

Appunti di Progettazione del Software

Canale 1 · Prof. Paolo Liberatore

Capitolo 3: Generalizzazione ed ereditarietà

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: Sottoinsiemi e generalizzazione in UML · extends · Proprietà ereditate
Tipi delle variabili · Gerarchie di classi · Costruttori e super
Una gerarchia di figure · Stub · Ridefinizione e toString

Indice

1	Sottoinsiemi e generalizzazione in UML	1
2	La generalizzazione in Java: extends	2
3	Associazioni ereditate e tipi delle variabili	3
4	Gerarchie di classi	4
5	Costruttori nelle sottoclassi	6
6	Una gerarchia di figure	7
7	Ridefinire un metodo: <code>toString</code>	10
8	Esercizi	12
9	Soluzioni	13

1 Sottoinsiemi e generalizzazione in UML

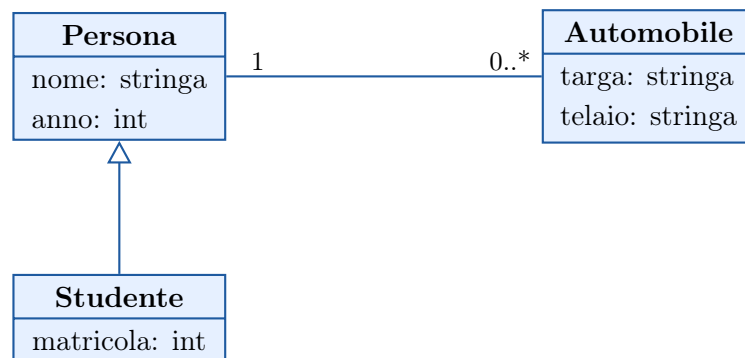
Definizione – Generalizzazione

Fra due classi A e B c'è una **generalizzazione** (o relazione **is-a**) se ogni istanza di A è anche istanza di B :

$$A \subseteq B.$$

A è la **sottoclasse**, B la **superclasse**. Nel diagramma delle classi si disegna una freccia con la punta a triangolo vuoto, da A verso B .

Gli studenti universitari sono persone che in più hanno un numero di matricola: **Studente** è una sottoclasse di **Persona**.



In **Studente** non si riscrivono nome e anno: li ha perché ogni studente è una persona. La freccia dice anche che gli studenti sono un sottoinsieme *di interesse per l'applicazione*: di sottoinsiemi delle persone ce ne sono infiniti, e si disegnano solo quelli che servono.

Nota – Linea e freccia

	Associazione	Generalizzazione
nel diagramma	linea	freccia a triangolo vuoto
come insiemi	$R \subseteq \text{Persona} \times \text{Automobile}$	$\text{Studente} \subseteq \text{Persona}$
in Java	campi scritti a mano (Capitolo 2)	parola chiave extends

Confondere i due simboli è un errore frequente negli esercizi: la linea collega istanze di classi diverse, la freccia dice che una classe è contenuta nell'altra.

Osservazione – Il criterio è il sottoinsieme, non i campi comuni

Si dichiara una generalizzazione solo se una classe è davvero un sottoinsieme dell'altra. Una città con nome, anno di fondazione e numero di abitanti ha gli stessi campi di una persona (una stringa e un intero) più uno, esattamente come uno studente; ma una città non è una persona, e fra **Città** e **Persona** non c'è generalizzazione. Quando si costruisce la gerarchia ci si chiede *chi sta dentro chi*, non quali componenti una classe aggiunge.

2 La generalizzazione in Java: **extends**

Definizione – extends

La dichiarazione `class A extends B { ... }` dice che la classe **A** è una sottoclasse di **B**: ogni oggetto di **A** è anche un oggetto di **B**.

Osservazione – Un falso amico

In inglese *extends* significa “estende”, ma in Java va letto “è un sottoinsieme di”. Leggerlo come “aggiunge campi” porta a scegliere male la gerarchia, come per la città.

