eecs441§3&4

Chatter with Maps and Programmatic UI Kotlin

Cover Page

DUE Wed, 10/13, 2 pm

The goal of this lab is two fold: First, to introduce you to programmatic UI development, without Layout Editor. Second, to integrate Google Maps with the Chatter app. We will first refactor out Layout Editor from our lab0 code. We'll assume that you're already familiar with ConstraintLayout and the concept of layout constraints.

In the maps-augmented Chatter app, we will add the **Map View**. On the map, there will be multiple markers. Each marker represents one `chatt`. If you click on a marker, it will display the poster's username, message, timestamp, and their **geodata**, consisting of their geolocation and velocity (compass-point facing and movement speed), captured at the time the `chatt` was post.

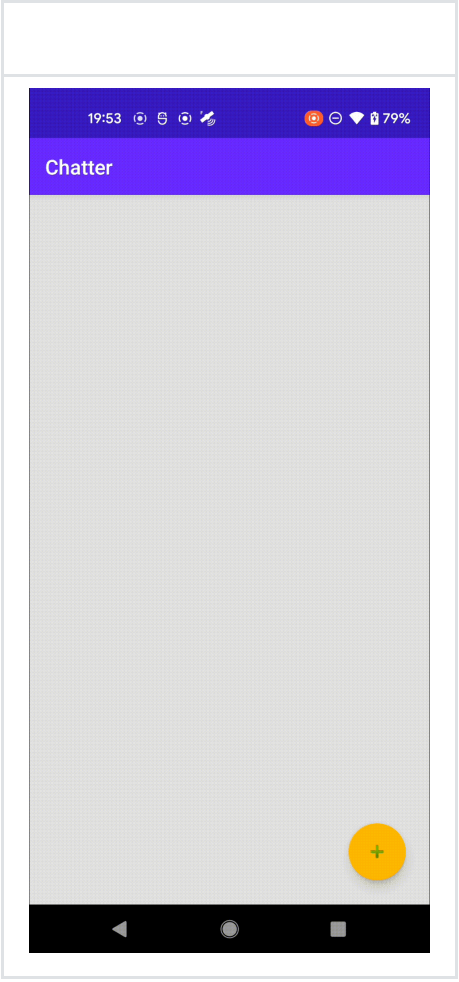
We will also implement a swiping gesture to allow users to switch from the default timeline view to the map view. From the map view, users can **not** post a `chatt`; they can only return to the timeline view. Once a user posts a `chatt`, they also can only return to the timeline view, not the map view. When a user transitions from the timeline view to the map view, the current trove of retrieved `chatts` will be passed along for display on the map view. User cannot initiate a new retrieval of `chatts` in the map view.

Gif demo

Post a new chatt and view chatts on Google Maps:

Note: Annotations in orange describe user actions or screens not recorded by the screen recorder and are not part of the app.

> Right click on the gif and open in a new tab to get a full-size view.
> To view the gif again, hit refresh on your browser (in the new tab where the gif is opened).



Part I: Converting Lab0 to use Programmatic UI

We have prepared a *Why Programmatic UI?* write up. Please give it a read if you're interested.

Preparing your GitHub repo

- On your laptop, navigate to `YOUR_LABSFOLDER/`
- Unzip your `lab0.zip` that you created as part of lab1
- Rename the newly unzipped `lab0` folder **lab2**
- Remove your lab2's `.gradle` directory by starting your Terminal program and run:

```
laptop$ cd YOUR_LABSFOLDER/lab2/kotlinChatter
laptop$ rm -rf .gradle
```

- Push your local `YOUR_LABSFOLDER/` repo to GitHub and make sure there're no git issues

In Android Studio's `File > Open` choose `YOUR_LABSFOLDER/lab2/kotlinChatter`.

Preparing project for programmatic UI

We will first factor out the XML layout files from our lab0 code.

In file `Gradle Scripts/build.gradle (Module:kotlinChatter.app)`, in the `android` block **remove**:

```
    buildFeatures {
        viewBinding true
    }
```

we will be creating views programmatically and will not be needing the services of `viewBinding`.

On the left/navigator pane of Android Studio, locate the folder `/app/res/layout`. We will be creating our layouts in code, so you can safely **delete** this whole folder. Right click on the folder, select `Delete...`, and click on the `DELETE` button.

We now design our screens programmatically.

Extensions.kt

As in previous labs, we'll collect all the extensions we'll be using into one file. Create a new Kotlin file called `Extensions.kt` and put the same `toast()` extension to `Context` from the previous lab in it. Then add the following extension function:

```
fun Context.dp2px(dp: Float): Int {
    return Math.ceil((dp * resources.displayMetrics.density).toDouble()).toInt()
}
```

ChattListItem

Since we have deleted the layout file `listitem_chatt.xml`, we need to replace it in code. Create a new Kotlin file, `ChattListItem`, and place the following three `TextView` variable declarations along with their initializations in it:

```
class ChattListItem(context: Context): ConstraintLayout(context) {
    val usernameTextView: TextView
    val timestampTextView: TextView
    val messageTextView: TextView

    init {
        usernameTextView = TextView(context).apply {
            id = generateViewId()
            textSize = 18.0f
        }
        timestampTextView = TextView(context).apply {
            id = generateViewId()
            textSize = 14.0f
        }
        messageTextView = TextView(context).apply {
            id = generateViewId()
            textSize = 18.0f
            setLineSpacing(0.0f, 1.2f)
        }
    }
}
```

We now provide layout constraints for our three `TextView`. Add the following code to `ChattListItem`'s initialization block above (inside the initialization block):

```
id = generateViewId()
addView(usernameTextView)
addView(timestampTextView)
addView(messageTextView)

val fill = LayoutParams(LayoutParams.MATCH_PARENT,
        LayoutParams.MATCH_PARENT).apply {
    setPadding(context.dp2px(5.82f), context.dp2px(7.27f),
        context.dp2px(5.82f), context.dp2px(13.82f))
 }
setLayoutParams(fill)

with (ConstraintSet()) {
    clone(this@ChattListItem)

    connect(usernameTextView.id, ConstraintSet.TOP, id, ConstraintSet.TOP)
    connect(usernameTextView.id, ConstraintSet.START, id, ConstraintSet.START)

    connect(timestampTextView.id, ConstraintSet.TOP, id, ConstraintSet.TOP)
    connect(timestampTextView.id, ConstraintSet.END, id, ConstraintSet.END)

    val margin = context.dp2px(8f)
    connect(messageTextView.id, ConstraintSet.TOP, usernameTextView.id, ConstraintSet.BOTTOM
    connect(messageTextView.id, ConstraintSet.START, id, ConstraintSet.START)

    applyTo(this@ChattListItem)
}
```

`ChattListItem` is a `ViewGroup` or `Layout` of type `ConstraintLayout`. In its initializer, we placed the three `TextView` as children `Views` of `ChattListItem`. We also generated identifiers for each of the `TextView` and for `ChattListItem` itself. We will use these IDs later when we place the UI elements relative to each other. When constructing a `ViewGroup/Layout` of type `ConstraintLayout`, each and every view **must** be given an ID or the app will crash.

