CSCI 448 – Lab 10A
Monday, March 27, 2023
LAB IS DUE BY **Friday, April 07, 2023, 11:59 PM**!!

Little Green Games was very impressed with your work on Samodelkin. They are now anticipating adding location based features to the game. The goal is to find other Samodelkin players in your vicinity and have your characters battle if you are close enough to each other.

As a proof of concept, they have asked you to develop an app that will track a user's location and plot it on a map. We'll take a more slimmed down view to our app as it will consist of a single screen – so no `NavGraph`, `Scaffold`, `IScreenSpec`, ViewModel, etc. In practice, what we are putting into MainActivity should properly be put into a `LocationScreenSpec` object. We'll be removing that layer of abstraction for simplicity in this smaller lab.

This lab will set up querying the device's location and plot that location on a map. We'll also do some other location related activities for good UX and features. Lab10A will be the longer piece since it requires getting the app set up and requesting permission for the location. Lab10B is more straight forward setting up the Map and interacting with it.

Step 0 – Application Setup

This step is largely a review of prior labs. It is up to you to determine the structure and file organization. Create a new `Empty Compose Activity (Material3)` project and name it `GeoLocatr`.

In the `build.gradle (Project)` file, update the `compose_version` to `1.4.0` and set the `kotlin` version to `1.8.0`.

In the `build.gradle (Module :app)` file, set the kotlin compiler to version `1.4.1`. Then update all dependencies to their latest versions. Additionally, add the following dependencies:

```
implementation 'androidx.lifecycle:lifecycle-runtime-compose:2.6.1'
implementation 'com.google.android.gms:play-services-location:21.0.1'
```

Step 1 – Get the `LocationScreen` and `MainActivity` in place

Part 1.I – Create the `LocationScreen`

Create a file named `LocationScreen.kt`. Create a composable function named `LocationScreen` that accepts the following parameters:

- A nullable `Location` called `location` `Location?`
- A `Boolean` called `locationAvailable` `Boolean`
- A function object with no input or output named `onGetLocation` `() -> Unit`
- A `String` called `address` `String`

Inside the function, setup a `Column` with the following composables nested inside:

- A `Text` displaying "`Latitude / Longitude`"

- A `Text` displaying the `latitude` and `longitude` values of the `location` parameter
- A `Text` displaying `"Address"`
- A `Text` displaying the `address` parameter
- A `Button` with
    - The `enabled` property set to the `locationAvailable` parameter
    - When clicked invokes the `onGetLocation` function parameter
    - Has a text label of `"Get Current Location"`

Set up a `Preview` composable to test your layout. If you followed the above naming scheme, the following function will interface with the function:

```
@Preview(showBackground = true)
@Composable
private fun PreviewLocationScreen() {
  val locationState = remember { mutableStateOf<Location?>(null) }
  val addressState = remember { mutableStateOf("") }
  LocationScreen(
    location = locationState.value,
    locationAvailable = true,
    onGetLocation = {
      locationState.value = Location("").apply {
        latitude = 1.35
        longitude = 103.87
      }
      addressState.value = "Singapore"
    },
    address = addressState.value
  )
}
```

Use the interactive mode and press the button to ensure your view is laid out as desired.

Part 1.II – Setup `MainActivity`

Refactor `MainActivity` to call `LocationScreen`. The `setContent` block should be structured similarly to the preview function above. Deploy your app to a device and verify the button updates the displayed location and address as it did in the preview.

We're now ready to get the device's true location and display its information.

