Branch: master ▾ Find file Copy path

android-guidelines / project_style_guidelines.md

radarhere Fixed typos

99a585b on Aug 26, 2016

7 contributors

Raw Blame History

1240 lines (712 sloc) 40.8 KB

Android Project Guidelines

The aim of this document is to define project guidelines. These should be followed throughout the Android project in order to help us to keep our code base clean and consistent. A lot of this will be enforced by code quality checks through Jenkins, but it's important to be mindful of the things that may not be picked up 🙂

1. Project Guidelines

1.1 Project Structure

When contributing work, the project should maintain the following structure:

```
src/androidTest
src/test
src/commonTest
src/main
```

androidTest - Directory containing functional tests
test - Directory containing unit tests
commonTest - Directory containing shared test code for AndroidTest & Test
main - Directory containing application code

The structure of the project should remain as defined above whenever you are modifying or adding new features.

Using this structure allows us to keep the application code separated from any test-related code. The CommonTest directory allows us to share classes between the functional and unit tests, such as mock model creation and dagger test configuration classes.

1.2 File Naming

1.2.1 Class Files

Any classes that you define should be named using UpperCamelCase, for example:

```
AndroidActivity, NetworkHelper, UserFragment, PerActivity
```

Any classes extending an Android framework component should **always** end with the component name. For example:

```
UserFragment, SignUpActivity, RateAppDialog, PushNotificationServer,
NumberView
```

We use UpperCamelCase as this helps to seperate the words used to create the name, making it easier to read. Naming classes to end with the framework component makes it super clear as to what the class is used for. For example, if you're looking to make changes to the RegistrationDialog then this naming convention makes it really easy to locate that class.

1.2.1 Resource Files

When naming resource files you should be sure to name them using lowercase letters and underscores instead of spaces, for example:

```
activity_main, fragment_user, item_post
```

This convention again makes it really easy to locate the specific layout file that you're looking for. Within android studio, the layout package is sorted in alphabetical order meaning that activity, fragment and other layout types becomes grouped - so you know where to begin looking for a file. Other than this, beginning the file name with the component name makes it clear what component/class the layout file is being used for.

1.2.2.1 Drawable Files

Drawable resource files should be named using the **ic_** prefix along with the size and color of the asset. For example, white accept icon sized at 24dp would be named:

```
ic_accept_24dp_white
```

And a black cancel icon sized at 48dp would be named:

```
ic_cancel_48dp_black
```

We use this naming convention so that a drawable file is recognisable by its name. If the colour and size are not stated in the name, then the developer needs to open the drawable file to find out this information. This saves us a little bit of time :)

Other drawable files should be named using the corresponding prefix, for example:

Type	Prefix	Example
Selector	selector_	selector_button_cancel
Background	bg_	bg_rounded_button
Circle	circle_	circle_white
Progress	progress_	progress_circle_purple
Divider	divider_	divider_grey

This convention again helps to group similar items within Android Studio. It also makes it clear as to what the item is used for. For example, naming a resource button_cancel could mean anything! Is this a selector resource or a rounded button background? Correct naming helps to clear any ambiguity that may arise.

When creating selector state resources, they should be named using the corresponding suffix:

State	Suffix	Example
Normal	_normal	btn_accept_normal
Pressed	_pressed	btn_accept_pressed
Focused	_focused	btn_accept_focused

State	Suffix	Example
Disabled	_disabled	btn_accept_disabled
Selected	_selected	btn_accept_selected

Using clear prefixes such as the above helps to make it absolutely obvious as to what a selector state resource is used for. Prefixing resources with the colour or any other identifier again requires the developer to open the selector file to be educated in what the different selector state resources are.

1.2.2.2 Layout Files

When naming layout files, they should be named starting with the name of the Android Component that they have been created for. For example:

Component	Class Name	Layout Name
Activity	MainActivity	activity_main
Fragment	MainFragment	fragment_main
Dialog	RateDialog	dialog_rate
Widget	UserProfileView	view_user_profile
AdapterView Item	N/A	item_follower

Note: If you create a layout using the merge tag then the layout_ prefix should be used.

