# 1 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"
CHAR           ::= "char"
CLASS          ::= "class"
CONTINUE       ::= "continue"
DEFLT          ::= "default"
DO             ::= "do"
DOUBLE         ::= "double"
ELSE           ::= "else"
EXTENDS        ::= "extends"
FALSE          ::= "false"
FOR            ::= "for"
IF             ::= "if"
IMPORT         ::= "import"
INSTANCEOF     ::= "instanceof"
INT            ::= "int"
LONG           ::= "long"
NEW            ::= "new"
NULL           ::= "null"
PACKAGE        ::= "package"
PRIVATE        ::= "private"
PROTECTED      ::= "protected"
PUBLIC         ::= "public"
RETURN         ::= "return"
STATIC         ::= "static"
SUPER          ::= "super"
SWITCH         ::= "switch"
THIS           ::= "this"
TRUE           ::= "true"
VOID           ::= "void"
WHILE          ::= "while"

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

```
53  RPAREN          ::= ")"
54  SEMI            ::= ";"
55
56  // Operators
57  ASSIGN          ::= "="
58  COLON           ::= ":"
59  DEC             ::= "--"
60  DIV             ::= "/"
61  DIV_ASSIGN      ::= "/="
62  EQUAL           ::= "=="
63  GE              ::= ">="
64  GT              ::= ">"
65  INC             ::= "++"
66  LAND            ::= "&&"
67  LE              ::= "<="
68  LNOT            ::= "!"
69  LOR             ::= "||"
70  LT              ::= "<"
71  MINUS           ::= "-"
72  MINUS_ASSIGN    ::= "-="
73  NOT             ::= "~"
74  NOT_EQUAL       ::= "!="
75  PLUS            ::= "+"
76  PLUS_ASSIGN     ::= "+="
77  QUESTION        ::= "?"
78  REM             ::= "%"
79  REM_ASSIGN      ::= "%="
80  STAR            ::= "*"
81  STAR_ASSIGN     ::= "*="
82
83  // Identifiers
84  IDENTIFIER      ::= ( "a"..."z" | "A"..."Z" | "_" | "$" )
85                      { "a"..."z" | "A"..."Z" | "_" | "0"..."9" | "$" }
86
87  // Literals
88  DIGITS          ::= ( "0"..."9" ) { "0"..."9" }
89  INT_LITERAL     ::= DIGITS
90  LONG_LITERAL    ::= INT_LITERAL ( "l" | "L" )
91  EXPONENT        ::= ( "e" | "E" ) [ ( "+" | "-" ) ] DIGITS
92  SUFFIX          ::= "d" | "D"
93  DOUBLE_LITERAL  ::= DIGITS "." [ DIGITS ] [ EXPONENT ] [ SUFFIX ]
94                    | "." DIGITS [ EXPONENT ] [ SUFFIX ]
95                    | DIGITS EXPONENT [ SUFFIX ]
96                    | DIGITS SUFFIX
97  ESC             ::= "\\" ( "n" | "r" | "t" | "b" | "f" | "'" | "\"" | "\\" )
98  STRING_LITERAL  ::= "\"" { ESC | ~( "\"" | "\\" | "\n" | "\r" ) } "\""
99  CHAR_LITERAL    ::= "'" ( ESC | ~( "'" | "\n" | "\r" | "\\" ) ) "'"
100
101 // End of file
102 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 memberDecl } RCURLY

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

```

```
53  localVariableDeclarationStatement ::= type variableDeclarators SEMI
54
55  variableDeclarators ::= variableDeclarator { COMMA variableDeclarator }
56
57  variableDeclarator ::= IDENTIFIER [ ASSIGN variableInitializer ]
58
59  variableInitializer ::= arrayInitializer | expression
60
61  arrayInitializer ::= LCURLY [ variableInitializer { COMMA variableInitializer } [ COMMA ] ] RCURLY
62
63  arguments ::= LPAREN [ expression { COMMA expression } ] RPAREN
64
65  type ::= basicType | referenceType
66
67  basicType ::= BOOLEAN | CHAR | DOUBLE | INT | LONG
68
69  referenceType ::= basicType LBRACK RBRACK { LBRACK RBRACK }
70                  | qualifiedIdentifier { LBRACK RBRACK }
71
72  statementExpression ::= expression
73
74  expression ::= assignmentExpression
75
76  assignmentExpression ::= conditionalExpression
77                             [ ( ASSIGN | DIV_ASSIGN | MINUS_ASSIGN
78                               | PLUS_ASSIGN | REM_ASSIGN | STAR_ASSIGN ) assignmentExpression ]
79
80  conditionalExpression ::= conditionalOrExpression [ QUESTION expression COLON conditionalExpression ]
81
82  conditionalOrExpression ::= conditionalAndExpression { LOR conditionalAndExpression }
83
84  conditionalAndExpression ::= equalityExpression { LAND equalityExpression }
85
86  equalityExpression ::= relationalExpression { ( EQUAL | NOT_EQUAL ) relationalExpression }
87
88  relationalExpression ::= additiveExpression [ ( GE | GT | LE | LT ) additiveExpression
89                                              | INSTANCEOF referenceType ]
90
91  additiveExpression ::= multiplicativeExpression { ( MINUS | PLUS ) multiplicativeExpression }
92
93  multiplicativeExpression ::= unaryExpression { ( DIV | REM | STAR ) unaryExpression }
94
95  unaryExpression ::= ( DEC | INC ) unaryExpression
96                    | ( MINUS | PLUS ) unaryExpression
97                    | simpleUnaryExpression
98
99  simpleUnaryExpression ::= LNOT unaryExpression
100                         | NOT unaryExpression
101                         | LPAREN basicType RPAREN unaryExpression
102                         | LPAREN referenceType RPAREN simpleUnaryExpression
103                         | postfixExpression
104
105 postfixExpression ::= primary { selector } { DEC | INC }
106
```

```
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
```

# 3 Semantics

```
JArrayExpression:
- The thing indexed must be an array
- The index must be an int

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 both be a double, int, or long (addition) or lhs must be a
    string (concatenation)
- JDivAssignOp, JMinusAssign, JRemAssignOp, JStarAssignOp
  - lhs must be legal
  - lhs and rhs must both be a double, int, or long

JBinaryExpression:
- JDivideOp, JMultiplyOp, JRemainderOp, JSubtractOp
  - lhs and rhs must both be a double, int, or long
- JPlusOp
  - lhs and rhs must both be a double, int, or long (addition) or either must be a
    string (concatenation)

JBooleanBinaryExpression:
- JEqualOp, JNotEqualOp:
  - lhs and rhs must have the same type
```

```
- JLogicalAndOp, JLogicalOrOp:
  - lhs and rhs must be booleans

JBreakStatement:
- Must not be outside of a switch or loop

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 both be a double, int, or long

JCompilationUnit:
- Imports must be valid

JConditionalExpression (e ? e1 : e2):
- e must be a boolean
- e1 and e2 must have the same type

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

JContinueStatement:
- Must not be outside of a loop

JDoStatement:
- The condition must be a boolean

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

JForStatement:
- The condition must be a boolean

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 ints

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

JSwitchStatement:
- The condition must be an int
- Each case expression must be an int literal
- No two case constants may have the same value
- No more than one default label may be present

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 a double, int, or long
```

```
- JPostDecrementOp, JPostIncrementOp, JPreDecrementOp, JPreIncrementOp:
  - The operand must have an LValue
  - The operand must be a double, int, or long

JWhileStatement:
- The condition must be a boolean
```