Android Fragments

In this Lecture, you will learn:

- Fragments
- Creating Fragments
-

Fragment

A Fragment is a modular section of an activity with its own user interface. It has its own lifecycle, receives its own events, and it can be added or removed while the activity is running. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities.

A fragment is hosted in an activity. Fragment's lifecycle depends on 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 in the resumed state, you can manipulate fragments, such as add or remove them. With such a fragment operation, you can add it to a back stack managed by the activity. Each back stack entry is a record of the fragment transaction. The back stack allows the user to reverse a fragment transaction (navigate backwards), by pressing the Back button.

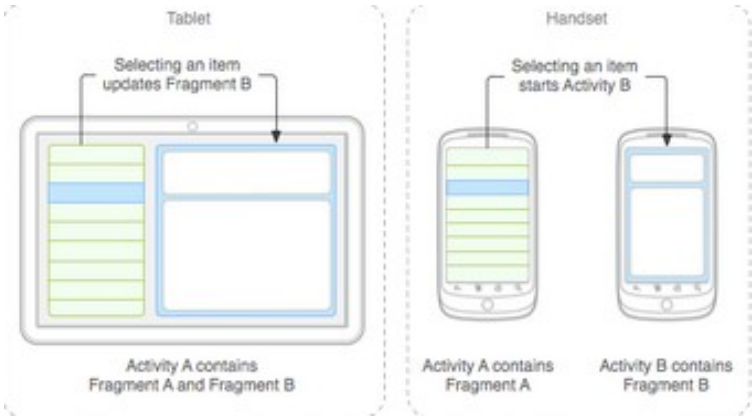
When you add a fragment as a part of activity layout, it lives in a ViewGroup inside the activity's view hierarchy. The fragment defines its own view layout. You can insert a fragment into your activity layout by declaring the fragment in the activity's layout file, as a <fragment> element, or from your application code by adding it to an existing ViewGroup.

Design:

There is more psace on large screens (eg, tablet screen) to combine and interchange UI components. Fragments allow such designs without need to manage complex changes to the view hierarchy. By dividing the layout of an activity into fragments, you can modify the activity's appearance at runtime and preserve those changes in a back stack.

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 it, just select an article and read it in the same activity.

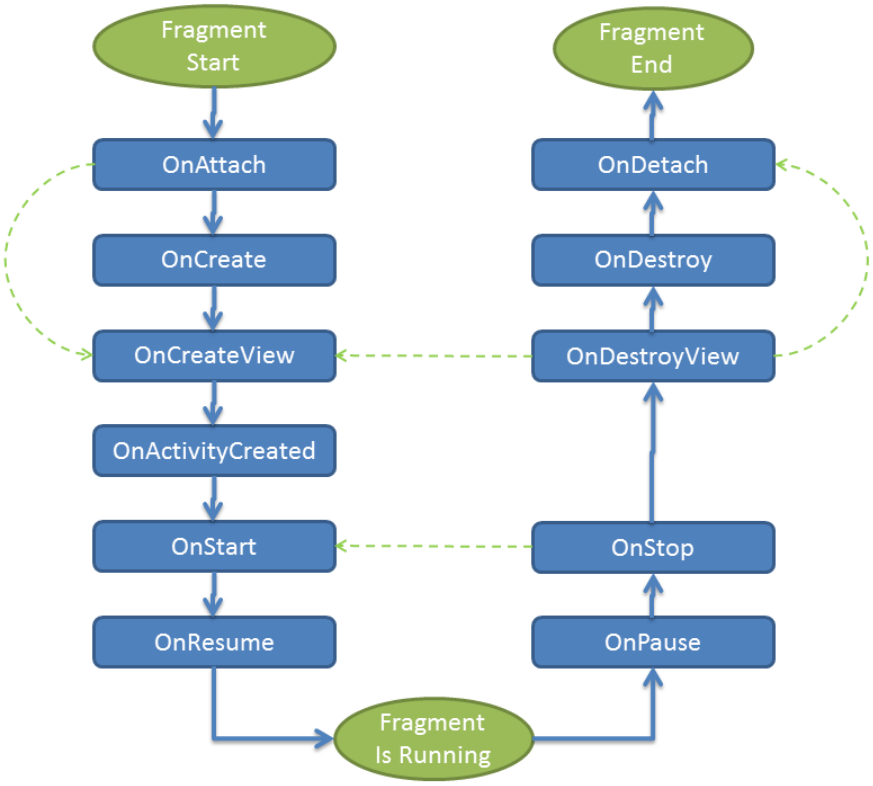


Design each fragment as a modular and reusable component. You can include one fragment in multiple activities, so design for reuse and avoid directly manipulating one fragment from another fragment. When supporting both tablets and handsets, you can reuse your fragments in different layout configurations 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.

