Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 4

Declarations

Glossary

Entity
: Distinguishable part of a program

Identifier
: Name of a program entity

Path
: Sequence of identifiers which references a program entity in a given scope

Introduction

Declarations in Kotlin are used to introduce entities (values, types, etc.); most declarations are *named*, i.e. they also assign an identifier to their own entity, however, some declarations may be *anonymous*.

Every declaration is accessible in a particular *scope*, which is dependent both on where the declaration is located and on the declaration itself. Every named declaration introduces a binding for this name in the scope it is declared in. For most of the declarations, this scope is the declaration scope introduced by the **parent** declaration, e.g. the declaration this declaration is syntactically nested in. See scoping section for details.

4.1 Classifier declaration

***classDeclaration*:**

[*modifiers*]
('`class`' | (['`fun`' {*NL*}] '`interface`'))
{*NL*}

simpleIdentifier
[{*NL*} *typeParameters*]
[{*NL*} *primaryConstructor*]
[{*NL*} `':'` {*NL*} *delegationSpecifiers*]
[{*NL*} *typeConstraints*]
[({*NL*} *classBody*) | ({*NL*} *enumClassBody*)]

***objectDeclaration*:**
[*modifiers*]
`'object'`
{*NL*}
simpleIdentifier
[{*NL*} `':'` {*NL*} *delegationSpecifiers*]
[{*NL*} *classBody*]

Classifier declarations introduce new types to the program, of the forms described here. There are three kinds of classifier declarations:

- class declarations;
- interface declarations;
- object declarations.

  Important: object literals are similar to object declarations and are considered to be anonymous classifier declarations, despite being expressions.

4.1.1 Class declaration

A simple class declaration consists of the following parts.

- Name c;
- Optional primary constructor declaration $ptor$;
- Optional supertype specifiers $S_1, \ldots, S_s$;
- Optional body b, which may include the following:
  - secondary constructor declarations $stor_1, \ldots, stor_c$;
  - instance initialization blocks $init_1, \ldots, init_i$;
  - property declarations $prop_1, \ldots, prop_p$;
  - function declarations $md_1, \ldots, md_m$;
  - companion object declaration $companionObj$;
  - nested classifier declarations $nested$.

and creates a simple classifier type $c : S_1, \ldots, S_s$.

Supertype specifiers are used to create inheritance relation between the declared type and the specified supertype. You can use classes and interfaces as supertypes, but not objects or inner classes.

> Note: if supertype specifiers are absent, the declared type is considered to be implicitly derived from `kotlin.Any`.

It is allowed to inherit from a single class only, i.e., multiple class inheritance is not supported. Multiple interface inheritance is allowed.

Instance initialization block describes a block of code which should be executed during object creation.

Property and function declarations in the class body introduce their respective entities in this class' scope, meaning they are available only on an entity of the corresponding class.

Companion object declaration `companion object CO { ... }` for class `C` introduces an object, which is available under this class' name or under the path `C.CO`. Companion object name may be omitted, in which case it is considered to be equal to `Companion`.

Nested classifier declarations introduce new classifiers, available under this class' name. Further details are available here.

A parameterized class declaration, in addition to what constitutes a simple class declaration, also has a type parameter list $T_1, \ldots, T_m$ and extends the rules for a simple class declaration w.r.t. this type parameter list. Further details are described here.

Examples:

```
// An open class with no supertypes
//
open class Base

// A class inherited from `Base`
//
// Has a single read-only property `i`
//   declared in its primary constructor
//
class B(val i: Int) : Base()

// An open class with no superclasses
//
// Has a single read-only property `i`
//   declared in its body
//
// Initial value for the property is calculated
//   in the init block
//
open class C(arg: Int) {
    val i: Int

    init {
        i = arg * arg
```

```
    }
}

// A class inherited from `C`
//   Does not have a primary constructor,
//   thus does not need to invoke the supertype constructor
//
// The secondary constructor delegates to the supertype constructor
class D : C {
    constructor(s: String) : super(s.toInt())
}

// An open class inherited from `Base`
//
// Has a companion object with a mutable property `name`
class E : Base() {
    companion object /* Companion */ {
        var name = "I am a companion object of E!"
    }
}
```

Example:

```
class Pair(val a: Int, val b: Int) : Comparable<Pair> {

    fun swap(): Pair = Pair(b, a)

    override fun compareTo(other: Pair): Int {
        val f = a.compareTo(other.a)
        if (f != 0) return f
        return b.compareTo(other.b)
    }

    companion object {
        fun duplet(a: Int) = Pair(a, a)
    }
}
```

Constructor declaration

There are two types of class constructors in Kotlin: primary and secondary.

A primary constructor is a concise way of describing class properties together with constructor parameters, and has the following form

$$ptor : (p_1, \ldots, p_n)$$

where each of p_i may be one of the following:

- regular constructor parameter *name* : *type*;
- read-only property constructor parameter `val` *name* : *type*;
- mutable property constructor parameter `var` *name* : *type*.

Property constructor parameters, together with being regular constructor parameters, also declare class properties of the same name and type.

> Important: if a property constructor parameter with type T is specified as `vararg`, its corresponding class property type is the result of *array type specialization* of type `Array<out` T`>`.

One can consider primary constructor parameters to have the following syntactic expansion.

```
class Foo(i: Int, vararg val d: Double, var s: String) : Super(i, d, s) {}

class Foo(i: Int, vararg d_: Double, s_: String) : Super(i, d_, s_) {
  val d = d_
  var s = s_
}
```

When accessing property constructor parameters inside the class body, one works with their corresponding properties; however, when accessing them in the supertype specifier list (e.g., as an argument to a superclass constructor invocation), we see them as actual parameters, which cannot be changed.

If a class declaration has a primary constructor and also includes a class supertype specifier, that specifier must represent a valid invocation of the supertype constructor.

A secondary constructor describes an alternative way of creating a class instance and has only regular constructor parameters.

If a class has a primary constructor, any secondary constructor must delegate to either the primary constructor or to another secondary constructor via `this(...)`.

If a class does not have a primary constructor, its secondary constructors must delegate to either the superclass constructor via `super(...)` (if the superclass is present in the supertype specifier list) or to another secondary constructor via `this(...)`. If the only superclass is `kotlin.Any`, delegation is optional.

In all cases, it is forbidden if two or more secondary constructors form a delegation loop.

Class constructors (both primary and secondary) may have variable-argument parameters and default parameter values, just as regular functions. Please refer to the function declaration reference for details.

If a class does not have neither primary, nor secondary constructors, it is assumed to implicitly have a default parameterless primary constructor. This also means that, if a class declaration includes a class supertype specifier, that specifier must represent a valid invocation of the supertype constructor.

Examples:

```
open class Base

class POKO : Base() {}

class NotQuitePOKO : Base {
    constructor() : super() {}
}

class Primary(val s: String) : Base() {
    constructor(i: Int) : this(i.toString()) {}

    constructor(d: Double) : this(d.toInt()) {}

    // Error, has primary ctor,
    //   needs to delegate to primary or secondary ctor
    // constructor() : super() {}
}

class Secondary : Base {
    constructor(i: Int) : super() {}

    constructor(s: String) : this(s.toInt()) {}

    // Ok, no primary ctor,
    //   can delegate to `super(...)`
    constructor() : super() {}
}
```

Constructor declaration scopes

Similar to function declarations, a constructor introduces two scopes: a constructor parameter scope and a constructor body scope, see function declaration section for details. The constructor parameter scope is upward-linked to the static classifier declaration scope of its classifier. In addition to this, primary constructor parameter scope is downward-linked to the classifier initialization scope. There is also no primary constructor body scope as primary constructor has no body.

Nested and inner classifiers

If a classifier declaration ND is *nested* in another classifier declaration PD, it

creates a nested classifier type — a classifier type available under the path PD . ND. In all other aspects, nested classifiers are equivalent to regular ones.

Inner classes are a special kind of nested classifiers, which introduce types of objects associated (linked) with other (parent) objects. An inner class declaration ID nested in another classifier declaration PD may reference an *object* of type ID associated with it.

This association happens when instantiating an object of type ID, as its constructor may be invoked only when a receiver of type PD is available, and this receiver becomes associated with the new instantiated object of type ID.

Inner classes cannot be declared in interface declarations, as interfaces cannot be instantiated.

Inner classes cannot be declared in a statement scope, as such scope does not have an object to associate the inner class with.

Inner classes cannot be declared in object declarations, as object declarations also create a single named value of their type, which makes additional association unnecessary.

