Kotlin language specification

Version 1.7-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 1

Syntax and grammar

1.1 Notation

This section uses a BNF-based notation similar to EBNF with the following conventions:

- Any sequence of characters given in single-quotes and monospace font denote a terminal sequence;
- Special terminal sequences that needs specification are given in angle brackets: <...>;
- Normal parentheses are used sparingly to specify priority between other operations;
- A sequence of rules A and B: (A B);
- Choice between rules A and B: (A | B);
- Optional use of rule A: [A];
- Repetition of rule A: {A}.

Rule names starting with capital letters denote lexical rules, while rule names starting with lowercase letters denote syntactic rules.

> Note: this notation is similar to ISO EBNF as per standard ISO/IEC 14977, but does not employ any special symbols for concatenation or termination and does not use some of the additional notation symbols

1.2 Lexical grammar

1.2.1 Whitespace and comments

***LF*:** *<unicode character Line Feed U+000A>*

CR:
<unicode character Carriage Return U+000D>

ShebangLine:
`'#!'` {*<any character excluding CR and LF >*}

DelimitedComment:
`'/*'` { *DelimitedComment* | *<any character>* } `'*/'`

LineComment:
`'//'` {*<any character excluding CR and LF >*}

WS:
<one of the following characters: SPACE U+0020, TAB U+0009, Form Feed U+000C>

NL: *LF* | (*CR* [*LF*])

Hidden:
DelimitedComment | *LineComment* | *WS*

1.2.2 Keywords and operators

RESERVED:
`'...'`

DOT:
`'.'`

COMMA:
`','`

LPAREN:
`'('`

RPAREN:
`')'`

LSQUARE:
`'['`

RSQUARE:
`']'`

LCURL:
`'{'`

RCURL:
`'}'`

MULT:
`'*'`

***MOD*:**
'%'

***DIV*:**
'/'

***ADD*:**
'+'

***SUB*:**
'-'

***INCR*:**
'++'

***DECR*:**
'--'

***CONJ*:**
'&&'

***DISJ*:**
'||'

***EXCL_WS*:**
'!' *Hidden*

***EXCL_NO_WS*:**
'!'

***COLON*:**
':'

***SEMICOLON*:**
';'

***ASSIGNMENT*:**
'='

***ADD_ASSIGNMENT*:**
'+='

***SUB_ASSIGNMENT*:**
'-='

***MULT_ASSIGNMENT*:**
'*='

***DIV_ASSIGNMENT*:**
'/='

***MOD_ASSIGNMENT*:**
'%='

ARROW:
'->'

DOUBLE_ARROW:
'=>'

RANGE:
'..'

COLONCOLON:
'::'

DOUBLE_SEMICOLON:
';;'

HASH:
'#'

AT_NO_WS:
'@'

AT_POST_WS:
'@' (*Hidden* | *NL*)

AT_PRE_WS:
(*Hidden* | *NL*) '@'

AT_BOTH_WS:
(*Hidden* | *NL*) '@' (*Hidden* | *NL*)

QUEST_WS:
'?' *Hidden*

QUEST_NO_WS:
'?'

LANGLE:
'<'

RANGLE:
'>'

LE: '<='

GE:
'>='

EXCL_EQ:
'!='

EXCL_EQEQ:
'!=='

AS_SAFE:
'as?'

EQEQ:
    `'=='`

EQEQEQ:
    `'==='`

SINGLE_QUOTE:
    `'\''`

RETURN_AT:
    `'return@'` *Identifier*

CONTINUE_AT:
    `'continue@'` *Identifier*

BREAK_AT:
    `'break@'` *Identifier*

THIS_AT:
    `'this@'` *Identifier*

SUPER_AT:
    `'super@'` *Identifier*

FILE:
    `'file'`

FIELD:
    `'field'`

PROPERTY:
    `'property'`

GET:
    `'get'`

SET:
    `'set'`

RECEIVER:
    `'receiver'`

PARAM:
    `'param'`

SETPARAM:
    `'setparam'`

DELEGATE:
    `'delegate'`

PACKAGE:
    `'package'`

IMPORT:
    `'import'`

CLASS:
    `'class'`

INTERFACE:
    `'interface'`

FUN:
    `'fun'`

OBJECT:
    `'object'`

VAL:
    `'val'`

VAR:
    `'var'`

TYPE_ALIAS:
    `'typealias'`

CONSTRUCTOR:
    `'constructor'`

BY:
    `'by'`

COMPANION:
    `'companion'`

INIT:
    `'init'`

THIS:
    `'this'`

SUPER:
    `'super'`

TYPEOF:
    `'typeof'`

WHERE:
    `'where'`

IF: `'if'`

ELSE:
    `'else'`

WHEN:
    `'when'`

TRY:
    `'try'`

CATCH:
    `'catch'`

FINALLY:
    `'finally'`

FOR:
    `'for'`

DO:
    `'do'`

WHILE:
    `'while'`

THROW:
    `'throw'`

RETURN:
    `'return'`

CONTINUE:
    `'continue'`

BREAK:
    `'break'`

AS: `'as'`

IS: `'is'`

IN: `'in'`

NOT_IS:
    `'!is'` (*Hidden* | *NL*)