The news articles application can embed in two fragments in Activity A, when running on a tablet-sized device. But on a handset screen, Activity A includes only the fragment for the list of articles, and when user selects an article, it starts Activity B, which includes the second fragment to read the article. Thus, the application reuses fragments in different combinations.

Creating a Fragment:

To create a fragment, create a subclass of Fragment. The Fragment class has callback methods similar to an activity. To convert an existing application to use fragments, you might simply move code from your activity's callback methods into respective callback methods of fragment.



Important lifecycle methods:
Here are some of the important lifecycle methods of fragments.

onCreate()
Called when creating the fragment. Here initialize essential components of the fragment that you want to retain when the fragment is paused or stopped, then resumed.

onCreateView()
Called when the fragment draws its user interface for the first time. Return a View from this method that is the root of your fragment's layout. Return null if the fragment does not provide a UI.

onPause()
Called when user is leaving the fragment. Commit any changes that should be persisted beyond the current user session.

Fragment Subclasses:
Some subclasses you can 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 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. (Preferred method for displaying a list is to use RecyclerView instead of ListView. In this case you would need to create a fragment that includes a RecyclerView in its layout.)

PreferenceFragmentCompat
Displays a hierarchy of Preference objects as a list. This is used to create a settings screen for your application.

Adding user interface:
To provide a layout for fragment, implement onCreateView() callback method. Return a View that is the root of your fragment's layout, using inflate from an XML layout resource. Example:

```
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);
    }
}
```

The container parameter passed to onCreateView() is the parent ViewGroup (from the activity's layout) where fragment layout is inserted. The savedInstanceState is a Bundle that provides data about the previous instance of the fragment, if the fragment is being resumed.

The inflate() method takes three arguments:
1. Resource ID: of the layout to inflate.
2. ViewGroup: the parent of the inflated layout.
3. A boolean: indicating whether the inflated layout should be attached to the ViewGroup during inflation. (Here 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.)

Adding fragment to an activity:
Usually, a fragment contributes a portion of UI to the host activity. There are two ways to add a fragment to activity layout:

1. Declare the fragment inside the activity's layout file:
Specify layout properties for the fragment as if its a view. For example, here is 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. The system inserts the View returned by the fragment in place of the <fragment> element.

2. Programmatically add the fragment to an existing ViewGroup:
At runtime, specify a ViewGroup in which the fragment is to be placed. To make fragment transactions in your activity (such as add, remove, or replace a fragment), use FragmentTransaction. Get an instance of FragmentTransaction from your FragmentActivity as:

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

Add a fragment using 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. Call **commit()** for the changes to take effect.

Managing Fragments:
To manage the fragments in your activity, use FragmentManager. To get it, call getSupportFragmentManager() from your activity. Some functions of FragmentManager include:
→ Get fragments in the activity, with findFragmentById() (for fragments with a UI) or findFragmentByTag().

→ Pop fragments off the back stack, with popBackStack()
→ Register a listener for changes to the back stack, with addOnBackStackChangedListener().

Performing Fragment Transactions

You can add, remove, replace, and perform other actions in response to user interaction. Each set of changes is called a transaction. You can also save each transaction to a back stack managed by the activity, allowing user to navigate backward through the fragment changes. Acquire an instance of FragmentTransaction as:

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

The transactions can be performed using methods such as add(), remove(), and replace(). Apply the transaction to the activity, with commit() method.

Before calling commit(), you can call addToBackStack() to add the transaction to a back stack of fragment transactions. This back stack is managed by the activity allowing to return to the previous fragment state, by pressing the Back button. For example, replacing one fragment with another, and preserve the previous state in the back stack:

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

The newFragment replaces fragment 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. FragmentActivity automatically retrieve fragments from the back stack via onBackPressed().

Adding multiple changes to the transaction before calling commit(), the changes are added to the back stack as a single transaction and the Back button reverses them all together. The order in which you add changes to a FragmentTransaction doesn't matter, except that 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.

Calling commit() doesn't perform the transaction immediately. Rather, it schedules it to run on the activity's UI thread (the "main" thread). 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.

