Lab 5: ComicViewer (Fragments)

Goal

In this lab, you'll practice using another UI pattern that uses fragments: a Tabbed Activity that uses a View Pager. We will also learn about a mechanism for doing expensive calculations off the UI thread: AsyncTasks.

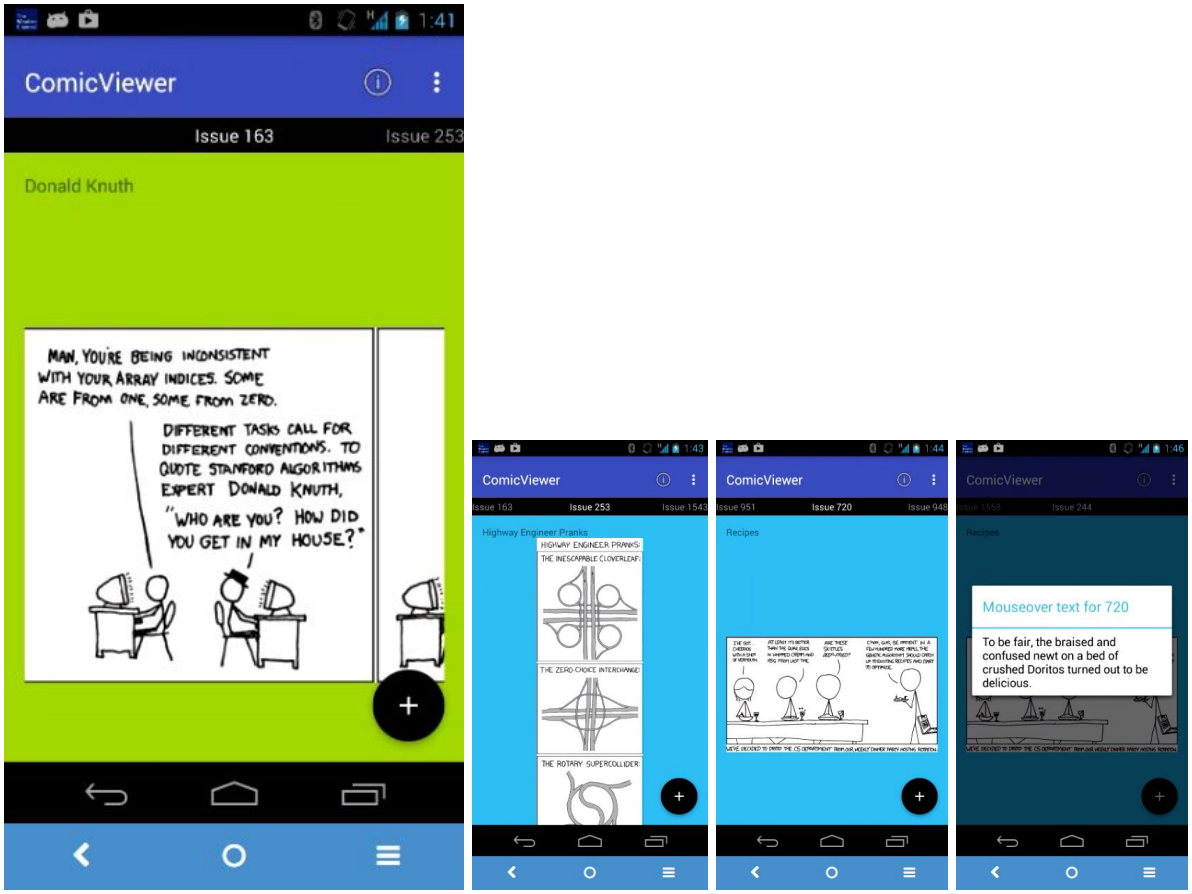
Heads-up: I think this lab is fun and very useful, but it is significantly longer than previous labs. I give you more guidance for the new topics, but please leave yourself enough time to get help as needed.

