

Lab 4: Non-UI Thread processing, Communication, Location Services

(Due @ 3.25pm on Monday, February 24, 2020)

Introduction:

In this lab will show you how to work with Non-UI threads and the inbuilt GPS on your android device. You will learn how to create a Non-UI thread and make it interact with the Main thread. You will also learn how to access your android devices location by using its inbuilt GPS to make a simple but intuitive location tracking app. In Milestone 1 you will create a simple application that uses non-ui threads to run a process in the background and also making it interact with the main thread. In Milestone 2, you will learn how to create a location tracker that asks users for permission to access the inbuilt GPS in your android device.

Milestone 1 - In this milestone we will develop a basic app to understand the use of Non-UI threads in Android.

- As you learned in the lectures, on the Android platform, applications operate, by default, on one thread. This thread is called the *UI thread*.
- It is often called that because this single thread displays the user interface and listens for events that occur when the user interacts with the app.
- Developers quickly learn that if code running on that thread hogs that single thread and prevents user interaction (for more than 5 seconds), it causes Android to throw up the infamous Android Not Responsive (ANR) error.

So how do you prevent ANR? Your application must create other threads and put long running work on non-UI threads. There are options on how to accomplish the creation of alternate threads. You can create and start your own `java.lang.Thread`. You can create and start an `AsyncTask` – Android's own thread simplification mechanism. The non-UI thread then handles long running processing – like downloading a file – while the UI thread sticks to displaying the UI and reacting to user events.

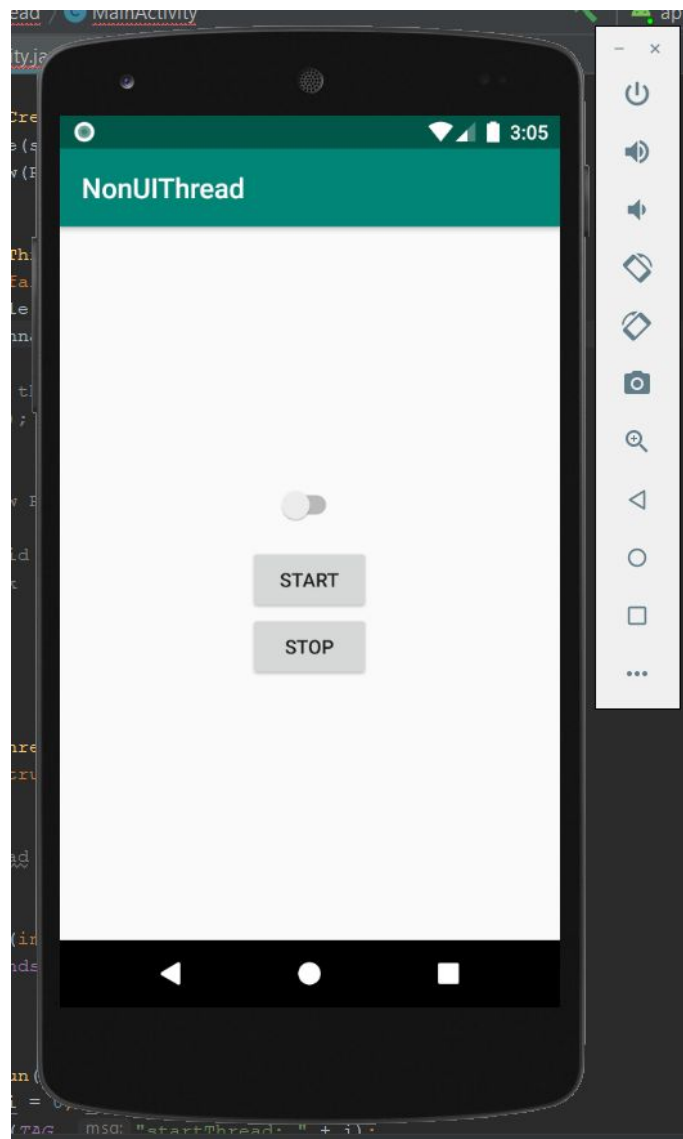
Let us begin:

1.) Every time we start android we run on single threads by default, which is called the main thread. Another name for it is the UI thread because it is responsible for managing and updating the user interface.

To start off let us make a clone of the repository as we have done before:

- Clone the Lab 4 Milestone 1 repository by selecting File > New > Project from Version Control > Git
- Then click the following link and accept the invitation:
<https://classroom.github.com/a/XABvyRFB>
- You'll get a link in the form:
<https://github.com/CS-407-Spring2020/lab-4-milestone-1-yourgithubnamehere>

Now, set up your app to have two buttons and one toggle switch in the format as shown in the screenshot below.



You can either do this by writing the code in the corresponding xml file or dragging and dropping the buttons from the design tab.

2.) When you have completed this step you can notice that if you click the “Start” button it becomes darker in colour for a second and then turns back to the normal colour. When a button is clicked, in the background the “OnClick” function is called and once that function completes its execution, the button returns to its default colour. All this happens sequentially in the background every time any button is clicked.

Now the problem is, what if the OnClick function takes a long time to complete execution? The answer is that the app will remain frozen until the completion of the OnClick function. So how can we run long running tasks?

We can run them in the background using Non-UI threads.

3.) Let us simulate this task by making use of the buttons we have set up in the previous step. First let us define the the functions that will execute when we click on the start and stop functions.

In the MainActivity.java file:

- 1.) Write a function called startThread() corresponding to the “Start” button.
- 2.) Write a function called stopThread() corresponding to the “Stop” button.

```
public void startThread(View view) {  
  
}  
  
public void stopThread(View view) {  
  
}
```

Now update the same in the activity_main.xml file, so that file should look something like this:

```

<Switch
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="16dp" />

<Button
    android:id="@+id/button_start_thread"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="startThread"
    android:text="Start" />

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="stopThread"
    android:text="Stop" />

```

Note: That your file may look different based on your layout preferences, but the main thing we want you to focus on is that in each button, the OnClick functions are assigned properly.

