Programmazione II

14. Generics

https://docs.oracle.com/javase/tutorial/extra/generics/index.html

https://www.baeldung.com/java-generics

Dragan Ahmetovic

Tipi e Metodi Generici

Tipi Generici

I limiti del Polimorfismo Parametrico per Subtyping sono superati dai **Tipi Generici**

- Specifica del tipo tra parentesi angolari permette di **vincolare i tipi di oggetti permessi**
  - es: `ArrayList<Integer> = new ArrayList<Integer>();`
  - se possibile **Inferenza di Tipo**, si può omettere l'**Argomento di Tipo** dall'**assegnamento**
  - es: `ArrayList<Integer> = new ArrayList<>();`
  - le parentesi angolari rimaste <> sono chiamate **Diamond**

Metodi Generici

Anche i **metodi** (su tipo non generico) possono essere generici

- Rispetto ai **parametri** o **tipo di ritorno**
  - es: `Classe.metodo<Integer>(lista);`
  - Se parametri sono del tipo generico è possibile omettere il **Witness** (se solo il tipo di ritorno **NO**)
  - es: `Classe.metodo(lista);`

Definizione

Definizione di un Tipo Generico

La definizione di un Tipo Generico è effettuata in fase di creazione della Classe

- **Inizializzazione** della classe (o interfaccia)
  - `class ClassName<T,U> {...}`
  - I **Tipi Generici** utilizzati sono messi **tra parentesi angolari dopo il nome della classe**
  - Il **Parametro di Tipo Generico** è per convenzione **lettera singola maiuscola**
  - Altrimenti sarebbe facile **confondere Parametro Generico e Tipo** di qualcosa
- Utilizzo dei **parametri di tipo generico** nella classe:
  - Si possono **specificare attributi** del Tipo del Parametro Generico
  - es: `T genericAttribute;`
  - Si possono **specificare parametri** dei metodi del Tipo del Parametro Generico
  - esempio: `public void method(T attribute);`
  - Si possono **restituire oggetti** del Tipo del Parametro Generico
  - esempio: `public U method();`

Definizione

Esempio di classe generica

```
1  public class Tupla<T,U> {
2  //OVERVIEW: tupla di tipi generici
3    private final T val1;
4    private final U val2;
5
6  //costruttori
7    public Tupla(T val1, U val2) {
8    //MODIFIES: this
9    //EFFECTS: inizializza this con due valori di tipi generici anche diversi
10     if(val1 == null || val2 == null)
11       throw new NullPointerException();
12
13     this.val1 = val1;
14     this.val2 = val2;
15   }
16
17 //metodi
18   public T getVal1() {
19     return this.val1;
20   }
21
22   public U getVal2() {
23     return val2;
24   }
25 }
```

Definizione

Metodi Generici

È anche possibile creare **metodi** indipendenti che **accettano parametri** o **output** Generici

- Questi **non richiedono che la classe stessa sia dichiarata come generica**
- I **tipi generici** utilizzati sono messi **tra parentesi angolari tra i modificatori di metodo**
- Definizione del metodo con **input del Tipo di Parametro Generico**:
  - `public <T> void method(T parameter)`
- Definizione del metodo con **output del Tipo di Parametro Generico**:
  - `public <T> T method(T parameter)`
- Definizione di un metodo con **diversi tipi generici come input e output**
  - `public <T,U> T method(U parameter)`

Definizione

Esempio di metodo generico

```
1  class StampaGenerico {
2  //OVERVIEW: classe statica che stampa un oggetto di tipo generico
3
4    static <T> void stampaInfo(T obj) {
5    //MODIFIES: System.out
6    //EFFECTS: stampa classe e valore di obj
7      System.out.println(obj.getClass() + ": " + obj);
8    }
9
10   public static void main(String[] args) {
11     StampaGenerico.stampa("Hello World");
12
13     StampaGenerico.stampa(5.3);
14
15     StampaGenerico.stampa(new Punto2D(4,5));
16   }
17 }
```

Bounds e Wildcards

Relazioni di Sotto/Super tipo e Generics

Nell'utilizzo dei Generici si presentano due questioni riguardanti le **relazioni di Sotto/Super tipo**

- Dati S e T dove $S \prec T$, in che relazione sono $S<U>$ e $T<U>$?
  - In questo caso vale $S<U> \prec T<U>$
  - Infatti è possibile **sostituire** $S<U>$ a $T<U>$ nel codice
  - es: `List<String> = new ArrayList<String>()`
- Dati S e T dove $S \prec T$, in che relazione sono $U<S>$ e $U<T>$?
  - Precisamente, $U<S>$ è **sottotipo** di $U<T>$?
  - es: dati `List<Dog> ld` e `List<Animal> la`, è possibile fare `la = ld`?
  - **Non è possibile** perché se aggiungo un `Cat` a `la`, `ld` non è più corretto
  - $U<S>$ **non può essere sottotip**o di $U<T>$
- Nota: invece, nel caso degli array è possibile fare `[]Object ao = new [4]String`
  - Gli **array conservano l'informazione di tipo** - `ao` rimane un array di String
  - Gli **array sono di dimensione fissa**, non c'è il rischio di prima

