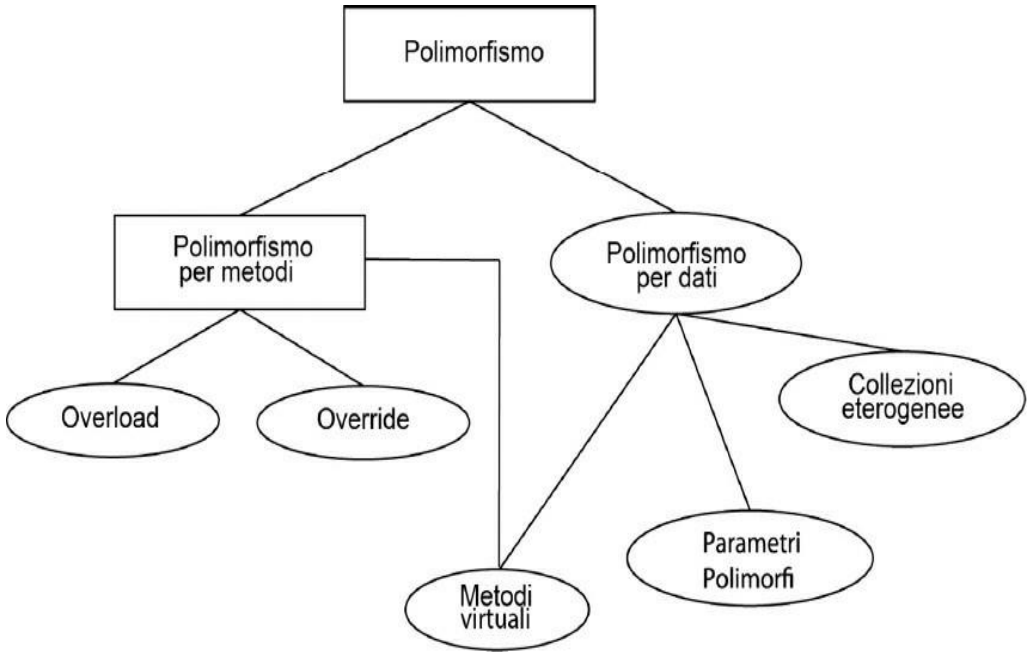
CORSO JAVA AESYS
Parte Prima

2022

www.aesystech.it

Intro polimorfismo

Il polimorfismo (dal greco “molte forme”) è un altro concetto che dalla realtà è stato importato nella programmazione ad oggetti. Esso permette di riferirci con un unico termine a entità diverse. Ad esempio, sia un telefono fisso sia uno smartphone consentono di telefonare, dato che entrambi i mezzi sono definibili come telefoni. Telefonare, quindi, può essere considerata un’azione polimorfica (ha diverse implementazioni). Il polimorfismo in Java è argomento complesso, che si dirama in vari sottoargomenti. Utilizzando una convenzione con la quale rappresenteremo mediante rettangoli i concetti, e con ovali i concetti che hanno una reale implementazione in Java, cercheremo di schematizzare il polimorfismo e le sue espressioni.



Override 1/2

L'override (che potremmo tradurre con "riscrittura") è il termine object oriented che viene utilizzato per descrivere la caratteristica che hanno le sottoclassi di ridefinire un metodo ereditato da una superclasse. Ovviamente non esisterà override senza ereditarietà. Una sottoclasse è sempre più specifica della classe che estende, e quindi potrebbe ereditare metodi che hanno bisogno di essere ridefiniti per funzionare correttamente nel nuovo contesto. Ad esempio, supponiamo che una ridefinizione della classe Punto (che per convenzione assumiamo bidimensionale) dichiari un metodo distanzaDallOrigine() che calcola la distanza tra un punto di determinate coordinate e l'origine degli assi cartesiani. Questo metodo se ereditato all'interno di un'eventuale sottoclasse PuntoTridimensionale, ha bisogno di essere ridefinito per calcolare la distanza voluta, tenendo conto anche della terza coordinata. Vediamo quanto appena descritto sotto forma di codice:

```
public class Punto {    private int x, y;

   public void setX(int x) {       this.x = x;
   }
   public int getX() {       return x;
   }
   public void setY(int y) {       this.y = y;
   }
   public int getY() {       return y;
   }
   public double distanzaDallOrigine() {       int tmp = (x*x) +
(y*y);       return Math.sqrt(tmp);
   } }

public class PuntoTridimensionale extends Punto {    private int z;

   public void setZ(int z) {       this.z = z;
   }
   public int getZ() {       return z;
   }

   public double distanzaDallOrigine() {
      int tmp = (getX()*getX()) + (getY()*getY())
         + (z*z); // N.B.: x e y non sono ereditate
```

