CSCI 4237

Software Design for Handheld Devices

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Lecture - Permissions & Location

Mike Cobb

Upcoming

- 4/9 Project 2 Check-in #1
- 4/16 quiz 4
- 4/23 project 2 presentations
- 4/30 Make up date/project 2 due
- 5/7 Final Exam Day (online-asynchronous)
-

Last Time

- Firebase DB
- Crashlytics
- Analytics

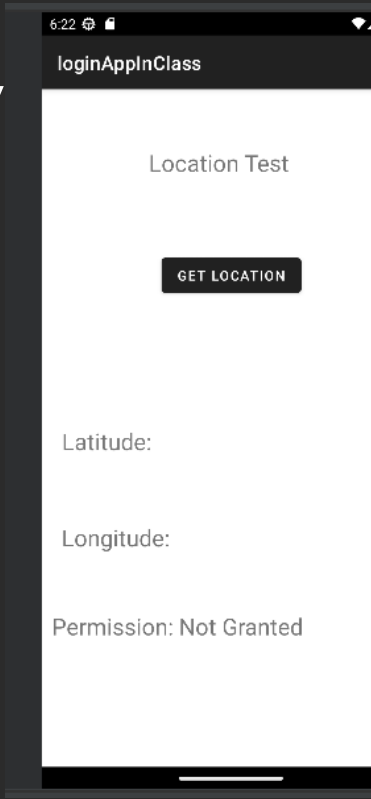
Today

- Quiz 3
- Permissions
- Location

After Quiz

- In your yelp app (or new project), add a new activity
- Need a button and 4 text Views.
- In code bind the button and the three views below the button.

Once done. Add a shortcut button on the login screen To go to this activity.

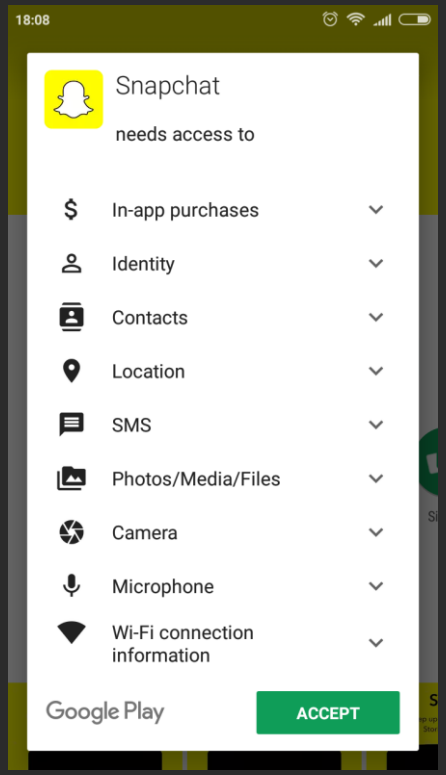


Quiz 3

- 15 minutes
- Auto submit
- Short answer
- 3 questions (20 points)
- Q2 has 5 parts. all the answer needs is:
    - Crashlytics, analytics, none, or both

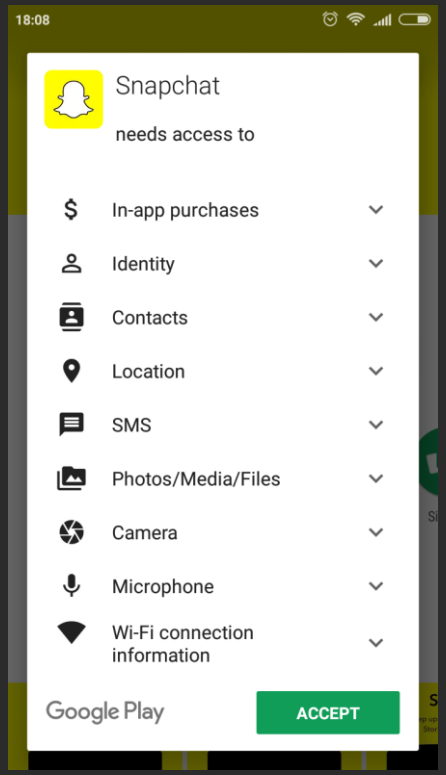
Permissions

Permissions



In order to use data or hardware from other parts of the device, an app needs to declare *permissions*.

Permissions



In order to use data or hardware from other parts of the device, an app needs to declare *permissions*.

Either at *installation time* or at *runtime*, these permissions are granted to the app.

Permissions

You declare permissions in your Android Manifest file.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="edu.gwu.androidtweetsfall2021">

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

Permissions

You declare permissions in your Android Manifest file.

