CORSO JAVA AESYS
Parte Prima

2022

www.aesystech.it

Ereditarietà ed interfacce

seconda parte

- Generalizzazione e specializzazione
- Il modificatore abstract
- Interfacce
- Interfacce funzionali

Generalizzazione e specializzazione 1/2

Sono due termini che definiscono i processi che portano all'implementazione dell'ereditarietà.
Si parla di generalizzazione se, a partire da un certo numero di classi, si definisce una superclasse che ne raccoglie le caratteristiche comuni. Viceversa si parla di specializzazione quando partendo da una classe, si definiscono una o più sottoclassi allo scopo di ottenere oggetti più specializzati.
L'utilità della specializzazione è semplice da evidenziare: supponiamo di voler creare una classe MioBottone, le cui istanze siano visualizzate come bottoni da utilizzare su un'interfaccia grafica. Partire da zero creando pixel per pixel è molto complicato. Se invece estendiamo la classe Button del package java.awt, dobbiamo solo aggiungere il codice che personalizzerà il MioBottone.

Abbiamo notato come il test "is a", fallendo, ci sconsigli l'implementazione dell'ereditarietà. Eppure queste due classi hanno campi in comune e non sembra che sia un evento casuale. In effetti sia il Triangolo sia il Trapezio sono ("is a") entrambi poligoni. La soluzione a questo problema è naturale. Basta generalizzare le due astrazioni in una classe Poligono, che potrebbe essere estesa dalle classi Triangolo e Rettangolo. Per esempio:

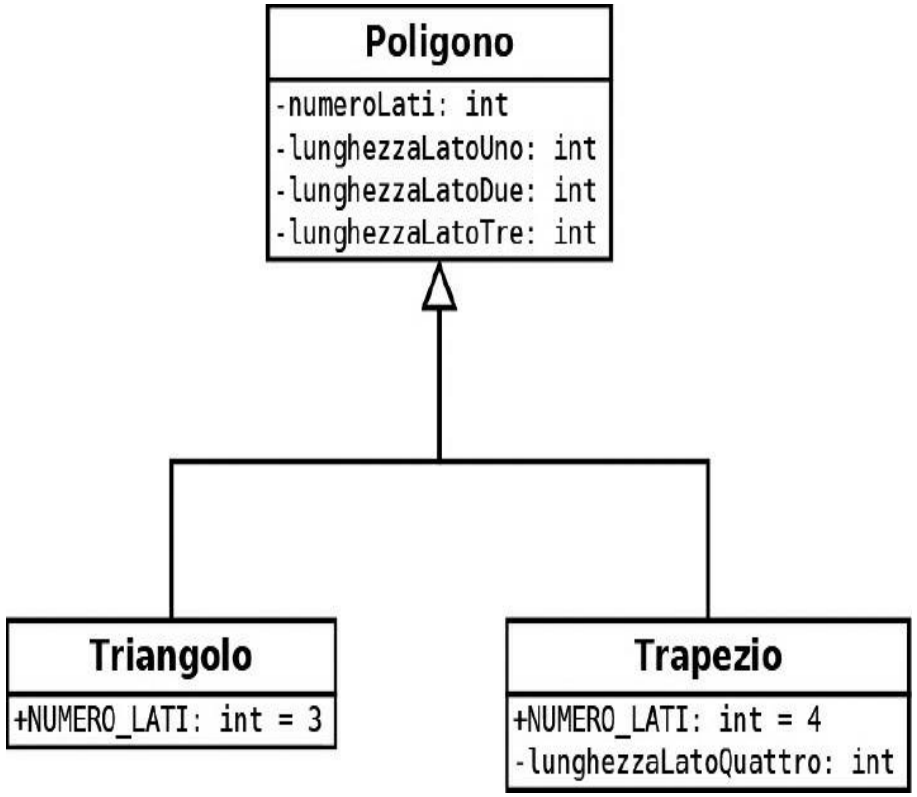
```
public class Poligono {     public int numeroLati;
public float lunghezzaLatoUno;     public float
lunghezzaLatoDue;     public float
lunghezzaLatoTre;
   //...
}

public class Triangolo extends Poligono {     public final int
NUMERO_LATI = 3;     public Triangolo() {}
   //...
}

public class Trapezio extends Poligono {     public final int
NUMERO_LATI = 4;     public float lunghezzaLatoQuattro;
   //... }
```

Generalizzazione e specializzazione 2/2

Se fossimo partiti dalla classe Poligono per poi definire le due sottoclassi, avremmo parlato invece di specializzazione.
Usando il processo di specializzazione è facile pensare anche ad altre eventuali sottoclassi come Esagono, Pentagono, Ottagono, Quadrato e così via. La classe Poligono quindi è la più generica, e in un eventuale programma che usa poligoni, l'unica classe che probabilmente non istanzieremo sarà proprio la classe
Poligono. In questi casi la classe Poligono va dichiarata abstract, e questo implicherà che non potrà essere istanziata (otteremo un errore in compilazione se ci provassimo). Quindi Poligono rimarrebbe importante solo per permetterci di sfruttare i vantaggi dell'ereditarietà, ma non esisteranno oggetti istanziati direttamente da questa classe. Tuttavia, per quanto detto, oggetti istanziati dalle sue sottoclassi Trapezio e Triangolo potranno essere considerati tutti poligoni!



