**Contents**

|  |  |  |
| --- | --- | --- |
| 1 | So, What’s a Fragment? | 1 |
| 2 | Fragments: Implemented in Honeycomb (3.0) or Later | 1 |
| 3 | Fragments: Implemented in Donut (1.6) or Later | 1 |
| 4 | Downloading the Compatibility Package | 2 |
| 5 | Using the Compatibility Package | 2 |
| 6 | Fragment Lifecycle | 3 |
| 7 | Creating the Fragment Class | 4 |
| 8 | Creating a Fragment Layout | 5 |
| 9 | Creating a Fragment Layout (example) | 5 |
| 10 | “Statically” Including a Fragment in an Activity Layout | 6 |
| 11 | Dynamically Adding a Fragment to an Activity | 7 |
| 12 | Handling Run-Time Configuration Changes | 7 |
| 13 | Saving Fragment State | 8 |
| 14 | Retaining Fragments Across Activity Re-Creation | 8 |
| 15 | Using Fragments with no Layouts | 9 |
| 16 | Finding Fragments | 9 |
| 17 | Fragment Operations | 10 |
| 18 | Performing Fragment Transactions | 10 |
| 19 | Managing the Fragment Back Stack | 11 |
| 20 | Integrating Fragment Action Bar/Options Menu Items | 12 |
| 21 | Integrating Fragment Action Bar/Options Menu Items (example) | 12 |

|  |  |  |
| --- | --- | --- |
| 22 | Communication Between the Fragment and the Activity | 13 |
| 23 | Best Practices: Loose Coupling of Activities and Fragments | 14 |
| 24 | Best Practices: Define Fragment Interfaces to Invoke Activity Behavior | 14 |
| 25 | Best Practices: The Activity as a Switchboard | 15 |
| 26 | Advanced Fragment Initialization | 15 |
| 27 | Advanced Fragment Initialization: Arguments | 15 |
| 28 | Implementing Dialogs Using Fragments | 16 |
| 29 | Using a Fragment-Based Dialog | 17 |
| 30 | Example: A Simple Confirmation Dialog Fragment | 17 |
| 31 | Additional Fragment Subclasses | 19 |

**1 So, What’s 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.

**2 Fragments: Implemented in Honeycomb (3.0) or Later**

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

**3 Fragments: Implemented in Donut (1.6) or Later**

Google provides the Compatibility Package, a Java library that you can include in an application, implementing support for fragments and other Honeycomb features (loaders).

• You can use the Compatibility Package with applications targeting 1.6 (API 4) or later.

• For each of the classes in the Compatibility Package, the APIs work almost exactly the same as their counterparts in the latest Android platform. Therefore, you can usually refer to the online documentation for information about the supported APIs.

**Note**

If you write an application using the Compatibility Package, your application uses the class definitions from the Com- patibility Package even when running on a device with 3.0 or later installed. The package does not try to switch to the device’s “native” implementation of the classes.

The primary classes related to fragments are:

**android.support.v4.app.FragmentActivity**

The base class for all activities using compatibility-based fragment (and loader) features

**android.support.v4.app.Fragment**

The base class for all fragment definitions

**android.support.v4.app.FragmentManager**

The class for interacting with fragment objects inside an activity

**android.support.v4.app.FragmentTransaction**

The class for performing an atomic set of fragment operations

**Important**

To use the Compatibility Package features, your activity must use android.support.v4.app.FragmentActivity

as a base class, rather than android.app.Activity or one of its subclasses.

**4 Downloading the Compatibility Package**

You can download and install the Compatibility Package using the Android SDK and AVD Manager tool.

• You can find the Compatibility Package for download under "Android Repository" as "Android Compatibility package".

