

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

Due Oct 11 by 3:45pm **Points** 10 **Submitting** a file upload
Available Oct 4 at 12am - Oct 11 at 3:45pm 8 days

This assignment was locked Oct 11 at 3:45pm.

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

Introduction:

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 device's location by using its inbuilt GPS to make a simple 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 have 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 mock file downloader app to understand the use of non-UI threads in Android.

Background:

As you learned in the lectures, on the Android platform, applications operate, by default, on one thread. This thread is called the *UI Thread* or *Main 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 the 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 additional threads and put long running tasks on these non-UI threads. You can create such an alternative thread by either:

- Creating and starting your own `java.lang.Thread`
- Creating and starting an `AsyncTask` (Android's 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.

Setup:

Create a new project and set up your app to have two buttons and one toggle switch, as shown in Figure M1-1.

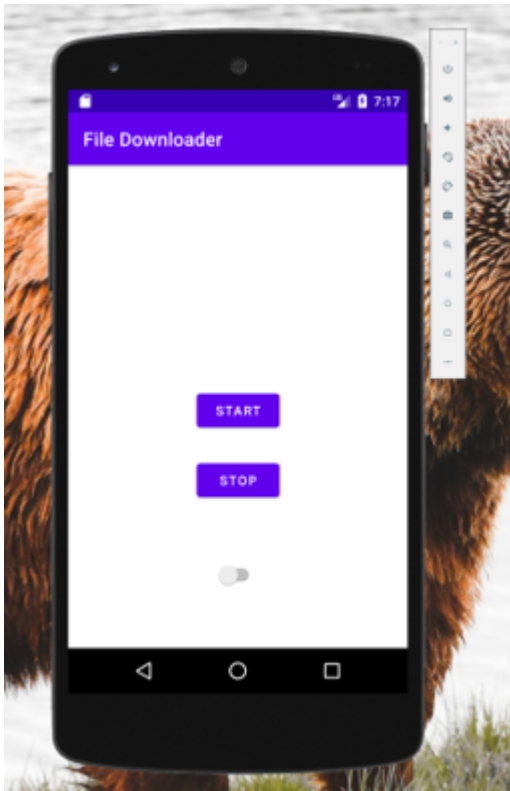


Figure M1-1: Our basic app layout for Milestone 1

Now, let's simulate a file download (we won't actually be downloading any files, just pretending we are as it is a common example of a task that should be done on a non-UI thread).

To do this, we will define a `mockFileDownloader()` method that will output the progress of our 'download' to Logcat as if we were actually downloading a file (see Figure M1-2). We will do this through the use of the `Thread.sleep()` method. This method causes whatever thread it is running on to pause for a number of milliseconds. We put it inside a for loop so it will pause each time through the for loop for 1 second. We also update our 'Download Progress' percentage in the for loop. This makes it look like our 'Download Progress' percentage is increasing by 10% each second.

We want our 'file download' to start when we click the 'Start' button. So we need to now define a `startDownload()` method (see Figure M1-2). In this method, we want to call our `mockFileDownloader()` method. We also need to associate this method with our 'Start' button by assigning its `OnClick` attribute to be 'startDownload' so that it runs when the button is clicked.

```

public class MainActivity extends AppCompatActivity {

    private static final String TAG = "MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void mockFileDownloader() {
        for (int downloadProgress = 0; downloadProgress <= 100; downloadProgress=downloadProgress+10) {
            Log.d(TAG, msg: "Download Progress: " + downloadProgress + "%");
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void startDownload(View view){
        mockFileDownloader();
    }
}

```

Figure M1-2: Code for the mockFileDownloader() and startDownload() methods

The Problem With Using Only the Main Thread:

Now, before you test the code we wrote above, consider what you think will happen. Specifically, what do you think will happen with regards to the UI and in Logcat after you press the start button?

