C. Java Syntax

C.1 Introduction and Notation

This appendix describes the lexical and syntactic grammars[1] for the Java programming language; the former specifies how individual tokens in the language are composed, and the latter specifies how language constructs are formed. The grammars are based on the second edition of the Java Language Specification.

We employ the following notation to describe the grammars, similar to what we used in Appendix B to specify *j*-- syntax.

- // indicates comments;
- Non-terminals are written in the form of Java (mixed-case) identifiers;
- Terminals are written in **`bold`**;
- Token representations are enclosed in **`<>`**;
- `[`*x*`]` indicates zero or one occurrence of *x*;
- `{`*x*`}` indicates zero or more occurrences of *x*;
- *x*`|`*y* indicates *x* or *y*;
- `~`*x* indicates negation of *x*;
- Parentheses are used for grouping;
- Level numbers in expressions indicate precedence

C.2 Lexical Grammar

C.2.1 White Space

White space in Java is defined as the ASCII space (**`SP`**), horizontal tab (**`HT`**), and form feed (**`FF`**) characters, as well as line terminators: line feed (**`LF`**), carriage return (**`CR`**), and carriage return (**`CR`**) followed by line feed (**`LF`**).

C.2.2 Comments

Java supports two kinds of comments: the traditional comment where all the text from the ASCII characters **`/*`** to the ASCII characters **`*/`** is ignored, and single-line comments where all the text from the ASCII characters **`//`** to the end of the line is ignored.

C.2.3 Reserved Words

The following tokens are reserved for use as keywords in Java and cannot be used as identifiers.

```
abstract   goto      continue      for         new     switch     default
if         package   synchronized  boolean     do      private    this
```

[1]Adapted from ANTLR (http://www.antlr.org) Parser Generator examples.

```
break      double    implements    protected   throw   byte       else
import     public    transient     case        void    return     throws
catch      extends   interface     short       try     char       final
int        static    instanceof    finally     long    strictfp   native
const      float     volatile      super       while   class
```

C.2.4 Operators

The following tokens serve as operators in Java.

```
?      =      ==     !      ~      !=     /      /=     +      +=     ++
-      -=     --     *      *=     %      %=     >>     >>=    >>>    >>>=
>=     >      <<     <<=    <=     <      ^      ^=     |      |=     ||
&      &=     &&
```

C.2.5 Separators

The following tokens serve as separators in Java.

```
,       .       [       {       (       )       }       ]       ;
```

C.2.6 Identifiers

The following regular expression describes identifiers in Java.

```
<identifier> = (a-z | A-Z | _ | $) {a-z | A-Z | _ | 0-9 | $}
```

C.2.7 Literals

An escape (`ESC`) character in Java is a `\` followed `n`, `r`, `t`, `b`, `f`, `'`, `"`, or `\`. An octal escape (`OCTAL_ESC`) -- provided for compatibility with C -- is an octal digit (`0-7`), or an octal digit followed by another octal digit, or one of `0`, `1`, `2` or `3` followed by two octal digits. Besides the `true, false,` and `null` literals, Java supports `int`, `long`, `float`, `double`, `char` and `String` literals as described by the following regular expressions.

```
<int_literal> = 0 | (1-9) {0-9} // decimal
              | 0 (x | X) ((0-9) | (A-F) | (a-f)) {(0-9) | (A-F) | (a-f)} // hexadecimal
              | 0 (0-7) {0-7} // octal
<long_literal> = <int_literal> (l | L)
<float_literal> = (0-9) {0-9} . {0-9} [(e | E) [+ | -] (0-9) {0-9}] [d | D | f | F]
                | . {0-9} [(e | E) [+ | -] (0-9) {0-9}] [d | D | f | F]
                | (0-9) {0-9} [(e | E) [[+ | -] (0-9) {0-9}] (d | D | f | F)
                | (0-9) {0-9} ((e | E) ([+ | -] (0-9) {0-9}) [d | D | f | F]
                | (0x | 0X) . (0-9) | (a-f) | (A-F) {(0-9) | (a-f) | (A-F)}
```

```
                        (p | P) [+ | -] (0-9){0-9} [d | D | f | F] // hexadecimal
                | (0x | 0X) (0-9) | (a-f) | (A-F) {(0-9) | (a-f) | (A-F)}
                        [. {(0-9) | (a-f) | (A-F)}]
                        (p | P) [+ | -] (0-9){0-9} [d | D | f | F] // hexadecimal

<double_literal> = {0-9} [ [ . ] {0-9} [(e | E) [ + | -] (0-9) {0-9} ]] [ d | D ]
<char_literal> = ' (ESC | OCTAL_ESC | ~(' | \ | LF | CR)) '
<string_literal> = " {ESC | OCTAL_ESC | ~(" | \ | LF | CR)} "
```

