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.

This document describes how to build your application to use fragments, including how fragments can maintain their state when added to the activity's back stack, share events with the activity and other fragments in the activity, contribute to the activity's action bar, and more.

Design Philosophy

Android introduced fragments in Android 3.0 (API level 11), primarily to support more dynamic and flexible UI designs on large screens, such as tablets. Because a tablet's screen is much larger than that of a handset, there's more room to combine and interchange UI components. Fragments allow such designs without the need for you to manage complex changes to the view hierarchy. By dividing the layout of an activity into fragments, you become able to modify the activity's appearance at runtime and preserve those changes in a back stack that's managed by the activity.

For example, 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. Thus, instead of using one activity to select an article and another activity to read the article, the user can select an article and read it all within the same activity, as illustrated in the tablet layout in figure 1.

You should design each fragment as a modular and reusable activity component. That is, because each fragment defines its own layout and its own behavior with its own lifecycle callbacks, you can include one fragment in multiple activities, so you should design for reuse and avoid directly manipulating one fragment from another fragment. This is especially important because a modular fragment allows you to change your fragment combinations for different screen sizes. When designing your application to support both tablets and handsets, you can reuse your fragments in different layout configurations to optimize the user experience based on

the available screen space. For example, on a handset, it might be necessary to separate fragments to provide a single-pane UI when more than one cannot fit within the same activity.

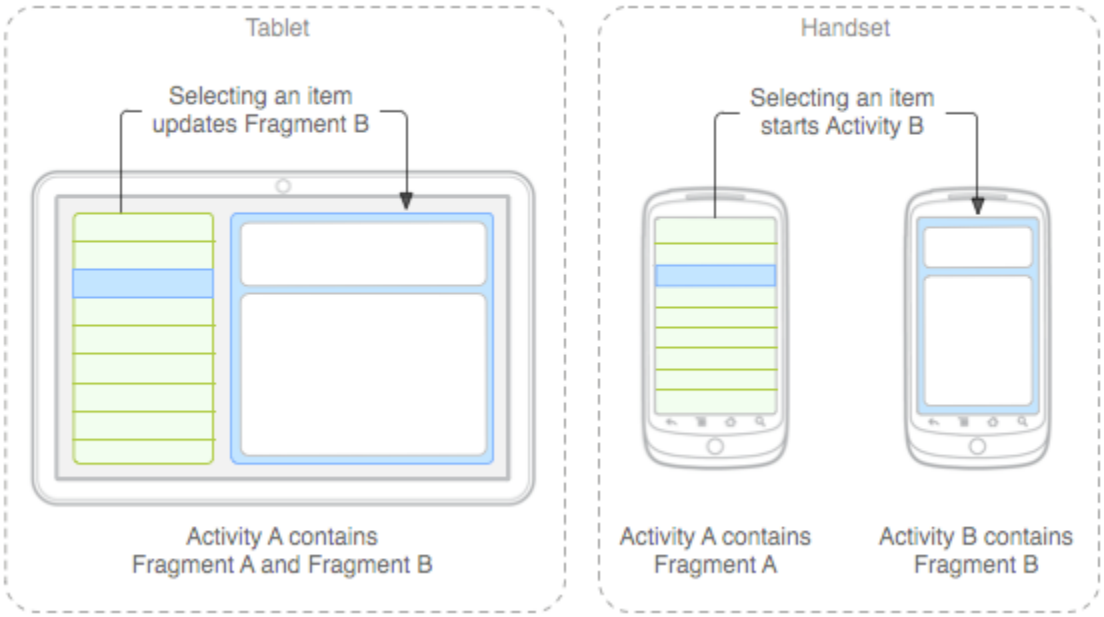

Figure 1. An example of how two UI modules defined by fragments can be combined into one activity for a tablet design, but separated for a handset design.

