📚 Detekt 1.23.4 Complete Rule Set

Presented By: Ali Khaleqi Yekta

December 2023

Note: To view the rule set in its original format, visit the official docs at https://detekt.dev/docs/intro.

AbsentOrWrongFileLicense

This rule will report every Kotlin source file which doesn't have the required license header. The rule validates each Kotlin source and operates in two modes: if licenseTemplateIsRegex = false (or missing) the rule checks whether the input file header starts with the read text from the passed file in the licenseTemplateFile configuration option. If licenseTemplateIsRegex = true the rule matches the header with a regular expression produced from the passed template license file (defined via licenseTemplateFile configuration option).

Active by default: No

Debt: 5min

CommentOverPrivateFunction

This rule reports comments and documentation that has been added to private functions. These comments get reported
because they probably explain the functionality of the private function. However, private functions should be small
enough and have an understandable name so that they are self-explanatory and do not need this comment in the first
place.

Instead of simply removing this comment to solve this issue prefer to split up the function into smaller functions
with better names if necessary. Giving the function a better, more descriptive name can also help in
solving this issue.

Active by default: No

Debt: 20min

CommentOverPrivateProperty

This rule reports comments and documentation above private properties. This can indicate that the property has a
confusing name or is not in a small enough context to be understood.
Private properties should be named in a self-explanatory way and readers of the code should be able to understand
why the property exists and what purpose it solves without the comment.

Instead of simply removing the comment to solve this issue, prefer renaming the property to a more self-explanatory
name. If this property is inside a bigger class, it makes sense to refactor and split up the class. This can
increase readability and make the documentation obsolete.

Active by default: No

Debt: 20min

DeprecatedBlockTag

This rule reports use of the `@deprecated` block tag in KDoc comments. Deprecation must be specified using a `@Deprecated` annotation as adding a `@deprecated` block tag in KDoc comments has no effect and is not supported. The `@Deprecated` annotation constructor has dedicated fields for a message and a type (warning, error, etc.). You can also use the `@ReplaceWith` annotation to specify how to solve the deprecation automatically via the IDE.

Active by default: No

Debt: 5min

Noncompliant Code:

```
/**
* This function prints a message followed by a new line.
*
* @deprecated Useless, the Kotlin standard library can already do this. Replace with println.
*/
fun printThenNewline(what: String) {
    // ...
}
```

Compliant Code:

```
/**
* This function prints a message followed by a new line.
*/
@Deprecated("Useless, the Kotlin standard library can already do this.")
@ReplaceWith("println(what)")
fun printThenNewline(what: String) {
    // ...
}
```

EndOfSentenceFormat

This rule validates the end of the first sentence of a KDoc comment.
It should end with proper punctuation or with a correct URL.

Active by default: No

Debt: 5min

KDocReferencesNonPublicProperty

This rule will report any KDoc comments that refer to non-public properties of a class. Clients do not need to know the implementation details.

Active by default: No

Debt: 5min

Noncompliant Code:

```
/**
* Comment
* [prop1] - non-public property
* [prop2] - public property
*/
class Test {
    private val prop1 = 0
    val prop2 = 0
}
```

Compliant Code:

```
/**
* Comment
* [prop2] - public property
*/
class Test {
    private val prop1 = 0
    val prop2 = 0
}
```

OutdatedDocumentation

This rule will report any class, function or constructor with KDoc that does not match the declaration signature.
If KDoc is not present or does not contain any @param or @property tags, rule violation will not be reported.
By default, both type and value parameters need to be matched and declarations orders must be preserved. You can
turn off these features using configuration options.

Active by default: No

Debt: 10min

Noncompliant Code:

```
/**
* @param someParam
* @property someProp
*/
class MyClass(otherParam: String, val otherProp: String)

/**
* @param T
* @param someParam
*/
fun <T, S> myFun(someParam: String)
```

Compliant Code:

```
/**
* @param someParam
* @property someProp
*/
class MyClass(someParam: String, val someProp: String)

/**
* @param T
* @param S
* @param someParam
*/
fun <T, S> myFun(someParam: String)
```

UndocumentedPublicClass

This rule reports public classes, objects and interfaces which do not have the required documentation.
Enable this rule if the codebase should have documentation on every public class, interface and object.

By default, this rule also searches for nested and inner classes and objects. This default behavior can be changed
with the configuration options of this rule.

Active by default: No

Debt: 20min

UndocumentedPublicFunction

This rule will report any public function which does not have the required documentation.
If the codebase should have documentation on all public functions enable this rule to enforce this.
Overridden functions are excluded by this rule.

Active by default: No

Debt: 20min

UndocumentedPublicProperty

This rule will report any public property which does not have the required documentation.
This also includes public properties defined in a primary constructor.
If the codebase should have documentation on all public properties enable this rule to enforce this.
Overridden properties are excluded by this rule.

Active by default: No

Debt: 20min

ComplexCondition

Complex conditions make it hard to understand which cases lead to the condition being true or false. To improve
readability and understanding of complex conditions consider extracting them into well-named functions or variables
and call those instead.

Active by default: Yes - Since v1.0.0

Debt: 20min

Noncompliant Code:

```
val str = "foo"
val isFoo = if (str.startsWith("foo") && !str.endsWith("foo") && !str.endsWith("bar") && !str.endsWith("_")) {
    // ...
}
```

Compliant Code:

```
val str = "foo"
val isFoo = if (str.startsWith("foo") && hasCorrectEnding()) {
    // ...
}

fun hasCorrectEnding() = return !str.endsWith("foo") && !str.endsWith("bar") && !str.endsWith("_")
```

ComplexInterface

Complex interfaces which contain too many functions and/or properties indicate that this interface is handling too
many things at once. Interfaces should follow the single-responsibility principle to also encourage implementations
of this interface to not handle too many things at once.

Large interfaces should be split into smaller interfaces which have a clear responsibility and are easier
to understand and implement.

Active by default: No

Debt: 20min

CyclomaticComplexMethod

Complex methods are hard to understand and read. It might not be obvious what side-effects a complex method has.
Prefer splitting up complex methods into smaller methods that are in turn easier to understand.
Smaller methods can also be named much clearer which leads to improved readability of the code...

This rule uses McCabe's Cyclomatic Complexity (MCC) metric to measure the number of linearly independent paths through a function's source code (https://www.ndepend.com/docs/code-metrics#CC)...

The higher the number of independent paths, the more complex a method is.
Complex methods use too many of the following statements.
Each one of them adds one to the complexity count.

- **Conditional statements** - `if`, `else if`, `when`
- **Jump statements** - `continue`, `break`
- **Loops** - `for`, `while`, `do-while`, `forEach`
- **Operators** `&&`, `||`, `?:`
- **Exceptions** - `catch`, `use`
- **Scope Functions** - `let`, `run`, `with`, `apply`, and `also` -> Reference

Active by default: Yes - Since v1.0.0

Debt: 20min

Aliases: ComplexMethod

LabeledExpression

This rule reports labeled expressions. Expressions with labels generally increase complexity and worsen the
maintainability of the code. Refactor the violating code to not use labels instead. Labeled expressions referencing an outer class with a label from an inner class are allowed, because there is no
way to get the instance of an outer class from an inner class in Kotlin.

Active by default: No

Debt: 20min

Noncompliant Code:

```
val range = listOf<String>("foo", "bar")
loop@ for (r in range) {
    if (r == "bar") break@loop
    println(r)
}

class Outer {
    inner class Inner {
        fun f() {
            val i = this@Inner // referencing itself, use `this instead
        }
    }
}
```

Compliant Code:

```
val range = listOf<String>("foo", "bar")
for (r in range) {
    if (r == "bar") break
    println(r)
}

class Outer {
    inner class Inner {
        fun f() {
            val outer = this@Outer
        }
        fun Int.extend() {
            val inner = this@Inner // this would reference Int and not Inner
        }
    }
}
```

LargeClass

This rule reports large classes. Classes should generally have one responsibility. Large classes can indicate that
the class does instead handle multiple responsibilities. Instead of doing many things at once prefer to
split up large classes into smaller classes. These smaller classes are then easier to understand and handle less
things.

Active by default: Yes - Since v1.0.0

Debt: 20min

LongMethod

Methods should have one responsibility. Long methods can indicate that a method handles too many cases at once.
Prefer smaller methods with clear names that describe their functionality clearly.

Extract parts of the functionality of long methods into separate, smaller methods.

Active by default: Yes - Since v1.0.0

Debt: 20min

LongParameterList

Reports functions and constructors which have more parameters than a certain threshold.

Active by default: Yes - Since v1.0.0

Debt: 20min

MethodOverloading

This rule reports methods which are overloaded often.
Method overloading tightly couples these methods together which might make the code harder to understand.

Refactor these methods and try to use optional parameters instead to prevent some of the overloading.

Active by default: No

Debt: 20min

NamedArguments

Reports function invocations which have more arguments than a certain threshold and are all not named. Calls with
too many arguments are more difficult to understand so a named arguments help.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
fun sum(a: Int, b: Int, c: Int, d: Int) {
}
sum(1, 2, 3, 4)
```

Compliant Code:

```
fun sum(a: Int, b: Int, c: Int, d: Int) {
}
sum(a = 1, b = 2, c = 3, d = 4)
```

NestedBlockDepth

This rule reports excessive nesting depth in functions. Excessively nested code becomes harder to read and increases
its hidden complexity. It might become harder to understand edge-cases of the function.

Prefer extracting the nested code into well-named functions to make it easier to understand.

Active by default: Yes - Since v1.0.0

Debt: 20min

NestedScopeFunctions

Although the scope functions are a way of making the code more concise, avoid overusing them: it can decrease
your code readability and lead to errors. Avoid nesting scope functions and be careful when chaining them:
it's easy to get confused about the current context object and the value of this or it.

Reference

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
// Try to figure out, what changed, without knowing the details
first.apply {
    second.apply {
        b = a
        c = b
    }
}
```

Compliant Code:

```
// 'a' is a property of current class
// 'b' is a property of class 'first'
// 'c' is a property of class 'second'
first.b = this.a
second.c = first.b
```

ReplaceSafeCallChainWithRun

Chains of safe calls on non-nullable types are redundant and can be removed by enclosing the redundant safe calls in
a `run {}` block. This improves code coverage and reduces cyclomatic complexity as redundant null checks are removed.

This rule only checks from the end of a chain and works backwards, so it won't recommend inserting run blocks in the
middle of a safe call chain as that is likely to make the code more difficult to understand...

The rule will check for every opportunity to replace a safe call when it sits at the end of a chain, even if there's
only one, as that will still improve code coverage and reduce cyclomatic complexity.

Active by default: No

Requires Type Resolution

Debt: 10min

Noncompliant Code:

```
val x = System.getenv()
    ?.getValue("HOME")
    ?.toLowerCase()
    ?.split("/") ?: emptyList()
```

Compliant Code:

```
val x = getenv()?.run {
    getValue("HOME")
        .toLowerCase()
        .split("/")
} ?: emptyList()
```

StringLiteralDuplication

This rule detects and reports duplicated String literals. Repeatedly typing out the same String literal across the
codebase makes it harder to change and maintain.

Instead, prefer extracting the String literal into a property or constant.

Active by default: No

Debt: 5min

Noncompliant Code:

```
class Foo {

    val s1 = "lorem"
    fun bar(s: String = "lorem") {
        s1.equals("lorem")
    }
}
```

Compliant Code:

```
class Foo {
    val lorem = "lorem"
    val s1 = lorem
    fun bar(s: String = lorem) {
        s1.equals(lorem)
    }
}
```

TooManyFunctions

This rule reports files, classes, interfaces, objects and enums which contain too many functions.
Each element can be configured with different thresholds.

Too many functions indicate a violation of the single responsibility principle. Prefer extracting functionality
which clearly belongs together in separate parts of the code.

Active by default: Yes - Since v1.0.0

Debt: 20min

Noncompliant Code:

```
fun myFunc() {
coroutineScope(Dispatchers.IO)
}
```

Compliant Code:

```
fun myFunc(dispatcher: CoroutineDispatcher = Dispatchers.IO) {
coroutineScope(dispatcher)
}

class MyRepository(dispatchers: CoroutineDispatcher = Dispatchers.IO)
```

RedundantSuspendModifier

`suspend` modifier should only be used where needed, otherwise the function can only be used from other suspending
functions. This needlessly restricts use of the function and should be avoided by removing the `suspend` modifier
where it's not needed.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
suspend fun normalFunction() {
    println("string")
}
```

Compliant Code:

```
fun normalFunction() {
    println("string")
}
```

SleepInsteadOfDelay

Report usages of `Thread.sleep` in suspending functions and coroutine blocks. A thread can
contain multiple coroutines at one time due to coroutines' lightweight nature, so if one coroutine invokes `Thread.sleep`, it could potentially halt the execution of unrelated coroutines
and cause unpredictable behavior.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
suspend fun foo() {
    Thread.sleep(1_000L)
}
```

Compliant Code:

```
suspend fun foo() {
    delay(1_000L)
}
```

SuspendFunSwallowedCancellation

`suspend` functions should not be called inside `runCatching`'s lambda block, because `runCatching` catches all the `Exception`s. For Coroutines to work in all cases, developers should make sure to propagate `CancellationException` exceptions. This means `CancellationException` should never be:

- caught and swallowed (even if logged)
- caught and propagated to external systems
- caught and shown to the user

they must always be rethrown in the same context...