We next provide layout constraints for our four UI elements: the three `TextView`s and the parent `ChattListItem`. We will be placing this ConstraintLayout container within a `ListView`, thereby forming a three-level view hierarchy: `ListView` > `ChattListItem` > `three TextViews`. Each item in `ListView` will be displayed according to the layout we specify for `ChattListItem`.

▶ px, dp, sp

`ConstraintSet`

After we've set the layout parameters for `ChattListItem`, we are ready to specify the constraints within the layout. All the constraints within a layout is encapsulated within a class called `ConstraintSet`. This class is used only when designing UI programmatically. It is not used when building UI using the Layout Editor. We

first create an instance of this class, then we call its `clone()` method to give a copy of the `Layout` for which we want to specify constraints.

> ⚠ **Important**: First add **all** the UI elements you want in a `Layout` before cloning it for `ConstraintSet`.

We now specify the top left/right corner of each `TextView`'s position and let ConstraintLayout determine the bottom right corner of each based on the provided coordinates and the content sizes of these `TextView`s. We set the `usernameTextView` flushed to the top and start (left) edge of the parent (`ChattListItem`). We set the `timestampTextView` also flushed to the top but to the end (right) edge of the parent. The `messageTextView` we set flushed to the start edge of the parent but its top we set 8 dp below the bottom of `usernameTextView`.

Like padding, margins are given in units of pixel (`px`)), hence the call to `dp2px()`. Recall that the **margins** specified for a `View` or `Layout` (in this case, `messageTextView`) apply **outside** it, the margins in layout parameters specify a View/Layout relationship to its parent's boundaries or to other, "sibling", Views/Layouts within the parent ViewGroup.

With all the necessary constraints specified, we call the `applyTo()` method of `ConstraintSet` to apply the set of constraints to the `Layout`, `ChattListItem`.

In the references section, we further provide references to `ConstraintSet` and programming layout in general. Google's Layout Inspector can still be used when building UI programmatically.

`ChattListAdapter`

Recall that the `getView()` method of `ChattListAdapter` first checks if a recycled `View` has been passed in. If so, it re-uses the view and re-populates the recycled view. If not, it creates a new `View`. We update the `getView()` method to create a `ChattListItem` whenever it needs a new view. Not having a layout file means that we no longer need to load/inflate it and since we are not using `viewBinding` any more, we also remove it from `getView()`. Replace your `getView()` with the following:

```
override fun getView(position: Int, convertView: View?, parent: ViewGroup): View {
    val listItemView = convertView as? ChattListItem ?: ChattListItem(context)

    listItemView.setBackgroundColor(Color.parseColor(if (position % 2 == 0) "#E0E0E0" else "#EEE

    getItem(position)?.run {
        listItemView.usernameTextView.text = username
        listItemView.messageTextView.text = message
        listItemView.timestampTextView.text = timestamp
    }

    return listItemView
}
```

`MainView`

To replace the deleted layout file `activity_main.xml`, create a new Kotlin file, `MainView`, and place the following three variable declarations in it:

```
class MainView(context: Context): ConstraintLayout(context) {
    val chattListView: ListView
    val postButton: FloatingActionButton
    val refreshContainer: SwipeRefreshLayout
}
```

First, we construct a `ListView` to display `chatt`s retrieved from the back end. For the `postButton`, we want to use a standard Material Design's "add document" button. We create `FloatingActionButton` with a golden yellow background, and for the icon we use the `ic_input_add` icon that is part of the Android SDK. Put the following in the initialization block of `MainView`:

```
    init {
        chattListView = ListView(context)

        postButton = FloatingActionButton(context).apply {
            id = generateViewId()
            setBackgroundTintList(ColorStateList.valueOf(Color.parseColor("#FFC107")))
            setImageResource(android.R.drawable.ic_input_add)
        }
```

We put the `ListView` inside a `SwipeRefreshLayout` refresh container so that we can pull down to refresh the timeline. Add the following code in `MainView`'s initialization block right below the above code:

```
        refreshContainer = SwipeRefreshLayout(context).apply {
            id = generateViewId()
            addView(chattListView)
        }
```

Now we prepare the `ConstraintLayout` that represents our main `ViewGroup`. Append the code below to the initialization block of `MainView`, which also closes it off:

```
        id = generateViewId()
        setBackgroundColor(Color.parseColor("#E0E0E0"))
        addView(postButton)
        addView(refreshContainer)

        val fill = LayoutParams(LayoutParams.WRAP_CONTENT,LayoutParams.WRAP_CONTENT).apply {
            val pad = context.dp2px(8f)
            setPadding(pad, pad, pad, pad)
        }
        setLayoutParams(fill)

        with (ConstraintSet()) {
            clone(this@MainView)

            val margin = context.dp2px(16f)
            connect(postButton.id, ConstraintSet.BOTTOM, id, ConstraintSet.BOTTOM, margin)
            connect(postButton.id, ConstraintSet.END, id, ConstraintSet.END, margin)
```

```
            applyTo(this@MainView)
        }
    }
```

We generated view identifiers for `postButton` and `refreshContainer` as well as for `MainView` itself. As with `ChattListItem`, these IDs will be used to refer to these UI elements when they are placed relative to each other.

We next provide layout constraints for our three UI elements. We first set the layout parameters for `MainView`. We set its width and height to wrap its contents. Then we set its padding. After we've set the layout parameters for `MainView`, we create an instance of `ConstraintSet` and give it the `Layout` we want to specify constraints for. Note again that all UI elements to be placed within a `Layout` must be added to the `Layout` before we call the `clone()` method of `ConstraintSet`.

We now specify the bottom right corner of `postButton` to be 16 dp off the sides of the parent (`MainView`). This is in addition to the parent's padding. The `SwipeRefreshLayout` container will take over the whole screen by default, so we don't need to set its layout. Finally, we call the `applyTo()` method of `ConstraintSet` to apply the set of constraints to the `MainView`.