Le classi della Sezione 1, per ora senza costruttori (Sezione 5):

```
public class Persona {  
    String nome;  
    int anno;    // anno di nascita  
}  
  
public class Studente extends Persona {  
    int matricola;  
}
```

Proposizione – Ereditarietà

Una sottoclasse ha tutte le proprietà della superclasse senza doverle dichiarare: campi, metodi e associazioni. In più può avere proprietà proprie.

Il principio è un sillogismo:

1. ogni persona ha un nome e un anno di nascita;
2. ogni studente è una persona;
3. quindi ogni studente ha un nome e un anno di nascita.

Nel programma i campi ereditati si usano come quelli propri:

```

Studente s = new Studente();
s.nome = "Anna";
s.anno = 2005;
s.matricola = 1234567;
System.out.println(s.nome + " " + s.anno + " " + s.matricola);
// stampa: Anna 2005 1234567
    
```

Una sottoclasse può anche aggiungere solo metodi, senza campi nuovi, oppure non aggiungere niente: resta un sottoinsieme dichiarato, con le stesse proprietà della superclasse.

3 Associazioni ereditate e tipi delle variabili

Le associazioni della superclasse valgono anche per le sottoclassi. Se ogni persona può possedere automobili, anche uno studente può: se **Persona** ha l'insieme **auto** del Capitolo 2, ce l'ha anche **Studente**. E il proprietario di un'automobile, che è una persona, può essere uno studente.

Proposizione – Compatibilità dei tipi

In una variabile di tipo B (una variabile locale, un campo, un elemento di una collezione) si può mettere un riferimento a un oggetto di una sottoclasse di B . Non vale il contrario: in una variabile di una sottoclasse non si può mettere un oggetto della superclasse.

Con s studente e p persona (creata con `new Persona()`):

```

Automobile a = new Automobile("AB123CD", "ZFA1990000123456");
a.proprietario = s;           // proprietario è una Persona: va bene

HashSet<Persona> rubrica = new HashSet<Persona>();
rubrica.add(p);
rubrica.add(s);               // in una rubrica ci sono studenti e non studenti

HashSet<Studente> iscritti = new HashSet<Studente>();
iscritti.add(s);
iscritti.add(p);              // errore di compilazione
    
```

L'ultima riga non compila: *incompatible types: Persona cannot be converted to Studente*. Gli iscritti sono studenti, e **p** è una persona che non è uno studente.

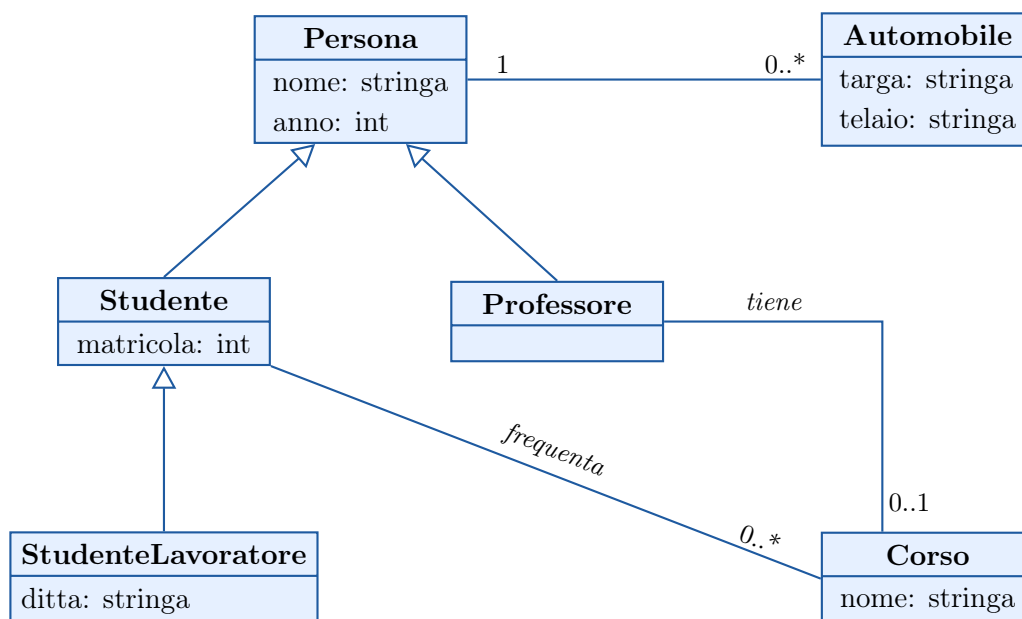
Nota – Il tipo della variabile e la classe dell'oggetto

