# Kotlin style guide

This document serves as the complete definition of Google's Android coding standards for source code in the Kotlin Programming Language. A Kotlin source file is described as being in Google Android Style if and only if it adheres to the rules herein.

Like other programming style guides, the issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, this document focuses primarily on the hard-and-fast rules that we follow universally, and avoids giving advice that isn't clearly enforceable (whether by human or tool).

Last update: 2023-09-06 (/kotlin/guides-changelog)

## Source files

All source files must be encoded as UTF-8.

### Naming

If a source file contains only a single top-level class, the file name should reflect the case-sensitive name plus the `.kt` extension. Otherwise, if a source file contains multiple top-level declarations, choose a name that describes the contents of the file, apply PascalCase (camelCase is acceptable if the filename is plural), and append the `.kt` extension.

```
// MyClass.kt
class MyClass { }
```

```
// Bar.kt
class Bar { }
fun Runnable.toBar(): Bar = // …
```

```
// Map.kt
fun <T, O> Set<T>.map(func: (T) -> O): List<O> = // …
fun <T, O> List<T>.map(func: (T) -> O): List<O> = // …
```

```
// extensions.kt
fun MyClass.process() = // …
fun MyResult.print() = // …
```

## Special Characters

### Whitespace characters

Aside from the line terminator sequence, the **ASCII horizontal space character (0x20)** is the only whitespace character that appears anywhere in a source file. This implies that:

- All other whitespace characters in string and character literals are escaped.
- Tab characters are *not* used for indentation.

### Special escape sequences

For any character that has a special escape sequence (`\b`, `\n`, `\r`, `\t`, `\'`, `\"`, `\\`, and `\$`), that sequence is used rather than the corresponding Unicode (e.g., `\u000a`) escape.

### Non-ASCII characters

For the remaining non-ASCII characters, either the actual Unicode character (e.g., `∞`) or the equivalent Unicode escape (e.g., `\u221e`) is used. The choice depends only on which makes the code **easier to read and understand.** Unicode escapes are discouraged for printable characters at any location and are strongly discouraged outside of string literals and comments.

| Example | Discussion |
| --- | --- |
| `val unitAbbrev = "µs"` | Best: perfectly clear even without a comment. |

| Example | Discussion |
| --- | --- |
| `val unitAbbrev = "\u03bcs" / μs` | Poor: there's no reason to use an escape with a printable character. |
| `val unitAbbrev = "\u03bcs"` | Poor: the reader has no idea what this is. |
| `return "\ufeff" + content` | Good: use escapes for non-printable characters, and comment if necessary. |

## Structure

A `.kt` file comprises the following, in order:

- Copyright and/or license header (optional)
- File-level annotations
- Package statement
- Import statements
- Top-level declarations

Exactly one blank line separates each of these sections.

### Copyright / License

If a copyright or license header belongs in the file it should be placed at the immediate top in a multi-line comment.

```
/*
 * Copyright 2017 Google, Inc.
 *
 * ...
 */
```

Do not use a KDoc-style (https://kotlinlang.org/docs/reference/kotlin-doc.html) or single-line-style comment.

```
/**
 * Copyright 2017 Google, Inc.
 *
 * ...
 */
```

```
// Copyright 2017 Google, Inc.
//
// ...
```

### File-level annotations

Annotations with the "file" use-site target (https://kotlinlang.org/docs/reference/annotations.html#annotation-use-site-targets) are placed between any header comment and the package declaration.

### Package statement

The package statement is not subject to any column limit and is never line-wrapped.

### Import statements

Import statements for classes, functions, and properties are grouped together in a single list and ASCII sorted.

Wildcard imports (of any type) are **not allowed.**

Similar to the package statement, import statements are not subject to a column limit and they are never line-wrapped.

### Top-level declarations

A `.kt` file can declare one or more types, functions, properties, or type aliases at the top-level.

The contents of a file should be focused on a single theme. Examples of this would be a single public type or a set of extension functions performing the same operation on multiple receiver types. Unrelated declarations should be separated into their own files and public declarations within a single file should be minimized.

No explicit restriction is placed on the number nor order of the contents of a file.

Source files are usually read from top-to-bottom meaning that the order, in general, should reflect that the declarations higher up will inform understanding of those farther down. Different files may choose to order their contents differently. Similarly, one file may contain 100 properties, another 10 functions, and yet another a single class.