Not only does this approach makes it easy to find files in the directory hierarchy, but it really helps when needing to identify what corresponding class a layout file belongs to.

1.2.2.3 Menu Files

Menu files do not need to be prefixed with the menu_ prefix. This is because they are already in the menu package in the resources directory, so it is not a requirement.

1.2.2.4 Values Files

All resource file names should be plural, for example:

```
attrs.xml, strings.xml, styles.xml, colors.xml, dimens.xml
attrs.xml, strings.xml, styles.xml, colors.xml, dimens.xml
```

2. Code Guidelines

2.1 Java Language Rules

2.1.1 Never ignore exceptions

Avoid not handling exceptions in the correct manner. For example:

```
public void setUserId(String id) {
        try {
        mUserId = Integer.parseInt(id);
        } catch (NumberFormatException e) { }
}
```

This gives no information to both the developer and the user, making it harder to debug and could also leave the user confused if something goes wrong. When catching an exception, we should also always log the error to the console for debugging purposes and if necessary alert the user of the issue. For example:

```
public void setCount(String count) {
        try {
        count = Integer.parseInt(id);
        } catch (NumberFormatException e) {
                count = 0;
        Log.e(TAG, "There was an error parsing the count " + e);

DialogFactory.showErrorMessage(R.string.error_message_parsing_count);
        }
}
```

Here we handle the error appropriately by:

- Showing a message to the user notifying them that there has been an error
- Setting a default value for the variable if possible
- Throw an appropriate exception

2.1.2 Never catch generic exceptions

Catching exceptions generally should not be done:

```
public void openCustomTab(Context context, Uri uri) {
        Intent intent = buildIntent(context, uri);
        try {
        context.startActivity(intent);
        } catch (Exception e) {
        Log.e(TAG, "There was an error opening the custom tab " + e);
        }
}
```

Why?

Do not do this. In almost all cases it is inappropriate to catch generic Exception or Throwable (preferably not Throwable because it includes Error exceptions). It is very dangerous because it means that Exceptions you never expected (including RuntimeExceptions like ClassCastException) get caught in application-level error handling. It obscures the failure handling properties of your code, meaning if someone adds a new type of Exception in the code you're calling, the compiler won't help you realize you need to handle the error differently. In most cases you shouldn't be handling different types of exception the same way. - taken from the Android Code Style Guidelines

Instead, catch the expected exception and handle it accordingly:

```
public void openCustomTab(Context context, Uri uri) {
        Intent intent = buildIntent(context, uri);
        try {
        context.startActivity(intent);
        } catch (ActivityNotFoundException e) {
        Log.e(TAG, "There was an error opening the custom tab " + e);
        }
}
```

2.1.3 Grouping exceptions

Where exceptions execute the same code, they should be grouped in-order to increase readability and avoid code duplication. For example, where you may do this:

```
public void openCustomTab(Context context, @Nullable Uri uri) {
        Intent intent = buildIntent(context, uri);
        try {
        context.startActivity(intent);
        } catch (ActivityNotFoundException e) {
        Log.e(TAG, "There was an error opening the custom tab " + e);
        } catch (NullPointerException e) {
        Log.e(TAG, "There was an error opening the custom tab " + e);
        } catch (SomeOtherException e) {
                // Show some dialog
    }
}
```

You could do this:

```
public void openCustomTab(Context context, @Nullable Uri uri) {
        Intent intent = buildIntent(context, uri);
        try {
        context.startActivity(intent);
        } catch (ActivityNotFoundException e | NullPointerException e) {
        Log.e(TAG, "There was an error opening the custom tab " + e);
        } catch (SomeOtherException e) {
                // Show some dialog
    }
}
```

2.1.4 Using try-catch over throw exception

Using try-catch statements improves the readability of the code where the exception is taking place. This is because the error is handled where it occurs, making it easier to both debug or make a change to how the error is handled.

