Attributierte Grammatik für Java für Workgroups

```
Program -> (Class)*
        Program.classes = concatAll($Class_i$.node)
Class -> class IDENT { (Field | Method | MainMethod)* }
        Class.node = new Class(
                name = IDENT.lexval,
                fields = concatAll($Field_i$.node),
                methods = concatAll($Method_i$.node),
                mainMethods = concatAll($MainMethod_i$.node)
        )
Field -> public Type IDENT ;
        Field.node = new Field(type=Type.node, name=IDENT.lexval)
Method -> public Type IDENT ( Parameters ) Block
        Method.node = new Method(
                retType = Type.node,
                name = IDENT.lexval,
                params = Parameters.node,
                block = Block.node
        )
MainMethod -> public static void $IDENT_1$ ( String [ ] $IDENT_2$ ) Block
        MainMethod.node = new MainMethod(
                name = $IDENT_1$.lexval,
                argName = $IDENT_2$.lexval,
                block = Block.node
        )
Parameters -> ($Parameter_1$ (, $Parameter_n$)* )?
        Parameters.node = new Parameters(concatAll($Parameter_i$.node))
Parameter -> Type IDENT
        Parameter.node = new Parameter(type = Type.node, name = IDENT.lexval)
Type -> BasicType
        Type.node = BasicType.node
Type -> BasicType ([ ])*
        Type.node = new ArrayType(basicType = BasicType.node, dimension = count([ ]))

BasicType -> int | boolean | void
        BasicType.node = new PrimitiveType(int | boolean | void)
BasicType -> IDENT
        BasicType.node = new ClassType(IDENT.lexval)

Statement -> Block
        Statement.node = Block.node
Statement -> EmptyStatement
        Statement.node = null
```

```
Statement -> IfStatement
        Statement.node = IfStatement.node
Statement -> ExpressionStatement
        Statement.node = ExpressionStatement.node
Statement -> WhileStatement
        Statement.node = WhileStatement.node
Statement -> ReturnStatement
        Statement.node = ReturnStatement.node
Block -> { BlockStatement* }
        Block.node = concatAll(BlockStatement_i.node)
BlockStatement -> Statement
        BlockStatement.node = Statement.node
BlockStatement -> LocalVariableDeclaration
        BlockStatement.node = LocalVariableDeclaration.node

IfStatement -> if ( Expression ) Statement
        IfStatement.node = new IfStatement(
                cond=Expression.node,
                then=Statement.node,
                else=null
        )
IfStatement -> if ( Expression ) Statement else Statement
        IfStatement.node = new IfStatement(
                cond=Expression.node,
                then=Statement.node,
                else=Statement.node
        )
WhileStatement -> while ( Expression ) Statement
        WhileStatement.node = new WhileStatement(
                cond=Expression.node,
                expr=Statement.node
        )
ExpressionStatement -> Expression ;
        ExpressionStatement.node ->Expression.node

ReturnStatement -> return Expression ;
        ReturnStatement.node = new ReturnStatement(expr=Expression.node)
ReturnStatement -> return ;
        ReturnStatement.node = new ReturnStatement(expr=null)

        Expression.node = ArrayAccess.node

Expression -> BinaryExpression
        Expression.node = BinaryExpression.node

BinaryExpression -> UnaryExpression
```

```
    BinaryExpression.node = UnaryExpression.node
// BINOP ∈ (= | || | && | == | != | < | <= | > | >= | + | - | * | / | %)
BinaryExpression -> BinaryExpression$_1$ BINOP BinaryExpression$_2$
    BinaryExpression.node = new BinaryExpression(
        lhs = BinaryExpression$_1$.node
        operation = BINOP.lexval
        rhs = BinaryExpression$_2$.node
    )



// UNOP ∈ ( - | ! )
UnaryExpression$_1$ -> UNOP UnaryExpression$_2$
    UnaryExpression$_1$.node = new UnaryExpression(
        expression = UnaryExpression$_2$.node
        operation = UNOP.lexval
    )
UnaryExpression -> PostfixExpression
    UnaryExpression.node = PostfixExpression.node
PostfixExpression -> PrimaryExpression
    PostfixExpression.node = PrimaryExpression.node
PostfixExpression$_1$ -> PostfixExpression$_2$ . IDENT
    PostfixExpression$_1$.node = new FieldAccess(
        left = PostfixExpression$_2$.node,
        name = IDENT.lexval
    )
PostfixExpression$_1$ -> PostfixExpression$_2$ . IDENT ( Arguments )
    PostfixExpression$_1$.node = new MethodInvokation(
        left = PostfixExpression$_2$.node,
        name = IDENT.lexval,
        args = Arguments.node
    )
PostfixExpression$_1$ -> PostfixExpression$_2$ [ Expression ]
    PostfixExpression$_1$.node = new ArrayAccess(
        array = PostfixExpression$_2$.node,
        index = Expression.node
    )
Arguments -> (Expression$_1$ (, Expression$_n$)* )?
    Arguments.node = new Arguments(concatAll(Expression$_i$.node))

PrimaryExpression -> NewArrayExpression
    PrimaryExpression.node = NewArrayExpression.node
PrimaryExpression -> NewObjectExpression
    PrimaryExpression.node = NewObjectExpression.node
PrimaryExpression -> INTEGER_LITERAL
    PrimaryExpression.node = new IntLiteral(value = INTEGER_LITERAL.lexval)
```

PrimaryExpression -> **true**
        PrimaryExpression.node = `new BoolLiteral`(value = `true`)
PrimaryExpression -> **false**
        PrimaryExpression.node = `new BoolLiteral`(value = `false`)
PrimaryExpression -> **null**
        NullLiteral.node = `new NullLiteral`()
PrimaryExpression -> **this**
        ThisLiteral.node = `new ThisLiteral`()
PrimaryExpression -> **IDENT**
        PrimaryExpression.node = `new VarRef`(name=**IDENT**.lexval)
NewArrayExpression -> **new** BasicType **[** Expression **]** (**[ ]**)*

```
        NewArrayExpression.node = new NewArrayExpression(
                arrayType = new ArrayType(
                        basicType = BasicType.node,
                        dimension = count([ ]) + 1
                ),
                size = Expression.node
        )
```

NewObjectExpression -> **new IDENT ( )**
        NewObjectExpression.node = `new NewObjectExpression`(name = **IDENT**.lexval)