI * raggio * raggio ;  
    }  
    // peso() è definito nella superclasse  
}
```

Gerarchia di classi e polimorfismo

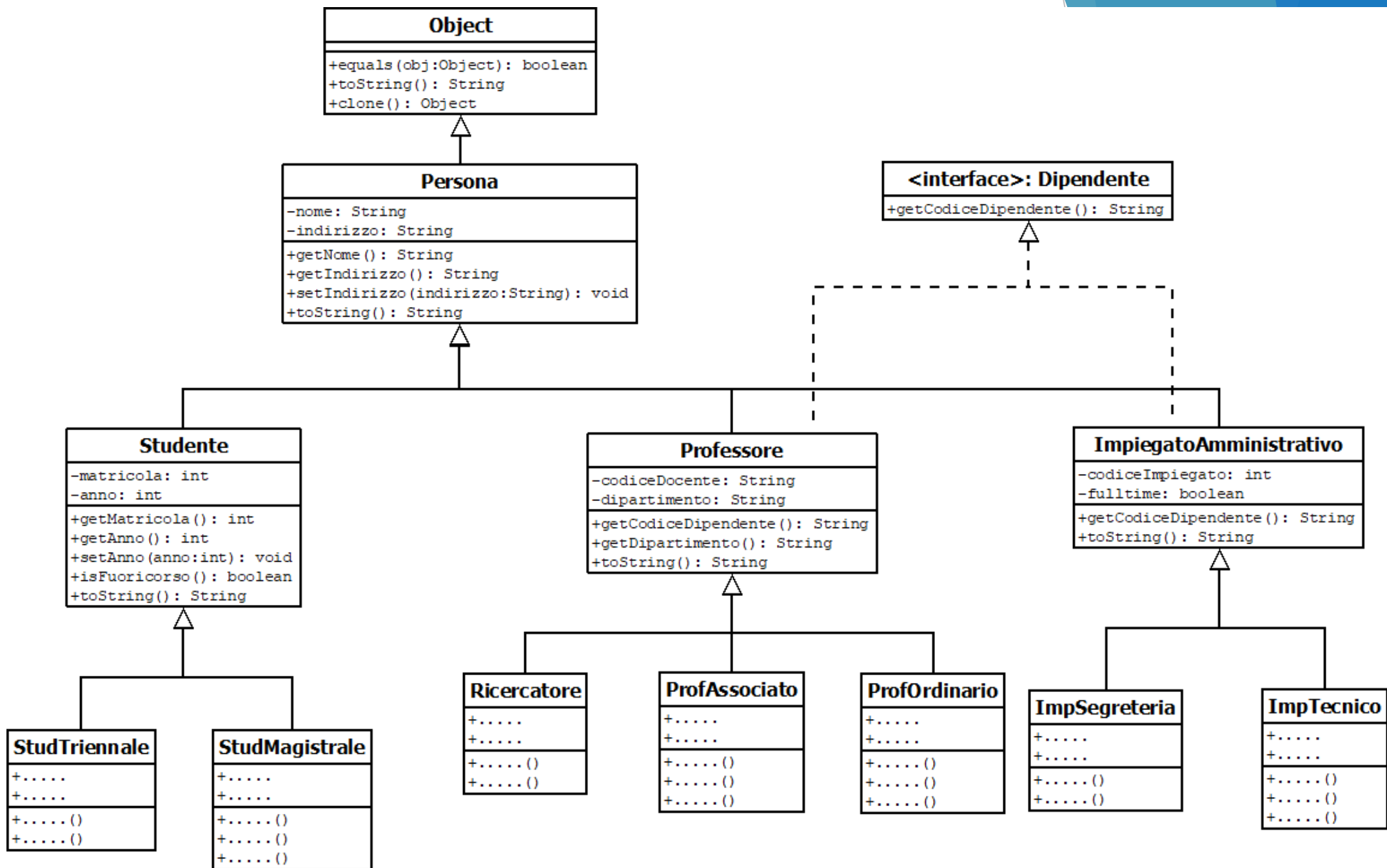
Il costrutto `extends` quindi consente di creare una **gerarchia di classi** rappresentabile come un **albero** la cui radice è `Object`

- ▶ In Java ad ogni **classe** (e interfaccia) è associato un **tipo di oggetto**
- ▶ **Nominal Subtyping** implica che la gerarchia di classi (estesa con anche le interfacce) è una rappresentazione della **relazione di sottotipo** `<:` il cui elemento "**Top**" è **Object**
- ▶ La regola di **subsumption** del sistema di tipi automaticamente ci consente di ottenere un meccanismo di **polimorfismo per sottotipo**

Esempio di gerarchia di classi



Esempio di gerarchia di classi (con interfacce)



