

Android - Fragments

Fragments were introduced in Android 3.0 (API level 11), this version of android was designed for tablets. The tablet's screen size allow us to split it into small parts and give each one his own logic and life cycle and make him independent, these are Fragments. For example, in a single screen a news application can use one fragment to show a list of articles on the left and another fragment to display an article on the right—both fragments appear in one activity, side by side, and each fragment has its own set of lifecycle callback methods and handle their own user input events.

The Fragment has its own layout (xml file) and its own logic (java file), so we can include one fragment in multiple activities, for example Login window, (That is why we should avoid using one fragment from another fragment in order for them to be independent so we can reuse them). A fragment contributes its own layout to the activity. We can think of a fragment as a sub activity, a light-weighted activity. When the activity paused or destroyed all of its fragments are also paused and destroyed but while the activity is running we can add and remove as many fragments as we want - this is called fragment transactions. The fragment can easily get the context by calling his `getActivity()` function that returns his hosting activity.

Adding a fragment

First we need to create the fragment:

1. Create an xml layout file representing the layout for this Fragment.
2. Create a new java file and extend the Fragment class (or his sub classes - discussed later).
3. In the fragment class file override the **onCreateView** function and inflate the proper xml file you created in stage one: `inflater.inflate(R.layout.example_fragment, container, false);`
The inflater to inflate the xml and the container to hold the fragment's view is passed to the `oncreateView` from the system. Using the container will cause this fragment to have the same layout params as his container and will make the container to be the parent view, if we don't want this we can pass null. the last argument is a boolean indicating whether we want to attach the fragment view to the container, here we use false since we are already doing this. after setting the layout you can get references to each of the view's widgets and perform any additional customizations such as adding buttons listeners, and then we must return the newly customized view.

Adding the Fragment to the activity:

You can add the fragment either programmatically or in the layout xml file:

At any time while your activity is running, you can add fragments to your activity layout. You simply need to specify a ViewGroup in which to place the fragment.

1. Get an instance of the `FragmentManager` from the activity
2. Get an instance of the `FragmentTransaction` by calling **`beginTransaction()`** on your manager.
3. Create an instance of your fragment to be added.
4. Use the `FragmentTransaction` add function to add the new instance to the fragments root layout. The first argument passed to `add()` is the `ViewGroup` in which the fragment should be placed, specified by resource ID (this will be the container mentioned below), and the second parameter is the fragment to add. You can optionally add a third parameter which is a string indicating the fragment's tag name, this will help you get a reference to the fragment later on.

Note: If you there is no special container for the fragment in your layout and you just need the activity's root layout you can pass the **`android.R.id.content` as the fragment container id, this will automatically be your root layout.**

5. call the `FragmentTransaction` commit function to perform the transactions.

The full code can be done in a single line:

```
getFragmentManager().beginTransaction().add(R.id.fragment_container, new  
ExampleFragment(),"tag_name").commit();
```

adding the fragment through the xml layout file:

Use the `<fragment>` tag inside the desired viewgroup like it was a regular widget but you must supply the `android:name` attribute indicating the name of the fragment's java file

Note: Each fragment requires a unique identifier that the system can use to restore the fragment if the activity is restarted (and which you can use to capture the fragment to perform transactions, such as remove it). You can either Supply the `android:id` attribute with a unique ID, or Supply the `android:tag` attribute with a unique string. If you provide neither of the previous two, the system uses the ID of the container view.

Manage Fragment transactions

To manage the fragments in your activity, you need to use `FragmentManager`. To get it, call `getFragmentManager()` from your activity. With the manager you can get fragments that exist in the activity, with `findFragmentById()` or `findFragmentByTag()` (read section 3 in adding the fragments). As demonstrated in the previous section, you can also use `FragmentManager` to open a `FragmentTransaction`, which allows you to perform transactions, such as **add**, **remove**, **replace**, and perform other actions with them. You can also save each transaction to a back stack managed by the activity, allowing the user to navigate backward through the fragment changes (similar to navigating backward through activities).

You can set up all the changes you want to perform for a given transaction using methods such as `add()`, `remove()`, and `replace()`. Then, to apply the transaction to the activity, you must call `commit()`. All fragment are added to the activity as a stack when the last added one is on the top. All transaction are executed when calling commit so you can do multiple transactions at once.

remove(fragment) receives a fragment instance currently visible and removes it from the fragments stack. You can get that instance with the fragment's tag and the `findFragmentByTag(tag)` explained above. **replace(container_id, fragment, tag)** Replaces an existing fragment that was added to a container. This is essentially the same as calling `remove(Fragment)` for all currently added fragments that were added with the same `containerViewId` and then `add(int, Fragment, String)` with the same arguments given here. If no fragment is currently visible then it simply add the new fragment to the container.

