DHBW Stuttgart

Java-TX Compiler in Java-TX

Julian Schmidt

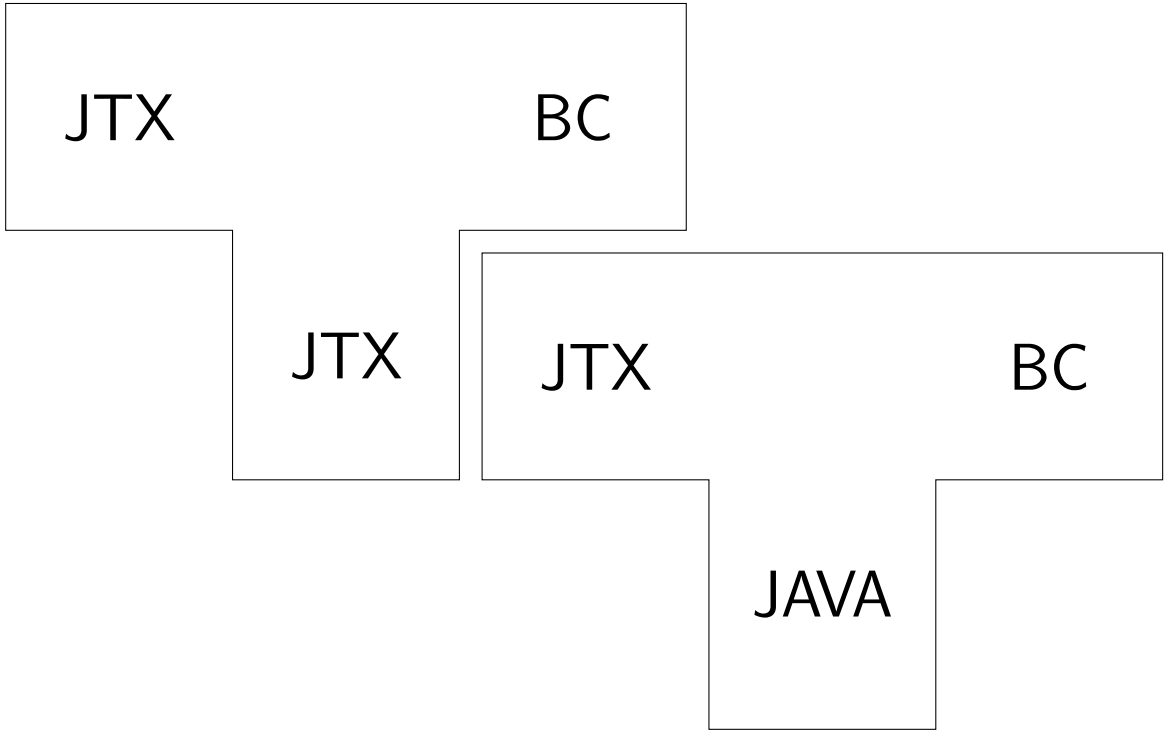
Agenda

1. Motivation
2. Aufbau der Umgebung
3. Bugs/Probleme
   1. Überschreiben von Methoden
   2. Kompatibilität mit funktionalen Interfaces
4. Fazit

Motivation I

- Welche Features fehlen noch in Java-TX?
- Welche Bugs gibt es?
- Wie performant is Java-TX für größere Projekte?
- Vorteile/Nachteile zu Java in der Praxis

Motivation II



Vergleich Sourcecode

```
1  public class FunNClass extends ClassOrInterface
     {
2    private static GenericDeclarationList
       createGenerics(List<GenericRefType>
       funNParams) {
3      var generics = new ArrayList<GenericTypeVar
         >();
4      for (GenericRefType param : funNParams) {
5        generics.add(...);
6      }
7      return new GenericDeclarationList(generics,
         new NullToken());
8    }
9  }
```

```
1  public class FunNClass extends ClassOrInterface
     {
2    private static createGenerics(funNParams) {
3
4
5      var generics = new ArrayList<GenericTypeVar
         >();
6      for (GenericRefType param : funNParams) {
7          generics.add(...);
8      }
9      return new GenericDeclarationList(generics,
         new NullToken());
10   }
11 }
```

