CSCD 372 – Android Mobile Development

Module 8: Fragments

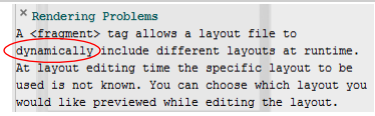
Fragment

- A Fragment is a section of UI functionality
  - introduced in Android 3.0 (Honeycomb), and the compatibility support package allows Fragments for Android 1.6 (API 4) +
  - one reason for Fragments is to make it easier to create different UI configurations for tablets and phones
  - e.g., what a phone needs to display on two separate screens could be displayed side-by-side on a tablet
  - by implementing these screens as fragments, the app can choose to place one or both fragments on an activity layout
- Other reasons for Fragments
  - when a new Activity is started, the ActionBar is redrawn (because it belongs to an Activity), whereas it remains in place when a new Fragment is displayed
  - Retained (setRetainInstance) Fragments (called worker Fragment when w/o view) are now the preferred way to keep objects alive across configuration changes, and to reconnect such objects to re-created Activities

Creating & Displaying Fragments

- Similar to an Activity ...
  - a Fragment is represented by the combination of an XML layout and a class that extends Fragment
  - Fragments have a lifecycle similar to an Activity
  - a Fragment can be statically placed in an Activity Layout, but that's a trivial exercise we'll cover as part of a lab assignment
- To dynamically use a Fragment in an Activity:
  - for the Activity, use placeholder layouts (e.g., FrameLayout) or a ViewGroup (multiple Fragments), and make sure id's are assigned
  - create an XML layout for the Fragment (just like you would for an Activity)
  - add a class (or more) extending Fragment
  - inflate a fragment's view in onCreateView() of the Fragment class
  - attach a fragment to an Activity using a FragmentTransaction object
  - the Basic Wizard has a checkbox to place the content in a Fragment
  - the Empty Wizard does NOT offer this

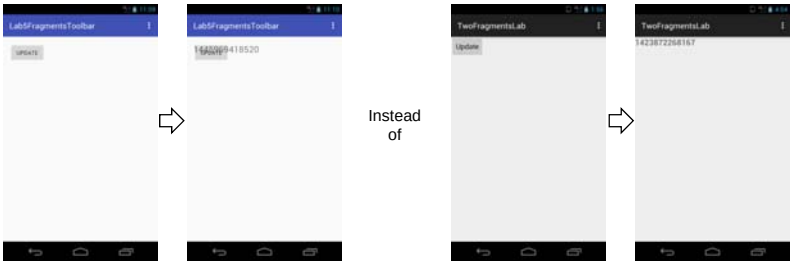
You might see this at some point



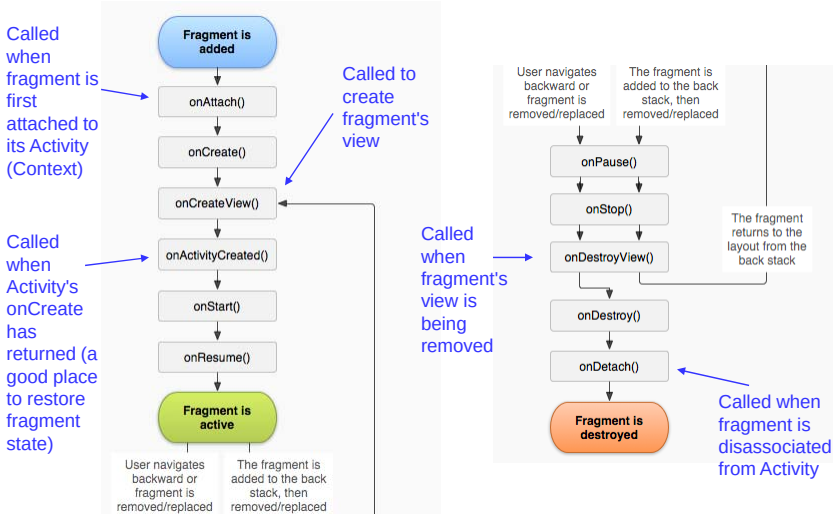
- This is saying that <fragment> is a placeholder for the layout of some fragment, and it doesn't (yet) know what that layout is
  - it requires, however, that you give it a Fragment class name (in the android:name attribute)
  - that class will determine what is inflated here
- Can that class choose different layouts to inflate there?
  - It could, but would be weird and I have yet to see that being done
  - you generally have 0 or 1 layout for each class derived from Fragment
  - when *I* say dynamic fragments, I mean changing the fragment *class* as well as the layout, and something other than <fragment> is used as the placeholder
  - the underlying class doesn't change for a <fragment> tag, so I will be referring to that as a *static* fragment

