DiKTat - Kotlin linter

ANDREY KULESHOV, Lomonosov Moscow State University, Russia
PETR TRIFANOV, Lomonosov Moscow State University, Russia
DENIS KUMAR, Higher School of Economics, Russia
ALEXANDER TSAY, Higher School of Economics, Russia

1 INTRODUCTION

It is necessary to follow a specific style of code during software development. Otherwise code will become less readable and the developer will not be able to understand other programmers' intent. It can lead to functional defects and bugs in the code. Static analyzers, in it's turn, have methods for detecting and auto-correcting style errors and bugs. Modern linters and static analyzers are extremely useful not only for simple code-style analysis but also for bugs detection and automatic code fixing.

There are many methods and techniques used by existing analyzers to find bugs (path-sensitive data flow analysis [?], alias analysis [?], type analysis [?], symbolic execution [?], abstract interpretation [?]).

Senior developer can write the same comment again and again in hundreds of code reviews. Static analysis reduces this bureaucracy as it can be thought of as an automated code review process, because it can detect those issues in code automatically. And of course it perfectly reduces the human factor in the review process. There are two main tasks that can be solved by static code analysis: identifying errors (bugs) in programs and recommending code formatting (fixes). This means that the analyzer allows you, for example, to check whether the source code complies with the accepted coding convention and automatically fix found issues. Also, a static analyzer can be used to determine the level of maintainability of a code. It shows how easy is it to read, modify and adapt a given code of software by detecting code-smells and design patterns used in the code. Static analysis tools allow you to identify a large number of errors in the design phase, which significantly reduces the development cost of the entire project. Static analysis covers the entire code - it checks even those code fragments that are difficult to test. It does not depend on the compiler used and the environment in which the compiled program will be executed.

This white-paper covers the work that was done to create a static analyzer for Kotlin language, called diKTat. It also briefly describes it's implementation and functionality. You can treat this document as a "how-to" instruction for diKTat.

2 DEFINITION

Before we will move one, it is necessary to define some terms for better understanding of context. The first and basic concept that should be introduced is **Rule** (marked with R_i). Rule in diKTat is the logic described in a special class named with "`Rule`" suffix, which checks whether code meets a certain paragraph of code-style. The set - is a well-defined collection of distinct objects, considered as an object in its own right. So we can define a **Ruleset** - a set of such code analysis Rules. We will mark any of such set of Rules with R.

Inspection is the part of any Rule. It is an algorithm that can detect (marked with W_i) or fix (marked with F_i) invalid code. It is very important to understand that $Rule \neq Inspection(Inspection \subset Rule)$. We will use I_i notation to mark each separate inspection. So it is obvious that: $I_i = W_i \cup F_i$, where $i \in \mathbb{N}$. Using the same logic we can say that $R = \bigcup_i R_i$ where $R_i = \bigcup_j I_j$.

Abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language (Kotlin in our case). Each node of the tree denotes a construct occurring in the source code. **CI/CD** - continuous integration (CI) and continuous delivery (CD) is a methodology that allows application development teams to make changes to code more frequently and reliably [?]. **KDoc** - is the language used to document Kotlin code (the equivalent of Java's JavaDoc).

3 KOTLIN

This section explains why we are focused exactly on the static analysis for Kotlin language.

Kotlin is a cross-platform, statically typed, general-purpose programming language with type inference. Kotlin is designed to interoperate fully with Java, and the JVM version of Kotlin's standard library depends on the Java Class Library, but type inference allows its syntax to be more concise. This language can be called "Java on steroids" as it takes best features from different languages and puts it on the top of JVM-world. Kotlin mainly targets the JVM, but also is compiled to JavaScript (e.g. for frontend web applications using React or Thymeleaf) or native code (via LLVM), e.g. for native iOS apps sharing business logic with Android apps.

Kotlin has quickly skyrocketed in popularity. It's used by Google, Square, Pinterest, Pivotal, Netflix, Atlassian and many other companies. It's the fastest-growing programming language, according to GitHub, growing over 2,5 times in the past year (2019). It was voted one of the five most loved languages, according to Stack Overflow. There are even meetups and conferences focused only on Kotlin[1].

Kotlin is used in a lot of ways. For example, it can be used for backend development using `ktor` framework (Kotlin framework developed by JetBrains), and `Spring` framework that also has first-party support for kotlin (`Spring` is one of the most popular framework on Java for Web development). Kotlin/JS provides the ability to transpile your Kotlin code to JavaScript, as well as providing JS variant of Kotlin standard library and interopability with existing JS dependencies, both for Node.js and browser. There are numerous ways how Kotlin/JS can be used. For instance, Kotlin/JS is used to create frontend web applications, server-side and serverless applications, libraries for use with JavaScript and TypeScript.

Support for multiplatform programming is one of key benefits. It reduces time for writing and maintaining the same code for different platforms while retaining the flexibility and benefits of native programming. We think that it is the main reason why Kotlin is so popular in the community of mobile developers.

Kotlin supports well asynchronous or non-blocking programming. Whether we're creating server-side, desktop or mobile applications, it's important that we provide an experience that is not only fluid from the user's perspective, but scalable when needed. Kotlin has chosen a very flexible approach one by providing Coroutine support as a first-party library `kotlinx.coroutines` with a kotlin compiler plugin. A coroutine is a concurrent

[1]https://www.businessinsider.com/kotlin-programming-language-explained-popularity-2019-5#:~:text=Kotlin%20has%20quickly%20skyrocketed%20in,times%20in%20the%20past%20year

design pattern that you can use on Android to simplify code that executes asynchronously. Coroutines (in a form of kotlinx.coroutines library and kotlin compiler plugin) were added to Kotlin in version 1.3 and are based on established concepts from other languages. Also, coroutines do not only open the doors to an easy asynchronous programming in Kotlin, but also provide a wealth of other possibilities such as concurrency, actors, etc.

On Android, coroutines help to manage long-running tasks that might otherwise block the main thread and cause your app to become unresponsive. Over 50% of professional developers who use coroutines have reported that productivity had increased. So designers of Kotlin made a correct decision. Coroutines help you to write cleaner and more concise code for your applications.

The state of Kotlin in the Q3 of 2020 (according to the latest Kotlin Census and statistical data):

- 4,7 million users
- 65% of users use Kotlin in production
- Kotlin is primary language for 56% of users, which means the main or only one they use at work
- 100+ people are on the Kotlin development team at JetBrains
- 350+ independent contributors develop the language and its ecosystem outside of JetBrains

Kotlin is used by developers of open-source operating systems like HarmonyOS and Android. In 2019 Google announced that the Kotlin programming language is now its preferred language for Android app developers. In the same year Stack Overflow stated that Kotlin is fourth most loved language in community. Nowadays there are over 60% of android developers who use Kotlin as their main language. [2]

Kotlin's popularity can be explained by the rising number of Android users (last year, 124.4m in the USA) and, thus, Android-based devices. 80% of Kotlin programmers use the language to build Android apps, 31% for back-end applications, 30% for SDK/libraries. Kotlin is also interoperable with Java, which allows developers to use all existing Android libraries in a Kotlin app. Now (2020) Kotlin is in the top-10 of PYPL rating:

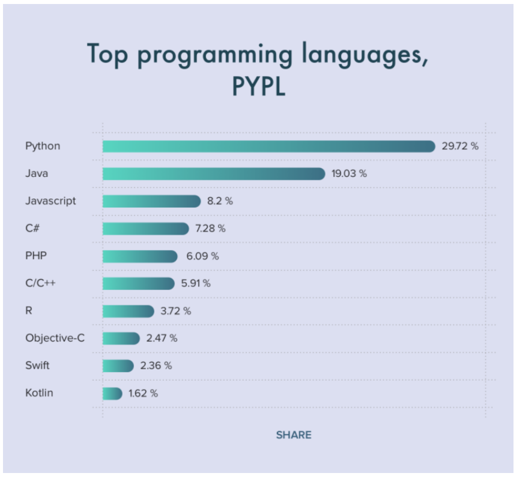


Fig. 1. Top programming languages, 2020

Overall, Kotlin is a modern language that gains its popularity incredibly fast. It is mostly used by Android developers, but other "branches of programming" are gaining popularity as well, for example Spring framework (the most popular Java framework) supports Kotlin. It supports both OO (object-oriented) and FP (function-oriented) programming paradigms. Since the release 1.4 Kotlin claims to bring major updates every 6 month.

[2]https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/

4 DIKTAT

4.1 What is diKTat?

DiKTat [3] - is a formal strict code-style for Kotlin language and a linter with a set of rules that implement this code-style. Basically, it is a collection of Kotlin code style rules implemented as AST visitors on top of KTlint framework [4]. Diktat detects and automatically fixes code style errors and code smells based on the configuration of rules. DiKTat is a highly configurable framework, that can be extended further by adding custom rules. It can be run as command line application or with maven or gradle plugins. In this paper, we will explain how diKTat works, describe its advantages and disadvantages and compare it with other static analyzers for Kotlin. The main idea is to use diktat in your CI/CD pipeline.

4.2 Why diKTat?

DiKTat permits formal flexible description or Rules and Inspections expressed by means of yml file. We looked at similar existing projects and realized that their functionality does not give us a chance to implement our own configurable code style. Most of rules which we wanted to implement were missing in other analyzers. Mostly all of those analyzers had hardcoded logic and prohibited configuration. That's why we decided that we need to create convenient, user friendly and easily configured tool for developers.

First of all, diKTat has its own highly configurable Ruleset R_{diktat} that contains unique Inspections, missing in other Kotlin static analyzers. You just need to set your own options which fit your project the most. In case you don't want to do this - you can use the default configuration, but some of complex inspections will be disabled. Basically, Ruleset is an `yml` file with a description of each rule.

Secondly, DiKTat has its own plugins and can be run via Maven, Gradle and command line. Developer can use build automation system that he prefers.

Finally, developer can disable with diKTat each inspection from the code using special annotations on the line where he wants to suppress an Inspection.

5 COMPARATIVE ANALYSIS

5.1 About ktlint

Ktlint is a popular an anti-bikeshedding Kotlin linter with a built-in formatter created by Pinterest[5]. It tries to reflect official code style from `kotlinlang.org` and Android Kotlin Style Guide and then automatically apply these rules to your codebase. Ktlint can check and automatically fix code. It claims to be simple and easy to use. As it is focused more on checking code-style and code-smell related issues, ktlint inspections work with Abstract Syntax Tree generated by Kotlin parser. Ktlint framework has some basic utilities to make the work with Kotlin AST easier, but anyway all inspections work with original ASTNode provided by Kotlin parser.

Ktlint has been developed since 2016 and since then it has 3.8k stars, 309 forks and 390 closed PRs (2020). It looks to be the most popular and mature linter in the Kotlin community right now with approximately 15k lines of code written.

Ktlint has its own set of rules, which are divided on standard and experimental rules. But unfortunately the number of fixers and checkers in the standard ruleset is very few (20 rules) and inspections are trivial.

Ktlint can be used as a plugin for Maven, Gradle or command line app. `.editorconfig` file should be modified to configure rules. This is the - only configuration that ktlint provides and it contains just simple configuration like the number of spaces in indents. Actually user even can't configure specific rules (for example to disable or

[3]https://github.com/cqfn/diKTat
[4]https://github.com/pinterest/ktlint
[5]https://github.com/pinterest/

suppress any of them), instead you can provide some common settings like the number of spaces for indenting. In other words, ktlint has a "fixed hardcoded" code-style that is not very configurable.

If you want to implement your own rules you need to create a your own Ruleset. Ktlint is very user-friendly for creation of custom Rulesets. In this case ktlint will parse the code using a Kotlin parser and will trigger your inspection (as visitor) for each node of AST. Ktlint uses javas `ServiceLoader` to discover all available Rulesets. `ServiceLoader` is used to inject your own implementation of rules for the static analysis. In this case ktlint becomes both a third-party dependency and a framework. Basically you should provide implementation of `RuleSetProvider` interface.

Ktlint refers to article on Medium[6] on how to create a custom Ruleset and a Rule.

A lot of projects uses ktlint as their code formatting tool. For example, OmiseGo [7] (currently rebranding to OMG Network) - is a quite popular cryptocurrency.

To summarize: Ktlint is very mature and useful as a framework for creating your own checker&fixer of Kotlin code and doing AST-analysis. It can be very useful if you need only simple inspections that check (and fix) code-style issues (like indents).

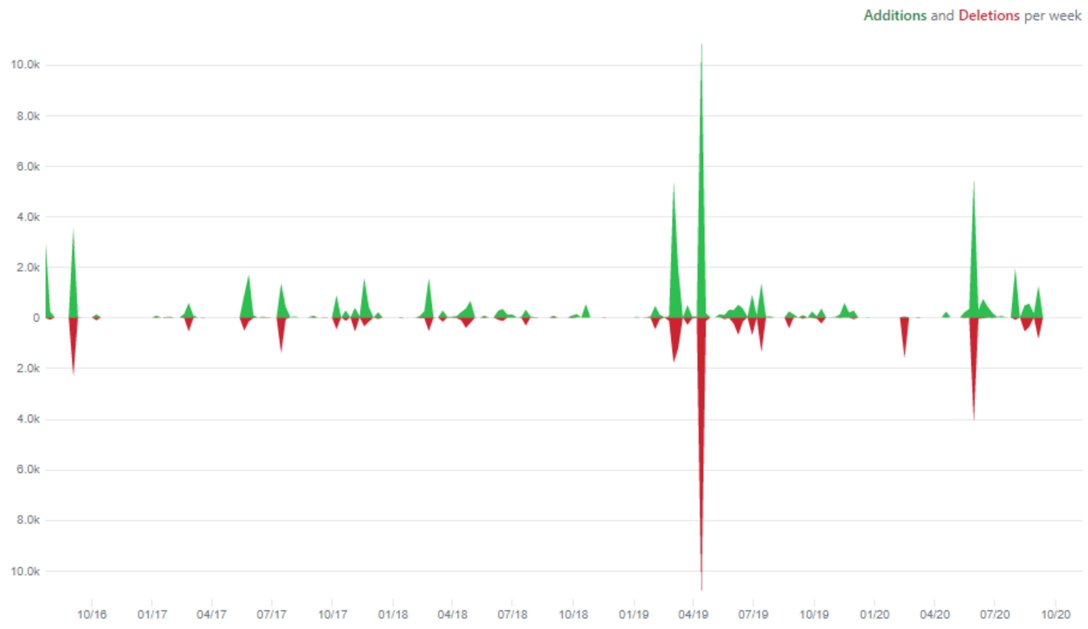


Fig. 2. Ktlint Code Frequency

5.2 About detekt

Detekt [8] is a static code analysis tool. It operates on an abstract syntax tree (AST) and meta-information provided by Kotlin compiler. On the top of that info, it does a complex analysis of the code. However, this project is more

[6]https://medium.com/mydevnotes/ktlint-improve-your-kotlin-code-quality-with-lint-checks-13a4456c4600
[7]https://github.com/omgnetwork/android-sdk
[8]https://github.com/detekt/detekt

focused on checking the code rather than fixing. Similarly, to ktlint, it has its own rules and inspections. Detekt uses wrapped ktlint to redefine RuleSet of ktlint as it's formatting rules.

Detekt supports detection of code smells, bugs searching and code-style checking. It has a highly configurable rule sets (can even make suppression of issues from the code). And the number of checkers is large: it has more than 100 inspections. Detekt has IntelliJ integration, third-party integrations for Maven, Bazel and Github actions and a mechanism for suppression of their warnings with @Suppress annotation from the code. It is being developed since 2016 and today it has 3.2k stars, 411 forks and 1850 closed PRs. It has about 45k lines of code. And its codebase is the biggest comparing to other analyzers. Detekt is used in such projects as fountain [9] or Kaspresso [10].

To summarize: Detekt is very useful as a Kotlin static analyser for CI/CD. It tries to find bugs in the code and is focused more on checking of the code. Detekt has 100+ rules which check the code.

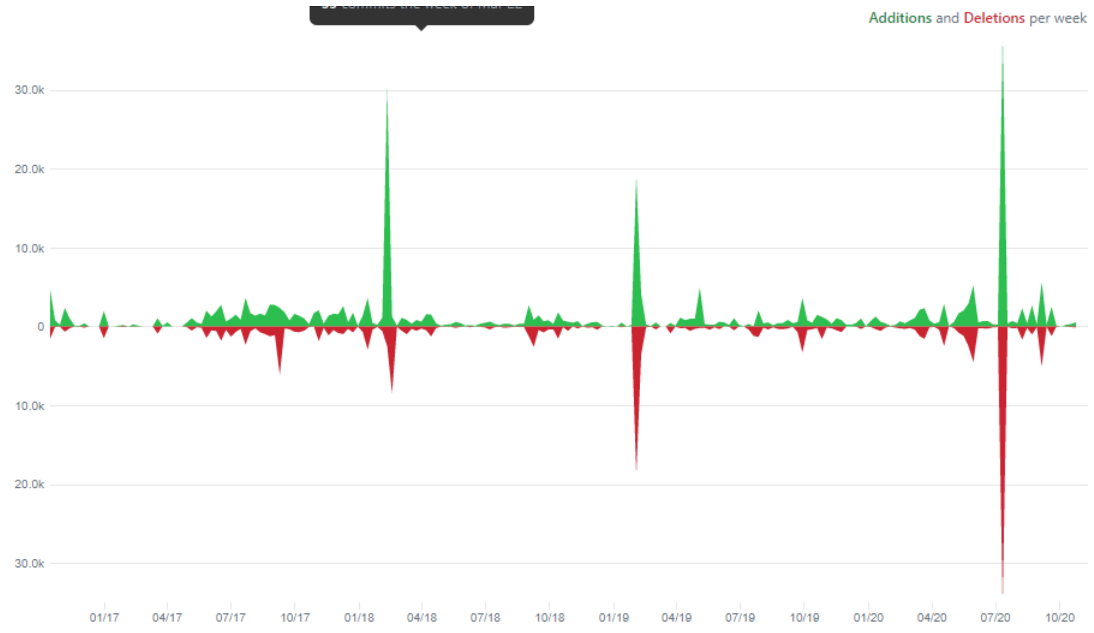


Fig. 3. Detekt Code Frequency

5.3 About ktfmt

Ktfmt formats is a program that formats Kotlin code, based on google-java-format. Its development started in Facebook at the end of 2019. It can be added to client's project through a Maven dependency, Gradle dependency, IntelliJ plugin or run through a command line. Ktfmt is not a configurable application, so to change any rule logic you need to download the project and redefine some constants. Ktfmt has 214 stars, 16 forks, 20 closed PRs and around 7500 lines of code.