2.1.5 Never 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 InputStreamfor 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. - taken from the Android code style guidelines

2.1.6 Fully qualify imports

When declaring imports, use the full package declaration. For example:

Don’t do this:

```
import android.support.v7.widget.*;
```

Instead, do this 😃

```
import android.support.v7.widget.RecyclerView;
```

2.1.7 Don't keep unused imports

Sometimes removing code from a class can mean that some imports are no longer needed. If this is the case then the corresponding imports should be removed alongside the code.

2.2 Java Style Rules

2.2.1 Field definition and naming

All fields should be declared at the top of the file, following these rules:

- Private, non-static field names should not start with m. This is right:

  userSignedIn, userNameText, acceptButton

Not this:

```
mUserSignedIn, mUserNameText, mAcceptButton
```

- Private, static field names do not need to start with an s. This is right:

  someStaticField, userNameText

Not this:

```
sSomeStaticField, sUserNameText
```

- All other fields also start with a lower case letter.

  int numOfChildren; String username;

- Static final fields (known as constants) are ALL_CAPS_WITH_UNDERSCORES.

  private static final int PAGE_COUNT = 0;

Field names that do not reveal intention should not be used. For example,

```
int e; //number of elements in the list
```

why not just give the field a meaningful name in the first place, rather than leaving a comment!

```
int numberOfElements;
```

That's much better!

2.2.1.2 View Field Naming

When naming fields that reference views, the name of the view should be the last word in the name. For example:

View	Name
TextView	usernameView
Button	acceptLoginView
ImageView	profileAvatarView
RelativeLayout	profileLayout

We name views in this way so that we can easily identify what the field corresponds to. For example, having a field named **user** is extremely ambiguous - giving it the name usernameView, userAvatarView or userProfieLayout helps to make it clear exactly what view the field corresponds with.

Previously, the names for views often ended in the view type (e.g acceptLoginButton) but quite often views change and it's easy to forgot to go back to java classes and update variable names.

2.2.2 Avoid naming with container types

Leading on from the above, we should also avoid the use of container type names when creating variables for collections. For example, say we have an arraylist containing a list of userIds:

Do:

```
List<String> userIds = new ArrayList<>();
```

Don't:

```
List<String> userIdList = new ArrayList<>();
```

If and when container names change in the future, the naming of these can often get forgotten about - and just like view naming, it's not entirely necessary. Correct naming of the container itself should provide enough information for what it is.

2.2.3 Avoid similar naming

Naming variables, method and / or classes with similar names can make it confusing for other developers reading over your code. For example:

```
hasUserSelectedSingleProfilePreviously

hasUserSelectedSignedProfilePreviously
```

Distinguishing the difference between these at a first glance can be hard to understand what is what. Naming these in a clearer way can make it easier for developers to navigate the fields in your code.

2.2.4 Number series naming

When Android Studio auto-generates code for us, it's easy to leave things as they are - even when it generate horribly named parameters! For example, this isn't very nice:

```
public void doSomething(String s1, String s2, String s3)
```

It's hard to understand what these parameters do without reading the code. Instead:

```
public void doSomething(String userName, String userEmail, String userId)
```

That makes it much easier to understand! Now we'll be able to read the code following the parameter with a much clearer understanding ☺

2.2.5 Pronouncable names

When naming fields, methods and classes they should:

- Be readable: Efficient naming means we'll be able to look at the name and understand it instantly, reducing cognitive load on trying to decipher what the name means.
- Be speakable: Names that are speakable avoids awkward conversations where you're trying to pronounce a badly named variable name.
- Be searchable: Nothing is worse than trying to search for a method or variable in a class to realise it's been spelt wrong or badly named. If we're trying to find a method that searches for a user, then searching for 'search' should bring up a result for that method.
- Not use Hungarian notation: Hungarian notation goes against the three points made above, so it should never be used!

