Lab on apps development for tablets, smartphones and smartwatches

Week 5: ViewModels and System Services

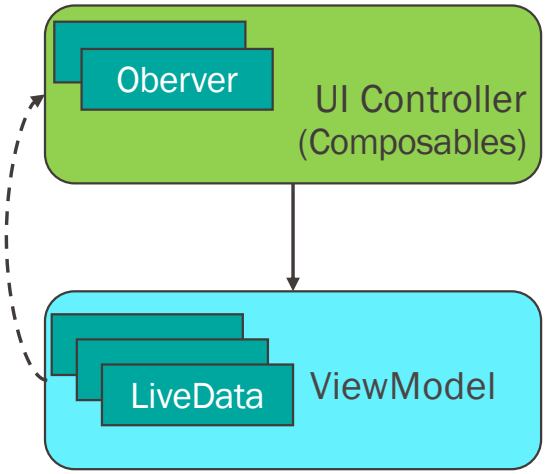
Giovanni Ansaloni

Dimitra Tatli, Riselda Kodra, Yuxuan Wang
Qunyou Liu, Amirhossein Shahbazinia, Christodoulos Kechris

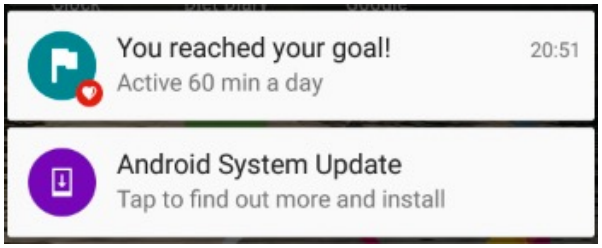
School of Engineering (STI) – Institute of Electrical and Micro Engineering (IEM)

Class outline

- **ViewModels and Livedata**



- System Services
  - Sensor services
  - Notifications
  - Alarms



- Broadcast receivers

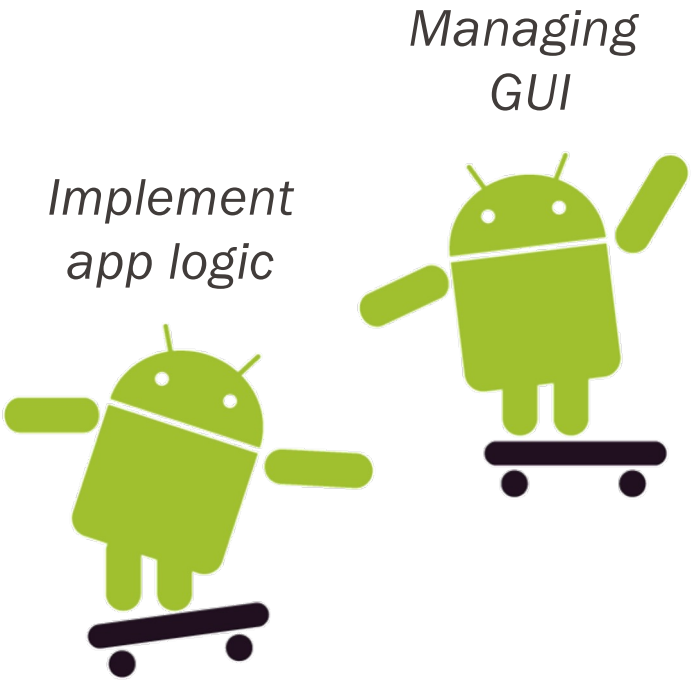
Where are we?

- Our Composables are becoming increasingly complicated
  - Listeners, navigation, menus, …

- Are Composable functions doing too much?

  1. Management of the Composables GUI
     - displaying views, listening to user actions

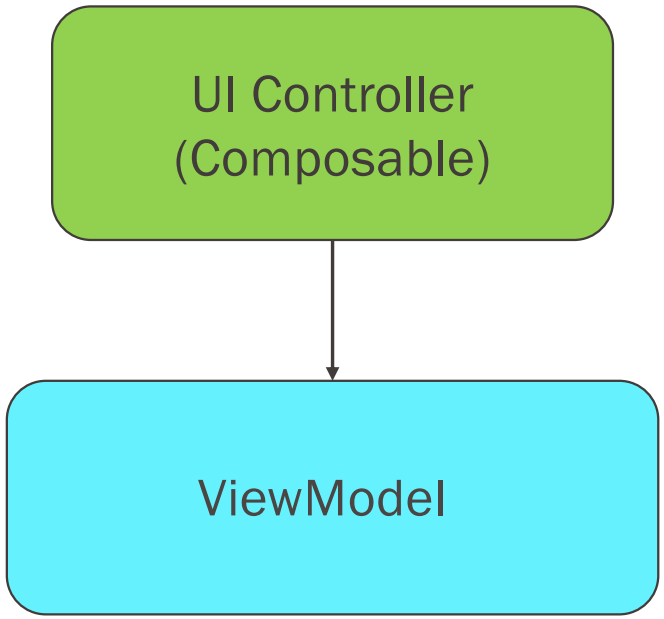
  2. Holding and manipulating data



Separation of concerns

1. UI controllers → Composables (e.g. Screens)
   - Manage the GUI
     → displaying/updating

2. ViewModels
   - Hold and manipulate data
     → implement the app "logic"



- UI controllers are re-created during configuration changes
  - rotations, …
- ViewModels persist during configuration changes

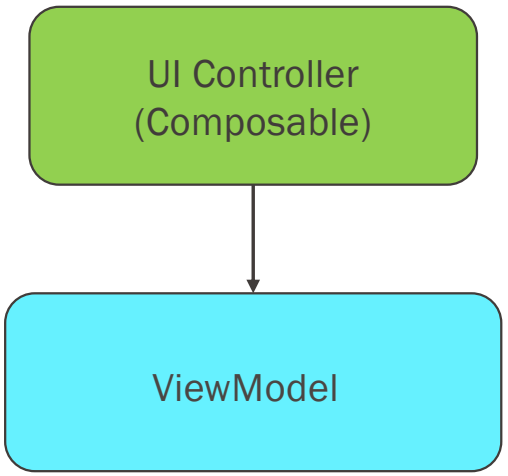
ViewModels

- Extend the ViewModel class

```
class MyViewModel : ViewModel(){...}
```

- Created/Linked to and by a UI controller
  - Commonly provided as a last default parameter in the constructor

```
@Composable
fun MyScreen(
    ...
    modifier: Modifier = Modifier,
    myViewModel: MyViewModel = viewModel()
) {
```