What is important is that each file uses **some** logical order, which its maintainer could explain if asked. For example, new functions are not just habitually added to the end of the file, as that would yield "chronological by date added" ordering, which is not a logical ordering.

### Class member ordering

The order of members within a class follow the same rules as the top-level declarations.

## Formatting

### Braces

Braces are not required for `when` branches and `if` expressions which have no more than one `else` branch and which fit on a single line.

```
if (string.isEmpty()) return

val result =
    if (string.isEmpty()) DEFAULT_VALUE else string

when (value) {
    0 -> return
    // …
}
```

Braces are otherwise required for any `if`, `for`, `when` branch, `do`, and `while` statements and expressions, even when the body is empty or contains only a single statement.

```
if (string.isEmpty())
    return  // WRONG!

if (string.isEmpty()) {
    return  // Okay
}

if (string.isEmpty()) return  // WRONG
else doLotsOfProcessingOn(string, otherParametersHere)

if (string.isEmpty()) {
    return  // Okay
} else {
    doLotsOfProcessingOn(string, otherParametersHere)
}
```

### Non-empty blocks

Braces follow the Kernighan and Ritchie style ("Egyptian brackets") for nonempty blocks and block-like constructs:

- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.
- Line break after the closing brace, *only if* that brace terminates a statement or terminates the body of a function, constructor, or *named* class. For example, there is *no* line break after the brace if it is followed by `else` or a comma.

```
return Runnable {
    while (condition()) {
        foo()
    }
}

return object : MyClass() {
    override fun foo() {
```

```
        if (condition()) {
            try {
                something()
            } catch (e: ProblemException) {
                recover()
            }
        } else if (otherCondition()) {
            somethingElse()
        } else {
            lastThing()
        }
    }
}
```

A few exceptions for enum classes (#enum-classes) are given below.

### Empty blocks

An empty block or block-like construct must be in K&R style.

```
try {
    doSomething()
} catch (e: Exception) {} // WRONG!
```

```
try {
    doSomething()
} catch (e: Exception) {
} // Okay
```

### Expressions

An `if/else` conditional that is used as an expression may omit braces *only* if the entire expression fits on one line.

```
val value = if (string.isEmpty()) 0 else 1  // Okay
```

```
val value = if (string.isEmpty())  // WRONG!
    0
else
    1
```

```
val value = if (string.isEmpty()) { // Okay
    0
} else {
    1
}
```

### Indentation

Each time a new block or block-like construct is opened, the indent increases by four spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block.

### One statement per line

Each statement is followed by a line break. Semicolons are not used.

### Line wrapping

Code has a column limit of 100 characters. Except as noted below, any line that would exceed this limit must be line-wrapped, as explained below.

Exceptions:

- Lines where obeying the column limit is not possible (for example, a long URL in KDoc)
- `package` and `import` statements
- Command lines in a comment that may be cut-and-pasted into a shell

### Where to break

The prime directive of line-wrapping is: prefer to break at a higher syntactic level. Also:

- When a line is broken at an operator or infix function name, the break comes after the operator or infix function name.
- When a line is broken at the following "operator-like" symbols, the break comes before the symbol:
  - The dot separator (`.`, `?.`).
  - The two colons of a member reference (`::`).
- A method or constructor name stays attached to the open parenthesis (`(`) that follows it.
- A comma (`,`) stays attached to the token that precedes it.
- A lambda arrow (`->`) stays attached to the argument list that precedes it.

**Note:** The primary goal for line wrapping is to have clear code, not necessarily code that fits in the smallest number of lines.