MainActivity

Following the Model-View-Controller architecture, `MainActivity` is the controller for our `MainView` view above. So the first thing `MainActivity` must do is to construct the View and to populate it. We no longer have use for `ActivityViewBinding` type. Replace it with the `MainView` type. Update the `onCreate()` method of `MainActivity` to the following:

```
class MainActivity: AppCompatActivity() {
    private lateinit var chattListAdapter: ChattListAdapter
    private lateinit var view: MainView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        view = MainView(this)

        chattListAdapter = ChattListAdapter(this, chatts)
        view.chattListView.setAdapter(chattListAdapter)

        view.postButton.setOnClickListener {
            startActivity(Intent(this, PostActivity::class.java))
        }

        view.refreshContainer.setOnRefreshListener { refreshTimeline() }

        setContentView(view)

        refreshTimeline()
    }
```

Finally, remove the method `startPost()` from your `MainActivity` class. It has been incorporated into the `onCreate()` method above as a lambda expression.

PostView

To replace the deleted layout file `activity_post.xml`, create another new Kotlin file, `PostView`, and place the following content in it:

```
class PostView(context: Context): ConstraintLayout(context) {
    val usernameTextView: TextView
    val messageTextView: EditText

    init {
        usernameTextView = TextView(context).apply {
            id = generateViewId()
            textSize = 24.0f
            setText(R.string.username)
        }

        messageTextView = EditText(context).apply {
            id = generateViewId()
            textSize = 18.0f
            setLineSpacing(0.0f, 1.2f)
            setText(R.string.message)
        }

        id = generateViewId()
        addView(usernameTextView)
        addView(messageTextView)

        val fill = LayoutParams(LayoutParams.MATCH_PARENT,
                LayoutParams.MATCH_PARENT).apply {
            val pad = context.dp2px(8f)
            setPadding(pad, pad, pad, pad)
        }
        setLayoutParams(fill)

        with (ConstraintSet()) {
            clone(this@PostView)

            connect(usernameTextView.id, ConstraintSet.TOP, id, ConstraintSet.TOP, context.dp2px(21.
            connect(usernameTextView.id, ConstraintSet.START, id, ConstraintSet.START)
            connect(usernameTextView.id, ConstraintSet.END, id, ConstraintSet.END)

            connect(messageTextView.id, ConstraintSet.TOP, usernameTextView.id, ConstraintSet.TOP, c
            connect(messageTextView.id, ConstraintSet.START, id, ConstraintSet.START)

            applyTo(this@PostView)
        }
    }
}
```

I think by now you should be able to tell what this code does?

`PostActivity`

Similar to how we refactored `MainActivity` above, first construct the `PostView` view for which `PostActivity` is the controller. Replace the `PostViewBinding` type with `PostView` and update the `onCreate()` method of `PostActivity` accordingly:

```
class PostActivity: AppCompatActivity() {
    private lateinit var view: PostView
    private var enableSend = true

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        view = PostView(this)
        setContentView(view)
    }
```

the functions `onPrepareOptionsMenu()`, `onOptionsItemSelected()`, and `submitChatt()` remain unchanged.

There is no change to the files `Chatt.kt` and `ChattStore.kt` from lab0.

Congratulations! You've converted your lab0 from using Layout Editor to using programmatic UI. Try to build and run the code and confirm that it behaves the same as your lab0 version.

> If Android Studio complains of `external_file_lib_dex_archives/debug` not found during build, make sure you've removed the hidden directory `YOUR_LABSFOLDER/lab2/.gradle` as shown above, then do `File > Invalidate Caches/Restart...` click on the `INVALIDATE AND RESTART` button and try to build again after Android Studio restarted.

▶ Programmatic UI version of Chatter

Part II: Adding Maps to Chatter

Setting up the back end

As when we added support for images in the previous lab, we first prepare `Chatter` back end to support geodata.

Install updates

Every time you ssh to your server, you will see something like:

```
N updates can be applied immediately.
```

if `N` is not 0, run the following:

```
server$ sudo apt update
server$ sudo apt upgrade
```

Failure to update your packages could lead to the lab back end not performing correctly and also make you vulnerable to security hacks.

If you see `*** System restart required ***` when you ssh to your server, please run:

```
server$ sync
server$ sudo reboot
```

Your ssh session will be ended at the server. Wait a few minutes for the system to reboot before you ssh to your server again.

Modified `Chatter` API data formats

In this lab, we will add the user's **geodata**, consisting of their geolocation and velocity (facing and speed) at the time of posting, to a `chatt`.

As in previous labs, the `chatt`s retrieval API will send back all accumulated chatts in the form of a JSON object consisting of a dictionary entry with key "chatts" and value being an array of string arrays. In addition to the three elements "username", "message", "timestamp", each string array now carries an additional element which **is itself an array** containing the user's geodata (in order): latitude (lat), longitue (lon), location (corresponding to the lat/lon), compass point facing, and speed.

```
{
    "chatts": [["username0", "message0", "timestamp0", "[lat0, lon0, \"loc0\", \"facing0\", \"speed0
               ["username1", "message1", "timestamp1", "[lat1, lon1, \"loc1\", \"facing1\", \"speed1
               ...
              ]
}
```

To post a `chatt`, the client correspondingly sends a JSON object consisting of "username", "message", and "geodata", where the geodata conforms to the format above. For example:

```
{
   "username": "YOUR_UNIQNAME",
   "message": "Hello world!",
   "geodata": "[42.29, -83.72, \"Ann Arbor\", \"South\", \"walking\"]"
}
```

Notice that both the "facing" and "speed" elements are descriptive, as we'll explain further later.

Database table

As in previous labs, we first create a new table in our `chatterdb` database. Let's call this the `maps` table. It should have all the three columns of `username`, `message`, and `time` as in the `chatts` table of lab0. In

addition, it should have a `geodata` column of type `text`. Remember to give user `chatter` access to the new table.

> If you're not sure how to do any of the above, please review lab0 and lab1 back-end specs.

Editing `views.py`

Now, let's edit our `~/441/chatter/app/views.py` to handle geodata uploads. Make a copy of your `postchatt()` function inside your `views.py` file and name the copy `postmaps()`. Add code to your `postmaps()` to extract the geodata from the JSON object and insert it into the `maps` table, along with the rest of its associated `chatt`. To the back-end database, the geodata is just a string. Note that your `INSERT` statement should target the `maps` table, not the `chatts` table.

Next, make a copy of your `getchatts()` function inside your `views.py` file and name the copy `getmaps()`. In `getmaps()`, replace the `chatts` table in the `SELECT` statement with the `maps` table. This statement will retrieve all data we need (including geodata).

Save and exit `views.py`.

Routing for new urls

For the newly added `getmaps()` and `postmaps()` functions, add the following new routes to the `urlpatterns` array in `~/441/chatter/routing/urls.py`:

```
    path('getmaps/', views.getmaps, name='getmaps'),
    path('postmaps/', views.postmaps, name='postmaps'),
```

Remember to restart Gunicorn after you've updated `views.py` and `urls.py`.

