Polimorfismo in Java

Esaminiamo i tre tipi di polimorfismo partendo dal fatto che un'espressione che può assumere valori diversi in base al tipo cui viene applicata.

Il termine polimorfismo, dal greco *πολυμορφοσ* (polymorfos, "avere molte forme"), nell'ambito dei linguaggi di programmazione si riferisce in generale alla possibilità data ad una determinata espressione di assumere valori diversi in relazione ai tipi di dato a cui viene applicata.

Secondo questa generica definizione, è dunque polimorfa anche l'espressione:

```
a + b
```

che, in base al tipo di `a` e `b`, potrebbe rappresentare un valore di tipo `int`, `float` (o numerico in generale, operando in ogni caso l'opportuna operazione di somma aritmetica) oppure addirittura di tipo `String` (ed in tal caso operare il concatenamento delle String `a` e `b`).

Tipi di polimorfismo

Tecnicamente parlando in un linguaggio non tipato o dinamicamente tipato (e.g. il Javascript) ogni espressione è polimorfa, mentre in un linguaggio staticamente tipato come Java, la questione è assai più articolata; nel contesto della programmazione OO e del linguaggio Java si possono infatti distinguere almeno 3 tipi di polimorfismo:

- ah hoc polymorphism
- inclusion polymorphism
- parametric polymorphism (o generic programming)

In questa guida affronteremo i primi due tipi. Il terzo è legato ad una feature di Java detta **generics** (simili ai templates del C++) che affronteremo più avanti nel corso della guida.

Per la verità, fuori dal mondo 'Object Oriented' (o quantomeno fuori da quello che potremmo definire "OO classico", la scuola che ha generato linguaggi come Java e C++, per intendersi), sconfinando nel territorio dei linguaggi fuzionali, il parametric polymorphysm diventa quello predominante al quale si aggiungono altre varianti di "cambiamento di forma" più sofisticate.

Ad hoc polymorphism, overload dei metodi

Detta più comunemente "**method overloading**" (oppure "*operator overloding*" nel caso che sia applicato ad operatori come nell'esempio `a + b`) questa forma di polimorfismo è nota ed utilizzata sin dagli anni '60 e consiste concretamente nella possibilità di "ridefinire" un medesimo metodo per set di parametri diversi.

Prendiamo per esempio la situazione in cui abbiamo la classe Triangolo e dobbiamo implementare il metodo *"area"*. Dovremo utilizzare una delle formule per il calcolo dell'area, ad esempio:

```
Area = base * altezza / 2
```

per applicare la quale è necessario conoscere `base` e `altezza`. Oppure la formula:

$$Area = \frac{1}{2} det \begin{pmatrix} x1 & y1 & 1 \\ x2 & y2 & 1 \\ x3 & y3 & 1 \end{pmatrix}$$

che invece prevede la conoscenza delle coordinate `(xi, yi), {i=1,2,3}` dei vertici del triangolo.

Entrambe le formule potrebbero essere utili ed entrambe avrebbero soprattutto il diritto di avere implementazione in un metodo che si chiama `area`: *ad hoc polymorphism* ci permette proprio di ridefinire il metodo area per set di parametri diversi:

```
public double area(double base, double altezza) {
        return base * altezza * 0.5;
}
public double area(double x1, double y1,
                   double x2, double y2,
                   double x3, double y3) {
        return 0.5 * (x1*y2 + y1*x3 + x2*y3 - x3*y2 - y3*x1 - x2*y1);
}
```

Poiché i 2 metodi hanno parametri diversi (2 double il primo, 6 double il secondo) Java è in grado di capire, durante la compilazione (a "compile-time"), quale è il metodo che intendiamo utilizzare e decidere quindi quale invocare.

Se il linguaggio non permettesse l'overloading (es. il C) potremmo comunque implementare i 2 metodi ma saremmo costretti a utilizzare per loro due nomi diversi (per esempio `areaDataBaseEdAltezza` e `areaDatiIVertici`) e ottenere il medesimo risultato.

