|  |
| --- |
|  |
| Android Coding Guidelines |
|  |
|  |
| **Version 1.5**  **05** |
|  |

|  |
| --- |
|  |

**Revision history**

Contents

[1 Introduction 3](#_Toc502337418)

[2 Overview 3](#_Toc502337419)

[3 Naming Conventions 4](#_Toc502337420)

[4 Coding Style 5](#_Toc502337421)

[5 Code Documentation 7](#_Toc502337422)

[6 Maintainability 7](#_Toc502337423)

[7 Robustness 7](#_Toc502337424)

[8 Third party Libraries 8](#_Toc502337425)

[9 Performance 9](#_Toc502337426)

[10 Code organization guidelines 10](#_Toc502337427)

[1. For code organization, we follow the Appitecture as our base structure, more on Appitecture could be found in the Appitecture ppt. 10](#_Toc502337428)

[2. ListView adapters should be kept in the same activity(class) unless adapter logic is too complex and big in volume(More than 300 lines) 10](#_Toc502337429)

[11 UI Guidelines 11](#_Toc502337430)

[12 Prefered Design Patterns 12](#_Toc502337431)

[13 References 20](#_Toc502337432)

# Introduction

The document is intended for detailing the coding guidelines for applications developed on the Android platform, and consists of a number of recommendations for well written Java code.

The rest of this document has the following contents:

* Overview
* Code Sample
* Naming Conventions
* Code Disposition
* Coding Style
* Code Documentation
* Maintainability
* Robustness
* Performance
* Code organization guidelines
* References

# Overview

Here are some other important rules of sound software design are:

* ***Minimize calls***. There should be as few calls as possible and that the amount of data sent in each call is as small as possible. In a tradeoff between "chunky" (few large calls) and "chatty" (many small calls), "chunky" is generally the preference.
* ***Minimize resources needed***. Resources used should be released as soon as possible and held for as short time as possible.
* ***Minimize code duplication***. All general functionality is placed in separate general components or abstract base classes.
* ***Modularity***. Functionality is split into logical tiers and layers and grouped in classes to simplify change management, development of new functionality and integration.

Following these rules will assure performance, scalability and stability of the system.

# Naming Conventions

Following is the recommended naming conventions:

1. No prefixes (like Hungarian notation or underscores) anywhere!
2. Use English in all names.
3. Use lower case ASCII characters for package naming.
4. Class names should be nouns, in mixed case with the first letter of each internal word capitalized.
5. Keep your class names simple and descriptive. Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).
6. Interface names should be capitalized like class names.
7. Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.
8. Use camelCasing for everything private, like parameters, variables, and methods.
9. Common exceptions: int i, int j, String s, Exception ex, Database db, etc.
10. Name classes and properties using <noun> or <noun><type> pattern where <type> is inherited type (e.g. Customer, SampleForm, SampleException, SampleEventArgs, Color, BackColor).
11. Do not include class name within property name (e.g. name c~~ustomerName~~).
12. Name methods using <verb>, <verb><noun>, or <verb><object> pattern (e.g. initialize(), getCustomer()). If any return value is not obvious, it should also be reflected in method name (e.g. getInvoiceStatus()).
13. Use fully descriptive names and avoid abbreviations and acronyms (CustomerPriority ~~CustPrio~~). Avoid common names and reserved words (e.g. ~~Java~~, ~~for~~, ~~while~~).
14. Avoid fully qualified names, use the import statement instead.
15. Variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores (“\_”). Try to keep the all constant variables in one place. Constant should be final.
16. Naming for resource layout files (xml files) should be in small letters, words separated by underscores (“\_”).
17. Naming for id in views should follow camelCasing similar to methods.
18. Use short and descriptive names for the values defined under /values folder.

e.g. string defined in the strings.xml should have small and descriptive name.