Using `runCatching` increases this risk of mis-handling cancellation. If you catch and don't rethrow all the `CancellationException`, your coroutines are not cancelled even if you cancel their `CoroutineScope`. This can very easily lead to:

- unexpected crashes
- extremely hard to diagnose bugs
- memory leaks
- performance issues
- battery drain

See the reference for more details.
If your project wants to use `runCatching` and `Result` objects, it is recommended to write a `coRunCatching` utility function which immediately re-throws `CancellationException`; and forbid `runCatching` and `suspend` combinations by activating this rule.

Active by default: No

Requires Type Resolution

Debt: 10min

Noncompliant Code:

```
@Throws(IllegalStateException::class)
suspend fun bar(delay: Long) {
    check(delay <= 1_000L)
    delay(delay)
}

suspend fun foo() {
    runCatching {
        bar(1_000L)
    }
}
```

Compliant Code:

```
@Throws(IllegalStateException::class)
suspend fun bar(delay: Long) {
    check(delay <= 1_000L)
    delay(delay)
}

suspend fun foo() {
    try {
        bar(1_000L)
    } catch (e: IllegalStateException) {
        // handle error
    }
}

// Alternate
@Throws(IllegalStateException::class)
suspend fun foo() {
    bar(1_000L)
}
```

SuspendFunWithCoroutineScopeReceiver

Suspend functions that use `CoroutineScope` as receiver should not be marked as `suspend`.
A `CoroutineScope` provides structured concurrency via its `coroutineContext`. A `suspend`
function also has its own `coroutineContext`, which is now ambiguous and mixed with the
receiver`s.

See https://kotlinlang.org/docs/coroutines-basics.html#scope-builder-and-concurrency

Active by default: No

Requires Type Resolution

Debt: 10min

Aliases: SuspendFunctionOnCoroutineScope

Noncompliant Code:

```
suspend fun CoroutineScope.foo() {
    launch {
      delay(1.seconds)
    }
}
```

Compliant Code:

```
fun CoroutineScope.foo() {
    launch {
      delay(1.seconds)
    }
}

// Alternative
suspend fun foo() = coroutineScope {
    launch {
      delay(1.seconds)
    }
}
```

SuspendFunWithFlowReturnType

Functions that return `Flow` from `kotlinx.coroutines.flow` should not be marked as `suspend`.
`Flows` are intended to be cold observable streams. The act of simply invoking a function that
returns a `Flow`, should not have any side effects. Only once collection begins against the
returned `Flow`, should work actually be done.

See https://kotlinlang.org/docs/flow.html#flows-are-cold

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 10min

Noncompliant Code:

```
suspend fun observeSignals(): Flow<Unit> {
    val pollingInterval = getPollingInterval() // Done outside of the flow builder block.
    return flow {
        while (true) {
            delay(pollingInterval)
            emit(Unit)
        }
    }
}

private suspend fun getPollingInterval(): Long {
    // Return the polling interval from some repository
    // in a suspending manner.
}
```

Compliant Code:

```
fun observeSignals(): Flow<Unit> {
    return flow {
        val pollingInterval = getPollingInterval() // Moved into the flow builder block.
        while (true) {
            delay(pollingInterval)
            emit(Unit)
        }
    }
}

private suspend fun getPollingInterval(): Long {
    // Return the polling interval from some repository
    // in a suspending manner.
}
```

- `allowedExceptionNameRegex` (default: `'_|(ignore|expected).*'`)

  ignores exception types which match this regex

EmptyClassBlock

Reports empty classes. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

EmptyDefaultConstructor

Reports empty default constructors. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

EmptyDoWhileBlock

Reports empty `do` / `while` loops. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

EmptyElseBlock

Reports empty `else` blocks. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

EmptyFinallyBlock

Reports empty `finally` blocks. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

EmptyForBlock

Reports empty `for` loops. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

EmptyFunctionBlock

Reports empty functions. Empty blocks of code serve no purpose and should be removed.
This rule will not report functions with the override modifier that have a comment as their only body contents
(e.g., a `// no-op` comment in an unused listener function).

Set the `ignoreOverridden` parameter to `true` to exclude all functions which are overriding other
functions from the superclass or from an interface (i.e., functions declared with the override modifier).

Active by default: Yes - Since v1.0.0

Debt: 5min

EmptyIfBlock

Reports empty `if` blocks. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

EmptyInitBlock

Reports empty `init` expressions. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

EmptyKtFile

Reports empty Kotlin (.kt) files. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

EmptySecondaryConstructor

Reports empty secondary constructors. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

EmptyTryBlock

Reports empty `try` blocks. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.6.0

Debt: 5min

EmptyWhenBlock

Reports empty `when` expressions. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

EmptyWhileBlock

Reports empty `while` expressions. Empty blocks of code serve no purpose and should be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

- `methodNames` (default: `['equals', 'finalize', 'hashCode', 'toString']`)

  methods which should not throw exceptions

Noncompliant Code:

```
class Foo {

    override fun toString(): String {
        throw IllegalStateException() // exception should not be thrown here
    }
}
```

InstanceOfCheckForException

This rule reports `catch` blocks which check for the type of exception via `is` checks or casts.
Instead of catching generic exception types and then checking for specific exception types the code should
use multiple catch blocks. These catch blocks should then catch the specific exceptions.

Active by default: Yes - Since v1.21.0

Debt: 20min

Noncompliant Code:

```
fun foo() {
    try {
        // ... do some I/O
    } catch(e: IOException) {
        if (e is MyException || (e as MyException) != null) { }
    }
}
```

Compliant Code:

```
fun foo() {
    try {
        // ... do some I/O
    } catch(e: MyException) {
    } catch(e: IOException) {
    }
}
```

NotImplementedDeclaration

This rule reports all exceptions of the type `NotImplementedError` that are thrown. It also reports all `TODO(..)`
functions.
These indicate that functionality is still under development and will not work properly. Both of these should only
serve as temporary declarations and should not be put into production environments.

Active by default: No

Debt: 20min

Noncompliant Code:

```
fun foo() {
    throw NotImplementedError()
}

fun todo() {
    TODO("")
}
```

ObjectExtendsThrowable

This rule reports all `objects` including `companion objects` that extend any type of `Throwable`. Throwable instances are not intended for reuse as they are stateful and contain
mutable information about a specific exception or error. Hence, global singleton `Throwables`
should be avoided.

See https://kotlinlang.org/docs/object-declarations.html#object-declarations-overview
See https://kotlinlang.org/docs/object-declarations.html#companion-objects

Active by default: No

Requires Type Resolution

Debt: 10min

Noncompliant Code:

```
object InvalidCredentialsException : Throwable()

object BanException : Exception()

object AuthException : RuntimeException()
```

Compliant Code:

```
class InvalidCredentialsException : Throwable()

class BanException : Exception()

class AuthException : RuntimeException()
```

PrintStackTrace

This rule reports code that tries to print the stacktrace of an exception. Instead of simply printing a stacktrace
a better logging solution should be used.

Active by default: Yes - Since v1.16.0

Debt: 20min

Noncompliant Code:

```
fun foo() {
    Thread.dumpStack()
}

fun bar() {
    try {
        // ...
    } catch (e: IOException) {
        e.printStackTrace()
    }
}
```

Compliant Code:

```
val LOGGER = Logger.getLogger()

fun bar() {
    try {
        // ...
    } catch (e: IOException) {
        LOGGER.info(e)
    }
}
```

RethrowCaughtException

This rule reports all exceptions that are caught and then later re-thrown without modification.
It ignores cases:

1. When caught exceptions that are rethrown if there is work done before that.
2. When there are more than one catch in try block and at least one of them has some work.

Active by default: Yes - Since v1.16.0

Debt: 5min

Noncompliant Code:

```
fun foo() {
    try {
        // ...
    } catch (e: IOException) {
        throw e
    }
}
```

Compliant Code:

```
fun foo() {
    try {
        // ...
    } catch (e: IOException) {
        throw MyException(e)
    }
    try {
        // ...
    } catch (e: IOException) {
        print(e)
        throw e
    }
    try {
        // ...
    } catch (e: IOException) {
        print(e.message)
        throw e
    }

    try {
        // ...
    } catch (e: IOException) {
        throw e
    } catch (e: Exception) {
        print(e.message)
    }
}
```

ReturnFromFinally

Reports all `return` statements in `finally` blocks.
Using `return` statements in `finally` blocks can discard and hide exceptions that are thrown in the `try` block.
Furthermore, this rule reports values from `finally` blocks, if the corresponding `try` is used as an expression.

Active by default: Yes - Since v1.16.0

Requires Type Resolution

Debt: 20min

Noncompliant Code:

```
fun foo() {
    try {
        throw MyException()
    } finally {
        return // prevents MyException from being propagated
    }
}

val a: String = try { "s" } catch (e: Exception) { "e" } finally { "f" }
```

SwallowedException

Exceptions should not be swallowed. This rule reports all instances where exceptions are `caught` and not correctly
passed (e.g. as a cause) into a newly thrown exception.

The exception types configured in `ignoredExceptionTypes` indicate nonexceptional outcomes.
These by default configured exception types are part of Java.
Therefore, Kotlin developers have to handle them by using the catch clause.
For that reason, this rule ignores that these configured exception types are caught.

Active by default: Yes - Since v1.16.0

Debt: 20min

Noncompliant Code:

```
fun foo() {
    try {
        // ...
    } catch(e: IOException) {
        throw MyException(e.message) // e is swallowed
    }
    try {
        // ...
    } catch(e: IOException) {
        throw MyException() // e is swallowed
    }
    try {
        // ...
    } catch(e: IOException) {
        bar() // exception is unused
    }
}
```

Compliant Code:

```
fun foo() {
    try {
        // ...
    } catch(e: IOException) {
        throw MyException(e)
    }
    try {
        // ...
    } catch(e: IOException) {
        println(e) // logging is ok here
    }
}
```

ThrowingExceptionFromFinally

This rule reports all cases where exceptions are thrown from a `finally` block.
Throwing exceptions from a `finally`
block should be avoided as it can lead to confusion and discarded exceptions.

Active by default: Yes - Since v1.16.0

Debt: 20min

Noncompliant Code:

```
fun foo() {
    try {
        // ...
    } finally {
        throw IOException()
    }
}
```

ThrowingExceptionInMain

This rule reports all exceptions that are thrown in a `main` method.
An exception should only be thrown if it can be handled by a "higher" function.

Active by default: No

Debt: 20min

Noncompliant Code:

```
fun main(args: Array<String>) {
    // ...
    throw IOException() // exception should not be thrown here
}
```

ThrowingExceptionsWithoutMessageOrCause

This rule reports all exceptions which are thrown without arguments or further description.
Exceptions should always call one of the constructor overloads to provide a message or a cause.
Exceptions should be meaningful and contain as much detail about the error case as possible. This will help to track
down an underlying issue in a better way.

Active by default: Yes - Since v1.16.0

Debt: 5min

Noncompliant Code:

```
fun foo(bar: Int) {
    if (bar < 1) {
        throw IllegalArgumentException()
    }
    // ...
}
```

Compliant Code:

```
fun foo(bar: Int) {
    if (bar < 1) {
        throw IllegalArgumentException("bar must be greater than zero")
    }
    // ...
}
```

ThrowingNewInstanceOfSameException

Exceptions should not be wrapped inside the same exception type and then rethrown. Prefer wrapping exceptions in more
meaningful exception types.

Active by default: Yes - Since v1.16.0

Debt: 5min

Noncompliant Code:

```
fun foo() {
    try {
        // ...
    } catch (e: IllegalStateException) {
        throw IllegalStateException(e) // rethrows the same exception
    }
}
```

Compliant Code:

```
fun foo() {
    try {
        // ...
    } catch (e: IllegalStateException) {
        throw MyException(e)
    }
}
```

TooGenericExceptionCaught

This rule reports `catch` blocks for exceptions that have a type that is too generic.
It should be preferred to catch specific exceptions to the case that is currently handled.
If the scope of the caught
exception is too broad it can lead to unintended exceptions being caught.

Active by default: Yes - Since v1.0.0

Debt: 20min

Noncompliant Code:

```
fun foo() {
    try {
        // ... do some I/O
    } catch(e: Exception) { } // too generic exception caught here
}
```

Compliant Code:

```
fun foo() {
    try {
        // ... do some I/O
    } catch(e: IOException) { }
}
```

TooGenericExceptionThrown

This rule reports thrown exceptions that have a type that is too generic. It should be preferred to throw specific
exceptions to the case that has currently occurred.

Active by default: Yes - Since v1.0.0

Debt: 20min

Noncompliant Code:

```
fun foo(bar: Int) {
    if (bar < 1) {
        throw Exception() // too generic exception thrown here
    }
    // ...
}
```

Compliant Code:

```
fun foo(bar: Int) {
    if (bar < 1) {
        throw IllegalArgumentException("bar must be greater than zero")
    }
    // ...
}
```

- `indentSize` (default: `4`)

  indentation size

AnnotationSpacing

See ktlint docs for documentation.

Active by default: Yes - Since v1.22.0

ArgumentListWrapping

See ktlint docs for documentation.

Active by default: Yes - Since v1.22.0

BlockCommentInitialStarAlignment

See ktlint docs for documentation.

Active by default: Yes - Since v1.23.0

ChainWrapping

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

ClassName

See ktlint docs for documentation.

Active by default: No

CommentSpacing

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

CommentWrapping

See ktlint docs for documentation.

Active by default: Yes - Since v1.23.0

ContextReceiverMapping

See ktlint docs for documentation.

Active by default: No

DiscouragedCommentLocation

See ktlint docs for documentation.