4.) Now we will create and run a thread in the background to show how we defer a time consuming task to the background and how we can make this background task interact with the main thread.

As their name suggests, the `startThread()` method is invoked to start a Non-UI thread and similarly the `stopThread()` method is invoked to end that particular thread. One common way to create thread is by implementing the `Runnable` interface. In the screenshots below is the code that you can use to create and terminate a thread.

```

35     class ExampleRunnable implements Runnable {
36
37         @Override
38         public void run() {
39
40         }
41     }

```

To specify the number of seconds that we want the thread to sleep for, we can include a constructor as shown below:

```

33
34
35 class ExampleRunnable implements Runnable {
36
37     int seconds;
38
39     @
40     ExampleRunnable(int seconds) {
41         this.seconds = seconds;
42     }
43
44     @Override
45     public void run() {

```

5.) Now let us write some code to start the thread and terminate it.

```

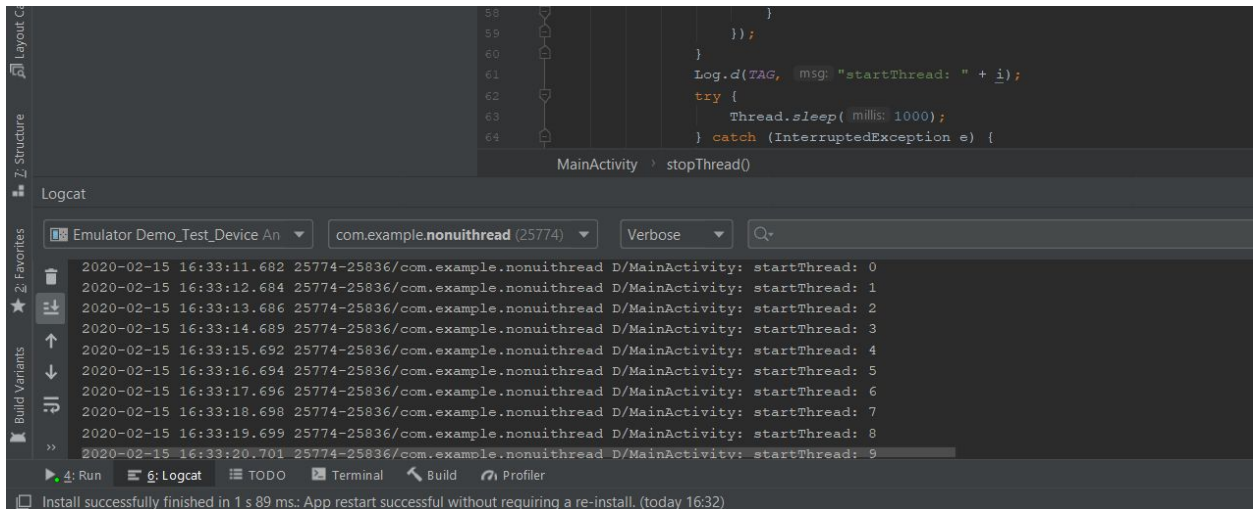
8
9 public class MainActivity extends AppCompatActivity {
10
11     private Button buttonStartThread;
12
13     private volatile boolean stopThread = false;
14
15     @Override
16     protected void onCreate(Bundle savedInstanceState) {
17         super.onCreate(savedInstanceState);
18         setContentView(R.layout.activity_main);
19
20         buttonStartThread = findViewById(R.id.button_start_thread);
21     }
22
23     public void startThread(View view) {
24         stopThread = false;
25         ExampleRunnable runnable = new ExampleRunnable( seconds: 10);
26         new Thread(runnable).start();
27     }
28
29     public void stopThread(View view) {
30         stopThread = true;
31     }

```

6.) Now let us make the thread execute a time consuming task. Let us make the thread sleep in the background for 10 seconds, this can be done by calling `Thread.sleep()` as shown in the screenshot below.

Now to make this newly created Non-UI thread interact with the Main thread we will use the `runOnUiThread()` function.

So how do we know that a thread is actually running in the background? We can verify this by making the background thread print a statement in the logcat.



To accomplish this we will set up a TAG for the mainactivity using the line pointed at by the arrow.



Using this tag we can make a print statement everytime a second elapses by using the code:

`Log.d(TAG, \"startThread: \" + i);` as shown in the screenshot below.

