**------------------------- Task 1 ---------------------------**

**I. Coding Rules:**

**1. File naming**

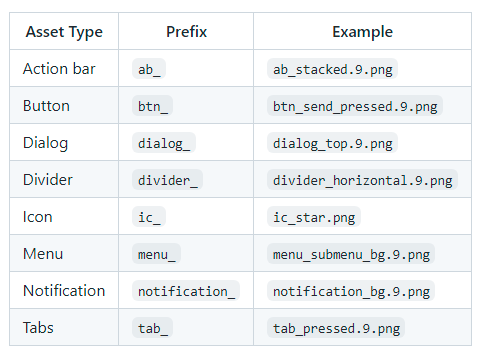
**- Class files**:  [UpperCamelCase](http://en.wikipedia.org/wiki/CamelCase)

example: SignInActivity,SignInFragment,ImageUploaderService,ChangePasswordDialog

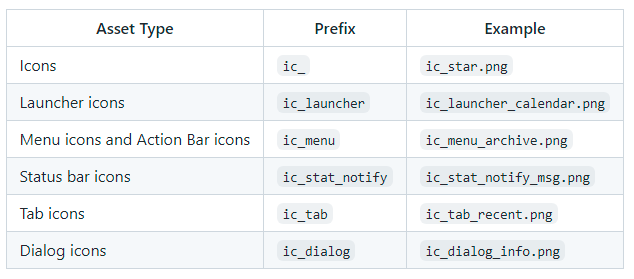
**- Resources files:  lowercase\_underscore.**

- **Drawable files :**

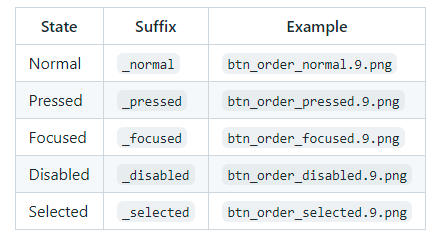
**+** Naming conventions for drawables :

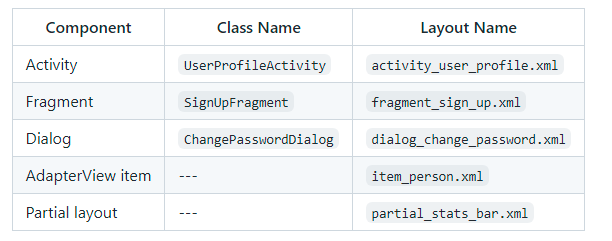


**+** Naming conventions for icons :



**+** Naming conventions for selector states:

****

**- 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. ****

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. For example, when creating layout files that are intended to be part of other layouts. In this case you should use the prefix partial\_.

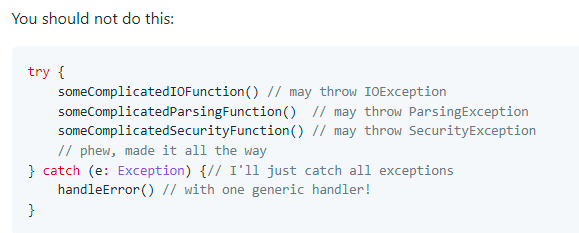
**- Menu files:** Similar to layout files, menu files should match the name of the component. For example, if we are defining a menu file that is going to be used in the UserActivity, then the name of the file should be activity\_user.xml

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

**2. Java language rules:**

**- Don't ignore exceptions :** You must handle every Exception in your code in some principled way. The specific handling varies depending on the case.

- **Don't catch generic exception :**

****

- **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.

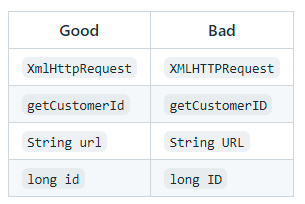
- **Fully qualify imports:**

This is bad: import foo.\*

This is good: import foo.Bar

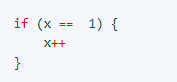
**3. Kotlin language rules:**

- **Treat acronyms as words**



- **Use spaces for indentation:**

Use **4 space** indents for blocks:



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



- **Annotations style:**

**Classes, Methods and Constructors**

When annotations are applied to a class, method, or constructor, they are listed after the documentation block and should appear as **one annotation per line** .

****

**Fields**

Annotations applying to fields should be listed **on the same line**, unless the line reaches the maximum line length.

****

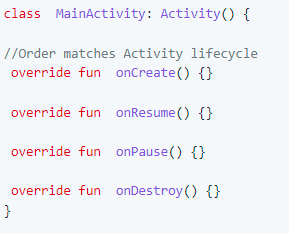
- **Class member ordering:**

There is no single correct solution for this but usin a **logical** and **consistent** order will improve code learnability and readability. It is recommendable to use following order:

1. Constants
2. Fields
3. Constructors
4. Override methods and callbacks (public or private)
5. Public methods
6. Private methods
7. Inner classes or interfaces



If your class is extending an **Android component** 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(), then the correct order is:

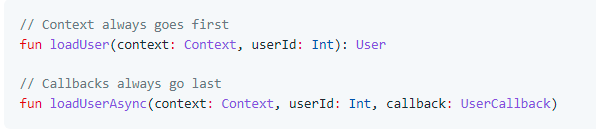


- **Parameter ordering in methods :**

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.

