

17 - Ereditarietà, tipi e gerarchie

Programmazione e analisi di dati
Modulo A: Programmazione in Java

Paolo Milazzo

Dipartimento di Informatica, Università di Pisa
<http://www.di.unipi.it/~milazzo>
milazzo@di.unipi.it

Corso di Laurea Magistrale in Informatica Umanistica
A.A. 2013/2014

Sommario

- 1 Ancora sull'ereditarietà
 - Esempio: gestione di un teatro
 - Late binding
- 2 Ereditarietà e controllo dei tipi
- 3 Gerarchie di classi di Java
 - La classe Object e i suoi metodi

Ereditarietà

Abbiamo visto che tramite i meccanismi di **ereditarietà** di Java si possono definire classi che **estendono** altre classi già definite

```
public class Studente extends Persona { ... }
```

Questo consente di:

- raggruppare i membri comuni di più classi (es. Studente e Professore) in un'unica classe (es. Persona)
- definire una nuova classe che è un “caso particolare” di una già definita ri-definendo (**overriding**) solo i metodi che differiscono

Persona-Studente-Professore: richiamo (1)

```
public class Persona {  
  
    private String nome;           // nome e cognome  
    private String indirizzo;     // indirizzo  
  
    // costruttore  
    public Persona(String nome, String indirizzo) {  
        this.nome = nome;  
        this.indirizzo = indirizzo;  
    }  
  
    public String getNome() { return nome; }  
  
    public String getIndirizzo() { return indirizzo; }  
  
    public void setIndirizzo(String indirizzo) {  
        this.indirizzo = indirizzo;  
    }  
  
    // visualizza i dati della persona  
    public void visualizza() {  
        System.out.println("    Nome: " + nome);  
        System.out.println("Indirizzo: " + indirizzo);  
        System.out.println();  
    }  
}
```

Persona-Studente-Professore: richiamo (2)

```
public class Studente extends Persona {  
  
    private int matricola;    // numero di matricola  
    private int anno;        // anno di frequentazione  
  
    private static int ultimaMatricola = 0;  
  
    // costruttore  
    public Studente(String nome, String indirizzo) {  
  
        super(nome, indirizzo);  
  
        this.matricola = ultimaMatricola + 1;  
        ultimaMatricola++;  
        this.anno = 1;  
    }  
}
```

(segue)

Persona-Studente-Professore: richiamo (3)

(segue Studente)

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

public void setAnno(int anno) {
    if (anno>0) this.anno = anno;
}

public boolean isFuoricorso() { return (anno>5); }

// stampa le informazioni sullo studente
public void visualizza() {
    System.out.println("    Nome: " + getNome());
    System.out.println("Indirizzo: " + getIndirizzo());
    System.out.println("Matricola: " + matricola);
    System.out.println("    Anno: " + anno);
    if (isFuoricorso())
        System.out.println("    ( Studente fuoricorso )");
    else
        System.out.println("    ( Studente in corso )");
    System.out.println();
}
}
```

Persona-Studente-Professore: richiamo (4)

```
public class Professore extends Persona {

    private String codiceDocente; // codice del docente
    private String dipartimento; // dipartimento di appartenenza

    // costruttore
    public Professore(String nome, String indirizzo,
                     String codiceDocente,
                     String dipartimento) {

        super(nome, indirizzo);
        this.codiceDocente = codiceDocente;
        this.dipartimento = dipartimento;
    }

    public String getCodiceDocente() { return codiceDocente; }

    public String getDipartimento() { return dipartimento; }

    // stampa le informazioni sul professore
    public void visualizza() {
        System.out.println("        Nome: Prof. " + getNome());
        System.out.println("    Indirizzo: " + getIndirizzo());
        System.out.println("        Codice: " + codiceDocente);
        System.out.println("Dipartimento: " + dipartimento);
        System.out.println();
    }
}
```

Gestione di un teatro (1)

Usando ancora l'esempio Persona-Studente-Professore visto nella lezione precedente proviamo a scrivere un programma per la **gestione di un teatro**.

Realizziamo una classe **Spettacolo** che rappresenta uno spettacolo teatrale:

- includerà il **titolo** dello spettacolo e il nome della **compagnia** teatrale che lo rappresenta
- includerà la **data** e l'**ora** dello spettacolo
- includerà l'elenco degli **spettatori** (persone) assumendo che vi siano 100 posti disponibili
- consentirà di **aggiungere nuovi spettatori** (prenotare posti) fino all'esaurimento dei posti
- consentirà di **stampare l'elenco** degli spettatori (prenotazioni)

Gestione di un teatro (2)

