Ignite Android Coding Guidelines

Ignite Android Code Guidelines 1.pdf

Introduction

Purpose

This document defines the Swift programming standards. The document makes no proposals for the design of Android programs, but concentrates entirely on the mechanics of Kotlin programming.

Note several important points:

- Projects for particular customers or technologies may require the use of their own coding standards – use them in these cases.
- Updates to existing code should continue to use the coding standards of the original (provided there were some) – DO NOT change working code simply to change the layout, because you WILL introduce bugs.

Scope

These standards are to be applied to all software written in Kotlin.

HARMAN's Coding Standards

This document defines the coding standards for source code in the Kotlin Programming Language for HARMAN delivered Open Source Projects.

Coding Guidelines

Source File Structure

A source file consists of the following, in this exact order:

1	License or Copyright information	• Each Kotlin file must specify the following license at the very top of the file	See Appendix A

2	File description	• Every file should have a comment at the top describing its contents. This description section must appear right after the license information. File description will contain module information, module summary, traceability records and modification records	See Appendix B.
3	File lavel Annotation	• Annotations are means of attaching metadata to code. To declare an annotation, put the `annotation` modifier in front of a class:	`@file:CustomName package org.example` `@Target (AnnotationTarget.FILE) annotation class CustomName`
4	Package Statements	The package statement is not subject to any column limit and is never line-wrapped	package org.example
5	Import Statements	Import statements for classes, functions, and properties are grouped together in a single list and ASCII sorted. Wildcard imports (of any type) are **not allowed.** Similar to the package statement, import statements are not subject to a column limit and they are never line-wrapped.	import org.example.Message
6	Top Lavel decleration	A `.kt` file can declare one or more types, functions, properties, or type aliases at the top-level. The contents of a file should be focused on a single theme. Examples of this would be a single public type or a set of extension functions performing the same operation on multiple receiver types. Unrelated declarations should be separated into their own files and public declarations within a single file should be minimized. No explicit restriction is placed on the number nor order of the contents of a file. Source files are usually read from top-to-bottom meaning that the order, in general, should reflect that the declarations higher up will inform understanding of those farther down. Different files may choose to order their contents differently. Similarly, one file may contain 100 properties, another 10 functions, and yet another a single class. What is important is that each file uses **some** logical order, which its maintainer could explain if asked. For example, new functions are not just habitually added to the end of the file, as that would yield “chronological by date added” ordering, which is not a logical ordering.	
	Class member ordering	The order of members within a class follow the same rules as the top-level declarations.	

Formatting

#	Code Format	Criteria Description	Remarks
1	Braces	Braces are not required for `when` branches and `if` expressions which have no more than one `else` branch and which fit on a single line	`if (string.isEmpty()) return` `val result = if (string.isEmpty()) DEFAULT_VALUE else string` `when (value) { 0 -> return // … }`

2	Non Empty Blocks	Braces follow the Kernighan and Ritchie style ("Egyptian brackets") for nonempty blocks and block-like constructs: • No line break before the opening brace. • Line break after the opening brace. • Line break before the closing brace. • Line break after the closing brace, *only if* that brace terminates a statement or terminates the body of a function, constructor, or *named* class. For example, there is *no* line break after the brace if it is followed by `else` or a comma. : 	``` return object : MyClass() { override fun foo() { if (condition()) { try { something() } catch (e: ProblemException) { recover() } } else if (otherCondition()) { somethingElse() } else { lastThing() } } } ```
3	Empty Blocks	An empty block or block-like construct must be in K&R style	``` try { doSomething() } catch (e: Exception) { } // Okay ```
4	Expressions	An `if/else` conditional that is used as an expression may omit braces *only* if the entire expression fits on one line	``` val value = if (string.isEmpty()) 0 else 1 ```
5	indentation	Each time a new block or block-like construct is opened, the indent increases by four spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block. Each statement is followed by a line break. Semicolons are not used.	
6	Where to beak	The prime directive of line-wrapping is: prefer to break at a higher syntactic level. Also: • When a line is broken at an operator or infix function name, the break comes after the operator or infix function name. • When a line is broken at the following “operator-like” symbols, the break comes before the symbol: ◦ The dot separator (`.`, `?.`). ◦ The two colons of a member reference (`::`). • A method or constructor name stays attached to the open parenthesis (`(`) that follows it. • A comma (`,`) stays attached to the token that precedes it. • A lambda arrow (`->`) stays attached to the argument list that precedes it.	
7	Functions	When a function signature does not fit on a single line, break each parameter declaration onto its own line. Parameters defined in this format should use a single indent (+4). The closing parenthesis (`)`) and return type are placed on their own line with no additional indent. When a function contains only a single expression it can be represented as an expression functions.	``` fun <T> Iterable<T>.joinToString(separator: CharSequence = ", ", prefix: CharSequence = "", postfix: CharSequence = ""): String { // … } override fun toString(): String { return "Hey" } ```

