# **ANDROID CODE STANDARD**

# ***Last edit: Phatvt***

# **1. Project guidelines**

## **1.1 Project structure**

New projects should follow the Android Gradle project structure that is defined on the [Android Gradle plugin user guide](http://tools.android.com/tech-docs/new-build-system/user-guide#TOC-Project-Structure). The [BoilerPlate](https://github.com/ribot/android-boilerplate) project is a good reference.

## **1.2 Package structure**

#### **App**

* ui
  + fragment
  + activity
  + dialog
  + widget
  + adapter
* service
* data
  + remote
  + local
  + model
* util

## **1.3 File naming**

### **1.3.1 Class files**

Class names are written in [UpperCamelCase](http://en.wikipedia.org/wiki/CamelCase).

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.

### **1.3.1 Resources files**

Resources file names are written in **lowercase\_underscore**.

#### **1.3.1.1 Drawable files**

Naming conventions for drawables:

|  |  |  |
| --- | --- | --- |
| **Asset Type** | **Prefix** | **Example** |
| Action bar | ab\_ | ab\_stacked.9.png |
| Button | btn\_ | btn\_send\_pressed.9.png |
| Dialog | dialog\_ | dialog\_top.9.png |
| Divider | divider\_ | divider\_horizontal.9.png |
| Icon | ic\_ | ic\_star.png |
| Menu | menu\_ | menu\_submenu\_bg.9.png |
| Notification | notification\_ | notification\_bg.9.png |
| Tabs | tab\_ | tab\_pressed.9.png |

***Naming conventions for icons (taken from*** [***Android iconography guidelines***](http://developer.android.com/design/style/iconography.html)***):***

|  |  |  |
| --- | --- | --- |
| **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 |

***Naming conventions for selector states:***

|  |  |  |
| --- | --- | --- |
| **State** | **Suffix** | **Example** |
| Normal | \_normal | btn\_order\_normal.9.png |
| Pressed | \_pressed | btn\_order\_pressed.9.png |
| Focused | \_focused | btn\_order\_focused.9.png |
| Disabled | \_disabled | btn\_order\_disabled.9.png |
| Selected | \_selected | btn\_order\_selected.9.png |

#### 

#### **1.3.1.2 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 |
| Partial layout | --- | partial\_stats\_bar.xml |

A slighly 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\_

#### **1.3.1.3 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 use in the UserActivity, then the name of the file should be activity\_user.xml

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

#### **1.3.1.4 Values files**

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

# 

# **2. Code guidelines**

## **2.1 Java language rules**

### **2.1.1 Don't ignore exceptions**

You must never do the following:

void setServerPort(String value) {  
 try {  
 serverPort = Integer.parseInt(value);  
 } catch (NumberFormatException e) { }  
}

*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](https://source.android.com/source/code-style.html))

See alternatives [here](https://source.android.com/source/code-style.html#dont-ignore-exceptions).

### **2.1.2 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 (Exception e) { // I'll just catch all exceptions   
 handleError(); // with one generic handler!  
}

See the reason why and some alternatives [here](https://source.android.com/source/code-style.html#dont-catch-generic-exception)

### **2.1.3 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](https://source.android.com/source/code-style.html#dont-use-finalizers))

### **2.1.4 Fully qualify imports**

This is bad: import foo.\*;

This is good: import foo.Bar;

See more info [here](https://source.android.com/source/code-style.html#fully-qualify-imports)

## **2.2 Java style rules**

### **2.2.1 Fields definition and naming**

Fields should be defined at the **top of the file** and they should follow the naming rules listed below.

* Private, non-static field names start with **m**.
* Private, static field names start with **s**.
* Other fields start with a lower case letter.
* Static final fields (constants) are ALL\_CAPS\_WITH\_UNDERSCORES.