Difference for the Lab

- **After using two <fragment> tags I ask you to use a single <FrameLayout> in the portrait orientation**
  - **and then use FragmentTransactions to replace the "fragment" displayed there (class AND layout, not just the layout)**
- **Here's what would happen if we did those transactions on a <fragment> placeholder**
  - **with name attribute of MainFragment**



Fragment Lifecycle

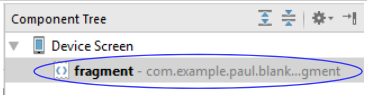


Where to Save and Restore

- **In the Fragment**
  - **onSaveInstanceState(): call the superclass version and save the Fragment's state to the Bundle**
  - **onActivityCreated(): call the superclass version and restore the Fragment's state from the passed Bundle (if not null)**
- **In the attaching Activity**
  - **try to NOT keep a reference to your fragment in your Activity (caveats coming), but if you feel you must...**
  - **in onCreate, you can ask the Fragment manager (getFragmentManager) for a reference to a fragment, either**
    - by tag (findFragmentByTag): a good way to get reconnected to a retained worker fragment
    - or by placeholder ID (findFragmentByID): a good way to get reconnected to a fragment that has a view – recall that Android will take care of repopulating your Activity's view hierarchy, including any fragments contained in placeholders
  - **if the Fragment is retained, then the Activity could also put a reference to the Fragment in the outgoing bundle and retrieve it from the incoming bundle**
    - see FragmentManager#putFragment, and getFragment

Basic with Fragment Wizard

- **content_main.xml:**



- **fragment_main.xml:**

```
<RelativeLayout xmlns:android="http: ...
xmlns:tools="http://schemas.android.com/tools"
    ...
    tools:showIn="@layout/activity_main">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />
</RelativeLayout>
```

Hardcoded string!

- **Placeholder fragment:**

```
public (static) class MainActivityFragment extends Fragment {
   public MainActivityFragment() {
   }

   @Override
   public View onCreateView(LayoutInflater inflater, ViewGroup container,
                            Bundle savedInstanceState) {
      return inflater.inflate(R.layout.fragment_main, container, false);
   }
}
```

Nested Fragment Classes

- In this simple case the Fragment only contains:
  - a no-argument constructor (required by the Android system so it can create your Fragment on the fly)
  - onCreateView, where its layout (if it has one) is inflated to a View object and returned
- Note that if nested in an Activity, a Fragment class *must* be static
  - because a non-static nested (i.e., *inner*) class has a hidden reference to its containing class
  - this gives it access to the containing class's members (something that a static nested class cannot do)
  - as you are by now aware, Android destroys and rebuilds Activities behind the scenes – it does the same for Fragments
  - this hidden reference creates problems for Android when it rebuilds Activities and Fragments (prevents the Activity from being collected)
  - a previous version of the "Blank with Fragment" wizard generated the Fragment as a nested class in MainActivity, but it now places it in a separate file

