Programmazione Java
Polimorfismo

Davide Di Ruscio
Dipartimento di Informatica
Università degli Studi dell'Aquila

diruscio@di.univaq.it

Sommario

» Polimorfismo
- Upcasting
- Selezione dei metodi
- Comportamento dei metodi polimorfi dentro i costruttori
- Downcasting

Polimorfismo

- I linguaggi procedurali (esempio Pascal) sono basati sull'idea che procedure e funzioni, e i loro operandi, hanno un unico tipo
- Tali linguaggi sono detti monomorphic, cioè ogni valore e variabile può avere uno ed un solo tipo
- Linguaggi O.O. sono detti polymorphic, cioè i valori e le variabili possono avere più di un tipo
- Dal greco polymorphos "avere molte forme"

Polimorfismo > upcasting

- Un oggetto può essere utilizzato come oggetto del proprio tipo o come oggetto del suo tipo base
- Prendere un riferimento a un oggetto e trattarlo come un riferimento al suo tipo base si dice upcasting

Polimorfismo > upcasting


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

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

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 `Music.tune()` accetta un riferimento `Instrument`, ma anche qualunque tipo di dati derivato da Instrumento

(Vedere music.Music.java)

Polimorfismo > upcasting


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

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

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
    }
}
```

Per esempio un riferimento `Wind` viene passato a `tune()` senza la necessità di cast

Polimorfismo > upcasting

» Senza il meccanismo dell'upcasting è necessario scrivere un metodo `tune()` per ogni tipo di `Instrument`

» Per esempio, in casso di aggiunta delle classi `Stringed` e `Brass` sottoclassi di `Instrument`

```
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);         // No upcasting
     tune(frenchHorn);      // No upcasting
   }
  }
```

Polimorfismo > selezione dei metodi (1)

- Il collegamento tra una chiamata di un metodo ed un blocco di codice viene detto *binding*
- Quando si effettua il binding prima dell'esecuzione del programma si parla di *early binding*
- Quando il collegamento avviene in fase di esecuzione, si parla di *late binding*
- In Java il binding è sempre late a meno che un metodo non sia stato dichiarato static o final (nota che i metodi private sono implicitamente final)

Polimorfismo > selezione dei metodi (2)

» Come detto, un oggetto ha due tipi che possono essere diversi

- Vero tipo (statico)
- Tipo dinamico (run time type)

```
Automobile a = new Fiat();
```

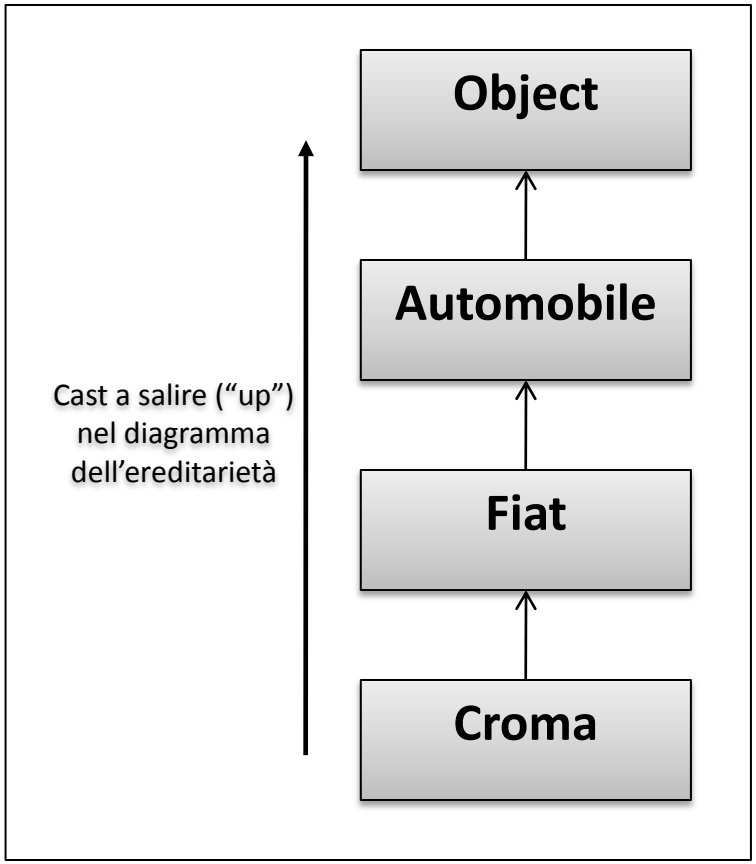
Il tipo statico di `a` è `Automobile`

Il tipo dinamico di `a` è `Fiat`

