Kotlin Coding Convention

Structure

Resource structure

Name	Path	Description
XML Layouts	`res/layout/`	This is where we put our XML layout files.
XML Menus	`res/menu/`	This is where we put our AppBar menu actions.
Anim	`res/anim/`	This is where we put animation (API < 11).
Animator	`res/animator/`	This is where we put animator (API > 11).
Drawables	`res/drawable`	This is where we put XML drawables.
Resource	`res/drawable-xhdpi|xxhdpi|xxxhdpi`	This is where we put images.
Asset	`res/assets`	This is where we put databases, raw datas, etc
Colors	`res/values/colors.xml`	This is where we put color definitions.
Dimensions	`res/values/dimens.xml`	This is where we put dimension values.
String	`res/values/strings.xml`	This is where we put strings.
Styles	`res/values/styles.xml`	This is where we put style values.

File structure

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

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

- **Copyright and/or license header (optional)**
  If a **copyright or license header** belongs in the file it should be placed at the immediate top in a **multi-line comment**.
- *GOOD**

```
/*
* Copyright 2017 Google, Inc.
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at:
```

```
* http://www.apache.org/licenses/LICENSE-2.0
* ...
*/
```

- *BAD**

```
/**
* Copyright 2017 Google, Inc.
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at:
* http://www.apache.org/licenses/LICENSE-2.0
* ...
*/
```

- *BAD**

```
// Copyright 2017 Google, Inc.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at:
// http://www.apache.org/licenses/LICENSE-2.0
// ...
```

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

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

Class

- First letter of classes is UpperCase.
- Name of class only accept in range **[A-Z], [a-z]** and follow **Camel Case**.
- Classes name must be **noun**.
- For classes that extend an Android component, the name of the class should end with the name of the component; for example: SignInActivity, SignInFragment, ImageUploaderService, ChangePasswordDialog.
- Open brace ‘ { ‘ appears at the end of the same line as the declaration statement.
- Closing brace ‘ } ‘ starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the ‘ } ‘ should appear immediately after the ‘ { ‘
- Write KotlinDoc for each class, Kotlin doc must define what are classes working for.

See Example

```
/**
 * Copyright © Monstarlab Vietnam Co., Ltd.
 * Created by mvn-...-dn on 16/04/2021.
 * ...
 */
class SomeClass {
...
}
```

- Class member ordering There is no single correct solution for this but using a **logical** and **consistent** order will improve code learnability and readability. It is recommendable to use the following order:
  - `Constants`
  - `Fields`
  - `Constructors`
  - `Override methods and callbacks`
  - `Public methods`
  - `Internal methods`
  - `Private methods`
  - `Inner classes and interfaces`

`(TODO: inner class can be change position or some positions others)`

Example

```
 class MainActivity : Activity() {
   var title: String
   var tvTitle: TextView

   override fun onCreate() {
        ...
   }

   internal fun setName(): String {
        ...
   }

   private fun setUpView() {
        ...
   }
}
```

If your class is extending an **Android components** such as an Activity or a Fragment, it is a good practice to order the override methods so that they **match the component's lifecycle**. For example, if you have an Activity that implements **onCreate()**, **onDestroy()**, **onPause()** and **onResume()**, correct order as below:

```
 class MainActivity : Activity() {

  override fun onCreate() {}
```

```
  override fun onResume() {}

  override fun onPause() {}

  override fun onDestroy() {}
}
```

Method

- Short method content: Method content should be short and focus on the feature of method. Avoid repeating code.
- Method names not over `30 characters`. Except for inputting text or URL.
- Code lines not over `30 lines`.
- Method name must start with verb is first letter.
- First letter of method name is **LOWERCASE**.
- `@SuppressWarnings`: The `@SuppressWarnings` annotation should only be used under circumstances where it is impossible to eliminate a warning. If a warning passes this "**impossible to eliminate**" test, the `@SuppressWarnings` annotation must be used, so as to ensure that all warnings reflect actual problems in the code.

Example

- Write document for methods if it’s necessary.

```
/**
* Calculate sum of two integers
*
* @param a number
* @param b number
* @return the sum of a and b
*/
fun calculateSum(a:Int, b:Int): Int {
    ...
}
```

- Limit method block line less than **30** lines, limit method arguments less than **5**. Separate code if function has long line method.
- If statement convention.

GOOD

```
 if (...) {
     ...
 }
```

BAD

```
if (...)
    ...
```

- Use `// TODO` for dummy data or need to change later.

```
// TODO Remove after api finish
```

- Catch exception.