Attaching the Fragment

- A <fragment> widget contains an android:name parameter identifying the class
  - it is thus the approach for "static" Fragments, *which are loaded for you when the container xml is inflated*
- For dynamic holders, the default Fragment is typically attached in Activity#onCreate()
  - beginTransaction() returns a FragmentTransaction object, which is used to add the Fragment to the Activity layout in the placeholder position
  - note that this is only done when the Activity is FIRST being created (Fragments are automatically reattached, if the same Activity view exists, when an Activity is recreated)

```
protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.activity_main);
     if (savedInstanceState == null) {
         getFragmentManager()
              .beginTransaction()
              .add(R.id.container, new StartupFragment())
              .commit();
     }
```

use getSupportFragmentManager() for API<14

Fragment Transactions

- We can now respond to some event by replacing the existing Fragment with another
  - e.g., in response to a Navigation event (maybe from a Toolbar)

```
public void onSomeEvent(View view) {
   FragmentTransaction trans =
           getFragmentManager().beginTransaction() ;
   trans.replace(R.id.container, new SomeOtherFragment()) ;
   trans.commit() ;
}
```

  - FragmentTransaction methods:

add():	adds a Fragment to an Activity state (and optionally, but not necessarily, to a container)
remove():	removes a fragment from a container
replace():	replaces a fragment that is in a container
detach():	detaches a fragment from the UI, but retains it state so it can be reattached, but only to the same container
hide():	hide a fragment that has been added to a container (see also show)

Fragment Lifecycle Control

- Some things to note:
  - a Fragment is closely tied to the Activity it is in, and its lifecycle is dependent on that Activity
  - the commit call is non-blocking: an added Fragment will not yet exist when that call returns
- Fragment#setRetainInstance(true)
  - you can call this to ensure that a Fragment's state (its member variables) are retained across Activity re-creation during configuration changes (e.g., rotation)
  - if one of those member variables is a reference back to the Activity, you need to update it in onAttach(), as it will have a new containing activity (null it on onDetach)
  - likewise for any references to View elements, in onCreateView, as the View (if it has one) will be re-created
  - this cannot be used for Fragments that are on the back stack
  - a popular use of this is to retain connections to Threads (or AsyncTasks) across new Activity instances (in a Viewless Worker Fragment)

Swiped Fragment Nav (Overview)

- Use the "Tabbed Activity Wizard"
  - select "Swipe Views" for the Nav Style (the other options for Action Bar Tabs and Action Bar Spinner actually use Toolbar)
  - the main activity layout now contains a custom view based on ViewPager (R.id.container), ...
  - ViewPager uses an Adapter derived from FragmentPagerAdapter
  - A shell for the adapter is generated for you (called SectionsPagerAdapter)
  - MainActivity#onCreate() instantiates the adapter and gives it to the ViewPager
  - The Adapter is responsible for providing the Pager with the appropriate Fragment to display in an overridden method: SectionsPagerAdapter#getItem(int position)
  - the wizard-generated code for that creates a new instance of the PlaceholderFragment class (generated as a static nested class) each time getItem() is called

SwipedNavEx

- The Main Activity Creates the ViewPager and gives it an Adapter

```
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);

   // new Toolbar code is generated here
   ...

   // Create the adapter that will return a fragment for each
   // section (swipe page) of the activity.
   mSectionsPagerAdapter = new
            SectionsPagerAdapter(getFragmentManager());

   // Set up the ViewPager with the adapter
   mViewPager = (ViewPager) findViewById(R.id.container);
   mViewPager.setAdapter(mSectionsPagerAdapter);
   ...
}
```

FragmentPagerAdapter

- **getItem() is where our PagerAdapter provides the page**
  - **you CANNOT use the same fragment *object* for more than one page**
  - **but you can, of course, use multiple instances of the same class (which is the assumption of the Wizard), and in this case using a <fragment> tag in the layout is appropriate**
  - **this modified example uses 2 classes for 4 pages (one class uses a static Factory method and the other uses a constructor)**

```
public class SectionsPagerAdapter extends FragmentPagerAdapter {
   ...
   // this is called to instantiate the fragment for the given page
   // create a new instance of whatever fragment you want
   @Override
   public Fragment getItem(int position) {
      if (position >= 3) return new SpecialFragment() ;
      return PlaceholderFragment.newInstance(position + 1);
   }

   @Override
   public int getCount() {
      return 4 ;  // show 4 pages
   }
   ...
```

