Java Generics

- Sunto di quanto letto nel libro
- *Java Generics and Collections*
- *di Maurice Naftalin & Philip Wadler*
- *O'REILLY*

Le Novità

- I GENERICI sono (forse) la modifica più corposa fra quelle inserite nella versione 5 del Java.
-
- Altre importanti sono:

- Boxing/Unboxing
- Foreach
- Varargs

-

L'unione fa la forza

- Le novità si apprezzano molto di più se usate insieme, formano un'unione sinergica:

```
List<Integer> ints = Arrays.asList(1,2,3);
for(int i : ints){
   System.out.printf("%02d%n",i);
}
```

- 01
- 02
- 03

Generici

- Un'interfaccia o una classe possono essere dichiarate in modo da dipendere da uno o più parametri di tipizzazione, che sono scritti fra parentesi angolari e dovrebbero essere specificati quando si dichiara una variabile appartenente all'interfaccia o alla classe o quando si crea una nuova istanza della classe.

Erasure - 1

- Il bytecode dello stesso codice con e senza generici è identico:

- ***`List`***
- ***`List<String>`***
- ***`<List<List<String>>`***

sono identici in bytecode e corrispondono alla plain old List che tutti conosciamo.

Erasure - 2

- E' il processo che converte il programma scritto con i generici nella forma senza generici che rispecchia più da vicino il bytecode prodotto.
- Il termine *Erasure* non è proprio corretto in quanto vengono rimossi i generici, ma aggiunti i cast.

Erasure - 3

- **Cast-iron guarantee:**
- il cast implicito aggiunto dalla compilazione dei generici non DEVE MAI FALLIRE.
- Questa regola si applica solo per il codice che non presenta *unchecked warnings*.

Erasure - 4

- I vantaggi dell'implementazione via Erasure sono:
-

- mantiene le cose semplici (non c'è nulla di nuovo)
- mantiene le cose piccole (una sola implementazione di List)
- semplifica l'evoluzione (la stessa libreria può essere acceduta da codice *generico* e da codice legacy)

Erasure - 5

- Un'altra conseguenza dell'Erasure è che it tipi array DIFFERISCONO in modo CHIAVE dai tipi parametrici:
- ***`new String[size]`***
- differisce sostanzialmente da
- ***`new ArrayList<String>()`***
- in quanto l'array mantiene l'informazione del tipo, il generico no

Subtyping & Wildcards - 1

- Il meccanismo del subtyping è centrale in tutto il linguaggio Java (*extends, implements*) e si fonda sul principio di sostituzione:
- *Ad una variabile di un dato tipo può essere assegnato un valore di ogni sottotipo e un metodo con un argomento di un dato tipo può essere chiamato con un argomento di ogni sottotipo.*

Subtyping & Wildcards - 2

- Esempio:
-
- ```
  List<Number> numbers = new ArrayList<Number>();
  ```
-
- ```
  numbers.add(2);
  ```
-
- ```
  numbers.add(3.14d);
  ```
-
- ```
  assert numbers.toString().equals("[2, 3.14]");
  ```
-

- Qui il principio vale fra List e ArrayList e fra Number e Integer e Double rispettivamente.

Subtyping & Wildcards - 3

- ATTENZIONE che List<Integer> NON E' UN SOTTOTIPO di List<Number> in quanto viene violato il principio di sostituzione come si evince dal seguente esempio:

```
List<Integer> integers = Arrays.asList(1, 2);

List<Number> numbers = integers; // non compila!

numbers.add(3.14d);

assert integers.toString().equals("[1, 2, 3.14]");
```

Subtyping & Wildcards - 4

- Gli array invece si comportano in modo differente, in quanto Integer[] viene effettivamente considerato un sottotipo di Number[] (in barba al principio di sostituzione).
- Per fare in modo che i generici si comportino similarmente agli array si devono utilizzare le *wildcard*

Subtyping & Wildcards - 5

- Wildcard con EXTENDS - 1

```
interface Collection<E> {
   ...
   public boolean addAll(Collection<? extends E> c)
   ...
}

-----------------

Collection<Number> numbers = new
ArrayList<Number>();
numbers.addAll(Arrays.<Integer>asList(1,2));
numbers.addAll(Arrays.<Double>asList(2.78d,
3.14d));
```

