Android Project Guideline V1

Why?

Because each time we start a project we found few ways to develop the app. Its human nature to do it differently each time. But as a large organization with bunch of developer we should strict to a proper guideline. And each of us should follow the guideline.

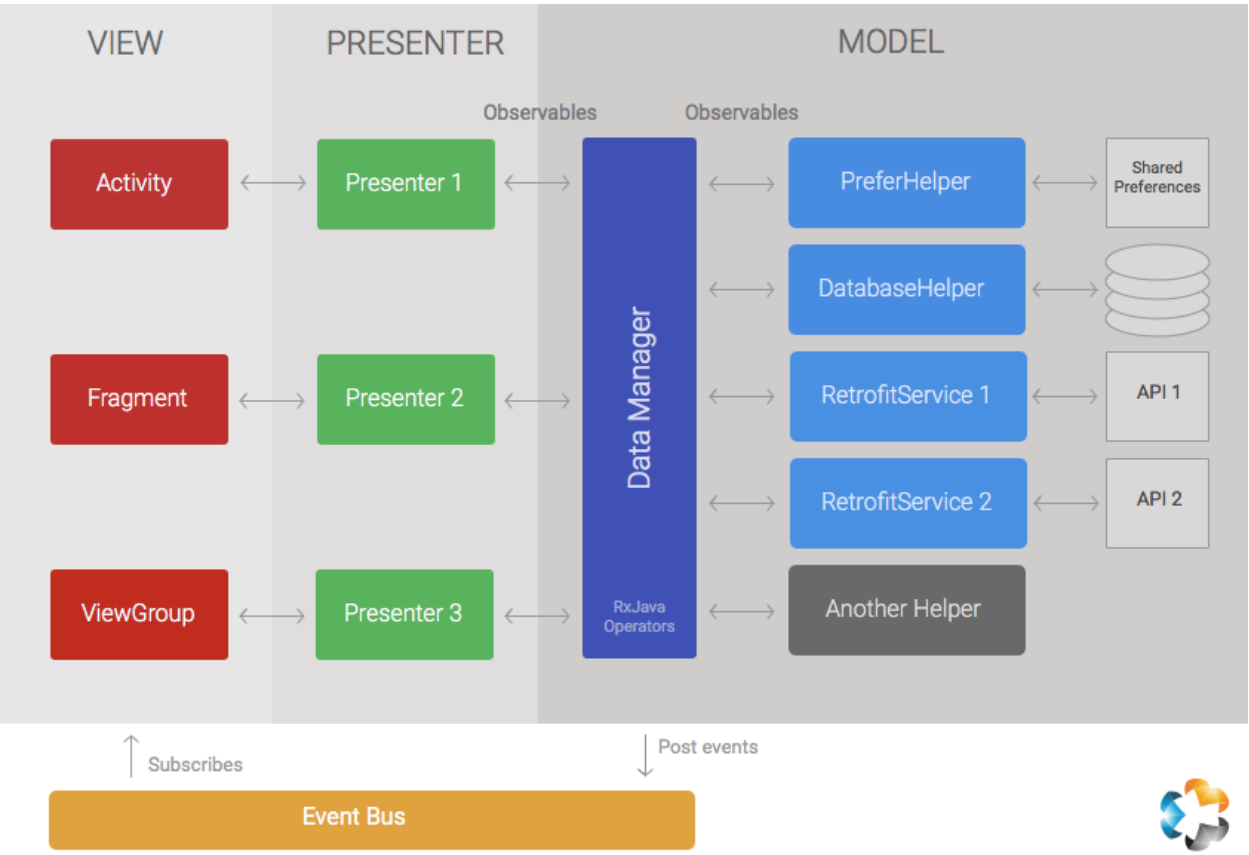
What will be the architecture/strategy?

MVP (model view presenter) for now.

Project Architecture

The architecture of our Android apps is based on the MVP (Model View Presenter) pattern.