For example—to continue with the news application example—the application can embed two fragments in *Activity A*, when running on a tablet-sized device. However, on a handset-sized screen, there's not enough room for both fragments, so *Activity A* includes only the fragment for the list of articles, and when the user selects an article, it starts *Activity B*, which includes the second fragment to read the article. Thus, the application supports both tablets and handsets by reusing fragments in different combinations, as illustrated in figure 1.

For more information about designing your application with different fragment combinations for different screen configurations, see the guide to Supporting Tablets and Handsets.

Creating a Fragment

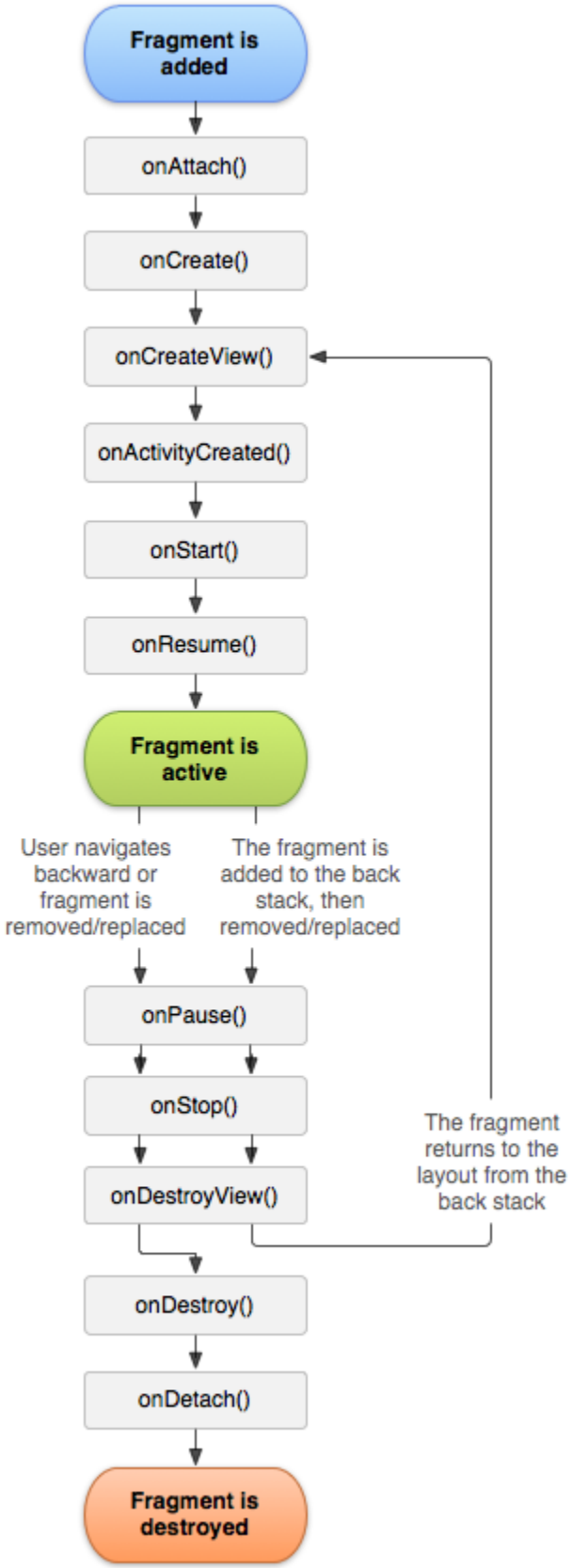
Fragment is added
onAttach()
onCreate()
onCreateView()
onActivityCreated()
onStart()
onResume()
Fragment is active
User navigates backward or fragment is removed/replaced
The fragment is added to the back stack, then removed/replaced
onPause()
onStop()
onDestroyView()
The fragment returns to the layout from the back stack
onDestroy()
onDetach()
Fragment is destroyed

Figure 2. The lifecycle of a fragment (while its activity is running).