- Link viewModel if exist

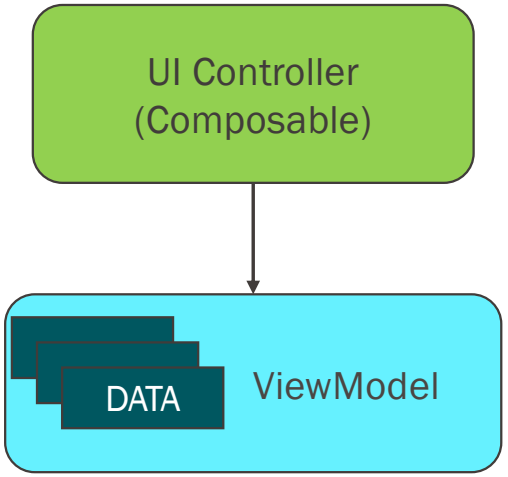
  or

- or create one otherwise

UI-controller vs ViewModel

- UI controller: everything (and only) what is related to GUI
  - declaration/references to Composables
  - state variables
  - events
    - e.g. onMyButtonClicked()
- ViewModel
  - all other variables, methods etc.. implementing the app logic
- UI Controllers can access methods in ViewModels

```
viewModel.viewModelFun()
```



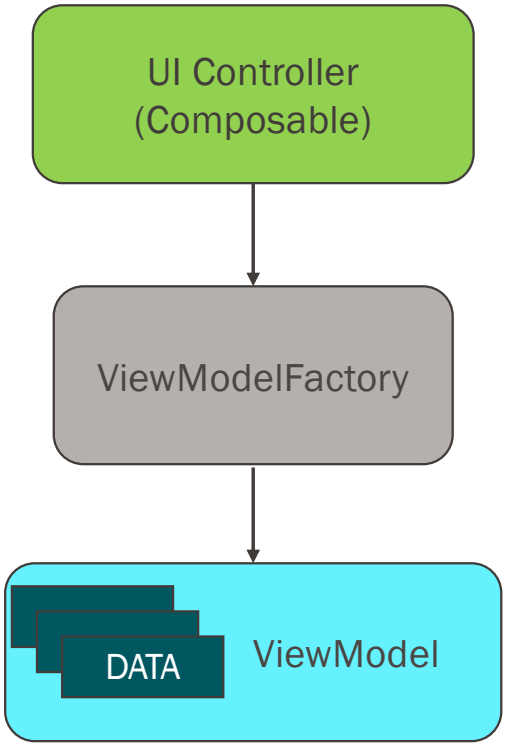
Initialize ViewModels with a Factory class

- The initial state of ViewModels can be parametric
  - i.e. depend on the arguments passed to the UI controller that creates the ViewModel

- ViewModel using parameters such as:

```
class MyViewModel(myParameter: Int) : ViewModel(){
```

...are created using a ViewModelFactory



ViewModels parameters

- Parameters passed during Viewmodel creation via a Factory

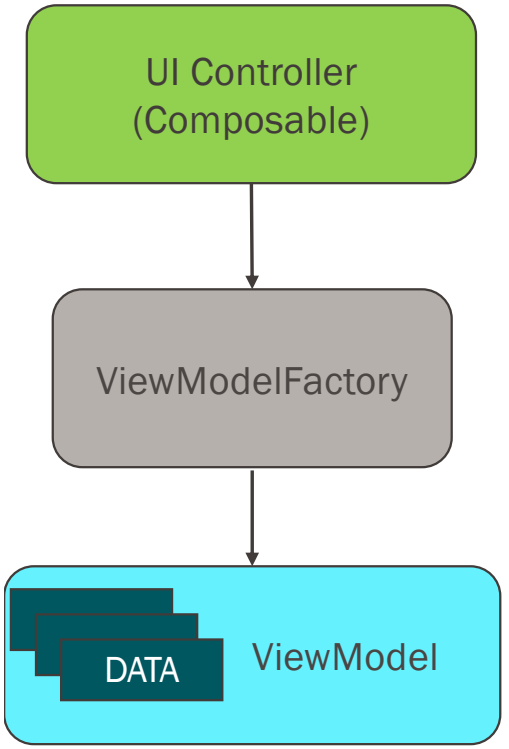
```
@Composable
fun MyScreen(
    modifier: Modifier = Modifier,
    myViewModel: MyViewModel = viewModel(
        factory = MyViewModelFactory(myParameter))
) {
```

- ViewModelFactory → class that returns a ViewModel instance

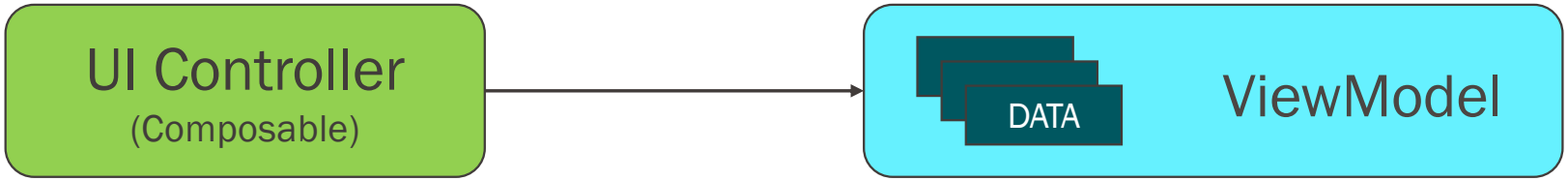
```
class MyViewModelFactory(private val myParameter: Int) :
                                        ViewModelProvider.Factory {

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
            return MyViewModel(myParameter) as T
    }
}
```

parameter

parametric ViewModel



UI controllers and ViewModels

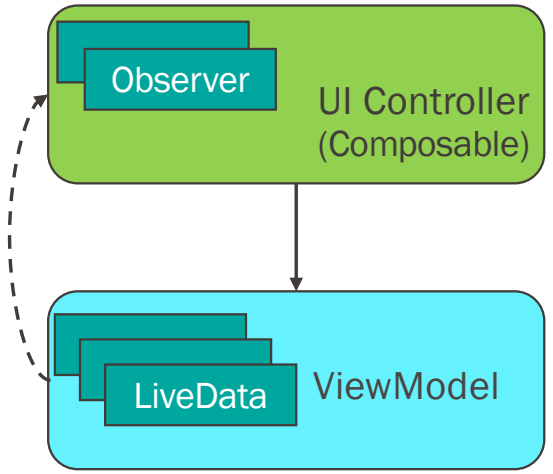


- Does not manipulate data to be displayed in the UI
- Contain code for displaying data, managing events/state variables
- Does contain a reference to the associated ViewModel
- Destroyed and re-created at configuration changes

