# Tool Bar and Fragments

**What this lecture we will see**

* Tool Bars
  + Setting Up the App Bar
  + Add a toolbar to an Activity
  + Creating Action to the toolbar
  + Reacting to action selection
  + Styling and Custom the toolbar
* Fragments
  + Fragment Lifecycle
  + An Example: Creating the First Fragment Layout
  + Adding the Fragment to the Activity

## Tool Bar

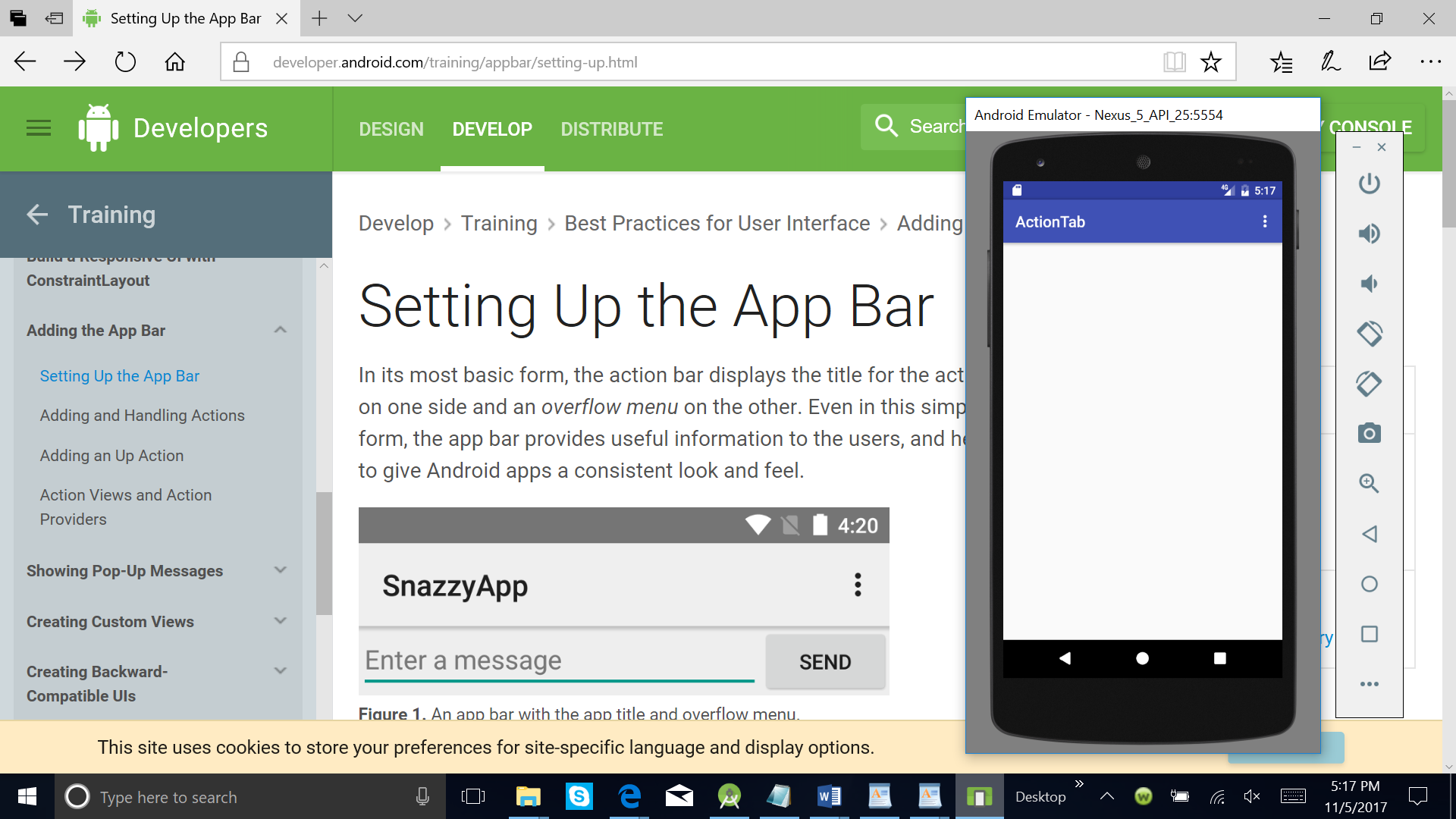
The *toolbar bar* (formerly known as action bar) is represented as of Android 5.0 via the Toolbar view group. It can be freely positioned into your layout file. It can display the activity title, icon, actions which can be triggered, additional views and other interactive items. It can also be used for navigation in your application.



App bar structure

## Setting Up the App Bar

In its most basic form, the action bar displays the title for the activity on one side and an overflow menu on the other. Even in this simple form, the app bar provides useful information to the users, and helps to give Android apps a consistent look and feel.



The most recent features of App Bar are added to the support library's version of Toolbar, and they are available on any device that can use the support library.

For this reason, you should use the support library's Toolbar class to implement your activities' app bars. Using the support library's toolbar helps ensure that your app will have consistent behavior across the widest range of devices.

## Add a Toolbar to an Activity

Let create a new Android project In Android Studio and we choose the **Basic Action Option**. To set up a [Toolbar](https://developer.android.com/reference/android/support/v7/widget/Toolbar.html) as your activity's app bar we need:

1. Add the [v7 appcompat](https://developer.android.com/tools/support-library/features.html#v7-appcompat) support library to your project.

