Java

polimorfismo

G. Prencipe
prencipe@di.unipi.it

Introduzione

- È un altro degli ingredienti fondamentali della OOP
- Permette di organizzare il codice e la leggibilità e di ottenere programmi estensibili
- Viene detto anche *dynamic binding*
- Partiamo da un esempio

Esempio

```
public class Note {
  private String noteName;
  private Note(String noteName) {
    this.noteName = noteName;
  }
  public String toString() { return noteName; }
  public static final Note // Costanti
    MIDDLE_C = new Note("Middle C"),
    C_SHARP  = new Note("C Sharp"),
    B_FLAT   = new Note("B Flat");
} // Non si possono creare oggetti addizionali perché il
   costruttore è private
```

Esempio

```
public class Instrument {
  public void play(Note n) {
    System.out.println("Instrument.play() " + n);
  }
} ///:~
```

Esempio

```
// Oggetti Wind sono Instruments
// perché hanno la stessa interfaqccia
public class Wind extends Instrument {
  // Ridefinisce l'interfaccia
  public void play(Note n) {
    System.out.println("Wind.play() " + n);
  }
} ///:~
```

Esempio

```
public class Music {
   public static void tune(Instrument i) {
    i.play(Note.MIDDLE_C);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    tune(flute); // Upcasting: il metodo tune accetta un
   // Instrument, ma noi gli passiamo un Wind
   }
} ///:~
```

Dimenticare il tipo

- Nell'esempio visto, sembra quasi che ci dimentichiamo del tipo di un oggetto
- Non sarebbe più "normale" se semplicemente **tune()** accettasse direttamente un riferimento a un **Wind** (piuttosto che a un **Instrument**)?
- Domanda: cosa accadrebbe? Perché non vogliamo fare così?

Dimenticare il tipo

- Nell'esempio visto, sembra quasi che ci dimentichiamo del tipo di un oggetto
- Non sarebbe più "normale" se semplicemente **tune()** accettasse direttamente un riferimento a un **Wind** (piuttosto che a un **Instrument**)?
- Domanda: cosa accadrebbe?
  - Semplice: bisognerebbe scrivere un metodo **tune()** per ogni tipo di **Instrument** nel sistema!!

Esempio

```
class Stringed extends Instrument {
  public void play(Note n) {
    System.out.println("Stringed.play() " + n);
  }
}
class Brass extends Instrument {
  public void play(Note n) {
    System.out.println("Brass.play() " + n);
  }
}
```

Esempio

```
public class Music2 {
  public static void tune(Wind i) {
    i.play(Note.MIDDLE_C);
  }
  public static void tune(Stringed i) {
    i.play(Note.MIDDLE_C);
  }
  public static void tune(Brass i) {
    i.play(Note.MIDDLE_C);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    Stringed violin = new Stringed();
    Brass frenchHorn = new Brass();
    tune(flute); // No upcasting
    tune(violin);
    tune(frenchHorn);
  }
} ///:~
```

Polimorfismo

- Chiaramente funziona, ma
  - Bisogna scrivere metodi specifici per ogni **Intrument** aggiunto
  - Inoltre il compilatore non ci avverte se dimentichiamo di aggiungere un metodo per una delle sottoclassi!!
- Con il *polimorfismo* possiamo scrivere un unico metodo che prende la superclasse come argomento, e funziona per ognuna delle sottoclassi!!

Binding

- Il processo di collegare una chiamata a un metodo al corpo del metodo è detto *binding*
- Quando il *binding* è effettuato prima che il programma venga eseguito, viene detto *early binding*
- Con il polimorfismo, il compilatore non può sapere su quale **Instrument** verrà invocato il metodo **tune()**
- In questo caso si ha *late binding*
  - Il *binding* avviene a run-time
- *Tutti* i metodi in Java usano *late binding*, a meno che un metodo non sia dichiarato **final**

Classi e metodi astratti

- Una classe *astratta* è una classe che viene definita solo per stabilire una *interfaccia comune* per tutte le sue sottoclassi
- Non viene fornita l'implementazione completa per quella classe
  - Si definisce solo l'interfaccia o parte dell'implementazione