Vergleich Sourcecode

```
1 public class FunNClass extends ClassOrInterface
    {
2   private static GenericDeclarationList
      createGenerics(List<GenericRefType>
      funNParams) {
3     var generics = new ArrayList<GenericTypeVar
        >();
4     for (GenericRefType param : funNParams) {
5       generics.add(...);
6     }
7     return new GenericDeclarationList(generics,
        new NullToken());
8   }
9 }
```

```
1  public class FunNClass extends ClassOrInterface
     {
2    private static GenericDeclarationList
       createGenerics(Iterable<? extends
       GenericRefType> var0) {
3      List var1 = null;
4      var1 = (List)(new ArrayList());
5      Iterator var10000 = var0.iterator();
6      while(var10000.hasNext()) {
7        GenericRefType var2 = (GenericRefType)
           var10000.next();
8        var1.add(...);
9      }
10     return new GenericDeclarationList(var1, new
         NullToken());
11   }
12 }
```

Aufbau der Umgebung I

1. Compiler soll sukzessive in Java-TX umgeschrieben werden
2. Umgebung mit .java und .jav Dateien
3. Ziel: Auf JVM ausführbare .class Dateien
4. Java-TX Compiler kann .java Dateien lesen
   $\rightarrow$ Abhängigkeiten zu Java Dateien möglich
5. Java-TX Compiler muss vor javac aufgerufen werden

Aufbau der Umgebung II - make

Erster Ansatz mit make:

```
1   # Use find to locate all .java and .jav files recursively
2   JAVASOURCES := $(shell find $(SRCDIR) -name '*.java')
3   JAVSOURCES := $(shell find $(SRCDIR) -name '*.jav')
4
5   # Convert .java/.jav files to .class files with the same directory structure
6   JAVACLASSES := $(patsubst $(SRCDIR)/%.java,$(DESTDIR)/%.class,$(JAVASOURCES))
7   JAVCLASSES := $(patsubst $(SRCDIR)/%.jav,$(DESTDIR)/%.class,$(JAVSOURCES))
8
9   # Rule for compiling .jav files
10  $(DESTDIR)/%.class: $(SRCDIR)/%.jav
11  java -jar $(JTX) -d "$(dir $@)" -cp "$(SRCDIR):$(DESTDIR):target/dependencies/" $<
12  # Rule for compiling .java files
13  $(DESTDIR)/%.class: $(SRCDIR)/%.java
14  $(JC) -nowarn -d $(DESTDIR) -cp "$(SRCDIR):$(DESTDIR):target/dependencies/*" $(JFLAGS) $<
```

Aufbau der Umgebung III

Probleme:

1. javac compiliert und trackt Änderungen der Abhängigkeiten automatisch
2. javac ist sehr langsam wenn für jede Datei einzeln aufgerufen (viele mehrfache Compilierungen)

```
1 javac src/main/java/de/dhbwstuttgart/
    typedeployment/TypeInsert.java
2 javac src/main/java/de/dhbwstuttgart/
    typedeployment/TypeInsertPlacer.java
3 ...
4 javac src/main/java/Main.java
```

$\sim$ 5min Compilerzeit

```
1 javac src/main/java/de/dhbwstuttgart/
    typedeployment/TypeInsert.java src/main/java/
    de/dhbwstuttgart/typedeployment/
    TypeInsertPlacer.java ... src/main/java/Main.
    java
```