C.3 Syntactic Grammar

```
compilationUnit ::= [package qualifiedIdentifier ;]
                    {import qualifiedIdentifierStar ;}
                    {typeDeclaration}
                    EOF

qualifiedIdentifier ::= <identifier> {. <identifier>}

qualifiedIdentifierStar ::= <identifier> {. <identifier>} [ . *]

typeDeclaration ::= typeDeclarationModifiers (classDeclaration | interfaceDeclaration)
                  | ;

typeDeclarationModifiers ::= { public | protected | private | static | abstract
                             | final | strictfp }

classDeclaration ::= class <identifier> [extends qualifiedIdentifier]
                     [implements qualifiedIdentifier {, qualifiedIdentifier}]
                       classBody

interfaceDeclaration ::= interface <identifier> // can't be final
                         [extends qualifiedIdentifier {, qualifiedIdentifier}]
                           interfaceBody

modifiers ::= { public | protected | private | static | abstract
            | transient | final | native | threadsafe | synchronized
            | const | volatile | strictfp} // const is reserved, but not valid

classBody ::= { { ;
                | static block
                | block
                | modifiers memberDecl
                }
              }
```

```
interfaceBody ::= { { ;
                    | modifiers interfaceMemberDecl
                    }
                  }

memberDecl ::= classDeclaration // inner class
             | interfaceDeclaration // inner interface
             | <identifier> // constructor
                 formalParameters
                    [throws qualifiedIdentifier {, qualifiedIdentifier}] block
             | (void | type) <identiier> // method
                  formalParameters {[]}
                    [throws qualifiedIdentifier {, qualifiedIdentifier}] (block | ;)
             | type variableDeclarators ; // field

interfaceMemberDecl ::= classDeclaration // inner class
                      | interfaceDeclaration // inner interface
                      | (void | type) <identifier>  // method
                          formalParameters { [ ] }
                             [throws qualifiedIdentifier {, qualifiedIdentifier}] ;
                      | type variableDeclarators ; // fields; must have inits

block ::= { {blockStatement} }

blockStatement ::= localVariableDeclarationStatement
                 | typeDeclarationModifiers classDeclaration
                 | statement

statement ::= block
            | if parExpression statement [else statement]
            | for ( [forInit] ; [expression] ; [forUpdate] ) statement
            | while parExpression statement
            | do statement while parExpression ;
            | try block
                {catch ( formalParameter ) block}
                  [finally block] // must be present if no catches
            | switch parExpression { {switchBlockStatementGroup} }
            | synchronized parExpression block
            | return [expression] ;
            | throw expression ;
            | break [<identifier>] ;
```

| **continue** [**<identifier>**] **;**
| **;**
| **<identifier>** **:** statement
| statementExpression **;**

formalParameters ::= **(** [formalParameter {**,** formalParameter}] **)**

formalParameter ::= [**final**] type **<identifier>** { **[ ]** }

parExpression ::= **(** expression **)**

forInit **::=** statementExpression {**,** statementExpression}
| [**final**] type variableDeclarators

forUpdate **::=** statementExpression {**,** statementExpression}

switchBlockStatementGroup **::=** switchLabel {switchLabel} {blockStatement}

switchLabel **::=** **case** expression **:** // must be constant
| **default** **:**

