Advanced Mobile Application Development
Week 12: Fragment Lifecycle

Fragments

A Fragment is a piece of behavior or user interface that can be placed in an activity which enables a more modular activity design. You can think of it conceptually as a kind of subactivity.
Important fragment concepts:

- A fragment has its own layout and its own behavior with its own lifecycle callbacks.
- You can add or remove fragments in an activity while the activity is running.
- You can combine multiple fragments in a single activity to build a multi-pane UI.
- A fragment can be used in multiple activities.
- The fragment life cycle is closely related to the Activity lifecycle but not exactly the same
- When the activity is paused, all the fragments available in the activity will also be stopped.
- A fragment can implement a behavior that has no user interface component.

Fragment Lifecycle

https://developer.android.com/guide/components/fragments
The Fragment lifecycle is similar, but not exactly the same, as the Activity lifecycle. (slides)
Lifecycle methods:
onAttach: when the fragment attaches to its host activity
onCreate: when a new fragment instance is initialized. Good place to do initial setup. Just like in an activity
onCreateView: when a fragment creates its portion of the view hierarchy, which is added to its activity's view hierarchy
onActivityCreated: when the fragment's activity has finished its own *onCreate() method*
onStart: when the fragment is visible
onResume: when the fragment is visible and running; a fragment cannot resume until its activity resumes and often does so in quick succession after the activity
There's more. These lifecycle events happen when you remove a fragment:
onPause: when the fragment is no longer interacting with the user; it occurs when either the fragment is about to be removed or replaced, or the host activity takes a pause
onStop: when the fragment is no longer visible; it occurs either after the fragment is about to be removed or replaced or when the host activity stops
onDestroyView: when the view and related resources are removed from the activity's view hierarchy and destroyed
onDestroy: when the fragment does its final clean up
onDetach: when the fragment is detached from its host activity
As you can see, the fragment's lifecycle is intertwined with the activity's lifecycle, but it has extra events that are particular to the fragment's view hierarchy, state and attachment to its activity.

Fragments and Activities

Because we want fragments to be reusable we want them to be as independent from their activity as possible.
We'll use an interface to decouple the fragment and the activity
An interface defines the minimum requirements for one object to usefully talk to another

Define the interface in fragment A (master)
Create an instance of the interface and attach the activity to it in the **onAttach()** method
onAttach() is where when the fragment gets attached to its activity

The activity that hosts fragment A implements the listener and overrides the method that will handle passing data from fragment A to fragment B (detail)

Back Stack
With activities the back button goes to the previous activity
With fragments the back button should show the previous fragment
But if we just update the fragment with different data we won't have the previous fragment on the back stack so instead we'll create a new fragment and replace it each time.

Rotation
Do you remember what happens in the Android lifecycle when the device is rotated?
If the activity uses a fragment the fragment gets destroyed along with the activity so after rotation the fragment will default back to the first item
Override **onSaveInstanceState()** to save the current state of the fragment
In **onCreateView()** restore the saved state if there is one

Multiple Screen Support
You can customize your apps appearance and flow based on different specifications

- Screen size
- Screen density
- Orientation
- Resolution
- Density independent pixel (dp)

Create layout specific folders for the configuration you want to target and provide modified layout files in those folders.
Based on the device the app is running on the system will use the layout file in the matching resource folder, same as it does with images. If there's no matching resource folder it will use the default.

As of Android 3.2(API 13) the generalized sizes of small, medium, large, and xlarge are deprecated.
Instead you now specific smallest width(sw) or available screen width
https://developer.android.com/training/multiscreen/screensizes.html
`res/layout-sw600dp/main_activity.xml` would be used for screens 600dp wide and bigger
`res/layout/main_activity.xml` would be used for all other screen widths

Even with a flexible layout there are times you'll want alternative layouts for different devices. Using fragments we can split activities up and based on the screen real estate (or other characteristics) decide which fragments will be displayed in the current activity.

Fragment vs Activity
So when should we use Activities and when should we use Fragments? Like most things on Android there are different thoughts in this debate.