```
class Persona {...}

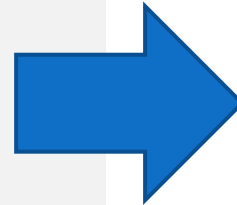
interface Dipendente {...}

class Studente extends Persona {...}
class Professore extends Persona
    implements Dipendente {...}
class ImpAmministrativo extends Persona
    implements Dipendente {...}

class StudTriennale extends Studente {...}
class StudMagistrale extends Studente {...}

class Ricercatore extends Professore {...}
class ProfAssociato extends Professore {...}
class ProfOrdinario extends Professore {...}

class ImpSegreteria extends ImpAmministrativo {...}
class ImpTecnico extends ImpAmministrativo {...}
```



```
Persona <: Object
Dipendente <: Object

Studente <: Persona
Professore <: Persona
Professore <: Dipendente
ImpAmministrativo <: Persona
ImpAmministrativo <: Dipendente

StudTriennale <: Studente
StudMagistrale <: Studente

Ricercatore <: Professore
ProfAssociato <: Professore
ProfOrdinario <: Professore

ImpSegreteria <: ImpAmministrativo
ImpTecnico <: ImpAmministrativo
```

```
class Persona { }
```

```
in
```

Proprietà transitiva $\frac{T_1 <: T_2 \quad T_2 <: T_3}{T_1 <: T_3}$

Subsumption $\frac{\Gamma \vdash_e \text{exp} : S \quad S <: T}{\Gamma \vdash_e \text{exp} : T}$

New $\Gamma \vdash_e \text{new } T() : T$

Assegnamento $\frac{\Gamma \vdash_e \text{exp} : T}{\Gamma \vdash_c T \ x = \text{exp}}$

Con queste regole possiamo derivare:

$\Gamma \vdash_c \text{Persona } p = \text{new StudTriennale}();$

```
Persona <: Object
```

```
Dipendente <: Object
```

```
Studiante <: Persona
```

```
Professore <: Persona
```

```
Professore <: Dipendente
```

```
ImpAmministrativo <: Persona
```

```
ImpAmministrativo <: Dipendente
```

```
StudTriennale <: Studiante
```

```
StudMagistrale <: Studiante
```

```
Ricercatore <: Professore
```

```
ProfAssociato <: Professore
```

```
ProfOrdinario <: Professore
```

```
ImpSegreteria <: ImpAmministrativo
```

```
ImpTecnico <: ImpAmministrativo
```


Structural vs Nominal Subtyping

Teorema

Sia $<:$ la relazione di tipo inferita dalla gerarchia di classi di Java. Per ogni coppia di classi S e T tale che $S <: T$, ogni membro pubblico di T è anche membro pubblico di S

Dimostrazione

Banale... segue dalla definizione di `extends` e `implements` e dalla transitività di $<:$

Conseguenza

Structural subtyping $<:_s$ è una relazione **più debole** del **Nominal Subtyping** $<:_n$

$$S <:_n T \Rightarrow S <:_s T$$

$$S <:_s T \not\Rightarrow S <:_n T$$

Overloading, overriding e
dynamic dispatch

Invocare un metodo...

```
public class Esami {  
    ...  
    public boolean verbalizza(Professore p, Studente s, ...) { ... }  
}  
  