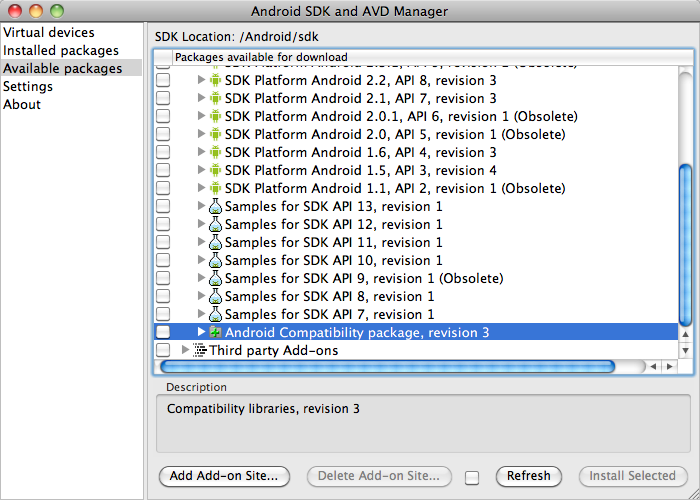


Figure 1: Installing the Compatibility Package

**5 Using the Compatibility Package**

To use the Compatibility Package, you must add it to your Android project and include it in your Build Path.

• Using the Eclipse ADT plugin, you can do this automatically by:

1. Selecting your Android project in the Package Explorer view

2. Bringing up the context-menu

3. Selecting Android Tools → Add Compatibility Library

**6 Fragment Lifecycle**

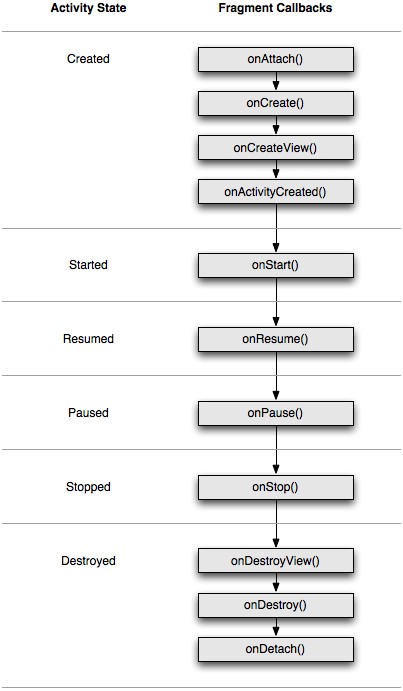


Figure 2: Fragment Lifecycle

As with the Activity class, the Fragment base class is an example of the template method design pattern.

• The Fragment base class defines a set of methods that you override to provide the custom behavior of your fragment

implementation.

Of primary importance are the fragment lifecycle callback methods.

• The lifecycle of the activity in which the fragment lives directly affects the lifecycle of the fragment.

• 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 managing interaction with the activity:

**onAttach(Activity)**

Called when the fragment has been associated with the activity.

**onCreateView(LayoutInflater, ViewGroup, Bundle)**

Called to create the view hierarchy associated with the fragment.

**onActivityCreated(Bundle)**

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.

**Important**

When overriding these lifecycle methods — with the exception of onCreateView() — you **must** call through to the super class’s implementation of the method. Otherwise, an exception occurs at run-time.

**7 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.

**8 Creating a 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.

**9 Creating a Fragment Layout (example)**

For example, you could define a simple layout resource such as this:

res/layout/first\_fragment.xml

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android=["http://schemas.android.com/apk/res/android](http://schemas.android.com/apk/res/android)" android:orientation="vertical"

android:layout\_width="match\_parent" android:layout\_height="match\_parent" android:id=["@+id](mailto:@+id)/first\_fragment\_root">

<TextView android:layout\_height="wrap\_content" android:text=["@string](mailto:@string)/text\_first\_fragment\_title" android:layout\_width="match\_parent" android:gravity="center\_horizontal" />

<EditText android:layout\_height="wrap\_content" android:layout\_width="match\_parent" android:id=["@+id](mailto:@+id)/edit\_first\_msg" />

<Button android:layout\_height="wrap\_content" android:layout\_gravity="center\_horizontal" android:layout\_width="wrap\_content" android:id=["@+id](mailto:@+id)/button\_first" android:text=["@string](mailto:@string)/button\_first\_text" />

</LinearLayout>

Then to initialize your fragment’s layout:

public class FirstFragment extends Fragment implements OnClickListener {

[@Override](mailto:@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.

