Programmazione II

13. Polimorfismo (PDJ 8)

Dragan Ahmetovic

Polimorfismo

Definizione:

La capacità di associare ad **un singolo simbolo dati di tipo diverso** o **comportamenti diversi**

- Meccanismi:
  - **Su comportamenti**
  - **Su tipi di dati**
- **Polimorfismo ad-hoc**
  - Procedure con comportamenti diversi in base al tipo su cui sono chiamate o dei loro parametri
- **Subtyping**
  - Possibilità di associare oggetti di tipo diverso ad un unico atomo (variabile/parametro/espressione)
- **Polimorfismo parametrico**
  - Abilità di una procedura (o un tipo) di comportarsi in maniera uniforme per parametri di tipo diverso

Polimorfismo Ad-hoc

Definizione:

Procedure con **comportamenti diversi** in base al **tipo su cui sono chiamate** o **dei loro parametri**

- Polimorfismo su comportamenti (su metodi)
- **Overloading** - sostituire il comportamento di un metodo in base ai parametri che riceve
  - Il cambio di comportamento dipende dai parametri ricevuti
- **Overriding** - sostituire il comportamento di un metodo in base al tipo di dato su cui è definito
  - Il cambio di comportamento dipende dal tipo di oggetto su cui il metodo è chiamato
- Overloading e Overriding **possono accadere in combinazione**
- Motivazioni:
  - Sintattico-semantica: operazioni **concettualmente simili** chiamate allo stesso modo
  - Specializzazione: fornire **comportamento più appropriato** ai tipi dei parametri

Subtyping (interface polymorphism / inclusion polymorphism)

Definizione:

Associare **oggetti di tipo diverso** ad un **unico atomo** (variabile/parametro/espressione)

- Polimorfismo su tipi di dati (su oggetti)
- Capacità di un atomo di avere un **Tipo Apparente**
  - Tipo di dato specificato **in fase di inizializzazione** per l'atomo considerato
  - Viene utilizzato per verificare i **comportamenti validi per l'atomo**
- Capacità di associare all'atomo un oggetto di un diverso **Tipo Concreto**
  - Tipo di dato dell'oggetto effettivamente utilizzato ed **assegnato all'atomo**
  - Viene utilizzato per identificare i **comportamenti effettivi dell'atomo**
- Motivazioni:
  - Permettere l'**utilizzo di oggetti di tipo diverso alla stessa maniera**
  - **Non vincolare** i tipi utilizzati - limitarsi alla richiesta dei comportamenti necessari

Polimorfismo Parametrico

Definizione:

Abilità di una **procedura o tipo** di comportarsi in **maniera uniforme per parametri di diversi tipi**

- **Senza definire singolarmente i diversi tipi** per cui è valido (come nell'overloading)
- Polimorfismo su tipi di dati e su comportamenti
- Comportamenti validi per parametri appartenenti a una **gerarchia di tipi**
  - Permettono le **stesse operazioni a prescindere dal tipo di dato**
  - Permettono **output di tipo appropriato** in base al tipo dei parametri
  - es: ArrayList, Iterator, Comparable possono funzionare con Integer, Double, String ...
- Motivazioni:
  - Possibilità di definire tipi in grado di **ospitare dati di diverso tipo**
  - Abilità di definire **comportamenti validi per oggetti e parametri qualsiasi**
- Possibile Polimorfismo Parametrico **per Subtyping** o attraverso **Generic Typing** (**Generics**)

Polimorfismo Parametrico

Esempio IntSet

```
1  public class IntSet {
2    private int[] els; //assumiamo rappresentazione come array
3
4    public void insert(int x) {
5      if(els == null)
6        els = new int[]{x}; //se non ci sono elementi inizializzo col primo
7      else if(!this.contains(x)) { //se non c'e'
8        int[] tmp = new int[els.length+1];  //nuovo array esteso di 1
9        for(int i=0; i < els.length; i++) //copio tutto
10         tmp[i] = els[i];
11       tmp[els.length] = x; //ultimo valore
12       this.els = tmp; //rimetto in els
13     }
14   }
15
16   public boolean contains(int x) {
17     for(int i : els)
18       if(i == x)
19         return true;
20     return false;
21   }
22 }
```