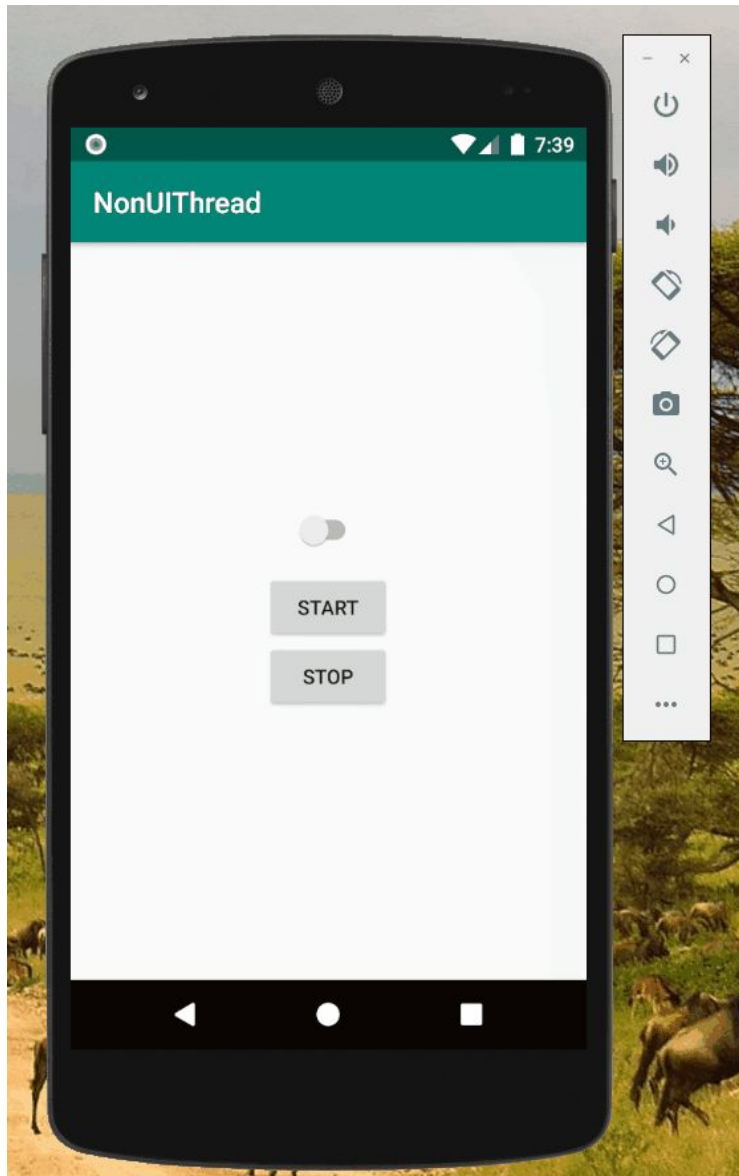
Now exactly at the 5th second we will change the text of the “START” button to “50%” signifying that the background thread has been sleeping for 5 seconds out of the 10 seconds that we mentioned earlier. We can make a change in the UI thread by using `runOnUiThread()`.

```

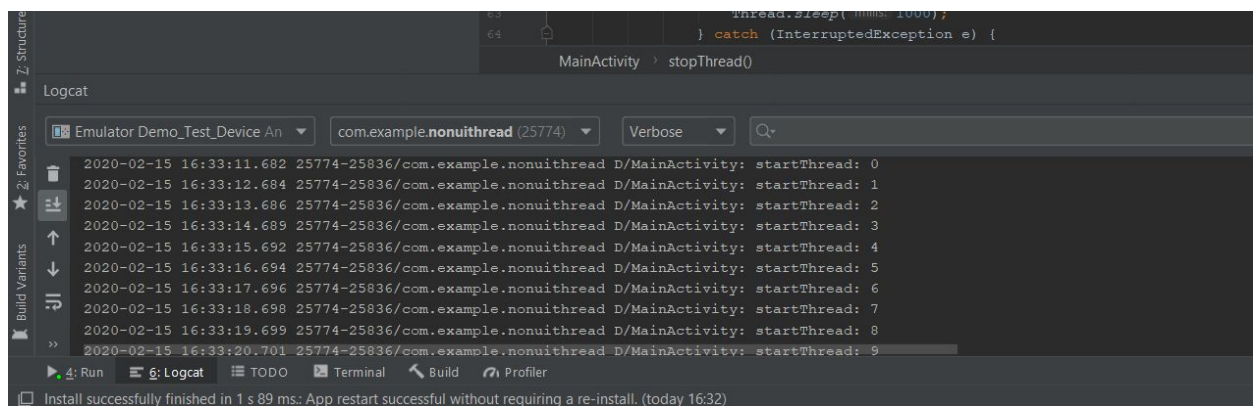
@Override
public void run() {
    for (int i = 0; i < seconds; i++) {
        if (stopThread) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    buttonStartThread.setText("Start");
                }
            });
            return;
        }
        if (i == 5) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    buttonStartThread.setText("50%");
                }
            });
        }
        Log.d(TAG, "msg: " + "startThread: " + i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            buttonStartThread.setText("Start");
        }
    });
}
}

```

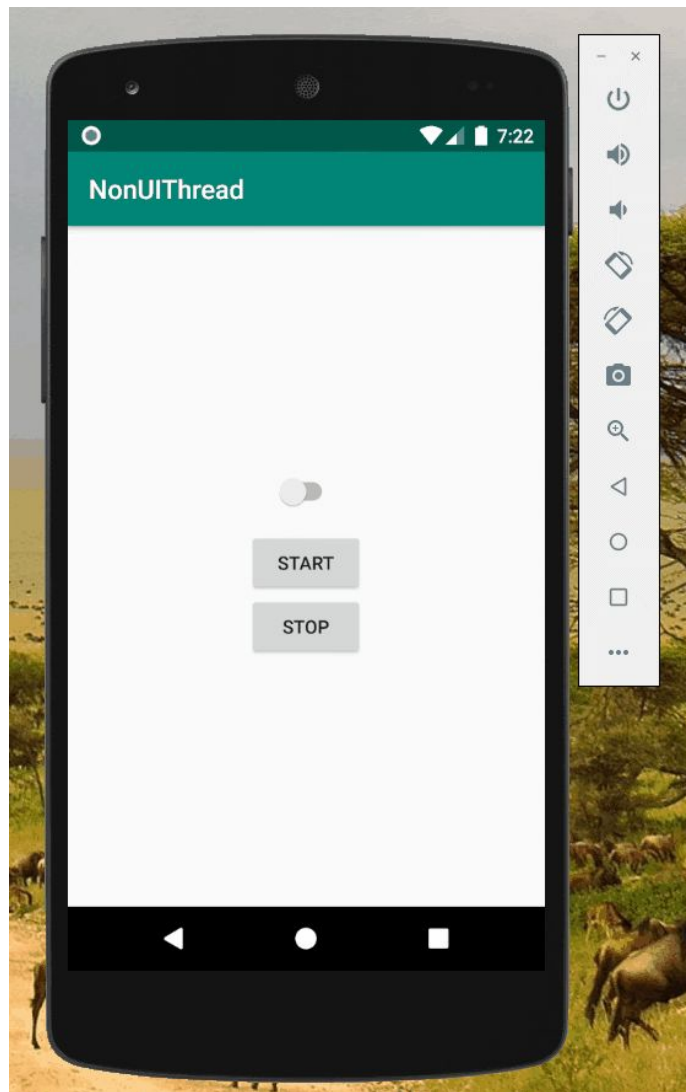
You can also notice that since the thread is sleeping in the background, you can click on the toggle switch and the app won't be frozen. If the main thread would be made to sleep for 10 seconds then we would not be able to toggle the switch or even use the stop button as shown below and the app will crash.