Ridefinire i metodi ereditati (override o "Inclusion polymorphism")

A differenza del polimorfismo 'ad hoc' questa forma di polimorfismo è strettamente legata al concetto di ereditarietà e di sub-typing e consiste nella possibilità che una sottoclasse *A* di una data classe *B* ridefinisca uno dei metodi della super-classe e che quindi quando verrà utilizzata una istanza della classe *A* le invocazioni al metodo ridefinito (spesso detto **overridden**) eseguiranno il codice definito nella sotto-classe.

Ricordiamo che le istanze della classe *A* dovranno poter essere utilizzate in ogni espressione che preveda l'utilizzo di una istanza della classe *B* e a queste espressioni si applica dunque la definizione di espressione polimorfa data all'inizio di questa pagina.

In Java per default tutti i metodi non `private` sono ridefinibili ma è possibile specificare la keyword **final** per istruire il compilatore a non ammetterne la ridefinizione.

```
public class B {
        public int metodo(int i, int j) {
                return i+j;
    }
        public final int metodoFinal(int i, int j) {
                return i+j;
        }
}
```

```
public class A extends B {
	public int metodo(int i, int j) {
		return i-j;
	}
	// questa ridefinizione non è ammessa in quanto il metodo è final
	// Infatti se de-commentiamo queste righe il compilatore
	// segnala un errore
	/*
	public int metodoFinal(int i, int j) {
		return i-j;
	}
	*/
}
```

Se osserviamo invece il seguente main program:

```
public static void main(String[] args) {
	B b = new B();
	A a = new A();
	B aa = new A(); // A è sottoclasse di B quindi è
			// assegnabile a variabili di tipo B
	int c;
	c = b.metodo(5, 7);
	System.out.println(c);
	c = a.metodo(5, 7);
	System.out.println(c);
	c = aa.metodo(5, 7);
	System.out.println(c);
}
```

che fornisce come output

```
12
-2
-2
```

è possibile convincersi che la sovrascrittura dei metodi nel subtyping è operazione dinamica (diversamente dall'overloading che avviene sempre a compile time):

- La prima riga dell'output ci dice che è stato chiamato il metodo `metodo` così come implementato nella classe `B` e sia la variabile che l'istanza sono di tipo B.
- Il secondo risultato indica che è stata utilizzata l'implementazione data nella classe *A*, poiché sia la variabile che l'istanza sono di tipo A.
- Il fatto che l'ultima riga di output sia `-1` implica invece che nonostante la variabile `aa` sia di tipo `B`, l'oggetto ritornato dal costruttore `A()`, benchè assegnabile a una variabile di tipo `B`, ha dentro di sé riferimenti ai metodi come definiti per la classe A.

Tale associazione tra istanze e metodi è di solito implementata attraverso una tavola di "puntatori" a metodi che viene associata a run-time ad ogni istanza generata durante l'esecuzione e che servono per risolvere, nella maniera opportuna, le chiamate ai metodi degli oggetti in un programma.

Covariant return types

In object Oriented programming, un "covariant return type" è un metodo il cui valore di ritorno può essere sostituito con un sotto tipo quando il metodo è sovrascritto in una sottoclasse.

I **covariant return types** sono stati (parzialmente) introdotti in Java dalla versione 1.5 (quindi l'esempio sotto non compila in versioni precedenti):

```
public class D {
        public B dammiUnB() {
                // ...
        }
}
public class E extends D {
   public A dammiUnB() {
     // ritorna un'istanza di A
   }
}
```

Qui si vede che la classe `E`, derivata dalla classe `D` non solo ridefinisce (override) il metodo `dammiUnB` ma ne cambia anche il valore di ritorno specificando che non sarà una generica istanza di tipo `B` ad essere ritornata ma una istanza di tipo `A` che è una sottoclasse di `B`.