Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 9

Operator overloading

Some syntax forms in Kotlin are *defined by convention*, meaning that their semantics are defined through syntactic expansion of one syntax form into another syntax form.

Particular cases of definition by convention include:

- Arithmetic and comparison operators;
- `invoke` convention;
- Operator-form assignments;
- For-loop statements;
- Delegated properties;
- Destructuring declarations.

  Important: another case of definition by convention is safe navigation, which is covered in more detail in its respective section.

There are several points shared among all the syntax forms defined using definition by convention:

- The expansions are hygienic: if they introduce new identifiers that were not present in original syntax, all such identifiers are not accessible outside the expansion and cannot clash with any other declarations in the program;
- The expressions captured by an expansion are using *call-by-need* evaluation strategy, meaning that they are evaluated only once during first usage specified in the expansion even if the expansion itself has more than one usage of such an expression;
- An expansion may lead to another expansion, following the same rules;
- All call expressions that are produced by expansion are only allowed to use operator functions.

This expansion of a particular syntax form to a different piece of code is usually defined in the terms of *operator* functions.

Operator functions are function which are declared with a special keyword `operator` and are not different from regular functions when called via function calls. However, operator functions can also be used in definition by convention.

> Note: it is not important whether an operator function is a member or an extension, nor whether it is suspending or not. The only requirements are the ones listed in the respected sections.
>
> For example, for an operator form `a + b` where `a` is of type `A` and `b` is of type `B` any of the following function definitions are applicable:

```
class A {
    // member function
    operator fun plus(b: B) = ...
    // suspending member function
    suspend operator fun plus(b: B) = ...
}

// extension function
operator fun A.plus(b: B) = ...
// suspending extension function
suspend operator fun A.plus(b: B) = ...
```

> Assuming additional implicit receiver of this type is available, it may also be an extension defined in another type:

```
object Ctx {
    // extension that is a member of some context type
    operator fun A.plus(b: B) = ...

    fun add(a: A, b: B) = a + b
}
```

> Note: different platforms may add additional criteria on whether a function may be considered a suitable candidate for operator convention.

The details of individual expansions are available in the sections of their respective operators, here we would like to describe how they *interoperate*.

For example, take the following declarations:

```
class A {
    operator fun inc(): A { ... }
}

object B {
    operator fun get(i: Int): A { ... }
    operator fun set(i: Int, value: A) { ... }
}
```

```
object C {
    operator fun get(i: Int): B { ... }
}
```

The expression `C[0][0]++` is expanded (see the Expressions section for details) using the following rules:

- The operations are expanded in order of their priority.
- First, the increment operator is expanded, resulting in:

  ```
  C[0][0] = C[0][0].inc()
  ```

- Second, the assignment to an indexing expression (produced by the previous expansion) is expanded, resulting in:

  ```
  C[0].set(C[0][0].inc())
  ```

- Third, the indexing expressions are expanded, resulting in:

  ```
  C.get(0).set(C.get(0).get(0).inc())
  ```

  Important: although the resulting expression contains several instances of the subexpression `C.get(0)`, as all these instances were created from the same original syntax form, the subexpression is evaluated only once, making this code roughly equivalent to:

  ```
  val $tmp = C.get(0)
  $tmp.set($tmp.get(0).inc())
  ```

9.1 Destructuring declarations

A special case of definition by convention is the destructuring declaration of properties, which is available for local properties, parameters of lambda literals and the iteration variable of for-loops. See the corresponding sections for particular syntax.

This convention allows to introduce a number (one or more) of properties in the place of one by immediately *destructuring* the property during construction. The immediate value (that is, the initializing expression of the local property, the value acquired from the operator convention of a for-loop statement, or an argument passed into a lambda body) is assigned to a number of placeholders $p_0, \ldots, p_N$ where each placeholder is either an identifier or a special ignoring placeholder `_` (note that `_` is not a valid identifier in Kotlin). For each identifier the corresponding operator function `componentK` with K being equal to the position of the placeholder in the declaration (**starting from 1**) is called without arguments and the result is assigned to a fresh value referred to as the identifier used. For each ignoring placeholder, no calls are performed and nothing is assigned. Each placeholder may be provided with an optional type signature

T_M which is used in type inference as any property type would. Note that an ignoring placeholder may also be provided with a type signature, in which case although the call to corresponding `componentM` function is not performed, it still must be checked for function applicability during type inference.

Examples:

```
val (x: A, _, z) = f()
```

is expanded to

```
val $tmp = f()
val x: A = $tmp.component1()
val z = $tmp.component3()
```

where `component1` and `component3` are suitable operator functions available on the value returned by `f()`

```
for((x: A, _, z) in f()) { ... }
```

is expanded to (as per for-loop expansion)

```
when(val $iterator = f().iterator()) {
    else -> while ($iterator.hasNext()) {
                val $tmp = $iterator.next()
                val x: A = $tmp.component1()
                val z = $tmp.component3()
                ...
            }
}
```

where `iterator()`, `next()`, `hasNext()`, `component1()` and `component3` are all suitable operator functions available on their respective receivers.

```
foo { (x: A, _, z) -> ... }
```

is expanded to

```
foo { $tmp ->
    val x: A = $tmp.component1()
    val z = $tmp.component3()
    ...
}
```

where `component1()` and `component3` are all suitable operator functions available on the value of lambda argument.