CSCI 4237

Software Design for Handheld Devices

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Lecture 12 - Screen Rotation & Notifications

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) 8am-6pm

Last Time

- Permissions
- Location

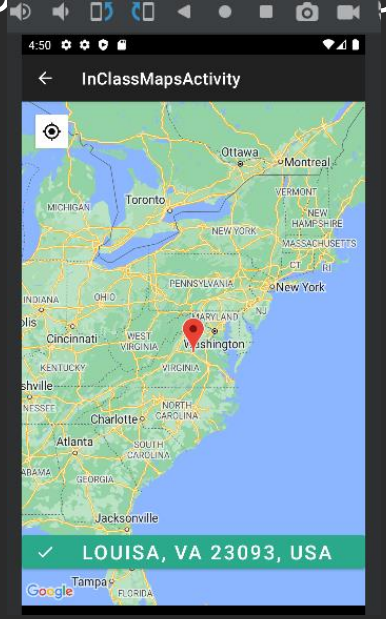
Today

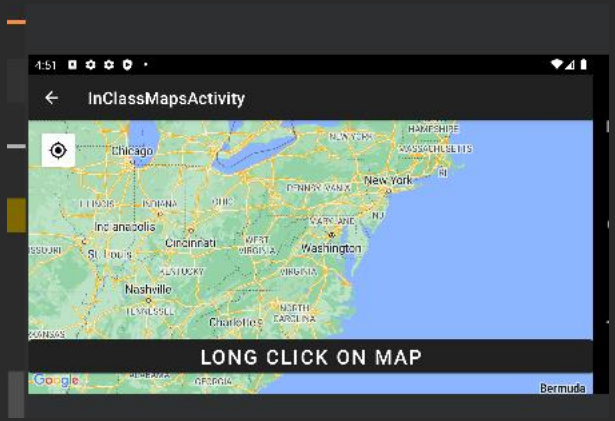
- Screen Rotation
- Notifications

Screen Rotation

Motivation

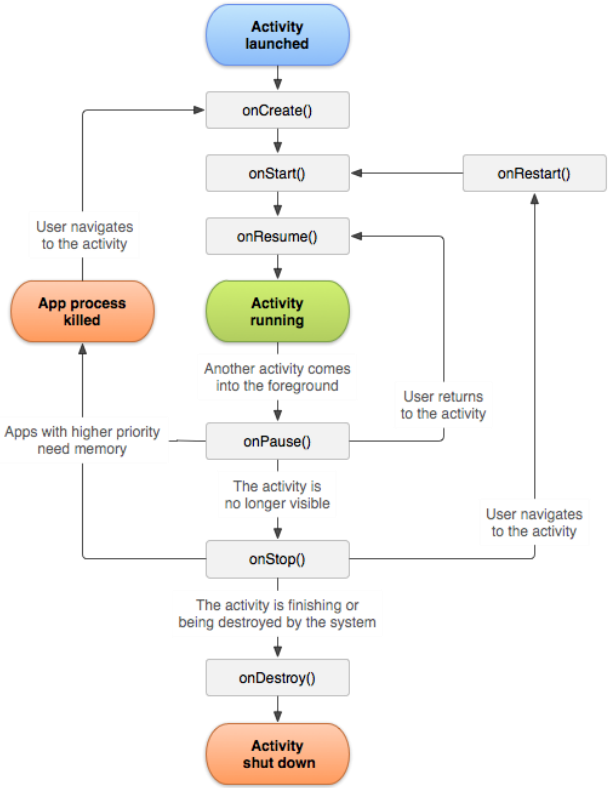
If you select a location on our Maps screen and rotate the screen - some information is lost...





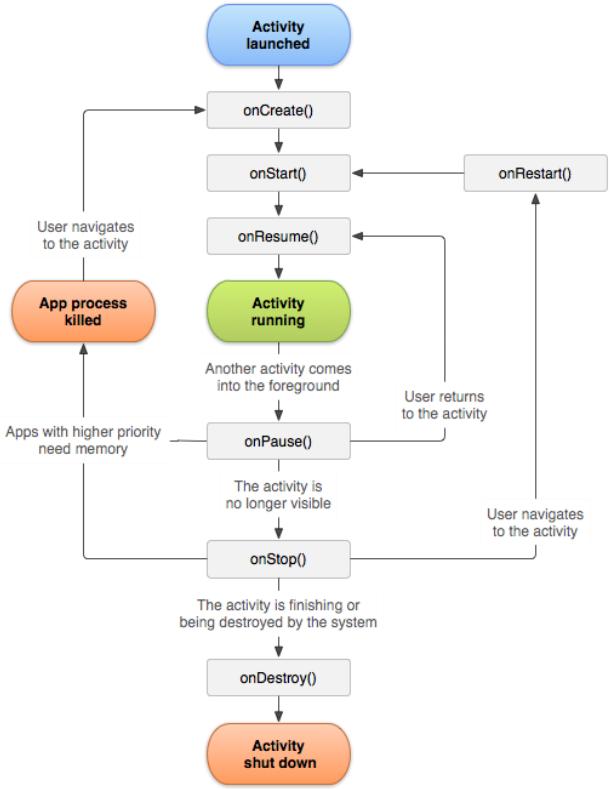
Recall: The Activity Lifecycle

- Your Activity goes through a series of events as the user interacts with your app, known as the Activity Lifecycle.



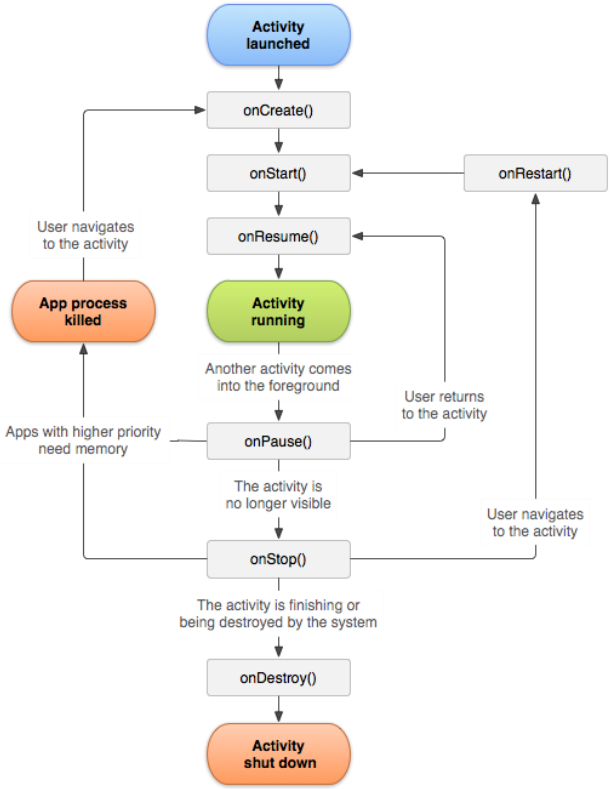
Recall: The Activity Lifecycle

- When you rotate the screen:



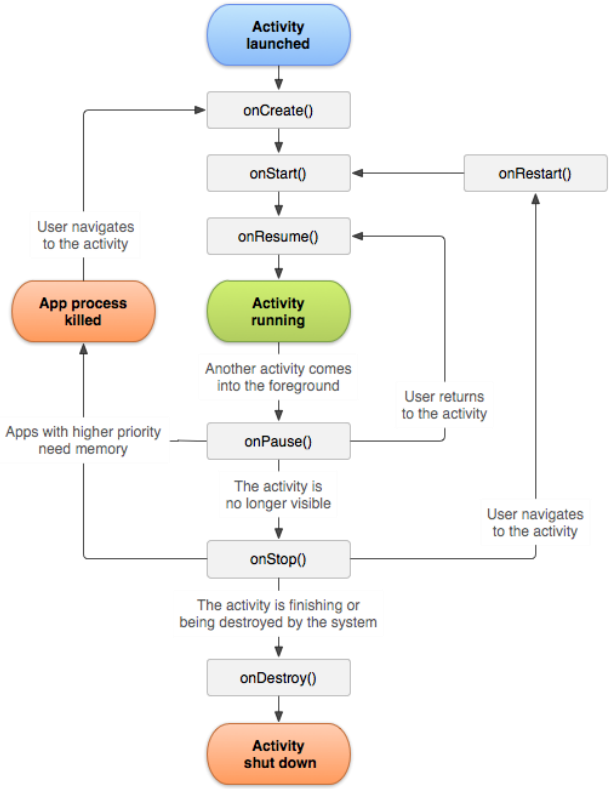
Recall: The Activity Lifecycle

- When you rotate the screen:
  - Android **recreates** your Activity.