- ✓ **View (UI layer)**: this is where Activities, Fragments and other standard Android components live. It's responsible for displaying the data received from the presenters to the user. It also handles user interactions and inputs (click listeners, etc) and triggers the right action in the Presenter if needed.
- ✓ **Presenter**: presenters subscribe to RxJava Observables provided by the *DataManager*. They are in charge of handling the subscription lifecycle, analysing/modifying the data returned by the *DataManager* and calling the appropriate methods in the View in order to display the data.
- ✓ **Model (Data Layer)**: this is responsible for retrieving, saving, caching and massaging data. It can communicate with local databases and other data stores as well as with restful APIs or third party SDKs. It is divided in two parts: a group of helpers and a *Manager*. The number of helpers vary between project and each of them has a very specific function, e.g. talking to an API or saving data in *SharedPreferences*. The *Manager* combines and transforms the outputs from different helpers using Rx operators so it can:
    1. Provide meaningful data to the Presenter,
    2. Group actions that will always happen together. This layer also contains the actual model classes that define how the data structure is.



Looking at the diagram from right to left:

- ✓ **Helpers (Model):** A set of classes, each of them with a very specific responsibility. Their function can range from talking to APIs or a database to implementing some specific business logic. Every project will have different helpers but the most common ones are:
  - o **DatabaseHelper**: It handles inserting, updating and retrieving data from a local SQLite database. Its methods return Rx Observables that emit plain java objects (models)
  - o **PreferencesHelper**: It saves and gets data from *SharedPreferences*, it can return Observables or plain java objects directly.
  - o **Retrofit services**: Retrofit interfaces that talk to Restful APIs, each different API will have its own Retrofit service. They return Rx Observables.
- ✓ **Data Manager (Model):** It's a key part of the architecture. It keeps a reference to every helper class and uses them to satisfy the requests coming from the presenters. Its methods make extensive use of Rx operators to combine, transform or filter the output coming from the helpers in order to generate the desired output ready for the Presenters. It returns observables that emit data models.
- ✓ **Presenters**: Subscribe to observables provided by the *DataManager* and process the data in order to call the right method in the View.

- ✓ **Activities, Fragments, ViewGroups (View):** Standard Android components that implement a set of methods that the Presenters can call. They also handle user interactions such as clicks and act accordingly by calling the appropriate method in the Presenter. These components also implement framework-related tasks such as managing the Android lifecycle, inflating views, etc.
- ✓ **Event Bus:** It allows the View components to be notified of certain types of events that happen in the Model. Generally the *DataManager* posts events which can then be subscribed to by Activities and Fragments. The event bus is only used for very specific actions that are not related to only one screen and have a broadcasting nature, e.g. the user has signed out.

Project Guideline

Git repo and maintenance

A git repo must be created first. Scrum master should create it, and assign permissions among team.

- Each project must contain a README.md file which contain:
  - What this project all about?
  - How to setup environment?
  - How the project organized?
- Before development a *development* branch should create. Everyone should create their own branch from that *development* branch with convention as *FEATURE_JIRA_TICKET_TASK_HINT* e.g. *FEATURE_BNA-123_Upload_image*. Soon he/she finish his task, he/she should push and request for merge with *development* branch.
- For release a well-tested production ready app should marge from *development* to *master* branch. Android **keystore** for app release should pass to top level management via email. Make sure you have putted a *TAG* for each release on git.
- For bugs, a hotfix branch should create first from the release branch with format of *HOTFIX_JIRA_TICKET_TASK_HINT* and sync between development and master branch.

Project Structure

New projects should follow the Android Gradle project structure that is defined on the Android Gradle plugin user guide.

To start a new project go here and download the ZIP from here of master branch.

Then change project name and project package name. See here

N.B: There is no need to push back to this repo. Made your own project repo from that.