Fragment flexibility:

- Lighter weight to add/replace than starting a new activity
- You can combine multiple fragments in a single activity to build a multi-pane UI
- A fragment can be used in multiple activities
- In apps with bottom navigation fragments make it easy to always have the bottom navigation present and keep track of which tab is selected. If we did this using Activities we'd have to copy

the bottom navigation code, track its state across activities, and implement the animated transition between activities.

Fragment complexity:

- coordination with the activity it's housed in
- a different lifecycle
- managing the back stack

An Activity must be the app entry point, after that it's up to you. In general Android suggests using Fragments whenever you can as they're more flexible. Some developers go as far as only having one Activity/app and the rest are Fragments. Fragments are also used in their new Navigation component/graph so they're definitely not going away.

Master Detail
Create a new project called MasterDetail
Master/Detail Flow template
Minimum SDK: API 21 (21 is the minimum API for Material Design)

Run your app on a phone and a table in landscape orientation to see what the template gave you.
The Master/Detail Flow template creates an app with a master/detail structure. The master has a list of items where selecting an item presents additional information relating to that item within a detail pane. The template creates an app structure that is flexible so on tablet sized Android device it displays in two-pane mode in landscape orientation where the master list appears in a narrow vertical panel along the left-hand edge of the screen and the detail pane on the remaining right side. On smaller, phone sized Android devices, the master list takes up the entire screen and the detail pane appears on a separate screen which appears when a selection is made from the master list. In this mode, the detail screen includes an action bar entry to return to the master list.
The template creates the following files for this structure:

activity_item_list.xml – The top level layout file for the master list is a CoordinatorLayout widget containing an app bar with a toolbar, a floating action button and includes the item_list.xml file in a FrameLayout. This layout is loaded by the ItemListActivity class.

item_list.xml – The layout file used to display the master list of items in single-pane mode where the master list and detail pane appear on different screens. This file consists of a RecyclerView object configured to use the LinearLayoutManager. The RecyclerView element declares that each item in the master list is to be displayed using the layout declared within the item_list_content.xml file.

item_list.xml (w900dp) – The layout file for the master list in the two-pane mode used on tablets in landscape (where the master list and detail pane appear side by side). This file contains a horizontal LinearLayout parent within which resides a RecyclerView to display the master list, and a FrameLayout to contain the content of the detail pane. As with the single-pane variant of this file, the RecyclerView element declares that each item in the list be displayed using the layout contained within the item_list_content.xml file.

item_content_list.xml – This file contains the layout to be used for each item in the master list. By default, this consists of two TextView objects embedded in a horizontal LinearLayout.

activity_item_detail.xml – The top level layout file used for the detail pane when running in single-pane mode. This layout contains an app bar, collapsing toolbar, scrolling view and a floating action button. This layout file is loaded by the ItemDetailActivity class.

item_detail.xml – The layout file for the detail pane which contains a single TextView by default. In single-pane mode, this fragment is loaded into the layout defined by the activity_item_detail.xml file. In two-pane mode, this layout is loaded into the FrameLayout area of the item_list.xml (w900dp) file so that it appears adjacent to the master list. This layout file is loaded by the ItemDetailFragment class.

ItemListActivity.java – The activity class responsible for displaying and managing the master list (declared in the activity_item_list.xml file) and for both displaying and responding to the selection of items in that list.

ItemDetailActivity.java – This class displays the layout defined in the activity_item_detail.xml file. The class also initializes and displays the fragment containing the detail content defined in the item_detail.xml and ItemDetailFragment.java files.

ItemDetailFragment.java – The fragment class responsible for displaying the item_detail.xml layout and populating it with the content to be displayed in the detail pane. This fragment is initialized and displayed within the ItemDetailActivity.java file to provide the content displayed within the activity_item_detail.xml layout for single-pane mode and the item_list.xml (w900dp) layout for two-pane mode.

DummyContent.java – A class file intended to provide sample data for the template. This class can either be modified or replaced entirely. By default, the content provided by this class simply consists of a number of string items.