SwipedNavEx

- **Sections (pages) 1-3 use a static Factory method**
  - **newInstance() takes the place of an explicit value constructor, by creating and initializing a new instance of PlaceholderFragment**
  - **Android will recreate the Activity (and fragments) at will, and knows only the no-argument constructor for that**
  - **the static Factory method approach (newInstance) encapsulates explicit construction in the class, and with Android typically passes initializing info through a Bundle arg**

```
public static PlaceholderFragment newInstance(int sectnNumber)
{
   PlaceholderFragment fragment = new PlaceholderFragment();
   Bundle args = new Bundle();
   args.putInt(ARG_SECTION_NUMBER, sectnNumber);
   fragment.setArguments(args);

   return fragment;
}
```

MUST we instantiate a Fragment ...

- **... every time the page is swiped?**
  - this is not well-understood "out there", but although it may appear to be that way, that isn't what actually happens
  - the Fragment manager will handle creating these pages as needed, and may keep them in memory once created, and remember the views they are attached to – so it does not necessarily call getItem() every time the page is swiped
  - in this case, each Fragment will be created at most once (***for each configuration change (e.g., a rotation***)
- **FragmentPagerAdapter:**
  - ***"each page is persistently kept in the fragment manager as long as the user can return to the page"***
  - if the Activity is killed or changes configuration, the fragment will also typically be recreated (unless it has been retained)
  - the Fragment's view hierarchy may be destroyed when not visible (but Android will repopulate the GUIs)
  - so, this adapter is appropriate when there is only a handful of fragments to be paged through

Must I instantiate in getItem?

- **You COULD ...**
  - instantiate your fragments in your Activity, and just return the appropriate instance in getItem() (SwipedNavMod)
  - there is likely little to be gained in doing so (except perhaps code complexity), and you might be needlessly instantiating a Fragment that is never viewed
  - you're risking an error that creates a mismatch between the references held by the FragmentManager and your Activity
  - furthermore, if you use this Adapter ...
- **FragmentStatePagerAdapter**
  - "when pages are not visible to the user, their entire fragment ***may*** be destroyed, keeping only any saved state"
  - use this adapter if you have a large number of pages, in order to help Android conserve memory
  - in this case you WILL need to let the FragmentManager dictate creation, and instantiate new instances in getItem()

Therefore ...

- ... hanging on to Fragment references in your Activity is discouraged
  - don't do it unless you have a good reason and really know what you are doing
- What if I need one Fragment to communicate to another?
  - first of all, such communication should ALWAYS be relayed through method calls to the Activity
  - the Activity can relay this with a method call to the other fragment without holding a reference to it using:

    `getSupportFragmentManager().findFragmentById(...) ;`

    (this takes the container id, you can also find a fragment by tag)
  - If that returns null, then the Fragment doesn't exist. If you need it to, then you can create it, give it an argument Bundle, and add it to the Fragment Manager, with or without displaying it, using an add() or replace() Transaction
  - so while this scenario may give you a reason to want to hang on to Fragment references in your Activity, that can still be avoided with proper use of the Fragment Manager

Fragment Construction

- What is the difference between the Fragment constructor and Fragment#onCreate()?
- The constructor is of course called when the Fragment is instantiated
  - remember that the constructor used by Android cannot take creation arguments
  - any argument bundle does not yet exist
- Fragment#onCreate() is called sometime later
  - and Android will provide it with any argument Bundle
  - so if you're instantiating with arguments, you generally initialize here instead of in an explicit value constructor
  - note, however, that the view hierarchy does not yet exist here, so wait to do any view initialization until onCreateView()
  - BTW, you can wait to process the argument bundle in onCreateView() as well, but note that onCreateView() may be called more often than onCreate() – heavy sigh