List of permissions

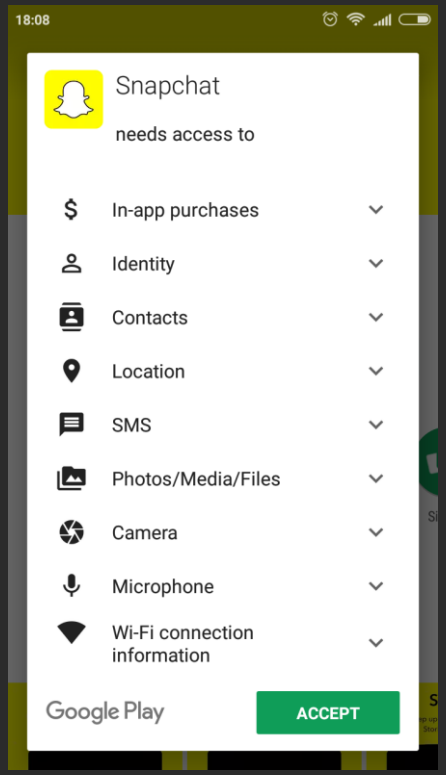
```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="edu.gwu.androidtweetsfall2021">

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

Fine vs Coarse

- ACCESS_FINE_LOCATION includes GPS data in reporting user location

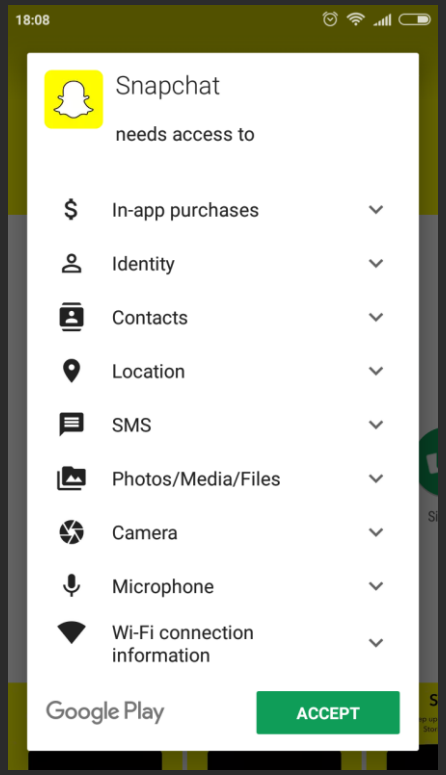
- ACCESS_COARSE_LOCATION includes data from the most battery-efficient non-GPS provider available (for example, the network).

Permissions



Before Android 6.0 (Marshmallow, API 23), **all** permissions were granted at installation.

Permissions



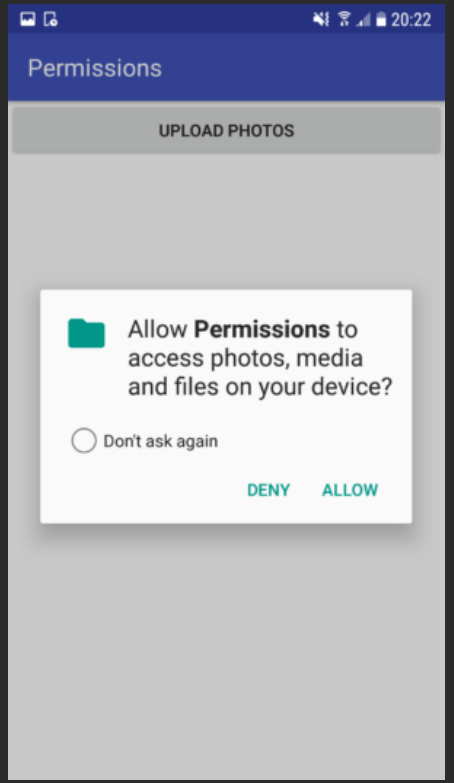
Before Android 6.0 (Marshmallow, API 23), **all** permissions were granted at installation.

Not user-friendly, leads to apps taking advantage of users who don't read the permissions or how they're used.

Read the fine print

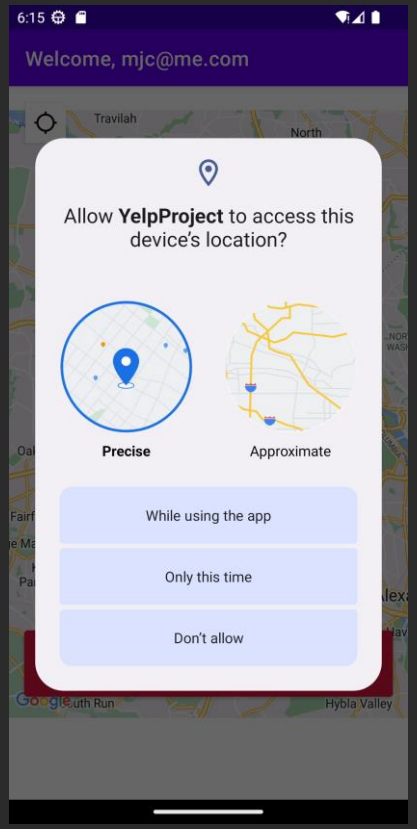
- “If you’ve read this far, then you are one of the very few Tin Leg customers to review all of their policy documentation," read the text, revealing a secret contest inviting Andrews to claim a $10,000 reward.
- Andrews was the first to reach out to Squaremouth about the contest, the company said, 23 hours after the event launched and after 73 policies spelling out the $10,000 reward had been issued.

Permissions



Android 6.0 (and higher) introduced *runtime permissions* - which the app **must** prompt the user to grant.

Permissions



The options for newer devices change to conditionally allowance:

- Accept (while in app)
- Accept (only this session)
- Deny
- Deny - Don't ask again (occurs if the user denies more than once).

Permissions

Permissions are divided into two groups.

Permissions

Permissions are divided into two groups.

- Normal - granted automatically at installation.

Permissions

Permissions are divided into two groups.

- Normal - granted automatically at installation.
  - INTERNET
  - BLUETOOTH
  - VIBRATE
  - SET_WALLPAPER
  - SET_ALARM
  - ...

Permissions

Permissions are divided into two groups.

- Normal - granted automatically at installation.
  - User will see these when they choose to install your app from Google Play.

Permissions

Permissions are divided into two groups.

- Normal - granted automatically at installation.
- Dangerous - not granted automatically, must prompt.

Permissions

Permissions are divided into two groups.

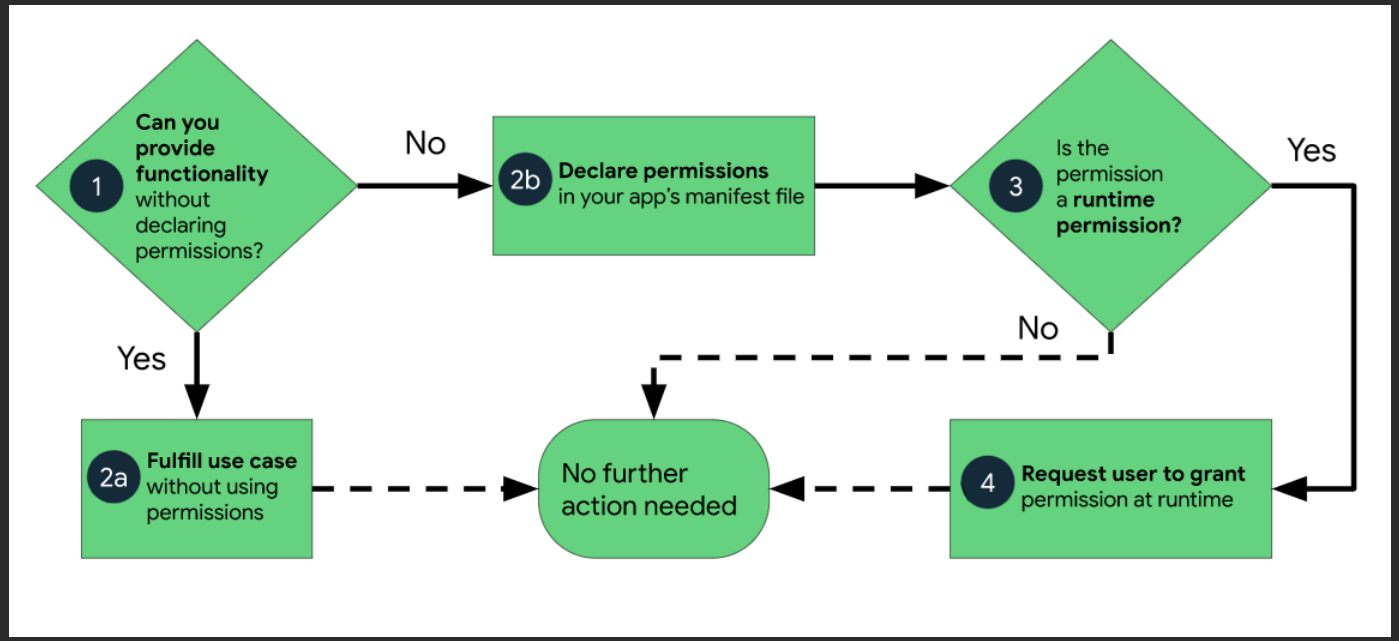
- Normal - granted automatically at installation.
- Dangerous - not granted automatically, must prompt.
  - CAMERA
  - RECORD_AUDIO
  - ACCESS_FINE_LOCATION / ACCESS_COARSE_LOCATION
  - SEND_SMS / READ_SMS / RECEIVE_SMS
  - WRITE_EXTERNAL_STORAGE

Permissions

Permissions are divided into two groups.

- Normal - granted automatically at installation.
- Dangerous - not granted automatically, must prompt.
  - Generally, permissions that can compromise the user's privacy are marked "dangerous."
  - The documentation lists the category for each permission.

Permissions



Dangerous Permissions

Your app has to handle:

- Requesting the permission (if device is >= Marshmallow)

Dangerous Permissions

Your app has to handle:

- Requesting the permission (if device is >= Marshmallow)
- Handling permission acceptance

Dangerous Permissions

Your app has to handle:

- Requesting the permission (if device is >= Marshmallow)
- Handling permission acceptance
- Handling permission denial
- Handling permission denial “forever” (Don’t Ask Again)

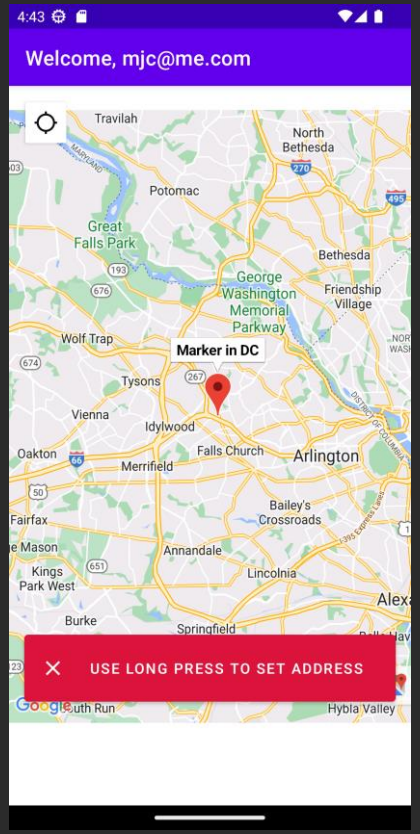
Dangerous Permissions

Your app has to handle:

- Requesting the permission (if device is >= Marshmallow)
  - Determining if an additional explanation for the permission is needed.

- Handling permission acceptance
- Handling permission denial
- Handling permission denial “forever” (Don’t Ask Again)

Requesting Permissions



- Let’s request the GPS permission (ACCESS_FINE_LOCATION) for the button on our maps screen.

Permissions

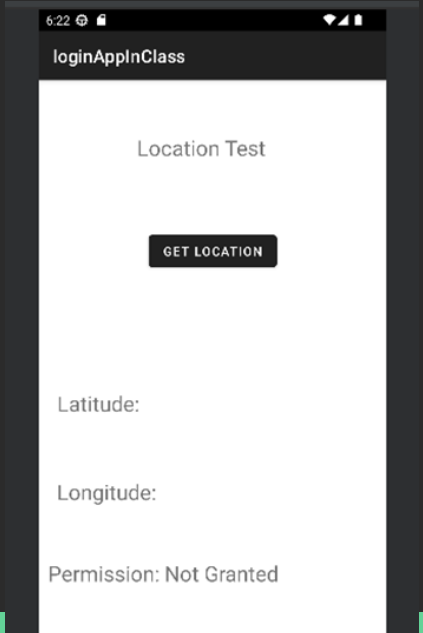
First, we need to declare the permission in the Manifest.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="edu.gwu.androidtweetsfall2021">

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

Requesting Permissions

When the user presses the “Current Location” button, we need to check if the app already has the location permission.

If not, we request it.



Requesting Permissions

We'll encapsulate this into two functions:

```
private fun checkLocationPermission() {
    // Determine if the user has the location permission
    // If not, we can ask for it
}

private fun useCurrentLocation() {
    // Called assuming we have the location permission granted
}
```

Requesting Permissions

At runtime, check if the permission has already been granted:

```
import android.Manifest
import android.content.pm.PackageManager

// In checkLocationPermission()

if (ContextCompat.checkSelfPermission(this, android.
Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED) {
   // Permission is already granted

} else {
   // Permission has not been granted

}
```

Make sure you import Android.Manifest

Requesting Permissions

Use ActivityCompat.requestPermissions()

```
ActivityCompat.requestPermissions(
    this,
    arrayOf(Manifest.permission.ACCESS_FINE_LOCATION),
    200
)
```

Requesting Permissions

Use ActivityCompat.requestPermissions()

Context (e.g. the Activity)

```
ActivityCompat.requestPermissions(
    this,
    arrayOf(Manifest.permission.ACCESS_FINE_LOCATION),
    200
)
```

Requesting Permissions

Use ActivityCompat.requestPermissions()

List of permissions to prompt

```
ActivityCompat.requestPermissions(
    this,
    arrayOf(Manifest.permission.ACCESS_FINE_LOCATION),
    200
)
```

Requesting Permissions

Use ActivityCompat.requestPermissions()

The “request code”. An arbitrary number that you can use to refer back to this request later.

```
ActivityCompat.requestPermissions(
    this,
    arrayOf(Manifest.permission.ACCESS_FINE_LOCATION),
    200
)
```

Requesting Permissions

```
if (ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION)
   == PackageManager.PERMISSION_GRANTED) {
   // Permission is already granted, we can check the location

} else {
   // Permission has not been granted, try to request the permission
   ActivityCompat.requestPermissions(
       this,
       arrayOf(Manifest.permission.ACCESS_FINE_LOCATION),
       200
   )
}
```

Receiving Permissions Result

The user will be presented the permissions dialog. When they pick an option, the OS will call your Activity's `onRequestPermissionsResult` method.

Receiving Permissions Result

```
override fun onRequestPermissionsResult(
  requestCode: Int,
  permissions: Array<String>,
  grantResults: IntArray
) {

}
```

Receiving Permissions Result

The request code used earlier to request permissions.

```
override fun onRequestPermissionsResult(
  requestCode: Int,
  permissions: Array<String>,
  grantResults: IntArray
) {

}
```

Receiving Permissions Result

Permissions that were requested

```
override fun onRequestPermissionsResult(
  requestCode: Int,
  permissions: Array<String>,
  grantResults: IntArray
) {

}
```

Receiving Permissions Result

Either PERMISSION_GRANTED or PERMISSION_DENIED for each of the permissions

```
override fun onRequestPermissionsResult(
  requestCode: Int,
  permissions: Array<String>,
  grantResults: IntArray
) {

}
```

Receiving Permissions Result

```
// Inside onRequestPermissionsResult

if (requestCode == 200) {

}
```

Receiving Permissions Result

```
if (requestCode == 200) {
    // We only requested one permission, so its result is the first element
    if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {
        // User granted the permission :)