To summarize: no one knows why Facebook has invested their money in this tool. Nothing new was introduced. If they really needed to have new rules - they could create their own Ruleset for ktlint or detekt.

[9] https://github.com/xmartlabs/fountain
[10] https://github.com/KasperskyLab/Kaspresso

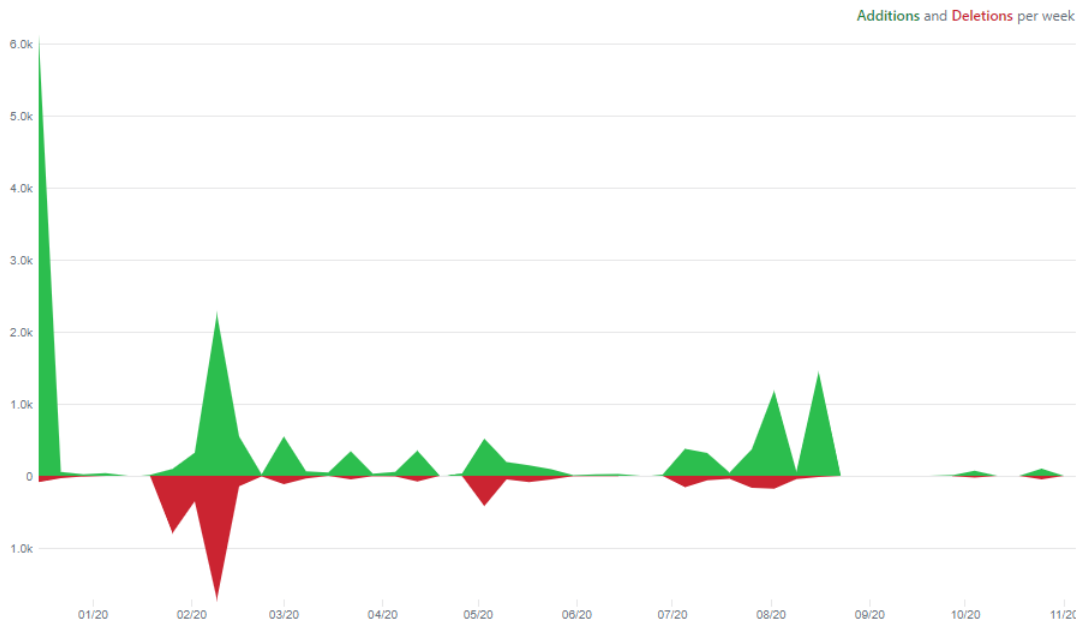


Fig. 4. Ktfmt Code Frequency

5.4 About diKTat

Diktat is a static code analysis tool as well as ktlint and detect. But diktat is not only a tool, but also a coding convention that describes in details all the rules that you should follow when writing a code on Kotlin. Its development has started in 2020 and at the time of writing this article diKTat has 168 stars and 13 forks. DiKTat operates on AST provided by kotlin compiler. So why diKTat is better?

First of all, it supports much more rules than ktlint. Its ruleset includes more than 100 rules, that can both check and fix your code.

Secondly, diKTat is configurable. A lot of rules have their own settings, and all of them can be easily understood. For example, you can choose whether you need a copyright, choose a length of line or you can configure your indents.

Third, diKTat is very easy to configure. You don't need to spend hours only to understand what each rule does. Diktat's ruleset is a `.yml` file, where each rule is commented out with the description. Also you can suppress error on the particular lines of code using `@Suppress` annotation in your code.

DiKTat can be used as a CI/CD tool in order to avoid merging errors in the code. Overall it can find code smells and code style issues. Also it can find pretty not obvious bugs by complex AST analysis. Diktat works with maven, gradle and as command-line application powered by ktlint.

To summarize: diktat contains a strict coding convention that was not yet introduced by other linters. It works both as a checker and as a fixer. Diktat has much more inspections (100+) and is very configurable (each inspection can be disabled/configured separately), so you can configure it for your particular project.

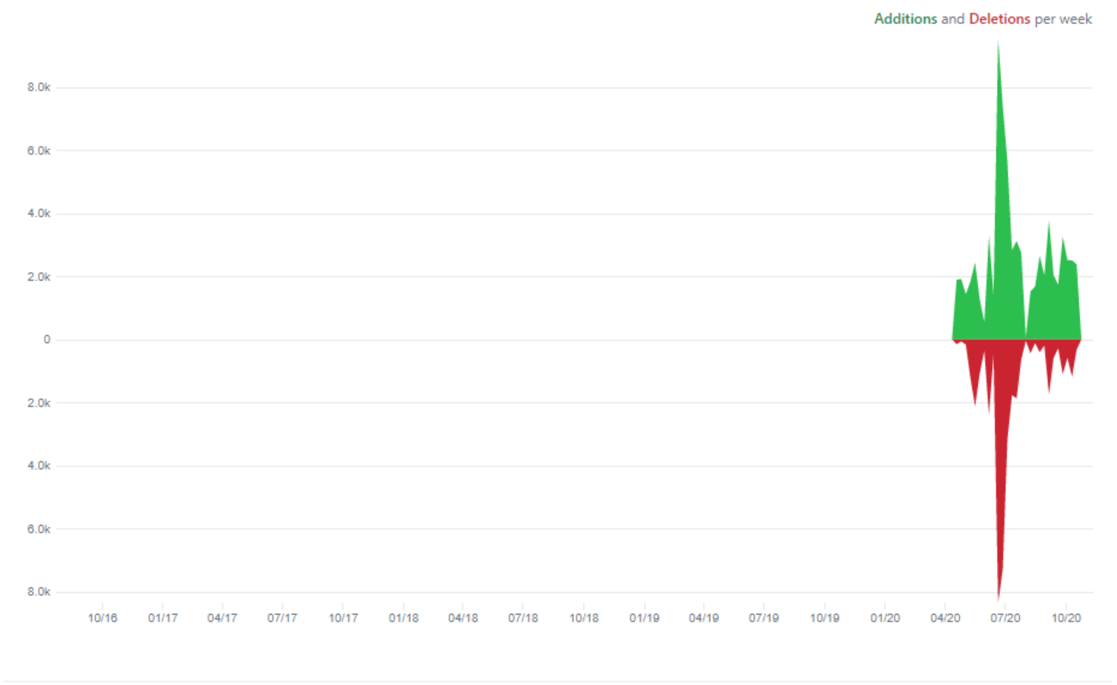


Fig. 5. DiKTat Code Frequency

5.5 A few words about Jetbrains

Jetbrains invented Kotlin and created one of the best IDEs for Java and Kotlin called IntelliJ. This IDE supports a built-in linter. However, it is not a well-configurable tool, you are not able to specify your own coding convention and it is not useful for CI/CD as it is highly coupled with UI. Unfortunately such static analysis is not so effective as it cannot prevent merging of the code with bugs into the repository. As experience shows - many developers simply ignore those static analysis errors until they are blocked from merging their pull requests. So it is not so suitable for CI/CD, but very good for finding and fixing issues inside your IDE.

5.6 Summary

To sum up, four linters, excepting diKTat, were mentioned above and each of them has it's own strengths and weaknesses. Diktat, in its turn, is uniting its strengths and providing new features in code linting and fixing tools.

Comparing table				
	diKTat	ktlint	detekt	ktfmt
starting year	2020	2016	2016	2019
stars	168	3.2k	3.8k	214
forks	13	299	411	16
closed PRs	321	390	1850	20
lines of code	32k	15k	45k	7,5k
number of rules	>100	≈ 20	>100	≈ 10
is configurable	yes	no	yes/no	no
maven/gradle plugin	both	both	gradle only	no
web version	yes	yes	no	no

6 HOW DOES DIKTAT WORK

Diktat does AST-analysis, using Abstract Syntax Tree for creation of internal representation (IR) from the parsed code by the kotlin-compiler. This chapter describes how diktat works.

6.1 ktlint

To quickly and efficiently analyze the program code, you first need to transform it into a convenient data structure. This is exactly what ktlint does - it parses plain text code into an abstract syntax tree. So we decided not to choose the way of development that was chosen by Facebook in ktfmt and not invent our own framework for parsing the code. We decided to write our own Ruleset on the top of ktlint framework. The good thing is that we were able to inject our set of rules to ktlint via Java's `ServiceLoader`[11]

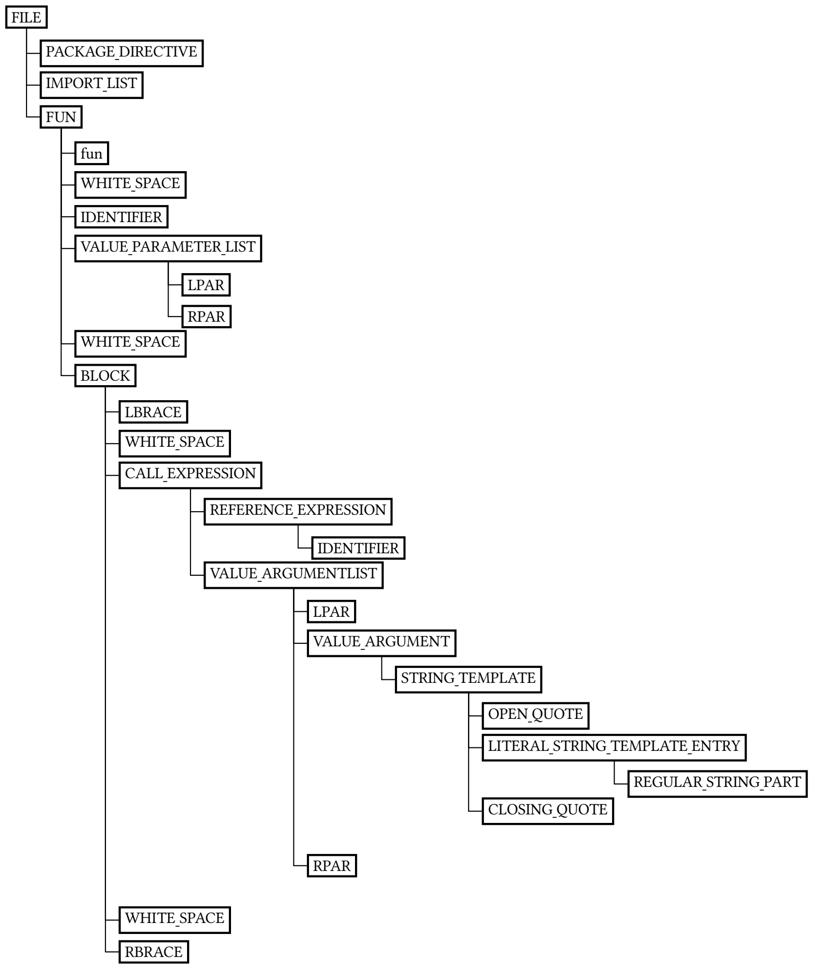
In ktlint, the transformation of code happens in the *prepareCodeForLinting*[12] method. This method uses kotlin-compiler-embeddable library to create a root node of type FILE. For example, this simple code:

```kotlin
fun main() {
  println("Hello World")
}
```

Listing 1. Simple function that will be transformed to AST

will be converted into the following AST:

[11]https://docs.oracle.com/javase/8/docs/api/java/util/ServiceLoader.html
[12]https://github.com/pinterest/ktlint/blob/master/ktlint-core/src/main/kotlin/com/pinterest/ktlint/core/KtLint.kt



If there are error elements inside the constructed tree, then the corresponding error is displayed. If the code is valid and parsed without errors, for each rule in the Ruleset, the *visit* method is called for the root node itself and its "children" are sequentially passed. When you run program, you can pass flags to ktlint - one of them is `-F`. This flag means that the rule will not only report an error, but will also try to fix it.

6.2 DiKTat

Another feature of ktlint is that at it's startup you can provide a JAR file with additional ruleset(s), that will be discovered by the `ServiceLoader` and then all AST nodes will be passed to these rules. DiKTat uses this approach.

The only modification Diktat makes to the framework is that it adds a mechanism to disable Inspection from the code using annotations or configuration file. The set of all rules is described in the *DiktatRuleSetProvider*[13] class. This class overrides the *get()* method of the *RuleSetProvider*[14] interface, which returns a set of rules to be

[13]https://github.com/cqfn/diKTat/blob/v0.1.3/diktat-rules/src/main/kotlin/org/cqfn/diktat/ruleset/rules/DiktatRuleSetProvider.kt
[14]https://github.com/pinterest/ktlint/blob/master/ktlint-core/src/main/kotlin/com/pinterest/ktlint/core/RuleSetProvider.kt

"traversed". But before returning this set Diktat is reading the configuration file where the user has independently configured all the Inspections. If there is no configuration file, then a warning will be displayed and Inspections will use the default configuration file. Each rule must implement the *visit* method of the abstract Rule class, which describes the logic of the rule. By the way it is a special case of a famous pattern Visitor [?]. Implementation example of the simple Rule that contains one Inspections can be found below.

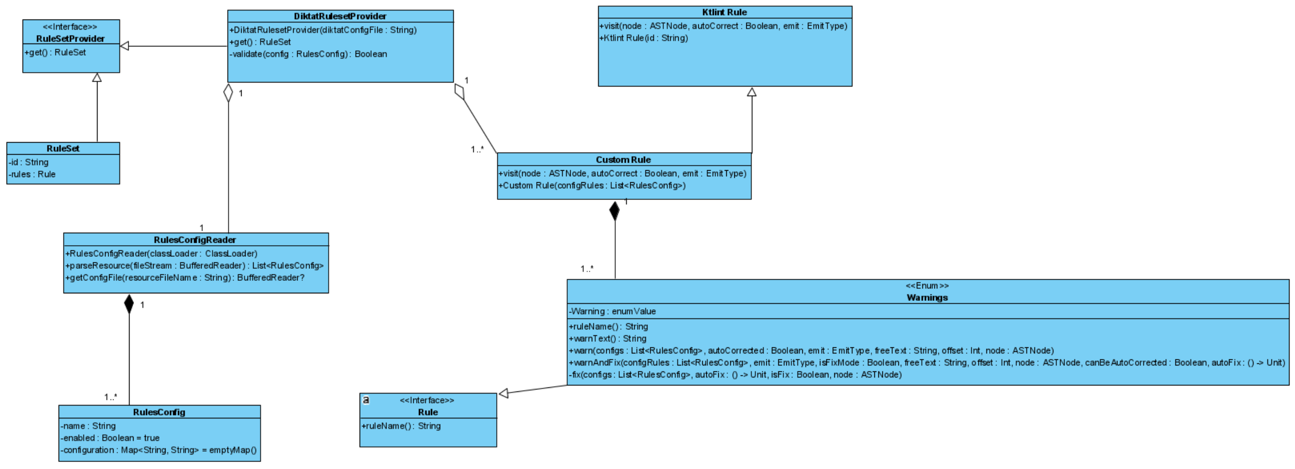
```kotlin
class SingleLineStatementsRule(private val configRules: List<RulesConfig>) : Rule("statement")
    {
    private var isFixMode: Boolean = false
    private lateinit var emitWarn: EmitType

    override fun visit(node: ASTNode,
                       autoCorrect: Boolean,
                       emit: EmitType) {
        emitWarn = emit
        isFixMode = autoCorrect

        // all the work is done with ASTNode - this is the type, provided by Kotlin compiler
        node.getChildren(TokenSet.create(SEMICOLON)).forEach {
            if (!it.isFollowedByNewline()) {
                // configuration is checked by warning mechanism under the hood
                // warnings are mapped to proper paragraph of a code standard
                MORE_THAN_ONE_STATEMENT_PER_LINE.warnAndFix(
                        configRules,
                        emitWarn,
                        isFixMode,
                        it.extractLineOfText(),
                        it.startOffset,
                        it
                // this lambda provides the logic that will be used to fix the code
                ) {
                    if (it.treeParent.elementType == ENUM_ENTRY) {
                        node.treeParent.addChild(PsiWhiteSpaceImpl("\n"), node.treeNext)
                    } else {
                        if (!it.isBeginByNewline()) {
                            val nextNode = it.parent(
                                    { parent -> parent.treeNext != null },
                                    strict = false
                            )?.treeNext

                            node.appendNewlineMergingWhiteSpace(nextNode, it)
                        }
                        node.removeChild(it)
                    }
                }
            }
        }
    }
}
```

Listing 2. Example of the Rule.

The example above describes the Rule that checks that there are no statements separated by a semicolon. The list of configurations is passed to the parameter of this rule so that the error is displayed only when the Rule is enabled (further it will be described how to enable or disable the Rule). The class fields and the *visit()* method are described below. The first parameter in method is of *ASTNode* type, this type is produced by the Kotlin compiler. It is important to understand that these visitors are called for each and every node of AST that is provided by the compiler. This is not optimal from the perspective of the performance, but makes the code much more readable and isolated.

Then a check occurs: if the code contains a line in which more than one statement per line and this rule is enabled, then the rule will be executed and, depending on the mode in which the user started ktlint, the rule will either simply report an error or fix it. In our case, when an error is found, the method is called to report and fix the error - *warnAndFix()*.