> Note: for information on how type parameters of parent and nested / inner classifiers interoperate, we delegate you to the type system section of the specification.
>
> Note: unlike object declarations, in object literals only inner classes are allowed, as types of object literals are anonymous, making their nested classifiers available only through explicit receiver, effectively forcing them to be inner.
>
> Examples:

```
interface Quz {
    interface Bar
    class Nested
    // Error: no parent object to reference,
    //   as interfaces cannot be instantiated
    // inner class Inner
}

class Foo {
    interface Bar
    class Nested
    inner class Inner
}

object Single {
    interface Bar
    class Nested
```

```
    // Error: value of type Single is available as-is,
    //   no reason to make an inner class
    // inner class Inner
}

fun foo() {
    // Error: interfaces cannot be local
    // interface Bar

    class Nested

    // Error: inner classes cannot be local
    // inner class Inner
}

fun test() {
    val fooV = Foo()

    Quz.Nested()
    Foo.Nested()
    fooV.Inner()

    Single.Nested()

    val anon = object {
        // Error: cannot reference <anon>.Bar
        // interface Bar
        // Error: cannot reference <anon>.Nested
        // class Nested
        inner class Inner
    }

    anon.Inner()
}
```

Inheritance delegation

In a classifier (an object or a class) declaration C, any supertype I inheritance may be *delegated to* an arbitrary value v if:

- The supertype I is an interface type;
- v has type T such that $T <: I$.

The inheritance delegation uses a syntax similar to property delegation using the `by` keyword, but is specified in the classifier declaration header and is a very different concept. If inherited using delegation, each method M of I (whether they have a default implementation or not) is delegated to the corresponding

method of v as if it was overridden in C with all the parameter values directly passed to the corresponding method in v, unless the body of C itself has a suitable override of M (see the method overriding section).

The particular means on how v is stored inside the classifier object is platform-defined.

Due to the initialization order of a classifier object, the expression used to construct v can not access any of the classifier object properties or methods excluding the parameters of the primary constructor.

Example:

```
interface I {
    fun foo(value: Int): Double
    val bar: Long
}
interface J : I {
    fun fee(): Int
}

class C(delegatee: I): I by delegatee
```

is expanded to

```
interface I {
    fun foo(value: Int): Double
    val bar: Long
}
interface J : I {
    fun fee(): Int
}

class C(delegatee: I): I {
    val I$delegate = delegate

    override fun foo(value: Int): Double = I$delegate.foo(value)
    override val bar: Long
        get() = I$delegate.bar
}
```

Please note that the expression used as delegate is accessed exactly once when creating the object, e.g. if the delegate expression contains a mutable property access, this mutable property is accessed once during object construction and its subsequent changes do not affect the delegated interface functions. See classifier initialization section for details on the evaluation order of classifier initialization entities.

For example (assuming interface `I` from the previous example is defined):

```
var mut = object: J {...}

class D: I by mut // D delegates I to mutable property
```

is expanded to

```
var mut = object: J {...}
class D: I {
    val I$delegate = mut // mut is accessed only once

    override fun foo(value: Int): Double = I$delegate.foo(value)
    override val bar: Long
        get() = I$delegate.bar
}

mut = x1
val d1 = D() // d1 methods are delegated to x1
mut = x2
val d2 = D() // d2 methods are delegated to x2
// but d1 methods are still delegated to x1
```

Abstract classes

A class declaration can be marked `abstract`. Such classes *cannot* be instantiated directly; they are used as superclasses for other classes or objects.

Abstract classes may contain one or more abstract members: members without implementation, which should be implemented in a subtype of this abstract class.

4.1.2 Data class declaration

A data class *dataClass* is a special kind of class, which represents a product type constructed from a number of data properties $(dp_1, \ldots, dp_m)$, described in its primary constructor. Non-property constructor parameters are not allowed in the primary constructor of a data class. As such, data classes allow Kotlin to reduce the boilerplate and generate a number of additional data-relevant functions.

- `equals() / hashCode() / toString()` functions compliant with their contracts:
  - `equals(that)` returns true iff:
    - `that` has the same runtime type as `this`;
    - `this.prop == that.prop` returns `true` for every data property `prop`;
  - `hashCode()` returns the same numbers for values `A` and `B` if they are equal w.r.t. the generated `equals`;

  - `toString()` returns a string representations which is guaranteed to include the class name along with all the data properties' string representations.
- A `copy()` function for shallow object copying with the following properties:
  - It has the same number of parameters as the primary constructor with the same names and types;
  - It calls the primary constructor with the corresponding parameters at the corresponding positions;
  - It has defaults for all the parameters defaulting to the value of the corresponding property in `this` object.
- A number of `componentN()` functions for destructuring declaration:
  - For the data property at position N (**starting from 1**), the generated `component`N function has the same type as this property and returns the value of this property;
  - It has an `operator` modifier, allowing it to be used in destructuring declarations;
  - The number of these functions is the same as the number of data properties.

All these functions consider only data properties $\{dp_i\}$; e.g., your data class may include regular property declarations in its body, however, they will *not* be considered in the `equals()` implementation or have a `componentN()` generated for them.

There are several rules as to how these generated functions may be explicified or inherited.

> Note: a generated function is explicified, if its implementation (with matching function signature) is provided explicitly in the body of the data class. A generated function is inherited, if its implementation (with matching function signature) is taken from a supertype of the data class.

The declarations of `equals`, `hashCode` and `toString` may be explicified similarly to how overriding works in normal classes. If a correct explicit implementation is available, no function is generated. Other functions (`copy`, `componentN`) **cannot** be explicified.

The declarations of `equals`, `hashCode` and `toString` may be inherited from the base class, if it provides a `final` version with a matching signature. If a correct inherited implementation is available, no function is generated. Other functions (`copy`, `componentN`) **cannot** be inherited.

In addition, for every generated function, if any of the base types provide an open function with a matching signature, it is automatically overridden by the generated function as if it was generated with an `override` modifier.

> Note: data classes or their supertypes may also have functions which have a matching name and/or signature with one of the generated

functions. As expected, these cases result in either override or overload conflicts the same way they would with a normal class declaration, or they create two separate functions which follow the rules of overloading.

Data classes have the following restrictions:

- Data classes are closed and cannot be inherited from;
- Data classes must have a primary constructor with property constructor parameters only, which become data properties for the data class;
- There must be at least one data property in the primary constructor;
- Data properties cannot be specified as `vararg` constructor arguments.

For example, the following data class declaration

```
data class DC(val x: Int, val y: Double)
```

is equivalent to

```
class DC(val x: Int, val y: Double) {
    override fun equals(other: Any?): Boolean {
         if(other !is DC) return false
         return x == other.x && y == other.y
    }

    override fun hashCode(): Int = x.hashCode() + 31 * y.hashCode()

    override fun toString(): String = "DC(x=$x,y=$y)"

    operator fun component1(): Int = x

    operator fun component2(): Double = y

    fun copy(x: Int = this.x, y: Double = this.y): DC = DC(x, y)
}
```

The following data class declaration

```
data class DC(val x: Int) {
    override fun equals(other: Any?) = false
    override fun toString(): String = super.toString()
}
```

may be equivalent to

```
class DC(val x: Int) {
    override fun equals(other: Any?) = false

    override fun hashCode(): Int = x.hashCode()

    override fun toString(): String = super.toString()
```

```
    operator fun component1(): Int = x

    fun copy(x: Int = this.x): DC = DC(x)
}
```

(note how `equals` and `toString` implementations are explicified in the second declaration)

Disclaimer: the implementations of these methods given in this examples are not guaranteed to exactly match the ones generated by kotlin compiler, please refer to the descriptions of these methods above for guarantees

Data object declaration

Note: as of Kotlin 1.9, this feature is experimental.

A data object *dataObject* is a special kind of object, which extends the data class abstraction (product type of one or more data properties) to a case of unit type: product type of zero data properties.

Note: unit type has only one possible value, thus it is also known as singleton type.

Similarly to data classes, there are a number of functions with predefined behaviour generated for data objects.

- `equals() / hashCode() / toString()` functions compliant with their contracts:
  - `equals(that)` returns true iff `that` has the same runtime type as `this`;
  - `hashCode()` returns the same numbers for values `A` and `B` if they are equal w.r.t. the generated `equals`;
  - `toString()` returns a string representations which is guaranteed to include the object name.

Note: `copy()` and `componentN()` functions are not generated, as they are not relevant for a unit type.

- `copy()` function is not needed as unit type has a single possible value;
- `componentN()` functions are not needed as unit type has no data properties.

Unlike data classes, however, for data objects the only generated function which can be exemplified or inherited is `toString()`; `equals()` and `hashCode()` for a data object always work as specified above. This is to ensure data objects do not violate the unit type invariant of "being inhabited by only one value", which would be possible if one were to provide a custom `equals()` implementation.

If either `equals()` or `hashCode()` function would be exemplified or inherited by a data object, it is a compile-time error.

Data objects have the same restrictions are regular objects.

> Note: companion objects and object literals cannot be data objects.

4.1.3 Enum class declaration

Enum class E is a special kind of class with the following properties:

- It has a number of predefined values that are declared in the class itself (*enum entries*);
- No other values of this class can be constructed;
- It implicitly inherits the built-in class `kotlin.Enum<E>` (and cannot have any other base classes);
- It is implicitly final and cannot be inherited from;
- It cannot have type parameters of any kind;
- It has special syntax to accommodate for the properties described above.

