

1 Lexical Grammar

```
1 // Whitespace -- ignored
2 " " | "\t" | "\n" | "\r" | "\f"
3
4 // Single line comment -- ignored
5 "//" { ~( "\n" | "\r" ) } ( "\n" | "\r" ["\n"] )
6
7 // Multiline comment -- ignored
8 "/*" { ~( "*/" ) } "*/"
9
10 // Reserved words
11 ABSTRACT      ::= "abstract"
12 BOOLEAN      ::= "boolean"
13 BREAK        ::= "break"
14 CASE         ::= "case"
15 CHAR         ::= "char"
16 CLASS        ::= "class"
17 CONTINUE     ::= "continue"
18 DEFLT        ::= "default"
19 DO           ::= "do"
20 DOUBLE       ::= "double"
21 ELSE         ::= "else"
22 EXTENDS      ::= "extends"
23 FALSE        ::= "false"
24 FOR          ::= "for"
25 IF           ::= "if"
26 IMPORT       ::= "import"
27 INSTANCEOF   ::= "instanceof"
28 INT          ::= "int"
29 LONG         ::= "long"
30 NEW          ::= "new"
31 NULL         ::= "null"
32 PACKAGE      ::= "package"
33 PRIVATE      ::= "private"
34 PROTECTED    ::= "protected"
35 PUBLIC       ::= "public"
36 RETURN       ::= "return"
37 STATIC       ::= "static"
38 SUPER        ::= "super"
39 SWITCH       ::= "switch"
40 THIS         ::= "this"
41 TRUE         ::= "true"
42 VOID         ::= "void"
43 WHILE        ::= "while"
44
45 // Separators
46 COMMA        ::= ","
47 DOT          ::= "."
48 LBRACK       ::= "["
49 LCURLY       ::= "{"
50 LPAREN       ::= "("
51 RBRACK       ::= "]"
52 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

```
1 compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
2                 { IMPORT qualifiedIdentifier SEMI }
3                 { typeDeclaration }
4                 EOF
5
6 qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
7
8 typeDeclaration ::= modifiers classDeclaration
9
10 modifiers ::= { ABSTRACT | PRIVATE | PROTECTED | PUBLIC | STATIC }
11
12 classDeclaration ::= CLASS IDENTIFIER [ EXTENDS qualifiedIdentifier ] classBody
13
14 classBody ::= LCURLY { modifiers memberDecl } RCURLY
15
16 memberDecl ::= IDENTIFIER formalParameters block
17              | ( VOID | type ) IDENTIFIER formalParameters ( block | SEMI )
18              | type variableDeclarators SEMI
19
20 block ::= LCURLY { blockStatement } RCURLY
21
22 blockStatement ::= localVariableDeclarationStatement
23                 | statement
24
25 statement ::= block
26             | BREAK SEMI
27             | CONTINUE SEMI
28             | DO statement WHILE parExpression SEMI
29             | FOR LPAREN [ forInit ] SEMI [ expression ] SEMI [ forUpdate ] RPAREN statement
30             | IF parExpression statement [ ELSE statement ]
31             | RETURN [ expression ] SEMI
32             | SEMI
33             | SWITCH parExpression LCURLY { switchBlockStatementGroup } RCURLY
34             | WHILE parExpression statement
35             | statementExpression SEMI
36
37 formalParameters ::= LPAREN [ formalParameter { COMMA formalParameter } ] RPAREN
38
39 formalParameter ::= type IDENTIFIER
40
41 parExpression ::= LPAREN expression RPAREN
42
43 forInit ::= statementExpression { COMMA statementExpression }
44          | type variableDeclarators
45
46 forUpdate ::= statementExpression { COMMA statementExpression }
47
48 switchBlockStatementGroup ::= switchLabel { switchLabel } { blockStatement }
49
50 switchLabel ::= CASE expression COLON
51              | DEFLT COLON
52
```

```

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

```

```

107 selector ::= DOT qualifiedIdentifier [ arguments ]
108           | LBRACK expression RBRACK
109
110 primary ::= parExpression
111           | NEW creator
112           | THIS [ arguments ]
113           | SUPER ( arguments | DOT IDENTIFIER [ arguments ] )
114           | qualifiedIdentifier [ arguments ]
115           | literal
116
117 creator ::= ( basicType | qualifiedIdentifier )
118           ( arguments
119           | LBRACK RBRACK { LBRACK RBRACK } [ arrayInitializer ]
120           | newArrayDeclarator
121           )
122
123 newArrayDeclarator ::= LBRACK [ expression ] RBRACK { LBRACK [ expression ] RBRACK }
124
125 literal ::= CHAR_LITERAL | DOUBLE_LITERAL | FALSE | INT_LITERAL
126           | LONG_LITERAL | NULL | STRING_LITERAL | TRUE

```

3 Semantics