- Does contain the data the UI controller displays
- Contains code for data processing
- Does not contain reference to the associated UI controller
- Destroyed only when the associated Activity goes away permanently

LiveData and Observers

- Data values are stored in ViewModels,
  but the UI controller should be aware of (some of) them



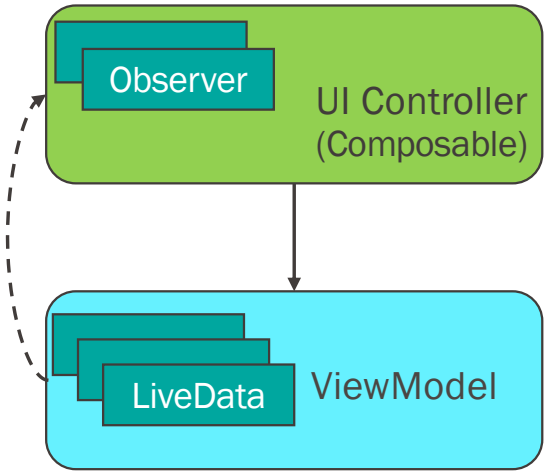
- When should the UI Controller update the GUI?
  - Reflecting changes in the ViewModel

- Android provides for
  - LiveData in ViewModel Classes
    - changes in LiveData are notified to the UI controller
    - lifeCycle-aware
      → notify only if/when UI controller is active and/or becomes visible
  - Observers in UI controllers

LiveData

- LiveData are declared in ViewModels
  - Wrapper that can contain any kind of object or primitive type
    - Examples:

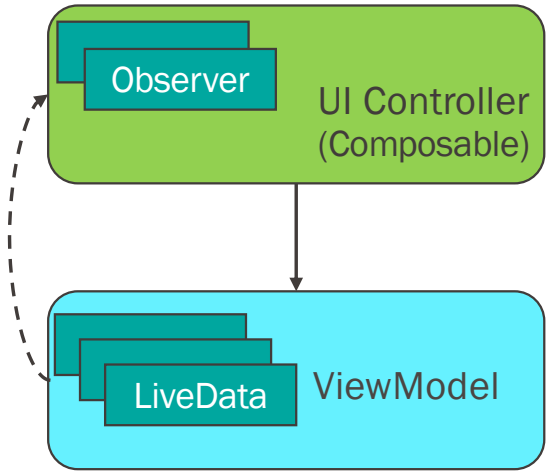
```
val score: LiveData<Int>

val word: LiveData<String>
```



- Observers in UI Controller link Composable and LiveData

Observers in Composables

- observeAsState observe LiveData from the ViewModel and transforms it into state
  - Every time the LiveData updates, the UI that depends on ”word” is recomposed.




```
@Composable
fun GameScreen(
    modifier: Modifier = Modifier,
    gameViewModel: GameViewModel = viewModel()
) {
    val word by gameViewModel.word.observeAsState(initial = "")

    …
}
```

observed LiveData in ViewModel

observer state variable in UI controller

MutableLiveData and LiveData

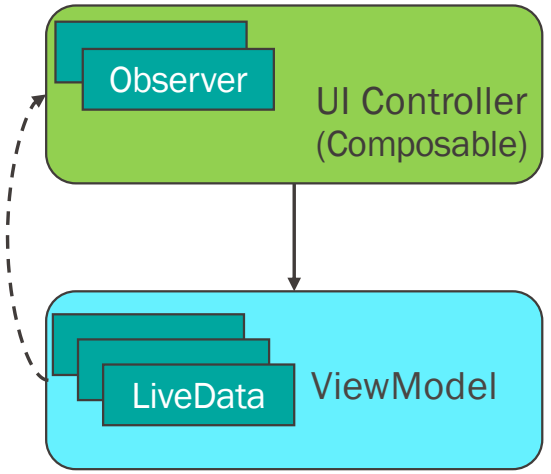
- LiveData should be readable,
  but not writable by the UI controller
  → separation of concerns

- Use MutableLiveData in ViewModels
  - declare it private to make it writable only inside the class

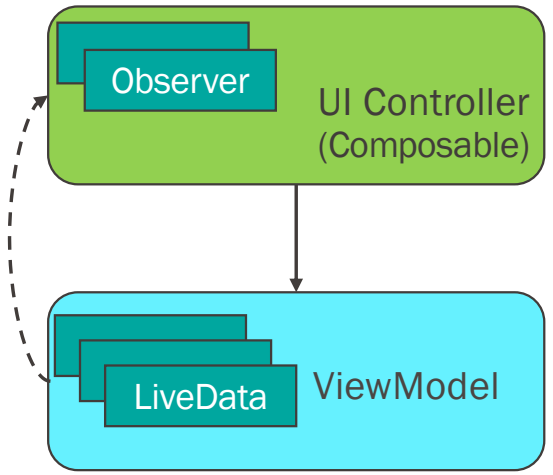
```
private val _word = MutableLiveData<String>()

_word.value = "myWord"
```

- Associate LiveData with MutableLiveData with getter

```
val word: LiveData<String>
    get() = _word
```



Summing up



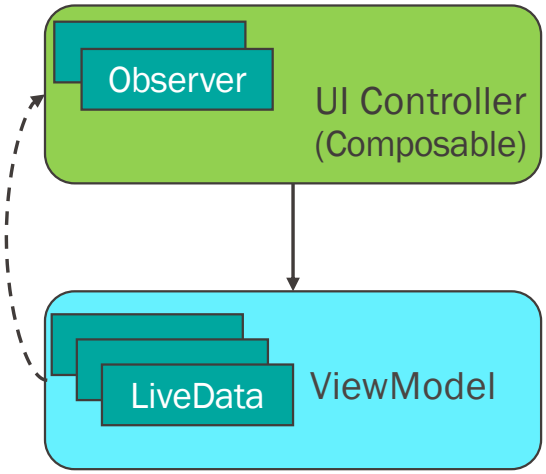
- Separation of concerns
  - UI Controller vs. ViewModel
- ViewModels can be parametrized via ViewModelFactory
- LiveData/MutableLiveData incapsulates observed data
  - MutableLiveData → readable/writable, private of ViewModel
  - LiveData → readable by UI controller
- Observers automatically update the UI when LiveData changes
  - `val <observers> by myViewModel.<LiveDataVar>.observeAsState()`

