Coding conventions

Commonly known and easy-to-follow coding conventions are vital for any programming language. Here we provide guidelines on the code style and code organization for projects that use Kotlin.

Configure style in IDE

Two most popular IDEs for Kotlin - IntelliJ IDEA and Android Studio provide powerful support for code styling. You can configure them to automatically format your code in consistence with the given code style.

Apply the style guide

1. Go to **Settings/Preferences | Editor | Code Style | Kotlin**.
2. Click **Set from...**.
3. Select **Kotlin style guide**.

Verify that your code follows the style guide

1. Go to **Settings/Preferences | Editor | Inspections | General**.
2. Switch on **Incorrect formatting** inspection. Additional inspections that verify other issues described in the style guide (such as naming conventions) are enabled by default.

For more information, see the Migrate to Kotlin code style with IntelliJ IDEA guide.

Source code organization

Directory structure

In pure Kotlin projects, the recommended directory structure follows the package structure with the common root package omitted. For example, if all the code in the project is in the `org.example.kotlin` package and its subpackages, files with the `org.example.kotlin` package should be placed directly under the source root, and files in `org.example.kotlin.network.socket` should be in the `network/socket` subdirectory of the source root.

> On JVM: In projects where Kotlin is used together with Java, Kotlin source files should reside in the same source root as the Java source files, and follow the same directory structure: each file should be stored in the directory corresponding to each package statement.

Source file names

If a Kotlin file contains a single class or interface (potentially with related top-level declarations), its name should be the same as the name of the class, with the `.kt` extension appended. It applies to all types of classes and interfaces. If a file contains multiple classes, or only top-level declarations, choose a name describing what the file contains, and name the file accordingly. Use upper camel case, where the first letter of each word is capitalized. For example, `ProcessDeclarations.kt`.

The name of the file should describe what the code in the file does. Therefore, you should avoid using meaningless words such as `Util` in file names.

Multiplatform projects

In multiplatform projects, files with top-level declarations in platform-specific source sets should have a suffix associated with the name of the source set. For example:

- **jvm**Main/kotlin/Platform. **jvm** .kt
- **android**Main/kotlin/Platform. **android** .kt
- **ios**Main/kotlin/Platform. **ios** .kt

As for the common source set, files with top-level declarations should not have a suffix. For example, `commonMain/kotlin/Platform.kt`.

Technical details

We recommend following this file naming scheme in multiplatform projects due to JVM limitations: it doesn't allow top-level members (functions, properties).

To work around this, the Kotlin JVM compiler creates wrapper classes (so-called "file facades") that contain top-level member declarations. File facades have an internal name derived from the file name.

In turn, JVM doesn't allow several classes with the same fully qualified name (FQN). This might lead to situations when a Kotlin project cannot be compiled to JVM:

```
root
|- commonMain/kotlin/myPackage/Platform.kt // contains 'fun count() { }'
|- jvmMain/kotlin/myPackage/Platform.kt // contains 'fun multiply() { }'
```

Here both `Platform.kt` files are in the same package, so the Kotlin JVM compiler produces two file facades, both of which have FQN `myPackage.PlatformKt`. This produces the "Duplicate JVM classes" error.

The simplest way to avoid that is renaming one of the files according to the guideline above. This naming scheme helps avoid clashes while retaining code readability.

> There are two scenarios where these recommendations may seem redundant, but we still advise to follow them:
>
> - Non-JVM platforms don't have issues with duplicating file facades. However, this naming scheme can help you keep file naming consistent.
> - On JVM, if source files don't have top-level declarations, the file facades aren't generated, and you won't face naming clashes.
>
>   However, this naming scheme can help you avoid situations when a simple refactoring or an addition could include a top-level function and result in the same "Duplicate JVM classes" error.

Source file organization

Placing multiple declarations (classes, top-level functions or properties) in the same Kotlin source file is encouraged as long as these declarations are closely related to each other semantically, and the file size remains reasonable (not exceeding a few hundred lines).

In particular, when defining extension functions for a class which are relevant for all clients of this class, put them in the same file with the class itself. When defining extension functions that make sense only for a specific client, put them next to the code of that client. Avoid creating files just to hold all extensions of some class.