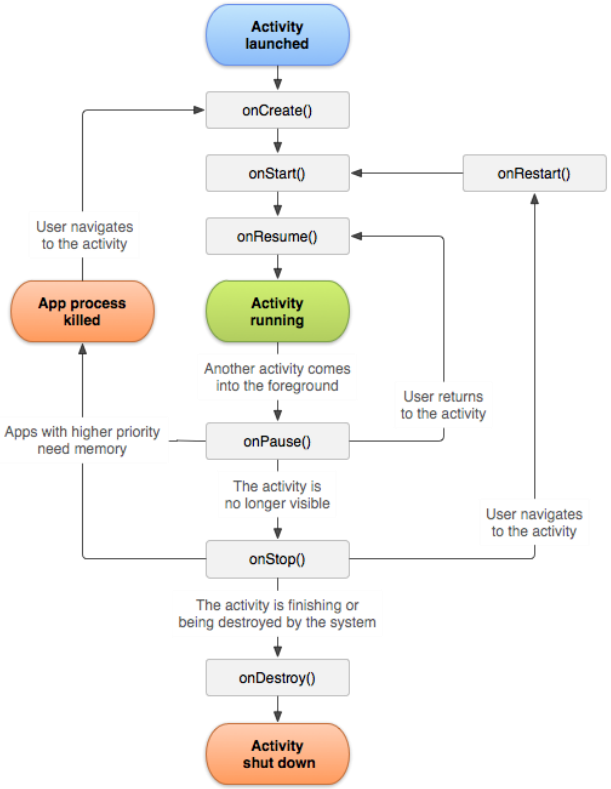
Recall: The Activity Lifecycle

- When you rotate the screen:
  - Android **recreates** your Activity.
  - Previous state is **erased** (e.g. new instance).



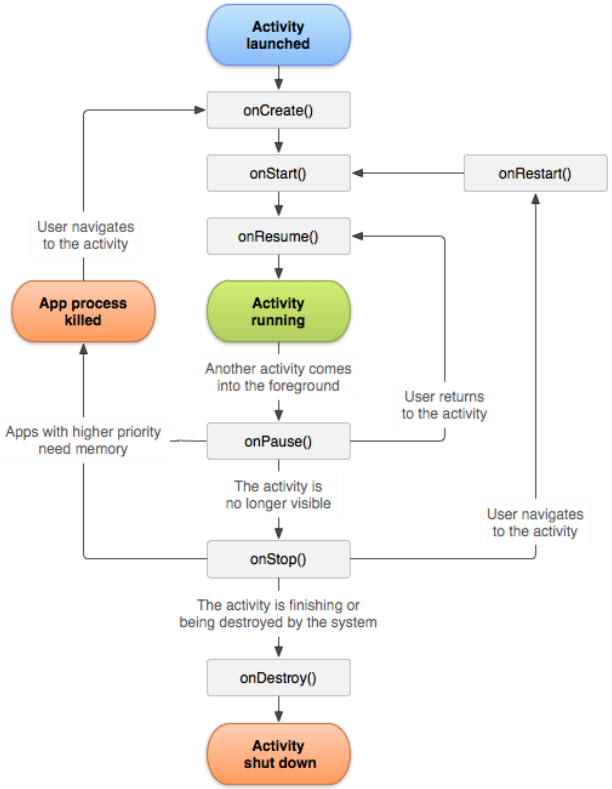
Recall: The Activity Lifecycle

- When you rotate the screen:
  - Android **recreates** your Activity.
  - Previous state is **erased** (e.g. new instance).
  - onCreate is called again, etc.



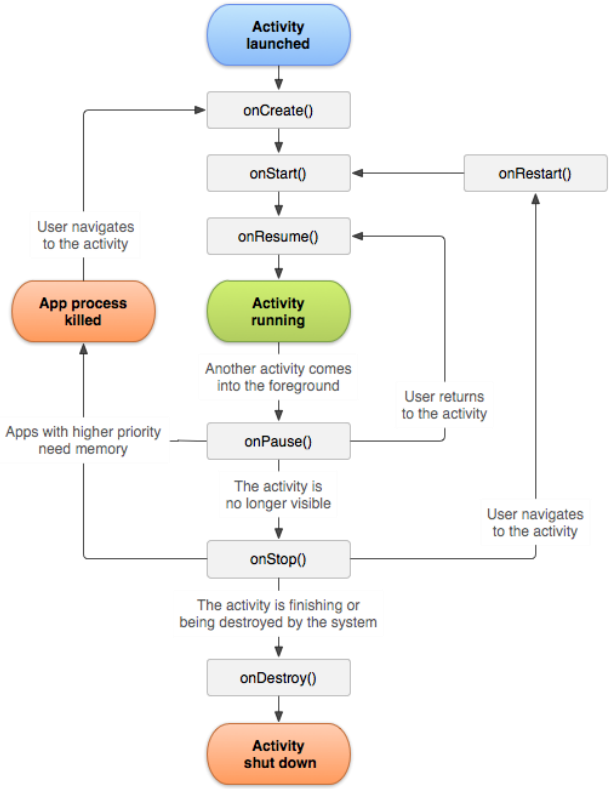
Recall: The Activity Lifecycle

- When you rotate the screen:
  - Android **recreates** your Activity.
  - Previous state is **erased** (e.g. new instance).
  - onCreate is called again, etc.
- More generally, this occurs on “configuration changes”



Recall: The Activity Lifecycle

- When you rotate the screen:
  - Android **recreates** your Activity.
  - Previous state is **erased** (e.g. new instance).
  - onCreate is called again, etc.
- More generally, this occurs on “configuration changes”
  - Change language
  - Revoke permissions via Settings
  - etc.



Handling Configuration Change (e.g. Rotation)

Android provides a way for an Activity to save data before a configuration change and restore it afterwards.

onSaveInstanceState

- Before restarting, Android calls **onSaveInstanceState** and passes it a **Bundle**.

- Lets see if this works by just adding a log statement to it
- We will also test out the rotation.