All warnings that contain similar logic (e.g. regarding formatting of function KDocs) are checked in the same Rule. This way we can make small optimisation and check similar parts of AST only once. Whether the Inspection is enabled or disabled is checked inside *warn()* or *warnAndFix()* methods. The same works for the suppression of Inspections: right before emitting the warning we check whether any of current node's parents has a Suppress annotation. The diagram below describes it's architecture:



<<Interface>>
RuleSetProvider
+get() : RuleSet
DiktatRulesetProvider
+DiktatRulesetProvider(diktatConfigFile : String)
+get() : RuleSet
-validate(config : RulesConfig) : Boolean
Ktlint Rule
+visit(node : ASTNode, autoCorrect : Boolean, emit : EmitType)
+Ktlint Rule(id : String)
RuleSet
-id : String
-rules : Rule
Custom Rule
+visit(node : ASTNode, autoCorrect : Boolean, emit : EmitType)
+Custom Rule(configRules : List<RulesConfig>)
RulesConfigReader
+RulesConfigReader(classLoader : ClassLoader)
+parseResource(fileStream : BufferedReader) : List<RulesConfig>
+getConfigFile(resourceFileName : String): BufferedReader?
<<Enum>>
Warnings
-Warning : enumValue
+ruleName(): String
+warnText() : String
+warn(configs : List<RulesConfig>, autoCorrected : Boolean, emit : EmitType, freeText : String, offset : Int, node : ASTNode)
+warnAndFix(configRules : List<RulesConfig>, emit : EmitType, isFixMode : Boolean, freeText : String, offset : Int, node : ASTNode, canBeAutoCorrected : Boolean, autoFix : () -> Unit)
-fix(configs : List<RulesConfig>, autoFix : () -> Unit, isFix : Boolean, node : ASTNode)
RulesConfig
-name : String
-enabled : Boolean = true
-configuration : Map<String, String> = emptyMap()
<<Interface>>
Rule
+ruleName(): String
1
1..*

Fig. 6. Diktat class diagram

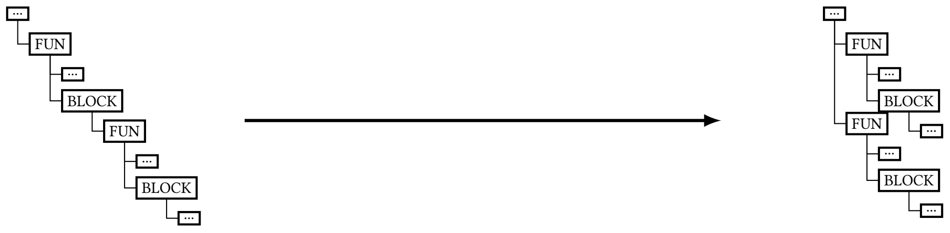
6.3 Examples of unique inspections

As already described above, DiKTat has more rules than existing analogues, therefore, it will find and fix more errors and shortcomings and, thereby, make the code cleaner and better. To better understand how detailed are checks in Diktat,let's mention a few examples:

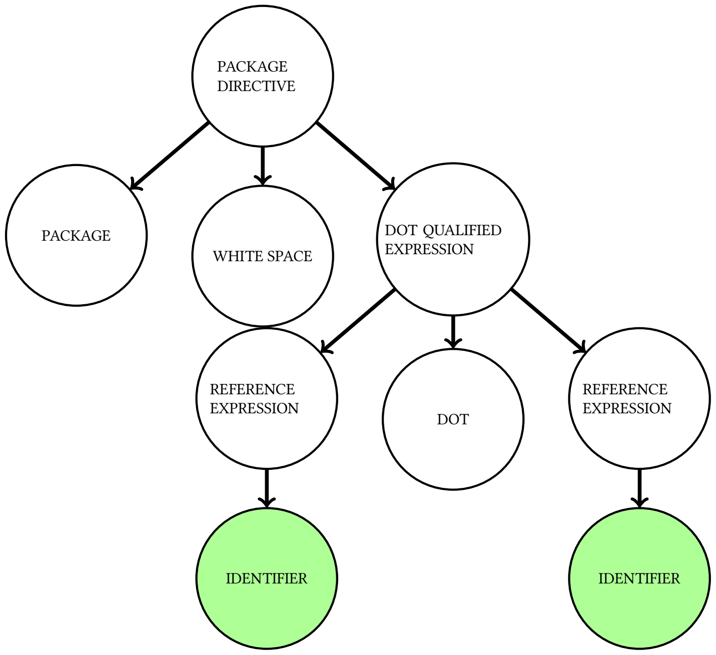
(1) **Package.** In DiKTat, there are about 6 Inspections that are checking the package naming. For comparison: detekt has only one rule, where the package name is simply checked by a pattern, in ktlint there are zero inspections.
(2) **KDoc.** KDoc is an important part of good code to make it easier to understand and navigate the program. In DiKTat there are 15 rules on KDoc, in detekt there are only 7. Therefore, DiKTat will make and correct KDoc in more detail and correctly.

There are also many unique Inspections that other analyzers do not have. Here are some of them:

(1) **COMMENTED_OUT_CODE** – This Inspection checks if the code contains commented code blocks.
(2) **FILE_CONTAINS_ONLY_COMMENTS** – This rule checks file contains not only comments.
(3) **LOCAL_VARIABLE_EARLY_DECLARATION** – This rule checks that local variables are declared close to the point where they are first used.
(4) **AVOID_NESTED_FUNCTIONS** - This rule checks for nested functions and warns and fixes if it finds any. An example of changing the tree when this rule is triggered and DiKTat is run with fix mode:



(5) **FLOAT_IN_ACCURATE_CALCULATIONS** - Inspection that checks that floating-point numbers are not used for accurate calculations (see the corresponding Rule from the code style to get more information).
(6) **PACKAGE_NAMING** - This Inspection checks that package name is in a proper format and is separated by dots. This inspection is very demonstrative to show how the work with AST is done.
This inspection receives different nodes and checks them one by one. First it checks their element type (type of the node in AST). When it's element type equals to `PACKAGE_DIRECTIVE` it gets the file name and collects all nodes with `IDENTIFIER` type as it is shown on the following graph:



In order to collect elements with a proper type, Inspection has to do a tree traversal. Tree traversal is done by a special method called `findAllNodesWithCondition()` (see Listing 3). This function searches for a node with a given condition (in this case it is when node's type equals to `IDENTIFIER`). As a basis it uses

DFS (Depth-first search): it goes recursively in depth of the tree and compares types of AST nodes. When it finds necessary node it returns it and the result from it's parent search as a list, otherwise it returns empty list. At the end the Inspection checks the package name based on identifiers and file name.

```kotlin
/**
 * This method performs tree traversal and returns all nodes which satisfy the
    condition
 */
fun ASTNode.findAllNodesWithCondition(condition: (ASTNode) -> Boolean, withSelf:
    Boolean = true): List<ASTNode> {
    val result = if (condition(this) && withSelf) mutableListOf(this) else
    mutableListOf()
    return result + this.getChildren(null).flatMap {
        it.findAllNodesWithCondition(condition)
    }
}
```

Listing 3. Method for AST traversal

7 KILLER-FEATURES

As described above, diKTat is configurable and user-friendly. But these are not all of it's advantages and features. Below we will present and describe unusual and important killer-features of diKTat.

7.1 Configuration file

It's worth starting with the configuration file. This is a file in which the user can manually turn rules on and off or configure the rules settings. Below is one of the rules in the configuration file.

```yaml
- name: DIKTAT_COMMON
  configuration:
    # put your package name here - it will be autofixed and checked
    domainName: your.name.here
- name: COMMENTED_OUT_CODE
  enabled: true
- name: HEADER_MISSING_OR_WRONG_COPYRIGHT
  enabled: true
  configuration:
    isCopyrightMandatory: true
    copyrightText: 'Copyright (c) Your Company Name Here. 2010-2020'
    testDirs: test
- name: FILE_IS_TOO_LONG
  enabled: true
  configuration:
    maxSize: '2000'
    ignoreFolders: ' '
```

Listing 4. Part of configuration file.

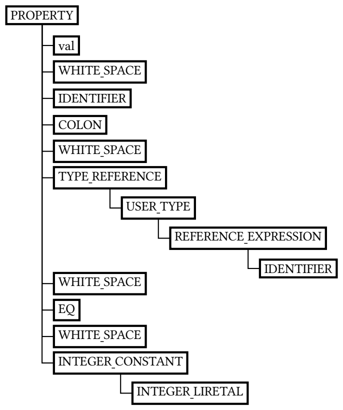
Each Inspection in this file has 3 fields: `name` - the name of the Inspection, `enabled` - whether the rule is enabled or disabled (all rules are enabled by default), `configuration` - parameters for the Inspection. With the first two, everything is obvious. The third parameter is less obvious. The configuration is a set of "properties" to configure this rule. For example, for an Inspection `FILE_IS_TOO_LONG`, that checks the number of lines in a Kotlin file, the user can configure the maximum number of lines allowed in the file - by changing the "maxSize" in the configuration, or the user can specify paths to folders that do not need to be checked - by writing the path in "ignoreFolders".

7.2 Create ASTNode

Another feature is a special mechanism that allows you to construct an abstract syntax tree node from the text. It is extremely useful for creating automatic fixers, because you do not need to think about the AST implementation and you simply need to provide a text block with a code. Everything will be done under the hood by the framework. This algorithm can parse the code even partially, when you do not need to save the hierarchy of the file (with imports/packages/classes). For example it can parse and provide you a sub-tree for these lines of code:

```
val nodeFromText: ASTNode = KotlinParser().createNode("val age: Int = 21")
```

Listing 5. Example of creating an AST from text of code.



As you can see in the example, we pass the text of the source code, that we want to transform, to the method. What's going on inside this method? First of all, special system properties (used by Kotlin parser) are set (for example: set "idea.io.use.nio2" to true). If the text of the code contains high-level keywords like `import` or `package`, then the method builds a tree with a root node of the FILE type, otherwise it tries with a different root type. In both cases, at the end, if the tree contains a node with type `ERROR_ELEMENT`, it means that some of the code and the method was unable to build the tree and, therefore, throws an exception.
This helps us to implement such complex inspections like the detection of commented code (and distinguish real comments from commented code blocks), helps easily fix the code without manually building sub-trees in visitors.

7.3 Suppress annotation

What if the user wants one of the diKTat Inspections not to check a particular piece of code? The *SUPPRESS* annotation will help us with it. This annotation can be used to ignore a certain Inspection in a certain code block. For instance, if we run this code:

```kotlin
/**
 * This is example
 */

package org.cqfn.diktat

/**
 * Simple class
 */
class User(private val name: String, private val age: Int) {
  /**
   * Function with incorrect name
   *
   * @return is username longer than age
   */
  fun IsInCoRrEcTnAMe() = name.length > age
}
```

Listing 6. Function with incorrect name.

Diktat will raise the warning:

`[FUNCTION_NAME_INCORRECT_CASE] function/method name should be in lowerCamelCase`

But if there is a `@Suppress` before this method, then there will be no warnings during the run:

```kotlin
/**
 * This is example
 */

package org.cqfn.diktat

/**
 * Simple class
 */
@Suppress("FUNCTION_NAME_INCORRECT_CASE")
class User(private val name: String, private val age: Int) {
  /**
   * Function with incorrect name
   *
   * @return is username longer than age
   */
  fun IsInCoRrEcTnAMe() = name.length > age
}
```

Listing 7. Function with incorrect name, but with suppressed Inspection.

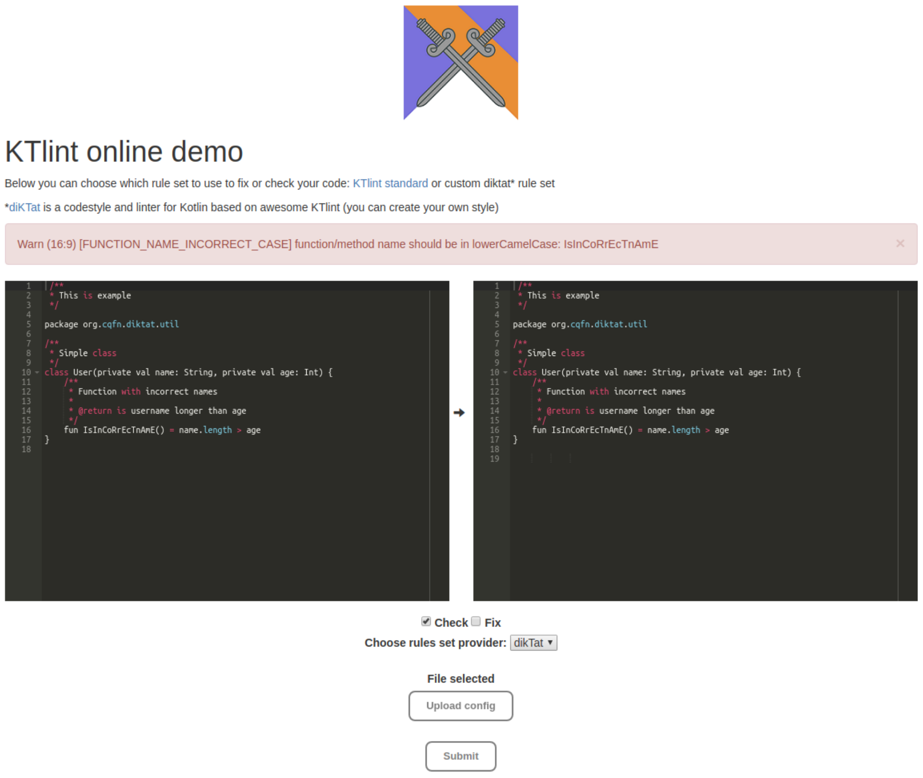
The example shows that the method has a suppress annotation. Therefore, the `FUNCTION_NAME_INCORRECT_CASE` rule will be ignored on this method and there will be no error. The search method for a given annotation goes up recursively to the root element of type `FILE`, looking for the annotation. This means that `@Suppress` can be placed not only in front of knowingly incorrect code, but also at the upper levels of the abstract syntax tree. In our example, the annotation is not in front of the method, but in front of the class and it still works. Also, you can put several annotations:

```
@set:[Suppress("WRONG_DECLARATION_ORDER") Suppress("IDENTIFIER_LENGTH")]
```

Listing 8. Several suppression annotations

7.4 WEB

It worth mentioning that there is a web version of diKTat. This is a handy tool that can be used quickly without any installations, and it is very simple. The link can be found in or you can find it in "How to use diKTat" chapter or in ktlint project as reference.[15]



KTlint online demo

Below you can choose which rule set to use to fix or check your code: KTlint standard or custom diktat* rule set

*diKTat is a codestyle and linter for Kotlin based on awesome KTlint (you can create your own style)

Warn (16:9) [FUNCTION_NAME_INCORRECT_CASE] function/method name should be in lowerCamelCase: IsInCoRrEcTnAmE

```
1  /**
2   * This is example
3   */
4
5  package org.cqfn.diktat.util
6
7  /**
8   * Simple class
9   */
10 class User(private val name: String, private val age: Int) {
11     /**
12      * Function with incorrect names
13      *
14      * @return is username longer than age
15      */
16     fun IsInCoRrEcTnAmE() = name.length > age
17 }
18
```

```
1  /**
2   * This is example
3   */
4
5  package org.cqfn.diktat.util
6
7  /**
8   * Simple class
9   */
10 class User(private val name: String, private val age: Int) {
11     /**
12      * Function with incorrect names
13      *
14      * @return is username longer than age
15      */
16     fun IsInCoRrEcTnAmE() = name.length > age
17 }
18
19
```

Fig. 7. Diktat-web online demo

7.5 Options validation

As it has been mention earlier, diktat has a highly customizable configuration file, but manually editing it is error-prone, for example, name of the rule can be incorrect due to a typo. Diktat will validate configuration file on startup and suggest the closest name based on *Levenshtein* method.

7.6 Self checks

Diktat fully supports self checking of it's own code using both release and snapshot (master) versions. Diktat uses it's self to validate the code inside during it's CI/CD process.

[15]https://github.com/pinterest/ktlint#online-demo

8 HOW TO USE DIKTAT

8.1 CLI-application

You can run diKTat as a CLI-application. To do this you simply need to install ktlint/diktat and to run it via console with a special option `-disabled_rules=standard` that we have introduced in ktlint [16] :

```
$ ./ktlint -R diktat.jar -disabled_rules=standard "path/to/project/**/*.kt"
```

After the run, all detected errors will displayed. Each warning contains of a rule name, a description of the rule and line/column where this error appears in the code. It also will contain `"cannot be auto-corrected"` note if the Inspection does not have autofixer. The format of warning is the following:

/path/to/project/file.kt:6:5: [WARNING_ID_NAME] free text of the warning (cannot be auto-corrected)

Please also note, that as diktat is using ktlint framework - the format of the reported warnings can be changed: it can be xml, json and other formats that are supported by ktlint. Please refer to ktlint documentation [17] to see the information about custom reporters.

8.2 Maven plugin

Maven plugin was introduced for diktat since the version 0.1.3. The following code snippet from `pom.xml` shows how to use diktat with Maven plugin:

```
<plugin>
      <groupId>org.cqfn.diktat</groupId>
      <artifactId>diktat-maven-plugin</artifactId>
      <version>${diktat.version}</version>
      <executions>
          <execution>
              <id>diktat</id>
              <phase>none</phase>
              <goals>
                  <goal>check</goal>
                  <goal>fix</goal>
              </goals>
              <configuration>
                  <inputs>
                      <input>${project.basedir}/src/main/kotlin</input>
                      <input>${project.basedir}/src/test/kotlin</input>
                  </inputs>
                  <diktatConfigFile>diktat-analysis.yml</diktatConfigFile>
                  <excludes>
                     <exclude>${project.basedir}/src/test/kotlin/excluded</exclude>
                  </excludes>
              </configuration>
          </execution>
      </executions>
  </plugin>
```

Listing 9. DiKTat with Maven plugin

[16] https://github.com/pinterest/ktlint/pull/977/files
[17] https://github.com/pinterest/ktlint#creating-a-reporter

To run diktat in only-check mode use the command: `mvn diktat:check@diktat`. To run diktat in autocorrect mode use the command: `mvn diktat:fix@diktat`.

8.3 Gradle plugin

This plugin is available since version 0.1.5. The following code snippet shows how to configure Gradle plugin for diktat.

```
plugins {
    id("org.cqfn.diktat.diktat-gradle-plugin") version "0.1.7"
}
```

Listing 10. DiKTat with Gradle plugin

Or use buildscript syntax:

```

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.cqfn.diktat:diktat-gradle-plugin:0.1.7")
    }
}
apply(plugin = "org.cqfn.diktat.diktat-gradle-plugin")
```

Listing 11. DiKTat with Gradle plugin

Then you can configure diktat using diktat extension:

```
diktat {
    inputs = files("src/**/*.kt")  // file collection that will be checked by diktat
    debug = true  // turn on debug logging
    excludes = files("src/test/kotlin/excluded")  // these files will not be checked by diktat
}
```

Listing 12. DiKTat extension

You can run diktat checks using task `diktatCheck` and automatically fix errors with tasks `diktatFix`.

8.4 Configuratifon file

As described above, diKTat has a configuration file. Note that you should place the *diktat-analysis.yml* file containing the diktat configuration in the parent directory of your project when running as a CLI application. Diktat-maven-plugin and diktat-gradle-plugin have a separate option to setup the path where the configuration file is stored.

