Advanced Mobile Application Development
Week 12: Fragments

Fragments

We've been using Activities in our apps but sometimes you want more flexibility for your layouts and don't want to design a dozen different layouts.
Fragments enable your app to be more modular, reusable, and adaptable (slide)
Modularity

- Fragments allow you to break your activities up into smaller modular components
- Divide complex activity code across fragments for better organization and maintenance

Reusability

- Separating behavior or UI parts into fragments allow you to share them across multiple activities
- Fragments can easily be reused and adapted for different device sizes, orientation, or other criteria

Adaptability

- Representing sections of a UI as different fragments lets you utilize different layouts depending on screen orientation and size (slide)
- You can one or more fragments embedded in an activity (slide)

Creating Fragments

To create a fragment you subclass the **Fragment** class. The Fragment class has a lot of the same methods as the **Activity** class.
https://developer.android.com/guide/components/fragments.html

- *Fragment* subclasses require an empty default constructor. Android Studio creates one for you when you use the Fragment template.
- It also has a **ListFragment** subclass which displays a list of items using an adapter just like activities
  - **onListItemClick()** handles click events

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)

Fragment 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.

Adding to a User Interface
Create the layout for a fragment using the **<fragment>** element
- Specify which fragment using the **class** attribute
- Inflate the xml layout resource in onCreateView()

<FrameLayout> is a view group that let's you specify an area in your layout where you can add a fragment later programmatically

Adding to an Activity
You can then ad a fragment to an activity by either
- Adding the fragment to the activity's layout file
  - When the system creates the activity's layout it instantiates each fragment by calling its onCreateView() method to retrieve the fragment's layout
- Programmatically add the fragment to an activity
  - Use the **FragmentTransaction** class to programmatically interact with fragments
  - The **FragmentManager** class manages an activity's fragments
    - **getFragmentManager()** gets reference to a fragment
    - **beginTransaction()** starts a fragment transaction
    - **replace(id, fragment)** replaces a fragment
    - **commit()** commits the changes

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 or available screen width
http://developer.android.com/guide/practices/screens_support.html
https://developer.android.com/training/multiscreen/screensizes.html

Superheroes
Create a new project called Superheroes
Minimum SDK: API 21 (can pick an earlier version such as 17 if needed)
Check Phone and Tablet, leave the rest unchecked
Empty Activity template
Activity name: MainActivity
Check Generate Layout File
Layout Name: activity_main
Uncheck backwards compatibility

Hero java class
Create a new Java class called Hero.

```
import java.util.Arrays;
import java.util.ArrayList;

public class Hero {
   private String universe;
   private ArrayList<String> superheroes = new ArrayList<>();

   //constructor
   private Hero(String univ, ArrayList<String> heroes){
      this.universe = univ;
      this.superheroes = new ArrayList<String>(heroes);
   }

   public static final Hero[] heroes = {
         new Hero("DC", new ArrayList<String>(Arrays.asList("Superman", "Batman", "Wonder
Woman", "The Flash", "Green Arrow", "Catwoman"))),
         new Hero("Marvel", new ArrayList<String>(Arrays.asList("Iron Man", "Black Widow",
"Captain America", "Jean Grey", "Thor", "Hulk")))
   };
```

```
    public String getUniverse(){
        return universe;
    }

    public ArrayList<String> getSuperheroes(){
        return superheroes;
    }

    public String toString(){
        return this.universe;
    }
}
```

In Java Arrays are a fixed length while ArrayLists are dynamic in size so you can add to them.
Also, ArrayList can not contains primitive data types (like int, float, double) it can only contain objects while Array can contain both primitive data types as well as objects. If you try to add a primitive data type to an ArrayList it actually creates an object for it so it can store the data.

Universe List Fragment
Let's create a fragment that will be used for our first view which will have an image and a list of the superhero universes.

File | New | Fragment | Fragment(blank)
Name: UniverseListFragment
Check Create layout XML
Fragment layout name: fragment_universe_list
Uncheck both include check boxes to keep code simpler
Language: Java
Finish

Android Studio creates a new fragment file called UniverseListFragment.java in the app/src/main/java folder, and a new layout file called fragment_universe_list.xml in the app/src/res/layout folder.

Go into fragment_universe_list.xml
You can see that fragment layout code looks a lot like activity layout code. You can use all the same views and layouts you've been using in activities when you're creating fragments.
Copy the superheroes image file or any other images you're going to use into the drawables folder.
Change the FrameLayout, which is meant to display a single item, to a ConstraintLayout by going into the Design view and right click on FrameLayout in the component tree and pick Convert FrameLayout to ConstraintLayout, selecting to flatten the hierarchy.
Delete the TextView that was included and add an ImageView using the superhero image and give it an id of imageView. Add a ListView and give it the id listView.