Active by default: No

EnumEntryNameCase

See ktlint docs for documentation.

Active by default: Yes - Since v1.22.0

EnumWrapping

See ktlint docs for documentation.

Active by default: No

Filename

See ktlint docs for documentation.

This rules overlaps with naming>MatchingDeclarationName from the standard rules, make sure to enable just one.

Active by default: Yes - Since v1.0.0

FinalNewline

See ktlint docs for documentation.

This rules overlaps with style>NewLineAtEndOfFile
from the standard rules, make sure to enable just one. The pro of this rule is that it can auto-correct the issue.

Active by default: Yes - Since v1.0.0

FunKeywordSpacing

See ktlint docs for documentation.

Active by default: Yes - Since v1.23.0

FunctionName

See ktlint docs for documentation.

Active by default: No

FunctionReturnTypeSpacing

See ktlint docs for documentation.

Active by default: Yes - Since v1.23.0

FunctionSignature

See ktlint docs for documentation.

Active by default: No

FunctionStartOfBodySpacing

See ktlint docs for documentation.

Active by default: Yes - Since v1.23.0

FunctionTypeReferenceSpacing

See ktlint docs for documentation.

Active by default: Yes - Since v1.23.0

IfElseBracing

See ktlint docs for documentation.

Active by default: No

IfElseWrapping

See ktlint docs for documentation.

Active by default: No

ImportOrdering

See ktlint docs for documentation.

For defining import layout patterns see the KtLint Source Code

Active by default: Yes - Since v1.19.0

Indentation

See ktlint docs for documentation.

Active by default: Yes - Since v1.19.0

KdocWrapping

See ktlint docs for documentation.

Active by default: Yes - Since v1.23.0

MaximumLineLength

See ktlint docs for documentation.

This rules overlaps with style>MaxLineLength
from the standard rules, make sure to enable just one or keep them aligned. The pro of this rule is that it can
auto-correct the issue.

Active by default: Yes - Since v1.0.0

ModifierListSpacing

See ktlint docs for documentation.

Active by default: Yes - Since v1.23.0

ModifierOrdering

See ktlint docs for documentation.

This rules overlaps with style>ModifierOrder
from the standard rules, make sure to enable just one. The pro of this rule is that it can auto-correct the issue.

Active by default: Yes - Since v1.0.0

MultiLineIfElse

See ktlint docs for documentation.

Active by default: Yes - Since v1.22.0

MultilineExpressionWrapping

See ktlint docs for documentation.

Active by default: No

NoBlankLineBeforeRbrace

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

NoBlankLineInList

See ktlint docs for documentation.

Active by default: No

NoBlankLinesInChainedMethodCalls

See ktlint docs for documentation.

Active by default: Yes - Since v1.22.0

NoConsecutiveBlankLines

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

NoConsecutiveComments

See ktlint docs for documentation.

Active by default: No

NoEmptyClassBody

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

NoEmptyFirstLineInClassBody

See ktlint docs
for documentation.

Active by default: No

NoEmptyFirstLineInMethodBlock

See ktlint docs for documentation.

Active by default: Yes - Since v1.22.0

NoLineBreakAfterElse

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

NoLineBreakBeforeAssignment

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

NoMultipleSpaces

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

NoSemicolons

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

NoSingleLineBlockComment

See ktlint docs for documentation.

Active by default: No

NoTrailingSpaces

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

NoUnitReturn

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

NoUnusedImports

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

NoWildcardImports

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

NullableTypeSpacing

See ktlint docs for documentation.

Active by default: Yes - Since v1.23.0

PackageName

See ktlint docs for documentation.

Active by default: Yes - Since v1.22.0

ParameterListSpacing

See ktlint docs for documentation.

Active by default: No

ParameterListWrapping

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

ParameterWrapping

See ktlint docs for documentation.

Active by default: Yes - Since v1.23.0

PropertyName

See ktlint docs for documentation.

Active by default: No

PropertyWrapping

See ktlint docs for documentation.

Active by default: Yes - Since v1.23.0

SpacingAroundAngleBrackets

See ktlint docs for documentation.

Active by default: Yes - Since v1.22.0

SpacingAroundColon

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

SpacingAroundComma

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

SpacingAroundCurly

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

SpacingAroundDot

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

SpacingAroundDoubleColon

See ktlint docs for documentation.

Active by default: Yes - Since v1.22.0

SpacingAroundKeyword

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

SpacingAroundOperators

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

SpacingAroundParens

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

SpacingAroundRangeOperator

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

SpacingAroundUnaryOperator

See ktlint docs for documentation.

Active by default: Yes - Since v1.22.0

SpacingBetweenDeclarationsWithAnnotations

See ktlint docs
for documentation.

Active by default: Yes - Since v1.22.0

SpacingBetweenDeclarationsWithComments

See ktlint docs

for documentation.

Active by default: Yes - Since v1.22.0

SpacingBetweenFunctionNameAndOpeningParenthesis

See ktlint docs for
documentation.

Active by default: Yes - Since v1.23.0

StringTemplate

See ktlint docs for documentation.

Active by default: Yes - Since v1.0.0

StringTemplateIndent

See ktlint docs for documentation.

Active by default: No

TrailingCommaOnCallSite

See ktlint docs for documentation.

The default config comes from ktlint and follows these conventions:

- Kotlin coding convention recommends
  trailing comma encourage the use of trailing commas at the declaration site and leaves it at your discretion for the call site.
- Android Kotlin style guide does not include
  trailing comma usage yet.

Active by default: No

TrailingCommaOnDeclarationSite

See ktlint docs for documentation.

The default config comes from ktlint and follows these conventions:

- Kotlin coding convention recommends
  trailing comma encourage the use of trailing commas at the declaration site and leaves it at your discretion for the call site.
- Android Kotlin style guide does not include
  trailing comma usage yet.

Active by default: No

TryCatchFinallySpacing

See ktlint docs for documentation.

Active by default: No

TypeArgumentListSpacing

See ktlint docs for documentation.

Active by default: No

TypeParameterListSpacing

See ktlint docs for documentation.

Active by default: No

UnnecessaryParenthesesBeforeTrailingLambda

See ktlint docs
for documentation.

Active by default: Yes - Since v1.23.0

Wrapping

See ktlint docs for documentation.

Active by default: Yes - Since v1.20.0

Noncompliant Code:

```
data class C(val a: String) // violation: public data class
```

Compliant Code:

```
internal data class C(val a: String)
```

LibraryCodeMustSpecifyReturnType

Functions/properties exposed as public APIs of a library should have an explicit return type.
Inferred return type can easily be changed by mistake which may lead to breaking changes.

See also: Kotlin 1.4 Explicit API

Active by default: Yes - Since v1.2.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
// code from a library
val strs = listOf("foo, bar")
fun bar() = 5
class Parser {
    fun parse() = ...
}
```

Compliant Code:

```
// code from a library
val strs: List<String> = listOf("foo, bar")
fun bar(): Int = 5

class Parser {
    fun parse(): ParsingResult = ...
}
```

LibraryEntitiesShouldNotBePublic

Library typealias and classes should be internal or private.

Active by default: Yes - Since v1.16.0

Debt: 5min

Noncompliant Code:

```
// code from a library
class A
```

Compliant Code:

```
// code from a library
internal class A
```

- `allowedPattern` (default: `'^(is|has|are)'`)

  naming pattern

- ~~`ignoreOverridden`~~ (default: `true`)

  Deprecated: This configuration is ignored and will be removed in the future

  ignores properties that have the override modifier

Noncompliant Code:

```
val progressBar: Boolean = true
```

Compliant Code:

```
val hasProgressBar: Boolean = true
```

ClassNaming

Reports class or object names that do not follow the specified naming convention.

Active by default: Yes - Since v1.0.0

Debt: 5min

Aliases: ClassName

ConstructorParameterNaming

Reports constructor parameter names that do not follow the specified naming convention.

Active by default: Yes - Since v1.0.0

Debt: 5min

EnumNaming

Reports enum names that do not follow the specified naming convention.

Active by default: Yes - Since v1.0.0

Debt: 5min

ForbiddenClassName

Reports class names which are forbidden per configuration. By default, this rule does not report any classes.
Examples for forbidden names might be too generic class names like `...Manager`.

Active by default: No

Debt: 5min

FunctionMaxLength

Reports when very long function names are used.

Active by default: No

Debt: 5min

FunctionMinLength

Reports when very short function names are used.

Active by default: No

Debt: 5min

FunctionNaming

Reports function names that do not follow the specified naming convention.
One exception are factory functions used to create instances of classes.
These factory functions can have the same name as the class being created.

Active by default: Yes - Since v1.0.0

Debt: 5min

Aliases: FunctionName

FunctionParameterNaming

Reports function parameter names that do not follow the specified naming convention.

Active by default: Yes - Since v1.0.0

Debt: 5min

InvalidPackageDeclaration

Reports when the file location does not match the declared package.

Active by default: Yes - Since v1.21.0

Debt: 5min

Aliases: PackageDirectoryMismatch

LambdaParameterNaming

Reports lambda parameter names that do not follow the specified naming convention.

Active by default: No

Debt: 5min

MatchingDeclarationName

"If a Kotlin file contains a single non-private class (potentially with related top-level declarations),
its name should be the same as the name of the class, with the .kt extension appended.
If a file contains multiple classes, or only top-level declarations,
choose a name describing what the file contains, and name the file accordingly.
Use camel humps with an uppercase first letter (e.g. ProcessDeclarations.kt).

The name of the file should describe what the code in the file does.
Therefore, you should avoid using meaningless words such as "Util" in file names." - Official Kotlin Style Guide

More information at: https://kotlinlang.org/docs/coding-conventions.html

Active by default: Yes - Since v1.0.0

Debt: 5min

Noncompliant Code:

```
class Foo // FooUtils.kt

fun Bar.toFoo(): Foo = ...
fun Foo.toBar(): Bar = ...
```

Compliant Code:

```
class Foo { // Foo.kt
    fun stuff() = 42
}

fun Bar.toFoo(): Foo = ...
```

MemberNameEqualsClassName

This rule reports a member that has the same as the containing class or object.
This might result in confusion.
The member should either be renamed or changed to a constructor.
Factory functions that create an instance of the class are exempt from this rule.

Active by default: Yes - Since v1.2.0

Debt: 5min

Noncompliant Code:

```
class MethodNameEqualsClassName {

    fun methodNameEqualsClassName() { }
}

class PropertyNameEqualsClassName {

    val propertyEqualsClassName = 0
}
```

Compliant Code:

```
class Manager {

    companion object {
        // factory functions can have the same name as the class
        fun manager(): Manager {
            return Manager()
        }
    }
}
```

NoNameShadowing

Disallows shadowing variable declarations.

Shadowing makes it impossible to access a variable with the same name in the scope.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
fun test(i: Int, j: Int, k: Int) {
    val i = 1
    val (j, _) = 1 to 2
    listOf(1).map { k -> println(k) }
    listOf(1).forEach {
        listOf(2).forEach {
        }
    }
}
```

Compliant Code:

```
fun test(i: Int, j: Int, k: Int) {
    val x = 1
    val (y, _) = 1 to 2
    listOf(1).map { z -> println(z) }
    listOf(1).forEach {
        listOf(2).forEach { x ->
        }
    }
}
```

NonBooleanPropertyPrefixedWithIs

Reports when property with 'is' prefix doesn't have a boolean type.

Please check the chapter 8.3.2 at Java Language Specification

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
val isEnabled: Int = 500
```

Compliant Code:

```
val isEnabled: Boolean = false
```

ObjectPropertyNaming

Reports property names inside objects that do not follow the specified naming convention.

Active by default: Yes - Since v1.0.0

Debt: 5min

PackageNaming

Reports package names that do not follow the specified naming convention.

Active by default: Yes - Since v1.0.0

Debt: 5min

Aliases: PackageName, PackageDirectoryMismatch

TopLevelPropertyNaming

Reports top level constant that which do not follow the specified naming convention.

Active by default: Yes - Since v1.0.0

Debt: 5min

VariableMaxLength

Reports when very long variable names are used.

Active by default: No

Debt: 5min

VariableMinLength

Reports when very short variable names are used.

Active by default: No

Debt: 5min

VariableNaming

Reports variable names that do not follow the specified naming convention.

Active by default: Yes - Since v1.0.0

Debt: 5min

Noncompliant Code:

```
listOf(1, 2, 3, 4).map { it*2 }.filter { it < 4 }.map { it*it }
```

Compliant Code:

```
listOf(1, 2, 3, 4).asSequence().map { it*2 }.filter { it < 4 }.map { it*it }.toList()

listOf(1, 2, 3, 4).map { it*2 }
```

ForEachOnRange

Using the forEach method on ranges has a heavy performance cost. Prefer using simple for loops.

Benchmarks have shown that using forEach on a range can have a huge performance cost in comparison to
simple for loops. Hence, in most contexts, a simple for loop should be used instead.
See more details here: https://sites.google.com/a/athaydes.com/renato-athaydes/posts/kotlinshiddencosts-benchmarks
To solve this CodeSmell, the forEach usage should be replaced by a for loop.

Active by default: Yes - Since v1.0.0

Debt: 5min

Noncompliant Code:

```
(1..10).forEach {
    println(it)
}
(1 until 10).forEach {
    println(it)
}
(10 downTo 1).forEach {
    println(it)
}
```

Compliant Code:

```
for (i in 1..10) {
    println(i)
}
```

SpreadOperator

In most cases using a spread operator causes a full copy of the array to be created before calling a method.
This has a very high performance penalty. Benchmarks showing this performance penalty can be seen here: https://sites.google.com/a/athaydes.com/renato-athaydes/posts/kotlinshiddencosts-benchmarks

The Kotlin compiler since v1.1.60 has an optimization that skips the array copy when an array constructor
function is used to create the arguments that are passed to the vararg parameter. When type resolution is enabled in
detekt this case will not be flagged by the rule since it doesn't suffer the performance penalty of an array copy.

Active by default: Yes - Since v1.0.0

Debt: 20min

Noncompliant Code:

```
val strs = arrayOf("value one", "value two")
val foo = bar(*strs)

fun bar(vararg strs: String) {
    strs.forEach { println(it) }
}
```

Compliant Code:

```
// array copy skipped in this case since Kotlin 1.1.60
val foo = bar(*arrayOf("value one", "value two"))

// array not passed so no array copy is required
val foo2 = bar("value one", "value two")

fun bar(vararg strs: String) {
    strs.forEach { println(it) }
}
```

UnnecessaryPartOfBinaryExpression

Unnecessary binary expression add complexity to the code and accomplish nothing. They should be removed.
The rule works with all binary expression included if and when condition. The rule also works with all predicates.
The rule verify binary expression only in case when the expression use only one type of the following
operators || or &&.

Active by default: No

Debt: 5min

Noncompliant Code:

```
val foo = true
val bar = true

if (foo || bar || foo) {
}
```

Compliant Code:

```
val foo = true
if (foo) {
}
```

UnnecessaryTemporaryInstantiation

Avoid temporary objects when converting primitive types to String. This has a performance penalty when compared
to using primitive types directly.
To solve this issue, remove the wrapping type.

Active by default: Yes - Since v1.0.0

Debt: 5min

Noncompliant Code:

```
val i = Integer(1).toString() // temporary Integer instantiation just for the conversion
```

Compliant Code:

```
val i = Integer.toString(1)
```

- `forbiddenTypePatterns` (default: `['kotlin.String']`)

  Specifies those types for which referential equality checks are considered a rule violation. The types are defined by a list of simple glob patterns (supporting `*` and `?` wildcards) that match the fully qualified type name.

Noncompliant Code:

```
val areEqual = "aString" === otherString
val areNotEqual = "aString" !== otherString
```

Compliant Code:

```
val areEqual = "aString" == otherString
val areNotEqual = "aString" != otherString
```

CastNullableToNonNullableType

Reports cast of nullable variable to non-null type. Cast like this can hide `null` problems in your code. The compliant code would be that which will correctly check for two things (nullability and type) and not just one (cast).

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
fun foo(bar: Any?) {
    val x = bar as String
}
```

Compliant Code:

```
fun foo(bar: Any?) {
    val x = checkNotNull(bar) as String
}

// Alternative
fun foo(bar: Any?) {
    val x = (bar ?: error("null assertion message")) as String
}
```

CastToNullableType

Reports unsafe cast to nullable types.
`as String?` is unsafed and may be misused as safe cast (`as? String`).

Active by default: No

Debt: 5min

Noncompliant Code:

```
fun foo(a: Any?) {
    val x: String? = a as String? // If 'a' is not String, ClassCastException will be thrown.
}
```

Compliant Code:

```
fun foo(a: Any?) {
    val x: String? = a as? String
}
```

Deprecation

Deprecated elements are expected to be removed in the future. Alternatives should be found if possible.

Active by default: No

Requires Type Resolution

Debt: 20min

Aliases: DEPRECATION

DontDowncastCollectionTypes

Down-casting immutable types from kotlin.collections should be discouraged.
The result of the downcast is platform specific and can lead to unexpected crashes.
Prefer to use instead the `toMutable<Type>()` functions.

Active by default: No

Requires Type Resolution

Debt: 10min

Noncompliant Code:

```
val list : List<Int> = getAList()
if (list is MutableList) {
    list.add(42)
}

(list as MutableList).add(42)
```

Compliant Code:

```
val list : List<Int> = getAList()
list.toMutableList().add(42)
```

DoubleMutabilityForCollection

Using `var` when declaring a mutable collection or value holder leads to double mutability.
Consider instead declaring your variable with `val` or switching your declaration to use an
immutable type.

By default, the rule triggers on standard mutable collections, however it can be configured
to trigger on other types of mutable value types, such as `MutableState` from Jetpack Compose.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Aliases: DoubleMutability

Noncompliant Code:

```
var myList = mutableListOf(1,2,3)
var mySet = mutableSetOf(1,2,3)
var myMap = mutableMapOf("answer" to 42)
```

Compliant Code:

```
// Use val
val myList = mutableListOf(1,2,3)
val mySet = mutableSetOf(1,2,3)
val myMap = mutableMapOf("answer" to 42)

// Use immutable types
var myList = listOf(1,2,3)
var mySet = setOf(1,2,3)
var myMap = mapOf("answer" to 42)
```

~~DuplicateCaseInWhenExpression~~

Rule deprecated as compiler performs this check by default

Flags duplicate `case` statements in `when` expressions.

If a `when` expression contains the same `case` statement multiple times they should be merged. Otherwise, it might be
easy to miss one of the cases when reading the code, leading to unwanted side effects.

Active by default: Yes - Since v1.0.0

Debt: 10min

Noncompliant Code:

```
when (i) {
    1 -> println("one")
    1 -> println("one")
    else -> println("else")
}
```

Compliant Code:

```
when (i) {
    1 -> println("one")
    else -> println("else")
}
```

ElseCaseInsteadOfExhaustiveWhen

This rule reports `when` expressions that contain an `else` case even though they have an exhaustive set of cases.

This occurs when the subject of the `when` expression is either an enum class, sealed class or of type boolean.
Using `else` cases for these expressions can lead to unintended behavior when adding new enum types, sealed subtypes
or changing the nullability of a boolean, since this will be implicitly handled by the `else` case.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
enum class Color {
    RED,
    GREEN,
    BLUE
}

when(c) {
    Color.RED -> {}
    Color.GREEN -> {}
    else -> {}
}
```

Compliant Code:

```
enum class Color {
    RED,
    GREEN,
    BLUE
}

when(c) {
    Color.RED -> {}
    Color.GREEN -> {}
    Color.BLUE -> {}
}
```

EqualsAlwaysReturnsTrueOrFalse

Reports `equals()` methods which will always return true or false.

Equals methods should always report if some other object is equal to the current object.
See the Kotlin documentation for Any.equals(other: Any?):
https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-any/equals.html

Active by default: Yes - Since v1.2.0

Debt: 20min

Noncompliant Code:

```
override fun equals(other: Any?): Boolean {
    return true
}
```

Compliant Code:

```
override fun equals(other: Any?): Boolean {
    return this === other
}
```

EqualsWithHashCodeExist

When a class overrides the equals() method it should also override the hashCode() method.

All hash-based collections depend on objects meeting the equals-contract. Two equal objects must produce the
same hashcode. When inheriting equals or hashcode, override the inherited and call the super method for
clarification.

Active by default: Yes - Since v1.0.0

Debt: 5min

Noncompliant Code:

```
class Foo {

    override fun equals(other: Any?): Boolean {
        return super.equals(other)
    }
}
```

Compliant Code:

```
class Foo {

    override fun equals(other: Any?): Boolean {
        return super.equals(other)
    }

    override fun hashCode(): Int {
        return super.hashCode()
    }
}
```

ExitOutsideMain

Reports the usage of `System.exit()`, `Runtime.exit()`, `Runtime.halt()` and Kotlin's `exitProcess()`
when used outside the `main` function.
This makes code more difficult to test, causes unexpected behaviour on Android, and is a poor way to signal a
failure in the program. In almost all cases it is more appropriate to throw an exception.

Active by default: No

Requires Type Resolution

Debt: 10min

Noncompliant Code:

```
fun randomFunction() {
    val result = doWork()
    if (result == FAILURE) {
        exitProcess(2)
    } else {
        exitProcess(0)
    }
}
```

Compliant Code:

```
fun main() {
    val result = doWork()
    if (result == FAILURE) {
        exitProcess(2)
    } else {
        exitProcess(0)
    }
}
```

ExplicitGarbageCollectionCall

Reports all calls to explicitly trigger the Garbage Collector.
Code should work independently of the garbage collector and should not require the GC to be triggered in certain
points in time.

Active by default: Yes - Since v1.0.0

Debt: 20min

Noncompliant Code:

```
System.gc()
Runtime.getRuntime().gc()
System.runFinalization()
```

HasPlatformType

Platform types must be declared explicitly in public APIs to prevent unexpected errors.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
class Person {
    fun apiCall() = System.getProperty("propertyName")
}
```

Compliant Code:

```
class Person {
    fun apiCall(): String = System.getProperty("propertyName")
}
```

IgnoredReturnValue

This rule warns on instances where a function, annotated with either `@CheckReturnValue` or `@CheckResult`,
returns a value but that value is not used in any way. The Kotlin compiler gives no warning for this scenario
normally so that's the rationale behind this rule.

fun returnsValue() = 42
fun returnsNoValue() {}

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 20min

Noncompliant Code:

```
returnsValue()
```

Compliant Code:

```
if (42 == returnsValue()) {}
val x = returnsValue()
```

ImplicitDefaultLocale

Prefer passing [java.util.Locale] explicitly than using implicit default value when formatting
strings or performing a case conversion.

The default locale is almost always inappropriate for machine-readable text like HTTP headers.
For example, if locale with tag `ar-SA-u-nu-arab` is a current default then `%d` placeholders
will be evaluated to a number consisting of Eastern-Arabic (non-ASCII) digits.
[java.util.Locale.US] is recommended for machine-readable output.

Active by default: Yes - Since v1.16.0

Debt: 5min

Noncompliant Code:

```
String.format("Timestamp: %d", System.currentTimeMillis())
```

Compliant Code:

```
String.format(Locale.US, "Timestamp: %d", System.currentTimeMillis())
```

ImplicitUnitReturnType

Functions using expression statements have an implicit return type.
Changing the type of the expression accidentally, changes the functions return type.
This may lead to backward incompatibility.
Use a block statement to make clear this function will never return a value.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
fun errorProneUnit() = println("Hello Unit")
fun errorProneUnitWithParam(param: String) = param.run { println(this) }
fun String.errorProneUnitWithReceiver() = run { println(this) }
```

Compliant Code:

```
fun blockStatementUnit() {
    // code
}

// explicit Unit is compliant by default; can be configured to enforce block statement
fun safeUnitReturn(): Unit = println("Hello Unit")
```

InvalidRange

Reports ranges which are empty.
This might be a bug if it is used for instance as a loop condition. This loop will never be triggered then.
This might be due to invalid ranges like (10..9) which will cause the loop to never be entered.

Active by default: Yes - Since v1.2.0

Debt: 10min

Noncompliant Code:

```
for (i in 2..1) {}
for (i in 1 downTo 2) {}

val range1 = 2 until 1
val range2 = 2 until 2
```

Compliant Code:

```
for (i in 2..2) {}
for (i in 2 downTo 2) {}

val range = 2 until 3
```

IteratorHasNextCallsNextMethod

Verifies implementations of the Iterator interface.
The hasNext() method of an Iterator implementation should not have any side effects.
This rule reports implementations that call the next() method of the Iterator inside the hasNext() method.

Active by default: Yes - Since v1.2.0

Debt: 10min

Noncompliant Code:

```
class MyIterator : Iterator<String> {

    override fun hasNext(): Boolean {
        return next() != null
    }
}
```

IteratorNotThrowingNoSuchElementException

Reports implementations of the `Iterator` interface which do not throw a NoSuchElementException in the
implementation of the next() method. When there are no more elements to return an Iterator should throw a
NoSuchElementException.

See: https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html#next()

Active by default: Yes - Since v1.2.0

Debt: 10min

Noncompliant Code:

```
class MyIterator : Iterator<String> {

    override fun next(): String {
        return ""
    }
}
```

Compliant Code:

```
class MyIterator : Iterator<String> {

    override fun next(): String {
        if (!this.hasNext()) {
            throw NoSuchElementException()
        }
        // ...
    }
}
```

LateinitUsage

Reports usages of the lateinit modifier.

Using lateinit for property initialization can be error-prone and the actual initialization is not
guaranteed. Try using constructor injection or delegation to initialize properties.

Active by default: No

Debt: 20min

Noncompliant Code:

```
class Foo {
    private lateinit var i1: Int
    lateinit var i2: Int
}
```

MapGetWithNotNullAssertionOperator

Reports calls of the map access methods `map[]` or `map.get()` with a not-null assertion operator `!!`.
This may result in a NullPointerException.
Preferred access methods are `map[]` without `!!`, `map.getValue()`, `map.getOrDefault()` or `map.getOrElse()`.

Based on an IntelliJ IDEA inspection MapGetWithNotNullAssertionOperatorInspection.

Active by default: Yes - Since v1.21.0

Debt: 5min

Noncompliant Code:

```
val map = emptyMap<String, String>()
map["key"]!!

val map = emptyMap<String, String>()
map.get("key")!!
```

Compliant Code:

```
val map = emptyMap<String, String>()
map["key"]

val map = emptyMap<String, String>()
map.getValue("key")

val map = emptyMap<String, String>()
map.getOrDefault("key", "")

val map = emptyMap<String, String>()
map.getOrElse("key", { "" })
```

MissingPackageDeclaration

Reports when the package declaration is missing.

Active by default: No

Debt: 5min

~~MissingWhenCase~~

Rule deprecated as compiler performs this check by default

Turn on this rule to flag `when` expressions that do not check that all cases are covered when the subject is an enum
or sealed class and the `when` expression is used as a statement.

When this happens it's unclear what was intended when an unhandled case is reached. It is better to be explicit and
either handle all cases or use a default `else` statement to cover the unhandled cases.

Active by default: Yes - Since v1.2.0

Requires Type Resolution

Debt: 20min

Noncompliant Code:

```
enum class Color {
    RED,
    GREEN,
    BLUE
}

fun whenOnEnumFail(c: Color) {
    when(c) {
        Color.BLUE -> {}
        Color.GREEN -> {}
    }
}
```

Compliant Code:

```
enum class Color {
    RED,
    GREEN,
    BLUE
}

fun whenOnEnumCompliant(c: Color) {
    when(c) {
        Color.BLUE -> {}
        Color.GREEN -> {}
        Color.RED -> {}
    }
}

fun whenOnEnumCompliant2(c: Color) {
    when(c) {
        Color.BLUE -> {}
        else -> {}
    }
}
```

NullCheckOnMutableProperty

Reports null-checks on mutable properties, as these properties' value can be changed - and thus make the null-check invalid - after the execution of the if-statement.

Active by default: No

Requires Type Resolution

Debt: 10min

Noncompliant Code:

```
class A(private var a: Int?) {
fun foo() {
    if (a != null) {
      println(2 + a!!)
    }
}
}
```

Compliant Code:

```
class A(private val a: Int?) {
fun foo() {
    if (a != null) {
      println(2 + a)
    }
}
}
```

NullableToStringCall

Reports `toString()` calls with a nullable receiver that may return the string "null".

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
fun foo(a: Any?): String {
    return a.toString()
}

fun bar(a: Any?): String {
    return "$a"
}
```

Compliant Code:

```
fun foo(a: Any?): String {
    return a?.toString() ?: "-"
}

fun bar(a: Any?): String {
    return "${a ?: "-"}"
}
```

PropertyUsedBeforeDeclaration

Reports properties that are used before declaration.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
class C {
    private val number
        get() = if (isValid) 1 else 0

    val list = listOf(number)

    private val isValid = true
}

fun main() {
    println(C().list) // [0]
}
```

Compliant Code:

```
class C {
    private val isValid = true

    private val number
        get() = if (isValid) 1 else 0

    val list = listOf(number)
}

fun main() {
    println(C().list) // [1]
}
```

~~RedundantElseInWhen~~

Rule deprecated as compiler performs this check by default

Reports `when` expressions that contain a redundant `else` case. This occurs when it can be
verified that all cases are already covered when checking cases on an enum or sealed class.

Active by default: Yes - Since v1.2.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
enum class Color {
    RED,
    GREEN,
    BLUE
}

fun whenOnEnumFail(c: Color) {
    when(c) {
        Color.BLUE -> {}
        Color.GREEN -> {}
        Color.RED -> {}
        else -> {}
    }
}
```

Compliant Code:

```
enum class Color {
    RED,
    GREEN,
    BLUE
}

fun whenOnEnumCompliant(c: Color) {
    when(c) {
        Color.BLUE -> {}
        Color.GREEN -> {}
        else -> {}
    }
}

fun whenOnEnumCompliant2(c: Color) {
    when(c) {
        Color.BLUE -> {}
        Color.GREEN -> {}
        Color.RED -> {}
    }
}
```

UnconditionalJumpStatementInLoop

Reports loops which contain jump statements that jump regardless of any conditions. This implies that the loop is only executed once and thus could be rewritten without a loop altogether.

Active by default: No

Debt: 10min

Noncompliant Code:

```
for (i in 1..2) break
```

Compliant Code:

```
for (i in 1..2) {
    if (i == 1) break
}
```

UnnecessaryNotNullCheck

Reports unnecessary not-null checks with `requireNotNull` or `checkNotNull` that can be removed by the user.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
var string = "foo"
println(requireNotNull(string))
```

Compliant Code:

```
var string : String? = "foo"
println(requireNotNull(string))
```

UnnecessaryNotNullOperator

Reports unnecessary not-null operator usage (!!) that can be removed by the user.

Active by default: Yes - Since v1.16.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
val a = 1
val b = a!!
```

Compliant Code:

```
val a = 1
val b = a
```

UnnecessarySafeCall

Reports unnecessary safe call operators (`?.`) that can be removed by the user.

Active by default: Yes - Since v1.16.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
val a: String = ""
val b = a?.length
```

Compliant Code:

```
val a: String? = null
val b = a?.length
```

UnreachableCatchBlock

Reports unreachable catch blocks.
Catch blocks can be unreachable if the exception has already been caught in the block above.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
fun test() {
    try {
        foo()
    } catch (t: Throwable) {
        bar()
    } catch (e: Exception) {
        // Unreachable
        baz()
    }
}
```

Compliant Code:

```
fun test() {
    try {
        foo()
    } catch (e: Exception) {
        baz()
    } catch (t: Throwable) {
        bar()
    }
}
```

UnreachableCode

Reports unreachable code.
Code can be unreachable because it is behind return, throw, continue or break expressions.
This unreachable code should be removed as it serves no purpose.

Active by default: Yes - Since v1.0.0

Requires Type Resolution

Debt: 10min

Noncompliant Code:

```
for (i in 1..2) {
    break
    println() // unreachable
}

throw IllegalArgumentException()
println() // unreachable

fun f() {
    return
    println() // unreachable
}
```

UnsafeCallOnNullableType

Reports unsafe calls on nullable types. These calls will throw a NullPointerException in case
the nullable value is null. Kotlin provides many ways to work with nullable types to increase
null safety. Guard the code appropriately to prevent NullPointerExceptions.

Active by default: Yes - Since v1.2.0

Requires Type Resolution

Debt: 20min

Noncompliant Code:

```
fun foo(str: String?) {
    println(str!!.length)
}
```

Compliant Code:

```
fun foo(str: String?) {
    println(str?.length)
}
```

UnsafeCast

Reports casts that will never succeed.

Active by default: Yes - Since v1.16.0

Requires Type Resolution

Debt: 20min

Aliases: UNCHECKED_CAST

Noncompliant Code:

```
fun foo(s: String) {
    println(s as Int)
}

fun bar(s: String) {
    println(s as? Int)
}
```

Compliant Code:

```
fun foo(s: Any) {
    println(s as Int)
}
```

UnusedUnaryOperator

Detects unused unary operators.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
val x = 1 + 2
    + 3 + 4
println(x) // 3
```

Compliant Code:

```
val x = 1 + 2 + 3 + 4
println(x) // 10
```

UselessPostfixExpression

Reports postfix expressions (++, --) which are unused and thus unnecessary.
This leads to confusion as a reader of the code might think the value will be incremented/decremented.
However, the value is replaced with the original value which might lead to bugs.

Active by default: Yes - Since v1.21.0

Debt: 20min

Noncompliant Code:

```
var i = 0
i = i--
i = 1 + i++
i = i++ + 1

fun foo(): Int {
    var i = 0
    // ...
    return i++
}
```

Compliant Code:

```
var i = 0
i--
i = i + 2
i = i + 2

fun foo(): Int {
    var i = 0
    // ...
    i++
    return i
}
```

WrongEqualsTypeParameter

Reports equals() methods which take in a wrongly typed parameter.

Correct implementations of the equals() method should only take in a parameter of type Any?

See: https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-any/equals.html

Active by default: Yes - Since v1.2.0

Debt: 10min

Noncompliant Code:

```
class Foo {

    fun equals(other: String): Boolean {
        return super.equals(other)
    }
}
```

Compliant Code:

```
class Foo {

    fun equals(other: Any?): Boolean {
        return super.equals(other)
    }
}
```

UseEntityAtName

If a rule [report]s issues using [Entity.from] with [KtNamedDeclaration.getNameIdentifier],
then it can be replaced with [Entity.atName] for more semantic code and better baseline support.

Active by default: Yes - Since v1.22.0

Debt: 5min

ViolatesTypeResolutionRequirements

If a rule uses the property [BaseRule.bindingContext] should be annotated with `@RequiresTypeResolution`.
And if the rule doesn't use that property it shouldn't be annotated with it.

Active by default: Yes - Since v1.22.0

Requires Type Resolution

Debt: 5min

- `singleLine` (default: `'never'`)

  single-line braces policy

- `multiLine` (default: `'always'`)

  multi-line braces policy

Noncompliant Code:

```
// singleLine = 'never'
if (a) { b } else { c }

if (a) { b } else c

if (a) b else { c; d }

// multiLine = 'never'
if (a) {
    b
} else {
    c
}

// singleLine = 'always'
if (a) b else c

if (a) { b } else c

// multiLine = 'always'
if (a) {
    b
} else
    c

// singleLine = 'consistent'
if (a) b else { c }
if (a) b else if (c) d else { e }

// multiLine = 'consistent'
if (a)
    b
else {
    c
}

// singleLine = 'necessary'
if (a) { b } else { c; d }

// multiLine = 'necessary'
if (a) {
    b
    c
} else if (d) {
    e
} else {
    f
}
```

Compliant Code:

```
// singleLine = 'never'
if (a) b else c

// multiLine = 'never'
if (a)
    b
else
    c

// singleLine = 'always'
if (a) { b } else { c }

if (a) { b } else if (c) { d }

// multiLine = 'always'
if (a) {
    b
} else {
    c
}

if (a) {
    b
} else if (c) {
    d
}

// singleLine = 'consistent'
if (a) b else c

if (a) { b } else { c }

if (a) { b } else { c; d }

// multiLine = 'consistent'
if (a) {
    b
} else {
    c
}

if (a) b
else c

// singleLine = 'necessary'
if (a) b else { c; d }

// multiLine = 'necessary'
if (a) {
    b
    c
} else if (d)
    e
else
    f
```

BracesOnWhenStatements

This rule detects `when` statements which do not comply with the specified policy. Keeping braces consistent will improve readability and avoid possible errors.
Single-line `when` statement is: a `when` where each of the branches are single-line (has no line breaks `\n`).
Multi-line `when` statement is: a `when` where at least one of the branches is multi-line (has a break line `\n`).

Available options are:

- `never`: forces no braces on any branch.
  Tip: this is very strict, it will force a simple expression, like a single function call / expression.
  Extracting a function for "complex" logic is one way to adhere to this policy.
- `necessary`: forces no braces on any branch except where necessary for multi-statement branches.
- `consistent`: ensures that braces are consistent within `when` statement.
  If there are braces on one of the branches, all branches should have it.
- `always`: forces braces on all branches.

Active by default: No

Debt: 5min

Noncompliant Code:

```
// singleLine = 'never'
when (a) {
    1 -> { f1() } // Not allowed.
    2 -> f2()
}
// multiLine = 'never'
when (a) {
    1 -> { // Not allowed.
        f1()
    }
    2 -> f2()
}
// singleLine = 'necessary'
when (a) {
    1 -> { f1() } // Unnecessary braces.
    2 -> f2()
}
// multiLine = 'necessary'
when (a) {
    1 -> { // Unnecessary braces.
        f1()
    }
    2 -> f2()
}

// singleLine = 'consistent'
when (a) {
    1 -> { f1() }
    2 -> f2()
}
// multiLine = 'consistent'
when (a) {
    1 ->
        f1() // Missing braces.
    2 -> {
        f2()
        f3()
    }
}

// singleLine = 'always'
when (a) {
    1 -> { f1() }
    2 -> f2() // Missing braces.
}
// multiLine = 'always'
when (a) {
    1 ->
        f1() // Missing braces.
    2 -> {
        f2()
        f3()
    }
}
```

Compliant Code:

```
// singleLine = 'never'
when (a) {
    1 -> f1()
    2 -> f2()
}
// multiLine = 'never'
when (a) {
    1 ->
        f1()
    2 -> f2()
}
// singleLine = 'necessary'
when (a) {
    1 -> f1()
    2 -> { f2(); f3() } // Necessary braces because of multiple statements.
}
// multiLine = 'necessary'
when (a) {
    1 ->
        f1()
    2 -> { // Necessary braces because of multiple statements.
        f2()
        f3()
    }
}

// singleLine = 'consistent'
when (a) {
    1 -> { f1() }
    2 -> { f2() }
}
when (a) {
    1 -> f1()
    2 -> f2()
}
// multiLine = 'consistent'
when (a) {
    1 -> {
        f1()
    }
    2 -> {
        f2()
        f3()
    }
}

// singleLine = 'always'
when (a) {
    1 -> { f1() }
    2 -> { f2() }
}
// multiLine = 'always'
when (a) {
    1 -> {
        f1()
    }
    2 -> {
        f2()
        f3()
    }
}
```

CanBeNonNullable

This rule inspects variables marked as nullable and reports which could be declared as non-nullable instead.

It's preferred to not have functions that do "nothing".
A function that does nothing when the value is null hides the logic,
so it should not allow null values in the first place.
It is better to move the null checks up around the calls,
instead of having it inside the function.

This could lead to less nullability overall in the codebase.

Active by default: No

Requires Type Resolution

Debt: 10min

Noncompliant Code:

```
class A {
    var a: Int? = 5

    fun foo() {
        a = 6
    }
}

class A {
    val a: Int?
        get() = 5
}

fun foo(a: Int?) {
    val b = a!! + 2
}

fun foo(a: Int?) {
    if (a != null) {
        println(a)
    }
}

fun foo(a: Int?) {
    if (a == null) return
    println(a)
}
```

Compliant Code:

```
class A {
    var a: Int = 5

    fun foo() {
        a = 6
    }
}

class A {
    val a: Int
        get() = 5
}

fun foo(a: Int) {
    val b = a + 2
}

fun foo(a: Int) {
    println(a)
}
```

CascadingCallWrapping

Requires that all chained calls are placed on a new line if a preceding one is.

Active by default: No

Debt: 5min

Noncompliant Code:

```
foo()
.bar().baz()
```

Compliant Code:

```
foo().bar().baz()

foo()
.bar()
.baz()
```

ClassOrdering

This rule ensures class contents are ordered as follows as recommended by the Kotlin Coding Conventions:

- Property declarations and initializer blocks
- Secondary constructors
- Method declarations
- Companion object

Active by default: No

Debt: 5min

Noncompliant Code:

```
class OutOfOrder {
    companion object {
        const val IMPORTANT_VALUE = 3
    }

    fun returnX(): Int {
        return x
    }

    private val x = 2
}
```

Compliant Code:

```
class InOrder {
    private val x = 2

    fun returnX(): Int {
        return x
    }

    companion object {
        const val IMPORTANT_VALUE = 3
    }
}
```

CollapsibleIfStatements

This rule detects `if` statements which can be collapsed. This can reduce nesting and help improve readability.

However, carefully consider whether merging the if statements actually improves readability, as collapsing the
statements may hide some edge cases from the reader.

Active by default: No

Debt: 5min

Noncompliant Code:

```
val i = 1
if (i > 0) {
    if (i < 5) {
        println(i)
    }
}
```

Compliant Code:

```
val i = 1
if (i > 0 && i < 5) {
    println(i)
}
```

DataClassContainsFunctions

This rule reports functions inside data classes which have not been marked as a conversion function.

Data classes should mainly be used to store data. This rule assumes that they should not contain any extra functions
aside functions that help with converting objects from/to one another.
Data classes will automatically have a generated `equals`, `toString` and `hashCode` function by the compiler.

Active by default: No

Debt: 20min

Noncompliant Code:

```
data class DataClassWithFunctions(val i: Int) {
    fun foo() { }
}
```

DataClassShouldBeImmutable

This rule reports mutable properties inside data classes.

Data classes should mainly be used to store immutable data. This rule assumes that they should not contain any
mutable properties.

Active by default: No

Debt: 20min

Noncompliant Code:

```
data class MutableDataClass(var i: Int) {
    var s: String? = null
}
```

Compliant Code:

```
data class ImmutableDataClass(
    val i: Int,
    val s: String?
)
```

DestructuringDeclarationWithTooManyEntries

Destructuring declarations with too many entries are hard to read and understand.
To increase readability they should be refactored to reduce the number of entries or avoid using a destructuring
declaration.

Active by default: Yes - Since v1.21.0

Debt: 10min

Noncompliant Code:

```
data class TooManyElements(val a: Int, val b: Int, val c: Int, val d: Int)
val (a, b, c, d) = TooManyElements(1, 2, 3, 4)
```

Compliant Code:

```
data class FewerElements(val a: Int, val b: Int, val c: Int)
val (a, b, c) = TooManyElements(1, 2, 3)
```

DoubleNegativeLambda

Detects negation in lambda blocks where the function name is also in the negative (like `takeUnless`).
A double negative is harder to read than a positive. In particular, if there are multiple conditions with `&&` etc. inside
the lambda, then the reader may need to unpack these using DeMorgan's laws.
Consider rewriting the lambda to use a positive version
of the function (like `takeIf`).

Active by default: No

Debt: 5min

Noncompliant Code:

```
fun Int.evenOrNull() = takeUnless { it % 2 != 0 }
```

Compliant Code:

```
fun Int.evenOrNull() = takeIf { it % 2 == 0 }
```

EqualsNullCall

To compare an object with `null` prefer using `==`. This rule detects and reports instances in the code where the `equals()` method is used to compare a value with `null`.

Active by default: Yes - Since v1.2.0

Debt: 5min

Noncompliant Code:

```
fun isNull(str: String) = str.equals(null)
```

Compliant Code:

```
fun isNull(str: String) = str == null
```

EqualsOnSignatureLine

Requires that the equals sign, when used for an expression style function, is on the same line as the
rest of the function signature.

Active by default: No

Debt: 5min

Noncompliant Code:

```
fun stuff(): Int
    = 5

fun <V> foo(): Int where V : Int
    = 5
```

Compliant Code:

```
fun stuff() = 5

fun stuff() =
    foo.bar()

fun <V> foo(): Int where V : Int = 5
```

ExplicitCollectionElementAccessMethod

In Kotlin functions `get` or `set` can be replaced with the shorter operator — `[]`, see Indexed access operator.
Prefer the usage of the indexed access operator `[]` for map or list element access or insert methods.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
val map = mutableMapOf<String, String>()
map.put("key", "value")
val value = map.get("key")
```

Compliant Code:

```
val map = mutableMapOf<String, String>()
map["key"] = "value"
val value = map["key"]
```

ExplicitItLambdaParameter

Lambda expressions are one of the core features of the language. They often include very small chunks of
code using only one parameter. In this cases Kotlin can supply the implicit `it` parameter
to make code more concise. It fits most use cases, but when faced larger or nested chunks of code,
you might want to add an explicit name for the parameter. Naming it just `it` is meaningless and only
makes your code misleading, especially when dealing with nested functions.

Active by default: Yes - Since v1.21.0

Debt: 5min

Noncompliant Code:

```
a?.let { it -> it.plus(1) }
foo.flatMapObservable { it -> Observable.fromIterable(it) }
listOfPairs.map(::second).forEach { it ->
    it.execute()
}
collection.zipWithNext { it, next -> Pair(it, next) }
```

Compliant Code:

```
a?.let { it.plus(1) } // Much better to use implicit it
foo.flatMapObservable(Observable::fromIterable) // Here we can have a method reference

// For multiline blocks it is usually better come up with a clear and more meaningful name
listOfPairs.map(::second).forEach { apiRequest ->
    apiRequest.execute()
}

// Lambdas with multiple parameter should be named clearly, using it for one of them can be confusing
collection.zipWithNext { prev, next ->
    Pair(prev, next)
}
```

ExpressionBodySyntax

Functions which only contain a `return` statement can be collapsed to an expression body. This shortens and
cleans up the code.

Active by default: No

Debt: 5min

Noncompliant Code:

```
fun stuff(): Int {
    return 5
}
```

Compliant Code:

```
fun stuff() = 5

fun stuff() {
    return
        moreStuff()
            .getStuff()
            .stuffStuff()
}
```

ForbiddenAnnotation

This rule allows to set a list of forbidden annotations. This can be used to discourage the use
of language annotations which do not require explicit import.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
@SuppressWarnings("unused")
class SomeClass()
```

Compliant Code:

```
@Suppress("unused")
class SomeClass()
```

ForbiddenComment

This rule allows to set a list of comments which are forbidden in the codebase and should only be used during
development. Offending code comments will then be reported.

The regular expressions in `comments` list will have the following behaviors while matching the comments:

- **Each comment will be handled individually.**
  - single line comments are always separate, consecutive lines are not merged.
  - multi line comments are not split up, the regex will be applied to the whole comment.
  - KDoc comments are not split up, the regex will be applied to the whole comment.

- **The following comment delimiters (and indentation before them) are removed** before applying the regex:
  `//`, `// `, `/*`, `/* `, `/**`, `*` aligners, `*/`, ` */`

- **The regex is applied as a multiline regex**,
  see Anchors for more info.
  To match the start and end of each line, use `^` and `$`.
  To match the start and end of the whole comment, use `\A` and `\Z`.
  To turn off multiline, use `(?-m)` at the start of your regex.

- **The regex is applied with dotall semantics**, meaning `.` will match any character including newlines,
  this is to ensure that freeform line-wrapping doesn't mess with simple regexes.
  To turn off this behavior, use `(?-s)` at the start of your regex, or use `[^\r\n]*` instead of `.*`.

- **The regex will be searched using "contains" semantics** not "matches", so partial comment matches will flag forbidden comments.
  In practice this means there's no need to start and end the regex with `.*`.

Active by default: Yes - Since v1.0.0

Debt: 10min

Noncompliant Code:

```
val a = "" // TODO: remove please
/**
* FIXME: this is a hack
*/
fun foo() { }
/* STOPSHIP: */
```

ForbiddenImport

Reports all imports that are forbidden.

This rule allows to set a list of forbidden [imports].
This can be used to discourage the use of unstable, experimental or deprecated APIs.

Active by default: No

Debt: 10min

Noncompliant Code:

```
import kotlin.jvm.JvmField
import kotlin.SinceKotlin
```

ForbiddenMethodCall

Reports all method or constructor invocations that are forbidden.

This rule allows to set a list of forbidden [methods] or constructors. This can be used to discourage the use
of unstable, experimental or deprecated methods, especially for methods imported from external libraries.

Active by default: No

Requires Type Resolution

Debt: 10min

Noncompliant Code:

```
import java.lang.System
fun main() {
    System.gc()
    System::gc
}
```

ForbiddenSuppress

Report suppressions of all forbidden rules.

This rule allows to set a list of [rules] whose suppression is forbidden.
This can be used to discourage the abuse of the `Suppress` and `SuppressWarnings` annotations.

This rule is not capable of reporting suppression of itself, as that's a language feature with precedence.

Active by default: No

Debt: 10min

Noncompliant Code:

```
package foo

// When the rule "MaximumLineLength" is forbidden
@Suppress("MaximumLineLength", "UNCHECKED_CAST")
class Bar
```

Compliant Code:

```
package foo

// When the rule "MaximumLineLength" is forbidden
@Suppress("UNCHECKED_CAST")
class Bar
```

ForbiddenVoid

This rule detects usages of `Void` and reports them as forbidden.
The Kotlin type `Unit` should be used instead. This type corresponds to the `Void` class in Java
and has only one value - the `Unit` object.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
runnable: () -> Void
var aVoid: Void? = null
```

Compliant Code:

```
runnable: () -> Unit
Void::class
```

FunctionOnlyReturningConstant

A function that only returns a single constant can be misleading. Instead, prefer declaring the constant
as a `const val`.

Active by default: Yes - Since v1.2.0

Debt: 10min

Noncompliant Code:

```
fun functionReturningConstantString() = "1"
```

Compliant Code:

```
const val constantString = "1"
```

LoopWithTooManyJumpStatements

Loops which contain multiple `break` or `continue` statements are hard to read and understand.
To increase readability they should be refactored into simpler loops.

Active by default: Yes - Since v1.2.0

Debt: 10min

Noncompliant Code:

```
val strs = listOf("foo, bar")
for (str in strs) {
    if (str == "bar") {
        break
    } else {
        continue
    }
}
```

MagicNumber

This rule detects and reports usages of magic numbers in the code. Prefer defining constants with clear names
describing what the magic number means.

Active by default: Yes - Since v1.0.0

Debt: 10min

Noncompliant Code:

```
class User {

    fun checkName(name: String) {
        if (name.length > 42) {
            throw IllegalArgumentException("username is too long")
        }
        // ...
    }
}
```

Compliant Code:

```
class User {

    fun checkName(name: String) {
        if (name.length > MAX_USERNAME_SIZE) {
            throw IllegalArgumentException("username is too long")
        }
        // ...
    }

    companion object {
        private const val MAX_USERNAME_SIZE = 42
    }
}
```

MandatoryBracesLoops

This rule detects multi-line `for` and `while` loops which do not have braces. Adding braces would improve readability and avoid possible errors.

Active by default: No

Debt: 5min

Noncompliant Code:

```
for (i in 0..10)
    println(i)

while (true)
    println("Hello, world")

do
    println("Hello, world")
while (true)
```

Compliant Code:

```
for (i in 0..10) {
    println(i)
}

for (i in 0..10) println(i)

while (true) {
    println("Hello, world")
}

while (true) println("Hello, world")

do {
    println("Hello, world")
} while (true)

do println("Hello, world") while (true)
```

MaxChainedCallsOnSameLine

Limits the number of chained calls which can be placed on a single line.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
a().b().c().d().e().f()
```

Compliant Code:

```
a().b().c()
.d().e().f()
```

MaxLineLength

This rule reports lines of code which exceed a defined maximum line length.

Long lines might be hard to read on smaller screens or printouts. Additionally, having a maximum line length
in the codebase will help make the code more uniform.

Active by default: Yes - Since v1.0.0

Debt: 5min

MayBeConst

This rule identifies and reports properties (`val`) that may be `const val` instead. Using `const val` can lead to better performance of the resulting bytecode as well as better interoperability with
Java.

Active by default: Yes - Since v1.2.0

Debt: 5min

Aliases: MayBeConstant

Noncompliant Code:

```
val myConstant = "abc"
```

Compliant Code:

```
const val MY_CONSTANT = "abc"
```

ModifierOrder

This rule reports cases in the code where modifiers are not in the correct order. The default modifier order is
taken from: Modifiers order

Active by default: Yes - Since v1.0.0

Debt: 5min

Noncompliant Code:

```
lateinit internal val str: String
```

Compliant Code:

```
internal lateinit val str: String
```

MultilineLambdaItParameter

Lambda expressions are very useful in a lot of cases, and they often include very small chunks of
code using only one parameter. In this cases Kotlin can supply the implicit `it` parameter
to make code more concise. However, when you are dealing with lambdas that contain multiple statements,
you might end up with code that is hard to read if you don't specify a readable, descriptive parameter name
explicitly.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
val digits = 1234.let {
    println(it)
    listOf(it)
}

val digits = 1234.let { it ->
    println(it)
    listOf(it)
}

val flat = listOf(listOf(1), listOf(2)).mapIndexed { index, it ->
    println(it)
    it + index
}
```

Compliant Code:

```
val digits = 1234.let { explicitParameterName ->
    println(explicitParameterName)
    listOf(explicitParameterName)
}

val lambda = { item: Int, that: String ->
    println(item)
    item.toString() + that
}

val digits = 1234.let { listOf(it) }
val digits = 1234.let {
    listOf(it)
}
val digits = 1234.let { it -> listOf(it) }
val digits = 1234.let { it ->
    listOf(it)
}
val digits = 1234.let { explicit -> listOf(explicit) }
val digits = 1234.let { explicit ->
    listOf(explicit)
}
```

MultilineRawStringIndentation

This rule ensures that raw strings have a consistent indentation.

