# Coding Guidelines | Android

1. **Java language rules**
   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)

See alternatives here.

* 1. **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

* 1. **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 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.* - (Android code style guidelines)

* 1. **Fully qualify imports**

This is bad: import foo.\*;

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

1. **Java style rules**
   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;

}

* 1. **Treat acronyms as words**

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

* 1. **Use spaces for indentation** Use **4 space** indents for blocks: if (x ==

1) {

x++;

}

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

Instrument i =

someLongExpression(that, wouldNotFit, on, one, line);

* 1. **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!

* 1. **Annotations**
     1. **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.

* + 1. **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;

* 1. **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)

* 1. **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

* 1. **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);

* 1. **Class member ordering**

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

1. 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() {}

}

* 1. **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);

* 1. **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";

* 1. **Arguments in Fragments and Activities**

When data is passed into an Activityor 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 Frag ment.

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 not need for them to be exposed outside the class.

* 1. **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.

* + 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.

**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);

**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");

**2.14 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;

}

});

}

1. **XML style rules**
   1. **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>

* 1. **Resources naming**

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

* + 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>

* + 1. **Strings**

String names start with a prefix that identifies the section they belong to. For example registration\_email\_hint or registration\_name\_hint. If a string **doesn't belong**to any section, then you should follow the rules below:

|  |  |
| --- | --- |
| **Prefix** | **Description** |
| error\_ | An error message |
| msg\_ | A regular information message |
| title\_ | A title, i.e. a dialog title |
| action\_ | An action such as "Save" or "Create" |

* + 1. **Styles and Themes**

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

* 1. **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

***Use TODO Comments***

Use TODO comments for code that is temporary, a short-term solution, or good-enough but not perfect. TODOs should include the string TODO in all caps, followed by a colon:

// TODO: Remove this code after the UrlTable2 has been checked in.

and

// TODO: Change this to use a flag instead of a constant.

***Some General Guidelines:***

Don't hard code any values inside the code. Move them to properties files on constant files. This makes it easy to manage and change. You don't need to put any comments on the code if the naming is proper. Write comments only in case you write a complex business logic.

Don't make unnecessary API/database calls. Use caching for less frequent changing data.

Follow the KISS principle. It means Keep It Simple Stupid. (<http://en.wikipedia.org/wiki/KISS_principle)>Don't write overly complicated code.

Don't add unnecessary permissions in the manifest file. This sometimes cares a user.

Have the key store added to project itself. This avoid different developers using different key stores. Password should be kept and shared separately.

Have proper versioning of different version. In case of minor fixes just update the minor version.

Don't put your name in Packages, Classes or Methods ( During code review I found some people have created packages with his/her name). Use names which relates to your application.

Create reusable UI components along with their action. Avoid copy pasting the code and creating replicas. Java Classes

com.example

activities - Contains all the activities. Classes are all named with Activity at the end. adapters – contains all the adapters

authenticator – all classes related to different authenticators data – data management (Content Providers, Sqllite helpers) fragments – All fragments

helpers - Contains helper classes. A helper class is a place to put code that is used in more than one place. Most of the methods are static. Example DateHelper

Interfaces – Contains all the interfaces models - Contains all local models.

You can have sub-package according to logical entities like different departments.

***Logging:***

Use slf4j logging framework. Write error/fatal logs in production mode while info logs for development. Write proper logs in code at different needed places.

User at-least 3 level of logging – INFO (general informational messages), ERROR (business error cases) & FATAL (System failures). Print full exception stack in mail and in mail.

We should have a logger class where we define which logs to come for debug mode . Only error logs should be enabled for release

binary.

***Error Handling:***

Do proper error handling. The beauty of java is you can catch everything. This helps in controlling error messages to users and avoiding crashes.

The thumb rule is: No screen should show a technical error. All errors should caught and presented with appropriate error messages. Make the API calls cancellable. This will avoid crashes in case use has moved to different activity before getting a response.

***Unit Testing:***

Use JUnit framework for unit testing.

Make sure that you cover 70-80% of code using test cases.

This will reduce bugs in QA and reduce the over all effort in project development.

**Android Best Practices**

***https://developer.android.com/distribute/essentials/quality/core.html***

***Avoid Memory Leaks:***

In Java, non-static anonymous classes hold an implicit reference to their enclosing class. If you're not careful, storing this reference can result in the Activity being retained when it would otherwise be eligible for garbage collection. Activity objects hold a reference to their entire view hierarchy and all its resources, so if you leak one, you leak a lot of memory.

