JAVA - Generics

Metodi Avanzati di Programmazione

Laurea Triennale in Informatica

Università degli Studi di Bari Aldo Moro

Docente: Pierpaolo Basile

Introduzione…

- Un meccanismo messo a disposizione dai linguaggi OO per supportare la generalizzazione di una soluzione computazionale è rappresentato dal polimorfismo per inclusione
- Infatti, è possibile implementare un metodo m che prende in input un oggetto della classe base A come argomento. Tale metodo potrà essere utilizzato su tutte le sottoclassi della classe A. (Ammesso che A non sia dichiarata final).
- La limitazione della applicabilità del metodo m ad un albero di ereditarietà con radice A può essere superato con l'utilizzo di un'interfaccia

…Introduzione…

- In alcuni casi, anche il vincolarsi ad una interfaccia può sembrare troppo restrittivo
- Questo è il concetto che sta alla base della astrazione generica
  - Con le Java Generics è possibile implementare il concetto di tipo parametrizzato, che permette di creare componenti (spesso dei contenitori) che sono semplici da utilizzare con più tipi

Containers e Java Generics...

- Una delle motivazioni principali delle Java Generics è rivolta alla possibilità di creare classi contenitori
- Consideriamo una semplice classe che contiene un solo oggetto di classe Object
- La classe è in grado di contenere oggetti di qualsiasi classe

…Containers e Java Generics…

```
public class Holder2 {
   private Object a;
   public Holder2(Object a) { this.a = a; }
   public void set(Object a) { this.a = a; }
   public Object get() { return a; }
   public static void main(String[] args) {
      Holder2 h2 =new Holder2(new Automobile());
      Automobile a = (Automobile) h2.get();
      h2.set("Not an Automobile");
      String s = (String)h2.get();
      h2.set(1); // Autoboxes to Integer
      Integer x = (Integer)h2.get();
   }
}
```

...Containers e Java Generics...

In alcuni casi nei quali è necessario inserire all'interno di un contenitore più oggetti omogenei, è possibile ricorrere alle Java Generics:

- Permettono di specificare il tipo degli oggetti contenuti
- Demandano al compilatore la possibilità di effettuare controlli di tipo al “compile time”.

Nell'esempio precedente, è possibile ricorrere alle generics

…Containers e Java Generics

```
public class Holder3<T> {
  private T a;
  public Holder3(T a) { this.a = a; }
  public void set(T a) { this.a = a; }
  public T get() { return a; }
}
```

Containers e Java Generics: definizione di Tuple...

In alcuni casi è possibile che sia necessario definire una funzione che restituisca non un singolo valore ma una coppia di valori o una tripla, ecc...

In questi casi si può definire una classe specifica per definire il tipo restituito.

Una soluzione differente e più generale sta nell'uso delle Java Generics per la definizione di ***tuple.***

Tuple

```
public class TwoTuple<A,B> {
  public final A first;
  public final B second;
  public TwoTuple(A a, B b) { first = a; second = b; }
  public String toString() {
    return "(" + first + ", " + second + ")";
  }
}
```

…Containers e Java Generics: definizione di Tuple.

A questo punto è possibile definire le funzioni che fanno uso delle “classi tupla” definite

```
TwoTuple tt = new TwoTuple<String,Integer>("hi", 47);
System.out.println(tt.toString());
```

Containers e Java Generics

Se consideriamo il tipo astratto di dati STACK, è possibile definire uno stack senza specificare il tipo degli oggetti che memorizzerà

```
public class LinkedStack<T> {

  private class Node<U> {
    U item;
    Node<U> next;
    Node() { }
    Node(U item, Node<U> next) {
      this.item = item;
      this.next = next;
    }
    boolean end() {
    return item == null && next == null; }
  }


  private Node<T> top = new Node<T>();

  public void push(T item) {
    top = new Node<T>(item, top);
  }

  public T pop() {
    T result = top.item;
    if(!top.end())
      top = top.next;
    return result;
  }
}
```

Containers e Java Generics

Particolarità di questa definizione:

- `Node<U>` è una **classe interna** a `LinkedStack<T>`. Questo significa che può essere utilizzata all'interno di `LinkedStack<T>`. Potrebbe anche essere visibile all'esterno, ma in questo caso è definita privata, quindi non è visibile all'esterno.

Containers e Java Generics

La classe `Node<U>` è di servizio alla definizione di `LinkedStack<T>`. Per questa ragione non è resa accessibile se non a quest'ultima.

In generale non è possibile creare un oggetto di una classe interna a meno che non si disponga di un oggetto della classe esterna.

Tuttavia nel caso di classe interna dichiarata **static** (detta anche *nested*) non è necessario disporre di tale oggetto.

