

```

j-- Grammar
1 // Lexical grammar
2
3 // Whitespace -- ignored
4 " " | "\t" | "\n" | "\r" | "\f"
5
6 // Single line comment -- ignored
7 "/*" { ~( "\n" | "\r" ) } ( "\n" | "\r" ["\n"] )
8
9 // Reserved words
10 ABSTRACT      ::= "abstract"
11 BOOLEAN      ::= "boolean"
12 CHAR         ::= "char"
13 CLASS        ::= "class"
14 ELSE         ::= "else"
15 EXTENDS      ::= "extends"
16 FALSE        ::= "false"
17 IF           ::= "if"
18 IMPORT       ::= "import"
19 INSTANCEOF   ::= "instanceof"
20 INT          ::= "int"
21 NEW         ::= "new"
22 NULL        ::= "null"
23 PACKAGE     ::= "package"
24 PRIVATE     ::= "private"
25 PROTECTED   ::= "protected"
26 PUBLIC      ::= "public"
27 RETURN      ::= "return"
28 STATIC      ::= "static"
29 SUPER       ::= "super"
30 THIS        ::= "this"
31 TRUE        ::= "true"
32 VOID        ::= "void"
33 WHILE       ::= "while"
34
35 // Separators
36 COMMA       ::= ","
37 DOT         ::= "."
38 LBRACK      ::= "["
39 LCURLY      ::= "{"
40 LPAREN      ::= "("
41 RBRACK      ::= "]"
42 RCURLY      ::= "}"
43 RPAREN     ::= ")"
44 SEMI        ::= ";"
45
46 // Operators
47 ASSIGN      ::= "="
48 DEC         ::= "--"
49 EQUAL       ::= "=="
50 GT          ::= ">"
51 INC         ::= "++"
52 LAND        ::= "&&"
53 LE          ::= "<="
54 LNOT        ::= "!"
55 MINUS       ::= "-"
56 PLUS        ::= "+"
57 PLUS_ASSIGN ::= "+="
58 STAR        ::= "*"
59
60 // Identifiers
61 IDENTIFIER  ::= ( "a"..."z" | "A"..."Z" | "_" | "$" ) { "a"..."z" | "A"..."Z" | "_" | "0"..."9" | "$" }
62
63 // Literals
64 INT_LITERAL ::= ( "0"..."9" ) { "0"..."9" }
65 ESC         ::= "\\\" ( "n" | "r" | "t" | "b" | "f" | "'" | "\"" | "\\\" )
66 STRING_LITERAL ::= "\\\" { ESC | ~( "\\\" | "\\\" | \"n\" | \"r\" ) } "\\\"
67 CHAR_LITERAL ::= "'" ( ESC | ~( "'" | \"n\" | \"r\" | "\\\" ) ) "'"
68
69 // End of file
70 EOF         ::= "<end of file>"
71
72 compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
73                  { IMPORT qualifiedIdentifier SEMI }
74                  { typeDeclaration }
75                  EOF
76
77 qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
78

```

```

79 typeDeclaration ::= modifiers classDeclaration
80
81 modifiers ::= { ABSTRACT | PRIVATE | PROTECTED | PUBLIC | STATIC }
82
83 classDeclaration ::= CLASS IDENTIFIER [ EXTENDS qualifiedIdentifier ] classBody
84
85 classBody ::= LCURLY { modifiers memberDecl } RCURLY
86
87 memberDecl ::= IDENTIFIER formalParameters block
88               | ( VOID | type ) IDENTIFIER formalParameters ( block | SEMI )
89               | type variableDeclarators SEMI
90
91 block ::= LCURLY { blockStatement } RCURLY
92
93 blockStatement ::= localVariableDeclarationStatement
94                  | statement
95
96 statement ::= block
97              | IF parExpression statement [ ELSE statement ]
98              | RETURN [ expression ] SEMI
99              | SEMI
100             | WHILE parExpression statement
101             | statementExpression SEMI
102
103 formalParameters ::= LPAREN [ formalParameter { COMMA formalParameter } ] RPAREN
104
105 formalParameter ::= type IDENTIFIER
106
107 parExpression ::= LPAREN expression RPAREN
108
109 localVariableDeclarationStatement ::= type variableDeclarators SEMI
110
111 variableDeclarators ::= variableDeclarator { COMMA variableDeclarator }
112
113 variableDeclarator ::= IDENTIFIER [ ASSIGN variableInitializer ]
114
115 variableInitializer ::= arrayInitializer | expression
116
117 arrayInitializer ::= LCURLY [ variableInitializer { COMMA variableInitializer } [ COMMA ] ] RCURLY
118
119 arguments ::= LPAREN [ expression { COMMA expression } ] RPAREN
120
121 type ::= referenceType | basicType
122
123 basicType ::= BOOLEAN | CHAR | INT
124
125 referenceType ::= basicType LBRACK RBRACK { LBRACK RBRACK }
126                  | qualifiedIdentifier { LBRACK RBRACK }
127
128 statementExpression ::= expression
129
130 expression ::= assignmentExpression
131
132 assignmentExpression ::= conditionalAndExpression [ ( ASSIGN | PLUS_ASSIGN ) assignmentExpression ]
133
134 conditionalAndExpression ::= equalityExpression { LAND equalityExpression }
135
136 equalityExpression ::= relationalExpression { EQUAL relationalExpression }
137
138 relationalExpression ::= additiveExpression [ ( GT | LE ) additiveExpression
139                               | INSTANCEOF referenceType ]
140
141 additiveExpression ::= multiplicativeExpression { ( MINUS | PLUS ) multiplicativeExpression }
142
143 multiplicativeExpression ::= unaryExpression { STAR unaryExpression }
144
145 unaryExpression ::= INC unaryExpression
146                  | MINUS unaryExpression
147                  | simpleUnaryExpression
148
149 simpleUnaryExpression ::= LNOT unaryExpression
150                        | LPAREN basicType RPAREN unaryExpression
151                        | LPAREN referenceType RPAREN simpleUnaryExpression
152                        | postfixExpression
153
154 postfixExpression ::= primary { selector } { DEC }
155
156 selector ::= DOT qualifiedIdentifier [ arguments ]
157            | LBRACK expression RBRACK
158