```
public class Spettacolo {

    private String titolo;        // titolo dello spettacolo
    private String compagnia;     // compagnia teatrale
    private String dataora;      // data e ora spettacolo

    // array che contiene gli spettatori (classe Persona)
    private Persona[] spettatori;

    // contatore dei posti prenotati (inizialmente zero)
    private int postiPrenotati = 0;

    // costante condivisa che indica la capienza del teatro
    private static final int CAPIENZA=100;

    // costruttore
    public Spettacolo(String titolo, String compagnia,
                      String dataora) {
        this.titolo = titolo;
        this.compagnia = compagnia;
        this.dataora = dataora;

        // inizializza l'array (inizialmente pieno di null)
        this.spettatori = new Persona[CAPIENZA];
    }
}
```

(segue)

Gestione di un teatro (3)

(segue Spettacolo)

```
// restituisce il numero di posti ancora disponibili
public int postiDisponibili() {
    return CAPIENZA-postiPrenotati;
}

// consente di prenotare un nuovo posto
// restituisce false se non ci sono posti disponibili
public boolean prenota(Persona spettatore) {
    if (postiDisponibili()>0) {
        spettatori[postiPrenotati] = spettatore;
        postiPrenotati++;
        return true;
    }
    else return false;
}

// stampa l'elenco delle prenotazioni
public void stampaPrenotazioni() {
    System.out.println("Spettacolo " + titolo);
    System.out.println("Del " + dataora);
    System.out.println(); // riga vuota

    for (int i=0; i<postiPrenotati; i++) {
        spettatori[i].visualizza();
    }
}
}
```

Gestione di un teatro (4)

Un main....

```
public class UsaSpettacolo {
    public static void main(String args[]) {

        // crea un paio di persone
        Persona p1 = new Persona("Mario Rossi","Via Garibaldi 23");
        Persona p2 = new Persona("Federico Bianchi","Via Mazzini 44");

        // crea uno spettacolo
        Spettacolo s = new Spettacolo("Macbeth","Attori dilettanti",
                                     "25/12/2013 - 21.00");

        // prenota posti per le due persone
        boolean ok1 = s.prenota(p1);
        boolean ok2 = s.prenota(p2);

        // controlla che sia andato tutto bene
        if (!ok1 || !ok2) System.out.println("Problemi...");

        // stampa l'elenco delle prenotazioni
        s.stampaPrenotazioni();

        // stampa i posti ancora disponibili
        System.out.println("Posti disponibili: " +s.postiDisponibili());
    }
}
```

Gestione di un teatro (5)

Risultato dell'esecuzione:

Spettacolo Macbeth
Del 25/12/2013 - 21.00

Nome: Mario Rossi
Indirizzo: Via Garibaldi 23

Nome: Federico Bianchi
Indirizzo: Via Mazzini 44

Posti disponibili: 98

Gestione di un teatro (6)

Benissimo... e se volessi prenotare un posto per uno studente o un professore?

- La classe in questi casi non è Persona, ma Studente o Professore

Ragioniamo:

- studenti e professori **sono** a tutti gli effetti persone...
- corrispondentemente, grazie ai meccanismi di ereditarietà, le classi Studente e Professore **possiedono tutti i membri** della classe Persona
 - ▶ magari ne hanno qualcuno **in più**
 - ▶ magari ne hanno **ridefinito** qualcuno

ma ci sono tutti!

- una classe (come Spettacolo) che usa Persona usa quindi membri che sono anche in Studente e Professore
- **quindi**: non c'e' motivo per cui Spettacolo non debba funzionare con studenti e professori

Gestione di un teatro (7)

Regola generale:

E' sempre possibile utilizzare un'oggetto di una sottoclasse (es. *Studente*)
dovunque sia richiesto un oggetto della superclasse (es. *Persona*)

Attenzione:

- Affinché possa avvenire tale sostituzione, tra le due classi deve valere una relazione superclasse-sottoclasse sancita dalla parola chiave **extends**
- Non è sufficiente che le due classi contengano gli stessi metodi...
(senza **extends**)

Gestione di un teatro (8)

Un altro main

```
public class UsaSpettacolo2 {
    public static void main(String args[]) {

        // crea una persona, uno studente e un professore
        Persona pers = new Persona("Mario Rossi","Via Garibaldi 23");
        Studente stud = new Studente("Gianni Bianchi","Via Gramsci 88");
        Professore prof = new Professore("Luca Neri","Via Belli 11",
                                         "a11233","Dip. Matematica");