(**How you setup the Android Support Libraries in your development project depends on what features you want to use and what range of Android platform versions you want to support with your application**.

[**https://o7planning.org/en/10525/how-to-add-external-libraries-to-android-project-in-android-studio**](https://o7planning.org/en/10525/how-to-add-external-libraries-to-android-project-in-android-studio)

[**AppCompatActivity**](https://developer.android.com/reference/android/support/v7/app/AppCompatActivity.html) **- Provides Material color themes, widget tinting, and** [**app bar**](https://developer.android.com/training/appbar/index.html) **support to earlier devices. Use of this class requires that you use Theme.AppCompat themes for consistent visual presentation**.)

1. Make sure the activity extends [AppCompatActivity](https://developer.android.com/reference/android/support/v7/app/AppCompatActivity.html).:

public class MainActivity extends AppCompatActivity {  
  // ...  
}

**Note:** Make this change for every activity in your app that uses a [Toolbar](https://developer.android.com/reference/android/support/v7/widget/Toolbar.html) as an app bar.

1. In the app manifest, set the [<application>](https://developer.android.com/guide/topics/manifest/application-element.html) element to use one of appcompat's [NoActionBar](https://developer.android.com/reference/android/support/v7/appcompat/R.style.html#Theme_AppCompat_NoActionBar) themes. Using one of these themes prevents the app from using the native [ActionBar](https://developer.android.com/reference/android/app/ActionBar.html) class to provide the app bar. For example:

<application  
    android:theme="@style/Theme.AppCompat.Light.NoActionBar"  
    />

1. Add a [Toolbar](https://developer.android.com/reference/android/support/v7/widget/Toolbar.html) to the activity's layout. For example, the following layout code adds a [Toolbar](https://developer.android.com/reference/android/support/v7/widget/Toolbar.html) and gives it the appearance of floating above the activity:

<android.support.design.widget.AppBarLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:theme="@style/AppTheme.AppBarOverlay">  
  
 <android.support.v7.widget.Toolbar  
 android:id="@+id/toolbar"  
 android:layout\_width="match\_parent"  
 android:layout\_height="?attr/actionBarSize"  
 android:background="?attr/colorPrimary"  
 app:popupTheme="@style/AppTheme.PopupOverlay" />  
  
</android.support.design.widget.AppBarLayout>

Position the toolbar at the top of the activity's [layout](https://developer.android.com/guide/topics/ui/declaring-layout.html), since you are using it as an app bar.

1. In the activity's [onCreate()](https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) method, call the activity's [setSupportActionBar()](https://developer.android.com/reference/android/support/v7/app/AppCompatActivity.html#setSupportActionBar(android.support.v7.widget.Toolbar)) method, and pass the activity's toolbar. This method sets the toolbar as the app bar for the activity. For example:

@Override  
**protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 Toolbar toolbar = (Toolbar) findViewById(R.id.***toolbar***);  
 setSupportActionBar(toolbar);  
}

Your app now has a basic action bar. By default, the action bar contains just the name of the app and an overflow menu. The options menu initially contains just the **Settings** item. You can add more actions to the action bar and the overflow menu.

## [Creating actions in the toolbar](http://www.vogella.com/tutorials/AndroidActionBar/article.html#creating-actions-in-the-toolbar)

Entries in the toolbar are typically called actions. While it is possible to create entries in the action bar via code, it is typically defined in an XML resource file.

Each menu definition is contained in a separate file in the res/menu folder. The Android tooling automatically creates a reference to menu item entries in the R file, so that the menu resource can be accessed.

An activity adds entries to the action bar in its onCreateOptionsMenu() method.

<menu xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 tools:context="com.example.user.actionbar.MainActivity">  
  
 <item  
 android:id="@+id/menuitem\_search"

app:showAsAction="ifRoom"

android:title="@string/ui\_menu\_search">  
 </item>  
 <item  
 android:id="@+id/menuitem\_send"  
 android:title="@string/ui\_menu\_send">  
 </item>  
 <item  
 android:id="@+id/menuitem\_add"  
 android:title="@string/ui\_menu\_add">  
 </item>  
 <item  
 android:id="@+id/menuitem\_share"  
 android:title="@string/ui\_menu\_share">  
 </item>  
 <item  
 android:id="@+id/menuitem\_feedback"  
 android:title="@string/ui\_menu\_feedback">  
 </item>  
 <item  
 android:id="@+id/menuitem\_about"  
 android:title="@string/ui\_menu\_about">  
 </item>  
 <item  
 android:id="@+id/menuitem\_quit"  
 android:title="@string/ui\_menu\_quit">  
 </item>  
  
</menu>

The MenuInflater class allows to inflate actions defined in an XML file and adds them to the action bar. MenuInflater can get accessed via the getMenuInflater() method from your *activity*. The following example code demonstrates the creation of actions.

@Override

public boolean onCreateOptionsMenu(Menu menu) {

MenuInflater inflater = getMenuInflater();

inflater.inflate(R.menu.mainmenu, menu);

return true;

}

## [Reacting to action selection](http://www.vogella.com/tutorials/AndroidActionBar/article.html#reacting-to-action-selection)

If an action is selected, the onOptionsItemSelected() method in the corresponding action is called. It receives the selected action as parameter. The usage of this method is demonstrated in the following code snippet.