La classe di un oggetto la decide **new** e non cambia più: **new Persona()** crea una persona che *non* è uno studente, **new Studente()** uno studente, che è anche una persona. Una variabile di tipo **Persona** può contenere un riferimento all'uno o all'altro.

4 Gerarchie di classi

Una classe può avere più sottoclassi, e una sottoclasse può avere a sua volta sottoclassi.

- **Professori**: persone che tengono un corso. **Professore** *extends* **Persona** ha l'associazione *tiene* con **Corso**, che le altre persone non hanno.
- **Studenti lavoratori**: studenti iscritti come lavoratori, di cui interessa la ditta. **StudenteLavoratore** *extends* **Studente**, quindi è anche un sottoinsieme delle persone.



Il nome di un'associazione, come *tiene*, si scrive accanto alla linea. Nello schema:

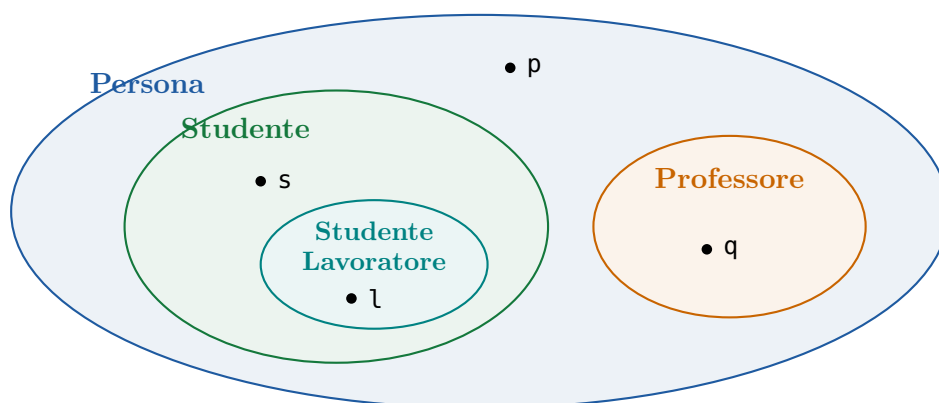
- l'attributo **nome** di **Persona** vale in tutte le sottoclassi, ma non in **Automobile**, e il nome di un corso è un attributo diverso;
- professori, studenti e studenti lavoratori possono possedere automobili, come tutte le persone;
- solo i professori tengono un corso; uno studente può frequentarne, e con lui uno studente lavoratore.

Conta solo la direzione delle frecce, non la posizione dei rettangoli: una sottoclasse disegnata di lato resta una sottoclasse.

Proposizione – Classi disgiunte

In Java due classi sono sempre in uno di due casi:

- una è sottoclasse dell'altra, direttamente o attraverso altre classi;
- sono **disgiunte**: nessun oggetto appartiene a entrambe.



*Le classi della gerarchia come insiemi: p è una persona che non è né studente né professore, l è studente lavoratore, studente e persona. **Studente** e **Professore** sono disgiunti.*

Osservazione – Un limite di Java

Niente vieta a un professore di iscriversi a un corso di laurea, ma con queste classi un oggetto è un professore oppure uno studente, non entrambi. Java rappresenta bene i sottoinsiemi, ma non due sottoinsiemi che si sovrappongono solo in parte.