2.2.6 Treat acronyms as words

Any acronyms for class names, variable names etc should be treated as words - this applies for any capitalisation used for any of the letters. For example:

Do	Don't
setUserId	setUserID
String uri	String URI
int id	int ID
parseHtml	parseHTML
generateXmlFile	generateXMLFile

2.2.7 Avoid justifying variable declarations

Any declaration of variables should not use any special form of alignment, for example:

This is fine:

```
private int userId = 8;
private int count = 0;
private String username = "hitherejoe";
```

Avoid doing this:

```
private String username = "hitherejoe";
private int userId      = 8;
private int count       = 0;
```

This creates a stream of whitespace which is known to make text difficult to read for certain learning difficulties.

2.2.8 Use spaces for indentation

For blocks, 4 space indentation should be used:

```
if (userSignedIn) {
    count = 1;
}
```

Whereas for line wraps, 8 spaces should be used:

```
String userAboutText =
        "This is some text about the user and it is pretty long, can you
see!"
```

2.2.9 If-Statements

2.2.9.1 Use standard brace style

Braces should always be used on the same line as the code before them. For example, avoid doing this:

```
class SomeClass
{
        private void someFunction()
        {
        if (isSomething)
        {

        }
        else if (!isSomethingElse)
        {

        }
        else
        {

        }
        }
}
```

And instead, do this:

```
class SomeClass {
        private void someFunction() {
        if (isSomething) {

        } else if (!isSomethingElse) {

        } else {

        }
        }
}
```

Not only is the extra line for the space not really necessary, but it makes blocks easier to follow when reading the code.

2.2.9.2 Inline if-clauses

Sometimes it makes sense to use a single line for if statements. For example:

```
if (user == null) return false;
```

However, it only works for simple operations. Something like this would be better suited with braces:

```
if (user == null) throw new IllegalArgumentExeption("Oops, user object is
required.");
```

2.2.9.3 Nested if-conditions

Where possible, if-conditions should be combined to avoid over-complicated nesting. For example:

Do:

```
if (userSignedIn && userId != null) {

}
```

Try to avoid:

```
if (userSignedIn) {
    if (userId != null) {

    }
}
```

This makes statements easier to read and removes the unnecessary extra lines from the nested clauses.

2.2.9.4 Ternary Operators

Where appropriate, ternary operators can be used to simplify operations.

For example, this is easy to read:

```
userStatusImage = signedIn ? R.drawable.ic_tick : R.drawable.ic_cross;
```

and takes up far fewer lines of code than this:

```
if (signedIn) {
    userStatusImage = R.drawable.ic_tick;
} else {
    userStatusImage = R.drawable.ic_cross;
}
```

Note: There are some times when ternary operators should not be used. If the if-clause logic is complex or a large number of characters then a standard brace style should be used.

2.2.10 Annotations

2.2.10.1 Annotation practices

Taken from the Android code style guide:

@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 should always be used where possible. For example, using the @Nullable annotation should be used in cases where a field could be expected as null. For example:

```
@Nullable TextView userNameText;

private void getName(@Nullable String name) { }
```

2.2.10.2 Annotation style

Annotations that are applied to a method or class should always be defined in the declaration, with only one per line:

```
@Annotation
@AnotherAnnotation
public class SomeClass {

  @SomeAnotation
  public String getMeAString() {

  }

}
```

When using the annotations on fields, you should ensure that the annotation remains on the same line whilst there is room. For example:

```
@Bind(R.id.layout_coordinator) CoordinatorLayout coordinatorLayout;


@Inject MainPresenter mainPresenter;
```

We do this as it makes the statement easier to read. For example, the statement '@Inject SomeComponent mSomeName' reads as 'inject this component with this name'.

2.2.11 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. - taken from the Android code style guidelines

2.2.12 Unused elements

All unused **fields**, **imports**, **methods** and **classes** should be removed from the code base unless there is any specific reasoning behind keeping it there.

2.2.13 Order Import Statements

Because we use Android Studio, so imports should always be ordered automatically. However, in the case that they may not be, then they should be ordered as follows:

1. Android imports
2. Imports from third parties
3. java and javax imports
4. Imports from the current Project

Note:

- 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)

2.2.14 Logging

Logging should be used to log useful error messages and/or other information that may be useful during development.

