Lab. Programmazione (CdL Informatica) & Informatica (CdL Matematica) a.a. 2022-23

Monica Nesi

Università degli Studi dell'Aquila

7 & 13 Dicembre 2022

Tipi (insiemi) e sottotipi (sottoinsiemi)

Sia T_A un tipo di dato (insieme di elementi).

Supponiamo che vi sia la necessità di definire un *sottotipo* (sottoinsieme) T_B di T_A e di dover operare sugli elementi di T_B in modo particolare.

Poiché T_B è un sottotipo di T_A, ogni elemento di T_B è anche un elemento di T_A.

Ciò implica che ogni elemento di T_B possiede tutte le caratteristiche di un qualsiasi elemento di T_A e quindi tutte le operazioni definite su T_A possono essere applicate anche sugli elementi di T_B.

Tuttavia, oltre a queste caratteristiche, ogni elemento di T_B possiede ulteriori caratteristiche che lo rendono un elemento del sottoinsieme T_B.

Classi e sottoclassi

Nel linguaggio Java che cosa significa definire una classe che implementa il sottotipo T_B di T_A, dove il tipo T_A è implementato da una classe A?

T_B viene implementato da una classe B che è definita come *classe derivata* o *sottoclasse* o *estensione* della classe A.

Il termine *estensione* si riferisce al fatto che gli elementi della sottoclasse B sono oggetti *più ricchi* con un numero maggiore di qualità/caratteristiche che estendono gli oggetti della classe A. La classe che viene estesa è detta *classe genitrice* o *superclasse*.

Una classe può avere *più* sottoclassi.

Ogni classe ha *una sola* classe genitrice.

Iterando la definizione di sottoclassi si ottiene una *gerarchia di classi*.

Definizione di una sottoclasse

In Java una sottoclasse viene definita tramite la parola riservata `extends`:

```
public class B extends A {
...
}
```

Questo costrutto definisce `B` come sottoclasse o estensione della classe `A`, che è quindi la superclasse di `B`.

Le variabili istanza degli oggetti della sottoclasse sono quelle degli oggetti della superclasse `A` + le *nuove* variabili istanza che rappresentano le ulteriori caratteristiche degli oggetti della sottoclasse `B`.

Nella definizione della sottoclasse si dichiarano *soltanto le nuove variabili istanza*, in quanto le altre vengono *ereditate* dalla superclasse `A`.

Definizione di una sottoclasse: esempio

Sia data la classe `StudUniv` per gli studenti universitari (definita in una lezione precedente).
Uno studente universitario *fuori corso* è uno studente universitario caratterizzato *anche* dall'anno di fuori corso. Scrivere una classe `StudUnivFC` per gli studenti universitari fuori corso.

Gli oggetti della classe `StudUnivFC` hanno le variabili istanza della classe `StudUniv` (che vengono ereditate) + l'anno di fuori corso, ovvero la nuova caratteristica che hanno *solo* le istanze di `StudUnivFC`.

```
public class StudUnivFC extends StudUniv {
 private int annoFC;     // nuova var. ist.
 ...
}
```

Costruttore di una sottoclasse

Il costruttore di una sottoclasse viene definito richiamando il costruttore della superclasse tramite il costrutto `super`, a cui vengono passati gli eventuali parametri che il costruttore della superclasse si aspetta.

Il costruttore della superclasse ha il compito di inizializzare le variabili istanza ereditate, a cui il costruttore della sottoclasse non può accedere, in quanto dichiarate `private`.

Nella sottoclasse StudUnivFC definiamo il costruttore:

```
public StudUnivFC (String nome, String cognome,
int m, int fc) {
   super(nome,cognome,m);
   this.annoFC= fc;
}
```

N.B. La chiamata al costruttore della superclasse con `super` deve essere la *prima istruzione* del corpo del costruttore della sottoclasse.

Metodi

Gli oggetti della sottoclasse *ereditano* i metodi della superclasse, ovvero tali oggetti hanno questi metodi che possono essere invocati sulle istanze della sottoclasse.

I metodi *ereditati* dalla superclasse (se vanno bene come definiti nella superclasse) *non* vengono riscritti.

Occorre invece definire i metodi *nuovi* della sottoclasse che operano sulle *nuove* variabili istanza.

Esempio: Definire nella classe `StudUnivFC` i metodi seguenti:
- un metodo che restituisce l'anno di fuori corso;
- un metodo che incrementa l'anno di fuori corso;
- un metodo che restituisce una stringa con le informazioni relative ad uno studente universitario fuori corso.