Requirements

1. (5 pts) When the app loads, the user is presented with the first of 5 jokes/comics displayed in a ViewPager. The user can swipe to the right to see others, and back to the original ones.
2. (1 pt) The background colors change in a predictable pattern, cycling through green, blue, yellow, and red as you move from left to right in the list of comics.
3. (1 pt) Pressing the FAB loads another comic.
4. (1 pt) Pressing an info menu item will launch a dialog that displays the mouseover alt-text (or the punchline, in the fallback "joke" option).
5. (1 pt) It displays a comic image.
6. (1 pt) The comic can be zoomed and scrolled. This is helpful on smaller devices like phones.
7. Your app must look exactly like the screenshots if you are using a phone. If your image is resized on a tablet, feel free to change font and image sizes as needed to display nicely.
8. If the screen is rotated, the app doesn't crash.
9. Use string resources for all text displayed on the screen. Recall the alt-shift hint.
10. If it loads xkcd comics, it uses clean ones (see given list).

Screenshots/Testing

For your demo, show similar screenshots to these.



a. Starting the app shows a comic, title, and issue number.
b. Scrolling right shows a different comic in a different color. the issue numbers are random, but the color sequence is regular.
c. Pressing the FAB creates a 6th comic (same color as 2nd comic, since 5 ≡ 1 (mod 4).
d. Pressing the info menu item shows the mouseover alt-text.

Hints/guidelines

Since this is a new UI pattern, I'll guide you through the process I used while creating the app. In summary, I (1) customized the starting code so the ViewPager displayed issue numbers, (2) implemented the FAB, (3) loaded xkcd json data, (4) displayed the image, and (5) took care of scroll/zoom.

View Pager

I found that when I created the app, that the Tabbed Activity template (the last option) was very helpful, and strongly suggest you use it. Choose the default names and setting. On the last

page, we'll choose the default Swipe Views (ViewPager) but preview the ActionBarTabs and ActionBarSpinner on that page first).

It's a good idea when playing with a new template to run it as-is to see how it behaves. Do that now.

To learn the most from this lab, I want you **right now** to examine at least the following files it stubbed in and see how much you understand. Then read on to check your understanding.

activity_main.xml,
fragment_main.xml,
MainActivity.java onCreate()
PlaceholderFragment (an inner class in MainActivity)
SectionsPagerAdapter (also an inner class in MainActivity)

You should always strive to be an active learner, talking to yourself as you work and judging how well you understand something.

If you have looked at the code, congrats for taking charge of your learning. And if not, please go back to the code now and read it before you read on.

Discussion

It created this template code.
res/layout:

activity_main.xml. Most of this is just a coordinator layout with toolbar and FAB, so should look familiar. But rather than containing content_main.xml, it has a ViewPager with id=container.

fragment_main.xml. Just a RelativeLayout with a TextView, but we'll customize it to show jokes and images.

MainActivity.java. MainActivity's onCreate() is typical, but it also creates the Adapter, captures the ViewPager and binds the two together. It has two important inner classes:

PlaceholderFragment. It's methods should look really familiar. Note (1) only a default constructor. (2) a newInstance method to pass data in. In this case, it just passes in a section number (currently 1, 2, or 3) to display it. In HistoricalDocs, we passed in a … Doc object. Here, we'll pass in a Comic. (3) an onCreateView() method that inflates the fragment_main.xml file and populates it.

SectionsPagerAdapter. This class has 3 methods. **getItem()** returns a new fragment using the newInstance pattern. **getCount()** is just like that of any other adapter. We won't use **getPageTitle()** since they are only used when you use the ActionBar Tabs and ActionBar Spinner options, and they will look bad on a phone, since we'll have too many tabs. (I tried.)

SectionsPagerAdapter extends FragmentPagerAdapter, and according to the javadoc, “keeps every loaded fragment in memory”. Sounds like a good place to start for this app.

I bet you understood the starting code pretty well!

I recommend you rename the classes to **ComicFragment** and **ComicsPagerAdapter**. I also moved them to their own files. It helped later on with the static newInstance() method, but there may be other ways to get it to work.