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

We can set the `Tag` for the log as a `static final` field at the top of the class, for example:

```
private static final String TAG = MyActivity.class.getName();
```

All verbose and debug logs must be disabled on release builds. On the other hand - information, warning and error logs should only be kept enabled if deemed necessary.

```
if (BuildConfig.DEBUG) {
    Log.d(TAG, "Here's a log message");
}
```

Note: Timber is the preferred logging method to be used. It handles the tagging for us, which saves us keeping a reference to a TAG.

2.2.15 Field Ordering

Any fields declared at the top of a class file should be ordered in the following order:

1. Enums
2. Constants
3. Dagger Injected fields
4. Butterknife View Bindings
5. private global variables
6. public global variables

For example:

```
public static enum {
```

```
public static enum {
        ENUM_ONE, ENUM_TWO
}

public static final String KEY_NAME = "KEY_NAME";
public static final int COUNT_USER = 0;

@Inject SomeAdapter someAdapter;

@BindView(R.id.text_name) TextView nameText;
@BindView(R.id.image_photo) ImageView photoImage;

private int userCount;
private String errorMessage;

public int someCount;
public String someString;
```

Using this ordering convention helps to keep field declarations grouped, which increases both the locating of and readability of said fields.

2.2.16 Class member ordering

To improve code readability, it’s important to organise class members in a logical manner. The following order should be used to achieve this:

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

For example:

```
public class MainActivity extends Activity {

    private int count;

    public static newInstance() { }

    @Override
    public void onCreate() { }

    public setUsername() { }
    private void setupUsername() { }

    static class AnInnerClass { }

    interface SomeInterface { }

}
```

Any lifecycle methods used in Android framework classes should be ordered in the corresponding lifecycle order. For example:

```
public class MainActivity extends Activity {

    // Field and constructors

    @Override
    public void onCreate() { }

    @Override
    public void onStart() { }

    @Override
    public void onResume() { }

    @Override
    public void onPause() { }

    @Override
    public void onStop() { }

    @Override
    public void onRestart() { }

    @Override
    public void onDestroy() { }

    // public methods, private methods, inner classes and interfaces

}
```

2.2.17 Method parameter ordering

When defining methods, parameters should be ordered to the following convention:

```
public Post loadPost(Context context, int postId);
public void loadPost(Context context, int postId, Callback callback);
```

Context parameters always go first and **Callback** parameters always go last.

2.2.18 String constants, naming, and values

When using string constants, they should be declared as final static and use the follow conventions:

[Strings table]

2.2.19 Enums

Enums should only be used where actually required. If another method is possible, then that should be the preferred way of approaching the implementation. For example:

Instead of this:

```
public enum SomeEnum {
    ONE, TWO, THREE
}
```

Do this:

```
private static final int VALUE_ONE = 1;
private static final int VALUE_TWO = 2;
private static final int VALUE_THREE = 3;
```

2.2.20 Arguments in fragments and activities

When we pass data using an Intent or Bundle, the keys for the values must use the conventions defined below:

Activity

Passing data to an activity must be done using a reference to a KEY, as defined as below:

```
private static final String KEY_NAME =
"com.your.package.name.to.activity.KEY_NAME";
```

Fragment

Passing data to a fragment must be done using a reference to an EXTRA, as defined as below:

```
private static final String EXTRA_NAME = "EXTRA_NAME";
```

When creating new instances of a fragment or activity that involves passing data, we should provide a static method to retrieve the new instance, passing the data as method parameters. For example:

Activity

```
public static Intent getStartIntent(Context context, Post post) {
    Intent intent = new Intent(context, CurrentActivity.class);
    intent.putParcelableExtra(EXTRA_POST, post);
    return intent;
}
```

Fragment

```
public static PostFragment newInstance(Post post) {
    PostFragment fragment = new PostFragment();
    Bundle args = new Bundle();
    args.putParcelable(ARGUMENT_POST, post);
    fragment.setArguments(args)
    return fragment;
}
```

2.2.21 Line Length Limit

Code lines should exceed no longer than 100 characters, this makes the code more readable. Sometimes to achieve this, we may need to:

- Extract data to a local variable
- Extract logic to an external method
- Line-wrap code to separate a single line of code to multiple lines

Note: For code comments and import statements it’s ok to exceed the 100 character limit.