Let's first modify this so the data is a list of names and urls and we'll load the web pages in the detail view.
In DummyContent.java update the data.

```
static {
    Add some sample items.
   addItem(new DummyItem("1", "eBookFrenzy",
         "https://www.ebookfrenzy.com"));
   addItem(new DummyItem("2", "Amazon",
         "https://www.amazon.com"));
   addItem(new DummyItem("3", "New York Times",
         "https://www.nytimes.com"));
}
```

Now we need to modify the detail layout so it can show a web page.
Replace the TextView with a WebView. I couldn't delete the TextView or add a WebView in Design mode. I had to go into the text view and modify the xml.

```
<WebView xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:id="@+id/website_detail"
```

```
    tools:context= ".ItemDetailFragment">
</WebView>
```

Web Views
https://developer.android.com/guide/webapps/webview.html
A WebView is an Android UI component that is an embedded browser that can be integrated to Android application to display web pages.
It uses the WebKit rendering engine to display web pages and includes methods to navigate forward and backward through a history, zoom in and out, and perform text searches and more.

We need to update ItemDetailFragment to load a web page.
First modify onCreate()to display the web site name in the app bar by changing the last line.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    …
        if (appBarLayout != null) {
            appBarLayout.setTitle(mItem.name);
        }
    }
}
```

Then we need to update onCreateView() to find the view with the ID of website_detail (this was formally the TextView but is now a WebView) and extract the URL of the web site from the selected item. An instance of the WebViewClient class is created and assigned the shouldOverrideUrlLoading() callback method. This method is implemented so as to force the system to use the WebView instance to load the page instead of the Chrome browser. WebViews don't enable JavaScript by default so we enable it.

```
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    View rootView = inflater.inflate(R.layout.item_detail, container, false);

    if (mItem != null) {
        WebView webView = rootView.findViewById(R.id.website_detail);
        //webView.loadUrl(mItem.url);
        webView.setWebViewClient(new WebViewClient(){
            @Override
            public boolean shouldOverrideUrlLoading(WebView view, WebResourceRequest request) {
                //tells Android to use webview instead of opening in a browser
                return super.shouldOverrideUrlLoading(view, request);
            }
        });
        webView.getSettings().setJavaScriptEnabled(true);
        webView.loadUrl(mItem.url);
    }
    return rootView;
}
```

Also update ItemListActivity to make sure that the web site names appear in the master list. In onBindViewHolder() method modify the setText() method call to reference the name.
holder.**mContentView**.setText(**mValues**.get(position).**name**);

The final step is to add internet permission to the AndroidManifest file. This will enable the WebView object to access the internet and download web pages.

```
<uses-permission android:name="android.permission.INTERNET" />
```

Run your app on a phone and a tablet in landscape orientation.

XML Data

Parsing XML Data https://developer.android.com/training/basics/network-ops/xml.html
Android/Java has multiple ways of dealing with XML, using XMLPullParser is recommended.
If your XML file is in your project, you create an xml directory in resources and then get access to it using getResources().getXml(R.xml.*filename*); which returns a XMLResourceParser. This is more efficient than adding it as an asset and opening it as an input stream.
If you're downloading it from the Internet you should do so in an asynchronous thread so as to not hold up the UI. We will do this when we deal with JSON.
XMLResourceParser is a subclass of XMLPullParser
https://developer.android.com/reference/org/xmlpull/v1/XmlPullParser.html
The current event state of the parser can be determined by calling the getEventType() method.

Event types:

- START_DOCUMENT: the beginning of the file
- START_TAG: an XML start tag was read
  - the tag name can be retrieved using the getName() method
- TEXT: text content was read
  - the text content can be retrieved using the getText() method
- END_TAG: an XML end tag was read
  - the tag name can be retrieved using the getName() method
- END_DOCUMENT: end of documents, no more events

The method next() advances the parser to the next event.
You basically walk through the document and test for the tags that you're interested in and extract that data.

Context

You're going to see Context in Android a lot. Context is basically the current environment and is needed to access application-wide resources. This includes anything you usually use the R class to reference or getting the path to a specific directory or file in the filesystem.
There are different methods you can use to get Context:

- getApplicationContext()
- getContext()
- getBaseContext()
- or this (when in the activity class)

When those aren't available you can extend the Application class as a way to access the Application Context, which is what we'll be doing.

Load XML

To add our XML file we need to create an XML directory in resources and add it there.
Right click on res folder File | New | Android resource directory
Name: xml
Type: xml
Copy and past the xml file in the xml directory

To load the XML file let's create a new class to do this.

