



# Android lecture 4

ViewModels/LiveData, Adapters, Broadcasts, Scheduling,  
Services

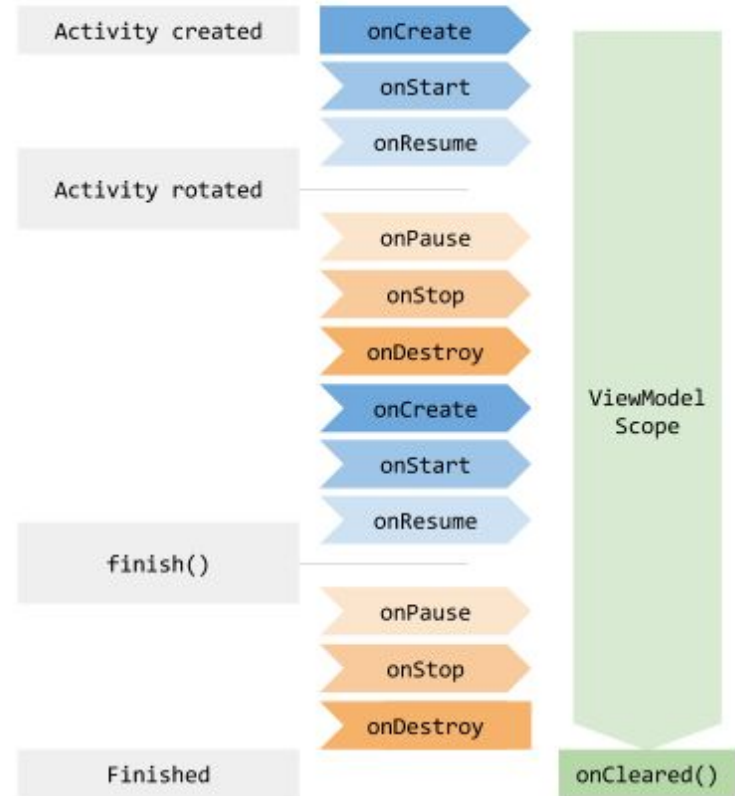
# ViewModel

# ViewModel

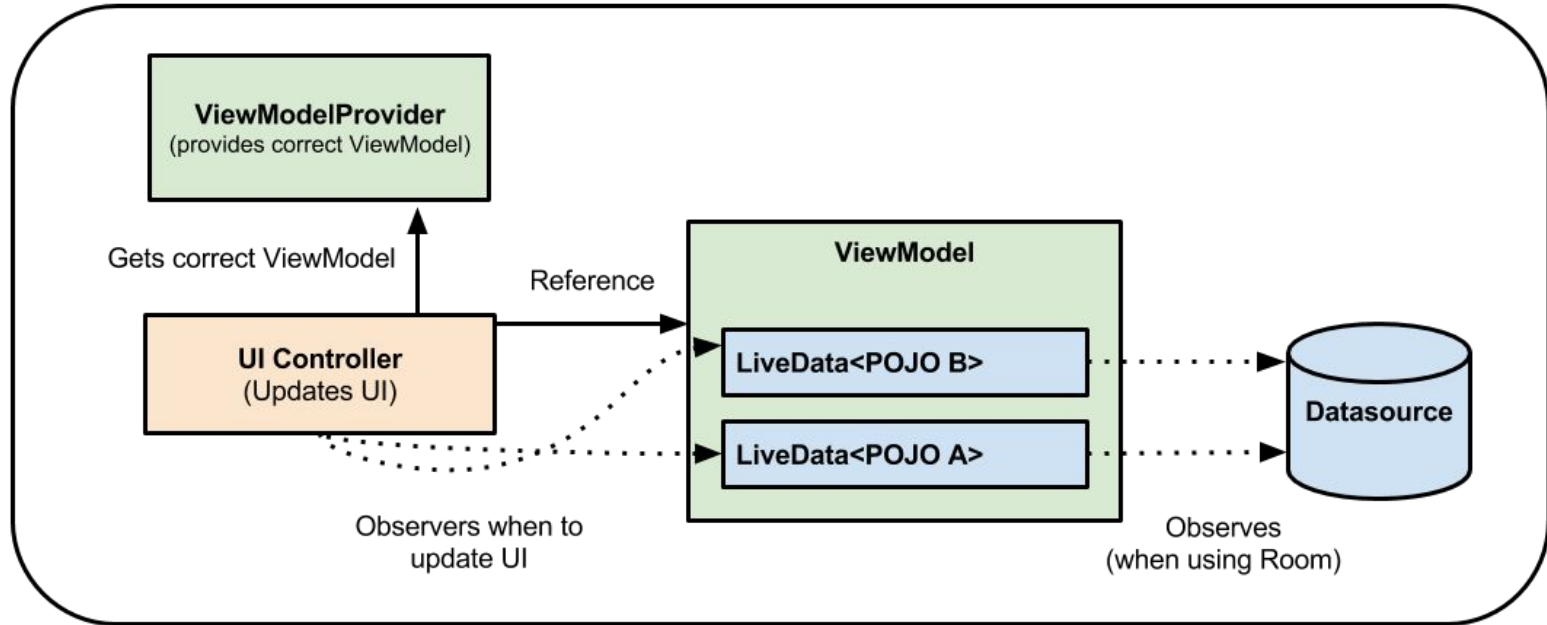
- Prepare data for UI controller (activity, fragment)
- Automatically retained during configuration change