So in your app when you click start you can notice this printing in the logcat:



The final app will look like this when you hit the “Start” button:



Deliverables for Milestone 1 - Show a TA or a peer mentor:

1. The app running as it is in the GIF shown above to receive full credit for milestone1.

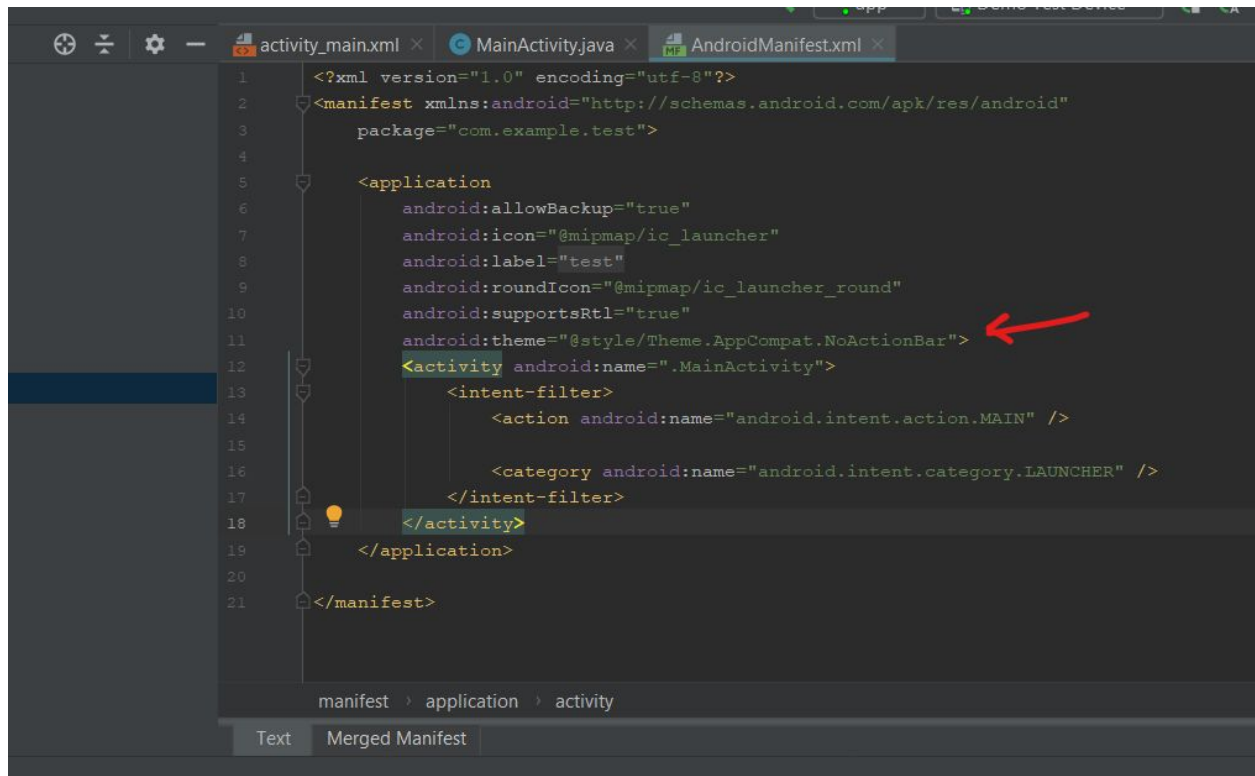
Milestone 2 - In Milestone 2, you will learn how to create a location tracker that asks users for permission to access the inbuilt GPS in your android device.

1.) Clone a new repository:

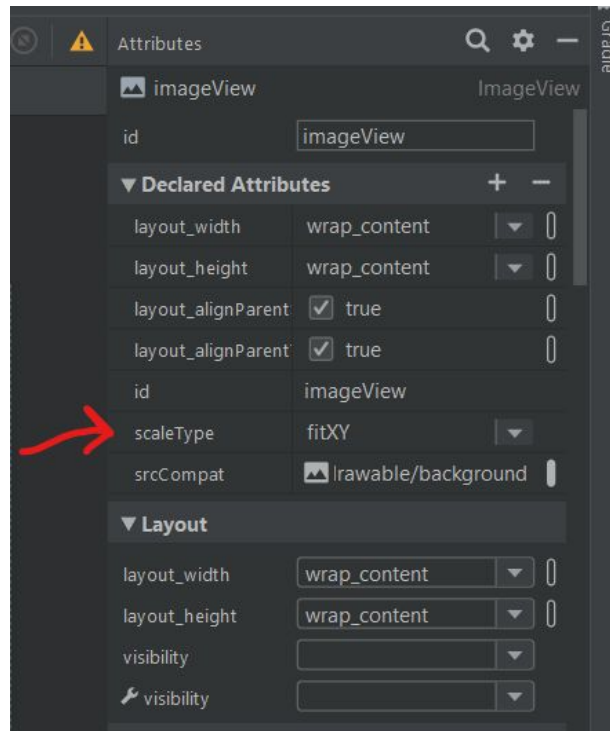
- Clone the Lab 4 Milestone 2 repository by selecting File > New > Project from Version Control > Git
- Then click the following link and accept the invitation:
<https://classroom.github.com/a/jiXf4WSW>
- You'll get a link in the form:
<https://github.com/CS-407-Spring2020/lab-4-milestone-2-yourgithubnamehere>