To create a fragment, you must create a subclass of `Fragment` (or an existing subclass of it). The `Fragment` class has code that looks a lot like an `Activity`. It contains callback methods similar to an activity, such as `onCreate()`,`onStart()`, `onPause()`, and `onStop()`. In fact, if you're converting an existing Android application to use fragments, you might simply move code from your activity's callback methods into the respective callback methods of your fragment.

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

`onCreate()`

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()`

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` 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()`

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).

Most applications should implement at least these three methods for every fragment, but there are several other callback methods you should also use to handle various stages of the fragment lifecycle. All the lifecycle callback methods are discussed more later, in the section about Handling the Fragment Lifecycle.

There are also a few subclasses that you might want to extend, instead of the base `Fragment` class:

`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, because you can incorporate a fragment dialog into the back stack of fragments managed by the activity, allowing the user to return to a dismissed fragment.

`ListFragment`

Displays a list of items that are managed by an adapter (such as a `SimpleCursorAdapter`), similar to`ListActivity`. It provides several methods for managing a list view, such as the `onListItemClick()` callback to handle click events.

`PreferenceFragment`

Displays a hierarchy of `Preference` objects as a list, similar to `PreferenceActivity`. This is useful when creating a "settings" activity for your application.

Adding a user interface

A fragment is usually used as part of an activity's user interface and contributes its own layout to the activity.

To provide a layout for a fragment, you must implement the `onCreateView()` callback method, which the Android system calls when it's time for the fragment to draw its layout. Your implementation of this method must return a `View`that is the root of your fragment's layout.

> **Note:** If your fragment is a subclass of `ListFragment`, the default implementation returns a `ListView` from`onCreateView()`, so you don't need to implement it.

To return a layout from `onCreateView()`, you can inflate it from a layout resource defined in XML. To help you do so,`onCreateView()` provides a `LayoutInflater` object.

For example, here's a subclass of `Fragment` that loads a layout from the `example_fragment.xml` file:

```
public static class ExampleFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.example fragment, container, false);
    }
}
```

Creating a layout

In the sample above,`R.layout.example_fragment` is a reference to a layout resource named `example_fragment.xml`saved in the application resources. For information about how to create a layout in XML, see the User Interface documentation.

The `container` parameter passed to `onCreateView()` is the parent `ViewGroup` (from the activity's layout) in which your fragment layout will be inserted. The `savedInstanceState` parameter is a `Bundle` that provides data about the previous instance of the fragment, if the fragment is being resumed (restoring state is discussed more in the section about Handling the Fragment Lifecycle).

The `inflate()` method takes three arguments:

- The resource ID of the layout you want to inflate.
- The `ViewGroup` to be the parent of the inflated layout. Passing the `container` is important in order for the system to apply layout parameters to the root view of the inflated layout, specified by the parent view in which it's going.
- A boolean indicating whether the inflated layout should be attached to the `ViewGroup` (the second parameter) during inflation. (In this case, this is false because the system is already inserting the inflated layout into the `container`—passing true would create a redundant view group in the final layout.)

Now you've seen how to create a fragment that provides a layout. Next, you need to add the fragment to your activity.

Adding a fragment to an activity

Usually, a fragment contributes a portion of UI to the host activity, which is embedded as a part of the activity's overall view hierarchy. There are two ways you can add a fragment to the activity layout:

- **Declare the fragment inside the activity's layout file.**

In this case, you can specify layout properties for the fragment as if it were a view. For example, here's the layout file for an activity with two fragments:

```
<?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.ArticleListFragment"
            android:id="@+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/viewer"
            android:layout_weight="2"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
</LinearLayout>
```

The `android:name` attribute in the `<fragment>` specifies the `Fragment` class to instantiate in the layout.

When the system creates this activity layout, it instantiates each fragment specified in the layout and calls the `onCreateView()` method for each one, to retrieve each fragment's layout. The system inserts the `View` returned by the fragment directly in place of the `<fragment>` element.

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 container view.

- **Or, programmatically add the fragment to an existing `ViewGroup`.**

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.