<string name=*"enter\_email\_add\_to\_reset"*>To reset your password, please type the email address you use to sign in.</string>

Bad e.g

<string name=*"to\_reset\_your\_password\_please\_type\_the\_email\_address\_you\_use\_to\_sign\_in\_ "*>To reset your password, please type the email address you use to sign in.</string>

1. Package name should be like,

com.<Customer/company name>.<appname/projectname>.android

e.g.

com.sandvik.startvalue.android.

1. For other details regarding comments, indentation, declaration in the java coding guidelines,

<http://www.oracle.com/technetwork/java/codeconv-138413.html>

# Coding Style

Following is the recommended coding style.

1. Always state the scope (private, public, etc) of classes, methods and class variables.
2. With the exception of zero and one, never hard-code a numeric value; always declare a constant instead.
3. Always use zero-based arrays and with indexed collection, use zero-based indices.
4. All the string visible to end-user should be placed in res/strings.xml for ease of localization.
5. Use StringBuilder for building long strings.
6. Use TextUtils.isEmpty(String str) for checking empty strings as it verifies the null condition as well.
7. Always use a default case in a switch statement that asserts.
8. Prefer generic flavor before non-generic if both implementations are available.
9. Use application logging and debugging.
10. Remove logs from the code while releasing build for google play.
11. Don’t make any instance or class variable public without good reason.
12. Avoid using an object to access a class (static) variable or method. Use a class name instead.
13. For call backs from the network layer to UI layer use interfaces or handler (ANDROID specific).
14. Colors used within the project must be in color.xml file of /res/values.
15. Menu (commands) for each activity must be in menu.xml file of /res/values.
16. Avoid using splash screen for applications as separate launcher activities; use it along with the first screen to be displayed as a separate view.
17. Whenever you use layout\_margins please check the effects of orientation change (Landscape -> Potrait and Potrait -> Landscape) for the same.
18. Any control in the screen should have equal margin from all the sides.
19. While creating databases make sure to use less no. of columns or merge few of them in to a single column. Consider using binary operators like & for the same. More on binary operators on the below link <http://www.mssqltips.com/tip.asp?tip=1218>.
20. In case using databases always try using managedQuery for firing a query to the DB as the entire query maintenance is handled by the Android system itself i.e. closing and deactivating the cursor when the Activity class is destroyed. The managedQuery is available only for the Activity class. In case simple query is used on a cursor object make sure the cursor object is deactivated, closed and set to null in order to avoid memory leaks.
21. All the blocking calls must be done on a separate thread and not on the UI thread, so that it doesn’t block the UI.
22. Always use dip (Density independent pixels) in layout xml files for defining height and width of the component.
23. Use sp for defining height of the text used.
24. Use px to define width/height whenever exact size component required in pixels.
25. Understand the difference between “GONE” and “INVISIBLE” for a view.
26. Manipulating any UI operation from the network thread must be done in a UI thread only example updating a listview once the data is fetched from the network. The example for the same are as below:
    1. Activity.runOnUiThread(Runnable)
    2. View.post(Runnable)
    3. View.postDelayed(Runnable)
    4. Handler
    5. AsyncTask
27. Always use layout files to create user interface unless and until there is specific requirement.
28. Temporary files created must be cleared at interval or every app termination.
29. Progress dialog should behave same in the entire app.
30. Should handle connection related issues like, no internet connection, server connection time out.
31. While making network connection, explicitly mention request type(POST ,GET)
32. Encode all the data which is sent with the request with URL encoding.
33. Direct network calls must be avoided from UI(Activity) classes. Responsibility for network related calls and the response handling must be delegated to the Manager (Controller) classes.
34. Create generic data models for data received from the network.
35. Parsing of the data should be done in entity.
36. Use tools like MAT to analyze the memory.
37. View the below link for Android specific coding style.

<http://source.android.com/source/code-style.html>

1. For auto code style please use following profile. Import them in to eclipse. After importing these three profile all you have to do is press CTRL + SHIFT + F, for auto formatting.