**10 “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.

**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). There are three ways to provide an ID for a fragment:

• Supply the android:id attribute with a unique ID.

• Supply the android:tag attribute with a unique string.

• If you provide neither of the previous two, the system uses the ID of the fragment’s container view.

For example:

Activity Layout

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android=["http://schemas.android.com/apk/res/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.ArticleListFragment" android:id=["@+id](mailto:@+id)/list"

android:layout\_weight="1" android:layout\_width="0dp" android:layout\_height="match\_parent" />

<fragment android:name="com.example.news.ArticleReaderFragment" android:id=["@+id](mailto:@+id)/viewer"

android:layout\_weight="2" android:layout\_width="0dp" android:layout\_height="match\_parent" />

</LinearLayout>

**11 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.

**Note**

If you’re using the Compatibility Package, use FragmentActivity.getSupportFragmentManager().

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()

// Or: FragmentManager fragmentManager = getSupportFragmentManager() FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction(); ExampleFragment fragment = new ExampleFragment(); fragmentTransaction.add(R.id.fragment\_container, fragment); fragmentTransaction.commit();

But. . . There’s a catch.

**12 Handling Run-Time Configuration Changes**

When the system destroys and re-creates an activity because of a run-time configuration change, the activity automatically re-instantiates existing fragments.

• This isn’t a problem for “static” fragments declared in the activity’s layout.

• But for “dynamic” fragments, you need to test for this situation to prevent creating a second instance of your fragment.

To test whether the system is re-creating the activity, check whether the Bundle argument passed to your activity’s

onCreate() is null.

• It it is non-null, the system is re-creating the activity. In this case, the activity automatically re-instantiates existing fragments.

• If it null you can safely instantiate your dynamic fragment. For example:

public void onCreate(Bundle savedInstanceState) {

// ...

if (savedInstanceState != null) {

FragmentManager fragmentManager = getFragmentManager()

// Or: FragmentManager fragmentManager = getSupportFragmentManager() FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction(); ExampleFragment fragment = new ExampleFragment(); fragmentTransaction.add(R.id.fragment\_container, fragment); fragmentTransaction.commit();

}

}

**13 Saving Fragment State**

The Fragment class supports the onSaveInstanceState(Bundle) method (but not the onRestoreInstanceState()

method) in much the same way as the Activity class.

• The default implementation saves the state of all the fragment’s views that have IDs.

• You can override this method to store additional fragment state information.

• If the system is re-creating the fragment from a previous saved state, it provides a reference to the Bundle contain- ing that state to the onCreate(), onCreateView(), and onActivityCreated() methods; otherwise, the argument is set to null.

**14 Retaining Fragments Across Activity Re-Creation**

By default, when an activity is re-created (such as in response to a run-time configuration change), its fragments are also destroyed and re-created automatically.

Executing Fragment.setRetainInstance(boolean) with a value of true requests the system to retain the current instance of the fragment if the activity is re-created.

• If set, when the activity is re-created, the fragment’s onDestroy() and onCreate() methods are not invoked.

• All other fragment lifecycle methods are invoked in their typical sequence.

**Note**

If you use setRetainInstance(true), then the Bundle argument to onCreateView() and and

onActivityCreated() is null because the fragment is not re-created.

“Retained” fragments can be quite useful for propagating state information — especially thread management — across activity instances.

• For example, a fragment can serve as a “host” for an instance of Thread or AsyncTask, managing its operation.

• The Activity.onRetainNonConfigurationInstance() method, which has traditionally been used for this purpose, is deprecated as of API 11 in favor of the fragment’s setRetainInstance() capability.

**Important**

If you are using the compatibility library, the FragmentActivity class overrides onRetainNonConfigurationInstance() making it **final** to implement the fragment (and loader) retention capability.

**15 Using Fragments with no Layouts**

A fragment is not required to have a user interface.

• For example, if the sole purpose of a fragment is to maintain state information or manage a thread, it might not need a user interface.

When using a fragment with no user interface:

• There is no need to override the onCreateView() method.

• You must add the fragment to the activity using FragmentTransaction.add(Fragment, String), provid- ing a unique String tag to identify the fragment.