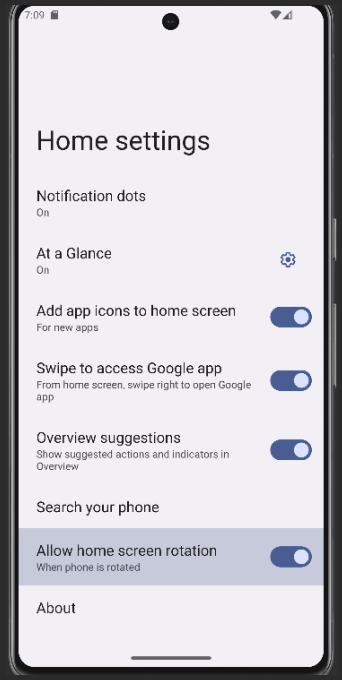
onSaveInstanceState

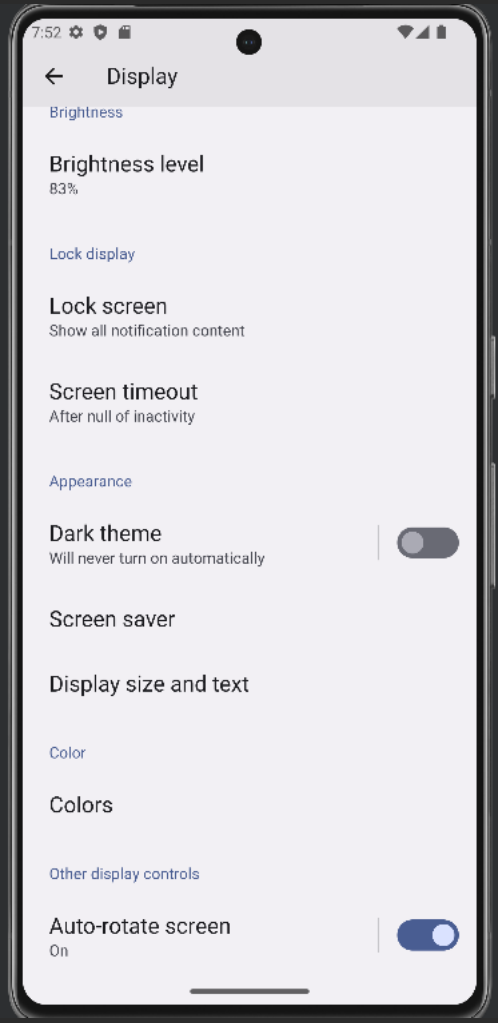
- Before restarting, Android calls **onSaveInstanceState** and passes it a **Bundle**.
- A **Bundle** acts like a Map (like an Intent) for you to put data in.

onSaveInstanceState

- Before restarting, Android calls **onSaveInstanceState** and passes it a **Bundle**.
- A **Bundle** acts like a Map (like an Intent) for you to put data in.
- The OS **persists** the Bundle. When your Activity restarts, it passes it as a param to **onCreate**.

Need to allow rotations in Phone





What do we need to do?

- Cleanup some of the map logic.
- Declare a default value for lat, long
- Setup the onSaveInstanceState funciton to save the current location
- Reload the location data if the savedInstanceState is any other than null

Persisting our Map

Let’s persist our lat,long on our MapsActivity so that we don’t lose the user’s long click/location.

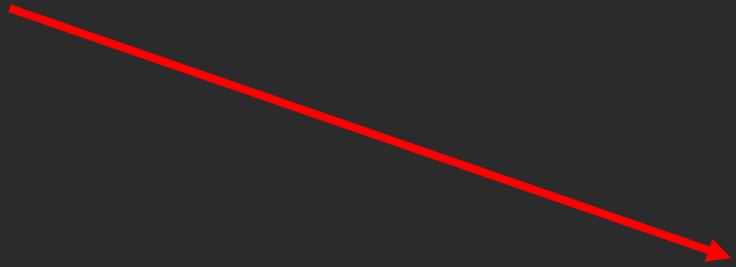
```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    Log.d(“Map”, “calling the onSaveInstanceState”)
   //save the lat/long
}
```

Persisting our List<YelpListing>

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)

    // We can put stuff into the Bundle, similar to an Intent
    outState.putInt("SOME_INT", 5)
    outState.putBoolean("SOME_BOOL", true)
    outState.putDouble(“latitude”, currentLatitude)

}
```

Restoring State

Recognize this?



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

Restoring State

After restarting your Activity, Android will pass the saved Bundle to onCreate(). When an Activity is launched for the first time, it will be null.

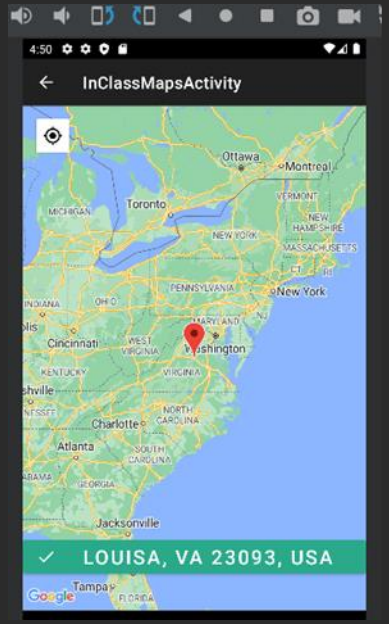
```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
```

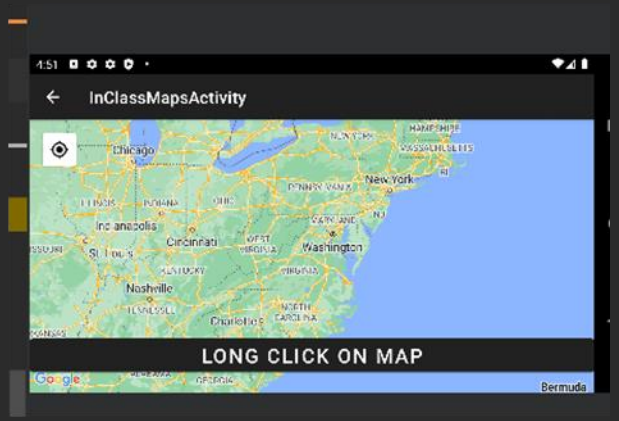
Restoring State

```
if (savedInstanceState != null) {
if (savedInstanceState!=null){
            Log.d("Map","savedInstance is not null")
            val latitude=savedInstanceState.getDouble("Latitude")
            val longitude=savedInstanceState.getDouble("Longitude")

            val latLng=LatLng(latitude,longitude)
            Log.d("Map","loading , $latitude, $longitude")

            doGeoCoding(latLng)

