



Aufbau des Compilers

AwesomeJavaCompiler

Nils Olbert, Tim Röthel, Julian Zepf,
David Deutsch, Lukas Lautenschlager, Daniel Bornbaum, Nico Dreher

Exportiert am: 03.06.2019

Version: 12

Inhaltsverzeichnis

1 ASTGenerator	4
2 ByteCodeGenerator:	7
3 Common	8
4 TAST-Generator.....	9
5 Test	10
6 Benutzung des Compilers.....	11

Der Compiler besteht aus 4 großen Packages, welche die einzelnen Komponenten der Software abbilden:

- ASTGenerator: Komponente, welche für den Aufbau des abstrakten Syntaxbaums zuständig ist.
- ByteCodeGenerator: Erzeugt aus dem getypten, abstrakten Syntaxbaum den Bytecode.
- Common: Stellt die zentrale Komponente des Compilers dar. Hier werden alle gemeinsam genutzten Ressourcen (Bsp. Enumeration für AccessModifiers), sowie Interfaces für die anderen Komponenten bereitgestellt.
- TASTGenerator: Erzeugt aus dem abstrakten Syntaxbaum einen getypten abstrakten Syntaxbaum.

Zusätzlich zu den 4 Hauptkomponenten gibt es noch das Test Package, welches umfangreiche Tests für die einzelnen Komponenten, sowie Integrationstests bereitstellt.

Zur Implementierung der einzelnen Komponenten wurde das Visitor Pattern genutzt.

Im Folgenden werden die einzelnen Packages nochmals genauer erläutert. Auch die Verantwortlichkeiten wurden in den Beschreibung festgehalten. Das Package Common und die Packages Expressions, GeneralElements und Statement des AST- und TASTGenerator wurden von allen Projektbeteiligten gemeinsam entwickelt und von den Verantwortlichen modifiziert.

1 ASTGenerator

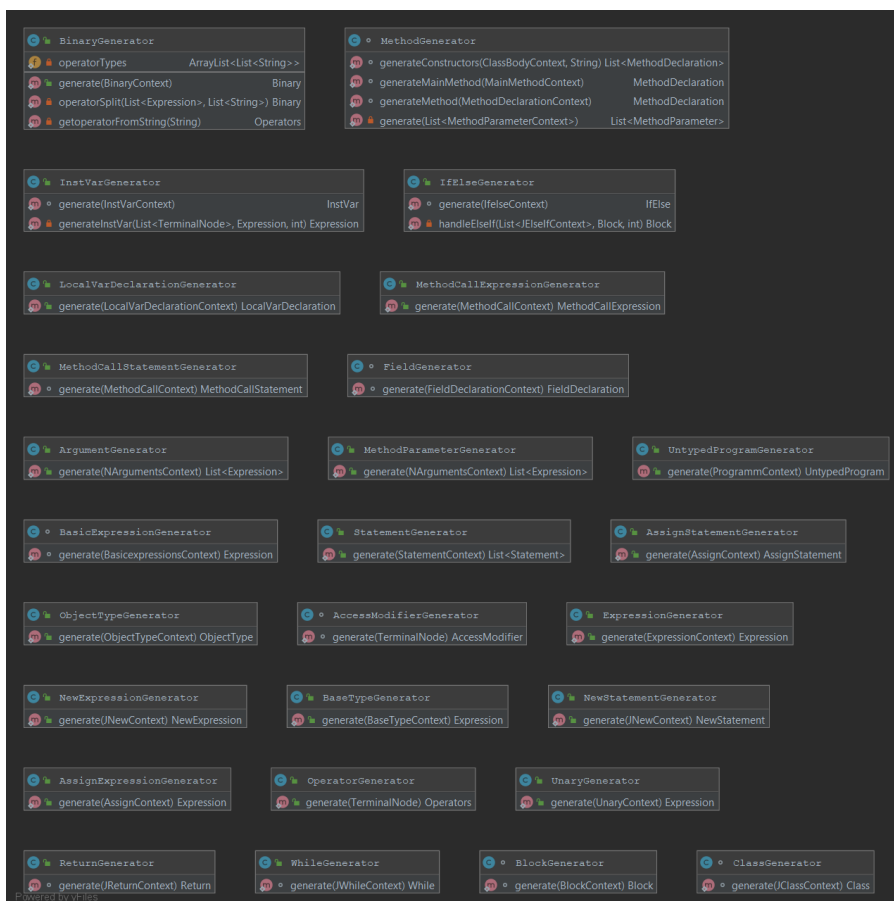
Hauptverantwortliche: Daniel Bornbaum und David Deutsch