8.5 DiKTat-web

The easiest way to use diktat without any downloads or installations is the web version of the app. You can try it by the following link: https://ktlint-demo.herokuapp.com/demo. Web app supports both checking and fixing, using either ktlint or diktat ruleset. For diktat you can also upload a custom configuration file.

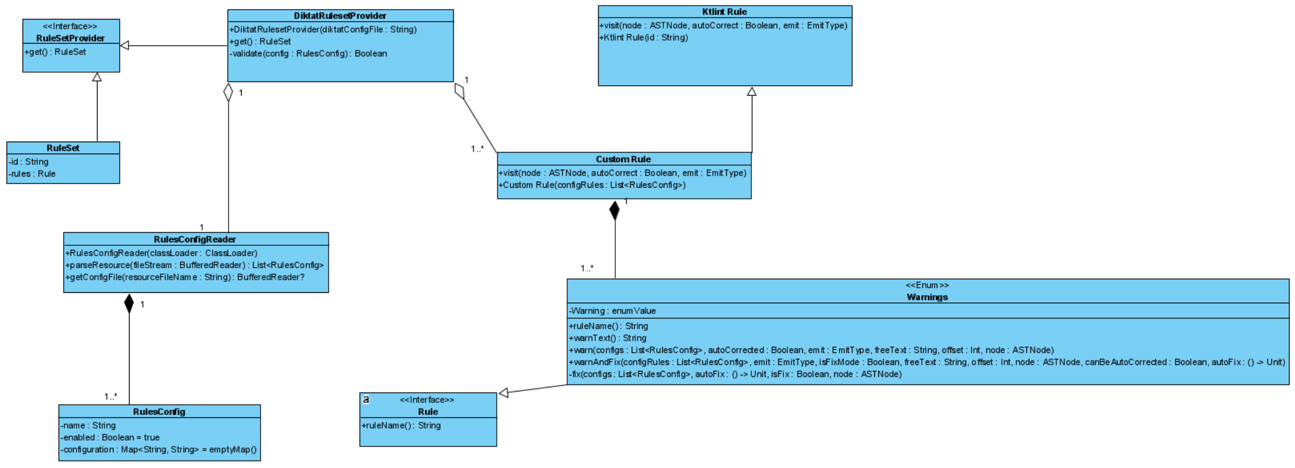
9 CONCLUSION & FUTURE WORK

DiKTat is a static code analyzer that finds and fixes code style inconsistencies. DiKTat is configurable, easy-to-use and it implements CI/CD pipelines, which distinguishes it from analogues. We offer many convenient ways to use diktat in projects, so you can use it as Maven/Gradle plugin, CLI tool, Web or github actions. It supports more than 100 rules, where each of them has clear explanation and can be configured by user. For diktat we have instroduced the coding convention for Kotlin code that now has 6 chapters and will be extended in the future.

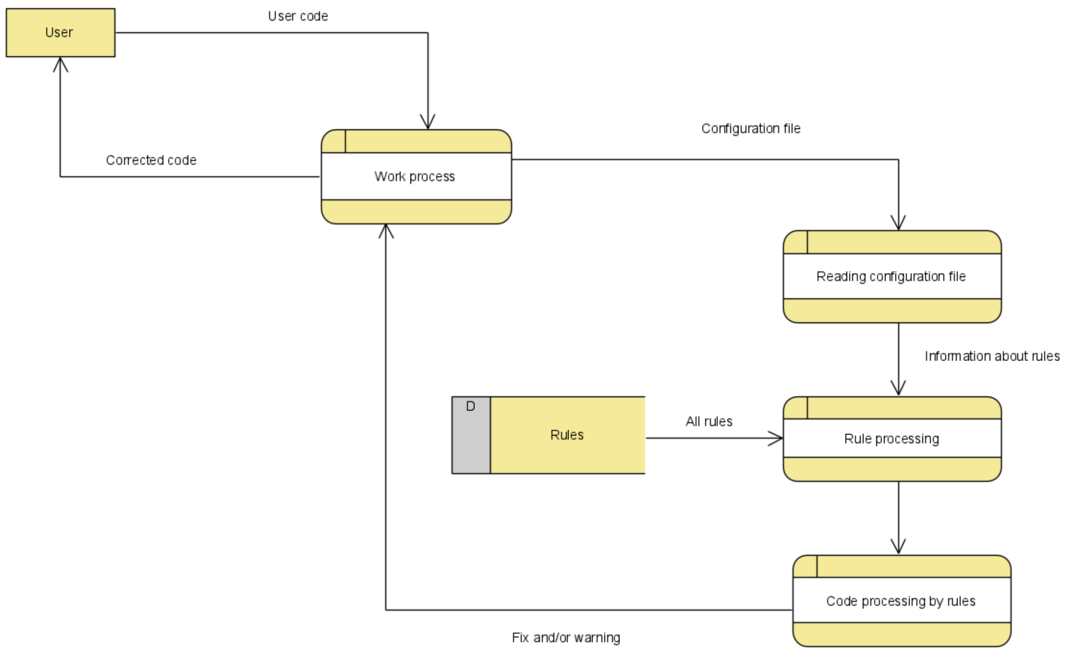
When the development of diKTat will be finished, we are going to support rules, update frameworks and track latest Kotlin releases to keep diKTat up to date. We are also planning to implement some number of Inspections that can detect real bugs in Kotlin code.

10 APPENDIX

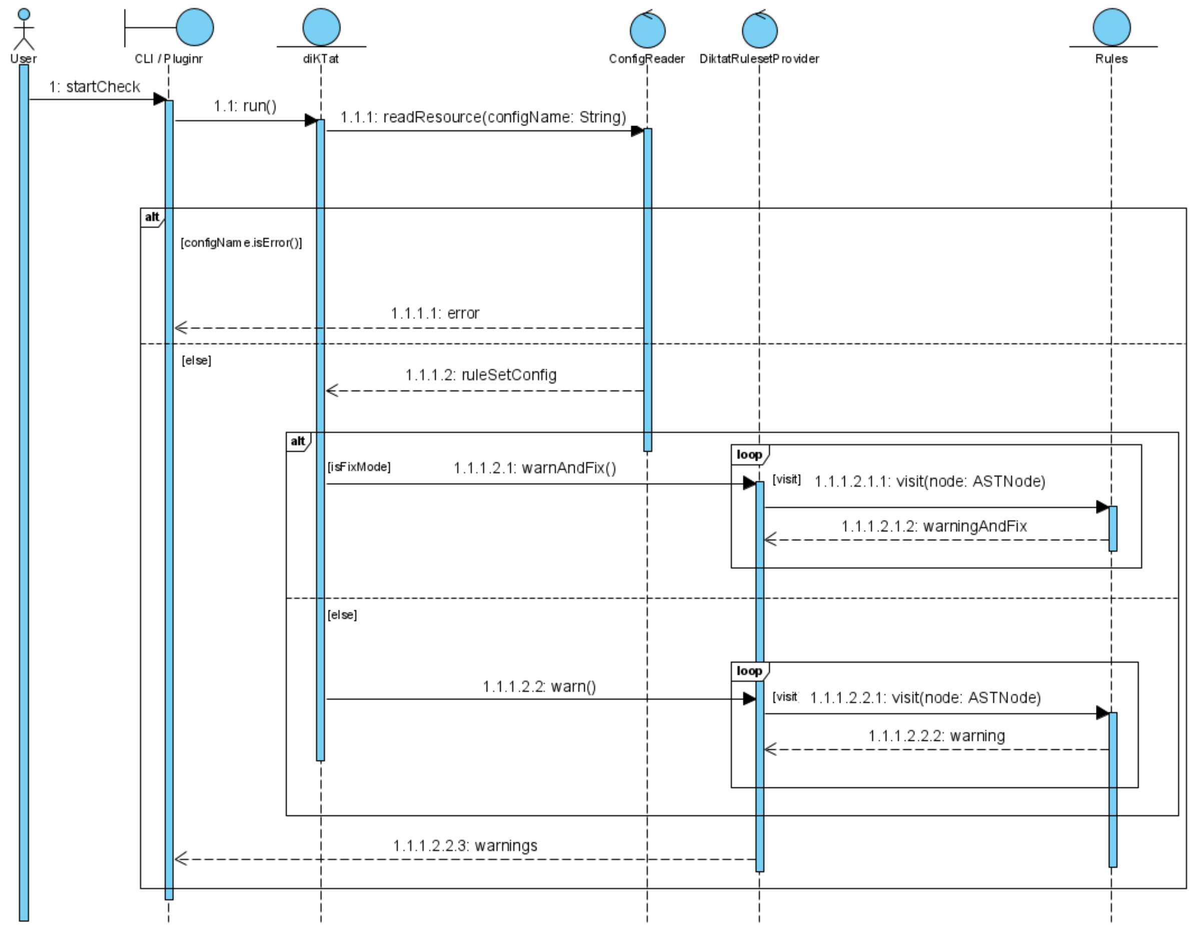
10.1 Class diagram


<<Interface>>
RuleSetProvider
+get() : RuleSet
DiktatRulesetProvider
+DiktatRulesetProvider(diktatConfigFile : String)
+get() : RuleSet
-validate(config : RulesConfig) : Boolean
Ktlint Rule
+visit(node : ASTNode, autoCorrect : Boolean, emit : EmitType)
+Ktlint Rule(id : String)
RuleSet
-id : String
-rules : Rule
Custom Rule
+visit(node : ASTNode, autoCorrect : Boolean, emit : EmitType)
+Custom Rule(configRules : List<RulesConfig>)
RulesConfigReader
+RulesConfigReader(classLoader : ClassLoader)
+parseResource(fileStream : BufferedReader) : List<RulesConfig>
+getConfigFile(resourceFileName : String) : BufferedReader?
<<Enum>>
Warnings
-Warning : enumValue
+ruleName() : String
+warnText() : String
+warn(configs : List<RulesConfig>, autoCorrected : Boolean, emit : EmitType, freeText : String, offset : Int, node : ASTNode)
+warnAndFix(configRules : List<RulesConfig>, emit : EmitType, isFixMode : Boolean, freeText : String, offset : Int, node : ASTNode, canBeAutoCorrected : Boolean, autoFix : () -> Unit)
-fix(configs : List<RulesConfig>, autoFix : () -> Unit, isFix : Boolean, node : ASTNode)
<<Interface>>
Rule
+ruleName() : String
RulesConfig
-name : String
-enabled : Boolean = true
-configuration : Map<String, String> = emptyMap()
1
1..*

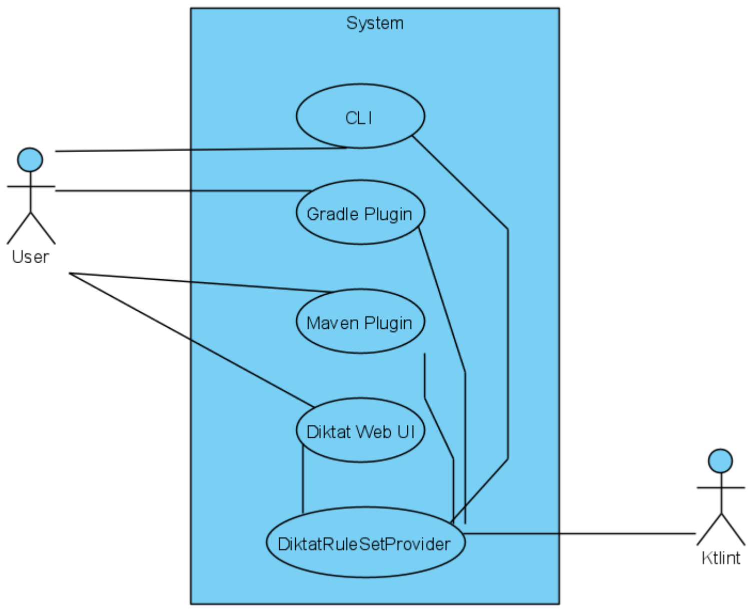
10.2 Data Flow diagram



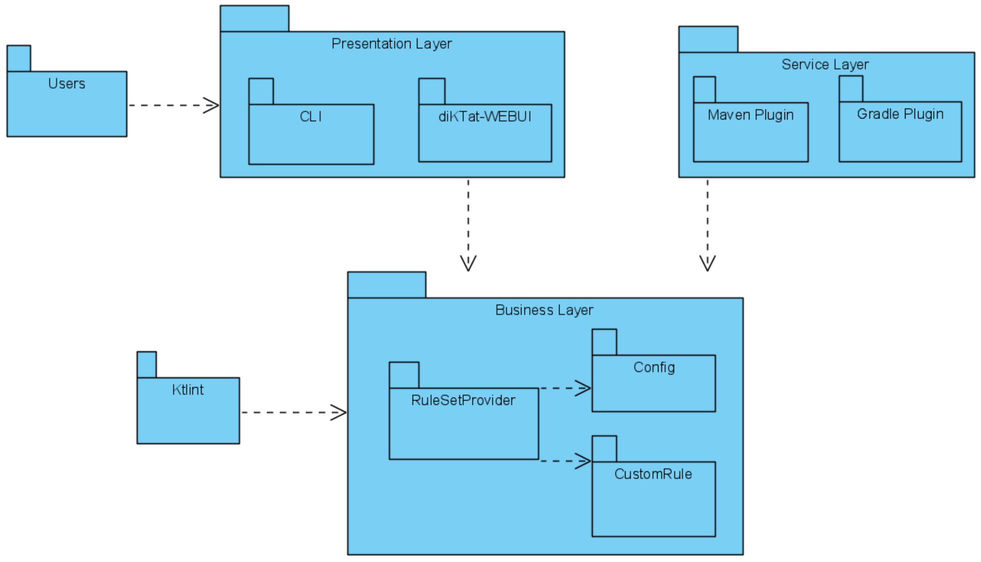
10.3 Sequence diagram



10.4 Use-case diagram



10.5 Package diagram



AVAILABLE RULES

Available Rules

diKTat rule	code style	autofix	config
VARIABLE_NAME_INCORRECT	1.1.1	no	no
VARIABLE_HAS_PREFIX	1.1.1	yes	no
IDENTIFIER_LENGTH	1.1.1	no	no
GENERIC_NAME	1.1.1	yes	no
BACKTICKS_PROHIBITED	1.1.1	no	no
FILE_NAME_INCORRECT	1.1.1	yes	no
EXCEPTION_SUFFIX	1.1.1	yes	no
CONFUSING_IDENTIFIER_NAMING	1.1.1	no	no
PACKAGE_NAME_MISSING	1.2.1	yes	no
PACKAGE_NAME_INCORRECT_CASE	1.2.1	yes	no
PACKAGE_NAME_INCORRECT_PREFIX	1.2.1	yes	no
PACKAGE_NAME_INCORRECT_SYMBOLS	1.2.1	no	no
PACKAGE_NAME_INCORRECT_PATH	1.2.1	yes	no
INCORRECT_PACKAGE_SEPARATOR	1.2.1	yes	no
CLASS_NAME_INCORRECT	1.3.1	yes	no
OBJECT_NAME_INCORRECT	1.3.1	yes	no
ENUM_VALUE	1.3.1	yes	enumStyle: snakeCase, pascalCase
FUNCTION_NAME_INCORRECT_CASE	1.4.1	yes	no
CONSTANT_UPPERCASE	1.5.1	yes	no
VARIABLE_NAME_INCORRECT_FORMAT	1.6.1	yes	no
FUNCTION_BOOLEAN_PREFIX	1.6.2	yes	no
MISSING_KDOC_TOP_LEVEL	2.1.1	no	no
MISSING_KDOC_CLASS_ELEMENTS	2.1.1	no	no
MISSING_KDOC_ON_FUNCTION	2.1.1	yes	no
KDOC_NO_CONSTRUCTOR_PROPERTY	2.1.1	yes	no
KDOC_EXTRA_PROPERTY	2.1.1	no	no
KDOC_NO_CONSTRUCTOR_PROPERTY_WITH_COMMENT	2.1.1	yes	no
KDOC_WITHOUT_PARAM_TAG	2.1.2	yes	no
KDOC_WITHOUT_RETURN_TAG	2.1.2	yes	no
KDOC_WITHOUT_THROWS_TAG	2.1.2	yes	no
KDOC_EMPTY_KDOC	2.1.3	no	no
KDOC_WRONG_SPACES_AFTER_TAG	2.1.3	yes	no
KDOC_WRONG_TAGS_ORDER	2.1.3	yes	no
KDOC_NEWLINES_BEFORE_BASIC_TAGS	2.1.3	yes	no
KDOC_NO_NEWLINES_BETWEEN_BASIC_TAGS	2.1.3	yes	no
KDOC_NO_NEWLINE_AFTER_SPECIAL_TAGS	2.1.3	yes	no
KDOC_NO_DEPRECATED_TAG	2.1.3	yes	no
KDOC_CONTAINS_DATE_OR_AUTHOR	2.1.3	no	no
KDOC_NO_EMPTY_TAGS	2.2.1	no	no
HEADER_WRONG_FORMAT	2.2.1	yes	no
HEADER_MISSING_OR_WRONG_COPYRIGHT	2.2.1	yes	mandatoryCopyright
WRONG_COPYRIGHT_YEAR	2.2.1	yes	no
HEADER_MISSING_IN_NON_SINGLE_CLASS_FILE	2.2.1	no	no
HEADER_NOT_BEFORE_PACKAGE	2.2.1	yes	no
KDOC_TRIVIAL_KDOC_ON_FUNCTION	2.3.1	no	no
WRONG_NEWLINES_AROUND_KDOC	2.4.1	yes	no
FIRST_COMMENT_NO_SPACES	2.4.1	yes	no
COMMENT_WHITE_SPACE	2.4.1	yes	maxSpaces
IF_ELSE_COMMENTS	2.4.1	yes	no
COMMENTED_OUT_CODE	2.4.2	no	no
FILE_IS_TOO_LONG	3.1.1	no	maxSize ignoreFolders
FILE_CONTAINS_ONLY_COMMENTS	3.1.2	no	no
FILE_INCORRECT_BLOCKS_ORDER	3.1.2	yes	no
FILE_NO_BLANK_LINE_BETWEEN_BLOCKS	3.1.2	yes	no
FILE_UNORDERED_IMPORTS	3.1.2	yes	no
FILE_WILDCARD_IMPORTS	3.1.2	no	allowedWildcards
FILE_NAME_MATCH_CLASS	3.1.2	yes	no
WRONG_ORDER_IN_CLASS_LIKE_STRUCTURES	3.1.4	yes	no
BLANK_LINE_BETWEEN_PROPERTIES	3.1.4	yes	no
WRONG_DECLARATIONS_ORDER	3.1.4	yes	no
NO_BRACES_IN_CONDITIONALS_AND_LOOPS	3.2.1	yes	no