What you should be experiencing is that the UI has 'frozen' (you can't click on any buttons or the toggle). However, you can see that our mock file downloader is in fact running by looking at Logcat - it should be outputting an increasing percentage each second (see Figure M1-3). Your app may even crash if you press the 'Start' button too many times and wait. Why do you think this is?

```

2021-09-25 19:42:07.648 16837-16875/com.example.filedownloader D/MainActivity: Download Progress: 0%
2021-09-25 19:42:08.650 16837-16875/com.example.filedownloader D/MainActivity: Download Progress: 10%
2021-09-25 19:42:09.653 16837-16875/com.example.filedownloader D/MainActivity: Download Progress: 20%
2021-09-25 19:42:10.655 16837-16875/com.example.filedownloader D/MainActivity: Download Progress: 30%
2021-09-25 19:42:11.661 16837-16875/com.example.filedownloader D/MainActivity: Download Progress: 40%
2021-09-25 19:42:12.666 16837-16875/com.example.filedownloader D/MainActivity: Download Progress: 50%
2021-09-25 19:42:13.674 16837-16875/com.example.filedownloader D/MainActivity: Download Progress: 60%
2021-09-25 19:42:14.679 16837-16875/com.example.filedownloader D/MainActivity: Download Progress: 70%
2021-09-25 19:42:15.688 16837-16875/com.example.filedownloader D/MainActivity: Download Progress: 80%
2021-09-25 19:42:16.696 16837-16875/com.example.filedownloader D/MainActivity: Download Progress: 90%
2021-09-25 19:42:17.699 16837-16875/com.example.filedownloader D/MainActivity: Download Progress: 100%

```

Figure M1-3: Logcat expected output

Recall from the previous labs that when a button is clicked, several things occur. First, the UI updates and the button changes to a darker color. Second, the 'OnClick' Function is called and runs to completion. Third, the UI updates again and the button returns to its original color. All this happens sequentially on the Main Thread every time a button is clicked.

However, this presents us with a problem - what if the OnClick function takes a long time to complete execution, like it would if it was downloading a file (or, in our case, pretending to download a file)? The answer is that the app will remain frozen until the completion of the OnClick function.

So how can we handle long running tasks without freezing? The answer is that we can run them in the background using non-UI threads. For our app, that means deferring running the `mockFileDownload()` method (which is currently running on our main thread) to a non-UI thread.

Creating a Non-UI Thread:

The first step to creating a non-UI thread is to implement a *runnable interface* (see Figure M1-4). This interface will be used to tell our thread what task we want it to work on - specifically we put whatever we want it to do in the `run()` method. For us, this means that we put our call to `mockFileDownloader()` in there (remember to delete the `mockFileDownloader()` call from `startDownload()`).

```
class ExampleRunnable implements Runnable {  
    @Override  
    public void run() {  
        mockFileDownloader();  
    }  
}
```

Figure M1-4: Code for a Runnable Interface that runs our `mockFileDownloader()`

The next step is to actually create and run our thread (see Figure M1-5). Since we want our thread to start running when we press the 'Start' button, we want to put this code in the `startDownload()` method.

```
public void startDownload(View view){  
    ExampleRunnable runnable = new ExampleRunnable();  
    new Thread(runnable).start();  
}
```

Figure M1-5: Code for creating and running a new Thread using our Runnable Interface

At this point, consider what you think will happen when you run your code with these changes?

What you should be seeing is the Download Progress updates in Logcat still working as they were last time, but now, the UI of the app should NOT be frozen - we can click on buttons and the toggle switch and they will update their appearance! This is because we have migrated our 'file download' task off of the main thread and onto a non-UI thread where it can run in the background.

Updating UI Using Non-UI Threads:

As mentioned earlier in the lab, standard non-UI threads cannot be used to directly update our UI appearance. To see this, all we need to do is try to change the 'Start' button's text to 'Downloading...' while our `mockFileDownloader` is running (see Figure M1-6).

```

public class MainActivity extends AppCompatActivity {

    private static final String TAG = "MainActivity";
    private Button startButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        startButton = findViewById(R.id.startButton);
    }

    public void mockFileDownloader() {

        startButton.setText("DOWNLOADING...");

        for (int downloadProgress = 0; downloadProgress <= 100; downloadProgress=downloadProgress+10) {
            Log.d(TAG, "msg: \"Download Progress: \" + downloadProgress + \"%");
            try {
                Thread.sleep( mills: 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

Figure M1-6: Example code to demonstrate the issue with modifying UI from a non-UI thread

If we run this, we will get an error message telling us that this thread cannot modify the View and the app should crash. Essentially, it is telling us that we can only modify the UI from the UI thread.

Since this is the case we need some way of running portions of our code which are running on a non-UI thread on the UI thread. Lucky for us, we have the `runOnUiThread()` method. This method does exactly what it's name suggests - it runs its contents on the UI thread, which is exactly what we need to be able to do to modify the View and change our button's text (see Figure M1-7)! We can even add multiple `runOnUiThread()` methods, which will allow us to switch the button's text back to 'Start' once the 'download' has finished (see Figure M1-7).