2.) Let us make our app fullscreen. We can do this by making a change in AndroidManifest.xml file. This file can be found in: app > manifests > AndroidManifest.xml

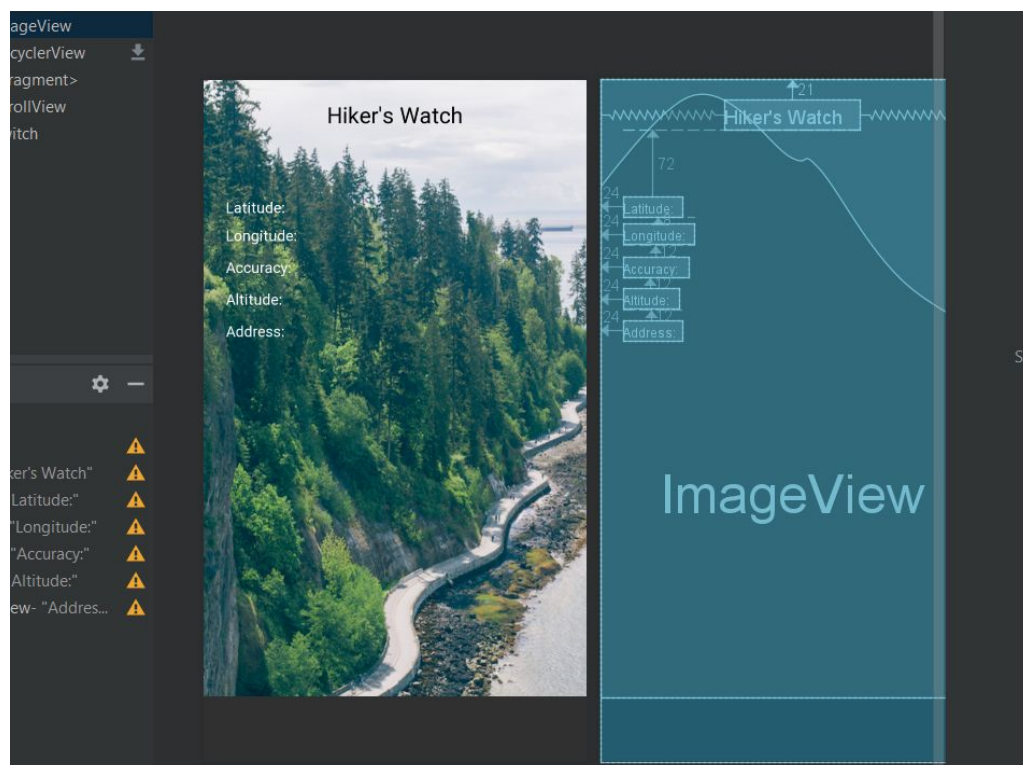
In this file change the android:theme from “@style/AppTheme” to “@style/Theme.AppCompat.NoActionBar”



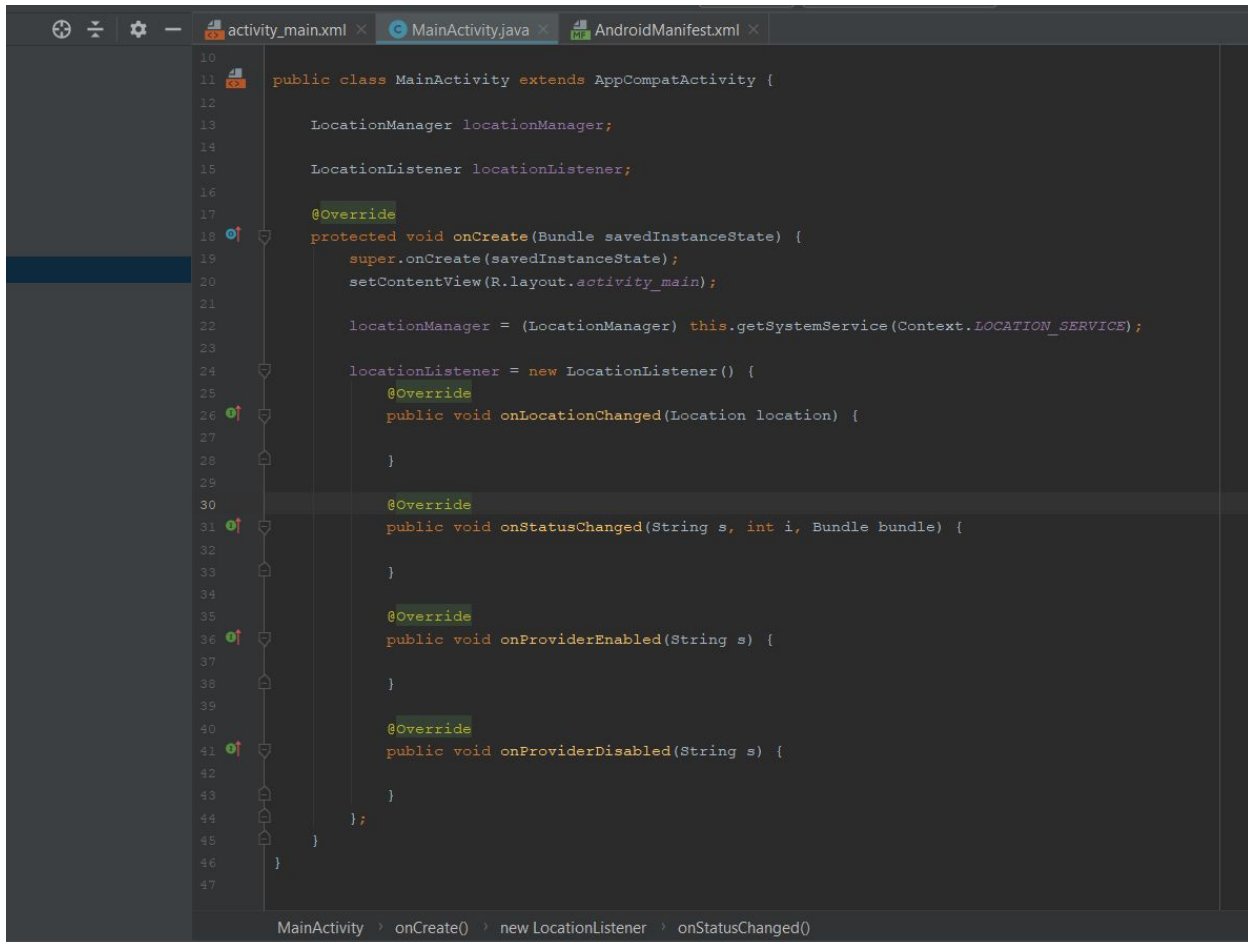
3.) You must have noticed that most applications have a nice background image. So let us set an image as the background image. We can do this by setting an imageview and changing the android: scaleType to “fitXY”. This can be done as so