Die Komponente stellt den Parser, sowie den Aufbau des abstrakten Syntaxbaums zu Verfügung. Als Eingabe dient hierzu ein InputStream, in welchen die zu kompilierende Java Klasse übergeben wird. Dieser Input Stream wird erst mit ANTLR in einen Parsebaum übersetzt, aus welchen dann der abstrakte Syntaxbaum aufgebaut wird.

Das Package astgenerator wird hierbei in 4 Teilbereiche untergliedert:

- Expressions: Enthält alle Klasse die einen Ausdruck abbilden.
- GeneralElements: Enthält Abbildungen der allgemeinen Struktur eines Java Programms (UntypedProgram, Class, FieldDeclaration, MethodDeclaration).
- Parser: Enthält die G4-Grammatik, sowie die erzeugten Dateien von ANTLR und die Generator-Klassen, die den AST erzeugen.
- Statement: Enthält alle Klassen zur Abbildung von Anweisungen.

Im Folgenden wird noch der Aufbau der Komponente mittels UML-Klassendiagramm eingegangen. Hierbei wurde nur die Teilkomponente des Package Parser betrachtet, welche für den Aufbau des abstrakten Syntaxbaum benötigt wird. Hierbei wird für die einzelnen Instanzen ein Generator zur Verfügung gestellt, aus welchem sich später der abstrakte Syntaxbaum ergibt. Mit der Generate Methode werden dann die von ANTLR zur Verfügung gestellte Parser Klasse genutzt (Klassenname: AwsomeJavaParser).



ANTLR benötigt zur Generierung eines Parsers eine Grammatik. Der Aufbau unserer Grammatik, welche für unseren Compiler entwickelt wurde, hat folgenden Aufbau:

Grammatik Awsome Java Compiler

```
1  grammar awesomeJava;
2
3  programm: jclass+;
4  jclass: 'class' Identifier classBody;
5  constructor: AccessModifier? Identifier LBracket nMethodParameters
   RBracket block;
6  classBody: CurlyLBracket (methodDeclaration|fieldDeclaration|constructor)*
   CurlyRBracket;
7  methodDeclaration: AccessModifier? (objectType|Void) Identifier LBracket
   nMethodParameters RBracket block;
8  fieldDeclaration: AccessModifier? objectType Identifier (Comma
   Identifier)* (Equal expression)? Semicolon;
9  methodParameter: objectType Identifier;
10 nMethodParameters: (methodParameter)? | methodParameter (Comma
   methodParameter)+;
11
12 nArguments: expression? | expression (Comma expression)* | instVar;
13 expression: basicexpressions | binary;
14 basicexpressions: baseType | instVar | Identifier | statementExpressions
   | unary | LBracket expression RBracket ;
15 instVar: This Dot Identifier|(This Dot)? (Identifier Dot)+ Identifier;
16 statementExpressions: assign | jNew | methodCall;
17 assign: (instVar | Identifier) (Equal|PlusEqual|MinusEqual) expression;
18 localVarDeclaration: objectType Identifier (Comma Identifier)* ((Equal|
   PlusEqual|MinusEqual) expression)?;
19 jNew: 'new' Identifier LBracket nArguments RBracket;
20 methodCall: methodCallPrefix? (Identifier LBracket nArguments RBracket
   Dot)*
21 (Identifier LBracket nArguments RBracket);
22 methodCallPrefix: (instVar|Identifier Dot);
23 statement: ifelse | localVarDeclaration Semicolon | jReturn Semicolon |
   jWhile | block
24 | statementExpressions Semicolon;
25 block: CurlyLBracket (statement)* CurlyRBracket;
26 ifelse: jIf jElseIf* jElse?;
27 jIf: If LBracket expression RBracket block;
28 jElseIf: Else If LBracket expression RBracket block;
29 jElse: Else block;
30 jWhile: 'while' LBracket expression RBracket block;
31 jReturn: 'return' expression;
32 unary: NotOperator expression;
33 binary: basicexpressions (operators basicexpressions)+;
34 baseType: JBoolean | JNull | This | JString | JCharacter | JInteger |
   Super;
35 objectType: 'int'|'char'|'boolean'|Identifier;
36 operators: LogicalOperator|Comperator|AddSubOperator|PointOperator;
37
38 AccessModifier: 'public' | 'protected' | 'private';
```

```
39 JBoolean: 'true'|'false';
40 JNull: 'null';
41 Void: 'void';
42 Super: 'super';
43 This: 'this';
44 If: 'if';
45 Else: 'else';
46 Equal: '=';
47 PlusEqual: '+=';
48 MinusEqual: '-=';
49 Comperator: '=='|'!='|'>='|'<='|'>'|'<';
50 NotOperator: '!';
51 OpBeforeOrAfterIdentifier: '++'|--';
52 PointOperator: '*'|'/';
53 AddSubOperator: '+'|'-'|'%';
54 LogicalOperator: '&&'||'|';
55 LBracket:'(';
56 RBracket:')';
57 CurlyLBracket:'{';
58 CurlyRBracket:'}';
59 Dot: '.';
60 Comma: ',';
61 Semicolon: ';';
62 Identifier: [A-Za-z][A-Za-z0-9]*;
63 JCharacter: '\\' [A-Za-z] '\\';
64 JInteger: [0-9]+;
65
66 WS: ([ \t\r\n]+) -> skip;
67 Comment: '/*' .*? '*/' -> skip;
68 LineComment: '//' ~[\r\n]* -> skip;
```

2 ByteCodeGenerator:

Hauptverantwortlicher: Nico Dreher

Die Komponente stellt den Teil des Compilers zu Verfügung, welcher aus dem getypten, abstrakten Syntaxbaum den Bytecode generiert.

Als Eingabe erhält diese ein TypedProgram (Abbildung des getypten abstrakten Syntaxbaums).