Step 2 – Get the Location

Part 2.I – Update the Manifest

In order to get the user's location, we need to add the following two permissions to the manifest:

- `ACCESS_FINE_LOCATION`
- `ACCESS_COARSE_LOCATION`

Part 2.II – Create the `LocationUtility`

Create a class called `LocationUtility`. This will handle all the Location related functionality. It's constructor should accept, but not store, a Context.

```
class LocationUtiity(context: Context) { }
```

Part 2.II.A – Add State

Add two `StateFlow` objects – one for the nullable `Location` object and one for the address `String`. Best practice would dictate a private `MutableStateFlow` and a public `StateFlow`.

```
private val mCurrentLocationStateFlow: MutableStateFlow<Location?>
     = MutableStateFlow(null)
val currentLocationStateFlow: StateFlow<Location?>
  get() = mCurrentLocationStateFlow.asStateFlow()

// repeat for mCurrentAddressStateFlow
```

Part 2.II.B – Ask for Permission

Now add a public method called `checkPermissionAndGetLocation`. It will accept two parameters:

- An `Activity` object named `activity`
- An `ActivityResultLauncher<Array<String>>` object named `permissionLauncher`

We will use both of these to check if the user has given permission to access their location The pseudocode for this process is as follows:

```
// check if permissions are granted
if activity.checkSelfPermission( ACCESS_FINE_LOCATION ) == PERMISSION_GRANTED
      || activity.checkSelfPermission( ACCESS_COARSE_LOCATION ) == PERMISSION_GRANTED
  // Section 1
  // permission has been granted to do what we need
  Log that we have permission
else
  // permission is currently not granted
  // check if we should ask for permission or not
  if ActivityCompat.shouldShowRequestPermissionRationale(activity, ACCESS_FINE_LOCATION)
      || ActivityCompat.shouldShowRequestPermissionRationale(activity, ACCESS_COARSE_LOCATION)
    // Section 2
    // user already said no, don't ask again
    Log permission was denied
    Display Toast "We must access your location to plot where you are"
  else
    // Section 3
    User hasn't previously declined, ask them
    Log asking for permission
    Launch permissionLauncher for ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION
```

With the permission check shell in place, we can now begin filling in gaps.

Part 2.II.C – Get the Location

To make the asynchronous location request, we'll need to setup both the request object and the callback object. Add the following private data members to the `LocationUtility` class.

- `locationRequest` : use the `LocationRequest.Builder` to specify a `PRIORITY_HIGH_ACCURACY` with zero millisecond interval request that will retrieve at most one update
- `locationCallback` : implement `LocationCallback` and override `onLocationResult`. Update the current location `StateFlow` with the last location that was retrieved.
- `fusedLocationProviderClient` : get from `LocationServices` for the `context` that was provided to the constructor

With those in place, we can now fill in the needed step for Section 1 to make the location request.

```
fusedLocationProviderClient
  .requestLocationUpdates(locationRequest,
                          locationCallback,
                          Looper.getMainLooper())
```

Part 2.III – Use the `LocationUtility`

We'll now switch back to `MainActivity` to integrate the `LocationUtility` and provide the ability to get the location properly.

First add a private `lateinit var` data member to the `MainActivity` class for the `LocationUtility`. Also add a private `lateinit var` data member for our `permissionLauncher` of type `ActivityResultLauncher<Array<String>>`.

Inside of `onCreate()`, create the `LocationUtility` instance.

Next, we'll create the `permissionLauncher` to register for multiple permission requests.

```
permissionLauncher =
  registerForActivityResult(ActivityResultContracts.RequestMultiplePermissions()) {
    // process if permissions were granted
  }
```

There's a number of steps to unpack here:

1. The `permissionLauncher` object is of type `ActivtyResultLauncher`. When we launch the activity, it expects an input of an Array of Strings. (This is done in Section 3 when we ask for permission)
2. We then ask the activity to register itself with the OS to receive the result of another activity call.
3. The contract for the activity intent interaction is a system defined contract, one that allows for multiple permissions to be requested at once.
4. The result of the contact is a map where each key corresponds to the string representation of a permission. The corresponding value is a Boolean denoting if the associated permission was granted or not.
5. The trailing lambda we are about to provide is the implementation of the callback to process the result of the contract.

We'll come back to completing Step 5 in just a moment. Let's work through the user flow.