```

public void mockFileDownloader() {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            startButton.setText("Downloading...");
        }
    });

    for (int downloadProgress = 0; downloadProgress <= 100; downloadProgress=downloadProgress+10) {
        Log.d(TAG, "msg: \"Download Progress: \" + downloadProgress + \"%");
        try {
            Thread.sleep( mills: 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            startButton.setText("Start");
        }
    });
}

```

Figure M1-7: Code to demonstrate the use of the `runOnUiThread()` method. Specifically used for changing the 'Start' button's text at the beginning and end of the download process.

Now, once you run the app, when you click the 'Start' button it should change its text to 'DOWNLOADING...', Logcat should still be displaying the 'download' progress, and you should be

able to interact with the UI and do things like toggle the switch (see Figure M1-8).

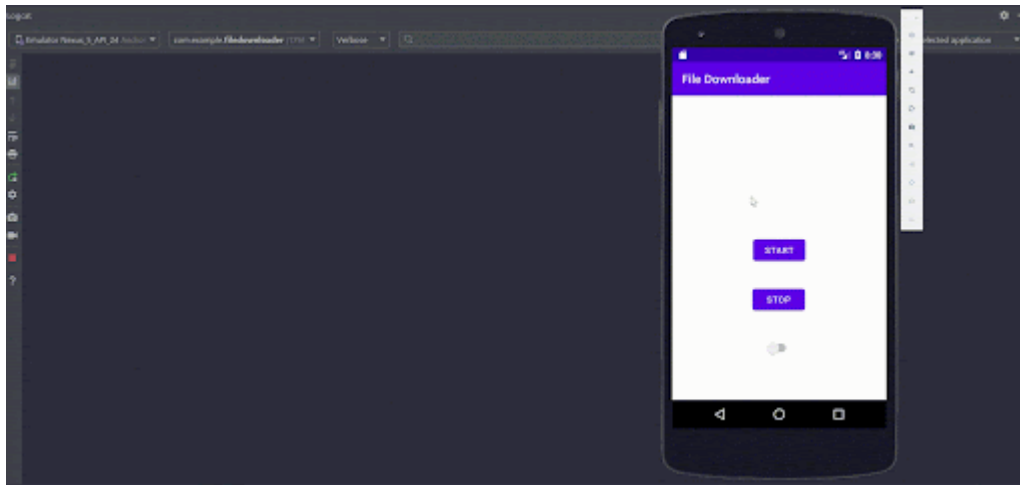


Figure M1-8: Gif showing the expected results of modifying our UI using the `runOnUiThread()` method

Terminating a Thread Early:

Now, we can turn our attention to the 'Stop' button. When we click this button we want to stop the 'download'. A thread automatically stops when it reaches the end of its `run()` method. For our non-UI thread, the only thing in its `run()` is a call to the `mockFileDownloader()`. This means, in order to terminate our thread early, we simply need to reset the 'Start' button text and return from `mockFileDownloader()` early. We can do this by defining a `stopThread` boolean variable, which we set to true when the stop button is clicked - indicating we want the download to stop- and false when the start button is clicked (see Figure M1-9). We then add code into `mockFileDownloader()` to reset the 'Start' button text and return early when the boolean is true (see Figure M1-10).

```
public class MainActivity extends AppCompatActivity {

    private static final String TAG = "MainActivity";
    private Button startButton;
    private volatile boolean stopThread = false;

    public void startDownload(View view){
        stopThread = false;
        ExampleRunnable runnable = new ExampleRunnable();
        new Thread(runnable).start();
    }

    public void stopDownload(View view){
        stopThread = true;
    }
}
```

Figure M1-9: The creation and usage of the `stopThread` variable used to track of the state of our non-UI thread

```

public void mockFileDownloader() {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            startButton.setText("Downloading...");
        }
    });

    for (int downloadProgress = 0; downloadProgress <= 100; downloadProgress=downloadProgress+10) {
        if (stopThread) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    startButton.setText("Start");
                }
            });
            return;
        }