        }
}
```

Motivation

If you select a location on our Maps screen and rotate the screen - some information is lost...





What is the order of events once a rotation is initiated?

- onSaveInstanceState (before the activity is destroyed and recreated)
- onCreate method runs....

What do we need to do here?

5 minutes with your team to discuss (No Code). Conceptually think thru this and brief the class on what you came up with.

Lock the orientation of the Screen?

It’s also a difficult problem to solve - you technically can lock your app to only one screen orientation.

- It doesn’t *completely* absolve you... as screen rotation is not the only type of configuration change (but arguably it’s the most common).

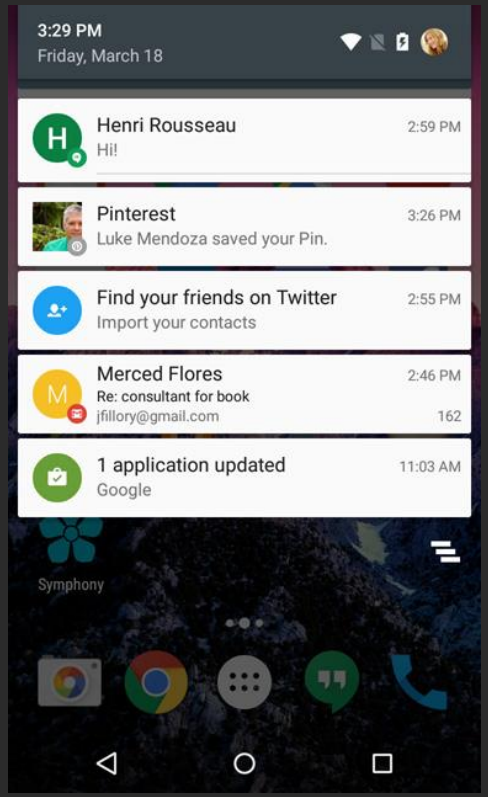
Low memory?

Android also has conditions (low memory) where it will choose to kill your app if it in the background and recreate it if the user navigates back.

i.e. apps at scale will most likely need to deal with configuration changes sooner or later.

Notifications

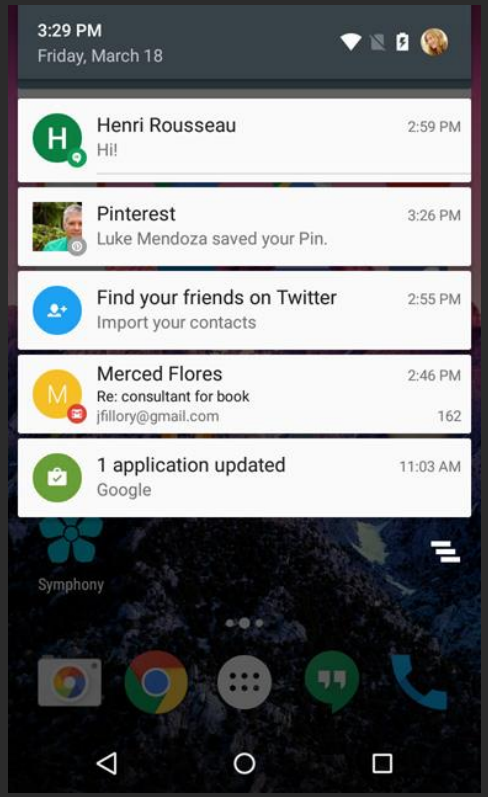
Notifications



Notifications are great for keeping users engaged with your app.

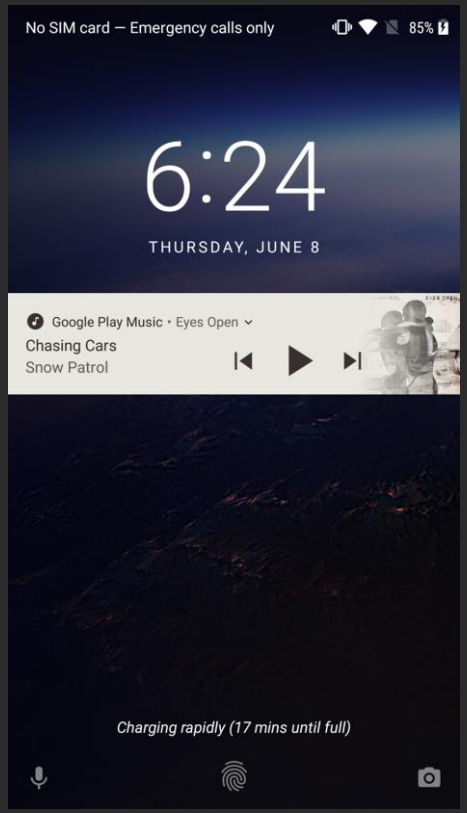
- Text / Email Notifications
- Inactivity Notifications
- Background Results
- Offers / New Features / etc.

Notifications



Broadly, we can think of notification *content* coming from two places:

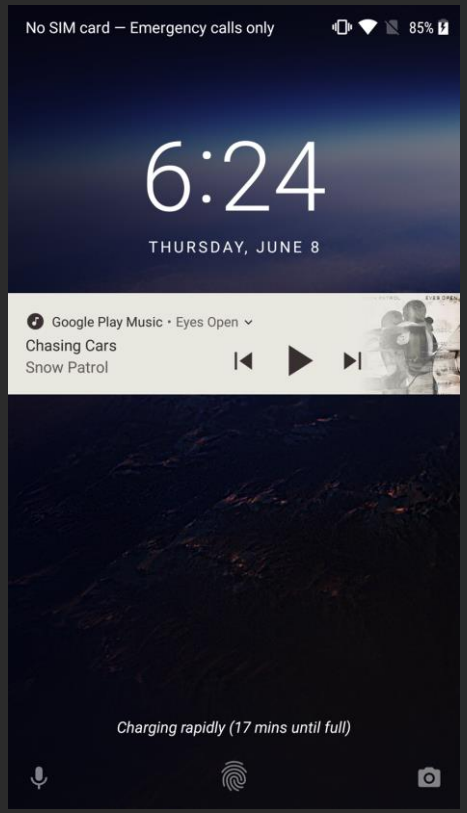
Notifications



Broadly, we can think of notification *content* coming from two places:

- Local - generated from the Android app itself.

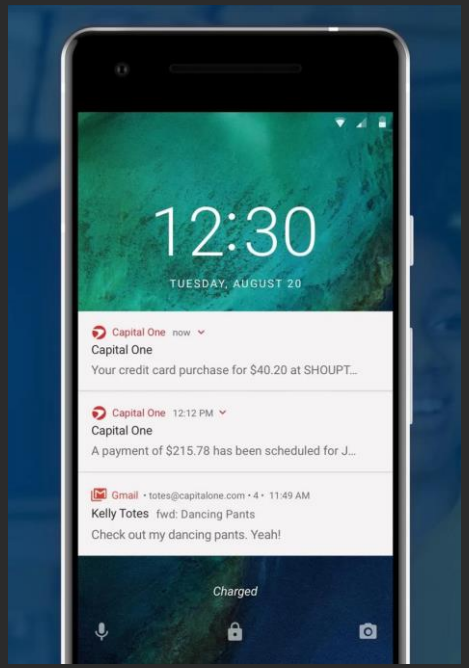
Notifications



Broadly, we can think of notification *content* coming from two places:

- Local - generated from the Android app itself.
  - Text / Email
  - Media Controls
  - Scheduled Notifications

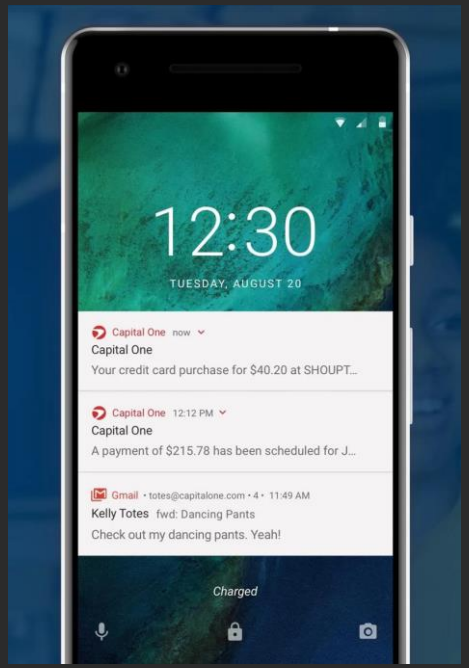
Notifications



Broadly, we can think of notification *content* coming from two places:

- Local - generated from the Android app itself.
- Remote - generated external from the app and pushed to the device.

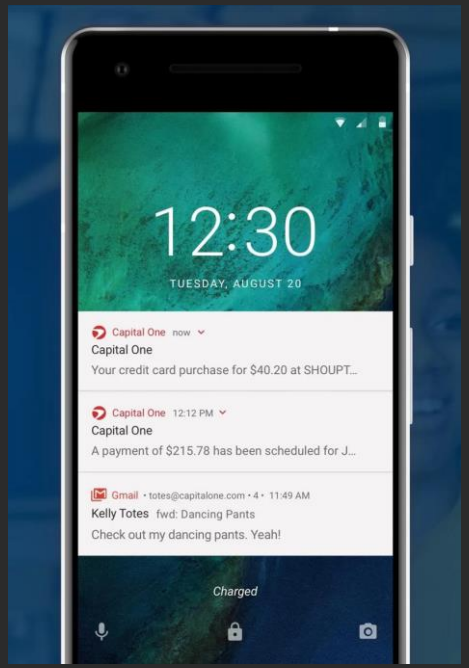
Notifications



Broadly, we can think of notification *content* coming from two places:

- Local - generated from the Android app itself.
- Remote - generated external from the app and pushed to the device.
  - Credit card transactions
  - Userbase campaigns

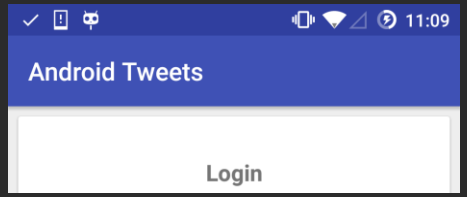
Notifications

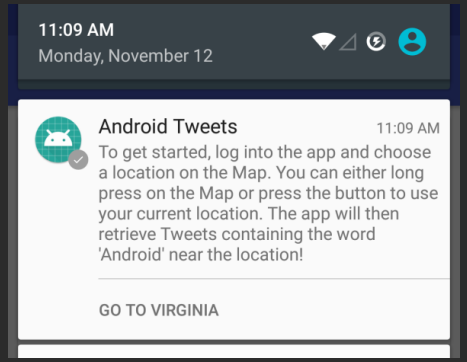


Broadly, we can think of notification *content* coming from two places:

- Local - generated from the Android app itself.
  - Android's NotificationManager
- Remote - generated external from the app and pushed to the device.
  - Firebase Cloud Messaging

Notifications





Let’s see how we can create a notification that posts when the user creates a new account.

- Tapping the notification will just open the app to the Login screen.
- There will be an action to open directly to the Tweets screen for Virginia.

Creating a Notification

Notifications are part of AndroidX’s core dependency.

