

Exercise Sheet 9

Assignment 9.1 Quiz

- Let C be a class, composed from the Mixins M and N . Suppose, M and N both implement a method $f()$. Is it true that
 - ☐ The conflicting methods $f()$ from M and N lead to a compiler error and have to be resolved manually
 - ☒ There is no compiler error, but one implementation of $f()$ from M or N overwrites the other
- Now assume, that M and N are Traits instead of Mixins. Is it true that
 - ☒ The conflicting methods $f()$ from M and N lead to a compiler error and have to be resolved manually
 - ☐ There is no compiler error, but one implementation of $f()$ from M or N overwrites the other
- (Attention: Several answers might be true for this question!)**
 $c_1 \sqcup c_2 = c_1 \uplus c_2$ is true for
 - ☒ $c_1 = \{a = 0x1\}, \quad c_2 = \{b = 0x1\}$
 - ☐ $c_1 = \text{mixin}(c_3)(c_2), \quad c_2 = \{a = 0x1\}, \quad c_3 = \{a = 0x2\}$
 - ☒ $c_1 = \text{mixin}(c_2)(c_3), \quad c_2 = \{a = 0x1\}, \quad c_3 = \{a = 0x2\}$
 - ☒ $c_1 = \text{mixin}(c_3)(c_4), \quad c_2 = c_3 \triangleright c_4, \quad c_3 = \{a = 0x1\}, \quad c_4 = \{a = 0x2\}$
- Why is exclusion an important composition operator for Traits?

Suggested Solution 9.1

Resolving conflicts is otherwise only possible via reimplementation of a method, leading to more redundant code. Consider the following Java 8 code as an example:

```
interface A {
    default int f() { return 0; }
}

interface B {
    default int f() { return 1; }
}

class C implements A,B {
    public int f() { return A.super.f(); }
}
```

Since Java 8 does not support exclusion, the method f in class C must be reimplemented.

Assignment 9.2 Having fun with Mixins

Reconsider the example from the lecture about synchronized file- and socket-streams. The following classes are given:

$$\begin{aligned}FileStream &= \{read = 0x1, write = 0x2\} \\SocketStream &= \{read = 0x3, write = 0x4\} \\SyncRW &= \{read = 0x5, write = 0x6\}\end{aligned}$$

Your task is to come up with a new class *SyncedFileStream* which mixes the class *SyncRW* into the class *FileStream*.

Suggested Solution 9.2

$$\text{mixin}(\text{SyncRW}) = \lambda x. \text{SyncRW} \triangleright x = \lambda x. \{super \mapsto x\} \sqcup (\text{SyncRW} \sqcup x)$$

Now define the actual new class:

$$\begin{aligned}\text{SyncedFileStream} &= \text{mixin}(\text{SyncRW}) \circ \text{FileStream} \\&= \{super \mapsto \text{FileStream}\} \sqcup (\text{SyncRW} \sqcup \text{FileStream}) \\&= \{super \mapsto \text{FileStream}\} \sqcup \{read = 0x5, write = 0x6\} \\&= \{super \mapsto \text{FileStream}, read = 0x5, write = 0x6\}\end{aligned}$$

Assignment 9.3 Mixins Ruby

Implement the *Stream Wrapper* scenario from the lecture based on Ruby Mixins

Suggested Solution 9.3

streams.rb

Assignment 9.4 Implementation differences: Traits vs. Mixins

A next mainstream implementation of traits comes with the virtual extension methods in Java 8.

- Implement a solution for the *Stream Wrapper* problem. You may use the following code:

```
interface Stream {
    int read();
}

interface FileStream extends Stream {
    default int read() { /* ... */ }
}

interface NetworkStream extends Stream {
    default int read() { /* ... */ }
}
```

```

interface Synch {
    default void acquireLock() { /* ... */ }
    default void releaseLock() { /* ... */ }
}

```

- Compare your solution to the one based on Mixins from the above assignment. What are the differences? Which one is more flexible w.r.t. software engineering aspects?

Suggested Solution 9.4

Traits.java

There are two solutions to come up with something like the Synch-Mixin:

- Either statically generate particular wrapper interfaces to forward statically to the right parent (as done in the `SynchFileStream`).
- Or generate a dynamic wrapper interface which opens up a new abstract method, which has to be dispatched to the right dynamic predecessor (kind of imitated via the interfaces `SynchWrapper` and `SynchNetworkStream`)

If we compare these solutions via Traits with the solution based on Mixins from above, we see that the overwriting of method `read()` has to be done explicitly, if we resort to Traits, where as with Mixins, we automatically overwrite. Mixins further offer the possibility to treat the parent class as of variable type, while in Java's *Virtual Extension Methods*, we cannot do so.