Esami e = new Esami();  
ProfAssociato prof = new ProfAssociato("Mario Rossi",...);  
StudTriennale stud = new StudTriennale("Rosa Bianchi",...);  
e.verbalizza(prof, stud, ...); // chiamata del metodo  
// ProfAssociato <: Professore, StudTriennale <: Studente, OK!
```

Che succede quando **invochiamo un metodo su un oggetto**?

- ▶ Innanzitutto, la chiamata deve superare i **controlli statici** operati dal compilatore
 - ▶ Il **metodo deve essere presente nella classe** che descrive il **tipo apparente** (o **tipo statico**) dell'oggetto, ossia quello della **dichiarazione**
 - ▶ I **parametri attuali** devono essere **compatibili** con il tipo dei **parametri formali** del metodo (sottotipi)

Overloading di metodi

In una classe Java si possono definire più metodi con lo stesso nome (**overloading** di metodi)...

- I metodi in overloading devono avere **FIRME DIVERSE** (parametri diversi in numero o tipo)

Metodo	Firma (Signature)
int getVal()	getVal()
int minimo(int x, int y)	minimo(int, int)
int minimo(int a, int b)	minimo(int, int)
double minimo(double x, double y)	minimo(double, double)
int minimo(int x, int y, int z)	minimo(int, int, int)

Quali di questi metodi potrebbero stare nella stessa classe?

Overloading di metodi

Domande...

- Perché due metodi con la stessa firma non possono stare nella stessa classe?
- Perché il nome dei parametri formali e il tipo del risultato del metodo non fanno parte della firma?

Metodo	Firma (Signature)
int getVal()	getVal()
int minimo(int x, int y)	minimo(int, int)
int minimo(int a, int b)	minimo(int, int)
double minimo(double x, double y)	minimo(double, double)
int minimo(int x, int y, int z)	minimo(int, int, int)

Quali di questi metodi potrebbero stare nella stessa classe?

Overloading di metodi

Domande...

- Perché due metodi con la stessa firma non possono stare nella stessa classe?
- Perché il nome dei parametri formali e il tipo del risultato del metodo non fanno parte della firma?

Risposte:

- La firma riassume le **informazioni** che possono essere **dedotte dalla chiamata** del metodo
 - es. da `obj.minimo(7, 9)` si può dedurre nome del metodo, numero e tipi dei parametri)e che possono essere **usate per capire quale metodo chiamare**
- Il nome dei parametri formali e il tipo restituito **non possono essere dedotti** dalla chiamata

Overloading di metodi

Domande...

- Perché due metodi con la stessa firma non possono stare nella stessa classe?
- Perché il nome dei parametri formali e il tipo del risultato del metodo non fanno parte della firma?

Formalmente:

$$\frac{\forall i \in [1, n]. \Gamma \vdash_e e_i : T_i \quad \text{obj} : S \quad T \text{ m}(T_1 x_1, \dots, T_n x_n) \text{ C} \in S}{\Gamma \vdash_e \text{obj.m}(e_1, \dots, e_n) : T}$$

dove $T \text{ m}(T_1 x_1, \dots, T_n x_n) \text{ C}$ è la definizione del metodo nella classe:

- es: `int somma(int x, int y) { return x+y; }`

Se il metodo non viene trovato?

Si ha un **errore in fase di compilazione**

- ▶ il type checker non può applicare la precedente regola di inferenza

Ma potrebbe essere un problema legato al fatto che il **controllo** è svolto sul **tipo apparente** (**tipo statico**)

- ▶ Se sappiamo che nel tipo effettivo il metodo ci sarà, possiamo fare un **downcast esplicito**
- ▶ Meglio se lo facciamo dopo aver controllato il **tipo effettivo** (**tipo dinamico**) con il predicato **instanceof**
 - ▶ controllo di tipo dinamico che si basa sul **descrittore dell'oggetto**, che contiene il nome della classe usata per fare new

Type coercions (cast) di oggetti in Java

Downcast esplicito con controllo dinamico:

```
Persona p = new Studente();  
if (p instanceof Studente) {  
    Studente s = (Studente) p;  
    int m = s.getMatricola();  
}
```

Gli **upcast** (da sottotipo a supertipo) in Java sono invece **impliciti** (non è necessario scriverli)

► Poiché il polimorfismo per sottotipo (**subsumption**) si applica ovunque

Non si possono fare cast tra tipi di oggetti che non siano in relazione di sottotipo

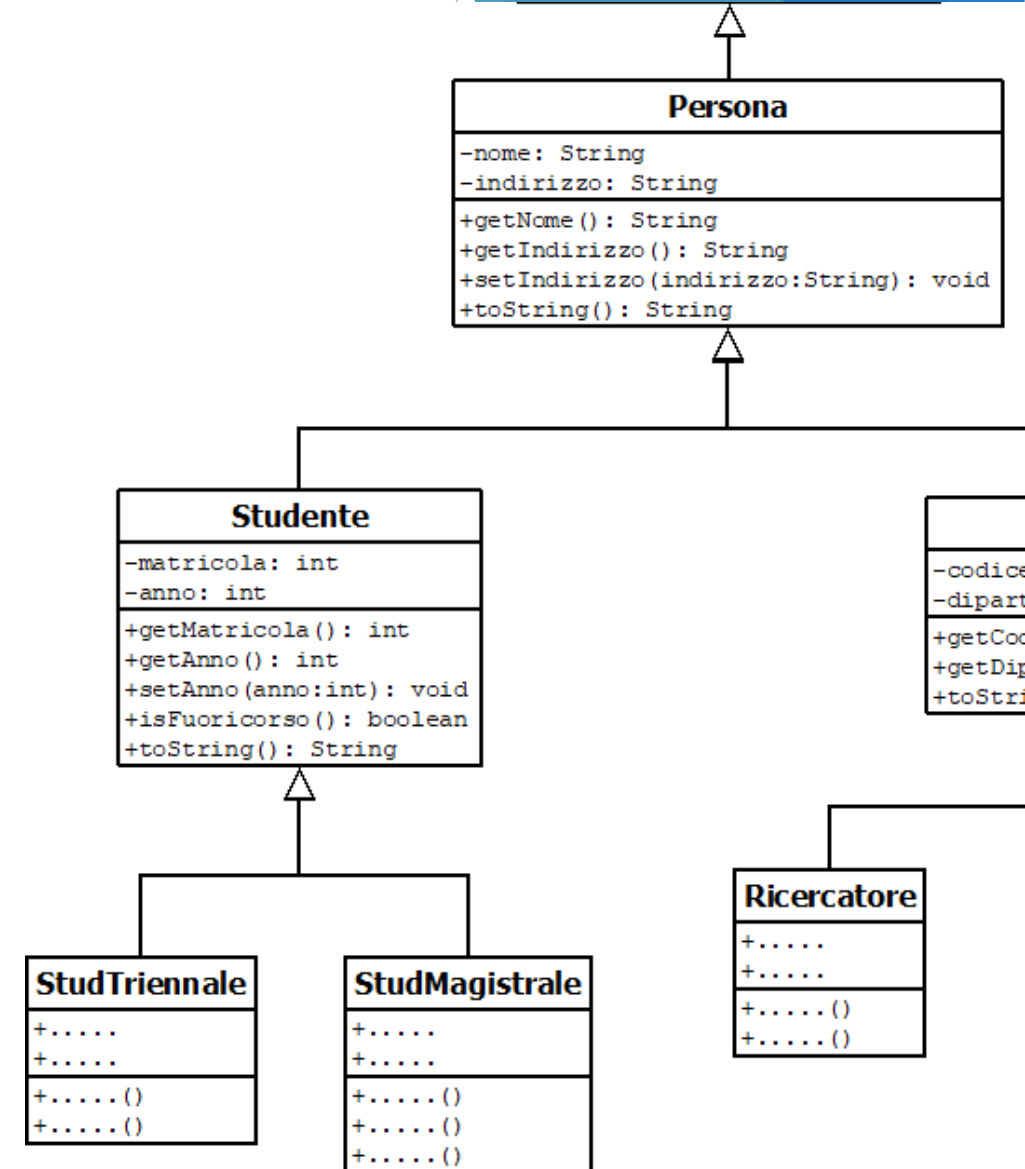
E a tempo di esecuzione?

Dynamic Dispatch dei metodi

Superati i controlli di tipo statici, a tempo di esecuzione la chiamata del metodo può trovarsi nei seguenti casi:

- ▶ Il metodo **è presente nella classe** dell'oggetto
- ▶ Il metodo **va cercato nelle superclassi** (le classi più in alto nella gerarchia)