5 Costruttori nelle sottoclassi

Proposizione – Costruttori e super

Se la superclasse ha solo costruttori con argomenti, allora:

- ogni sottoclasse deve avere un proprio costruttore;
- la prima istruzione di quel costruttore è `super(...)`, che chiama il costruttore della superclasse con gli argomenti che gli servono.

```
import java.util.HashSet;

public class Persona {
    String nome;
    int anno;
    HashSet<Automobile> auto;

    Persona(String nome, int anno) {
        this.nome = nome;
        this.anno = anno;
        this.auto = new HashSet<Automobile>();
    }
}
```

Il costruttore non riceve tutti i campi: l'insieme delle auto nasce vuoto. Da qui `new Persona()` non compila più, perché `Persona` non ha più il costruttore di default (nota “Costruttore di default” del Capitolo 2).

Per la stessa ragione non compila più nemmeno `Studente`: senza un costruttore proprio, la sottoclasse proverebbe a creare la parte `Persona` senza argomenti. Il compilatore segnala l'errore sulla sottoclasse, non su `Persona`: *constructor Persona in class Persona cannot be applied to given types*.

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

    Studente(String nome, int anno, int matricola) {
        super(nome, anno); // costruttore di Persona
        this.matricola = matricola;
    }
}

public class StudenteLavoratore extends Studente {
    String ditta;

    StudenteLavoratore(String nome, int anno, int matricola, String ditta) {
        super(nome, anno, matricola); // costruttore di Studente
        this.ditta = ditta;
    }
}
```

Il costruttore di **Studente** riceve nome, anno e matricola, perché uno studente ha tutti e tre. Non ripete però `this.nome = nome` e `this.anno = anno`: quel codice c'è già nel costruttore di **Persona**, e `super(nome, anno)` lo esegue. Oltre che con `new`, un costruttore si invoca così, dalla prima riga del costruttore di una sottoclasse.

Che cosa fa `new StudenteLavoratore(...)`.

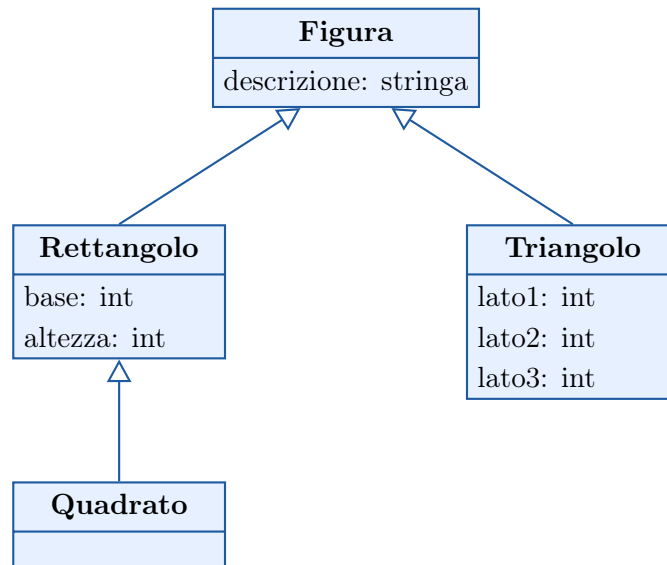
1. Crea l'oggetto, con tutti i campi: nome, anno, auto, matricola, ditta.
2. Il costruttore di **StudenteLavoratore** chiama quello di **Studente**, che chiama quello di **Persona**: si inizializzano nome, anno e l'insieme delle auto.
3. Si torna in **Studente**, che assegna la matricola, e infine in **StudenteLavoratore**, che assegna la ditta.

Nota – Da Java 25

Fino a Java 24 un'istruzione prima di `super(...)` è un errore. Da Java 25 il compilatore ammette prima di `super(...)` alcune istruzioni, purché non leggano i campi dell'oggetto né ne chiamino i metodi; scrivere `super(...)` come prima istruzione resta corretto in ogni versione.

6 Una gerarchia di figure

Ogni figura geometrica ha una descrizione. I rettangoli sono figure con base e altezza, i triangoli figure con tre lati, i quadrati rettangoli particolari.