Class layout

The contents of a class should go in the following order:

1. Property declarations and initializer blocks
2. Secondary constructors
3. Method declarations
4. Companion object

Do not sort the method declarations alphabetically or by visibility, and do not separate regular methods from extension methods. Instead, put related stuff together, so that someone reading the class from top to bottom can follow the logic of what's happening. Choose an order (either higher-level stuff first, or vice versa) and stick to it.

Put nested classes next to the code that uses those classes. If the classes are intended to be used externally and aren't referenced inside the class, put them in the end, after the companion object.

Interface implementation layout

When implementing an interface, keep the implementing members in the same order as members of the interface (if necessary, interspersed with additional private methods used for the implementation).

Overload layout

Always put overloads next to each other in a class.

Naming rules

Package and class naming rules in Kotlin are quite simple:

- Names of packages are always lowercase and do not use underscores (`org.example.project`). Using multi-word names is generally discouraged, but if you do need to use multiple words, you can either just concatenate them together or use camel case (`org.example.myProject`).
- Names of classes and objects use upper camel case:

```kotlin
open class DeclarationProcessor { /*...*/ }

object EmptyDeclarationProcessor : DeclarationProcessor() { /*...*/ }
```

Function names

Names of functions, properties and local variables start with a lowercase letter and use camel case with no underscores:

```kotlin
fun processDeclarations() { /*...*/ }
var declarationCount = 1
```

Exception: factory functions used to create instances of classes can have the same name as the abstract return type:

```kotlin
interface Foo { /*...*/ }

class FooImpl : Foo { /*...*/ }

fun Foo(): Foo { return FooImpl() }
```

Names for test methods

In tests (and **only** in tests), you can use method names with spaces enclosed in backticks. Note that such method names are only supported by Android runtime from API level 30. Underscores in method names are also allowed in test code.

```kotlin
class MyTestCase {
    @Test fun `ensure everything works`() { /*...*/ }

    @Test fun ensureEverythingWorks_onAndroid() { /*...*/ }
}
```

Property names

Names of constants (properties marked with `const`, or top-level or object `val` properties with no custom `get` function that hold deeply immutable data) should use all uppercase, underscore-separated names following the screaming snake case convention:

```kotlin
const val MAX_COUNT = 8
val USER_NAME_FIELD = "UserName"
```

Names of top-level or object properties which hold objects with behavior or mutable data should use camel case names:

```kotlin
val mutableCollection: MutableSet<String> = HashSet()
```

Names of properties holding references to singleton objects can use the same naming style as `object` declarations:

```kotlin
val PersonComparator: Comparator<Person> = /*...*/
```

For enum constants, it's OK to use either all uppercase, underscore-separated (screaming snake case) names (`enum class Color { RED, GREEN }`) or upper camel case names, depending on the usage.

Names for backing properties

If a class has two properties which are conceptually the same but one is part of a public API and another is an implementation detail, use an underscore as the prefix for the name of the private property:

```kotlin
class C {
    private val _elementList = mutableListOf<Element>()

    val elementList: List<Element>
        get() = _elementList
}
```

Choose good names

The name of a class is usually a noun or a noun phrase explaining what the class *is*: `List`, `PersonReader`.

The name of a method is usually a verb or a verb phrase saying what the method *does*: `close`, `readPersons`. The name should also suggest if the method is mutating the object or returning a new one. For instance `sort` is sorting a collection in place, while `sorted` is returning a sorted copy of the collection.

The names should make it clear what the purpose of the entity is, so it's best to avoid using meaningless words (`Manager`, `Wrapper`) in names.

When using an acronym as part of a declaration name, follow these rules:

- For two-letter acronyms, use uppercase for both letters. For example, `IOStream`.
- For acronyms longer than two letters, capitalize only the first letter. For example, `XmlFormatter` or `HttpInputStream`.

Formatting

Indentation

Use four spaces for indentation. Do not use tabs.