To make fragment transactions in your activity (such as add, remove, or replace a fragment), you must use APIs from `FragmentTransaction`. You can get an instance of`FragmentTransaction` from your `Activity` like this:

```
FragmentManager fragmentManager = getFragmentManager()
FragmentTransaction fragmentTransaction =
fragmentManager.beginTransaction();
```

You can then add a fragment using the `add()` method, specifying the fragment to add and the view in which to insert it. For example:

```
ExampleFragment fragment = new ExampleFragment();
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.commit();
```

The first argument passed to `add()` is the `ViewGroup` in which the fragment should be placed, specified by resource ID, and the second parameter is the fragment to add.

Once you've made your changes with `FragmentTransaction`, you must call `commit()` for the changes to take effect.

Adding a fragment without a UI

The examples above show how to add a fragment to your activity in order to provide a UI. However, you can also use a fragment to provide a background behavior for the activity without presenting additional UI.

To add a fragment without a UI, add the fragment from the activity using `add(Fragment, String)` (supplying a unique string "tag" for the fragment, rather than a view ID). This adds the fragment, but, because it's not associated with a view in the activity layout, it does not receive a call to `onCreateView()`. So you don't need to implement that method.

Supplying a string tag for the fragment isn't strictly for non-UI fragments—you can also supply string tags to fragments that do have a UI—but if the fragment does not have a UI, then the string tag is the only way to identify it. If you want to get the fragment from the activity later, you need to use `findFragmentByTag()`.

For an example activity that uses a fragment as a background worker, without a UI, see the `FragmentRetainInstance.java` sample.

Managing Fragments

To manage the fragments in your activity, you need to use `FragmentManager`. To get it, call `getFragmentManager()` from your activity.

Some things that you can do with `FragmentManager` include:

- Get fragments that exist in the activity, with `findFragmentById()` (for fragments that provide a UI in the activity layout) or `findFragmentByTag()` (for fragments that do or don't provide a UI).
- Pop fragments off the back stack, with `popBackStack()` (simulating a *Back* command by the user).
- Register a listener for changes to the back stack, with `addOnBackStackChangedListener()`.

For more information about these methods and others, refer to the `FragmentManager` class documentation.

As demonstrated in the previous section, you can also use `FragmentManager` to open a `FragmentTransaction`, which allows you to perform transactions, such as add and remove fragments.

Performing Fragment Transactions

A great feature about using fragments in your activity is the ability to add, remove, replace, and perform other actions with them, in response to user interaction. Each set of changes that you commit to the activity is called a transaction and you can perform one using APIs in `FragmentTransaction`. 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 acquire an instance of `FragmentTransaction` from the `FragmentManager` like this:

```
FragmentManager fragmentManager = getFragmentManager();
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
```

Each transaction is a set of changes that you want to perform at the same time. 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()`.

Before you call `commit()`, however, you might want to call `addToBackStack()`, 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.

For example, here's how you can replace one fragment with another, and preserve the previous state in the back stack:

```
// Create new fragment and transaction
Fragment newFragment = new ExampleFragment();
FragmentTransaction transaction = getFragmentManager().beginTransaction();

// Replace whatever is in the fragment_container view with this fragment,
// and add the transaction to the back stack
transaction.replace(R.id.fragment_container, newFragment);
transaction.addToBackStack(null);

// Commit the transaction
transaction.commit();
```

In this example, `newFragment` replaces whatever fragment (if any) is currently in the layout container identified by the `R.id.fragment_container` ID. By calling`addToBackStack()`, the replace transaction is saved to the back stack so the user can reverse the transaction and bring back the previous fragment by pressing the *Back* button.

If you add multiple changes to the transaction (such as another `add()` or `remove()`) 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.

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

If you do not call `addToBackStack()` when you perform a transaction that removes a fragment, then that fragment is destroyed when the transaction is committed and the user cannot navigate back to it. Whereas, if you do call `addToBackStack()` when removing a fragment, then the fragment is *stopped* and will be resumed if the user navigates back.