Override 2/2

L’override sembra semplice quindi da implementare, ma non è tutto così scontato:
ci sono regole da rispettare:

1. il metodo riscritto nella sottoclasse deve avere la stessa firma (nome eparametri) del metodo della superclasse;
2. il tipo di ritorno del metodo della sottoclasse deve coincidere con quello del metodo che si sta riscrivendo, o deve essere di un tipo che estende il tipo di ritorno del metodo della superclasse;
3. il metodo ridefinito nella sottoclasse non deve essere meno accessibiledel metodo originale della superclasse.

Per quanto riguarda la prima regola, dando per scontato che il nome del metodo della sottoclasse deve essere identico a quello della superclasse, se parliamo di firme differenti, parliamo di lista di parametri differenti. In casi come questo saremmo di fronte ad un overload in luogo di un override. Infatti non essendo riscritto con la stessa firma, erediteremmo nella sottoclasse anche il metodo della superclasse, e ci sarebbero due metodi con lo stesso nome e differente lista di parametri.

AESYS
colors your business

Collezioni eterogenee ed operatore instanceof

Una collezione eterogenea è una collezione composta da oggetti diversi (ad esempio un array di Object che in realtà immagazzina altri tipi di oggetti). Anche la possibilità di sfruttare collezioni eterogenee è garantita dal polimorfismo per dati. Infatti un array dichiarato di Object potrebbe contenere ogni tipo di oggetto:

```
Object arr[] = new Object[3];
arr[0] = new Punto(); //arr[0], arr[1], arr[2] sono arr[1] = "Hello World!"; //reference ad Object che
puntano arr[2] = new Date(); //ad oggetti istanziati da sottoclassi
```

Il che equivale a scrivere:

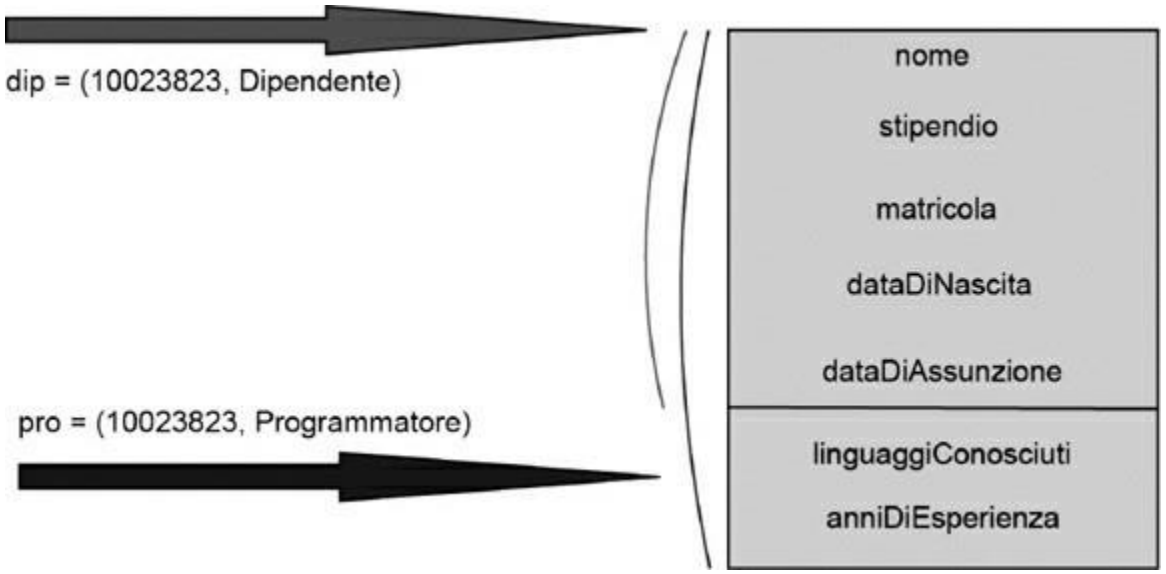
```
Object arr[] = {new Punto(), "Hello World!", new Date()};
```

Tra gli operatori di Java esiste un operatore binario che può essere utile in questi contesti: instanceof. Mediante questo costrutto si può testare a che tipo di oggetto punta un reference.
In particolare l'operatore instanceof ha come operandi un reference (il primo operando) e una classe (il secondo operando). Questo operatore restituisce true se il primo operando è un reference che punta ad un oggetto istanziato dal secondo operando o ad un oggetto istanziato da una sottoclasse specificata dal secondo operando. Se non si verifica una di queste due condizioni ritorna false.