Example:

public class MyClass {  
 public static final int SOME\_CONSTANT = 42;  
 public int publicField;  
 private static MyClass sSingleton;  
 int mPackagePrivate;  
 private int mPrivate;  
 protected int mProtected;  
}

**2.2.3 Treat acronyms as words**

|  |  |
| --- | --- |
| **Good** | **Bad** |
| XmlHttpRequest | XMLHTTPRequest |
| getCustomerId | getCustomerID |
| String url | String URL |
| long id | long ID |

### 

### **2.2.4 Use spaces for indentation**

Use **4 space** idents for blocks:

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

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

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

### **2.2.5 Use standard brace style**

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

~~class MyClass {  
 int func() {  
 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 do~~ **~~not~~** ~~use braces e.g.~~

~~if (condition) body();~~

~~This is~~ **~~bad~~**~~:~~

~~if (condition)  
 body(); // bad!~~

Good :

if (condition)

{  
 body();

}

### **2.2.6 Use standard Java annotations**

According to the Android code style guide, the standard practices for some of the predefined annotations in Java are:

* @Override: The @Override annotation **must be used** whenever a method overrides the declaration or implementation from a super-class. For example, if you use the @inheritdocs Javadoc tag, and derive from a class (not an interface), you must also annotate that the method @Overrides the parent class's method.
* @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.

More information about annotations guidelines can be found [here](http://source.android.com/source/code-style.html#use-standard-java-annotations).

### **2.2.7 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](https://source.android.com/source/code-style.html#limit-variable-scope))

### **2.2.8 Order import statements**

If you are using an IDE such as Android Studio, you don't have to worry about this because your IDE is already obeying these rules. If not, have a look below.

The ordering of import statements is:

1. Android imports
2. Imports from third parties (com, junit, net, org)
3. java and javax
4. Same project imports

To exactly match the IDE settings, the imports should be:

* Alphabetical within each grouping, with capital letters before lower case letters (e.g. Z before a).
* There should be a blank line between each major grouping (android, com, junit, net, org, java, javax).

More info [here](https://source.android.com/source/code-style.html#limit-variable-scope)

### **2.2.9 Logging guidelines**

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

* Log.v(String tag, String msg) (verbose)
* Log.d(String tag, String msg) (debug)
* Log.i(String tag, String msg) (information)
* Log.w(String tag, String msg) (warning)
* Log.e(String tag, String msg) (error)

As a general rule, we use the class name as tag and we define it as a static final field at the top of the file. For example:

public class MyClass {  
 private static final String TAG = "MyClass";  
  
 public myMethod() {  
 Log.e(TAG, "My error message");  
 }  
}

VERBOSE and DEBUG logs **must** be disable on relase builds. It is also recommendable to disable INFORMATION, WARNING and ERROR logs but you may want to keep them enable if you think they may be useful to identify issues on release builds. If you decide to leave them enable, you have to make sure that they are not leaking private information such as email addresses, user ids, etc.

To only show logs on debug builds:

if (BuildConfig.DEBUG) Log.d(TAG, "The value of x is " + x);

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

Example:

public class MainActivity extends Activity {  
  
 private String mTitle;  
 private TextView mTextViewTitle;  
  
 public void setTitle(String title) {  
 mTitle = title;  
 }  
  
 @Override   
 public void onCreate() {  
 ...  
 }  
  
 private void setUpView() {  
 ...  
 }  
  
 static class AnInnerClass {  
  
 }  
  
}

If your class is extending and **Android component** such as an Activity or a Fragment, it is a good practise 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:

public class MainActivity extends Activity {  
  
 //Order matches Activity lifecycle   
 @Override   
 public void onCreate() {}   
  
 @Override   
 public void onResume() {}  
  
 @Override   
 public void onPause() {}  
  
 @Override   
 public void onDestory() {}  
  
}

### **2.2.11 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:

// Context always go first  
public User loadUser(Context context, int userId);  
  
// Callbacks always go last  
public void loadUserAsync(Context context, int userId, UserCallback callback);

### **2.2.13 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 indicaded below.

|  |  |
| --- | --- |
| **Element** | **Field Name Prefix** |
| SharedPreferences | PREF\_ |
| Bundle | BUNDLE\_ |
| Fragment Arguments | ARGUMENT\_ |
| 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:

// Note the value of the field is the same as the name to avoid duplication issues  
static final String PREF\_EMAIL = "PREF\_EMAIL";  
static final String BUNDLE\_AGE = "BUNDLE\_AGE";  
static final String ARGUMENT\_USER\_ID = "ARGUMENT\_USER\_ID";  
  
// Intent-related items use full package name as value  
static final String EXTRA\_SURNAME = "com.myapp.extras.EXTRA\_SURNAME";  
static final String ACTION\_OPEN\_USER = "com.myapp.action.ACTION\_OPEN\_USER";

### **2.2.14 Arguments in Fragments and Activities**

When data is passed into an Activityor Fragment via Intents or a Bundles, the keys for the different values **must** follow the rules described in the section above.

