Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 6

Scopes and identifiers

Kotlin program is logically divided into *scopes*.

A scope is a syntactically-delimited region of code which constitutes a context in which entities and their names can be introduced. Scopes can be nested, with entities introduced in outer scopes possibly available in the inner scopes if linked.

The top level of a Kotlin file is also a scope, containing all the scopes within a file.

All the scopes are divided into two kinds: declaration scopes and statement scopes. These two kinds of scopes differ in how the identifiers may refer to the entities defined in the scopes.

Declaration scopes include:

- The project modules;
- The project packages;
- The top level scopes of non-script Kotlin files;
- The bodies of classifier declarations;
- The bodies of object literals;
- Function parameter scope containing the declared value parameters in a function declaration or a non-primary constructor declaration;
- Primary constructor parameter scope containing the declared value parameters in a primary constructor.

Statement scopes include:

- The top level scopes of script Kotlin files;
- Scopes produced by control structure bodies of different expressions;
- The bodies of function declarations;
- The bodies of function literals;
- The bodies of getters and setters of properties;
- The bodies of constructors;

- The bodies of instance initialization blocks in classifier declarations;
- Special initialization scope for a body of a classifier declaration.

All declarations in a particular scope introduce new *bindings* of identifiers in this scope to their respective entities in the program. These entities may be types or values, where values refer to objects, functions or properties (which may also be delegated). Top-level scopes additionally allow to introduce new bindings using `import` directives from other top-level scopes.

In most situations, it is not allowed to bind several values to the same identifier in the same scope, but it is allowed to bind a value to an identifier already available in the scope through linked scopes or imports.

An exception to this rule are function declarations, which are matched by signatures and allow defining several functions with the same name in the same scope. When calling functions, a process called overload resolution allows for differentiating between such functions. Overload resolution also applies to properties if they are used as functions through `invoke`-convention, but it does not allow defining several properties with the same name and with the same receivers in the same scope.

> Note: platforms may introduce additional restrictions on which identifiers may be declared together in the same or linked scopes.

The main difference between declaration scopes and statement scopes is that names in the statement scope are bound in the order of appearance. It is not allowed to access a value through an identifier in code which (syntactically) precedes the binding itself. On the contrary, in declaration scopes it is fully allowed, although initialization cycles may occur leading to unspecified behaviour.

> Note: Kotlin compiler may attempt to detect and report such initialization cycles as compile-time warnings or errors.

It also means that statement scopes nested inside declaration scopes may access values declared afterwards in parent declaration scopes, but any values declared inside a statement scope can be accessed only after their declaration point.

Examples:

- In declaration scope:

```
// x refers to the property defined below
//   even if there is another property
//   called x in outer scope or imported
fun foo() = x + 2
val x = 3
```

- In statement scope:

```
// x refers to another property
//   defined in outer scope or imported
//   or is a compile-time error
```

```
fun foo() = x + 2
val x = 3
```

6.1 Linked scopes

Scopes `A` and `B` in a Kotlin program may be *downwards-linked* (`A ~> B`), meaning identifiers from `A` can be used in `B` without the need for additional qualification. If scopes `A` and `B` are downwards-linked, scopes `B` and `A` are considered *upwards-linked* (`B <~ A`).

> Note: link relation is transitive, unless specified otherwise.

Scopes are downwards-linked (DLD) or upwards-linked (ULD) as follows:

- A statement scope is DLD to any directly nested scope;
- An object declaration scope is DLD to any nested scopes;
- An object declaration scope is non-transitively ULD to the companion object scopes of its superclasses;
- An object declaration scope is non-transitively ULD to the companion object scopes of its parent classifier superclasses;
- An object declaration scope is ULD to the companion object declaration scope of its parent classifier;
- A companion object declaration scope is DLD to any nested scopes;
- A companion object declaration scope is non-transitively ULD to the companion object scopes of its superclasses;
- A companion object declaration scope is non-transitively ULD to the companion object scopes of its parent classifier superclasses;
- A companion object declaration scope is ULD to the companion object declaration scope of the *parent* of its parent classifier;
- A classifier or nested class declaration scope is DLD to any nested statement scopes;
- A classifier or nested class declaration scope is ULD to its companion object declaration scope;
- An inner class declaration scope is DLD to any nested statement scopes;
- An inner class declaration scope is ULD to the classifier declaration scope of its parent classifier;
- A function or non-primary constructor parameter scope is ULD to the scope containing the function declaration and DLD to the function body;
- A primary constructor parameter scope is ULD to the scope containing the classifier declaration (but not the classifier declaration scope itself) and DLD to the classifier initialization scope;
- The instance initialization blocks are ULD to the classifier initialization scope.

> Important: linked scopes **do not** cover cases when identifiers from supertypes are used in subtypes, as this is covered by the inheritance rules.

6.2 Identifiers and paths

Kotlin program operates with different *entities*, such as classes, interfaces, values, etc. An entity can be referenced using its *path*: a sequence of identifiers which references this entity in a given scope.

Kotlin supports two kinds of paths.

- Simple paths `P`, which consist of a single identifier
- Qualified paths `P.m`, which consist of a path `P` and a member identifier `m`

Besides identifiers which are introduced by the developer (e.g., via declaring classes or introducing variables), there are several predefined identifiers with special semantics.

- `this` – an identifier which references the default receiver available in the current scope, further details are available here
- `this@label` – an identifier which references the default receiver available in the selected scope, further details are available here
- `super<Klazz>` – an identifier which references the supertype `Klazz` available in the current scope, further details are available here
- `super<Klazz>@label` – an identifier which references the supertype `Klazz` available in the selected scope, further details are available here

6.3 Labels

Labels are special syntactic marks which allow one to reference certain code fragments or elements. Lambda expressions and loop statements are allowed to be labeled, with label identifier associated with the corresponding entity.

> Note: in Kotlin version 1.3 and earlier, labels were allowed to be placed on any expression or statement.

Labels can be *redeclared*, meaning that the same label identifier may be reused with different parts of code (or even on the same expression/loop) several times. Labels are *scoped*, meaning that they are only available in the scope they were declared in.

Labels are used by certain expressions, such as `break`, `continue` and `return`, to specify exactly what entity the expression corresponds to. Please refer to the corresponding sections for details.

When resolving labels (determining which label an expression refers to), the *closest* label with the matching identifier is chosen, i.e., a label in an innermost scope syntactically closest to the point of its use.