```
implementation 'androidx.core:core-ktx:1.12.0'
```

Creating a Notification

Construct a notification using the NotificationCompat.Builder.

There are many pieces to customize.

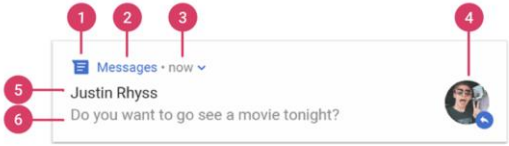


Figure 7. A notification with basic details

The most common parts of a notification are indicated in figure 7 as follows:

1. Small icon: This is required and set with `setSmallIcon()`.
2. App name: This is provided by the system.
3. Time stamp: This is provided by the system but you can override with `setWhen()` or hide it with `setShowWhen(false)`.
4. Large icon: This is optional (usually used only for contact photos; do not use it for your app icon) and set with `setLargeIcon()`.
5. Title: This is optional and set with `setContentTitle()`.
6. Text: This is optional and set with `setContentText()`.

Creating a Notification

Construct a notification using the NotificationCompat.Builder.

We will just use the signup to our Yelps App to send a welcome notification

Creating a Notification

Construct a notification using the NotificationCompat.Builder.

- The 2nd parameter is the notification category (channel). Only relevant for Android Oreo (API 26) and higher (more in a bit).

```
val mBuilder = NotificationCompat.Builder(this, "default")
```

Creating a Notification

Construct a notification using the NotificationCompat.Builder.

```
val mBuilder = NotificationCompat.Builder(this, "default")
    .setSmallIcon(R.drawable.ic_check)
    .setContentTitle(“Yelps")
    .setContentText("Welcome to Yelp Businesses!")
```

Creating a Notification

Display the notification by using the NotificationManager

```
val mBuilder = NotificationCompat.Builder(this, "default")
    .setSmallIcon(R.drawable.ic_check)
    .setContentTitle(“Yelps")
    .setContentText("Welcome to Yelp Businesses!")

NotificationManagerCompat.from(this).notify(0, mBuilder.build())
```

Creating a Notification

Display the notification by using the NotificationManager (note, this won't show *yet* for Oreo or higher (>= 8.0))

```
val mBuilder = NotificationCompat.Builder(this, "default")
    .setSmallIcon(R.drawable.ic_check)
    .setContentTitle(“Yelps")
    .setContentText("Welcome to Yelp Businesses!")

NotificationManagerCompat.from(this).notify(0, mBuilder.build())
```

Creating a Notification

The first parameter to notify is **an ID**. If you want to **update** the notification in the future, call notify again with the same ID.

```
val mBuilder = NotificationCompat.Builder(this, "default")
    .setSmallIcon(R.drawable.ic_check)
    .setContentTitle(“Yelps")
    .setContentText("Welcome to Yelp Businesses!")

NotificationManagerCompat.from(this).notify(0, mBuilder.build())
```

Creating a Notification

So ideally, we would use something more random, or an incrementing counter or timestamp.

```
NotificationManagerCompat.from(this).notify(..., mBuilder.build())
```

Creating a Notification

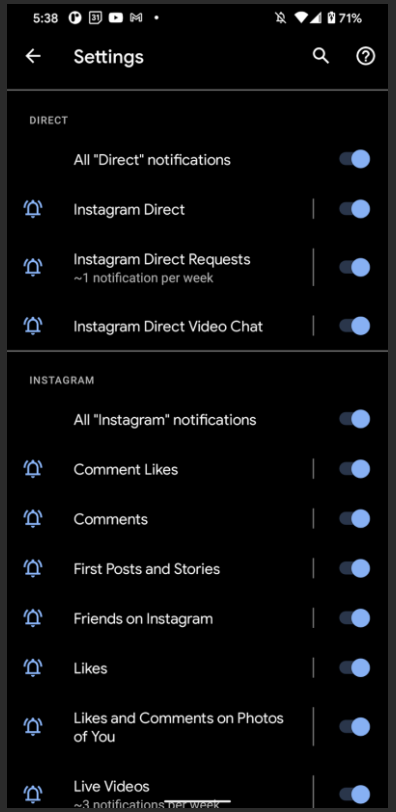
Now - about the NotificationChannel “default”

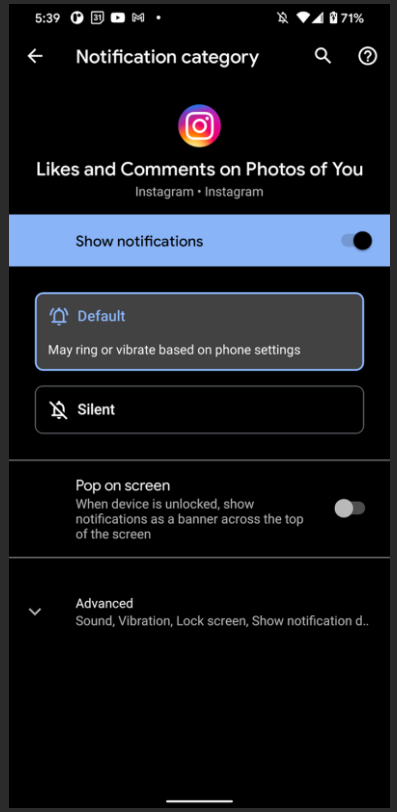
```
val mBuilder = NotificationCompat.Builder(this, "default")
```

Notification Channels

For Android Oreo and higher, you need explicitly create a “notification channel” to use with your notifications.

- This allows a user to selectively subscribe / unsubscribe from different notification types.





Notification Channels

For Android Oreo and higher, you need explicitly create a “notification channel” to use with your notifications.

```
private fun createNotificationChannel() {
   // Only needed for Android Oreo and higher
   if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
       val name = "Default Notifications"
       val descriptionText = "The app's default notification set"
       val importance = NotificationManager.IMPORTANCE_DEFAULT

       val channel = NotificationChannel("default", name, importance)
       channel.description = descriptionText

       // Register the channel with the system
       val notificationManager: NotificationManager =
           getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
       notificationManager.createNotificationChannel(channel)
   }
}
```

Notification Channels

You only have to do this once - for example, in onCreate.

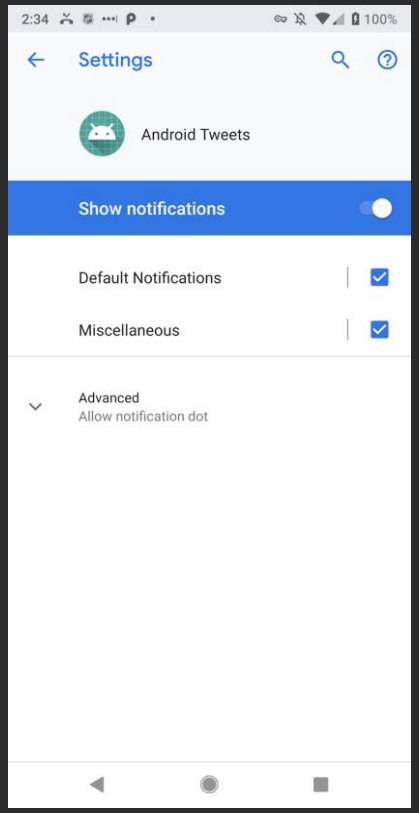
```
private fun createNotificationChannel() {
    // Only needed for Android Oreo and higher
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val name = "Default Notifications"
        val descriptionText = "The app's default notification set"
        val importance = NotificationManager.IMPORTANCE_DEFAULT

        val channel = NotificationChannel("default", name, importance)
        channel.description = descriptionText