NOT_IN:
    `'!in'` (*Hidden* | *NL*)

OUT:
    `'out'`

DYNAMIC:
    `'dynamic'`

PUBLIC:
    `'public'`

PRIVATE:
    `'private'`

PROTECTED:
    `'protected'`

INTERNAL:
    `'internal'`

ENUM:
    `'enum'`

SEALED:
    `'sealed'`

ANNOTATION:
    `'annotation'`

DATA:
    `'data'`

INNER:
    `'inner'`

TAILREC:
    `'tailrec'`

OPERATOR:
    `'operator'`

INLINE:
    `'inline'`

INFIX:
    `'infix'`

EXTERNAL:
    `'external'`

SUSPEND:
    `'suspend'`

OVERRIDE:
    `'override'`

ABSTRACT:
    `'abstract'`

FINAL:
    `'final'`

OPEN:
    `'open'`

CONST:
    `'const'`

LATEINIT:
    `'lateinit'`

VARARG:
    `'vararg'`

NOINLINE:
    `'noinline'`

CROSSINLINE:
    `'crossinline'`

REIFIED:
    `'reified'`

EXPECT:
    `'expect'`

ACTUAL:
    `'actual'`

1.2.3 Literals

DecDigitNoZero:
    `'1'` | `'2'` | `'3'` | `'4'` | `'5'` | `'6'` | `'7'` | `'8'` | `'9'`

DecDigit:
    `'0'` | `'1'` | `'2'` | `'3'` | `'4'` | `'5'` | `'6'` | `'7'` | `'8'` | `'9'`

DecDigitOrSeparator:
    DecDigit | `'_'`

DecDigits:
    DecDigit {*DecDigitOrSeparator*} *DecDigit*
    | *DecDigit*

DoubleExponent:
    (`'e'` | `'E'`) [(`'+'` | `'-'`)] *DecDigits*

RealLiteral:
    FloatLiteral | *DoubleLiteral*

FloatLiteral:
    DoubleLiteral (`'f'` | `'F'`)
    | *DecDigits* (`'f'` | `'F'`)

DoubleLiteral:
    [*DecDigits*] `'.'` *DecDigits* [*DoubleExponent*]
    | [*DecDigits*] [*DoubleExponent*]

IntegerLiteral:
    DecDigitNoZero {*DecDigitOrSeparator*} *DecDigit*
    | *DecDigit*

***HexDigit*:**
DecDigit
| 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
| 'a' | 'b' | 'c' | 'd' | 'e' | 'f'

***HexDigitOrSeparator*:**
HexDigit | '_'

HexLiteral
'0' ('x' | 'X') *HexDigit* {*HexDigitOrSeparator*} *HexDigit*
| '0' ('x' | 'X') *HexDigit*

BinDigit
'0' | '1'

BinDigitOrSeparator
BinDigit | '_'

BinLiteral
'0' ('b' | 'B') *BinDigit* {*BinDigitOrSeparator*} *BinDigit*
| '0' ('b' | 'B') *BinDigit*

UnsignedLiteral
(*IntegerLiteral* | *HexLiteral* | *BinLiteral*) ('u' | 'U') ['L']

LongLiteral
(*IntegerLiteral* | *HexLiteral* | *BinLiteral*) 'L'

BooleanLiteral
'true' | 'false'

NullLiteral
'null'

CharacterLiteral
''' (*EscapeSeq* | <any character excluding CR, LF, ''' or '\'>) '''

UniCharacterLiteral
'\' 'u' *HexDigit HexDigit HexDigit HexDigit*

EscapedIdentifier
'\' ('t' | 'b' | 'r' | 'n' | ''' | '"' | '\' | '$')

EscapeSeq
UniCharacterLiteral | *EscapedIdentifier*

1.2.4 Identifiers

Letter
<any unicode character of categories Lu, Ll, Lt, Lm or Lo>

QuotedSymbol
<any character excluding CR, LF and '`'>

UnicodeDigit
<any unicode character of category Nd>

Identifier
(*Letter* | '_') {*Letter* | '_' | *UnicodeDigit*}
| '`' *QuotedSymbol* {*QuotedSymbol*} '`'

Kotlin supports *escaping* identifiers by enclosing any sequence of characters into backtick (`) characters, allowing to use any name as an identifier. This allows not only using non-alphanumeric characters (like `@` or `#`) in names, but also using keywords like `if` or `when` as identifiers. Actual set of characters that is allowed to be escaped may, however, be a subject to platform restrictions. Consult particular platform sections for details.

> Note: for example, the following characters are not allowed in identifiers used as declaration names on the JVM platform even when escaped due to JVM restrictions: `.`, `;`, `[`, `]`, `/`, `<`, `>`, `:`, `\\` .

Escaped identifiers are treated the same as corresponding non-escaped identifier if it is allowed. For example, an escaped identifier `` `foo` `` and non-escaped identifier `foo` may be used interchangeably and refer to the same program entity.