$\sim$ 2sec Compilerzeit

Aufbau der Umgebung IV - compile script

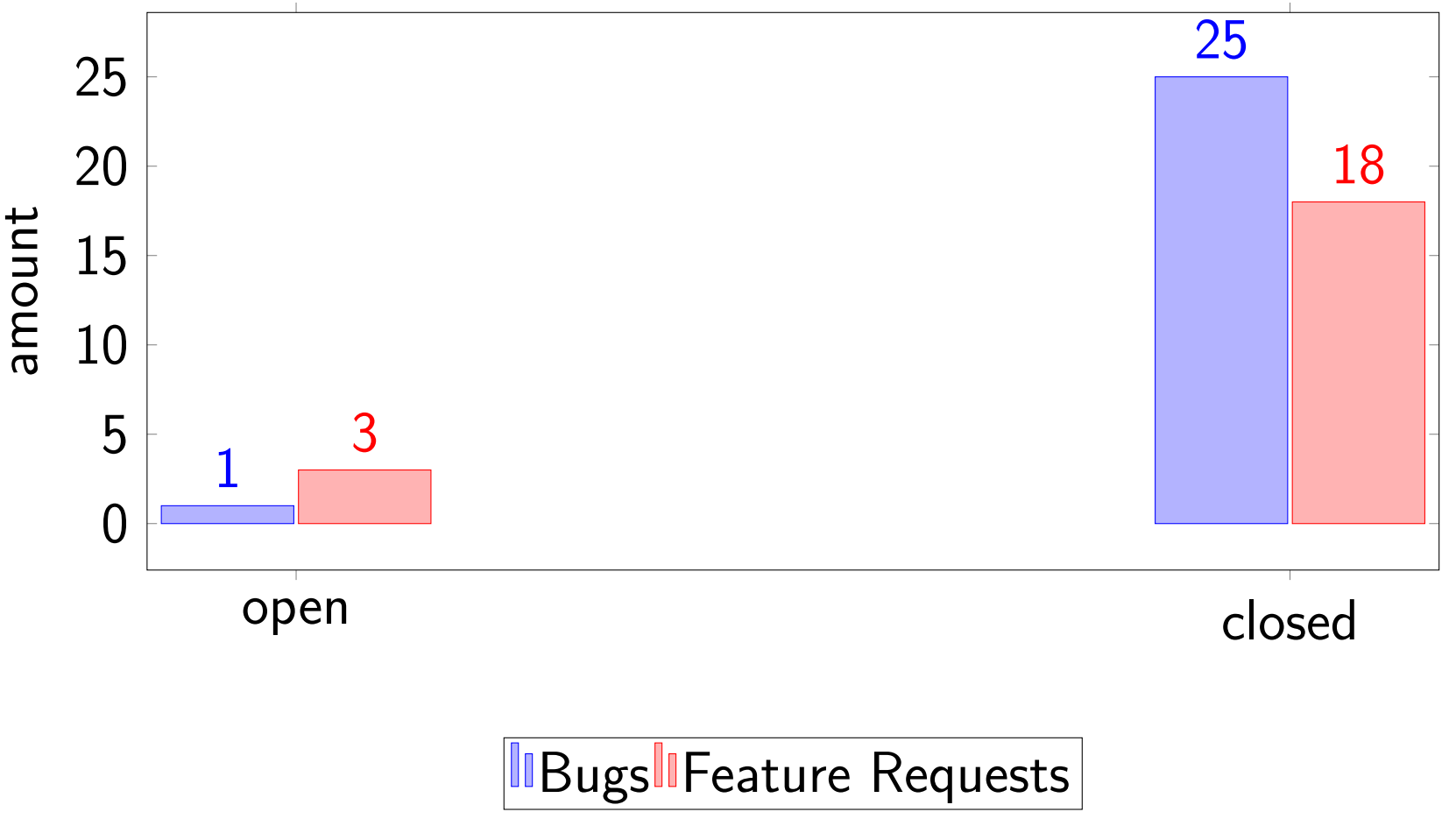
Gegeben: Quellverzeichnis, Zielverzeichnis

1. Suche rekursiv alle .java und .jav Dateien im Quellverzeichnis und speichere sie jeweils in einer Liste
2. Überprüfe für jede Quelldatei, ob die zugehörige .class Datei im Zielverzeichnis existiert und ob die Zieldatei neuer als die Quelldatei ist
   - Wenn ja, gehe weiter zur nächsten Datei
   - Wenn nein, füge die Quelldatei zur Liste der zu kompilierenden Dateien hinzu
3. Rufe den Java-TX Compiler mit allen Dateien in der jav-Liste als Argumente auf

   ```
   java -jar $JAVATX_COMPILER_PATH -d $DESTDIR -cp "$SRCDIR:$DESTDIR:target/dependencies/" "${JAV_CHANGED[@]}"
   ```
4. Rufe den javac Compiler mit allen Dateien in der java-Liste als Argumente auf

   ```
   javac -d $DESTDIR -cp "$SRCDIR:$DESTDIR:target/dependencies/*" $JAVAC_FLAGS "${JAVA_CHANGED[@]}"
   ```

Bugübersicht



Primitive Typen in Java-TX

- Java erlaubt neben Referenztypen auch primitive Datentypen (int, boolean, ...)
- Java-TX erlaubt primitive Datentypen zwar im Quellcode, wandelt diese aber in die korrespondierende Wrapperklasse um (Integer, Boolean, ...)

```
1 int a = 10;
2 boolean b = true;
3 float c = 10.0f;
```

```
1 Integer var1 = null;
2 var1 = 10;
3 Boolean var2 = null;
4 var2 = true;
5 Float var3 = null;
6 var3 = 10.0F;
```