> Note: for the purposes of overload resolution, enum entries are considered to be static member callables of the enum class type

Enum class body uses special kind of syntax (see grammar) to declare enum entries in addition to all other declarations inside the class body. Enum entries have their own bodies that may contain their own declarations, similar to object declarations.

> Note: an enum class can have zero enum entries. This makes objects of this class impossible to construct.

Every enum entry of class `E` implicitly overrides members of `kotlin.Enum<E>` in the following way:

- ```
  public final val name: String
  ```

  defined to be the same as the name of the entry as declared in code;

- ```
  public final val ordinal: Int
  ```

  defined to be the ordinal of the entry, e.g. the position of this entry in the list of entries, starting with 0;

- ```
  public override final fun compareTo(other: E): Int
  ```

  (a member of `kotlin.Comparable<E>`) defined by default to compare entries by their ordinals, but may be overridden to have different behaviour both in the enum class declaration and in entry declarations;

- ```
  public override fun toString(): String
  ```

(a member of `kotlin.Any`) defined by default to return the entry name, but may be overridden to have different behaviour both in the enum class declaration and in entry declarations.

In addition to these, every enum class type `E` has the following **static** members declared implicitly:

- `public final static val entries: EnumEntries<E>`

  This property returns an instance of a special immutable `EnumEntries<E>` list of all possible enum values in the order they are declared;

- `public final static fun valueOf(value: String): E`

  This function returns an object corresponding to the entry with the name equal to `value` parameter of the call or throws an exception otherwise.

  Important: `static` is not a valid Kotlin keyword and is only used here for clarity. The static members are handled differently by the overload resolution.

Kotlin standard library also introduces a function to access all enum values for a specific enum class called `kotlin.enumEntries<T>`. Please refer to the standard library documentation for details.

Note: the `entries` property is available since Kotlin 1.9.

For backwards compatibility, in addition to the `entries` property, every enum class type `E` has the following **static** member function declared implicitly.

- `public final static fun values(): kotlin.Array<E>`

  This function returns an array of all possible enum values in the order they are declared. Every invocation of this function returns a new array to disallow changing its contents.

  Important: `values` function is effectively deprecated and `entries` property should be used instead.

Kotlin standard library also introduces another function to access all enum values for a specific enum class called `kotlin.enumValues<T>` (which is deprecated for subsequent removal). Please refer to the standard library documentation for details.

Example:

```
enum class State { LIQUID, SOLID, GAS }

...
State.SOLID.name // "SOLID"
State.SOLID.ordinal // 1
State.GAS > State.LIQUID // true
State.SOLID.toString() // "SOLID"
```

```
State.valueOf("SOLID") // State.SOLID
State.valueOf("Foo") // throws exception
State.values() // arrayOf(State.LIQUID, State.SOLID, State.GAS)

...

// enum class can have additional declarations that may be overridden in its value
enum class Direction(val symbol: Char) {
    UP('^') {
        override val opposite: Direction
            get() = DOWN
    },
    DOWN('v') {
        override val opposite: Direction
            get() = UP
    },
    LEFT('<') {
        override val opposite: Direction
            get() = RIGHT
    },
    RIGHT('>') {
        override val opposite: Direction
            get() = LEFT
    };
    abstract val opposite: Direction
}
```

4.1.4 Annotation class declaration

Annotations class is a special kind of class that is used to declare annotations. Annotation classes have the following properties:

- They cannot have any secondary constructors;
- All the primary constructor parameters must use the property syntax;
- They implicitly implement `kotlin.Annotation` interface (and cannot implement additional interfaces);
- They cannot have any specified base classes;
- They are implicitly closed and cannot be inherited from;
- They may not have any member functions, properties not declared in the primary constructor or any overriding declarations;
- They cannot have companion objects;
- They cannot have nested classes;
- The types of primary constructor parameters are limited to:
  - `kotlin.String`;
  - `kotlin.KClass`;
  - Built-in number types;

- Other annotation types;
- Arrays of any other allowed type.

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.

Note: annotation classes can have type parameters, but cannot use them as types for their primary constructor parameters. Their main use is for various annotation processing tools, which can access the type arguments from the source code.

The main use of annotation classes is when specifying code annotations for other entities. Additionally, annotation classes can be instantiated directly, for cases when you require working with an annotation instance directly. For example, this is needed for interoperability with some Java annotation APIs, as in Java you can implement an annotation interface and then instantiate it.

Note: before Kotlin 1.6, annotation classes could not be instantiated directly.

Examples:

```
// a couple annotation classes
annotation class Super(val x: Int, val f: Float = 3.14f)
annotation class Duper(val supers: Array<Super>)

// the same classes used as annotations
@Duper(arrayOf(Super(2, 3.1f), Super(3)))
class SuperClass {
    @Super(4)
    val x = 3
}

// annotation class without parameters
annotation class Transmogrifiable

@Transmogrifiable
fun f(): Int = TODO()

// variable argument properties are supported
annotation class WithTypes(vararg val classes: KClass<out Annotation>)

@WithTypes(Super::class, Transmogrifiable::class)
val x = 4
```

4.1.5 Value class declaration

A class may be declared a **value** class by using `inline` or `value` modifier in its declaration. Value classes must adhere to the following limitations:

- Value classes are closed and cannot be inherited from;
- Value classes cannot be `inner`, `data` or `enum` classes;
- Value classes must have a primary constructor with a single property constructor parameter, which is the data property of the class;
- This property cannot be specified as `vararg` constructor argument;
- This property must be declared `public`;
- They must not override `equals` and `hashCode` member functions of `kotlin.Any`;
- They must not have any base classes besides `kotlin.Any`;
- No other properties of this class may have backing fields.

  Note: `inline` modifier for value classes is supported as a legacy feature for compatibility with Kotlin 1.4 experimental inline classes and will be deprecated in the future.

  Note: before Kotlin 1.8, value classes supported only properties of [a runtime-available types].

Value classes implicitly override `equals` and `hashCode` member functions of `kotlin.Any` by delegating them to their only data property. Unless `toString` is overridden by the value class definition, it is also implicitly overridden by delegating to the data property. In addition to these, an value class is allowed by the implementation to be **inlined** where applicable, so that its data property is operated on instead. This also means that the property may be boxed back to the value class by using its primary constructor at any time if the compiler decides it is the right thing to do.

> Note: when inlining a data property of a non-runtime-available type U (i.e., a non-reified type parameter), the property is considered to be of type, which is the runtime-available upper bound of U.

Due to these restrictions, it is highly discouraged to use value classes with the reference equality operators.

> Note: in the future versions of Kotlin, value classes may be allowed to have more than one data property.

4.1.6 Interface declaration

Interfaces differ from classes in that they cannot be directly instantiated in the program, they are meant as a way of describing a contract which should be satisfied by the interface's subtypes. In other aspects they are similar to classes, therefore we shall specify their declarations by specifying their differences from class declarations.

- An interface can be declared only in a declaration scope;
  - Additionally, an interface cannot be declared in an object literal;
- An interface cannot have a class as its supertype;
  - This also means it is not considered to have `kotlin.Any` as its supertype for the purposes of inheriting and overriding callables;
  - However, it is still considered to be a subtype of `kotlin.Any` w.r.t. subtyping;
- An interface cannot have a constructor;
- Interface properties cannot have initializers or backing fields;
- Interface properties cannot be delegated;
- An interface cannot have inner classes;
- An interface and all its members are implicitly open;
- All interface member properties and functions are implicitly public;
  - Trying to declare a non-public member property or function in an interface is an compile-time error;
- Interface member properties and functions without implementation are implicitly abstract.

Functional interface declaration

A *functional interface* is an interface with a **single** abstract function and no other abstract properties or functions.

A function interface declaration is marked as `fun interface`. It has the following additional restrictions compared to regular interface declarations.

- A functional interface can have only one abstract member function, which must be non-parameterized;
- A functional interface cannot have any abstract member properties;

A functional interface has an associated function type, which is the same as the function type of its single abstract member function.

> Important: the associated function type of a functional interface is different from the type of said functional interface.

If one needs an object of a functional interface type, they can use the regular ways of implementing an interface, either via an anonymous object declaration or as a complete class. However, as functional interface essentially represents a single function, Kotlin supports the following additional ways of providing a functional interface implementation from function values (expressions with function type).

- If an expression `L` is used as an argument of functional type `T` in a function call, and the type of `L` is a subtype of the associated function type of `T`, this argument is considered as an instance of `T` with expression `L` used as its abstract member function implementation.

  Example:

```
fun interface FI {
    fun bar(s: Int): Int
}

fun doIt(fi: FI) {}

fun foo() {
    doIt { it }
    doIt { s: Int -> s + 42 }
    doIt { s: Number -> s.toInt() }

    doIt(fun(s): Int { return s; })

    val l = { s: Number -> s.toInt() }

    doIt(l)
}
```

- When encountered in a function call as the *function being called*, a functional interface name `T` is considered to be representing a function of type `(T) -> T`, which allows conversion-like function calls as in the examples below.