We'll need to collect the location and address state from the utility and provide this to `LocationScreen` to display.

```
~~val locationState = remember { mutableStateOf<Location?>(null) }~~
val locationState = locationUtility
     .currentLocationStateFlow
     .collectAsStateWithLifecycle(lifecycle = this@MainActivity.lifecycle)
// repeat for address
```

`LocationScreen` is where the location request is triggered. It triggers the implementation of `onGetLocation`. This is where we will now use our utility to check for permission and potentially get the location.

```
onGetLocation = {
  locationUtility.checkPermissionAndGetLocation(this@MainActivity,
                                                permissionLauncher)
}
```

What happens?

- In the app, when the user pressed the get location button it will first check if permission was previously given.
    - If yes, then the request is made and our location callback is called. This updates the location state, which then gets displayed on screen.
    - If no, then we launch the permission request. The launcher callback is called. Now to complete Step 5.

We could step through the map of permissions and check each one. We could then disable the button if we don't have permission. (But, we want the button to remain enabled so if they try again we can remind them about the permissions.) And from a UX view point, if they gave permission we don't want the user to have to press the get location button a second time to then get their location. So we will "cheat" and call our `locationUtility` method a second time to get the location.

```
permissionLauncher =
  registerForActivityResult(ActivityResultContracts.RequestMultiplePermissions()) {
    // process if permissions were granted
    locationUtility.checkPermissionAndGetLocation(this@MainActivity, permissionLauncher)
  }
```

This is smart decision since our function internally performs the permission check. It will know if we have permission or not. And, it will provide a seamless flow for the user. When returning from the permission ask, if they said yes they see the location. If they said no they see the toast why they should have said yes.

At this point build, deploy, run, and test the app. You should now be able to see the user's latitude and longitude once permission is given.

Part 2.IV – Be a Good Programmer

The last part of getting the location is to clean up our location request. What may occur, the user presses the get location button and the request hangs. During the time waiting for the location to come back, the user leaves the app. Then the location callback is called and ultimately it tries to update a variable that no longer exists.

Add to your `LocationUtility` class a `removeLocationRequest()` method. This will remove the callback from the provider.

```
fun removeLocationRequest() {
  fusedLocationProviderClient.removeLocationUpdates(locationCallback)
}
```

Now in `MainActivity`, override the `onDestroy()` method (which corresponds to when the `Activity` would be destroyed and the `LocationUtility` instance will be destroyed) and call the newly made function.

Step 3 – Get the Address

The latitude and longitude are not all that useful to the average user. We would much rather tell the user the street address of where they are. Android uses a `Geocoder` service that will decode a latitude and longitude to the nearest street address.

On `LocationUtility`, add a private data member for the `Geocoder` object.

```
private val geocoder = Geocoder(context)
```

Then add the following function to perform the lookup:

```
suspend fun getAddress(location: Location?) {
  val addressTextBuilder = StringBuilder()
  if (location != null) {
    try {
      val addresses = geocoder.getFromLocation(location.latitude,
                                               location.longitude,
                                               1)
      if (addresses != null && addresses.isNotEmpty()) {
        val address = addresses[0]
        for (i in 0..address.maxAddressLineIndex) {
          if (i > 0) {
            addressTextBuilder.append("\n")
          }
          addressTextBuilder.append( address.getAddressLine(i) )
        }
      }
    } catch (e: IOException) {
      Log.e(LOG_TAG, "Error getting address", e)
    }
  }
  mCurrentAddressStateFlow.update { addressTextBuilder.toString() }
}
```

We'll ignore the warning about `getFromLocation()` being deprecated. The new version is available in API 33, but we're supporting back to API 29 so the new solution does not apply. We do mark the method as a blocking function so we will perform the lookup on another coroutine.

We're ready to hook this up through `MainActivity`. In `MainActivity`, after we created our state objects and before we call the `LocationScreen` composable, we'll create a `LaunchedEffect` side effect. This will monitor `locationState`. When the `locationState` value changes, it will then trigger the call to update the `addressState` as well.

```
LaunchedEffect(locationState.value) {
  locationUtility.getAddress(locationState.value)
}
```

Run the app, get the location, and find out where you're standing!

Step 4 – Good UI/UX: Ensure Location is Available

Part of being good programmers is making sure the device is in the right configuration, i.e. does the user have location settings turned on. The system has the ability to inspect the location settings and enable location from within the app.