@Override  
**public boolean** onOptionsItemSelected(MenuItem item) {  
 **switch** (item.getItemId()) {  
 **case** R.id.***menuitem\_search***:  
 Toast.*makeText*(**this**, getString(R.string.***ui\_menu\_search***),  
 Toast.***LENGTH\_SHORT***).show();  
 **return true**;  
 **case** R.id.***menuitem\_send***:  
 Toast.*makeText*(**this**, getString(R.string.***ui\_menu\_send***),  
 Toast.***LENGTH\_SHORT***).show();  
 **return true**;  
 **case** R.id.***menuitem\_add***:  
 Toast.*makeText*(**this**, getString(R.string.***ui\_menu\_add***),  
 Toast.***LENGTH\_SHORT***).show();  
 **return true**;  
 **case** R.id.***menuitem\_share***:  
 Toast.*makeText*(**this**, getString(R.string.***ui\_menu\_share***),  
 Toast.***LENGTH\_SHORT***).show();  
 **return true**;  
 **case** R.id.***menuitem\_feedback***:  
 Toast.*makeText*(**this**, getString(R.string.***ui\_menu\_feedback***),  
 Toast.***LENGTH\_SHORT***).show();  
 **return true**;  
 **case** R.id.***menuitem\_about***:  
 Toast.*makeText*(**this**, getString(R.string.***ui\_menu\_about***),  
 Toast.***LENGTH\_SHORT***).show();  
 **return true**;  
 **case** R.id.***menuitem\_quit***:  
 Toast.*makeText*(**this**, getString(R.string.***ui\_menu\_quit***),  
 Toast.***LENGTH\_SHORT***).show();  
 finish(); *// close the activity* **return true**;  
 }  
 **return false**;  
}

## Styling the Toolbar

The Toolbar can be customized in many ways leveraging various style properties including android:theme, app:titleTextAppearance, app:popupTheme. Each of these can be mapped to a style. Start with:

<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/ToolbarTheme"  
 app:titleTextAppearance="@style/Toolbar.TitleText"  
 app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>

## Now, we need to create the custom styles in res/values/styles.xml with:

<**resources**>  
  
*<!-- Base application theme. -->*<**style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar"**>  
 *<!-- Customize your theme here. -->* <**item name="colorPrimary"**>@color/colorPrimary</**item**>  
 <**item name="colorPrimaryDark"**>@color/colorPrimaryDark</**item**>  
 <**item name="colorAccent"**>@color/colorAccent</**item**>  
</**style**>  
  
<**style name="AppTheme.NoActionBar"**>  
 <**item name="windowActionBar"**>false</**item**>  
 <**item name="windowNoTitle"**>true</**item**>  
  
</**style**>  
 <**style name="ToolbarTheme" parent=**

**"@style/ThemeOverlay.AppCompat.Dark.ActionBar"**>  
 *<!-- android:textColorPrimary is the color of the title*

*text in the Toolbar -->* <**item name=**

**"android:textColorPrimary"**>@android:color/holo\_blue\_light</**item**>  
 *<!-- actionMenuTextColor is the color of the text of action*

*(menu) items -->* <**item name=**

**"actionMenuTextColor"**>@android:color/holo\_green\_light</**item**>  
 *<!-- Tints the input fields like checkboxes and text fields -->* <**item name="colorAccent"**>@color/cursorAccent</**item**>  
 *<!-- Applies to views in their normal state. -->* <**item name="colorControlNormal"**>@color/controlNormal</**item**>  
 *<!-- Applies to views in their activated state*

*(i.e checked or switches) -->* <**item name="colorControlActivated"**>@color/controlActivated</**item**>

</**style**>  
 *<!-- This configures the styles for the title within the Toolbar -->* <**style name="Toolbar.TitleText" parent=**