Subtyping & Wildcards - 6

• Wildcard con EXTENDS - 2

-
  - Le wildcard si possono usare anche nella dichiarazione delle variabili, anche se...
-
- ```
  List<Integer> integers = Arrays.asList(1, 2);
  ```
-
- ```
  List<? extends Number> numbers = integers;
  ```
-
- ```
  numbers.add(3.14d); //non compila!
  ```
-
- ```
  assert numbers.toString().equals("[1, 2, 3.14]");
  ```
-

Subtyping & Wildcards - 7

- Wildcard con SUPER – 1
- Consideriamo il seguente metodo di Collections:

```
public static <T> void copy(
                    List<? super T> dst,
                    List<? extends T> src);
```

Subtyping & Wildcards - 8

- Wildcard con SUPER – 2
- Come per ogni metodo *generico* il tipo può essere implicito o esplicitato. Nel codice che segue:

```
List<Object> objects = Arrays.<Object>asList(1, 2, "tre");
List<Integer> integers = Arrays.asList(5, 4);
Collections.copy(objects, integers);
assert objects.toString().equals("[5, 4, tre]");
```

- La riga che copia si poteva anche scrivere:

```
Collections.<Object>copy(objects, integers);
Collections.<Number>copy(objects, integers);
Collections.<Integer>copy(objects, integers);
```

- Implicitamente viene preso Integer che è il più specifico.

Subtyping & Wildcards - 9

- Wildcard con SUPER – 3
-
- Piccolo Qiz:
- In che senso le tre signature che segueno NON sono equivalenti?
-

```
•    public static <T> void copy1(List<? super T> dst,
                                  List<T> src);
•
•    public static <T> void copy2(List<T> dst,
                                  List<? extends T> src);
•
•    public static <T> void copy3(List<? super T> dst,
                                  List<? extends T> src);
```

Subtyping & Wildcards - 10

- The Get and Put PRINCIPLE:
-
- si usa un'*extends wildcard* quando si devono solo estrarre valori da una struttura, si usa una super wildcard quando si devono solo inserire valori in una struttura e non si usano wildcard quando si devono sia estrarre che inserire valori nella stessa struttura.

Subtyping & Wildcards - 11

- PROIBITO!
-
- E' utile osservare che per nessuna ragione se non la semplicità, un metodo non può esprimere un range ereditario di tipi, come per esempio:

```
double sumAndCount(
        Collection<? extends Number super Integer> coll,
        int n){
    count(coll, n);
    return sum(coll);
}
```

Subtyping & Wildcards - 12

- Arrays vs Generics, runtime vs compile time
- array subtyping è covariante:

  S[] è sottotipo di ***T[]*** se ***S*** è sottotipo di ***T***
- generic subtyping è invariante:

  List<S> non è sottotipo di ***List<T>*** se ***S*** è sottotipo di ***T tranne nel caso S==T***
- le wildcard reintroducono la convarianza:

  List<S> è sottotipo di ***List<? extends T>*** se ***S*** è sottotipo di ***T***
- le wildcard introducono la controvarianza:

  List<S> è sottotipo di ***List<? super T>*** se ***S*** è supertipo di ***T***

Subtyping & Wildcards - 13

- GLI ARRAY SONO UN TIPO DEPRECATO?
- Gli array covarianti possono essere considerati un artefatto per aggirare la mancanza dei generici nelle prime versioni del Java, per permettere metodi come:
- ***`public static void sort(Object[] oo);`***
- ma con l'avvento dei generici e la possibilità di scrivere:
- ***`public static <T> void sort(T[] oo);`***
- risultano ormai quasi un baco e vengono mantenuti per la compatibilità all'indietro.