IdentifierOrSoftKey
Identifier
| *ABSTRACT*
| *ANNOTATION*
| *BY*
| *CATCH*
| *COMPANION*
| *CONSTRUCTOR*
| *CROSSINLINE*
| *DATA*
| *DYNAMIC*
| *ENUM*
| *EXTERNAL*
| *FINAL*
| *FINALLY*
| *IMPORT*
| *INFIX*
| *INIT*
| *INLINE*
| *INNER*
| *INTERNAL*
| *LATEINIT*
| *NOINLINE*
| *OPEN*
| *OPERATOR*
| *OUT*

| *OVERRIDE*
| *PRIVATE*
| *PROTECTED*
| *PUBLIC*
| *REIFIED*
| *SEALED*
| *TAILREC*
| *VARARG*
| *WHERE*
| *GET*
| *SET*
| *FIELD*
| *PROPERTY*
| *RECEIVER*
| *PARAM*
| *SETPARAM*
| *DELEGATE*
| *FILE*
| *EXPECT*
| *ACTUAL*
| *CONST*
| *SUSPEND*

Some of the keywords used in Kotlin are allowed to be used as identifiers even when not escaped. Such keywords are called *soft keywords*. You can see the complete list of soft keyword in the rule above. All other keywords are considered *hard keywords* and may only be used as identifiers if escaped.

Note: for example, this is a valid property declaration in Kotlin:

```
val field = 2
```

even though `field` is a keyword

1.2.5 String mode grammar

QUOTE_OPEN
`'"'`

TRIPLE_QUOTE_OPEN
`'"""'`

FieldIdentifier
`'$'` *IdentifierOrSoftKey*

Opening a double quote (QUOTE_OPEN) rule puts the lexical grammar into the special “line string” mode with the following rules. Closing double quote (QUOTE_CLOSE) rule exits this mode.

QUOTE_CLOSE
'"'

LineStrRef
FieldIdentifier

LineStrText
{<any character except '\', '"' or '$'>} | '$'

LineStrEscapedChar
EscapedIdentifier | *UniCharacterLiteral*

LineStrExprStart
'${'

Opening a triple double quote (TRIPLE_QUOTE_OPEN) rule puts the lexical grammar into the special "multiline string" mode with the following rules. Closing triple double quote (TRIPLE_QUOTE_CLOSE) rule exits this mode.

TRIPLE_QUOTE_CLOSE
[*MultilineStringQuote*] '"""'

MultilineStringQuote
'"""' {'"'}

MultiLineStrRef
FieldIdentifier

MultiLineStrText
{<any character except '"' or '$'>} | '$'

MultiLineStrExprStart
'${'

1.2.6 Tokens

These are all the valid tokens in one rule. Note that syntax grammar ignores tokens *DelimitedComment*, *LineComment* and *WS*.

KotlinToken
ShebangLine
| *DelimitedComment*
| *LineComment*
| *WS*
| *NL*
| *RESERVED*
| *DOT*
| *COMMA*
| *LPAREN*
| *RPAREN*
| *LSQUARE*

| *RSQUARE*
| *LCURL*
| *RCURL*
| *MULT*
| *MOD*
| *DIV*
| *ADD*
| *SUB*
| *INCR*
| *DECR*
| *CONJ*
| *DISJ*
| *EXCL_WS*
| *EXCL_NO_WS*
| *COLON*
| *SEMICOLON*
| *ASSIGNMENT*
| *ADD_ASSIGNMENT*
| *SUB_ASSIGNMENT*
| *MULT_ASSIGNMENT*
| *DIV_ASSIGNMENT*
| *MOD_ASSIGNMENT*
| *ARROW*
| *DOUBLE_ARROW*
| *RANGE*
| *COLONCOLON*
| *DOUBLE_SEMICOLON*
| *HASH*
| *AT_NO_WS*
| *AT_POST_WS*
| *AT_PRE_WS*
| *AT_BOTH_WS*
| *QUEST_WS*
| *QUEST_NO_WS*
| *LANGLE*
| *RANGLE*
| *LE*
| *GE*
| *EXCL_EQ*
| *EXCL_EQEQ*
| *AS_SAFE*
| *EQEQ*
| *EQEQEQ*
| *SINGLE_QUOTE*
| *RETURN_AT*
| *CONTINUE_AT*

| *BREAK_AT*
| *THIS_AT*
| *SUPER_AT*
| *FILE*
| *FIELD*
| *PROPERTY*
| *GET*
| *SET*
| *RECEIVER*
| *PARAM*
| *SETPARAM*
| *DELEGATE*
| *PACKAGE*
| *IMPORT*
| *CLASS*
| *INTERFACE*
| *FUN*
| *OBJECT*
| *VAL*
| *VAR*
| *TYPE_ALIAS*
| *CONSTRUCTOR*
| *BY*
| *COMPANION*
| *INIT*
| *THIS*
| *SUPER*
| *TYPEOF*
| *WHERE*
| *IF*
| *ELSE*
| *WHEN*
| *TRY*
| *CATCH*
| *FINALLY*
| *FOR*
| *DO*
| *WHILE*
| *THROW*
| *RETURN*
| *CONTINUE*
| *BREAK*
| *AS*
| *IS*
| *IN*
| *NOT_IS*