2.2.21.1 Line-wrapping techniques

When it comes to line-wraps, there's a few situations where we should be consistent in the way we format code.

Breaking at Operators

When we need to break a line at an operator, we break the line before the operator:

```
int count = countOne + countTwo - countThree + countFour * countFive - 
countSix
        + countOnANewLineBecauseItsTooLong;
```

If desirable, you can always break after the `=` sign:

```
int count =
        countOne + countTwo - countThree + countFour * countFive + 
countSix;
```

Method Chaining

When it comes to method chaining, each method call should be on a new line.

Don't do this:

```
Picasso.with(context).load("someUrl").into(imageView);
```

Instead, do this:

```
Picasso.with(context)
        .load("someUrl")
        .into(imageView);
```

Long Parameters

In the case that a method contains long parameters, we should line break where appropriate. For example when declaring a method we should break after the last comma of the parameter that fits:

```
private void someMethod(Context context, String someLongStringName, String 
text,
                            long thisIsALong, String anotherString) {
}
```

And when calling that method we should break after the comma of each parameter:

```
someMethod(context,
        "thisIsSomeLongTextItsQuiteLongIsntIt",
        "someText",
        01223892365463456,
        "thisIsSomeLongTextItsQuiteLongIsntIt");
```

2.2.22 Method spacing

There only needs to be a single line space between methods in a class, for example:

Do this:

```
public String getUserName() {
    // Code
}

public void setUserName(String name) {
    // Code
}

public boolean isUserSignedIn() {
    // Code
}
```

Not this:

```
public String getUserName() {
    // Code
}


public void setUserName(String name) {
    // Code
}


public boolean isUserSignedIn() {
    // Code
}
```

2.2.23 Comments

2.2.23.1 Inline comments

Where necessary, inline comments should be used to provide a meaningful description to the reader on what a specific piece of code does. They should only be used in situations where the code may be complex to understand. In most cases however, code should be written in a way that it easy to understand without comments 🙂

Note: Code comments do not have to, but should try to, stick to the 100 character rule.

2.2.23.2 JavaDoc Style Comments

Whilst a method name should usually be enough to communicate a methods functionality, it can sometimes help to provide JavaDoc style comments. This helps the reader to easily understand the methods functionality, as well as the purpose of any parameters that are being passed into the method.

```
/**
 * Authenticates the user against the API given a User id.
 * If successful, this returns a success result
 *
 * @param userId The user id of the user that is to be authenticated.
 */
```

2.2.23.3 Class comments

When creating class comments they should be meaningful and descriptive, using links where necessary. For example:

```
/**
  * RecyclerView adapter to display a list of {@link Post}.
  * Currently used with {@link PostRecycler} to show the list of Post
items.
  */
```

Don’t leave author comments, these aren’t useful and provide no real meaningful information when multiple people are to be working on the class.

```
/**
  * Created By Joe 18/06/2016
  */
```

2.2.24 Sectioning code

2.2.24.1 Java code

If creating 'sections' for code, this should be done using the following approach, like this:

```
public void method() { }

public void someOtherMethod() { }

/********* Mvp Method Implementations  ********/

public void anotherMethod() { }

/********* Helper Methods  ********/

public void someMethod() { }
```

Not like this:

```
public void method() { }

public void someOtherMethod() { }

// Mvp Method Implementations

public void anotherMethod() { }
```

This makes sectioned methods easier to located in a class.