8	Properties	When a property initializer does not fit on a single line, break after the equals sign (=) and use an indent. Properties declaring a `get` and/or `set` function should place each on their own line with a normal indent (+4). Format them using the same rules as functions. Read-only properties can use a shorter syntax which fits on a single line.	`private val defaultCharset: Charset? =  EncodingRegistry.getInstance().getDefaultCharsetForPropertiesFiles(file)` `var directory: File? = null  set(value) {  // …  }` `val defaultExtension: String get() = "kt"`
9	Vertical Whitespaces	A single blank line appears: • *Between* consecutive members of a class: properties, constructors, functions, nested classes, etc. ◦ **Exception:** A blank line between two consecutive properties (having no other code between them) is optional. Such blank lines are used as needed to create logical groupings of properties and associate properties with their backing property, if present. ◦ **Exception:** Blank lines between enum constants are covered below. • Between statements, *as needed* to organize the code into logical subsections. • *Optionally* before the first statement in a function, before the first member of a class, or after the last member of a class (neither encouraged nor discouraged). • As required by other sections of this document (such as the Structure section). Multiple consecutive blank lines are permitted, but not encouraged or ever required.	
10	Horizontal whitespaces	Beyond where required by the language or other style rules, and apart from literals, comments, and KDoc, a single ASCII space also appears in the following places only: • Separating any reserved word, such as `if`, `for`, or `catch` from an open parenthesis (`(`) that follows it on that line.	
11	Specific Construct Enum classes	An enum with no functions and no documentation on its constants may optionally be formatted as a single line. When the constants in an enum are placed on separate lines, a blank line is not required between them except in the case where they define a body.	`enum class Answer { YES, NO, MAYBE }`
12	Annotations	Member or type annotations are placed on separate lines immediately prior to the annotated construct. Annotations without arguments can be placed on a single line When only a single annotation without arguments is present, it may be placed on the same line as the declaration.	`@Retention(SOURCE) @Target(FUNCTION, PROPERTY_SETTER, FIELD) annotation class Global` `@JvmField @Volatile var disposable: Disposable? = null` `@Volatile var disposable: Disposable? = null` `@Test fun selectAll() {  // …  }`

Naming

Names provide a language for building a common understanding in the code. Names should allow software developers to look at code and determine what is the meaning of the name and how to use it. A name should be descriptive enough to allow any developer knowledgeable in the domain of the software to relate it to a concrete or abstract principle in the domain

#	Code Format	Criteria Description	Remarks

1	Naming	Identifiers use only ASCII letters and digits, and, in a small number of cases noted below, underscores. Thus each valid identifier name is matched by the regular expression `\w+`. Special prefixes or suffixes, like those seen in the examples `name_`, `mName`, `s_name`, and `kName`, are not used except in the case of backing properties	
2	Package Name	Package names are all lowercase, with consecutive words simply concatenated together (no underscores).	`package com.example.deepspace`
3	Type Name	Class names are written in PascalCase and are typically nouns or noun phrases. For example, `Character` or `ImmutableList`. Interface names may also be nouns or noun phrases (for example, `List`), but may sometimes be adjectives or adjective phrases instead (for example `Readable`). Test classes are named starting with the name of the class they are testing, and ending with `Test`. For example, `HashTest` or `HashIntegrationTest`	
4	Function Name	Function names are written in camelCase and are typically verbs or verb phrases. For example, `sendMessage` or `stop`. Underscores are permitted to appear in test function names to separate logical components of the name. Functions annotated with `@Composable` that return `Unit` are PascalCased and named as nouns, as if they were types. Function names should not contain spaces because this is not supported on every platform (notably, this is not fully supported in Android).	`@Test fun pop_emptyStack() {` `  // …` `}` `@Composable` `fun NameTag(name: String) {` `  // …` `}`
5	Constant name	Constant names use UPPER_SNAKE_CASE: all uppercase letters, with words separated by underscores. But what *is* a constant, exactly? Constants are `val` properties with no custom `get` function, whose contents are deeply immutable, and whose functions have no detectable side-effects. This includes immutable types and immutable collections of immutable types as well as scalars and string if marked as `const`. If any of an instance's observable state can change, it is not a constant. Merely intending to never mutate the object is not enough.	`const val NUMBER = 5` `val NAMES = listOf("Alice", "Bob")` `val AGES = mapOf("Alice" to 35, "Bob" to 32)` `val COMMA_JOINER = Joiner.on(',') // Joiner is immutable` `val EMPTY_ARRAY = arrayOf()`

6	Non Constant Names	Non-constant names are written in camelCase. These apply to instance properties, local properties, and parameter names	`val variable = "var"` `val nonConstScalar = "non-const"` `val mutableCollection: MutableSet = HashSet()` `val mutableElements = listOf(mutableInstance)` `val mutableValues = mapOf("Alice" to mutableInstance, "Bob" to mutableInstance2)` `val logger = Logger.getLogger(MyClass::class.java.name)` `val nonEmptyArray = arrayOf("these", "can", "change")`
7	Backing Properties	When a backing property is needed, its name should exactly match that of the real property except prefixed with an underscore.	`private var _table: Map? = null` `val table: Map` `get() {` `if (_table == null) {` `_table = HashMap()` `}` `return _table ?: throw AssertionError()` `}`
8	Type variable name	Each type variable is named in one of two styles: • A single capital letter, optionally followed by a single numeral (such as `E`, `T`, `X`, `T2`) • A name in the form used for classes, followed by the capital letter `T` (such as `RequestT`, `FooBarT`)	