```

```

159 primary ::= parExpression
160           | NEW creator
161           | THIS [ arguments ]
162           | SUPER ( arguments | DOT IDENTIFIER [ arguments ] )
163           | qualifiedIdentifier [ arguments ]
164           | literal
165
166 creator ::= ( basicType | qualifiedIdentifier )
167           ( arguments
168             | LBRACK RBRACK { LBRACK RBRACK } [ arrayInitializer ]
169             | newArrayDeclarator
170           )
171
172 newArrayDeclarator ::= LBRACK expression RBRACK { LBRACK expression RBRACK } { LBRACK RBRACK }
173
174 literal ::= CHAR_LITERAL | FALSE | INT_LITERAL | NULL | STRING_LITERAL | TRUE

```

```

Java Grammar
1 // Lexical grammar
2
3 // Whitespace -- ignored
4 " " | "\t" | "\n" | "\r" | "\f"
5
6 // Single line comment -- ignored
7 "/*" { ~( "\n" | "\r" ) } ( "\n" | "\r" ["\n"] )
8
9 // Multiline comment -- ignored
10 "/*" { ~( "*/" ) } "*/"
11
12 // Reserved words
13 ABSTRACT      ::= "abstract"
14 BOOLEAN      ::= "boolean"
15 BREAK        ::= "break"
16 CASE         ::= "case"
17 CATCH        ::= "catch"
18 CHAR         ::= "char"
19 CLASS        ::= "class"
20 CONTINUE     ::= "continue"
21 DEFLT        ::= "default"
22 DO           ::= "do"
23 DOUBLE       ::= "double"
24 ELSE         ::= "else"
25 EXTENDS      ::= "extends"
26 FALSE        ::= "false"
27 FINALLY      ::= "finally"
28 FOR          ::= "for"
29 IF           ::= "if"
30 IMPLEMENTS   ::= "implements"
31 IMPORT        ::= "import"
32 INSTANCEOF   ::= "instanceof"
33 INT          ::= "int"
34 INTERFACE    ::= "interface"
35 LONG         ::= "long"
36 NEW          ::= "new"
37 NULL         ::= "null"
38 PACKAGE      ::= "package"
39 PRIVATE      ::= "private"
40 PROTECTED    ::= "protected"
41 PUBLIC       ::= "public"
42 RETURN       ::= "return"
43 STATIC       ::= "static"
44 SUPER        ::= "super"
45 SWITCH       ::= "switch"
46 THIS         ::= "this"
47 THROW        ::= "throw"
48 THROWS       ::= "throws"
49 TRUE         ::= "true"
50 TRY          ::= "try"
51 VOID         ::= "void"
52 WHILE        ::= "while"
53
54 // Separators
55 COMMA        ::= ","
56 DOT          ::= "."
57 LBRACK       ::= "["
58 LCURLY       ::= "{"
59 LPAREN       ::= "("
60 RBRACK       ::= "]"

