Prof. Dr. A. Poetzsch-Heffter
Mathias Weber, M.Sc.

University of Kaiserslautern
Department of Computer Science
Software Technology Group

Advanced Aspects of Object-Oriented Programming (SS 2013)

Practice Sheet 3 (Hints and Comments)

Exercise 1 Reflection and Annotations

a) The class `java.lang.reflect.Proxy` allows you to create dynamic proxy instances which implement a list of interfaces specified at runtime. Statically, a class implementing this list of interfaces does not need to exist.

b) See enclosed source.

c) This technique only works for interface types.

Exercise 2 Inheritance and Subtyping

a) "In programming language theory, subtyping or subtype polymorphism is a form of type polymorphism in which a subtype is a datatype that is related to another datatype (the supertype) by some notion of substitutability, meaning that program constructs, typically subroutines or functions, written to operate on elements of the supertype can also operate on elements of the subtype. If S is a subtype of T, the subtyping relation is often written S <: T, to mean that any term of type S can be safely used in a context where a term of type T is expected." (Source: `http://en.wikipedia.org/wiki/Subtype_polymorphism`

The main purpose of inheritance is the reuse of code in different contexts. It makes implementing subtypes easier, because the subtype implementation only needs to describe the difference from the supertype, every unchanged behavior is given by the supertype implementation. If two classes are related with an inheritance relation, they also stand in a subtype relation.

b) The output is `woof woof`. The method `bark` is a static method, and therefore is bound statically. As `woofer` and `nipper` are variables of the static type `Dog`, the implementation in class `Dog` is chosen for calls.

Note: Java allows static methods to be called either on an object or directly on a class. In both cases they are statically bound, and thus the runtime type of the called object does not influence the method choice.

c) Before the Java 1.5 Specification was released, it was not possible to tell the compiler a method should override a method in the superclass. This lead to many mistakes where methods accidentally did not override the method of the superclass but instead added an overloaded method to the subclass.

When annotating a method with `@Override` this method must override a method of the superclass. If this is not the case, the java compiler will signal an error.

§15.12.4.4 of the Java Language Specification describes how methods are invoked at runtime. The method dispatch of `x.m` for a non-static method `m` and `T` is the static type of `x` starts by looking up the method in the runtime type of `x` called `S`. If `S` has an implementation of `m` and this implementation overrides the method `T.m` than this method is called. Otherwise the lookup procedes up the class hierarchy.

d) Yes, it does. This can be checked by adding the `@Override` annotation to the method. Note: `PhDAssistant` and `Person` are in the same package and widening of the accessibility (changing the modifier from default to `public`) is not affecting the subsignature relation between the two implementations. `Assistant` does not have a print method, but this does not change overriding, it just affects how (if possible) to call the overridden method.

Exercise 3 Super-Calls

```
public class A {
   public void m() { System.out.println(''A''); }
}
public class B extends A{
   public void m() { super.m(); }
}
```

```
public class C extends B{
   public static void main(String ... arg) {
      new C().m();
   }
}
```

With static binding new C().m() results in a call to the method m of class A. With dynamic binding it would end in an endless recursion, because the supertype of the current this would be B, so the call would again be dispatched to m in B.

Exercise 4 *StoJas* Extension

a) **Syntax:** Extend the syntax to include the `for` statement.

```
Statement = ...
          | for ( VarId = Exp; Exp; Statement ) { Statement } ;
```

Context Conditions: For `VarId = Exp` the context conditions used for normal assignments have to be fulfilled, especially the loop variable `VarId` has to be declared before the loop; the second expression (the termination expression) has to be of type boolean; the first statement is an arbitrary statement, usually this should be a statement that changes the loop variable, but we do not force it to be of that kind.

Static Semantics: No changes needed.

Dynamic Semantics: We add a new rule for the new statement. This rule expresses the semantics of the for statement as the semantics of an equivalent while statement.

$$\frac{\text{S : x = e1; while (e2) \{ s2; s1 \}} \rightarrow \text{SQ}}{\text{S: for (x = e1; e2; s1) \{ s2 \};} \rightarrow \text{SQ}}$$

b) **Syntax:** For the static methods we use a similar syntax as for the super calls and we extend the method declarations with the keyword static.

```
MethodDecl = ...
          | static TypeId MethodId(TypeId p) { Statement }
Statement = ...
          | VarId = ClassId@MethodId( Exp )  ;
```

Context Conditions: The statement of the method body is not allowed to use the variable `this` (neither reading nor writing).

Static Semantics: The set `Method` in the static semantics can now contain both static and non-static methods. The rest of the semantics does not change.

Dynamic Semantics: Same as the super call, as it is a statically bound call as well.

$$\frac{\text{S[p:=S(e), res:=init(rtyp(C@m))] : body(C@m)} \rightarrow \text{SQ}}{\text{S: x = C@m(e);} \rightarrow \text{S[x := SQ(res), \$:=SQ(\$)]}}$$

c) **Syntax:** Include static field declarations and statements to read and write the static fields.

```
FieldDecl = ...
          | static TypeId FieldId ;
Statement = ...
          | VarId = ClassId@FieldId  ;
          | ClassId@FieldId = Exp ;
```

Context Conditions: Anything that holds for non-static fields has to hold for static fields as well. (Type compatibility in assignments, etc.)

To implement the static fields, there are two possibilities. A) Model them as a part of the state, B) use a class-object and consider them to be the instance variables of the class object. We look at a version A here, for a solution using class objects you may refer to Exercise 4 of the year 2008.

Static Semantics: The state space of the program is enlarged. The state now contains not only instance variables `<c,o,f>` and the store but also the static fields. So first of all, we define the set of static fields, it is similar to the instance variable set, but as static fields belong to classes and not to instances, there is no object reference.

```
StatFields = { <c,f> | f is a static field of class c }
```

The state now is defined as

$$State = (VarId \cup \{\$\} \cup \{ClassId@FieldId\}) \rightarrow (Value \cup Store)$$

Dynamic Semantics: We introduce two new rules for read and write access to the static fields.

$$\frac{}{\text{S: x = C@f; } \rightarrow \text{S[x := S(C@f)]}} \qquad \frac{}{\text{S: C@f = e; } \rightarrow \text{S[C@f := S(e)]}}$$

When a static field changes, this change remains valid when returning from a method call. Same as a change in the store. Therefor we have to copy the changes originating of an execution of a method body to the calling state. The final states of the call rules therefor change to:

$$\text{S[x := SQ(res), \$:= SQ(\$), C@f := SQ(C@f) for all C@f]}$$

For static local variables (not supported by Java) the extension would look similar, just use `C@m@v` *for the variables and make sure, that they do not get reinitialized at each method call.*