Programmazione a Oggetti

Polimorfismo
e
Tipi Generici

Polimorfismo

Abilità di una funzione di eseguire il suo task ***su argomenti di tipi diversi***

Dipende di solito, dal fatto che la funzione ***non ha necessità di entrare nella struttura "fine"*** *di un tipo*

Esempi

La funzione **`length`** *che calcola la lunghezza di una lista* ***non dipende dal tipo degli elementi della lista***

Un ***algoritmo di sort*** *dipende solo dal fatto che gli elementi abbiano un tipo per cui è* ***definita una relazione d' ordine***

Vantaggi del Polimorfismo

Astrazione

L'algoritmo è scritto nella sua forma più generale

Riuso del Codice

E' possibile usare la stessa funzione in più casi. Di conseguenza:

Eleganza e sinteticità

Polimorfismo in Java

Esistono sostanzialmente due forme:

polimorfismo indotto dal subtyping: *un metodo con parametro* **`S`** *accetta argomenti di ogni tipo* **`T`**, *a patto che* **`T<:S`**

uso di Tipi Generici

Esempio: Insiemi

La classe **`IntSet`** *non* ***subisce grandi cambiamenti se consideriamo insiemi di oggetti qualsiasi****: è sufficiente usare opportunamente il tipo* **`Object`**

```
public class Set{
    private Vector els;
public void insert(Object x)
//MODIFICA:this
//POST: inserisce x in this
//this_post = this + {x}
```

Drawback

Ogni volta che si recupera un elemento (ad esempio con **`choose()`***) è* ***necessario fare un downcast*** *da* **`Object`** *al tipo atteso.*

Poco safe*: non c'è alcun controllo su cosa inserisco nell'insieme. Di solito gli insiemi sono* ***omogenei***

Set revisited

In JAVA è possibile ***parametrizzare*** *una classe rispetto a un* ***tipo generico:***

```
public class Set<T>{…}
```

`T` *è una* ***variabile di tipo*** *e indica un qualsiasi tipo.*

Creare un oggetto dalla classe `Set<T>` *richiederà di* ***specificare un tipo concreto****:*

```
Set<Integer> s = new Set<Integer>();
```

Set revisited

La variabile di tipo **`T`** *sarà utilizzabile ovunque necessario:*

Come ***tipo dei parametri****:*

```
public void insert(T x)
```

Come ***tipo del valore di ritorno*** *di un metodo:*

```
public T choose()
```

Come ***tipo di variabili*** *di istanza:*

Classi Generiche

Osservate che esiste ***un'unica classe*** **`Set<T>`**!

(questo ha varie conseguenze…)

Inoltre: se **`S<:T`**, ***non è vero che***

`Set<S> <: Set<T>`

(Questa forma di subtyping viceversa vale per alcune strutture dati, come per esempio gli array)

Le variabili di tipo possono essere usate anche nella ***definizione di interfacce***

Vincoli superiori

A volte è conveniente fare ***ipotesi più precise*** *su chi sono gli elementi (generici) di una collezione.*

Esempio: *si suppone che gli elementi di una collezione ordinata, siano ordinabili:*

```
interface Comparable<E>
    {boolean compareTo(E x);}

interface SortedColletion<E extends
    Comparable<E>>
```

Segnaposto

Supponiamo di voler scrivere un metodo:

```
static double sum(Set<??>)
```

per calcolare la sommatoria degli elementi di un insieme.

In tal caso, dovremmo supporre che l'insieme contenga elementi sottotipi di **`Number`**. ***(Sarà necessario invocare il metodo*** **`doubleValue()`** ***sugli elementi dell'insieme).*** *Tuttavia la segnatura:*

```
static double sum(Set<Number>)
```

non raggiunge lo scopo perché **`Set<Integer>`** ***non è sottotipo*** *di* **`Set<Number>`**

Segnaposto

Java permette tuttavia di scrivere:

```
static double sum(Set<? extends Number>)
```

`Number` *è* ***il vincolo superiore*** *per il tipo che ci si aspetta.*

Abbiamo in questo caso che:

`Set<Number>` ***è sottotipo*** *di* **`Set<? extends Number>`**

ma anche:

`Set<? Extends Integer>` ***è sottotipo*** *di* **`Set<? extends Number>`**

Vincoli inferiori

A volte è utile stabilire quali siano ***i vincoli inferiori di un tipo****: è possibile scrivere:*

`Set<? super Number>`

Mentre i ***vincoli superiori*** *sono utili per gli* ***argomenti****, i* ***limiti inferiori*** *potrebbero essere utili per i tipi dei* ***valori tornati*** *da un metodo.*

La scrittura **`Set<?>`** *è un'abbreviazione per*

`Set<? extends Object>` *e* ***non*** *per* **`Set<Object>`**

è sottotipo *di* **`Set<? extends Number>`**

Sottotipaggio di C<T>

Sarebbe in generale ***unsafe*** *dedurre*

`C<S> <: C<T>` *da* `S<:T`

Considerate il tipo definito dall'interfaccia:

```
Interface Blocco<A,B>
  { B valuta(A a);}
```

e un metodo `void m(Blocco<A,B> b)`

nel codice di `m` *il* ***blocco b*** *può essere usato per* ***invocare*** `valuta` *e* ***usare il valore tornato****:*

```
D d = new D();
C c = b.valuta(d);
```

Il codice è ben tipato se `B<:C` ***e*** `D<:A`

Sottotipaggio di C<T>

Sarebbe quindi ***scorretto*** *dedurre:*

A′<:A *e* **B′<:B**

⇒ **Blocco<A′,B′> <: Blocco<A,B>**

Viceversa per mantenere le disuguaglianze, posso ***far crescere il primo tipo*** *e far* ***descrescere il secondo****.*

Infatti è vicecersa ***corretto*** *dedurre:*

A<:A′ *e* **B′<:B**

⇒ **Blocco<A′,B′> <: Blocco<A,B>**

Il metodo potrebbe quindi avere tipo:

void m(Blocco<? super A,? extends B> b)