4.) Now that we have set a background image. Let us create a few textviews to display our location information. Design the app layout like the way it is in the screenshot below:



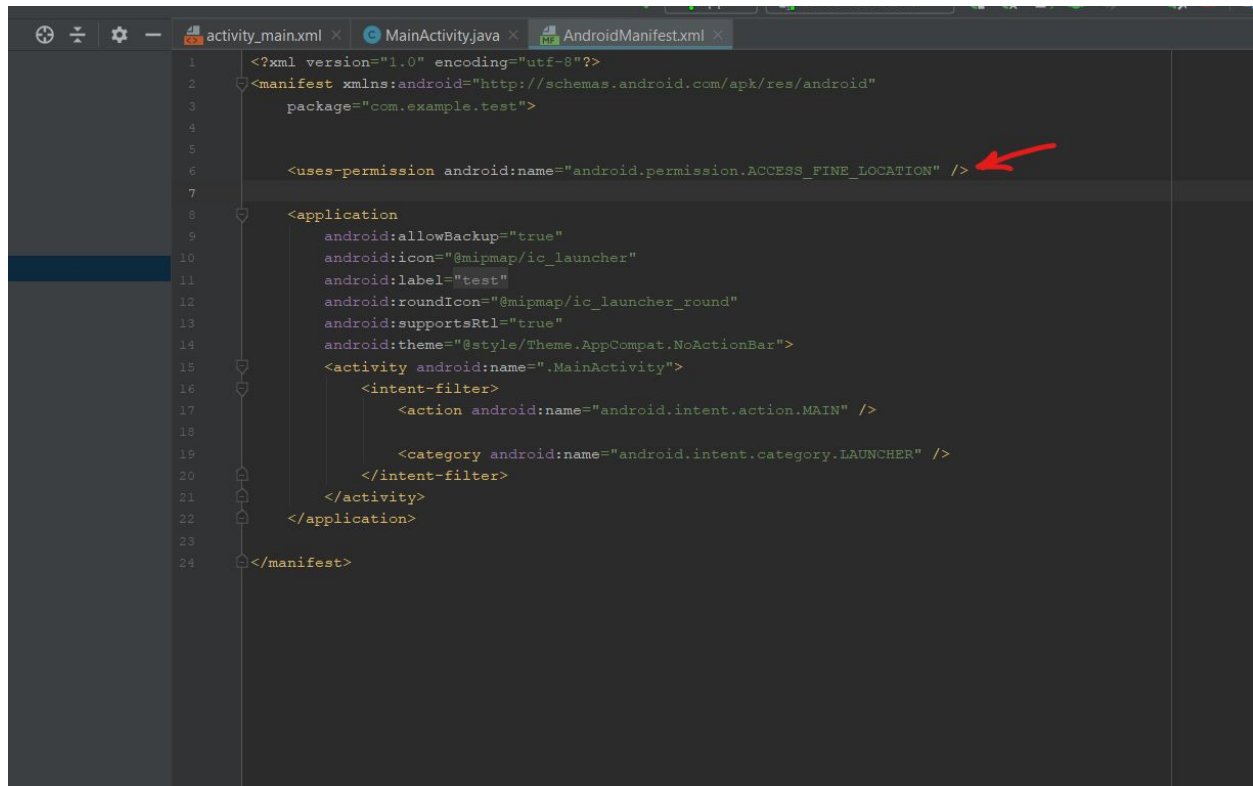
5.) Now that we have completed the aesthetic portion of the app let us write the code to get the location from our device and display it in these textviews. To do this we write the code in MainActivity.java. We will start by defining a LocationManager and a LocationListener. The line: `locationManager = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);` gets the location data from the android device. Using this location data in the location listener we can get the details that we need from the android device. In this app we will be working in the `onLocationChanged()` method as we will be displaying the location every time there has been a change.

A screenshot of an IDE window showing the MainActivity.java file. The code defines a MainActivity class that extends AppCompatActivity. It declares a LocationManager variable and a LocationListener interface. The onCreate method calls super.onCreate, setContentView, and initializes the locationManager. A new LocationListener is created with several overridden methods: onLocationChanged, onStatusChanged, onProviderEnabled, and onProviderDisabled. The breadcrumb at the bottom reads: MainActivity > onCreate() > new LocationListener > onStatusChanged().

```
10
11 public class MainActivity extends AppCompatActivity {
12
13     LocationManager locationManager;
14
15     LocationListener locationListener;
16
17     @Override
18     protected void onCreate(Bundle savedInstanceState) {
19         super.onCreate(savedInstanceState);
20         setContentView(R.layout.activity_main);
21
22         locationManager = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);
23
24         locationListener = new LocationListener() {
25             @Override
26             public void onLocationChanged(Location location) {
27
28             }
29
30             @Override
31             public void onStatusChanged(String s, int i, Bundle bundle) {
32
33             }
34
35             @Override
36             public void onProviderEnabled(String s) {
37
38             }
39
40             @Override
41             public void onProviderDisabled(String s) {
42
43             }
44         };
45     }
46 }
47
```