```
Int method (Automobile a){…};
```

Il tipo statico di `a` è `Automobile`

Il tipo dinamico di `a` è sconosciuto



Polimorfismo > selezione dei metodi (3)

» La selezione del tipo (statico o dinamico) usato per scegliere il metodo da eseguire segue le seguenti regole:

- in caso di *overloading*, viene scelto il tipo statico
- in caso di *overriding*, viene scelto il tipo dinamico

» In particolare, data la seguente invocazione:

```
object1.method(object2)
```

i seguenti passi vengono eseguiti:

1. Trova il *tipo statico* di object1
2. Nella class che definisce il *tipo statico* di object1, trova il metodo più specifico che accetta il *tipo statico* di object2.
3. Nell'esecuzione del metodo selezionato al passo 2, il corpo eseguito sarà quello del *tipo dinamico* di object1

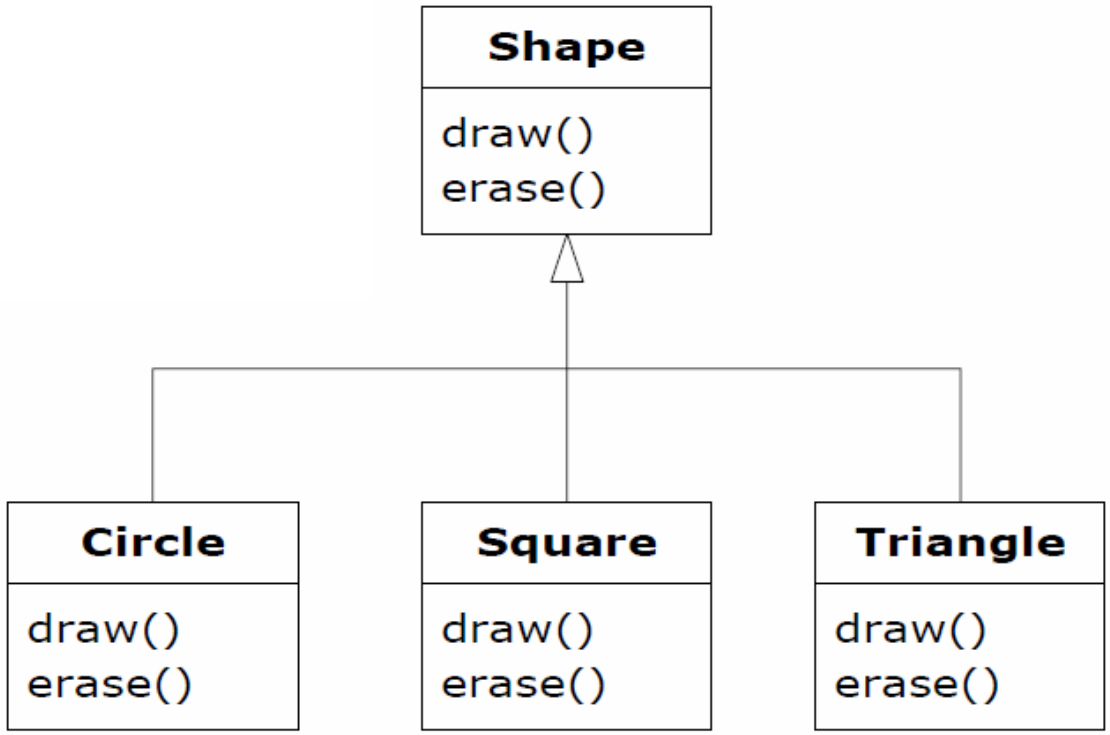
Polimorfismo > selezione dei metodi (4)

- Due errori (`NoSuchMethod` e `Ambiguous`) possono essere rilevati a tempo di compilazione nella scelta del passo 2
- Nessun errore può essere rilevato a tempo di esecuzione

(Vedere MethodSelection*.java)

Polimorfismo > selezione dei metodi (5)

» Esempio:



(Vedere Shapes.java)

Comportamento dei metodi polimorfi dentro i costruttori (1)

- Nella lezione 5 è stata presentata la gerarchia delle chiamate dei costruttori
- Che cosa accade dentro un costruttore se viene chiamato un metodo dell'oggetto in costruzione soggetto al dynamic binding ?

Comportamento dei metodi polimorfi dentro i costruttori (2)