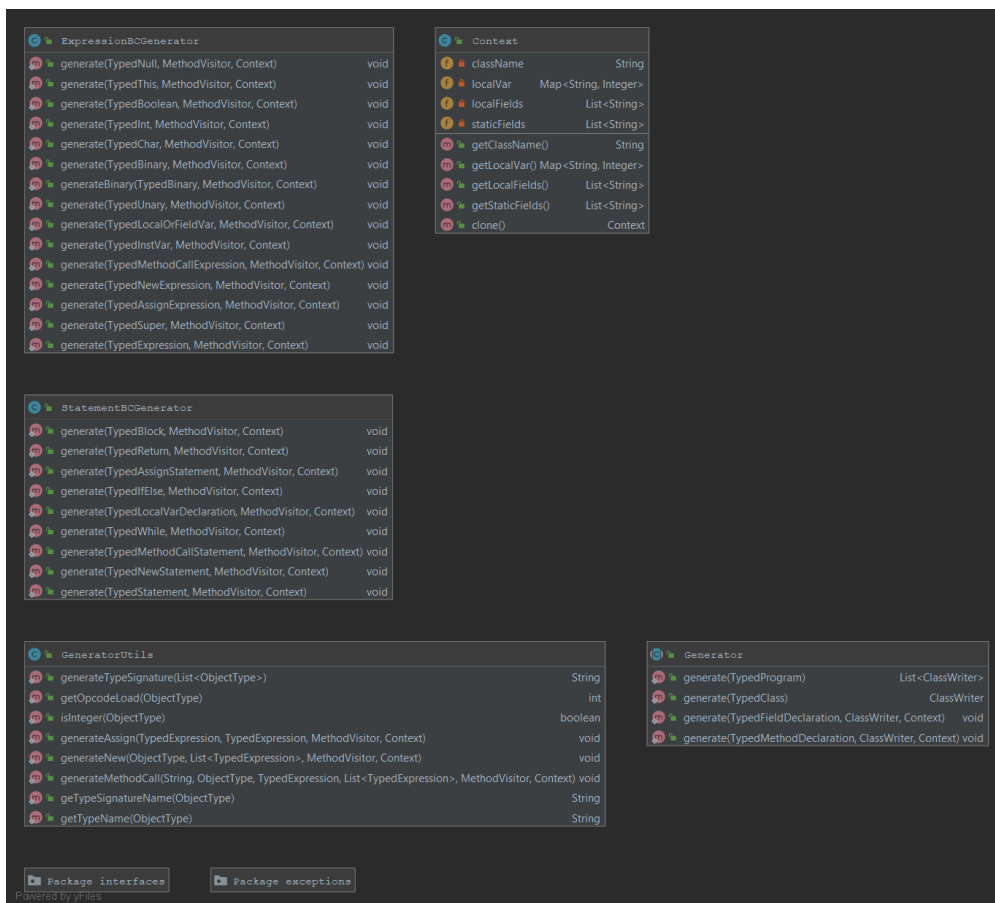
Dieser Syntaxbaum wird dann von der Komponente in ByteCode mit Hilfe von ASM umgewandelt.

Das folgende Bild beschreibt den Aufbau des ByteCodeGenerators in Form eines UML-Klassendiagramms.

Der wichtigste Teil ist der Generator selbst, welcher überwiegend aus der generate Methode besteht.

Diese Methode wird hierbei für die unterschiedlichen Objekte des getypten, abstrakten Syntaxbaum überschrieben und erstellt anhand dessen den ByteCode, welcher durch die JVM ausführbar ist.

Die Context Klasse unterstützt hierbei, um sichtbare Variablen im aktuellen Bereich zu speichern.