        Log.d(TAG, msg, "Download Progress: " + downloadProgress + "%");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                startButton.setText("Start");
            }
        });
    }
}

```

Figure M1-10: Code showing how to use our stopThread boolean in mockFileDownloader to end the function early

Your Turn!:

The final task for you for this milestone is to finish up our pretend file downloader app by having it display the download progress (the percentage currently being output to Logcat), on the app itself. The final product should look like the gif in Figure M1-11.

**Tip:* Depending on your approach, you may receive an error message along the lines of 'Variable 'downloadProgress' is accessed from within the inner class, needs to be final or effectively final'. The easiest way to remedy this is to simply follow the suggested fix by Android Studio and copy the value you are working with to an effectively final variable (hover over the error and press 'Alt-Shift-Enter' and it will do it automatically for you).

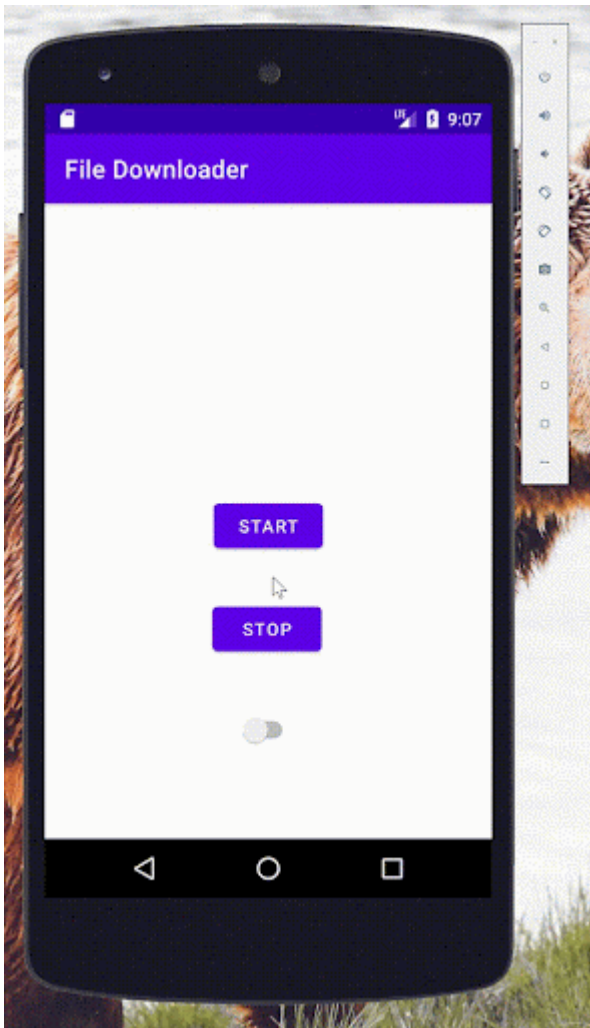


Figure M1-11: Gif showing the expected final results of this Milestone. Your finished product should resemble and function like this app (feel free to make it prettier if you'd like though haha)

Milestone 2:

In Milestone 2, we will learn how to create a location tracker that asks users for permission to access the inbuilt GPS in their Android device.

Setup:

Create a new project with an Empty Activity. Next, let us make our app fullscreen by modifying the AndroidManifest.xml file. This file can be found in: app → manifests → AndroidManifest.xml. Once located, set android:theme to “@style/Theme.AppCompat.NoActionBar” (See Figure M2-1).


```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.gpsdemo">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="GPS Demo"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.AppCompat.NoActionBar">

        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Figure M2-1: Updated android:theme attribute to make our app fullscreen

Now, we will add a background image to our app. To do this we will navigate to our activity_main.xml file's Design View and create an ImageView (just as you would create a TextView, see Figure M2-2). Once created, the Android Studio Resource Manager will pop up and you will then be prompted to select the image you wish to appear in your ImageView. Chances are, you won't have any suitable background images already available on Android Studio. As such, we will need to import an image (any image you would like)! There are two main ways of doing this:

1. Simply drag the image file you wish to import on top of the resource manager (the window that popped up when you created your image view).
2. Click the '+' in the upper left corner and select 'Import Drawables' and choose whichever image you would like to use.

Once you have imported your background image, you simply need to select it and press 'ok'. The final thing we need to do is set the scaleType attribute of our ImageView to be 'fitXY' (this will expand the image to cover the entire background). You can find this attribute under 'Common Attributes' in the Attributes menu of our ImageView in the Design View (see Figure M2-2).