GOOD

```
try {
    ...
} catch (et1: ExceptionType1) {
    ...
} catch (et2: ExceptionType2) {
    ...
}

finally {
   ...
}
```

BAD

```
try {
    ...
} catch (e: Exception) {
    ...
}

finally {
   ...
}
```

XML Conventions

- `View ID` is followed by camel-case

Example

```
btnLogin, tvCaption
```

- Prefix for `id` in XML

Name	Prefix	Name	Prefix
Button	btn	Datepicker	datePicker
EditText	edt	Timepicker	timePicker
TextView	tv	Videoview	videoView
Checkbox	chk	Seekbar	seekBar
RadioButton	rb	RatingBar	ratingBar
ToggleButton	tb	Processbar	processBar
Spinner	spn	ImageView	img
Menu	menu	ImageButton	imgBtn
ListView	lv	RecyclerView	recycleView

GalleryView	gv	CardView	cardView
LinearLayout	ll	ScrollView	scrollView
RelativeLayout	rl	ToolBar	toolbar
Calendar	calendar	FrameLayout	fr

- XML File Name
    - Item for ListView, ReyclerView, GridView: `item_*`.
    - Custom for view: `custom_*`.
    - Custom for dialog: `dialog_*`.
    - Item of shape, selector, etc:
        - If item is a common using with least 2 item view : `common_bg_*`.
        - Otherwise: `bg_ prefix _*`.
    - Resource Name:
        - Icon:
            - Normal: `ic_*`.
            - State: `ic_*_hover , ic_*_normal , ic_*_select`.
        - Image:
            - Name: `bg_*`.
            - Folder: `drawable-xhdpi, drawable-xxhdpi, drawable-xxxhdpi`.

Example:

```
@+id/tvSubjectQuestions
@+id/imgTakeCamera
```

- String resource name:
    - Name: `ScreenName_Description`.
    - Ex: `home_show_dialog_ok`.
- Style resource:
    - Name: Using `Camel` but upper first character.
    - Ex: `StyleForButton` , `StyleForListContact`.
    - Dimension can hard code in `style`.
- Color resource:
    - Hex color must using Lowercase letters.
    - When define alpha (transparent), must to show percent of alpha.
    - Name of color must folow `Camel` convention.

```
<color name="red">#f44336</color>
<color name="redLight">#30f44336</color>
```

- Other naming
    - Model (example: Student, Teacher)

    - Activity: `*Activity` (example: UserActivity)
    - Fragment: `*Fragment` (example: HomeFragment)
    - Adapter: `*Adapter` (example: PageAdapter)
- `Padding` , `Margin` : accept to use hard code in XML file.
- Text size must use `dp` instead `sp` .
- **Do not** make a deep hierarchy of `ViewGroups` . here
- Also keep `dimens.xml` DRY, define generic constants. here
- Use multiple style files to avoid a single huge one. here
- When an `XML` element doesn't have any contents, you **must** use self closing tags.

GOOD

```
<TextView
    android: id="@+id/tvProfile"
    android: layout_width="wrap_content"
    android: layout_height="wrap_content" />
```

BAD

```
<TextView
    android: id="@+id/text_view_profile"
    android: layout_width="wrap_content"
    android: layout_height="wrap_content" >
</TextView>
```

Basic

Variables

- Write in **lowerCamelCase**.
- Single character value must be avoided, except for temporary looping variables.

GOOD

```
var studentName: String
```

BAD

```
var StudentName: String
```

- Use `non-null` value as much as possible
- Property priority
    1. `non-null & val`
    2. `non-null & var`
    3. `nullable & var`
- Use a **lateinit** or **Delegates.notNull()** if you cannot set a initial value. **lateinit** is better than **Delegates.notNull()** because **Delegates.notNull()** uses reflection. But primitive values is not applied to **lateinit**.

```
// Non-null & val
private val hoge: Hoge = Hoge()
private val drawablePadding: Int by lazy {
activity.resources.getDimensionPixelSize(R.dimen.drawable_padding) }

// Non-null & var
private lateinit var hoge: Hoge
private var hoge: Hoge = Delegates.notNull()

// Nullable & var
private var hoge: Hoge? = null
```

Type Inference

- Type inference should be preferred where possible to explicitly decleard types.