To ignore formatting in some code you can use ignore formatting tag in the code.

Style profile has been uploaded to the,

<https://www.assembla.com/code/sogeti-nl-empty-repos/git/nodes>

# Code Documentation

Following is the recommended use of comments and other code documentation.

1. Avoid comments that state the obvious. Code should be self-explanatory. Good code with readable variable and method names should not require comments.
2. Document only operational assumptions, algorithm insights and so on.
3. Separate comments from the leading slashes with a single blank and only end with period if comment includes more than one sentence.
4. Include Task-List keyword flags to enable comment-filtering (e.g. // TODO:, // HACK:, etc).

# Maintainability

Following is the recommended guidelines to achieve maintainability of the code.

1. Avoid duplicated code but explore opportunities for reuse (e.g. consider implementing classes that encapsulate the logic, or creating helper classes).
2. Keep methods and files as short as possible (in general methods should be less than 350 lines and files less than 750 lines).
3. Do everything you can to encourage high [cohesion](http://en.wikipedia.org/wiki/Cohesion_%28computer_science%29) in all classes (i.e. avoid any form of coupling).
4. Avoid long parameter lists (in general not than 5 parameters - use structures or similar types instead).
5. Avoid use of "control flags" to drive the internal behavior of a method, explore opportunities for specialized classes or overloaded methods.
6. Avoid the use of global variables to drive the behavior of several classes.
7. Practice encapsulation, like information hiding, and avoid inappropriate intimacy between classes.
8. Replace magic numbers and literals with constants that have meaningful names (exception; zero and one).
9. Use <include/> and <merge/> tags in case the same xml layout is used in multiple layouts. The common example of the same would be title in all the screens.
10. All your drawables ( images ) must be in /res/drawable folder only. In case the build is suppose to support different device screens use /res/drawable-hdpi, /res/drawable-mdpi and /res/drawable-ldpi. Please refer the below link for more details on the same <http://developer.android.com/guide/practices/screens_support.html>.

# Robustness

Following is the recommended guidelines to achieve robustness of the code.

1. Perform early parameter checking (e.g. boundary checks, type checks, assertions, etc.) before executing main body of logic.
2. Checks return values received from service or method calls.
3. Always check for nulls when appropriate.
4. Avoids "Apocalypse Ready" designs (that handle exceptions that will probably never happen).
5. Catch exceptions and handle them at the top of the call-stack (lower in the call stack, exceptions are only caught to log or gather information, add information to the exception, perform cleanup, or attempt to recover).
6. Prefer the use of standard framework-defined exceptions when possible.
7. Throw exceptions only for clearly abnormal cases, and do not use exceptions to control application flow.

# Third party Libraries

Please choose carefully as per application needs and requirements.

1. **v7 Support library**: There are several libraries designed to be used with Android 2.1 (API level 7) and higher. These libraries provide specific feature sets and can be included in your application independently from each other.

* **v7 appcompat library**: This library adds support for the [Action Bar](http://developer.android.com/guide/topics/ui/actionbar.html) user interface [design pattern](http://developer.android.com/design/patterns/actionbar.html). This library includes support for [material design](http://developer.android.com/design/material/) user interface implementations.
* **v7 cardview library**: This library adds support for the [CardView](http://developer.android.com/reference/android/support/v7/widget/CardView.html) widget, which lets you show information inside cards that have a consistent look on any app.
* **v7 recyclerview library**: The recyclerview library adds the [RecyclerView](http://developer.android.com/reference/android/support/v7/widget/RecyclerView.html) class. This class provides support for the [RecyclerView widget](http://developer.android.com/training/material/lists-cards.jd#RecyclerView), a view for efficiently displaying large data sets by providing a limited window of data items.

