Fragments in Android

Activity's Little Helpers

A Fragment

A fragment is a class implementing a portion of an activity.

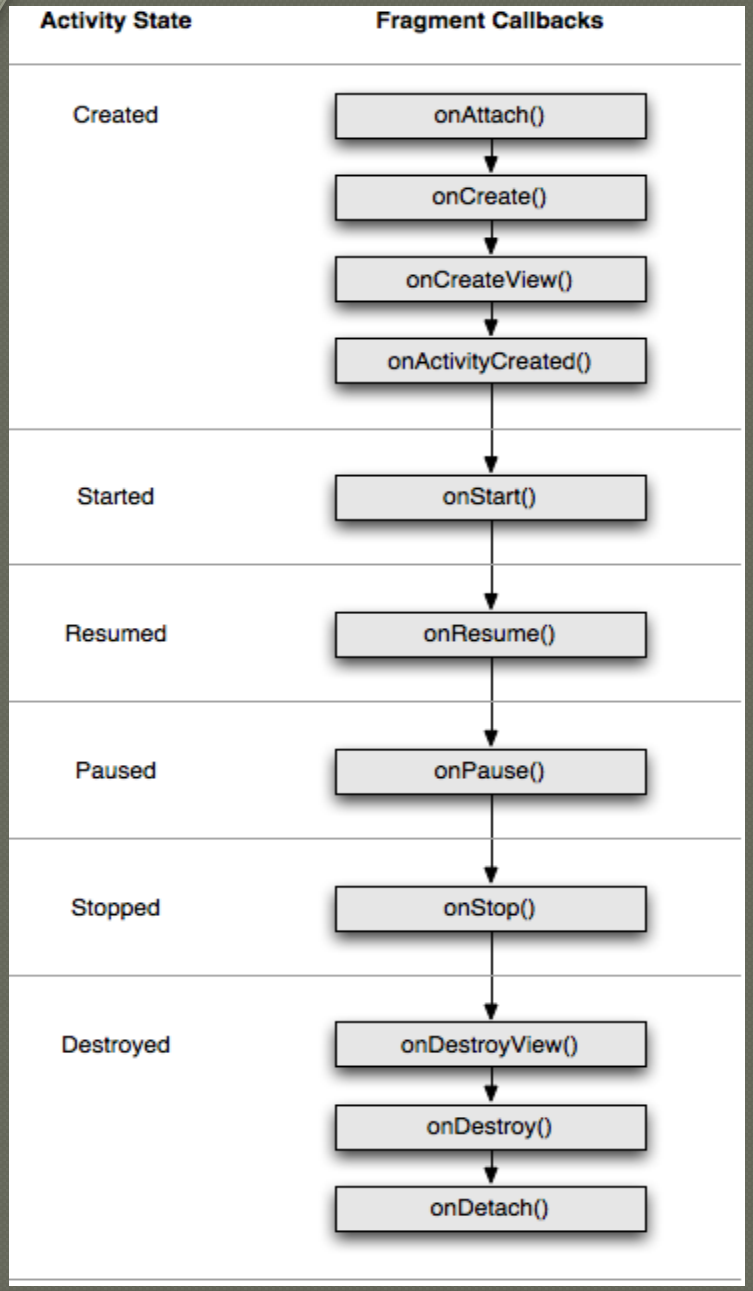
• A fragment represents a particular operation or interface running within a larger activity.
• Fragments enable more modular activity design, making it easier to adapt an application to different screen orientations
and multiple screen sizes.
• Fragments must be embedded in activities; they cannot run independent of activities.
• Most fragments define their own layout of views that live within the activity's view hierarchy.
– However, a fragment can implement a behavior that has no user interface component.
• A fragment has its own lifecycle, closely related to the lifecycle of its host activity.
• A fragment can be a static part of an activity, instantiated automatically during the activity's creation.
• Or, you can create, add, and remove fragments dynamically in an activity at run-time.

Fragments were added to the Android API in Honeycomb, API 11.

The primary classes related to fragments are:
android.app.Fragment
The base class for all fragment definitions

android.app.FragmentManager
The class for interacting with fragment objects inside an activity

android.app.FragmentTransaction
The class for performing an atomic set of fragment operations



Called when the fragment has been associated with the activity.

Called to create the view hierarchy associated with the fragment.

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

Called when the view hierarchy associated with the fragment is being removed.

Called when the fragment is being disassociated from the activity.

Creating the Fragment Class

Each fragment must be implemented as a subclass of Fragment.

Many of the Fragment methods are analogous to those found in Activity, and you should use them in a similar fashion.

onCreate(Bundle)
Initialize resources used by your fragment except those related to the user interface.
onCreateView(LayoutInflater, ViewGroup, Bundle)
Create and return the view hierarchy associated with the fragment.
onResume()
Allocate “expensive” resources (in terms of battery life, monetary cost, etc.), such as registering for location updates, sensor updates, etc.
onPause()
Release “expensive” resources. Commit any changes that should be persisted beyond the current user session.

Fragment Layout

To provide a layout for a fragment, your fragment's class must implement the onCreateView() callback method.

• The Android system invokes this method when it's time for the fragment to create its layout.

• This method must return a View that is the root of your fragment's layout. As with an activity, you can create your layout programmatically by directly instantiating and configuring view objects, or declaratively by providing an XML layout file and inflating the layout.