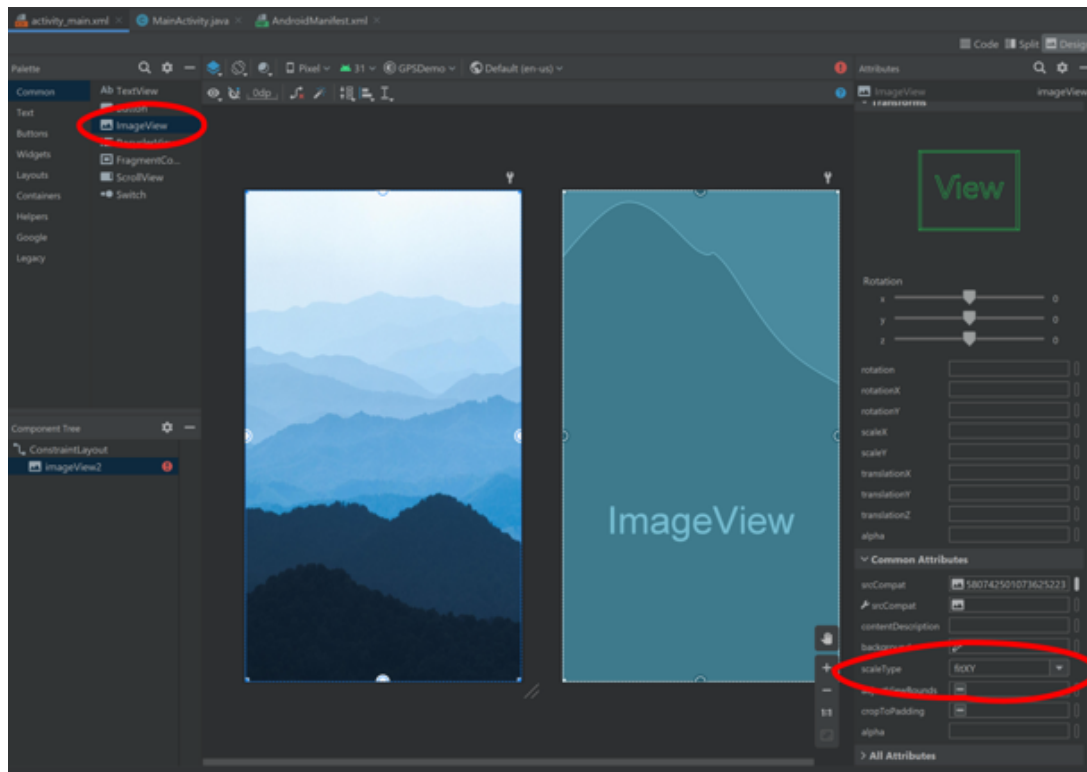


Figure M2-2: Options for creating an ImageView and modifying its scaleType attribute

Now that we have set a background image, let's create a few TextViews to display the location information our app will be gathering. Finish your app layout so that it has the same TextViews and looks something similar to the screenshot in Figure M2-3.

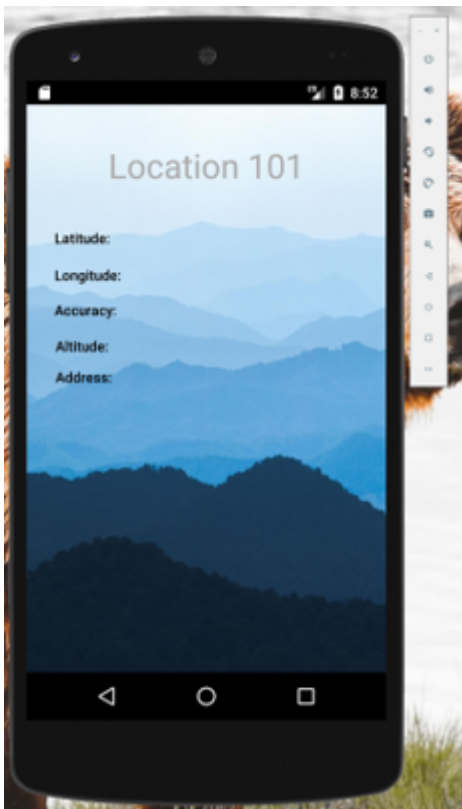


Figure M2-3: Expected layout (stylistic choices like background, color, and app name are left to you)

Getting and Using Location Data:

Now that we have finished setting up our app's layout, we turn our attention to getting the location data of our user and displaying it in our TextViews.

