Programmieren in Java

http://proglang.informatik.uni-freiburg.de/teaching/java/2011/

Java-Übung Blatt 8 (Generics)

2011-06-27

Hinweise

- Schreiben Sie Identifier *genau so*, wie sie auf dem Blatt stehen (inklusive Groß- und Kleinschreibung), nicht nur ungefähr.
- Exportieren Sie das Projekt als ZIP-Datei mit Namen `accountname-projectname.zip`. Beispiel: Peter Lustig (`lustigp`) gibt `ex02_1` ab als Datei `lustigp-ex02_1.zip`
- ZIP-Export in Eclipse: Rechtsklick auf die Wurzel des Projects, "Export...", "General/Archive File..."
- Mailen Sie die ZIP-Files an Ihren Tutor.

Abgabe: Montag, 4. Juli 2011, um 9.00 Uhr.

Anmeldung zum Projekt Die Anmeldung zum Projekt ist eröffnet! Das Projekt beginnt am Montag, 11. Juli 2011. Bitte finden sie sich in Gruppen von 2-3 Personen zusammen und melden Sie sich bis Freitag per Email bei `geffken@informatik...` mit `Anmeldung zum [Android|Desktop]-Projekt: Programmieren in Java` als Betreff an. Es wird ein Desktop- und ein Android-Projekt geben, zwischen denen sich Ihre Gruppe entscheiden muss. Geben Sie in der Email die Namen und Benutzernamen aller Gruppenmitglieder an. Alle Gruppenmitglieder einer Gruppe sollten in der Lage sein eine gemeinsame Übungsgruppe zu besuchen. Bitte vermerken Sie diesen Termin in der Anmeldung. Nähere Informationen zum Ablauf des Projektes finden Sie vor Projektbeginn auf der Vorlesungsseite.

Vier Seiten? Diesmal ist der Text besonders lang, weil reich an Erklärungstext. Keine Sorge: die Lösungen sind nicht auch besonders lang.

Aufgabe 1 (Von Pair zu Map, 5 Punkte)
Projekt: `ex08_1`. Package: `pair`.
Ein Hauptanwendungsgebiet für Generics sind Containerklassen (Listen, Mengen, Abbildungen,...). Wir fangen mit etwas ganz Einfachem an: dem Paar. Holen Sie sich das Skelett von der Homepage.

(a) Schreiben Sie eine generische Klasse `Pair<X,Y>`. Sie soll folgendermaßen verwendbar sein:

```
1  Brot br1 = ...; Wurst wu1 = ...;
2  Pair<Brot, Wurst> wurstbrot = new Pair<Brot, Wurst>(br1, wu1);
3  Brot br2 = wurstbrot.first(); // ==> ergibt br1
4  Wurst wu2 = wurstbrot.second(); // ==> ergibt wu1
```

Testen: ein Testfall, der in etwa so viel abdeckt wie das Beispiel.

(b) Fügen Sie Ihrer Klasse eine Methode `flip()` hinzu, die ein Paar zurückgibt, das dieselben Werte in umgekehrter Reihenfolge enthält: wenn Sie `flip()` auf (a, b) aufrufen, bekommen Sie ein neues Paar (b, a).
Testen: ein Testfall.

(c) Im Skelett finden Sie Listen (Interface `IList<E>`). Wenn man Paare in eine Liste steckt, kann man daraus ein Abbildung ("Map") bauen. Implementieren Sie mit der Klasse `PairListMap<K,V>` folgendes Interface:

```
1   public interface Map<K,V> {
2     /** Associate key with value: next time someone calls get(k)
3      * with a k which is equals() to key, the get−call shall
4      * return value */
5     void put(K key, V value);
6     /** Get the value associated with key, or null if none was
7      * associated with key.
8      */
9     V get(K key);
10    /** the number of key−value mappings in this map. */
11    int size();
12  }
```

Für Vergleiche können Sie die auf jedem Objekt definierte Methode `equals(Object other)` verwenden. Wenn ein neuer Wert zu einem schon bekannten Schlüssel eingetragen wird, dürfen Sie das alte Paar in der Liste belassen (und `size()` darf es mitzählen).
Test: drei Testfälle zum Thema "was man puttet, wird gegettet".