        // crea uno spettacolo
        Spettacolo s = new Spettacolo("Macbeth","Attori dilettanti",
                                     "25/12/2013 - 21.00");

        // prenota posti per tutti
        boolean ok1 = s.prenota(pers);
        boolean ok2 = s.prenota(stud);
        boolean ok3 = s.prenota(prof);

        // controlla che sia andato tutto bene
        if (!ok1 || !ok2 || !ok3) System.out.println("Problemi...");

        // stampa l'elenco delle prenotazioni
        s.stampaPrenotazioni();

        // stampa i posti ancora disponibili
        System.out.println("Posti disponibili: " +s.postiDisponibili());
    }
}
```

Gestione di un teatro (9)

Risultato dell'esecuzione:

Spettacolo Macbeth

Del 25/12/2013 - 21.00

Nome: Mario Rossi

Indirizzo: Via Garibaldi 23

Nome: Gianni Bianchi

Indirizzo: Via Gramsci 88

Matricola: 1

Anno: 1

(Studente in corso)

Nome: Prof. Luca Neri

Indirizzo: Via Belli 11

Codice: a11233

Dipartimento: Dip. Matematica

Posti disponibili: 97

Late Binding

Nella stampa dell'elenco delle prenotazioni è stato richiamato il metodo `visualizza()` specifico della (sotto)classe di ogni spettatore

- La classe `Spettacolo` ha chiamato `visualizza()` **convinta** di lavorare con un oggetto di tipo `Persona`
- Durante l'esecuzione (non prima), la chiamata a `visualizza()` viene **collegata** alla (sotto)classe giusta (corrispondente al tipo vero dell'oggetto)
- Questo meccanismo di collegamento "ritardato" della chiamata di un metodo con la classe corrispondente viene detto:

LATE BINDING (oppure **BINDING DINAMICO**)

Sommario

- 1 Ancora sull'ereditarietà
 - Esempio: gestione di un teatro
 - Late binding
- 2 Ereditarietà e controllo dei tipi
- 3 Gerarchie di classi di Java
 - La classe Object e i suoi metodi

Ereditarietà e controllo dei tipi (1)

Abbiamo detto che un oggetto di una sottoclasse può essere usato ovunque sia richiesto un oggetto della superclasse

- Può essere **passato** a un metodo
- Può essere **restituito** da un metodo
- Può essere **assegnato** a una variabile

Quindi possiamo scrivere:

```
Persona p = new Studente("Guido Guidi", "Via Roma 12");
```

La variabile `p` è di tipo `Persona`

- su di essa potremo chiamare **solo** metodi che facciano parte della classe `Persona`
- se tali metodi sono soggetti a overriding in `Studente`, verrà usata la versione di `studente`
- non potranno essere chiamati altri metodi di `Studente` che non siano in `Persona`

Ereditarietà e controllo dei tipi (2)

```
Persona p = new Studente("Guido Guidi", "Via Roma 12");
```

Ogni oggetto in Java ha quindi un **tipo apparente** e un **tipo effettivo**

- Il tipo apparente è quello specificato come tipo della variabile corrispondente (nell'esempio `Persona`)
- Il tipo effettivo è quello con cui si è costruito l'oggetto (nell'esempio `Studente`)

Ereditarietà e controllo dei tipi (3)

Il **compilatore** si basa solo sul **tipo apparente**:

- in quanto il tipo effettivo può variare durante l'esecuzione

```
...
Persona p;
if
  (n==0) p = new Studente("Guido Guidi","Via Roma 12");
else
  p = new Persona("Guido Guidi","Via Roma 12");
int m = p.getMatricola(); //funziona? il compilatore non si fida...
```

- quindi, per sicurezza, il compilatore controlla che p venga usato sempre come un oggetto di tipo Persona

A tempo di esecuzione l'**interprete** (Java Virtual Machine) usa il **tipo effettivo**:

- l'interprete può controllare il tipo vero dell'oggetto
- vengono richiamati i metodi della classe corrispondente

Ereditarietà e controllo dei tipi (4)

In qualunque momento possiamo **forzare** il compilatore a considerare una variabile come se fosse un oggetto di una sottoclasse

- come nel caso dei tipi primitivi, possiamo usare un **type cast**

