Prof. Dr. A. Poetzsch-Heffter
Dipl.-Inform. Kathrin Geilmann

University of Kaiserslautern
Department of Computer Science
Software Technology Group

Advanced Aspects of Object-Oriented Programming (SS 2010)

Practice Sheet 5 (Hints and Comments)

Exercise 1 *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 contains 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:

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

Exercise 2 Covariance & Contravariance

Checked Exceptions have to be caught or declared to be thrown by a method.

a) With covariant exceptions try-catch blocks catch all thrown exceptions, even if a subtype of the catched exception is thrown. In a contra- or invariant scenario, the programmer has to update his catch statement every time the exception changes (e.g. due to an update in third party code). Depending on the context a program is used in, this would be uncomfortable or even impossible.

b) Checked exceptions guarantee at compile time, that they are handled by the program. Thus they do not lead to the abortion of the program. See JLS 11.2 for more about compile time checking of exceptions and the reasons for having unchecked exceptions as well.

Exercise 3 Generics

a)
```
static <A> void flip (Pair<A,A> p) {
        A tmp = p.fst;
        p.snd = p.fst;
        p.fst = tmp;
}
```

b)
```
class C implements Comparable<A> {

        public <T extends Comparable<A>> boolean equals (T o) {
                return this.compareTo(o) == o.compareTo(this);
                // depending on the type erasure and the dynamic dispatch,
                // this method might be called at lesser places than expected.
        }
}
```

c) The method can be written and compiled. But whenever you try to execute it, you will get a class cast exception. To implement such methods one can force the programmer to provide the array to the method. Even if it is not large enough one can use the given array to create a bigger one. Look at the Collection-API to see such methods.

```
public static <T> T[] toArray(List<T> l) {
        T[] res = (T[])(new Object[l.size()]);
        for (int i = 0; i < res.length; i++) {
                res[i] = l.get(i);
        }
        return res;
}
```

d) `Collection<?>` can hold a collection of anything, whereas `Collection<Object>` only holds collections to objects, i.e. `Collection<?> c = new ArrayList<String>();` is valid, wheras `Collection<Object> c = new ArrayList<String>();` is not (remember that generic types are invariant).

e) The output is `overloadedMethod?(Collection <?>)`, because overloading is resolved at compile time and at compile time the parameter t has type List<T>, such that the most specific method that can be applied is the one for the generic collection. (see JLS 15.12)