Back to SwipedNavEx

- In onCreateView() we read the initializing arguments and modify the fragment view accordingly

```
public View onCreateView(LayoutInflater inflater, ...) {
   View rootView = inflater.inflate(R.layout.fragment_main,
                   container, false);

   if (getArguments() != null)
      this.section = getArguments().getInt(ARG_SECTION_NUMBER) ;

   TextView sectionView = (TextView)
                          rootView.findViewById(R.id.section_label) ;
   sectionView.setText(Integer.toString(section)) ;

   ImageView sectionImage = (ImageView)findViewById(R.id.imageView) ;
   switch (section) {
      case 1:
         sectionImage.setImageResource(R.drawable.beatles1) ; break;
      case 2:
         sectionImage.setImageResource(R.drawable.beatles2) ; break;
      ...
   }
   return rootView;
}
```

Fragment / Activity Interaction

- If the Fragment is not nested in the Activity
  - it can of course still call public methods in the Activity class:
    ```
    MainActivity activity = (MainActivity) getActivity() ;
    activity.doSomething() ;
    ```
  - but, of course, this requires that the type of the attached Activity be known at compile time (MainActivity in this case)
- If the Activity class is NOT known
  - (as in designing your Fragments for re-usability)
  - a Fragment can require that the containing activity implement a specific callback interface
  - the Fragment class defines the callback interface
  - in onAttach() the Fragment attempts to cast the containing class (passed as an argument) to the interface type
  - in the following example, the attached Activity must implement MyFragment.MyListener (with method onMyEvent)
  - New→Fragment will generate template code for all this

onAttach() Example

```
public class MyFragment extends Fragment
{
   // define the interface that is required in the Activity
   public interface MyListener {
      public void onMyEvent(String event) ;
   }

   // keep a local reference to the activity
   MyListener listener ;

   // when someone attaches me, try casting the attaching activity
   // to my required reference type, which is an interface
   @Override
   public void onAttach(Activity activity) {
      super.onAttach(activity) ;
      try {
         listener = (MyListener) activity ;
      } catch (ClassCastException e) {
         throw new ClassCastException(activity.toString() +
                   " must implement MyListener");
      }
   }
   ...
```

onAttach() Example

```
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState)
  {
     View v = inflater.inflate(R.layout.myfragment, container, false) ;

     // simple example where a handler in the fragment needs
     // to call a specific method in the Activity
     // here we create the local button handler as an AIC
     // that makes a call back to the attached activity
     Button button = (Button) v.findViewById(R.id.button) ;
     button.setOnClickListener(new OnClickListener() {
           public void onClick(View v) {
              mListener.onMyEvent("it happened") ;
           }
     } );

     return v;
  }
}  // end of Fragment class
```

Frag-sensitive Menus (see ex)

- A Fragment can modify the Menu / ActionBar when it is attached to the Activity
  - it should have its own menu resource file
  - it must make this call in onCreateView():

    ```
    setHasOptionsMenu(true) ;
    ```

  - it must also override onCreateOptionsMenu(), which is similar, but slightly different than the same method in an Activity:

    ```
    public void onCreateOptionsMenu(Menu menu,
                MenuInflater inflater) {
       // Inflate the menu (onto ActionBar if present)
       inflater.inflate(R.menu.menu_fragment2, menu);
    }
    ```

- Handlers for any added menu event(s)
  - can be placed in the override of onOptionsItemSelected() in the Activity class (to which I say yay! for centralization)

Tabbed+Swiped Navigation (see ex)

- If you want tabbed navigation with swiping:
  - try the "Tabbed Activity Wizard" again
  - and select "Action Bar Tabs (with ViewPager)" for the Navigation Style
  - this combines swiping with tabbed navigation
  - more code will be added for the tabs, but completing the implementation is identical to what we just did with swiping
- Like straight tabbed nav, this nav style is deprecated for ActionBar
  - so this is another example where the Wizard now generates a Toolbar to replace the ActionBar
  - note the smooth movement between pages and the indicator that slides across the Tab labels