Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Prof. Dr. W.-P. de Roever
Harald Fecher, Martin Steffen

SIEGEL

Fool

Sommersemester 2005

Serie 1

2005

Thema: OO Intro (Aufgaben mit Lösungshinweisen)

Ausgabetermin: 2005

Abgabe:

Aufgabe 1 (Feature check-up) Find out, which (sub)-typing features we have discussed in the lecture can be found in *Java*:

- nominal/structural subtyping? both?
- multiple subtyping?
- late binding
- method specialization
  - on override
  - on inheritancce?
- Self-type specialization
- type-case?
- (bounded) type operators?
- F-Bounded polymorphism

Lösung:

Subtyping: Basically, *Java* uses nominal subtyping, for classes as well as for interfaces. This is rather understandable from the fact that *Java* uses the classes also as types at the same time, which makes in unavoidable for classes. But also for interfaces (in the sense of *Java*) which correspond roughly to the notion of object types that has been discussed in the lecture, *Java* uses nominal subtyping.

This does not mean that there are no "nominal" parts in the subtyping relation at all, of course, the *names* of the members (the fields and the methods) are compared by their names.

Multiple subtyping: As for multiple subtyping, the situation is as follows. First of all, there is no multiple *inheritance* (as for instance in C^{++}), it least not in the sense we use the word, i.e., inheritance means code reuse. In connection with the class types, there is in consequence no multiple subtyping. For interfaces, however, *Java* allows that one interface *extends* more than one interface (which must "contradict each" other in this case). So one can say that *Java* supports multiple subtyping for the interface types.

Late binding: Sure. Some other words (not 100% synonymous) for this feature are dynamic binding, dynamic dispatch or also (strange enough) message passing. It's such a typical feature of object-oriented languages, that without it one would not even call the language object-oriented without it.

Method specialization: *Java* is disappointingly weak there .... On the other hand it is flexible on method overloading, something which we haven't discussed much in the lecture. For instance, the code in Listing **??**, which is the simplest instance of method specialization on method override (co-variant in the return type), does not compile.

Listing 1: method specialization

```
// The following example tries  specialization on method override
class U1{}
class U2 extends U1{}

class T{
    public U1 m() {System.out.println("T.m"); return new U1();};
}

class S extends T {
    public U2 m() {System.out.println("S.m"); return new U2();};
}
```

Since specialization (in covariant position) of the self-type on override is even more specific than the specialization of the previous example it comes as no surprise that neither the Example of Listing **??** does not compile. Note also that the code is no example of self-type specialization, since the return type of the two methods is explicitely given as T, respectively S.

Listing 2: method specialization

```
// ``self-type'' specialization on method override?
class T{
    public T m() {System.out.println("T.m"); return this;
    };
}

class S extends T {
    public S m() {System.out.println("S.m"); return this;
    };
}
```

type case: not literally, but there is the *instanceof*-operator.

bounded type operators ... In *Java* 1.5, there are type operators, but we haven't tried out whether there's F-bounded polymorphism or something.

Aufgabe 2 Unter welchen Bedingungen ist $(A \to ((A \times B) \to C))$ ein Subtyp von $(C \to ((D \times A) \to B))$? Begründen Sie Ihre Aussage mittels den Regeln auf Seite 20f von [**?**].

Lösung: Geht bei Gleichheit?

Aufgabe 3 Was sind die Vorteile von 'embedding' (in objektbasierten Sprachen) gegenüber von 'delegation' und umgekehrt?

Lösung: Well, not too clear, which one "is better". One is more flexible, but perhaps more confusing or difficult to handle.