Programmazione Orientata agli Oggetti in Linguaggio Java

Ereditarietà e Polimorfismo: Polimorfismo - d

Regole Sintattiche e Semantiche

versione 1.2

G. Mecca – Università della Basilicata – mecca@unibas.it

Sommario

- Regole per la Scrittura del Codice
  - ⇨Metodi Finali
  - ⇨Metodi Statici
  - ⇨Metodi Privati
  - ⇨Proprietà
  - ⇨Riferimenti e Tipi
  - ⇨Cast

Regole per la Scrittura del Codice

- A questo punto
  - ⇨possiamo riassumere alcune regole per la scrittura del codice
- In particolare
  - ⇨come impedire la riscrittura di un metodo
  - ⇨regole sui metodi statici e privati
  - ⇨regole sulle proprietà
  - ⇨regole sui riferimenti e i tipi

Metodi Finali

- Annotazione n. 1
  - ⇨per le ragioni di prestazioni esposte, nei linguaggi esiste un modo per impedire il late binding di un metodo
  - ⇨tipicamente si impedisce che il metodo possa essere sovrascritto

Metodi Finali

- In Java
  - ⇨questo si ottiene utilizzando la parola chiave final
- Esempio
  - ⇨public final String toString() {...}
  - ⇨in questo caso, il compilatore impedisce che il metodo sia ridefinito lungo l'eventuale gerarchia

Metodi Finali

- I vari usi di final
  - ⇨vicino ad una proprietà: la rende costante
  - ⇨vicino ad un metodo: ne impedisce l'overriding
  - ⇨vicino ad una classe: impedisce di estendere la classe; misura ancora più drastica, anche in questo caso l'overriding è impedito

Metodi Finali

- Terminologia
  - ⇨metodo virtuale: metodo soggetto a ridefinizione >> binding dinamico
  - ⇨metodo finale: metodo non soggetto a ridefinizione >> binding statico
- Attenzione
  - ⇨un metodo può anche essere parzialmente virtuale; in questo caso il binding è complesso; per semplicità: binding dinamico

Metodi Finali

- Esempio
  - ⇨java.lang.Object: toString() -> virtuale
  - ⇨miopackage.Prova: toString() -> virtuale, sovrascrive toString() di Object
  - ⇨miopackage.Prova2 extends Prova: public final toString() -> finale, sovrascrive toString() di Prova ma non consente altre sovrascritture
  - ⇨miopackage.Prova3 extends Prova2: non è consentito sovrascrivere toString()

Metodi Statici

- Annotazione n. 2
  - ⇨i meccanismi riguardano esclusivamente i metodi di oggetto
  - ⇨i metodi statici vengono ereditati
  - ⇨ma non possono essere oggetto di overriding
- In altri termini
  - ⇨ai metodi statici viene sempre applicato il binding statico

Metodi Statici

```
public class Superclasse {
  public void stampa() {
     System.out.println("Superclasse");
  }
}

public class Sottoclasse
                extends Superclasse {
  public void stampa() {
     System.out.println("Sottoclasse");
  }
}
// in Principale
public static void main(String[] args) {
   Superclasse s = new Sottoclasse();
   s.stampa(); // "Sottoclasse"
}
```

```
public class Superclasse {
  public static void stampa() {
     System.out.println("Superclasse");
  }
}

public class Sottoclasse
                extends Superclasse {
  public static void stampa() {
     System.out.println("Sottoclasse");
  }
}
// in Principale
public static void main(String[] args) {
   Superclasse.stampa(); // Superclasse
}
```

Metodi Statici

- Di conseguenza
  - ⇨i componenti di tipo classe non sono polimorfi
- In effetti
  - ⇨è la ragione principale per cui programmare con gli oggetti è preferibile a programmare con le classi
  - ⇨l'altra è la visibilità globale

Metodi Privati

- Annotazione n. 3
  - ⇨i metodi privati non possono essere sovrascritti
  - ⇨in effetti non vengono nemmeno ereditati
- Nota
  - ⇨non è possibile cambiare il livello di visibilità di un metodo da private a public (errore sintattico)

Metodi Privati

- In particolare
  - supponiamo che ci siano due metodi privati con lo stesso prototipo nella superclasse e nella sottoclasse
  - in questo caso la macchina virtuale utilizzerebbe il binding statico per le chiamate del metodo
  - ovvero sarebbe sempre l'oggetto di turno e non l'oggetto puntato ad eseguire il metodo

Metodi Privati

- Un esempio di binding statico

```
public class Superclasse {

public void esegui() {
   this.stampa();
}

private void stampa() {
   System.out.println("Superclasse");
}
}
```

```
public class Sottoclasse
        extends Superclasse {

private void stampa() {
   System.out.println("Sottoclasse");
}

}
```

```
// in Principale
public static void main(String[] args) {
   Superclasse s = new Sottoclasse();
   s.esegui();
}
```

risultato:
Superclasse

NOTA: e se i due metodi fossero pubblici ???