For curly braces, put the opening brace at the end of the line where the construct begins, and the closing brace on a separate line aligned horizontally with the opening construct.

```kotlin
if (elements != null) {
    for (element in elements) {
        // ...
    }
}
```

In Kotlin, semicolons are optional, and therefore line breaks are significant. The language design assumes Java-style braces, and you may encounter surprising behavior if you try to use a different formatting style.

Horizontal whitespace

- Put spaces around binary operators (`a + b`). Exception: don't put spaces around the "range to" operator (`0..i`).
- Do not put spaces around unary operators (`a++`).
- Put spaces between control flow keywords (`if`, `when`, `for`, and `while`) and the corresponding opening parenthesis.
- Do not put a space before an opening parenthesis in a primary constructor declaration, method declaration or method call.

```kotlin
class A(val x: Int)

fun foo(x: Int) { ... }

fun bar() {
    foo(1)
}
```

- Never put a space after `(`, `[`, or before `]`, `)`.
- Never put a space around `.` or `?.`: `foo.bar().filter { it > 2 }.joinToString()`, `foo?.bar()`.
- Put a space after `//`: `// This is a comment`.
- Do not put spaces around angle brackets used to specify type parameters: `class Map<K, V> { ... }`.
- Do not put spaces around `::`: `Foo::class`, `String::length`.
- Do not put a space before `?` used to mark a nullable type: `String?`.

As a general rule, avoid horizontal alignment of any kind. Renaming an identifier to a name with a different length should not affect the formatting of either the declaration or any of the usages.

Colon

Put a space before `:` in the following scenarios:

- When it's used to separate a type and a supertype.
- When delegating to a superclass constructor or a different constructor of the same class.
- After the `object` keyword.

Don't put a space before `:` when it separates a declaration and its type.

Always put a space after `:`.

```kotlin
abstract class Foo<out T : Any> : IFoo {
    abstract fun foo(a: Int): T
}

class FooImpl : Foo() {
    constructor(x: String) : this(x) { /*...*/ }

    val x = object : IFoo { /*...*/ }
}
```

Class headers

Classes with a few primary constructor parameters can be written in a single line:

```kotlin
class Person(id: Int, name: String)
```

Classes with longer headers should be formatted so that each primary constructor parameter is in a separate line with indentation. Also, the closing parenthesis should be on a new line. If you use inheritance, the superclass constructor call, or the list of implemented interfaces should be located on the same line as the parenthesis:

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name) { /*...*/ }
```

For multiple interfaces, the superclass constructor call should be located first and then each interface should be located in a different line:

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name),
    KotlinMaker { /*...*/ }
```

For classes with a long supertype list, put a line break after the colon and align all supertype names horizontally:

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne {

    fun foo() { /*...*/ }
}
```

To clearly separate the class header and body when the class header is long, either put a blank line following the class header (as in the example above), or put the opening curly brace on a separate line:

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne
{
    fun foo() { /*...*/ }
}
```

Use regular indent (four spaces) for constructor parameters. This ensures that properties declared in the primary constructor have the same indentation as properties declared in the body of a class.

Modifiers order

If a declaration has multiple modifiers, always put them in the following order:

```
public / protected / private / internal
expect / actual
final / open / abstract / sealed / const
external
override
lateinit
tailrec
vararg
suspend
inner
enum / annotation / fun // as a modifier in `fun interface`
companion
inline / value
infix
operator
data
```

Place all annotations before modifiers:

```
@Named("Foo")
private val foo: Foo
```

Unless you're working on a library, omit redundant modifiers (for example, `public`).

Annotations

Place annotations on separate lines before the declaration to which they are attached, and with the same indentation:

```
@Target(AnnotationTarget.PROPERTY)
annotation class JsonExclude
```

Annotations without arguments may be placed on the same line:

```
@JsonExclude @JvmField
var x: String
```

A single annotation without arguments may be placed on the same line as the corresponding declaration:

```
@Test fun foo() { /*...*/ }
```

File annotations

File annotations are placed after the file comment (if any), before the `package` statement, and are separated from `package` with a blank line (to emphasize the fact that they target the file and not the package).

```
/** License, copyright and whatever */
@file:JvmName("FooBar")

package foo.bar
```

Functions

If the function signature doesn't fit on a single line, use the following syntax:

```kotlin
fun longMethodName(
    argument: ArgumentType = defaultValue,
    argument2: AnotherArgumentType,
): ReturnType {
    // body
}
```

Use regular indent (four spaces) for function parameters. It helps ensure consistency with constructor parameters.