```
Persona p = new Studente("Guido Guidi", "Via Roma 12");  
int m = ((Studente) p).getMatricola(); // forza il compilatore  
Studente s = (Studente) p; // forza il compilatore
```

- quando si usa un type cast bisogna essere **sicurissimi** che il tipo effettivo sarà compatibile con quello che forziamo...
- ...altrimenti si avrà un errore durante l'esecuzione che **interromperà il programma!**

Il type cast può essere usato solo tra classi che sono in una relazione supertipo-sottotipo

```
Studente s = (Studente) p; // OK, Studente sottotipo di Persona  
Persona p2 = (Persona) s; // OK, Persona supertipo di Studente  
// (comunque poco utile)  
Spettacolo s = (Spettacolo) p; // NO! Spettacolo non in relazione  
// con Persona
```

Ereditarietà e controllo dei tipi (5)

Il tipo effettivo di un oggetto può essere **testato** usando il predicato **instanceof**

```
p instanceof Studente
```

il quale è

- un'espressione **booleana**
- che equivale a true se l'oggetto p ha **tipo effettivo** Studente

```
if (p instanceof Studente) {  
    int m = ((Studente) p).getMatricola();  
    System.out.println("Matricola: " + m);  
}
```

Anche dopo aver testato il tipo effettivo con instanceof bisogna usare il type cast

- grazie al controllo siamo sicuri che il type cast non darà errore!

Esempio: gestione di un teatro

Arricchiamo l'esempio del teatro prevedendo prezzi dei biglietti

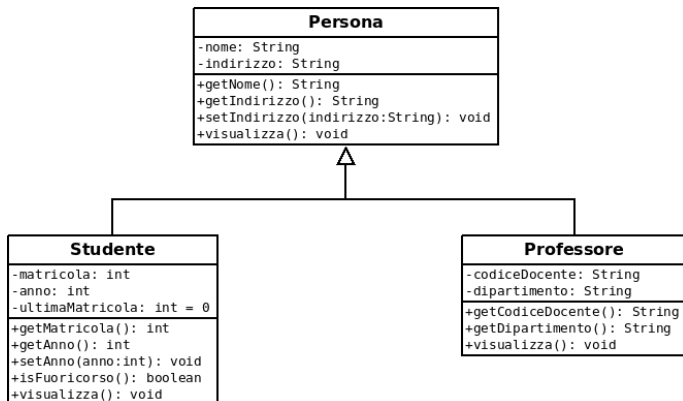
```
public class Spettacolo {  
  
    // nuove variabili statiche con i prezzi dei biglietti  
    private static double prezzoIntero = 10.0;  
    private static double prezzoRidotto = 7.0  
  
    ... // membri visti prima  
  
    // nuovo metodo che calcola il prezzo in base allo spettatore  
    // prezzo ridotto per professori e studenti in corso  
    public double getPrezzo(Persona p) {  
        if (p instanceof Professore) {  
            return prezzoRidotto;  
        } else if (p instanceof Studente) {  
            if (!(Studente) p).isFuoricorso()  
                return prezzoRidotto;  
        } else  
            return prezzoIntero;  
    }  
}
```


Sommario

- 1 Ancora sull'ereditarietà
 - Esempio: gestione di un teatro
 - Late binding
- 2 Ereditarietà e controllo dei tipi
- 3 Gerarchie di classi di Java
 - La classe Object e i suoi metodi

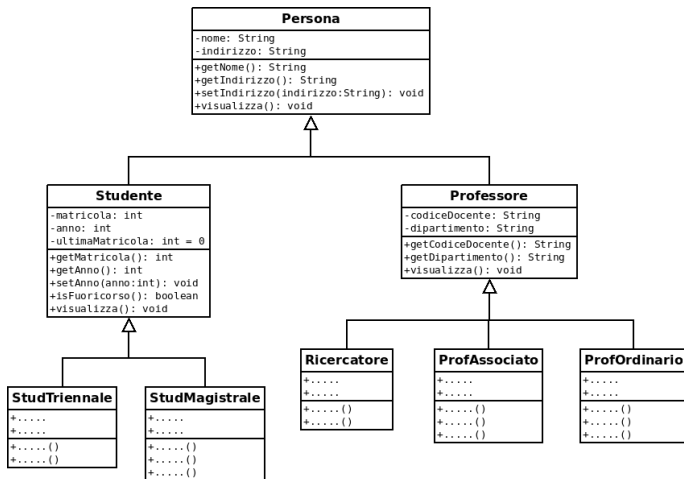
Gerarchie di classi (1)

Abbiamo visto come i meccanismi dell'ereditarietà consentono di specificare relazioni superclasse-sottoclasse



Gerarchie di classi (2)

Usando questo meccanismo ripetutamente, si possono stabilire gerarchie di classi



Gerarchie di classi (3)

L'ereditarietà è **transitiva**

- Esempio: StudMagistrale è una sottoclasse di Persona (per transitività, grazie a Studente)

L'ereditarietà è **singola**

- ogni classe può estendere **una sola** altra classe
- la rappresentazione grafica di una gerarchia prende quindi la forma di un albero rovesciato (vedere grafico precedente)
- ereditarietà cicliche sono proibite (Classe1 extends Classe2 e Classe2 extends Classe1)

La classe Object (1)

Nella Libreria Standard di Java esiste la classe `Object` (package `java.lang`)

Le classi che non estendono nessuna altra classe, **estendono implicitamente** `Object`

- ... ossia