AESYS
colors your business

Casting di oggetti

Nella figura possiamo notare come i due reference abbiano lo stesso valore numerico (indirizzo) ma differente intervallo di puntamento. Da questo dipende l'accessibilità all'oggetto.
Nel caso tentassimo di assegnare al reference *pro* l'indirizzo di *dip* senza l'utilizzo del casting, otterremmo un errore in compilazione ed un relativo messaggio che ci richiede un casting esplicito. Ancora una volta il comportamento del compilatore conferma la robustezza del linguaggio.



Il compilatore non può stabilire se ad un certo indirizzo risiede un determinato oggetto piuttosto che un altro. È solo in esecuzione che la Java Virtual Machine può sfruttare l'operatore instanceof per risolvere il dubbio.
Il casting di oggetti non si deve considerare come strumento di programmazione standard, piuttosto come un utile mezzo per risolvere problemi progettuali. Una progettazione ideale farebbe a meno del casting di oggetti. Per quanto ci riguarda, all'interno di un progetto, la necessità del casting ci porta a pensare ad una forzatura e quindi ad un eventuale aggiornamento della progettazione. Tuttavia ci sono casi in cui è inevitabile l'utilizzo.

AESYS
colors your business

Invocazione virtuale dei metodi

Un'invocazione ad un metodo m può definirsi virtuale quando m è definito in una classe A, ridefinito in una sottoclasse B (override) e invocato su un'istanza di B, tramite un reference di A (polimorfismo per dati). Quando s'invoca in maniera virtuale il metodo m, il compilatore "pensa" di invocare il metodo m della classe A (virtualmente). In realtà viene invocato il metodo ridefinito nella classe B. Un esempio classico è quello del metodo toString() della classe Object. Abbiamo già accennato al fatto che esso viene sottoposto a override in molte classi della libreria standard. Consideriamo la classe Date del package java.util.
In essa il metodo toString() è riscritto in modo tale da restituire informazioni sull'oggetto Date (giorno, mese, anno, ora, minuti, secondi, giorno della settimana, ora legale...). Consideriamo il seguente frammento di codice:

```
Object obj = new Date(); String s1 = obj.toString();
```

Il reference s1 conterrà la stringa che contiene informazioni riguardo l'oggetto Date, unico oggetto istanziato. Possiamo vederne la schematizzazione in figura 8.5. Il reference obj può accedere solamente all'interfaccia pubblica della classe Object e quindi anche al metodo toString(). Il reference punta però a un'area di memoria dove risiede un oggetto della classe Date, nella quale il metodo toString() ha una diversa implementazione rispetto a quella che fornisce la classe Object.

Polimorfismo e interfacce

Il polimorfismo per dati funziona anche con le interfacce. Questo significa che possiamo usare reference di interfacce per puntare ad oggetti che implementano queste interfacce. Nel capitolo precedente abbiamo asserito che le interfacce possono essere definite per astrarre comportamenti e aggettivi, per poi essere implementati in classi concrete. Precedentemente avevamo definito l'interfaccia Volante in questo modo:

public interface Volante { void atterra(); void decolla();

Ogni classe che deve astrarre un concetto di oggetto volante (come un aereo, un elicottero o un uccello) deve implementare l'interfaccia. Quindi avevamo riscritto la classe Aereo nel seguente modo:

```
public class Aereo extends Veicolo implements Volante {
  @Override
  public void atterra() {
    // override del metodo di Volante
  }  @Override
  public void decolla() {
    // override del metodo di Volante
  }  @Override
  public void accelera() {
    // override del metodo di Veicolo
  }  @Override
  public void decelera() {
    // override del metodo di Veicolo
  }
```

Potremmo quindi creare parametri polimorfi per sfruttare l'interfaccia Volante:

```
public class TorreDiControllo {  public void
autorizzaAtterraggio(Volante v) {
    v.atterra();  }

  public void autorizzaDecollo(Volante v) {
    v.decolla();
  }
}
```