Now we need to add the fragment to the main activity which was created for us when we created the project. In activity_main.xml remove the TextView and add the fragment. We want the fragment on the left side of the screen.

```
<fragment
   android:layout_width="150dp"
   android:layout_height="match_parent"
   android:id="@+id/universe_frag"
   android:name="com.example.aileen.superheroes.UniverseListFragment"
   tools:layout="@layout/fragment_universe_list"
/>
```

The name attribute specifies the fully qualified name of your fragment.
The tools:layout tag is to tell it how to lay out the fragment.
Add constraints if needed using Infer Constraints.

Look at the code generated in UniverseListFragment.java

Note it extends the Fragment class instead of Activity.

You **MUST** change **import** android.support.v4.app.Fragment; to **import** android.app.Fragment; or your app will crash (error will reference xml).

The onCreateView() method gets called each time Android needs the fragment's layout, and it's where you indicate which layout the fragment uses. You'll need to implement this method almost every time you create a fragment because it's called whenever you're creating a fragment with a layout. inflater.inflate() is a fragment's equivalent to an activity's setContentView() method. It inflates the layout from the XML into the view.

All fragments must have a public constructor with no arguments. Android uses it to reinstantiate the fragment when needed, and if it's not there, you'll get a runtime exception. A public no- argument constructor has automatically been created for us.

Now lets update UniverseListFragment.java to load our values into our fragment's listview.

```
@Override
public void onStart(){
   super.onStart();
   View view = getView();
   if (view != null){
      //load data into fragment
      //get the list view
      ListView listUniverse = (ListView) view.findViewById(R.id.listView);
      //define an array adapter
      ArrayAdapter<Hero> listAdapter = new ArrayAdapter<Hero>(getActivity(),
android.R.layout.simple_list_item_1, Hero.heroes);
      //set the array adapter on the list view
      listUniverse.setAdapter(listAdapter);
   }
}
```

Need to use onStart() because in onCreateView() the view isn't created yet so we can't access it.

```
import android.widget.ArrayAdapter;
import android.widget.ListView;
```

If you run it at this point you should be able to see the fragment with the image and data in the list view.

Hero Detail Fragment

Now let's create our detail fragment to hold the list of superheroes. This is going to hold a list view of the superheroes and eventually a button to add superheroes.

File | New | Fragment | Fragment(blank)
Name: HeroDetailFragment
Check Create layout XML
Fragment layout name: fragment_hero_detail
Uncheck both include check boxes
Language: Java
Finish

Go into fragment_hero_detail.xml
Change the layout to have a ListView with the id herolistView. You can change it to ConstraintLayout or since it will only have a single item we can leave it as a FrameLayout.

```
<ListView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/herolistView" />
```

For now let's populate the list view with the heroes of the first universe (DC) just to get it working.
HeroDetailFragment.java
You **MUST** change **import** android.support.v4.app.Fragment; to **import** android.app.Fragment; or your app will crash.

```
//array adapter
private ArrayAdapter<String> adapter;

@Override public void onStart(){
    super.onStart();

    View view = getView();
    ListView listHeroes = (ListView) view.findViewById(R.id.herolistView);

    // get hero data
    ArrayList<String> herolist = new ArrayList<String>();
    herolist = Hero.heroes[0].getSuperheroes();

    //set array adapter
   adapter = new ArrayAdapter<String>(getActivity(), android.R.layout.simple_list_item_1, herolist);
```

```
    //bind array adapter to the list view
    listHeroes.setAdapter(adapter);
}

import android.widget.ListView;
import java.util.ArrayList;
import android.widget.ArrayAdapter;
```

When we used an array adapter to populate a list view with an activity we could use "this" to get the current context because activity is a type of context, the Activity class is a subclass of the Context class. The Fragment class is not a subclass of the Context class so we must use getActivity() to get the activity that the fragment is associated with. (using getContext() requires API 23 for the fragment class)

Eventually we'll want the universe fragment to tell the hero fragment which superheroes to display. To do this we'll need the id of the universe and a setter method to set it. Later we'll use it to update the fragment view.

```
private long universeId; //id of the universe chosen

//set the universe id
public void setUniverse(long id){
    this.universeId = id;
}
```

Now we need to add the fragment to the main activity layout so it appears to the right of UniverseListFragment.
activity_main.xml

```
<fragment
    android:id="@+id/detail_frag"
    android:layout_width="200dp"
    android:layout_height="match_parent"
    android:name="com.example.aileen.superheroes.HeroDetailFragment"
    tools:layout="@layout/fragment_hero_detail"
    app:layout_constraintRight_toRightOf="parent"
/>
```

Now when you run it you should see both fragments side by side.
You might need to change the dimensions of your image if it's cut off.
Cntrl fn F11 to rotate the emulator on a Mac; cntrl F11 on a PC.

