

# Advanced Aspects of Object-Oriented Programming (SS 2009)

## Practice Sheet 6

Date of Issue: 26.05.09  
Deadline: 08.06.09  
(until 10 a.m. as PDF via E-Mail)

### Exercise 1 Aliasing

- Give an example for dynamic aliasing.
- Define the relationship between capturing and aliasing.
- For which inputs does the following method `addAll` not work?

```
class MyStack<T> {  
    private ArrayList<T> elems = new ArrayList<T>();  
  
    public void push(T t) {  
        elems.add(0,t);  
    }  
  
    public T pop() {  
        return elems.remove(0);  
    }  
  
    public void addAll(MyStack<T> other) {  
        for (int i = 0; i < other.elems.size(); i++) {  
            push(other.elems.get(i));  
        }  
    }  
}
```

- Provide two collections such that the call to the `equals` method leads to a stack overflow error.

```
Collection<Object> c1 = ...  
...  
Collection<Object> c2 = ...  
...  
c1.equals(c2);
```

### Exercise 2 Immutable Classes in the JDK

Select five classes from JDK 5 which conform to the definition of immutability as given in the lecture and explain the reason for their immutability.

### Exercise 3 Syntactic Immutability Criteria

We define a class as immutable if all fields are private and methods of the class can only have read access to the fields. Compare this notion of immutability to the one in the lecture. Give syntactic criteria which guarantee immutability.

### Exercise 4 Immutability and `hashCode()`

When initializing object fields, which can only be done by computationally expensive means, it is useful only to compute them when their results are actually needed. Normally the result is then saved after it is used the first time. One possible usage scenario of the proposed solution is the computation of the result of the `hashCode()` method. Implement this method for an immutable class of your choice and make sure you do not break immutability.

## Exercise 5 Recursive Datatypes and Immutability

Consider the following datatype definition, which allows to create binary search trees:

```
datatype 'a tree = Leaf of 'a
                | Node of 'a tree * 'a * 'a tree
```

The following ML-function allows to insert values into a binary search tree:

```
fun insert smaller x (Leaf l) = if smaller (x, l)
                               then Node ((Leaf x), l, (Leaf l))
                               else Node ((Leaf l), x, (Leaf x))
  | insert smaller x (Node (t1, l, t2)) = if smaller (x, l)
                                           then Node (insert smaller x t1, l, t2)
                                           else Node (t1, l, insert smaller x t2)
```

- Implement immutable Java classes representing the tree datatype. Each datatype constructor should be realized by its own Java class.
- Add a method `insert(int x)` to each of the classes, respecting the immutability property. As we do not want to provide a `smaller` method to compare values of objects, we require that our type variable of the tree type is constrained to types implementing `Comparable`.
- Write a program, which creates a tree consisting only of a leaf with value 5 and insert the values 4, 7, 5 and 6. Draw the resulting object graph.