

# Android - Material Design

## Intro and Support library

Material Design is The new design language that gives design guidelines for Android apps, it was introduced with the release of Android 5.0.

This design pattern introduced new components. For ensuring backward compatibility, Google introduced the Android Design Support Library which brings a number of important material design components to developers with backwards compatible (We can use the design elements even from Android 2.1) The library includes a navigation drawer view, floating labels for editing text, a floating action button, snackbar, tabs and a motion and scroll framework to tie them together.

In this example we will create an app compatible with the material design guidelines

To include the support library we need to add the following dependencies to the *build.gradle* (Module:app) file under the **dependencies**

compile 'com.android.support:design:23.0.0' (or any other build version you have installed)

## Color Palette

The material design guide also provides app themes that can be customized. We will customize the app's [Color Palette](#) by setting the primary, primary-dark and accent colors as specified in the material design guide.

Create a resource file in the *res/values* folder called *colors.xml*. Modify it as below:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="primary">#3F51B5</color>
    <color name="primary_dark">#303F9F</color>
    <color name="accent">#FF4081</color>
</resources>
```

Modify res/values/styles.xml as below:

```
<style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
    <item name="colorPrimary">@color/primary</item>
    <item name="colorPrimaryDark">@color/primary_dark</item>
    <item name="colorAccent">@color/accent</item>
</style>
```

Please note that we need to use the NoActionBar in order for the navigation view to take the whole parent. So what we need to do is to create our own ToolBar and set it as the Activity's Action bar.

To create the Tool bar with action bar appearance and size bar we use the following xml syntax:

```
<android.support.v7.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    android:theme="@style/ThemeOverlay.AppCompat.Dark" />
```

Then we need to get a reference to the ToolBar and set it as the action bar.

Note: You can get a reference of the Action Bar and set its Home icon to the [infamous hamburger menu icon](#). First add the icon to your drawable folder, then get a reference to the action bar and set the up button enable but replace regular back arrow with the hamburger icon.

## Navigation View

The Navigation Drawer is a common component in Android apps. It is one of the ways to build a navigation hierarchy on Android. Implementing it with the Design Support Library is much simpler than it was before:

To get started, We need to set our root layout to **android.support.v4.widget.DrawerLayout** give it an id and inside it we add the root relative layout that was before with the ToolBar. Then we add the **android.support.design.widget.NavigationView** as a subview inside the Drawer layout. Notice the **layout\_gravity** property which can be start/end if the drawer appears from the left or right of the screen (it must correlate view's `openDrawer(GravityCompat.START/END)`) that we will use to slide that drawer upon the hamburger press. Other two important properties are the **app:headerLayout="@layout/drawer\_header"**, which define a custom layout for the navigation headline (it is optional) and the **app:menu="@menu/drawer"**, which must reference the menu items xml file that will be inflated and can also be changed at runtime.

When creating the menu file you can use the group tag and specify a few items with the **checkableBehavior** that can be all/single or none, You can also create a sub menu while creating another menu inside an item.

In the example we did The first part shows a collection of checkable menu items. The checked item will appear highlighted in the navigation drawer, ensuring the user knows which navigation item is currently selected. In the second part, we use a subheader to separate the second group of items from the first.

In the Java we can get the home button click(the one with the hamburger) in the already overridden function `onOptionsItemSelected`. The button id will be the `android.R.id.home` platform resource for the home button. Then get a reference to the `DrawerLayout` and call the `openDrawer` function passing it the Gravity (or the `GravityCompat` in case of the support library `START` or `END` (same as in the xml file).

To capture click events on the menu items we need to set an `OnNavigationItemSelectedListener` on the `NavigationView`: Get a reference to the navigation view in the `onCreate` function and use the `setNavigationItemSelectedListener` passing it a `NavigationView.OnNavigationItemSelectedListener` instance and override the `onNavigationItemSelectedListener` function. Inside you can `setChecked(true)` the menuitem passed(this will only affect the menu items marked as checkable - not the bottom options), close the drawer with the `DrawerLayout's closeDrawers()` function and do whatever else you want

## Floating Action Button

A floating action button is a round button that appear floating on the layout that marks a primary action on your interface. By default colored using the colorAccent from your theme.

The normal size is 56dp, it also supports the mini size 40dp. To add the floating button just add the **android.support.design.widget.FloatingActionButton** to the xml file, give it an id, a drawable source and place it wherever you want on the screen and that's it. you can set a custom image with setImageDrawable().

To receive the click on the button just get a reference to it and use the setOnClickListener.

## Snackbar & CoordinatorLayout

Traditionally, if you wanted to present quick brief feedback to the user, you would use a Toast. Now there is another option – the Snackbar. Snackbars provide lightweight feedback about an operation. They show a brief message at the bottom of the screen on mobile and lower left on larger devices, they contain text with an optional single action. They automatically remove themselves after the given time by animating off the screen. Users can also swipe them away before the timeout.

You can also interact with the Snackbar through actions and swiping to dismiss, they are definitely the new Toasts.

To display a Snackbar we use the static method make, passing the view to be displayed on, the message to show and the time length. After creating it we can add actions to it using the setAction method and specify the action's title and the onClickListener. When finished configuring the Snack Bar we call show().

Snackbar will attempt to find an appropriate parent of the Snackbar's view to ensure that it is anchored to the bottom. If we test the app now we will see that the snackbar overlaps the floating action button and the solution to this is the **coordinator layout**.

The Design library introduces the CoordinatorLayout, a layout which provides an additional level of control over touch events between child views, something which many of the components in the Design library take advantage of.

An example of this is when you add a `FloatingActionButton` as a child of your `CoordinatorLayout` and then pass that `CoordinatorLayout` to your `Snackbar.make()` call, then he will coordinate between them

Instead of the snackbar displaying over the floating action button, as seen previously, the `FloatingActionButton` takes advantage of additional callbacks provided by `CoordinatorLayout` to automatically move upward as the snackbar animates in and returns to its position when the snackbar animates out. To test this just replace the `RelativeLayout` that holds the Toolbar and the Floating action button with coordinator layout and also pass it as the view which the snack bar will be displayed on.

Note: the Coordinate layout won't accept the `RelativeLayout` `layout_alignParentBottom="true"` and `layout_alignParentRight="true"` (the Floating action bar properties) just replace them with `android:layout_gravity="bottom|right"`.

Important note: the Floating action bar comes with built in Behavior class that define it's behaviour in the coordinate layout - we can change it or add something like that to other elements if we want them to act the same way

## Floating Labels for EditText

The Design support library also introduced an improved capability of `EditText`: Normally when typing is started on an `EditText`, the placeholder hint that was on the field is hidden. Now you can wrap an `EditText` in a `TextInputLayout` causing the hint text to become a floating label above the `EditText`, ensuring that users never lose context in what they are entering.