Classi e metodi astratti

- Quindi non ha senso creare oggetti di una classe astratta
  - E infatti non si possono creare oggetti di una classe astratta, altrimenti il compilatore si lamenta
- All'interno della classe astratta è possibile definire metodi senza darne una implementazione
  - Essi sono detti *metodi astratti*

Classi e metodi astratti

- La loro sintassi è

  abstract void f()
- Se una classe contiene uno o più metodi astratti, anch'essa deve essere qualificata come **abstract**

Classi e metodi astratti

- Se si scrive una sottoclasse della classe astratta e se ne vogliono creare oggetti, si devono fornire implementazioni dei metodi astratti della superclasse
- Altrimenti, anche la sottoclasse diviene astratta
  - In questo caso il compilatore forza l'inserimento della parola chiave **abstract**

Classi e metodi astratti

- È possibile creare una classe astratta senza inserire metodi astratti
  - È utile se vogliamo comunque evitare la creazione di oggetti di quella classe
- Vediamo un esempio

Esempio

```
abstract class Instrument {
  private int i;
  public abstract void play(Note n);
  public String what() {
        return "Instrument";
  }
  public abstract void adjust();
}
```

Esempio

```
class Wind extends Instrument {
  public void play(Note n) {
  System.out.println("Wind.play() " + n);
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}
class Percussion extends Instrument {
  public void play(Note n) {
    System.out.println("Percussion.play() " + n);
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}
```

Esempio

```
public class Music4 {
  static void tune(Instrument i) {
    i.play(Note.MIDDLE_C);
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
    tune(e[i]);
  }
public static void main(String[] args) {
    // Upcasting durante la creazione dell'array
    Instrument[] orchestra = {new Wind(), new Percussion(), new Stringed(),
                              new Brass(), new Woodwind()
    };
    tuneAll(orchestra);
  }}
```

Classi astratte

- Si nota che comunque le classi astratte, essendo sottoclassi di **Object**, ereditano i tutti i suoi metodi
- Quindi ogni classe astratta fruisce sicuramente di un certo numero di metodi non astratti

Costruttori e polimorfismo

- Analizziamo l'esempio **Sandwich.java** e il suo output
- L'ordine con cui vengono invocati i costruttori è il seguente
  1. Il costruttore della superclasse è invocato. Questo passo è ripetuto ricorsivamente in modo che prima il costruttore della radice della gerarchia è invocato e poi gli altri (a discendere)
  2. I membri della classe sono inizializzati secondo l'ordine della dichiarazione
  3. Il corpo del costruttore della sottoclasse è invocato

Costruttori e polimorfismo

- L'ordine con cui sono invocati i costruttori è importante
- Infatti, al momento della creazione di un oggetto di una sottoclasse, si deve assumere che tutti i membri della superclasse devono essere correttamente inizializzati
  - L'unico modo per garantire questo è di invocare prima il costruttore della superclasse
  - Anche per questo motivo, è buona pratica inizializzare i membri oggetto quando sono dichiarati (come per **b**, **c**, e **i** nell'esempio **Sandwich.java**)

Costruttori e polimorfismo

- Cosa accade se invochiamo all'interno di un costruttore un metodo?
  - Sappiamo che in Java abbiamo *dynamic binding*
- Non sempre si hanno i risultati voluti, ed è facile introdurre comportamenti non controllati (derivati appunto dal binding dinamico)
- Vediamo un esempio: **PolyConstructors.java**

Esempio

- Domanda: cosa notiamo di strano nel risultato?
- In particolare nel risultato della prima invocazione di **draw()**?

Esempio

- Domanda: cosa notiamo di strano nel risultato?
- In particolare nel risultato della prima invocazione di **draw()**?
- Il valore di **radius** è 0 (nemmeno 1, che è il valore di inizializzazione in **RoundGlyph**)
- Perché?
- Rivediamo l'ordine di inizializzazione per i costruttori….

