Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 17

Annotations

Annotations are a form of syntactically-defined metadata which may be associated with different entities in a Kotlin program. Annotations are specified in the source code of the program and may be accessed on a particular platform using platform-specific mechanisms both by the compiler (and source-processing tools) and at runtime (using reflection facilities). Values of annotation types can also be created directly, but are usually operated on using platform-specific facilities.

> Note: before Kotlin 1.6, annotation types could not be created directly.

17.1 Annotation values

An annotation value is a value of a special annotation type. An annotation type is a special kind of class type which is allowed to include read-only properties of the following types:

- Integer types;
- Enum types;
- String type;
- Other annotation types;
- Arrays of any type listed above.

> Important: when we say "other annotation types", we mean an annotation type cannot reference itself, either directly or indirectly. For example, if annotation type `A` references annotation type `B` which references an array of `A`, it is prohibited and reported as a compile-time error.

Annotation classes are not allowed to have any member functions, constructors or mutable properties. They are also not allowed to have declared supertypes and are considered to be implicitly derived from `kotlin.Annotation`.

17.2 Annotation retention

The retention level of an annotation declares which compilation artifacts (for a particular compiler on a particular platform) *retain* this kind of annotation. There are the following types of retention available:

- Source retention (accessible by source-processing tools);
- Binary retention (retained in compilation artifacts);
- Runtime retention (accessible at runtime).

  Each subsequent level inherits what is accessible on the previous levels.

For availability and particular ways of accessing the metadata specified by these annotations please refer to the corresponding platform-specific documentation.

17.3 Annotation targets

The *target* of a particular type of annotations is the kind of program entity which this annotations may be placed on. There are the following targets available:

- A class declaration (including annotation classes);
- An annotation class declaration;
- A type parameter;
- A property declaration;
- A property backing field;
- A property getter;
- A property setter;
- A local property declaration;
- A value parameter in function or constructor declaration;
- A constructor;
- A function declaration;
- A type usage;
- An arbitrary expression;
- A Kotlin file;
- A type alias declaration.

17.4 Annotation declarations

Annotations are declared using annotation class declarations. See the corresponding section for details.

Annotations may be declared *repeatable* (meaning that the same annotation may be applied to the same entity more than once) or *non-repeatable* (meaning that only one annotation of a particular type may be applied to the same entity).

17.5 Built-in annotations

17.5.1 `kotlin.annotation.Retention`

`kotlin.annotation.Retention` is an annotation which is only used on annotation classes to specify their annotation retention level. It has the following single field:

- `val value: AnnotationRetention = AnnotationRetention.RUNTIME`

  The retention level of the annotated annotation.

`kotlin.annotation.AnnotationRetention` is an enum class with the following values (see Annotation retention section for details):

- `SOURCE`;
- `BINARY`;
- `RUNTIME`.

17.5.2 `kotlin.annotation.Target`

`kotlin.annotation.Target` is an annotation which is only used on annotation classes to specify targets those annotations are valid for. It has the following single field:

- `vararg val allowedTargets: AnnotationTarget`

  The allowed annotation targets of the annotated annotation.

`kotlin.annotation.AnnotationTarget` is an enum class with the following values (see Annotation targets section for details):

- `CLASS`;
- `ANNOTATION_CLASS`;
- `TYPE_PARAMETER`;
- `PROPERTY`;
- `FIELD`;
- `LOCAL_VARIABLE`;
- `VALUE_PARAMETER`;
- `CONSTRUCTOR`;
- `FUNCTION`;
- `PROPERTY_GETTER`;
- `PROPERTY_SETTER`;
- `TYPE`;
- `EXPRESSION`;
- `FILE`;
- `TYPEALIAS`.

17.5.3 kotlin.annotation.Repeatable

`kotlin.annotation.Repeatable` is an annotation which is only used on annotation classes to specify whether this particular annotation is repeatable. Annotations are non-repeatable by default.

17.5.4 kotlin.RequiresOptIn / kotlin.OptIn

`kotlin.RequiresOptIn` is an annotation class with two fields:

- ```
  val message: String = ""
  ```

  The message describing the particular opt-in requirements.

- ```
  val level: Level = Level.ERROR
  ```

  The severity level of the experimental status with two possible values: `Level.WARNING` and `Level.ERROR`.

This annotation is used to introduce implementation-defined experimental language or standard library features.

`kotlin.OptIn` is an annotation class with a single field:

- ```
  vararg val markerClass: KClass<out Annotation>
  ```

  The classes which this annotation allows to use.

This annotation is used to explicitly mark declarations which use experimental features marked by `kotlin.RequiresOptIn`.