```

```

61 RCURLY      ::= "}"
62 RPAREN     ::= ")"
63 SEMI       ::= ";"
64
65 // Operators
66 LSHIFT     ::= "<<"
67 LSHIFT_ASSIGN ::= "<<="
68 AND        ::= "&"
69 AND_ASSIGN ::= "&="
70 RSHIFT     ::= ">>"
71 RSHIFT_ASSIGN ::= ">>="
72 ASSIGN     ::= "="
73 COLON      ::= ":"
74 DEC        ::= "--"
75 DIV        ::= "/"
76 DIV_ASSIGN ::= "/="
77 EQUAL      ::= "=="
78 GE         ::= ">="
79 GT         ::= ">"
80 INC        ::= "++"
81 LAND       ::= "&&"
82 LE         ::= "<="
83 LNOT       ::= "!"
84 LOR        ::= "||"
85 LRSRIFT    ::= ">>>"
86 LRSRIFT_ASSIGN ::= ">>>="
87 LT         ::= "<"
88 MINUS      ::= "-"
89 MINUS_ASSIGN ::= "-="
90 NOT        ::= "~"
91 NOT_EQUAL  ::= "!="
92 OR         ::= "|"
93 OR_ASSIGN  ::= "|="
94 PLUS       ::= "+"
95 PLUS_ASSIGN ::= "+="
96 QUESTION  ::= "?"
97 REM        ::= "%"
98 REM_ASSIGN ::= "%="
99 STAR       ::= "*"
100 STAR_ASSIGN ::= "*="
101 XOR        ::= "^"
102 XOR_ASSIGN ::= "^="
103
104 // Identifiers
105 IDENTIFIER ::= ( "a"..."z" | "A"..."Z" | "_" | "$" ) { "a"..."z" | "A"..."Z" | "_" | "0"..."9" | "$" }
106
107 // Literals
108 DIGITS     ::= ( "0"..."9" ) { "0"..."9" }
109 INT_LITERAL ::= DIGITS
110 LONG_LITERAL ::= INT_LITERAL ( "l" | "L" )
111 EXPONENT   ::= ( "e" | "E" ) [ ( "+" | "-" ) ] DIGITS
112 SUFFIX     ::= "d" | "D"
113 DOUBLE_LITERAL ::= DIGITS "." [ DIGITS ] [ EXPONENT ] [ SUFFIX ]
114             | "." DIGITS [ EXPONENT ] [ SUFFIX ]
115             | DIGITS EXPONENT [ SUFFIX ]
116             | DIGITS [ EXPONENT ] SUFFIX
117 ESC        ::= "\\\" ( "n" | "r" | "t" | "b" | "f" | "'" | "\"" | "\\\" )
118 STRING_LITERAL ::= "\\\" { ESC | ~( "\\\" | "\\\" | "\\n" | "\\r" ) } "\\\"
119 CHAR_LITERAL  ::= "'" ( ESC | ~( "'" | "\\n" | "\\r" | "\\\" ) ) "'"
120
121 // End of file
122 EOF         ::= "<end of file>"
123
124 // Syntactic grammar
125
126 compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
127                 { IMPORT qualifiedIdentifier SEMI }
128                 { typeDeclaration }
129                 EOF
130
131 qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
132
133 typeDeclaration ::= modifiers ( classDeclaration | interfaceDeclaration )
134
135 modifiers ::= { ABSTRACT | PRIVATE | PROTECTED | PUBLIC | STATIC }
136
137 classDeclaration ::= CLASS IDENTIFIER [ EXTENDS qualifiedIdentifier ]
138                 [ IMPLEMENTS qualifiedIdentifier { COMMA qualifiedIdentifier } ] classBody
139
140 interfaceDeclaration ::= INTERFACE IDENTIFIER