    } else {
        // User denied the permission :(

    }
}
```

Receiving Permissions Result

In many cases, this is all the code needed and either the user accepts the permission or not (and either you move forward with the requested action for now).

Permissions -
Extra Denial Handling

Receiving Permissions Result

In many cases, this is all the code needed and either the user accepts the permission or not (and either you move forward with the requested action for now).

But, if needed, there are some extra ways you can handle permission denial.

Handling Denial

If you received PERMISSION_DENIED, there are 3 scenarios:

Handling Denial

If you received PERMISSION_DENIED, there are 3 scenarios:

- The user clicked Deny

Handling Denial

If you received PERMISSION_DENIED, there are 3 scenarios:

- The user clicked Deny
- The user clicked Deny and checked Don’t ask again

Handling Denial

If you received PERMISSION_DENIED, there are 3 scenarios:

- The user clicked Deny
- The user clicked Deny and checked Don’t ask again
- The user *previously* checked Don’t ask again and denied (e.g. automatic denial)

Handling Denial

If you received PERMISSION_DENIED, there are 3 scenarios:

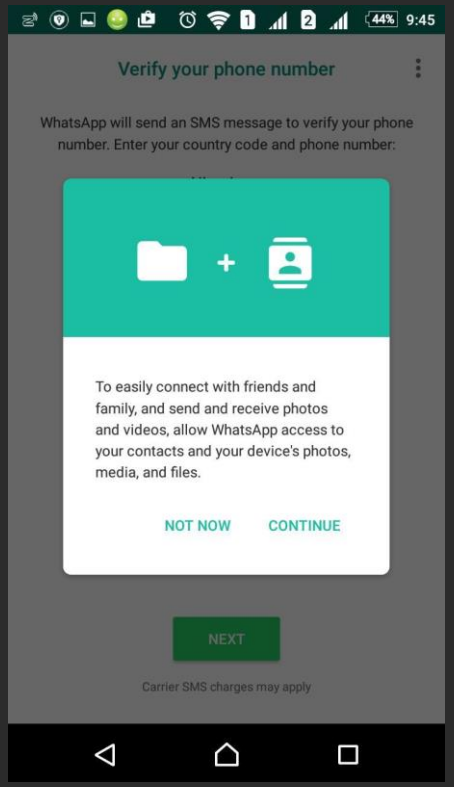
- The user clicked Deny
- The user clicked Deny and checked Don’t ask again
- The user *previously* checked Don’t ask again and denied (e.g. automatic denial)

Handling Denial

If you received PERMISSION_DENIED, there are 3 scenarios:

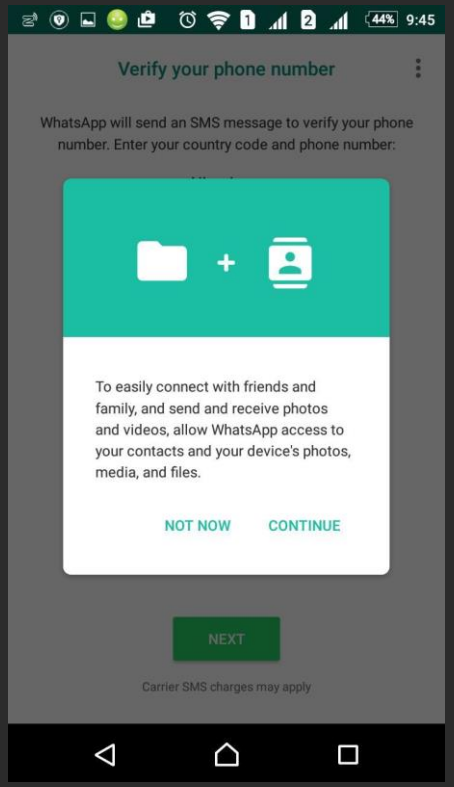
- The user clicked Deny
  - Maybe the user is confused why the permission is needed? We could try providing a better explanation next time we need to prompt them.
- The user clicked Deny and checked Don’t ask again