Displaying Issue numbers

We want to show 5 comics. We’ll download the comic from the xkcd site shortly, so what we are doing now is just a placeholder to make sure you can get the fragments working with a model object. I created a “ComicWrapper” model object to do this, with a hardcoded list of 5 ComicWrappers in my ComicsPagerAdapter .

A ComicWrapper object has an integer `xkcdIssue` (the 614 in https://xkcd.com/**614**/). It also has a int `color`, cycling through those specified above. I used some built in ones: `android.R.color.holo_green_light`, `android.R.color.holo_blue_light`, `android.R.color.holo_orange_light`, `android.R.color.holo_red_light`

When you create new ComicWrappers, grab a new color. Also grab a new random issue number from the list using a method in Utils.java. (These are my “sanctioned” xkcd issues - to my knowledge, they are mostly tech-focused and not very crude. If you find any to be offensive ones, please let me know and I’ll remove them from the list. Or if you have other clean favorites, please send them my way too! :))

Now is the time to show off your understanding of fragments. To pass a ComicWrapper to the ComicFragment, you’ll need to update the Fragment’s newInstance() factory method to take a ComicWrapper parameter, which also means ComicWrapper needs to be parcelable. All stuff we did earlier in this unit.

Update the ComicFragment to display the issue number in some way and to use the background color passed.

[One neat way to do this is to add a PagerTitleStrip to your ViewPager - it will autofill with whatever your adapter’s getPageTitle method returns. See here.]

Make sure you can scroll through the 5 pages and make sure they change colors and issue numbers.

Implement the FAB

Easy; just add another ComicWrapper with a random issue number and the correct background color.

Fallback option if you are short on time. You have done the heavily fragment-based part of the lab. If you want to skip the xkcd's, you may add a string to your ComicWrapper model and hardcode some jokes into it. If you make a menu item to show the punchline (hint below), then you'll be up to 8 of 10 points. Continue on only if you think xkcd is cool enough or if you want to learn about AsyncTasks to read json and images from online sources.

xkcd issue format and Comic class

Randall Munroe has made every issue available through a standard **secured (note the https in the URLs)** interface, with each issue represented in json (https://xkcd.com/about/). For example, https://xkcd.com/614/ (full page) is served by: https://xkcd.com/614/info.0.json, this json object:


```
/* from http://xkcd.com/614/info.0.json:
{"month": "7",
"num": 614,
"link": "",
"year": "2009",
"news": "",
"safe_title": "Woodpecker",
"transcript": "[[A man with a beret and a woman are standing on a boardwalk, leaning on a
handrail.]]\nMan: A woodpecker!\n<<Pop pop pop>>\nWoman: Yup.\n\n[[The woodpecker is banging its head
against a tree.]]\nWoman: He hatched about this time last year.\n<<Pop pop pop pop>>\n\n[[The woman walks
away.  The man is still standing at the handrail.]]\n\nMan: ... woodpecker?\nMan: It's your
birthday!\n\nMan: Did you know?\n\nMan: Did... did nobody tell you?\n\n[[The man stands,
looking.]]\n\n[[The man walks away.]]\n\n[[There is a tree.]]\n\n[[The man approaches the tree with a
present in a box, tied up with ribbon.]]\n\n[[The man sets the present down at the base of the tree and
looks up.]]\n\n[[The man walks away.]]\n\n[[The present is sitting at the bottom of the tree.]]\n\n[[The
woodpecker looks down at the present.]]\n\n[[The woodpecker sits on the present.]]\n\n[[The woodpecker
pulls on the ribbon tying the present closed.]]\n\n((full width panel))\n[[The woodpecker is flying, with
an electric drill dangling from its feet, held by the cord.]]\n\n{{Title text: If you don't have an
extension cord I can get that too.  Because we're friends!  Right?}}",
"alt": "If you don't have an extension cord I can get that too.  Because we're friends!  Right?",
"img": "http:\/\/imgs.xkcd.com\/comics\/woodpecker.png",
"title": "Woodpecker",
"day": "24"}
*/
```


Of note are the title (Woodpecker"), the mouseover text (the "alt" tag), and a link to the image (img tag): http://imgs.xkcd.com//comics//woodpecker.png

We can load the comics using this format.