The content of a multi line raw string should have the same indentation as the enclosing expression plus the
configured indentSize. The closing triple-quotes (`"""`) must have the same indentation as the enclosing expression.

Active by default: No

Debt: 5min

Noncompliant Code:

```
val a = """
Hello World!
How are you?
""".trimMargin()

val a = """
    Hello World!
    How are you?
    """.trimMargin()
```

Compliant Code:

```
val a = """
    Hello World!
    How are you?
""".trimMargin()

val a = """
    Hello World!
      How are you?
""".trimMargin()
```

NestedClassesVisibility

Nested classes inherit their visibility from the parent class
and are often used to implement functionality local to the class it is nested in.
These nested classes can't have a higher visibility than their parent.
However, the visibility can be further restricted by using a private modifier for instance.
In internal classes the *explicit* public modifier for nested classes is misleading and thus unnecessary,
because the nested class still has an internal visibility.

Active by default: Yes - Since v1.16.0

Debt: 5min

Noncompliant Code:

```
internal class Outer {
    // explicit public modifier still results in an internal nested class
    public class Nested
}
```

Compliant Code:

```
internal class Outer {
    class Nested1
    internal class Nested2
}
```

NewLineAtEndOfFile

This rule reports files which do not end with a line separator.

Active by default: Yes - Since v1.0.0

Debt: 5min

NoTabs

This rule reports if tabs are used in Kotlin files.
According to
Google's Kotlin style guide
the only whitespace chars that are allowed in a source file are the line terminator sequence
and the ASCII horizontal space character (0x20). Strings containing tabs are allowed.

Active by default: No

Debt: 5min

NullableBooleanCheck

Detects nullable boolean checks which use an elvis expression `?:` rather than equals `==`.

Per the Kotlin coding conventions
converting a nullable boolean property to non-null should be done via `!= false` or `== true`
rather than `?: true` or `?: false` (respectively).

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
value ?: true
value ?: false
```

Compliant Code:

```
value != false
value == true
```

ObjectLiteralToLambda

An anonymous object that does nothing other than the implementation of a single method
can be used as a lambda.

See SAM conversions,
Functional (SAM) interfaces

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
object : Foo {
    override fun bar() {
    }
}
```

Compliant Code:

```
Foo {
}
```

OptionalAbstractKeyword

This rule reports `abstract` modifiers which are unnecessary and can be removed.

Active by default: Yes - Since v1.0.0

Debt: 5min

Noncompliant Code:

```
abstract interface Foo { // abstract keyword not needed

    abstract fun x() // abstract keyword not needed
    abstract var y: Int // abstract keyword not needed
}
```

Compliant Code:

```
interface Foo {

    fun x()
    var y: Int
}
```

OptionalUnit

It is not necessary to define a return type of `Unit` on functions or to specify a lone Unit statement.
This rule detects and reports instances where the `Unit` return type is specified on functions and the occurrences
of a lone Unit statement.

Active by default: No

Debt: 5min

Noncompliant Code:

```
fun foo(): Unit {
    return Unit
}
fun foo() = Unit

fun doesNothing() {
    Unit
}
```

Compliant Code:

```
fun foo() { }

// overridden no-op functions are allowed
override fun foo() = Unit
```

~~OptionalWhenBraces~~

Same functionality is implemented in BracesOnWhenStatements

This rule reports unnecessary braces in when expressions. These optional braces should be removed.

Active by default: No

Debt: 5min

Noncompliant Code:

```
val i = 1
when (i) {
    1 -> { println("one") } // unnecessary curly braces since there is only one statement
    else -> println("else")
}
```

Compliant Code:

```
val i = 1
when (i) {
    1 -> println("one")
    else -> println("else")
}
```

PreferToOverPairSyntax

This rule detects the usage of the Pair constructor to create pairs of values.

Using `<value1>` to `<value2>` is preferred.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
val pair = Pair(1, 2)
```

Compliant Code:

```
val pair = 1 to 2
```

ProtectedMemberInFinalClass

Kotlin classes are `final` by default. Thus classes which are not marked as `open` should not contain any `protected` members. Consider using `private` or `internal` modifiers instead.

Active by default: Yes - Since v1.2.0

Debt: 5min

Noncompliant Code:

```
class ProtectedMemberInFinalClass {
    protected var i = 0
}
```

Compliant Code:

```
class ProtectedMemberInFinalClass {
    private var i = 0
}
```

RedundantExplicitType

Local properties do not need their type to be explicitly provided when the inferred type matches the explicit type.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
fun function() {
    val x: String = "string"
}
```

Compliant Code:

```
fun function() {
    val x = "string"
}
```

RedundantHigherOrderMapUsage

Redundant maps add complexity to the code and accomplish nothing. They should be removed or replaced with the proper
operator.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
fun foo(list: List<Int>): List<Int> {
    return list
        .filter { it > 5 }
        .map { it }
}

fun bar(list: List<Int>): List<Int> {
    return list
        .filter { it > 5 }
        .map {
            doSomething(it)
            it
        }
}

fun baz(set: Set<Int>): List<Int> {
    return set.map { it }
}
```

Compliant Code:

```
fun foo(list: List<Int>): List<Int> {
    return list
        .filter { it > 5 }
}

fun bar(list: List<Int>): List<Int> {
    return list
        .filter { it > 5 }
        .onEach {
            doSomething(it)
        }
}

fun baz(set: Set<Int>): List<Int> {
    return set.toList()
}
```

RedundantVisibilityModifierRule

This rule checks for redundant visibility modifiers.
One exemption is the
explicit API mode
In this mode, the visibility modifier should be defined explicitly even if it is public.
Hence, the rule ignores the visibility modifiers in explicit API mode.

Active by default: No

Debt: 5min

Aliases: RedundantVisibilityModifier

Noncompliant Code:

```
public interface Foo { // public per default

    public fun bar() // public per default
}
```

Compliant Code:

```
interface Foo {

    fun bar()
}
```

ReturnCount

Restrict the number of return methods allowed in methods.

Having many exit points in a function can be confusing and impacts readability of the code.

Active by default: Yes - Since v1.0.0

Debt: 10min

Noncompliant Code:

```
fun foo(i: Int): String {
    when (i) {
        1 -> return "one"
        2 -> return "two"
        else -> return "other"
    }
}
```

Compliant Code:

```
fun foo(i: Int): String {
    return when (i) {
        1 -> "one"
        2 -> "two"
        else -> "other"
    }
}
```

SafeCast

This rule inspects casts and reports casts which could be replaced with safe casts instead.

Active by default: Yes - Since v1.0.0

Debt: 5min

Noncompliant Code:

```
fun numberMagic(number: Number) {
    val i = if (number is Int) number else null
    // ...
}
```

Compliant Code:

```
fun numberMagic(number: Number) {
    val i = number as? Int
    // ...
}
```

SerialVersionUIDInSerializableClass

Classes which implement the `Serializable` interface should also correctly declare a `serialVersionUID`.
This rule verifies that a `serialVersionUID` was correctly defined and declared as `private`.

More about `SerialVersionUID`

Active by default: Yes - Since v1.16.0

Debt: 5min

Noncompliant Code:

```
class IncorrectSerializable : Serializable {

    companion object {
        val serialVersionUID = 1 // wrong declaration for UID
    }
}
```

Compliant Code:

```
class CorrectSerializable : Serializable {

    companion object {
        const val serialVersionUID = 1L
    }
}
```

SpacingBetweenPackageAndImports

This rule verifies spacing between package and import statements as well as between import statements and class
declarations.

Active by default: No

Debt: 5min

Noncompliant Code:

```
package foo
import a.b
class Bar { }
```

Compliant Code:

```
package foo

import a.b

class Bar { }
```

StringShouldBeRawString

This rule reports when the string can be converted to Kotlin raw string.
Usage of a raw string is preferred as that avoids the need for escaping strings escape characters like \n, \t, ".
Raw string also allows us to represent multiline string without the need of \n.
Also, see Kotlin coding convention for
recommendation on using multiline strings

Active by default: No

Debt: 5min

Noncompliant Code:

```
val windowJson = "{\n" +
    "  \"window\": {\n" +
    "    \"title\": \"Sample Quantum With AI and ML Widget\",\n" +
    "    \"name\": \"main_window\",\n" +
    "    \"width\": 500,\n" +
    "    \"height\": 500\n" +
    "  }\n" +
    "}"

val patRegex = "/^(\\/[^\\/]+){0,2}\\/?\$/gm\n"
```

Compliant Code:

```
val windowJson = """
    {
        "window": {
            "title": "Sample Quantum With AI and ML Widget",
            "name": "main_window",
            "width": 500,
            "height": 500
        }
    }
""".trimIndent()

val patRegex = """/^(\/[^\/]+){0,2}\/?$/gm"""
```

ThrowsCount

Functions should have clear `throw` statements. Functions with many `throw` statements can be harder to read and lead
to confusion. Instead, prefer limiting the number of `throw` statements in a function.

Active by default: Yes - Since v1.0.0

Debt: 10min

Noncompliant Code:

```
fun foo(i: Int) {
    when (i) {
        1 -> throw IllegalArgumentException()
        2 -> throw IllegalArgumentException()
        3 -> throw IllegalArgumentException()
    }
}
```

Compliant Code:

```
fun foo(i: Int) {
    when (i) {
        1,2,3 -> throw IllegalArgumentException()
    }
}
```

TrailingWhitespace

This rule reports lines that end with a whitespace.

Active by default: No

Debt: 5min

TrimMultilineRawString

All the Raw strings that have more than one line should be followed by `trimMargin()` or `trimIndent()`.

Active by default: No

Debt: 5min

Noncompliant Code:

```
"""
Hello World!
How are you?
"""
```

Compliant Code:

```
"""
|  Hello World!
|  How are you?
""".trimMargin()

"""
Hello World!
How are you?
""".trimIndent()

"""Hello World! How are you?"""
```

UnderscoresInNumericLiterals

This rule detects and reports long base 10 numbers which should be separated with underscores
for readability. For `Serializable` classes or objects, the field `serialVersionUID` is explicitly ignored. For floats and doubles, anything to the right of the decimal point is ignored.

Active by default: No

Debt: 5min

Noncompliant Code:

```
const val DEFAULT_AMOUNT = 1000000
```

Compliant Code:

```
const val DEFAULT_AMOUNT = 1_000_000
```

UnnecessaryAbstractClass

This rule inspects `abstract` classes. In case an `abstract class` does not have any concrete members it should be
refactored into an interface. Abstract classes which do not define any `abstract` members should instead be
refactored into concrete classes.

Active by default: Yes - Since v1.2.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
abstract class OnlyAbstractMembersInAbstractClass { // violation: no concrete members

    abstract val i: Int
    abstract fun f()
}

abstract class OnlyConcreteMembersInAbstractClass { // violation: no abstract members

    val i: Int = 0
    fun f() { }
}
```

Compliant Code:

```
interface OnlyAbstractMembersInInterface {
    val i: Int
    fun f()
}

class OnlyConcreteMembersInClass {
    val i: Int = 0
    fun f() { }
}
```

UnnecessaryAnnotationUseSiteTarget

This rule inspects the use of the Annotation use-site Target. In case that the use-site Target is not needed it can
be removed. For more information check the kotlin documentation:
Annotation use-site targets

Active by default: No

Debt: 5min

Noncompliant Code:

```
@property:Inject private val foo: String = "bar" // violation: unnecessary @property:

class Module(@param:Inject private val foo: String) // violation: unnecessary @param:
```

Compliant Code:

```
class Module(@Inject private val foo: String)
```

UnnecessaryApply

`apply` expressions are used frequently, but sometimes their usage should be replaced with
an ordinary method/extension function call to reduce visual complexity

Active by default: Yes - Since v1.16.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
config.apply { version = "1.2" } // can be replaced with `config.version = "1.2"`
config?.apply { environment = "test" } // can be replaced with `config?.environment = "test"`
config?.apply { println(version) } // `apply` can be replaced by `let`
```

Compliant Code:

```
config.apply {
    version = "1.2"
    environment = "test"
}
```

UnnecessaryBackticks

This rule reports unnecessary backticks.

Active by default: No

Debt: 5min

Noncompliant Code:

```
class `HelloWorld`
```

Compliant Code:

```
class HelloWorld
```

UnnecessaryBracesAroundTrailingLambda

In Kotlin functions the last lambda parameter of a function is a function then a lambda expression passed as the
corresponding argument can be placed outside the parentheses.
see Passing trailing lambdas.
Prefer the usage of trailing lambda.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
fun test() {
    repeat(10, {
        println(it)
    })
}
```

Compliant Code:

```
fun test() {
    repeat(10) {
        println(it)
    }
}
```