```

public class Figura {
    String descrizione;

    Figura(String descrizione) {
        this.descrizione = descrizione;
    }
}

public class Rettangolo extends Figura {
    int base;
    int altezza;

    Rettangolo(String descrizione, int base, int altezza) {
        super(descrizione);
        this.base = base;
        this.altezza = altezza;
    }

    double area() {
        return base * altezza;
    }

    double perimetro() {
        return 2 * (base + altezza);
    }
}

```

Con `new Rettangolo("tavolo", 4, 5)`, `perimetro()` restituisce $2 \cdot (4+5) = 18$ e `area()` restituisce 20.

```
public class Triangolo extends Figura {
    int lato1;
    int lato2;
    int lato3;

    Triangolo(String descrizione, int lato1, int lato2, int lato3) {
        super(descrizione);
        this.lato1 = lato1;
        this.lato2 = lato2;
        this.lato3 = lato3;
    }

    double perimetro() {
        return lato1 + lato2 + lato3;
    }

    double area() {
        return 0;    // stub: la formula di Erone è da scrivere
    }
}
```

Definizione – Stub

Un **metodo stub** ha l'intestazione completa ma un corpo provvisorio, che restituisce un valore qualsiasi del tipo dichiarato. Si può già chiamare, e il resto del programma si scrive e si prova prima che il metodo sia finito.

Un metodo dichiarato **double** deve restituire un valore: senza **return**, **area()** non compila (*missing return statement*), e **return 0** basta a farlo compilare. Lo stub serve anche a dividere il lavoro: chi ha bisogno di una classe che un altro non ha ancora finito lavora intanto con i suoi stub.

Il quadrato non ha componenti in più del rettangolo: è un rettangolo con base e altezza uguali. Il suo costruttore riceve quindi un solo lato e lo passa due volte a **super**:

```
public class Quadrato extends Rettangolo {
    Quadrato(String descrizione, int lato) {
        super(descrizione, lato, lato);
    }
}
```

Il costruttore di una sottoclasse non deve avere gli stessi parametri di quello della superclasse: riceve ciò che serve a costruire i suoi oggetti. **Quadrato** eredita **area()** e **perimetro()**: con lato 2 danno 4 e 8.

Osservazione – Quadrato extends Rettangolo, non il contrario

Il rettangolo ha un dato in più del quadrato, ma **Rettangolo extends Quadrato** sarebbe sbagliato. Una sottoclasse ha due conseguenze:

1. eredita le proprietà della superclasse (ereditarietà);
2. un suo oggetto va dovunque serve un oggetto della superclasse (compatibilità).

Con la gerarchia rovesciata la prima funzionerebbe ancora, la seconda no: in un insieme di rettangoli non entrerebbero i quadrati. Decide il sottoinsieme: i quadrati sono rettangoli, non il viceversa.

```
HashSet<Rettangolo> r = new HashSet<Rettangolo>();
r.add(new Rettangolo("tavolo", 4, 5));
r.add(new Quadrato("piastrella", 2));    // un quadrato è un rettangolo
r.add(new Triangolo("vela", 3, 2, 4));    // errore di compilazione
```

L'ultima riga non compila: *incompatible types: Triangolo cannot be converted to Rettangolo*. Avere tre lati invece di due non rende il triangolo un rettangolo con un lato in più. Con **Rettangolo extends Quadrato** l'errore passerebbe alla riga del quadrato: *Quadrato cannot be converted to Rettangolo*.

7 Ridefinire un metodo: toString

Definizione – toString

Il metodo **public String toString()** restituisce una descrizione dell'oggetto come stringa. Con **x** riferimento a un oggetto, **System.out.println(x)** stampa la stringa restituita da **x.toString()**.

Ogni classe ha un **toString**, anche senza scriverlo: quello predefinito dà il nome della classe e un codice, per esempio **Figura@1dbd16a6**. Per stampare i dati dell'oggetto lo si scrive nella classe.

Proposizione – Ridefinizione (overriding)

Se una sottoclasse definisce un metodo con lo stesso nome e gli stessi parametri di uno ereditato:

- per gli oggetti della sottoclasse si esegue il metodo della sottoclasse;
- vale qualunque sia il tipo della variabile: conta la classe dell'oggetto, decisa da **new**.