# ViewModel - lifecycle

Illustration of lifecycle when activity is rotated



# ViewModel - architecture



# ViewModel - obtain

```
class DemoViewModel() : ViewModel() {  
    private val _data: MutableLiveData<String> = MutableLiveData()  
    fun getData(): LiveData<String> = _data  
}
```

```
class DemoFragment() : Fragment() {  
  
    val demoViewModel: DemoViewModel by viewModels()  
}
```

# ViewModel

Never hold reference to View, Lifecycle or activity context (even transitively)

# ViewModelProvider.AndroidViewModelFactory

- Creates instances of ViewModel
- Default implementation uses
  - non-param constructor
  - Constructor which accept Application as the only parameter
- Often required with DI framework



# LiveData

- Observable data holder class
- Lifecycle aware
-

```
class DemoFragment() : Fragment() {  
  
    val demoViewModel: DemoViewModel by viewModels()  
  
    override fun onStart() {  
        super.onStart()  
        demoViewModel.getData().observe(viewLifecycleOwner) {data: String ->  
  
        }  
    }  
}
```

# Demo time

Fetching user data using view model

# Adapter views

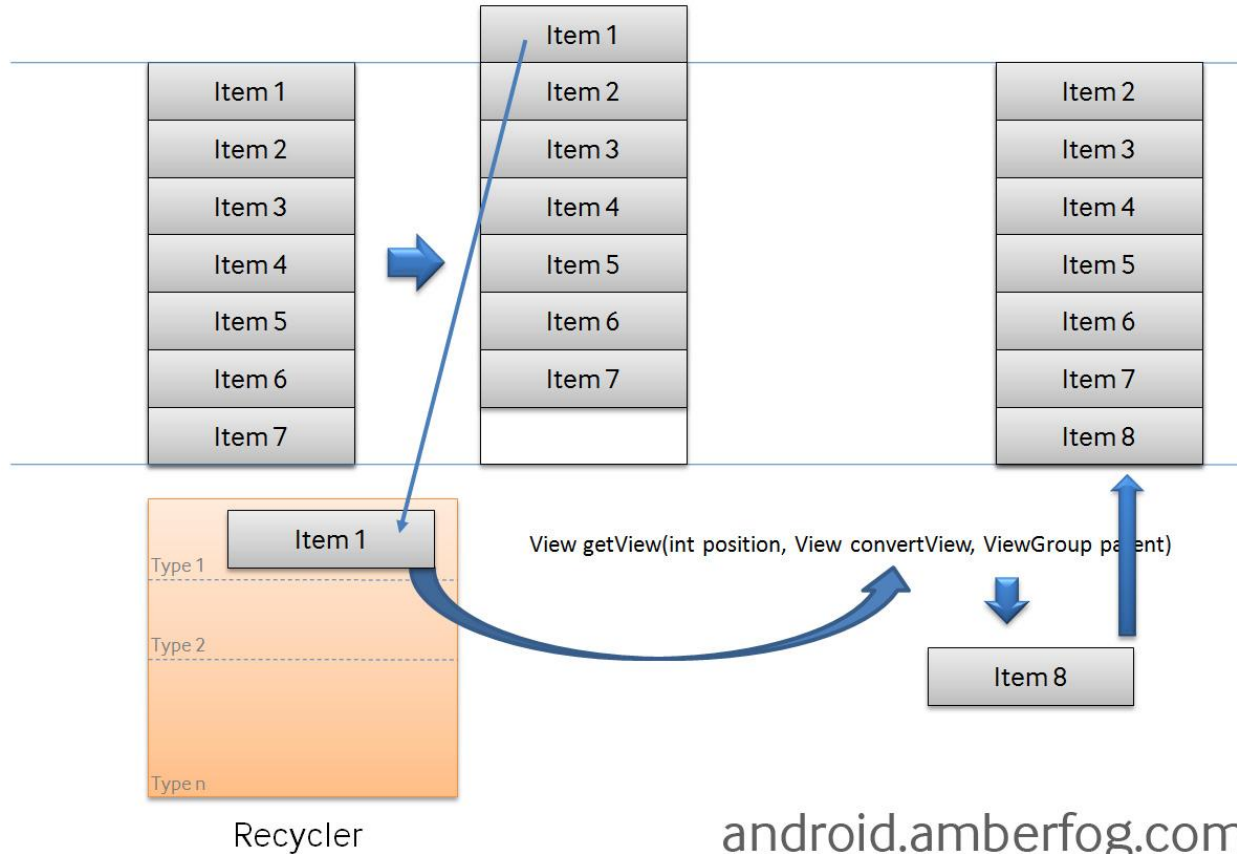
# Adapter views

- Views hold multiple items
- Horizontal scrolling
  - ListView
  - GridView
  - Spinner

# Adapter

- Bridge between data and view
- Responsible for creating view for every item
- For inserting items into ListView, Spinner
- BaseAdapter
  - Common base implementation of adapter
  - `int getCount()`
  - `Object getItem(int position)`
  - `getItemId(int position)`
  - `View getView(int position, View convertView, ViewGroup parent)`
- Subclasses
  - `ArrayAdapter<T>`
  - `CursorAdapter`, `SimpleCursorAdapter`

# View recycling



# ViewHolder pattern

- Remember views
- findViewById is expensive operation
  - Traversing view for complex item
  - Impact on scroll smoothness



# RecyclerView

- AndroidX library
- Uses holder pattern, simplify recycling

# Recycler view - Layout managers

- Measuring and positioning items in list
  - LinearLayoutManager
  - GridLayoutManager
  - StaggeredGridLayoutManager

# Recycler view - ViewHolders

- View caching

# RecyclerView

- `RecyclerView.Adapter<ViewHolderType>`
  - `onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolderType`
  - `getItemCount(): Int`
  - `onBindViewHolder(viewHolder: RepositoryViewHolder, position: Int)`

# Demo time

- Recycler view bind view holder
- Recycler view fill data

# Broadcast receivers

## Intent filters

# IntentFilter

- Intent contains
  - Component name
    - Explicit intent
  - Action