  Example:

  ```
  fun interface FI {
      fun bar(s: Int): Int
  }

  fun foo() {
      val fi = FI { it }
      val fi2 = FI { s: Int -> s + 42 }
      val fi3 = FI { s: Number -> s.toInt() }
      val fi4 = FI({ it })

      val lambda = { s: Int -> s + 42 }
      val fi5 = FI(lambda)
  }
  ```

  Informally: this feature is known as "Single Abstract Method" (SAM) conversion.

  Note: in Kotlin version 1.3 and earlier, SAM conversion was not available for Kotlin functional interfaces.

  Note: SAM conversion is also available on Kotlin/JVM for Java functional interfaces.

4.1.7 Object declaration

Object declarations are similar to class declaration in that they introduce a new classifier type, but, unlike class or interface declarations, they also introduce a value of this type in the same declaration. No other values of this type may be declared, making object a single existing value of its type.

> Note: This is similar to *singleton pattern* common to object-oriented programming in introducing a type which includes a single global value.

Similarly to interfaces, we shall specify object declarations by highlighting their differences from class declarations.

- An object can only be declared in a declaration scope;
  - Additionally, an object cannot be declared in an object literal;
- An object type cannot be used as a supertype for other types;
- An object cannot have an explicit primary or secondary constructor;
- An object cannot have a companion object;
- An object cannot have inner classes;
- An object cannot be parameterized, i.e., cannot have type parameters.

> Note: an object is assumed to implicitly have a default parameterless primary constructor.
>
> Note: this section is about declaration of *named* objects. Kotlin also has a concept of *anonymous* objects, or object literals, which are similar to their named counterparts, but are expressions rather than declarations and, as such, are described in the corresponding section.
>
> Note: besides regular object declarations, Kotlin supports data object declarations.

4.1.8 Local class declaration

A class (but not an interface or an object) may be declared *locally* inside a statement scope (namely, inside a function). Such declarations are similar to object literals in that they may capture values available in the scope they are declared in:

```
fun foo() {
    val x = 2
    class Local {
        val y = x
    }
    Local().y // 2
}
```

Enum classes and annotation classes cannot be declared locally.

4.1.9 Classifier initialization

When creating a class or object instance via one of its constructors *ctor*, it is initialized in a particular order, which we describe here.

A primary *pctor* or secondary constructor *ctor* has a corresponding superclass constructor *sctor* defined as follows.

- For primary constructor *pctor*, a corresponding superclass constructor *sctor* is the one from the supertype specifier list;
- For secondary constructor *ctor*, a corresponding supertype constructor *sctor* is the one ending the constructor delegation chain of *ctor*;
- If an explicit superclass constructor is not available, `Any()` is implicitly used.

When a classifier type is initialized using a particular secondary constructor *ctor* delegated to primary constructor *pctor* which, in turn, is delegated to the corresponding superclass constructor *sctor*, the following happens, in this *initialization order*:

- The superclass object is initialized as if created by invoking *sctor* with the specified parameters;
- Interface delegation expressions are invoked and the result of each is stored in the object to allow for interface delegation, *in the order of appearance of delegation declarations in the supertype specifier list*;
- *pctor* is invoked using the specified parameters, initializing all the properties declared by its property parameters *in the order of appearance in the constructor declaration*;
- Each property initialization code as well as the initialization blocks in the class body are invoked *in the order of appearance in the class body*;
- *ctor* body is invoked using the specified parameters.

  Note: this means that if an `init`-block appears between two property declarations in the class body, its body is invoked between the initialization code of these two properties.

The initialization order stays the same if any of the entities involved are omitted, in which case the corresponding step is also omitted (e.g., if the object is created using the primary constructor, the body of the secondary one is not invoked).

If any step in the initialization order creates a loop, it results in unspecified behaviour.

If any of the properties are accessed before they are initialized w.r.t initialization order (e.g., if a method called in an initialization block accesses a property declared *after* the initialization block), the value of the property is unspecified. It stays unspecified even after the "proper" initialization is performed.

> Note: this can also happen if a property is captured in a lambda expression used in some way during subsequent initialization steps.

Examples:

```
open class Base(val v: Any?) {
    init {
        println("2: $this")
    }
}

interface I

class Init(val a: Number) : Base(0xC0FFEE) /* (2) */,
    I by object : I {
        init { println("2.5") }
    } /* (2.5) */ {

    init {
        println("3: $this") /* (3) */
    }

    constructor(v: Int) : this(v as Number) {
        println("10: $this") /* (10) */
    }

    val b: String = a.toString() /* (4) */

    init {
        println("5: $this") /* (5) */
    }

    var c: Any? = "b is $b" /* (6) */

    init {
        println("7: $this") /* (7) */
    }

    val d: Double = 42.0 /* (8) */

    init {
        println("9: $this") /* (9) */
    }

    override fun toString(): String {
        return "Init(a=$a, b='$b', c=$c, d=$d)"
    }
}
```

```
fun main() {
    Init(5)
    // 2: Init(a=null, b='null', c=null, d=0.0)
    // 3: Init(a=5, b='null', c=null, d=0.0)
    // 5: Init(a=5, b='5', c=null, d=0.0)
    // 7: Init(a=5, b='5', c=b is 5, d=0.0)
    // 9: Init(a=5, b='5', c=b is 5, d=42.0)
    // 10: Init(a=5, b='5', c=b is 5, d=42.0)

    // Here we can see how the undefined values for
    //   uninitialized properties may leak outside
}
```

4.1.10 Classifier declaration scopes

Every classifier declaration introduces two declarations scope syntactically bound by the classifier body, if any: the **static** classifier body scope and the **actual** classifier body scope Every function, property or inner classifier declaration contained within the classifier body are declared in the actual classifier body scope of this classifier. All non-primary constructors of the classifier, as well as any non-inner nested classifier, including the companion object declaration (if it exists) and enum entries (if this is an enum class), are declared in the static classifier body scope. Static classifier body scope is upwards-linked to the actual classifier body scope. For an object declaration, static classifier body scope and the actual classifier body scoped are one and the same.

In addition to this, objects and classes introduce a special *object initialization scope*, which is not syntactically delimited. The scopes of each initialization expression of every property in the class body, as well as the scopes of each initialization block, is upward-linked to the object initialization scope, which itself is upward-linked to the actual classifier body scope.

If a classifier declares a primary constructor, the parameters of this constructor are bound in the special *primary constructor parameter scope*, which is downward-linked to the initialization scope and upward-linked to the scope the classifier is declared in. The interface delegation expressions (if any) are resolved in the primary constructor parameter scope if it exists and in the scope the classifier is declared in otherwise.

4.2 Function declaration

***functionDeclaration*:**
[*modifiers*]
`'fun'`
[{*NL*} *typeParameters*]
[{*NL*} *receiverType* {*NL*} `'.'`]

{*NL*}
simpleIdentifier
{*NL*}
functionValueParameters
[{*NL*} `':'` {*NL*} *type*]
[{*NL*} *typeConstraints*]
[{*NL*} *functionBody*]

***functionBody*:**
block
| (`'='` {*NL*} *expression*)

Function declarations assign names to functions — blocks of code which may be called by passing them a number of arguments. Functions have special *function types* which are covered in more detail here.

A simple function declaration consists of four main parts:

- Name f;
- Parameter list $(p_1 : P_1[= v_1], \ldots, p_n : P_n[= v_n])$;
- Return type R;
- Body b.

and has a function type $f : (p_1 : P_1, \ldots, p_n : P_n) \to R$.

Parameter list $(p_1 : P_1[= v_1], \ldots, p_n : P_n[= v_n])$ describes function parameters, i.e. inputs needed to execute the declared function. Each parameter $p_i : P_i = v_i$ introduces p_i as a name of value with type P_i available inside function body b; therefore, parameters are final and cannot be changed inside the function. A function may have zero or more parameters.

A parameter may include a default value v_i, which is used if the corresponding argument is not specified in function invocation; v_i must be an expression which evaluates to type $V <: P_i$.

Return type R, if omitted, is calculated as follows.

- If function body b is present in the expression form and it may be inferred to have a valid type $B : B \not\equiv$ `kotlin.Nothing`, $R \equiv B$.
- If function body b is present in the block form, $R \equiv$ `kotlin.Unit`.

In other cases return type R cannot be omitted and must be specified explicitly.

> As type `kotlin.Nothing` has a special meaning in Kotlin type system, it must be specified explicitly, to avoid spurious `kotlin.Nothing` function return types.

Function body b is optional; if it is omitted, a function declaration creates an *abstract* function, which does not have an implementation. This is allowed only inside an abstract class or an interface. If a function body b is present, it should evaluate to type B which should satisfy $B <: R$.

A parameterized function declaration consists of five main parts.

- Name f;
- Type parameter list $T_1, \ldots, T_m$;
- Parameter list $(p_1 : P_1 = v_1, \ldots, p_n : P_n = v_n)$;
- Return type R;
- Body b.

and extends the rules for a simple function declaration w.r.t. type parameter list. Further details are described here.

4.2.1 Function signature

In some cases we need to establish whether one function declaration *matches* another, e.g., for checking overridability. To do that, we compare *function signatures*, which consist of the following.