Prefer using an expression body for functions with the body consisting of a single expression.

```kotlin
fun foo(): Int {     // bad
    return 1
}

fun foo() = 1        // good
```

Expression bodies

If the function has an expression body whose first line doesn't fit on the same line as the declaration, put the `=` sign on the first line and indent the expression body by four spaces.

```kotlin
fun f(x: String, y: String, z: String) =
    veryLongFunctionCallWithManyWords(andLongParametersToo(), x, y, z)
```

Properties

For very simple read-only properties, consider one-line formatting:

```kotlin
val isEmpty: Boolean get() = size == 0
```

For more complex properties, always put `get` and `set` keywords on separate lines:

```kotlin
val foo: String
    get() { /*...*/ }
```

For properties with an initializer, if the initializer is long, add a line break after the `=` sign and indent the initializer by four spaces:

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

Control flow statements

If the condition of an `if` or `when` statement is multiline, always use curly braces around the body of the statement. Indent each subsequent line of the condition by four spaces relative to the statement start. Put the closing parentheses of the condition together with the opening curly brace on a separate line:

```kotlin
if (!component.isSyncing &&
    !hasAnyKotlinRuntimeInScope(module)
) {
    return createKotlinNotConfiguredPanel(module)
}
```

This helps align the condition and statement bodies.

Put the `else`, `catch`, `finally` keywords, as well as the `while` keyword of a `do-while` loop, on the same line as the preceding curly brace:

```
if (condition) {
    // body
} else {
    // else part
}

try {
    // body
} finally {
    // cleanup
}
```

In a `when` statement, if a branch is more than a single line, consider separating it from adjacent case blocks with a blank line:

```
private fun parsePropertyValue(propName: String, token: Token) {
    when (token) {
        is Token.ValueToken ->
            callback.visitValue(propName, token.value)

        Token.LBRACE -> { // ...
        }
    }
}
```

Put short branches on the same line as the condition, without braces.

```
when (foo) {
    true -> bar() // good
    false -> { baz() } // bad
}
```

Method calls

In long argument lists, put a line break after the opening parenthesis. Indent arguments by four spaces. Group multiple closely related arguments on the same line.

```
drawSquare(
    x = 10, y = 10,
    width = 100, height = 100,
    fill = true
)
```

Put spaces around the `=` sign separating the argument name and value.

Wrap chained calls

When wrapping chained calls, put the `.` character or the `?.` operator on the next line, with a single indent:

```kotlin
val anchor = owner
    ?.firstChild!!
    .siblings(forward = true)
    .dropWhile { it is PsiComment || it is PsiWhiteSpace }
```

The first call in the chain should usually have a line break before it, but it's OK to omit it if the code makes more sense that way.

Lambdas

In lambda expressions, spaces should be used around the curly braces, as well as around the arrow which separates the parameters from the body. If a call takes a single lambda, pass it outside parentheses whenever possible.

```kotlin
list.filter { it > 10 }
```

If assigning a label for a lambda, do not put a space between the label and the opening curly brace:

```kotlin
fun foo() {
    ints.forEach lit@{
        // ...
    }
}
```

When declaring parameter names in a multiline lambda, put the names on the first line, followed by the arrow and the newline:

```kotlin
appendCommaSeparated(properties) { prop ->
    val propertyValue = prop.get(obj)  // ...
}
```

If the parameter list is too long to fit on a line, put the arrow on a separate line:

```kotlin
foo {
    context: Context,
    environment: Env
    ->
    context.configureEnv(environment)
}
```

Trailing commas

A trailing comma is a comma symbol after the last item in a series of elements:

```kotlin
class Person(
    val firstName: String,
    val lastName: String,
    val age: Int, // trailing comma
)
```

Using trailing commas has several benefits:

- It makes version-control diffs cleaner – as all the focus is on the changed value.
- It makes it easy to add and reorder elements – there is no need to add or delete the comma if you manipulate elements.