Detailed information can be found at:

<http://developer.android.com/tools/support-library/features.html>

1. **V4 library**: This library is provided by the Google to use some of the API that is not present in the older version of android. Detailed information can be found at,

<http://developer.android.com/tools/extras/support-library.html>

1. **GSON library**: Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Detailed information can be found at:

<https://code.google.com/p/google-gson/>

1. **Calligraphy library:** Custom fonts can be used easily with this library, for detailed information please visit below links:

<https://github.com/chrisjenx/Calligraphy>

<http://blog.goyello.com/2014/08/01/how-to-use-custom-fonts-in-android-apps-and-not-get-fat-3/>

1. **Volley library:** Volley is an HTTP library that makes networking for Android apps easier and most importantly, faster. Volley is available through the open [AOSP](https://android.googlesource.com/platform/frameworks/volley) repository. For detailed information please visit below links:

<http://developer.android.com/training/volley/index.html>

1. **OkHttp library:** OkHttp is an HTTP client that’s efficient and allows support for connection pooling which reduces request latency. Detailed information with example can be found at:

<http://square.github.io/okhttp/>

1. **Retrofit library:** Retrofit turns your REST API into a Java interface. Detailed information with example can be found at: <http://square.github.io/retrofit/>
2. **Picasso library:** A powerful image downloading and caching library for android. Detailed information with example can be found at: <http://square.github.io/picasso/>
3. **ActionBarSherlock** : This is an extension of the [compatibility library](http://developer.android.com/sdk/compatibility-library.html) designed to facilitate the use of the action bar design pattern across all versions of Android with a single API. Detailed information with example can be found at:

<http://actionbarsherlock.com/>

1. **MergeAdapter:** MergeAdapter accepts a mix of Adapters and Views and presents them as one contiguous whole to whatever ListView it is poured into. This is good for cases where you have multiple data sources, or if you have a handful of ordinary Views to mix in with lists of data, or the like.

<https://github.com/commonsguy/cwac-merge>

1. **Jackson:** An SAX based parser for JSON objects.

**Download:** <http://jackson.codehaus.org/1.9.9/jackson-all-1.9.9.jar>

**More Info:** <https://github.com/FasterXML/jackson-core>

# Performance

Following is the recommended guidelines to achieve performance of the code.

1. Exit loops as soon as conditions are met.
2. Avoid using static variable as much as possible.
3. Do not re-evaluate expressions from within the loop controller statements.
4. Logic that always gets same results does not occur within the body of loops.
5. Avoid [premature optimization](http://en.wikipedia.org/wiki/Optimization_(computer_science)).
6. Use RelativeLayout instead of LinearLayout as its less UI expensive for Android. In case a LinearLayout has to be used think of a rationale for the same.
7. User view-holder pattern in case of ListViews.
8. Recycle bitmaps once not in use, to avoid out of memory exception.
9. After creating a xml file use “Hierarchy Viewer” to optimize the same. More on the same at the below link: <http://developer.android.com/guide/developing/tools/hierarchy-viewer.html>.
10. Android best practices could be found at the below link:

<http://developer.android.com/guide/practices/index.html>

1. Videos for improving performance of an Android system below:
   1. <http://www.google.com/events/io/2010/sessions/android-ui-design-patterns.html>.
   2. <http://www.google.com/events/io/2010/sessions/writing-zippy-android-apps.html>.
   3. <http://www.google.com/events/io/2010/sessions/world-of-listview-android.html>.
   4. <http://www.google.com/events/io/2010/sessions/casting-wide-net-android-devices.html>
   5. <http://www.google.com/events/io/2011/sessions/memory-management-for-android-apps.html>

# Code organization guidelines

For Code Organization we follow below guidelines:

## For code organization, we follow the Appitecture as our base structure, more on Appitecture could be found in the Appitecture ppt.