• For example:

FragmentManager fragmentManager = getFragmentManager()

// Or: FragmentManager fragmentManager = getSupportFragmentManager() FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction(); BackgroundFragment fragment = new BackgroundFragment(); fragmentTransaction.add(fragment, "thread\_manager"); fragmentTransaction.commit();

**16 Finding Fragments**

The FragmentManager class has methods for finding a fragment contained within an activity:

**findFragmentById(int id)**

Finds a fragment with the specified ID.

**findFragmentByTag(String tag)**

Finds a fragment with the specified tag.

Both of these methods return a reference to the fragment, or null if no matching fragment is found.

**17 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.

**Note**

Revision 3 and later of the Compatibility Package also supports the attach() and detach() methods.

**18 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.

**Tip**

Most of the FragmentTransaction operations return a reference to the same FragmentTransaction

instance, allowing method chaining.

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.

**19 Managing the Fragment Back Stack**

Similar to the way the system automatically mains 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 com- mitting 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.

**20 Integrating Fragment Action Bar/Options Menu Items**

A fragment can add its own items to the activity’s action bar or options menu.

• First, you must invoke Fragment.setHasOptionsMenu() in the fragment’s onCreate() method.

• Then you can provide definitions for the fragment’s onCreateOptionsMenu() and onOptionsItemSelected() methods, and optionally onPrepareOptionsMenu(), onOptionsMenuClosed(), and onDestroyOptionsMenu() methods, if needed.

• You implement these methods exactly as if they appeared within an activity class definition.

The corresponding methods in the Activity base class (or FragmentActivity base class) automatically invoke these methods for each fragment it contains.

• If you override these methods in your activity class, you can processing any menu items managed directly by the activity.

• Then, if appropriate, invoke the super version of the method to propagate the call to the fragments within the activity. Don’t forget to do this, or your fragment’s items will be ignored!

The activity automatically updates the action bar/options menu based on the dynamic state of its fragments.

• Items are added if needed when fragments are added, shown, or attached.

• Items are removed if no longer needed when fragments are removed, hidden, or detached.

**Note**

A fragment that adds items to the options menu does **not** need to have a view hierarchy. For example, a fragment could add Start and Stop items to the menu to control its behavior, but not display any other user interface.

**21 Integrating Fragment Action Bar/Options Menu Items (example)**

Activity source code

public class MyActivity extends Activity {

// ...

[@Override](mailto:@Override)

public boolean onCreateOptionsMenu(Menu menu) { super.onCreateOptionsMenu(menu); getMenuInflater().inflate(R.menu.activity\_options, menu); return true;

}

[@Override](mailto:@Override)

public boolean onOptionsItemSelected(MenuItem item) {

switch (item.getItemId()) {

case R.id.menu\_activity\_info:

// Handle activity menu item return true;

default:

// Handle fragment menu items

return super.onOptionsItemSelected(item);

}

}

// ...

}

Fragment source code

public class MyFragment extends Fragment {

// ...

[@Override](mailto:@Override)

public void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); setHasOptionsMenu(true);

}

[@Override](mailto:@Override)

public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {

inflater.inflate(R.menu.myfragment\_options, menu);

}

[@Override](mailto:@Override)

public boolean onOptionsItemSelected(MenuItem item) {

switch (item.getItemId()) {

case R.id.menu\_first\_info:

// Handle fragment menu item return true;

default:

// Not one of ours. Perform default menu processing return super.onOptionsItemSelected(item);

}

}

// ...

}

**22 Communication Between the Fragment and the Activity**

A given instance of a fragment is directly tied to the activity that contains it.

• Your activity can call public methods in the fragment through a reference to the fragment object.

– If you don’t retain a reference to a fragment when creating it, you can find it with the FragmentManager, using

findFragmentById() or findFragmentByTag().

• The fragment can access its containing activity instance with the Fragment.getActivity() method.

– For example, you could obtain a reference to a view in the activity’s layout:

View listView = getActivity().findViewById(R.id.list);

**Note**

The Fragment class is not derived from Context. If your fragment needs a Context object, either use the activity or — especially if the Context might be retained after activity destruction — invoke getApplicationContext() on the hosting activity.