- It simplifies code generation, for example, for object initializers. The last element can also have a comma.

Trailing commas are entirely optional – your code will still work without them. The Kotlin style guide encourages the use of trailing commas at the declaration site and leaves it at your discretion for the call site.

To enable trailing commas in the IntelliJ IDEA formatter, go to **Settings/Preferences | Editor | Code Style | Kotlin**, open the **Other** tab and select the **Use trailing comma** option.

Enumerations

```
enum class Direction {
    NORTH,
    SOUTH,
    WEST,
    EAST, // trailing comma
}
```

Value arguments

```
fun shift(x: Int, y: Int) { /*...*/ }
shift(
    25,
    20, // trailing comma
)
val colors = listOf(
    "red",
    "green",
    "blue", // trailing comma
)
```

Class properties and parameters

```
class Customer(
    val name: String,
    val lastName: String, // trailing comma
)
class Customer(
    val name: String,
    lastName: String, // trailing comma
)
```

Function value parameters

```
fun powerOf(
    number: Int,
    exponent: Int, // trailing comma
) { /*...*/ }
constructor(
    x: Comparable<Number>,
    y: Iterable<Number>, // trailing comma
) {}
fun print(
    vararg quantity: Int,
    description: String, // trailing comma
) {}
```

Parameters with optional type (including setters)

```
val sum: (Int, Int, Int) -> Int = fun(
    x,
    y,
    z, // trailing comma
): Int {
    return x + y + x
}
println(sum(8, 8, 8))
```

Indexing suffix

```
class Surface {
    operator fun get(x: Int, y: Int) = 2 * x + 4 * y - 10
}
fun getZValue(mySurface: Surface, xValue: Int, yValue: Int) =
    mySurface[
        xValue,
        yValue, // trailing comma
    ]
```

Parameters in lambdas

```
fun main() {
    val x = {
            x: Comparable<Number>,
            y: Iterable<Number>, // trailing comma
        ->
        println("1")
    }
    println(x)
}
```

when entry

```
fun isReferenceApplicable(myReference: KClass<*>) = when (myReference) {
    Comparable::class,
    Iterable::class,
    String::class, // trailing comma
        -> true
    else -> false
}
```

Collection literals (in annotations)

```
annotation class ApplicableFor(val services: Array<String>)
@ApplicableFor([
    "serializer",
    "balancer",
    "database",
    "inMemoryCache", // trailing comma
])
fun run() {}
```

Type arguments

```
fun <T1, T2> foo() {}
fun main() {
    foo<
            Comparable<Number>,
            Iterable<Number>, // trailing comma
            >()
}
```

Type parameters

```
class MyMap<
        MyKey,
        MyValue, // trailing comma
        > {}
```

Destructuring declarations

```
data class Car(val manufacturer: String, val model: String, val year: Int)
val myCar = Car("Tesla", "Y", 2019)
val (
    manufacturer,
    model,
    year, // trailing comma
) = myCar
val cars = listOf<Car>()
fun printMeanValue() {
    var meanValue: Int = 0
    for ((
        _,
        _,
        year, // trailing comma
    ) in cars) {
        meanValue += year
    }
    println(meanValue/cars.size)
}
printMeanValue()
```

Documentation comments

For longer documentation comments, place the opening `/**` on a separate line and begin each subsequent line with an asterisk:

```
/**
 * This is a documentation comment
 * on multiple lines.
 */
```

Short comments can be placed on a single line:

```
/** This is a short documentation comment. */
```

Generally, avoid using `@param` and `@return` tags. Instead, incorporate the description of parameters and return values directly into the documentation comment, and add links to parameters wherever they are mentioned. Use `@param` and `@return` only when a lengthy description is required which doesn't fit into the flow of the main text.

```
// Avoid doing this:

/**
 * Returns the absolute value of the given number.
 * @param number The number to return the absolute value for.
 * @return The absolute value.
 */
fun abs(number: Int): Int { /*...*/ }

// Do this instead:

/**
 * Returns the absolute value of the given [number].
 */
fun abs(number: Int): Int { /*...*/ }
```

Avoid redundant constructs

In general, if a certain syntactic construction in Kotlin is optional and highlighted by the IDE as redundant, you should omit it in your code. Do not leave unnecessary syntactic elements in code just "for clarity".