2.2.24.2 Strings file

String resources defined within the string.xml file should be section by feature, for example:

```
// User Profile Activity
<string name="button_save">Save</string>
<string name="button_cancel">Cancel</string>

// Settings Activity
<string name="message_instructions">...</string>
```

Not only does this help keep the strings file tidy, but it makes it easier to find strings when they need altering.

2.2.24.3 RxJava chaining

When chaining Rx operations, every operator should be on a new line, breaking the line before the period `.`. For example:

```
return dataManager.getPost()
            .concatMap(new Func1<Post, Observable<? extends Post>>() {
                @Override
                 public Observable<? extends Post> call(Post post) {
                     return mRetrofitService.getPost(post.id);
                 }
            })
            .retry(new Func2<Integer, Throwable, Boolean>() {
                 @Override
                 public Boolean call(Integer numRetries, Throwable
throwable) {
                     return throwable instanceof RetrofitError;
                 }
            });
```

This makes it easier to understand the flow of operation within an Rx chain of calls.

2.2.25 Butterknife

2.2.25.1 Event listeners

Where possible, make use of Butterknife listener bindings. For example, when listening for a click event instead of doing this:

```
mSubmitButton.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        // Some code here...
    }
  };
```

Do this:

```
@OnClick(R.id.button_submit)
public void onSubmitButtonClick() { }
```

2.3 XML Style Rules

2.3.1 Use self=-closing tags

When a View in an XML layout does not have any child views, self-closing tags should be used.

Do:

```
<ImageView
    android:id="@+id/image_user"
    android:layout_width="90dp"
    android:layout_height="90dp" />
```

Don't:

```
<ImageView
    android:id="@+id/image_user"
    android:layout_width="90dp"
    android:layout_height="90dp">
</ImageView>
```

2.3.2 Resource naming

All resource names and IDs should be written using lowercase and underscores, for example:

```
text_username, activity_main, fragment_user,
error_message_network_connection
```

The main reason for this is consistency, it also makes it easier to search for views within layout files when it comes to altering the contents of the file.

2.3.2.1 ID naming

All IDs should be prefixed using the name of the element that they have been declared for.

Element	Prefix

Element	Prefix
ImageView	image_
Fragment	fragment_
RelativeLayout	layout_
Button	button_
TextView	text_
View	view_

For example:

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

Views that typically are only one per layout, such as a toolbar, can simply be given the id of it's view type. E.g. `toolbar` .

2.3.2.2 Strings

All string names should begin with a prefix for the part of the application that they are being referenced from. For example:

Screen	String	Resource Name
Registration Fragment	“Register now”	registration_register_now
Sign Up Activity	“Cancel”	sign_up_cancel
Rate App Dialog	“No thanks”	rate_app_no_thanks

If it’s not possible to name the referenced like the above, we can use the following rules:

Prefix	Description

Prefix	Description
error_	Used for error messages
title_	Used for dialog titles
action_	Used for option menu actions
msg_	Used for generic message such as in a dialog
label_	Used for activity labels

Two important things to note for String resources:

- String resources should never be reused across screens. This can cause issues when it comes to changing a string for a specific screen. It saves future complications by having a single string for each screens usage.
- String resources should **always** be defined in the strings file and never hardcoded in layout or class files.

2.3.2.3 Styles and themes

When defining both Styles & Themes, they should be named using UpperCamelCase. For example:

```
AppTheme.DarkBackground.NoActionBar
AppTheme.LightBackground.TransparentStatusBar

ProfileButtonStyle
TitleTextStyle
```

2.3.3 Attributes ordering

Ordering attributes not only looks tidy but it helps to make it quicker when looking for attributes within layout files. As a general rule,

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

For example:

```
<Button
    android:id="@id/button_accept"
    style="@style/ButtonStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_alignParentStart="true"
    android:padding="16dp"
    android:text="@string/button_skip_sign_in"
    android:textColor="@color/bluish_gray" />
```

Note: This formatting can be carried out by using the format feature in android studio -

`cmd + shift + L`

Doing this makes it easy to navigate through XML attributes when it comes to making changes to layout files.