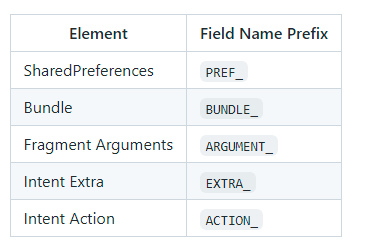
Examples:

****

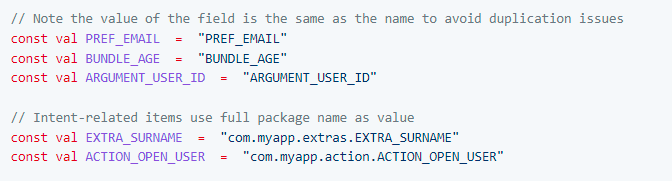
- **String constants, naming, and values :**

Many elements of the Android SDK such as SharedPreferences, Bundle, or Intent use a key-value

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

****

Example:

****

- **Line length limit :**

Code lines should not exceed **100 characters**. 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:

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

- **Line-wrapping strategies :**

**Break at operators**

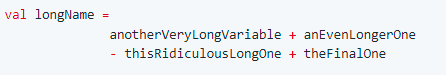
When the line is broken at an operator, the break comes **before** the operator. For example:

val longName = anotherVeryLongVariable + anEvenLongerOne- thisRidiculousLongOne

* theFinalOne

**Assignment Operator Exception**

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



**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 .



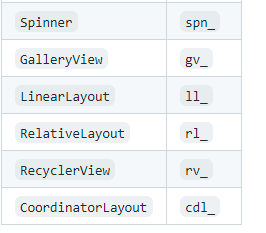
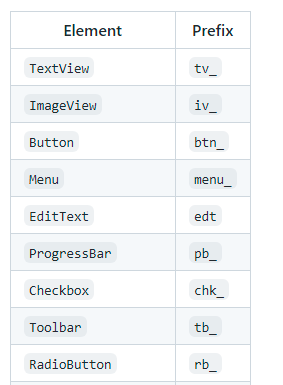
**Long parameters case**

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



- **Resources naming :** Resource IDs and names are written in **lowercase\_underscore**.

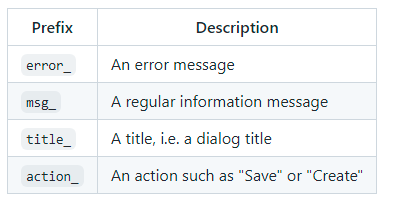
- **ID naming :**

****

****

- **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:

****

- **Styles and Themes :** Unlike the rest of resources, style names are written in **UpperCamelCase**.

- **Attributes ordering:**

As a general rule you should try to group similar attributes together. A good way of ordering the most common attributes is:

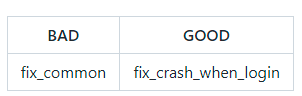
1. View Id
2. Style
3. Layout width and layout height
4. Other layout attributes, sorted alphabetically
5. Remaining attributes, sorted alphabetically

**4. Workflows**

- **Attributes ordering:**

* Branch name must be cleared
* Easy to get meaning
* After branch has been merged > it must be deleted!

Example: update\_user\_profile

****

- **Rebase - Single commits :**

* Always create PR and send to reviewer
* ONLY 1 commit in 1 PR
* Less than 10 files changed in 1 commit

- **NO HARD CODE - NO DEAD CODE :**

* No hard string > define constants or common function
* If code didn't use anymore, please delete

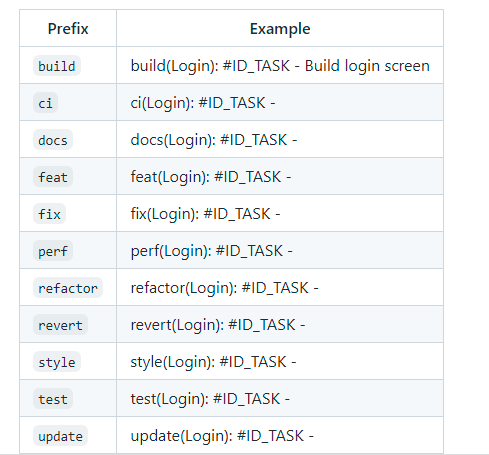
- **Code commits :**

EVERY COMMIT must be followed this rule:

[action](scope function): [message detail].

* action: should be in: [build, ci, chore, docs, feat, fix, perf, refactor, revert, style, test, update]
* scope function: should be task/function implement
* message detail: clear message about your task or what you did on this commit

Example: update(Login): #ID\_TASK - Changed login logics

****