Costruttori e polimorfismo

1. La memoria allocata per l'oggetto viene inizializzata a zero (binario) prima che qualsiasi altra cosa accada
2. I costruttori per la superclasse sono invocati, come detto in precedenza
   1. A questo punto il metodo *overridden* **draw()** viene invocato (prima che il costruttore per **RoundGlyph** sia chiamato!!)-->**radius** è 0
3. I membri sono inizializzati nell'ordine della dichiarazione
4. Il corpo del costruttore della sottoclasse è eseguito

Costruttori e polimorfismo

- Quindi, una buona pratica nella scrittura dei costruttori è di fare il *minimo indispensabile* per portare l'oggetto in uno stato *buono*
- Se possibile, evitare di invocare metodi
  - Gli unici metodi sicuri che si possono invocare sono quelli **final** (o **private**, che sono automaticamente **final**) della superclasse
  - Su di essi, infatti, non è possibile fare *overriding*

Regola di design

- Non è sempre facile capire se è meglio utilizzare composizione o ereditarietà per il riutilizzo del codice
- Una regola di massima è la seguente:
  - *Utilizzare ereditarietà per esprimere differenze di comportamento, e membri (quindi composizione) per esprimere variazioni di stato*
- Un esempio può chiarire: **Transmogrify.java**

Ereditarietà ed estensione

- Quando creiamo una sottoclasse di una data superclasse, l'interfaccia viene ereditata
- Se non aggiungiamo nuovi metodi alla sottoclasse, l'interfaccia della sottoclasse è *esattamente la stessa* di quella della superclasse
- In questo caso si parla di *ereditarietà pura*

Ereditarietà ed estensione

- Con l'ereditarietà pura, tra superclasse è sottoclasse si stabilisce una relazione del tipo *"è-un"*
  - Esempio: **Cerchio** è una **FormaGeom**
- Se invece nella sottoclasse introduciamo nuovi metodi, allora stiamo estendendo l'interfaccia
  - Si stabilisce una relazione del tipo *"è-come"*
  - Infatti, la sottoclasse ha tutti i metodi disponibili per la superclasse, ma ha qualcosa in più

Ereditarietà ed estensione

- Con l'ereditarietà pura non ci sono problemi, visto che il polimorfismo si occupa di tutto
  - È possibile invocare metodi indifferentemente su oggetti della superclasse o della sottoclasse
  - Si ha *upcasting*
- Con l'estensione, i metodi aggiunti alla sottoclasse *non possono essere invocati* su oggetti della superclasse

Ereditarietà ed estensione

- Nel caso di estensione, il problema è che se facciamo *upcast* su un oggetto della sottoclasse non possiamo più invocare su di esso i metodi estesi (della sottoclasse)
- Ma noi sappiamo che quell'oggetto è un oggetto della sottoclasse
- Quindi, come facciamo a riscoprire il tipo esatto di quell'oggetto?
  - Con il *downcast*

Ereditarietà ed estensione

- In Java ogni *cast* è controllato
- Quindi, se i tipi (conseguenti a un cast) non tornano, otteniamo un errore **ClassCastException** a run-time
- Il controllo dei tipi a run-time è detto *run-time type identification (RTTI)*
- Vediamo un esempio: **RTTI.java**

Esercizi

1. Creare una gerarchia di classi **Roditori**: **Topo**, **Castori**, **Coniglio**, ecc….
   1. Definire la superclasse **abstract**. Nella superclasse fornire metodi (**abstract** dove possibile) comuni a tutti i **Roditori**
   2. Fare *override* di questi metodi nelle sottoclassi con comportamenti diversi a seconda del tipo di **Roditore**
   3. Creare un array di **Roditori**, e invocare i metodi della superclasse e controllare cosa accade

Esercizi

2. Creare una classe astratta **TestAstratto** con nessun metodo
   1. Creare una sottoclasse **SottoTA** con un metodo **metodoUno()**
   2. Creare un metodo **statico** **metodoDue()** che prende un riferimento alla superclasse, fa il *downcast* alla sottoclasse, e chiama **metodoUno()**
   3. Nel **main** dimostrare che il tutto funziona
   4. Inserire una dichiarazione astratta del **metodoUno()** nella superclasse per eliminare la necessità del downcast

Java

polimorfismo

fine