Update Detail List

Now lets get the detail view to show the heroes based on which universe the user selects.
We want the fragments to be reusable so it's best if they rely as little as possible on the activity using it.
We'll decouple the fragment and the activity by using an interface.
When we define an interface, we're saying what the minimum requirements are for one object to talk usefully to another. It means that we'll be able to get the fragment to talk to any kind of activity, so long as that activity implements the interface.

Update UniverseListFragment.java to create an interface, define a listener, assign the listener to the context, and tell the listener when an item is clicked.

```
//create interface
interface UniverseListListener{
   void itemClicked(long id);
}

//create listener
private UniverseListListener listener;

@Override public void onAttach(Context context){
   super.onAttach(context);
   //attaches the context to the listener
   listener = (UniverseListListener) context;
}
```

Update the class to implement the AdapterView OnItemClickListener
```
public class UniverseListFragment extends Fragment implements AdapterView.OnItemClickListener

@Override public void onItemClick(AdapterView<?> parent, View view, int position, long id){
   if (listener != null){
      //tells the listener an item was clicked
      listener.itemClicked(id);
   }
}
```

And update onStart() to attach the listener to the listview (add this after setAdapter() in the if body)
```
      //attach the listener to the listview
      listUniverse.setOnItemClickListener(this);

import android.content.Context;
import android.widget.AdapterView;
```

When an item is clicked in the fragment, the itemClicked() method in the activity will be called so that's where we can have the fragment update its details. Instead of updating the fragment details we're going to replace the detail fragment with a brand-new detail fragment each time we want its text to change. This is so the back button does what a user would expect and go to the previous data if they had clicked on different items, and not automatically the previous view.

Now we need to update MainActivity.java to have our main activity implement the interface.
Update MainActivity.java
```
public class MainActivity extends Activity implements UniverseListFragment.UniverseListListener
```

Next we create a new HeroDetailFragment when the user clicks on a universe.

```
@Override public void itemClicked(long id){
   //create new fragment instance
   HeroDetailFragment frag = new HeroDetailFragment();
```

```
    //set the id of the universe selected
    frag.setUniverse(id);
    //create new fragment transaction
    FragmentTransaction ft = getFragmentManager().beginTransaction();
    //replace the fragment in the fragment container
    ft.replace(R.id.fragment_container, frag);
    //add fragment to the back stack
    ft.addToBackStack(null);
    //set the transition animation-optional
    ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
    //commit the transaction
    ft.commit();
}

import android.app.FragmentTransaction;
```

Fragments don't have a onBackPressed() method so when you press the back key the app looks for the previous activity or if there is none, exits. To have the back button work correctly with fragments we need to override the onBackPressed() method and have it check if there are fragments on the backstack to pop them out, otherwise follow the default behavior.

```
@Override public void onBackPressed() {
    if (getFragmentManager().getBackStackEntryCount() > 0 ){
        getFragmentManager().popBackStack();
    } else {
        super.onBackPressed();
    }
}
```

We also need to change the activity_main.xml layout file. Instead of inserting HeroDetailFragment directly, we'll use a frame layout.

A frame layout is a type of view group that's used to block out an area on the screen. You define it using the <FrameLayout> element. You use it to display single items—in our case, a fragment. We'll put our fragment in a frame layout so that we can control its contents programmatically. Whenever an item in the UniverseListFragment list view gets clicked, we'll replace the contents of the frame layout with a new instance of HeroDetailFragment that displays details of the correct superheroes.

In activity_main.xml replace the hero detail fragment with a FrameLayout, change the id to fragment_container, and keep the same constraints.

```
<FrameLayout
    android:layout_width="200dp"
    android:layout_height="match_parent"
    app:layout_constraintRight_toRightOf="parent"
    android:id="@+id/fragment_container" />
```

Now that we have that set up let's update HeroDetailFragment.java so based on which universe the user selected the correct heroes are shown. We just need to change the one line where we have Hero.heroes[0] now use the universeId.

```
herolist = Hero.heroes[(int) universeId].getSuperheroes();
```

Now when you run it you should be able to select a universe and see the correct superheroes.

Rotation

You'll notice that if you select any item other than the first, when you rotate the device the list automatically goes back to the first one. That's because the detail fragment is getting destroyed and recreated along with the activity. (control fn F11 to rotate)

In HeroDetailFragment.java save the current universeId being displayed in the fragment.

```
@Override public void onSaveInstanceState(Bundle savedInstanceState){
    savedInstanceState.putLong("universeId", universeId);
}
```

Then update onCreateView() to check to see if there was a saved instance and retrieve the universe id.

```
if (savedInstanceState !=null){
    universeId = savedInstanceState.getLong("universeId");
}
```