Subtyping & Wildcards - 14

- Wildcard Capture
-
- quando un metodo generico è invocato il ***type parameter*** può essere scelto per interpretare lo ***unknown type*** rappresentato da una wildcard.
- Si considerino gli EQUIVALENTI:

-
- ```
  public static void reverse(List<?> l);
  ```
-
- ```
  public static <T> void reverse(List<T> l)
  ```

Subtyping & Wildcards - 15

```
public static void reverse1(List<?> list){
    List<?> tmp = new ArrayList<Object>(list);
    for(int i=0; i<list.size();i++){
        //non compila!
        list.set(i, tmp.get(list.size()-i-1));
    }
}

public static <T> void reverse2(List<T> list){
    List<T> tmp = new ArrayList<T>(list);
    for(int i=0; i<list.size();i++){
        list.set(i, tmp.get(list.size()-i-1));
    }
}

public static void reverse(List<?> list){
    reverse2(list);
}
```

Subtyping & Wildcards - 16

Restrizioni all'uso delle wildcard

1) al top-level nella creazione di istanze di classe (new)

```
new ArrayList<?>();
```

2) nei type parameter dei metodi generici

```
List<?> list = Lists.<?>factory();
```

3) nei supertypes (extends, implements)

```
class AnyList extends ArrayList<?>{...}
```

4)si noti che forme del tipo

```
new ArrayList<List<?>>();
```

sono permesse.

Comparison & Bounds - 1

Consideriamo l'interfaccia di comparazione

```
interface Comparable<T> {
     public int compareTo(T o);
}
```

1) anti-simmetria;
3) transitività;
4) congruenza (anche nel lancio di eccezioni);
5) compatibilità con `equals()`.

Comparison & Bounds - 2

```
public static <T extends Comparable<T>> T max(Collection<T> coll){
    T candidate = coll.iterator().next();
    for(T elt : coll){
        if (candidate.compareTo(elt) < 0){
            candidate = elt;
        }
    }
    return candidate;
}
```

`T` è BOUNDED da `Comparable<T>` e a differenza delle wildcards si può usare solo `extends` e mai `super`.
Come si vede è ammessa la ricorsività, anche mutua.

Comparison & Bounds - 3

Quando è possibile è tile accrescere la generalità per aumentare l'utilità del metodo. Se è possibile si dovrebbero rimpiazzare i tipi con wildcard.

```
public static <T extends Comparable<? super T>> T
                    max(Collection<? extends T> coll)
```

Si osservi che il Get and Put principle è rispettato. Nelle librerie java la signature del metodo è in realtà

```
public static <T extends Object & Comparable<? super
T>> T max(Collection<? extends T> coll)
```

per motivi di compatibità all'indietro.

Comparison & Bounds - 4

- Enumeration
-

```
public abstract class Enum<E extends Enum<E>>
                                    implements Comparable<E>{...}
```

-

```
class Season extends Enum<Season>{...}
```

Comparison & Bounds - 5

- Enumeration

```
public abstract class Enum<E extends Enum<E>>
                              implements Comparable<E>{
  private final String name;
  private final int ordinal;
  protected Enum(String name, int ordinal){
     this.name=name; this.ordinal=ordinal;
  }
  public final String name(){return name;}
  public final int ordinal(){return ordinal;}
  @Override
  public String toString() {return name;}
  public int compareTo(E o) {
     return ordinal - o.ordinal;//!!!
  }
}
```

Comparison & Bounds - 6

- Enumeration

```
// == enum Season { WINTER, SPRING, SUMMER, FALL }
final class Season extends Enum<Season>{
   private Season(String name, int ord){super(name, ord);}
   public static final Season WINTER = new Season("WINTER",0);
   //...
   public static final Season FALL = new Season("FALL",3);
   private static final Season[] VALUES = {WINTER,/*...*/FALL};
   public static Season[] values(){return VALUES.clone();}
   public static Season valueOf(String name){
      for(Season e : VALUES){
         if(e.name().equals(name)){return e;}
      }
      throw new IllegalArgumentException();
   }
}
```

Comparison & Bounds - 7

- Enumeration
-

•E' importante osservare che sia:

•

```
•  class Enum implements Comparable<Enum>
```

•

•che

•

```
•  class Enum<E> implements Comparable<E>
```

•

•risultano entrambe troppo generiche, non forzando il binding fra E e la sua Enum e non rendendo la comparazione ristretta al solo tipo E.