```
abstract class Glyph {
    abstract void draw();
    Glyph() {
      System.out.println("Glyph() before draw()");
      draw();
      System.out.println("Glyph() after draw()");
    }
  }

class RoundGlyph extends Glyph {
    private int radius = 1;
    RoundGlyph(int r) {
      radius = r;
      System.out.println("RoundGlyph.RoundGlyph(), radius = " + radius);
    }
    void draw() {
     System.out.println("RoundGlyph.draw(), radius = " + radius);
    }
  }



public class PolyConstructors {
    public static void main(String[] args) {
      new RoundGlyph(5);
    }
  }
```

Il metodo `draw()` è abstract quindi è destinato ad essere ridefinito

(Vedere PolyConstructors.java)

Comportamento dei metodi polimorfi dentro i costruttori (2)


```
abstract class Glyph {
    abstract void draw();
    Glyph() {
      System.out.println("Glyph() before draw()");
      draw();
      System.out.println("Glyph() after draw()");
    }
  }

class RoundGlyph extends Glyph {
    private int radius = 1;
    RoundGlyph(int r) {
      radius = r;
      System.out.println("RoundGlyph.RoundGlyph(), radius
    }
    void draw() {
     System.out.println("RoundGlyph.draw(), radius = " + r
    }
  }



public class PolyConstructors {
    public static void main(String[] args) {
      new RoundGlyph(5);
    }
  }
```

- La creazione di un oggetto di tipo `RoundGlyph` prevede l'invocazione del costruttore di `Glyph` che a sua volta invoca `draw()` che però e' abstract
- In questo caso viene invocato il metodo `draw()` ridefinito in `RoundGlyph` (prima che venga chiamato il costruttore di `RoundGlyph`)

Comportamento dei metodi polimorfi dentro i costruttori (2)


```
abstract class Glyph {
    abstract void draw();
    Glyph() {
      System.out.println("Glyph() before draw()");
      draw();
      System.out.println("Glyph() after draw()");
    }
  }

class RoundGlyph extends Glyph {
    private int radius = 1;
    RoundGlyph(int r) {
      radius = r;
      System.out.println("RoundGlyph.RoundGlyph(), radius = " + radius);
    }
    void draw() {
     System.out.println("RoundGlyph.draw(), radius = " + radius);
    }
  }
```

```
public class PolyConstructors {
    public static void main(String[] args) {
      new RoundGlyph(5);
    }
  }
```

OUTPUT

```
Glyph() before draw()
RoundGlyph.draw(), radius = 0 // radius ha il
                              // valore
                              // dell'inizializzazione
                              // di default
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5
```

Downcasting

- Il *downcast* si può utilizzare per recuperare le informazioni sul tipo specifico di un oggetto
- Il downcast non è sicuro come l'upcast (in cui la classe base non può avere un'interfaccia più grande di quella della classe derivata, e quindi tutti i messaggi inviati attraverso l'interfaccia della classe base saranno sicuramente accettati)
- In Java avviene un controllo a run-time per garantire che l'oggetto in questione sia in effetti del tipo che si pensa. In caso contrario viene sollevata una ClassCastException (vedremo in seguito)
- Il controllo dinamico dei tipi in fase di esecuzione è chiamato *run-time type identification* (RTTI)

Downcasting > esempio


```
class Useful {
  public void f() {}
  public void g() {}
}

class MoreUseful extends Useful {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}

public class RTTI {
  public static void main(String[] args) {

    Useful a = new Useful();
    Useful b = new MoreUseful();

    a.f();
    b.g();
    // Errore a tempo di compilazione:  method not found in Useful
    //! b.u();
    ((MoreUseful)b).u(); // Downcast/RTTI - Non si ha errore perchè di fatto b ha il metodo u()
    ((MoreUseful)a).u(); // Exception thrown, a non ha il metodo u()
  }
}
```