BRACES_BLOCK_STRUCTURE_ERROR	3.2.2	yes	openBraceNewline closeBraceNewline
WRONG_INDENTATION	3.3.1	yes	extendedIndentOfParameters alignedParameters extendedIndentAfterOperators extendedIndentBeforeDot indentationSize
EMPTY_BLOCK_STRUCTURE_ERROR	3.4.1	yes	allowEmptyBlocks styleEmptyBlockWithNewline
LONG_LINE	3.5.1	yes	lineLength
MORE_THAN_ONE_STATEMENT_PER_LINE	3.6.1	yes	no
REDUNDANT_SEMICOLON	3.6.2	yes	no
WRONG_NEWLINES	3.6.2	yes	no
TOO_MANY_BLANK_LINES	3.7.1	yes	no
WRONG_WHITESPACE	3.8.1	yes	no
TOO_MANY_CONSECUTIVE_SPACES	3.8.1	yes	maxSpaces saveInitialFormattingForEnums
ENUMS_SEPARATED	3.9.1	yes	no
LOCAL_VARIABLE_EARLY_DECLARATION	3.10.2	no	no
WHEN_WITHOUT_ELSE	3.11.1	yes	no
ANNOTATION_NEW_LINE	3.12.1	yes	no
WRONG_MULTIPLE_MODIFIERS_ORDER	3.14.1	yes	no
LONG_NUMERICAL_VALUES_SEPARATED	3.14.2	yes	maxNumberLength maxBlockLength
STRING_CONCATENATION	3.15.1	no	no
STRING_TEMPLATE_CURLY_BRACES	3.15.2	yes	no
STRING_TEMPLATE_QUOTES	3.15.2	yes	no
FLOAT_IN_ACCURATE_CALCULATIONS	4.1.1	no	no
SAY_NO_TO_VAR	4.1.3	no	no
SMART_CAST_NEEDED	4.2.1	yes	no
TYPE_ALIAS	4.2.2	no	typeReferenceLength
NULLABLE_PROPERTY_TYPE	4.3.1	yes	no
GENERIC_VARIABLE_WRONG_DECLARATION	4.3.2	yes	no
AVOID_NULL_CHECKS	4.3.3	no	no
TOO_LONG_FUNCTION	5.1.1	no	maxFunctionLength isIncludeHeader
NESTED_BLOCK	5.1.2	no	maxNestedBlockQuantit
AVOID_NESTED_FUNCTIONS	5.1.3	yes	no
INVERSE_FUNCTION_PREFERRED	5.1.4	yes	-
LAMBDA_IS_NOT_LAST_PARAMETER	5.2.1	no	no
TOO_MANY_PARAMETERS	5.2.2	no	maxParameterListSize
WRONG_OVERLOADING_FUNCTION_ARGUMENTS	5.2.3	no	no
SINGLE_CONSTRUCTOR_SHOULD_BE_PRIMARY	6.1.1	yes	no
USE_DATA_CLASS	6.1.2	no	no
EMPTY_PRIMARY_CONSTRUCTOR	6.1.3	yes	no
MULTIPLE_INIT_BLOCKS	6.1.4	yes	no
USELESS_SUPERTYPE	6.1.5	yes	no
CLASS_SHOULD_NOT_BE_ABSTRACT	6.1.6	yes	no
NO_CORRESPONDING_PROPERTY	6.1.7	no	no
CUSTOM_GETTERS_SETTERS	6.1.8	no	no
WRONG_NAME_OF_VARIABLE_INSIDE_ACCESSOR	6.1.9	no	no
TRIVIAL_ACCESSORS_ARE_NOT_RECOMMENDED	6.1.10	yes	no
COMPACT_OBJECT_INITIALIZATION	6.1.11	yes	no
EXTENSION_FUNCTION_SAME_SIGNATURE	6.2.2	no	no
AVOID_USING_UTILITY_CLASS	6.4.1	no	no
OBJECT_IS_PREFERRED	6.4.2	yes	no

DIKTAT CODING STYLE GUIDE, V.0.0.1

TABLE OF CONTENTS

DIKTAT CODING STYLE GUIDE
INTERNATIONAL VERSION, V.0.0.1

<img src="logo.svg" width="64px"/>

Purpose of this document

The purpose of this document is to provide a specification that software developers could reference to enhance their ability to write consistent, easy-to-read, and high-quality code.

Such a specification will ultimately improve software development efficiency and product competitiveness.

For the code to be considered high-quality, it must entail the following characteristics:

1. Simplicity
2. Maintainability
3. Reliability
4. Testability
5. Efficiency
6. Portability
7. Reusability

General principles

Like other modern programming languages, Kotlin is an advanced programming language that complies with the following general principles:

1. Clarity — a necessary feature of programs that are easy to maintain and refactor.
2. Simplicity — a code is easy to understand and implement.
3. Consistency — enables a code to be easily modified, reviewed, and understood by the team members. Unification is particularly important when the same team works on the same project, utilizing similar styles enabling a code to be easily modified, reviewed, and understood by the team members.

Also, we need to consider the following factors when programming on Kotlin:

1. Writing clean and simple Kotlin code

Kotlin combines two of the main programming paradigms: functional and object-oriented.
Both of these paradigms are trusted and well-known software engineering practices.
As a young programming language, Kotlin is built on top of well-established languages such as Java, C++, C#, and Scala.
This enables Kotlin to introduce many features that help a developer write cleaner, more readable code while also reducing the number of complex code structures. For example, type and null safety, extension functions, infix syntax, immutability, val/var differentiation, expression-oriented features, "when" statements, much easier work with collections, type auto conversion, and other syntactic sugar.
2. Following Kotlin idioms
The author of Kotlin, Andrey Breslav, mentioned that Kotlin is both pragmatic and practical, but not academic.
Its pragmatic features enable ideas to be transformed into real working software easily. Kotlin is closer to natural languages than its predecessors, and it implements the following design principles: readability, reusability, interoperability, security, and tool-friendliness (https://blog.jetbrains.com/kotlin/2018/10/kotlinconf-2018-announcements/).
3. Using Kotlin efficiently
Some Kotlin features can help you to write higher-performance code: including rich coroutine library, sequences, inline functions/classes, arrays of basic types, tailRec, and CallsInPlace of contract.

Terminology

Rules — conventions that should be followed when programming.
Recommendations — conventions that should be considered when programming.
Explanation — necessary explanations of rules and recommendations.
Valid Example — recommended examples of rules and recommendations.
Invalid Example — not recommended examples of rules and recommendations.
Unless otherwise stated, this specification applies to versions 1.3 and later of Kotlin.

Exceptions

Even though exceptions may exist, it is essential to understand why rules and recommendations are needed.
Depending on a project situation or personal habits, you can break some of the rules. However, remember that one exception may lead to many and eventually can destroy code consistency. As such, there should be very few exceptions.
When modifying open-source code or third-party code, you can choose to use the code style from this open-source project (instead of using the existing specifications) to maintain consistency.
Software that is directly based on the Android native operating system interface, such as the Android Framework, remains consistent with the Android style.

1. NAMING

In programming, it is not always easy to meaningfully and appropriately name variables, functions, classes, etc. Using meaningful names helps to clearly express your code's main ideas and functionality and avoid misinterpretation, unnecessary coding and decoding, "magic" numbers, and inappropriate abbreviations.
Note: The source file encoding format (including comments) must be UTF-8 only. The ASCII horizontal space character (0x20, that is, space) is the only permitted whitespace character. Tabs should not be used for indentation.

1.1 Identifiers

This section describes the general rules for naming identifiers.

1.1.1 Identifiers naming conventions.

For identifiers, use the following naming conventions:
1. All identifiers should use only ASCII letters or digits, and the names should match regular expressions **\w{2,64}**.
Explanation: Each valid identifier name should match the regular expression **\w{2,64}**.
{2,64} means that the name length is 2 to 64 characters, and the length of the variable name should be proportional to its life range, functionality, and responsibility.
Name lengths of less than 31 characters are generally recommended. However, this depends on the project. Otherwise, a class declaration with generics or inheritance from a superclass can cause line breaking.
No special prefix or suffix should be used in names. The following examples are inappropriate names: name_, mName, s_name, and kName.
2. Choose file names that would describe the content. Use camel case (PascalCase) and **.kt** extension.
3. Typical examples of naming:

Meaning	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

4. The usage of () and free naming for functions and identifiers are prohibited. For example, the following code is not recommended:

```kotlin
val `my dummy name-with-minus` = "value"
```

The only exception is function names in **Unit tests.**
5. Backticks () should not be used for identifiers, except the names of test methods (marked with @Test annotation):

```
@Test fun `my test`() { /*...*/ }
```

6. The following table contains some characters that may cause confusion. Be careful when using them as identifiers. To avoid issues, use other names instead.

Expected	Confusing name	Suggested name
0 (zero)	O, D	obj, dgt
1 (one)	I, l	it, ln, line
2 (two)	Z	n1, n2
5 (five)	S	xs, str
6 (six)	e	ex, elm
8 (eight)	B	bt, nxt
n,h	h,n	nr, head, height
rn, m	m,rn	mbr, item

Exceptions:
- The i,j,k variables used in loops are part of the industry standard. One symbol can be used for such variables.
- The **e** variable can be used to catch exceptions in catch block: **catch (e: Exception) {}**
- The Java community generally does not recommend the use of prefixes. However, when developing Android code, you can use the s and m prefixes for static and non-public non-static fields, respectively.

Note that prefixing can also negatively affect the style and the auto-generation of getters and setters.

Type	Naming style
Interfaces, classes, annotations, enumerated types, and object type names	Camel case, starting with a capital letter. Test classes have a Test suffix. The filename is 'TopClassName'.kt.
Class fields, local variables, methods, and method parameters	Camel case starting with a low case letter. Test methods can be underlined with '_'; the only exception is [backing properties]
Static constants and enumerated values	Only uppercase underlined with '_'
Generic type variable	Single capital letter, which can be followed by a number, for example: ‘E, T, U, X, T2‘
Exceptions	Same as class names, but with a suffix Exception, for example: ‘AccessException‘ and ‘NullPointerException‘

1.2 Packages

Rule 1.2.1 Package names dots.

Package names are in lower case and separated by dots. Code developed within your company should start with **your.company.domain.** Numbers are permitted in package names.

Each file should have a **package** directive.

Package names are all written in lowercase, and consecutive words are concatenated together (no underscores). Package names should contain both the product or module names and the department (or team) name to prevent conflicts with other teams. Numbers are not permitted. For example: **org.apache.commons.lang3**, **xxx.yyy.v2**.

Exceptions:
- In certain cases, such as open-source projects or commercial cooperation, package names should not start with **your.company.domain.**
- If the package name starts with a number or other character that cannot be used at the beginning of the Java/Kotlin package name, then underscores are allowed. For example: **com.example._123name**.
- Underscores are sometimes permitted if the package name contains reserved Java/Kotlin keywords, such as **org.example.hyphenated_name**, **int_.example**.

Valid example:

```
package your.company.domain.mobilecontrol.views
```

1.3 Classes

This section describes the general rules for naming classes, enumerations, and interfaces.

1.3.1 Classes.

Classes, enumerations, and interface names use **UpperCamelCase** nomenclature. Follow the naming rules described below:

1. A class name is usually a noun (or a noun phrase) denoted using the camel case nomenclature, such as UpperCamelCase. For example: **Character** or **ImmutableList**.

An interface name can also be a noun or noun phrase (such as **List**) or an adjective or adjective phrase (such as **Readable**).

Note that verbs are not used to name classes. However, nouns (such as **Customer**, **WikiPage**, and **Account**) can be used. Try to avoid using vague words such as **Manager** and **Process**.

2. Test classes start with the name of the class they are testing and end with 'Test'. For example, **HashTest** or **HashIntegrationTest**.

Invalid example:

```
class marcoPolo {}
class XMLService {}
interface TAPromotion {}
class info {}
```

Valid example:

```
class MarcoPolo {}
class XmlService {}
interface TaPromotion {}
class Order {}
```

1.4 Functions

This section describes the general rules for naming functions.

1.4.1 Function names should be in camel case.

Function names should use **lowerCamelCase** nomenclature. Follow the naming rules described below:
1. Function names are usually verbs or verb phrases denoted with the camel case nomenclature (**lowerCamelCase**).
For example: **sendMessage**, **stopProcess**, or **calculateValue**.
To name functions, use the following formatting rules:
a) To get, modify, or calculate a certain value: get + non-boolean field(). Note that the Kotlin compiler automatically generates getters for some classes, applying the special syntax preferred for the 'get' fields: kotlin private val field: String get() { }. kotlin private val field: String get() { }.

```
private val field: String
get() {
}
```

Note: The calling property access syntax is preferred to call getter directly. In this case, the Kotlin compiler automatically calls the corresponding getter.
b) **is** + boolean variable name()
c) **set** + field/attribute name(). However, note that the syntax and code generation for Kotlin are completely the same as those described for the getters in point a.
d) **has** + Noun / adjective ()
e) verb()
Note: Note: Verb are primarily used for the action objects, such as **document.print ()**
f) verb + noun()
g) The Callback function allows the names that use the preposition + verb format, such as: **onCreate()**, **onDestroy()**, **toString()**.
Invalid example:

```
fun type(): String
fun Finished(): Boolean
fun visible(boolean)
fun DRAW()
fun KeyListener(Listener)
```

Valid example:

```
fun getType(): String
fun isFinished(): Boolean
fun setVisible(boolean)
fun draw()
fun addKeyListener(Listener)
```

2. An underscore (_) can be included in the JUnit test function name and should be used as a separator. Each logical part is denoted in **lowerCamelCase**, for example, a typical pattern of using underscore: **pop_emptyStack**.

1.5 Constants

This section describes the general rules for naming constraints.

1.5.1 Using UPPER case and underscore characters in a constraint name.

Constant names should be in UPPER case, words separated by underscore. The general constant naming conventions are listed below:
1. Constants are attributes created with the **const** keyword or top-level/**val** local variables of an object that holds immutable data. In most cases, constants can be identified as a **const val** property from the **object**/**companion object**/file top level. These variables contain fixed constant values that typically should never be changed by programmers. This includes basic types, strings, immutable types, and immutable collections of immutable types. The value is not constant for the object, which state can be changed.
2. Constant names should contain only uppercase letters separated by an underscores. They should have a val or const val modifier to make them final explicitly. In most cases, if you need to specify a constant value, then you need to create it with the "const val" modifier. Note that not all **val** variables are constants.
3. Objects with immutable content, such as **Logger** and **Lock**, can be in uppercase as constants or have camel case as regular variables.

4. Use meaningful constants instead of **magic numbers**. SQL or logging strings should not be treated as magic numbers, nor should they be defined as string constants.

Magic constants, such as **NUM_FIVE** = **5** or **NUM_5** = **5** should not be treated as constants. This is because mistakes will easily be made if they are changed to **NUM_5** = **50** or 55.

These constants typically represent business logic values, such as measures, capacity, scope, location, tax rate, promotional discounts, and power base multiples in algorithms.

You can avoid using magic numbers with the following method:

- Using library functions and APIs. For example, instead of checking that **size** == **0**, use **isEmpty()** function. To work with **time**, use built-ins from **java.time API**.

- Enumerations can be used to name patterns. Refer to [Recommended usage scenario for enumeration in 3.9].

Invalid example:

```
var int MAXUSERNUM = 200;
val String sL = "Launcher";
```

Valid example:

```
const val int MAX_USER_NUM = 200;
const val String APPLICATION_NAME = "Launcher";
```

1.6 Non-constant fields

This section describes the general rules for naming variables.

1.6.1 Non-constant field name.

Non-constant field names should use camel case and start with a lowercase letter.

A local variable cannot be treated as constant even if it is final and immutable. Therefore, it should not use the preceding rules. Names of collection type variables (sets, lists, etc.) should contain plural nouns.

For example: **var namesList: List<String>**

Names of non-constant variables should use **lowerCamelCase**. The name of the final immutable field used to store the singleton object can use the same camel case notation.

Invalid example:

```
customername: String
user: List<String> = listof()
```

Valid example:

```
var customerName: String
val users: List<String> = listOf();
val mutableCollection: MutableSet<String> = HashSet()
```

1.6.2 Boolean variable names with negative meaning.

Avoid using Boolean variable names with a negative meaning. When using a logical operator and name with a negative meaning, the code may be difficult to understand, which is referred to as the "double negative".

For instance, it is not easy to understand the meaning of !isNotError.

The JavaBeans specification automatically generates isXxx() getters for attributes of Boolean classes.

However, not all methods returning Boolean type have this notation.

For Boolean local variables or methods, it is highly recommended that you add non-meaningful prefixes, including is (commonly used by JavaBeans), has, can, should, and must. Modern integrated development environments (IDEs) such as Intellij are already capable of doing this for you when you generate getters in Java. For Kotlin, this process is even more straightforward as everything is on the byte-code level under the hood.

Invalid example:

```
val isNoError: Boolean
val isNotFound: Boolean
fun empty()
fun next();
```

Valid example:

```
val isError: Boolean
val isFound: Boolean
val hasLicense: Boolean
val canEvaluate: Boolean
val shouldAbort: Boolean
```

```kotlin
fun isEmpty()
fun hasNext()
```

2. COMMENTS

The best practice is to begin your code with a summary, which can be one sentence.

Try to balance between writing no comments at all and obvious commentary statements for each line of code.

Comments should be accurately and clearly expressed, without repeating the name of the class, interface, or method.

Comments are not a solution to the wrong code. Instead, you should fix the code as soon as you notice an issue or plan to fix it (by entering a TODO comment, including a Jira number).

Comments should accurately reflect the code's design ideas and logic and further describe its business logic.

As a result, other programmers will be able to save time when trying to understand the code.

Imagine that you are writing the comments to help yourself to understand the original ideas behind the code in the future.

2.1 General form of Kdoc

KDoc is a combination of JavaDoc's block tags syntax (extended to support specific constructions of Kotlin) and Markdown's inline markup.

