Android Code Style Guidelines

1. Naming

1.1 Files Naming

1.1.1 Class files

Class names are written in (UpperCamelCase).

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***.

1.1.2 Resource files

Resources file names are written in **lowercase_underscore**.

1.1.2.1 Drawable files

Images

Asset Type	Prefix	Example
Action Bar	ab_	ab_stacked.png
Button	btn_	btn_send_pressed.png
Dialog	dialog_	dialog_top.png
Divider	divider_	divider_horizontal.png

Icon	ic_	ic_star.png
Menu	menu_	menu_submenu_bg.png
Notification	notification_	notification_bg.png
Tabs	tab_	tab_pressed.png

Icons

Naming conventions for icons (taken from Android iconography guidelines)

Asset Type	Prefix	Example
Icons	ic_	ic_star.png
Launcher icons	ic_launcher_	ic_launcher_calendar.png
Menu icons and Action Bar icons	ic_menu_	ic_menu_archive.png
Status bar icons	ic_stat_notify_	ic_stat_notify_msg.png
Tab icons	ic_tab_	ic_tab_recent.png
Dialog icons	ic_dialog_	ic_dialog_info.png

Selectors

Naming conventions for selector states:

State	Suffix	Example
Normal	_normal	btn_order_normal.png
Pressed	_pressed	btn_order_pressed.png
Focused	_focused	btn_order_focused.png
Disabled	_disabled	btn_order_disabled.png
Selected	_selected	btn_order_selected.png

Note: We are trying no use PNG/JPG images, but only vectorial images, therefore this table is a bit deprecated.

1.1.2.2 Layout files

Layout files should match the name of the Android components that they are intended for but moving the top level component name to the beginning. For example, if we are creating a layout for the ***SignInActivity***, the name of the layout file should be ***activity_sign_in.xml***.

Component	Class Name	Layout Name
Activity	UserProfileActivity	activity_user_profile.xml
Fragment	SignUpFragment	fragment_sign_up.xml
Dialog	ChangePasswordDialog	dialog_change_password.xml
AdapterView item	--	item_person.xml

A slightly different case is when we are creating a layout that is going to be inflated by an ***Adapter***, e.g to populate a ***ListView***. In this case, the name of the layout should start with ***item_.***

Note that there are cases where these rules will not be possible to apply.

1.1.2.3 Layout files

Similar to layout files, menu files should match the name of the component.

For example, if we are defining a menu file that is going to be used in the ***UserActivity***, then the name of the file should be **menu_*activity_user.xml***

A good practice is to not include the word menu as part of the name because these files are already located in the menu directory.

1.1.2.1 Value files

Resource files in the values folder should be **plural**, e.g. ***`strings.xml`, `styles.xml`, `colors.xml`, `dimens.xml`, `attrs.xml***

1.2 Resources naming

Resource IDs and names are written in **lowercase_underscore.**

1.2.1 ID naming

IDs should be defined in the following way:

1. Namespace which can be the feature we are working on, or the widget, etc.
2. The purpose of the element
3. The type of the element (btn, text etc).

Element	Suffixes
BottomNavigationView	_bottom_navigation
Button	_button
ConstraintLayout	_constraint_layout
EditText	_edit_text
FrameLayout	_frame_layout
ImageView	_image
LinearLayout	_linear_layout
RecyclerView	_recycler
TextView	_text
Toolbar	_toolbar

1.2.2 Strings

String names start with a prefix that identifies the section they belong to. For example ***registration_email_hint*** or ***registration_name_hint***.

If a string **doesn't belong** to any section, then you should follow the rules below:

Prefix	Description
error_	An error message
msg_	A regular information message
title_	A title, i.e. a dialog title
action_	An action such as "Save" or "Create"

1.2.3 Styles and Themes

Unlike the rest of the resources, style names are written in **UpperCamelCase**.

For example:

```
<style name="FormLabel">
    <item name="android:layout_width">match_parent</item>
    <item name="android:layout_height">wrap_content</item>
    <item name="android:textColor">@color/header_text_color</item>
</style>
```

1.3 Code naming

1.3.1 Class

Class names are written in **UpperCamelCase.**

For example: ***UserProfileActivity***, or **TeaserStandardViewModel.**

1.3.2 String constants, naming, and values

Many elements of the Android SDK such as ***SharedPreferences***, ***Bundle***, or ***Intent*** use a key-value pair approach so it's very likely that even for a small app you end up having to write a lot of **String** constants.

