

Einführung in die Programmierung
WS 2018/19

Übungsblatt 13: Listen, Generics

Besprechung: 04.02.2019 - 08.02.2019

Aufgabe 13-1 *Generics*

Das Minimum eines Arrays kann bestimmt werden, indem man das Array sortiert und das erste Element zurückgibt. Beispiele sehen Sie in den unten angegebenen Klassen für Integer und Strings. Implementieren Sie eine generische Klasse `Minimum<T>`, sodass für Arrays mit beliebigem Typ `T` das Minimum - wie oben beschrieben - bestimmt wird. Wenn das nicht möglich ist, geben Sie einen entsprechenden Hinweis. Können Sie mit Ihrer Klasse auch das Minimum in einem `int`-Array finden? Was passiert, falls die Datentypen im Array sowohl vom Typ `String` als auch vom Typ `Integer` sind? Beispiel: `Object[] objectarray = {new Integer(1), "hallo"};`

```
import java.util.Arrays;

public class MinimumInteger {

    private Integer[] array;

    public MinimumInteger(Integer[] x){
        this.array = x;
    }

    public Integer getMinimum(){
        Arrays.sort(array);
        return array[0];
    }
}
```

```
import java.util.Arrays;

public class MinimumString {

    private String[] array;

    public MinimumString(String[] x){
        this.array = x;
    }

    public String getMinimum(){
        Arrays.sort(array);
        return array[0];
    }
}
```

Lösungsvorschlag:

`int`-Arrays können nicht benutzt werden, da `int` primitiv und kein Objekt ist. Daher gibt es auch keine Typisierung dafür.

Zwei Objekte innerhalb des Arrays müssen nicht unbedingt vergleichbar sein (`Integer/String`, `array-/array,...`). Weil die Typen inkompatibel sind, wirft `compareTo` (in `Arrays.sort()`) eine `Exception`.

Aufgabe 13-2 *Listen*

In der Vorlesung haben Sie Listen kennengelernt. Die `List`-Klasse sowie die `Entry`-Klasse können Sie in der Form herunterladen, wie sie bereits besprochen wurden. Im Folgenden sollen Sie weitere Erweiterungen vornehmen.

- (a) Implementieren Sie die Methode `T last()`, die das letzte Listenelement vom Typ `T` zurückgibt.

- (b) Implementieren Sie die Methode `void postfix(Entry<T> entry)` analog zur `prefix`-Methode. Ihre Methode soll allerdings ein Element an das Ende der Liste hinzufügen.
- (c) Implementieren Sie die Methode `void concat(List<> list)`, die an den letzten Listeneintrag den Kopfeintrag der übergebenen Liste anhängt. Für eine Liste `l = (1, 2, 3)` ist dann `l.concat(4, 5) = (1, 2, 3, 4, 5)`.
- (d) Implementieren Sie die Methode `void clone()`, die eine tiefe Kopie der aktuellen Liste zurückgibt. Kommentieren Sie hier besonders genau Ihren Code!
- (e) Implementieren Sie die Methode `List<T> concatClone(List<> list)`, die eine neue Liste erzeugt, die erst die Listenelemente der gegebenen Liste und danach die Elemente von `list` enthält. Die Reihenfolge der Elemente soll beibehalten werden. Zum Beispiel ist `(1, 2, 3, 6, 7, 9)` die Konkatenation von `(1, 2, 3)` und `(6, 7, 9)`.
Diskutieren Sie knapp die Vorteile beider Konkatenationsmethoden. Ein Vorteil pro Methode reicht dabei.
- (f) Implementieren Sie die Methode `List<T> reverse()`, die eine neue Liste erstellt, die die Listenelemente in umgekehrter Reihenfolge enthält. Die umgekehrte Liste von `(1, 2, 3, 6, 7, 9)` ist `(9, 7, 6, 3, 2, 1)`.
- (g) * Implementieren Sie die Methode `removeDuplicates()`, die aus einer Liste alle Elementduplikate herauslöscht und nur noch die ersten Vorkommen eines Elements enthält. Ein Löschen der Duplikate in `(7, 2, 3, 2, 7, 9)` ergibt die Liste `(7, 2, 3, 9)`.
- (h) Statt einem Attribut `private int size` können wir dies auch dynamisch berechnen. Sei im Folgenden eine Methode gegeben, die die Länge einer Liste dynamisch berechnet:

```
public int sizeDyn() {
    int size = 0;
    if(this.head != null){
        Entry<T> entry = this.head;
        while(entry != null){
            size++;
            entry = entry.getNext();
        }
    }
    return size;
}
```

Terminiert die Methode? Begründen Sie Ihre Antwort. Bei Nicht-Terminierung geben Sie außerdem ein knappes Codebeispiel an.

Lösungsvorschlag:

Terminiert nicht. Da `Entries` direkt in die Liste eingetragen werden können, kann ein `Entry` rekursiv auf sich selbst zeigen. Alternativ kann eine Subliste zyklisch sein. Beispiel:

```
List<Integer> circle = new List<>();
Entry<Integer> entry = new Entry<>(1);
circle.prefix(entry);
entry.setNext(entry);
System.out.println(circle.size());
System.out.println(circle.sizeDyn());
```

- (i) Diskutieren Sie in maximal 3 Sätzen den Unterschied zwischen Arrays und Listen.

Lösungsvorschlag:

Arrays haben feste statische Längen, Listen dürfen dynamisch sein. Zugriff auf Arrays direkt, aber Listen müssen durchlaufen werden. Fehlerquellen bei Listen sehr hoch durch Pointer, aber flexibler als Arrays (Stacks, Queues, Bäume,...).

Aufgabe 13-3 *Bäume*

- (a) Beschreiben Sie knapp, wie man die Listenklasse strukturell erweitern muss, damit man binäre Bäume modellieren kann. Gehen Sie zur Einfachheit von einer maximal öffentlichen Sichtbarkeit der Attribute aus und ignorieren Sie die Datenkapselung. Zur Erinnerung:

```
public class List<T> {
    public Entry<T> head;
}

public class Entry<T> {
    public T element;
    public Entry<T> next;
}
```

Lösungsvorschlag:

Ein zweiter Pointer in der Entry-Klasse ermöglicht die zwei Kinder eines Knotens in einem Binärbaum.

- (b) Sind auch Bäume mit variablem Knotengrad möglich? Beschreiben Sie kurz. Code ist für diese Aufgabe nicht nötig.

Lösungsvorschlag:

Ja, nutze einfach Listen von Pointern statt nur einem Pointer:

```
public class Entry<T> {
    public T element;
    public List<Entry<T>> next;
}
```