CSCI 448 – Lab 12B
Friday, April 21, 2023
LAB IS DUE BY **Friday, April 28, 2023 11:59 PM**!!

To complete the notification process, we need to bring the user back into the app and show them where they are. This will be performed via a NavGraph with a Deep Link set to a specific location.

Step 1 – Setup the NavGraph

Part 1.I – Add Dependencies

To use the navigation components, we need to add the navigation dependencies to our Module `build.gradle` file:

```
implementation 'androidx.navigation:navigation-runtime-ktx:2.5.3'
implementation 'androidx.navigation:navigation-compose:2.6.0-alpha09'
```

Part 1.II – Create the NavGraph

As with much of this app, we'll not follow best practice to encapsulate our NavGraph and screen specifications. Instead, we'll opt for the quicker proof of concept approach. In `MainActivity setContent` we'll need to wrap our `LocationScreen` call with a `NavHost` and `composable`.

Begin by creating a constant in the `companion object` (we'll add to this list as we go on):

```
private const val ROUTE_LOCATION = "location"
```

Now, put the NavGraph in place.

```
val navHostController = rememberNavController()
NavHost(
  navController = navHostController,
  startDestination = ROUTE_LOCATION
) {
  composable(
    route = ROUTE_LOCATION
  ) {
    LocationScreen(...)
  }
}
```

Rerun the app and nothing should change from before.

Step 2 – Utilize a Deep Link

We are going to perform two separate, but related, steps concurrently:

1. Provide a starting location to display on our map without the user pressing get location
2. Implement the ability to navigate to a specific destination within our NavGraph

Part 2.I – Add Navigation Arguments

The argument names will be referenced in several places, so add these as additional constants.

```
private const val ARG_LATITUDE = "lat"
private const val ARG_LONGITUDE = "long"
```

Then add two string arguments to the composable destination:

```
composable(
  route = ROUTE_LOCATION,
  arguments = listOf(
    navArgument(name = ARG_LATITUDE) {
      type = NavType.StringType
      nullable = true
    },
    navArgument(name = ARG_LONGITUDE) {
      type = NavType.StringType
      nullable = true
    }
  )
) { navBackStackEntry ->
  ...
}
```

Be sure to capture the `navBackStackEntry` in the body of the composable. We'll now use this to get the values of the arguments when navigated to.

```
navBackStackEntry.arguments?.let { args ->
  val lat = args.getString(ARG_LATITUDE)?.toDoubleOrNull()
  val long = args.getString(ARG_LONGITUDE)?.toDoubleOrNull()
  if (lat != null && long != null) {
    val startingLocation = Location("").apply {
      latitude = lat
      longitude = long
    }
    locationUtility.setStartingLocation(startingLocation)
  }
}
```

We will use the arguments to tell the `LocationUtility` our starting location. Add the referenced `setStartingLocation` method to `LocationUtility`. This method will accept a nullable `Location` object and use it to update the private current location state.

The app can now be rerun and again there should be no change, or error, from last time.

Part 2.II – Specify The DeepLink

We'll now create a deep link to navigate to the corresponding destination AND specify the latitude/longitude to load.

The deep link is specified by a URI. Add more constants to store the components of the URI:

```
private const val SCHEME = "https"
private const val HOST = "geolocatr.labs.csci448.mines.edu"
private const val BASE_URI = "$SCHEME://$HOST"
```

The template for the complete URI will follow:

```
SCHEME://HOST/ROUTE/{ARG1}/{ARG2}
```

which for our specific destination would be

```
https://geolocatr.labs.csci448.mines.edu/location/{lat}/{long}
```

and a specific route to navigate to would follow the form

```
https://geolocatr.labs.csci448.mines.edu/location/37.4223983/-122.0847983
```

Create a private method that builds the full URI to correspond to the destination.

```
private fun formatUriString(location: Location? = null): String {
  val uriStringBuilder = StringBuilder()
  uriStringBuilder.append(BASE_URI)
  uriStringBuilder.append("/$ROUTE_LOCATION/")
  if (location == null) {
    uriStringBuilder.append("{$ARG_LATITUDE}")
  } else {
    uriStringBuilder.append(location.latitude)
  }
  uriStringBuilder.append("/")
  if (location == null) {
    uriStringBuilder.append("{$ARG_LONGITUDE}")
  } else {
    uriStringBuilder.append(location.longitude)
  }
  return uriStringBuilder.toString()
}
```

The last part of the deep link specification is to put it on the composable destination:

```
composable(
  route = ROUTE_LOCATION,
  arguments = ...,
  deepLinks = listOf(
    navDeepLink { uriPattern = formatUriString() }
  )
) { ... }
```

Part 2.III – Navigate To The DeepLink

To launch to a specific deep link destination, we need an explicit intent that corresponds to the URI destination. Create a public static method in the MainActivity companion object called createPendingIntent.

```
fun createPendingIntent(context: Context, location: Location): PendingIntent {
  val deepLinkIntent = Intent(
    Intent.ACTION_VIEW,
    formatUriString(location).toUri(),
    context,
    MainActivity::class.java
  )
  return TaskStackBuilder.create(context).run {
    addNextIntentWithParentStack(deepLinkIntent)
    getPendingIntent(
      0,
      PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE
    )
  }
}
```

Inside LocationAlarmReceiver::onReceive when we are making the notification we now need to attach the pending intent to be launched when the user clicks on the notification.

```
val startingLocation = Location("").apply {
  latitude = lat
  longitude = long
}
val deepLinkPendingIntent = MainActivity
  .createPendingIntent(context, startingLocation)
val notification = NotificationCompat.Builder(context, channelId)
  ...
  .setContentIntent(deepLinkPendingIntent)
  .setAutoCancel(true)
  .build()
```

At this point, run the app and when the notification appears be sure you have closed the app. From your home screen, click on the notification. The notification will disappear and you'll be brought into the app. The location info will display at the top but the map may not update.

Part 2.IV – Autoupdate the Map

If the map takes a long time to load, then the camera cannot be updated until the map is ready. In the LocationScreen composable, add one more mutable state object

```
val mapReadyState = remember { mutableStateOf(false) }
```

The `GoogleMap` composable has another argument we can leverage, `onMapLoaded`. This will fire after the map is loaded and ready to be interacted with. Supply a function for this argument to update the `mapReadyState` to `true`.

```
GoogleMap(
  ...,
  onMapLoaded = { mapReadyState.value = true }
) { ... }
```

What do we do with this state? We provide it to our LaunchedEffect as well. Now, if the location changes or the mapReadyState changes we'll update our camera to go to the location (if it's not null).

```
~~LaunchedEffect(location) {~~
LaunchedEffect(location, mapReadyState.value) {
  ...
}
```

Rerun the app, click the notification, and you should have a smooth experience to see where you were 10 seconds ago.

Step 3 - Submission

You will want to start with a fresh install before any permissions have been granted. Submit a video with the following features for the final sign off:

- Open the app
- Press the get location button
- *(notify button becomes enabled)*
- Press the notify me button
- Go to recents and close the app
- *(…Wait for notification to appear…)*
- Pull down top window shade
- Press notification
- (*user is brought back into app with location auto populated)*

Please name this video file `<your_user_name>_L12`. For instance, my submission would be `jpaone_L12.webm`.

LAB IS DUE BY **Friday, April 28, 2023 11:59 PM**!!