When using one of these components, you **must** define the keys as a **static final** fields and they should be prefixed as indicated below.

Element	Field Name Prefix
Shared Preferences	PREF_
Fragment Arguments	ARG_
Intent Extra	EXTRA_
Intent Action	ACTION_

Note that the arguments of a Fragment - ***Fragment.getArguments()*** - are also a Bundle.

However, because this is a quite common use of Bundles, we define a different prefix for them.

Example:

```
companion object {
    // Note the value of the field is the same as the name to avoid duplication issues
    private const val PREF_EMAIL = "PREF_EMAIL"
    private const val ARG_USER_ID = "ARGUMENT_USER_ID"

    // Intent-related items use full package name as value
    private const val EXTRA_SURNAME = "com.myapp.extras.EXTRA_SURNAME"
    private const val ACTION_OPEN_USER = "com.myapp.action.ACTION_OPEN_USER"
}
```

2. Code style rules

2.1 Treat acronyms as words

Good	Bad
XmlHttpRequest	XMLHTTPRequest
getCustomerId	getCustomerID
url: String	URL: String
id: Long	ID: Long

2.2 Use spaces for indentation

Use **4 space** indents for blocks:

```
if (x == 1) {
    x += 1
}
```

Use **8 space** indents for line wraps:

```
Instrument i =
        someLongExpression(that, wouldNotFit, on, one, line)
```

2.3 Use standard brace style

Braces go on the same line as the code before them.

```
class MyClass {
    fun func(): Int {
        if (something) {
            // ...
        } else if (somethingElse) {
            // ...
        } else {
            // ...
        }
    }
}
```

Braces around the statements are required unless the condition and the body fit on one line.

If the condition and the body fit on one line and that line is shorter than the max line length, then braces are not required, e.g.

```
if (condition) body() else somethingElse()
```

This is **Bad**:

```
if (condition)
    body()  // Bad!
```

This is **Good**:

```
// Please always use curly braces!
if (condition) {
    body() // Good!
}
```

2.4 Limit variable scope

_The scope of local variables should be kept to a minimum (Effective Java Item 29).

By doing so, you increase the readability and maintainability of your code and reduce the likelihood of error.

Each variable should be declared in the innermost block that encloses all uses of the variable._

_Local variables should be declared at the point they are first used.

Nearly every local variable declaration should contain an initializer.

If you don't yet have enough information to initialize a variable sensibly, you should postpone the declaration until you do._

- (Android code style guidelines)

2.5 Logging guidelines

Use the logging methods provided by the ***Timber*** class to print out error messages or other information that may be useful for developers to identify issues:

- ***Timber.v(msg: String)*** (verbose)
- ***Timber.d(msg: String)*** (debug)
- ***Timber.i(msg: String)*** (information)
- ***Timber.w(msg: String)*** (warning)
- ***Timber.e(msg: String)*** (error)

2.6 Ordering

2.6.1 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:

1. Override attributes
2. Public attributes
3. Private attributes
4. Constructors
5. Override methods and callbacks
6. Public methods

7. Private methods
8. Inner classes or interfaces
9. Companion Object

Also, within each class member, it is recommended to follow the alphabetical ordering.

Example:

```
class MainActivity: AppCompatctivity() {

    private textViewTitle: TextView
    private title: String

    override fun onCreate() {
        ...
    }

    fun setTitle(title: String) {
            mTitle = title;
    }

    private fun setUpHeader() {
        ...
    }

    private fun setUpView() {
        ...
    }

    class MyInnerClass {

    }

    companion object {
        private const val TAG = "MainActivity"
    }
}
```

2.6.2 Parameter ordering in methods

Generally parameters should be in **alphabetical** order.

Exception: ***Schedulers*** must always be at first.

```
fun createViewModel(
    ioScheduler: Scheduler,
    uiScheduler: Scheduler,
    mapper: Mapper,
    model: Model,
    repository: Repository
): ViewModel
```

When programming for Android, it is quite common to define methods that take a **Context**.

If you are writing a method like this, then the **Context** must be the **first** parameter.