```
public class Prova { .... }
```

corrisponde in realtà a

```
public class Prova extends Object { .... }
```

- tutte le altre (sotto)classi estendono `Object` per transitività

Quindi: **tutte** le classi sono sottoclassi di `Object`

La classe Object (2)

La classe Object fornisce a tutte gli oggetti di tutte le classi alcuni **metodi fondamentali**

Alcuni dei metodi di Object:

- **toString()**: Restituisce una rappresentazione testuale (una stringa) che descrive l'oggetto
 - ▶ utile per stampare
- **equals()**: Consente di confrontare due oggetti
 - ▶ già visto con la classe String

Il metodo toString() (1)

Il metodo toString() restituisce una **rappresentazione testuale** dell'oggetto su cui è invocato.

Dentro a Object, il metodo toString() è definito in modo da restituire una stringa della forma

`<classe>@<hash_code>`

dove <classe> è il nome della classe e <hash_code> è un codice alfanumerico che cambia da oggetto a oggetto.

```
Persona p = new Persona("Mario Bianchi", "Via Firenze 13");
String s = p.toString();
System.out.println(s);    // stampa Persona@7519ca2c
```

Il metodo toString() (2)

Il metodo toString() viene richiamato da System.out.println() quando gli si passano oggetti

```
Persona p = new Persona("Mario Bianchi", "Via Firenze 13");  
System.out.println(p); // stampa Persona@7519ca2c
```

Può quindi essere utile **ridefinire** toString() nelle proprie classi (overriding)

```
public String toString()
```

In questo modo sarà possibile passare i propri oggetti a System.out.println() direttamente

Il metodo toString() (3)

Esempio:

```
public class Rettangolo {
    public int base;
    public int altezza;

    public Rettangolo(int x, int y) {
        base = x; altezza=y;
    }

    public String toString() {
        return "Rettangolo di base " + base + " e altezza " + altezza;
    }
}
```

```
....
Rettangolo r = new Rettangolo(10,20);
System.out.println(r);
...
// stampa "Rettangolo di base 10 e altezza 20"
```

Il metodo equals() (1)

Il metodo `equals()` serve per confrontare oggetti:

- Riceve un oggetto (di tipo `Object`) come parametro
- Restituisce `true` se l'oggetto corrente (`this`) e quello ricevuto sono istanze della stessa classe e sono da considerare equivalenti

L'implementazione di `equals()` in `Object` è basata su `==`, quindi non è affidabile!

Tutte le principali classi della Libreria Standard di Java ridefiniscono `equals()`

- Ad esempio: la classe `String` (... `s1.equals(s2)`...)

Il metodo equals() (2)

Se si vuole consentire di confrontare oggetti delle proprie classi è bene ridefinire equals()

- Attenzione alla firma! Il metodo è così definito

```
public boolean equals(Object o) { ... }
```

- Quindi anche nelle proprie classi bisogna usare un parametro di tipo Object

Il metodo equals() (3)

Esempio:

```
public class Rettangolo {
    public int base;
    public int altezza;

    public Rettangolo(int x, int y) {
        base = x; altezza=y;
    }

    public boolean equals(Object o) {
        // meglio controllare il tipo effettivo!
        if (o instanceof Rettangolo) {
            Rettangolo r = (Rettangolo) o;
            return ((r.base==this.base)&&(r.altezza==this.altezza));
        }
        else return false;
    }
}
```

```
....
Rettangolo r1 = new Rettangolo(10,20);
Rettangolo r2 = new Rettangolo(10,20);
if (r1.equals(r2)) System.out.println("Sono uguali");
...
// stampa "Sono uguali"
```