Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 10

Packages and imports

A Kotlin project is structured into **packages**. A package contains one or more Kotlin files, with files linked to a package using a *package header*. A file may contain exactly one or zero package headers, meaning each file belongs to exactly one package.

> Note: an absence of a package header in a file means it belongs to the special *root package*.

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

> Note: Packages are orthogonal from modules. A module may contain many packages, and a single package can be spread across several modules.

The name of a package is a simple or a qualified path, which creates a package hierarchy.

> Note: unlike many other languages, Kotlin packages *do not* require files to have any specific locations w.r.t. itself; the connection between a file and its package is established only via a package header. It is strongly recommended, however, that the folder structure of a project does correspond to the package hierarchy.

10.1 Importing

Program entities declared in one package may be freely used in any file in the same package with the only two restrictions being module boundaries and visibility constraints. In order to use an entity from a file belonging to a different package, the programmer must use *import directives*.

***importList*:**
  {*importHeader*}

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

***importAlias*:**
  `'as'` *simpleIdentifier*

An import directive contains a simple or a qualified path, with the name of an imported entity as its last component. A path may include not only a package, but also an object or a type, in which case it refers to the companion object of that type. The last component may reference any named declaration within that scope (that is, top-level scope of all files in the package or an object declaration scope) may be imported using their names.

There are two special kinds of imports: star-imports ending in an asterisk (`*`) and renaming imports employing the `as` operator.

Star-imports import all named entities inside the corresponding scope, but have lesser priority during overload resolution of functions and properties.

Renaming imports work just like regular imports, but introduce the entity into the current file with the specified name, such that an unqualified access to this entity is possible *only using the newly specified name*. This means that renaming imports of entities from the same package effectively *change* their unqualified name.

Example:

```
package foo

import foo.foo as baz

fun foo() {} // (1)
fun bar() {} // (2)

fun test() {
    // Qualified access is unchanged by the renaming import
    foo.foo() // resolves to (1)
    foo.bar() // resolved to (2)

    // Unqualified access considers the rename of `foo` to `baz`
    foo() // Unresolved reference
    bar() // resolves to (2)
    baz() // resolves to (1)
}
```

Imports from objects have certain limitations: only object members may be imported and star-imports are not allowed.

Imports are local to their files, meaning if an entity is introduced into file A.kt from package `foo.bar`, it does not introduce that entity to any other file from package `foo.bar`.

There are some packages which have all their entities *implicitly imported* into any Kotlin file, meaning one can access such entity without explicitly using import directives.

> Note: one may, however, import these entities explicitly if they choose to do so.

The following packages of the standard library are implicitly imported:

- `kotlin`
- `kotlin.annotation`
- `kotlin.collections`
- `kotlin.comparisons`
- `kotlin.io`
- `kotlin.ranges`
- `kotlin.sequences`
- `kotlin.text`
- `kotlin.math`

> Note: platform implementations may introduce additional implicitly imported packages, for example, to extend Kotlin code with the platform-specific functionality. An example of this would be `java.lang` package implicitly imported on the JVM platform.

Importing certain entities may be disallowed by their visibility modifiers.

- `public` entities can be imported anywhere
- `internal` entities can be imported only within the same module
- `protected` entities cannot be imported
- top-level `private` entities can be imported within their declaring file
- other `private` entities cannot be imported

10.2 Modules

A module is a concept on the boundary between the code itself and the resulting application, thus it depends on and influences both of them. A Kotlin module is a set of Kotlin files which are considered to be interdependent and must be handled together during compilation.

In a simple case, a module is a set of files compiled at the same time in a given project.

- A set of files being compiled with a single Kotlin compiler invocation
- A Maven module
- A Gradle project

In a more complicated case involving multi-platform projects, a module may be distributed across several compilations, projects and/or platforms.

For the purposes of Kotlin/Core, modules are important for `internal` visibility. How modules influence particular platforms is described in their respective sections of this specification.