**23 Best Practices: Loose Coupling of Activities and Fragments**

Avoid tight coupling between activities and fragments.

• Especially if you make the fragment “too aware” of its hosting activity and the other fragments that might (or might not) be present in the activity, you increase its complexity and reduce its reuse.

• On the other hand, activities typical need more awareness of the fragments they and other activities host.

**24 Best Practices: Define Fragment Interfaces to Invoke Activity Behavior**

Define interfaces in fragments to call back to their hosting activities.

• Rather than having a fragment directly access the internals of a hosting activity, create a nested interface within the fragment defining a set of callback methods.

• The hosting activity can then implement the interface, or you can provide another listener object that implements the interface.

• If you plan always to use the activity as an event listener, you can automatically “register” it in the fragment’s onAttach()

method. Otherwise, you can provide a listener registration method.

• For example:

Fragment source code

public class TitlesFragment extends ListFragment {

private OnTitleSelectedListener listener;

public interface OnTitleSelectedListener {

public void onTitleSelected(int index);

}

public void onAttach(Activity activity) {

super.onAttach(activity);

try {

showSecondFragmentListener

= (OnTitleSelectedListener) activity;

} catch (ClassCastException e) {

throw new ClassCastException(activity.toString()

+ " must implement OnTitleSelectedListener");

}

}

[@Override](mailto:@Override)

public void onListItemClick(ListView l, View v,

int position, long id) {

listener.onTitleSelected(position);

}

// ...

}

Activity source code

public class MyActivity extends Activity

implements TitlesFragment.OnTitleSelectedListener {

public void onTitleSelected(int index) {

// ...

}

// ...

}

**25 Best Practices: The Activity as a Switchboard**

Let the activity serve as an intermediary between fragments.

• Fragments don’t have intent filters; an Intent can’t directly trigger a fragment.

• Let the activity respond to intents and fragment callbacks.

• The activity will know if it can “route” a event to a fragment it contains, or if it needs to start another activity to handle the event.

**26 Advanced Fragment Initialization**

Every fragment must have a default empty constructor.

• The system invokes the default constructor to re-instantiate the fragment when restoring its activity’s state. Fragments generally should not implement additional constructors or override the default constructor.

• The first place application code can run where the fragment is ready to be used is in onAttach().

**27 Advanced Fragment Initialization: Arguments**

• Immediately after instantiating a fragment, can can provide it with a set of arguments.

– Create a Bundle object, and put in it any values to want to provide as arguments to the fragment.

– Invoke setArguments(Bundle) on the fragment, passing the Bundle.

• Within the fragment, invoke getArguments() when you’re ready to retrieve the Bundle of arguments.

• Often, especially in the case of a fragment supporting only one or two arguments, it’s easiest to define a static factory method to instantiate the fragment and supply the arguments.

Fragment source code

public static class DetailsFragment extends Fragment {

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;

}

public int getShownIndex() {

return getArguments().getInt("index", 0);

}

// ...

}

Activity source code

DetailsFragment detail = DetailsFragment.newInstance(2);

**28 Implementing Dialogs Using Fragments**

As of Honeycomb (API 11), the Activity class’s “managed dialog” methods are deprecated in favor of fragments. The DialogFragment class serves as a base class for fragment-based dialog management.

• DialogFragment implements a fragment that displays a dialog window, floating on top of its activity’s window.

• This fragment contains a Dialog object, which it displays as appropriate based on the fragment’s state.

• You should control dialog (showing and dismissing it) through the DialogFragment methods, not with direct calls on the dialog.

You can implement a DialogFragment so that it can be used only as a dialog, or so that it can be displayed as either a dialog or as a “normal” fragment managed by a ViewGroup within the activity.

• If you plan to use the fragment only as a dialog, override the onCreateDialog() method and return an instance of

Dialog or one of its subclasses.

• If you want to use the fragment as either a dialog or a normal fragment, override onCreateView() and return a view hierarchy.

**29 Using a Fragment-Based Dialog**

The DialogFragment class provides an overloaded show() method, which posts the dialog as part of a transaction.