The basic format of KDoc is shown in the following example:

```kotlin
 /**
 * There are multiple lines of KDoc text,
 * Other ...
 */
fun method(arg: String) {
    // ...
}
```

It is also shown in the following single-line form:

```kotlin
 /** Short form of KDoc. */
```

Use a single-line form when you store the entire KDoc block in one line (and there is no KDoc mark @XXX). For detailed instructions on how to use KDoc, refer to Official Document.

2.1.1 Using KDoc for the public.

At a minimum, KDoc should be used for every public, protected, or internal decorated class, interface, enumeration, method, and member field (property).
Other code blocks can also have KDocs if needed.
Instead of using comments or KDocs before properties in the primary constructor of a class - use **@property** tag in a KDoc of a class.
All properties of the primary constructor should also be documented in a KDoc with a **@property** tag.
Incorrect example:

```kotlin
/**
 * Class description
 */
class Example(
 /**
  * property description
  */
  val foo: Foo,
  // another property description
  val bar: Bar
)
```

Correct example:

```kotlin
/**
 * Class description
 * @property foo property description
 * @property bar another property description
 */
class Example(
  val foo: Foo,
  val bar: Bar
)
```

Exceptions:
* For setters/getters of properties, obvious comments (like **this getter returns field**) are optional. Note that Kotlin generates simple **get/set** methods under the hood.
* It is optional to add comments for simple one-line methods, such as shown in the example below:

```
val isEmpty: Boolean
    get() = this.size == 0
```

or

```
fun isEmptyList(list: List<String>) = list.size == 0
```

Note: You can skip KDocs for a method's override if it is almost the same as the superclass method.

2.1.2 Describing methods that have arguments.

When the method has such details as arguments, return value, or can throw exceptions, it must be described in the KDoc block (with @param, @return, @throws, etc.).
Valid examples:

```
/**
 * This is the short overview comment for the example interface.
 *     / * Add a blank line between the comment text and each KDoc tag underneath * /
 * @since 1.6
 */
 protected abstract class Sample {
    /**
     * This is a long comment with whitespace that should be split in
     * comments on multiple lines if the line comment formatting is enabled.
     *     / * Add a blank line between the comment text and each KDoc tag underneath * /
     * @param fox A quick brown fox jumps over the lazy dog
     * @return battle between fox and dog
     */
    protected abstract fun foo(Fox fox)
     /**
      * These possibilities include: Formatting of header comments
      *     / * Add a blank line between the comment text and each KDoc tag underneath * /
      * @return battle between fox and dog
      * @throws ProblemException if lazy dog wins
      */
    protected fun bar() throws ProblemException {
        // Some comments / * No need to add a blank line here * /
        var aVar = ...

        // Some comments  / * Add a blank line before the comment * /
        fun doSome()
    }
 }
```

2.1.3 Only one space between the Kdoc tag and content. Tags are arranged in the order.

There should be only one space between the Kdoc tag and content. Tags are arranged in the following order: @param, @return, and @throws.
Therefore, Kdoc should contain the following:
- Functional and technical description, explaining the principles, intentions, contracts, API, etc.
- The function description and @tags (**implSpec**, **apiNote**, and **implNote**) require an empty line after them.

- **@implSpec**: A specification related to API implementation, and it should let the implementer decide whether to override it.
- **@apiNote**: Explain the API precautions, including whether to allow null and whether the method is thread-safe, as well as the algorithm complexity, input, and output range, exceptions, etc.
- **@implNote**: A note related to API implementation, which implementers should keep in mind.
- One empty line, followed by regular **@param**, **@return**, **@throws**, and other comments.
- The conventional standard "block labels" are arranged in the following order: **@param**, **@return**, **@throws**.

Kdoc should not contain:
- Empty descriptions in tag blocks. It is better not to write Kdoc than waste code line space.
- There should be no empty lines between the method/class declaration and the end of Kdoc (*/ symbols).
- **@author** tag. It doesn't matter who originally created a class when you can use **git blame** or VCS of your choice to look through the changes history.

Important notes:
- KDoc does not support the **@deprecated** tag. Instead, use the **@Deprecated** annotation.
- The **@since** tag should be used for versions only. Do not use dates in **@since** tag, it's confusing and less accurate.

If a tag block cannot be described in one line, indent the content of the new line by *four spaces* from the **@** position to achieve alignment (**@** counts as one + three spaces).

Exception:

When the descriptive text in a tag block is too long to wrap, you can indent the alignment with the descriptive text in the last line. The descriptive text of multiple tags does not need to be aligned.

See [3.8 Horizontal space].

In Kotlin, compared to Java, you can put several classes inside one file, so each class should have a Kdoc formatted comment (as stated in rule 2.1).

This comment should contain @since tag. The right style is to write the application version when its functionality is released. It should be entered after the **@since** tag.

Examples:

```
/**
 * Description of functionality
 *
 * @since 1.6
 */
```

Other KDoc tags (such as @param type parameters and @see.) can be added as follows:

```
/**
 * Description of functionality
 *
 * @apiNote: Important information about API
 *
 * @since 1.6
 */
```

2.2 Adding comments on the file header

This section describes the general rules of adding comments on the file header.

2.2.1 Formatting of comments in the file header.

Comments on the file header should be placed before the package name and imports. If you need to add more content to the comment, subsequently add it in the same format.

Comments on the file header must include copyright information, without the creation date and author's name (use VCS for history management).

Also, describe the content inside files that contain multiple or no classes.

The following examples for Huawei describe the format of the *copyright license*:

Chinese version: 版权所有**(c)** 华为技术有限公司**2012-2020**

English version: **Copyright (c) Huawei Technologies Co., Ltd. 2012-2020. All rights reserved.**

2012 and **2020** are the years the file was first created and the current year, respectively.

Do not place **release notes** in header, use VCS to keep track of changes in file. Notable changes can be marked in individual KDocs using **@since** tag with version.

Invalid example:

```
/**
 * Release notes:
 * 2019-10-11: added class Foo
 */

class Foo
```

Valid example:

```
/**
 * @since 2.4.0
 */
class Foo
```

- The **copyright statement** can use your company's subsidiaries, as shown in the below examples:
Chinese version: 版权所有**(c)** 海思半导体**2012-2020**
English version: **Copyright (c) Hisilicon Technologies Co., Ltd. 2012-2020. All rights reserved.**
- The copyright information should not be written in KDoc style or use single-line comments. It must start from the beginning of the file.
The following example is a copyright statement for Huawei, without other functional comments:

```
/*
 * Copyright (c) Huawei Technologies Co., Ltd. 2012-2020. All rights reserved.
 */
```

The following factors should be considered when writing the file header or comments for top-level classes:
- File header comments must start from the top of the file. If it is a top-level file comment, there should be a blank line after the last Kdoc */ symbol. If it is a comment for a top-level class, the class declaration should start immediately without using a newline.
- Maintain a unified format. The specific format can be formulated by the project (for example, if you use an existing opensource project), and you need to follow it.
- A top-level file-Kdoc must include a copyright and functional description, especially if there is more than one top-level class.
- Do not include empty comment blocks. If there is no content after the option **@apiNote**, the entire tag block should be deleted.
- The industry practice is not to include historical information in the comments. The corresponding history can be found in VCS (git, svn, etc.). Therefore, it is not recommended to include historical data in the comments of the Kotlin source code.

2.3 Comments on the function header

Comments on the function header are placed above function declarations or definitions. A newline should not exist between a function declaration and its Kdoc. Use the preceding «c2.1,KDoc» style rules.

As stated in Chapter 1, the function name should reflect its functionality as much as possible. Therefore, in the Kdoc, try to describe the functionality that is not mentioned in the function name.

Avoid unnecessary comments on dummy coding.

The function header comment's content is optional, but not limited to function description, return value, performance constraints, usage, memory conventions, algorithm implementation, reentrant requirements, etc.

2.4 Code comments

This section describes the general rules of adding code comments.

2.4.1 Add a blank line between the body of the comment and Kdoc tag-blocks.

It is a good practice to add a blank line between the body of the comment and Kdoc tag-blocks. Also, consider the following rules:
- There must be one space between the comment character and the content of the comment.
- There must be a newline between a Kdoc and the presiding code.
- An empty line should not exist between a Kdoc and the code it is describing. You do not need to add a blank line before the first comment in a particular namespace (code block) (for example, between the function declaration and first comment in a function body).

Valid Examples:

```
/**
 * This is the short overview comment for the example interface.
 *
 * @since 1.6
 */
 public interface Example {
    // Some comments  /* Since it is the first member definition in this code block, there is
    no need to add a blank line here */
    val aField: String = ...
                     /* Add a blank line above the comment */
    // Some comments
    val bField: String = ...
                      /* Add a blank line above the comment */
    /**
    * This is a long comment with whitespace that should be split in
    * multiple line comments in case the line comment formatting is enabled.
    *                /* blank line between description and Kdoc tag */
    * @param fox A quick brown fox jumps over the lazy dog
    * @return the rounds of battle of fox and dog
    */
   fun foo(Fox fox)
                     /* Add a blank line above the comment */
    /**
     * These possibilities include: Formatting of header comments
     *
     * @return the rounds of battle of fox and dog
     * @throws ProblemException if lazy dog wins
     */
   fun bar() throws ProblemException {
       // Some comments  /* Since it is the first member definition in this range, there is
   no need to add a blank line here */
       var aVar = ...

       // Some comments  /* Add a blank line above the comment */
       fun doSome()
   }
}
```

- Leave one single space between the comment on the right side of the code and the code.

If you use conditional comments in the **if-else-if** scenario, put the comments inside the **else-if** branch or in the conditional block, but not before the **else-if**. This makes the code more understandable.

When the if-block is used with curly braces, the comment should be placed on the next line after opening the curly braces.

Compared to Java, the **if** statement in Kotlin statements returns a value. For this reason, a comment block can describe a whole **if-statement**.

Valid examples:

```

val foo = 100  // right-side comment
val bar = 200  /* right-side comment */

// general comment for the value and whole if-else condition
val someVal = if (nr % 15 == 0) {
    // when nr is a multiple of both 3 and 5
    println("fizzbuzz")
} else if (nr % 3 == 0) {
    // when nr is a multiple of 3, but not 5
    // We print "fizz", only.
    println("fizz")
} else if (nr % 5 == 0) {
    // when nr is a multiple of 5, but not 3
    // we print "buzz" only.
    println("buzz")
} else {
    // otherwise, we print the number.
    println(x)
}
```

- Start all comments (including KDoc) with a space after the first symbol (//, /*, / and *)

Valid example:

```
val x = 0  // this is a comment
```

2.4.2 Do not comment on unused code blocks.

Do not comment on unused code blocks, including imports. Delete these code blocks immediately.
A code is not used to store history. Git, svn, or other VCS tools should be used for this purpose.
Unused imports increase the coupling of the code and are not conducive to maintenance. The commented out code cannot be appropriately maintained.
In an attempt to reuse the code, there is a high probability that you will introduce defects that are easily missed.
The correct approach is to delete the unnecessary code directly and immediately when it is not used anymore.
If you need the code again, consider porting or rewriting it as changes could have occurred since you first commented on the code.

2.4.3 Code delivered to the client should not contain TODO.

The code officially delivered to the client typically should not contain TODO/FIXME comments.
TODO comments are typically used to describe modification points that need to be improved and added. For example, refactoring FIXME comments are typically used to describe known defects and bugs that will be subsequently fixed and are not critical for an application.
They should all have a unified style to facilitate unified text search processing.
Example:

```
// TODO(<author-name>): Jira-XXX - support new json format
// FIXME: Jira-XXX - fix NPE in this code block
```

At a version development stage, these annotations can be used to highlight the issues in the code, but all of them should be fixed before a new product version is released.

3. GENERAL FORMATTING

3.1 File-related rules

This section describes the rules related to using files in your code.

3.1.1 Avoid files that are too long.

If the file is too long and complicated, it should be split into smaller files, functions, or modules. Files should not exceed 2000 lines (non-empty and non-commented lines).
It is recommended to horizontally or vertically split the file according to responsibilities or hierarchy of its parts.
The only exception to this rule is code generation - the auto-generated files that are not manually modified can be longer.

3.1.2 Code blocks in the source file should be separated by one blank line.

A source file contains code blocks in the following order: copyright, package name, imports, and top-level classes. They should be separated by one blank line.
a) Code blocks should be in the following order:
1. Kdoc for licensed or copyrighted files
2. **@file** annotation
3. Package name
4. Import statements
5. Top-class header and top-function header comments
6. Top-level classes or functions
b) Each of the preceding code blocks should be separated by a blank line.
c) Import statements are alphabetically arranged, without using line breaks and wildcards (wildcard imports - *).
d) **Recommendation**: One **.kt** source file should contain only one class declaration, and its name should match the filename
e) Avoid empty files that do not contain the code or contain only imports/comments/package name

3.1.3 Import statements order.

From top to bottom, the order is the following:
1. Android
2. Imports of packages used internally in your organization
3. Imports from other non-core dependencies
4. Java core packages
5. kotlin stdlib
Each category should be alphabetically arranged. Each group should be separated by a blank line. This style is compatible with Android import order.
Valid example:

```
import android.* // android
import androidx.* // android
import com.android.* // android

import com.your.company.* // your company's libs
```

```kotlin
import your.company.* // your company's libs

import com.fasterxml.jackson.databind.ObjectMapper // other third-party dependencies
import org.junit.jupiter.api.Assertions

import java.io.IOException // java core packages
import java.net.URL

import kotlin.system.exitProcess  // kotlin standard library
import kotlinx.coroutines.*  // official kotlin extension library
```

3.1.4 Order of declaration parts of class-like code structures.

The declaration parts of class-like code structures (class, interface, etc.) should be in the following order: compile-time constants (for objects), class properties, late-init class properties, init-blocks, constructors, public methods, internal methods, protected methods, private methods, and companion object. Blank lines should separate their declaration.

Notes:

1. There should be no blank lines between properties with the following **exceptions**: when there is a comment before a property on a separate line or annotations on a separate line.
2. Properties with comments/Kdoc should be separated by a newline before the comment/Kdoc.
3. Enum entries and constant properties (**const val**) in companion objects should be alphabetically arranged.

The declaration part of a class or interface should be in the following order:
- Compile-time constants (for objects)
- Properties
- Late-init class properties
- Init-blocks
- Constructors
- Methods or nested classes. Put nested classes next to the code they are used by.

If the classes are meant to be used externally, and are not referenced inside the class, put them after the companion object.
- Companion object

Exception:
All variants of a **(private) val** logger should be placed at the beginning of the class (**(private) val log**, **LOG**, **logger**, etc.).

3.2 Braces

This section describes the general rules of using braces in your code.

3.2.1 Using braces in conditional statements and loop blocks.

Braces should always be used in **if**, **else**, **for**, **do**, and **while** statements, even if the program body is empty or contains only one statement. In special Kotlin **when** statements, you do not need to use braces for single-line statements.

Valid example:

```kotlin
when (node.elementType) {
    FILE -> {
        checkTopLevelDoc(node)
        checkSomething()
     }
    CLASS -> checkClassElements(node)
}
```

Exception: The only exception is ternary operator in Kotlin (a single line **if ()** <> **else** <>)

Invalid example:

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

Valid example:

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

```
if (condition) {
    println("test")
} else {
    println(0)
}
```

3.2.2 Opening braces are placed at the end of the line in.

For *non-empty* blocks and block structures, the opening brace is placed at the end of the line.
Follow the K&R style (1TBS or OTBS) for *non-empty* code blocks with braces:
- The opening brace and first line of the code block are on the same line.
- The closing brace is on its own new line.
- The closing brace can be followed by a newline character. The only exceptions are **else**, **finally**, and **while** (from **do-while** statement), or **catch** keywords. These keywords should not be split from the closing brace by a newline character.
Exception cases:
1) For lambdas, there is no need to put a newline character after the first (function-related) opening brace. A newline character should appear only after an arrow (->) (see [point 5 of Rule 3.6.2]).

```
arg.map { value ->
    foo(value)
}
```

2) for **else**/**catch**/**finally**/**while** (from **do-while** statement) keywords closing brace should stay on the same line:

```
do {
    if (true) {
        x++
    } else {
        x--
    }
} while (x > 0)
```

Valid example:

```
        return arg.map { value ->
            while (condition()) {
                method()
            }
            value
        }

        return MyClass() {
            @Override
              fun method() {
                if (condition()) {
                    try {
                        something()
                    } catch (e: ProblemException) {
                        recover()
                    }
                } else if (otherCondition()) {
                    somethingElse()
                } else {
                    lastThing()
                }
            }
        }
```

3.3 Indentation

Only spaces are permitted for indentation, and each indentation should equal **four spaces** (tabs are not permitted).

If you prefer using tabs, simply configure them to change to spaces in your IDE automatically.

These code blocks should be indented if they are placed on the new line, and the following conditions are met:
- The code block is placed immediately after an opening brace.
- The code block is placed after each operator, including the assignment operator (+/-/&&/=/etc.)
- The code block is a call chain of methods:

```
someObject
    .map()
    .filter()
```

- The code block is placed immediately after the opening parenthesis.
- The code block is placed immediately after an arrow in lambda:

```
arg.map { value ->
    foo(value)
}
```

Exceptions:
1. Argument lists:
a) Eight spaces are used to indent argument lists (both in declarations and at call sites).
b) Arguments in argument lists can be aligned if they are on different lines.
2. Eight spaces are used if there is a newline after any binary operator.
3. Eight spaces are used for functional-like styles when the newline is placed before the dot.
4. Supertype lists:
a) Four spaces are used if the colon before the supertype list is on a new line.
b) Four spaces are used before each supertype, and eight spaces are used if the colon is on a new line.
Note: there should be an indentation after all statements such as **if**, **for**, etc. However, according to this code style, such statements require braces.

```
if (condition)
    foo()
```

Exceptions:
- When breaking the parameter list of a method/class constructor, it can be aligned with **8 spaces**. A parameter that was moved to a new line can be on the same level as the previous argument:

```
fun visit(
        node: ASTNode,
        autoCorrect: Boolean,
        params: KtLint.Params,
        emit: (offset: Int, errorMessage: String, canBeAutoCorrected: Boolean) -> Unit
) {
}
```

- Such operators as +/-/* can be indented with **8 spaces**:

```
val abcdef = "my splitted" +
                " string"
```

- A list of supertypes should be indented with **4 spaces** if they are on different lines or with **8 spaces** if the leading colon is also on a separate line

```
class A :
    B()

class A
    :
        B()
```

3.4 Empty blocks

Avoid empty blocks, and ensure braces start on a new line. An empty code block can be closed immediately on the same line and the next line. However, a newline is recommended between opening and closing braces **{}** (see the examples below.)