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

Calling `commit()` does not perform the transaction immediately. Rather, it schedules it to run on the activity's UI thread (the "main" thread) as soon as the thread is able to do so. If necessary, however, you may call `executePendingTransactions()` from your UI thread to immediately execute transactions submitted by `commit()`. Doing so is usually not necessary unless the transaction is a dependency for jobs in other threads.

> **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 its okay that you lose the commit, use`commitAllowingStateLoss()`.

Communicating with the Activity

Although a `Fragment` is implemented as an object that's independent from an `Activity` and can be used inside multiple activities, a given instance of a fragment is directly tied to the activity that contains it.

Specifically, the fragment can access the `Activity` instance with `getActivity()` and easily perform tasks such as find a view in the activity layout:

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

Likewise, your activity can call methods in the fragment by acquiring a reference to the `Fragment` from `FragmentManager`, using `findFragmentById()` or `findFragmentByTag()`. For example:

```
ExampleFragment fragment = (ExampleFragment)
getFragmentManager().findFragmentById(R.id.example_fragment);
```

Creating event callbacks to the activity

In some cases, you might need a fragment to share events with the activity. A good way to do that is to define a callback interface inside the fragment and require that the host activity implement it. When the activity receives a callback through the interface, it can share the information with other fragments in the layout as necessary.

For example, if a news application has two fragments in an activity—one to show a list of articles (fragment A) and another to display an article (fragment B)—then fragment A must tell the activity when a list item is selected so that it can tell fragment B to display the article. In this case, the `OnArticleSelectedListener` interface is declared inside fragment A:

```
public static class FragmentA extends ListFragment {
    ...
    // Container Activity must implement this interface
    public interface OnArticleSelectedListener {
        public void onArticleSelected(Uri articleUri);
    }
    ...
}
```

Then the activity that hosts the fragment implements the `OnArticleSelectedListener` interface and overrides `onArticleSelected()` to notify fragment B of the event from fragment A. To ensure that the host activity implements this interface, fragment A's `onAttach()` callback method (which the system calls when adding the fragment to the activity) instantiates an instance of `OnArticleSelectedListener` by casting the `Activity` that is passed into `onAttach()`:

```
public static class FragmentA extends ListFragment {
    OnArticleSelectedListener mListener;
    ...
    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        try {
            mListener = (OnArticleSelectedListener) activity;
        } catch (ClassCastException e) {
            throw new ClassCastException(activity.toString() + " must implement
OnArticleSelectedListener");
        }
    }
    ...
}
```

If the activity has not implemented the interface, then the fragment throws a `ClassCastException`. On success, the `mListener` member holds a reference to activity's implementation of `OnArticleSelectedListener`, so that

fragment A can share events with the activity by calling methods defined by the `OnArticleSelectedListener` interface. For example, if fragment A is an extension of `ListFragment`, each time the user clicks a list item, the system calls `onListItemClick()` in the fragment, which then calls `onArticleSelected()` to share the event with the activity:

```
public static class FragmentA extends ListFragment {
    OnArticleSelectedListener mListener;
    ...
    @Override
    public void onListItemClick(ListView l, View v, int position, long id) {
        // Append the clicked item's row ID with the content provider Uri
        Uri noteUri = ContentUris.withAppendedId(ArticleColumns.CONTENT_URI, id);
        // Send the event and Uri to the host activity
        mListener.onArticleSelected(noteUri);
    }
    ...
}
```

The `id` parameter passed to `onListItemClick()` is the row ID of the clicked item, which the activity (or other fragment) uses to fetch the article from the application's `ContentProvider`.

More information about using a content provider is available in the Content Providers document.