| *NOT_IN*
| *OUT*
| *DYNAMIC*
| *PUBLIC*
| *PRIVATE*
| *PROTECTED*
| *INTERNAL*
| *ENUM*
| *SEALED*
| *ANNOTATION*
| *DATA*
| *INNER*
| *TAILREC*
| *OPERATOR*
| *INLINE*
| *INFIX*
| *EXTERNAL*
| *SUSPEND*
| *OVERRIDE*
| *ABSTRACT*
| *FINAL*
| *OPEN*
| *CONST*
| *LATEINIT*
| *VARARG*
| *NOINLINE*
| *CROSSINLINE*
| *REIFIED*
| *EXPECT*
| *ACTUAL*
| *Identifier*
| *RealLiteral*
| *IntegerLiteral*
| *HexLiteral*
| *BinLiteral*
| *LongLiteral*
| *BooleanLiteral*
| *NullLiteral*
| *CharacterLiteral*
| *QUOTE_OPEN*
| *QUOTE_CLOSE*
| *TRIPLE_QUOTE_OPEN*
| *TRIPLE_QUOTE_CLOSE*
| *LineStrRef*
| *LineStrText*
| *LineStrEscapedChar*

| *LineStrExprStart*
| *MultilineStringQuote*
| *MultiLineStrRef*
| *MultiLineStrText*
| *MultiLineStrExprStart*

EOF
<end of input>

1.3 Syntax grammar

The grammar below replaces some lexical grammar rules with explicit literals (where such replacement in trivial and always correct, for example, for keywords) for better readability.

***kotlinFile*:**
[*shebangLine*]
{*NL*}
{*fileAnnotation*}
packageHeader
importList
{*topLevelObject*}
EOF

***script*:**
[*shebangLine*]
{*NL*}
{*fileAnnotation*}
packageHeader
importList
{*statement semi*}
EOF

***shebangLine*:**
ShebangLine (*NL* {*NL*})

***fileAnnotation*:**
(*AT_NO_WS* | *AT_PRE_WS*)
`'file'`
{*NL*}
`':'`
{*NL*}
((`'['` (*unescapedAnnotation* {*unescapedAnnotation*}) `']'`) | *unescapedAnnotation*)
{*NL*}

***packageHeader*:**
[`'package'` *identifier* [*semi*]]

importList:
    {*importHeader*}

importHeader:
    `'import'` *identifier* [(`'.'` `'*'`) | *importAlias*] [*semi*]

importAlias:
    `'as'` *simpleIdentifier*

topLevelObject:
    declaration [*semis*]

typeAlias:
    [*modifiers*]
    `'typealias'`
    {*NL*}
    simpleIdentifier
    [{*NL*} *typeParameters*]
    {*NL*}
    `'='`
    {*NL*}
    type

declaration:
    classDeclaration
    | *objectDeclaration*
    | *functionDeclaration*
    | *propertyDeclaration*
    | *typeAlias*

classDeclaration:
    [*modifiers*]
    (`'class'` | ([`'fun'` {*NL*}] `'interface'`))
    {*NL*}
    simpleIdentifier
    [{*NL*} *typeParameters*]
    [{*NL*} *primaryConstructor*]
    [{*NL*} `':'` {*NL*} *delegationSpecifiers*]
    [{*NL*} *typeConstraints*]
    [({*NL*} *classBody*) | ({*NL*} *enumClassBody*)]

primaryConstructor:
    [[*modifiers*] `'constructor'` {*NL*}] *classParameters*

classBody:
    `'{'`
    {*NL*}
    classMemberDeclarations
    {*NL*}
    `'}'`

***classParameters*:**
'('
{*NL*}
[*classParameter* {{*NL*} ',' {*NL*} *classParameter*} [{*NL*} ',']]
{*NL*}
')'

***classParameter*:**
[*modifiers*]
['val' | 'var']
{*NL*}
simpleIdentifier
':'
{*NL*}
type
[{*NL*} '=' {*NL*} *expression*]

***delegationSpecifiers*:**
annotatedDelegationSpecifier {{*NL*} ',' {*NL*} *annotatedDelegationSpecifier*}

***delegationSpecifier*:**
constructorInvocation
| *explicitDelegation*
| *userType*
| *functionType*
| ('suspend' {*NL*} *functionType*)