9	Camel Case	Sometimes there is more than one reasonable way to convert an English phrase into camel case, such as when acronyms or unusual constructs like “IPv6” or “iOS” are present. To improve predictability, use the following scheme. Beginning with the prose form of the name: 1. Convert the phrase to plain ASCII and remove any apostrophes. For example, “Müller’s algorithm” might become “Muellers algorithm”. 2. Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens).*Recommended:* if any word already has a conventional camel-case appearance in common usage, split this into its constituent parts (e.g., “AdWords” becomes “ad words”). Note that a word such as “iOS” is not really in camel case per se; it defies any convention, so this recommendation does not apply. 3. Now lowercase everything (including acronyms), then do one of the following: • Uppercase the first character of each word to yield pascal case. • Uppercase the first character of each word except the first to yield camel case. 4. Finally, join all the words into a single identifier. Note that the casing of the original words is almost entirely disregarded.	
10	Image Name	Use underscore for adding icons e.g Trip Start Icon: icn_trip_start.png or icn_trip_start.xml (SVG)	
11	Localisation Key Name	Use underscore for keys name e.g Trip Start Title :- “trip_start_title” = “Trip Start” Add a New Boundary title:- “add_a_new_boundary” = “Add a New Boundary”	

Documentation - Doc

#	Criteria	Criteria Description	Remarks
#	Criteria	Criteria Description	Remarks
1	General	The language used to document Kotlin code (the equivalent of Java's Javadoc) is called **KDoc**. In essence, KDoc combines Javadoc's syntax for block tags (extended to support Kotlin's specific constructs) and Markdown for inline markup.	https://kotlinlang.org/docs/kotlin-doc.html#kdoc-syntax

Best Practices

SOLID Principles

	Acronym	Description
1	S	The Single Responsibility Principle states that **a class should do one thing and therefore it should have only a single reason to change**.
2	O	The Open-Closed Principle requires that **classes should be open for extension and closed to modification.**
3	L	The Liskov Substitution Principle states that subclasses should be substitutable for their base classes.
4	I	Segregation means keeping things separated, and the Interface Segregation Principle is about separating the interfaces.
5	D	The Dependency Inversion principle states that the classes should depend upon interfaces or abstract classes instead of concrete classes and functions.

Appendix**:**

A. License Header

```
/**
*******************************************************************************
* COPYRIGHT (c) <<yyyy>> Harman International Industries, Inc                 *
*                                                                             *
* All rights reserved                                                         *
*                                                                             *
* This software embodies materials and concepts which are                     *
* confidential to Harman International Industries, Inc. and is                *
* made available solely pursuant to the terms of a written license            *
* agreement with Harman International Industries, Inc.                        *
*                                                                             *
* Designed and Developed by Harman International Industries, Inc.             *
*-----------------------------------------------------------------------------*
* MODULE OR UNIT: <<name of the component or module>>                         *
*******************************************************************************
*/
```

B. File description

Doc coding documentation formatting

Just like with Javadoc, KDoc comments start with `/**` and end with `*/`. Every line of the comment may begin with an asterisk, which is not considered part of the contents of the comment.

By convention, the first paragraph of the documentation text (the block of text until the first blank line) is the summary description of the element, and the following text is the detailed description.

Every block tag begins on a new line and starts with the `@` character.

Here's an example of a class documented using KDoc:

```
/** * A group of *members*. * * This class has no useful logic; it's just a documentation example. * * @param T the type of a member in this group. * @property name the name of this group. * @constructor Creates an empty group. */ class Group<T>(val name: String) { /** * Adds a [member] to this group. * @return the new size of the group. */ fun add(member: T): Int { ... } }
```

Block tags

KDoc currently supports the following block tags:

@param name

Documents a value parameter of a function or a type parameter of a class, property or function. To better separate the parameter name from the description, if you prefer, you can enclose the name of the parameter in brackets. The following two syntaxes are therefore equivalent:

```
@param name description. @param[name] description.
```

@return

Documents the return value of a function.

@constructor

Documents the primary constructor of a class.

@receiver

Documents the receiver of an extension function.

@property name

Documents the property of a class which has the specified name. This tag can be used for documenting properties declared in the primary constructor, where putting a doc comment directly before the property definition would be awkward.

@throws class, @exception class

Documents an exception which can be thrown by a method. Since Kotlin does not have checked exceptions, there is also no expectation that all possible exceptions are documented, but you can still use this tag when it provides useful information for users of the class.

@sample identifier

Embeds the body of the function with the specified qualified name into the documentation for the current element, in order to show an example of how the element could be used.

@see identifier

Adds a link to the specified class or method to the **See also** block of the documentation.

@author

Specifies the author of the element being documented.

@since

Specifies the version of the software in which the element being documented was introduced.

@suppress

Excludes the element from the generated documentation. Can be used for elements which are not part of the official API of a module but still have to be visible externally.