Azzolini Riccardo 2018-11-13

Gerarchia di classi

1 Gerarchia

In generale, una classe rappresenta una categoria di oggetti caratterizzati dallo stesso tipo di stato e dallo stesso comportamento.

Spesso è possibile individuare più classi con delle caratteristiche in comune: ad esempio, sia `Pasticceria` che `Libreria` potrebbero fornire un metodo `pagamentoCC`. Tale metodo è in realtà proprio di una *categoria più ampia*, `Negozio`, di cui `Pasticceria` e `Libreria` sono *sotto-categorie*. Le proprietà e i comportamenti di una categoria di oggetti vengono infatti *ereditati* dalle sotto-categorie.

Java consente di rappresentare queste relazioni gerarchiche tra classi:

- la classe corrispondente alla categoria più ampia prende il nome di **superclasse**
- le classi che descrivono le sotto-categorie si dicono **sottoclassi**
- le sottoclassi **ereditano** lo stato e il comportamento della superclasse: ad esempio, quando viene invocato un metodo su un'istanza di una sottoclasse, la JVM cerca il codice da eseguire risalendo la gerarchia
- si dice che una sottoclasse **estende** la superclasse perché aggiunge proprietà e comportamenti, oltre a quelli ereditati

2 Organizzazione della gerarchia

La gerarchia delle classi di Java è organizzata ad albero:

- ogni classe estende *al più una* classe (la sua superclasse diretta)
- ogni classe può essere estesa da *più* sottoclassi
- la radice dell'albero è `Object`: tutte le altre classi sono sue sottoclassi

La classe `Object` ha vari metodi (`toString`, `equals`, ecc.), che vengono ereditati da tutte le classi e sono quindi disponibili anche se il programmatore non li definisce esplicitamente.

3 La classe `Rettangolo`

Le sue istanze rappresentano rettangoli.

3.1 Costruttori

```
public Rettangolo(double x, double y);
```

3.2 Metodi

```
public double getArea();
public double getPerimetro();
public double getBase();
public double getAltezza();
public boolean equals(Rettangolo r);
public boolean haAreaMaggiore(Rettangolo r);
public boolean haPerimetroMaggiore(Rettangolo r);
public String toString(); // es. "base = 3.5, altezza = 2.0"
```

4 La classe `Quadrato`

Le sue istanze rappresentano quadrati.

4.1 Costruttori

```
public Quadrato(double x);
```

4.2 Metodi

```
public double getArea();
public double getPerimetro();
public double getLato();
public boolean equals(Quadrato q);
public boolean haAreaMaggiore(Quadrato q);
public boolean haPerimetroMaggiore(Quadrato q);
public String toString(); // es. "lato = 5.1"
```

La classe `Quadrato`:

- eredita i metodi di `Rettangolo`

- ne ridefinisce alcuni (es. `toString`)
- ha dei metodi in più (es. `getLato`)

In pratica, grazie all'ereditarietà, per implementare `Quadrato` è necessario scrivere solo ciò che la differenzia da `Rettangolo`.

5 Gerarchia e tipo

Ogni classe definisce un tipo, i cui valori sono tutte le sue istanze possibili.

Il tipo di un oggetto è effettivamente determinato dall'insieme dei messaggi a cui può rispondere: agli oggetti di tipo `A` è possibile inviare tutti i messaggi specificati nella classe `A`.

In presenza di ereditarietà, le sottoclassi ereditano i metodi della superclasse. Di conseguenza, le istanze della sottoclasse sono in grado di rispondere anche ai messaggi definiti nella superclasse, e possono quindi essere trattate come istanze di quest'ultima. In altre parole, il tipo determinato da una sottoclasse è un **sottotipo** di quello determinato dalla superclasse.

La relazione tra sottoclasse e superclasse è infatti chiamata "è un" ("is-a"): ogni oggetto della sottoclasse *è un* oggetto della superclasse.

6 Promozione di tipi riferimento

Il tipo di una superclasse è più ampio (è un **supertipo**) di quello di una sua sottoclasse. Per questo, i riferimenti a istanze di una sottoclasse possono essere **promossi** implicitamente al tipo della superclasse.

6.1 Esempio

```
Quadrato q = new Quadrato(6);
Rettangolo r = q;
// oppure
Rettangolo r = new Quadrato(6);
```

7 Overriding e polimorfismo

Una sottoclasse può **ridefinire** (o **riscrivere**, **override**) i metodi della superclasse.

Se un metodo è stato ridefinito, e ne esistono quindi più versioni (con la stessa segnatura ma in classi diverse), quella da eseguire è determinata dalla JVM *in fase di esecuzione* in base **tipo dell'oggetto** a cui è rivolta l'invocazione del metodo, e non al tipo del riferimento all'oggetto.

Di conseguenza, una stessa chiamata può *assumere più forme* (**polimorfismo**), cioè invocare metodi diversi, a seconda del tipo di oggetto a cui viene rivolta.

7.1 Fasi di compilazione ed esecuzione

In fase di compilazione, viene verificata l'esistenza del metodo chiamato per il *tipo del riferimento*.

In fase di esecuzione, invece, viene selezionato il metodo da eseguire in base al *tipo effettivo dell'oggetto*. Per fare ciò, la JVM ricerca il metodo partendo dalla classe dell'oggetto e risalendo la gerarchia. È garantito che esso verrà prima o poi trovato, risalendo al massimo fino al tipo del riferimento (per il quale l'esistenza del metodo è stata verificata dal compilatore).

7.2 Esempi

```
Rettangolo r = new Quadrato(6);
out.println(r.toString());
```

Nonostante la variabile `r` sia di tipo `Rettangolo`, viene chiamato il metodo `toString` di `Quadrato` (cioè la versione ridefinita).

```
Object o;
// ...
o.getBase();
```

Questo codice, invece, non viene accettato dal compilatore perché il metodo `getBase` non è definito per il tipo del riferimento, `Object`, quindi non si può garantire che, in fase di esecuzione, il metodo verrà effettivamente trovato, qualunque sia l'oggetto a cui fa riferimento `o`.