```

1 JArrayExpression:
2 - The thing indexed must be an array
3 - The index must be an int
4
5 JArrayInitializer:
6 - A non-array object must not be initialized with the array sequence {...}
7 - Each initializer must have the same type as the component type
8
9 JAssignment:
10 - JAssignOp:
11   - lhs must be legal
12   - lhs and rhs must have the same type
13 - JPlusAssignOp:
14   - lhs must be legal
15   - lhs and rhs must both be a double, int, or long (addition) or lhs must be a
16     string (concatenation)
17 - JDivAssignOp, JMinusAssign, JRemAssignOp, JStarAssignOp
18   - lhs must be legal
19   - lhs and rhs must both be a double, int, or long
20
21 JBinaryExpression:
22 - JDivideOp, JMultiplyOp, JRemainderOp, JSubtractOp
23   - lhs and rhs must both be a double, int, or long
24 - JPlusOp
25   - lhs and rhs must both be a double, int, or long (addition) or either must be a
26     string (concatenation)
27
28 JBooleanBinaryExpression:
29 - JEqualOp, JNotEqualOp:
30   - lhs and rhs must have the same type

```

31 - JLogicalAndOp, JLogicalOrOp:
32 - lhs and rhs must be booleans
33
34 JBreakStatement:
35 - Must not be outside of a switch or loop
36
37 JCastOp:
38 - Source type must be compatible with the target type
39
40 JClassDeclaration:
41 - Super type must be accessible from the base type
42 - Super type must not be final
43 - A non-abstract class must not declare abstract methods
44
45 JComparisonExpression:
46 - lhs and rhs must both be a double, int, or long
47
48 JCompilationUnit:
49 - Imports must be valid
50
51 JConditionalExpression (e ? e1 : e2):
52 - e must be a boolean
53 - e1 and e2 must have the same type
54
55 JConstructorDeclaration:
56 - A constructor must not be static or abstract
57 - Signature must not exist already
58
59 JContinueStatement:
60 - Must not be outside of a loop
61
62 JDoStatement:
63 - The condition must be a boolean
64
65 JFieldDeclaration:
66 - A field must not be abstract
67 - Name must not exist already
68
69 JFieldSelection:
70 - The target must be a reference type
71 - The field must be declared
72 - The field must be accessible
73 - A non-static field must not be referenced from a static context
74 - A final field must not be assigned a value
75
76 JForStatement:
77 - The condition must be a boolean
78
79 JIfStatement:
80 - The condition must be a boolean
81
82 JInstanceOfOp:
83 - lhs and rhs must be reference types and assignable from one to the other
84

85 JMessageExpression:
86 - The target must be a reference type
87 - The message must exist
88 - The message must be accessible
89 - A non-static message must not be referenced from a static context
90
91 JMethodDeclaration:
92 - An abstract method cannot have a body
93 - A method without body must be abstract
94 - A private method cannot be abstract
95 - A static method cannot be abstract
96 - Signature must not exist already
97 - A non-void method must have a return statement
98
99 JNewArrayOp:
100 - Dimensions must be ints
101
102 JNewOp:
103 - The constructor being invoked must not instantiate an abstract type
104 - The constructor being invoked must exist
105
106 JReturnStatement:
107 - Must not return a value from a constructor
108 - Must not return a value from a void method
109 - The type of return value in a non-void method must match return type of the method
110 - A non-void method must have a return value
111
112 JSuperConstruction:
113 - super(...) must be the first statement in the constructor's body
114 - A super constructor with the given argument types must exist
115
116 JSwitchStatement:
117 - The condition must be an int
118 - Each case expression must be an int literal
119 - No two case constants may have the same value
120 - No more than one default label may be present
121
122 JThisConstruction:
123 - this(...) must be the first statement in the constructor's body
124 - A constructor with the given argument types must exist
125
126 JVariable:
127 - The variable name must exist
128 - The variable must be initialized
129 - The variable must be a valid lhs to =
130
131 JVariableDeclaration:
132 - The variable must not shadow another local variable
133
134 JUnaryExpression:
135 - JLogicalNotOp:
136 - The operand must be a boolean
137 - JNegateOp, JUnaryPlusOp:
138 - The operand must be a double, int, or long

- 139 - JPostDecrementOp, JPostIncrementOp, JPreDecrementOp, JPreIncrementOp:
- 140 - The operand must have an LValue
- 141 - The operand must be a double, int, or long
- 142
- 143 JWhileStatement:
- 144 - The condition must be a boolean