***constructorInvocation*:**
userType valueArguments

***annotatedDelegationSpecifier*:**
{*annotation*} {*NL*} *delegationSpecifier*

***explicitDelegation*:**
(*userType* | *functionType*)
{*NL*}
'by'
{*NL*}
expression

***typeParameters*:**
'<'
{*NL*}
typeParameter
{{*NL*} ',' {*NL*} *typeParameter*}
[{*NL*} ',']
{*NL*}
'>'

***typeParameter*:**
[*typeParameterModifiers*] {*NL*} *simpleIdentifier* [{*NL*} `':'` {*NL*} *type*]

***typeConstraints*:**
`'where'` {*NL*} *typeConstraint* {{*NL*} `','` {*NL*} *typeConstraint*}

***typeConstraint*:**
{*annotation*}
simpleIdentifier
{*NL*}
`':'`
{*NL*}
type

***classMemberDeclarations*:**
{*classMemberDeclaration* [*semis*]}

***classMemberDeclaration*:**
declaration
| *companionObject*
| *anonymousInitializer*
| *secondaryConstructor*

***anonymousInitializer*:**
`'init'` {*NL*} *block*

***companionObject*:**
[*modifiers*]
`'companion'`
{*NL*}
`'object'`
[{*NL*} *simpleIdentifier*]
[{*NL*} `':'` {*NL*} *delegationSpecifiers*]
[{*NL*} *classBody*]

***functionValueParameters*:**
`'('`
{*NL*}
[*functionValueParameter* {{*NL*} `','` {*NL*} *functionValueParameter*} [{*NL*}
`','`]]
{*NL*}
`')'`

***functionValueParameter*:**
[*parameterModifiers*] *parameter* [{*NL*} `'='` {*NL*} *expression*]

***functionDeclaration*:**
[*modifiers*]
`'fun'`
[{*NL*} *typeParameters*]

[{*NL*} *receiverType* {*NL*} '.']
{*NL*}
simpleIdentifier
{*NL*}
functionValueParameters
[{*NL*} ':' {*NL*} *type*]
[{*NL*} *typeConstraints*]
[{*NL*} *functionBody*]

***functionBody*:**
block
| ('=' {*NL*} *expression*)

***variableDeclaration*:**
{*annotation*} {*NL*} *simpleIdentifier* [{*NL*} ':' {*NL*} *type*]

***multiVariableDeclaration*:**
'('
{*NL*}
variableDeclaration
{{*NL*} ',' {*NL*} *variableDeclaration*}
[{*NL*} ',']
{*NL*}
')'

***propertyDeclaration*:**
[*modifiers*]
('val' | 'var')
[{*NL*} *typeParameters*]
[{*NL*} *receiverType* {*NL*} '.']
({*NL*} (*multiVariableDeclaration* | *variableDeclaration*))
[{*NL*} *typeConstraints*]
[{*NL*} (('=' {*NL*} *expression*) | *propertyDelegate*)]
[(*NL* {*NL*}) ';']
{*NL*}
(([*getter*] [{*NL*} [*semi*] *setter*]) | ([*setter*] [{*NL*} [*semi*] *getter*]))

***propertyDelegate*:**
'by' {*NL*} *expression*

***getter*:**
[*modifiers*] 'get' [{*NL*} '(' {*NL*} ')' [{*NL*} ':' {*NL*} *type*] {*NL*} *functionBody*]

***setter*:**
[*modifiers*] 'set' [{*NL*} '(' {*NL*} *functionValueParameterWithOptionalType* [{*NL*} ','] {*NL*} ')' [{*NL*} ':' {*NL*} *type*] {*NL*} *functionBody*]

***parametersWithOptionalType*:**
'('

{*NL*}
[*functionValueParameterWithOptionalType* {{*NL*} ',' {*NL*} *functionValueParameterWithOptionalType*} [{*NL*} ',']]
{*NL*}
')'

***functionValueParameterWithOptionalType*:**
[*parameterModifiers*] *parameterWithOptionalType* [{*NL*} '=' {*NL*} *expression*]

***parameterWithOptionalType*:**
simpleIdentifier {*NL*} [':' {*NL*} *type*]

***parameter*:**
simpleIdentifier
{*NL*}
':'
{*NL*}
type

***objectDeclaration*:**
[*modifiers*]
'object'
{*NL*}
simpleIdentifier
[{*NL*} ':' {*NL*} *delegationSpecifiers*]
[{*NL*} *classBody*]

***secondaryConstructor*:**
[*modifiers*]
'constructor'
{*NL*}
functionValueParameters
[{*NL*} ':' {*NL*} *constructorDelegationCall*]
{*NL*}
[*block*]

***constructorDelegationCall*:**
('this' | 'super') {*NL*} *valueArguments*

***enumClassBody*:**
'{'
{*NL*}
[*enumEntries*]
[{*NL*} ';' {*NL*} *classMemberDeclarations*]
{*NL*}
'}'

***enumEntries*:**
enumEntry {{*NL*} ',' {*NL*} *enumEntry*} {*NL*} [',']

***enumEntry*:**
[*modifiers* {*NL*}] *simpleIdentifier* [{*NL*} *valueArguments*] [{*NL*} *classBody*]

***type*:**
[*typeModifiers*] (*functionType* | *parenthesizedType* | *nullableType* | *typeReference* | *definitelyNonNullableType*)