You can test your back-end APIs using curl, HTTPie, or Postman. On Postman, you can use the example JSON above. After a successful POST, your database should contain:

```
chatterdb=# SELECT * FROM maps;
 username |   message    |   time   | geodata
----------+--------------+----------+---------
 YOUR_UNIQNAME  | Hello world! | 2020-06-21 18:28:43.354443 | [42.29, -83.72, "Ann Arbor", "South",
(1 row)
```

Submitting your back end

- Commit new changes to the local repo with:

```
server$ cd ~/441/chatter
server$ git commit -m "lab2 back end"
```

  and push new changes to the remote GitHub repo with:

```
server$ git push
```

- If `git push` fails due to new changes made to the remote repo, you will need to run `git pull` first. Then you may have to resolve any conflicts before you can `git push` again.

Once you are done with the back end, we'll move on to the front end.

Maps front end

In this section, we will go through the details about how to get user's geolocation information (latitude (lat), longitude (lon), human-readable location (loc), and velocity data (facing and speed)).

Overview

To add support for maps in our app, we need to accomplish three things on the front end:

1. working with Android's `LocationManager` to obtain the user's lat/lon and with Android's `GeoCoder` to determine place name from the lat/lon,
2. working with Android's `SensorManager` to obtain the user's bearing, and
3. working with Google Maps to display user's geodata information.

Location Service

Add the following line to your app-level gradle file, `/Gradle Scripts/build.gradle (Module: kotlinChatter.app)`:

```
dependencies {
    ...
    implementation 'com.google.android.gms:play-services-location:18.0.0'
}
```

Bring up the `Project Structure` window (⌘; on the Mac, `Ctl-Alt-Shift-s` on Windows). If the last item on the left pane, `Suggestions` , shows a number next to it, click on the item and click `Update` on all of the suggested updates, click `Apply` , click `OK` .

Requesting permission

We must first request user's permission to access the device's location. In your `AndroidManifest.xml` file, find `android.permisssion.INTERNET` and add the following lines right below it:

> "Fine" location uses GPS, WiFi, and cell-tower localization to determine device's location. "Coarse" location uses only WiFi and/or cell-tower localization, with city-block level accuracy.

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

The second line is to prevent Android 12 from throwing a `SecurityException` if we read the sensors more frequently than the allowed sampling rate.

Next, follow up the permission tag added to `AndroidManifest.xml` above with code in the `onCreate()` method of your `MainACtivity` to prompt user for access permission. We'll be reading the device's current location when posting a chatt in `PostActivity` and to zoom in to the user's current location in `MapsActivity`, it thus makes sense to ask for location access permission in `MainActivity` since we don't intend for either of the other Activities to be launchable from outside `Chatter`.

As in earlier labs, set up an Android's `ActivityResultContracts` to prompt user for permission to access fine location. The name of the contract is `RequestPermission` (singular). Once the contract is created, register it with the following callback handler, in the form of a lambda expression:

```
{ granted ->
    if (!granted) {
        toast("Fine location access denied", false)
        finish()
    }
}
```

Since we have no further use for the contract and registered launcher, you can launch it immediately with `Manifest.permission.ACCESS_FINE_LOCATION` as the launch argument.

`Chatt`

We add a new stored property `geodata` to the `Chatt` class to hold the geodata associated with each `chatt`:

```
class Chatt(var username: String? = null,
            var message: String? = null,
            var timestamp: String? = null,
            var geodata: GeoData? = null)
```

`GeoData`

We need to create a new `GeoData` class to store the additional geodata. Let's put our new `GeoData` class in a new `GeoData.kt` file:

```
class GeoData(var lat: Double = 0.0, var lon: Double = 0.0, var loc: String = "",
              var facing: String = "unknown", var speed: String = "unkown")
```

Post `chatt` with geodata

In `PostActivity`, since we already requested location permission in `MainActivity`, we tell Android Studio not to warn us about "MissingPermission" when we try to read the device's location. We add implementation of `SensorEventListener` interface to the class declaration to read the device's bearing. Your class delcaration for `PostActivity` should look something like this:

```
@SuppressLint("MissingPermission") // checked in MainActivity
class PostActivity: AppCompatActivity(), SensorEventListener {
```

Then add the following class member variables to `PostActivity`:

```
    private var lat = 0.0
    private var lon = 0.0
    private var speed = -1.0f

    private lateinit var sensorManager: SensorManager
    private var accelerometer: Sensor? = null
    private var magnetometer: Sensor? = null
```

We use Android's `FusedLocationProvider` to obtain the user's current location (latitude (lat), longitude (lon)) and speed. "Fused" here means that it uses all of GPS, WiFi, and cell-tower localization to balance between battery consumption and accuracy in determining the device's location. To determine bearing, we read the accelerometer and magnetometer sensors. Add the following to your `PostActivity.onCreate()`:

```
        LocationServices.getFusedLocationProviderClient(applicationContext)
            .getCurrentLocation(LocationRequest.PRIORITY_HIGH_ACCURACY, CancellationTokenSource().t
            .addOnCompleteListener {
                if (it.isSuccessful) {
                    lat = it.result.latitude
                    lon = it.result.longitude
                    speed = it.result.speed
                    if (!enableSend) {
                        submitChatt()
                    }
                } else {
                    Log.e("PostActivity getFusedLocation", it.exception.toString())
                }
            }

        // read sensors to determine bearing
        sensorManager = applicationContext.getSystemService(Context.SENSOR_SERVICE) as SensorManage
        accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
        magnetometer = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD)
        accelerometer?.let {
            sensorManager.registerListener(this, it, SensorManager.SENSOR_DELAY_FASTEST)
        }
        magnetometer?.let {
            sensorManager.registerListener(this, it, SensorManager.SENSOR_DELAY_FASTEST)
        }
```

We use `applicationContext` in the call to `LocationServices.getFusedLocationProviderClient()` to guard against the activity being destroyed due to orientation change.

The call to `getCurrentLocation()` usually returns in less than a second, but could take longer. When the user taps `Send` button, we check whether `getCurrentLocation()` has completed. If it has, `speed` would be set to a non-negative value. In which case, we go ahead and call `submitChatt()` to post the `chatt` with

geodata. Otherwise, we simply disable send, but let the completion listener of `getCurrentLocation()` to perform the actual posting. Thus the completion closure for `addOnCompletionListener()` above checks whether send is enabled. If not, it calls `submitChatt()`.