- Name f;
- Type parameter list $T_1, \ldots, T_m$ (if present);
- Parameter list $P_1, \ldots, P_n$.

Two function signatures A and B are considered *matching*, if the following is true.

- Name of A is the same as the name of B;
- Formal parameter types of A are pairwise equal to the formal parameter types of B w.r.t. possible type parameter substitutions;
- If the number of type parameters is the same, type parameters of A must be pairwise equivalent to the type parameters of B.

  Important: a platform implementation may change which function signatures are considered matching, depending on the platform's specifics.

4.2.2 Named, positional and default parameters

Kotlin supports *named* parameters out-of-the-box, meaning one can bind an argument to a parameter in function invocation not by its position, but by its name, which is equal to the argument name.

```
fun bar(a: Int, b: Double, s: String): Double = a + b + s.toDouble()

fun main(args: Array<String>) {
    println(bar(b = 42.0, a = 5, s = "13"))
}
```

Note: it is prohibited to bind the same named parameter to an argument several times, such invocations should result in a compile-time error.

All the names of named parameters are resolved at compile-time, meaning that performing a call with a parameter name not used at declaration-site is a compile-time error.

If one wants to mix named and positional arguments, the argument list must conform to the following form: $PoN_1, \ldots, PoN_M, N_1, \ldots, N_Q$, where PoN_i is an i-th argument in either positional or named form, N_j is a named argument irregardless of its position.

> Note: in Kotlin version 1.3 and earlier, PoN_i were restricted to positional arguments only.

If one needs to provide a named argument to a variable length parameter, it can be achieved via either regular named argument `arg = arr` or a spread operator expression form `arg = *arr`. In both cases type of `arr` must be a subtype of ATS(`kotlin.Array(out` T)) for a variable length parameter of type T.

> Note: in Kotlin version 1.3 and earlier, only the spread operator expression form for named variable length arguments was supported.

Kotlin also supports *default* parameters — parameters which have a default value used in function invocation, if the corresponding argument is missing. Note that default parameters cannot be used to provide a value for positional argument *in the middle* of the positional argument list; allowing this would create an ambiguity of which argument for position i is the correct one: explicit one provided by the developer or implicit one from the default value.

```
fun bar(a: Int = 1, b: Double = 42.0, s: String = "Hello"): Double =
    a + b + s.toDouble()

fun main(args: Array<String>) {
    // Valid call, all default parameters used
    println(bar())
    // Valid call, defaults for `b` and `s` used
    println(bar(2))
    // Valid call, default for `b` used
    println(bar(2, s = "Me"))

    // Invalid call, default for `b` cannot be used
    println(bar(2, "Me"))
}
```

In summary, argument list should have the following form:

- Zero or more arguments in either positional or named form;
- Zero or more named arguments.

Missing arguments are bound to their default values, if they exist.

The evaluation order of argument list is described in Function calls and property access section of this specification.

4.2.3 Variable length parameters

One of the parameters may be designated as being variable length (aka *vararg*). A parameter list $(p_1, \ldots, \text{vararg } p_i : P_i = v_i, \ldots, p_n)$ means a function may be called with any number of arguments in the i-th position. These arguments are represented inside function body b as a value p_i of type, which is the result of *array type specialization* of type `kotlin.Array`(`out` P_i).

> Important: we also consider variable length parameters to have such types for the purposes of type inference and calls with named parameters.

If a variable length parameter is not last in the parameter list, all subsequent arguments in the function invocation should be specified as named arguments.

If a variable length parameter has a default value, it should be an expression which evaluates to a value of type, which is the result of *array type specialization* of type `kotlin.Array`(`out` P_i).

A value of type $Q <:$ ATS(`kotlin.Array`(`out` P_i)) may be *unpacked* to a variable length parameter in function invocation using spread operator; in this case array elements are considered to be separate arguments in the variable length parameter position.

> Note: this means that, for variable length parameters corresponding to specialized array types, unpacking is possible only for these specialized versions; for a variable length parameter of type `Int`, for example, unpacking is valid only for `IntArray`, and not for `Array<Int>`.

A function invocation may include several spread operator expressions corresponding to the vararg parameter. These may also be freely mixed with non-spread-expression arguments.

> Examples

```
fun foo(vararg i: Int) { ... }
fun intArrayOf(vararg i: Int): IntArray = i
...
// i is [1, 2, 3]
foo(1, 2, 3)
// i is [1, 2, 3]
foo(*intArrayOf(1, 2, 3))
// i is [1, 2, 3, 4, 5]
foo(1, 2, *intArrayOf(3, 4), 5)
// i is [1, 2, 3, 4, 5, 6]
foo(*intArrayOf(1, 2, 3), 4, *intArrayOf(5, 6))
```

4.2.4 Extension function declaration

An *extension function declaration* is similar to a standard function declaration,

but introduces an additional special function parameter, the *receiver parameter*. This parameter is designated by specifying the receiver type (the type before `.` in function name), which becomes the type of this receiver parameter. This parameter is not named and must always be supplied (either explicitly or implicitly), e.g. it cannot be a variable-argument parameter, have a default value, etc.

Calling such a function is special because the receiver parameter is not supplied as an argument of the call, but as the *receiver* of the call, be it implicit or explicit. This parameter is available inside the scope of the function as the implicit receiver or `this`-expression, while nested scopes may introduce additional receivers that take precedence over this one. See the receiver section for details. This receiver is also available (as usual) in nested scope using labeled `this` syntax using the name of the declared function as the label.

For more information on how a particular receiver for each call is chosen, please refer to the overloading section.

> Note: when declaring extension functions inside classifier declarations, this receiver takes precedence over the classifier object, which is usually the current receiver inside nested functions

For all other purposes, extension functions are not different from non-extension functions.

Examples:

```
fun Int.foo() { println(this + 1) } // this has type Int

fun main(args: Array<String>) {
    2.foo() // prints "3"
}

class Bar {
    fun foo() { println(this) } // this has type Bar
    fun Int.foo() { println(this) } // this has type Int
}
```

4.2.5 Inlining

A function may be declared `inline` using a special `inline` modifier. This allows the compiler to inline the function at call-site, replacing the call with the body of the function with arguments mapped to corresponding parameters. It is unspecified whether inlining will actually be performed, however.

Declaring a function `inline` has two additional effects:

- It allows type parameters of the function to be declared `reified`, making them runtime-available and allowing usage of specific expressions involving these parameters, such as type checks and class literals. Calling such a

function is only allowed in a context where a particular type argument provided for this type parameter is also a runtime-available type.
- Any parameter of this function of a function type is treated as *inlined* parameter unless it has one of two special modifiers: `crossinline` or `noinline`. If a particular argument corresponding to inline parameter is a lambda literal, this lambda literal is considered *inlined* and, in particular, affects the way the return expressions are handled in its body. See the corresponding section for details.

Inlined parameters are not allowed to escape the scope of the function body, meaning that they cannot be stored in variables, returned from the function or captured by other values. They may only be called inside the function body or passed to other functions as inline arguments.

Crossinline parameters may not be stored or returned from the function, but may be captured (for example, by object literals or other noinline lambda literals).

Noinline parameters may be treated as any other values. They may also be passed to other functions as `noinline` or `crossinline` arguments.

Particular platforms may introduce additional restrictions or guarantees for the inlining mechanism.

> Important: for extension functions, the extension receiver is considered to be effectively `noinline`.

Examples:

```
fun bar(value: Any?) {}

inline fun inlineParameter(arg: () -> Unit) { arg() }
inline fun noinlineParameter(noinline arg: () -> Unit) { arg() }
inline fun crossinlineParameter(crossinline arg: () -> Unit) { arg() }

inline fun foo(inl: () -> Unit,
               crossinline cinl: () -> Unit,
               noinline noinl: () -> Unit) {
    // all arguments may be called
    inl()
    cinl()
    noinl()
    // all arguments may be passed as inline
    inlineParameter(inl)
    inlineParameter(cinl)
    inlineParameter(noinl)
    // only noinline arguments may be passed as noinline
    noinlineParameter(inl) // not allowed
    noinlineParameter(cinl) // not allowed
    noinlineParameter(noinl)
```

```
        // noinline/crossinline arguments may be passed as crossinline
        crossinlineParameter(inl) // not allowed
        crossinlineParameter(cinl)
        crossinlineParameter(noinl)
        // only noinline arguments may be passed to non-inline functions
        bar(inl) // not allowed
        bar(cinl) // not allowed
        bar(noinl)
        // noinline/crossinline parameters may be captured in lambda literals
        bar({ inl() }) // not allowed
        bar({ cinl() })
        bar({ noinl() })
    }
```

4.2.6 Infix functions

A function may be declared as an *infix* function by using a special `infix` modifier. An infix function can be called in an infix form, i.e., `a foo b` instead of `a.foo(b)`.

To be a valid infix function, function F must satisfy the following requirements.