***typeReference*:**
userType
| `'dynamic'`

***nullableType*:**
(*typeReference* | *parenthesizedType*) {*NL*} (*quest* {*quest*})

***quest*:**
QUEST_NO_WS
| *QUEST_WS*

***userType*:**
simpleUserType {{*NL*} `'.'` {*NL*} *simpleUserType*}

***simpleUserType*:**
simpleIdentifier [{*NL*} *typeArguments*]

***typeProjection*:**
([*typeProjectionModifiers*] *type*)
| `'*'`

***typeProjectionModifiers*:**
typeProjectionModifier {*typeProjectionModifier*}

***typeProjectionModifier*:**
(*varianceModifier* {*NL*})
| *annotation*

***functionType*:**
[*receiverType* {*NL*} `'.'` {*NL*}]
functionTypeParameters
{*NL*}
`'->'`
{*NL*}
type

***functionTypeParameters*:**
`'('`
{*NL*}
[*parameter* | *type*]
{{*NL*} `','` {*NL*} (*parameter* | *type*)}
[{*NL*} `','`]
{*NL*}
`')'`

***parenthesizedType*:**
'('
{*NL*}
type
{*NL*}
')'

***receiverType*:**
[*typeModifiers*] (*parenthesizedType* | *nullableType* | *typeReference*)

***parenthesizedUserType*:**
'('
{*NL*}
(*userType* | *parenthesizedUserType*)
{*NL*}
')'

***definitelyNonNullableType*:**
[*typeModifiers*]
(*userType* | *parenthesizedUserType*)
{*NL*}
'&'
{*NL*}
[*typeModifiers*]
(*userType* | *parenthesizedUserType*)

***statements*:**
[*statement* {*semis statement*}] [*semis*]

***statement*:**
{*label* | *annotation*} (*declaration* | *assignment* | *loopStatement* | *expression*)

***label*:**
simpleIdentifier (*AT_NO_WS* | *AT_POST_WS*) {*NL*}

***controlStructureBody*:**
block
| *statement*

***block*:**
'{'
{*NL*}
statements
{*NL*}
'}'

***loopStatement*:**
forStatement
| *whileStatement*
| *doWhileStatement*

***forStatement*:**
'for'
{*NL*}
'('
{*annotation*}
(*variableDeclaration* | *multiVariableDeclaration*)
'in'
expression
')'
{*NL*}
[*controlStructureBody*]

***whileStatement*:**
'while'
{*NL*}
'('
expression
')'
{*NL*}
(*controlStructureBody* | ';')

***doWhileStatement*:**
'do'
{*NL*}
[*controlStructureBody*]
{*NL*}
'while'
{*NL*}
'('
expression
')'

***assignment*:**
((*directlyAssignableExpression* '=') | (*assignableExpression assignmentAndOperator*)) {*NL*} *expression*

***semi*:**
(';' | *NL*) {*NL*}

***semis*:**
';' | *NL* {';' | *NL*}

***expression*:**
disjunction

***disjunction*:**
conjunction {{*NL*} '||' {*NL*} *conjunction*}

***conjunction*:**
equality {{*NL*} '&&' {*NL*} *equality*}

***equality*:**
comparison {*equalityOperator* {*NL*} *comparison*}

***comparison*:**
genericCallLikeComparison {*comparisonOperator* {*NL*} *genericCallLikeComparison*}

***genericCallLikeComparison*:**
infixOperation {*callSuffix*}

***infixOperation*:**
elvisExpression {(*inOperator* {*NL*} *elvisExpression*) | (*isOperator* {*NL*} *type*)}

***elvisExpression*:**
infixFunctionCall {{*NL*} *elvis* {*NL*} *infixFunctionCall*}

***elvis*:**
QUEST_NO_WS ':'

***infixFunctionCall*:**
rangeExpression {*simpleIdentifier* {*NL*} *rangeExpression*}

***rangeExpression*:**
additiveExpression {'..' {*NL*} *additiveExpression*}

***additiveExpression*:**
multiplicativeExpression {*additiveOperator* {*NL*} *multiplicativeExpression*}

***multiplicativeExpression*:**
asExpression {*multiplicativeOperator* {*NL*} *asExpression*}

***asExpression*:**
prefixUnaryExpression {{*NL*} *asOperator* {*NL*} *type*}

***prefixUnaryExpression*:**
{*unaryPrefix*} *postfixUnaryExpression*

***unaryPrefix*:**
annotation
| *label*
| (*prefixUnaryOperator* {*NL*})

***postfixUnaryExpression*:**
primaryExpression {*postfixUnarySuffix*}

***postfixUnarySuffix*:**
postfixUnaryOperator
| *typeArguments*
| *callSuffix*
| *indexingSuffix*
| *navigationSuffix*

***directlyAssignableExpression*:**
(*postfixUnaryExpression assignableSuffix*)
| *simpleIdentifier*
| *parenthesizedDirectlyAssignableExpression*