Unit return type

If a function returns Unit, the return type should be omitted:

```
fun foo() { // ": Unit" is omitted here

}
```

Semicolons

Omit semicolons whenever possible.

String templates

Don't use curly braces when inserting a simple variable into a string template. Use curly braces only for longer expressions:

```
println("$name has ${children.size} children")
```

Use multi-dollar string interpolation to treat the dollar sign chars `$` as string literals:

```kotlin
val KClass<*>.jsonSchema : String
    get() = $$"""
        {
            "$schema": "https://json-schema.org/draft/2020-12/schema",
            "$id": "https://example.com/product.schema.json",
            "$dynamicAnchor": "meta",
            "title": "$${simpleName ?: qualifiedName ?: "unknown"}",
            "type": "object"
        }
        """
```

Idiomatic use of language features

Immutability

Prefer using immutable data to mutable. Always declare local variables and properties as `val` rather than `var` if they are not modified after initialization.

Always use immutable collection interfaces (`Collection` , `List` , `Set` , `Map`) to declare collections which are not mutated. When using factory functions to create collection instances, always use functions that return immutable collection types when possible:

```kotlin
// Bad: use of a mutable collection type for value which will not be mutated
fun validateValue(actualValue: String, allowedValues: HashSet<String>) { ... }

// Good: immutable collection type used instead
fun validateValue(actualValue: String, allowedValues: Set<String>) { ... }

// Bad: arrayListOf() returns ArrayList<T>, which is a mutable collection type
val allowedValues = arrayListOf("a", "b", "c")

// Good: listOf() returns List<T>
val allowedValues = listOf("a", "b", "c")
```

Default parameter values

Prefer declaring functions with default parameter values to declaring overloaded functions.

```kotlin
// Bad
fun foo() = foo("a")
fun foo(a: String) { /*...*/ }

// Good
fun foo(a: String = "a") { /*...*/ }
```

Type aliases

If you have a functional type or a type with type parameters which is used multiple times in a codebase, prefer defining a type alias for it:

```kotlin
typealias MouseClickHandler = (Any, MouseEvent) -> Unit
typealias PersonIndex = Map<String, Person>
```

If you use a private or internal type alias for avoiding name collision, prefer the `import ... as ...` mentioned in Packages and Imports.

Lambda parameters

In lambdas which are short and not nested, it's recommended to use the `it` convention instead of declaring the parameter explicitly. In nested lambdas with parameters, always declare parameters explicitly.

Returns in a lambda

Avoid using multiple labeled returns in a lambda. Consider restructuring the lambda so that it will have a single exit point. If that's not possible or not clear enough, consider converting the lambda into an anonymous function.

Do not use a labeled return for the last statement in a lambda.

Named arguments

Use the named argument syntax when a method takes multiple parameters of the same primitive type, or for parameters of `Boolean` type, unless the meaning of all parameters is absolutely clear from context.

```
drawSquare(x = 10, y = 10, width = 100, height = 100, fill = true)
```

Conditional statements

Prefer using the expression form of `try`, `if`, and `when`.

```
return if (x) foo() else bar()
```

```
return when(x) {
    0 -> "zero"
    else -> "nonzero"
}
```

The above is preferable to:

```
if (x)
    return foo()
else
    return bar()
```

```
when(x) {
    0 -> return "zero"
    else -> return "nonzero"
}
```

if versus when

Prefer using `if` for binary conditions instead of `when`. For example, use this syntax with `if`:

```
if (x == null) ... else ...
```

Instead of this one with `when`:

```
when (x) {
    null -> // ...
    else -> // ...
}
```

Prefer using `when` if there are three or more options.