Replace the call to `submitChatt()` in the `onOptionsItemSelected()` method of your `PostActivity` with:

```
            if (speed < 0f) {
                toast("Getting location fix . . .")
            } else {
                submitChatt()
            }
```

Sensor readings for bearing

To implement the `SensorEventListener` interface means `GeoData` must provide `onAccuracyChanged()` and `onSensorChanged()` methods, though in this case we're not doing anything in the former. In `onSensorChanged()`, we record whether it's a change in the accelerometer or the magnetometer reading. To obtain the bearing of the device, we need readings from both. Add to your `PostActivity` class the following methods:

```
override fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {}

var gravity: FloatArray = emptyArray<Float>().toFloatArray()
var geomagnetic: FloatArray = emptyArray<Float>().toFloatArray()

override fun onSensorChanged(event: SensorEvent) {
    if (event.sensor.type == Sensor.TYPE_ACCELEROMETER)
        gravity = event.values
    if (event.sensor.type == Sensor.TYPE_MAGNETIC_FIELD)
        geomagnetic = event.values
}
```

We read the device's currrent location by calling `getCurrentLocation()` once. There is, unfortunately, no equivalent single-call API to determine bearing. Instead, we had to register a listener for updates to the magnetometer and accelerometer sensors respectively. To conserve energy, add the following `onDestroy()` Activity lifecycle event handler method to your `PostActivity` class, to unregister our sensor listeners:

```
override fun onDestroy() {
    super.onDestroy()
    accelerometer?.let {
        sensorManager.unregisterListener(this, it)
    }
    magnetometer?.let {
        sensorManager.unregisterListener(this, it)
    }
}
```

More familiar bearing and speed readouts

Bearing is familiarly expressed in terms of compass directions, so 0° (or 360°) is North, 90° is East, 180° South, and 270° West:

```
fun convertBearing(): String {
    if (gravity.isNotEmpty() && geomagnetic.isNotEmpty()) {
        val R = FloatArray(9)
        val I = FloatArray(9)
        if (SensorManager.getRotationMatrix(R, I, gravity, geomagnetic)) {
            val orientation = FloatArray(3)
            SensorManager.getOrientation(R, orientation)
            // the 3 elements of orientation: azimuth, pitch, and roll,
            // bearing is azimuth = orientation[0], in rad
            val bearingdeg = (Math.toDegrees(orientation[0].toDouble()) + 360).rem(360)
            val compass = arrayOf("North", "NE", "East", "SE", "South", "SW", "West", "NW", "Nor
            val index = (bearingdeg / 45).toInt()
            return compass[index]
        }
    }
    return "unknown"
}
```

Speed of movement is similarly converted to more familiar transportation modes (in m/s):

```
fun convertSpeed(): String {
    return when (speed) {
        in 1.2..4.9 -> "walking"
        in 5.0..6.9 -> "running"
        in 7.0..12.9 -> "cycling"
        in 13.0..89.9 -> "driving"
        in 90.0..138.9 -> "in train"
        in 139.0..224.9 -> "flying"
        else -> "resting"
    }
}
```

We use Android's `Geocoder` to perform reverse geocoding, to translate the lat/lon coordinates into more familiar geographic location (loc) names:

```
fun convertLoc(): String {
    val locations = Geocoder(applicationContext, Locale.getDefault()).getFromLocation(lat, lon,
    if (locations.size > 0) {
        val geoloc = locations[0]
        return geoloc.locality ?: geoloc.subAdminArea ?: geoloc.adminArea ?: geoloc.countryName
        ?: "unknown"
    }
    return "unknown"
}
```

We also need to update `submitChatt()` in `PostActivity` to upload the geodata, after converting the location, heading, and speed info into more human familar formats, along with the `chatt`. Change the

declaration for `chatt` in `submitChatt()` to:

```
val chatt = Chatt(username = view.usernameTextView.text.toString(),
    message = view.messageTextView.text.toString(),
    geodata = GeoData(lat, lon, convertLoc(),
        convertBearing(), convertSpeed()))
```

Subsequently, we update `postChatt()` in `ChattStore.kt` to pass along the geodata. Here's the updated top part of `postChatt(_:)` :

```
fun postChatt(context: Context, chatt: Chatt) {
    val geoObj = chatt.geodata?.run{ JSONArray(listOf(lat, lon, loc, facing, speed)) }

    val jsonObj = mapOf(
        "username" to chatt.username,
        "message" to chatt.message,
        "geodata" to geoObj?.toString()
    )
    // ...
```

Staying in the `postChatt()` method, find the declaration of `postRequest` and replace `postchatt` with `postmaps` in the url construction.

We are now ready to retrieve `chatts` from the back end.

`getChatts()`

Again, find the declaration of `getRequest` and replace `getchatts` with `getmaps` in the url construction.

To construct `Chatt` objects from retrieved JSON data, we modify the `getChatts()` function in the `ChattStore` class. Find the `if (chattEntry.length() == nFields) {` block in `getChatts()` and replace the **content** of the `if` block with:

```
val geoArr = if (chattEntry[3] == JSONObject.NULL) null else JSONArray(chatt
chatts.add(Chatt(username = chattEntry[0].toString(),
    message = chattEntry[1].toString(),
    timestamp = chattEntry[2].toString(),
    geodata = geoArr?.let { GeoData(
        lat = it[0].toString().toDouble(),
        lon = it[1].toString().toDouble(),
        loc = it[2].toString(),
        facing = it[3].toString(),
        speed = it[4].toString()
    )}
))
```

Each string array returned by the back end represents a single `chatt` . The fourth entry in each string array, `chattEntry[3]` , contains the string holding an "inner" array of geodata. If this string is not `null` , we convert it into a `JSONArray` and construct a `GeoData` using elements of this array to initialize the `GeoData` .

We then use this `GeoData` instance, along with the other elements of the "outer" array, to construct a `Chatt`.

Viewing geolocation data

We support three ways to view the geodata retrieved with the `chatts` from the back end:

1. as descriptive text displayed with each `chatt`,
2. on a map, viewing the posting locations of all retrieved `chatts`, and
3. on a map, viewing the posting location of a single `chatt`.

As text alongside a `chatt`

Let's update `ChattListItem` and add an additional `TextView` to display geolocation data textually while displaying `chatts`. Add to property declaratation of `ChattListItem`:

```
    val geodataTextView: TextView
```

and initialize it in the `init` block, after the initialization of `messageTextView`:

```
        geodataTextView = TextView(context).apply {
            id = generateViewId()
            setAutoSizeTextTypeUniformWithConfiguration(12, 14, 1, TypedValue.COMPLEX_UNIT_SP)
            setLineSpacing(0.0f, 1.2f)
        }
```

The data we display can get rather long: we want to automatically scale the text (`setAutoSizeTextType`) so that all the information is always visible.