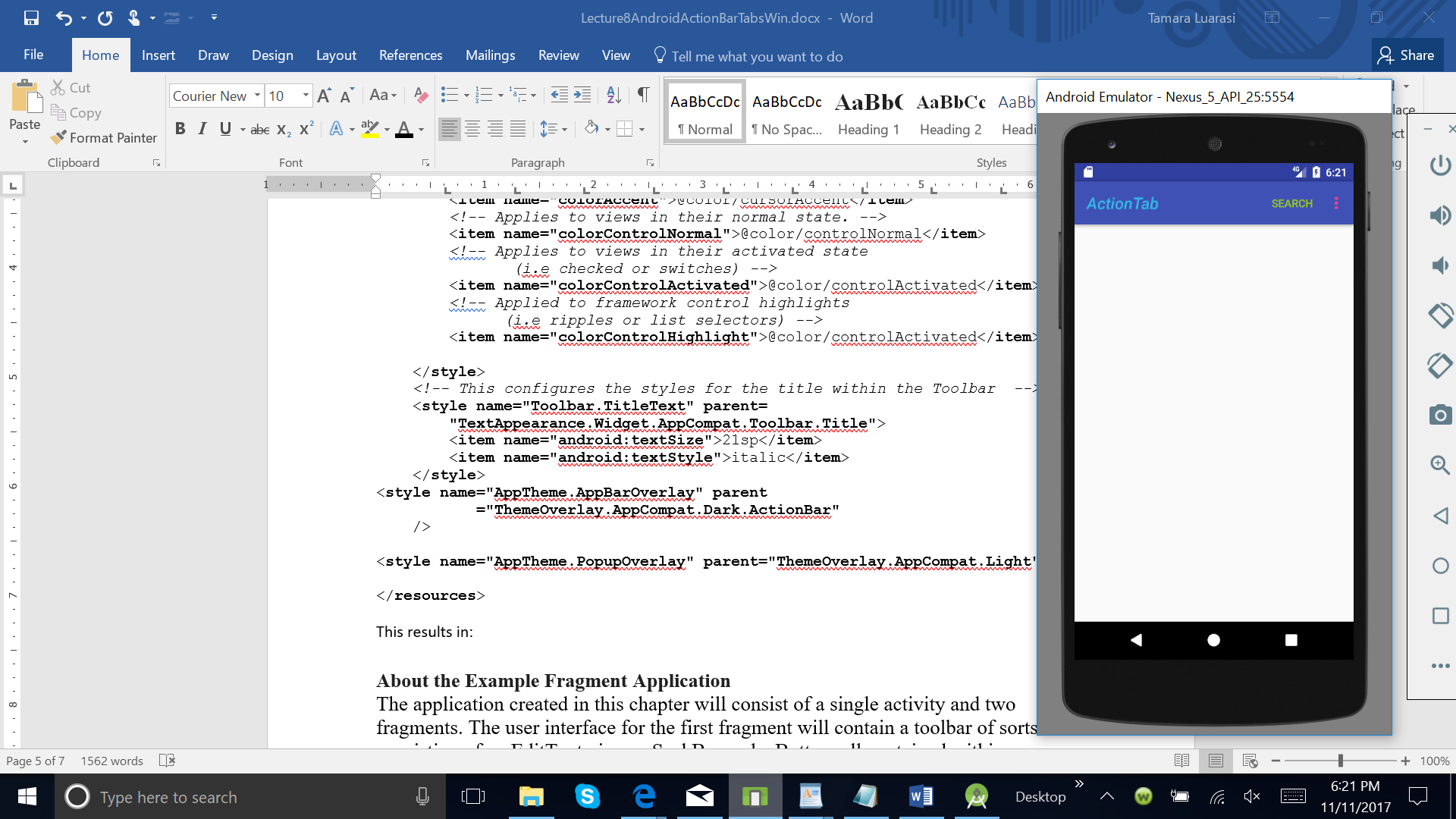
**"TextAppearance.Widget.AppCompat.Toolbar.Title"**>  
 <**item name="android:textSize"**>21sp</**item**>  
 <**item name="android:textStyle"**>italic</**item**>  
 </**style**>  
<**style name="AppTheme.AppBarOverlay" parent**

**="ThemeOverlay.AppCompat.Dark.ActionBar"**

*<!-- TITLE\_COLOR\_GOES\_HERE -->  
<!--SUBTITLE\_COLOR\_GOES\_HERE-->*

/>  
  
<**style name="AppTheme.PopupOverlay" parent="ThemeOverlay.AppCompat.Light"** />  
*<!--MENU\_POPUP\_BACK\_COLOR-->  
<!--MENU\_POPUP\_ITEMS\_COLOR-->*