```
// in Figura
public String toString() {
    return "Figura [descrizione=" + descrizione + "]";
}

// in Rettangolo
public String toString() {
    return "Rettangolo [descrizione=" + descrizione
        + ", base=" + base + ", altezza=" + altezza + "]";
}

// in Quadrato
public String toString() {
    return "Quadrato [descrizione=" + descrizione + ", lato=" + altezza + "]";
}
```

Il `toString` di una sottoclasse sostituisce quello della superclasse, non lo completa: i campi ereditati, come `descrizione`, si scrivono anche qui, se devono comparire. Va dichiarato `public`: senza, il compilatore rifiuta la ridefinizione (*attempting to assign weaker access privileges; was public*).

```
Figura f = new Quadrato("piastrella", 2);
System.out.println(new Rettangolo("tavolo", 4, 5));
System.out.println(f);
// Rettangolo [descrizione=tavolo, base=4, altezza=5]
// Quadrato [descrizione=piastrella, lato=2]
```

`f` è di tipo `Figura`, ma l'oggetto è un quadrato: si esegue il `toString` di `Quadrato`. Allo stesso modo, in un `HashSet<Figura>` si mettono rettangoli, triangoli e quadrati, e stampando l'insieme ogni elemento compare con il proprio `toString`. Esercizio 1.

Riepilogo

- **Generalizzazione:** $A \subseteq B$, ogni istanza della sottoclasse è istanza della superclasse; in UML una freccia a triangolo vuoto verso la superclasse (la linea è un'associazione).
- **extends:** `A extends B` si legge “*A* è un sottoinsieme di *B*”; si usa solo per un vero sottoinsieme, non per classi con campi simili.
- **Ereditarietà:** la sottoclasse ha campi, metodi e associazioni della superclasse, più i propri.
- **Compatibilità:** in una variabile di tipo *B* va un oggetto di una sottoclasse, non il contrario.
- **Classi disgiunte:** due classi Java sono una dentro l'altra oppure disgiunte.
- **Costruttori:** se la superclasse ha solo costruttori con argomenti, ogni sottoclasse ne ha uno che comincia con `super(...)`.
- **Stub:** metodo con l'intestazione completa e un corpo provvisorio.
- **Ridefinizione:** un metodo con lo stesso nome e gli stessi parametri sostituisce quello ereditato; si esegue quello della classe dell'oggetto, come per `toString`.

8 Esercizi

Esercizio 1 – Perimetro totale

Con le classi `Figura`, `Rettangolo`, `Triangolo` e `Quadrato` delle Sezioni 6 e 7, scrivere un metodo che, dato un `HashSet<Figura>`, restituisca la somma dei perimetri delle figure. Serve una piccola modifica alla classe `Figura`.

Soluzione a pagina 13 →

9 Soluzioni

Soluzione dell'Esercizio 1 – Perimetro totale

Si scorre l'insieme sommando i perimetri:

```
static double perimetroTotale(HashSet<Figura> figure) {  
    double totale = 0;  
    for (Figura f : figure) {  
        totale = totale + f.perimetro();  
    }  
    return totale;  
}
```

Così com'è non compila: `f` è di tipo `Figura`, e `Figura` non ha un metodo `perimetro()` (*cannot find symbol*). La modifica è uno stub in `Figura`:

```
// in Figura  
double perimetro() {  
    return 0;    // stub: le sottoclassi lo ridefiniscono  
}
```

Ora `f.perimetro()` compila, e per la ridefinizione si esegue il perimetro della classe dell'oggetto: quello di `Rettangolo` per rettangoli e quadrati, quello di `Triangolo` per i triangoli. Con il rettangolo 4×5 , il triangolo di lati 3, 2, 4 e il quadrato di lato 2 il totale è $18 + 9 + 8 = 35$. Una figura creata con `new Figura(...)` conta 0.

[← Torna all'Esercizio 1, pagina 12](#)