Adding items to the Action Bar

Your fragments can contribute menu items to the activity's Options Menu (and, consequently, the Action Bar) by implementing `onCreateOptionsMenu()`. In order for this method to receive calls, however, you must call `setHasOptionsMenu()` during `onCreate()`, to indicate that the fragment would like to add items to the Options Menu (otherwise, the fragment will not receive a call to `onCreateOptionsMenu()`).

Any items that you then add to the Options Menu from the fragment are appended to the existing menu items. The fragment also receives callbacks to `onOptionsItemSelected()` when a menu item is selected.

You can also register a view in your fragment layout to provide a context menu by calling `registerForContextMenu()`. When the user opens the context menu, the fragment receives a call to `onCreateContextMenu()`. When the user selects an item, the fragment receives a call to `onContextItemSelected()`.

> **Note:** Although your fragment receives an on-item-selected callback for each menu item it adds, the activity is first to receive the respective callback when the user selects a menu item. If the activity's implementation of the on-item-selected callback does not handle the selected item, then the event is passed to the fragment's callback. This is true for the Options Menu and context menus.

For more information about menus, see the Menus and Action Bar developer guides.

Handling the Fragment Lifecycle

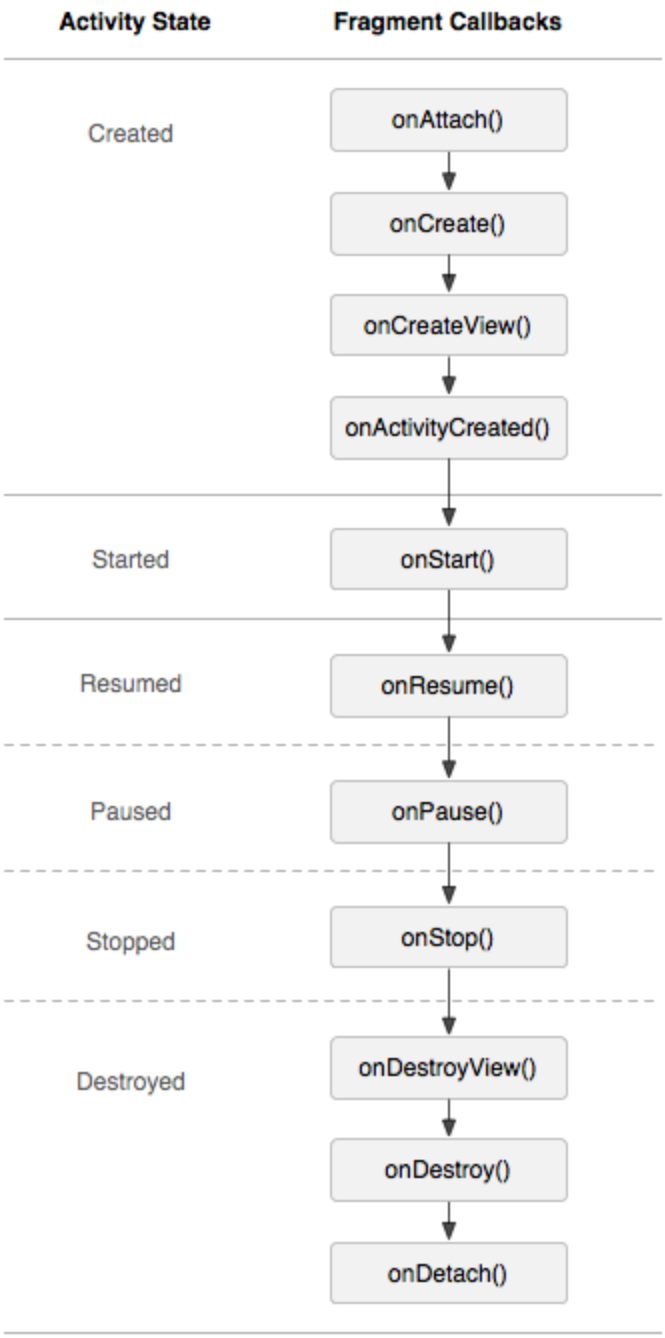

Figure 3. The activity lifecycle's affect on the fragment lifecycle.