- F has a dispatch or an extension receiver
- F has exactly one parameter

4.2.7 Local function declaration

A function may be declared *locally* inside a statement scope (namely, inside another function). Such declarations are similar to function literals in that they may capture values available in the scope they are declared in. Otherwise they are similar to regular function declarations.

```
fun foo() {
    var x = 2

    fun bar(): Int {
        return x
    }

    println(bar()) // 2

    x = 42
    println(bar()) // 42
}
```

4.2.8 Tail recursion optimization

A function may be declared *tail-recursive* by using a special `tailrec` modifier. A tail-recursive function that contains a recursive call to itself may be optimized

to a non-recursive form by a particular platform in order to avoid problems of recursion such as a possibility of stack overflows possible on some platforms.

In order to be applicable for such an optimization, the function must adhere to tail recursive form: for all paths containing recursive calls the result of the recursive call must also be the result of the function. If a function declaration is marked with the `tailrec` modifier, but is not actually applicable for the optimization, it must produce a compile-time warning.

Examples:

```
// this is not a tail-recursive function
// so tailrec modifier will produce a warning
tailrec fun factorial(i: Int): Int {
    if (i == 0) return 1
    return i * factorial(i - 1)
}
// this is a tail-recursive function
tailrec fun factorialTC(i: Int, result: Int = 1): Int {
    if (i == 0) return result
    return factorialTC(i - 1, i * result)
}
```

`factorialTC` declaration given above should be compiled to loop form similar to the following declaration

```
fun factorialLoop(i: Int, result: Int = 1): Int {
    var $i: Int = i
    var $result: Int = result
    while(true) {
       if ($i == 0) return $result
       else {
           $i = $i - 1
           $result = $i * $result
       }
    }
}
```

4.2.9 Function declaration scopes

Every function declaration body introduces a *function body scope*, which is a statement scope containing everything declared inside the function body and is delimited by the function body itself.

In addition to this scope, function parameters exist in a special *function parameter scope*, which is upward-linked to the scope the function is declared in and downward-linked to the function body scope.

4.3 Property declaration

***propertyDeclaration*:**
[*modifiers*]
('val' | 'var')
[{*NL*} *typeParameters*]
[{*NL*} *receiverType* {*NL*} '.']
({*NL*} (*multiVariableDeclaration* | *variableDeclaration*))
[{*NL*} *typeConstraints*]
[{*NL*} (('=' {*NL*} *expression*) | *propertyDelegate*)]
[{*NL*} ';']
{*NL*}
(([*getter*] [{*NL*} [*semi*] *setter*]) | ([*setter*] [{*NL*} [*semi*] *getter*]))

Kotlin uses *properties* to represent object-like entities, such as local variables, class fields or top-level values.

Property declarations may create read-only (`val`) or mutable (`var`) entities in their respective scope.

Properties may also have custom getter or setter — special functions which are used to read or write the property value. Getters and setters cannot be called directly, but rather define how the corresponding properties are evaluated when accessed.

4.3.1 Read-only property declaration

A read-only property declaration `val x: T = e` introduces `x` as a name of the result of `e`.

A read-only property declaration may include a custom getter in the form of

```
val x: T = e
    get(): T { ... } // (1)
```

or

```
val x: T = e
    get(): T = ... // (2)
```

in which case `x` is used as a synonym to the getter invocation. All of the right-hand value `e`, the type `T` in both positions, and the getter are optional, however, at least one of them must be specified. More so, if we cannot infer the resulting property type from the type of `e` or from the type of getter in expression form (2), the type `T` must be specified explicitly either as the property type, or as the getter return type. In case both `e` and `T` are specified, the type of `e` must be a subtype of `T` (see subtyping for more details).

The initializer expression `e`, if given, serves as the starting value for the property backing field (see getters and setters section for details) and is evaluated when

the property is created. Properties that are not allowed to have backing fields (see getters and setters section for details) are also not allowed to have initializer expressions.

> Note: although a property with an initializer expression looks similar to an assignment, it is different in several key ways: first, a read-only property cannot be assigned, but may have an initializer expression; second, the initializer expression never invokes the property setter, but assigns the property backing field value directly.

4.3.2 Mutable property declaration

A mutable property declaration `var x: T = e` introduces `x` as a name of a mutable variable with type `T` and initial value equals to the result of `e`. The rules regarding the right-hand value `e` and the type `T` match those of a read-only property declaration.

A mutable property declaration may include a custom getter and/or custom setter in the form of

```
var x: T = e
    get(): TG { ... }
    set(value: TS) { ... }
```

in which case `x` is used as a synonym to the getter invocation when read from and to the setter invocation when written to.

4.3.3 Local property declaration

If a property declaration is local, it creates a local entity which follows most of the same rules as the ones for regular property declarations. However, local property declarations cannot have custom getters or setters.

Local property declarations also support *destructuring* declaration in the form of

```
val (a: T, b: U, c: V, ...) = e
```

which is a syntactic sugar for the following expansion

```
val a: T = e.component1()
val b: U = e.component2()
val c: V = e.component3()
...
```

where `componentN()` should be a valid operator function available on the result of `e`. Some of the entries in the destructuring declaration may be replaced with an *ignore marker* `_`, which signifies that no variable is declared and no `componentN()` function is called.

As with regular property declaration, type specification is optional, in which case the type is inferred from the corresponding `componentN()` function. De-

structuring declarations cannot use getters, setters or delegates and must be initialized in-place.

4.3.4 Getters and setters

As mentioned before, a property declaration may include a custom getter and/or custom setter (together called *accessors*) in the form of

```
var x: T = e
    get(): TG { ... }
    set(anyValidArgumentName: TS): RT { ... }
```

These functions have the following requirements

- $TG \equiv T$;
- $TS \equiv T$;
- $RT \equiv$ `kotlin.Unit`;
- Types TG, TS and RT are optional and may be omitted from the declaration;
- Read-only properties may have a custom getter, but not a custom setter;
- Mutable properties may have any combination of a custom getter and a custom setter
- Setter argument may have any valid identifier as argument name.

  Note: Regular coding convention recommends `value` as the name for the setter argument

One can also omit the accessor body, in which case a *default* implementation is used (also known as default accessor).

```
var x: T = e
    get
    set
```

This notation is usually used if you need to change some aspects of an accessor (i.e., its visibility) without changing the default implementation.

Getters and setters allow one to customize how the property is accessed, and may need access to the property's *backing field*, which is responsible for actually storing the property data. It is accessed via the special `field` property available inside accessor body, which follows these conventions

- For a property declaration of type `T`, `field` has the same type `T`
- `field` is read-only inside getter body
- `field` is mutable inside setter body

However, the backing field is created for a property only in the following cases

- A property has no custom accessors;
- A property has a default accessor;
- A property has a custom accessor, and it uses `field` property;
- A mutable property has a custom getter or setter, but not both.

In all other cases a property has no backing field. Properties without backing fields are not allowed to have initializer expressions.

Read/write access to the property is replaced with getter/setter invocation respectively. Getters and setters allow for some modifiers available for function declarations (for example, they may be declared `inline`, see grammar for details).

Properties themselves may also be declared `inline`, meaning that both getter and setter of said property are `inline`. Additionally, `inline` properties are not allowed to have backing fields, i.e., they must have custom accessors which do not use the `field` property.

4.3.5 Delegated property declaration

A delegated read-only property declaration `val x: T by e` introduces `x` as a name for the *delegation* result of property `x` to the entity `e` or to the delegatee of `e` provided by `provideDelegate`. For the former, one may consider these properties as regular properties with a special *delegating* getters:

```
val x: T by e
```

is the same as

```
val x$delegate = e
val x: T
    get(): T = x$delegate.getValue(thisRef, ::x)
```

Here every access to such property (`x` in this case) becomes an overloadable form which is expanded into the following:

```
e.getValue(thisRef, property)
```

where

- `e` is the delegating entity; the compiler needs to make sure that this is accessible in any place `x` is accessible;
- `getValue` is a suitable operator function available on `e`;
- `thisRef` is the receiver object for the property. This argument is `null` for local properties;
- `property` is an object of the type `kotlin.KProperty<*>` that contains information relevant to `x` (for example, its name, see standard library documentation for details).

A delegated mutable property declaration `var x: T by e` introduces `x` as a name of a mutable entity with type `T`, access to which is *delegated* to the entity `e` or to the delegatee of `e` provided by `provideDelegate`. As before, one may

view these properties as regular properties with special *delegating* getters and setters:

```
var x: T by e
```

is the same as

```
val x$delegate = e
var x: T
    get(): T = x$delegate.getValue(thisRef, ::x)
    set(value: T) { x$delegate.setValue(thisRef, ::x, value) }
```

Read access is handled the same way as for a delegated read-only property. Any write access to `x` (using, for example, an assignment operator `x = y`) becomes an overloadable form with the following expansion:

```
e.setValue(thisRef, property, y)
```

where

- `e` is the delegating entity; the compiler needs to make sure that this is accessible in any place `x` is accessible;
- `setValue` is a suitable operator function available on `e`;
- `thisRef` is the receiver object for the property. This argument is `null` for local properties;
- `property` is an object of the type `kotlin.KProperty<*>` that contains information relevant to `x` (for example, its name, see standard library documentation for details);
- `y` is the value `x` is assigned to. In case of complex assignments (see the assignment section), as they are all overloadable forms, first the assignment expansion is performed, and after that, the expansion of the delegated property using normal assignment.