Package name

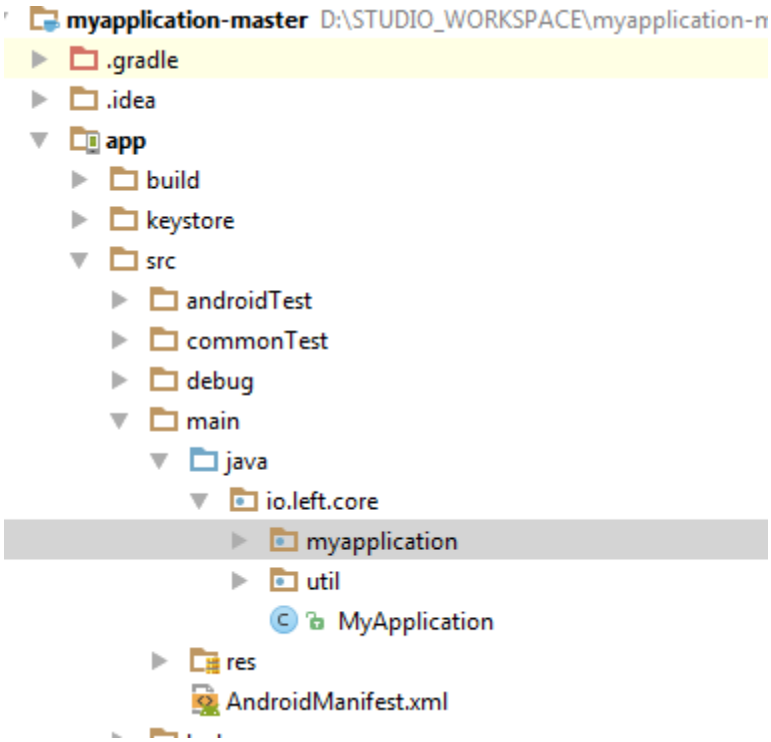
For w3engineers in-house project **com.w3engineers.core**

For left project **io.left.core**

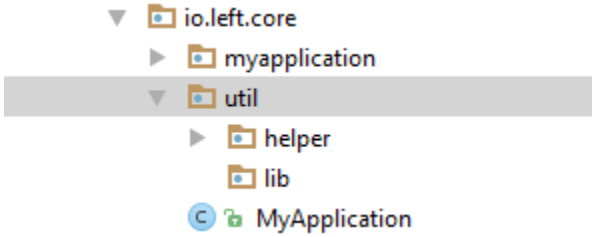
Under core package there will be 2 more package

1. myapplication [N.B: this will be changed according to each individual app name]

2. util [all independent tasks that we can use on each individual projects will be here]



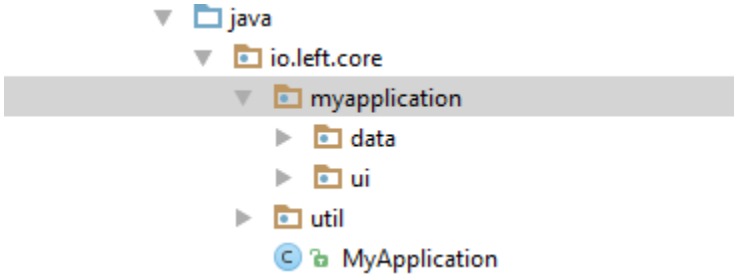
util package structure



util contains 2 inner package named

- ✓ helper [All generic tasks like TimeUtil, NetworkUtil will be here]
- ✓ lib [All 3rd party library that is related to data like read / write or push or anything where data is accessed must have a util class under lib folder. Application packages will communicate via this util class.]

myapplication package structure:

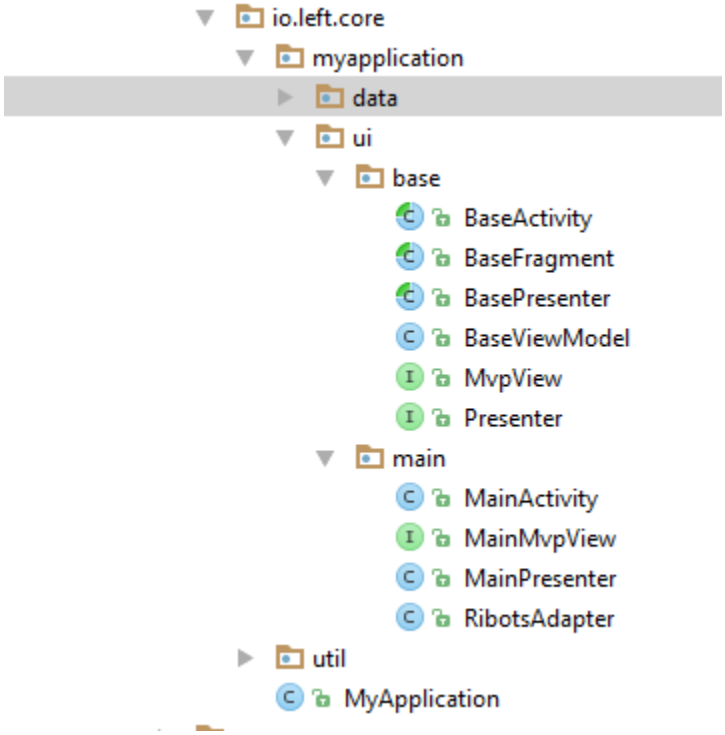


myapplication package contain 2 inner package named

- ✓ data [local database, shared preference, file, and remote server tasks will be here]
- ✓ ui [all activity/ fragment/ or other view components will be here]

NB: Only these 2 packages will remain here always. It is strongly recommended.

ui package structure



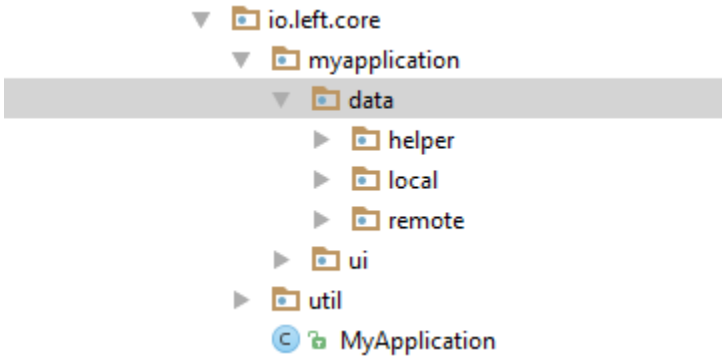
This will follow the MVP pattern for each view components. For each view component like Activity/ Fragment a separate package will be created under ui package.

for MainActivity here we create a package named main. for any new Activity a new package will be created with that Activity name.

- ✓ Each Activity must extend BaseActivity.
- ✓ Each Fragment must extend BaseFragment.
- ✓ Each Presenter must extend BasePresenter.
- ✓ Each View must implement BaseView (**MvpView**).

data package structure

data package contain 3 main package:



- ✓ helper
- ✓ local
- ✓ remote

helper package structure

here helper package is used to make more flexibility.

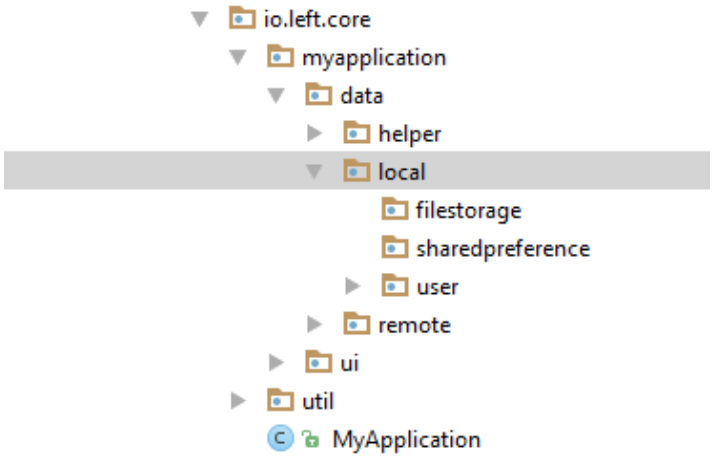
remote package structure

remote package contain all server related task that includes:

- ✓ httprequest (okhttp3.0 / retrofit2.0)
- ✓ firebase push

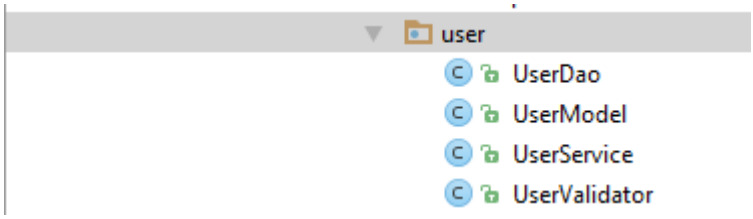
local package structure:

local package contain all local storage related task that includes:



- ✓ shared preference [sharedpreference]
- ✓ file r/w [filestorage]
- ✓ sqlite db [Room persistency library for now]

user package structure



For Room persistent library since it already use model and dao for each table so here we will create separate package for each individual table.

here user is a table name.

