j-- Grammar

```
// Lexical grammar

// Whitespace -- ignored
" " | "\t" | "\n" | "\r" | "\f"

// Single line comment -- ignored
"//" { ~( "\n" | "\r" ) } ( "\n" | "\r" ["\n"] )

// Reserved words
ABSTRACT       ::= "abstract"
BOOLEAN        ::= "boolean"
CHAR           ::= "char"
CLASS          ::= "class"
ELSE           ::= "else"
EXTENDS        ::= "extends"
FALSE          ::= "false"
IF             ::= "if"
IMPORT         ::= "import"
INSTANCEOF     ::= "instanceof"
INT            ::= "int"
NEW            ::= "new"
NULL           ::= "null"
PACKAGE        ::= "package"
PRIVATE        ::= "private"
PROTECTED      ::= "protected"
PUBLIC         ::= "public"
RETURN         ::= "return"
STATIC         ::= "static"
SUPER          ::= "super"
THIS           ::= "this"
TRUE           ::= "true"
VOID           ::= "void"
WHILE          ::= "while"

// Separators
COMMA          ::= ","
DOT            ::= "."
LBRACK         ::= "["
LCURLY         ::= "{"
LPAREN         ::= "("
RBRACK         ::= "]"
RCURLY         ::= "}"
RPAREN         ::= ")"
SEMI           ::= ";"

// Operators
ASSIGN         ::= "="
DEC            ::= "--"
EQUAL          ::= "=="
GT             ::= ">"
INC            ::= "++"
LAND           ::= "&&"
LE             ::= "<="
LNOT           ::= "!"
MINUS          ::= "-"
PLUS           ::= "+"
PLUS_ASSIGN    ::= "+="
STAR           ::= "*"

// Identifiers
IDENTIFIER     ::= ( "a"..."z" | "A"..."Z" | "_" | "$" ) { "a"..."z" | "A"..."Z" | "_" | "0"..."9" | "$" }

// Literals
INT_LITERAL    ::= ( "0"..."9" ) { "0"..."9" }
ESC            ::= "\\" ( "n" | "r" | "t" | "b" | "f" | "'" | "\"" | "\\" )
STRING_LITERAL ::= "\"" { ESC | ~( "\"" | "\\" | "\n" | "\r" ) } "\""
CHAR_LITERAL   ::= "'" ( ESC | ~( "'" | "\n" | "\r" | "\\" ) ) "'"

// End of file
EOF            ::= "<end of file>"

compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF

qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }

```

```
typeDeclaration ::= modifiers classDeclaration

modifiers ::= { ABSTRACT | PRIVATE | PROTECTED | PUBLIC | STATIC }

classDeclaration ::= CLASS IDENTIFIER [ EXTENDS qualifiedIdentifier ] classBody

classBody ::= LCURLY { modifiers memberDecl } RCURLY

memberDecl ::= IDENTIFIER formalParameters block
             | ( VOID | type ) IDENTIFIER formalParameters ( block | SEMI )
             | type variableDeclarators SEMI

block ::= LCURLY { blockStatement } RCURLY

blockStatement ::= localVariableDeclarationStatement
                 | statement

statement ::= block
            | IF parExpression statement [ ELSE statement ]
            | RETURN [ expression ] SEMI
            | SEMI
            | WHILE parExpression statement
            | statementExpression SEMI

formalParameters ::= LPAREN [ formalParameter { COMMA formalParameter } ] RPAREN

formalParameter ::= type IDENTIFIER

parExpression ::= LPAREN expression RPAREN

localVariableDeclarationStatement ::= type variableDeclarators SEMI

variableDeclarators ::= variableDeclarator { COMMA variableDeclarator }

variableDeclarator ::= IDENTIFIER [ ASSIGN variableInitializer ]

variableInitializer ::= arrayInitializer | expression

arrayInitializer ::= LCURLY [ variableInitializer { COMMA variableInitializer } [ COMMA ] ] RCURLY

arguments ::= LPAREN [ expression { COMMA expression } ] RPAREN

type ::= referenceType | basicType

basicType ::= BOOLEAN | CHAR | INT

referenceType ::= basicType LBRACK RBRACK { LBRACK RBRACK }
                | qualifiedIdentifier { LBRACK RBRACK }

statementExpression ::= expression

expression ::= assignmentExpression

assignmentExpression ::= conditionalAndExpression [ ( ASSIGN | PLUS_ASSIGN ) assignmentExpression ]

