CSCI 448 – Lab 05A
Friday, February 7, 2020
LAB IS DUE BY **Friday, February 21, 2020 2:50 PM**!!

Congratulations! After successfully completing your first app, GeoQuiz, your app knowledge has been noticed and you've recently been hired by WhoDunIt Inc. They have been working on a new app called CriminalIntent and have put you on the development team.

This app will allow the user the opportunity to create a record of when a crime occurred. In the future, we will add in who the suspect is while making use of additional Android features.

There will be a bit of a delay before the big payoff – there are several features that need to be implemented and the development team has a big presentation on St. Patrick's Day. The team has several sprints they'll push out every few weeks to show progress towards the big unveiling.

The three sprint cycles are:

- Sprint 1 (Lab05): Entering and viewing a list of crimes
- Sprint 2 (Lab06): Creating a side-by-side view of crimes
- Sprint 3 (Lab07): Choosing a suspect and sending the crime report

Lab05 will have 4 parts:

A. Creation of the Crime Detail interface
B. Creation of the Crime List interface
C. Populating the List with a known Database
D. Connecting the List to the Detail

It won't be until Lab06 that we see the benefit of structuring our app with Fragments instead of solely Activities, but we need the initial framework in place. Here are your first set of stories to complete for Lab05A:

- The MainActivity needs to host CrimeDetailFragment
- When the app opens, the user needs to be able to enter the crime information

Step 0 – Take a look around

As a new hire, some starter code has been given to you. There are some files to be aware of:

- `res/values/string.xml` – contains string labels that are used in layouts
- `res/layout/fragment_detail.xml` – contains the premade Crime Detail layout, will be used in this lab
- `java/data/Crime.kt` – class definition for a single Crime instance
- `java/ui/MainActivity.kt` – this class will host all our Fragments and contain the Fragment Manager
- `java/ui/detail/CrimeDetailFragment.kt` – this class will contain the actual Crime Details Controller

Run the app. You should just see a "`Hello World`" displayed.

Let's hook everything up. There will be two files we'll be working with:

1. `java/ui/detail/CrimeDetailFragment.kt` – we will need to complete this class to act as our primary controller and display the Crime Detail layout
2. `java/ui/MainActivity.kt` – we will need to complete this class to host the fragment

Step 1 – MainActivity needs to host CrimeDetailFragment

From this point forward, all of our apps (labs, assignments, final projects) will use Fragments. Very frequently, our Activity will be hosting only one Fragment. For these situations, our setup will always be the same.

Part I – Creating the Layout

The lead software engineer has left you some documentation on what the layout will need to look like. Refer to BNR Listing 8.11 for the layout structure and supporting information. This change will take place at TODO 1. (*Hint: there is a TODO tab at the bottom of Android Studio that lists all of the TODO comments in your project. Clicking the TODO opens the file.) (Hint 2: TODO 1 is in activity_main.xml)*

Part II – Host the Fragment

Just like the OS has the `ActivityManager` to manage activities, each Activity has a `FragmentManager` to manage fragments. Your lead has left you more notes surrounding Listing 8.12. The steps are also broken out below.

We will ask the `supportFragmentManager` (since we are using SupportFragments from the Support Library) if there currently exists a fragment with the id of our fragment container. This result could be null if we haven't created a fragment yet. We'll now fill in TODO 2.

```
val currentFragment =
        supportFragmentManager.findFragmentById(R.id.fragment_container)
```

If the returned fragment is not null, then there's nothing we need to do since the fragment exists already. We are only concerned if the fragment is null, in which case we will now call our function `createFragment()` function.

```
val currentFragment =
        supportFragmentManager.findFragmentById(R.id.fragment_container)
if(currentFragment == null ) {
    val fragment = CrimeFragment()
}
```

Lastly we will create a fragment transaction, add the fragment, and commit the transaction. By creating transactions, the FragmentManager can keep a history stack to support back functionality.

```
val currentFragment =
          supportFragmentManager.findFragmentById(R.id.fragment_container)
if(currentFragment == null ) {
    val fragment = createFragment()
    supportFragmentManager
          .beginTransaction()
          .add(R.id.fragment_container, fragment)
          .commit()
}
```

At this point, we can run the app. We should see nothing. That is to be expected. Once the next story is complete we'll see the new layout.

Step 2 – When the app opens, the user needs to be able to enter the crime information

The last step is to replicate the functionality that we used to place in the activity – namely connecting the Model to the View.

Before we add the Controller functionality, let's first interface with the Fragment methods.

Part I – Log the Fragment Life Cycle

Just like we have logged the Activity Life Cycle methods, we will also log all of the Fragment Life Cycle methods. These mimic the activity methods.

Override each of the following functions, call the super, and print a Log message that each has occurred.

- `onAttach(context: Context)`
- `onCreate(savedInstanceState: Bundle?)`
- `onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View?`
- `onViewCreated(view: View, savedInstanceState: Bundle?)`
- `onActivityCreated(savedInstanceState: Bundle?)`
- `onStart()`
- `onResume()`
- `onPause()`
- `onStop()`
- `onDestroyView()`
- `onDestroy()`
- `onDetach()`

Part II – Set the View

When working with an activity, we set the view layout to use in the `onCreate()` method. When working with fragments, the process is a bit different. Since fragments can be created and reused, the view setting and creating is handled separately in two additional fragment life cycle methods.

The first method is onCreateView(). It takes three parameters and returns a View. This function looks like and is duplicated in the lead's documentation book Listing 8.6:

```
override fun onCreateView(inflater: LayoutInflater,
                          container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
  Log.d(LOG_TAG, "onCreateView() called")
  return super.onCreateView(inflater, container, savedInstanceState)
  val view = inflater.inflate(R.layout.fragment_detail, container, false)
  return view
}
```

Note we are **not** calling a super method here. The inflater object is responsible for taking our XML layout and exploding it into objects and applying all the XML attributes to the object properties. We then return the root node object of this layout.

We're now ready to test everything! Run and install the app. We should see the crime details layout. Fantastic! Let's plug in some data now.

Part III – Hook up the Controller

This final part will not have any effect until we finish Lab05D, but we will put it in place to see more of the Fragment's Life Cycle. First let's add a reference to our Model. Since this Fragment is showing the details for a single crime, we will add a private property to the class for a Crime object. Normally when creating a class property, we have to assign it a value, but we do not know what the value will be until the Fragment is created. We are going to use a new qualifier for this property, the lateinit.

```
private lateinit var crime: Crime
```

The lateinit will not create the variable until the first time it is referenced. At that point, we will assign a value. When does that happen? In the Fragment `onCreate()`. After logging the call, set our crime variable to be a new crime object.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d(logTag, "onCreate() called")
    crime = Crime()
}
```

The Model is now connected to the Controller. The last step is to connect the Controller to the View – setting dynamic text fields and connecting event listeners.

For this step, we can follow the book for BNR Listings 8.7 – 8.10. You will need to change some of the IDs to match the code started by the original development team.
Run your app again, take notice the date has been placed on the button, and sleep well knowing that any changes made to the View are currently being backed in the Model.

At this point, our app matches the end of BNR Chapter 8, though we have structured it differently to satisfy our two stories.

LAB IS DUE BY **Friday, February 21, 2020 2:50 PM**!!