## Functions

When a function signature does not fit on a single line, break each parameter declaration onto its own line. Parameters defined in this format should use a single indent (+4). The closing parenthesis (`)`) and return type are placed on their own line with no additional indent.

```
fun <T> Iterable<T>.joinToString(
    separator: CharSequence = ", ",
    prefix: CharSequence = "",
    postfix: CharSequence = ""
): String {
    // …
}
```

### Expression functions

When a function contains only a single expression it can be represented as an expression function (https://kotlinlang.org/docs/reference/functions.html#single-expression-functions).

```
override fun toString(): String {
    return "Hey"
}
```

```
override fun toString(): String = "Hey"
```

### Properties

When a property initializer does not fit on a single line, break after the equals sign (`=`) and use an indent.

```
private val defaultCharset: Charset? =
    EncodingRegistry.getInstance().getDefaultCharsetForPropertiesFiles(file)
```

Properties declaring a `get` and/or `set` function should place each on their own line with a normal indent (+4). Format them using the same rules as functions.

```
var directory: File? = null
    set(value) {
        // …
    }
```

Read-only properties can use a shorter syntax which fits on a single line.

```
val defaultExtension: String get() = "kt"
```

## Whitespace

### Vertical

A single blank line appears:

- *Between* consecutive members of a class: properties, constructors, functions, nested classes, etc.
    - **Exception:** A blank line between two consecutive properties (having no other code between them) is optional. Such blank lines are used as needed to create logical groupings of properties and associate properties with their backing property, if present.
    - **Exception:** Blank lines between enum constants are covered below.
- Between statements, *as needed* to organize the code into logical subsections.
- *Optionally* before the first statement in a function, before the first member of a class, or after the last member of a class (neither encouraged nor discouraged).
- As required by other sections of this document (such as the Structure (#structure) section).

Multiple consecutive blank lines are permitted, but not encouraged or ever required.

### Horizontal

Beyond where required by the language or other style rules, and apart from literals, comments, and KDoc, a single ASCII space also appears in the following places only:

- Separating any reserved word, such as `if`, `for`, or `catch` from an open parenthesis (`(`) that follows it on that line.

```
// WRONG!
for(i in 0..1) {
}
```

```
// Okay
for (i in 0..1) {
}
```

- Separating any reserved word, such as `else` or `catch`, from a closing curly brace (`}`) that precedes it on that line.

```
// WRONG!
}else {
}
```

```
// Okay
} else {
}
```

- Before any open curly brace (`{`).

```
// WRONG!
if (list.isEmpty()){
}
```

```
// Okay
if (list.isEmpty()) {
}
```

- On both sides of any binary operator.

```
// WRONG!
val two = 1+1
```

```
// Okay
val two = 1 + 1
```

This also applies to the following “operator-like” symbols:

- the arrow in a lambda expression (`->`).

```
// WRONG!
ints.map { value->value.toString() }
```

```
// Okay
ints.map { value -> value.toString() }
```

But not:

- the two colons (`::`) of a member reference.

```
// WRONG!
val toString = Any :: toString
```

```
// Okay
val toString = Any::toString
```

- the dot separator (`.`).

```
// WRONG
it . toString()
```

```
// Okay
it.toString()
```

- the range operator (`..`).

```
// WRONG
for (i in 1 .. 4) {
  print(i)
}
```

```
// Okay
for (i in 1..4) {
  print(i)
}
```

- Before a colon (`:`) only if used in a class declaration for specifying a base class or interfaces, or when used in a `where` clause for generic constraints (https://kotlinlang.org/docs/reference/generics.html#generic-constraints).

```
// WRONG!
class Foo: Runnable
```

```
// Okay
class Foo : Runnable
```

```
// WRONG
fun <T: Comparable> max(a: T, b: T)
```

```
// Okay
fun <T : Comparable> max(a: T, b: T)
```

```
// WRONG
fun <T> max(a: T, b: T) where T: Comparable<T>
```

```
// Okay
fun <T> max(a: T, b: T) where T : Comparable<T>
```

- After a comma (`,`) or colon (`:`).

```
// WRONG!
val oneAndTwo = listOf(1,2)
```

```
// Okay
val oneAndTwo = listOf(1, 2)
```

```
// WRONG!
class Foo :Runnable
```

```
// Okay
class Foo : Runnable
```

- On both sides of the double slash (`//`) that begins an end-of-line comment. Here, multiple spaces are allowed, but not required.

```
// WRONG!
var debugging = false//disabled by default
```

```
// Okay
var debugging = false // disabled by default
```

This rule is never interpreted as requiring or forbidding additional space at the start or end of a line; it addresses only interior space.

