Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 5

Inheritance

Kotlin is an object-oriented language with its object model based on **inheritance**.

5.1 Classifier type inheritance

Classifier types may be inherited from each other: the type inherited *from* is called the *base type*, while the type which inherits the base type is called the *derived type*. The following limitations are imposed on the possible inheritance structure.

A class or object type is allowed to inherit from only one class type (called its **direct superclass**) and multiple interface types. As specified in the declaration section, if the superclass of a class or object type is not specified, it is assumed to be `kotlin.Any`. This means, among other things, that every class or object type always has a direct superclass.

A class is called **closed** and cannot be inherited from if it is not explicitly declared as either `open` or `abstract`.

> Note: classes are neither `open` nor `abstract` by default.

A `data class`, `enum class` or `annotation class` cannot be declared `open` or `abstract`, i.e., are always closed and cannot be inherited from. Declaring a class `sealed` also implicitly declares it `abstract`.

An interface type may be inherited from any number of other interface types (and only interface types), if the resulting type is well-formed.

Object types cannot be inherited from.

Inheritance is the primary mechanism of introducing subtyping relations between user-defined types in Kotlin. When a classifier type A is declared with base types

$B_1, \ldots, B_m$, it introduces subtyping relations $A <: B_1, \ldots, A <: B_m$, which are then used in overload resolution and type inference mechanisms.

5.1.1 Abstract classes

A class declared `abstract` cannot be instantiated, i.e., an object of this class cannot be created directly. Abstract classes are implicitly `open` and their primary purpose is to be inherited from. Abstract classes (similarly to interfaces) allow for `abstract` property and function declarations in their scope.

5.1.2 Sealed classes and interfaces

A class or interface (but not a functional interface) may be declared `sealed`, making it special from the inheritance point-of-view.

- A `sealed` class is implicitly `abstract` (and these two modifiers are exclusive);
- A `sealed` class or interface can only be inherited from by types declared in the same package and in the same module, and which have a fully-qualified name (meaning local and anonymous types cannon be inherited from `sealed` types);
- `Sealed` classes and interfaces allow for exhaustiveness checking of when expressions for values of such types. Any sealed type `S` is associated with its *direct non-sealed subtypes*: a set of non-sealed types, which are either direct subtypes of `S` or transitive subtypes of `S` via some number of other *sealed* types. These direct non-sealed subtypes form the boundary for exhaustiveness checks.

5.1.3 Inheritance from built-in types

Built-in types follow the same rules as user-defined types do. Most of them are closed class types and cannot be inherited from. Function types are treated as interfaces and can be inherited from as such.

5.2 Matching and subsumption of declarations

A callable declaration D *matches* to a callable declaration B if the following are true.

- B and D have the same name;
- B and D are declarations of the same kind (property declarations or function declarations);
- Function signature of D (if any) matches function signature of B (if any).

A callable declaration D *subsumes* a callable declaration B if the following are true.

- B and D match;
- The classifier of B (where it is declared) is a supertype of the classifier of D.

The notions of matching and subsumption are used when talking about how declarations are inherited and overridden.

5.3 Inheriting

A callable declaration (that is, a property or member function declaration) inside a classifier declaration is said to be *inheritable* if:

- Its visibility (and the visibility of its getter and setter, if present) is not `private`.

If the declaration B of the base classifier type is inheritable, no other inheritable declaration from the base classifier types subsume B, no declarations in the derived classifier type override B, then B is *inherited* by the derived classifier type.

As Kotlin is a language with single inheritance (only one supertype can be a class, any number of supertypes can be an interface), there are several additional rules which refine how declarations are inherited.

- If a derived class type inherits a declaration from its superclass, no other matching *abstract* declarations from its superinterfaces are inherited.
- If a derived classifier type inherits *several* matching *concrete* declarations from its supertypes, it is a compile-time error (this means a derived classifier type should override such declarations).
- If a derived *concrete* classifier type inherits an *abstract* declaration from its supertypes, it is a compile-time error (this means a derived classifier type should override such declaration).
- If a derived classifier type inherits both an *abstract* and a *concrete* declaration from its superinterfaces, it is a compile-time error (this means a derived classifier type should override such declarations).

5.4 Overriding

A callable declaration (that is, a property or member function declaration) inside a classifier declaration is said to be *overridable* if:

- Its visibility (and the visibility of its getter and setter, if present) is not `private`;
- It is declared as `open`, `abstract` or `override` (interface methods and properties are implicitly `abstract` if they don't have a body or `open` if they do).

It is illegal for a declaration to be both `private` and either `open`, `abstract` or `override`, such declarations should result in a compile-time error.

If the declaration B of the base classifier type is overridable, the declaration D of the derived classifier type subsumes B, and D has an `override` modifier, then D is *overriding* the base declaration B.

A function declaration D which overrides function declaration B should satisfy the following conditions.

- Return type of D is a subtype of return type of B;
- Suspendability of D and B must be the same.

A property declaration D which overrides property declaration B should satisfy the following conditions.

- Mutability of D is not stronger than mutability of B (where read-only `val` is stronger than mutable `var`);
- Type of D is a subtype of type of B; except for the case when both D and B are mutable (`var`), then types of D and B must be equivalent.

Otherwise, it is a compile-time error.

If the base declaration is not overridable and/or the overriding declaration does not have an `override` modifier, it is not permitted and should result in a compile-time error.

If the overriding declaration *does not* have its visibility specified, its visibility is implicitly set to be the same as the visibility of the overridden declaration.

If the overriding declaration *does* have its visibility specified, it must not be stronger than the visibility of the overridden declaration.

Examples:

```
open class B {
    protected open fun f() {}
}
class C : B() {
    open override fun f() {}
    // `f` is protected, as its visibility is
    //   inherited from the base declaration
}
class D : B() {
    public open override fun f() {}
    // this is correct, as public visibility is
    //   weaker that protected visibility
    //   from the base declaration
}

open class P {
```

```
    open fun g() {}
}

class Q : P() {
    protected open override fun g() {}
    // this is an error, as protected visibility is
    //   stronger that public visibility
    //   from the base declaration
}
```

Important: platforms may introduce additional cases of both *overridability* and *subsumption* of declarations, as well as limit the overriding mechanism due to implementation limitations.

Note: Kotlin does not have a concept of full hiding (or shadowing) of declarations.

Note: if a declaration binds a new function to the same name as was introduced in the base class, but which does not subsume it, it is neither a compile-time error nor an overriding declaration. In this case these two declarations follow the normal rules of overloading. However, these declarations may still result in a compile-time error as a result of conflicting overload detection.