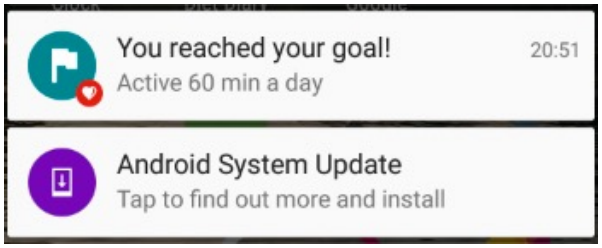
Class outline

- ViewModels and Livedata

- **System Services**
  - Sensor services
  - Notifications
  - Alarms

- **Broadcast Receivers**





System Services

- Allows applications to access OS and hardware features
  - **Sensor Service**
  - **Notification Service**
  - **Alarm Service**
  - Power Manager Service
  - Vibrator Service, Audio Service
  - Telephony Service, Connectivity Service, Wi-Fi Service
  - …
- Available via Service Managers

```
val sensorMgr = app.getSystemService(SENSOR_SERVICE) as SensorManager

val alarmMgr = app.getSystemService(Context.ALARM_SERVICE) as AlarmManager

val notificationMgr = app.getSystemService(Context.NOTIFICATION_SERVICE)
                                                    as NotificationManager
```

System Services

- Service Managers require an application Context
  - Handle to characteristics of application, OS, and device
    → AndroidViewModel instead of ViewModel

```
class myViewModel(private val app: Application, <MY_OTHER_INPUTS>) :
AndroidViewModel(app) {
```

- AndroidViewModel does not require, but supports viewModelFactory

```
class myViewModelFactory(private val application: Application,
                         <MY_OTHER_INPUTS>) : ViewModelProvider.Factory {

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
            return myViewModel(application, <MY_OTHER_INPUTS>) as T
    }
}
```

Sensor service

- Many types of sensors:
  - Accelerometer, Gyroscope, Light, …
  - GPS
  - Heart Rate
  - …
- Each Sensor contains information about the vendor, type, …

- List of sensors on a device:

```
var sensorlist = sensorMgr.getSensorList(Sensor.<TYPE>)
```

  - *<TYPE>* → TYPE_ACCELEROMETER, TYPE_LIGHT, TYPE_ALL

- Handle to a sensor of a given type

```
val slightSens = sensorMgr.getDefaultSensor(Sensor.TYPE_LIGHT)
```

Monitoring sensor events

- To get sensor data, two callbacks must be implemented via the SensorEventListener interface
  - the accuracy of a sensor changes: onAccuracyChanged()
  - a sensor reports a new value: onSensorChanged()

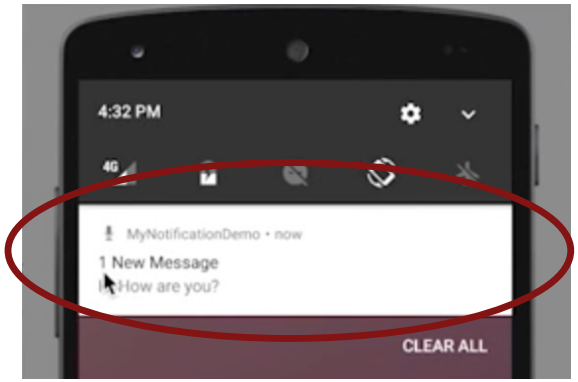
- Steps:
  1. Implement SensorEventListener
  2. Create the sensor manager
  3. Provide methods to register/unregister the listener
     → called in onPause()/onResume (or corresponding lifeCycle effects)
  4. Do something when sensor accuracy/value changes

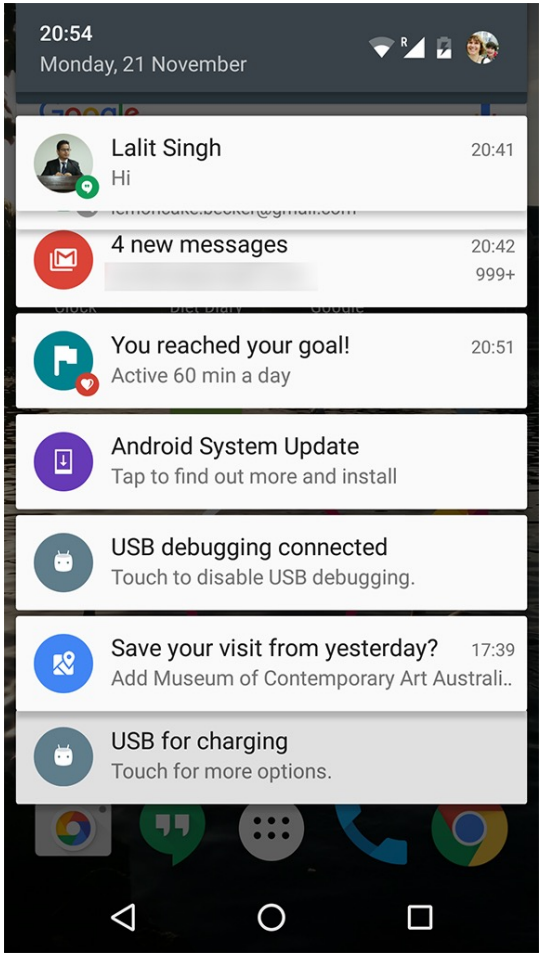
```
class myViewModel(private val app: Application) :
                    AndroidViewModel(app), SensorEventListener{          1.

    val  sensorMgr = app.getSystemService(SENSOR_SERVICE) as SensorManager   2.

    fun registerSensor() {
        val lightSens = sensorMgr.getDefaultSensor(Sensor.TYPE_LIGHT)
        sensorMgr.registerListener(this, lightSens,
                                   SensorManager.SENSOR_DELAY_NORMAL);
    }

    fun unregisterSensor() {                                             3.
        sensorMgr.unregisterListener(this)
    }

    override fun onSensorChanged(event: SensorEvent?) {
        //read value to be incapsulated in LiveData variable
        val flux: Float = event?.values?.get(0) ?: 0F
    }

    override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {     4.
        //Do something if sensor accuracy changes
    }
```