Polimorfismo Parametrico

Esempio StringSet

```
1  public class StringSet {
2    private String[] els; //cambia la rep
3
4    public void insert(String x) { //cambia il tipo dei parametri
5      if(els == null)
6        els = new String[]{x};
7      else if(!this.contains(x)) {
8        String[] tmp = new String[els.length+1];
9        for(int i=0; i < els.length; i++)
10         tmp[i] = els[i];
11       tmp[els.length] = x;
12       this.els = tmp;
13     }
14   }
15
16   public boolean contains(String x) {
17     for(String i : els)
18       if(i.equals(x)) //equals perche' oggetti
19         return true;
20     return false;
21   }
22 }
```

Polimorfismo Parametrico per Subtyping

Subtyping come soluzione per Polimorfismo Parametrico

Il meccanismo **storico** per il Polimorfismo Parametrico in java è attraverso **Subtyping**

- Utilizzo di un **Supertipo** S come **Tipo Apparente** di parametro (di input/output)
  - Permette l'utilizzo anche dei dati di tipo $T \prec S$ come parametri
- Essendo `Object` Supertipo di tutti i tipi, fornisce un **livello universale** di Polimorfismo
  - Ovvero, con `Object` è possibile un Set di tipi (non-primitivi) qualsiasi
  - Per oggetti primitivi si possono usare i tipi oggetto corrispondenti con (un)boxing
  - Si è **limitati** dalla possibilità di **utilizzare solo metodi di** `Object` o loro Override
  - Per questo motivo `Object` ha alcuni metodi di necessità universale (`toString`, `equals`, `hashCode`)
- Per permetere l'utilizzo di altri comportamenti condivisi si può usare un Supertipo
  - Es: `Solido` (Lab 6 ex 4) per poter confrontare i volumi tra diversi solidi
- Per output di **Tipo Apparente** S e **Tipo Concreto** T **serve casting** per usare i metodi di T

Polimorfismo Parametrico per Subtyping

Esempio Set di Object

```
1  public class Set {
2    private Object[] els; //tipo Object
3
4    public void insert(Object x) {
5      if(els == null) els = new Object[]{x};
6      else if (!this.contains(x)) {
7        Object[] tmp = new Object[els.length+1];
8        for (int i=0; i < els.length; i++)
9          tmp[i] = els[i];
10       tmp[els.length] = x;
11       this.els = tmp;
12     }
13   }
14
15   public boolean contains(Object x) {
16     for(Object i : els)
17       if(i.equals(x)) return true;
18     return false;
19   }
20
21   public Object choose() {
22       if(this.els == null) throw new NoSuchElementException(); //se vuoto
23       return els[0]; //ritorna primo
24   }
25
26   public String toString() {
27     String ret = "";
28     for(Object o : els) ret += o + " ";
29     return ret;
30   }
31 }
```

Polimorfismo Parametrico per Subtyping

Problema di Type Safety

La possibilità di **utilizzare un Supertipo** e di **richiedere casting manuale** può creare **problemi**

- Usando `Object` **non è possibile vincolare** in fase di utilizzo **il tipo permesso**
  - A seconda di quello che capita **a runtime** ci possono finire dentro dati di tipo diverso
- L'**errore avviene quando si accede** al tipo inserito in maniera errata, **durante il casting**
  - Per questo, il casting è considerato un'operazione "**Unsafe**"
  - Il **compilatore non sa se si può fare**, potrebbe causare **runtime exception**
- Possibile proteggersi mediante un **controllo** di `instanceof` o `try/catch` in fase di **casting**
  - Oneroso per il programmatore e **facile da dimenticare**
  - Fare il controllo internamente al tipo renderebbe inutile il polimorfismo
    (se devo prevedere nel metodo polimorfico i diversi tipi di output perdo la generalizzabilità)
- Questi limiti vengono superati dall'utilizzo di Generics che vedremo dopo

