Insufficienza della JCF "classica": verso i tipi generici

Leggere cap. 16 di Programmazione di base e avanzata con Java

Sorgente:

Prof. Enrico Denti

Fondamenti di Informatica T-2 - Corso di Laurea in Ingegneria Informatica

Universita' di Bologna

JCF "CLASSICA": PROBLEMI

- **Usare il tipo `Object` per fare *contenitori generici* si è rivelato causa di *parecchi problemi***
  - **MOTIVO: equivale ad abolire il controllo di tipo!**
  - **CONSEGUENZA: *operazioni sintatticamente corrette* possono risultare *semanticamente errate,* causando a run-time errori inaspettati**
- **MORALE:**
  ***una JCF basata su Object non è "type safe": la correttezza è affidata a "commenti sul corretto uso"* anziché ai controlli del compilatore.**

Vediamo perché.

ESEMPIO

Si consideri il seguente frammento di codice:

```
List myIntList = new ArrayList(); // list of integers
myIntList.add(113);
Integer i = (Integer) myIntList.get(0);
```

- ***Come dice il commento e come suggerisce il nome,*** **qui si definisce una lista *con l'idea che sia una lista di interi***
- **Tuttavia, nulla esprime formalmente tale vincolo: a livello formale, `myIntList` è semplicemente una lista di `Object`**
- **Ergo, se il vincolo viene violato il compilatore non se ne può accorgere, *ponendo le premesse per il disastro:***

```
List myIntList = new ArrayList(); // list of integers
myIntList.add("ahahahah");        // errore semantico
Integer i = (Integer) myIntList.get(0);
```

ESEMPIO

A run time, ovviamente, succede il disastro!

```
Exception in thread "main"
java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Integer
```

IL NOCCIOLO DEL PROBLEMA

- ***nonostante ciò non fosse espresso formalmente,*** **la lista *sottintendeva l'ipotesi* di essere una lista di interi**
- **Se quell'ipotesi viene violata, a run-time *accade il disastro***
- **MA il compilatore non può intercettare la violazione, perché il vincolo è rimasto inespresso: dopo tutto, `myIntList` è solo una normale lista… di `Object`!**

CONCLUSIONE

La compilazione corretta non ha impedito di ritrovarsi con un programma sbagliato, perché *l'errore di progetto non è stato "smascherato" dal type system.*

MORALE: la JCF classica non è "type safe"
(sicura sotto il profilo dei tipi),
perché non garantisce che un programma che passi la compilazione sia corretto.

Ulteriori problemi nascono con le compatibilità fra tipi indotte dall'ereditarietà: *le discuteremo più avanti.*

VERSO UN NUOVO APPROCCIO

- Occorre prendere atto che *è intrinsecamente sbagliato abolire il controllo di tipo*
  - si dovrebbe rinforzarlo, non indebolirlo!
  - vogliamo VERA *TYPE SAFETY:*
    "se si compila, è certamente corretto"
- La genericità è necessaria, *ma nel modo giusto* che non può essere "`Object` ovunque"
- Per questi motivi, Java 1.5 e C# introducono il nuovo concetto di tipo parametrico (generico)
  - un modo per poter esprimere genericità in tipo
  - senza abusare del povero `Object`...

TIPI GENERICI

- **Anziché esprimere genericità usando il tipo più generale possibile (`Object`), *si introduce una notazione esplicita* per questo scopo:**

`<TIPO>`

- ***Il tipo diventa un parametro* con cui etichettare le collections**
  - **in ogni situazione: dichiarazione di riferimenti, argomenti di funzioni, classi, metodi di classe…**
- **Così *il type system si rafforza anziché indebolirsi***
  - **il compilatore può effettuare *controlli stringenti* di tipo per garantire TYPE SAFETY.**

RIPRENDENDO IL PROBLEMA..

- **Anziché collezioni di `Object`**
  - ***in cui di fatto si può mettere qualunque cosa…***
- **definiamo collezioni di `T`**
  - ***essendo `T` un TIPO GENERICO***

PRIMA

```
List myIntList = new ArrayList(); // list of integers
myIntList.add(113);
Integer i = (Integer) myIntList.get(0);
```

ORA

```
List<Integer> myList = new ArrayList<Integer>();
myList.add(113);
Integer i = myList.get(0); // non serve più il cast
myList.add("ahahahah");    // TYPE ERROR!
```

PROBLEMA: SOLUZIONE

- **Anziché collezioni di `Object`**
  - ***in cui di fatto si può mettere qualunque cosa…***
- **definiamo collezioni di `T`**
  - ***essendo `T` un TIPO GENERICO***

- ***Stavolta, è esplicitamente detto che `myList` è una lista di interi: il compilatore lo sa e può agire di conseguenza.***
- **Eventuali violazioni vengono subito intercettate.**
- **Valgono le regole generali sui tipi: una collezione di `T` può ospitare istanze o di tipo `T` o di sottotipi di `T`.**