```
public class DataActivity {}
```

Since this is not in our Activity, we won't be able to access the R class in order to access our XML file. So we'll create a new class that extends the Application class that will have a method to get our applications context.

```
public class MyApplication extends Application {
   private static Context context;

   public void onCreate() {
      super.onCreate();
      MyApplication.context = getApplicationContext();
   }

   public static Context getAppContext() {
      return MyApplication.context;
   }
}
```

And you must add this to the <application> tag in the Android manifest file.

```
android:name=".MyApplication"
```

Now we'll create a file in the DataActivity class to load our XML file.

```
public List<DummyContent.DummyItem> loadXML() throws XmlPullParserException, IOException {
   String new_name = new String();
   String new_url = new String();
   int id_counter = 0;
   List<DummyContent.DummyItem> heroes=new ArrayList<DummyContent.DummyItem>();
   //string for debugging purposes only
   StringBuffer stringBuffer = new StringBuffer();
   //get xml file
   XmlResourceParser xpp = MyApplication.getAppContext().getResources().getXml(R.xml.marvel);
   //advances the parser to the next event
   xpp.next();
   //gets the event type/state of the parser
   int eventType = xpp.getEventType();
   while (eventType != XmlPullParser.END_DOCUMENT) {
      switch (eventType) {
         case XmlPullParser.START_DOCUMENT:
            // start of document
            break;
```

```
            case XmlPullParser.START_TAG:
                if (xpp.getName().equals("hero")) {
                    stringBuffer.append("\nSTART_TAG: " + xpp.getName());
                }
                if (xpp.getName().equals("name")) {
                    stringBuffer.append("\nSTART_TAG: " + xpp.getName());
                    eventType = xpp.next();
                    new_name = xpp.getText(); //gets the name of the hero
                }
                else if (xpp.getName().equals("url")) {
                    stringBuffer.append("\nSTART_TAG: " + xpp.getName());
                    eventType = xpp.next();
                    new_url = xpp.getText(); //gets the url of the hero
                }
                break;
            case XmlPullParser.END_TAG:
                if (xpp.getName().equals("hero")) {
                    id_counter++;
                    //create new item object
                    DummyContent.DummyItem new_item = new
DummyContent.DummyItem(String.valueOf(id_counter), new_name, new_url);
                    heroes.add(new_item);
                }
                break;
            case XmlPullParser.TEXT:
                break;
        }
        eventType = xpp.next();
    }
    return heroes;
}
```

Now let's update DummyContent.java so instead of static data it's using the XML data.

```
public void dataSetup(){
    List<DummyContent.DummyItem> xmlData=new ArrayList<DummyContent.DummyItem>();
    DataActivity xmlDataActivity = new DataActivity();
    if (ITEMS.size() == 0) {
        try {
            xmlData = xmlDataActivity.loadXML();
            for (int i = 0; i < xmlData.size(); i++) {
                addItem(xmlData.get(i));
            }
        } catch (XmlPullParserException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

And we call that at the end of onCreate() in ItemListActivity.java
DummyContent dummyContent = **new** DummyContent();
dummyContent.dataSetup();

Now when you run the app you should see the data from the XML file.

Since we're not using the Floating Action buttons I've commented them out of my layout and java files. I also converted item_list.xml to a constraint layout and added an image at the top. This will only show in single pane mode.
In item_list_content.xml I commented out the TextView for the id_text and in ItemListActivity.java I commented out all references to **mIdView.**