Then add the new `geodataTextView` as a child view of `ChattListItem` layout, right after adding `messageTextView`:

```
        addView(geodataTextView)
```

Finally add the following constraints for `geodataTextView` right before the calling `applyTo()`:

```
        connect(geodataTextView.id, ConstraintSet.TOP, messageTextView.id, ConstraintSet.BOTTOM,
        connect(geodataTextView.id, ConstraintSet.START, id, ConstraintSet.START)
```

We next modify our `ChattListAdapter` class to display our new geodata. Add the following code to the `run{}` code block of `getView()` in `ChattListAdapter` class,right before the block's closing right brace:

```
            geodata?.let {
                listItemView.geodataTextView.text =
                    "Posted from ${it.loc}, while facing ${it.facing} moving at ${it.speed} speed."
            } ?: run {
```

```
            listItemView.geodataTextView.text = ""
        }
```

With these changes, when you retrieve `chatts` from the back end, you should now see additional geodata displayed with posted `chatt`.

On a map

We support two ways to view the geodata on a map:

1. viewing the posting locations of all retrieved `chatts`, and
2. viewing the posting location of a single `chatt`.

To display the geodata associated with `chatts` on a map, we will need a separate Google `MapsActivity`. We will let user view the posting locations of all retrieved `chatts` by swiping left on the `chatts` timeline on `MainActivity` to transition to `MapsActivity`. We will implement this method first.

To create a Google Maps Activity, navigate to `File > New > Google > Google Maps Activity`. Click `Finish`. This will creates a `google_maps_api.xml` file in `/app/res/values` folder. We'll be using this file, so keep it. It also creates a `/res/layout/activity_maps.xml` layout file. We won't be using this file and you can safely delete it.

The Google Maps SDK automatically handles access to the Google Maps servers, map display, and response to user gestures such as clicks and drags. You can also add markers, polylines, ground overlays and info windows to your map. These objects provide additional information for map locations, and allow user interaction with the map.

Get and add API key

Next you need to obtain a Google Maps API key:

1. Copy the first link provided in the `/app/res/values/google_maps_api.xml` file and paste it into your browser. The link takes you to the Google Cloud Platform Console and supplies the required information to the Google Cloud Platform Console via URL parameters, thus reducing the manual input required from you.

2. Follow the instructions to create a new project on the Google Cloud Platform Console or select an existing project. ((You will need a gmail address, not a umich email address, to set up a Google Cloud Platform Console account.)

⚠ The Google API website is reconfigured very frequently. The instructions here have been through at least 4 reconfigurations of the site. If what you see on the site is so totally different from the description here that **you can't make your way through it**, please let the teaching staff know.

Or you can try the three-step instructions in Set up in Cloud Console document. If you do follow the instructions in this document however, please do **not** follow the instructions to use `Secret Gradle Plugin` and `local.properties`. These will render your lab not gradable by the teaching staff.

1. Set restrictions on your API key before using it in production: create an Android-restricted and API-restricted API key for your project.

2. Copy the resulting API key, go back to Android Studio, and paste the API key into the `<string>` element in the `google_maps_api.xml` file, replacing `YOUR_KEY_HERE` .

3. Add the following dependency to your app-level gradle file, `/Gradle Scripts/build.gradle (Module: kotlinChatter.app)` :

```
dependencies {
    ...
    implementation 'com.google.android.gms:play-services-maps:17.0.1'
}
```

Swipe left to view location of all `chatts` on map

Now that we have a Google `MapsActivity` , we will allow users to swipe left in `MainActivity` to access it. Add the following member variables to `MainActivity` class:

```
private var xdown: Float = 0f
private var ydown: Float = 0f
```

We will use these variables and the following code to detect swipe left gesture and launch `MapsActivity` (add it to your `MainActivity` class):

```
override fun dispatchTouchEvent(event: MotionEvent): Boolean {
    super.dispatchTouchEvent(event)

    when (event.action) {
        MotionEvent.ACTION_DOWN -> {
            xdown = event.x
            ydown = event.y
        }
        MotionEvent.ACTION_UP -> {
            if ((xdown - event.x) > 100 && abs(event.y - ydown) < 100) {
                startActivity(Intent(this, MapsActivity::class.java))
            }
        }
    }
    return false
}
```

> If Android Studio is unable to automatically resolve `abs()` , add the following to the top of the file:
>
> ```
> import kotlin.math.abs
> ```

⚠ Gesture navigation of Android 12 conflicts with our swipe left gesture. To test this lab, you may have to turn off Gesture navigation: go to `Settings > System > Gestures > System navigation` and

select `3-button navigation` instead of `Gesture navigation` or set the `Left edge` sensitivity to `Low` (accessed by clicking on the gear button next to `Gesture navigation`).

We can now move on to the implementations of `MapsActivity`!

`MapsActivity`

If `MapsActivity` was launched by user swiping left in `MainActivity`, we are going to implement three features:

1. to display **all** `chatts` in `ChattStore.chatts`
2. mark each `chatt` with a **marker** on the map and customize each marker's `Information Window` to show the `chatt` poster's geodata, and
3. to center and zoom our map onto the **user's current location**.

If, on the other hand, `MapsActivity` was launched by user tapping on a single `chatt` in `MainActivity`, we will implement the following three features:

1. to display **the single** `chatt` whose position in the `ChattStore.chatts` array is passed from `MainActivity` as intent data tagged with `"INDEX"`,
2. mark the `chatt` with a **marker** on the map and customize its `Information Window` to show the `chatt` poster's geodata, and
3. to center and zoom our map onto the **location of the `chatt` 's poster**.

In both cases, we want the current user's location marked on the map with Google Map's default blue dot, and location "bull's eye" showing on the map, which, when clicked, will center the map on the user's current location.

`onCreate()`

To use Google Maps programmatically, replace the content of `MapsActivity.onCreate()` with:

```
        super.onCreate(savedInstanceState)

        val mapFragment = SupportMapFragment.newInstance()
        supportFragmentManager
            .beginTransaction()
            .add(android.R.id.content, mapFragment)
            .commit()

        mapFragment.getMapAsync(this)
```

We instantiate Google's Map view fragment and attach it to the activity's default content view (`android.R.id.content`). Since we're not creating any custom layout, we don't need to call `setContentView()`. After attaching the map view fragment to the activity's content view, we call the SDK's asynchronous method to get the map to populate the fragment.