## ListView adapters should be kept in the same activity(class) unless adapter logic is too complex and big in volume(More than 300 lines)

# UI Guidelines

**Material Design:** Material design is a comprehensive guide for visual, motion, and interaction design across platforms and devices. Android now includes support for material design apps. To use material design in your Android apps, follow the guidelines defined in the [material design specification](http://www.google.com/design/spec) and use the new components and functionality available in Android 5.0 (API level 21) and above. <http://developer.android.com/design/material/index.html>

Please read and follow Android UI Design guidelines available at link below

<http://developer.android.com/design/index.html>

For icon design please check the link below:

<http://developer.android.com/guide/practices/ui_guidelines/icon_design.html>

For Android icons templates pack use the link below:

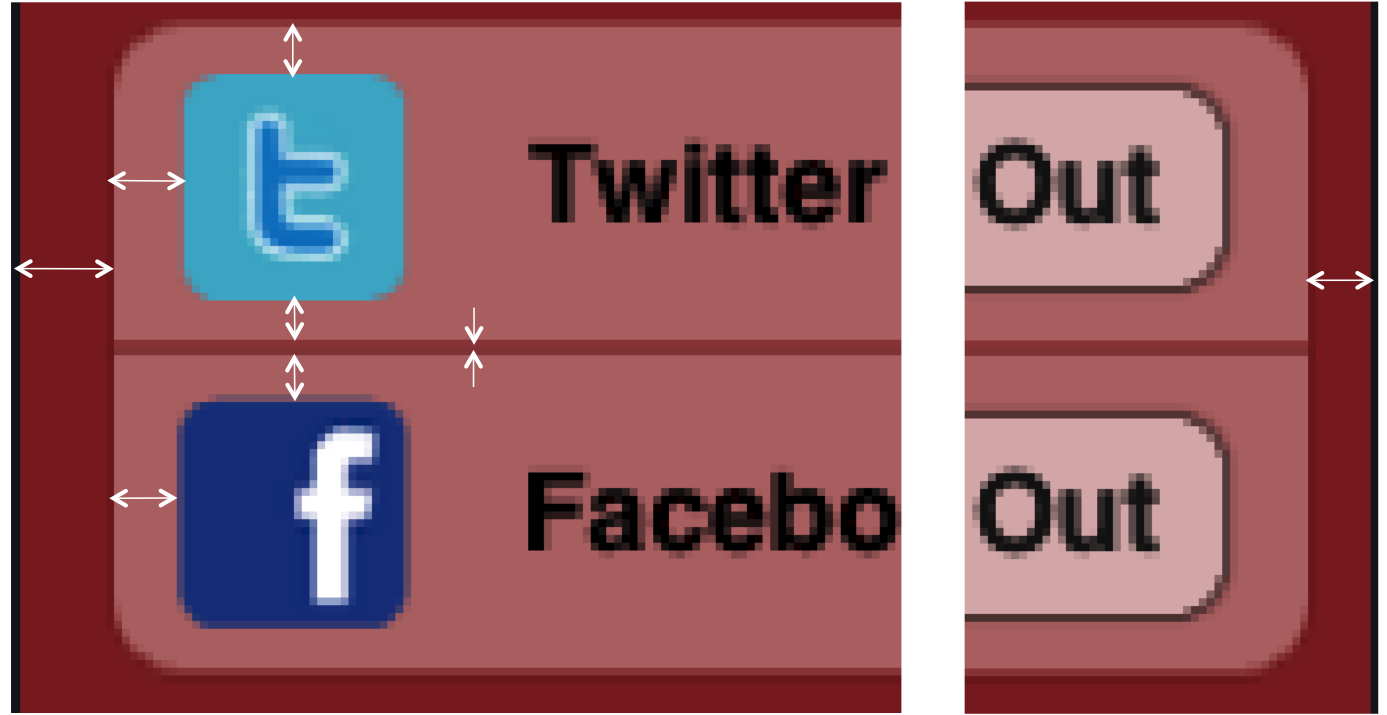
<http://developer.android.com/shareables/icon_templates-v4.0.zip>

For widget design guidelines please use the link below:

<http://developer.android.com/guide/practices/ui_guidelines/widget_design.html>

Please take care of following guidelines after developing/designing screens.