To use this just add you `EditText` in an **`android.support.design.widget.TextInputLayout`** and That's it!

## TabLayout (with ViewPager)

The Design library's `TabLayout` can also help you adding tabs to your app. It implements both fixed tabs, where the view's width is divided equally between all of the tabs, as well as scrollable tabs, where the tabs are not a uniform size and can scroll horizontally.

You can add the `TabLayout` with a view pager to enable horizontal scrolling. First add the `TabLayout` to the app. Add **`android.support.design.widget.TabLayout`** to your layout file give it an id, set the background property of the tabs (the tabs headline) to the android attribute

"?attr/colorPrimary", the `app:tabGravity` to center if you want them to be centered or `fill`(default) to be spread equally in the tab layout, and set the theme (causing the tab text and selection to look the same as the action bar). Then add the `ViewPager` and make him take all the space that's left with the `weight` property. You can use the `app:tabMode="scrollable"` when you have many number of tabs where there is insufficient space on the screen to fit all of them.

Create your displaying fragments class and the fragment adapter as an inner class in the main activity. The view pager will enable horizontal paging between the fragments and each fragment will be a tab.

Now we need to connect all of them together: create an instance of your adapter and set it as the view pager adapter, then get a reference to the tab layout and use the `setupWithViewPager` passing it the view pager, This ensures that tab selection events update the `ViewPager` and page changes update the selected tab.

Note: To add an icon to the tab, after assigning the `viewpager` as the source of the tabs get a reference to the tabs using the `TabLayout`'s `getTabAt(index)` and then set the icon using the `setIcon(RESID)`.

## AppBarLayout

The `AppBarLayout` allows the `Toolbar` and other views (such as tabs provided by `TabLayout`) to react to scroll events in a sibling view marked with a `ScrollingViewBehavior`. When the user scrolls through the `RecyclerView`, the `AppBarLayout` responds by using its children's scroll flags to control how they enter (scroll on screen) and exit (scroll off screen).

Flags include:

- **scroll**: This flag should be set for all views that want to scroll off the screen. For views that do not use this flag, they'll remain pinned to the top of the screen
- **enterAlways**: This flag ensures that any downward scroll will cause this view to become visible, enabling the 'quick return' pattern
- **enterAlwaysCollapsed**: When your view has declared a `minHeight` and you use this flag, your View will only enter at its minimum height (i.e., 'collapsed'), only re-expanding to its full height when the scrolling view has reached its top.

- **exitUntilCollapsed:** This flag causes the view to scroll off until it is 'collapsed' (its minHeight) before exiting

Note that all views using the scroll flag must be declared before views that do not use the flag. This ensures that all views exit from the top, leaving the fixed elements behind.

To see this in our app we first need to add a RecyclerView (All the material design components don't work well with ListView) to the fragment do the following steps:

1. Create the layout file for the fragments the layout will contain only a RecyclerView
2. Create the line layout in each of the cells - simple linear layout containing only textview
3. Create the RecyclerView adapter
4. Change the fragments onCreateView: create an array list of all the items to display in the view, inflate the fragments layout get a reference to the RecyclerView and set the adapter with the list we created and the layout manager to the LinearLayoutManager.
5. Last, we need to define ***an association between*** the AppBarLayout and **the View that will be scrolled**. Add an app:layout\_behavior to a RecyclerView or any other View capable of nested scrolling such as NestedScrollView. The support library contains a special string resource **@string/appbar\_scrolling\_view\_behavior** *that maps to **AppBarLayout.ScrollingViewBehavior**, which is used to notify the AppBarLayout when scroll events occur on this particular view.* The behavior must be established on the view that triggers the event.

After doing so just remove the LinearLayout from the main activity xml file, change the view pager height to match\_parent and remove the weight property and instead put both the Toolbar and the TabLayout childs of the android.support.design.widget.AppBarLayout. If you want only the Toolbar to disappear when scrolling give him the app:layout\_scrollFlags Property with the values mentioned above.

## CollapsingToolbarLayout

**CollapsingToolbarLayout** is a wrapper for Toolbar which implements a collapsing app bar. It is designed to be used as a direct child of a **AppBarLayout**. What we we get is an image that collapsed and expanded while scrolling.

To implement this: First give the AppBarLayout a fixed height then create the CollapsingToolbarLayout as his child giving it a background (either color or drawable) in the **app:contentScrim** attribute - this is the background that we want see when the image is collapsed. It is also needed to assign the **app:layout\_scrollFlags**, you can use the **scroll** and the **exitUntilCollapsed** flags together (scroll will make it scrolled out of the screen and the exitUntilCollapsed will ensure that we will stop when it is collapsed).

After doing so add an ImageView as a child of the CollapsingToolbarLayout and set the **app:layout\_collapseMode**, the values can be either **pin** or **parallax**. **pin** can be used in order to pin the bottom of the image to the toolbar and the upper part of the image will disappear when collapsing, or **parallax** that will cause the image to be hidden from the top and the bottom. The **layout\_collapseParallaxMultiplier** determine which part of the image will be hidden under the button content, it ranges from 0-1, 0 is like pin(hides only the top) and 1.0 means that top boundary of appbar's image is binded to the top edge of screen and doesn't move when scrolling(meaning the only bottom of the image will disappear while collapsing). When parameter is not set this corresponds to the value 0.5 and image will be overlapped above and below Synchronously.

After doing so add the Toolbar as the second child of the CollapsingToolbarLayout and set his **app:layout\_collapseMode** to "pin" in order to ensure that the tool won't disappear from the screen when collapsing.

## **Reference**

[\*\*Material Design with the Android Design Support Library\*\*](#), by Joyce Echessa