</**resources**>

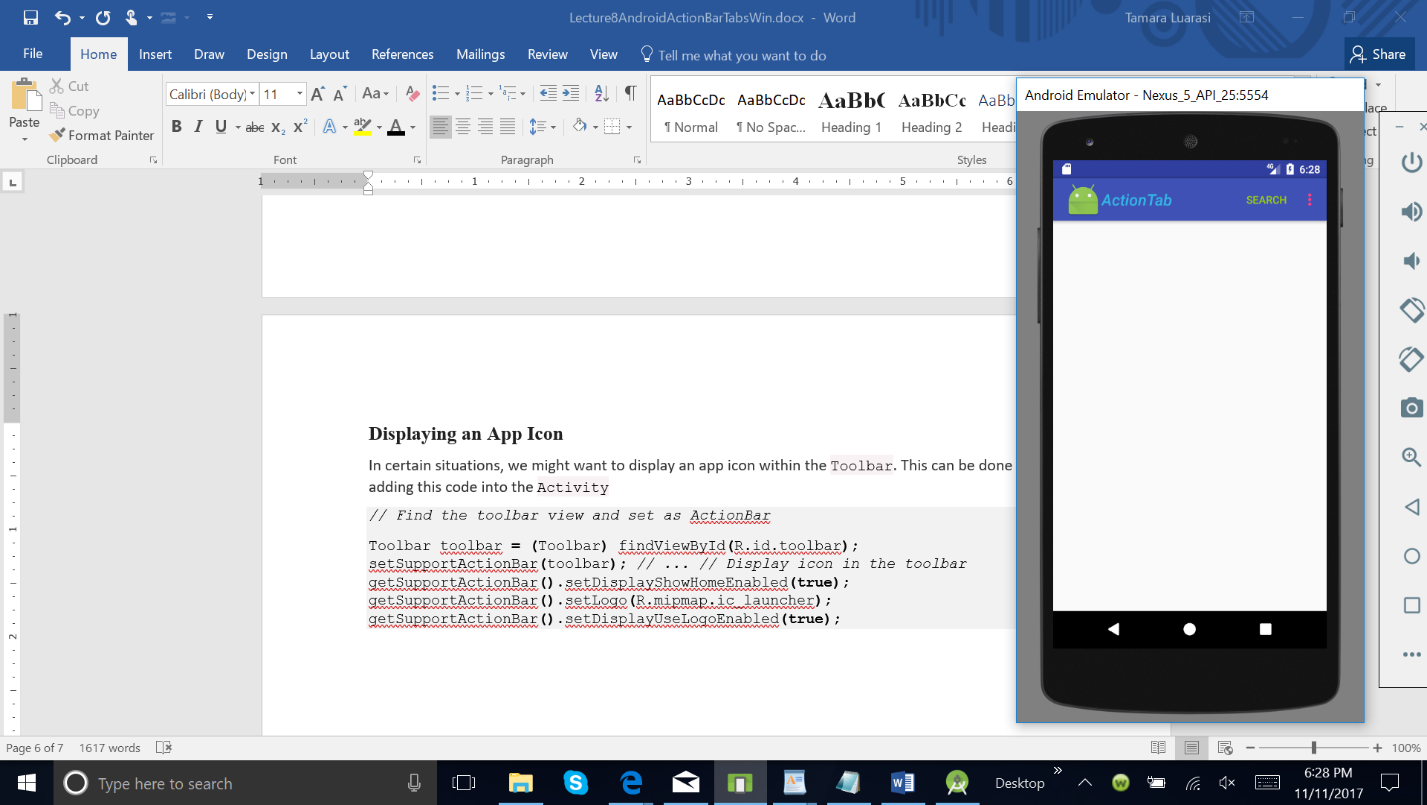
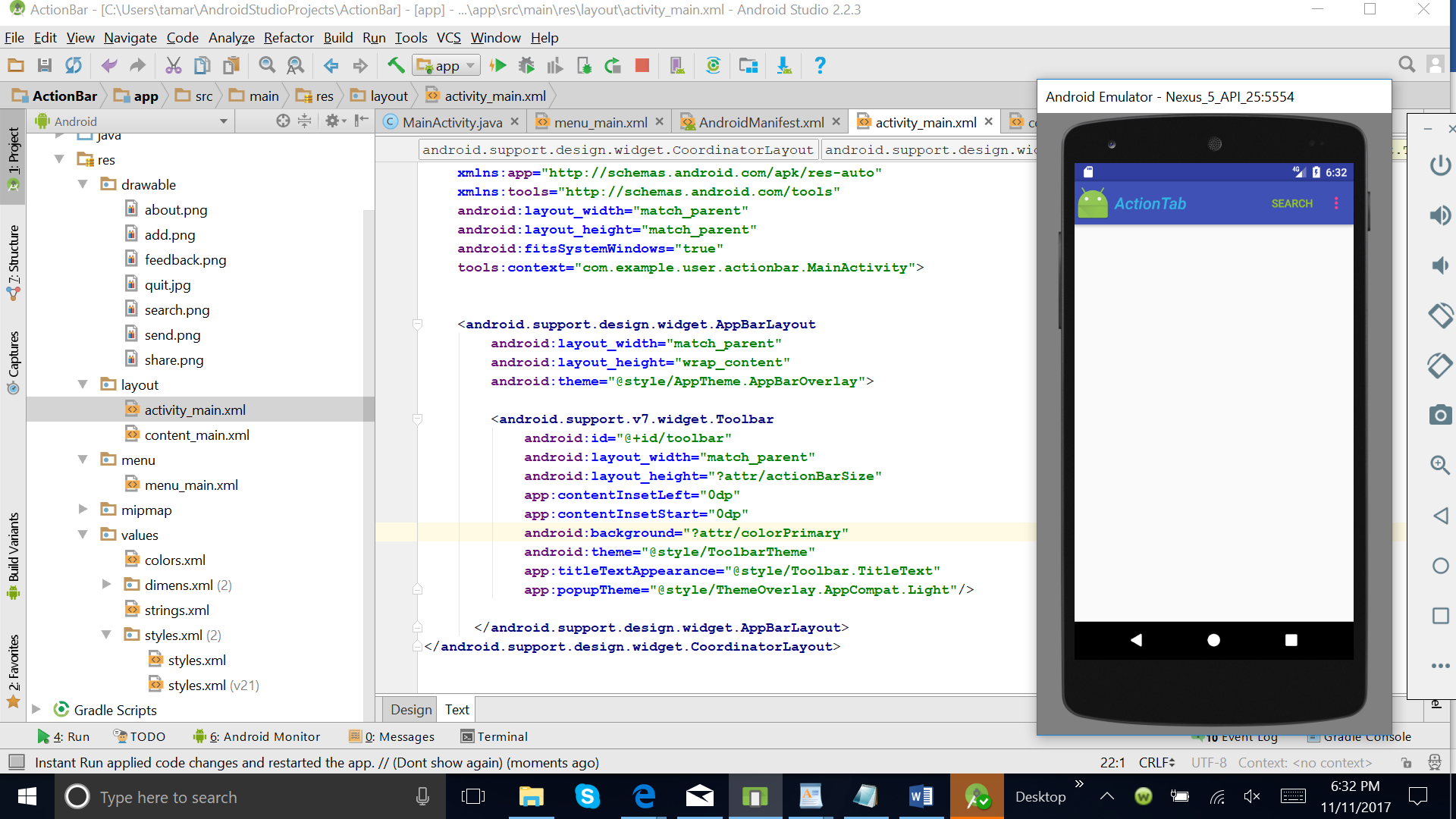
 This results in:

## Displaying an App Icon

In certain situations, we might want to display an app icon within the Toolbar. This can be done by adding this code into the Activity

// Find the toolbar view and set as ActionBar

Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar); setSupportActionBar(toolbar); // ... // Display icon in the toolbar getSupportActionBar().setDisplayShowHomeEnabled(true); getSupportActionBar().setLogo(R.mipmap.ic\_launcher); getSupportActionBar().setDisplayUseLogoEnabled(true);