(try to) listen every 3us

Notification service

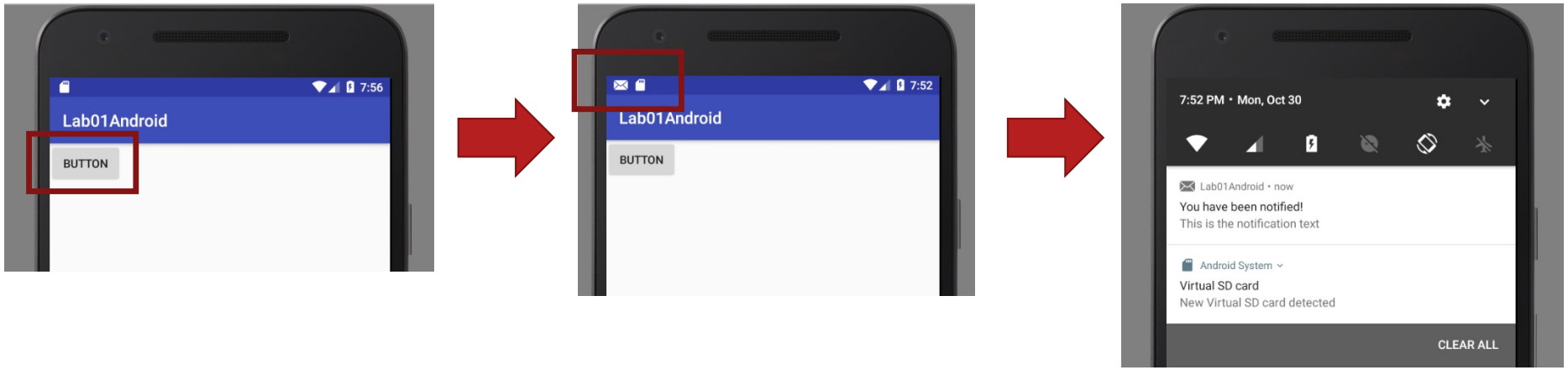
- What is a notification?
  - A message displayed outside of the regular app UI
    - Does not interrupt operations on the foreground app
  - To see details, user opens the notification drawer
  - At minimum, consists of
    - Small icon
    - Title
    - Detail text





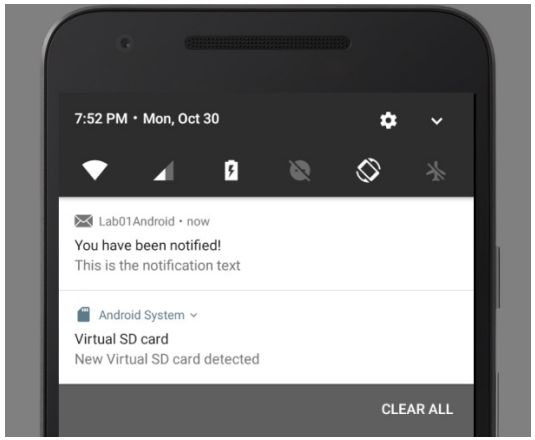
A very simple example

- ViewModel that launches a notification



How to create a Notification - setup

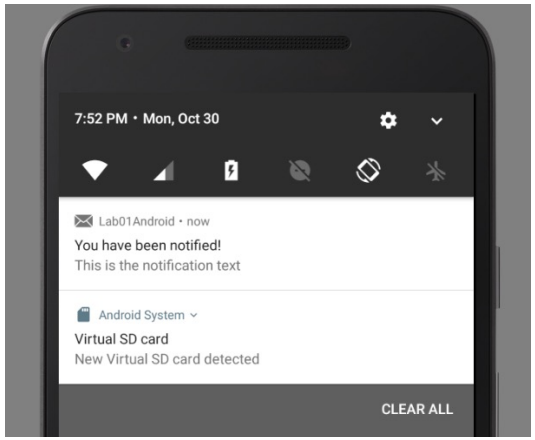
1. Getting a reference to NotificationManager SystemService
2. Creating a communication channel and attach it to the NotificationManager
   - required for Android version >= 8 (Oreo)




```
class myViewModel(private val app: Application) :
                        AndroidViewModel(app) {

1.  private val notificationManager =
    app.getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
    init{
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
2.      val notificationChannel = NotificationChannel(
                "myChannelID", "myChannel",
                NotificationManager.IMPORTANCE_HIGH)
        notificationManager.createNotificationChannel(notificationChannel)
        }
    }
```

How to create a Notification – display

3. Build the Notification message
   - Title / Content / Icon
4. Send the notification to the Notification Manager




```
fun displayNotification() {
    val notifyBuilder = NotificationCompat.Builder(app, "myChannelID")
            .setSmallIcon(R.drawable.cooked_egg)
            .setContentTitle("My notification")
            .setContentText("Time to work!")

    notificationManager.notify(NOTIFICATION_ID, notifyBuilder.build())
}
```

3.

4.

Notifications and pending intents

- Notifications can be interactive, responding when user tap on them
  - Notification may have buttons (e.g. “reply”)
    - *https://developer.android.com/training/notify-user/build-notification.html*

- Behavior of tapping on notification is specified in “PendingIntent” object
  - e.g. “app is opened when notification is tapped”
  - Described as part of notification

