Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 7

Statements

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

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

Kotlin does not explicitly distinguish between statements, expressions and declarations, i.e., expressions and declarations can be used in statement positions. This section focuses only on those statements that are *not* expressions or declarations. For information on those parts of Kotlin, please refer to the Expressions and Declarations sections of the specification.

> Example: Kotlin supports using conditionals both as expressions and as statements. As their use as expressions is more general, detailed information about conditionals is available in the Expressions section of the specification.

7.1 Assignments

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

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

An *assignment* is a statement that writes a new value to some program entity, denoted by its left-hand side. Both left-hand and right-hand sides of an assignment must be expressions, more so, there are several restrictions for the expression on the left-hand side.

For an expression to be *assignable*, i.e. be allowed to occur on the left-hand side of an assignment, it **must** be one of the following:

- An identifier referring to a mutable property;
- A navigation expression referring to a mutable property. If this navigation operator is the safe navigation operator, this introduces a special case of *safe assignment*;
- An indexing expression.

  Note: Kotlin assignments **are not** expressions and cannot be used as such.

7.1.1 Simple assignments

A *simple assignment* is an assignment which uses the assign operator `=`. If the left-hand side of an assignment refers to a mutable property, a value of that property is changed when an assignment is evaluated, using the following rules (applied in order).

- If a property has a setter (including delegated properties), it is called using the right-hand side expression as its argument;
- Otherwise, if a property is a mutable property, its value is changed to the evaluation result of the right-hand side expression.

If the left-hand side of an assignment is an indexing expression, the whole statement is treated as an overloaded operator with the following expansion:

$A[B_1, B_2, B_3, \ldots, B_N] = C$ is the same as calling A`.set`$(B_1, B_2, B_3, \ldots, B_N, C)$ where `set` is a suitable operator function.

7.1.2 Operator assignments

An *operator assignment* is a combined-form assignment which involves one of the following operators: `+=`, `-=`, `*=`, `/=`, `%=`. All of these operators are overloadable operator functions with the following expansions (applied in order):

- A `+=` B is exactly the same as one of the following:
  - A`.plusAssign(`B`)` if a suitable `plusAssign` operator function exists and is available;
  - A `=` A`.plus(`B`)` if a suitable `plus` operator function exists and is available.
- A `-=` B is exactly the same as one of the following:
  - A`.minusAssign(`B`)` if a suitable `minusAssign` operator function exists and is available;

  - A `=` A`.minus(`B`)` if a suitable `minus` operator function exists and is available.
- A `*=` B is exactly the same as one of the following:
  - A`.timesAssign(`B`)` if a suitable `timesAssign` operator function exists and is available;
  - A `=` A`.times(`B`)` if a suitable `times` operator function exists and is available.
- A `/=` B is exactly the same as one of the following:
  - A`.divAssign(`B`)` if a suitable `divAssign` operator function exists and is available;
  - A `=` A`.div(`B`)` if a suitable `div` operator function exists and is available;
- A `%=` B is exactly the same as one of the following:
  - A`.remAssign(`B`)` if a suitable `remAssign` operator function exists and is available;
  - A `=` A`.rem(`B`)` if a suitable `rem` operator function exists and is available.

  Note: before Kotlin version 1.3, there were additional overloadable functions for `%` called `mod`/`modAssign`

After the expansion, the resulting function call expression or simple assignment is processed according to their corresponding rules, and overload resolution and type checking are performed. If both expansion variants result in correctly resolved and inferred code, this should be reported as an operator overloading ambiguity. If only one of the expansion variants can be resolved correctly, this variant is picked as the correct one. If neither of variants result in correct code, the operator calls must be reported as unresolved.

> Example: consider the following compound operator statement: `x[y] += z`. The corresponding expansion variants are `x.get(y).plusAssign(z)` and `x.set(x.get(y).plus(z))` according to expansion rules for corresponding operators. If, for example, the call to `set` in the second variant results in resolution or inference error, the whole corresponding expansion is deemed unresolved and the first variant is picked if applicable.
>
> Note: although for most real-world use cases operators `++` and `--` are similar to operator assignments, in Kotlin they are expressions and are described in the corresponding section of this specification.

7.1.3 Safe assignments

If the left-hand side of an assignment involves a safe-navigation operator, it is treated as a special case of *safe assignment*. Safe assignments are expanded similar to safe navigation operator expressions:

- `a?.c` is exactly the same as

```
when(val $tmp = a) {
    null -> null
    else -> { $tmp.c }
}
```

For any right-hand combinations of operators present in `c`, which are expanded further, as usual.