Esempio StudUnivFC: metodi

Definiamo quindi i seguenti metodi nella sottoclasse `StudUnivFC`:

```
public int leggiAnnoFC() {
 return this.annoFC;
}

public void aggAnnoFC() {
 this.annoFC++;
}
```

Quindi, finora i metodi degli oggetti della sottoclasse sono questi due metodi + quelli definiti nella superclasse `StudUniv` ed ereditati.

Tuttavia, il metodo `info()` ereditato dalla superclasse, se invocato su un oggetto della sottoclasse `StudUnivFC`, non restituisce una descrizione completa dell'oggetto, in quanto il metodo della superclasse non conosce le nuove variabili istanza.

Sovrascrittura di metodi (Overriding)

Occorre quindi *ridefinire* il metodo `info()` (oppure `toString()`, a seconda di come è stato chiamato nella superclasse).

L'idea è che, quando tale metodo viene invocato su un'istanza della sottoclasse, non venga eseguito il metodo ereditato ma quello ridefinito nella sottoclasse.

Il metodo viene ridefinito utilizzando una forma di polimorfismo detta *overriding* (*sovrascrittura* di metodi).

L'overriding consiste nel definire nella sottoclasse un metodo con la *stessa intestazione* del metodo che compare nella superclasse (in particolare, la stessa *signature* o *firma* del metodo, data da nome + parametri del metodo).

Non solo... Il metodo sovrascritto calcola un risultato che può essere visto come il valore calcolato dal metodo della superclasse + altre informazioni che caratterizzano l'istanza della sottoclasse.

Overriding e riutilizzo del codice

Ne segue che il metodo sovrascritto nella sottoclasse può (meglio, *deve*) essere definito invocando lo stesso metodo presente nella superclasse $\Longrightarrow$ *riutilizzo del codice*.

Nel caso del metodo `info()` (o `toString()`), la stringa restituita dal metodo sovrascritto è la stringa restituita dal metodo nella superclasse + la descrizione delle nuove variabili istanza.

In `StudUnivFC` il metodo `info()` è sovrascritto come segue:

```
public String info() {     //overriding
 return super.info() + "␣" + this.annoFC +
"␣annofuoricorso.";
}
```

dove la chiamata `super.info()` invoca il metodo `info()` della superclasse per calcolare la prima parte della stringa.

La definizione della sottoclasse `StudUnivFC` è completata.

Super

In una sottoclasse definiamo:

- le nuove variabili istanza;

- il costruttore della sottoclasse invocando (come *prima riga di codice*) il costruttore della superclasse tramite `super(...)`;

- i metodi che operano sulle nuove variabili istanza;

- eventuali metodi che sovrascrivono metodi della superclasse tramite *overriding* + *riutilizzo del codice*.

Ricapitolando, la parola riservata `super` è utilizzata in due modi:
- `super(...)` per invocare il costruttore della superclasse;
- `super.m()` per invocare il metodo `m()` della superclasse (per distinguerlo dal metodo `m()` sovrascritto nella sottoclasse).

N.B. Non si scrive `this` davanti a `super.m()`.

Ereditarietà: alcune osservazioni

Il meccanismo dell'ereditarietà permette di risparmiare righe di codice, definendo in una sottoclasse solo le nuove caratteristiche, le nuove operazioni e ridefinendo opportunamente quelle operazioni della superclasse che si vuole far operare diversamente su istanze della sottoclasse.

Ciò viene fatto cercando di riutilizzare codice il più possibile e sfruttando il polimorfismo di Java (i.e., overloading e overriding).

Inoltre l'ereditarietà consente di fare sviluppi ed analisi top-down a partire da una classe generale fino a sottoclassi sempre più estese e con oggetti sempre più ricchi di informazioni.

Assegnamenti tra riferimenti ad oggetti di superclasse/sottoclasse

Date le classi `StudUniv` e `StudUnivFC`, si considerino gli oggetti:

```
StudUniv s1 =
  new StudUniv("Andrea","Rossi",134678);
StudUnivFC s2 =
  new StudUnivFC("Elena","Bianchi",111789,3);
```

Il metodo `info()` è applicabile su entrambi gli oggetti `s1` ed `s2`:

```
System.out.println(s1.info());
System.out.println(s2.info());
```

dando luogo alla stampa delle seguenti stringhe:

```
Rossi Andrea (matr. 134678)
Bianchi Elena (matr. 111789) 3 anno fuori corso.
```

Assegnamenti tra riferimenti ad oggetti di superclasse/sottoclasse (caso 1)

1. *Riferimento di sottoclasse* assegnato a *riferimento di superclasse*

È possibile fare un tale assegnamento? SI, ma...