        // Register the channel with the system
        val notificationManager: NotificationManager =
            getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(channel)
    }
}
```

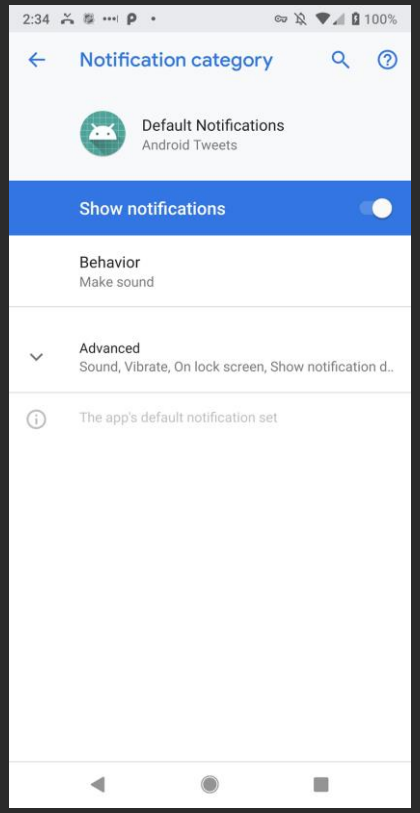
Notification Channels

You’re basically setting the text & notification behavior for the Settings app to display.

```
private fun createNotificationChannel() {
    // Only needed for Android Oreo and higher
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val name = "Default Notifications"
        val descriptionText = "The app's default notification set"
        val importance = NotificationManager.IMPORTANCE_DEFAULT

        val channel = NotificationChannel("default", name, importance)
        channel.description = descriptionText

        // Register the channel with the system
        val notificationManager: NotificationManager =
            getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(channel)
    }
}
```

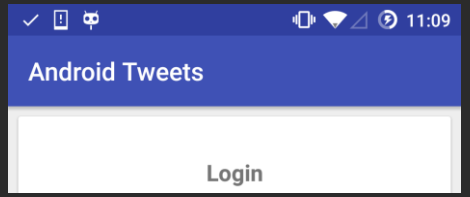
Notification Channels

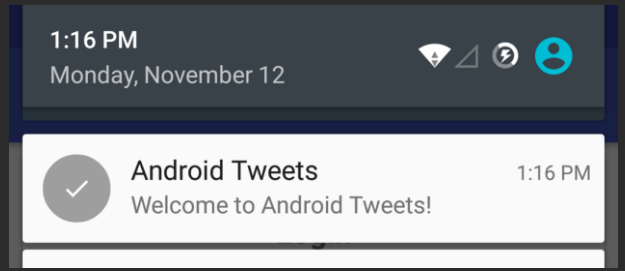


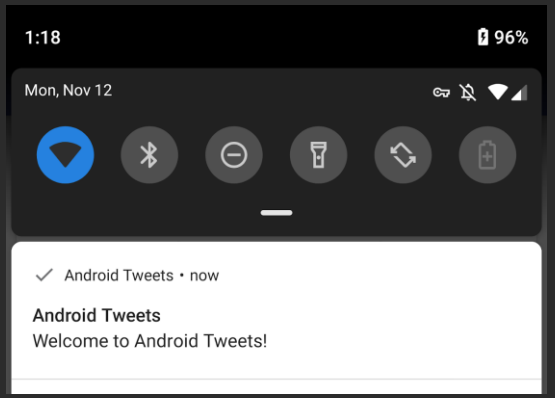


Creating a Notification

The resulting look depends on your Android version & phone make.



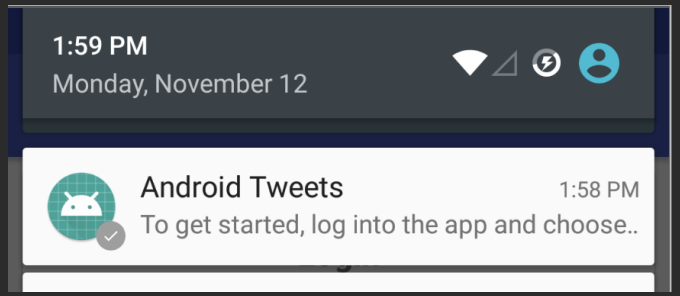




More Styles

Long text will get truncated...

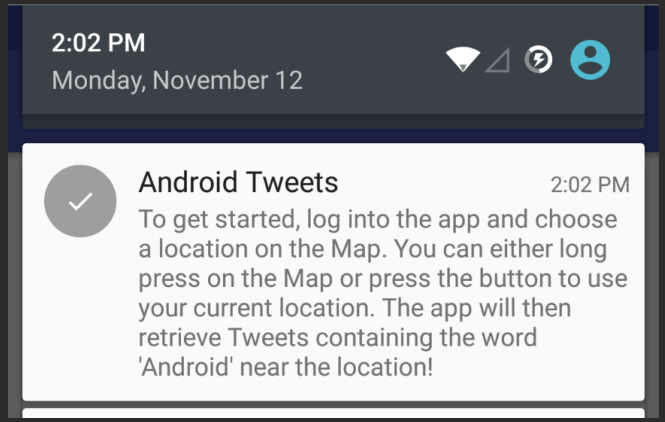
```
val mBuilder = NotificationCompat.Builder(this, "default")
    .setSmallIcon(R.drawable.ic_check_white)
    .setContentTitle("Android Tweets")
    .setContentText("To get started, log into the app and choose a location on the Map. You
can either long press on the Map or press the button to use your current location. The app
will then retrieve Tweets containing the word 'Android' near the location!")

// ...
```



More Styles

But you can set a “big text” style on your notification.

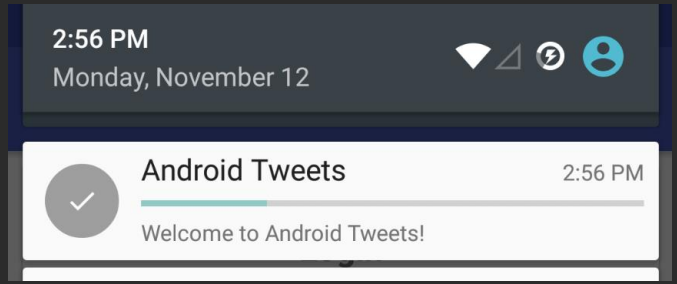
```
val mBuilder = NotificationCompat.Builder(this, "default")
     .setSmallIcon(R.drawable.ic_check_white)
     .setContentTitle("Android Tweets")
     .setStyle(NotificationCompat.BigTextStyle()
          .bigText("To get started, log into the app and
     choose a location on the Map. You can either long
     press on the Map or press the button to use your
     current location. The app will then retrieve Tweets
     containing the word 'Android' near the location!"))
// ...
```



More Styles

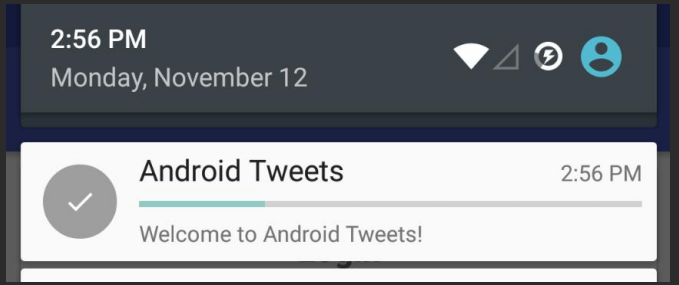
You can also set a progress bar (e.g. tracking a download). Use the same notification ID to update existing progress.

```
val mBuilder = NotificationCompat.Builder(this, "default")
      .setSmallIcon(R.drawable.ic_check_white)
      .setContentTitle("Android Tweets")
      .setContentText("Welcome to Android Tweets!")