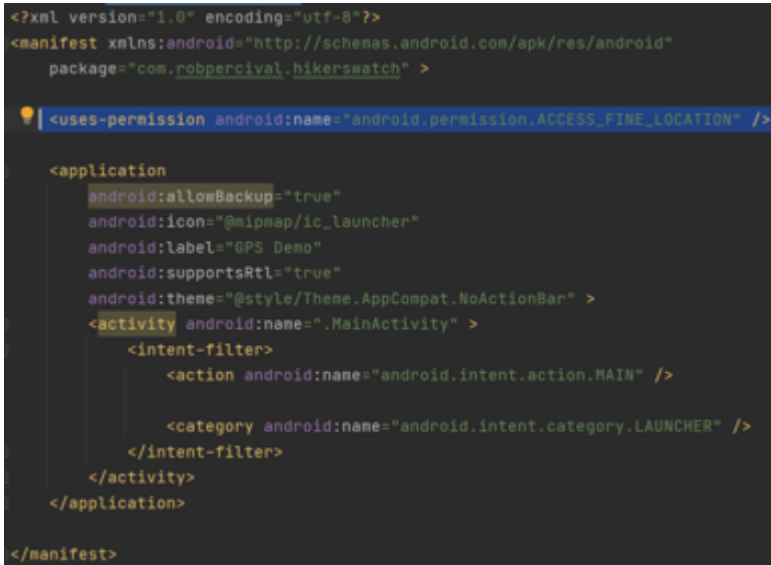
Most of this work will be done in our MainActivity.java file. We start by defining and creating a LocationManager (see Figure M2-4). This location manager gets the location data from the Android device. Next, we will create a LocationListener which will help us use the data from our LocationManager (see Figure M2-4).

```
LocationManager locationManager;  
LocationListener locationListener;  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
  
    locationManager = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);  
    locationListener = new LocationListener() {  
        @Override  
        public void onLocationChanged(@NonNull Location location) {  
            updateLocationInfo(location);  
        }  
        @Override  
        public void onStatusChanged(String s, int i, Bundle bundle){  
        }  
        @Override  
        public void onProviderEnabled(String s){  
        }  
        @Override  
        public void onProviderDisabled(String s){  
        }  
    };  
};
```

Figure M2-4: Code for creating a LocationManager and LocationListener

Requesting Permissions:

However, for devices running Android Marshmallow and above, we can't just access the location data automatically - we need to ask the device for permission. To do this, we need to add our permission tag to our AndroidManifest.xml file (see Figure M2-5) and then we need to prompt the user to actually give us those permissions in the onCreate() function (see Figure M2-6).



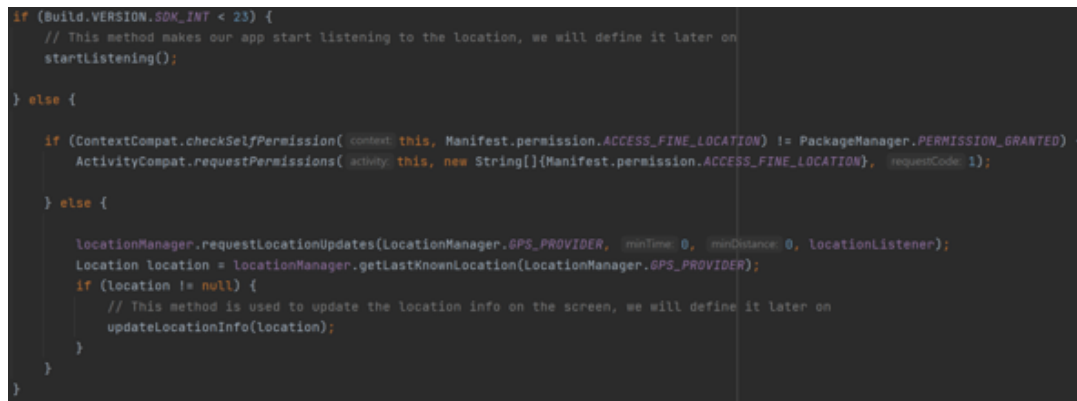
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.robpercival.hikerswatch" >

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="GPS Demo"
        android:supportRtl="true"
        android:theme="@style/Theme.AppCompat.NoActionBar" >
        <activity android:name=".MainActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Figure M2-5: The uses-permission tags necessary for accessing the location data we will be using

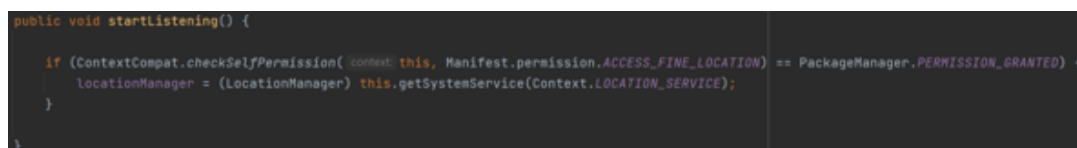


```
if (Build.VERSION.SDK_INT < 23) {
    // This method makes our app start listening to the location, we will define it later on
    startListening();
} else {
    if (ContextCompat.checkSelfPermission(context, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(activity, new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, 1);
    } else {
        locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, locationListener);
        Location location = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
        if (location != null) {
            // This method is used to update the location info on the screen, we will define it later on
            updateLocationInfo(location);
        }
    }
}
```

Figure M2-7: The prompting of the user to give us permission to access the device's location in onCreate()

Breaking down what we just did. We checked to see if the build version is greater than or equal to 23 (which means we do have to ask for permission). Then, if the user grants us permission, we request location updates and use the location manager to get the last known location of the device.