- The user *previously* checked Don’t ask again and denied (e.g. automatic denial)

Handling Denial



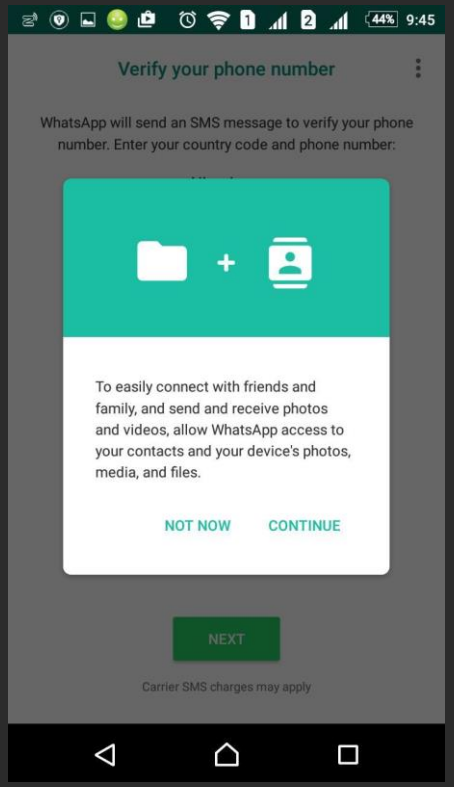
Generally, it’s a good practice to show a “rationale” or justification for requesting the permission - or, at least, after the user has denied once.

Handling Denial



Android provides a function **shouldShowRequestPermissionRationale** which returns a boolean indicating whether or not you can show a rationale.

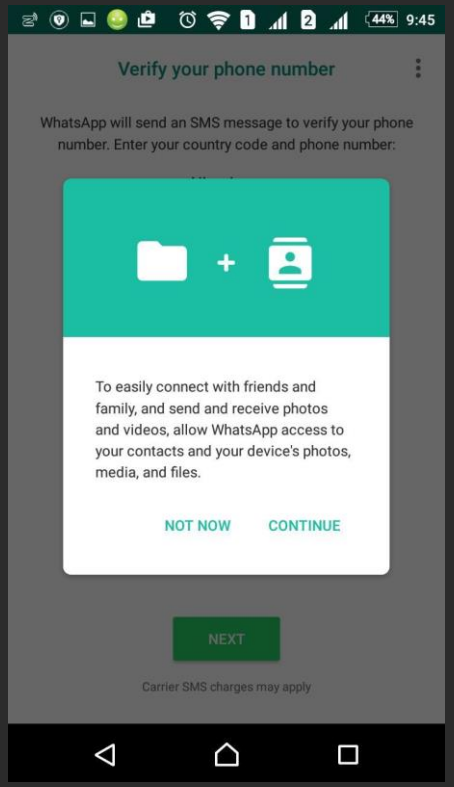
Handling Denial



shouldShowRequestPermissionRationale

"This method returns true if the app has requested this permission previously and the user denied the request."

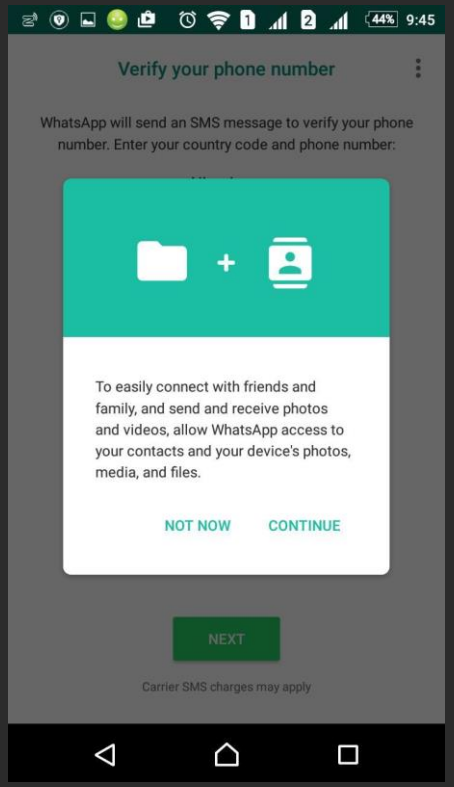
Handling Denial



shouldShowRequestPermissionRationale

“Note: If the user turned down the permission request in the past and chose the Don't ask again option in the permission request system dialog, this method returns false.”

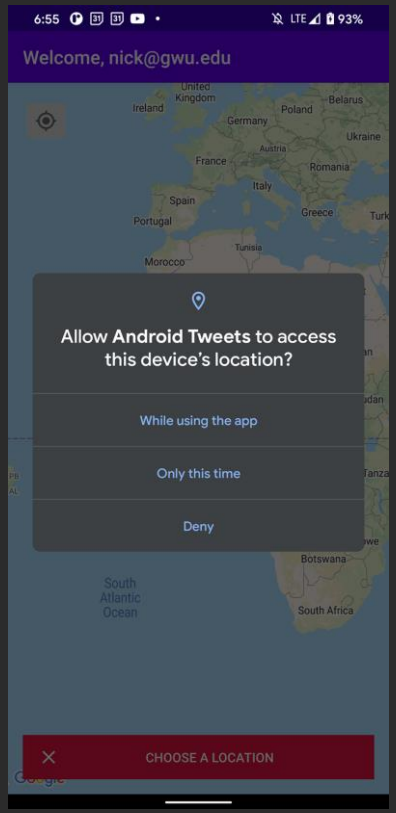
Handling Denial



shouldShowRequestPermissionRationale

- True - if the user has denied, but hasn't checked Don't show again.
- False - user has denied and checked Don't show again.
  - *Or*, they haven't yet been prompted at all.

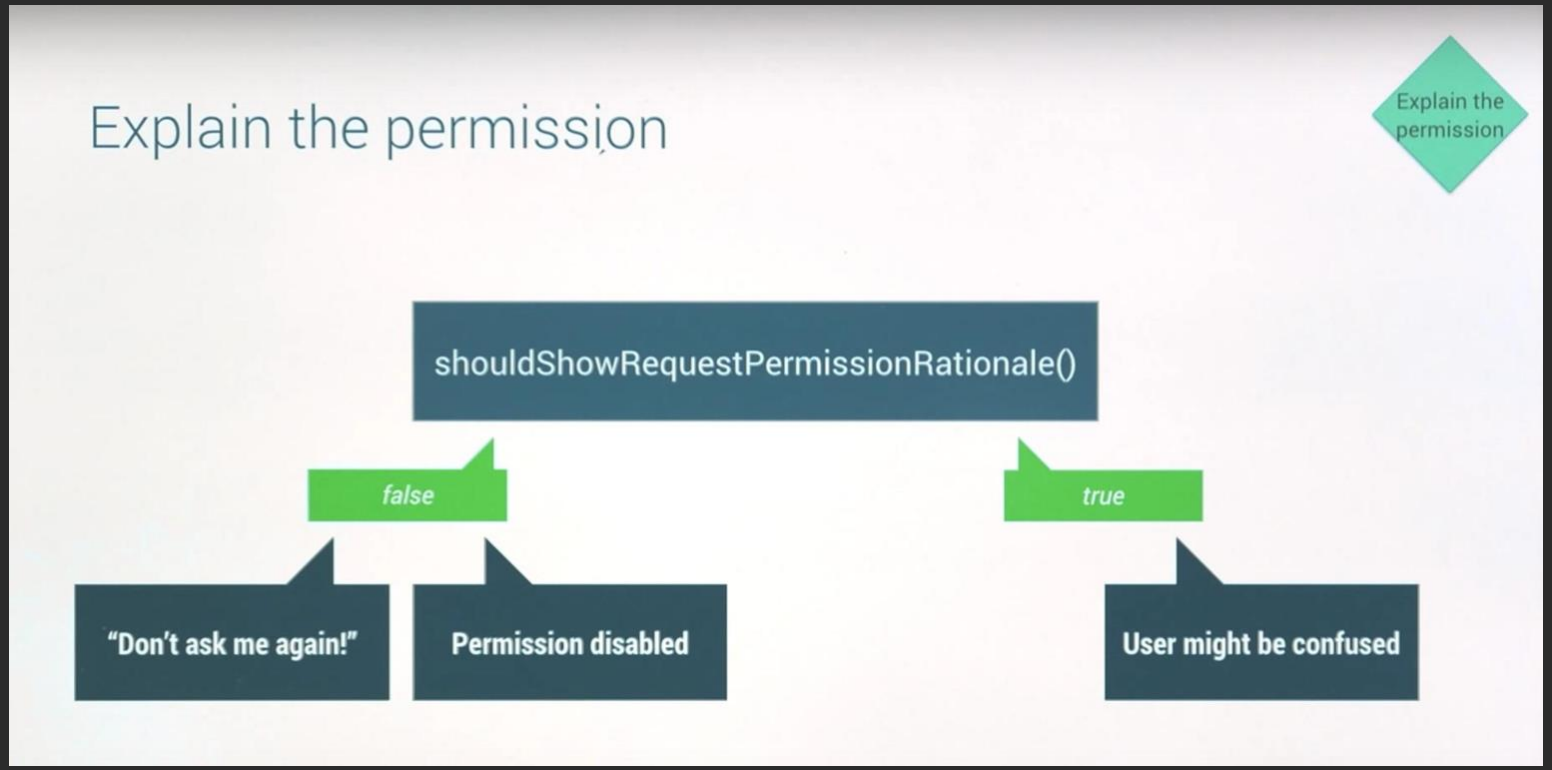
Handling Denial



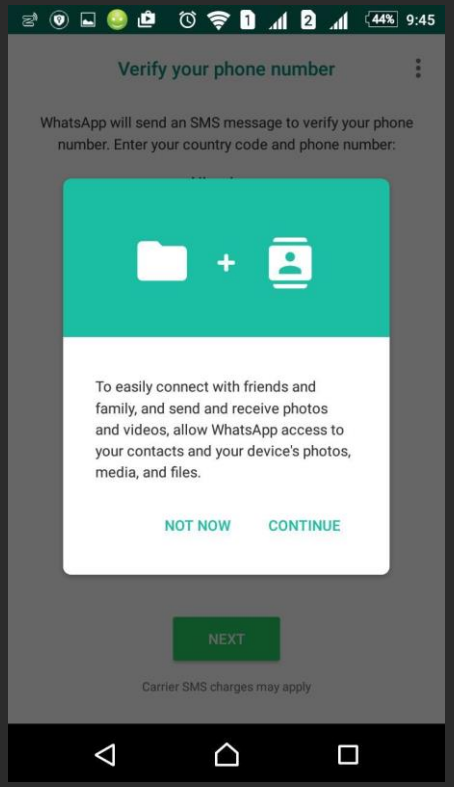
shouldShowRequestPermissionRationale

- On newer OS versions, it works similarly, but when “Only this time” option is selected, it’s as if the user is being prompted for the 1st time on each launch.

Handling Denial



Handling Denial



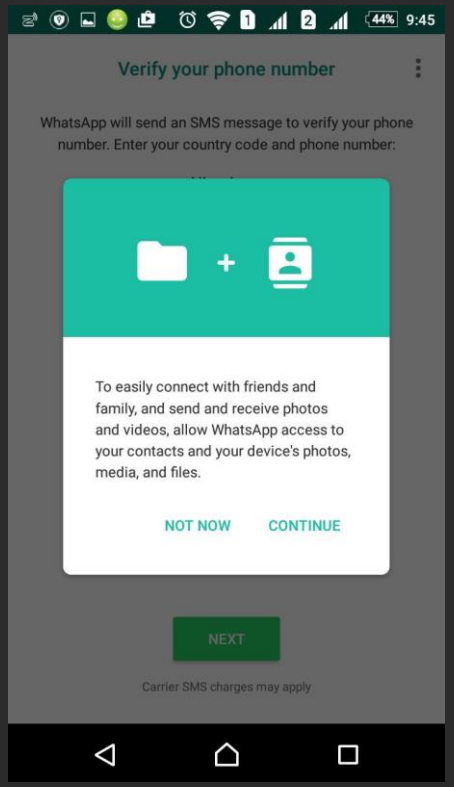
shouldShowRequestPermissionRationale

- You can use this *before* prompting the permission to determine if you should show some helpful message to convince the user.

Receiving Permissions Result