2.4 Tests style rules

2.4.1 Unit tests

Any Unit Test classes should be written to match the name of the class that the test are targeting, followed by the Test suffix. For example:

Class	Test Class
DataManager	DataManagerTest
UserProfilePresenter	UserProfilePresenterTest
PreferencesHelper	PreferencesHelperTest

All Test methods should be annotated with the `@Test` annotation, the methods should be named using the following template:

```
@Test
public void methodNamePreconditionExpectedResult() { }
```

So for example, if we want to check that the signUp() method with an invalid email address fails, the test would look like:

```
@Test
public void signUpWithInvalidEmailFails() { }
```

Tests should focus on testing only what the method name entitles, if there's extra conditions being tested in your Test method then this should be moved to it's own individual test.

If a class we are testing contains many different methods, then the tests should be split across multiple test classes - this helps to keep the tests more maintainable and easier to locate. For example, a DatabaseHelper class may need to be split into multiple test classes such as :

```
DatabaseHelperUserTest
DatabaseHelperPostsTest
DatabaseHelperDraftsTest
```

2.4.2 Espresso tests

Each Espresso test class generally targets an Activity, so the name given to it should match that of the targeted Activity, again followed by Test. For example:

Class	Test Class
MainActivity	MainActivityTest
ProfileActivity	ProfileActivityTest
DraftsActivity	DraftsActivityTest

When using the Espresso API, methods should be chained on new lines to make the statements more readable, for example:

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

Chaining calls in this style not only helps us stick to less than 100 characters per line but it also makes it easy to read the chain of events taking place in espresso tests.

3. Gradle Style

3.1 Dependencies

3.1.1 Versioning

Where applicable, versioning that is shared across multiple dependencies should be defined as a variable within the dependencies scope. For example:

```
final SUPPORT_LIBRARY_VERSION = '23.4.0'

compile "com.android.support:support-v4:$SUPPORT_LIBRARY_VERSION"
compile "com.android.support:recyclerview-v7:$SUPPORT_LIBRARY_VERSION"
compile "com.android.support:support-annotations:$SUPPORT_LIBRARY_VERSION"
compile "com.android.support:design:$SUPPORT_LIBRARY_VERSION"
compile "com.android.support:percent:$SUPPORT_LIBRARY_VERSION"
compile "com.android.support:customtabs:$SUPPORT_LIBRARY_VERSION"
```

This makes it easy to update dependencies in the future as we only need to change the version number once for multiple dependencies.

3.1.2 Grouping

Where applicable, dependencies should be grouped by package name, with spaces in-between the groups. For example:

```
compile "com.android.support:percent:$SUPPORT_LIBRARY_VERSION"
compile "com.android.support:customtabs:$SUPPORT_LIBRARY_VERSION"

compile 'io.reactivex:rxandroid:1.2.0'
compile 'io.reactivex:rxjava:1.1.5'

compile 'com.jakewharton:butterknife:7.0.1'
compile 'com.jakewharton.timber:timber:4.1.2'



compile 'com.github.bumptech.glide:glide:3.7.0'
```

`compile` , `testCompile` and `androidTestCompile` dependencies should also be grouped into their corresponding section. For example:

```
// App Dependencies
compile "com.android.support:support-v4:$SUPPORT_LIBRARY_VERSION"
compile "com.android.support:recyclerview-v7:$SUPPORT_LIBRARY_VERSION"
```

```
// Instrumentation test dependencies
androidTestCompile "com.android.support:support-
annotations:$SUPPORT_LIBRARY_VERSION"

// Unit tests dependencies
testCompile 'org.robolectric:robolectric:3.0'
```

Both of these approaches makes it easy to locate specific dependencies when required as it keeps dependency declarations both clean and tidy 🙌

3.1.3 Independent Dependencies

Where dependencies are only used individually for application or test purposes, be sure to only compile them using `compile` , `testCompile` or `androidTestCompile` . For example, where the robolectric dependency is only required for unit tests, it should be added using:

```
testCompile 'org.robolectric:robolectric:3.0'
```