conditionalAndExpression ::= equalityExpression { LAND equalityExpression }

equalityExpression ::= relationalExpression { EQUAL relationalExpression }

relationalExpression ::= additiveExpression [ ( GT | LE ) additiveExpression
                                            | INSTANCEOF referenceType ]

additiveExpression ::= multiplicativeExpression { ( MINUS | PLUS ) multiplicativeExpression }

multiplicativeExpression ::= unaryExpression { STAR unaryExpression }

unaryExpression ::= INC unaryExpression
                  | MINUS unaryExpression
                  | simpleUnaryExpression

simpleUnaryExpression ::= LNOT unaryExpression
                        | LPAREN basicType RPAREN unaryExpression
                        | LPAREN referenceType RPAREN simpleUnaryExpression
                        | postfixExpression

postfixExpression ::= primary { selector } { DEC }

selector ::= DOT qualifiedIdentifier [ arguments ]
           | LBRACK expression RBRACK

```

```
primary ::= parExpression
          | NEW creator
          | THIS [ arguments ]
          | SUPER ( arguments | DOT IDENTIFIER [ arguments ] )
          | qualifiedIdentifier [ arguments ]
          | literal

creator ::= ( basicType | qualifiedIdentifier )
                ( arguments
                | LBRACK RBRACK { LBRACK RBRACK } [ arrayInitializer ]
                | newArrayDeclarator
                )

newArrayDeclarator ::= LBRACK expression RBRACK { LBRACK expression RBRACK } { LBRACK RBRACK }

literal ::= CHAR_LITERAL | FALSE | INT_LITERAL | NULL | STRING_LITERAL | TRUE
```

Java Grammar

```
// Lexical grammar

// Whitespace -- ignored
" " | "\t" | "\n" | "\r" | "\f"

// Single line comment -- ignored
"//" { ~( "\n" | "\r" ) } ( "\n" | "\r" ["\n"] )

// Multiline comment -- ignored
"/*" { ~( "*/" ) } "*/"

// Reserved words
ABSTRACT       ::= "abstract"
BOOLEAN        ::= "boolean"
BREAK          ::= "break"
CASE           ::= "case"
CATCH          ::= "catch"
CHAR           ::= "char"
CLASS          ::= "class"
CONTINUE       ::= "continue"
DEFLT          ::= "default"
DO             ::= "do"
DOUBLE         ::= "double"
ELSE           ::= "else"
EXTENDS        ::= "extends"
FALSE          ::= "false"
FINALLY        ::= "finally"
FOR            ::= "for"
IF             ::= "if"
IMPLEMENTS     ::= "implements"
IMPORT         ::= "import"
INSTANCEOF     ::= "instanceof"
INT            ::= "int"
INTERFACE      ::= "interface"
LONG           ::= "long"
NEW            ::= "new"
NULL           ::= "null"
PACKAGE        ::= "package"
PRIVATE        ::= "private"
PROTECTED      ::= "protected"
PUBLIC         ::= "public"
RETURN         ::= "return"
STATIC         ::= "static"
SUPER          ::= "super"
SWITCH         ::= "switch"
THIS           ::= "this"
THROW          ::= "throw"
THROWS         ::= "throws"
TRUE           ::= "true"
TRY            ::= "try"
VOID           ::= "void"
WHILE          ::= "while"

// Separators
COMMA          ::= ","
DOT            ::= "."
LBRACK         ::= "["
LCURLY         ::= "{"
LPAREN         ::= "("
RBRACK         ::= "]"
```

```
RCURLY          ::= "}"
RPAREN          ::= ")"
SEMI            ::= ";"

// Operators
ALSHIFT         ::= "<<"
ALSHIFT_ASSIGN  ::= "<<="
AND             ::= "&"
AND_ASSIGN      ::= "&="
ARSHIFT         ::= ">>"
ARSHIFT_ASSIGN  ::= ">>="
ASSIGN          ::= "="
COLON           ::= ":"
DEC             ::= "--"
DIV             ::= "/"
DIV_ASSIGN      ::= "/="
EQUAL           ::= "=="
GE              ::= ">="
GT              ::= ">"
INC             ::= "++"
LAND            ::= "&&"
LE              ::= "<="
LNOT            ::= "!"
