

Advanced Aspects of Object-Oriented Programming (SS 2015)

Practice Sheet 1 (Hints and Comments)

Exercise 1 Java Programming

Straightforward implementation.

Exercise 2 The equals Method

a) In the following d1 and d2 reference instances of the class Date and d3 and d4 reference instances of the class NamedDate. We have to check calls to equals with all combinations of types.

- *Reflexivity* d1.equals(d1) ok; d3.equals(d3) ok; the given implementation is reflexiv.
- *Symmetry* d1.equals(d2) \Rightarrow d2.equals(d1) ok
d3.equals(d4) \Rightarrow d4.equals(d3) ok
d1.equals(d3) \Rightarrow d3.equals(d1) ok
d3.equals(d1) \Rightarrow d1.equals(d3) ok

The given implementation is symmetric.

- *Transitivity* Counterexample:

```
d3 = new NamedDate(2010, 5, 7, 'A');  
d4 = new NamedDate(2010, 5, 7, 'B');  
d1 = new Date(2010, 5, 7);
```

d3.equals(d1) and d1.equals(d4) are true but d3.equals(d4); is false

The given implementation is not transitive.

- *Consistency* ok
- *Non-Null* ok

b) The contract is fulfilled except for the call with null (fix it, by adding a check) but this solution breaks Liskov's substitution principle. The principle states that whenever a property is true for an object of type T it should be true for objects of type S as well, where S is a subtype of T.

In other words, whenever a type T is used it should be safe, i.e. leading to the same results, to use a subtype of T. This is not the case with the given implementation. Consider the following code:

```
HashMap<Date, String> m = new HashMap<Date, String>();  
m.put(new NamedDate(2010,4,5, 'A'), 'rainy');  
// the following calls may give different results (depending on  
// which object equals is called and which is used as parameter),  
// where the substitution principle expects the same  
m.containsKey(new Date(2010,4,5));  
m.containsKey(new NamedDate(2010,4,5, 'A'));
```

Breaking the substitution principle often results in unexpected behavior and should be avoided if possible.

c) String-Objects in Java are interned. That is, string literals create a string object, but they create the same string object for the same literal. (JLS 3.10.5). This result is, that a1 and a2 reference the same object, whereas b1 and b2 both create new string objects at runtime. The two strings have the same value (namely "B") i.e. they are equal, but they have not the same identity.

Exercise 3 Happy Birthday

- a) The problem with the given implementation is, that the current age of a `AgePerson` is used in the method `hashCode`. That changes the hash of the instance stored in the `HashMap` of the `AgeManager`.

The documentation of the interface `Map` states:

”Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on a such a map.

- b) To get rid of these problems, don't use mutable keys in maps. In general it is a good idea not to use mutable data to calculate hashes, instead base the hash calculation for a class on its immutable attributes.
- c) In the method `AgeManager.add` a anonymous inner class is created. The parameter `birthday` is used in its initializer block. The Java-Specification states in that case, that this parameter has to be `final`. (see JLS 8.1.3). This is because the instance of the inner class must maintain a separate copy of the variable, as it may out-live the function, in which the instance is created.