Generally, empty code blocks are prohibited; using them is considered a bad practice (especially for catch block).

They are only appropriate for overridden functions when the base class's functionality is not needed in the class-inheritor.

```
override fun foo() {
}
```

Valid examples (note once again that generally empty blocks are prohibited):

```
fun doNothing() {}

fun doNothingElse() {
}
```

Invalid examples:

```
try {
  doSomething()
} catch (e: Some) {}
```

Use the following valid code instead:

```
try {
   doSomething()
} catch (e: Some) {
}
```

3.5 Line length

Line length should be less than 120 symbols. The international code style prohibits **non-Latin** (**non-ASCII**) symbols.

(See [Identifiers]) However, if you still intend on using them, follow the following convention:

- One wide character occupies the width of two narrow characters.

The "wide" and "narrow" parts of a character are defined by its east Asian width Unicode attribute.

Typically, narrow characters are also called "half-width" characters.

All characters in the ASCII character set include letters (such as **a, A**), numbers (such as **0, 3**), and punctuation spaces (such as **,** , **{**), all of which are narrow characters.

Wide characters are also called "full-width" characters. Chinese characters (such as 中**,** 文), Chinese punctuation (，，；), full-width letters and numbers (such as Ａ、３) are "full-width" characters.

Each one of these characters represents two narrow characters.

- Any line that exceeds this limit (**120 narrow symbols**) should be wrapped, as described in the [Newline section].

Exceptions:

1. The long URL or long JSON method reference in KDoc.
2. The **package** and **import** statements.
3. The command line in the comment, enabling it to be cut and pasted into the shell for use.

3.6 Line breaks

This section contains the rules and recommendations on using line breaks.

3.6.1 Each line can have a maximum of one statement.

Each line can have a maximum of one code statement. This recommendation prohibits the use of code with **;** because it decreases code visibility.

Invalid example:

```
val a = ""; val b = ""
```

Valid example:

```
val a = ""
val b = ""
```

3.6.2 Rules for line-breaking.

1) Unlike Java, Kotlin allows you not to put a semicolon (;) after each statement separated by a newline character.
There should be no redundant semicolon at the end of the lines.
When a newline character is needed to split the line, it should be placed after such operators as **&&**/||/**+**/etc. and all infix functions (for example, **xor**).
However, the newline character should be placed before operators such as **.**, **?.**, **?:**, and **::**.
Note that all comparison operators, such as ==, >, <, should not be split.
Invalid example:

```
    if (node !=
            null && test != null) {}
```

Valid example:

```
        if (node != null &&
                test != null) {
        }
```

Note: You need to follow the functional style, meaning each function call in a chain with **.** should start at a new line if the chain of functions contains more than one call:

```
  val value = otherValue!!
          .map { x -> x }
          .filter {
              val a = true
              true
          }
          .size
```

Note: The parser prohibits the separation of the **!!** operator from the value it is checking.
Exception: If a functional chain is used inside the branches of a ternary operator, it does not need to be split with newlines.
Valid example:

```
if (condition) list.map { foo(it) }.filter { bar(it) } else list.drop(1)
```

2) Newlines should be placed after the assignment operator (=).
3) In function or class declarations, the name of a function or constructor should not be split by a newline from the opening brace **(**.
A brace should be placed immediately after the name without any spaces in declarations or at call sites.
4) Newlines should be placed right after the comma (**,**).
5) If a lambda statement contains more than one line in its body, a newline should be placed after an arrow if the lambda statement has explicit parameters.
If it uses an implicit parameter (**it**), the newline character should be placed after the opening brace (**{**).
The following examples illustrate this rule:
Invalid example:

```
    value.map { name -> foo()
        bar()
    }
```

Valid example:

```
value.map { name ->
    foo()
    bar()
}

val someValue = { node:String -> node }
```

6) When the function contains only a single expression, it can be written as expression function.
The below example shows the style that should not be used.
Instead of:

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

use:

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

7) If an argument list in a function declaration (including constructors) or function call contains more than two arguments, these arguments should be split by newlines in the following style.

Valid example:

```kotlin
class Foo(val a: String,
          b: String,
          val c: String) {
}

fun foo(
        a: String,
        b: String,
        c: String
) {

}
```

If and only if the first parameter is on the same line as an opening parenthesis, all parameters can be horizontally aligned by the first parameter.
Otherwise, there should be a line break after an opening parenthesis.
Kotlin 1.4 introduced a trailing comma as an optional feature, so it is generally recommended to place all parameters on a separate line
and append trailing comma.
It makes the resolving of merge conflicts easier.

Valid example:

```kotlin
fun foo(
        a: String,
        b: String,
) {

}
```

8) If the supertype list has more than two elements, they should be separated by newlines.

Valid example:

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

3.7 Using blank lines

Reduce unnecessary blank lines and maintain a compact code size. By reducing unnecessary blank lines, you can display more code on one screen, which improves code readability.

- Blank lines should separate content based on relevance and should be placed between groups of fields, constructors, methods, nested classes, **init** blocks, and objects (see [3.1.2]).
- Do not use more than one line inside methods, type definitions, and initialization expressions.
- Generally, do not use more than two consecutive blank lines in a row.
- Do not put newlines in the beginning or end of code blocks with curly braces.

Valid example:

```kotlin
fun baz() {

    doSomething()  // No need to add blank lines at the beginning and end of the code block
    // ...

}
```

3.8 Horizontal space

This section describes general rules and recommendations for using spaces in the code.

3.8.1: Usage of whitespace for code separation.

Follow the recommendations below for using space to separate keywords:
Note: These recommendations are for cases where symbols are located on the same line. However, in some cases, a line break could be used instead of a space.
1. Separate keywords (such as **if**, **when**, **for**) from the opening parenthesis with single whitespace.
The only exception is the **constructor** keyword, which should not be separated from the opening parenthesis.
2. Separate keywords like **else** or **try** from the opening brace ({) with single whitespace.
If **else** is used in a ternary-style statement without braces, there should be a single space between **else** and the statement after: **if (condition) foo() else bar()**
3. Use a **single** whitespace before all opening braces ({). The only exception is the passing of a lambda as a parameter inside parentheses:

```
    private fun foo(a: (Int) -> Int, b: Int) {}
    foo({x: Int -> x}, 5) // no space before '{'
```

4. Single whitespace should be placed on both sides of binary operators. This also applies to operator-like symbols.
For example:
- A colon in generic structures with the **where** keyword: **where T : Type**
- Arrow in lambdas: **(str: String) -> str.length()**
Exceptions:
- Two colons (**::**) are written without spaces:
Object::toString
- The dot separator (**.**) that stays on the same line with an object name:
object.toString()
- Safe access modifiers **?.** and **!!** that stay on the same line with an object name:
object?.toString()
- Operator **..** for creating ranges:
1..100
5. Use spaces after (**,**), (**:**), and (**;**), except when the symbol is at the end of the line.
However, note that this code style prohibits the use of (**;**) in the middle of a line ([see 3.3.2]).
There should be no whitespaces at the end of a line.
The only scenario where there should be no space after a colon is when the colon is used in the annotation to specify a use-site target (for example, **@param:JsonProperty**).
There should be no spaces before **,** , **:** and **;**.
Exceptions for spaces and colons:
- When **:** is used to separate a type and a supertype, including an anonymous object (after object keyword)
- When delegating to a superclass constructor or different constructor of the same class
Valid example:

```
  abstract class Foo<out T : Any> : IFoo { }

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

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

6. There should be *only one space* between the identifier and its type: **list: List<String>**
If the type is nullable, there should be no space before **?**.
7. When using **[]** operator (**get/set**) there should be **no** spaces between identifier and **[** : **someList[0]**.
8. There should be no space between a method or constructor name (both at declaration and at call site) and a parenthesis:
foo() {}. Note that this sub-rule is related only to spaces; the rules for whitespaces are described in [see 3.6.2].
This rule does not prohibit, for example, the following code:

```
fun foo
(
    a: String
)
```

9. Never put a space after **(**, **[**, **<** (when used as a bracket in templates) or before **)**, **]**, **>** (when used as a bracket in templates).
10. There should be no spaces between a prefix/postfix operator (like **!!** or **++**) and its operand.

3.8.2: No spaces for horizontal alignment.

Horizontal alignment refers to aligning code blocks by adding space to the code. Horizontal alignment should not be used because:
- When modifying code, it takes much time for new developers to format, support, and fix alignment issues.
- Long identifier names will break the alignment and lead to less presentable code.
- There are more disadvantages than advantages in alignment. To reduce maintenance costs, misalignment (???) is the best choice.
Recommendation: Alignment only looks suitable for **enum class**, where it can be used in table format to improve code readability:

```
enum class Warnings(private val id: Int, private val canBeAutoCorrected: Boolean, private val
    warn: String) : Rule {
    PACKAGE_NAME_MISSING         (1, true,  "no package name declared in a file"),
    PACKAGE_NAME_INCORRECT_CASE  (2, true,  "package name should be completely in a lower case
    "),
    PACKAGE_NAME_INCORRECT_PREFIX(3, false, "package name should start from the company's
    domain")
    ;
}
```

Valid example:

```
private val nr: Int // no alignment, but looks fine
private var color: Color // no alignment
```

Invalid example:

```
private val    nr: Int    // aligned comment with extra spaces
private val color: Color  // alignment for a comment and alignment for identifier name
```

3.9 Enumerations

Enum values are separated by a comma and line break, with ';' placed on the new line.

1) The comma and line break characters separate enum values. Put **;** on the new line:

```
enum class Warnings {
    A,
    B,
    C,
    ;
}
```

This will help to resolve conflicts and reduce the number of conflicts during merging pull requests.
Also, use trailing comma.
2) If the enum is simple (no properties, methods, and comments inside), you can declare it in a single line:

```
enum class Suit { CLUBS, HEARTS, SPADES, DIAMONDS }
```

3) Enum classes take preference (if it is possible to use it). For example, instead of two boolean properties:

```
val isCelsius = true
val isFahrenheit = false
```

use enum class:

```
enum class TemperatureScale { CELSIUS, FAHRENHEIT }
```

- The variable value only changes within a fixed range and is defined with the enum type.
- Avoid comparison with magic numbers of **-1, 0, and 1**; use enums instead.

```
enum class ComparisonResult {
    ORDERED_ASCENDING,
    ORDERED_SAME,
    ORDERED_DESCENDING,
    ;
}
```

3.10 Variable declaration

This section describes rules for the declaration of variables.

3.10.1 Declare one variable per line.

Each property or variable must be declared on a separate line.
Invalid example:

```
val n1: Int; val n2: Int
```

3.10.2 Variables should be declared near the line where they are first used.

Declare local variables close to the point where they are first used to minimize their scope. This will also increase the readability of the code. Local variables are usually initialized during their declaration or immediately after.
The member fields of the class should be declared collectively (see [Rule 3.1.2] for details on the class structure).

3.11 'When' expression

The **when** statement must have an 'else' branch unless the condition variable is enumerated or a sealed type.
Each **when** statement should contain an **else** statement group, even if it does not contain any code.
Exception: If 'when' statement of the **enum or sealed** type contains all enum values, there is no need to have an "else" branch.
The compiler can issue a warning when it is missing.

3.12 Annotations

Each annotation applied to a class, method or constructor should be placed on its own line. Consider the following examples:
1. Annotations applied to the class, method or constructor are placed on separate lines (one annotation per line).
Valid example:

```
@MustBeDocumented
@CustomAnnotation
fun getNameIfPresent() { /* ... */ }
```

2. A single annotation should be on the same line as the code it is annotating.
Valid example:

```
@CustomAnnotation class Foo {}
```

3. Multiple annotations applied to a field or property can appear on the same line as the corresponding field.
Valid example:

```
@MustBeDocumented @CustomAnnotation val loader: DataLoader
```

3.13 Block comments

Block comments should be placed at the same indentation level as the surrounding code. See examples below.
Valid example:

```
class SomeClass {
     /*
      * This is
      * okay
      */
      fun foo() {}
}
```

Note: Use /***...***/ block comments to enable automatic formatting by IDEs.

3.14 Modifiers and constant values

This section contains recommendations regarding modifiers and constant values.

3.14.1 Declaration with multiple modifiers.

If a declaration has multiple modifiers, always follow the proper sequence.
Valid sequence:

```
public / internal / protected / private
expect / actual
final / open / abstract / sealed / const
external
```

```
override
lateinit
tailrec
crossinline
vararg
suspend
inner
out
enum / annotation
companion
inline / noinline
reified
infix
operator
data
```

3.14.2: Separate long numerical values with an underscore.

An underscore character should separate long numerical values.
Note: Using underscores simplifies reading and helps to find errors in numeric constants.

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

3.15 Strings

This section describes the general rules of using strings.

3.15.1 Concatenation of Strings.

String concatenation is prohibited if the string can fit on one line. Use raw strings and string templates instead. Kotlin has significantly improved the use of Strings:
String templates, Raw strings.
Therefore, compared to using explicit concatenation, code looks much better when proper Kotlin strings are used for short lines, and you do not need to split them with newline characters.
Invalid example:

```
val myStr = "Super string"
val value = myStr + " concatenated"
```

Valid example:

```
val myStr = "Super string"
val value = "$myStr concatenated"
```

3.15.2 String template format.

Redundant curly braces in string templates
If there is only one variable in a string template, there is no need to use such a template. Use this variable directly.
Invalid example:

```
val someString = "${myArgument} ${myArgument.foo()}"
```

Valid example:

```
val someString = "$myArgument ${myArgument.foo()}"
```

Redundant string template
In case a string template contains only one variable - there is no need to use the string template. Use this variable directly.
Invalid example:

```
val someString = "$myArgument"
```

Valid example:

```
val someString = myArgument
```

4. VARIABLES AND TYPES

This section is dedicated to the rules and recommendations for using variables and types in your code.

4.1 Variables

The rules of using variables are explained in the below topics.

4.1.1 Do not use Float and Double types when accurate calculations are needed.

Floating-point numbers provide a good approximation over a wide range of values, but they cannot produce accurate results in some cases.
Binary floating-point numbers are unsuitable for precise calculations because it is impossible to represent 0.1 or any other negative power of 10 in a **binary representation** with a finite length.
The following code example seems to be obvious:

```
    val myValue = 2.0 - 1.1
    println(myValue)
```

However, it will print the following value: **0.8999999999999999**
Therefore, for precise calculations (for example, in finance or exact sciences), using such types as **Int**, **Long**, **BigDecimal**are recommended.
The **BigDecimal** type should serve as a good choice.
Invalid example:
Float values containing more than six or seven decimal numbers will be rounded.

```
 val eFloat = 2.7182818284f // Float, will be rounded to 2.7182817
```

Valid example: (when precise calculations are needed):

```
    val income = BigDecimal("2.0")
    val expense = BigDecimal("1.1")
    println(income.subtract(expense)) // you will obtain 0.9 here
```

4.1.2: Comparing numeric float type values.

Numeric float type values should not be directly compared with the equality operator (==) or other methods, such as **compareTo()** and **equals()**. Since floating-point numbers involve precision problems in computer representation, it is better to use **BigDecimal** as recommended in [Rule 4.1.1] to make accurate computations and comparisons. The following code describes these problems.
Invalid example:

```
val f1 = 1.0f - 0.9f
val f2 = 0.9f - 0.8f
if (f1 == f2) {
    println("Expected to enter here")
} else {
    println("But this block will be reached")
}

val flt1 = f1;
val flt2 = f2;
if (flt1.equals(flt2)) {
    println("Expected to enter here")
} else {
    println("But this block will be reached")
}
```

Valid example:

```
val foo = 1.03f
val bar = 0.42f
if (abs(foo - bar) > 1e-6f) {
    println("Ok")
} else {
    println("Not")
}
```

4.1.3 Try to use 'val' instead of 'var' for variable declaration.

Variables with the **val** modifier are immutable (read-only).
Using **val** variables instead of **var** variables increases code robustness and readability.
This is because **var** variables can be reassigned several times in the business logic.
However, in some scenarios with loops or accumulators, only **var**s are permitted.

4.2 Types

This section provides recommendations for using types.

4.2.1: Use Contracts and smart cast as much as possible.

The Kotlin compiler has introduced Smart Casts that help reduce the size of code.
Invalid example:

```
    if (x is String) {
        print((x as String).length) // x was already automatically cast to String - no need to use 'as' keyword here
    }
```

Valid example:

```
    if (x is String) {
        print(x.length) // x was already automatically cast to String - no need to use 'as' keyword here
    }
```

Also, Kotlin 1.3 introduced Contracts that provide enhanced logic for smart-cast.
Contracts are used and are very stable in **stdlib**, for example:

```
fun bar(x: String?) {
    if (!x.isNullOrEmpty()) {
        println("length of '$x' is ${x.length}") // smartcasted to not-null
    }
}
```

Smart cast and contracts are a better choice because they reduce boilerplate code and features forced type conversion.
Invalid example:

```
fun String?.isNotNull(): Boolean = this != null

fun foo(s: String?) {
    if (s.isNotNull()) s!!.length // No smartcast here and !! operator is used
}
```

Valid example:

```
fun foo(s: String?) {
    if (s.isNotNull()) s.length // We have used a method with contract from stdlib that helped compiler to execute smart cast
}
```

4.2.2: Try to use type alias to represent types making code more readable.

Type aliases provide alternative names for existing types.
If the type name is too long, you can replace it with a shorter name, which helps to shorten long generic types.
For example, code looks much more readable if you introduce a **typealias** instead of a long chain of nested generic types.
We recommend using a **typealias** if the type contains **more than two** nested generic types and is longer than **25 chars**.
Invalid example:

```
val b: MutableMap<String, MutableList<String>>
```

Valid example:

```
typealias FileTable = MutableMap<String, MutableList<String>>
val b: FileTable
```

You can also provide additional aliases for function (lambda-like) types:

```
typealias MyHandler = (Int, String, Any) -> Unit

typealias Predicate<T> = (T) -> Boolean
```

4.3 Null safety and variable declarations

Kotlin is declared as a null-safe programming language. However, to achieve compatibility with Java, it still supports nullable types.

4.3.1: Avoid declaring variables with nullable types.

To avoid **NullPointerException** and help the compiler prevent Null Pointer Exceptions, avoid using nullable types (with **?** symbol).
Invalid example:

```
val a: Int? = 0
```

Valid example:

```
val a: Int = 0
```

Nevertheless, when using Java libraries extensively, you have to use nullable types and enrich the code with **!!** and **?** symbols.
Avoid using nullable types for Kotlin stdlib (declared in official documentation).
Try to use initializers for empty collections. For example, if you want to initialize a list instead of **null**, use **emptyList()**.
Invalid example:

```
val a: List<Int>? = null
```

Valid example:

```
val a: List<Int> = emptyList()
```

4.3.2: Variables of generic types should have an explicit type declaration.

Like in Java, classes in Kotlin may have type parameters. To create an instance of such a class, we typically need to provide type arguments:

```
val myVariable: Map<Int, String> = emptyMap<Int, String>()
```

However, the compiler can inherit type parameters from the r-value (value assigned to a variable). Therefore, it will not force users to declare the type explicitly.
These declarations are not recommended because programmers would need to find the return value and understand the variable type by looking at the method.
Invalid example:

```
val myVariable = emptyMap<Int, String>()
```

Valid example:

```
val myVariable: Map<Int, String> = emptyMap()
```

4.3.3 Null-safety.

Try to avoid explicit null checks (explicit comparison with **null**)
Kotlin is declared as Null-safe language.
However, Kotlin architects wanted Kotlin to be fully compatible with Java; that's why the **null** keyword was also introduced in Kotlin.
There are several code-structures that can be used in Kotlin to avoid null-checks. For example: **?:**, **.let {}**, **.also {}**, e.t.c
Invalid example:

```kotlin
// example 1
var myVar: Int? = null
if (myVar == null) {
    println("null")
    return
}

// example 2
if (myVar != null) {
    println("not null")
    return
}

// example 3
val anotherVal = if (myVar != null) {
                    println("not null")
                    1
                } else {
                    2
                }
// example 4
if (myVar == null) {
    println("null")
} else {
    println("not null")
}
```

Valid example:

```kotlin
// example 1
var myVar: Int? = null
myVar?: run {
    println("null")
    return
}

// example 2
myVar?.let {
    println("not null")
    return
}

// example 3
val anotherVal = myVar?.also {
                    println("not null")
                    1
                    } ?: 2

// example 4
myVar?.let {
    println("null")
} ?: run { println("not null") }
```

Exceptions:
In the case of complex expressions, such as multiple **else-if** structures or long conditional statements, there is common sense to use explicit comparison with **null**.
Valid examples:

```
if (myVar != null) {
    println("not null")
} else if (anotherCondition) {
    println("Other condition")
}
```

```
if (myVar == null || otherValue == 5 && isValid) {}
```

Please also note, that instead of using **require(a != null)** with a not null check - you should use a special Kotlin function called **requireNotNull(a)**.

5. FUNCTIONS

This section describes the rules of using functions in your code.

5.1 Function design

Developers can write clean code by gaining knowledge of how to build design patterns and avoid code smells.
You should utilize this approach, along with functional style, when writing Kotlin code.
The concepts behind functional style are as follows:
Functions are the smallest unit of combinable and reusable code.
They should have clean logic, **high cohesion**, and **low coupling** to organize the code effectively.
The code in functions should be simple and not conceal the author's original intentions.
Additionally, it should have a clean abstraction, and control statements should be used straightforwardly.
The side effects (code that does not affect a function's return value but affects global/object instance variables) should not be used for state changes of an object.
The only exceptions to this are state machines.
Kotlin is designed to support and encourage functional programming, featuring the corresponding built-in mechanisms.
Also, it supports standard collections and sequences feature methods that enable functional programming (for example, **apply**, **with**, **let**, and **run**), Kotlin Higher-Order functions, function types, lambdas, and default function arguments.
As [previously discussed], Kotlin supports and encourages the use of immutable types, which in turn motivates programmers to write pure functions that avoid side effects and have a corresponding output for specific input.
The pipeline data flow for the pure function comprises a functional paradigm. It is easy to implement concurrent programming when you have chains of function calls, where each step features the following characteristics:
1. Simplicity
2. Verifiability
3. Testability
4. Replaceability
5. Pluggability
6. Extensibility
7. Immutable results
There can be only one side effect in this data stream, which can be placed only at the end of the execution queue.

5.1.1 Avoid functions that are too long.

The function should be displayable on one screen and only implement one certain logic.
If a function is too long, it often means complex and could be split or simplified. Functions should consist of 30 lines (non-empty and non-comment) in total.
Exception: Some functions that implement complex algorithms may exceed 30 lines due to aggregation and comprehensiveness.
Linter warnings for such functions **can be suppressed**.
Even if a long function works well, new problems or bugs may appear due to the function's complex logic once it is modified by someone else.
Therefore, it is recommended to split such functions into several separate and shorter functions that are easier to manage.
This approach will enable other programmers to read and modify the code properly.

5.1.2 Avoid deep nesting of function code blocks.

The nesting depth of a function's code block is the depth of mutual inclusion between the code control blocks in the function (for example: if, for, while, and when).

Each nesting level will increase the amount of effort needed to read the code because you need to remember the current "stack" (for example, entering conditional statements and loops).

Exception: The nesting levels of the lambda expressions, local classes, and anonymous classes in functions are calculated based on the innermost function. The nesting levels of enclosing methods are not accumulated.

Functional decomposition should be implemented to avoid confusion for the developer who reads the code.

This will help the reader switch between contexts.

5.1.3 Avoid using nested functions.

Nested functions create a more complex function context, thereby confusing readers.

With nested functions, the visibility context may not be evident to the code reader.

Invalid example:

```kotlin
fun foo() {
    fun nested():String {
        return "String from nested function"
    }
    println("Nested Output: ${nested()}")
}
```

5.1.4 Negated function calls.

Don't use negated function calls if it can be replaced with negated version of this function

Invalid example:

```kotlin
fun foo() {
    val list = listOf(1, 2, 3)

    if (!list.isEmpty()) {
        // Some cool logic
    }
}
```

Valid example:

```kotlin
fun foo() {
    val list = listOf(1, 2, 3)

    if (list.isNotEmpty()) {
        // Some cool logic
    }
}
```

5.2 Function arguments

The rules for using function arguments are described in the below topics.

5.2.1 The lambda parameter of the function should be placed at the end of the argument list.

With such notation, it is easier to use curly brackets, leading to better code readability.

Valid example:

```kotlin
// declaration
fun myFoo(someArg: Int, myLambda: () -> Unit) {
// ...
}

// usage
```

```
myFoo(1) {
println("hey")
}
```

5.2.2 Number of function parameters should be limited to five.

A long argument list is a code smell that leads to less reliable code.
It is recommended to reduce the number of parameters. Having **more than five** parameters leads to difficulties in maintenance and conflicts merging.
If parameter groups appear in different functions multiple times, these parameters are closely related and can be encapsulated into a single Data Class.
It is recommended that you use Data Classes and Maps to unify these function arguments.

5.2.3 Use default values for function arguments instead of overloading them.

In Java, default values for function arguments are prohibited. That is why the function should be overloaded when you need to create a function with fewer arguments.
In Kotlin, you can use default arguments instead.
Invalid example:

```
private fun foo(arg: Int) {
    // ...
}

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

Valid example:

```
 private fun foo(arg: Int = 0) {
     // ...
 }
```

6. CLASSES

6.1 Classes

This section describes the rules of denoting classes in your code.

6.1.1 Denoting a class with a single constructor.

When a class has a single constructor, it should be defined as a primary constructor in the declaration of the class. If the class contains only one explicit constructor, it should be converted to a primary constructor.
Invalid example:

```
class Test {
    var a: Int
    constructor(a: Int) {
        this.a = a
    }
}
```

Valid example:

```
class Test(var a: Int) {
    // ...
}

// in case of any annotations or modifiers used on a constructor:
class Test private constructor(var a: Int) {
    // ...
}
```

6.1.2 Prefer data classes instead of classes without any functional logic.

Some people say that the data class is a code smell. However, if you need to use it (which makes your code more simple), you can utilize the Kotlin **data class**. The main purpose of this class is to hold data,
but also **data class** will automatically generate several useful methods:
- equals()/hashCode() pair;
- toString()
- componentN() functions corresponding to the properties in their order of declaration;
- copy() function
Therefore, instead of using **normal** classes:

```
class Test {
    var a: Int = 0
        get() = field
        set(value: Int) { field = value}
}

class Test {
    var a: Int = 0
    var b: Int = 0

    constructor(a:Int, b: Int) {
        this.a = a
        this.b = b
    }
}

// or
class Test(var a: Int = 0, var b: Int = 0)

// or
class Test() {
    var a: Int = 0
    var b: Int = 0
}
```

prefer data classes:

```
data class Test1(var a: Int = 0, var b: Int = 0)
```

Exception 1: Note that data classes cannot be abstract, open, sealed, or inner; that is why these types of classes cannot be changed to a data class.
Exception 2: No need to convert a class to a data class if this class extends some other class or implements an interface.

6.1.3 Do not use the primary constructor if it is empty or useless.

The primary constructor is a part of the class header; it is placed after the class name and type parameters (optional) but can be omitted if it is not used.
Invalid example:

```
// simple case that does not need a primary constructor
class Test() {
    var a: Int = 0
    var b: Int = 0
}

// empty primary constructor is not needed here
// it can be replaced with a primary contructor with one argument or removed
class Test() {
    var a  = "Property"

    init {
        println("some init")
    }

    constructor(a: String): this() {
        this.a = a
    }
}
```

Valid example:

```
// the good example here is a data class; this example also shows that you should get rid of
    braces for the primary constructor
class Test {
    var a: Int = 0
    var b: Int = 0
}
```

***6.1.4 Do not use redundant init blocks in your class**.*

Several init blocks are redundant and generally should not be used in your class. The primary constructor cannot contain any code. That is why Kotlin has introduced **init** blocks.

These blocks store the code to be run during the class initialization.

Kotlin allows writing multiple initialization blocks executed in the same order as they appear in the class body.

Even when you follow (rule 3.2)[#r3.2], this makes your code less readable as the programmer needs to keep in mind all init blocks and trace the execution of the code.

Therefore, you should try to use a single **init** block to reduce the code's complexity. If you need to do some logging or make some calculations before the class property assignment, you can use powerful functional programming. This will reduce the possibility of the error if your **init** blocks' order is accidentally changed and

make the code logic more coupled. It is always enough to use one **init** block to implement your idea in Kotlin.

Invalid example:

```
class YourClass(var name: String) {
    init {
        println("First initializer block that prints ${name}")
    }

    val property = "Property: ${name.length}".also(::println)

    init {
        println("Second initializer block that prints ${name.length}")
    }
}
```

Valid example:

```
class YourClass(var name: String) {
    init {
        println("First initializer block that prints ${name}")
    }

    val property = "Property: ${name.length}".also { prop ->
        println(prop)
        println("Second initializer block that prints ${name.length}")
    }
}
```

The **init** block was not added to Kotlin to help you initialize your properties; it is needed for more complex tasks.
Therefore if the **init** block contains only assignments of variables - move it directly to properties to be correctly initialized near the declaration.
In some cases, this rule can be in clash with [6.1.1], but that should not stop you.
Invalid example:

```kotlin
class A(baseUrl: String) {
    private val customUrl: String
    init {
        customUrl = "$baseUrl/myUrl"
    }
}
```

Valid example:

```kotlin
class A(baseUrl: String) {
    private val customUrl = "$baseUrl/myUrl"
}
```

6.1.5 Explicit supertype qualification.

The explicit supertype qualification should not be used if there is no clash between called methods. This rule is applicable to both interfaces and classes.
Invalid example:

```kotlin
open class Rectangle {
    open fun draw() { /* ... */ }
}

class Square() : Rectangle() {
    override fun draw() {
        super<Rectangle>.draw() // no need in super<Rectangle> here
    }
}
```

6.1.6 Abstract class should have at least one abstract method.

Abstract classes are used to force a developer to implement some of its parts in their inheritors.
When the abstract class has no abstract methods, it was set **abstract** incorrectly and can be converted to a regular class.
Invalid example:

```kotlin
abstract class NotAbstract {
    fun foo() {}

    fun test() {}
}
```

Valid example:

```kotlin
abstract class NotAbstract {
    abstract fun foo()

    fun test() {}
}

// OR
class NotAbstract {
    fun foo() {}

    fun test() {}
}
```

6.1.7 When using the.

Kotlin has a mechanism of backing properties.
In some cases, implicit backing is not enough and it should be done explicitly:

```kotlin
private var _table: Map<String, Int>? = null
val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // Type parameters are inferred
        }
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

In this case, the name of the backing property (**_table**) should be the same as the name of the real property (**table**) but should have an underscore (_) prefix.
It is one of the exceptions from the [identifier names rule]

6.1.8 Avoid using custom getters and setters.

Kotlin has a perfect mechanism of properties.
Kotlin compiler automatically generates **get** and **set** methods for properties and can override them.
Invalid example:

```kotlin
class A {
    var size: Int = 0
        set(value) {
            println("Side effect")
            field = value
        }
        // user of this class does not expect calling A.size receive size * 2
        get() = field * 2
}
```

From the callee code, these methods look like access to this property: **A().isEmpty = true** for setter and **A().isEmpty** for getter.
However, when **get** and **set** are overridden, it isn't very clear for a developer who uses this particular class.
The developer expects to get the property value but receives some unknown value and some extra side-effect hidden by the custom getter/setter.
Use extra functions instead to avoid confusion.
Valid example:

```kotlin
class A {
    var size: Int = 0
    fun initSize(value: Int) {
        // some custom logic
    }

    // this will not confuse developer and he will get exactly what he expects
    fun goodNameThatDescribesThisGetter() = this.size * 2
}
```

Exception: Private setters are only exceptions that are not prohibited by this rule.

6.1.9 Never use the name of a variable in the custom getter or setter.

If you ignored [recommendation 6.1.8], be careful with using the name of the property in your custom getter/setter
as it can accidentally cause a recursive call and a **StackOverflow Error**. Use the **field** keyword instead.
Invalid example (very bad):

```kotlin
var isEmpty: Boolean
    set(value) {
        println("Side effect")
        isEmpty = value
    }
    get() = isEmpty
```

6.1.10 No trivial getters and setters are allowed in the code.

In Java, trivial getters - are the getters that are just returning the field value.
Trivial setters - are merely setting the field with a value without any transformation.
However, in Kotlin, trivial getters/setters are generated by default. There is no need to use it explicitly for all types of data structures in Kotlin.
Invalid example:

```
class A {
    var a: Int = 0
    get() = field
    set(value: Int) { field = value }

    //
}
```

Valid example:

```
class A {
    var a: Int = 0
    get() = field
    set(value: Int) { field = value }

    //
}
```

6.1.11 Use 'apply' for grouping object initialization.

In Java, before functional programming became popular, many classes from common libraries used the configuration paradigm.
To use these classes, you had to create an object with the constructor with 0-2 arguments and set the fields needed to run the object.
In Kotlin, to reduce the number of dummy code line and to group objects apply extension was added:
Invalid example:

```
class HttpClient(var name: String) {
    var url: String = ""
    var port: String = ""
    var timeout = 0

    fun doRequest() {}
}

fun main() {
    val httpClient = HttpClient("myConnection")
    httpClient.url = "http://example.com"
    httpClient.port = "8080"
    httpClient.timeout = 100

    httpCLient.doRequest()
}
```

Valid example:

```
class HttpClient(var name: String) {
    var url: String = ""
    var port: String = ""
    var timeout = 0

    fun doRequest() {}
}

fun main() {
    val httpClient = HttpClient("myConnection")
            .apply {
                url = "http://example.com"
                port = "8080"
                timeout = 100
            }
    httpClient.doRequest()
}
```

6.2 Extension functions

This section describes the rules of using extension functions in your code.

Extension functions is a killer-feature in Kotlin.

It gives you a chance to extend classes that were already implemented in external libraries and helps you to make classes less heavy.

Extension functions are resolved statically.

6.2.1 Use extension functions for making logic of classes less coupled.

It is recommended that for classes, the non-tightly coupled functions, which are rarely used in the class, should be implemented as extension functions where possible.

They should be implemented in the same class/file where they are used. This is a non-deterministic rule, so the code cannot be checked or fixed automatically by a static analyzer.

6.2.2 No extension functions with the same name and signature if they extend base and inheritor classes.

You should have ho extension functions with the same name and signature if they extend base and inheritor classes (possible_bug).esolved statically. There could be a situation when a developer implements two extension functions: one is for the base class and another for the inheritor.

This can lead to an issue when an incorrect method is used.

Invalid example:

```
open class A
class B: A()

// two extension functions with the same signature
fun A.foo() = "A"
fun B.foo() = "B"

fun printClassName(s: A) { println(s.foo()) }

// this call will run foo() method from the base class A, but
// programmer can expect to run foo() from the class inheritor B
fun main() { printClassName(B()) }
```

6.3 Interfaces

An **Interface** in Kotlin can contain declarations of abstract methods, as well as method implementations. What makes them different from abstract classes is that interfaces cannot store state.

They can have properties, but these need to be abstract or to provide accessor implementations.

Kotlin's interfaces can define attributes and functions.

In Kotlin and Java, the interface is the main presentation means of application programming interface (API) design and should take precedence over the use of (abstract) classes.

6.4 Objects

This section describes the rules of using objects in code.

6.4.1 Instead of using utility classes.

Avoid using utility classes/objects; use extensions instead. As described in [6.2 Extension functions], using extension functions is a powerful method. This enables you to avoid unnecessary complexity and class/object wrapping and use top-level functions instead.
Invalid example:

```kotlin
object StringUtil {
    fun stringInfo(myString: String): Int {
        return myString.count{ "something".contains(it) }
    }
}
StringUtil.stringInfo("myStr")
```

Valid example:

```kotlin
fun String.stringInfo(): Int {
    return this.count{ "something".contains(it) }
}

"myStr".stringInfo()
```

6.4.2 Objects should be used for Stateless Interfaces.

Kotlin's objects are extremely useful when you need to implement some interface from an external library that does not have any state.
There is no need to use classes for such structures.
Valid example:

```kotlin
interface I {
    fun foo()
}

object O: I {
    override fun foo() {}
}
```