Comparison & Bounds - 8

• Bridges

•l'implementazione per *erasure*, nel caso di interfacce parametriche come `Comparable<T>` implica che il compilatore inserisca dei metodi detti *bridges* (*NB. covariant overriding -> .class permettono + metodi con la stessa signature, .java no*):

```
•   //Legacy
•   class Integer implements Comparable{
•       //...
•       public int compareTo(Integer o) {
•           //questo è un bridge
•           //...
•       }
•       public int compareTo(Object o) {
•           return compareTo((Integer)o);
•       }
•   }
•   //Generic
•   class Integer implements Comparable<Integer>{
•       public int compareTo(Integer o) {
•           //il metodo bridge è generato dal compilatore
•           //..
•       }
•   }
```

Dichiarazioni - 1

- Static members
- i membri statici di una classe non possono fare riferimento ad un tipo parametrico di una classe generica (erasure).
- Nested Classes
- Se la outer class ha parametri di tipo e la inner class non è statica allora i tipi sono visibili dalla classe interna

Dichiarazioni - 2

- Nested Classes

```
public class LinkedCollection<E> extends AbstractCollection<E>{
   private class Node{
       private E element;
       private Node next = null;
       private Node(E elt){element = elt;}
   }
   private Node first = new Node(null);
   private Node last = first;
   //...
}
public class LinkedCollection<E> extends AbstractCollection<E>{
   private static class Node<T>{
       private T element;
       private Node<T> next = null;
       private Node(T elt){element = elt;}
   }
   private Node<E> first = new Node<E>(null);
   private Node<E> last = first;
   //...
}
```

Dichiarazioni - 3

- Come lavora l'erasure

•cancella tutti i parametri di tipo dai tipi parametrizzati e sostituisce ogni variabile tipizzata con l'erasure dei sui bound o con Object se non ci sono bound o con l'erasure del bound più a sinistra se ce ne sono molteplici.

•

•due metodi distinti non possono avere una signature con la stessa erasure

•

•non si possono estendere o implementare classi che hanno la stessa erasure (i bridges sarebbero troppo complicati)

Evolution not Revolution - 1

- Compatiblità
-

- backward compatibility: sia versioni legacy che generic per ciascuna applicazione/libreria (versioning nightmare).
-
- migration compatibility: lo stesso codice deve funzionare sia con versioni legacy che con versioni generiche di una libreria.
-
- binary compatibility: garantita dal meccanismo dell'erasure.
-
  - binary compatibility assicura migration compatiblity.
-

Evolution not Revolution - 2

- Evoluzioni
-

generic library / legacy client: il caso più comune e centrale nell'ottica della migration compatibility; per es. vecchie applicazioni che possono funzionare anche con jvm 5 e le relative librerie.

-

dovunque sia definito un tipo parametrizzato, Java riconosce anche una versione non parametrizzata detta *raw type*.

-

Java permette il passaggio di un raw type dove si aspetta un generico (nonostante sia un supertipo), ma segnala l'eccezione con un *unchecked conversion warning*

-

-

-

Evolution not Revolution - 3

Evoluzioni

legacy library / generic client: caso più insolito, ma che si può presentare se non si ha accesso al sorgente. In questo caso ha senso aggiornare le librerie per utilizzare i generici nelle signature dei metodi, senza però modificarne il body. Ci sono tre modi:

1)cambiamenti minimi ai sorgenti: modifica delle signatures

2)creazione di file stub: creazione di una libreria con signatures ma senza body, compilazione e sostituzione della libreria in esecuzione.

3)utilizzo di wrappers (sconsigliato): introduzione di una gerarchia parallela fra wrapper e libreria; errori qualora il codice utilizzi la object identity

Reification - 1

- In Java gli array materializzano l'informazione inerente al tipo dei componenti, mentre i tipi generici non materializzano l'informazione relativa ai tipi dei parametri.
-
- La mancata armonia fra gli array e i generici è uno degli aspetti peggiori del linguaggio Java.
-
- Per orientarsi si sono coniati il:
  - Principle of Truth in Advertising
  - Principle of Indecent Exposure

Reification - 2

In Java un tipo è *reifiable* se è completamente rappresentato in runtime.