Before you call `commit()`, however, you might want to call `addToBackStack(tag)`, in order to add the transaction to a back stack of fragment transactions. This back stack is managed by the activity and allows the user to return to the previous fragment state by pressing the *Back* button. Note: you will have to override the activity's `onBackPressed()` and restore the last fragment transaction by using `getFragmentManager().popBackStack` and passing it the same transaction you saved tag name.

Tip: For each fragment transaction, you can apply a transition animation, by calling `setTransition()` before you commit.

Caution: You can commit a transaction using `commit()` only prior to the activity *saving its state* (when the user leaves the activity). If you attempt to commit after that point, an exception will be

thrown. This is because the state after the commit can be lost if the activity needs to be restored. For situations in which it's okay that you lose the commit, use [commitAllowingStateLoss\(\)](#).

Communicating with the fragment

The activity can pass information to the fragment when it creates him. either in the fragment's constructor or, and this is more common, by using the fragment's bundle. For this you will have to create a static function inside your fragment class that receives the necessary information from the activity and return a new Instance of this fragment but before doing so create a new bundle and insert the information passed with as a key key-value pairs and use the newly created fragment `setArguments(bundle)` function to set the new bundle. Then later on in the rest of fragment's functions like `onCreateView` get that bundle using the fragment `getArguments()` function.

If you want the fragment to send events back to the activity which he is displayed on you will need to implement the Callback mechanism:

1. Inside the fragment create an interface with the functions that you want, meaning all the callback events that you want to notify the activity about.
2. declare a private member of the interface you just created, this will be your callback reference.
3. Override the in `onAttach(activity)` function that is called once the fragment is attached to your activity, inside it make the activity passed to the function to be the your callback(cast him to the interface type). Note: it is recommended here to throw a **ClassCastException** that will be thrown if the activity isn't implementing the interface.
4. Where ever you want to pass an event to the activity use the callback reference with the interface function you want(pass the arguments as needed).
5. Make your activity implements the callback interface, override the interface methods and perform the actions that you want.

Additional Fragment subclassed to extends

[DialogFragment](#) - Displays a floating dialog. Using this class to create a dialog is a good alternative to using the dialog helper methods in the [Activity](#) class which are now deprecated, meaning if you want more control over the dialog you can use this class. When subclassing this class there is no need to override **onCreateView** if you don't want custom view, but instead override the **onCreateDialog** function. There create the dialog with the Alert Builder as before and after all the customizations return the newly created dialog. You can also intercept the dialog lifecycle events by overriding the events in the fragments (onPrepare, onCancel, onDismiss and more). If you want to use customized view you can create your xml file and use the fragments onCreateView function as before inside it you can get the dialog with the **getDialog** function. To show a DialogFragment create an instance of it and call his **show** function passing it the fragment transaction and the tag.

[ListFragment](#) - Displays a list of items that are managed by an adapter, similar to [ListActivity](#). It provides several methods for managing a list view, such as the [onListItemClick\(\)](#) callback to handle click events. Note: If you inflate your own xml and not using the default list, then same as in ListActivity you will also have to add a ListView to the xml file with the following id: **android:id="android:list"**. In the onCreateView you can define your adapter and use the ListFragment **setAdapter()** function.

[PreferenceFragment](#) - Displays a hierarchy of [Preference](#) objects as a list, similar to [PreferenceActivity](#). This is useful when creating a "settings" activity for your application. To use the PreferenceFragment create a new preference xml file under the res/xml folder the main tag will be **<PreferenceScreen>** the device it to **<PreferenceCategory>** according to your needs. In each category use can use one the following **<CheckBoxPreference>** to display a checkbox, **<EditTextPreference>** to show a dialog with edit text to receive input or **<ListPreference>** do display a list of options to the user (radio buttons) and more. For each of the above you can give **android:title** to specify the title and **android:summary** for additional text and the most important is **android:key** to specify the key it will be saved in the SharedPreferences object. for the **<EditTextPreference>** you can also specify **android:dialogTitle** to display extra text in the dialog

open text field. For the <ListPreference> also add the **android:entries(the choices)** and **android:entryValues(the displayed choices)** both can be string arrays from the resources.

After finishing the xml file just create a new java file extend the PreferenceFragment. On his onCreate function just load the preferences using **addPreferencesFromResource(R.xml.preference)**. The settings will automatically be saved in the shared preferences. To retrieve them use the `PreferenceManager.getDefaultSharedPreferences(context)` and the key given before in the xml file. **Note:** it is very common to display this fragment on his own activity for the user experience, when he pushed the back button he won't close the top activity (or the app) but instead just the settings screen.

ViewPager with FragmentStatePagerAdapter

A ViewPager is a widget that allows us to easily switch views by swiping. It is usually use a `FragmentStatePagerAdapter` instance as his data source. The adapter creates the fragments upon the pager request and passes them to the pager.