First, I looked for a json to object mapper. I found Jackson, a standard tool for this. (Incidentally, it's also the one used by Firebase, which we will study in the next unit.) Jackson ObjectMapper class includes a convenient readValue method:

```
Comic comic = new ObjectMapper().readValue(new URL(urlString), Comic.class);
```

readValue() takes a URL to read and a class it can use as a template to deserialize the json into, and returns an object of that class. Beautiful!

To use this, you'll need a Comic class (in a separate file) with fields named the same as the json object.

```
public class Comic {
   private int num;
   private int month;
   private int day;
   private int year;
   private String link;
   private String news;
   private String transcript;
   private String safe_title;
   private String alt;
   private String img;
   private String title;
```

Create getters and setters for each field, a default (no-argument) constructor, and a toString() method for debugging. (Recall that you can do this quickly using alt-insert (cmd-N in Mac) in Android Studio.)

Also add the following dependencies to your gradle file (details):

```
dependencies {
   implementation fileTree(dir: 'libs', include: ['*.jar'])
   testImplementation 'junit:junit:4.12'
   implementation 'com.android.support:appcompat-v7:27.1.0'
   implementation 'com.android.support:design:27.1.0'
   implementation(
           [group: 'com.fasterxml.jackson.core', name: 'jackson-core', version: '2.4.1'],
           [group: 'com.fasterxml.jackson.core', name: 'jackson-annotations', version: '2.4.1'],
           [group: 'com.fasterxml.jackson.core', name: 'jackson-databind', version: '2.4.1']
   )
   implementation 'com.github.chrisbanes:PhotoView:2.1.3'
}
```

and add packaging options to the top of the gradle file (I am not sure why).

```
android {
   compileSdkVersion 27

   packagingOptions {
       exclude 'META-INF/LICENSE'
       exclude 'META-INF/NOTICE'
   }
```

Finally, since this will download from the internet, you'll need to give your app permission to access the internet. In your manifest, outside the application tag, add:

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

References for Jackson:
http://www.journaldev.com/2324/jackson-json-processing-api-in-java-example-tutorial
http://stackoverflow.com/questions/25265407/add-boon-or-jackson-json-to-android-studio-with-gradle

Android best practice is to do any network operations **off** the UI thread. So we'll use an **AsyncTask** to load the Comic.

AsyncTasks and Supplementary Video

Note: as the request of my past students, I made an optional video on AsyncTasks here (accompanying slides here) to supplement the following instructions. About 5 minutes is introducing the idea of AsyncTasks and 16 minutes is coding a basic version of the next part of the lab. While it's optional (no quiz and nothing to submit), you might find it helpful.

AsyncTasks are computations that run in the background, on a **different** thread than the UI. Let's write one to load the comic. We want this to run in the background, and once it is complete, we want to update that humor object's Comic object.

Start by creating **getComicTask**, a subclass of AsyncTask. AsyncTask is a generic class that takes three generic type parameters: the type of the input to the task, the type of the progress and the type of the result. In our case, we want a method that takes a url (String) and returns a Comic object; we don't care about progress, so we use type Void:

```
class GetComicTask extends AsyncTask<String, Void, Comic> {
```

You are required to override the doInBackground() method, so we start there with the network task we want to perform. If it weren't for exception handling, we could write the one liner above. Instead, we catch the exception. The syntax for the input argument, String… urlStrings may seen strange to you; it means that there are a variable number of strings coming in as an array. We only call the task with a single string, so we take array element 0:

```
@Override
protected Comic doInBackground(String... urlStrings) {
    String urlString = urlStrings[0];
    Comic comic = null;
    try {
        comic = new ObjectMapper().readValue(new URL(urlString), Comic.class);
    } catch (IOException e) {
        Log.d(Constants.TAG, "ERROR:" + e.toString());
    }
    return comic;
}
```

You'll notice that I used a Log message in here. I find that when working with network tasks, it helps to include lots of Log messages when developing to make it easier to debug.