**With Handler:**

**//This is Bad and leaks Memory**

public class SampleActivity extends Activity {

private final Handler mLeakyHandler = new Handler() { @Override

public void handleMessage(Message msg) {

// ...

}

}

}

#### //This is Good

public class SampleActivity extends Activity {

/\*\*

* Instances of static inner classes do not hold an implicit
* reference to their outer class.

\*/

private static class MyHandler extends Handler {

/\*\*

* Suppose that the garbage collector determines at a certain point in time that an object is weakly
* reachable. At that time it will atomically clear all weak references to that object. Also, a weak
* reference object is automatically made 'finalizable' and a finalizable object can eventually get
* its finalizer automatically invoked by the JVM.

\*/

private final WeakReference<SampleActivity> mActivity;

public MyHandler(SampleActivity activity) {

mActivity = new WeakReference<SampleActivity>(activity);

}

@Override

public void handleMessage(Message msg) { SampleActivity activity = mActivity.get(); if (activity != null) {

// ...

}

}

}

With Threads:

***This is Bad:***

/\*\*

* + Example illustrating how threads persist across configuration
  + changes (which cause the underlying Activity instance to be
  + destroyed). The Activity context also leaks because the thread
  + is instantiated as an anonymous class, which holds an implicit
  + reference to the outer Activity instance, therefore preventing
  + it from being garbage collected.

\*/

public class MainActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState)

{ super.onCreate(savedInstanceState); exampleOne();

}

private void exampleOne() { new Thread() {

@Override

public void run() { while (true) {

SystemClock.sleep(1000);

}

}

}.start();

}

}

**This is Good:**/\*\*

* + This example avoids leaking an Activity context by declaring the
  + thread as a private static inner class, but the threads still
  + continue to run even across configuration changes. The DVM has a
  + reference to all running threads and whether or not these threads
  + are garbage collected has nothing to do with the Activity lifecycle.
  + Active threads will continue to run until the kernel destroys your
  + application's process.

\*/

public class MainActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState);

exampleTwo();

}

private void exampleTwo()

{

new MyThread().start();

}

private static class MyThread extends Thread { @Override

public void run() { while (true) {

SystemClock.sleep(1000);

}

}

}

}

**Build:**

***Gradle:***

in Build.gradle pls make sure that repositories is included with mavenCenteral(). exp:

repositories { mavenCentral()

}

Also its good practice to define Min and target sdk version in grade defaultConfig {

applicationId “com.example.abc"

minSdkVersion 16

targetSdkVersion 21

versionCode 1

versionName "1.0"

}

**Android Specific Naming Convention**

**1. Class files**

Class names are written in UpperCamelCase.

For classes that extend an Android component, the name of the class should end with the name of the component; for example: SignInActivity, Si gnInFragment, ImageUploaderService, ChangePasswordDialog.**2. Resources files**

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

* 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):

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

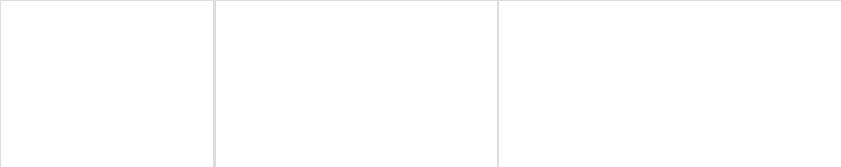
Selected \_selected btn\_order\_selected.9.png

* 1. **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\_.

* 1. **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.

* 1. **Values files**

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

1. **Resources naming**

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

* 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>

**Passing objects by Intent: Serializable vs Parcelable**

Serializable and Parcelable both are used to pass object by intent in Android application. Parcelable and Serialization are used for marshalingand unmarshaling Java objects.

In Parcelable, developers write custom code for marshaling and unmarshaling so it creates less garbage objects in comparison to Serialization. The performance of Parcelable over Serialization dramatically improves (around two times faster), because of this custom implementation.

Serialization is a marker interface, which implies the user cannot marshal the data according to their requirements. In Serialization, a marshaling operation is performed on a Java Virtual Machine (JVM) using the Java reflection API. This helps identify the Java objects member and behavior, but also ends up creating a lot of garbage objects. Due to this, the Serialization process is slow in comparison to Parcelable.

So Use **Parcelable t**o send objects by intents, also by other means like Handlers.