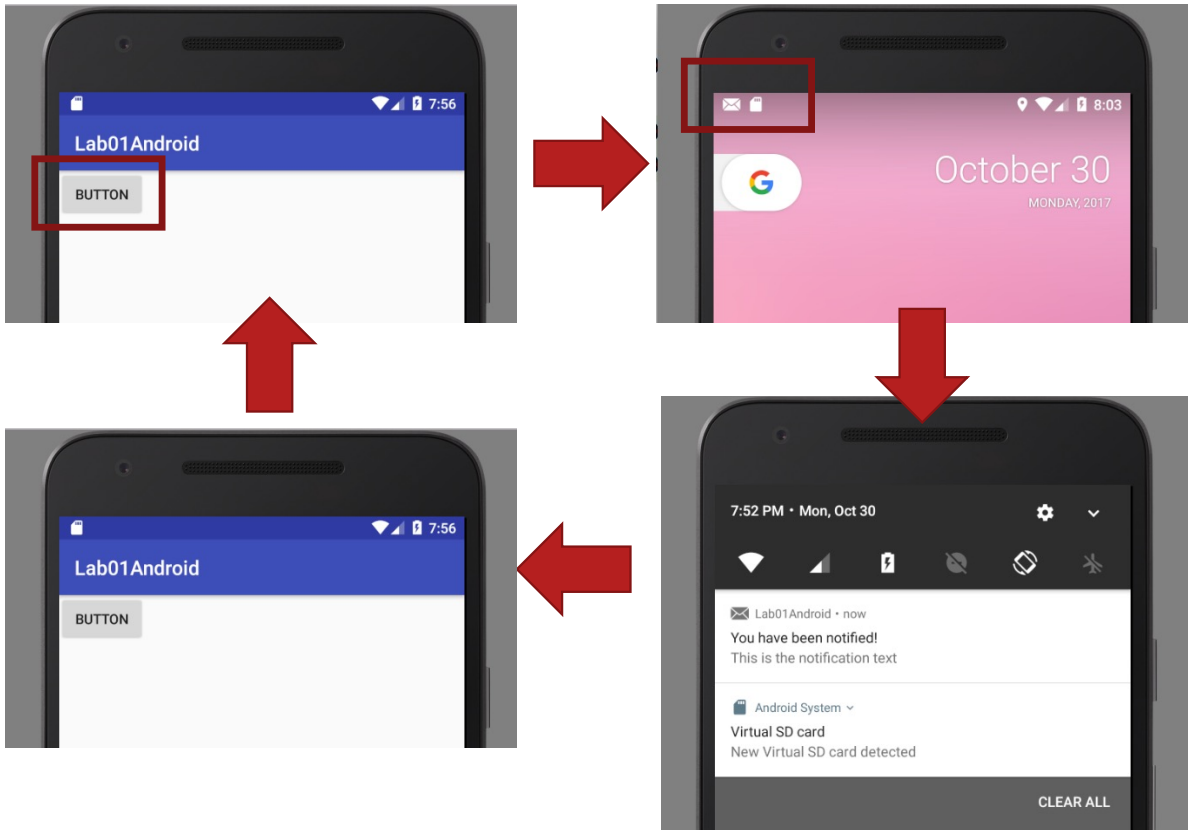
Creating a PendingIntent

1. Create an intent
2. Create a PendingIntent
3. Add it to notification builder
   - And then call notify()
4. *Only for this example:*
   Force quit the app
   (*activity*!!.finish())

```
fun displayNotification() {
   val contentIntent = Intent(app, MainActivity::class.java)                    1.

   val restartPendingIntent = PendingIntent.getActivity(                        2.
        app,
        NOTIFICATION_ID,
        contentIntent,
        PendingIntent.FLAG_UPDATE_CURRENT
   )

   val notifyBuilder = NotificationCompat.Builder(app, "myChannelID")
        .setSmallIcon(R.drawable.cooked_egg)
        .setContentTitle("My notification")
        .setContentText("Time to work!")
        .setContentIntent(restartPendingIntent)                                 3.
   notificationManager.notify(NOTIFICATION_ID, notifyBuilder.build())}
```

How does our example change now?

- When you launch the Notification, the app finishes
- When you click on the notification, the app is relaunched



Notification priorities

- Determines how the system displays the notification with respect to other notifications

- Use `<mNotifyBuilder>.setPriority()`
  - e.g. `.setPriority(NotificationCompat.PRIORITY_HIGH)`
  - `PRIORITY_MIN` (-2) to `PRIORITY_MAX` (2)

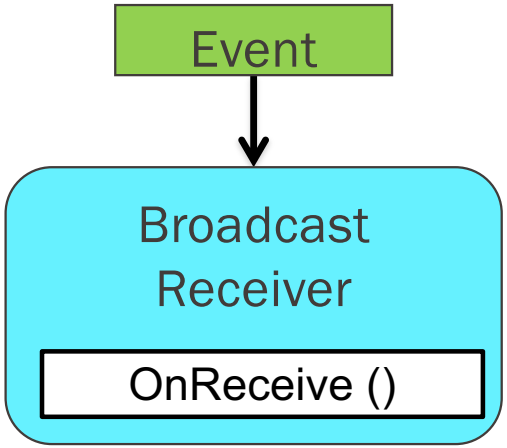
- Priority above 0 triggers heads-up notification on top of current UI
  - Used for important notifications such as phone calls
  - Use lowest priority possible

Broadcast receivers

- Broadcast Receivers respond to events
  - independent from Activity/Composables
  - even when app is closed
- Here, we will see them in conjunction to Alarms
  - Alarm service *setups* alarm event
  - Broadcasts receiver *captures* event
- Other patterns exists
  - Example: System broadcasts: delivered under certain events
    - After the system ends booting:

      `android.intent.action.BOOT_COMPLETED`
    - When the WiFi state changes:

      `android.intent.action.WIFI_STATE_CHANGED`



Creating Broadcast Receivers

1. Create a new class that extends BroadcastReceiver

2. Implement onReceive()
   - Receives the event Intent
   - Handles the event

3. Register the Broadcast Receiver Android Manifest

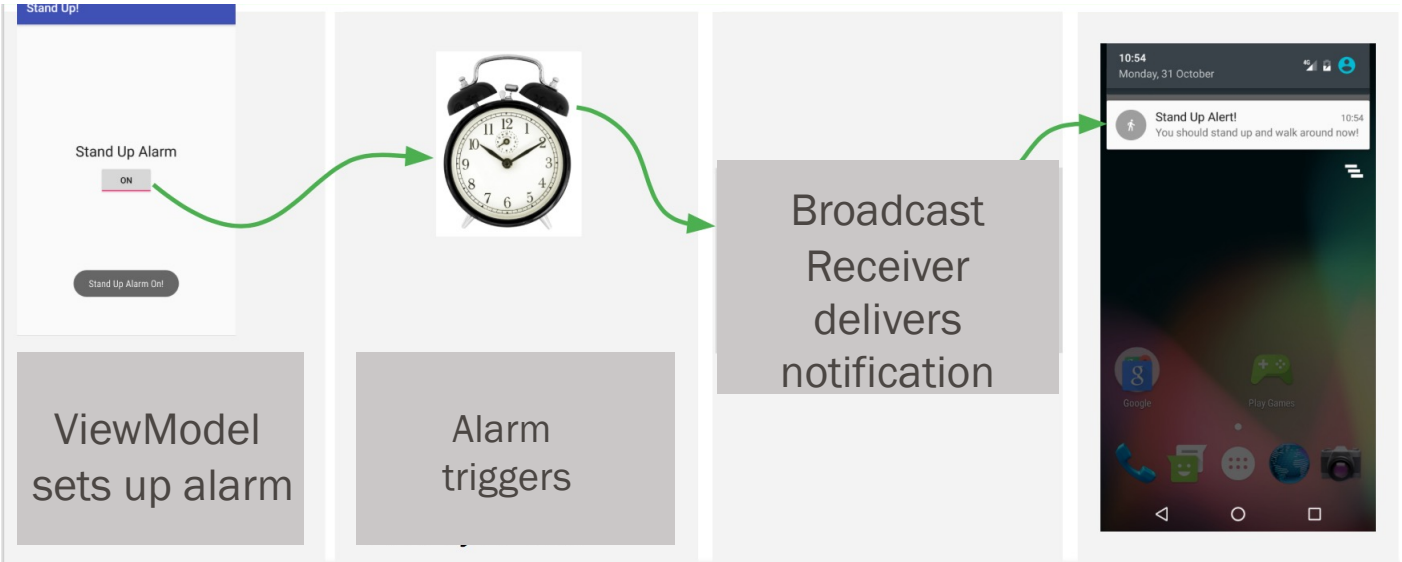
```
class AlarmReceiver: BroadcastReceiver() {

    override fun onReceive(context: Context, intent: Intent) {
        //Do something to deal with the alarm
    }

}
```