## Next (second screen above), we need to remove the left inset margin that pushes the icon over too far to the left by adding app:contentInsetStart to the Toolbar:

## <android.support.v7.widget.Toolbar

## android:id="@+id/toolbar"

## app:contentInsetLeft="0dp"

## app:contentInsetStart="0dp"

## ... >

## </android.support.v7.widget.Toolbar>

## Custom Title View

## A Toolbar is just a decorated ViewGroup and as a result, the title contained within can be completely customized by embedding a view within the Toolbar such as:

<**TextView  
 android:id="@+id/toolbar\_title"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Toolbar Title"  
 android:textColor="@android:color/white"  
 style="@style/TextAppearance.AppCompat.Widget.ActionBar.Title"  
 android:layout\_gravity="center"** />

This means that you can style the TextView like any other. You can access the TextView inside your activity with:

*/\* Inside the activity \*/*

*// Sets the Toolbar to act as the ActionBar for this Activity window.*

Toolbar toolbar **=** **(**Toolbar**)** findViewById**(**R**.**id**.**toolbar**);**

setSupportActionBar**(**toolbar**);** *// Remove default title text*

toolbar.setTitle(**"ToolBar Title"**)

getSupportActionBar**().**setDisplayShowTitleEnabled**(false);**

*// Get access to the custom title view*

TextView mTitle **=** **(**TextView**)** toolbar**.**findViewById**(**R**.**id**.**toolbar\_title**);**

Note that you **must hide the default title using** setDisplayShowTitleEnabled.

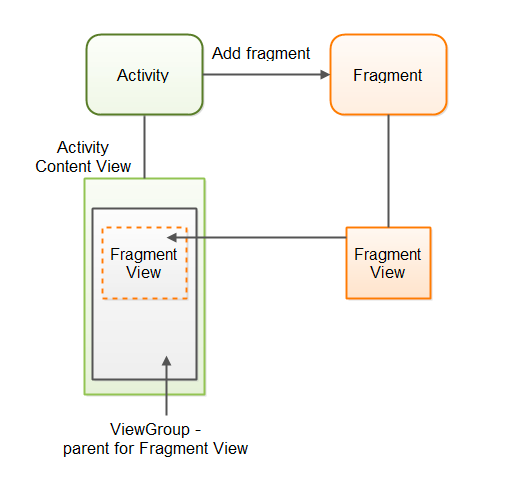
# Fragments

A Fragment represents a behavior or a portion of user interface in an Activity. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. You can think of a fragment as a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities).

A fragment must always be embedded in an activity and the fragment's lifecycle is directly affected by the host activity's lifecycle. For example, when the activity is paused, so are all fragments in it, and when the activity is destroyed, so are all fragments. However, while an activity is running (it is in the resumed lifecycle state), you can manipulate each fragment independently, such as add or remove them. When you perform such a fragment transaction, you can also add it to a back stack that's managed by the activity—each back stack entry in the activity is a record of the fragment transaction that occurred. The back stack allows the user to reverse a fragment transaction (navigate backwards), by pressing the Back button.

When you add a fragment as a part of your activity layout, it lives in a ViewGroup inside the activity's view hierarchy and the fragment defines its own view layout. You can insert a fragment into your activity layout by declaring the fragment in the activity's layout file, as a <fragment> element, or from your application code by adding it to an existing ViewGroup. However, a fragment is not required to be a part of the activity layout; you may also use a fragment without its own UI as an invisible worker for the activity.

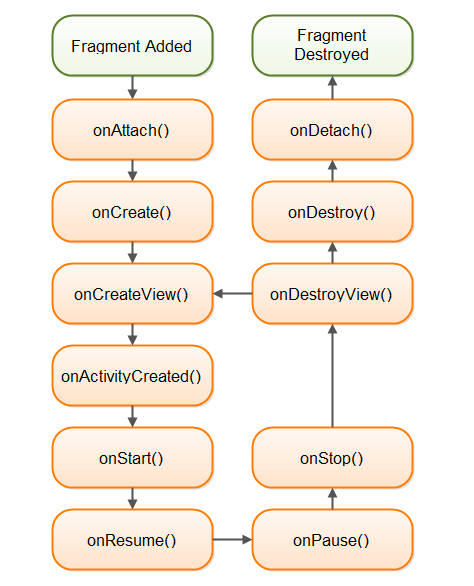
The following diagram shows depicts what happens when a fragment is added to an activity:

[[](https://cdn.journaldev.com/wp-content/uploads/2015/10/android-fragments-activity.png)](https://cdn.journaldev.com/wp-content/uploads/2015/10/android-fragments-activity.png)

First the activity obtains a reference to the fragment. Then it gets a reference to the ViewGroup the fragment’s view will be rendered inside. Then the activity adds the fragment. The fragment then creates its view and returns it to the activity. The view is then inserted into the ViewGroup parent, and the fragment is alive.

## Fragment Lifecycle

Android fragment lifecycle is illustrated in below image.

[[](https://cdn.journaldev.com/wp-content/uploads/2015/10/android-fragment-lifecycle.png)](https://cdn.journaldev.com/wp-content/uploads/2015/10/android-fragment-lifecycle.png)

Usually, you should implement at least the following lifecycle methods:

[onCreate()](https://developer.android.com/reference/android/support/v4/app/Fragment.html#onCreate(android.os.Bundle))

The system calls this when creating the fragment. Within your implementation, you should initialize essential components of the fragment that you want to retain when the fragment is paused or stopped, then resumed.