***parenthesizedDirectlyAssignableExpression*:**
'('
{*NL*}
directlyAssignableExpression
{*NL*}
')'

***assignableExpression*:**
prefixUnaryExpression
| *parenthesizedAssignableExpression*

***parenthesizedAssignableExpression*:**
'('
{*NL*}
assignableExpression
{*NL*}
')'

***assignableSuffix*:**
typeArguments
| *indexingSuffix*
| *navigationSuffix*

***indexingSuffix*:**
'['
{*NL*}
expression
{{*NL*} ',' {*NL*} *expression*}
[{*NL*} ',']
{*NL*}
']'

***navigationSuffix*:**
memberAccessOperator {*NL*} (*simpleIdentifier* | *parenthesizedExpression* |
'class')

***callSuffix*:**
[*typeArguments*] (([*valueArguments*] *annotatedLambda*) | *valueArguments*)

***annotatedLambda*:**
{*annotation*} [*label*] {*NL*} *lambdaLiteral*

***typeArguments*:**
'<'
{*NL*}

typeProjection
{{*NL*} ',' {*NL*} *typeProjection*}
[{*NL*} ',']
{*NL*}
'>'

***valueArguments*:**
'(' {*NL*} [*valueArgument* {{*NL*} ',' {*NL*} *valueArgument*} [{*NL*} ',']
{*NL*}] ')'

***valueArgument*:**
[*annotation*]
{*NL*}
[*simpleIdentifier* {*NL*} '=' {*NL*}]
['*']
{*NL*}
expression

***primaryExpression*:**
parenthesizedExpression
| *simpleIdentifier*
| *literalConstant*
| *stringLiteral*
| *callableReference*
| *functionLiteral*
| *objectLiteral*
| *collectionLiteral*
| *thisExpression*
| *superExpression*
| *ifExpression*
| *whenExpression*
| *tryExpression*
| *jumpExpression*

***parenthesizedExpression*:**
'('
{*NL*}
expression
{*NL*}
')'

***collectionLiteral*:**
'[' {*NL*} [*expression* {{*NL*} ',' {*NL*} *expression*} [{*NL*} ','] {*NL*}] ']'

***literalConstant*:**
BooleanLiteral
| *IntegerLiteral*
| *HexLiteral*
| *BinLiteral*

| *CharacterLiteral*
| *RealLiteral*
| 'null'
| *LongLiteral*
| *UnsignedLiteral*

stringLiteral:
lineStringLiteral
| *multiLineStringLiteral*

lineStringLiteral:
'"' {*lineStringContent* | *lineStringExpression*} '"'

multiLineStringLiteral:
'"""' {*multiLineStringContent* | *multiLineStringExpression* | '"'}
TRIPLE_QUOTE_CLOSE

lineStringContent:
LineStrText
| *LineStrEscapedChar*
| *LineStrRef*

lineStringExpression:
'${'
{*NL*}
expression
{*NL*}
'}'

multiLineStringContent:
MultiLineStrText
| '"'
| *MultiLineStrRef*

multiLineStringExpression:
'${'
{*NL*}
expression
{*NL*}
'}'

lambdaLiteral:
'{'
{*NL*}
[[*lambdaParameters*] {*NL*} '->' {*NL*}]
statements
{*NL*}
'}'

lambdaParameters:
lambdaParameter {{*NL*} ',' {*NL*} *lambdaParameter*} [{*NL*} ',']

lambdaParameter:
variableDeclaration
| (*multiVariableDeclaration* [{*NL*} ':' {*NL*} *type*])

anonymousFunction:
'fun'
[{*NL*} *type* {*NL*} '.']
{*NL*}
parametersWithOptionalType
[{*NL*} ':' {*NL*} *type*]
[{*NL*} *typeConstraints*]
[{*NL*} *functionBody*]

functionLiteral:
lambdaLiteral
| *anonymousFunction*

objectLiteral:
'object' [{*NL*} ':' {*NL*} *delegationSpecifiers* {*NL*}] [{*NL*} *classBody*]

thisExpression:
'this'
| *THIS_AT*

superExpression:
('super' ['<' {*NL*} *type* {*NL*} '>'] [*AT_NO_WS simpleIdentifier*])
| *SUPER_AT*

ifExpression:
'if'
{*NL*}
'('
{*NL*}
expression
{*NL*}
')'
{*NL*}
(*controlStructureBody* | ([*controlStructureBody*] {*NL*} [';'] {*NL*} 'else'
{*NL*} (*controlStructureBody* | ';')) | ';')

whenSubject:
'(' [{*annotation*} {*NL*} 'val' {*NL*} *variableDeclaration* {*NL*} '=' {*NL*}]
expression ')'

whenExpression:
'when'
{*NL*}

[*whenSubject*]
{*NL*}
'{'
{*NL*}
{*whenEntry* {*NL*}}
{*NL*}
'}'

whenEntry:
(*whenCondition* {{*NL*} ',' {*NL*} *whenCondition*} [{*NL*} ','] {*NL*} '->'
{*NL*} *controlStructureBody* [*semi*])
| ('else' {*NL*} '->' {*NL*} *controlStructureBody* [*semi*])

whenCondition:
expression
| *rangeTest*
| *typeTest*

rangeTest:
inOperator {*NL*} *expression*

typeTest:
isOperator {*NL*} *type*

tryExpression:
'try' {*NL*} *block* ((({*NL*} *catchBlock* {{*NL*} *catchBlock*}) [{*NL*} *finallyBlock*]) | ({*NL*} *finallyBlock*))