      // Max progress, current progress, indeterminate
      .setProgress(100, 25, false)

// ...
```



More Styles

How does a background downloader work and constantly update the notification?



- The same notification can be continually updated by using the same Notification ID.

- I don’t cover Services - but they are like long-running threads that you can have running in the background, even without UI.
  - Common for downloading & media playback.

Adding a Tap Action

When the user taps on the notification, the simplest thing we could do is open the first screen of our app.

Adding a Tap Action

We can’t just add an “onClickListener” like a normal button:

Adding a Tap Action

We can’t just add an “onClickListener” like a normal button:

- The user can click the notification when our app isn’t running (e.g. none of our code is loaded).

Adding a Tap Action

We can’t just add an “onClickListener” like a normal button:

- The user can click the notification when our app isn’t running (e.g. none of our code is loaded).
- Unlike our normal UIs, the OS is responsible for managing and displaying notifications.

Adding a Tap Action

We can’t just add an “onClickListener” like a normal button:

- The user can click the notification when our app isn’t running (e.g. none of our code is loaded).
- Unlike our normal UIs, the OS is responsible for managing and displaying notifications.
- **Another program** (the OS) has to handle the click event on our behalf.
  - The click event can even be handled when our app is not loaded into memory!

Pending Intent

A **PendingIntent** is like an **Intent**, but meant to be executed in the future.

Pending Intent

A **PendingIntent** is like an **Intent**, but meant to be executed in the future.

- You can give a PendingIntent to another application and it can execute it *as if it was your app*.

Pending Intent

A **PendingIntent** is like an **Intent**, but meant to be executed in the future.

- You can give a PendingIntent to another application and it can execute it *as if it was your app*.
- There is a predefined set of actions you can build a PendingIntent for (for example, “start an activity”).

Notification Tap Event

Building a pending intent to launch the login screen - start with the normal intent.

```
val intent = Intent(this, MainActivity::class.java)
```

Notification Tap Event

Now we build a PendingIntent to start an activity & wraps our Intent.

```
val intent = Intent(this, MainActivity::class.java)

// A PendingIntent explicitly to start an activity
val pendingIntent = PendingIntent.getActivity(this, 0, intent, 0)

val pendingMainActivityIntent= PendingIntent.getActivity(this,0,
   mainActivityIntent,
   PendingIntent.FLAG_IMMUTABLE)

Four parameters: Context, request code, intent, and flag
```

Notification Tap Event

The two `0` params are the requestCode and flags parameters, which are unneeded here (but can be used to identify a specific PendingIntent later).

```
val intent = Intent(this, MainActivity::class.java)

// A PendingIntent explicitly to start an activity
val pendingIntent = PendingIntent.getActivity(this, 0, intent, 0)
```

Notification Tap Event

Call setContentIntent on the builder to specify the PendingIntent to fire on click.

```
val mBuilder = NotificationCompat.Builder(this, "default")
    .setSmallIcon(R.drawable.ic_check_white)
    .setContentTitle("Android Tweets")
    .setContentText("Welcome to Android Tweets!")
    .setContentIntent(pendingIntent)

// ...
```

Notification Tap Event

To have the notification auto-dismiss after clicking, you can setAutoCancel

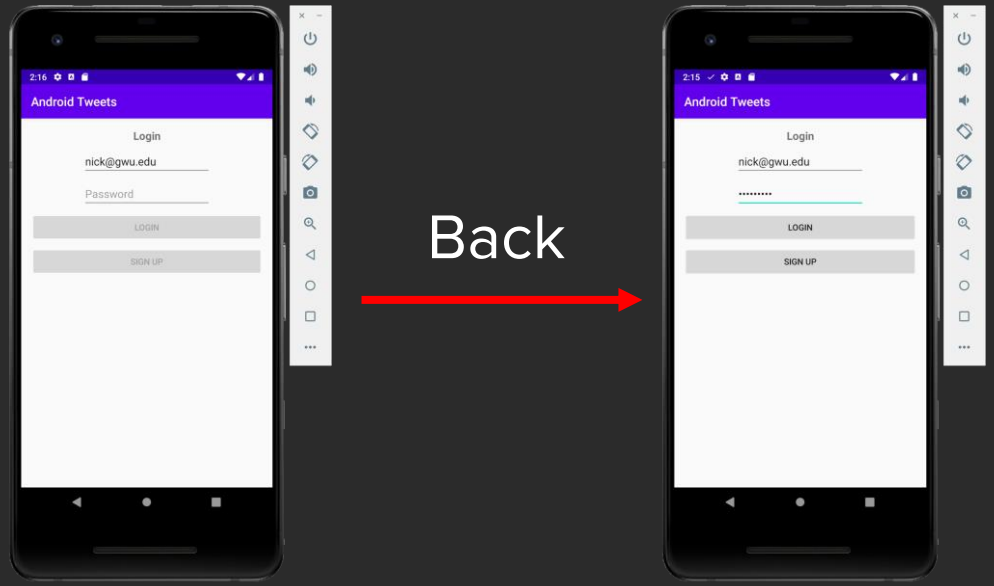
```
val mBuilder = NotificationCompat.Builder(this, "default")
    .setSmallIcon(R.drawable.ic_check_white)
    .setContentTitle("Android Tweets")
    .setContentText("Welcome to Android Tweets!")
    .setContentIntent(pendingIntent)
    .setAutoCancel(true)