MainActivity > onCreate() > new LocationListener > onStatusChanged()

6.) Now we must ask the device for permission to use its location data. For devices running Android Marshmallow and above we need to ask the device for permission, otherwise we are not able to access the location data from the device. To do this we need to first add some in the AndroidManifest.xml



7.) Now that we have added the permission in the manifest file, we must write the code to request the permission from the user who will run this app. To do this we add the following in the MainActivity.java file in the onCreate() function:

```
if (Build.VERSION.SDK_INT >= 23 &&
    ActivityCompat.checkSelfPermission( context: this, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions( activity: this, new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, requestCode: 1);
}

locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, minTime: 0, minDistance: 0,locationListener);
Location location = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
if (location!=null) {
    // we will add code to update location info here
}
```

So if the build version is greater than or equal to 23 means that we do ask for permission. If the permission is granted then we can request location updates and use the location manager to get the last known location of the device. If our location is not null (i.e no error has occurred in receiving the location) then we can use it in the next steps.

8.) Now let us create a function in the MainActivity class to use the location listener and get the location details.

```
public void startListening() {  
    if (ContextCompat.checkSelfPermission( context: this, Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED) {  
        locationManager = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);  
    }  
}
```

9.) Once the user has given us the permission to use the location of the device we must process that permission result. We can do this by invoking the `onRequestPermissionsResult()` function in the MainActivity class as shown in the screenshot below:

```
@Override  
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {  
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);  
  
    if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {  
        startListening();  
    }  
}
```

Here again we check if the permission result (i.e grantResults in this function) is valid and is equal to `PERMISSION_GRANTED`.

Now we have successfully written the code to request the location permission from the user for the android device.

10.) Now we can add a `startListening()` function call in `onCreate()` :

```
startListening();  
  
if (Build.VERSION.SDK_INT >= 23 &&  
    ActivityCompat.checkSelfPermission( context: this, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {  
    ActivityCompat.requestPermissions( activity: this, new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, requestCode: 1);  
}  
  
else {  
    locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, minTime: 0, minDistance: 0, locationListener);  
    Location location = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);  
    if (location!=null) {  
        // we will add code to update location info here  
    }  
}
```

11.) Now lets us create a function to update the location info. We will call this function:
`updateLocationInfo(Location location)`

In this function we will set all the textviews with the corresponding information that we can retrieve from the location variable.

```
public void updateLocationInfo(Location location) {

    Log.i( tag: "LocationInfo", location.toString());

    TextView latTextView = (TextView) findViewById(R.id.latTextView);

    TextView lonTextView = (TextView) findViewById(R.id.lonTextView);

    TextView altTextView = (TextView) findViewById(R.id.altTextView);

    TextView accTextView = (TextView) findViewById(R.id.accTextView);

    latTextView.setText("Latitude: " + location.getLatitude());

    lonTextView.setText("Longitude: " + location.getLongitude());

    altTextView.setText("Altitude: " + location.getAltitude());

    accTextView.setText("Accuracy: " + location.getAccuracy());

}
```

12.) Now we must construct the address. To do this we must use a geocoder object. We do this right after the code in the above screenshot. By passing the latitude and longitude to the geocoder we can retrieve the address of the location. This can be done as shown in the screenshot below:

```
69         Geocoder geocoder = new Geocoder(getApplicationContext(), Locale.getDefault());
70
71         try {
72             String address = "Could not find address";
73             List<Address> listAddresses = geocoder.getFromLocation(location.getLatitude(), location.getLongitude(), (maxResults: 1);
74
75             if (listAddresses != null && listAddresses.size() > 0 ) {
76
77                 Log.i( tag: "PlaceInfo", listAddresses.get(0).toString());
78                 address = "Address: \n";
79
80                 if (listAddresses.get(0).getSubThoroughfare() != null) {
81                     address += listAddresses.get(0).getSubThoroughfare() + " ";
82                 }
83                 if (listAddresses.get(0).getThoroughfare() != null) {
84                     address += listAddresses.get(0).getThoroughfare() + "\n";
85                 }
86                 if (listAddresses.get(0).getLocality() != null) {
87                     address += listAddresses.get(0).getLocality() + "\n";
88                 }
89                 if (listAddresses.get(0).getPostalCode() != null) {
90                     address += listAddresses.get(0).getPostalCode() + "\n";
91                 }
92                 if (listAddresses.get(0).getCountryName() != null) {
93                     address += listAddresses.get(0).getCountryName() + "\n";
94                 }
95             }
96
97             TextView addressTextView = (TextView) findViewById(R.id.addressTextView);
98             addressTextView.setText(address);
99
100         } catch (IOException e) {
101             e.printStackTrace();
102         }
103     }
```

The multiple if checks are just checking if the geocoder returns valid data and if its valid it adds it to the address string. Once we have constructed the address string then we can update the same in the address text view.