[onCreateView()](https://developer.android.com/reference/android/support/v4/app/Fragment.html#onCreateView(android.view.LayoutInflater, android.view.ViewGroup, android.os.Bundle))

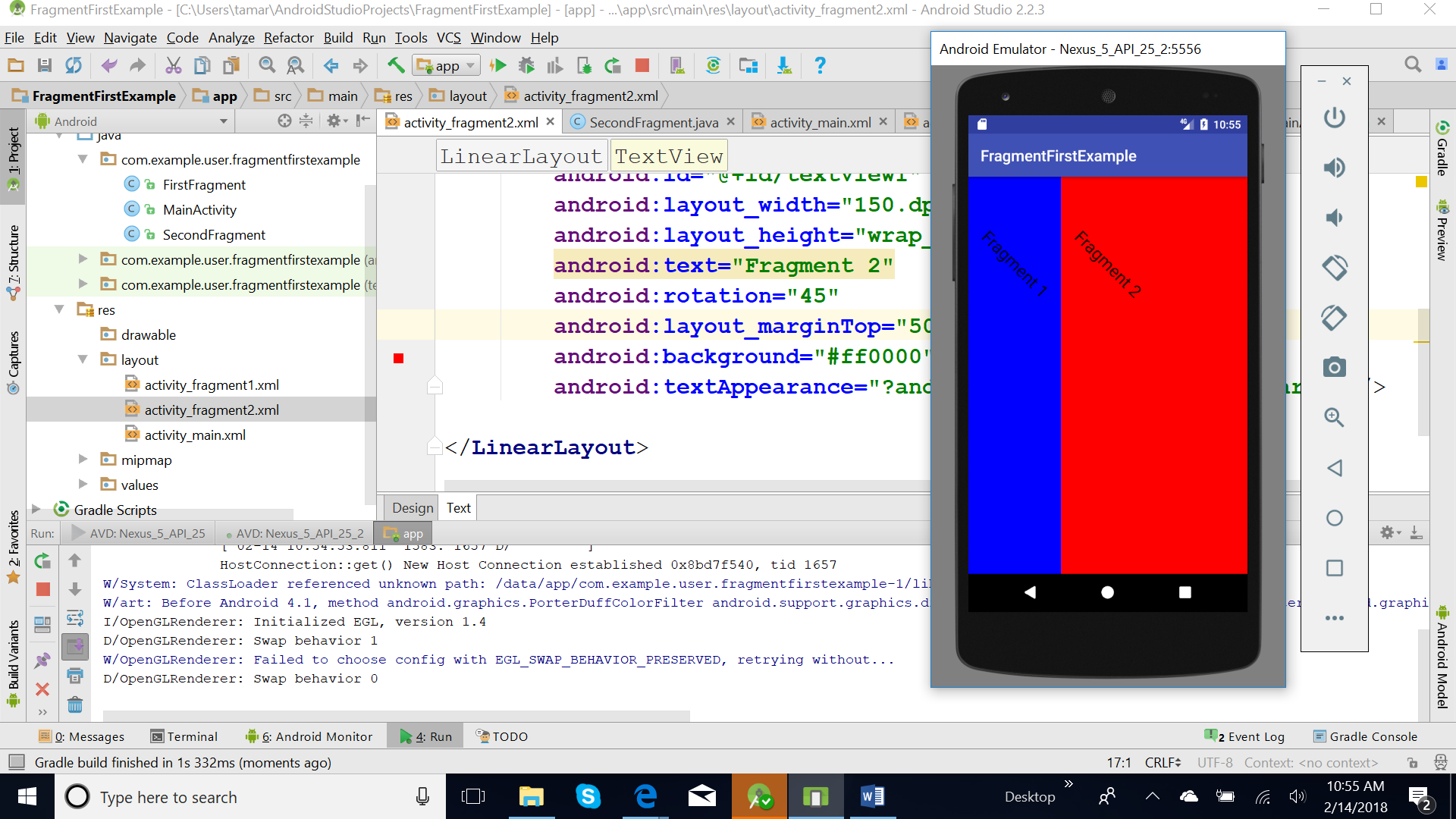
The system calls this when it's time for the fragment to draw its user interface for the first time. To draw a UI for your fragment, you must return a [View](https://developer.android.com/reference/android/view/View.html) from this method that is the root of your fragment's layout. You can return null if the fragment does not provide a UI.

[onPause()](https://developer.android.com/reference/android/support/v4/app/Fragment.html#onPause())

The system calls this method as the first indication that the user is leaving the fragment (though it does not always mean the fragment is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back).

**Example: Create some Fragments**

The first example let be a simple one, we are creating two embedded fragments in UI. We will create the following view:



1. Let’s create a new project in android studio. The name is FragmentFirstExample.
2. Create a Fragment Classes for each of two fragments and respective layouts.

To create a fragment, extend the Fragment class, then override key lifecycle methods to insert your app logic, similar to the way you would with an Activity class.

One difference when creating a Fragment is that you must use the onCreateView() callback to define the layout. In fact, this is the only callback you need in order to get a fragment running. For example, here's a simple fragment that specifies its own layout:

**import** android.os.Bundle;  
**import** android.support.v4.app.Fragment;  
**import** android.view.LayoutInflater;  
**import** android.view.View;  
**import** android.view.ViewGroup;  
  
**public class** FirstFragment **extends** Fragment {  
 @Override  
 **public** View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {  
 *// Inflate the layout for this fragment* **return** inflater.inflate(R.layout.***activity\_fragment1***, container,

**false**);  
 }  
}

Where ***activity\_fragment1*** is the layout of the fragment1 composed by just one TextField control.