// ...
```

Notification Tap Event

Note: sometimes with notifications, you may need to reinstall the app completely to see the changes take effect.

Notification Tap Event

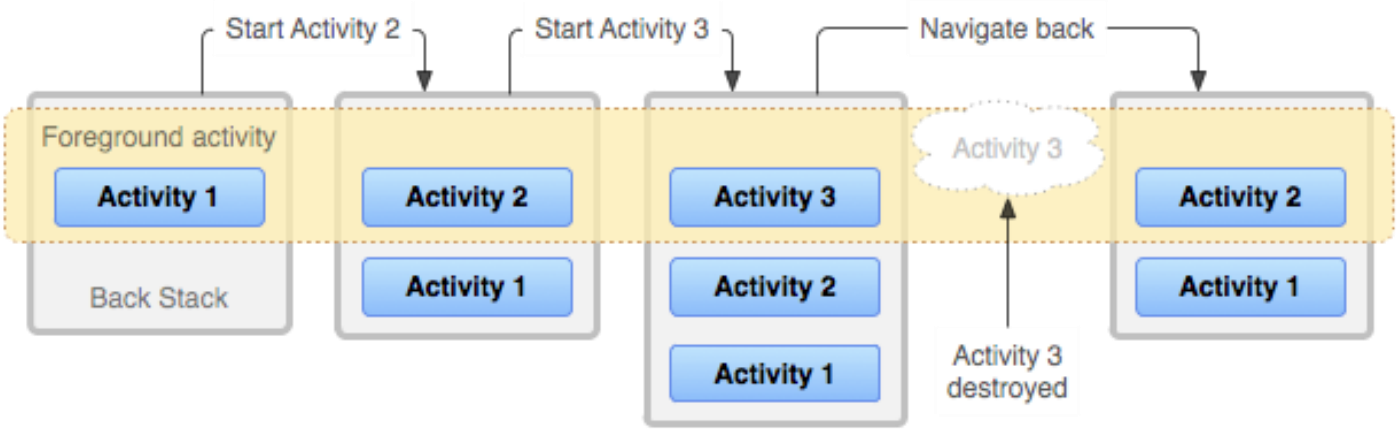
Notice: there's some odd behavior if the Activity is *already* launched, the user taps the notification, then hits Back.



Two login screens?

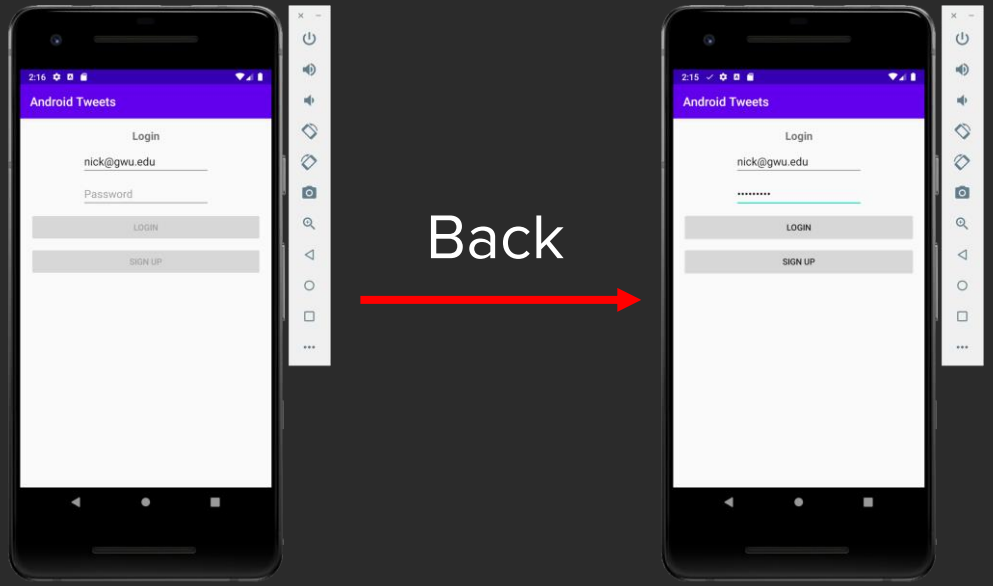
The Back Stack

Android keeps a stack of Activities the user has gone thru and pops the top of the stack whenever the user hits the Back button.



The Back Stack

That's in play here too - our PendingIntent launches a *second instance* of our Login screen. Now there's two on the stack.



The Back Stack

The Back Stack can be influenced by setting **flags** on your Intents.

- Sometimes you only want one instance of your Activity.
  - This is what we want in our scenario.
- Sometimes you want multiple Back Stacks.
  - Google Docs for Mobile does this - to allow you to have multiple documents open and switch between them via Recent Apps.

Intent Flags

We could review the list of common flags here to determine what to use...

- If the app isn't loaded, we need to create a *new* back stack.
- If the app is loaded, we want to basically "surface" the existing login screen in the stack to the user.

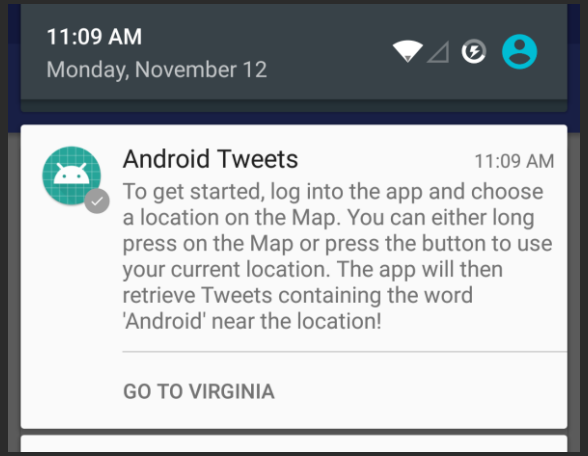
Intent Flags

Based on those requirements we could use both the SINGLE_TOP and CLEAR_TOP flags (need a binary OR).

```
// The “or” keyword does a binary OR operation in Kotlin
val intent = Intent(this, MainActivity::class.java)
intent.flags = Intent.FLAG_ACTIVITY_SINGLE_TOP or Intent.FLAG_ACTIVITY_CLEAR_TOP
```

Notification Actions

You can also specify button actions on your notification. PendingIntents are also used here to specify click actions.



Notification Actions

We can set up a PendingIntent to launch the YelpListings Activity with an Address set to Virginia (in reality, this could be controlled by the user location).

Back Stack Preparation

If we send the user straight to the Yelps Listings Activity and they press Back, we'd want to send to the Google Maps activity (and then back to the MainActivity)

Back Stack Preparation

To do this, the easiest way is to specify the intended “parent activities” in the Manifest.

```
<activity
   android:name=".MapsActivity"
   android:parentActivityName=".MainActivity"/>

<activity
   android:name=".YelpListings"
   android:parentActivityName=".MapsActivity"/>
```

Notification Actions

Setting up the Intent to launch the YelpsListing Activity.

```
val address = Address(Locale.US)
address.setAddressLine(0, "Richmond")
address.locality = "Richmond"
address.adminArea = "Virginia"
address.latitude = 37.5407
address.longitude = -77.4360

val yelpsIntent = Intent(this, YelpListing::class.java)
YelpsIntent.putExtra("address", address)
```

Notification Actions

This time, we'll use the **TaskStackBuilder** to create our PendingIntent. It will generate the intents to set up the back stack the way we want, using the parent activities.

Notification Actions

This time, we'll use the **TaskStackBuilder** to create our PendingIntent. It will generate the intents to set up the back stack the way we want, using the parent activities.

```
// Builder which will also build the intents to launch the
// previous activities when the user presses back

// Use the import from "androidx.core.app"
val YelpsPendingIntentBuilder = TaskStackBuilder.create(this)
YelpsPendingIntentBuilder.addNextIntentWithParentStack(yelpsIntent)
```

Notification Actions

This time, we'll use the **TaskStackBuilder** to create our PendingIntent. It will generate the intents to set up the back stack the way we want, using the parent activities.

```
// Create the actual PendingIntent from the builder
// The extra parameters are requestCode / flags which we don't need
val yelpsPendingIntent = yelpsPendingIntentBuilder.getPendingIntent(
    0,
    0
)
```

Notification Actions

Finally, we can call addAction on our NotificationBuilder to set up the button.

```
// The “0” is for an icon resource ID
val mBuilder = NotificationCompat.Builder(this, "default")
    .setSmallIcon(R.drawable.ic_check_white)
    .setContentTitle(“Yelps")
    .setContentText("Welcome to YelpListing!")
    .addAction(0, "Go To Virginia", yelpsPendingIntent)

// ...
```

Other Actions

What if tapping a notification or action you didn’t want to launch *any* UI and just trigger some code to run?

- Pause / resume a download
- Play / skip / rewind song

Other Actions

We haven’t talked about them, but instead of launching an activity, you can have a PendingIntent trigger *arbitrary code* in your application through the use of BroadcastReceivers.

Other Actions

We haven’t talked about them, but instead of launching an activity, you can have a PendingIntent trigger *arbitrary code* in your application through the use of BroadcastReceivers.

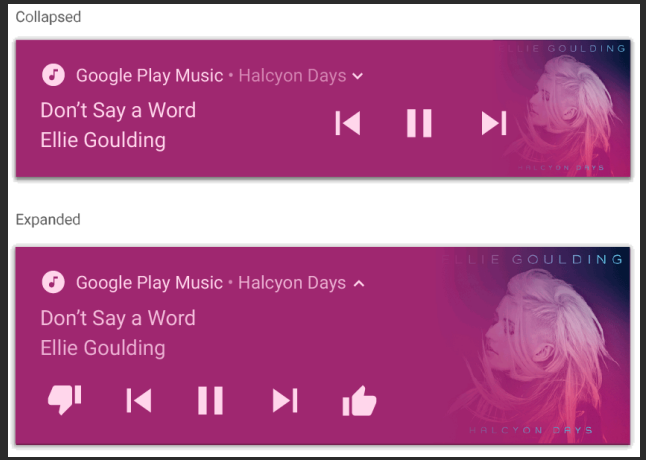
You can also create a completely custom notification UI.

Media Controls Notifications

The basis for notifications with media controls is via PendingIntents.

Android has a built in “template” for these types of notifications.

```
.setStyle(MediaNotificationCompat.MediaStyle()
```



Questions?

Helpful Links

Screen Rotation

- Handle Configuration Changes

Notifications

- Overview
- Creating a Notification
- Starting Activities
- Additional Notification Styles
- Custom Notification