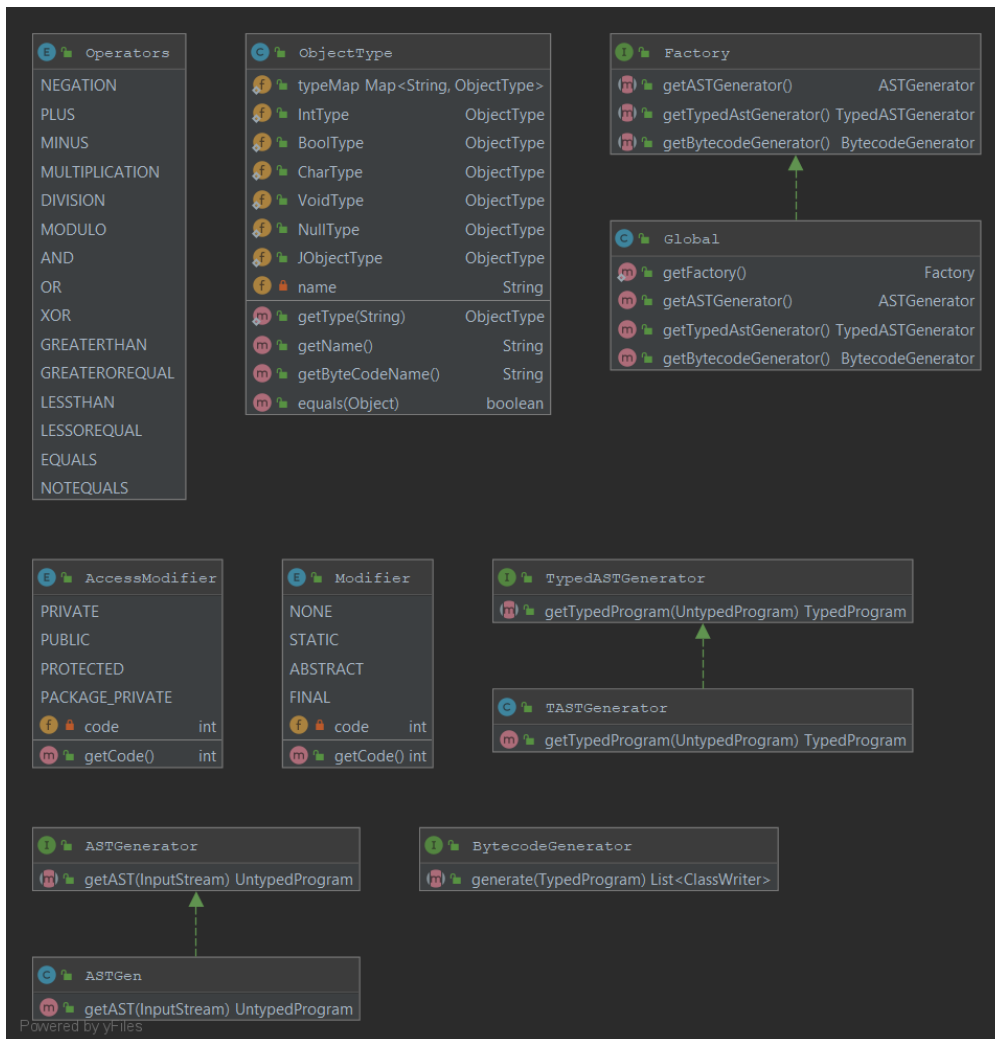


3 Common

Enthält allgemein benötigte Klassen und stellt die zentrale Komponente der Software dar.

In diesem Package befinden sich:

- Implementierungen der Schnittstellen der einzelnen Komponenten.
- Allgemeingültige Klassen, wie Enumerationen für Operatoren, AccessModifiers, etc.



5 Test

Hauptverantwortliche: Tim Röthel und Nils Olbert

Im Test Package befinden sich Tests und Testdaten für den Compiler.

In diesem befinden sich weitere Packages für die einzelnen Komponenten (bsp. ASTTests)

Hier finden sich Tests für die einzelnen Komponenten.

Weiterhin gibt es auch ein Package für Integrationstests, welches einen Test durchführt, der überprüft ob der Compiler eine Klasse kompilieren kann.

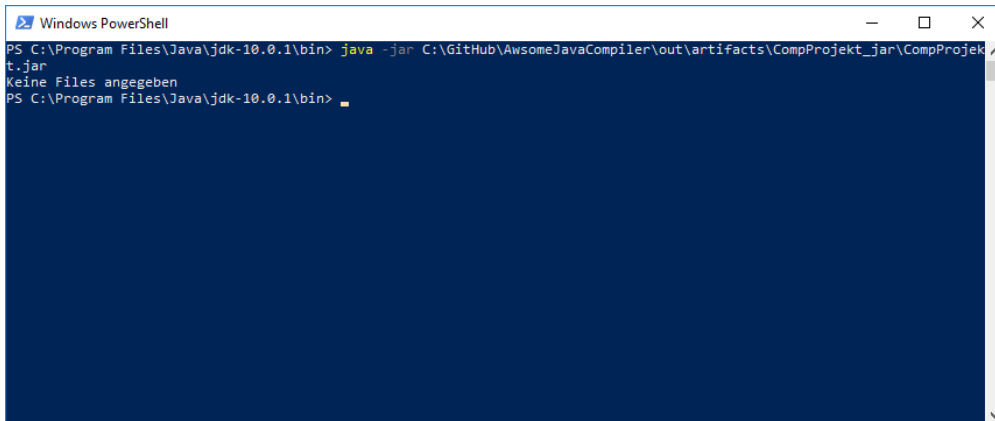
Die Testdaten umfassen auch alle Klassen, für welchen der Compiler funktioniert. Unter den Integrationstests sind auch durchgehende Beispiele vorhanden.