Bounds e Wildcards

Problema del Subtype tra Generici

```
1 List<Integer> li = new ArrayList<>();
2
3 //List<Object> lo = li; //Questo non e' permesso dal compilatore
4 List<Object> lo = (List)li; //Se invece si forza mediante cast ...
5
6 lo.add(new Object()); //... possibile aggiungere degli Object a lo ...
7
8 li.get(0); // ... Ma l'estrazione da li causa ClassCastException!
```

Bounds e Wildcards

Generici con Upper Bound

Potrebbe essere comunque **utile rendere permissibile** l'utilizzo di **tipi con Generici affini**

- Ad esempio, un metodo che somma una lista di `Number` o dei suoi sottotipi
  - `double add(List<Number> lst)` non può essere invocato con un parametro `List<Integer> lst`;
  - **non è un metodo generico**, è un metodo normale che **accetta un Generico come parametro**
- Possibile definire classi/metodi generici che **specificano i sottotipi permessi**
  - **Upper bound** - sintassi: `public static <T extends Number> double add(List<T> lst)`
  - Ovvero, accetta **tutti i sottotipi** di `Number` come Generic della Lista
  - Si possono anche controllare **molteplici upper bound**, es: `<T extends Number & Comparable>`
  - ovviamente al più un solo bound di **classe** e altri di **interfaccia**
- Potrebbe essere utile anche accettare parametri con **Generics** che sono **supertipi** di un tipo
  - es: consideriamo un metodo che legge da stdin una lista di interi: `readIntInto(List<Integer> dst)`
  - Sarebbe utile permettere di salvare gli elementi anche dentro una `List<Number>`
  - **Con Upper Bound non è possibile esprimere questo vincolo**

Bounds e Wildcards

Esempio di Metodo con Upper Bound

```
1  public class AddNumbers {
2  //OVERVIEW: classe statica che somma Number
3
4    public static <T extends Number> double add(ArrayList<T> lst) {
5    //EFFECTS: restituisce somma dei Number in lst
6
7      double sum = 0;
8
9      for (T n : lst) //it tipo di n e' T
10       sum += n.doubleValue(); //essendo sottotipo di Number ha doubleValue()
11
12     return sum;
13   }
14
15   public static void main(String[] args) {
16
17     //ArrayList<Integer> soddisfa i vincoli di Upper Bound
18     ArrayList<Integer> src = new ArrayList<Integer>(Arrays.asList(1,2,3,4,5));
19
20     System.out.println(add(src));
21   }
22 }
```

Bounds e Wildcards

Wildcards

Meccanismo che permette di usare sia **sottotipi** che **supertipi** Generici

- nella **dichiarazione delle variabili** o dei **parametri formali** dei metodi
  - Non si possono usare nella **definizione delle classi o metodi generici**
  - Attenzione: **stabiliscono il Tipo Apparente**; l'uso dell'effettivo **Tipo Concreto richiede casting**
- Unbounded wildcard <?>
  - Permette **generics di qualsiasi Tipo**
  - es `public static void printList(List<?> l)` - Attenzione: **NON è un metodo generico**
- Upper Bound wildcard <? `extends Tipo`>
  - Permette **generics che sono sottotipi di 'Tipo'**
  - es `public static double add(List<? extends Number> lst)`
- Lower Bound wildcard <? `super Tipo`>
  - Permette **generics che sono supertipi di 'Tipo'**
  - es `public static void readInto(ArrayList<? super Integer> dst)`

Bounds e Wildcards

Wildcard con Upper Bound

```
1  public class AddNumbers {
2  //OVERVIEW: classe statica che somma Number
3    public static double add(ArrayList<? extends Number> lst) { //Upper Bound Wildcard
4    //EFFECTS: restituisce somma dei Number in lst
5      double sum = 0;
6
7      for (Number n : lst) //uso di Upper Bound
8        sum += n.doubleValue(); //essendo sottotipo di Number ha doubleValue()
9
10     return sum;
11   }
12
13   public static void main(String[] args) {
14       //ArrayList<Integer> soddisfa i vincoli di Upper Bound
15     ArrayList<Integer> src = new ArrayList<Integer>(Arrays.asList(1,2,3,4,5));
16
17     System.out.println(add(src));
18   }
19 }
```

Bounds e Wildcards

Wildcard con Lower Bound

```
1  public class IntReader {
2  //OVERVIEW: classe per leggere Integer da Stdio
3    public static void readInto(ArrayList<? super Integer> dst) { //Lower bound Wildcard
4    //MODIFIES: dst
5    //EFFECTS: inserisce interi letti da System.in in dst
6      Scanner s = new Scanner(System.in);
7
8      System.out.println("Leggi:");
9      while(s.hasNextInt())
10       dst.add(s.nextInt()); //essendo superclasse di Integer puo' ricevere Integer
11   }
12
13   public static void main(String[] args) {
14     ArrayList<Number> dst = new ArrayList<>(); //Number soddisfa il Lower Bound
15     readInto(dst);
16
17     System.out.println("Letto:");
18     for(Number n : dst)
19       System.out.println(n);
20   }
21 }
```