localVariableDeclarationStatement ::= [**final**] type variableDeclarators **;**

variableDeclarators ::= variableDeclarator {**,** variableDeclarator}

variableDeclarator ::= **<identifier>** { **[ ]** } [**=** variableInitializer]

variableInitializer ::= arrayInitializer | expression

arrayInitializer ::= **{** [variableInitializer {**,** variableInitializer} [**,**]] **}**

arguments ::= **(** [expression {**,** expression}] **)**

type ::= referenceType | basicType

basicType ::= **boolean** | **byte** | **char** | **short** | **int** | **float** | **long** | **double**

referenceType ::= basicType **[ ]** {**[ ]**}
| qualifiedIdentifier {**[ ]**}

statementExpression ::= expression // but must have side-effect, eg **i++**

```
expression ::= assignmentExpression

// level 13
assignmentExpression ::= conditionalExpression // must be a valid lhs
                                [
                                 ( =
                                 | +=
                                 | -=
                                 | *=
                                 | /=
                                 | %=
                                 | >>=
                                 | >>>=
                                 | <<=
                                 | &=
                                 | |=
                                 | ^=
                                 ) assignmentExpression]

// level 12
conditionalExpression ::= conditionalOrExpression
                                 [? assignmentExpression : conditionalExpression]

// level 11
conditionalOrExpression ::= conditionalAndExpression { || conditionalAndExpression }

// level 10
conditionalAndExpression ::= inclusiveOrExpression { && inclusiveOrExpression }

// level 9
inclusiveOrExpression ::= exclusiveOrExpression { | exclusiveOrExpression }

// level 8
exclusiveOrExpression ::= andExpression { ^ andExpression }

// level 7
andExpression ::= equalityExpression { & equalityExpression}

// level 6
equalityExpression ::= relationalExpression { ( == | != ) relationalExpression }

// level 5
relationalExpression ::= shiftExpression
```

```
                        ( { ( < | > | <= | >= ) shiftExpression }
                        | instanceof referenceType )

// level 4
shiftExpression ::= additiveExpression { ( << | >> | >>> ) additiveExpression }

// level 3
additiveExpression ::= multiplicativeExpression { ( + | - ) multiplicativeExpression }

// level 2
multiplicativeExpression ::= unaryExpression { ( * | / | % ) unaryExpression }

// level 1
unaryExpression ::= ++ unaryExpression
                  | -- unaryExpression
                  | ( + | - ) unaryExpression
                  | simpleUnaryExpression

simpleUnaryExpression ::= ~ unaryExpression
                        | ! unaryExpression
                        | ( basicType ) unaryExpression //cast
                        | ( referenceType ) simpleUnaryExpression // cast
                        | postfixExpression

postfixExpression ::= primary {selector} { -- | ++ }

selector ::= . ( <identifier> [arguments]
               | this
               | super superSuffix
               | new innerCreator
               )
           | [ expression ]

superSuffix ::= arguments
              | . <identifier> [arguments]

primary ::= ( assignmentExpression )
          | this [arguments]
          | super superSuffix
          | literal
          | new creator
          | <identifier> {. <identifier>} [identifierSuffix]
          | basicType {[ ]} . class
```

| **void . class**

identifierSuffix **: :=** **[ ]** {**[ ]**} **. class**
| **[** expression **]**
| **.** (**class**
| **this**
| **super** arguments
| **new** innerCreator
)
| arguments

creator **: :=** type
(arguments [classBody]
| newArrayDeclarator [arrayInitializer]
)

newArrayDeclarator **: :=** **[** [expression] **]** { **[** [expression] **]** }

innerCreator ::= **<identifier>** arguments **[**classBody**]**

literal ::= **<int_literal>** | **<char_literal>** | **<string_literal>** | **<float_literal>**
| **<long_literal>** | **<double_literal>** | **true** | **false** | **null**

Further Reading

(Gosling et al, 2000). See Chapter 2 for a detailed description of the lexical and syntactic grammars for the Java language.