Giorno 5
OOP: Ereditarietà

Sommario

Nella prima parte si analizza l'ereditarietà in Java: Si introduce la **classe Object** ed alcuni suoi metodi, si parla del modificatore `protected` e delle classi astratte. Si introduce infine la gerarchia di classi, e nella seconda parte si parla di come si vanno a cercare in modo efficiente i metodi nelle superclassi.

1 Ereditarietà

L'eredità in java funziona per estensione esplicita, tramite `extends`:

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

Ed una classe che non estende altre classi estende di default la classe **Object**.

Nota: Una classe che ne estende un'altra **non ne eredita i membri privati**.

1.1 La classe 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
- ... `https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html`

1.1.1 Confronto con equals()

```
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)
```

1.1.2 Le stringhe (letterali) sono speciali

I letterali, (i.e. stringhe del tipo `"banana"`) sono **oggetti immutabili**: i metodi di String non modificano gli oggetti, ma ne restituiscono sempre dei nuovi come risultato.

1.1.3 Riconoscimento di letterali uguali

Il compilatore, per ottimizzare, riconosce letterali uguali ed ottimizza il codice **costruendo un solo oggetto**.

1.2 Visibilità (protected)

Oltre ai modificatori di accesso/visibilità che abbiamo visto, esiste: `protected`: rende visibili i membri **solo alle alle sotto-classi**, senza renderli del tutto pubblici.

1.3 Classi astratte

Una **classe astratta** è una **classe parzialmente definita**:

```
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 ();
}
```

Le classi astratte sono utilizzate estendendole, ed implementando i metodi astratti, in questo caso `volume` e `superficie`.

1.4 Gerarchia di classi

Il costrutto extends consente di creare una **gerarchia di classi**, rappresentabile come un albero la cui radice è Object. Dato che in Java alle classi sono associati *tipi di oggetto*, la gerarchia di classi è una rappresentazione della relazione di sottotipo, con $Top = Object$.

La regola di subsumption del sistema di tipi ci consente di ottenere un meccanismo di **polimorfismo per sottotipo**.

- Proprietà transitiva:

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

- Subsumption

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

- New

$$\Gamma \vdash_e new\ T() : T$$

- Assegnamento

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

1.4.1 Lo Structural subtyping è più debole del Nominal subtyping

Lo Structural è più debole del Nominal poiché nel nominal vale:

$$S <: T \implies \text{ogni membro pubblico di } T \text{ è membro pubblico di } S$$

La cui dimostrazione segue dalla definizione di extends/implements e dalla transitività di $<:$

2 Overloading, Overriding, Dynamic Dispatch

2.1 Overloading di metodi

Ad ogni metodo è associata una **signature**, che contiene il suo nome ed il tipo dei suoi parametri.

Esempi:

Metodo	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 a, double b)`	`minimo(double, double)`

Nota Si noti la signature **non tiene conto del nome dei parametri**, poiché la signature riassume solo le informazioni che possono essere dedotte dalla chiamata del metodo (quindi non i nomi dei parametri)

Più metodi di una stessa classe possono **avere lo stesso nome**, a patto che sia diversa la loro signature (ossia, nella pratica, il nome dei loro parametri). La tecnica di chiamare più metodi con lo stesso nome prende nome di **overloading**.

Formalmente:

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

Ossia: Se tutti i parametri attuali sono del tipo giusto, la signature è S e la dichiarazione del metodo ha signature S, allora si può assegnare tipo T alla chiamata del metodo.

La procedura di scegliere il metodo con la signature corretta si dice **static dispatch**;

(in generale, con il termine **dispatch** si indica il metodo con cui si collegano le chiamate di funzione/metodo alle loro definizioni)

2.1.1 Invocare un metodo

Quando si invoca un metodo, si controlla (staticamente) se questo è presente nella classe che descrive il *tipo apparente* (tipo statico) dell'oggetto. Se il metodo si trova nella classe che descrive il *tipo effettivo*, non sarà trovato automaticamente, ma si deve eseguire un downcast esplicito (se possibile dopo aver controllato il tipo effettivo con `instanceof`).

Se il metodo non è trovato, c'è un errore a tipo di compilazione.

2.2 Dynamic Dispatch ed Overriding

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

- Il metodo è presente nella classe dell'oggetto (ok!)
- Il metodo va cercato nelle superclassi

Nel secondo caso la ricerca del metodo parte dalla classe corrente (**tipo effettivo**) e risale la gerarchia. Questo prende il nome di **dynamic dispatch**.

2.2.1 Realizzazione efficiente

Ovviamente visitare l'albero ad ogni chiamata di metodo sarebbe molto inefficiente, perciò la JVM adotta una soluzione basata su:

- **Tabelle dei metodi** (**dispatch vector**), una tabella con puntatori al codice dei metodi;
- **Sharing strutturale**, i.e. la tabella (dispatch vector) di una sottoclasse **riprende la struttura** della tabella della superclasse **aggiungendo righe** per i nuovi metodi (aka **l'ordine dei metodi in comune è lo stesso**) – (n.b. le due tabelle sono distinte, è shared solo l'ordine!)

Queste due tecniche, permettono di accedere all'unica tabella che descrive le sottoclassi per indice, ed una chiamata di un metodo sarà tradotta nel bytecode con una chiamata `invokevirtual #i`, dove i è l'indice nella tabella.

2.2.2 Overriding

Che succede se "risalendo la gerarchia" si trovano due metodi con la stessa firma? **Si sceglie il primo incontrato**, aka quello definito più di recente. In termini di **dispatch vector**, le tabelle hanno la stessa struttura, ma i puntatori possono essere diversi a causa dell'overriding.

2.2.3 Ultime cose

- **Come si accede alle variabili d'istanza?** (intuitivamente, poi in realtà è più complicato) il compilatore aggiunge `this` come parametro implicito al metodo:

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

  Diventa

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

  ("a la Python")

- **Chiamata ai metodi statici:**

  I metodi statici hanno un **dispatch vector separato**, acceduto tramite `invokestatic` nel bytecode.

  $\Longrightarrow$ sostanzialmente **non sono metodi**