• If provided with a FragmentManager reference, the show() method creates a transaction, adds the fragment, and then commits. When the fragment is dismissed, a new transaction is executed automatically to remove it from the activity.

• If provided with a FragmentTransaction reference, the show() method adds the fragment and commits the transaction. In this case, you have the option of adding the transaction to the back stack prior to invoking show()

• In either version, the second argument is a String tag to identify the dialog fragment.

The DialogFragment class provides a dismiss() method to dismiss the fragment explicitly.

• If the fragment was added to the back stack, all back stack state up to and including this entry is popped. Otherwise, a new transaction is committed to remove the fragment.

• In the typical use case, the fragment dismisses itself with this method when its dialog is dismissed.

**30 Example: A Simple Confirmation Dialog Fragment**

Fragment source code

public class ConfirmationDialogFragment extends DialogFragment implements DialogInterface.OnClickListener {

private ConfirmationDialogFragmentListener listener;

public static ConfirmationDialogFragment newInstance(int title) { ConfirmationDialogFragment frag = new ConfirmationDialogFragment(); Bundle args = new Bundle();

args.putInt("title", title); frag.setArguments(args); return frag;

}

public interface ConfirmationDialogFragmentListener {

public void onPositiveClick();

public void onNegativeClick();

}

public void setConfirmationDialogFragmentListener( ConfirmationDialogFragmentListener listener) {

this.listener = listener;

}

[@Override](mailto:@Override)

public Dialog onCreateDialog(Bundle savedInstanceState) {

int title = getArguments().getInt("title");

return new AlertDialog.Builder(getActivity())

.setIcon(android.R.drawable.ic\_dialog\_alert)

.setTitle(title)

.setPositiveButton(android.R.string.ok, this)

.setNegativeButton(android.R.string.cancel, this)

.create();

}

[@Override](mailto:@Override)

public void onClick(DialogInterface dialog, int which) {

if (listener != null) {

switch (which) {

case DialogInterface.BUTTON\_POSITIVE:

listener.onPositiveClick();

default:

listener.onNegativeClick();

}

}

}

}

Activity source code

public class SimpleConfirmationDialogFragmentActivity extends FragmentActivity

implements OnClickListener, ConfirmationDialogFragmentListener {

[@Override](mailto:@Override)

public void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); setContentView(R.layout.main);

Button buttonPostDialog = (Button) findViewById(R.id.button\_post\_dialog);

buttonPostDialog.setOnClickListener(this);

}

[@Override](mailto:@Override)

public void onClick(View v) { ConfirmationDialogFragment confirmationDialog

= ConfirmationDialogFragment.newInstance(R.string.dialog\_format\_title); confirmationDialog.setConfirmationDialogFragmentListener(this); confirmationDialog.show(getSupportFragmentManager(), null);

}

[@Override](mailto:@Override)

public void onPositiveClick() {

Toast.makeText(this, android.R.string.ok, Toast.LENGTH\_LONG).show();

}

[@Override](mailto:@Override)

public void onNegativeClick() {

Toast.makeText(this, android.R.string.cancel, Toast.LENGTH\_LONG).show();

}

}

**31 Additional Fragment Subclasses**

There are some additional subclasses of Fragment designed for common uses:

**ListFragment**

A fragment that automatically manages a ListView. Analogous to the ListActivity class.

**PreferenceFragment**

A fragment that automatically manages a set of Preference objects. Using an XML preference resource, the fragment can automatically create an interface for displaying and editing a set of preferences. Analogous to the PreferenceActivity class.

**WebFragment**

A fragment that automatically creates and manages a WebView.

**Advantages**

Fragments are reusable component.

Cant exist without Activity class.

Will be affected by Activity life cycle.

Ex: If an Activity is paused All the fragments in it are destroyed.

Need not to register in manifest file.

Manually needs to add in addToBackStack().

0 or many fragments can place in single Activity.

One Fragment can be placed in any number of Activities.

**Differences between Activity and Fragment**

Activity Fragment

Must be Register in manifest file

Can exist without fragment also

Can exist independently

Contains 0 or more fragments

Does depend on the life cycle of fragments