The steps to use a ViewPager with `FragmentPagerAdapter` are as follows:

1. Add the ViewPager to your xml layout file and give it an id. Note: the widget currently belongs to the android support library so you should use the `<android.support.v4.view.ViewPager>`
2. create a new class (can be an inner class) and extend the **FragmentStatePagerAdapter**.
3. In that class create a constructor and override the **getCount()** and **getItem(int)** functions. in the `getCount` return the number of switching fragments and in the `getItem(int)` create the appropriate fragment according to the position passed (the `getItem` function parameter).
4. in your activity file (the one presenting the viewPager) get a reference to your pager from the xml file, create a new instance of the `FragmentStatePagerAdapter` subclass you created and use the pager's **setAdapter** function to set it as the pager data source.

Fragment's lifecycle

The lifecycle of the activity in which the fragment lives directly affects the lifecycle of the fragment, such that each lifecycle callback for the activity results in a similar callback for each fragment. For example, when the activity receives `onPause()`, each fragment in the activity receives `onPause()`.

Fragments have a few extra lifecycle callbacks, however, that handle unique interaction with the activity in order to perform actions such as build and destroy the fragment's UI. These additional callback methods are:

onAttach() - Called when the fragment has been associated with the activity (the **Activity** is passed in here and this is the best place to make him your callBack).

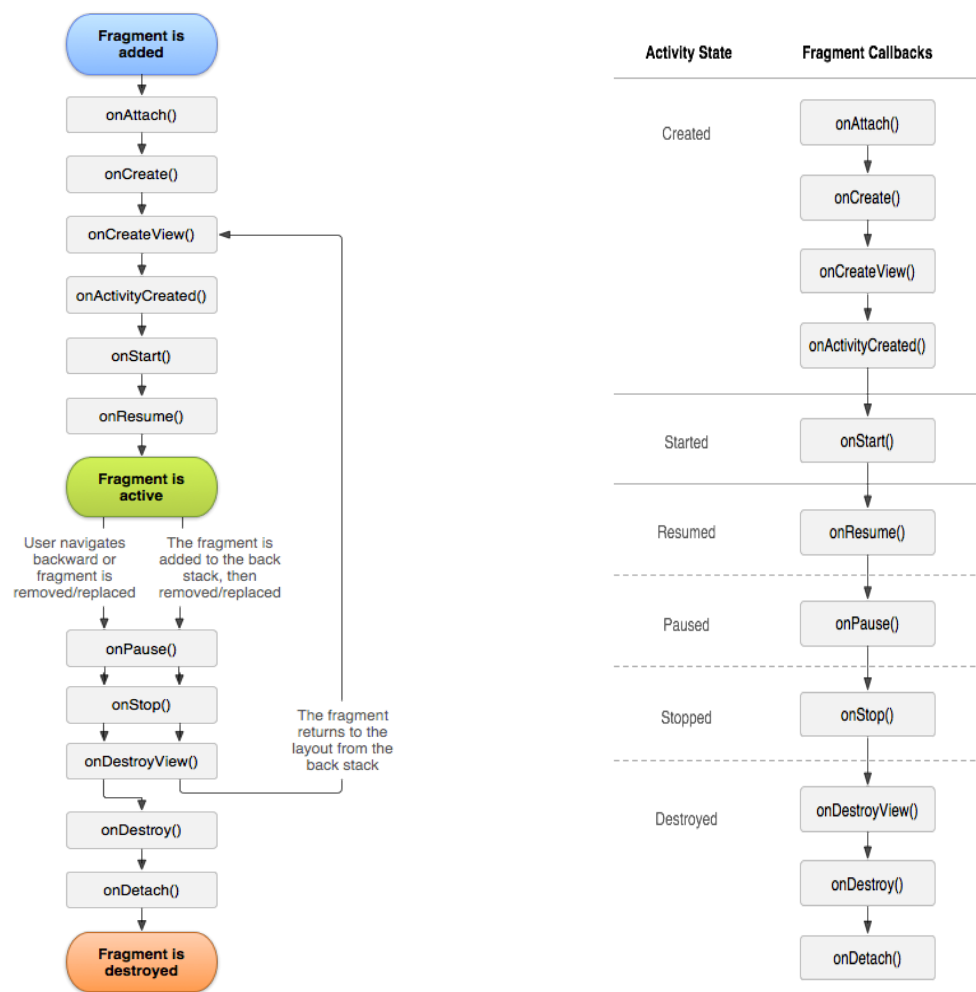
onCreateView() -Called to create the view hierarchy associated with the fragment.

onActivityCreated() -Called when the activity's **onCreate()** method has returned.

onDestroyView() - Called when the view hierarchy associated with the fragment is being removed.

onDetach() - Called when the fragment is being disassociated from the activity. the getActivity() function here will returned null.

Once the activity reaches the resumed state, you can freely add and remove fragments to the activity. Thus, only while the activity is in the resumed state can the lifecycle of a fragment change independently.



Reference: [Fragments in Android Developers](#)