Containers e Java Generics

Tornando all'esempio di contenitore generico, a questo punto è possibile creare uno Stack che contiene solo Stringhe:

```
LinkedStack<String> names = new LinkedStack<String>();
```

Controllo di tipo a compile time:

```
void printNames(LinkedStack<String> names) {
    String nextName = names.pop(); // no casting needed!
    names.push("John"); // ok
    names.push(new Integer(3)); // compile-time error!
}
```

Java Generics e Interfacce

Le Java Generics possono essere anche utilizzate per parametrizzare la dichiarazione di interfacce:

```
public interface Generator<T> { T next(); }
```

L'interfaccia ***Generator*** garantisce che il metodo ***next()*** restituisca un valore del tipo specificato dal parametro ***T***.

Un'altra interfaccia parametrizzata è ***Iterable<T> che*** forza l'implementazione del metodo:

Iterator<T> iterator()

Le classi che implementano tale interfaccia permettono ad un oggetto di essere usato nello statement "foreach".

Metodi Generici...

Oltre a parametrizzare la dichiarazione di intere classi, è possibile parametrizzare la dichiarazione di metodi all'interno di una classe.

Un metodo può essere definito generico indipendentemente dalla fatto che la classe sia generica oppure no.

Per di più, se un metodo definito in una classe parametrizzata è statico, tale metodo non accederà al parametro di tipo della classe.

...Metodi Generici...

Per definire un metodo come generico è sufficiente parametrizzare la sua dichiarazione:

Nell'esempio, la funzione ***f()*** è stata “sovraccaricata” (overloaded) ben sei volte.

f() accetterà anche valori di tipo primitivo mediante il meccanismo dell'autoboxing.

```
public class GenericMethods {
  public <T> void f(T x) {
    System.out.println(x.getClass().getName());
  }
  public static void main(String[] args) {
    GenericMethods gm = new GenericMethods();
    gm.f("");
    gm.f(1);
    gm.f(1.0);
    gm.f(1.0F);
    gm.f('c');
    gm.f(gm);
  }
}
```

…Metodi Generici…

Java introduce anche una semplificazione che permette di inferire il tipo in maniera automatica

```
List<String> ls = new ArrayList();
```

Tuttavia, questa inferenza di tipo ha dei limiti. Essa funziona solo per l'assegnazione. Se si passa il risultato della chiamata di un metodo come New.list() come argomento di un altro metodo, il compilatore non proverà a compiere l'inferenza di tipo e tratterà il risultato del metodo come se fosse di tipo Object.

Questo potrebbe causare degli errori di compilazione.

```
public class New {
  public static <K,V> Map<K,V> map() {
    return new HashMap<K,V>();
  }
  public static <T> List<T> list() {
    return new ArrayList<T>();
  }
  public static void main(String[] args) {
       //Il metodo map() non conosce i tipi passati
       // per argomento
    Map<String, List<String>> sls = New.map();
    List<String> ls = New.list();
  }
}
```

Non è possibile fare inferenza sul tipo in questo punto del codice

…Metodi Generici…

```
public class LimitsOfInference {
  static void f(List<String> list) {}
  public static void main(String[] args) {
     // f(New.list()); //non compila
  }
}
```

Il problema può essere risolto specificando esplicitamente il tipo al momento della invocazione del metodo.

...Metodi Generici...

```
public class LimitsOfInference {
  static void f(List<String> list) {}
  public static void main(String[] args) {
    f(New.<String>list());
  }
}
```

I metodi generici possono essere anche utilizzati in combinazione con metodi a numero variabile di argomenti

```
import java.util.*;

public class GenericVarargs {
  public static <T> List<T> makeList(T... args) {
    List<T> result = new ArrayList<T>();
    for(T item : args)
      result.add(item);
    return result;
  }
  public static void main(String[] args) {
    List<String> ls = makeList("A");
    System.out.println(ls);
    ls = makeList("A", "B", "C");
    System.out.println(ls);
    ls = makeList("ABCDEFFHIJKLMNOPQRSTUVWXYZ".split(""));
    System.out.println(ls);
  }
}
```

Il problema della cancellazione (erasure)

Le Java Generics lasciano comunque alcune questioni poco chiare. Per esempio, mentre è possibile ricorrere al letterale di classe per la classe ArrayList:

```
ArrayList.class
```

non è possibile ricorrere al letterale di classe per la classe ArrayList ottenuta parametrizzando il tipo del contenuto:

```
// ArrayList<Integer>.class
```

Il problema dell'erasure

Non è disponibile alcuna informazione sui tipi di parametri generici all'interno del codice generico.

Le generics del Java sono implementate usando l'*erasure*. Questo significa che ogni informazione di tipo è cancellata quando si ricorre all'astrazione generica.

Così ArrayList<String> e ArrayList<Integer> sono, di fatto, lo stesso tipo (ArrayList) al run time.

Wildcards

I tipi parametrici jolly (?), detti wildcard, servono a indebolire i vincoli di tipo.