- ✓ UserDao [room dao]
- ✓ UserModel [room model]
- ✓ UserService [manger for each table which receives data from Application layer to process data and then insert into db or fetch data from db and send back to application layer. it will be observable to broadcast data to app layer's observers.]
- ✓ UserValidator [(**Optional**) validation type task will be conducted here by Service Manger]

File naming

General Guidance while you start

- For each class and methods under that should contain,
    - Author
    - Purpose (not less than 50 character)
    - Date of creation
- Key idea here is, anyone can easily know about the class + method by reading and can ask more from the author
- Before adding/changing to any existing class/method a meeting should call including the author and decide by the agreement of each member of the team (junior to senior).

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.

Resource files

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

Drawable files

Naming conventions for drawables:

Asset Type	**Prefix**	**Example**
Action bar	actionbar_	actionbar_stacked.9.png
Button	button_	button_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

Asset Type	Prefix	Example
Notification	notification_	notification_bg.9.png
Tabs	tab_	tab_pressed.9.png

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

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

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 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**

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.

Values Files

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

Code Guideline

Java Language Rules

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)

See alternatives here.

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

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)

Fully qualify imports

This is bad: `import foo.*;`
This is good: `import foo.Bar;`
See more info here

Java style rules

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;
}
```

Treat acronyms as words

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

Use spaces for indentation

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

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

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

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 braces are not required, e.g.

```
if (condition) body();
```

This is **bad**:

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

Annotations

Annotations practices

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 annotation guidelines can be found here.

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** .

```
/* This is the documentation block about the class */
@AnnotationA
@AnnotationB
public class MyAnnotatedClass { }
```

Fields

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

```
@Nullable @Mock DataManager mDataManager;
```

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)

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:

- Alphabetically ordered 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

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 identify 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.class.getSimpleName();

    public myMethod() {
        Log.e(TAG, "My error message");
    }
}
```

VERBOSE and DEBUG logs **must** be disabled on release builds. It is also recommended to disable INFORMATION, WARNING and ERROR logs but you may want to keep them enabled if you think they may be useful to identify issues on release builds. If you decide to leave them enabled, 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);
```

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 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:

```
public class MainActivity extends Activity {

        //Order matches Activity lifecycle
    @Override
    public void onCreate() {}

    @Override
    public void onResume() {}

    @Override
    public void onPause() {}

    @Override
    public void onDestroy() {}

}
```

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 goes first
public User loadUser(Context context, int userId);
```

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

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
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";
```

Arguments in Fragments and Activities

When data is passed into an **Activity** or **Fragment** via an **Intent** or a **Bundle**, the keys for the different values **must** follow the rules described in the section above.
When an **Activity** or **Fragment** expects arguments, it should provide a public **static** method that facilitates the creation of the relevant **Intent** or **Fragment**.
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 is named **newInstance()** and 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 no need for them to be exposed outside the class.

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

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.

Break at operators

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

```
int 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.

```
int 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 **.**

```
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);
```

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");
```

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 mRetrofitService.getLocation(location.id);
                 }
            })
            .retry(new Func2<Integer, Throwable, Boolean>() {
                 @Override
                 public Boolean call(Integer numRetries, Throwable throwable) {
                     return throwable instanceof RetrofitError;
                 }
            });
}
```

XML style rules

Use self closing tags

When an XML element doesn't have any contents, 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>
```

Resources naming

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

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>
```

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

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"

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

Tests style rules

Unit tests

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`.

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 behavior.

- Template: `@Test void methodNamePreconditionExpectedBehaviour()`
- Example: `@Test void signInWithEmptyEmailFails()`

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. Generally you will be able to see what tests belong together because they have common test fixtures.

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. `SignInActivityTest`
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()))
```