```
if (shouldShowRequestPermissionRationale(Manifest.permission.ACCESS_FINE_LOCATION)) {
   // Show something like an AlertDialog before requesting the permission

} else {
   // No additional justification needed, can try requesting the permission

}
```

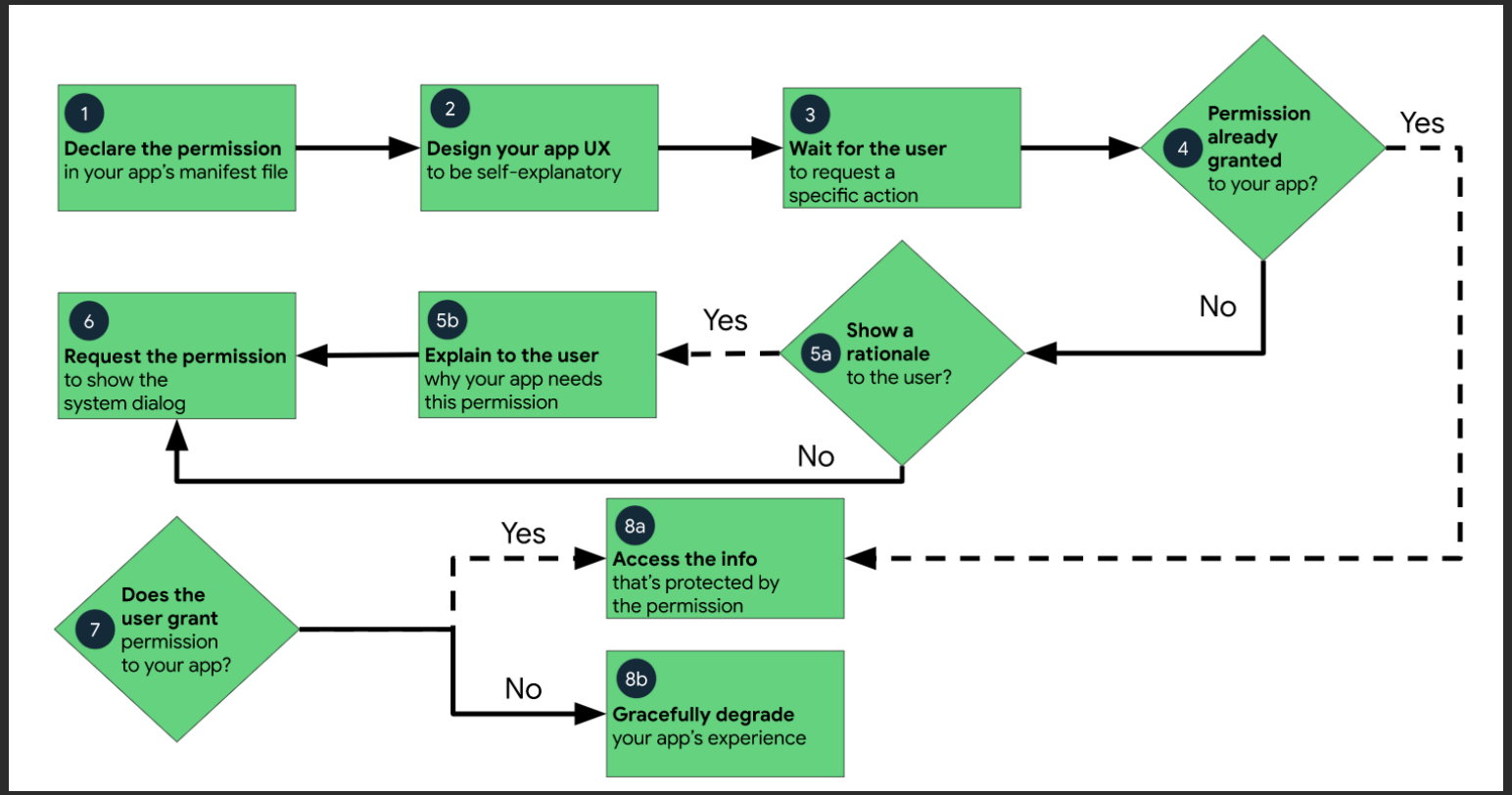
Handling Denial



shouldShowRequestPermissionRationale

- Also a good use case for a Dialog (as pictured in the left screenshot), rather than a full dedicated screen.
  - See the “extra” lecture on Dialogs under Lecture 4 on Blackboard.

Handling Denial



Handling Denial

If you received PERMISSION_DENIED, there are 3 scenarios:

- The user clicked Deny
- The user clicked Deny and checked Don’t ask again
- The user *previously* checked Don’t ask again and the system gave you an *automatic* denial

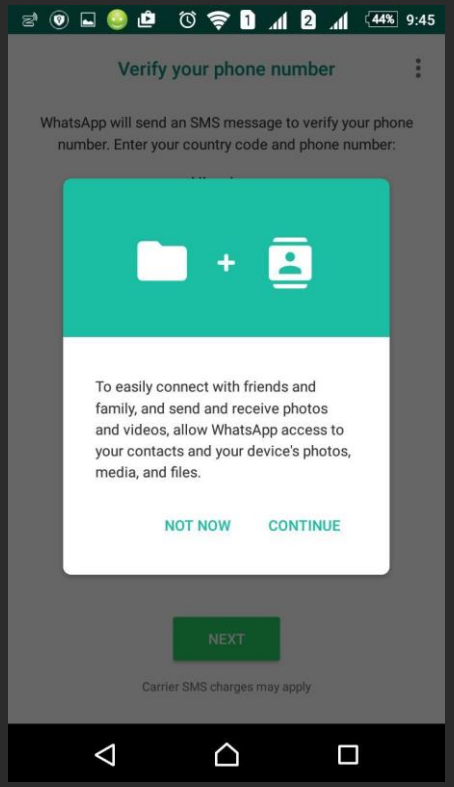
Handling Denial

If you received PERMISSION_DENIED, there are 3 scenarios:

- The user clicked Deny
- The user clicked Deny and checked Don’t ask again
- The user *previously* checked Don’t ask again and the system gave you an *automatic* denial

Generally, means you really shouldn’t ask again, but in case you need to…

Handling Denial



shouldShowRequestPermissionRationale

- Additionally, if this returns True, you can prompt the user again.
- So, you can **also** use this after receiving a permission denial to know if they denied it permanently (returns False).

Receiving Permissions Result

```
if (requestCode == 200) {
    // We only requested one permission, so its result is the first element
    if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {
        // User granted the permission :)

    } else {
        // User denied the permission :(

    }
}
```

Receiving Permissions Result

```
// In the else branch of handling the result...


if (shouldShowRequestPermissionRationale(
                Manifest.permission.ACCESS_FINE_LOCATION)) {
   // Do nothing, user declined, but can be prompted again later
   // (e.g. a regular decline)

} else {
   // User has denied & don't show again
   // (e.g. either this time, or was an automatic denial)
}
```

Receiving Permissions Result

If the user has permanently denied a permission, the only way to reverse that decision is for the user to do so in their Settings app.

Receiving Permissions Result

```
// In the else branch of handling the result...


if (shouldShowRequestPermissionRationale(
                    Manifest.permission.ACCESS_FINE_LOCATION)) {
   // Do nothing, user declined, but can be prompted again later

} else {
   // User has denied & don't show again
   Toast.makeText(
      this,
      "To use this feature, go into your Settings and enable the Location permission",
      Toast.LENGTH_LONG
   ).show()
}
```

Opening your app settings

If the user has permanently denied a permission, the only way to reverse that decision is for the user to do so in their Settings app.

We can make this easy using an Intent!

Opening your app settings

```
val myAppSettings = Intent(
    Settings.ACTION_APPLICATION_DETAILS_SETTINGS,
    Uri.parse("package:$packageName")
)

