3. Übungsbesprechung Programmkonstruktion

Karl Gmeiner

karl@complang.tuwien.ac.at

December 12, 2011

Rückblick und Ausblick

Test 1

- `String` ist kein elementarer Datentyp.
- `int i = 0; int a = i * foo();`. `foo()` wird ausgeführt!
- Einsichtnahme 19.12., 10:00, Argentinierstraße 8/4.

Test 3

- Praxis: Rekursive Datentypen, Iterator/Comparable/Comparator, Generizität
- Theorie: Kapitel 3+4+5

Aufgabe 1: Nicht-Terminierung, Schleifeninvarianten

Aufgabe

1. Warum terminiert die Schleife nicht?
2. Schleifeninvarianten finden (formale Variante von 1)
3. Korrekte Schleife angeben.

- Schleifendurchlauf für Fehlerfall.
  - Was sollte passieren?
  - Was passiert?
- Schleifen-Invarianten: Formale Bedingunge für Variablen angeben (wenn nötig Werte zwischenspeichern).

Aufgabe 2: Generischer Stack

```
public class Stack {
    private int[] array;
    private int pointer;

    public Stack(int maxCapacity) {
        this.array = new int[maxCapacity];
        this.pointer = 0;
    }

    public int pop() {
        return array[--pointer];
    }

    public void push(int value) {
        array[pointer++] = value;
    }
}
```

Problem mit Arrays und Generizität in Java

```
public class Stack<T> {

    // ...

    public Stack(int maxCapacity) {
        this.array = new T[maxCapacity]; // Nicht moeglich!
        this.pointer = 0;
    }

    // ...
}
```

Generizität in Java

Type erasure

- `<T>` wird beim Kompilieren gelöscht.
- `Stack<Integer>` wird zur Klasse `Stack`
- Typinformation zu `T` steht zur Laufzeit nicht zur Verfügung.

Nicht mögliche Operationen:

- `new T();`
- `obj instanceof T`
- `new T[maxCapacity];`

Möglicher Ersatz:

- `Object` verwenden
- Instanz von `Class` übergeben und Reflection verwenden.
- Auf `ArrayList` zurückgreifen

Stack mit generischem Container (ArrayList)

```
public class Stack {
    private ArrayList<Integer> array;
    private int pointer;

    public Stack(int maxCapacity) {
        this.array = new ArrayList<Integer>(maxCapacity);
        this.pointer = 0;
    }

    public int pop() {
        return array.get(--pointer);
    }

    public void push(int value) {
        array.set(pointer++, value);
    }
}
```

Exkurs: Iterator auf verkettete Listen

```
public class List<T> {
    private ListNode head;

    public List() { head = null; }
    public void add(T value) { head = new ListNode(value, head); }

    private class ListNode {
        private T value;
        private ListNode next;

        ListNode(int initValue, ListNode initNext) {
            this.value = initValue; this.next = initNext;
        }

        T getValue() { return value; }
        ListNode getNext() { return next; }
    }
}
```

Exkurs: Iterator auf verkettete Listen (2)

```
public class List<T> implements Iterable<T> {

    public Iterator<T> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<T> {
        ListNode current = head;

        public boolean hasNext() {
            return current != null;
        }

        public T next() {
            ListNode last = current;
            current = current.getNext(); // Naechstes Element
            return last.getValue();
        }
    } // remove fehlt!
}
```

Aufgabe 3: Comparable und Comparator

Generische Methoden

Methoden mit Typ-Parameter:

```
public class GenMethod {
    public static <T> T get(boolean first, T t0, T t1) {
        if(first) return t0;
        else return t1;
    }

    public static void main(String...args) {
        Object o = get(true, "hello", 4); // Typ-Inferenz
        // oder GenMethod.<Object> get(true, "hello", 4);
    }
}
```

<T extends Comparable<? super T>>

- T implementiert das Interface Comparable<? super T>>
- Typ T kann mit Objekten vom Typ T (oder einem Obertyp von T) verglichen werden.
- Lösung: class IntNumber implements Comparable<IntNumber>.

Methode compareTo(IntNumber o)

- Rückgabewert ist int
  - a.compareTo(b) < 0: a ist kleiner als b, $a < b$.
  - a.compareTo(b) > 0: a ist größer als b, $a > b$.
  - a.compareTo(b) == 0: a ist gleich b, $a == b$(*)

(*) Für alle c: a.compareTo(c) * b.compareTo(c) > 0

`static <T> void sort(List<T> list, Comparator<? super T> c)`

`Comparator<? super T> c`

- Objekt *c* implementiert `Comparator` für `T` oder einen Obertyp von `T` (Methode `compare(T a, T b)`).
- Sinnvoll, wenn `T` nicht `Comparable<T>` implementiert, oder
- wenn in einer anderen Reihenfolge als von `compareTo` vorgegeben sortiert werden soll.

Beispiel: Comparator für String, unabhängig von Groß-Klein-Schreibung

```
public class MyComparator implements Comparator<String> {
    public int compare(String a, String b) {
        a = a.toUpperCase();
        b = b.toUpperCase();

        return a.compareTo(b);
    }
}
```

```
List<String> strings = ...;
Comparator<String> c = new MyComparator();

Collections.sort(strings, c);
```