Enabling user's current location

Since we've requested access to location permission in `MainActivity`, we can suppress warning about not asking for access permissions here. Add the following annotation to the definition of `onMapReady()` (add the line directly above the definition):

```
@SuppressLint("MissingPermission") // checked in MainActivity
```

In `onMapReady()`, comment out the last three lines that move the camera to Sydney.

To view the user's current location and to zoom the map into that location, add the following code to `onMapReady()`:

```
mMap.isMyLocationEnabled = true
```

With `my location` enabled, when the map is displayed, if you click the button that looks like a bull's eye target at the upper right corner of the map, it should pan and zoom onto your current location.

Marking all `chatts`

Next we try to retrieve the index into the `chatts` array whose poster's location we want to display and zoom in on. Recall that we tagged the index with the label `"INDEX"` when creating the intent to start `MapsActivity`. If we couldn't find any data tagged with `"INDEX"` associated with the intent, it means the user had swiped left in `MainActivity` and we should display the poster locations of all `chatts` retrieved from the back end instead. In this latter case, we set the `index` variable's value to `-1` and once we have displayed all the posters' locations, we center and zoom in on the user's current location. Add the following code to `onMapReady()` right below the line enabling `my location`:

```
val index = intent.getIntExtra("INDEX", -1)
if (index < 0) {
    ChattStore.chatts.forEach {
        it?.let { renderChatt(it) }
    }

    // center and zoom in on user's current location
    LocationServices.getFusedLocationProviderClient(applicationContext)
        .getCurrentLocation(PRIORITY_HIGH_ACCURACY, CancellationTokenSource().token)
        .addOnCompleteListener {
            if (it.isSuccessful) {
                val pos = LatLng(it.result.latitude, it.result.longitude)
                mMap.animateCamera(CameraUpdateFactory.newLatLngZoom(pos, 16f))
            } else {
                Log.e("MapsActivity getFusedLocation", it.exception.toString())
            }
        }

    return
}
```

The call to `animateCamera()` centers and zooms the camera onto the users' current location. To conserve energy, we do not continuously center on the user's current location; if the user is moving, we only center

on the user's location upon the launch of `MapsActivity`.

Marking a single `chatt`

On the other hand, if we managed to retrieve the intent data tagged with `"INDEX"`, it indicates that the user wants to view a single `chatt`'s poster location. Add the following code to `onMapReady()` right below the above.

```
        // if coming from a chatt being tapped, show only the position of the
        // poster, centered and zoomed in on the poster
        val chatt = ChattStore.chatts[index] ?: return
        renderChatt(chatt)
        val geodata = chatt.geodata ?: return
        val pos = LatLng(geodata.lat, geodata.lon)
        mMap.animateCamera(CameraUpdateFactory.newLatLngZoom(pos, 16f))
```

Rendering `chatts`

We now turn to the second task: to display each post as a marker on the map. We do this using the `renderChatt()` function:

```
    private fun renderChatt(chatt: Chatt) {
        val geodata = chatt.geodata ?: return
        val pos = LatLng(geodata.lat, geodata.lon)

        val snippet = "${chatt.username}\n${chatt.message}\n\n"+
                "Posted from ${geodata.loc}, while facing ${geodata.facing}"+
                " moving at ${geodata.speed} speed."

        mMap.addMarker(MarkerOptions().position(pos)
            .title(chatt.timestamp)                // timestamp
            .snippet(snippet))
    }
```

If you now run the app you should be able to see your map populated with markers, each representing a posted `chatt`. You can test your map view by posting `chatts` with different geodata values either using Postman, HTTPie, or by manually entering different values in `postChatt()`. If you click on a marker you should be able to see the timestamp and username of the corresponding chatt. We will next create custom information window to display more information about each post.

`MapInfoWindow`

Google Maps allows you to display an information window when a marker is selected. But as we have seen above, the basic window is very rudimentary. To display a richer set of information, including each `chatt`'s username, message, timestamp, location, facing, and speed, we need to customize the information window. First, we design the layout for our new info window programmatically.

Create a new Kotlin Class/File, name it `MapInfoWindow` and put the following class in it:

```
class MapInfoWindow(context: Context): LinearLayout(context) {
    val titleTextView: TextView
```

```
    val snippetTextView: TextView

    init {
        titleTextView = TextView(context).apply {
            textSize = 14.0f
            setHorizontalGravity(Gravity.CENTER)
            ellipsize = TextUtils.TruncateAt.END
            maxLines = 1
            setTypeface(typeface, Typeface.BOLD)
            setTextColor(ColorStateList.valueOf(Color.parseColor("#0000FF")))
        }

        snippetTextView = TextView(context).apply {
            textSize = 14.0f
            ellipsize = TextUtils.TruncateAt.END
            maxLines = 10
            setTypeface(typeface, Typeface.BOLD)
            setTextColor(ColorStateList.valueOf(Color.parseColor("#000000")))
        }

        addView(titleTextView)
        addView(snippetTextView)

        val fill = LayoutParams(LayoutParams.MATCH_PARENT,
            LayoutParams.MATCH_PARENT).apply {
            val pad = context.dp2px(3.64f)
            setPadding(pad, pad, pad, pad)
            orientation = VERTICAL
        }
        setLayoutParams(fill)
    }
}
```

Note that since we're using `LinearLayout`, as opposed to `ConstratinLayout`, we don't need to assign an ID to each view here.

Next, we create a new custom info window adapter class to override the default info window adapter. Create another `Kotlin Class/File`, name it `MapInfoAdapter`, and put the following class in it:

```
class MapInfoAdapter(context: Context) : GoogleMap.InfoWindowAdapter {
    private var infoWindow: MapInfoWindow

    init {
        infoWindow = MapInfoWindow(context)
    }

    override fun getInfoWindow(marker: Marker): View? {
        return null
    }

    override fun getInfoContents(marker: Marker): View {
        infoWindow.titleTextView.text = marker.title
        infoWindow.snippetTextView.text = marker.snippet

        return infoWindow
```

```
    }
}
```

Finally, we instruct Google Maps to use our custom info window instead of the default. In `MapsActivity`, add the following line to the end of `renderChatt()`, right before the closing brace of the function:

```
        mMap.setInfoWindowAdapter(MapInfoAdapter(this))
```

Now, when a marker is selected, it should display our custom info window with the timestamp of each `chatt` along with the geodata of the poster.

Button to view a single `chatt`'s location on map

To enable user to view the poster location of a single `chatt`, we will add a `mapButton` to each `chatt` in the timeline. When user clicks on this button, pass the `chatt` associated with the button to `MapActivity` and start `MapActivity`.