myAppSettings.addCategory(Intent.CATEGORY_DEFAULT)
myAppSettings.flags = Intent.FLAG_ACTIVITY_NEW_TASK
startActivity(myAppSettings)
```

Opening your app settings

If you wanted to auto-recheck when the user returns to the app, one way would be to override one of the lifecycle functions like onResume.

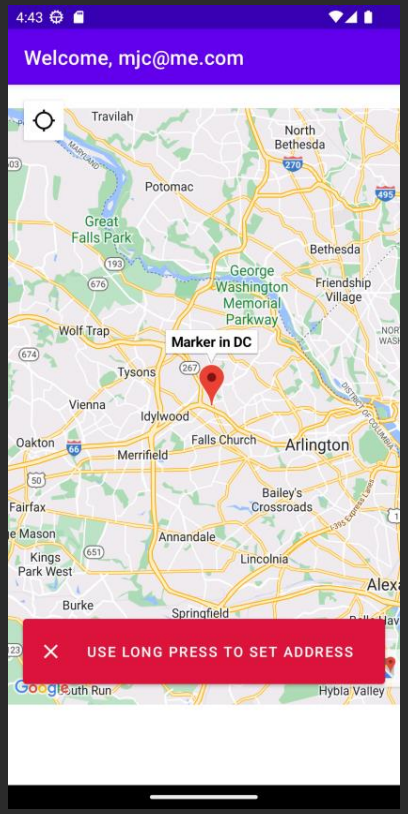
Opening your app settings

If you wanted to auto-recheck when the user returns to the app, one way would be to override one of the lifecycle functions like onResume.

There's another function called startActivityForResult that's normally used when you want callback when the user comes back from another Activity you launched, but needs some extra steps for it to work with the Settings menu...

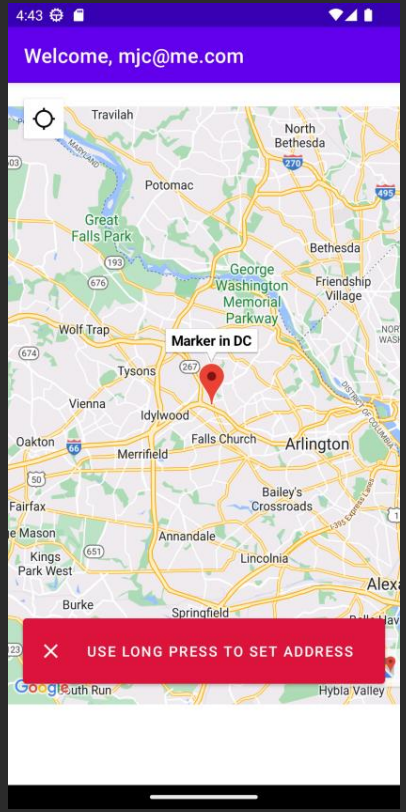
Location

Determining Location



Assuming the user granted the location permission, we can now access location information in our app!

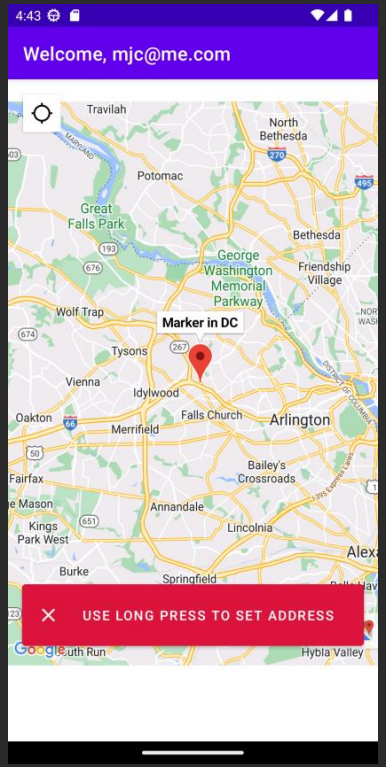
Determining Location



Two ways:

- ~~Framework Location APIs~~
- Google Play Services Location APIs

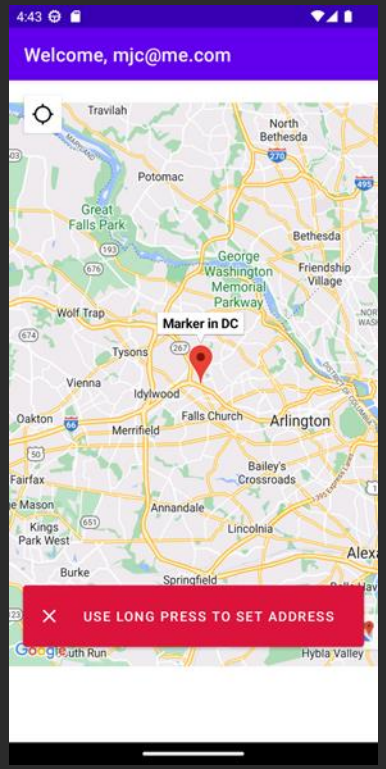
Determining Location



Two ways:

- Framework Location APIs
  - The “old” way. Subscribe to GPS or Network location updates and unsubscribe when you’re done.
- Google Play Services Location APIs

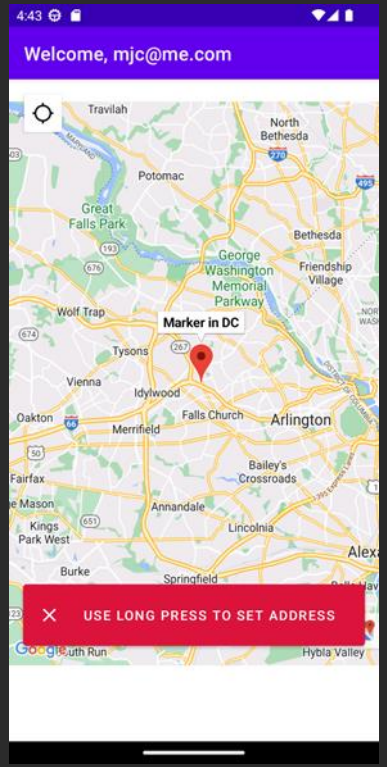
Determining Location



Two ways:

- Framework Location APIs
  - The “old” way. Subscribe to GPS or Network location updates and unsubscribe when you’re done.
  - But, it’s simple to use.
- Google Play Services Location APIs

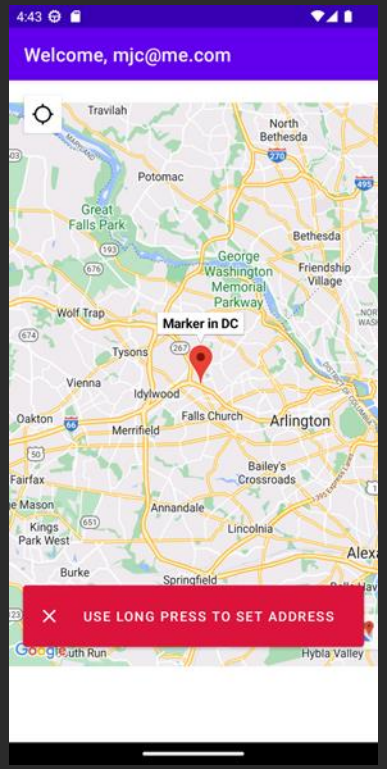
Determining Location



Two ways:

- Framework Location APIs
- Google Play Services Location APIs
  - The new way. Works similar to the Framework APIs, but adds more benefits.

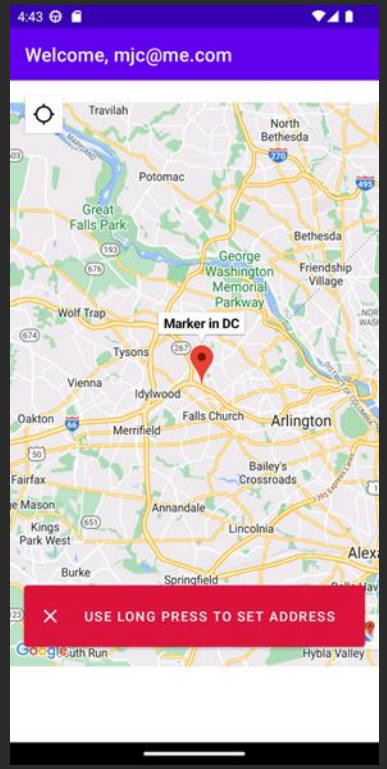
Determining Location



Two ways:

- Framework Location APIs
- Google Play Services Location APIs
  - The new way. Works similar to the Framework APIs, but adds more benefits:
    - Improved location accuracy (Google is smart).
    - Improved battery life management.

Determining Location



Two ways:

- Framework Location APIs
- Google Play Services Location APIs
  - The new way. Works similar to the Framework APIs, but adds more benefits.
  - More efficient, but slightly harder to use. Requires Google Play dependency, but no API key needed.

Play Services Location API

First, add the dependency to your app/build.gradle.

```
dependencies {
   // ...
  implementation(“com.google.android.gms:play-services-location:21.2.0”)
}
```

Autocompletes as you type….

Play Services Location API

Create a Location Provider in your class.

- The “FusedLocationProvider” manages the underlying technology (GPS, Wi-Fi, cellular) to determine your location and you can prioritize accuracy vs. battery life.

Play Services Location API

Create a Location Provider in your class.