In `LocationUtility`, add another `StateFlow` member that is a `Boolean` if the location is available or not.

Add a public method that will parse a nullable `LocationSettingsStates` object to determine if the location is usable:

```
fun verifyLocationSettingsStates(states: LocationSettingStates?) {
  mIsLocationAvailableStateFlow.update { states?.isLocationUsable ?: false }
}
```

Then add the following method to `LocationUtility`:

```
fun checkIfLocationCanBeRetrieved(
  activity: Activity,
  locationLauncher: ActivityResultLauncher<IntentSenderRequest>
) {
  val builder = LocationSettingsRequest.Builder()
    .addLocationRequest(locationRequest)
  val client = LocationServices.getSettingsClient(activity)
  client.checkLocationSettings(builder.build()).apply {
    addOnSuccessListener { response ->
      verifyLocationSettingsStates(response.locationSettingsStates)
    }
    addOnFailureListener { exc ->
      mIsLocationAvailableStateFlow.update { false }
      if (exc is ResolvableApiException) {
        locationLauncher
          .launch(IntentSenderRequest.Builder(exc.resolution).build())
      }
    }
  }
}
```

The failure listener gets triggered if location settings cannot be retrieved because location is turned off. The exception returned then contains an intent that can be used to turn the location back on.

We will need to setup the call to this method in `MainActivity`.

Add another `private lateinit var` for the `locationLauncher` of type `ActivityResultLauncher<IntentSenderRequest>`. In `onCreate()`, register this launcher:

```
locationLauncher = registerActivityResult(
    ActivityResultContracts.StartIntentSenderForResult()
) { result ->
  if (result.resultCode == RESULT_OK) {
    result.data?.let { data ->
      val states = LocationSettingsStates.fromIntent(data)
      locationUtility.verifyLocationSettingsStates(states)
    }
  }
}
```

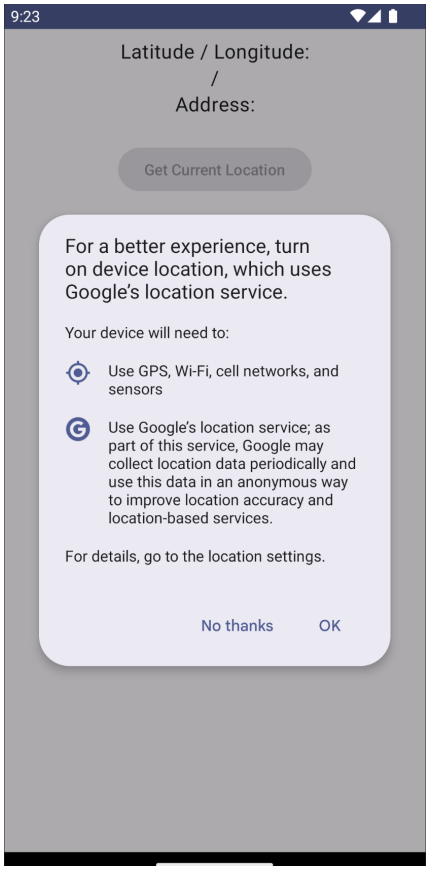
This callback will check if the location was turned on by the user or not.

When setting up the `LocationScreen` composable, collect the location available state and provide it to the `LocationScreen` call.

Finally, override `onStart()` and call the `LocationUtility` method:

```
locationUtility.checkIfLocationCanBeRetrieved(this, locationLauncher)
```

Build and deploy the app. Go into settings on the device and turn off the location. Then run the app. You should see a popup dialog similar to the image . Note that the location button is disabled. Once the user presses ok, the button becomes enabled and we can proceed as before.



Step 5 – Deploy Your App & Submit

At this point, `LocationUtility` is completed self-contained and can be ported between projects as long as we hook it up with the corresponding activity.

With location now working, you're ready to begin plotting Samodelkin players onto a map.

LAB IS DUE BY **Friday, April 07, 2023, 11:59 PM**!!