Example: The assignment

```
x?.y[0] = z
```

is expanded to

```
when(val $tmp = x) {
    null -> null
    else -> { $tmp.y[0] = z }
}
```

which, according to expansion rules for indexing assignments is, in turn, expanded to

```
when(val $tmp = x) {
    null -> null
    else -> { $tmp.y.set(0, z) }
}
```

7.2 Loop statements

Loop statements describe an evaluation of a certain number of statements repeatedly until a *loop exit condition* applies.

***loopStatement*:**

forStatement
| *whileStatement*
| *doWhileStatement*

Loops are closely related to the semantics of jump expressions, as these expressions, namely `break` and `continue`, are only allowed in a body of a loop. Please refer to the corresponding sections for details.

7.2.1 While-loop statements

***whileStatement*:**

`'while'`
{*NL*}
`'('`
expression
`')'`

{*NL*}
(*controlStructureBody* | ';')

A *while-loop statement* is similar to an `if` expression in that it also has a condition expression and a body consisting of zero or more statements. While-loop statement evaluating its body repeatedly for as long as its condition expression evaluates to true or a jump expression is evaluated to finish the loop.

> Note: this also means that the condition expression is evaluated before every evaluation of the body, including the first one.

The while-loop condition expression **must be a subtype** of `kotlin.Boolean`.

7.2.2 Do-while-loop statements

***doWhileStatement*:**

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

A *do-while-loop statement*, similarly to a while-loop statement, also describes a loop, with the following differences. First, it has a different syntax. Second, it evaluates the loop condition expression **after** evaluating the loop body.

> Note: this also means that the body is always evaluated at least once.

The do-while-loop condition expression **must be a subtype** of `kotlin.Boolean`.

7.2.3 For-loop statements

***forStatement*:**

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

> Note: unlike most other languages, Kotlin does not have a free-form condition-based for loops. The only form of a for-loop available in Kotlin is the "foreach" loop, which iterates over lists, arrays and other data structures.

A *for-loop statement* is a special kind of loop statement used to iterate over some data structure viewed as an iterable collection of elements. A for-loop statement consists of a loop body, a **container expression** and an **iteration variable declaration**.

The for-loop is actually an overloadable syntax form with the following expansion:

`for(VarDecl in C) Body` is the same as

```
when(val $iterator = C.iterator()) {
    else -> while ($iterator.hasNext()) {
                val VarDecl = __iterator.next()
                <... all the statements from Body>
            }
}
```

where `iterator`, `hasNext`, `next` are all suitable operator functions available in the current scope. `VarDecl` here may be a variable name or a set of variable names as per destructuring variable declarations.

> Note: the expansion is hygienic, i.e., the generated iterator variable never clashes with any other variable in the program and cannot be accessed outside the expansion.

7.3 Code blocks

***block*:**

'{'
{*NL*}
statements
{*NL*}
'}'

***statements*:**

[*statement* {*semis statement*}] [*semis*]

A *code block* is a sequence of zero or more statements between curly braces separated by newlines or/and semicolons. Evaluating a code block means evaluating all its statements in the order they appear inside of it.

> Note: Kotlin does **not** support code blocks as statements; a curly-braces code block in a statement position is a lambda literal.

A *last expression* of a code block is the last statement in it (if any) if and only if this statement is also an expression. A code block is said to contain no last

expression if it does not contain any statements or its last statement is not an expression (e.g., it is an assignment, a loop or a declaration).

> Informally: you may consider the case of a missing last expression as if a synthetic last expression with no runtime semantics and type `kotlin.Unit` is introduced in its place.

A *control structure body* is either a single statement or a code block. A *last expression* of a control structure body CSB is either the last expression of a code block (if CSB is a code block) or the single expression itself (if CSB is an expression). If a control structure body is not a code block or an expression, it has no last expression.

> Note: this is equivalent to wrapping the single expression in a new synthetic code block.

In some contexts, a control structure body is expected to have a value and/or a type. The value of a control structure body is:

- the value of its last expression if it exists;
- the singleton `kotlin.Unit` object otherwise.

The type of a control structure body is the type of its value.

7.3.1 Coercion to `kotlin.Unit`

When we expect the type of a control structure body to be `kotlin.Unit`, we relax the type checking requirements for its type by *coercing* it to `kotlin.Unit`. Specifically, we *ignore* the type mismatch between `kotlin.Unit` and the control structure body type.

> Examples:

```
fun foo() {
    val a /* : () -> Unit */ = {
        if (true) 42
        // CSB with no last expression
        // Type is defined to be `kotlin.Unit`
    }

    val b: () -> Unit = {
        if (true) 42 else -42
        // CSB with last expression of type `kotlin.Int`
        // Type is expected to be `kotlin.Unit`
        // Coercion to kotlin.Unit applied
    }
}
```