Note: 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 later, an exception is 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 Activity:
Although a Fragment is independent from a FragmentActivity and can be used inside multiple activities, a given instance of a fragment is directly tied to the activity that hosts it. Thus, fragment can access the

FragmentActivity instance with getActivity() and 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) getSupportFragmentManager().findFragmentById(R.id.my_fragment);

Creating event callbacks to the activity:
In some cases, you might need a fragment to share events or data with the activity and/or the other fragments hosted by the activity. To share data, create a shared ViewModel. If you need to propagate events that cannot be handled with a ViewModel, you can instead 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 for 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 listener;
    ...
    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        try {
            listener = (OnArticleSelectedListener) context;
        } catch (ClassCastException e) {
                throw new ClassCastException(context.toString() + " must
implement OnArticleSelectedListener");
        }
    }
    ...
}
```

If the activity hasn't 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 listener;
    ...
    @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
        listener.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.

Adding items to the App Bar

Fragments can contribute menu items to the activity's Options Menu (and app bar) by implementing onCreateOptionsMenu(). In order for this method to receive calls, call setHasOptionsMenu() during onCreate(), to indicate that the fragment would like to add items to the Options Menu. Otherwise, the fragment doesn't receive a call to onCreateOptionsMenu().

Any items added 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 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.

Handling the Fragment Lifecycle:

Managing the lifecycle of a fragment is similar to 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 isn't 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 but not visible to the user and is killed if the activity is killed.

Preserve the UI state of a fragment across configuration changes using a combination of onSaveInstanceState(Bundle), ViewModel, and persistent local storage.

An activity is placed into a back stack of activities managed by the system when it's stopped. 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 similar to managing the activity lifecycle; the same practices apply.

If you need a Context object within your Fragment, call getContext(). However, when the fragment isn't attached to activity yet, or was detached during the end of its lifecycle, getContext() returns null.

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. Thy 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.

Each successive state of the activity determines which callback methods a fragment may receive. When activity receives onCreate() callback, a fragment in the activity receives no more than the onActivityCreated() callback.

Once the activity reaches resumed state, 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 is pushed through its lifecycle.

Example:
To bring everything 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 play titles and another to show a summary of the play when selected from the list. It also demonstrates different configurations of the fragments, based on the screen configuration.

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 (listing play titles) when activity loads the layout, while the FrameLayout (for play summary) consumes space on the right side of the screen, but remains empty at first. It's not until the user selects an item from the list that a fragment is placed into the FrameLayout.

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. When the device is in portrait orientation, only the list of play titles is visible. When the user clicks a list item, the application starts a new activity to show the summary, instead of loading a second fragment.

How this is accomplished in the fragment classes. First is TitlesFragment, which shows the list of play titles. This fragment extends ListFragment and relies on it to handle most of the list view work.

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 dualPane;
```

```
    int curCheckPosition = 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);
        dualPane = detailsFrame != null && detailsFrame.getVisibility()
== View.VISIBLE;

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

        if (dualPane) {
            // 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(curCheckPosition);
        }
    }

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

```
    @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) {
        curCheckPosition = index;

        if (dualPane) {
            // 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)

getSupportFragmentManager().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 =
getSupportFragmentManager().beginTransaction();
                if (index == 0) {
                    ft.replace(R.id.details, details);
                } else {
                    ft.replace(R.id.a_item, 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
            // isn't displayed. Note this isn't 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;
    }
}
```

Recall from the TitlesFragment class, that, if the user clicks a list item and the current layout does not include the R.id.details view (which is where the DetailsFragment belongs), then the application starts the DetailsActivity activity to display the content of the item.

Here is the DetailsActivity, which simply embeds the DetailsFragment to display the selected play summary when the screen is in portrait orientation:

```
public static class DetailsActivity extends FragmentActivity {

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

        if (getResources().getConfiguration().orientation
                == Configuration.ORIENTATION_LANDSCAPE) {
            // If the screen is now in landscape mode, we can show the
            // dialog in-line with the list so we don't need this activity.
            finish();
            return;
        }

        if (savedInstanceState == null) {
            // During initial setup, plug in the details fragment.
            DetailsFragment details = new DetailsFragment();
            details.setArguments(getIntent().getExtras());
            getSupportFragmentManager().beginTransaction().add(android.R.
id.content, details).commit();
        }
    }
}
```