## Specific constructs

### Enum classes

An enum with no functions and no documentation on its constants may optionally be formatted as a single line.

```
enum class Answer { YES, NO, MAYBE }
```

When the constants in an enum are placed on separate lines, a blank line is not required between them except in the case where they define a body.

```
enum class Answer {
    YES,
    NO,

    MAYBE {
        override fun toString() = """¯\_(ツ)_/¯"""
    }
}
```

Since enum classes are classes, all other rules for formatting classes apply.

### Annotations

Member or type annotations are placed on separate lines immediately prior to the annotated construct.

```
@Retention(SOURCE)
@Target(FUNCTION, PROPERTY_SETTER, FIELD)
annotation class Global
```

Annotations without arguments can be placed on a single line.

```
@JvmField @Volatile
var disposable: Disposable? = null
```

When only a single annotation without arguments is present, it may be placed on the same line as the declaration.

```
@Volatile var disposable: Disposable? = null

@Test fun selectAll() {
    // …
}
```

`@[...]` syntax may only be used with an explicit use-site target, and only for combining 2 or more annotations without arguments on a single line.

```
@field:[JvmStatic Volatile]
var disposable: Disposable? = null
```

### Implicit return/property types

If an expression function body or a property initializer is a scalar value or the return type can be clearly inferred from the body then it can be omitted.

```
override fun toString(): String = "Hey"
// becomes
```

```
override fun toString() = "Hey"
```

```
private val ICON: Icon = IconLoader.getIcon("/icons/kotlin.png")
// becomes
private val ICON = IconLoader.getIcon("/icons/kotlin.png")
```

When writing a library, retain the explicit type declaration when it is part of the public API.

## Naming

Identifiers use only ASCII letters and digits, and, in a small number of cases noted below, underscores. Thus each valid identifier name is matched by the regular expression `\w+`.

Special prefixes or suffixes, like those seen in the examples `name_`, `mName`, `s_name`, and `kName`, are not used except in the case of backing properties (see Backing properties (#backing-properties)).

### Package Names

Package names are all lowercase, with consecutive words simply concatenated together (no underscores).

```
// Okay
package com.example.deepspace
// WRONG!
package com.example.deepSpace
// WRONG!
package com.example.deep_space
```

### Type names

Class names are written in PascalCase and are typically nouns or noun phrases. For example, `Character` or `ImmutableList`. Interface names may also be nouns or noun phrases (for example, `List`), but may sometimes be adjectives or adjective phrases instead (for example `Readable`).

Test classes are named starting with the name of the class they are testing, and ending with `Test`. For example, `HashTest` or `HashIntegrationTest`.

### Function names

Function names are written in camelCase and are typically verbs or verb phrases. For example, `sendMessage` or `stop`.

Underscores are permitted to appear in test function names to separate logical components of the name.

```
@Test fun pop_emptyStack() {
    // …
}
```

Functions annotated with `@Composable` that return `Unit` are PascalCased and named as nouns, as if they were types.

```
@Composable
fun NameTag(name: String) {
    // …
}
```

Function names should not contain spaces because this is not supported on every platform (notably, this is not fully supported in Android).

```
// WRONG!
fun `test every possible case`() {}
// OK
fun testEveryPossibleCase() {}
```

### Constant names

Constant names use UPPER_SNAKE_CASE: all uppercase letters, with words separated by underscores. But what *is* a constant, exactly?

Constants are `val` properties with no custom `get` function, whose contents are deeply immutable, and whose functions have no detectable side-effects. This includes immutable types and immutable collections of immutable types as well as scalars and string if marked as `const`. If any of an instance's observable state can change, it is not a constant. Merely intending to never mutate the object is not enough.

```
const val NUMBER = 5
val NAMES = listOf("Alice", "Bob")
val AGES = mapOf("Alice" to 35, "Bob" to 32)
val COMMA_JOINER = Joiner.on(',') // Joiner is immutable
val EMPTY_ARRAY = arrayOf
```

These names are typically nouns or noun phrases.

Constant values can only be defined inside of an `object` or as a top-level declaration. Values otherwise meeting the requirement of a constant but defined inside of a `class` must use a non-constant name.

Constants which are scalar values must use the `const` modifier (http://kotlinlang.org/docs/reference/properties.html#compile-time-constants).

### Non-constant names

Non-constant names are written in camelCase. These apply to instance properties, local properties, and parameter names.

```
val variable = "var"
val nonConstScalar = "non-const"
val mutableCollection: MutableSet
```

These names are typically nouns or noun phrases.

