1 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>"

2 Syntactic Grammar

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 memberDeclar } RCURLY

memberDeclar ::= 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
The \( j \)-Language Specification

```plaintext
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

3 Semantics

JArrayExpression:
- The thing indexed must be an array
- The index must be an integer

JArrayInitializer:
- A non-array object must not be initialized with the array sequence {...}
- Each initializer must have the same type as the component type

JAssignment:
- JAssignOp:
  - lhs must be legal
  - lhs and rhs must have the same type
- JPlusAssignOp:
  - lhs must be legal
  - lhs and rhs must be integers (addition) or lhs must be a string (concatenation)

JBinaryExpression:
- JMultiplyOp, JSubtractOp
  - lhs and rhs must be integers
- JPlusOp
  - lhs and rhs must be integers (addition) or one of them must be a string (concatenation)

JBooleanBinaryExpression:
- JEqualOp:
  - lhs and rhs must have the same type
- JLogicalAndOp:
  - lhs and rhs must be booleans

JCastOp:
- Source type must be compatible with the target type

JClassDeclaration:
- Super type must be accessible from the base type
- Super type must not be final
```
- A non-abstract class must not declare abstract methods

JComparisonExpression:
- lhs and rhs must be integers

JCompilationUnit:
- Imports must be valid

JConstructorDeclaration:
- A constructor must not be static or abstract
- Signature must not exist already

JFieldDeclaration:
- A field must not be abstract
- Name must not exist already

JFieldSelection:
- The target must be a reference type
- The field must be declared
- The field must be accessible
- A non-static field must not be referenced from a static context
- A final field must not be assigned a value

JIfStatement:
- The condition must be a boolean

JInstanceOfOp:
- lhs and rhs must be reference types and assignable from one to the other

JMessageExpression:
- The target must be a reference type
- The message must exist
- The message must be accessible
- A non-static message must not be referenced from a static context

JMethodDeclaration:
- An abstract method cannot have a body
- A method without body must be abstract
- A private method cannot be abstract
- A static method cannot be abstract
- Signature must not exist already
- A non-void method must have a return statement

JNewArrayOp:
- Dimensions must be integers

JNewOp:
- The constructor being invoked must not instantiate an abstract type
- The constructor being invoked must exist

JReturnStatement:
- Must not return a value from a constructor
- Must not return a value from a void method
- The type of return value in a non-void method must match return type of the method
- A non-void method must have a return value

    JSuperConstruction:
    - super(...) must be the first statement in the constructor's body
    - A super constructor with the given argument types must exist

    JThisConstruction:
    - this(...) must be the first statement in the constructor's body
    - A constructor with the given argument types must exist

    JVariable:
    - The variable name must exist
    - The variable must be initialized
    - The variable must be a valid lhs to =

    JVariableDeclaration:
    - The variable must not shadow another local variable

    JUnaryExpression:
    - JLogicalNotOp:
      - The operand must be a boolean
    - JNegateOp, JUnaryPlusOp:
      - The operand must be an integer
    - JPostDecrementOp, JPreIncrementOp:
      - The operand must have an LValue
      - The operand must be an integer

    JWhileStatement:
    - The condition must be a boolean