(d) Dass in der Liste alte Paare drinbleiben, wenn man mit `put` einem bekannten Schlüssel k einen neuen Wert v gibt, ist nicht schön. Besser wäre, wenn dann die alten Paare mit k hinausgeworfen würden.

Schreiben Sie einen Testfall, in dem Sie mit `put` und `size` prüfen, dass sich Ihre Map-Klasse "besser" verhält. Er sollte jetzt noch scheitern, sonst taugt er nichts.

Nutzen Sie dann in `put` die Methode `IList.filter(ISelect pred)`, um bei jedem `put` die Liste der Paare aufzuräumen. Die Methode liefert eine Liste, in der nur Listenelemente vorkommen, zu denen das `ISelect`-Objekt `pred` "ja" gesagt hat. Sie werden eine entsprechende `ISelect`-Klasse bauen müssen. Zu was muss sie "ja" sagen?

Aufgabe 2 (Benchmark, 4 Punkte)
Projekt: `ex08_2`. Package: `bench`.
Wenn ein Interface auf verschiedene Weisen implementiert ist, kann man mit Benchmarks die für ein bestimmtes Lastszenario bestgeeignete Implementation auswählen[1]. In dieser Aufgabe nutzen wir Generics, um eine Benchmarkklasse zu schreiben, die mit *beliebigen* Typen und *beliebigen dazu passenden* Lastszenarien umgehen kann – mal etwas anderes als immer nur Containerklassen.

Allgemein bauen wir eine Klasse `Benchmark<T>`, die die Geschwindigkeit zweier Objekte vom Typ `T` in einem gemeinsamen Szenario (implementiert als Objekt vom Typ `IScenario<T>`) vergleicht.

Wir schicken zwei Listenklassen aus der Standardbibliothek ins Wettrennen. Dort gibt es das Interface `java.util.List<E>`. Es enthält (unter anderem) die Methoden

- `void add(E elem)` fügt ein Element hinten an
- `int size()` Anzahl der Elemente
- `E get(int i)` ergibt das Element an `i`-ter Stelle (für $i = 0 \ldots$ `size()` $- 1$)

Das Interface `List<E>` wird von `ArrayList<E>` und `LinkedList<E>` implementiert. Wir werden einen Benchmark mit `T` $\equiv$ `List<Integer>` veranstalten und als Kandidaten jeweils eine Instanz von `ArrayList<Integer>` und `LinkedList<Integer>` ins Rennen schicken.

Holen Sie sich zunächst das Skelett.

[1] Diese Messung kann die Betrachtung der Worst-Case-Komplexität der Algorithmen nicht ersetzen, nur ergänzen.

(a) Im Skelett ist eine Szenario-Klasse vordefiniert. Definieren Sie eine weitere:

```
1  /** Scenario consists of a few thousand adds, then many gets
2   * to different addresses. */
3  public class RandomAccessScenario implements IScenario<List<Integer>> {
4    ...
5  }
```

Test: keiner verlangt (aber Testvorschlag: wenn man ein `XXXScenario` auf einer leeren Liste laufen lässt, hat sie nachher n Elemente).

(b) Schreiben Sie die Benchmarkklasse:

```
1  /** Generic benchmarker */
2  public class Benchmark<T> {
3    /** Run scenario on each of the input objects, measuring
4     * the time each run takes. Print a summary of the
5     * measurements to System.out. The input objects must be fresh!
6     */
7    public void compare(IScenario<T> scenario, T impl1, T impl2){
8    ...}
9  }
```

Sie soll mittels `System.nanoTime()` messen, wie lange das `scenario` auf `impl1` bzw `impl2` braucht, und das freundlich auf `System.out` ausgeben.