Managing the lifecycle of a fragment is a lot like managing the lifecycle of an activity. Like an activity, a fragment can exist in three states:

Resumed

The fragment is visible in the running activity.

Paused

Another activity is in the foreground and has focus, but the activity in which this fragment lives is still visible (the foreground activity is partially transparent or doesn't cover the entire screen).

Stopped

The fragment is not visible. Either the host activity has been stopped or the fragment has been removed from the activity but added to the back stack. A stopped fragment is still alive (all state and member information is retained by the system). However, it is no longer visible to the user and will be killed if the activity is killed.

Also like an activity, you can retain the state of a fragment using a `Bundle`, in case the activity's process is killed and you need to restore the fragment state when the activity is recreated. You can save the state during the fragment's`onSaveInstanceState()` callback and restore it during either `onCreate()`, `onCreateView()`, or`onActivityCreated()`. For more information about saving state, see the Activities document.

The most significant difference in lifecycle between an activity and a fragment is how one is stored in its respective back stack. An activity is placed into a back stack of activities that's managed by the system when it's stopped, by default (so that the user can navigate back to it with the *Back* button, as discussed in Tasks and Back Stack). However, a fragment is placed into a back stack managed by the host activity only when you explicitly request that the instance be saved by calling `addToBackStack()` during a transaction that removes the fragment.

Otherwise, managing the fragment lifecycle is very similar to managing the activity lifecycle. So, the same practices for managing the activity lifecycle also apply to fragments. What you also need to understand, though, is how the life of the activity affects the life of the fragment.

Coordinating with the activity 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).

`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 flow of a fragment's lifecycle, as it is affected by its host activity, is illustrated by figure 3. In this figure, you can see how each successive state of the activity determines which callback methods a fragment may receive. For example, when the activity has received its `onCreate()` callback, a fragment in the activity receives no more than the`onActivityCreated()` callback.

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.

However, when the activity leaves the resumed state, the fragment again is pushed through its lifecycle by the activity.

Example

To bring everything discussed in this document together, here's an example of an activity using two fragments to create a two-pane layout. The activity below includes one fragment to show a list of Shakespeare play titles and another to show a summary of the play when selected from the list. It also demonstrates how to provide different configurations of the fragments, based on the screen configuration.

> **Note:** The complete source code for this activity is available in `FragmentLayout.java`.

The main activity applies a layout in the usual way, during `onCreate()`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.fragment_layout);
}
```

The layout applied is `fragment_layout.xml`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent" android:layout_height="match_parent">

    <fragment class="com.example.android.apis.app.FragmentLayout$TitlesFragment"
            android:id="@+id/titles" android:layout_weight="1"
            android:layout_width="0px" android:layout_height="match_parent" />

    <FrameLayout android:id="@+id/details" android:layout_weight="1"
            android:layout_width="0px" android:layout_height="match_parent"
            android:background="?android:attr/detailsElementBackground" />

</LinearLayout>
```

Using this layout, the system instantiates the `TitlesFragment` (which lists the play titles) as soon as the activity loads the layout, while the `FrameLayout` (where the fragment for showing the play summary will go) consumes space on the right side of the screen, but remains empty at first. As you'll see below, it's not until the user selects an item from the list that a fragment is placed into the `FrameLayout`.

However, not all screen configurations are wide enough to show both the list of plays and the summary, side by side. So, the layout above is used only for the landscape screen configuration, by saving it at `res/layout-land/fragment_layout.xml`.