Guard conditions in when expression

Use parentheses when combining multiple boolean expressions in `when` expressions or statements with guard conditions:

```
when (status) {
    is Status.Ok if (status.info.isEmpty() || status.info.id == null) -> "no information"
}
```

Instead of:

```
when (status) {
    is Status.Ok if status.info.isEmpty() || status.info.id == null -> "no information"
}
```

Nullable Boolean values in conditions

If you need to use a nullable `Boolean` in a conditional statement, use `if (value == true)` or `if (value == false)` checks.

Loops

Prefer using higher-order functions (`filter` , `map` etc.) to loops. Exception: `forEach` (prefer using a regular `for` loop instead, unless the receiver of `forEach` is nullable or `forEach` is used as part of a longer call chain).

When making a choice between a complex expression using multiple higher-order functions and a loop, understand the cost of the operations being performed in each case and keep performance considerations in mind.

Loops on ranges

Use the `..<` operator to loop over an open-ended range:

```
for (i in 0..n - 1) { /*...*/ }  // bad
for (i in 0..<n) { /*...*/ }  // good
```

Strings

Prefer string templates to string concatenation.

Prefer multiline strings to embedding `\n` escape sequences into regular string literals.

To maintain indentation in multiline strings, use `trimIndent` when the resulting string does not require any internal indentation, or `trimMargin` when internal indentation is required:

```kotlin
fun main() {
    println("""
     Not
     trimmed
     text
     """
    )

    println("""
     Trimmed
     text
     """.trimIndent()
    )

    println()

    val a = """Trimmed to margin text:
            |if(a > 1) {
            |    return a
            |}""".trimMargin()

   println(a)
}
```

Learn the difference between Java and Kotlin multiline strings.

Functions vs properties

In some scenarios, functions with no arguments might be interchangeable with read-only properties. Although the semantics are similar, there are some stylistic conventions on when to prefer one to another.

Prefer a property over a function when the underlying algorithm:

- Does not throw.
- Is cheap to calculate (or cached on the first run).
- Returns the same result over invocations if the object state hasn't changed.

Extension functions

Use extension functions liberally. Every time you have a function that works primarily on an object, consider making it an extension function accepting that object as a receiver. To minimize API pollution, restrict the visibility of extension functions as much as it makes sense. As necessary, use local extension functions, member extension functions, or top-level extension functions with private visibility.

Infix functions

Declare a function as `infix` only when it works on two objects which play a similar role. Good examples: `and`, `to`, `zip`. Bad example: `add`.

Do not declare a method as `infix` if it mutates the receiver object.

Factory functions

If you declare a factory function for a class, avoid giving it the same name as the class itself. Prefer using a distinct name, making it clear why the behavior of the factory function is special. Only if there is really no special semantics, you can use the same name as the class.

```
class Point(val x: Double, val y: Double) {
    companion object {
        fun fromPolar(angle: Double, radius: Double) = Point(...)
    }
}
```

If you have an object with multiple overloaded constructors that don't call different superclass constructors and can't be reduced to a single constructor including parameters with default values, prefer to replace the overloaded constructors with factory functions.

Platform types

A public function/method returning an expression of a platform type must declare its Kotlin type explicitly:

```
fun apiCall(): String = MyJavaApi.getProperty("name")
```

Any property (package-level or class-level) initialized with an expression of a platform type must declare its Kotlin type explicitly:

```
class Person {
    val name: String = MyJavaApi.getProperty("name")
}
```

A local value initialized with an expression of a platform type may or may not have a type declaration:

```
fun main() {
    val name = MyJavaApi.getProperty("name")
    println(name)
}
```

Scope functions apply/with/run/also/let

Kotlin provides a set of functions to execute a block of code in the context of a given object: `let`, `run`, `with`, `apply`, and `also`. For the guidance on choosing the right scope function for your case, refer to Scope Functions.

Coding conventions for libraries

When writing libraries, it's recommended to follow an additional set of rules to ensure API stability:

- Always explicitly specify member visibility (to avoid accidentally exposing declarations as public API).
- Always explicitly specify function return types and property types (to avoid accidentally changing the return type when the implementation changes).
- Provide KDoc comments for all public members, except for overrides that do not require any new documentation (to support generating documentation for the library).