GOOD

```
val student = Student()
val age = 21
```

BAD

```
val student: Student = Student()
val age: Int = 21
```

Underscores in numeric literals

You can use underscores to make number constants more readable:

- With `Decimals`: Should use the underscore to group the 3 elements together.

```
val oneMillion = 1_000_000
```

- With `Hexadecimals`: Should use the underscore to group the 2 elements together after `0x`

```
val hexBytes = 0xFF_EC_DE_5E
```

- With `Binaries`: Should use the underscore to group the 8 elements together after `0b`.

```
val bytes = 0b11010010_01101001_10010100_10010010
```

- **Note**: With special case, self-defined programer. Example:

```
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
```

Strings

- Always prefer string interpolation if possible.

GOOD

```
val name = "${user.firstName} ${user.lastName}"
```

BAD

```
val name = user.firstName + " " + user.lastName
```

!!

- Do not use **!!**, it will erase the benefits of Kotlin.
- You use it only when you want to explicitly raise a Null Pointer Exception.

If-else expression

- Do not start a new line in variable declaration using if-else.

GOOD

```
val foo = 5
val bar = if(foo > 10) "kotlin" else "java"
```

BAD

```
val foo = 5
val bar = if (foo > 10) {
    "kotlin"
} else {
    "java"
}
```

When Expression

- Use **when** in case there are two or more branches of **if-else**.

GOOD

```
val hoge = 10
when {
    hoge > 10 -> print("10")
    hoge > 5 -> print("5")
    else -> print("different")
}
```

BAD

```
val hoge = 10
if (hoge > 10) {
 print("10")
} else if (hoge > 5) {
print("5")
} else {
print("different")
}
```

- Unlike `switch` statements in Java, `when` can be used either as an expression or as a statement. Separate cases using commas if they should be handled the same way. Always include the else case.

GOOD

```
when (Input) {
    1, 2 -> doSomethingForCaseOneOrTwo()
    3 -> doSomethingForCaseThree()
    else -> print("No case satisfied")
}
```

BAD

```
when (Input) {
    1 -> doSomeThingForCaseOne()
    2 -> doSomeThingForCaseOneOrTwo()
    3 -> doSomeThingForCaseThree()
}
```

Smart Cast

GOOD

```
dog as? Animal ?: throw IllegalArgumentException("not Animal!")
dog.foo()
```

BAD

```
if (dog !is Animal) {
    throw IllegalArgumentException("not Animal!")
}
dog.foo()
```

```
dog as Animal
dog.foo()
```

Collections

- Should be used `MutableList` if that `List` change data in future.

```
val students: MutableList<Int> = ArrayList()
```

- If you create the list to read only, you can use `List` instead of using `MutableList`.

```
val students: List<Int> = ArrayList()
```

- When get an item in a list.

GOOD

```
val array = ArrayList<Int>()
array[0]
```

BAD

```
val array = ArrayList<Int>()
array.get(0)
```

Use an IDE suggestion

- When call the activity

GOOD

```
activity.finish()
```

BAD

```
getActivity().finish()
```

Equality

- Should be used `equals` instead of `==` operator.

GOOD

```
a.equals(b)
!a.equals(b)
```

BAD

```
a == b
a != b
```

This Expression

- Don't use `this@label` if compiler understood that `this`.

GOOD

```
data class Test(var name: String) {
    fun showName(name: String) {
        this.name = name
    }
}
```

BAD

```
data class Test(var name: String) {
    fun showName(name: String) {
        this@Test.name = name
    }
}
```

Break line

Start a new line at `right vertical line` on Android studio. (About **100 characters**). Any line that would exceed this limit must be line-wrapped

- When a line is broken at an `assignment operator( =, +=, -=, *=, /=, %=)` the break comes after the symbol.

```
fun compare(a: String, b: String): Boolean =
             ...
```

- The break comes before the symbol `.` and `::` and `non-assignment operator`

```
addMarker(MarkerOptions()
                    .position(currentLocation)
                    .draggable(true)
                    .title(resources.getString(R.string.current_location))
```

```
fun getSomething(
        a: String,
        b: String,
        c: String,
        d: String,
        ::isSomething
){
   ...
}
```

- A `method` or `constructor` name stays attached to the open parenthesis `(` `(` that follows it and a `comma (,)` stays attached to the token that precedes it.

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

- A lambda arrow `->` stays attached to the argument list what precedes it.

```
val printSummary = { username: String, age: String, score: Int ->
                        println("User '$username' with '$age' get $score points.")
                    }
```

Annotation

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

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

- Annotations without arguments can be placed on a single line.

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

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

```
@Test fun selectAll() {
     ...
}
```

Loop (optional)

- You do not have to write for loop because there is `forEach` in collections package of Kotlin.