• The declarative approach usually is simpler and easier.

• In support of the declarative approach, the system provides a reference to a LayoutInflater and a ViewGroup from the activity's layout, which will serve as the parent of your fragment's layout.

Creating a Fragment Layout (example)

```
public class FirstFragment extends Fragment implements OnClickListener {

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
View view = inflater.inflate(R.layout.first_fragment, container, false);
 Button nextButton = (Button) view.findViewById(R.id.button_first);
 nextButton.setOnClickListener(this);
 return view;
 }
 // ...
 }
```

Note

The final false argument to LayoutInflator.inflate() prevents the inflator from automatically attaching the inflated view hierarchy to the parent container. This is important, because the activity automatically attaches the view hierarchy to the parent as appropriate.

“Statically” Including a Fragment in an Activity Layout

The easiest way to incorporate a fragment into an activity is by including it directly into the activity’s layout file.

- This works well if the fragment should always be present in the layout.
- However, this approach does not allow you to dynamically remove the fragment at run-time.

In the activity’s layout file, simply use the <fragment> element (yes, that’s really lowercase) where you want to include the fragment.

- Use the android:name attribute to provide the package-qualified class name of the fragment.
- Specify the layout attributes to control the size and position of the fragment.

For example: **Activity Layout**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="horizontal"
android:layout_width="match_parent"
android:layout_height="match_parent">
<fragment android:name="com.example.news.ListFragment"
            android:id="@+id/list"
            android:layout_weight="1"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
<fragment android:name="com.example.news.ReaderFragment"
            android:id="@+id/viewer"
            android:layout_weight="2"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
</LinearLayout>
```

Dynamically Adding a Fragment to an Activity

At any time while your activity is running, you can add fragments to your activity layout.

1. First, use Activity.getFragmentManager() to get a reference to the FragmentManager.
2. Invoke FragmentManager.beginTransaction() to get an instance of FragmentTransaction.
3. Instantiate an instance of your fragment.
4. Use the FragmentTransaction.add() to add the fragment to a ViewGroup in the activity, specified by its
ID. Optionally, you can also provide a String tag to identify the fragment.
5. Commit the transaction using FragmentTransaction.commit().

For example:

```
FragmentManager fragmentManager = getFragmentManager()
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
ExampleFragment fragment = new ExampleFragment();
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.commit();
```

Fragment Operations

You can perform many other operations on dynamic fragments other than adding them to an activity, such as removing them and changing their visibility.

• Each set of changes that you commit to the activity is called a transaction.
• You perform fragment operations using the methods in the FragmentTransaction class. Methods include:

add()
Add a fragment to the activity.
remove()
Remove a fragment from the activity. This operation destroys the fragment instance unless the transaction is added to the transaction back stack, described later.
replace()
Remove one fragment from the UI and replace it with another.
hide()
Hide a fragment in the UI (set its visibility to hidden without destroying the view hierarchy).
show()
Show a previously hidden fragment.
detach() (API 13)
Detach a fragment from the UI, destroying its view hierarchy but retaining the fragment instance.
attach() (API 13)
Reattach a fragment that has previously been detached from the UI, re-creating its view hierarchy.

Important
You can't remove(), replace(), detach(), or attach() a "static" fragment declared in an activity's Layout.

Performing Fragment Transactions

To perform a fragment transaction:

1. Obtain an instance of FragmentTransaction by calling FragmentManager.beginTransaction().
2. Perform any number of fragment operations using the transaction instance.
3. Call commit() to apply the transaction to the activity.

Important

You can commit a transaction using commit() only prior to the activity saving its state (i.e., before the system invokes Activity.onSaveInstanceState()). If you attempt to commit after that point, an exception will be thrown.

For example:

```
FragmentManager fragmentManager = getFragmentManager()
// Or: FragmentManager fragmentManager = getSupportFragmentManager()
fragmentManager.beginTransaction()
.remove(fragment1)
.add(R.id.fragment_container, fragment2)
.show(fragment3)
.hide(fragment4)
.commit();
```

The order in which you add changes to a FragmentTransaction doesn't matter, except:

- You must call commit() last.
- If you're adding multiple fragments to the same container, then the order in which you add them determines the order they appear in the view hierarchy.

Managing the Fragment Back Stack

Similar to the way the system automatically maintains a task back stack for activities, you have the option of saving fragment transactions onto a back stack managed by the activity.

• If you add fragment transactions to the back stack, then the user can navigate backward through the fragment changes by pressing the device's Back button.

• Once all fragment transactions have been removed from the back stack, pressing the Back button again destroys the activity.
To add a transaction to the back stack, invoke FragmentTransaction.addToBackStack(String) before committing the transaction.

• The String argument is an optional name to identify the back stack state, or null. The FragmentManager class has a popBackStack() method, which can return to a previous back stack state given its name.

• If you add multiple changes to the transaction and call addToBackStack(), then all changes applied before you call commit() are added to the back stack as a single transaction and the Back button will reverse them all together.

If you call addToBackStack() when removing or replacing a fragment:
• The system invokes onPause(), onStop(), and onDestroyView() on the fragment when it is placed on the back stack
• If the user navigates back, the system invokes onCreateView(), onActivityCreated(), onStart(), and onResume() on the fragment.