6 Benutzung des Compilers

Zur Benutzung des Compilers wird eine JAR bereitgestellt.

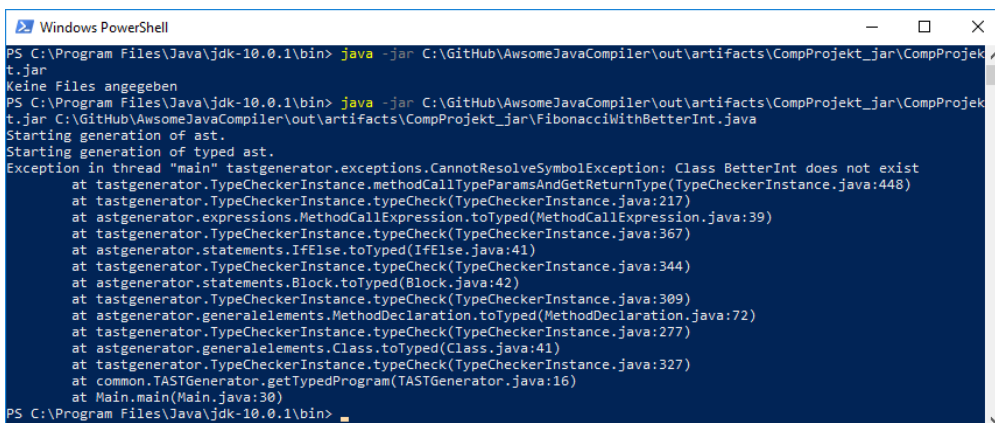
In den folgenden Beispielen wird beschrieben, wie der Compiler benutzt werden kann. Als Parameter werden die Dateipfade zu den Klassen übergeben.

Aufruf ohne Parameter:



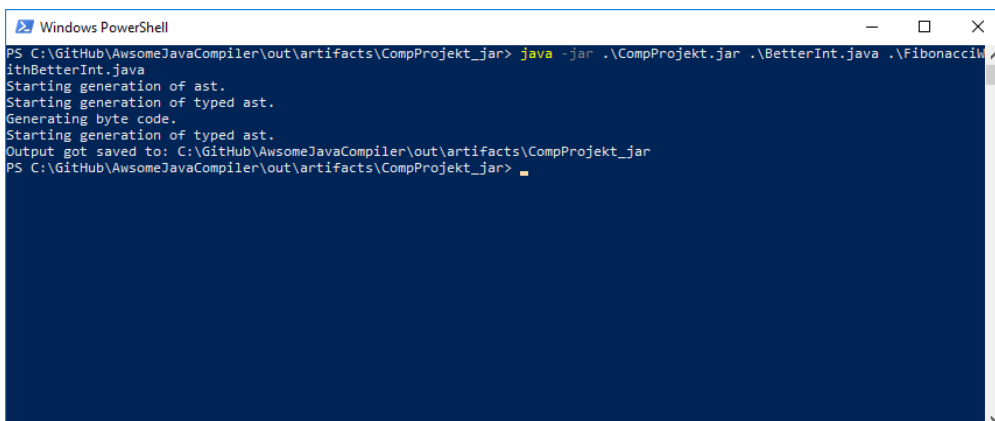
```
Windows PowerShell
PS C:\Program Files\Java\jdk-10.0.1\bin> java -jar C:\Github\AwesomeJavaCompiler\out\artifacts\CompProjekt_jar\CompProjekt.jar
Keine Files angegeben
PS C:\Program Files\Java\jdk-10.0.1\bin>
```