GOOD

```
(0..9).forEach { ... }

// If you want to know index
(0..9).forEachIndexed { index, value -> ... }
```

BAD

```
for (i in 0..9) { ... }
```

Use range

GOOD

```
val char = 'K'
if (char in 'A'..'Z') print("Hit!")
```

BAD

```
val char = 'K'
if (char >= 'A' && 'c' <= 'Z') print("Hit!")
```

OOP

Visibility Modifiers

- **Kotlin** have 4 **visibility modifiers: public, internal, protected, private.** But when we using we must define **visibility modifier** that have the smallest range of use.
- If **visibility modifiers** is **public** we can ignore **public** keyword:

GOOD

```
val name = "This is name"
```

BAD

```
public val name = "This is name"
```

Coroutine Scope

- GlobalScope.launch and GlobalScope.async are highly discouraged by the Kotlin documentation.
- Global scope is used to launch top-level coroutines which are operating on the whole application lifetime and are not cancelled prematurely.
- Application code usually should use an application-defined CoroutineScope. Using async or launch on the instance of GlobalScope is highly discouraged.

GOOD

```
val scope = CoroutineScope(Dispatchers.IO)
fun foo() {
    scope.launch {
        delay(1_000L)
    }
}

fun onDestroy() {
    scope.cancel()
}
```

BAD

```
fun foo() {
    GlobalScope.launch {
        delay(1_000L)
    }
}
```

Class

- **Constructor**: Initialize class properties as **primary constructor** parameters instead of in an **init** block.

GOOD

```
class Person (val firstName: String, val lastName) {
    ...
}
```

BAD

```
class Person (firstName: String, lastName: String) {
      val firstName: String
      val lastName: String

      init {
        this.firstName = firstName
        this.lastName = lastName
      }
}
```

Function

- 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.

GOOD

```
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.
}
```

BAD

```
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.
}
```

- If a function returns **Unit**, the return type should be ignored.

GOOD

```
fun showText (text: String) {
    println(text)
}
```

BAD

```
fun showText (text: String): Unit {
    println(text)
}
```

- If the function only executes an expression, the expression should be placed on the declaration line.

GOOD

```
fun sum (a: Int, b: Int): Int = a + b
```

BAD

```
fun sum (a: Int, b: Int): Int {
     return a + b
}
```

- If the function has an empty body, use the **Unit** type instead of empty bracket body.

GOOD

```
fun foo() = Unit
```

BAD

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

- Should use scope function.

GOOD

```
class Hoge {
    fun fun1() {}
    fun fun2() {}
}
val hoge = Hoge().apply {
   fun1()
   fun2()
}
```

BAD

```
class Hoge {
    fun fun1() {}
    fun fun2() {}
}
val hoge = Hoge()
hoge.fun1()
hoge.fun2()
```

- `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.

GOOD

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

BAD

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

Data class

Only using **data** keyword when class need `equals()`, `hashCode()`, `toString()`, `copy()`, `componentN()` function.

Enum classes

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

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

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

```
enum class Answer {
    YES,
    NO,

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

Generics

When define an object of a **generic** class we don't need define type of that object.

```
class Food<T>(kind: T) {
    var value = kind
}
```

GOOD

```
val fish = Food("Fish")
```

BAD

```
val fish: Food<String> = Food<String>("Fish")
```

Companion Object

- It should be placed at the beginning of its wrapper class.
- Name of constants is upper case and specified with keywords **const val**.

GOOD

```
class MyFragment: Fragment() {
    companion object {
        const val TYPE_VIEW_HEADER = 0
        const val TYPE_VIEW_FOOTER = 1
    }
}
```

BAD

```
class MyFragment: Fragment() {
    companion object {
        val TypeViewHeader = 0
        val TypeViewFooter = 1
    }
}
```

- When you want to use components in companion object, you should call directly.

GOOD

```
class MyClass {
   companion object {
       const val NUMBER = 10
   }
}

val x: Int = MyClass.NUMBER
```

BAD

```
class MyClass {
   companion object {
       const val NUMBER = 10
   }
}

val x: Int = MyClass.Companion.NUMBER
```

Extension function

- Extension function is used when actually needed.
- Do not define extension function of a class in this class.

GOOD

```
class Person(val name: String) {
    ...
}

fun Person.setName(name: String) {
    ...
}
```

BAD

```
class Person(val name: String) {
    ...
   fun Person.setName(name: String) {
        ...
   }
}
```

- Extension property should not use **this** keyword to access to properties of the object applied.

GOOD