Il modificatore abstract 1/2

Il modificatore abstract può essere applicato non solo a classi ma anche a metodi. Non ha senso invece parlare di variabili astratte.

Un metodo astratto non implementa un proprio blocco di codice e quindi è privo di comportamento. La definizione di un metodo astratto non possiede parentesi graffe ma termina con un punto e virgola. Un esempio di metodo astratto è il seguente:

```
public abstract void dipingiQuadro();
```

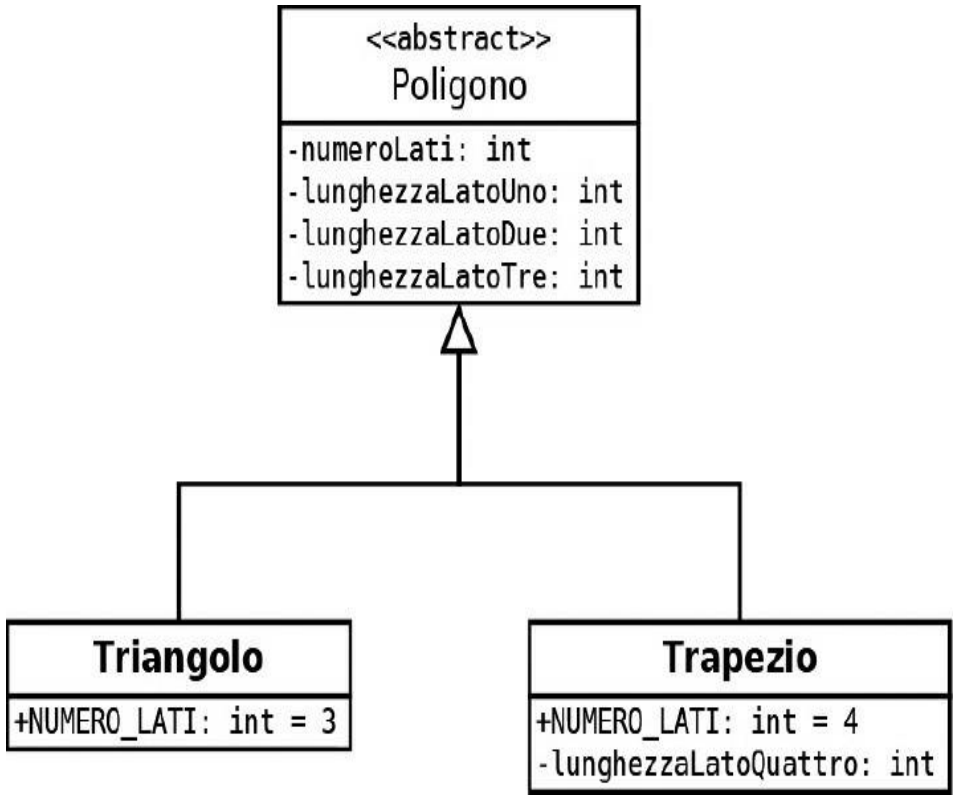
Questo metodo non potrà essere invocato (in quanto non è definito) ma potrà essere soggetto a riscrittura (override) in una sottoclasse come vedremo approfonditamente più avanti quando parleremo del polimorfismo.
Un metodo astratto potrà essere definito solamente all'interno di una classe astratta. In altre parole, una classe che contiene anche un solo metodo astratto deve essere dichiarata astratta.

Classi astratte

Una classe dichiarata astratta non può essere istanziata. Il programmatore che ha intenzione di marcare una classe con il modificatore abstract deve essere consapevole a priori che da quella classe non saranno istanziabili oggetti. (segue slide successiva)

Il modificatore abstract 2/2

Una classe dichiarata astratta non può essere istanziata. Il programmatore che ha intenzione di marcare una classe con il modificatore abstract deve essere consapevole a priori che da quella classe non saranno istanziabili oggetti. Ma perché creare una classe che non si può istanziare? Perché a volte c'è bisogno di creare classi generiche da estendere, e che rappresentano un concetto troppo astratto per essere istanziato nel contesto di un determinato programma. La classe Poligono della slide precedente per esempio, è un'ottima candidata a diventare una classe astratta. Infatti, in un ipotetico utilizzo delle classi che abbiamo definito, non istanzieremo mai oggetti dalla classe Poligono, ma solo dalle classi concrete Triangolo e Trapezio. Il modificatore abstract, è anche un importante strumento di progettazione. Infatti dichiarando una classe abstract dichiareremo anche che questa deve essere estesa, e quindi indirizzeremo lo sviluppo verso la creazione di sottoclassi della classe astratta.



