

Advanced Aspects of Object-Oriented Programming (SS 2011)

Practice Sheet 7 (Hints and Comments)

Exercise 1 Questions

Exercise 2 Aliasing

An alias is called dynamic, if the involved variables belong to the stack, e.g. local variables or method parameters.

- a) Desired dynamic alias: iterate through an array. The loop variable tmp aliases the array elements one after another.

```
class C {
    Object[] values;

    public void m () {
        for (Integer tmp : values) {
            ...
        }
    }
}
```

Undesired dynamic alias: Store an alias to an object in the local variable tmp; hideAll components and make the aliased one visible again. This breaks the invariant and is therefor an undesired alias.

```
class C {
    // invariant: either all components are visible or all are hidden.
    JComponent[] values;

    public void m () {
        JComponent tmp = values[0];
        hideAll();
        tmp.setVisible(true);
    }
    public void hideAll () {
        for (JComponent c : values) {
            c.setVisible(false);
        }
    }
}
```

- b) Aliasing means, that in a program state more than one reference to an object exists. Capturing occurs if a reference passed as a parameter to a method is stored in a field of the called object. If this is done, the reference remains available even after the method has been finished. I.e. Capturing happens, if a new static alias to a parameter object is established. Leaking means, that a reference to an object is passed outside. This means, that the caller of the leaking method can establish an alias to the leaked object.
- c) Instead of returning the reference to the original array, return a copy of the array. Depending on the characteristics of the type Identity and the security needs, it may be sufficient to return a new array, which references the original Identity objects, or one needs to return a deep copy of the array.

```
public Identity[] getSigners () {
    Identity[] signers = new Identity[this.signers.length];
    System.arraycopy(this.signers, 0, signers, 0, signers.length);
    return signers ;
}
```

Exercise 3 Immutable Classes in the JDK

- String: The best know immutable class of the JDK. In fact, it is not immutable with respect to the definition of the lecture, because it depends on the global setting for the current local.
- Number: The final classes implementing the abstract class Number. An object of Byte (Integer, etc) models exactly one value, which is not mutable. For the non-final ones a subtype could break immutability. Other wrapper classes like Character are good candidates for immutable classes as well.

- Classes of the reflection API: (Class, Package, etc) as long as the security manager is not changed.
- ...

A lot of classes are often used as if they are immutable, whereas technically they are not.

Exercise 4 Confined Types

	ConfinedList	ProofTreeNode	PTNIterator
C1	ok	ok	ok
C2	ok	ok	ok
C3	ok	see below	see below
C4	see below	ok	ok
C5	ok	ok	ok
C6	ok	ok	ok
C7	ok	ok	ok
C8	ok	ok	ok
Confined	no	no	no

C4 for ConfinedList ConfinedList inherits the method `iterator()`, which creates an object of the inner class `AbstractList.Itr` (in OpenJDK 1.6). This inner class captures the `this`-variable of `AbstractList`, the method `iterator()` can therefore not be anonymous, which breaks rule C4.

C3 for ProofTreeNode At line 19 in `ProofContainer` a `ProofTreeNode` is passed to a method that expects `Object`; at line 147 in `ProofTreeNode` widening to `Object` occurs.

C3 for PTNIterator In `ProofTreeNodeIt.getProofTreeNodes()` `PTNIterator` get widened to `Enumeration`.

Exercise 5 Encapsulation on Object Level

- No, the annotation is not possible due to the implementation of equals. Line 507: `co.data` accesses the data field but the referenced array is part of the representation of the list `co`. Representation objects are only accessible by the owner and by objects of the same representation domain, but the representation domain of the current `this` is different from the domain of the `co`.
- With alias modes it is possible to control the access to objects which belong to other objects. The alias modes work on object level, i.e. an alias mode is always relative to an object. This allows to hide the representation of one object to another one, even if they are created by the same source (i.e. they are of the same class). The confined types only allow to control that all interactions that may happen with object of a confined type are coded inside the package. This guarantees, that all modifications to these objects are under the control of the package developer. In the Confined Types approach, encapsulation is seen from a static point of view, whereas the alias modes (and ownership types in general) have a notion of encapsulation based on dynamic properties.