- Generell in Java: Methoden nicht anhand von Rückgabewert überschreibbar

```
1 public class Bar {
2   int foo(Object obj){return 0;}
3   boolean foo(Object obj){return false;}
4 }
```

```
1 Bar.java:3: Fehler: Methode foo(Object) ist bereits in Klasse Bar definiert
2     boolean foo(Object obj){return false;}
3             ^
```

  - Aber: Generell auf JVM lauffähig

Überschreiben von Methoden II

- Überschreiben von Java Methoden mit primitiven Datentypen als Parameter funktionierte nicht

```
public boolean equals(Object obj);
```

```
//Java-TX Code
import java.lang.Object;
import java.lang.Boolean;
public class Foo {
   equals(Object o){
        return false;
   }
}
```

```
//Inferierte Typen
public class Foo {
    public Foo() {}
    Boolean equals(Object var1) {
        return false;
    }
}
```

Überschreiben von Methoden III

- Lösung: Wenn Methodensignatur eines Supertyps sich nur in primitiven-/Wrapper-Datentypen unterscheidet, werden Typen vom Supertyp in Subtyp substituiert

```
1 import java.lang.Object;
2 import java.lang.Boolean;
3
4 public class Foo {
5    equals(Object o){
6         return false;
7    }
8 }
```

```
1 public class Foo {
2     public Foo() {}
3     boolean equals(Object var1) {
4         return false;
5     }
6 }
```

Kompatibilität mit funktionalen Interfaces I

- In Java haben Lambda Ausdrücke als Target Type ein funktionales Interface z.B. java.util.function.Function

```
1 Function<Integer, Integer> func = x -> x*2;
```

- Java-TX unterstützt echte Funktionstypen

```
1 var func = x -> x*2;
2 func: Fun1$$Ljava$lang$Integer$_$Ljava$lang$Integer$_$
```

- Für Kompatibilität müssen Funktionstypen mit Target Typen integriert werden

Kompatibilität mit funktionalen Interfaces II

- Problem: Lambda Ausdrücke haben in Java-TX einen FunN$$ Typ (für N = #Parameter)
- Aber: Java Bibliotheken wie Stream erwarten verschiedene funktionale Interfaces
- Lösung: Lambda Ausdrücke müssen je nach Kontext erwartetes funktionales Interface als Target Typ haben

Beispiel:

```
1  import java.util.function.Function;
2  import java.util.stream.Steam;
3  import java.util.List;
4  import java.util.ArrayList;
5
6  public class Main {
7      main() {
8        List<Integer> list = new ArrayList<>(List.of(1,2,3,4,5));
9        return list.stream().map(x -> x*2).toList();
10     }
```

Kompatibilität mit funktionalen Interfaces III

```
1 <R> Stream<R> map(Function<? super T, ? extends R> var1);
```

```
1 public interface Function<T, R> {
2    R apply(T var1);
3 }
```

- Eigentlich würde Java-TX Fun1$$ $\langle Integer, Integer \rangle$ inferieren
- Aber: Stream.map erwartet $Function\langle ?superT, ?extendsR \rangle$

```
1 ...
2 3: invokedynamic #41, 0              // InvokeDynamic #0:apply:(LFoo;)
    LFun1$$Ljava$lang$Integer$_$Ljava$lang$Integer$_$;
3 //Invalid Bytecode
4 5: invokeinterface #45, 2            // InterfaceMethod java/util/stream/Stream.map:(Ljava/util/
    function/Function;)Ljava/util/stream/Stream;
5
6 ...
```

Kompatibilität mit funktionalen Interfaces IV

```
1 ...
2 50: invokedynamic #60,  0              // InvokeDynamic #0:apply:(LFoo;)Ljava/util/function/
    Function;
3 55: invokeinterface #66,  2            // InterfaceMethod java/util/stream/Stream.map:(Ljava/util
    /function/Function;)Ljava/util/stream/Stream;
4 ...
```

Fazit

Vorteile:

- Programmierer muss weniger Typen explizit angeben
- Funktionstypen erlauben übersichtlichere Subtypisierung von anonymen Funktionen als Java

Nachteile:

- Alle verwendeten/berückstichtigten Typen müssen manuell importiert werden
  $\rightarrow$ Der Programmierer muss schon wissen, welche Typen in Frage kommen
- Es ist möglich, dass ein ungeschünschter Typ inferiert wird
- Aktuell begrenzte Sprachfeatures & vermutlich einige Bugs