UnnecessaryFilter

Unnecessary filters add complexity to the code and accomplish nothing. They should be removed.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
val x = listOf(1, 2, 3)
    .filter { it > 1 }
    .count()

val x = listOf(1, 2, 3)
    .filter { it > 1 }
    .isEmpty()
```

Compliant Code:

```
val x = listOf(1, 2, 3)
    .count { it > 2 }
}

val x = listOf(1, 2, 3)
    .none { it > 1 }
```

UnnecessaryInheritance

This rule reports unnecessary super types. Inheriting from `Any` or `Object` is unnecessary and should simply be
removed.

Active by default: Yes - Since v1.2.0

Debt: 5min

Noncompliant Code:

```
class A : Any()
class B : Object()
```

UnnecessaryInnerClass

This rule reports unnecessary inner classes. Nested classes that do not access members from the outer class do
not require the `inner` qualifier.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
class A {
    val foo = "BAR"

    inner class B {
        val fizz = "BUZZ"

        fun printFizz() {
            println(fizz)
        }
    }
}
```

UnnecessaryLet

`let` expressions are used extensively in our code for null-checking and chaining functions,
but sometimes their usage should be replaced with an ordinary method/extension function call
to reduce visual complexity.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
a.let { print(it) } // can be replaced with `print(a)`
a.let { it.plus(1) } // can be replaced with `a.plus(1)`
a?.let { it.plus(1) } // can be replaced with `a?.plus(1)`
a?.let { that -> that.plus(1) }?.let { it.plus(1) } // can be replaced with `a?.plus(1)?.plus(1)`
a.let { 1.plus(1) } // can be replaced with `1.plus(1)`
a?.let { 1.plus(1) } // can be replaced with `if (a != null) 1.plus(1)`
```

Compliant Code:

```
a?.let { print(it) }
a?.let { 1.plus(it) } ?.let { msg -> print(msg) }
a?.let { it.plus(it) }
val b = a?.let { 1.plus(1) }
```

UnnecessaryParentheses

This rule reports unnecessary parentheses around expressions.
These unnecessary parentheses can safely be removed.

Added in v1.0.0.RC4

Active by default: No

Debt: 5min

Noncompliant Code:

```
val local = (5 + 3)

if ((local == 8)) { }

fun foo() {
    function({ input -> println(input) })
}
```

Compliant Code:

```
val local = 5 + 3

if (local == 8) { }

fun foo() {
    function { input -> println(input) }
}
```

UntilInsteadOfRangeTo

Reports calls to '..' operator instead of calls to 'until'.
'until' is applicable in cases where the upper range value is described as some value subtracted by 1. 'until' helps to prevent off-by-one errors.

Active by default: No

Debt: 5min

Noncompliant Code:

```
for (i in 0 .. 10 - 1) {}
val range = 0 .. 10 - 1
```

Compliant Code:

```
for (i in 0 until 10) {}
val range = 0 until 10
```

UnusedImports

This rule reports unused imports. Unused imports are dead code and should be removed.
Exempt from this rule are imports resulting from references to elements within KDoc and
from destructuring declarations (componentN imports).

Active by default: No

Debt: 5min

UnusedParameter

An unused parameter can be removed to simplify the signature of the function.

Active by default: Yes - Since v1.23.0

Debt: 5min

Aliases: UNUSED_VARIABLE, UNUSED_PARAMETER, unused, UnusedPrivateMember

Noncompliant Code:

```
fun foo(unused: String) {
println()
}
```

Compliant Code:

```
fun foo(used: String) {
println(used)
}
```

UnusedPrivateClass

Reports unused private classes. If private classes are unused they should be removed. Otherwise, this dead code
can lead to confusion and potential bugs.

Active by default: Yes - Since v1.2.0

Debt: 5min

Aliases: unused

UnusedPrivateMember

Reports unused private functions.

If these private functions are unused they should be removed. Otherwise, this dead code
can lead to confusion and potential bugs.

Active by default: Yes - Since v1.16.0

Debt: 5min

Aliases: UNUSED_VARIABLE, UNUSED_PARAMETER, unused

UnusedPrivateProperty

An unused private property can be removed to simplify the source file.

This rule also detects unused constructor parameters since these can become properties of the class when they are declared with `val` or `var`.

Active by default: Yes - Since v1.23.0

Debt: 5min

Aliases: UNUSED_VARIABLE, UNUSED_PARAMETER, unused, UnusedPrivateMember

Noncompliant Code:

```
class Foo {
private val unused = "unused"
}
```

Compliant Code:

```
class Foo {
private val used = "used"

fun greet() {
    println(used)
}
}
```

UseAnyOrNoneInsteadOfFind

Turn on this rule to flag `find` calls for null check that can be replaced with a `any` or `none` call.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
listOf(1, 2, 3).find { it == 4 } != null
listOf(1, 2, 3).find { it == 4 } == null
```

Compliant Code:

```
listOf(1, 2, 3).any { it == 4 }
listOf(1, 2, 3).none { it == 4 }
```

UseArrayLiteralsInAnnotations

This rule detects annotations which use the arrayOf(...) syntax instead of the array literal [...] syntax.
The latter should be preferred as it is more readable.

Active by default: Yes - Since v1.21.0

Debt: 5min

Noncompliant Code:

```
@PositiveCase(arrayOf("..."))
```

Compliant Code:

```
@NegativeCase(["..."])
```

UseCheckNotNull

Turn on this rule to flag `check` calls for not-null check that can be replaced with a `checkNotNull` call.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
check(x != null)
```

Compliant Code:

```
checkNotNull(x)
```

UseCheckOrError

Kotlin provides a concise way to check invariants as well as pre- and post-conditions. Prefer them instead of manually throwing an IllegalStateException.

Active by default: Yes - Since v1.21.0

Debt: 5min

Noncompliant Code:

```
if (value == null) throw IllegalStateException("value should not be null")
if (value < 0) throw IllegalStateException("value is $value but should be at least 0")
when(a) {
    1 -> doSomething()
    else -> throw IllegalStateException("Unexpected value")
}
```

Compliant Code:

```
checkNotNull(value) { "value should not be null" }
check(value >= 0) { "value is $value but should be at least 0" }
when(a) {
    1 -> doSomething()
    else -> error("Unexpected value")
}
```

UseDataClass

Classes that simply hold data should be refactored into a `data class`. Data classes are specialized to hold data
and generate `hashCode`, `equals` and `toString` implementations as well.

Read more about data classes

Active by default: No

Debt: 5min

Noncompliant Code:

```
class DataClassCandidate(val i: Int) {
    val i2: Int = 0
}
```

Compliant Code:

```
data class DataClass(val i: Int, val i2: Int)

// classes with delegating interfaces are compliant
interface I
class B() : I
class A(val b: B) : I by b
```

UseEmptyCounterpart

Instantiation of an object's "empty" state should use the object's "empty" initializer for clarity purposes.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
arrayOf()
listOf() // or listOfNotNull()
mapOf()
sequenceOf()
setOf()
```

Compliant Code:

```
emptyArray()
emptyList()
emptyMap()
emptySequence()
emptySet()
```

UseIfEmptyOrIfBlank

This rule detects `isEmpty` or `isBlank` calls to assign a default value. They can be replaced with `ifEmpty` or `ifBlank` calls.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
fun test(list: List<Int>, s: String) {
    val a = if (list.isEmpty()) listOf(1) else list
    val b = if (list.isNotEmpty()) list else listOf(2)
    val c = if (s.isBlank()) "foo" else s
    val d = if (s.isNotBlank()) s else "bar"
}
```

Compliant Code:

```
fun test(list: List<Int>, s: String) {
    val a = list.ifEmpty { listOf(1) }
    val b = list.ifEmpty { listOf(2) }
    val c = s.ifBlank { "foo" }
    val d = s.ifBlank { "bar" }
}
```

UseIfInsteadOfWhen

Binary expressions are better expressed using an `if` expression than a `when` expression.

See if versus when

Active by default: No

Debt: 5min

Noncompliant Code:

```
when (x) {
    null -> true
    else -> false
}
```

Compliant Code:

```
if (x == null) true else false
```

UseIsNullOrEmpty

This rule detects null or empty checks that can be replaced with `isNullOrEmpty()` call.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
fun foo(x: List<Int>?) {
    if (x == null || x.isEmpty()) return
}
fun bar(x: List<Int>?) {
    if (x == null || x.count() == 0) return
}
fun baz(x: List<Int>?) {
    if (x == null || x.size == 0) return
}
```

Compliant Code:

```
if (x.isNullOrEmpty()) return
```

UseLet

`if` expressions that either check for not-null and return `null` in the false case or check for `null` and returns
`null` in the truthy case are better represented as `?.let {}` blocks.

Active by default: No

Debt: 5min

Noncompliant Code:

```
if (x != null) { x.transform() } else null
if (x == null) null else y
```

Compliant Code:

```
x?.let { it.transform() }
x?.let { y }
```

UseOrEmpty

This rule detects `?: emptyList()` that can be replaced with `orEmpty()` call.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
fun test(x: List<Int>?, s: String?) {
    val a = x ?: emptyList()
    val b = s ?: ""
}
```

Compliant Code:

```
fun test(x: List<Int>?, s: String?) {
    val a = x.orEmpty()
    val b = s.orEmpty()
}
```

UseRequire

Kotlin provides a much more concise way to check preconditions than to manually throw an IllegalArgumentException.

Active by default: Yes - Since v1.21.0

Debt: 5min

Noncompliant Code:

```
if (value == null) throw IllegalArgumentException("value should not be null")
if (value < 0) throw IllegalArgumentException("value is $value but should be at least 0")
```

Compliant Code:

```
requireNotNull(value) { "value should not be null" }
require(value >= 0) { "value is $value but should be at least 0" }
```

UseRequireNotNull

Turn on this rule to flag `require` calls for not-null check that can be replaced with a `requireNotNull` call.

Active by default: Yes - Since v1.21.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
require(x != null)
```

Compliant Code:

```
requireNotNull(x)
```

UseSumOfInsteadOfFlatMapSize

Turn on this rule to flag `flatMap` and `size/count` calls that can be replaced with a `sumOf` call.

Active by default: No

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
class Foo(val foo: List<Int>)
list.flatMap { it.foo }.size
list.flatMap { it.foo }.count()
list.flatMap { it.foo }.count { it > 2 }
listOf(listOf(1), listOf(2, 3)).flatten().size
```

Compliant Code:

```
list.sumOf { it.foo.size }
list.sumOf { it.foo.count() }
list.sumOf { it.foo.count { foo -> foo > 2 } }
listOf(listOf(1), listOf(2, 3)).sumOf { it.size }
```

UselessCallOnNotNull

The Kotlin stdlib provides some functions that are designed to operate on references that may be null. These
functions can also be called on non-nullable references or on collections or sequences that are known to be empty -
the calls are redundant in this case and can be removed or should be changed to a call that does not check whether
the value is null or not.

Active by default: Yes - Since v1.2.0

Requires Type Resolution

Debt: 5min

Noncompliant Code:

```
val testList = listOf("string").orEmpty()
val testList2 = listOf("string").orEmpty().map { _ }
val testList3 = listOfNotNull("string")
val testString = ""?.isNullOrBlank()
```

Compliant Code:

```
val testList = listOf("string")
val testList2 = listOf("string").map { }
val testList3 = listOf("string")
val testString = ""?.isBlank()
```

UtilityClassWithPublicConstructor

A class which only contains utility variables and functions with no concrete implementation can be refactored
into an `object` or a class with a non-public constructor.
Furthermore, this rule reports utility classes which are not final.

Active by default: Yes - Since v1.2.0

Debt: 5min

Noncompliant Code:

```
class UtilityClassViolation {

    // public constructor here
    constructor() {
        // ...
    }

    companion object {
        val i = 0
    }
}

open class UtilityClassViolation private constructor() {

    // ...
}
```

Compliant Code:

```
class UtilityClass {

    private constructor() {
        // ...
    }

    companion object {
        val i = 0
    }
}
object UtilityClass {

    val i = 0
}
```

VarCouldBeVal

Reports var declarations (both local variables and private class properties) that could be val,
as they are not re-assigned. Val declarations are assign-once (read-only), which makes understanding
the current state easier.

Active by default: Yes - Since v1.16.0

Requires Type Resolution

Debt: 5min

Aliases: CanBeVal

Noncompliant Code:

```
fun example() {
    var i = 1 // violation: this variable is never re-assigned
    val j = i + 1
}
```

Compliant Code:

```
fun example() {
    val i = 1
    val j = i + 1
}
```

WildcardImport

Wildcard imports should be replaced with imports using fully qualified class names.
This helps increase clarity of
which classes are imported and helps prevent naming conflicts.

Library updates can introduce naming clashes with your own classes which might result in compilation errors.

NOTE: This rule has a twin implementation NoWildcardImports in the formatting rule set (a wrapped KtLint rule).
When suppressing an issue of WildcardImport in the baseline file, make sure to suppress the corresponding NoWildcardImports issue.

Active by default: Yes - Since v1.0.0

Debt: 5min

Noncompliant Code:

```
import io.gitlab.arturbosch.detekt.*

class DetektElements {
    val element1 = DetektElement1()
    val element2 = DetektElement2()
}
```

Compliant Code:

```
import io.gitlab.arturbosch.detekt.DetektElement1
import io.gitlab.arturbosch.detekt.DetektElement2

class DetektElements {
    val element1 = DetektElement1()
    val element2 = DetektElement2()
}
```