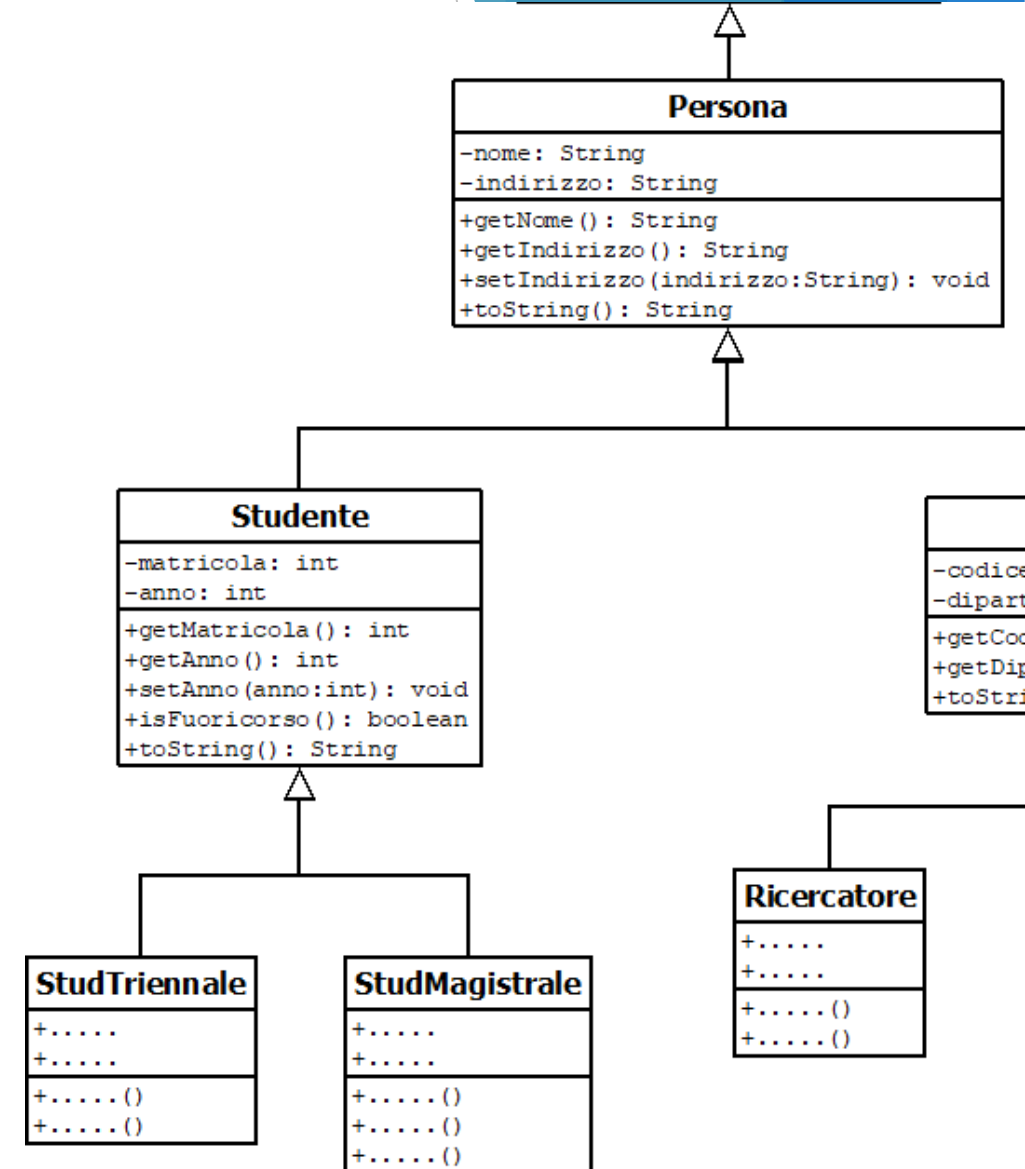
La JVM va alla ricerca del metodo partendo dalla classe corrente e risalendo la gerarchia (**DYNAMIC DISPATCH**)



La ricerca del metodo parte dalla classe che corrisponde al **TIPO EFFETTIVO** dell'oggetto.

La JVM trova l'indicazione sul tipo effettivo nel **descrittore del dato** (oggetto) in memoria, inizializzato al momento della creazione dell'oggetto

stud: <class Studente>	
nome:	"Mario"
indirizzo:	"Via..."
matricola:	14421
anno:	2



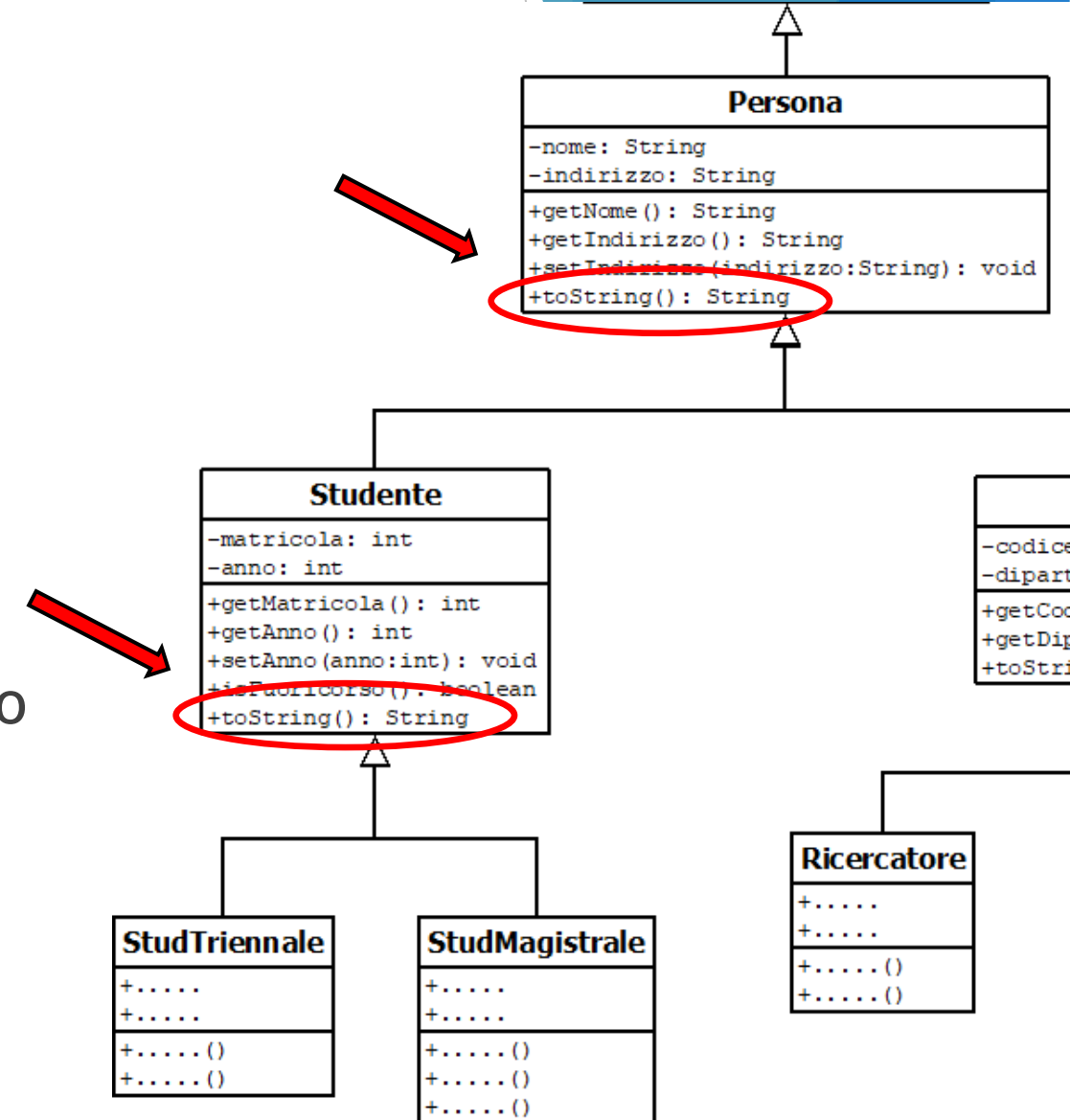
partendo dalla classe corrente e risalendo
la gerarchia (**DYNAMIC DISPATCH**)

Overriding

Risalendo la gerarchia si potrebbero incontrare **più metodi con la stessa firma**

- Classi che hanno ridefinito metodi delle superclassi (**overriding**)

In questo caso la JVM esegue il primo metodo incontrato risalendo la gerarchia (quello ridefinito più recentemente)



Dynamic dispatch: realizzazione efficiente nella JVM


Visitare l'albero della gerarchia di classi ad ogni chiamata di metodo è una soluzione molto inefficiente

La JVM adotta una soluzione che si basa su

- ▶ tabelle di metodi (dispatch vector)
(tabelle con puntatori al codice dei metodi)
- ▶ sharing strutturale
(la tabella di una sottoclasse riprende la struttura della tabella della superclasse aggiungendo righe per i nuovi metodi)

Nella JVM (a runtime)

Descrittore dell'oggetto x



x: <class B>	
a1	10
a2	21
a3	57

CLASS A	
m1	
m2	

CLASS B	
m1	
m2	
m3	