Interfacce

Dal punto di vista della progettazione, un'interfaccia è un'evoluzione del concetto di classe astratta. Dal punto di vista del codice, assomiglia invece ad una classe senza la sua implementazione interna. Un'interfaccia vuole rappresentare infatti quella che con l'incapsulamento abbiamo chiamato proprio interfaccia pubblica, ovvero quella parte dell'oggetto visibile all'esterno, che nasconde la sua implementazione interna.
Un'interfaccia non si può istanziare (non è una classe). Per essere utilizzata, ha bisogno di essere in qualche modo estesa, ma solo un'altra interfaccia può estendere un'interfaccia, e quindi è possibile creare gerarchie costituite da sole interfacce. Dato però che senza istanziare oggetti i programmi non funzionano, abbiamo bisogno che le classi ereditino dalle interfacce. Ecco che allora viene in aiuto la parola chiave implements, che si usa in maniera molto simile ad extends. La differenza tra implementare un'interfaccia ed estendere una classe, consiste essenzialmente nel fatto che, mentre possiamo estendere una sola classe alla volta, possiamo invece implementare un numero indefinito di interfacce, simulando di fatto l'ereditarietà multipla (che verrà affrontata in dettaglio nei prossimi paragrafi).
Un'interfaccia, essendo un'evoluzione di una classe astratta, non può essere istanziata.
Riassumendo, una classe può implementare un'interfaccia e un'interfaccia può estendere un'altra interfaccia.

Interfacce da Java 8

Dalla versione 8 di Java in poi, è possibile definire all'interno delle interfacce anche metodi statici.
Oggi quindi è possibile scrivere interfacce nel seguente modo:

```
public interface StaticMethodInterface {     static void metodoStatico() {
        System.out.println("Metodo Statico Chiamato!");
    }
```

E chiamare metodi statici direttamente dalle interfacce:

```
public class TestStaticMethodInterface {     public static void main(String args[]) {
StaticMethodInterface.metodoStatico();
    }
}
```

Usando le interfacce contenenti metodi statici ne tradiamo la natura contrattuale per la quale fu definita. Un'interfaccia come la precedente non ha bisogno di essere implementata, infatti i metodi statici di un'interfaccia non vengono ereditati. Il comando implements quindi non fa ereditare i metodi statici. Interfacce di questo tipo servono semplicemente per definire metodi statici e pubblici, in pratica definiscono funzioni. N.B.: Dalla versione 9 di Java è possibile anche creare metodi statici privati.

Metodi di default

Altra novità di Java 8 è la possibilità di dichiarare metodi concreti all'interno delle interfacce. Si parla di metodi di default (in inglese default methods) perché vengono dichiarati usando come modificatore la parola chiave default.

Interfacce funzionali

Il nome “interfaccia” in un certo senso ha perso il significato originario, anche se è sempre possibile usare le interfacce dichiarando solo metodi astratti. Con l’introduzione dei metodi privati infatti, le interfacce ormai sono quasi tecnicamente equivalenti alle classi astratte, a parte il fatto che non possono dichiarare variabili d’istanza. Anche se le interfacce possono oramai implementare metodi astratti e concreti come le classi astratte. Un’interfaccia dovrebbe rappresentare un comportamento che più classi potrebbero assumere, e un comportamento non si può istanziare. Una classe astratta dovrebbe invece rappresentare una generalizzazione troppo astratta per essere istanziata. Inoltre bisogna aver presente che una classe può estendere una sola classe astratta, mentre può implementare tante interfacce.
Per definizione prendono il nome di interfacce funzionali (in inglese functional interfaces) le interfacce che contengono un unico metodo astratto. Per indicare quest’ultimo viene spesso usato l’acronimo SAM, che sta per Single Abstract Method (in italiano singolo metodo astratto). Vengono chiamate così proprio perché è come se esistessero solo per dichiarare una funzione da implementare.
Le interfacce funzionali sono un ottimo esempio di come le interfacce dovrebbero essere utilizzate. Esiste anche un nuovo package che dichiara una serie di interfacce funzionali: il package java.lang.function.

Interfacce vs classi astratte

Le interfacce dovrebbero essere create per rappresentare comportamenti. Non possono dichiarare variabili d'istanza ma solo costanti statiche e pubbliche. Possono dichiarare metodi astratti, metodi di default, metodi statici. Dalla versione 9 di Java possono anche dichiarare metodi ausiliari privati, sia statici che non. I metodi privati non possono essere dichiarati come metodi di default.
È possibile implementare una sorta di ereditarietà multipla in Java, implementando più interfacce che hanno metodi di default. È possibile incappare nel famoso problema dell'ereditarietà a diamante, che si presenta quando viene ereditato lo stesso metodo da due interfacce diverse. Le regole in tali casi si possono riassumere nel fatto che il metodo ereditato si deve sempre riscrivere tranne che in due casi. Nel caso le due interfacce da cui viene ereditato il metodo sono una superclasse dell'altra, allora viene preso in considerazione direttamente il metodo della sottoclasse, perché più specifico. Un altro caso dove non c'è bisogno di riscrivere il metodo, è quando ereditiamo lo stesso metodo da una classe e da un'interfaccia: "vince sempre il metodo della classe".
La differenza essenziale tra le classi astratte e le interfacce risiede nel fatto che le classi astratte dovrebbero rappresentare concetti troppo generici per essere istanziati. Mentre le interfacce dovrebbero rappresentare comportamenti