#### Backing properties

When a backing property (https://kotlinlang.org/docs/reference/properties.html#backing-properties) is needed, its name should exactly match that of the real property except prefixed with an underscore.

```
private var _table: Map
```

### Type variable names

Each type variable is named in one of two styles:

- A single capital letter, optionally followed by a single numeral (such as `E`, `T`, `X`, `T2`)
- A name in the form used for classes, followed by the capital letter `T` (such as `RequestT`, `FooBarT`)

### Camel case

Sometimes there is more than one reasonable way to convert an English phrase into camel case, such as when acronyms or unusual constructs like “IPv6” or “iOS” are present. To improve predictability, use the following scheme.

Beginning with the prose form of the name:

1. Convert the phrase to plain ASCII and remove any apostrophes. For example, “Müller’s algorithm” might become “Muellers algorithm”.
2. Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens). *Recommended:* if any word already has a conventional camel-case appearance in common usage, split this into its constituent parts (e.g., “AdWords” becomes “ad words”). Note that a word such as “iOS” is not really in camel case per se; it defies any convention, so this recommendation does not apply.
3. Now lowercase everything (including acronyms), then do one of the following:
    - Uppercase the first character of each word to yield pascal case.
    - Uppercase the first character of each word except the first to yield camel case.
4. Finally, join all the words into a single identifier.

Note that the casing of the original words is almost entirely disregarded.

| Prose form | Correct | Incorrect |
| --- | --- | --- |
| "XML Http Request" | `XmlHttpRequest` | `XMLHTTPRequest` |
| "new customer ID" | `newCustomerId` | `newCustomerID` |
| "inner stopwatch" | `innerStopwatch` | `innerStopWatch` |
| "supports IPv6 on iOS" | `supportsIpv6OnIos` | `supportsIPv6OnIOS` |
| "YouTube importer" | `YouTubeImporter` | `YoutubeImporter*` |

(* Acceptable, but not recommended.)

**Note:** Some words are ambiguously hyphenated in the English language: for example “nonempty” and “non-empty” are both correct, so the method names **`checkNonempty`** and **`checkNonEmpty`** are likewise both correct.

## Documentation

### Formatting

The basic formatting of KDoc blocks is seen in this example:

```
/**
 * Multiple lines of KDoc text are written here,
 * wrapped normally…
 */
fun method(arg: String) {
    // …
}
```

...or in this single-line example:

```
/** An especially short bit of KDoc. */
```

The basic form is always acceptable. The single-line form may be substituted when the entirety of the KDoc block (including comment markers) can fit on a single line. Note that this only applies when there are no block tags such as `@return`.

#### Paragraphs

One blank line—that is, a line containing only the aligned leading asterisk (`*`)—appears between paragraphs, and before the group of block tags if present.

#### Block tags

Any of the standard "block tags" that are used appear in the order `@constructor`, `@receiver`, `@param`, `@property`, `@return`, `@throws`, `@see`, and these never appear with an empty description. When a block tag doesn't fit on a single line, continuation lines are indented 4 spaces from the position of the `@`.

### Summary fragment

Each KDoc block begins with a brief summary fragment. This fragment is very important: it is the only part of the text that appears in certain contexts such as class and method indexes.

This is a fragment–a noun phrase or verb phrase, not a complete sentence. It does not begin with "`A `Foo` is a...`", or "`This method returns...`", nor does it have to form a complete imperative sentence like "`Save the record.`". However, the fragment is capitalized and punctuated as if it were a complete sentence.

### Usage

At the minimum, KDoc is present for every `public` type, and every `public` or `protected` member of such a type, with a few exceptions noted below.

#### Exception: Self-explanatory functions

KDoc is optional for “simple, obvious” functions like `getFoo` and properties like `foo`, in cases where there really and truly is nothing else worthwhile to say but “Returns the foo”.

It is not appropriate to cite this exception to justify omitting relevant information that a typical reader might need to know. For example, for a function named `getCanonicalName` or property named `canonicalName`, don’t omit its documentation (with the rationale that it would say only `/** Returns the canonical name. */`) if a typical reader may have no idea what the term "canonical name" means!

### Exception: overrides

KDoc is not always present on a method that overrides a supertype method.