The type of a delegated property may be omitted at the declaration site, meaning that it may be inferred from the delegating function itself, as it is with regular getters and setters. If this type is omitted, it is inferred as if it was assigned the value of its expansion. If this inference fails, it is a compile-time error.

If the delegate expression has a suitable operator function called `provideDelegate`, a *provided* delegate is used instead. The provided delegate is accessed using the following expansion:

```
val x: T by e
```

is the same as

```
val x$delegate = e.provideDelegate(thisRef, ::x)
val x: T
    get(): T = x$delegate.getValue(thisRef, ::x)
```

and

```
var x: T by e
```

is the same as

```
val x$delegate = e.provideDelegate(thisRef, ::x)
val x: T
    get(): T = x$delegate.getValue(thisRef, ::x)
    set(value) { x$delegate.setValue(thisRef, ::x, value) }
```

where `provideDelegate` is a suitable operator function available using the receiver `e`, while `getValue` and `setValue` work the same way they do with normal property delegation. As is the case with`setValue` and `getValue`, `thisRef` is a reference to the receiver of the property or `null` for local properties, but there is also a special case: for extension properties `thisRef` supplied to `provideDelegate` is `null`, while `thisRef` provided to `getValue` and `setValue` is the actual receiver. This is due to the fact that, during the creation of the property, no receiver is available.

For both provided and standard delegates, the generated delegate value is placed in the same context as its corresponding property. This means that for a class member property it will be a synthetic member, for a local property it is a local value in the same scope as the property and for top-level (both extension and non-extension) properties it will be a top-level value. This affects this value's lifetime in the same way normal value lifetime works.

Example:

```
operator fun <V, R : V> Map<in String, V>.getValue(
        thisRef: Any?, property: KProperty<*>): R =
    getOrElse(property.name) {
        throw NoSuchElementException()
    } as R

operator fun <V> MutableMap<in String, V>.setValue(
        thisRef: Any?, property: KProperty<*>, newValue: V) =
    set(property.name, newValue)

fun handleConfig(config: MutableMap<String, Any?>) {
    val parent by config       // Any?
    val host: String by config // String
    var port: Int by config    // Int

    // Delegating property accesses to Map.getValue
    // Throwing NSEE as there is no "port" key in the map
    // println("$parent: going to $host:$port")

    // Delegating property access to Map.setValue
    port = 443
    // Map now contains "port" key
```

```
    // Delegating property accesses to Map.getValue
    // Not throwing NSEE as there is "port" key in the map
    println("$parent: going to $host:$port")
}

fun main() {
    handleConfig(mutableMapOf(
        "parent" to "",
        "host" to "https://kotlinlang.org/"
    ))
}
```

Example with `provideDelegate`:

```
operator fun <V> MutableMap<in String, V>.provideDelegate(
        thisRef: Any?,
        property: KProperty<*>): MutableMap<in String, V> =
    if (containsKey(property.name)) this
    else throw NoSuchElementException()

operator fun <V, R : V> Map<in String, V>.getValue(
        thisRef: Any?, property: KProperty<*>): R = ...

operator fun <V> MutableMap<in String, V>.setValue(
        thisRef: Any?, property: KProperty<*>, newValue: V) = ...

fun handleConfig(config: MutableMap<String, Any?>) {
    val parent by config       // Any?
    val host: String by config // String
    var port: Int by config    // Int
    // Throwing NSEE here as `provideDelegate`
    //   checks for "port" key in the map

    ...
}

fun main() {
    handleConfig(mutableMapOf(
        "parent" to "",
        "host" to "https://kotlinlang.org/"
    ))
}
```

4.3.6 Extension property declaration

An *extension property declaration* is similar to a standard property declaration, but, very much alike an extension function, introduces an additional parameter to

the property called *the receiver parameter*. This is different from usual property declarations, that do not have any parameters. There are other differences from standard property declarations:

- Extension properties cannot have initializers;
- Extension properties cannot have backing fields;
- Extension properties cannot have default accessors.

> Note: informally, on can say that extension properties have no state of their own. Only properties that use other objects' storage facilities and/or uses constant data can be extension properties.

Aside from these differences, extension properties are similar to regular properties, but, when accessing such a property one always need to supply a *receiver*, implicit or explicit. Like for regular properties, the type of the receiver must be a subtype of the receiver parameter, and the value that is supplied as the receiver is bound to the receiver parameter. For more information on how a particular receiver for each access is chosen, please refer to the overloading section.

The receiver parameter can be accessed inside getter and setter scopes of the property as the implicit receiver or `this`. It may also be accessed inside nested scopes using labeled `this` syntax using the name of the property declared as the label. For delegated properties, the value passed into the operator functions `getValue` and `setValue` as the receiver is the value of the receiver parameter, rather than the value of the outer classifier. This is also true for local extension properties: while regular local properties are passed `null` as the first argument of these operator functions, local extension properties are passed the value of the receiver argument instead.

> Note: when declaring extension properties inside classifier declarations, this receiver takes precedence over the classifier object, which is usually the current receiver inside nested properties

For all other purposes, extension properties are not different from non-extension properties.

> Examples:

```
val Int.foo: Int get() = this + 1

fun main(args: Array<String>) {
    println(2.foo.foo) // prints "4"
}

class Bar {
    val foo get() = this // returns type Bar
    val Int.foo get() = this // returns type Int
}
```

4.3.7 Property initialization

All non-abstract properties must be definitely initialized before their first use. To guarantee this, Kotlin compiler uses a number of analyses which are described in more detail here.

4.3.8 Constant properties

A property may be declared **constant**, meaning that its value is known during compilation, by using the special `const` modifier. In order to be declared `const`, a property must meet the following requirements:

- Its type is one of the following:
  - One of the the built-in integral types;
  - One of the the built-in floating types;
  - `kotlin.Boolean`;
  - `kotlin.Char`;
  - `kotlin.String`;
- It is declared in the top-level scope or inside an object declaration;
- It has an initializer expression and this initializer expression can be evaluated at compile-time. Integer literals and string interpolation expressions without evaluated expressions, as well as built-in arithmetic/comparison operations and string concatenation operations on those are such expressions, as well as other constant properties, but it is implementation-defined which other expressions qualify for this;
- It does not have getters, setters or delegation specifiers.

  Example:

```
// Correct constant properties
const val answer = 2 * 21
const val msg = "Hello World!"
const val calculated = answer + 45

// Incorrect constant property
const val emptyStringHashCode = "".hashCode()
```

4.3.9 Late-initialized properties

A mutable member property can be declared with a special `lateinit` modifier, effectively turning off the property initialization checks for it. Such a property is called late-initialized and may be used for values that are supposed to be initialized not during object construction, but during some other time (for example, a special initialization function). This means, among other things, that it is the responsibility of the programmer to guarantee that the property is initialized before its usage.

A property may be declared late-initialized if:

- It has no custom getters, setters or delegation;
- It is a member or a top-level property;
- It is mutable;
- It has declared non-nullable type which is also not one of the following types:
  - One of the built-in integer types;
  - One of the built-in floating types;
  - `kotlin.Boolean`;
  - `kotlin.Char`.

4.3.10 Property declaration scopes

Every property getter and setter introduce the same function parameter scope and function body scope as a corresponding function would, see function declaration scopes for details. Getter and setter parameter scopes are upward-linked to the scope property is declared in. Property itself introduces a new binding in the scope it is declared in.

Initialization expressions and delegate expressions for properties, however, are special. If the property declaration resides in a classifier body scope, its initialization expression or delegate expression is resolved in the initialization scope of the same classifier. If the property declaration is local or top-level, its initialization expression or delegate expression is resolved in the scope the property is declared in.

4.4 Type alias

***typeAlias*:**
[*modifiers*]
`'typealias'`
{*NL*}
simpleIdentifier
[{*NL*} *typeParameters*]
{*NL*}
`'='`
{*NL*}
type

Type alias introduces an alternative name for the specified type and supports both simple and parameterized types. If type alias is parameterized, its type parameters must be unbounded and cannot specify variance. Bounds and variance of these parameters is always defined to be the same as the corresponding parameters in the type being aliased, unless they are not referenced in the aliased type, in which case they are considered unbounded and invariant. Another restriction is that recursive type aliases are forbidden — the type alias name cannot be used in its own right-hand side.

At the moment, Kotlin supports only top-level type aliases. The scope where it is accessible is defined by its *visibility modifiers*.

Examples:

```
// simple typealias declaration
typealias IntList = List<Int>
// parameterized type alias declaration
// T has out variance implicitly
typealias IntMap<T> = Map<Int, T>
// type parameter may be unreferenced
typealias Strange<T> = String
```

4.5 Declarations with type parameters

Most declarations may be introduced as *generic*, introducing type parameters that must be explicitly specified or inferred when the corresponding declaration is used. For declarations that introduce new types this mechanism provides the means of introducing a parameterized type. Please refer to the corresponding section for details.