- tipo primitivo (`int`)
- classe o interfaccia non parametr. (`Runnable`)
- tipo parametrizzato con tutti i prametri unbunded wildcards (`Map<?,?>`)
- un raw type (`List, ArrayList`)
- un array i cui componenti sono reifiable (`int[], Number[], List<?>[]`)

Reification - 3

Non reifiable

- un variabile parametrica (`T`)
- un tipo parametrizzato con parametri espliciti. (`List<Number>`)
- un tipo parametrizzato con bounds

  (List `<? extends Number>`)
- Perciò `List<? extends Object>` `non è` reifiable mentre

  l'equiavalente `List<?>` sì!!! (ROTFL)

Reification - 4

Instance tests and Casts

Gli instance test e i cast dipendono dall'esame dei tipi in runtime e per tanto dipendono dalla reification. Per questo motivo un instance test o un cast a un tipo che non è reifiable usualmente solleva un warning.

Reification - 5

l'incubo di equals

```
•public boolean equals(Object o){
•   if(o instanceof List<E>){//compile error
•       Iterator<E> it1 = iterator();
•       Iterator<E> it2 = ((List<E>)o).iterator();//unchecked cast
•       while(it1.hasNext() && it2.hasNext()){
•           E e1 = it1.next();
•           E e2 = it2.next();
•           if(!(e1 == null ? e2 == null : e1.equals(e2))){
•               return false;
•           }
•       }
•       return !it1.hasNext() && it2.hasNext();
•   }else {return false}
•}
```

Reification - 6

l'incubo di equals

```
public boolean equals(Object o){
   if(o instanceof List<?>){
       Iterator<E> it1 = iterator();
       Iterator<?> it2 = ((List<?>)o).iterator();
       while(it1.hasNext() && it2.hasNext()){
           E e1 = it1.next();
           Object e2 = it2.next();
           if(!(e1 == null ? e2 == null : e1.equals(e2))){
               return false;
           }
       }
       return !it1.hasNext() && it2.hasNext();
   }else {return false}
}
```

Reification - 7

come se non bastasse

- *Un instance test di un tipo non reifiable è sempre un errore, un cast può anche funzionare:*

```
public static <T> List<T> asList(Collection<T> c){
  if(c instanceof List<?>){
     return (List<T>)c;
  }else throw new IllegalArgumentException();
}
```

Reification - 8

Exceptions

•Siccome il match delle clausole catch è realizzato tramite un test di instance, in Java nessuna sottoclasse di Throwable può essere parametrizzata!

Reification - 9

Creazione di Array

- In Java gli array trasportano informazioni sul tipo dei loro componenti. Queste informazioni vengono usate negli instance test e nei cast. Per questa ragione non è possibile creare un nuovo array se i suoi componenti non sono reifiable. Per tanto non si può scrivere
- `new T[SIZE]` o
- `new List<Integer>[SIZE].`
-
- L'impossibilità di creare array di generici è una delle restrizioni più severe del linguaggio Java.

Reification - 10

Putroppo non si può scrivere:

```
public static <T> T[] toArray(Collection<T> c){
    T[] a = new T[c.size()]; // compile-time error
    int i=0; for(T x : c){a[i++]=x;}
    return a;
}
```

e neppure:

```
public static <T> T[] toArray(Collection<T> c){
    T[] a = (T[])new Object[c.size()];//unchecked cast
    int i=0; for(T x : c){a[i++]=x;}
    return a;
}
public static void main(String[] args) {
    List<String> sl = Arrays.asList("uno","due");
    String[] sa = toArray(sl);//class cast error!
}
```

Reification - 11

L'aspetto importante dell'esempio precedente è la considerazione che gli unchecked warning corrompono la cast-iron guarantee, ma sopratutto il fatto che il punto in cui avviene il calss cast error può essere differente da quello in cui viene sollevato il warning.

Quindi gli unchecked warning possono essere davvero insidiosi e non dovrebbero essere mai sottovalutati o non attentamente considerati!

Reification - 13

Indecent Exposure Principle

Da quanto fin qui visto risulta importante enunciare il principio dell'indecent exposure che afferma:

Mai esporre pubblicamente un array i cui componenti non siano tipi reifiable.