```
val <T> List<T>.lastIndex: Int
   get() = size - 1
```

BAD

```
val <T> List<T>.lastIndex: Int
   get() = this.size - 1
```

- Do not create extension functions in same class.

GOOD

```
class Animal {
   fun Person.foo() {
   }
}
```

BAD

```
class Animal {
   fun Animal.foo() {
   }
}
```

- When adding extensions to external classes, create a extension package and make separate files for each type:
  - `StringExtensions.kt`
  - `BitmapExtensions.kt`
  - `...`

Destructuring Declarations

- Parameters should directly destructor without componentN().

GOOD

```
data class Person(var name: String, var age: Int) {
    ...
}

val person = Person("Nguyen Van A", 22)
```

```
val (name, age) = person
println("Name:$name,Age:$age")
```

BAD

```
data class Person(var name: String, var age: Int) {
    ...
}

val person = Person("Nguyen Van A", 22)
val name = person.component1()
val age = person.component2()
println("Name:$name,Age:$age")
```

High-Order Function

- Parameters in high-order function are placed in order: normal parameters, functions.

GOOD

```
fun <T> lock( lock: Lock, body: () -> T) {
    ...
}
```

BAD

```
fun <T> lock(body: () -> T, lock: Lock) {
    ...
}
```

Lambda

- Should ignore **()**, **->**, **name parameter** and use **it** keyword to declare the parameter explicitly if the function has only one parameter as lambda block.

GOOD

```
fun onClick(position: (Int) -> Unit) {
    ...
}

onClick { println(it) }
```

BAD

```
fun onClick(position: (Int) -> Unit) {
    ...
}

onClick({ position -> println(position) })
```

- Should place lambda block out of the round bracket of the function if a function has more than one parameter.

GOOD

```
fun onClick(x: String, data: (Int, String) -> Unit) {
    ...
}
onClick("abc") { position, content ->
   println("Position: $position, Content: $content")
}
```

BAD

```
fun onClick(x: String, data: (Int, String) -> Unit) {
    ...
}

onClick("abc", { position, content ->
   println("Position: $position, Content: $content")
})
```

Inline keyword

- An inline function is used for small functions(1 -> 5 lines of code), not used for big functions.
- The inline modifier can be used on accessors of properties when we do not want to create a backing field.

GOOD

```
inline var bar: Bar
    get() = ...
    set(v) {...}
```

BAD

```
var bar: Bar
    get() = ...
    set(v) {...}
```

Which use run or let

- Use **let** to substitute into functions.
- Use **run** to use outside functions.

```
class Foo
class Bar(val foo: Foo)
class Hoge {
    fun fun1() {}
    fun fun2(bar: Bar) {}
    fun fun3(foo: Foo) {}
}
```

GOOD

```
Hoge().run {
   fun1()
   Foo().let {
       fun2(Bar(it))
       fun3(it)
   }
}
```

BAD

```
Hoge().let {
   it.fun1()
   Foo().run {
       it.fun2(Bar(this))
       it.fun3(this)
   }
}
```

Environments

- Android Studio. download
- Setup Kotlin plugin. link

Security Conventions

- Remember cancel Thread, AsyncTask,… If go out other screen or turn back home screen.
- Remember enable minifyEnabled function in build.gradle while on release mode and config proguard carefully.

```
    buildTypes {
        release {
            debuggable false
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt' ), 'proguard-rules.pro'
            applicationVariants. all { variant ->
            appendVersionNameVersionCode(variant)
            }
        }
    }
```

- If you guys create keystore by yourself, let talk to PM or TL about that, and having backup that keystore carefully and remember information of keystore (keyAlias, keyPassword).
- Don’t push keystore file and information of keystore to GitHub, keep it in local as in local.properties file.
- More practice:
    - Proguard configure
    - Gradle configure

- *local.properties file**

```
sdk. dir=/Users/Administrator/Library/Android/sdk
keyAlias=sampletext1
keyPassword= sampletext2
```

- Verify input validation carefully.
- Verify Runtime Permission function available in Android 6.0 and above carefully. More details
- Avoid leak memory. More details
- Avoid out of memory. More details
- Avoid run time exception

Code management practices and dependency management practices

- Introduce Gradle Tool

References

- Android practices Github
- KotlinLang Page
- Android Developer Page
- Kotlin Code Conventions Page

Code review Checklist

- Pass Circle CI or Travis CI with checkstyle findbug lint tools: 70% follow convention.
- Pass Reviewer: 30% follow convention.
- Github

More References

- Android Boilerplate
- Android folder structure