It is implementation-defined how this annotation is processed.

> Note: before Kotlin 1.4.0, there were two other built-in annotations: `@Experimental` (now replaced by `@RequiresOptIn`) and `@UseExperimental` (now replaced by `@OptIn`) serving the same purpose which are now deprecated.

17.5.5 kotlin.Deprecated / kotlin.ReplaceWith

`kotlin.Deprecated` is an annotation class with the following fields:

- ```
  val message: String
  ```

  A message supporting the deprecation.

- ```
  val replaceWith: ReplaceWith = ReplaceWith("")
  ```

  An optional replacement for the deprecated code.

- ```
  val level: DeprecationLevel = DeprecationLevel.WARNING
  ```

  The deprecation level with three possible values: `DeprecationLevel.WARNING`, `DeprecationLevel.ERROR` and `DeprecationLevel.HIDDEN`.

`kotlin.ReplaceWith` is itself an annotation class containing the information on how to perform the replacement in case it is provided. It has the following fields:

- `val expression: String`

  The replacement code.

- `vararg val imports: String`

  An array of imports needed for the replacement code to work correctly.

`kotlin.Deprecated` is a built-in annotation supporting the deprecation cycle for declarations: marking some declarations as outdated, soon to be replaced with other declarations, or not recommended for use. It is implementation-defined how this annotation is processed, with the following recommendations:

- Attempting to use a declaration with deprecation level of `DeprecationLevel.WARNING` should produce a compile-time warning;
- Attempting to use a declaration with deprecation level of `DeprecationLevel.ERROR` should produce a compile-time error.

17.5.6 kotlin.Suppress

`kotlin.Suppress` is an annotation class with the following single field:

- `vararg val names: String`

  The names of features this annotation is suppressing.

`kotlin.Suppress` is used to optionally mark any piece of code as suppressing some language feature, such as a compiler warning, an IDE mechanism or a language feature. The names of features which one can suppress with this annotation are implementation-defined, as is the processing of this annotation itself.

17.5.7 kotlin.SinceKotlin

`kotlin.SinceKotlin` is an annotation class with the following single field:

- `val version: String`

  The version of Kotlin language.

`kotlin.SinceKotlin` is used to mark a declaration which is only available since a particular version of the language. These mostly refer to standard library declarations. It is implementation-defined how this annotation is processed.

17.5.8 kotlin.UnsafeVariance

`kotlin.UnsafeVariance` is an annotation class with no fields which is only applicable to types. Any type instance marked by this annotation explicitly

states that the variance errors arising for this particular type instance are to be ignored by the compiler.

17.5.9 `kotlin.DslMarker`

`kotlin.DslMarker` is an annotation class with no fields which is applicable only to other annotation classes. An annotation class annotated with `kotlin.DslMarker` is marked as a marker of a specific DSL (domain-specific language). Any type annotated with such a marker is said to belong to that specific DSL. This affects overload resolution in the following way: no two implicit receivers with types belonging to the same DSL are available in the same scope. See Overload resolution section for details.

17.5.10 `kotlin.PublishedApi`

`kotlin.PublishedApi` is an annotation class with no fields which is applicable to any declaration. It may be applied to any declaration with `internal` visibility to make it available to `public inline` declarations. See Declaration visibility section for details.

17.5.11 `kotlin.BuilderInference`

Marks the annotated function of function argument as eligible for builder-style type inference. See corresponding section for details.

> Note: as of Kotlin 1.9, this annotation is experimental and, in order to use it in one's code, one must explicitly enable it using opt-in annotations given above. The particular marker class used to perform this is implementation-defined.

17.5.12 `kotlin.RestrictSuspension`

In some cases we may want to limit which suspending functions can be called in another suspending function with an extension receiver of a specific type; i.e., if we want to provide a coroutine-enabled DSL, but disallow the use of arbitrary suspending functions. To do so, the type `T` of that extension receiver needs to be annotated with `kotlin.RestrictSuspension`, which enables the following limitations.

- Suspending functions with an extension receiver of type `T` are restricted from calling other suspending functions besides those accessible on this receiver.
- Suspending functions of type `T` can be called only on an extension receiver.

17.5.13 `kotlin.OverloadResolutionByLambdaReturnType`

This annotation is used to allow using lambda return type to refine function

applicability during overload resolution. Further details are available in the corresponding section.

> Note: as of Kotlin 1.9, this annotation is experimental and, in order to use it in one's code, one must explicitly enable it using opt-in annotations given above. The particular marker class used to perform this is implementation-defined.