Fehlgeschlagene Kompilierung:



```
Windows PowerShell
PS C:\Program Files\Java\jdk-10.0.1\bin> java -jar C:\Github\AwesomeJavaCompiler\out\artifacts\CompProjekt_jar\CompProjekt.jar C:\Github\AwesomeJavaCompiler\out\artifacts\CompProjekt_jar\FibonacciWithBetterInt.java
Keine Files angegeben
PS C:\Program Files\Java\jdk-10.0.1\bin> java -jar C:\Github\AwesomeJavaCompiler\out\artifacts\CompProjekt_jar\CompProjekt.jar C:\Github\AwesomeJavaCompiler\out\artifacts\CompProjekt_jar\FibonacciWithBetterInt.java
Starting generation of ast.
Starting generation of typed ast.
Exception in thread "main" tastgenerator.exceptions.CannotResolveSymbolException: Class BetterInt does not exist
    at tastgenerator.TypeCheckerInstance.methodCallTypeParamsAndGetReturnType(TypeCheckerInstance.java:448)
    at tastgenerator.TypeCheckerInstance.typeCheck(TypeCheckerInstance.java:217)
    at tastgenerator.expressions.MethodCallExpression.toTyped(MethodCallExpression.java:39)
    at tastgenerator.TypeCheckerInstance.typeCheck(TypeCheckerInstance.java:367)
    at tastgenerator.statements.IfElse.toTyped(IfElse.java:41)
    at tastgenerator.TypeCheckerInstance.typeCheck(TypeCheckerInstance.java:344)
    at tastgenerator.statements.Block.toTyped(Block.java:42)
    at tastgenerator.TypeCheckerInstance.typeCheck(TypeCheckerInstance.java:309)
    at tastgenerator.generalelements.MethodDeclaration.toTyped(MethodDeclaration.java:72)
    at tastgenerator.TypeCheckerInstance.typeCheck(TypeCheckerInstance.java:277)
    at tastgenerator.generalelements.Class.toTyped(Class.java:41)
    at tastgenerator.TypeCheckerInstance.typeCheck(TypeCheckerInstance.java:327)
    at common.TASTGenerator.getTypedProgram(TASTGenerator.java:16)
    at Main.main(Main.java:30)
PS C:\Program Files\Java\jdk-10.0.1\bin>
```

Erfolgreiche Kompilierung:



```
Windows PowerShell
PS C:\Github\AwesomeJavaCompiler\out\artifacts\CompProjekt_jar> java -jar .\CompProjekt.jar .\BetterInt.java .\FibonacciWithBetterInt.java
Starting generation of ast.
Starting generation of typed ast.
Generating byte code.
Starting generation of typed ast.
Output got saved to: C:\Github\AwesomeJavaCompiler\out\artifacts\CompProjekt_jar
PS C:\Github\AwesomeJavaCompiler\out\artifacts\CompProjekt_jar>
```

Kompilierung mit Ausführung (mittels einer eigenen Klasse mit Main-Methode):

```
public class Test
{
    public static void main(String[] args)
    {
        var x = new BetterInt(11);
        System.out.println(new Fibonacci().fibonacci(x).x);
    }
}
```



```
PS C:\Program Files\Java\jdk-10.0.1\bin> .\javac.exe -cp "C:\Git\AwesomeJavaCompiler\out\artifacts\CompProjekt.jar" C:\Git\AwesomeJavaCompiler\out\artifacts\CompProjekt.jar\Test.java
PS C:\Program Files\Java\jdk-10.0.1\bin> .\java.exe -cp "C:\Git\AwesomeJavaCompiler\out\artifacts\CompProjekt.jar" Test
89
PS C:\Program Files\Java\jdk-10.0.1\bin>
```