LOR             ::= "||"
LRSHIFT         ::= ">>>"
LRSHIFT_ASSIGN  ::= ">>>="
LT              ::= "<"
MINUS           ::= "-"
MINUS_ASSIGN    ::= "-="
NOT             ::= "~"
NOT_EQUAL       ::= "!="
OR              ::= "|"
OR_ASSIGN       ::= "|="
PLUS            ::= "+"
PLUS_ASSIGN     ::= "+="
QUESTION        ::= "?"
REM             ::= "%"
REM_ASSIGN      ::= "%="
STAR            ::= "*"
STAR_ASSIGN     ::= "*="
XOR             ::= "^"
XOR_ASSIGN      ::= "^="

// Identifiers
IDENTIFIER      ::= ( "a"..."z" | "A"..."Z" | "_" | "$" ) { "a"..."z" | "A"..."Z" | "_" | "0"..."9" | "$" }

// Literals
DIGITS          ::= ( "0"..."9" ) { "0"..."9" }
INT_LITERAL     ::= DIGITS
LONG_LITERAL    ::= INT_LITERAL ( "l" | "L" )
EXPONENT        ::= ( "e" | "E" ) [ ( "+" | "-" ) ] DIGITS
SUFFIX          ::= "d" | "D"
DOUBLE_LITERAL  ::= DIGITS "." [ DIGITS ] [ EXPONENT ] [ SUFFIX ]
                  | "." DIGITS [ EXPONENT ] [ SUFFIX ]
                  | DIGITS EXPONENT [ SUFFIX ]
                  | DIGITS [ EXPONENT ] SUFFIX
ESC             ::= "\\" ( "n" | "r" | "t" | "b" | "f" | "'" | "\"" | "\\" )
STRING_LITERAL  ::= "\"" { ESC | ~( "\"" | "\\" | "\n" | "\r" ) } "\""
CHAR_LITERAL    ::= "'" ( ESC | ~( "'" | "\n" | "\r" | "\\" ) ) "'"

// End of file
EOF             ::= "<end of file>"

// Syntactic grammar

compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF

qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }

typeDeclaration ::= modifiers ( classDeclaration | interfaceDeclaration )

modifiers ::= { ABSTRACT | PRIVATE | PROTECTED | PUBLIC | STATIC }

classDeclaration ::= CLASS IDENTIFIER [ EXTENDS qualifiedIdentifier ]
                         [ IMPLEMENTS qualifiedIdentifier { COMMA qualifiedIdentifier } ] classBody

interfaceDeclaration ::= INTERFACE IDENTIFIER
```

```
                              [ EXTENDS qualifiedIdentifier { COMMA qualifiedIdentifier } ] interfaceBody

classBody ::= LCURLY { modifiers memberDecl } RCURLY

interfaceBody ::= LCURLY { modifiers interfaceMemberDecl } RCURLY

memberDecl ::= IDENTIFIER formalParameters [ THROWS qualifiedIdentifier { COMMA qualifiedIdentifier } ] block
             | ( VOID | type ) IDENTIFIER formalParameters
                  [ THROWS qualifiedIdentifier { COMMA qualifiedIdentifier } ] ( block | SEMI )
             | type variableDeclarators SEMI

interfaceMemberDecl ::= ( VOID | type ) IDENTIFIER formalParameters
                            [ THROWS qualifiedIdentifier { COMMA qualifiedIdentifier } ] SEMI
                      | type variableDeclarators SEMI

block ::= LCURLY { blockStatement } RCURLY

blockStatement ::= localVariableDeclarationStatement
                 | statement

statement ::= block
            | BREAK SEMI
            | CONTINUE SEMI
            | DO statement WHILE parExpression SEMI
            | FOR LPAREN [ forInit ] SEMI [ expression ] SEMI [ forUpdate ] RPAREN statement
            | IF parExpression statement [ ELSE statement ]
            | RETURN [ expression ] SEMI
            | SEMI
            | SWITCH parExpression LCURLY { switchBlockStatementGroup } RCURLY
            | THROW expression SEMI
            | TRY block { CATCH LPAREN formalParameter RPAREN block } [ FINALLY block ]
            | WHILE parExpression statement
            | statementExpression SEMI

formalParameters ::= LPAREN [ formalParameter { COMMA formalParameter } ] RPAREN

formalParameter ::= type IDENTIFIER

parExpression ::= LPAREN expression RPAREN