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:orientation="vertical"  
 android:background="#0000ff"  
 >  
  
 <TextView  
 android:id="@+id/textView1"  
 android:layout\_width="150.dp"  
 android:layout\_height="wrap\_content"  
 android:text="Fragment 1"  
 android:rotation="45"  
 android:layout\_margintop="50dp"  
 android:background="#0000ff"  
 android:textAppearance="?android:attr/textAppearanceLarge" />  
  
</LinearLayout>

And the class of the fragment2

**import** android.os.Bundle;  
**import** android.support.v4.app.Fragment;  
**import** android.view.LayoutInflater;  
**import** android.view.View;  
**import** android.view.ViewGroup;  
  
**public class** SecondFragment **extends** Fragment {  
 @Override  
 **public** View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {  
 *// Inflate the layout for this fragment* **return** inflater.inflate(R.layout.***activity\_fragment2***, container, **false**);  
 }  
}

with the layout ***activity\_fragment2*** composed also by one TextField control.

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:orientation="vertical"  
 android:background="#ff0000"  
 >  
  
 <TextView  
 android:id="@+id/textView1"  
 android:layout\_width="150.dp"  
 android:layout\_height="wrap\_content"  
 android:text="Fragment 2"  
 android:rotation="45"  
 android:layout\_margintop="50dp"  
 android:background="#ff0000"  
 android:textAppearance="?android:attr/textAppearanceLarge" />  
  
</LinearLayout>

1. Add a Fragment to an Activity using XML

While fragments are reusable, modular UI components, each instance of a Fragment class must be associated with a parent FragmentActivity. You can achieve this association by defining each fragment within your activity layout XML file.

**Note:** FragmentActivity is a special activity provided in the Support Library to handle fragments on system versions older than API level 11. If the lowest system version you support is API level 11 or higher, then you can use a regular Activity.

In our example we have the activity:

**import** android.os.Bundle;  
**import** android.support.v7.app.AppCompatActivity;  
  
**public class** MainActivity **extends** AppCompatActivity {  
  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 }  
  
}

with the layout:

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="fill\_parent"  
 android:layout\_height="fill\_parent"  
 android:orientation="horizontal">  
  
 <fragment  
 android:id="@+id/fragment1"  
 android:name="com.example.user.fragmentfirstexample.FirstFragment"  
 android:layout\_weight="1"  
 android:layout\_width="0dp"  
 android:layout\_height="match\_parent"  
 />  
  
 <fragment  
 android:id="@+id/fragment2"  
 android:name="com.example.user.fragmentfirstexample.SecondFragment"  
 android:layout\_weight="2"  
 android:layout\_width="0dp"  
 android:layout\_height="match\_parent"  
  
 />  
</LinearLayout>

If you're using the v7 appcompat library, your activity should instead extend AppCompatActivity, which is a subclass of FragmentActivity.

We can test now our project. Let give now some life to our fragment and make them talks with activity.

# Assignment

The first Fragment has a list. When we click an item on the list some additional information will be displayed on the second fragment.

Some suggestions: there are needed the following elements

**Layout of first fragment**

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical" android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent">

.. .   
 a list here

. . .

</LinearLayout>

**Class of the first fragment**

. . .  
  
public class FirstFragment extends ListFragment {

// information  
    String[] . . . = . . .//content of the list

    String[] . . . =. . . //the information that will be displayed

    String[] . . . =. . . // the information that will be displayed

    @Override  
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {  
        View view =inflater.inflate(R.layout.firstfragment\_layout, container, false);  
        ArrayAdapter<String> adapter = new ArrayAdapter<String>(getActivity(),  
                android.R.layout.simple\_list\_item\_1, users);  
        setListAdapter(adapter);  
        return view;  
    }  
    @Override  
    public void onListItemClick(ListView l, View v, int position, long id) {

//activate the second fragment; create an object of the second fragment

Fragment2 txt = (Fragment2)getFragmentManager().findFragmentById(R.id.fragment2);  
 . . .// call some methods of the second fragment to display information

    }  
}

**Layout of second fragments**

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical" android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent"  
    android:background="#0079D6">  
 ….

. . .

Some textfieds here

</LinearLayout>

**Class of the second fragment**

public class SecondFragment extends Fragment {  
    TextView name,location;  
    @Override  
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {  
        View view = inflater.inflate(R.layout.details\_info, container, false);  
 //create textfieslds here and link them with UI (respective UI)

        return view;  
    }

//the method here that display information and it is used from the first class  
    public void . .. (. . . .){  
        . ..

    }  
}

**Layout of the MainActivity**

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent"  
    android:orientation="horizontal"  
    tools:context=". . . .MainActivity">  
  
    <fragment  
        android:layout\_height="match\_parent"  
        android:layout\_width="350px"  
        class=". . . .FirstFragment"  
        android:id="@+id/fragment1"/>  
    <fragment  
        android:layout\_width="match\_parent"  
        android:layout\_height="match\_parent"  
        class=". . .SecondFragment"  
        android:id="@+id/fragment2"/>  
</LinearLayout>

**Resources**

https://guides.codepath.com/android/Using-the-App-ToolBar#custom-title

<https://developer.android.com/guide/components/fragments.html>

<https://developer.android.com/training/basics/fragments/fragment-ui.html>