Se `s2` è assegnato ad `s1` e poi si cerca di applicare su `s1` un metodo definito nella sottoclasse:

```
s1 = s2;
int x = s1.leggiAnnoFC();
s1.aggAnnoFC();
```

si ha errore in fase di compilazione, in quanto tali metodi non appartengono alla classe a cui appartiene `s1`.

Binding Statico vs Dinamico

Cosa succede se si applica su `s1` un metodo presente nella superclasse che è sovrascritto nella sottoclasse, come `info()`?

Situazione: `s1` è dichiarato di tipo `StudUniv` (rif. superclasse), ma sta puntando ad un oggetto di tipo `StudUnivFC` (sottoclasse in cui il metodo `info()` è sovrascritto).

Occorre stabilire quale dei due metodi `info()` viene *legato* all'oggetto (riferito da) `s1`. Due possibili scelte:

- il legame avviene durante la fase di compilazione: si parla di *binding statico* o *early binding* ed il metodo legato ad `s1` può essere solo quello della sua classe `StudUniv`;

- il legame avviene durante l'esecuzione: si parla di *binding dinamico* o *late binding* ed il metodo legato ad `s1` è quello di `StudUnivFC`, ovvero la classe dell'istanza a cui sta puntando `s1`.

Binding Statico vs Dinamico (cont.)

Java adotta il *late binding* dei metodi.

Quindi, quando si invoca `s1.info()`, per il meccanismo del *late binding* viene eseguito il metodo sovrascritto nella sottoclasse, ovvero la classe a cui appartiene l'oggetto riferito da `s1`: `s1`:

```
System.out.println(s1.info());
```

dà luogo alla stampa:

```
Bianchi Elena (matr. 111789) 3 anno fuori corso.
```

Assegnamenti tra riferimenti ad oggetti di superclasse/sottoclasse (caso 2)

2. *Riferimento di superclasse* assegnato a *riferimento di sottoclasse*

È possibile fare un tale assegnamento? NO

Supponiamo di "perdere" il riferimento `s2` all'oggetto della sottoclasse e di voler assegnare `s1` ad un riferimento di sottoclasse:

```
s2 = null;
StudUnivFC s3 = s1;      // ERRORE
```

Si ha errore in fase di compilazione in quanto i tipi di `s1` e di `s3` non sono "compatibili".

Tuttavia, il riferimento di superclasse `s1` riferisce un oggetto della sottoclasse, quindi può essere assegnato al riferimento di sottoclasse `s3` *forzando* il tipo di `s1` tramite l'uso del *cast* `(StudUnivFC)s1`.

Cast ed instanceof

L'assegnamento

```
StudUnivFC s3 = (StudUnivFC)s1;
```

è corretto, in quanto l'espressione a destra è di tipo `StudUnivFC`.

Per evitare errori nel caso in cui `s1` non riferisca effettivamente un oggetto della sottoclasse, è consigliabile usare l'operatore booleano `instanceof`, come segue:

```
StudUnivFC s3;
if (s1 instanceof StudUnivFC)
  s3 = (StudUnivFC) s1;
else
  s3 = null;
System.out.println(s3.info());
```

Nuovamente viene stampata la stringa:

```
Bianchi Elena (matr. 111789) 3 anno fuori corso.
```

Sovrascrivere il metodo equals

Come abbiamo già visto, la classe `Object` contiene i metodi:

- `public boolean equals(Object obj)`
- `public String toString()`

Questi metodi dovrebbero essere ridefiniti in ogni classe in modo da sovrascrivere quelli della superclasse `Object`.

Consideriamo il metodo `equals` e la sua definizione nella classe `StudUniv` come *metodo sovrascritto*, assumendo che due oggetti della classe siano uguali se le variabili istanza corrispondenti sono uguali.

Facendo overriding, l'intestazione del metodo `equals` nella classe `StudUniv` è uguale a quella del metodo `equals` in `Object`:

```
public boolean equals(Object obj)
```

Sovrascrivere il metodo equals (cont.)

La definizione completa è la seguente:

```
public boolean equals(Object obj) {
  if (!(obj instanceof StudUniv))
    return false;
  StudUniv s = (StudUniv)obj;
  return this.nome.equals(s.nome) &&
this.cognome.equals(s.cognome) &&
this.matricola == s.matricola;
}
```

Confrontare questa definizione con quella data nel file contenente la classe `Corso`.

Come si può sovrascrivere il metodo `equals` nella sottoclasse `StudUnivFC` riutilizzando il metodo `equals` della superclasse `StudUniv`?