Thus, when the screen is in portrait orientation, the system applies the following layout, which is saved at `res/layout/fragment_layout.xml`:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <fragment class="com.example.android.apis.app.FragmentLayout$TitlesFragment"
            android:id="@+id/titles"
            android:layout width="match parent" android:layout height="match parent" />
</FrameLayout>
```

This layout includes only `TitlesFragment`. This means that, when the device is in portrait orientation, only the list of play titles is visible. So, when the user clicks a list item in this configuration, the application will start a new activity to show the summary, instead of loading a second fragment.

Next, you can see how this is accomplished in the fragment classes. First is `TitlesFragment`, which shows the list of Shakespeare play titles. This fragment extends`ListFragment` and relies on it to handle most of the list view work.

As you inspect this code, notice that there are two possible behaviors when the user clicks a list item: depending on which of the two layouts is active, it can either create and display a new fragment to show the details in the same activity (adding the fragment to the `FrameLayout`), or start a new activity (where the fragment can be shown).

```
public static class TitlesFragment extends ListFragment {
    boolean mDualPane;
    int mCurCheckPosition = 0;

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        // Populate list with our static array of titles.
        setListAdapter(new ArrayAdapter<String>(getActivity(),
                android.R.layout.simple_list_item_activated_1, Shakespeare.TITLES));

        // Check to see if we have a frame in which to embed the details
        // fragment directly in the containing UI.
        View detailsFrame = getActivity().findViewById(R.id.details);
        mDualPane = detailsFrame != null && detailsFrame.getVisibility() == View.VISIBLE;

        if (savedInstanceState != null) {
            // Restore last state for checked position.
            mCurCheckPosition = savedInstanceState.getInt("curChoice", 0);
        }

        if (mDualPane) {
```

```
            // In dual-pane mode, the list view highlights the selected item.
            getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
            // Make sure our UI is in the correct state.
            showDetails(mCurCheckPosition);
        }
    }

    @Override
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putInt("curChoice", mCurCheckPosition);
    }

    @Override
    public void onListItemClick(ListView l, View v, int position, long id) {
        showDetails(position);
    }

    /**
     * Helper function to show the details of a selected item, either by
     * displaying a fragment in-place in the current UI, or starting a
     * whole new activity in which it is displayed.
     */
    void showDetails(int index) {
        mCurCheckPosition = index;

        if (mDualPane) {
            // We can display everything in-place with fragments, so update
            // the list to highlight the selected item and show the data.
            getListView().setItemChecked(index, true);

            // Check what fragment is currently shown, replace if needed.
            DetailsFragment details = (DetailsFragment)
                    getFragmentManager().findFragmentById(R.id.details);
            if (details == null || details.getShownIndex() != index) {
                // Make new fragment to show this selection.
                details = DetailsFragment.newInstance(index);

                // Execute a transaction, replacing any existing fragment
                // with this one inside the frame.
                FragmentTransaction ft = getFragmentManager().beginTransaction();
                ft.replace(R.id.details, details);
                ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
                ft.commit();
            }

        } else {
            // Otherwise we need to launch a new activity to display
            // the dialog fragment with selected text.
            Intent intent = new Intent();
            intent.setClass(getActivity(), DetailsActivity.class);
            intent.putExtra("index", index);
            startActivity(intent);
        }
    }
}
```

The second fragment, `DetailsFragment` shows the play summary for the item selected from the list from `TitlesFragment`:

```
public static class DetailsFragment extends Fragment {
    /**
     * Create a new instance of DetailsFragment, initialized to
     * show the text at 'index'.
     */
    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);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        if (container == null) {
            // We have different layouts, and in one of them this
            // fragment's containing frame doesn't exist.  The fragment
            // may still be created from its saved state, but there is
            // no reason to try to create its view hierarchy because it
            // won't be displayed.  Note this is not needed -- we could
            // just run the code below, where we would create and return
            // the view hierarchy; it would just never be used.
            return null;
        }

        ScrollView scroller = new ScrollView(getActivity());
        TextView text = new TextView(getActivity());
        int padding = (int)TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP,
                4, getActivity().getResources().getDisplayMetrics());
        text.setPadding(padding, padding, padding, padding);
        scroller.addView(text);
        text.setText(Shakespeare.DIALOGUE[getShownIndex()]);
        return scroller;
    }
}
```