Now, a brief aside from the handling of the permissions. This is a good time to create a startListening() method in the MainActivity (see Figure M2-8).



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

Figure M2-8: The startListening() method used for getting location details

Now, back to the permissions! 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() method (see Figure M2-9). This method checks if the permission result

is valid and is equal to `PERMISSION_GRANTED` and then calls our `startListening()` method that we just defined.

```
@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();
    }
}
```

Figure M2-9: The `onRequestPermissionsResult()` method for processing permission results

Using the Location Data:

Now, let's create a function to update the location info on our screen. In this function, we will set all the `TextView`'s `Text` attributes to their corresponding location values (see Figure M2-10).

```
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());

}
```

Figure M2-10: The `updateLocationInfo()` method used for displaying the location data on the screen

Now, we are not done with our update method quite yet! You may have noticed that we have not updated our Address `TextView` yet - that is because in order to do so we must construct the address. To do this we will use a `Geocoder` object. By passing our `Geocoder` object our latitude and longitude we can retrieve the address of the location (see Figure M2-11).

```

Geocoder geocoder = new Geocoder(getApplicationContext(), Locale.getDefault());

try {
    String address = "Could not find address";
    List<Address> listAddresses = geocoder.getFromLocation(location.getLatitude(), location.getLongitude(), maxResults: 1);

    if (listAddresses != null && listAddresses.size() > 0 ) {

        Log.i( tag: "PlaceInfo", listAddresses.get(0).toString());
        address = "Address: \n";
        if (listAddresses.get(0).getSubThoroughfare() != null) {
            address += listAddresses.get(0).getSubThoroughfare() + " ";
        }
        if (listAddresses.get(0).getThoroughfare() != null) {
            address += listAddresses.get(0).getThoroughfare() + "\n";
        }
        if (listAddresses.get(0).getLocality() != null) {
            address += listAddresses.get(0).getLocality() + "\n";
        }
        if (listAddresses.get(0).getPostalCode() != null) {
            address += listAddresses.get(0).getPostalCode() + "\n";
        }
        if (listAddresses.get(0).getCountryName() != null) {
            address += listAddresses.get(0).getCountryName() + "\n";
        }
    }

    TextView addressTextView = (TextView) findViewById(R.id.addressTextView);
    addressTextView.setText(address);

} catch (IOException e) {
    e.printStackTrace();
}

```

Figure M2-11: Code for constructing our address and displaying in on the screen

The conditionals in this code are simply checking to make sure that the geocoder returns valid data. And then we are simply updating the address TextView with our newly constructed address.

Okay, we have created our update method, but now where should we call it? Well we want our location data to update both at the start of the app and whenever we move locations. As such, we need to call our update method twice. Once at the end of our onCreate() method and once in our LocationListener's onLocationChanged() method (see Figure M2-12).

```

LocationListener = new LocationListener() {
    @Override
    public void onLocationChanged(Location location) {
        updateLocationInfo(location);
    }

    @Override
    public void onStatusChanged(String s, int i, Bundle bundle) {
    }

    @Override
    public void onProviderEnabled(String s) {
    }

    @Override
    public void onProviderDisabled(String s) {
    }
};

if (Build.VERSION.SDK_INT < 23) {
    // This method makes our app start listening to the location, we will define it later on
    startListening();
} else {
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, requestCode: 1);
    } else {
        locationManager.requestLocationUpdates(locationManager.GPS_PROVIDER, 0, 0, locationListener);
        Location location = locationManager.getLastKnownLocation(locationManager.GPS_PROVIDER);
        if (location != null) {
            // This method is used to update the location info on the screen, we will define it later on
            updateLocationInfo(location);
        }
    }
}

```

Figure M2-12: Location of calls to our update method

Testing:

And with that we should have a functional app - now time for testing! When dealing with things like locations it would be very impractical if we actually had to physically travel to a bunch of different locations to test our app. Luckily for us, our Android Virtual Devices can simulate the location of our device. This can be done by using the extended settings on the virtual device. We can open this menu by clicking the three dots as shown in Figure M2-13. This will open a menu where we can update the location data by clicking on different spots on the map.