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

    <receiver
        android:name=".receiver.AlarmReceiver"
        android:enabled="true"
        android:exported="false">
    </receiver>

</application>
```

Alarms

- An Alarm in Android schedules something to happen at a set time
  - Fire intents at set times or intervals
  - App does not need to run for alarm to be active

→ Use with BroadcastReceiver



Types of Alarms and behaviors

- Measuring time
  - Elapsed Real Time: time since system boot
    - Independent of time zone
    - Used to measure intervals and relative time
    - Elapsed time includes time device was asleep
  - Real Time Clock (RTC): wall clock time
    - When time of day at local time zone matter

- Wake up behavior
  - Wakes up device if screen is off
    - Use only for critical operations

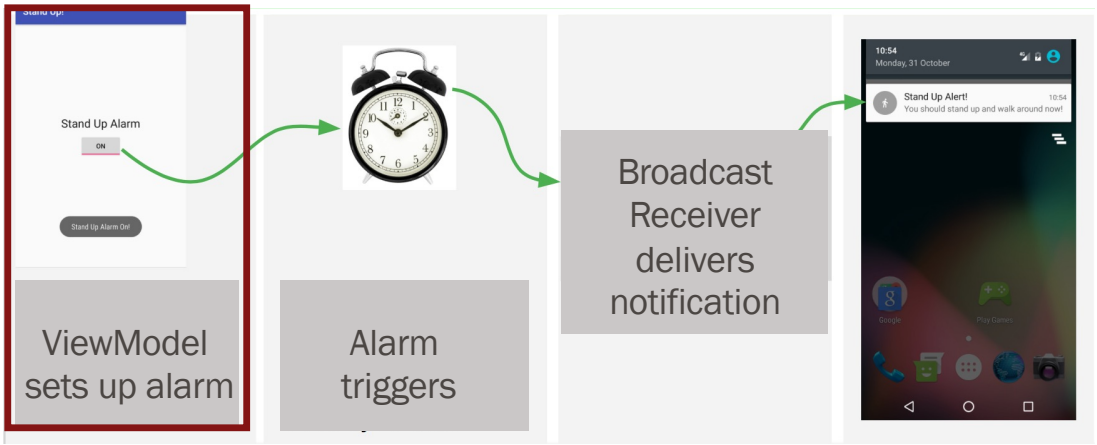
      → drains battery
  - Does not wake up device
    - Fires next time device is awake

	Elapsed Real Time (ERT)—since system boot	Real Time Clock (RTC)—time of day matters
Do not wake up device	ELAPSED_REALTIME	RTC
Wake up	ELAPSED_REALTIME_WAKEUP	RTC_WAKEUP

Setting up an Alarm

1. Create AlarmManager in the ViewModel

```
class myViewModel(private val app: Application) :
                        AndroidViewModel(app) {

    val alarmMgr = app.getSystemService(Context.ALARM_SERVICE)
            as AlarmManager
```

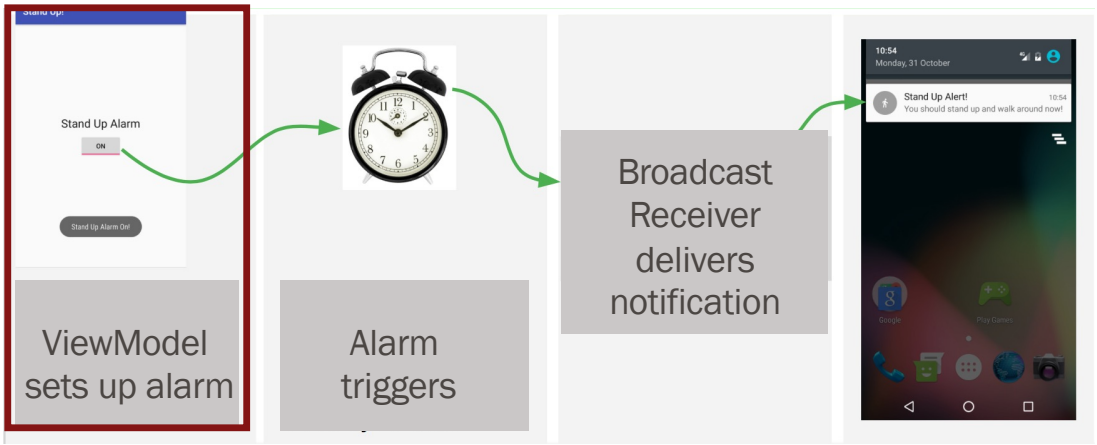


Setting up an Alarm

2. Also in ViewModel, set up a PendingIntent
   - associated to BroadcastReceiver
   - containing Intent processed by onReceive() in BroadcastReceiver

Broadcast Receiver name