Next, we must process the Comic once it is completed. We do this by overriding the AsyncTask's **onPostExecute()** method. It is passed the Comic object. It must use that info in a couple ways: first, add a Comic field to the ComicWrapper class and store the Comic there, so you'll be able to show the title and mouseover text later. Second, kick off a second AsyncTask to load the image. Each fragment has its own ComicWrapper object, so the AsyncTask will need to store that fragment and tell it to do these things when it finishes executing. I chose to use an interface for this. (If your GetComicTask is an inner class to the fragment, this isn't necessary.)

Here is the whole GetComicTask. I give it to you since it is your first one.

```
class GetComicTask extends AsyncTask<String, Void, Comic> {
    private ComicConsumer mComicConsumer;

    public GetComicTask(ComicConsumer activity) { mComicConsumer = activity; }

    @Override
    protected Comic doInBackground(String... urlStrings) {
        String urlString = urlStrings[0];
        Comic comic = null;
        try {
            comic = new ObjectMapper().readValue(new URL(urlString), Comic.class);
        } catch (IOException e) {
            Log.d(Constants.TAG, "ERROR:" + e.toString());
        }
        return comic;
    }

    @Override
    protected void onPostExecute(Comic comic) {
        super.onPostExecute(comic);
        mComicConsumer.onComicLoaded(comic);
    }

    public interface ComicConsumer {
        public void onComicLoaded(Comic comic);
    }
}
```

In onComicLoaded(), you'll need to display the title.

```
@Override
public void onComicLoaded(Comic comic) {
    Log.d("COMIC", "Comic Object\n" + comic);
    mComicWrapper.setComic(comic);
    mTextView.setText(comic.getSafe_title());
}
```

Finally, to run the GetComicTask, you need to make an instance of one and tell it to execute. I

did this in the fragment's onCreate() method, since that's the first place it had a ComicWrapper with the issue number that could be used as a URL.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (getArguments() != null) {
        mComicWrapper = getArguments().getParcelable(ARG_COMIC);
        String urlString = String.format("http://xkcd.com/%d/info.0.json", mComicWrapper.getIssue());
        new GetComicTask(this).execute(urlString);
    }

}
```

Update: change the http to https in the code above.

Run it now. You'll need an Internet connection for it to work.

Loading and displaying images

You'll add an ImageView to your fragment's layout, and capture it in code. To populate it, you'll load it from the URL of the actual comic, which is an image. You'll create another AsyncTask to load this. I suggest executing the task at the end of onComicLoaded(). The actual GetImageTask class will be similar to the GetComicTask, except for what it does in the background. The key lines are these:

```
InputStream in = new java.net.URL(urlString).openStream();
bitmap = BitmapFactory.decodeStream(in);
mImageView.setImageBitmap(mBitmap);
```

The first two lines go in doInBackground(), the third goes in the fragment's onImageLoaded() callback method that you'll write.

I leave you mostly on your own for this one, since it's so similar to the GetComicTask.

Test it.

Scrolling/zooming the image

The comic images show up pretty small, so you'll want to be able to zoom and scroll through them. After considering options, I chose this third party option. That link contains the gradle dependency you need to add (you need to change **both** of your gradle files) and an example usage of attaching the PhotoViewAttacher to your ImageView.

Showing the mouseover (alt text) on command

This is easy: just make an options menu item to show it and when they click it, then use it. You probably want to do this in the Fragment class by adding an onOptionsItemSelected() method there. There's one simple catch: whereas Activities are assumed to have options menus, you

need to tell the fragment that it has an options menu. In its onCreate, call

setHasOptionsMenu(true).

Your dialog just displays the mouseover text.

Test it out… xkcd goodness?

Credits
To Randall Munroe for xkcd.

Joke Credits
I grabbed jokes here when first creating the “fallback option” as the idea to load comics was first materializing in my mind.
http://stackoverflow.com/questions/234075/what-is-your-best-programmer-joke It is sad that stackoverflow is even a source of jokes…