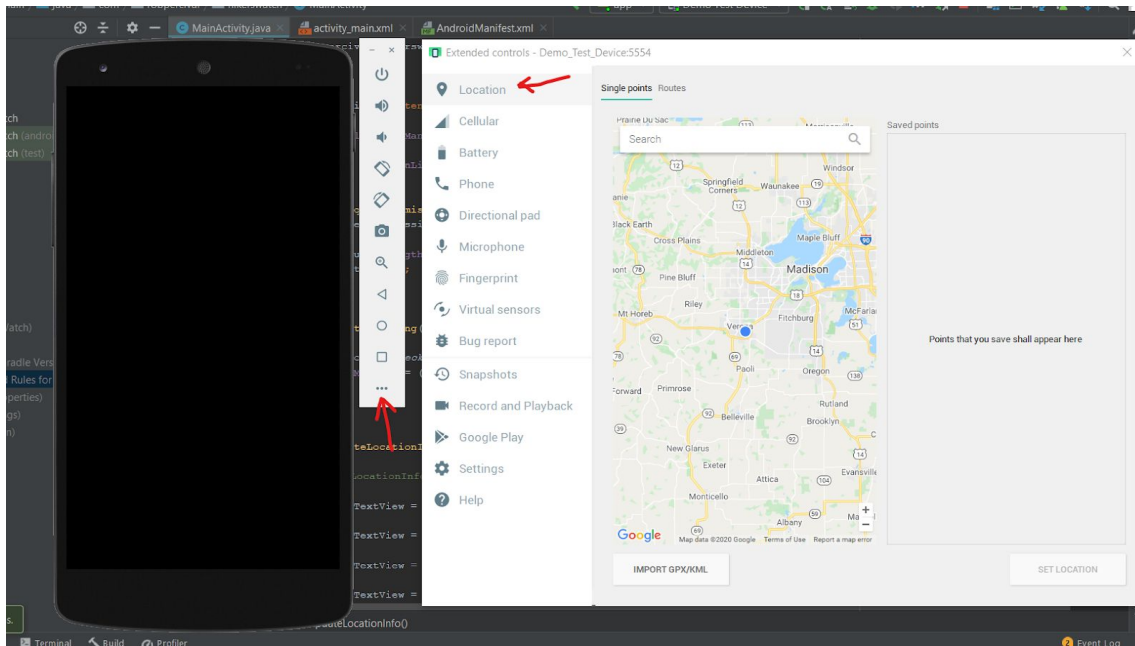
Now that we have the function `updateLocationInfo(Location,location)` ready, it needs to be called inside the `locationListener` we defined in Step 5.

```
105      @Override
106      protected void onCreate(Bundle savedInstanceState) {
107          super.onCreate(savedInstanceState);
108          setContentView(R.layout.activity_main);
109
110          locationManager = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);
111
112          locationListener = new LocationListener() {
113              @Override
114              public void onLocationChanged(Location location) {
115                  updateLocationInfo(location);
116              }
117          }
```

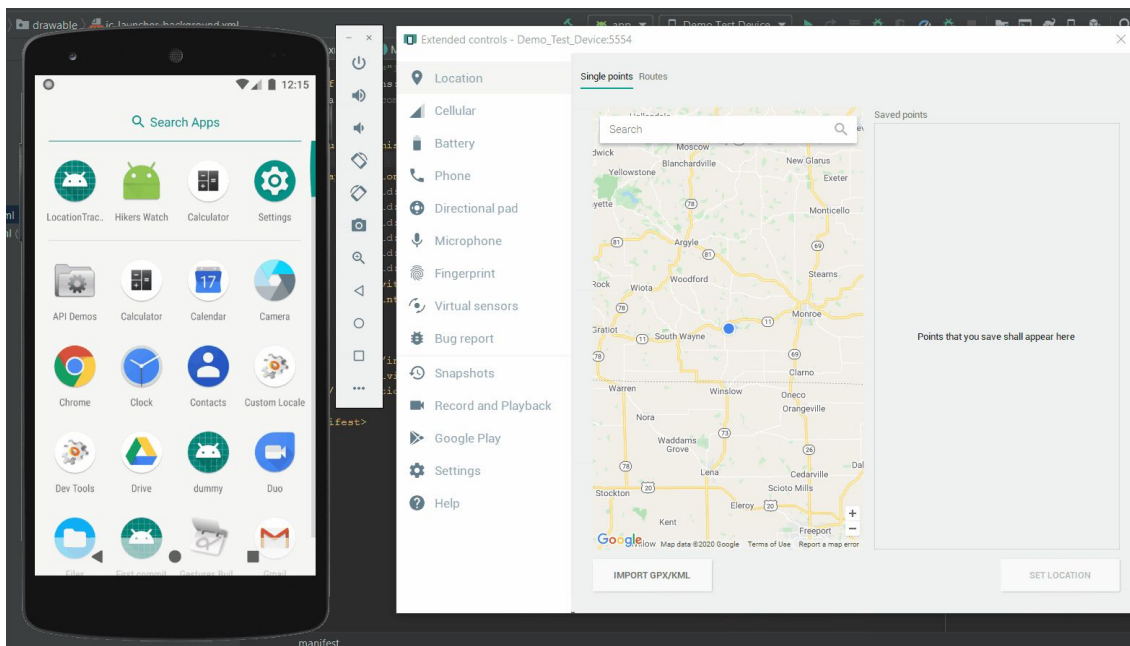
And in the other part in Step-10 where we had left a comment earlier.

```
locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, minTime: 0, minDistance: 0, locationListener);
Location location = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
if (location != null) {
    // We will add code here to update location information in the next steps
    updateLocationInfo(location);
}
```

13.) Now that we have successfully completed this step our app should be ready to use. Since most of you will be running this app on an Android Virtual Device we need to simulate the location of that virtual device. This can be done using the extended settings on the virtual device. We can open this menu by clicking the three dots as shown in the screenshot below. This will open a menu where we can update the location data by clicking on different spots on the map shown.



14.) The end product should look something like this:



(Note: Physical devices may not be able to get a location fix while inside a building. Please consult a TA for ways to simulate location on a physical device)

Deliverables for Milestone 2 - You can show the following to a TA or peer mentor to receive full credit for this milestone.

1. The popup that shows up to ask for permission to use the device's location.
2. Updation of the address details as shown in the above GIF

Conclusion:

Great job getting through this lab! Now you have had a taste of two strong utilities provided by android studio. Using threads you can accomplish many time consuming tasks in the background. There numerous uses of the location services provided by your android device and the example that we looked at in this lab is just one trivial application out of many that can make use of the location services.

References:

<https://codinginflow.com/tutorials/android/starting-a-background-thread>