```
fun onSetAlarm() {
        val alarmIntent = Intent(app, MyAlarmReceiver::class.java)
        val pd = PendingIntent.getBroadcast(app, REQUEST_CODE, alarmIntent, 0)
        alarmMgr.setExact(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                          SystemClock.elapsedRealtime() + 60 * 1000,
                          pd)
```

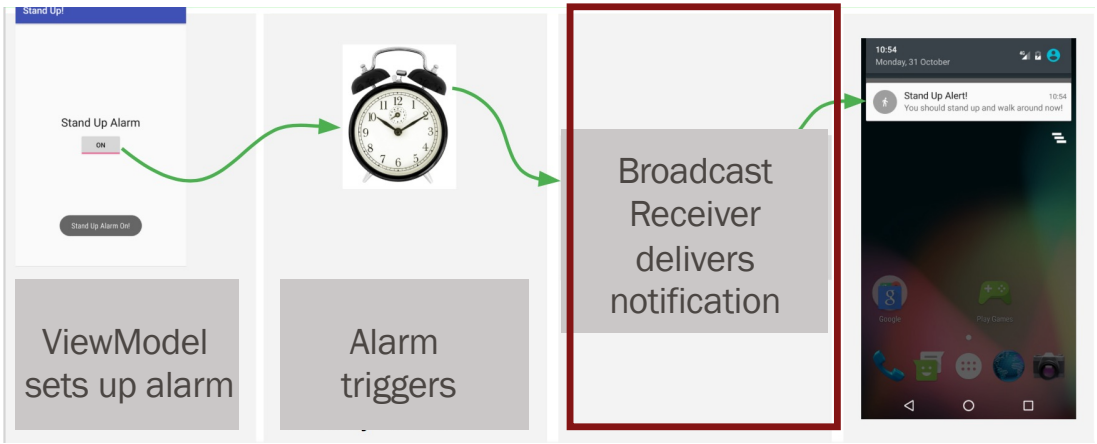


Setting up an Alarm

3. Advertise the Receiver in the Manifest XML file

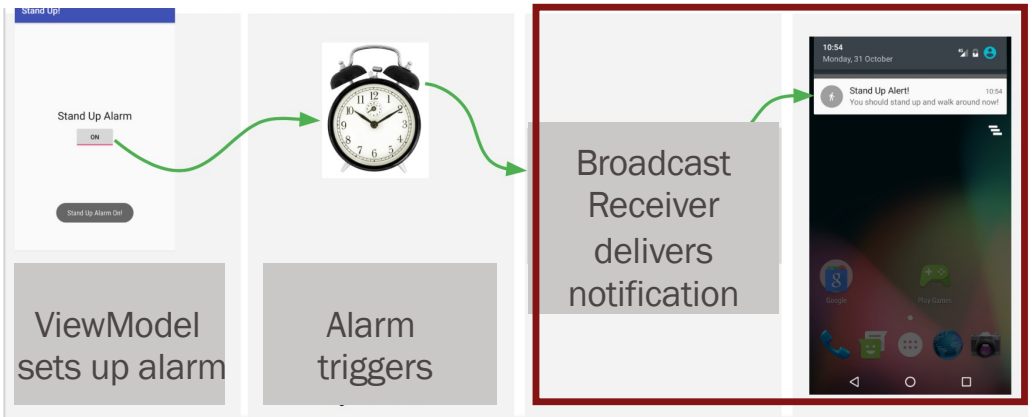
```
<application
  ...
  <activity android:name=".MainActivity">
     ...
  </activity>

  <receiver
    android:name=".MyAlarmReceiver"
    android:enabled="true">
  </receiver>

</application>
```

Broadcast Receiver name



Setting up an Alarm

4. Create a BroadcastReceiver
   - Do something in onReceive()
   - In this example, the broadcast receiver displays a notification when triggered by the alarm

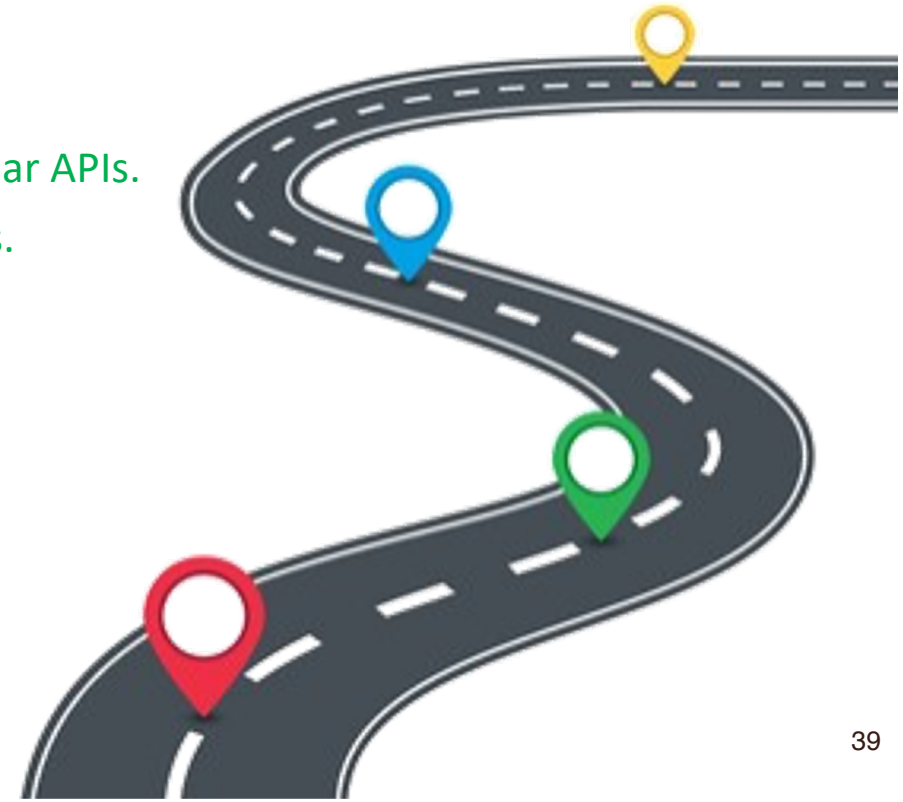



```
class Alarmreceiver : BroadcastReceiver() {

    private val NOTIFICATION_ID = 0

    override fun onReceive(context: Context, intent: Intent) {

        val notificationManager = getSystemService(
                context,
                NotificationManager::class.java
        ) as NotificationManager

        //create channel if
        // Build.VERSION.SDK_INT >= Build.VERSION_CODES.O

        val notifyBuilder = NotificationCompat
                .Builder(context,"myChannelID")
                .setSmallIcon(R.drawable.cooked_egg)
                .setContentTitle("My notification")
                .setContentText("Notification on time!")

        notificationManager
                .notify(NOTIFICATION_ID, notifyBuilder.build())
    }
}
```

Questions?

Where are we?

0. Course presentation. Introduction to Kotlin.
1. Android overview. Defining a GUI.
2. Dynamic applications: State and interactivity.
3. Complex GUIs: Screens and menus.
4. Apps under the hood: Life cycles
   Communication between Android and AndroidWear devices: Wear APIs.
5. Separating concerns: UI controllers and viewModels.
   Interfacing with sensors: System Services.
6. Interfacing with the cloud: Firebase.
   Displaying structured data: Lists.
7. Local databases: Room library.
   Integrating Google maps.
8. Bluetooth Low Energy.



Today's Lab

- Restructure tablet app
  - ViewModels
  - LiveData
  - Observers
- Acquire data from watch HR sensor
- Send HR data to tablet
- HR plot on tablet