The `mapButton` uses `R.drawable.border` as the background. Copy the `border.xml` resource file from your lab1's `/app/res/drawable/` here. Now we add a `mapButton` variable to the property declaration of `ChattListItem` class:

```
    val mapButton: ImageButton
```

and initialize it in the `init` block, after initializing `geodataTextView`:

```
        mapButton = ImageButton(context).apply {
            id = generateViewId()
            visibility = GONE
            setBackgroundResource(R.drawable.border)
            setImageResource(android.R.drawable.ic_menu_mylocation)
        }
```

add it to the `ChattListItem` layout, after adding `geodataTextView`:

```
        addView(mapButton)
```

and add the following constraints before the call to `applyTo()`:

```
            connect(mapButton.id, ConstraintSet.TOP, timestampTextView.id, ConstraintSet.BOTTOM, mar
            connect(mapButton.id, ConstraintSet.BOTTOM, geodataTextView.id, ConstraintSet.TOP, margi
            connect(mapButton.id, ConstraintSet.END, id, ConstraintSet.END)
            val dim = context.dp2px(40f)
            constrainWidth(mapButton.id, dim)
            constrainHeight(mapButton.id, dim)
```

Next in `ChattListAdapter`, when there's geodata associated with a `chatt`, we turn the `mapButton` visible and launch `MapsActivity` when the button is tapped. We also pass along to `MapsActivity` the `position` of the `chatt` entry tapped so that `MapsActivity` can retrieve the correct entry for display.

If there's no geodata associated with the `chatt`, we **explicitly** turn the `mapButton` invisible because list items are recycled and reused, we don't want a stray button to show up from a previous use. At the same time, we set the button's `onClickListener()` to `null`.

Replace the `geodata?.let{} ?: run {}` block in the `getView()` method of `ChattListAdapter` with:

```
geodata?.let {
    listItemView.geodataTextView.text =
        "Posted from ${it.loc}, while facing ${it.facing} moving at ${it.speed} speed."

    listItemView.mapButton.visibility = View.VISIBLE
    listItemView.mapButton.setOnClickListener { v ->
        if (v.id == listItemView.mapButton.id) {
            val intent = Intent(context, MapsActivity::class.java)
            intent.putExtra("INDEX", position)
            context.startActivity(intent)
        }
    }
} ?: run {
    listItemView.geodataTextView.text = ""
    listItemView.mapButton.visibility = View.INVISIBLE
    listItemView.mapButton.setOnClickListener(null)
}
```

To recap, by the end of this lab, if you tap on the map button associated with each `chatt`, you will see a map centered and zoomed in on **the poster's location**, with a marker at the location. Swiping left on `MainActivity` screen will bring you to the map view with **all** retrieved `chatts` shown as markers on the map and the map centered and zoomed in on **the user's current location**. In both cases, tapping the button that looks like a bull's eye target should pan and zoom onto the user's current location.

Simulating locations

To use the Android emulator to simulate your location, follow the instructions in our Getting Started with Android Development.

To simulate location on device, there are multiple apps in the Google Playstore that allows you to set fake GPS location at the same time you have `Chatter` running. I use "Fake GPS Location PROFESSIONAL" developed by "Just4Fun Utilities". Once you've installed the app, go to `Settings > System > Developer options > Select mock location app` and select the app. In the app, search for the desired simulated location and tap the play button. When you post a `chatt`, or when you view all `chatts`, the user's current location should be the simulated location. You can go back into the "Fake GPS Location" app and select a different simulated location and it should again be reflected in `Chatter`.

Submission guidelines

Unlike in previous labs, there is a **CRUCIAL** extra step to do before you push your lab to GitHub:

- Copy `debug.keystore` in (`~/.android/` for Mac or `C:\Users\<CurrentUser>\.android\` for Windows) to your `lab2` folder.

Without your `debug.keystore` we won't be able to run your app.

> **IMPORTANT:** If you work in team, remember to put your team mate's uniqnames in lab2 folder's `README.md` so that we'd know. Otherwise, we could mistakenly thought that you were cheating and accidentally report you to the Honor Council, which would be a hassle to undo.

Enter your uniqname (and that of your team mate's) and the link to your GitHub repo on the Lab Links sheet. The request for teaming information is redundant by design. If you're using a different GitHub repo from previous lab's, invite `eecs441staff@umich.edu` to your GitHub repo.

Push your lab2 to its GitHub repo as set up at the start of this spec. Using GitHub Desktop to do this, you can follow the steps below:

- Open GitHub Desktop and click on `Current Repository` on the top left of the interface
- Click on your `441` GitHub repo
- Add Summary to your changes and click `Commit to master`
- If you have a team mate and they have pushed changes to GitHub, you'll have to click `Pull Origin` and resolve any conflicts before . . .
- Finally click on `Push Origin` to push changes to GitHub

Go to the GitHub website to confirm that your project files for lab2 have been uploaded to GitHub repo under folder `lab2` .

References

Layout, screen density, margins

- Debug Your Layout with Layout Inspector and Layout Validation
- How to Find Device Metrics for Any Screen
- Difference Between dp, dip, sp, px, in, mm, pt in Android
  - Simpler dp to px conversion with queried density in Kotlin and as scaling factor from reference density (160dpi)
  - How to convert DP, PX, SP among each other?

Programmatic Layout

- Adding Views & Constraints to Android Constraint Layout Programmatically
- Managing Constraints using ConstraintSet

- Clone after addView
- generateViewId()
- ListAdapter
- Chaining Views in a ConstraintLayout Programmatically
- Create Android views and widgets programmatically
- LinearLayout

Buttons, icons, menu items, typefaces

- Setting FloatingActionButton background color
- Change FloatingActionButton icon
- Access default icon in SDK
- How to Dynamically or Programmatically Add Menu Items for an Android Activity
- setTypeface
- Autoscaling TextView

Mapping

- Get Started With Google Maps
- Add a SupportMapFragment dynamically
  - How do I add a Fragment to an Activity with a programmatically created content view Search for `android.R.id.content` .
  - Adding a GoogleMap to a Fragment Programmatically
- Location
- Getting City Name of Current Position
- Android Location Providers
- How do I get the current GPS location programmatically in Android?
- How can I replace TYPE_ORIENTATION (deprecated)?
  - FusedLocationProviderClient doesn't have bearing data, note rad to degree conversion

Array and JSON

- Java convert a Json string to an array
- Convert normal Java Array or ArrayList to Json Array in android
- How to initialize list in Kotlin
- Difference between List and Array types in Kotlin
- Kotlin when: A switch with Superpowers

Appendix: imports

Prepared for EECS 441 by Alexander Wu, Wendan Jiang, Benjamin Brengman, Ollie Elmgren, Yibo Pi, and Sugih Jamin	Last updated: July 4th, 2021