* Take screen shot of screens and zoom those to 500% and then review following.
* Accurate left/right alignment of text/elements falling one after another vertically.



* Even vertical spaces between two controls/labels
* Right foreground color and background colors. Prefer to set colors using RGB values supplied by graphics designer. Optionally you can use mspaint of Windows or apps like Fireworks/Photoshop to know RGB values of desired colors. Prefer to use hex values. And define them in color.xml



* Text font, font size and colors must be consistent all over the screen and app. The headings, titles, sub headings, paragraph text, icon text etc must be consistent all over the app.
* If wait indicators are blocking the UI then developer must hide the indicator when user taps on it or when user presses the back button. When user taps on wait indicator, one should also cancel the network request. Purpose is to avoid a state where app is unusable and user has to kill the app, this is bad user experience.

# Prefered Design Patterns

* **Factory Design Pattern**

1. When a class does not know which class of objects it must create.
2. A class specifies its sub-classes to specify which objects to create.
3. In programmer’s language (very raw form), you can use factory pattern where you have to create an object of any one of sub-classes depending on the data provided.

Example for the same below:

Let’s suppose an application asks for entering the name and sex of a person. If the sex is Male (M), it displays welcome message saying Hello Mr. <Name> and if the sex is Female (F), it displays message saying Hello Ms <Name>.

The skeleton of the code can be given here.

|  |  |
| --- | --- |
| public class Person { | |
|  | // name string public String name; // gender : M or F private String gender;  public String getName() { return name; }  public String getGender() { return gender; } |
| }// End of class | |

This is a simple class Person having methods for name and gender. Now, we will have two sub-classes, Male and Female which will print the welcome message on the screen.

|  |  |
| --- | --- |
| public class Male extends Person { | |
|  | public Male(String fullName) { System.out.println("Hello Mr. "+fullName); } |
| }// End of class | |

Also, the class Female

|  |  |
| --- | --- |
| public class Female extends Person { | |
|  | public Female(String fullNname) { System.out.println("Hello Ms. "+fullNname); } |
| }// End of class | |
|  | |
| Now, we have to create a client, or a SalutationFactory which will return the welcome message depending on the data provided.   |  |  | | --- | --- | | public class SalutationFactory { | | |  | public static void main(String args[]) { SalutationFactory factory = new SalutationFactory(); factory.getPerson(args[0], args[1]); }  public Person getPerson(String name, String gender) { if (gender.equals("M")) return new Male(name); else if(gender.equals("F")) return new Female(name); else return null; } | | }// End of class | |   This class accepts two arguments from the system at runtime and prints the names.  Running the program:  After compiling and running the code on my computer with the arguments Prashant and M:  java Jayshil D  The result returned is: “Hello Mr. Jayshil”. | |

* **Observer Design Pattern** (Loosely Coupled)

1. The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
2. The objects which are watching the state changes are called observer. Alternatively observer are also called listener.
3. The object which is being watched is called subject.

Example for the same below:

There is class of Persons, if the first name or last name of a person changes than the Observer in this case MyObserver will be notified of the changes, and can make appropriate actions.

package mydomain;

import java.beans.PropertyChangeListener;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class MyModel {

private List<Person> persons = new ArrayList<Person>();

private List<PropertyChangeListener> listener = new ArrayList<PropertyChangeListener>();

public class Person {

private String firstName;

private String lastName;

public Person(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

notifyListeners();

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

notifyListeners();

}

}

public List<Person> getPersons() {

return persons;

}

public MyModel() {

*// Just for testing we hard-code the persons here:*

persons.add(new Person("Lars", "Vogel"));

persons.add(new Person("Jim", "Knopf"));

}

private void notifyListeners() {

for (Iterator iterator = listener.iterator(); iterator.hasNext();) {

PropertyChangeListener name = (PropertyChangeListener) iterator

.next();

name.propertyChange(null);