```
class A {  
    int a1, a2;  
    void m1 ...;  
    void m2 ...;  
}
```

```
class B extends A  
    int a3;  
    void m1 ...;  
    void m3 ...;  
}
```




Tabella dei metodi della classe B
(dispatch vector)



Codice nell'ambiente delle classi

Sharing strutturale

La tabella della sottoclasse **riprende la struttura** (ordine delle righe) della tabella della superclasse

CLASS A	
m1	
m2	

CLASS B	
m1	
m2	
m3	


Esempio: il metodo **m2** è il **secondo** nella tabella di **A** e di **tutte le sue sottoclassi** (anche se magari i puntatori sono diversi a causa di overriding)

Dynamic dispatch: realizzazione efficiente nella JVM

Con questa soluzione, l'operazione di **dispatching dei metodi** la si può risolvere **STATICAMENTE**

- ▶ Il **compilatore determina l'offset** del metodo nella tabella (corrisponde alla posizione in tabella)
- ▶ L'offset è determinato sul **tipo apparente** dell'oggetto
- ▶ A tempo di esecuzione la JVM accede alla tabella della classe che è **tipo effettivo** usando quell'offset
- ▶ Anche se il tipo effettivo dovesse essere **diverso** da quello apparente, corrisponderà comunque ad una sua **sottoclasse**, e grazie allo **sharing strutturale** la JVM accederà comunque al **metodo giusto**
- ▶ Se la sottoclasse ha fatto **overriding** del metodo, il **puntatore** che si troverà in tabella **riferirà alla nuova versione** del metodo

Dynamic dispatch: realizzazione efficiente nella JVM



```
A x = new B();  
x.m2();
```

```
class A {  
    int a1, a2;  
    void m1 ...;  
    void m2 ...;  
}
```

Index

← 0

← 1

Esempio:

- ▶ x è di **tipo apparente A**
- ▶ il compilatore trova il metodo m2 in seconda posizione nel codice della classe A (**indice 1**)
- ▶ il compilatore **traduce** la chiamata nel **bytecode** usando la seguente operazione (in bytecode)

```
invokevirtual #1
```



Index del metodo

x: <class B>	
a1	10
a2	21
a3	57

x.m2 () ;

CLASS A	
m1	
m2	

CLASS B	
m1	
m2	
m3	

```
class A {
    int a1, a2;
    void m1 ...;
    void m2 ...;
}
```

```
class B extends A
    int a3;
    void m1 ...;
    void m3 ...;
}
```

- ▶ A tempo di **esecuzione** x avrà **tipo effettivo B**
- ▶ m2 in B ha la **stessa posizione** che in A (per lo **sharing strutturale**)
- ▶ **invokevirtual #1 chiama il metodo giusto!**
- ▶ In caso di **overriding** (es. m1) avrebbe chiamato la **nuova versione** del metodo (seguendo il puntatore)

Dynamic dispatch: realizzazione efficiente nella JVM

Questa soluzione fa sì che le chiamate di metodo possano essere eseguite in **tempo costante**

- ▶ **Niente visita** dell'albero delle classi
- ▶ **Accesso diretto** tramite offset e puntatore

Accesso alle variabili d'istanza

Come fa un **metodo (non statico)** ad **accedere alle variabili d'istanza** dell'oggetto?

► Il compilatore aggiunge **this** come **parametro (implicito)** al metodo

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

e (**intuitivamente**, in realtà è un po' più complicato) lo **compila** in

```
public String getMatricola(Studente this) { return this.matricola; }
```

a **tempo di esecuzione** sarà passato il riferimento all'oggetto su cui il metodo è chiamato

Chiamata di metodi statici

Metodi statici: appartengono alla classe e non possono accedere a variabili di istanza (non hanno il parametro implicito `this`)

A **run-time** sono implementati come procedure a top-level (globali)

- ▶ **Dispatch vector dedicato** ai metodi statici acceduto tramite l'operazione **invokestatic** nel bytecode
- ▶ Sostanzialmente non sono metodi...