? significa ‘qualunque tipo’

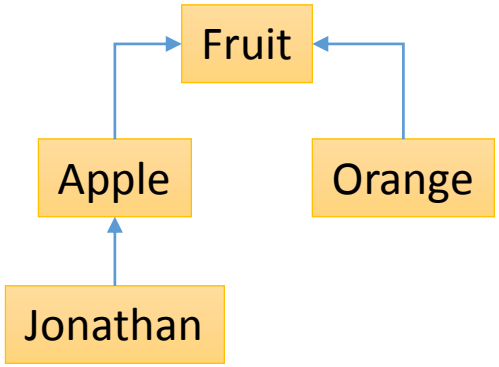
```
public class WildcardClassReferences {
  public static void main(String[] args) {
  Class<?> intClass = int.class;
  intClass = double.class;
  }
}
```

Wildcards

A volte un riferimento a un qualsiasi tipo può essere troppo 'generico'. Al fine di creare riferimenti più specifici (a un tipo e a tutti i suoi sottotipi) si può combinare la wildcard con la `extends`:

```
public class BoundedClassReferences {
  public static void main(String[] args) {
  Class<? extends Number> bounded = int.class;
  bounded = double.class;
  bounded = Number.class;
  }
}
```

```
class Fruit {}
class Apple extends Fruit {}
class Jonathan extends Apple {}
class Orange extends Fruit {}
public class CovariantArrays {
  public static void main(String[] args) {
    Fruit[] fruit = new Apple[10];
    fruit[0] = new Apple(); // OK
    fruit[1] = new Jonathan(); // OK
    // Runtime type is Apple[], not Fruit[] or Orange[]
    try {
      // Compiler allows you to add Fruit:
      fruit[0] = new Fruit(); // ArrayStoreException
                //errore a run-time, ma non a compile-time
    } catch(Exception e) { System.out.println(e); }
    try {
      // Compiler allows you to add Oranges:
      fruit[0] = new Orange(); // ArrayStoreException
    } catch(Exception e) { System.out.println(e); }
  }
}
```



Wildcards

Java non è in grado di controllare un eventuale errore a compile-time, ma solo al run-time.

Al contrario, con l'uso delle Generics:

```
List<Fruit> fList= new ArrayList<Apple>();
```

genererà un errore in fase di compilazione. Infatti non è possibile assegnare un generic che coinvolge oggetti di classe Apple a un generic che coinvolge oggetti di classe Fruit.

Wildcards

Questo comportamento deriva dal fatto che una List di Apple non è una List di Fruit (sebbene un oggetto di classe Apple è considerato un oggetto di classe Fruit).

Per poter gestire tale ereditarietà nelle generics si usano le Wildcards.

In questo esempio si mostra come List<? extends Fruit>

è considerato come una superclasse di ArrayList<Apple>.

Tuttavia, non è possibile invocare la *add* in quanto il compilatore non sa che *flist* è stata addirittura inizializzata come un *ArrayList* di *Apple*.

Infatti, teoricamente flist potrebbe anche essere una List<Orange>.

```
public class GenericsAndCovariance {
  public static void main(String[] args) {
    // Wildcards allow covariance:
    List<? extends Fruit> flist = new ArrayList<Apple>();
    // Compile Error: can't add any type of object:
    // flist.add(new Apple());
    // flist.add(new Fruit());
    // flist.add(new Object());
    flist.add(null); // Legal but uninteresting
    // We know that it returns at least Fruit:
    Fruit f = flist.get(0);
  }
}
```

Wildcards

Per questa ragione si perde la possibilità di passare qualunque oggetto, anche un oggetto di classe Object.

Al contrario, non sorgono problemi per la *get()* in quanto il compilatore sa che può restituire un oggetto della classe Fruit e sue sottoclassi.

```
public class Holder<T> {
  private T value;
  public Holder() {}
  public Holder(T val) { value = val; }
  public void set(T val) { value = val; }
  public T get() { return value; }
  public boolean equals(Object obj) {
    return value.equals(obj);
  }
  public static void main(String[] args) {
    Holder<Apple> apple = new Holder<Apple>(new Apple());
    Apple d = apple.get();
    apple.set(d);
    // Holder<Fruit> fruit = apple; // Cannot upcast
    Holder<? extends Fruit> fruit = apple; // OK Apple è sottoclasse di Fruit
    Fruit p = fruit.get();
    d = (Apple)fruit.get(); // Returns 'Fruit'
    try {
      Orange c = (Orange)fruit.get(); // No warning
    } catch(Exception e) { System.out.println(e); }
    // fruit.set(new Apple()); // Cannot call set()
    // fruit.set(new Fruit()); // Cannot call set()
    System.out.println(fruit.equals(d)); // OK
  }
}
```

Supertype Wildcards

List<? super Apple>

```
import java.util.*;

public class SuperTypeWildcards {
  static void writeTo(List<? super Apple> apples) {
    apples.add(new Apple());
    apples.add(new Jonathan());
    // apples.add(new Fruit()); // Fruit doesn’t extend Apple
  }
}
```