}

}

public void addChangeListener(PropertyChangeListener newListener) {

listener.add(newListener);

}

}

package myobserver;

import java.beans.PropertyChangeEvent;

import java.beans.PropertyChangeListener;

import mydomain.MyModel;

public class MyObserver implements PropertyChangeListener {

public MyObserver(MyModel model) {

model.addChangeListener(this);

}

*@Override*

public void propertyChange(PropertyChangeEvent arg0) {

System.out.println("Things are changing...");

}

}

package main;

import java.util.Iterator;

import mydomain.MyModel;

import mydomain.MyModel.Person;

import myobserver.MyObserver;

public class Main {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

MyModel model = new MyModel();

MyObserver observer = new MyObserver(model);

// We change the last name of the person, observer will get notified

for (Iterator iterator = model.getPersons().iterator(); iterator

.hasNext();) {

Person person = (Person) iterator.next();

person.setLastName(person.getLastName() + "new");

}

// We change the first name of the person, observer will get notified

for (Iterator iterator = model.getPersons().iterator(); iterator

.hasNext();) {

Person person = (Person) iterator.next();

person.setFirstName(person.getFirstName() + "new");

}

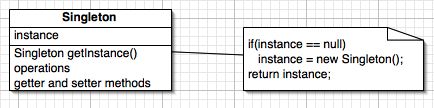
}

}

Note: The above class is a sample and doesnt follow the code organisation we follow, its purely taken from the prespective of Observer Pattern. The observer pattern allow for Open Closed Principle. The subject can registered an unlimited number of observers. If a new listener should register at the subject no code change in the subject is necessary. The subject and the observer are loosely coupled. If a controller is used then neither the subject nor the observer has direct knowledge about each other. If no controller is used then only the observer has knowledge about the subject. It’s a

* **Singleton Design Pattern**

1. Ensure that only one instance of a class is created
2. Provide a global point of access to the object
3. Allow multiple instances in the future without affecting a singleton class's clients



For safety reasons implement the cloneable interface and throw CloneNotSupportException so that clonning can not be done for you singleton class.

If the implementation also requires to be thread safe, synchronized in the getInstance() method of the class

**public** **class** MySingleton **implements** Cloneable {

**private** **static** MySingleton *\_instance*;

**private** MySingleton() {

}

**public** **static** **synchronized** MySingleton getInstance() {

**if** (*\_instance* == **null**) {

*\_instance* = **new** MySingleton();

}

**return** *\_instance*;

}

**protected** Object clone() **throws** CloneNotSupportedException {

**throw** **new** CloneNotSupportedException("A Singleton Class doesn't support Clone of Object, use getInstance() for Object");

}

}

* **Single Base Activity Pattern**

It is generally used, when there is a lot of common code which is being used in multiple activities.

Eg, progressBar/progressDialog, menu, menuEvents, inactivityModel. The example below shows whether the app is minimized or not and if its being inactive for more than 20 mins.

**public** **class** BaseActivity **extends** Activity

{

// To check if application is minimized

**public** **static** **boolean** *isAppMinimized*;

**public** **static** **boolean** *shiftToHomeScreen* = **false**;

**public** **static** **boolean** *isDirty* = **true**;

**public** **static** **boolean** *isHomeScreenDisplayed* = **true**;

**private** **static** Handler *handler* = **new** Handler();

@Override

**protected** **void** onResume() {

**super**.onResume();

i*sAppMinimized* = **false**;

**if**( *shiftToHomeScreen* )

{

*shiftToHomeScreen* = **false**;

*startHomeScreen*();

}

}

@Override

**protected** **void** onPause() {

**super**.onPause();

*isAppMinimized* = **true**;