```

```

141         [ EXTENDS qualifiedIdentifier { COMMA qualifiedIdentifier } ] interfaceBody
142
143 classBody ::= LCURLY { modifiers memberDecl } RCURLY
144
145 interfaceBody ::= LCURLY { modifiers interfaceMemberDecl } RCURLY
146
147 memberDecl ::= IDENTIFIER formalParameters [ THROWS qualifiedIdentifier { COMMA qualifiedIdentifier } ] block
148             | ( VOID | type ) IDENTIFIER formalParameters
149             [ THROWS qualifiedIdentifier { COMMA qualifiedIdentifier } ] ( block | SEMI )
150             | type variableDeclarators SEMI
151
152 interfaceMemberDecl ::= ( VOID | type ) IDENTIFIER formalParameters
153                     [ THROWS qualifiedIdentifier { COMMA qualifiedIdentifier } ] SEMI
154                     | type variableDeclarators SEMI
155
156 block ::= LCURLY { blockStatement } RCURLY
157
158 blockStatement ::= localVariableDeclarationStatement
159                 | statement
160
161 statement ::= block
162            | BREAK SEMI
163            | CONTINUE SEMI
164            | DO statement WHILE parExpression SEMI
165            | FOR LPAREN [ forInit ] SEMI [ expression ] SEMI [ forUpdate ] RPAREN statement
166            | IF parExpression statement [ ELSE statement ]
167            | RETURN [ expression ] SEMI
168            | SEMI
169            | SWITCH parExpression LCURLY { switchBlockStatementGroup } RCURLY
170            | THROW expression SEMI
171            | TRY block { CATCH LPAREN formalParameter RPAREN block } [ FINALLY block ]
172            | WHILE parExpression statement
173            | statementExpression SEMI
174
175 formalParameters ::= LPAREN [ formalParameter { COMMA formalParameter } ] RPAREN
176
177 formalParameter ::= type IDENTIFIER
178
179 parExpression ::= LPAREN expression RPAREN
180
181 forInit ::= statementExpression { COMMA statementExpression }
182         | type variableDeclarators
183
184 forUpdate ::= statementExpression { COMMA statementExpression }
185
186 switchBlockStatementGroup ::= switchLabel { switchLabel } { blockStatement }
187
188 switchLabel ::= CASE expression COLON
189             | DEFLT COLON
190
191 localVariableDeclarationStatement ::= type variableDeclarators SEMI
192
193 variableDeclarators ::= variableDeclarator { COMMA variableDeclarator }
194
195 variableDeclarator ::= IDENTIFIER [ ASSIGN variableInitializer ]
196
197 variableInitializer ::= arrayInitializer | expression
198
199 arrayInitializer ::= LCURLY [ variableInitializer { COMMA variableInitializer } [ COMMA ] ] RCURLY
200
201 arguments ::= LPAREN [ expression { COMMA expression } ] RPAREN
202
203 type ::= basicType | referenceType
204
205 basicType ::= BOOLEAN | CHAR | DOUBLE | INT | LONG
206
207 referenceType ::= basicType LBRACK RBRACK { LBRACK RBRACK }
208                | qualifiedIdentifier { LBRACK RBRACK }
209
210 statementExpression ::= expression
211
212 expression ::= assignmentExpression
213
214 assignmentExpression ::= conditionalExpression
215                      [ ( ALSHIFT_ASSIGN | AND_ASSIGN | ARSHIFT_ASSIGN | ASSIGN | DIV_ASSIGN
216                      | LSHIFT_ASSIGN | MINUS_ASSIGN | OR_ASSIGN | PLUS_ASSIGN | REM_ASSIGN
217                      | STAR_ASSIGN | XOR_ASSIGN ) assignmentExpression ]
218
219 conditionalExpression ::= conditionalOrExpression [ QUESTION expression COLON conditionalExpression ]
220

```

```

221 conditionalOrExpression ::= conditionalAndExpression { LOR conditionalAndExpression }
222
223 conditionalAndExpression ::= inclusiveOrExpression { LAND inclusiveOrExpression }
224
225 inclusiveOrExpression ::= exclusiveOrExpression { OR exclusiveOrExpression }
226
227 exclusiveOrExpression ::= andExpression { XOR andExpression }
228
229 andExpression ::= equalityExpression { AND equalityExpression }
230
231 equalityExpression ::= relationalExpression { ( EQUAL | NOT_EQUAL ) relationalExpression }
232
233 relationalExpression ::= shiftExpression [ ( GE | GT | LE | LT ) shiftExpression | INSTANCEOF referenceType ]
234
235 shiftExpression ::= additiveExpression { ( ALSHIFT | ARSHIFT | LRSIFT ) additiveExpression }
236
237 additiveExpression ::= multiplicativeExpression { ( MINUS | PLUS ) multiplicativeExpression }
238
239 multiplicativeExpression ::= unaryExpression { ( DIV | REM | STAR ) unaryExpression }
240
241 unaryExpression ::= DEC unaryExpression
242                   | INC unaryExpression
243                   | ( MINUS | PLUS ) unaryExpression
244                   | simpleUnaryExpression
245
246 simpleUnaryExpression ::= LNOT unaryExpression
247                       | NOT unaryExpression
248                       | LPAREN basicType RPAREN unaryExpression
249                       | LPAREN referenceType RPAREN simpleUnaryExpression
250                       | postfixExpression
251
252 postfixExpression ::= primary { selector } { DEC | INC }
253
254 selector ::= DOT qualifiedIdentifier [ arguments ]
255           | LBRACK expression RBRACK
256
257 primary ::= parExpression
258          | NEW creator
259          | THIS [ arguments ]
260          | SUPER ( arguments | DOT IDENTIFIER [ arguments ] )
261          | qualifiedIdentifier [ arguments ]
262          | literal
263
264 creator ::= ( basicType | qualifiedIdentifier )
265          ( arguments
266            | LBRACK RBRACK { LBRACK RBRACK } [ arrayInitializer ]
267            | newArrayDeclarator
268            )
269
270 newArrayDeclarator ::= LBRACK [ expression ] RBRACK { LBRACK [ expression ] RBRACK }
271
272 literal ::= CHAR_LITERAL | DOUBLE_LITERAL | FALSE | INT_LITERAL | LONG_LITERAL | NULL | STRING_LITERAL | TRUE

```