forInit ::= statementExpression { COMMA statementExpression }
          | type variableDeclarators

forUpdate ::= statementExpression { COMMA statementExpression }

switchBlockStatementGroup ::= switchLabel { switchLabel } { blockStatement }

switchLabel ::= CASE expression COLON
              | DEFLT COLON

localVariableDeclarationStatement ::= type variableDeclarators SEMI

variableDeclarators ::= variableDeclarator { COMMA variableDeclarator }

variableDeclarator ::= IDENTIFIER [ ASSIGN variableInitializer ]

variableInitializer ::= arrayInitializer | expression

arrayInitializer ::= LCURLY [ variableInitializer { COMMA variableInitializer } [ COMMA ] ] RCURLY

arguments ::= LPAREN [ expression { COMMA expression } ] RPAREN

type ::= basicType | referenceType

basicType ::= BOOLEAN | CHAR | DOUBLE | INT | LONG

referenceType ::= basicType LBRACK RBRACK { LBRACK RBRACK }
                | qualifiedIdentifier { LBRACK RBRACK }

statementExpression ::= expression

expression ::= assignmentExpression

assignmentExpression ::= conditionalExpression
                           [ ( ALSHIFT_ASSIGN | AND_ASSIGN | ARSHIFT_ASSIGN | ASSIGN | DIV_ASSIGN
                             | LRSHIFT_ASSIGN | MINUS_ASSIGN | OR_ASSIGN | PLUS_ASSIGN | REM_ASSIGN
                             | STAR_ASSIGN | XOR_ASSIGN ) assignmentExpression ]

conditionalExpression ::= conditionalOrExpression [ QUESTION expression COLON conditionalExpression ]

```

```
conditionalOrExpression ::= conditionalAndExpression { LOR conditionalAndExpression }

conditionalAndExpression ::= inclusiveOrExpression { LAND inclusiveOrExpression }

inclusiveOrExpression ::= exclusiveOrExpression { OR exclusiveOrExpression }

exclusiveOrExpression ::= andExpression { XOR andExpression }

andExpression ::= equalityExpression { AND equalityExpression }

equalityExpression ::= relationalExpression { ( EQUAL | NOT_EQUAL ) relationalExpression }

relationalExpression ::= shiftExpression [ ( GE | GT | LE | LT ) shiftExpression | INSTANCEOF referenceType ]

shiftExpression ::= additiveExpression { ( ALSHIFT | ARSHIFT | LRSHIFT ) additiveExpression }

additiveExpression ::= multiplicativeExpression { ( MINUS | PLUS ) multiplicativeExpression }

multiplicativeExpression ::= unaryExpression { ( DIV | REM | STAR ) unaryExpression }

unaryExpression ::= DEC unaryExpression
                  | INC unaryExpression
                  | ( MINUS | PLUS ) unaryExpression
                  | simpleUnaryExpression

simpleUnaryExpression ::= LNOT unaryExpression
                        | NOT unaryExpression
                        | LPAREN basicType RPAREN unaryExpression
                        | LPAREN referenceType RPAREN simpleUnaryExpression
                        | postfixExpression

postfixExpression ::= primary { selector } { DEC | INC }

selector ::= DOT qualifiedIdentifier [ arguments ]
           | LBRACK expression RBRACK

primary ::= parExpression
          | NEW creator
          | THIS [ arguments ]
          | SUPER ( arguments | DOT IDENTIFIER [ arguments ] )
          | qualifiedIdentifer [ arguments]
          | literal

creator ::= ( basicType | qualifiedIdentifier )
                ( arguments
                | LBRACK RBRACK { LBRACK RBRACK } [ arrayInitializer ]
                | newArrayDeclarator
                )

newArrayDeclarator ::= LBRACK [ expression ] RBRACK { LBRACK [ expression ] RBRACK }

literal ::= CHAR_LITERAL | DOUBLE_LITERAL | FALSE | INT_LITERAL | LONG_LITERAL | NULL | STRING_LITERAL | TRUE
```