Type parameters may be used as types inside the scope introduced by the declaration. When such a declaration is used, the parameters are substituted by types available inside the scope the declaration is used in.

The following declarations are not allowed to have type parameters:

- Non-extension property declarations;
- Object declarations (including companion object declarations);
- Constructor declarations;
- Getters and setters of property declarations;
- Enum class declarations;
- Classifier declarations inheriting from `kotlin.Throwable`.

Type parameters are allowed to specify *subtyping restrictions* on them in the form `T : U`, meaning $T <: U$ where T is a type parameter and U is some other type available in the scope the declaration is declared in. These either are written directly at the parameter placement syntax or using a special `where` syntax. Any number of restrictions is allowed on a single type, however, for a given type parameter `T`, only one restriction `T : U` can have `U` to be another type parameter.

These restrictions are turned into corresponding type constraints when the type parameters are substituted with types and are employed during type inference and overload resolution of any usage of the corresponding declaration. See the corresponding sections for details.

Type parameters do not introduce runtime-available types unless declared `reified`. Only type parameters of inline functions can be declared `reified`.

4.5.1 Type parameter variance

The declaration-site variance of a particular type parameter for a classifier declaration is specified using special keywords `in` (for covariant parameters) and `out` (for contravariant parameters). If the variance is not specified, the parameter is implicitly declared invariant. See the type system section for details.

A type parameter is **used in covariant position** in the following cases:

- It is used as an argument in another generic type and the corresponding parameter in that type is covariant;
- It is the return type of a function;
- It is a type of a property.

A type parameter is **used in contravariant position** in the following cases:

- It is used as an argument in another generic type and the corresponding parameter in that type is contravariant;
- It is a type of an parameter of a function;
- It is a type of a mutable property.

A type parameter is used in an invariant position if it is used as an argument in another generic type and the corresponding parameter in that type is invariant.

A usage of a contravariant type parameter in a covariant or invariant position, as well as usage of a covariant type parameter in a contravariant or invariant position, results in **variance conflict** and a compiler error, unless the containing declaration is private to the type parameter owner (in which case its visibility is restricted, see the visibility section for details). This applies only to member declarations of the corresponding class, extensions are not subject to this limitation.

This restrictions may be lifted in particular cases by annotating the corresponding type parameter usage with a special built-in annotation `kotlin.UnsafeVariance`. By supplying this annotation the author of the code explicitly declares that safety features that variance checks provide are not needed in this particular declarations.

Examples:

```
class Inv<T> {
    fun a(): T {...} // Ok, covariant usage
    fun b(value: T) {...} // Ok, contravariant usage
    fun c(p: Out<T>) {...} // Ok, covariant usage
    fun d(): Out<T> {...} // Ok, covariant usage
    fun e(p: In<T>) {...} // Ok, contravariant usage
    fun f(): In<T> {...} // Ok, contravariant usage
}

class Out<out T> { // T is covariant
```

```
    fun a(): T {...} // Ok, covariant usage
    fun b(value: T) {...} // ERROR, contravariant usage
    fun c(p: Inv<T>) {...} // ERROR, invariant usage
    fun d(): Inv<T> {...} // ERROR, invariant usage
}

class In<in T> { // T is contravariant
    fun a(): T {...} // ERROR, covariant usage
    fun b(value: T) {...} // Ok, contravariant usage
    fun c(p: Inv<T>) {...} // ERROR, invariant usage
    fun d(): Inv<T> {...} // ERROR, invariant usage
}
```

Any of these restrictions may be lifted using `@UnsafeVariance` annotation on the type argument:

```
class Out<out T> { // T is covariant
    fun b(value: @UnsafeVariance T) {...} // Ok
}

class In<in T> { // T is contravariant
    fun a(): @UnsafeVariance T {...} // Ok
}
```

Using `@UnsafeVariance` is inherently unsafe and should be used only when the programmer can guarantee that variance violations would not result in runtime errors. For example, receiving a value in a contravariant position for a covariant class parameter is usually OK if the function involved is guaranteed not to mutate internal state of the class.

For examples on how restrictions are lifted for private visibility (private-to-this), see visibility section

4.5.2 Reified type parameters

Type parameters of inline function or property declarations (and only those) can be declared `reified` using the corresponding keyword. A reified type parameter is a runtime-available type inside their declaration's scope, see the corresponding section for details. Reified type parameters can only be substituted by other runtime-available types when using such declarations.

Example:

```
fun <T> foo(value: Any?) {
    // ERROR, is-operator is only allowed for runtime-available types
    if(value is T) ...
}
```

```
inline fun <reified T> foo(value: Any?) {
    if(value is T) ... // Ok
}
```

4.5.3 Underscore type arguments

In case one needs to explicitly specify some type parameters via type arguments, but wants to use type inference for the rest, they can use an *underscore type argument*.

An underscore type argument does not add any type information to the constraint system *besides the presence of a type parameter*, i.e., parameterized declaration with different number of type parameters could be distinguished by different number of underscore type arguments.

If the type inference is successful, each underscore type argument is considered to be equal to the inferred type for their respective type parameter. If the type inference is not successful, it is a compile-time error.

Example:

```
fun <T> mk(): T = TODO()

interface MyRunnable<T> {
    fun execute(): T
}

class StringRunnable : MyRunnable<String> {
    override fun execute(): String = "test"
}

class IntRunnable : MyRunnable<Int> {
    override fun execute(): Int = 42
}

inline fun <reified S : MyRunnable<T>, T> run(): T = mk<S>().execute()

fun main() {
    val s = run<StringRunnable, _ /* inferred to String */>()
    assert(s == "test")

    val i = run<IntRunnable, _ /* inferred to Int */>()
    assert(i == 42)
}
```

Example:

```
fun <T> foo(t: T): T = TODO() // (1)
fun <T, R : List<T>> foo(t: T, d: Double = 42.0): T = TODO() // (2)
```

```
fun bar() {
    val a: Boolean = foo(true)
    // resolves to (1)
    // per overload resolution rules

    val b: Int = foo<_ /* U1 */>(42)
    // resolves to (1)
    // with U1 inferred to Int

    val c: Double = foo<_ /* U1 */, _ /* U2 */>(42.0)
    // resolves to (2)
    // with U1 inferred to Double and U2 inferred to List<Double>
}
```

4.6 Declaration visibility

Each declaration has a visibility property relative to the scope it is declared in. By default, all the declarations are `public`, meaning that they can be accessed from any other scope their outer scope can be accessed from. The only exception to this rule are overriding declarations that by default inherit the visibility from the declaration they override. Declarations may be also marked `public` explicitly.

Declarations marked as `private` can only be accessed from the same scope they are declared in. For example, all `private` top-level declarations in a file may only be accessed by code from the same file.

Some `private` declarations are special in that they have an even more restricted visibility, called "`private` to `this`". These include declarations that are allowed to lift certain variance rules in their types as long as they are never accessed outside `this` object, meaning that they can be accessed using `this` as the receiver, but are not visible on other instances of the same class even in the methods of this class. For example, for a class declaration C with type parameter T it is not allowed to introduce declarations involving T with conflicting variance, unless they are declared `private`. That is, if T is declared as covariant, any declarations with a type using T in a contravariant position (including properties with type T itself if they are mutable) and if T is declared as contravariant, any declarations with a type using T in a covariant position (including properties with type T itself) are forbidden, unless they are declared using `private` visibility, in which case they are instead treated as "`private` to `this`".

Example:

```
class Foo<out T>(val t: T) { // T is a covariant parameter
    // not allowed, T is in contravariant position
    public fun set1(t: T) {}
```

```
    // allowed, set2 is private-to-this
    private fun set2(t: T) {}
    private fun bar(other: Foo<T>) {
        // allowed, set2 is called on this
        this.set2(t)
        // not allowed, set2 is called on other
        other.set2(t)
    }
}
```

Note: the above does not account for `@UnsafeVariance` annotation that lifts any variance restrictions on type parameters

Declarations marked as `internal` may only be accessed from the same module, treated as `public` from inside the module and as `private` from outside the module.

Declarations in classifier declaration scope can also be declared `protected`, meaning that they can only be accessed from the same classifier type as well as any types inheriting from this type regardless of the scope they are declared in.

There is a partial order of *weakness* between different visibility modifiers:

- `protected` and `internal` are weaker than `private`;
- `public` is weaker than `protected` and `internal`.

Note: there is a certain restriction regarding `inline` functions that have a different visibility from entities they access. In particular, an `inline` function cannot access entities with a stronger visibility (i.e. `public inline` function accessing a `private` property). There is one exception to this: a `public inline` function can access `internal` entities which are marked with a special builtin annotation `@PublishedApi`.

Example:

```
class Foo<T>(internal val t: T) {
    // not allowed, t is internal, getValue is public
    inline fun getValue(): T = t
}
class Bar<T>(@PublishedApi internal val t: T) {
    // allowed through @PublishedApi
    inline fun getValue(): T = t
}
```