Proprietà

- Annotazione n. 4
  - ⇨il meccanismo dell'overriding e del binding dinamico si applica esclusivamente ai metodi
  - ⇨non si applica alle proprietà
  - ⇨se una sottoclasse ridefinisce una proprietà della superclasse tra le due proprietà non c'è nessun legame
  - ⇨il binding dei messaggi che riguardano la proprietà è statico

Proprietà

- Un esempio di ridefinizione delle proprietà

```
public class Superclasse {

protected int a = 10;

public void stampa() {
   System.out.println("Superclasse");
}

}
```

```
public class Sottoclasse
            extends Superclasse {
protected double a = 0.5;

public void stampa() {
   System.out.println("Sottoclasse");
}

}
```

```
// in Principale
public static void main(String[] args) {
   Superclasse s = new Sottoclasse();
   s.stampa();
   System.out.println(s.a);
}
```

risultato:
Sottoclasse
10

Proprietà

- Quindi
  - ⇨tutti i messaggi relativi alla proprietà vengono sempre eseguiti dall'oggetto di turno
- L'effetto della ridefinizione
  - ⇨la proprietà a di Sottoclasse "nasconde" la proprietà a ereditata da Superclasse
  - ⇨this.a si riferisce alla proprietà double
  - ⇨ma la proprietà del padre è comunque visibile attraverso super.a

Proprietà

- Accesso alla proprietà nascosta

```
public class Superclasse {

protected int a = 10;

public void stampa() {
   System.out.println("Superclasse");
}

}
```

```
public class Sottoclasse
        extends Superclasse {
protected double a = 0.5;

public void stampa() {
   System.out.println("Sottoclasse "
            + this.a + " "
            + super.a);
}

}
```

```
// in Principale
public static void main(String[] args) {
   Superclasse s = new Sottoclasse();
   s.stampa();
}
```

risultato:
Sottoclasse 0.5 10

Proprietà

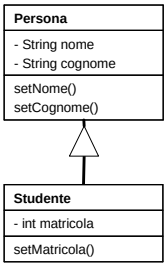
- Il significato di “nascondere”
  - la proprietà della superclasse non appare più come una proprietà ereditata
  - accessibile attraverso this
  - ma non è realmente nascosta, dal momento che resta accessibile attraverso super
- In altri termini
  - l’oggetto ha due “versioni” diverse della proprietà a disposizione

Riferimenti e Tipi

- Annotazione n. 5
  - attenzione all’uso dei riferimenti
  - abbiamo detto che un riferimento è tipato
  - e richiede un oggetto del tipo (interfaccia) corrispondente
  - questa regola è stringente
- In particolare
  - attraverso il riferimento è possibile chiamare solo metodi dell’interfaccia prevista

Riferimenti e Tipi

- Esempio
  - ⇨Persona e Studente




```
Studente s = new Studente();
Persona p = (Persona)s;

p.setNome("Mario"); // OK: il metodo appartiene al tipo Persona
p.setMatricola(12345); // ERRORE SINTATTICO: il metodo
                       // non appartiene al tipo Persona
```


Riferimenti e Tipi

- Ritorniamo alla stampa degli impegni
  - ⇨una versione alternativa
  - ⇨perfettamente funzionante

```
public static void stampa (ArrayList lista) {
   for (int i = 0; i < lista.size(); i++) {
      Object impegno = lista.get(i);
      System.out.println(impegno.toString());
   }
}
```

per poter chiamare toString()
il cast a Impegno non è necessario

Riferimenti e Tipi

- Viceversa
  - ⇨consideriamo il metodo che salva gli impegni su file, utilizzando saveString() di impegno

```
public static void salva (ArrayList lista, PrintWriter file) {
    for (int i = 0; i < lista.size(); i++) {
        Impegno impegno = (Impegno)lista.get(i);
        file.println(impegno.saveString());
    }
}
```

per poter chiamare saveString() il cast a Impegno è indispensabile: il metodo non fa parte dell'interfaccia di Object

Cast

ATTENZIONE
a upcast e downcast

- Annotazione n. 6
  - ⇨attenzione ai cast applicati agli oggetti
- Infatti
  - ⇨esistono due tipi di cast
- Upcast
  - ⇨cast verso un tipo più in alto nella gerarchia
- Downcast
  - ⇨cast verso un tipo più in basso

Cast

- Esempi
  - Riunione r = new Riunione();
  - Object o = (Object)r;
    // upcast da Riunione a Object
  - Impegno i = (Impegno)o;
    // downcast da Object a Impegno
  - Animale a = (Animale)new Volpe();
    // upcast da Volpe ad Animale
  - Volpe v = (Volpe)a;
    // downcast da Animale a Volpe

Cast

- Una differenza significativa
  - la correttezza degli upcast è verificabile staticamente (basta guardare la gerarchia)
  - la correttezza dei downcast è verificabile solo a tempo di esecuzione
  - possono sollevare ClassCastException
- Esempio:
  - Object o = new Object();
  - Volpe v = (Volpe)o; // ClassCastException

Cast

- Per questa ragione
  - ⇨Java rispetta la sua natura “fortemente tipata”
- In particolare
  - ⇨gli upcast sono eseguiti automaticamente senza bisogno di specificarli esplicitamente
  - ⇨i downcast (possibile fonte di errori) devono essere specificati esplicitamente dal programmatore

Cast

- Quindi, è possibile scrivere
  - ⇨Riunione r = new Riunione();
  - ⇨Object o = r;
  - ⇨Impegno i = (Impegno)o;
  - ⇨Animale a = new Volpe();
  - ⇨Volpe v = a;
    // ERRORE SINTATTICO: incompatible types

Riassumendo

- Regole per la Scrittura del Codice
  - ⇨Metodi Finali
  - ⇨Metodi Statici
  - ⇨Metodi Privati
  - ⇨Proprietà
  - ⇨Riferimenti e Tipi
  - ⇨Cast

Termini della Licenza