The opposite case are **callback** interfaces that should always be the **last** parameter.

Examples:

```
// Context always goes first
fun loadUser(context: Context, userId: Int): User

// Callbacks always go last
fun loadUserAsync(Context context, userId: Int, callback: UserCallback)
```

2.7 Line length limit

Code lines should not exceed **100 characters.**
Check the Android Studio configuration to set the 100 characters limit as hard wrap for Kotlin.
If the line is longer than this limit there are usually two options to reduce its length:

- Extract a local variable or method (preferable)
- Apply line-wrapping to divide a single line into multiple ones.

There are two __exceptions__ where it is possible to have lines longer than 100:

1. Lines that are not possible to split, e.g. long URLs in comments.
2. ***package*** and ***import*** statements.

2.7.1 Line-wrapping strategies

There isn't an exact formula that explains how to line-wrap and quite often different solutions are valid. However, there are a few rules that can be applied to common cases.

Assignment Operator Exception

An exception to the break at operators rule is the assignment operator `=`, where the line break should happen **after** the operator.

```
val longName =
        anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne + theFinalOne;
```

Method chain case

When multiple methods are chained in the same line - for example when using Builders - every call to a method should go in its own line, breaking the line before the `.`

```
// Don't do this
Picasso.with(context).load("http://my.cdn/images/image.jpg").into(imageView)

// Do this
Picasso
    .with(context)
    .load("http://my.cdn/images/image.jpg")
    .into(imageView)

// If we need to store the result in a variable, we can do it like that
val result = Picasso
    .with(context)
    .load("http://my.cdn/images/image.jpg")
    .into(imageView)
```

Long parameters case

When a method has many parameters or its parameters are very long, we should break the line after every comma `,`

```
// Don't do this
loadPicture(context, "http://my.cdn/images/image.jpg", mImageViewProfilePicture, clickListener, "Title of
the picture")

// Do this
loadPicture(
    clickListener = clickListener,
    context = context,
    imageView = mImageViewProfilePicture,
    title = "Title of the picture",
    url = "http://my.cdn/images/image.jpg",
)
```

2.8 XML

2.8.1 Use self closing tags

When an XML element doesn't have any contents, you **must** use self closing tags.

```
<!-- Don't do this! -->
<TextView
    android:id="@+id/text_view_profile"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" >
</TextView>

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

2.9 Add new line at the end

Ensure you add a new line at the end of your **Kotlin** files.

You can configure Android Studio to do this job for you with the following steps:

File > Preferences > Editor > General > Ensure line feed at file end on Save

3. Tests style rules

3.1 Unit tests

3.1.1 Class Name

Test classes should match the name of the class the tests are targeting, followed by ***Test***.

For example, if we create a test class that contains tests for the ***DatabaseHelper***, we should name it ***DatabaseHelperTest.***

In case we are testing extensions, and all the extensions are written in a separate file, then we should add the **KtTest** suffix.

For example, for **ActivityExtensions**, the test class name should be **ActivityExtensionsKtTest**.

3.1.2 Tested instance

The instance created for the class under test must be called ***underTest.***

3.1.3 Tests

Test methods are annotated with ***@Test*** and should generally start with the name of the method that is being tested, followed by a precondition and /or expected behaviour.

Test name should express a specific requirement

This requirement should be somehow derived from either a business requirement or a technical requirement.

It is recommended to use back quote notation to be able to write in English as method name.

```
// Template
@Test
fun `method name  given (input)  when (precondition)  then (expected behaviour)`() {
  // given part

  // when part

  // then part
}

// Example
@Test
fun `onCameraPermissionResult  given PHOTO request code  when permission IS GRANTED  it emits
GoToReporterAction with TAKE_PHOTO media source`() {
  val requestCode = REQUEST_CODE_CAMERA_PERMISSION_FOR_PHOTO

  val actionsObserver = underTest.actions.test()
  underTest.onCameraPermissionResult(
    isPermissionGranted = true,
    requestCode = requestCode
  )