```
// Requires a Context, so we can't initialize it until the
// Activity has been initialized.
private lateinit var locationProvider: FusedLocationProviderClient

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

   locationProvider = LocationServices.getFusedLocationProviderClient(this)
```

Play Services Location API

You can request the *last known location* from the location provider. It’s resolved asynchronously, so you need a callback.

```
// After having our permission granted...

locationProvider.lastLocation.addOnSuccessListener { location ->

}
```

Play Services Location API

You can request the *last known location* from the location provider. It's resolved asynchronously, so you need a callback.

```
locationProvider.lastLocation.addOnSuccessListener { location ->
  // Documentation states the location can be null in rare cases
  if (location != null) {
      val latLng = LatLng(location.latitude, location.longitude)

  }
}
```

Play Services Location API

You can request the *last known location* from the location provider. It's resolved asynchronously, so you need a callback.

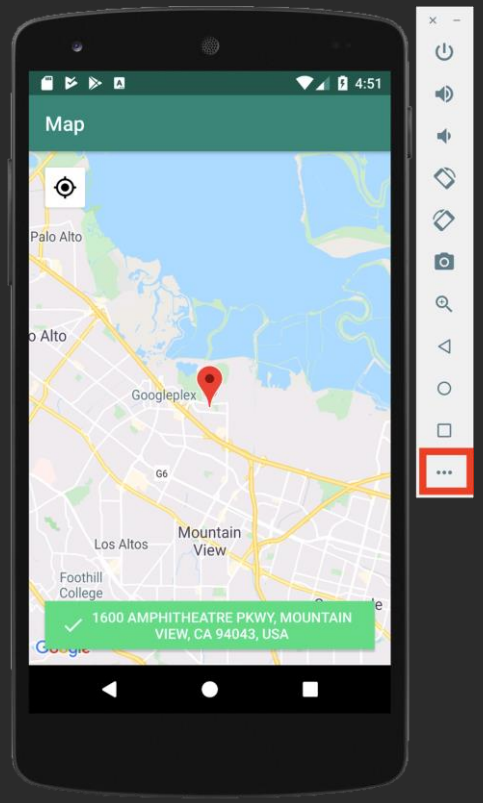
```
locationProvider.lastLocation.addOnSuccessListener { location ->
  // Documentation states the location can be null in rare cases
  if (location != null) {
    val latLng = LatLng(location.latitude, location.longitude)

    // Just like when there's a long-press on the Map, geocode the coordinates, place a marker, etc.
    doGeocoding(latLng)
  }
}
```

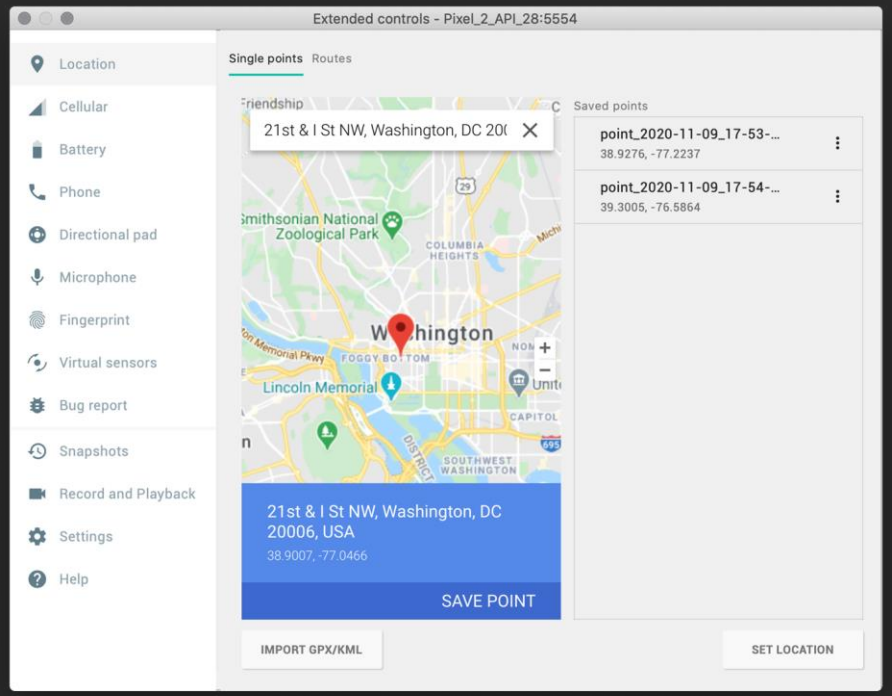
Play Services Location API

Our geocoding logic now needs to be referenced in more than one place - we can move it into a function.

```
private fun doGeocoding(coords: LatLng) {
   doAsync {
       val geocoder = Geocoder(this@MapsActivity)
       // ...
```

Testing Location on Emulator





Play Services Location API

The last known location is good if you don’t have an accuracy requirement and want something quick, but may not be appropriate for all use cases.

Play Services Location API

The last known location is good if you don’t have an accuracy requirement and want something quick, but may not be appropriate for all use cases.

If you need an up-to-date location, you can request a fresh (and constant) location updates.

Play Services Location API

Request fresh, constant location updates

```
locationProvider.requestLocationUpdates(
  LocationRequest.create(),
  locationCallback,
  null
)
```

Play Services Location API

Request fresh, constant location updates

```
locationProvider.requestLocationUpdates(
  LocationRequest.create(),
  locationCallback,
  null
)
```

The location request parameters. Using the default here, but can be customized (e.g. frequency of location updates, what level of accuracy, etc.)

Play Services Location API

Request fresh, constant location updates

We'll create this next. Receives a callback when the user's location is updated.

```
locationProvider.requestLocationUpdates(
  LocationRequest.create(),
  locationCallback,
  null
)
```

Play Services Location API

Request fresh, constant location updates

The (optional) thread for the callbacks to be received on.

```
locationProvider.requestLocationUpdates(
  LocationRequest.create(),
  locationCallback,
  null
)
```

Play Services Location API

If you did want constant location updates, you can specify an interval and priority:

```
// Refresh every second
val locationRequest = LocationRequest.create()
locationRequest.interval = 1000
locationRequest.priority = PRIORITY_HIGH_ACCURACY

locationProvider.requestLocationUpdates(
   locationRequest,
   locationCallback,
   null
)
```

Play Services Location API

Create a LocationCallback instance to receive location updates.

```
private val locationCallback = object : LocationCallback() {

    override fun onLocationResult(result: LocationResult) {
        // Do something with the LocationResult
    }

}
```

Play Services Location API

Docs for LocationResult:
https://developers.google.com/android/reference/com/google/android/gms/location/LocationResult#getLocations()

```
override fun onLocationResult(result: LocationResult) {
    // We only need one result, so stop listening for updates
    // Otherwise, this function would be called frequently
    // with new updates.
    locationProvider.removeLocationUpdates(this)

    // Most recent location is now the last one
    val location = result.lastLocation
```

Play Services Location API

```
override fun onLocationResult(result: LocationResult) {

    // We only need one result, so stop listening for updates
    // ...
    locationProvider.removeLocationUpdates(this)

    // Most recent location is now the last one
    val location = result.lastLocation
    val latLng = LatLng(location.latitude, location.longitude)

    // Do an Address lookup on the current location
    doGeocoding(latLng)
}
```

Play Services Location API

If you do this, you'll want to be smarter about when you call removeLocationUpdates (e.g. when the user leaves the screen)

```
override fun onLocationResult(result: LocationResult) {

   // You'd no longer unregister here (but instead when the user leaves the screen
   // e.g. in onDestroy or onPause)
   // locationProvider.removeLocationUpdates(this)

   // ...
}
```

Questions?

Helpful Links

- Permissions Overview
- Requesting Permissions
- Location - Framework APIs (Old)
- Location - Google Play Services APIs (New)
- Location - Last Known