When an Activity or Fragment expect arguments, it should provide a static public method that facilitates the creation of the Fragment or Intent.

In the case of Activities the method is usually called getStartIntent()

public static Intent getStartIntent(Context context, User user) {  
 Intent intent = new Intent(context, ThisActivity.class);  
 intent.putParcelableExtra(EXTRA\_USER, user);  
 return intent;  
}

For Fragments it's named newInstance() and it handles the creation of the Fragment with the right arguments.

public static UserFragment newInstance(User user) {  
 UserFragment fragment = new UserFragment;  
 Bundle args = new Bundle();  
 args.putParcelable(ARGUMENT\_USER, user);  
 fragment.setArguments(args)  
 return fragment;  
}

**Note 1**: these methods should go at the top of the class before onCreate()

**Note 2**: if we provide the methods described above, the keys for extras and arguments should be private because there is not need for them to be exposed outside the class.

### **~~2.2.15 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 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.~~

#### **2.2.15.1 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.

**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 .

Picasso.with(context).load("http://ribot.co.uk/images/sexyjoe.jpg").into(imageView);

Picasso.with(context)  
 .load("http://ribot.co.uk/images/sexyjoe.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 ,

loadPicture(context, "http://ribot.co.uk/images/sexyjoe.jpg", mImageViewProfilePicture, clickListener, "Title of the picture");

loadPicture(context,  
 "http://ribot.co.uk/images/sexyjoe.jpg",  
 mImageViewProfilePicture,  
 clickListener,  
 "Title of the picture");

### **2.2.16 RxJava chains styling**

Rx chains of operators require line-wrapping. Every operator must go in a new line and the line should be broken before the .

public Observable<Location> syncLocations() {  
 return mDatabaseHelper.getAllLocations()  
 .concatMap(new Func1<Location, Observable<? extends Location>>() {  
 @Override  
 public Observable<? extends Location> call(Location location) {  
 return mConcurService.getLocation(location.id);  
 }  
 })  
 .retry(new Func2<Integer, Throwable, Boolean>() {  
 @Override  
 public Boolean call(Integer numRetries, Throwable throwable) {  
 return throwable instanceof RetrofitError;  
 }  
 });  
}

## **2.3 XML style rules**

### **2.3.1 Use self closing tags**

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

This is good:

<TextView  
 android:id="@+id/text\_view\_profile"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content" />

This is **bad** :

<!-- Don't do this! -->  
<TextView  
 android:id="@+id/text\_view\_profile"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content" >  
</TextView>

### **2.3.2 Resources naming**

Resource IDs and names are written in **lowercase\_underscore**

#### **2.3.2.1 ID naming**

IDs should be prefixed with the name of the element in lowercase underscore. For example:

|  |  |
| --- | --- |
| **Element** | **Prefix** |
| TextView | text\_ |
| ImageView | image\_ |
| Button | button\_ |
| Menu | menu\_ |

Image view example:

<ImageView  
 android:id="@+id/image\_profile"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content" />

Menu example:

<menu>  
 <item  
 android:id="@+id/menu\_done"  
 android:title="Done" />  
</menu>

#### **2.3.2.2 Strings**

String names start with a prefix that indentifies 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" |

#### **2.3.2.3 Styles and Themes**

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

### **2.3.3 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

## **2.4 Tests style rules**

### **2.4.1 Unit tests**

The test classes should match the name of the class that the tests are targeting followed by Test. For example, If we create a test class that contains test for the DataManager, we should name it DataManagerTest.

The name of the tests must start with should followed by the expected behaviour. For example:

* shouldLoadUserData()
* shouldThrowExceptionWhenLoadingUser()

### **2.4.2 Espresso tests**

Every Espresso test class must target an Activity, therefore the name should match the name of the targeted Activity followed by Test, e.g. SignInActivityTest

When using the Espresso api is a common practise to place chained method in new lines.

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