Höhere Programmierkonzepte
Übung I

Prof. Dr. Nikolaus Wulff

Zum 28. Oktober 2020

Ko- und kontravariante generische Methodensignaturen widersetzen sich beim ersten Erlernen ziemlich hartnäckig dem anschaulichen Verständis – aber etwas Übung mag hier hilfreich sein, um tiefere Einsichten in die Problematik und eventuelle Lösungsansätze zu erlangen, wozu diese Übungsaufgaben dienen sollen.

1 Generische Erzeugung von Objekten

```
1   public static <T> List<T> createList(Class<T> type, int n) throws
        Exception {
2       List<T> list = new ArrayList<T>(n);
3       for (int j = 0; j < n; j++) {
4           list.add(type.newInstance());
5       }
6       return list;
7   }
8
9   public static <T> T[] createArray(Class<T> type, int n) throws
        Exception {
10      T[] array = ... // Help, implement me!
11      return array;
12  }
```

Listing 1: Generische Erzeugung von Arrays und Collections.

Der Quelltext (1) zeigt zwei generische statische Fabrikmethoden zum Erzeugen und Befüllen eines Feldes bzw. einer Liste mit beliebigen Java Objekten vom Typ `Class<T>` mit Hilfe der Reflection API. Einzige Voraussetzung an den Typ `T` ist, dass er einen public default Konstruktor zur Verfügung stellt.

Aufgabe

Leider bietet die zweite Methode lediglich die Signatur, vervollständigen Sie bitte die Implementierung.

2 Generische Algorithmen

```
1  class Buh {
2      private static int idGenerator = 0;
3      protected Integer id             = new Integer(++idGenerator);
4
5      @Override
6      public String toString() {
7          return String.format("%s-%d", getClass().getSimpleName(), id);
8      }
9  }
10
11 class Foo extends Buh implements Comparable<Foo> {
12
13     /* (non-Javadoc)
14      * @see java.lang.Comparable#compareTo(java.lang.Object)
15      */
16     public int compareTo(Foo that) {
17         return this.id.compareTo(that.id);
18     }
19 }
20
21 class Bar extends Buh {
22     // nothing special new, just another type ...
23 }
```

Listing 2: Drei generische Klassen zum Experimentieren.

Der Quelltext (2) zeigt drei beliebige Klassen `Buh`, `Bar` und `Foo` die eine eineindeutige generierte Id haben, so dass Instanzen gut zu unterscheiden sind – eine entsprechend angepasste `toString` Methode ist bereits implementiert und die `Foo` Klasse implementiert eine Vergleichsoperation.

Die `main` Methode des Quelltexts (3) zeigt die Verwendung der Fabrikmethoden aus dem ersten Teil, sowie eine Methode `mixit`, welche die beiden „foo“ und „bar“ Listen zusammenmischt, immer eine Foo, dann eine Bar Instanz und so weiter, wie auf der folgenden Konsolenausgabe ersichtlich:

```
[Foo-1, Foo-2, Foo-3, Foo-4]
[Bar-5, Bar-6, Bar-7, Bar-8]
[Foo-1, Bar-5, Foo-2, Bar-6, Foo-3, Bar-7, Foo-4, Bar-8]
```

Der Quelltext (3) ist nicht ganz vollständig, so dass hier noch Raum für weitere Experimente bleibt, um Generics noch besser zu verstehen – oder vollkommen zu verzweifeln...

Aufgabe

1. Implementieren Sie die fehlenden `equals` und `hashCode` Methoden unter Ausnutzung des `id` Feldes.

2. Wie lautet die Implementierung einer generischen `mixit` Methode, d.h. ohne das dort hart `Foo` oder `Bar` codiert wird?

3. Es ist ungünstig, dass die `mixList` untypisiert als `List<?>` codiert ist. Wie lautet der richtige passende Typ?

4. Bis lang wurden nur die `List` Kontainer Klassen verwendet. Wie ist eine entsprechende `mixit` Methode zu implementieren, die auf den bereits erzeugten Feldern mit gleicher Funktionalität arbeitet?

5. Macht es einen Unterschied, ob das gemischte Feld zurückgegeben oder als beschreibbarer Parameter an eine entsprechende `void mixit` Funktion übergeben wird?

6. Wie schaut es mit der Sortierbarkeit der drei Felder und Listen aus? Formulieren Sie entsprechende Sortieranweisungen, was passiert, was ist zu tun?

7. Wie muss ein generischer Filter formuliert werden, um aus einer Liste (oder einem Feld) bestimmte Typen herauszufiltern, so dass `Bar` und `Foo` Instanzen wieder entmischt werden?

```
1
2   public static <T> T[] sort(T[] a) {
3       Arrays.sort(a);
4       return a;
5   }
6
7
8   public static void main(String[] args) throws Exception {
9       final int n = 4;
10      List<Foo> fooList = createList(Foo.class, n);
11      List<Bar> barList = createList(Bar.class, n);
12      Foo[] fooArray = createArray(Foo.class, n);
13      Bar[] barArray = createArray(Bar.class, n);
14
15      List<?> mixList = mixit(fooList, barList);
16
17      System.out.println(fooList);
18      System.out.println(barList);
19      System.out.println(mixList);
20
21  }
```

Listing 3: Platz für einige generische Experimente.