Polimorfismo Parametrico per Subtyping

Esempio Type Safety

```
1   public static void main(String[] args) {
2     Set s = new Set();
3     s.insert(4);
4     int twice = (Integer)s.choose() * 2; //serve cast esplicito
5     System.out.println(twice);
6   }
```

Esempio Type Safety 2

```
1   public static void main(String[] args) {
2     Set s = new Set();
3     s.insert("4"); //se viene inserito tipo imprevisto compila comunque
4     int twice = (Integer)s.choose() * 2; // ClassCastException a runtime
5     System.out.println(twice);
6   }
```

Polimorfismo Parametrico per Subtyping

Equals e Polimorfismo Parametrico

Nel **Polimorfismo Parametrico** è cruciale la corretta **verifica dell'uguaglianza** tra gli elementi

- Es: nel `Set`, per inserire un elemento si deve controllare che non sia già stato inserito
  - Il controllo mediante == non va bene perchè si tratta di **dati non primitivi**
- La **semantica di** `equals` **deve essere chiara**, anche nei confronti di sottotipi
  - Considerare un `equals` **conservativo**, valido solo per il Tipo Concreto
  - Considerare un `equals` **rilassato**, con possibile uguaglianza con sottotipi
    è buona norma che ai fini dell'`equals` la rep considerata sia solo nel supertipo
- Bisogna definire l'**uguaglianza tra le** `Collection`
  - Considerare due `Collection` uguali se hanno gli **elementi uguali**?
  - Considerare due collezioni uguali se **puntano allo stesso dato sulla heap**?
- Problematico il trattamento di **oggetti mutabili**
  - Se un oggetto cambia e **diventa uguale** ad un'altro?

Polimorfismo Parametrico per Subtyping

Problema Equals nel Polimorfismo coi Sottotipi

```
1  public static void main(String[] args) {
2    Set s = new Set();
3    Punto2D p1 = new Punto2D(1, 3);  //Lab 6 ex 2
4    Punto3D p2 = new Punto3D(1, 3, 5); //Lab 6 ex 2, variante con equals solo su Punto3D
5    s.insert(p1);
6    s.insert(p2); //deve essere inserito o no?
7    System.out.println(s.contains(p1));  //true
8    System.out.println(s.contains(p2));  //true - sembra di si
9    System.out.println(s); // Punto2D - x: 1.0, y: 3.0 //e invece no!
10 }
```

Problema Equals nel Polimorfismo coi Mutabili

```
1  public static void main(String[] args) {
2    Set s = new Set();
3    ArrayList l1 = new ArrayList();
4    l1.add("ciao")
5    ArrayList l2 = new ArrayList();
6    s.insert(l1);
7    s.insert(l2);
8    System.out.println(s); //dentro set ci sono ["ciao"] e []
9    l2.add("ciao"); //modifico l2
10   System.out.println(s); //dentro set ci sono ["ciao"] e ["ciao"], errore ripetizione!
11 }
```

Limiti del Polimorfismo Parametrico per Subtyping

Problemi:

- Possibilità di **utilizzare solo i metodi del Supertipo** utilizzato
  - Nel caso di object solo `equals`, `toString` ...
- Necessità di **conversione esplicita (Casting)** di tipo dei valori restituiti
  - Aumento di **carico di lavoro** e **possibilità di errore** per lo sviluppatore
- **Nessuna garanzia** di avere un tipo corretto restituito (**Type Safety**)
  - **Runtime error** avviene solamente **durante il Casting**
  - **Impossibile controllare** a **compile time** o **durante inserimento**
  - Usare `try/catch` o controllo su `instanceof` durante il Casting è **oneroso**
- Problemi nel **controllo di uguaglianza** (soprattutto nelle strutture dati)
  - La semantica di `equals` deve essere **chiara in base ai tipi utilizzati**
  - In particolare nel caso dei **sottotipi** possono esserci assimetrie
  - nel caso dei **mutabili** possibile che il risultato di `equals` cambi con le mutazioni

Programmazione II

13. Polimorfismo (PDJ 8)

Dragan Ahmetovic