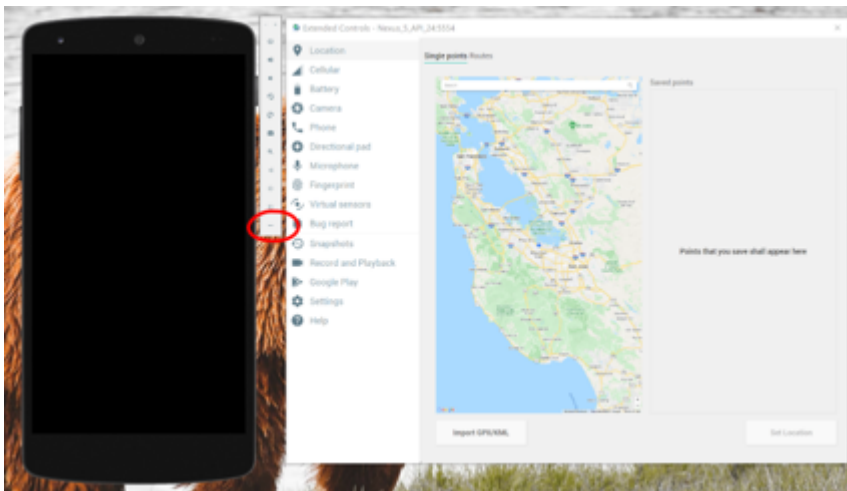


Figure M2-13: Navigation to extended settings to simulate location of device

Your final product should behave like and look similar to the gif in Figure M2-14 (Note, the way we have set things up, you may have to exit and restart the app once you have given it location permissions for it to work properly).

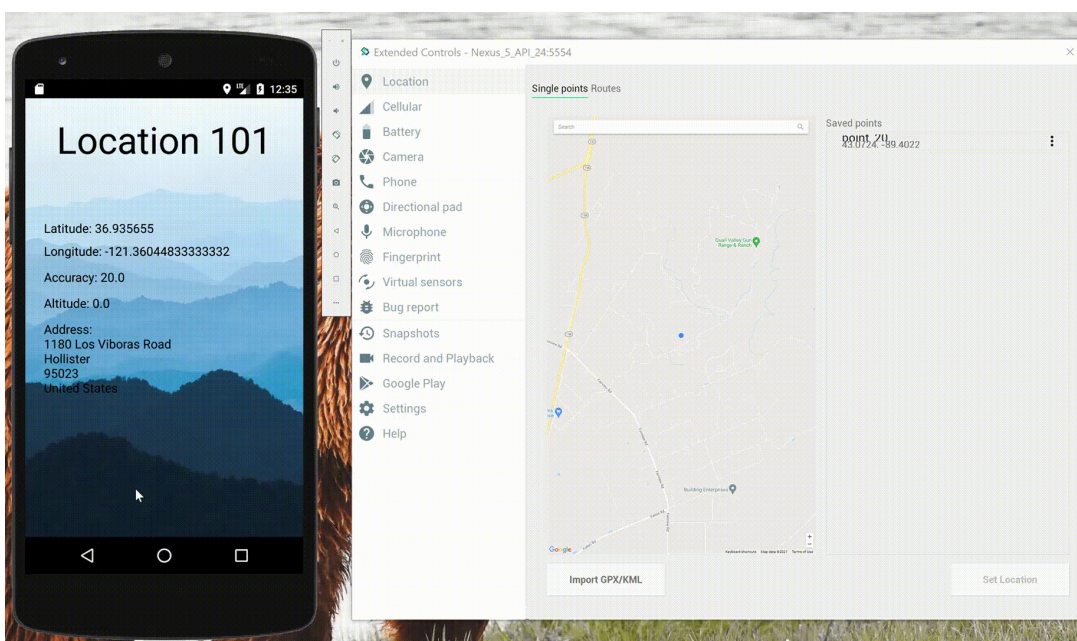


Figure M2-14: Expected behavior of finished app

Deliverables

- Show a TA or peer mentor your app from Milestone 1 functioning as the app in Figure M1-11 is to receive full credit for this milestone. ****Please have the emulator and app loaded and ready to grade when you come to get this milestone checked off****
- Show a TA or peer mentor your app from Milestone 2 functioning as the app in Figure M2-14 to receive full credit for this milestone. ****Please have the emulator and app loaded and ready to grade when you come to get this milestone checked off****
- Commit and push all the code for each milestone to personal Github repositories.

Some Rubric			
Criteria	Ratings		Pts
Milestone 1 Student app functions like figure M1-11	5 pts Full Marks	0 pts No Marks	5 pts
Milestone 2 Student app functions like figure M2-14	5 pts Full Marks	0 pts No Marks	5 pts
			Total Points: 10