  actionsObserver.assertValue { it is GoToReporterAction && it.mediaSource == TAKE_PHOTO }
}
```

Precondition and/or expected behaviour may not always be required if the test is clear enough without them.

Sometimes a class may contain a large amount of methods, that at the same time require several tests for each method.

In this case, it's recommendable to split up the test class into multiple ones.

For example, if the ***DataManager*** contains a lot of methods we may want to divide it into ***DataManagerSignInTest***, ***DataManagerLoadUsersTest***, etc.

3.1.4 Mocks

When a unit test requires a mock instance, then we should use one of the functions that provide mocks (or create it if it is missing).

The file name should match the class name with **Mocks** suffix.

For instance, if we need a mock for the **SportEventDateModel,** the file should be called **SportEventDateModelMocks.**

Create a function called **createMock** which returns a mock with all default values.

Also, create an extension for each field to be able to easily customize the behavior of a mock, when needed.

```
object SportEventDateModelMocks {

    fun createMock(
        date: String = Default.DATE,
        format: String = Default.FORMAT,
        replacer: String = Default.REPLACER,
        template: String = Default.TEMPLATE
    ): SportEventDateModel =
        mockk {
            mockDate() returns date
            mockFormat() returns format
            mockReplacer() returns replacer
            mockTemplate() returns template
        }

    fun SportEventDateModel.mockDate(): MockKStubScope<String, String> =
        every { this@mockDate.date }

    fun SportEventDateModel.mockFormat(): MockKStubScope<String, String> =
        every { this@mockFormat.format }

    fun SportEventDateModel.mockReplacer(): MockKStubScope<String, String> =
        every { this@mockReplacer.replacer }

    fun SportEventDateModel.mockTemplate(): MockKStubScope<String, String> =
        every { this@mockTemplate.template }

    object Default {
        const val DATE = "date"
        const val FORMAT = "format"
        const val REPLACER = "replacer"
        const val TEMPLATE = "template"
    }
}
```

Note. For BLU classes or repositories, where we just need to mock the output of methods, then we can just implement the extensions.

3.2 UI tests

3.2.1 Espresso tests

Every Espresso test class usually targets an Activity, therefore the name should match the name of the targeted Activity followed by ***Test***, e.g. ***SignIn ActivityTest***

When using the Espresso API it is a common practice to place chained methods in new lines.

```
onView(withId(R.id.view))
        .perform(scrollTo())
        .check(matches(isDisplayed()))
```

4. Coding rules

4.1 Don't ignore exceptions

You must **never** do the following:

```
fun setServerPort(value: String) {
    try {
        serverPort = Integer.parseInt(value)
    } catch (e: NumberFormatException) {
        // Exception ignored
    }
}
```

_While you may think that your code will never encounter this error condition or that it is not important to handle it, ignoring exceptions like above creates mines in your code for someone else to trip over some day.

You must handle every Exception in your code in some principled way.

The specific handling varies depending on the case._

(Android code style guidelines)

4.2 Don't catch generic exception

You should not do this:

```
try {
    someComplicatedIOFunction()        // may throw IOException
    someComplicatedParsingFunction()   // may throw ParsingException
    someComplicatedSecurityFunction()  // may throw SecurityException
    // phew, made it all the way
} catch (e: Exception) {               // I'll just catch all exceptions
    handleError()                      // with one generic handler!
}
```

4.3 Don't use finalizers

_We don't use finalizers.

There are no guarantees as to when a finalizer will be called, or even that it will be called at all. In most cases, you can do what you need from a finalizer with good exception handling.

If you absolutely need it, define a `close()` method (or the like) and document exactly when that method needs to be called.

See `InputStream` for an example. In this case it is appropriate but not required to print a short log message from the finalizer, as long as it is not expected to flood the logs._

(Android code style guidelines)

4.4 Fully qualify imports

- This is bad: ***import foo.****
- This is good: ***import foo.Bar***

4.5 Static imports

Use static imports for constants, methods etc.

Exception is only for **enums**. Enums can be imported using default import.

Example:

```
import foo.Baz
import foo.Constants.SOME_CONSTANT

fun example(baz: Baz) {
        when (baz) {
            Baz.VALUE_ONE -> doSomething(message = SOME_CONSTANT)
            Baz.VALUE_TWO -> doSomethingElse(message = SOME_CONSTANT)
        }
}

enum class Baz {
        VALUE_ONE,
        VALUE_TWO
}

object Constants {
        const val SOME_CONSTANT = "some_value"
}
```