ORA

```
List<Integer> myList = new ArrayList<Integer>();
myList.add(113);
Integer i = myList.get(0); // non serve più il cast
myList.add("ahahahah");    // TYPE ERROR!
```

ESEMPIO (REVISED)

```
import java.util.*;
public class ListaGenerica {
 public static void main(String args[]){
  List<Integer> list = new LinkedList<Integer>();
  for (int i=0; i<12; i++) list.add(i*i); //boxing
  System.out.println(list);
 }
}
```

```
---------- Java Run ----------
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

ESEMPIO (REVISED) - CONTROPROVA

```
import java.util.*;
public class ListaGenerica {
 public static void main(String args[]){
  List<Integer> list = new LinkedList<Integer>();
  for (int i=0; i<12; i++) list.add(i*i); //boxing
  list.add(3.1415); // non si compila!
  System.out.println(list);
 }
```

```
ListaGenerica.java:6: cannot find symbol
symbol  : method add(double)
location: interface java.util.List<java.lang.Integer>
  list.add(3.1415);
      ^
1 error
```

ESEMPIO (REVISED) - COMPROVA

- **Le classi wrapper numeriche (`Integer`, `Double`, `Float`) derivano tutte dalla classe-base astratta `Number`.**
- **Quindi, una lista di `Number` può ospitare ogni tipo di numeri… ad esempio, interi:**

```
import java.util.*;
public class ListaGenerica {
 public static void main(String args[]){
  List<Number> list = new LinkedList<Number>();
  for (int i=0; i<12; i++) list.add(i*i);
                        // OK: Integer extends Number
  list.add(3.1415); // OK: Double extends Number
  System.out.println(list);
 }
}
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 3.1415]
```

LA NUOVA JCF "GENERICA"

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Deque		ArrayDeque		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Tutta la JCF è stata riscritta con i generici

Le operazioni della nuova JCF "generica" si dicono *checked* (controllate) o *type-safe*

- **per converso, le operazioni della JCF "classica" si dicono *unchecked* e sono evidenziate con warning dal compilatore, *con esplicito invito a riscriverle.***

ESEMPIO

```
import java.util.*;
class TestBoxing {
 public static void main(String args[]) {
    List l = new ArrayList();   // USA JCF CLASSICA
    l.add(33); l.add(12);
    int i = (Integer) l.get(0); // HA BISOGNO DEL CAST
    System.out.println(l);
    System.out.println(i);
 }
}
```

```
---------- Java Compile ----------
Note: TestBoxing.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

ESEMPIO (DETTAGLI)

..e ricompilando con `-Xlint` (mosta tutti i warning) otteniamo infatti:

```
TestBoxing.java:5: warning: [rawtypes] found raw type: List
                List l = new ArrayList();
                ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List

TestBoxing.java:6: warning: [unchecked] unchecked call to add(E)
   as a member of the raw type List
                l.add(33); l.add(12);
                     ^
```

- il primo avvisa che stiamo usando un *raw type* come `List` (JCF classica) invece della corrispondente classe generica `List<E>`
- il secondo avvisa che, di conseguenza, la chiamata al metodo `add` è *unchecked* (non controllata sotto il profilo della type safety) perché invocata sul tipo grezzo `List` anziché su `List<E>`

ESEMPIO – VERSIONE GENERICA

```
import java.util.*;
class TestBoxing {
 public static void main(String args[]) {
    List<Integer> l = new ArrayList<Integer>();
    l.add(33); l.add(12);
    int i = l.get(0);    // CAST NON PIÙ NECESSARIO
    System.out.println(l);
    System.out.println(i);
 }
}
```

ESERCIZIO 4 – VERSIONE GENERICA

Obiettivo: conta le occorrenze delle parole digitate sulla linea di comando.

```
import java.util.*;
public class ContaFrequenza {
 public static void main(String args[]) {
   Map<String,Integer> m = new HashMap<String,Integer>();
   for (int i=0; i<args.length; i++) {
    Integer freq = m.get(args[i]);
    m.put(args[i], (freq==null ? new Integer(1) :
                    new Integer(freq.intValue() + 1)));
   }
   System.out.println(m.size() + " parole distinte:");
   System.out.println(m);
 }
}
```

```
>java ContaFrequenza cane gatto cane gatto gatto cane pesce
3 parole distinte: {cane=3, pesce=1, gatto=3}
```

ESERCIZIO 4 – VERSIONE GENERICA

Con boxing e unboxing automatico

```
import java.util.*;
public class ContaFrequenza {
 public static void main(String args[]) {
    Map<String,Integer> m = new HashMap<String,Integer>();
    for (int i=0; i<args.length; i++) {
     Integer freq = m.get(args[i]);
     m.put(args[i], (freq==null ? 1 : freq+ 1)));
    }
    System.out.println(m.size() + " parole distinte:");
    System.out.println(m);
 }
}
```

```
>java ContaFrequenza cane gatto cane gatto gatto cane pesce
3 parole distinte: {cane=3, pesce=1, gatto=3}
```