catchBlock:
'catch'
{*NL*}
'('
{*annotation*}
simpleIdentifier
':'
type
[{*NL*} ',']
')'
{*NL*}
block

finallyBlock:
'finally' {*NL*} *block*

jumpExpression:
('throw' {*NL*} *expression*)
| (('return' | *RETURN_AT*) [*expression*])
| 'continue'
| *CONTINUE_AT*

| 'break'
| *BREAK_AT*

callableReference:
[*receiverType*] '::' {*NL*} (*simpleIdentifier* | 'class')

assignmentAndOperator:
'+='
| '-='
| '*='
| '/='
| '%='

equalityOperator:
'!='
| '!=='
| '=='
| '==='

comparisonOperator:
'<'
| '>'
| '<='
| '>='

inOperator:
'in'
| *NOT_IN*

isOperator:
'is'
| *NOT_IS*

additiveOperator:
'+'
| '-'

multiplicativeOperator:
'*'
| '/'
| '%'

asOperator:
'as'
| 'as?'

prefixUnaryOperator:
'++'
| '--'
| '-'

| '+'
| *excl*

***postfixUnaryOperator*:**
'++'
| '--'
| ('!' *excl*)

***excl*:**
'!'
| *EXCL_WS*

***memberAccessOperator*:**
({*NL*} '.')
| ({*NL*} *safeNav*)
| '::'

***safeNav*:**
QUEST_NO_WS '.'

***modifiers*:**
annotation | *modifier* {*annotation* | *modifier*}

***parameterModifiers*:**
annotation | *parameterModifier* {*annotation* | *parameterModifier*}

***modifier*:**
(*classModifier* | *memberModifier* | *visibilityModifier* | *functionModifier* | *propertyModifier* | *inheritanceModifier* | *parameterModifier* | *platformModifier*)
{*NL*}

***typeModifiers*:**
typeModifier {*typeModifier*}

***typeModifier*:**
annotation
| ('suspend' {*NL*})

***classModifier*:**
'enum'
| 'sealed'
| 'annotation'
| 'data'
| 'inner'
| 'value'

***memberModifier*:**
'override'
| 'lateinit'

***visibilityModifier*:**
'public'

| 'private'
| 'internal'
| 'protected'

***varianceModifier*:**
'in'
| 'out'

***typeParameterModifiers*:**
typeParameterModifier {*typeParameterModifier*}

***typeParameterModifier*:**
(*reificationModifier* {*NL*})
| (*varianceModifier* {*NL*})
| *annotation*

***functionModifier*:**
'tailrec'
| 'operator'
| 'infix'
| 'inline'
| 'external'
| 'suspend'

***propertyModifier*:**
'const'

***inheritanceModifier*:**
'abstract'
| 'final'
| 'open'

***parameterModifier*:**
'vararg'
| 'noinline'
| 'crossinline'

***reificationModifier*:**
'reified'

***platformModifier*:**
'expect'
| 'actual'

***annotation*:**
(*singleAnnotation* | *multiAnnotation*) {*NL*}

***singleAnnotation*:**
((*annotationUseSiteTarget* {*NL*}) | *AT_NO_WS* | *AT_PRE_WS*) *unescapedAnnotation*

***multiAnnotation*:**
((*annotationUseSiteTarget* {*NL*}) | *AT_NO_WS* | *AT_PRE_WS*) `'['`
(*unescapedAnnotation* {*unescapedAnnotation*}) `']'`

***annotationUseSiteTarget*:**
(*AT_NO_WS* | *AT_PRE_WS*) (`'field'` | `'property'` | `'get'` | `'set'` |
`'receiver'` | `'param'` | `'setparam'` | `'delegate'`) {*NL*} `':'`

***unescapedAnnotation*:**
constructorInvocation
| *userType*

***simpleIdentifier*:**
Identifier
| `'abstract'`
| `'annotation'`
| `'by'`
| `'catch'`
| `'companion'`
| `'constructor'`
| `'crossinline'`
| `'data'`
| `'dynamic'`
| `'enum'`
| `'external'`
| `'final'`
| `'finally'`
| `'get'`
| `'import'`
| `'infix'`
| `'init'`
| `'inline'`
| `'inner'`
| `'internal'`
| `'lateinit'`
| `'noinline'`
| `'open'`
| `'operator'`
| `'out'`
| `'override'`
| `'private'`
| `'protected'`
| `'public'`
| `'reified'`
| `'sealed'`
| `'tailrec'`
| `'set'`

```
    | 'vararg'
    | 'where'
    | 'field'
    | 'property'
    | 'receiver'
    | 'param'
    | 'setparam'
    | 'delegate'
    | 'file'
    | 'expect'
    | 'actual'
    | 'const'
    | 'suspend'
    | 'value'
```

***identifier*:**
simpleIdentifier {{*NL*} `'.'` *simpleIdentifier*}

1.4 Documentation comments

Kotlin supports special comment syntax for code documentation purposes, called KDoc. The syntax is based on Markdown and Javadoc. Documentation comments start with `/**` and end with `*/` and allows external tools to generate documentation based on the comment contents.