Die Benchmarkklasse hat keine Ahnung von Listen! Wenn in der Klasse irgendwo das Wort `List` vorkommt, ist etwas verkehrt.
Test: keiner verlangt.

(c) Schreiben Sie eine Main-Klasse, die mit `Benchmark` vergleicht, wie lange *frische* `ArrayList<Integer>`- und `LinkedList<Integer>`-Objekte für Ihre beiden Szenarien brauchen. Achtung: jedes Objekt darf nur einmal in einem Szenario verwendet werden, dann ist es nicht mehr frisch.

Aufgabe 3 (Invarianz, 3 Punkte)
Abgabe als Textdatei.

Ihr Cousin Konstantin hat eine Programmiersprache *Kova* entworfen, die die gleichen Regeln und Sprachelemente wie Java hat, aber in der für beliebige generische Klassen `C<T>` gilt: falls `A` Subklasse von `B` ist, ist `C<A>` Subklasse von `C<B>` (also kann man `C<A>`-Instanzen dort anwenden, wo `C<B>` erwartet werden).

Schreiben Sie damit ein typkorrektes *Kova*-Programmstück, in dem eine Instanz von Klasse `Animal` erzeugt wird und dann die Methode `weaveNet()`, die nur in Klasse `Spider` vorhanden ist, auf diesem Objekt aufgerufen wird – ein Fehler zur Laufzeit, den das Typsystem eigentlich verhindern sollte.

Sie können voraussetzen, dass eine Klasse `List<T>` mit Methoden zum Einfügen und Herausholen von Elementen schon vorhanden ist und `Spider` eine Subklasse von `Animal` ist.

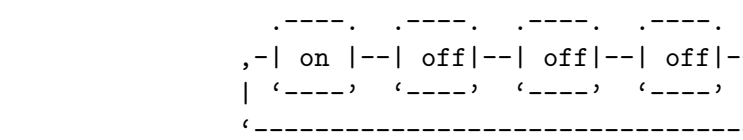
Nachholaufgaben für Thema 6: Zirkuläre Klassen

Mit dieser Aufgabe können Sie den Stoff von Thema 6 (dem Thema von Blatt 6) wiederholen und nochmal Punkte für Thema 6 erwerben.

Aufgabe 4 (Zelluläre Automaten, 4 Punkte)
Projekt: `ex06_4` Package: `cellular`

Wir betrachten ein System aus Zellen `Cell`, die entlang einer Linie angeordnet sind. Jede Zelle kennt ihren linken und rechten Nachbarn. Der Einfachheit halber ist der rechte Nachbar der rechtesten Zelle gerade wieder die linkeste Zelle (und umgekehrt).

Zellen können an oder aus sein (lebendig/tot, aktiv/passiv, etc. – wir nennen es einfach “an” und “aus”).



Ob eine Zelle an oder aus ist, entscheidet sie folgendermaßen: wenn von ihr und ihren beiden Nachbarn eine ungerade Anzahl an sind (also alle drei oder genau eine), ist sie an, sonst aus.

Simulieren Sie eine solche Zellkultur von 80 Zellen. Implementieren Sie die Zellen als Klasse `Cell`, die ihren linken und rechten Nachbarn kennt und Auskunft über ihren An-aus-Zustand liefern kann. Jede `Cell` soll ihren gegenwärtigen und ihren nächsten An-aus-Zustand speichern.

In der Simulation soll in jedem Zyklus zunächst jede Zelle ihren nächsten Zustand berechnen und dann jede Zelle ihren nächsten Zustand zu ihrem gegenwärtigen Zustand machen.

Bauen Sie eine `Main`-Klasse, die die Simulation durchführt und nach jedem Schritt die Zustände der Zellen als eine Zeile auf System.out ausgibt mit verschiedenen Zeichen für die beiden Zustände (hier für eine Kultur aus 22 Zellen):

```
-O--------------------
OOO--------------------
-O-O-----------------O
-O-OO---------------OO
-O---O-------------O--
OOO-OOO-----------OOO-
-O---O-O---------O-O--
```