Funzionamento dei Generics

Type Erasure

Per essere **retro-compatibile**, la **compilazione sostituisce generics con codice base**

- Processo di **Type Erasure**
  - Vengono **rimosse le parentesi angolari ed i Tipi di Parametri Generici**
  - I **riferimenti ai Tipi di Parametri Generici sono sostituiti con l'Upper Bound** (Object se nessuno)
  - Viene effettuato il **casting esplicito da/per tipo specificato** nelle parentesi angolari
  - Vengono aggiunti dei `metodi bridge` per sistemare overloading
- Si arriva al codice "Liskov-style"
- **Mischiare il codice con e senza generici** può creare `**unchecked warning**' e errori a runtime
  - Segno che la **Type Safety** del codice non è più garantita
  - Possibile avere problemi in fase di utilizzo come quando non si usano Generics
  - Gli unchecked warning si possono nascondere con `@SuppressWarnings("unchecked")`

Funzionamento dei Generics

Esempio Set Generico

```
1  public class Set<T> { //Uso Generico
2    private Object[] els; //tipo Object
3
4    public void insert(T x) {  //Tipo T, garantisco che sia compatibile
5      if(els == null)
6        els = new Object[]{x};
7      else if(this.contains(x))
8              return;
9      Object[] tmp = new Object[els.length+1];
10     for(int i=0; i < els.length; i++)
11       tmp[i] = els[i];
12     tmp[els.length] = x;
13     this.els = tmp;
14   }
15
16   @SuppressWarnings("unchecked") //Contiene solo T, evito warnings
17   public boolean contains(T x) {
18     for(Object i : els)
19       if(((T)i).equals(x)) //serve cast
20         return true;
21     return false;
22   }
23
24   @SuppressWarnings("unchecked") //Contiene solo T, evito warnings
25   public T choose() {
26     return (T)els[0]; //ritorna primo o NullPointerException se vuoto //serve cast
27   }
28
29   @SuppressWarnings("unchecked") //Contiene solo T, evito warnings
30   public String toString() {
31     String ret = "";
32     for(Object o : els)
33       ret += (T)o + " "; //serve cast
34     return ret;
35   }
36 }
```

Comparable e Comparator

Comparable

Un'interfaccia utile per **modellare dati in relazione d'ordine** è `**Comparable**

- Per comparare l'istanza di un tipo che la implementa con un'altro oggetto
  - In genere oggetto dello stesso (sotto)tipo
  - Restituisce valore intero positivo se this è maggiore, negativo se minore, 0 se sono uguali
  - Possibile usare il metodo `compare' presente in 'Double', 'Integer', etc`
- Possibile **ordinare** strutture dati o creare **strutture dati ordinate** (es: SortedSet)
  - es: `listSolidi.sort(null)' (gli elementi da ordinare devono implementare Comparable)`

Comparator

Se il tipo da comparare **non implementa Comparable** si può usare l'interfaccia **Comparator**

- Creo un tipo che implementa Comparator<Classe> dove Classe è il tipo da comparare
  - ridefinisco il metodo compare per confrontare i tipi desiderati
  - passo un'istanza del Comparator a chi deve farne uso (es: `listSolidi.sort(comparaSolidi)`)
  - è un'**interfaccia funzionale**, cioè fa da wrapper a un singolo metodo (vedremo la prossima volta)

Comparable e Comparator

Esempio Comparable

```
1  public abstract class Solido implements Comparable<Solido> { //Lab6 Ex4
2
3      public abstract double volume();
4
5      @Override
6      public int compareTo(Solido s) { //I sottotipi di Solido sono comparabili per volume
7          return Double.compare(this.volume(), s.volume());
8      }
9
10     public static void main(String[] args) {
11         ArrayList<Solido> l = new ArrayList<>();
12
13         ... //popolo la list
14
15         l.sort(null); //usa compareTo per ordinare la list
16     }
17 }
```

Comparable e Comparator

Esempio Comparator

```
1  public class ComparaSolidi implements Comparator<Solido>() {
2      @Override
3      public int compare(Solido s1, Solido s2) {
4          return Double.compare(s1.volume(), s2.volume());
5      }
6  }
7
8  public class Main{
9      public static void main(String[] args) {
10         ArrayList<Solido> l = new ArrayList<>();
11
12         ... //popolo la list
13
14         l.sort(new ComparaSolidi()); //passo un nuovo comparatore di solidi
15     }
16 }
```

Comparable e Comparator

Esempio Comparator Anonimo

```
1  public class Main{
2      public static void main(String[] args) {
3          ArrayList<Solido> l = new ArrayList<>();
4
5          ... //popolo la list
6
7          l.sort(new Comparator<Solido>(){ //passo al sort un comparator anonimo
8              @Override
9              public int compare(Solido s1, Solido s2) {
10                 return Double.compare(s1.volume(), s2.volume());
11             }
12         });
13     }
14 }
```

Programmazione II

14. Generics

https://docs.oracle.com/javase/tutorial/extra/generics/index.html

https://www.baeldung.com/java-generics

Dragan Ahmetovic