}

/\*\*

\* Logic for log out user if inactive for 20 min or more

\*/

**private** **static** **void** startHomeScreen()

{

// Code to start home screen

}

**private** **static** Runnable *runnable* = **new** Runnable()

{

@Override

**public** **void** run()

{

// Application Stale for more than 15 minutes

**if**( !*isDirty* )

{

**if**( !*isAppMinimized* )

{

*startHomeScreen*();

} **else** {

*shiftToHomeScreen* = **true**;

}

}

}

};

@Override

**public** **void** onUserInteraction()

{

**super**.onUserInteraction();

*baseActivity* = **this**;

startInactiveUserTimer();

}

**public** **void** startInactiveUserTimer()

{

*isDirty* = **true**;

*handler*.removeCallbacks( *runnable* );

**if**( !*isHomeScreenDisplayed* ) {

*handler*.postDelayed( *runnable*, howOften);

}

*isDirty* = **false**;

}

}

* **Continous timer based pattern**

In some cases we need timer which continously runs in the background and updates the listener after a specific time interval such as shopping bag, it is used to display remaining time for a item to expire.

Its not in the best interest to use timers for individual items, instead of that a single thread which would notify all the listeners after a specific time interval.

Note: Its an extension of the observer pattern

**private** Handler repeatHandler = **new** Handler();

**private** Runnable repeatRunnable = **new** Runnable() {

@Override

**public** **void** run() {

// Logic to be performed on the repeat event, like updating individual shopping bag items to decrement time

}

};

repeatHandler.postDelayed(repeatRunnable, howOften);

* **Fragment Implementation:**

A fragment is a self-contained, modular section of an application’s user interface and corresponding behavior that can be embedded within an activity. Fragments can be assembled to create an activity during the application design phase, and added to, or removed from an activity during application runtime to create a dynamically changing user interface. Fragments may only be used as part of an activity and cannot be instantiated as standalone application elements. That being said, however, a fragment can be thought of as a functional “sub-activity” with its own lifecycle similar to that of a full activity.

* **Passing arguments to fragment:**

Create a static newInstance() method with the data you want to pass to the fragment as a parameter and call this method to create a new instance of fragment as shown in below code snippet.

public static class DetailsFragment extends Fragment {  
    /\*\*  
     \* Create a new instance of DetailsFragment, initialized to  
     \* show the text at 'index'.  
     \*/  
    public static DetailsFragment newInstance(int index) {  
        DetailsFragment f = new DetailsFragment();  
  
        // Supply index input as an argument.  
        Bundle args = new Bundle();  
        args.putInt("index", index);  
        f.setArguments(args);  
  
        return f;  
    } ...

* **Adding fragment to backstack:**

Below code will push the CountingFragment on the backstack and pressing back will pop it return the user to whatever previous state the activity UI was in.

// Instantiate a new fragment.  
 Fragment newFragment = CountingFragment.newInstance(mStackLevel);  
  
    // Add the fragment to the activity, pushing this transaction  
    // on to the back stack.  
    FragmentTransaction ft = getFragmentManager().beginTransaction();  
    ft.replace(R.id.simple\_fragment, newFragment);  
    ft.setTransition(FragmentTransaction.TRANSIT\_FRAGMENT\_OPEN);  
    ft.addToBackStack(null);  
    ft.commit();

* **Communicating with other fragments:**

We want one Fragment to communicate with another, for example to change the content based on a user event. All Fragment-to-Fragment communication is done through the associated Activity. Two Fragments should never communicate directly. Detailed information with example can be found at: <http://developer.android.com/training/basics/fragments/communicating.html#Deliver>

# References

Android specifics: <http://developer.android.com/index.html>

Observer Design Pattern: <http://www.vogella.com/articles/DesignPatternObserver/article.html>

Factory Design Pattern: <http://www.allapplabs.com/java_design_patterns/factory_pattern.htm>

Singleton Design Pattern: <http://www.tekhnologia.com/2012/05/what-is-singleton-design-pattern.html>

Material Design: <http://www.google.com/design/spec/material-design/introduction.html>