    - Generic action to perform (send email, open web page, ....)
  - Data
    - Uri object that references MIME type of the data
  - Category
    - String with additional information about the kind of component that should handle the intent
  - Extras
    - Key-value pairs with additional data
  - Flags
    - Metadata, for example how the activity is launched

# IntentFilter

- Tells the system, which implicit intent is component able to respond
- Based on
  - Intent action
  - Intent category
  - Intent data

```
<intent-filter>  
  <action android:name="android.intent.action.SEND"/>  
  <category android:name="android.intent.category.DEFAULT"/>  
  <data android:mimeType="text/plain"/>  
</intent-filter>
```



# IntentFilter

- If there is more component which are able respond to the intent, system let user to decide which component/application want to use

# BroadcastReceiver

- Responds to broadcasts
- Broadcasts are system wide messages
  - Use package name prefix
- Registration
  - Static - AndroidManifest.xml
  - Dynamic - in the code at runtime
- By default runs on main thread in default process

# BroadcastReceiver

- Broadcast source
  - System
    - Incoming SMS
    - Incoming call
    - Screen turned off
    - Low battery
    - Removed SD card
  - Our app
- Normal vs ordered broadcasts
- Implicit vs explicit broadcasts

# Normal broadcast

- Asynchronous delivery (multiple receivers can receive intent at the same time)
- Cannot be aborted due to async behaviour
- More efficient

```
Context.sendBroadcast(intent)
```

# Ordered broadcasts

- Delivered to one receiver at a time
- Receiver can abort broadcast, it won't be passed to another receiver
- Order of receiver is controlled by the priority of the matching intent filter

# Implicit vs explicit broadcast

- Implicit
  - System-wide messages
  - [ACTION\\_TIMEZONE\\_CHANGED](#)
  - [ACTION\\_BOOT\\_COMPLETED](#)
  - [ACTION\\_TIME\\_CHANGED](#)
- Explicit
  - Target by class name

# BroadcastReceiver - Registration

- If contains intent filter any app can call the receiver
- Receivers are not enabled until first run of app
- Who can send the broadcast can be limited by permissions

```
<receiver android:enabled=["true" | "false"]  
    android:exported=["true" | "false"]  
    android:icon="drawable resource"  
    android:label="string resource"  
    android:name="string"  
    android:permission="string"  
    android:process="string" >  
    . . .  
</receiver>
```

# BroadcastReceiver - runtime registration

- Without specifying permission any app can send broadcast to you

## Register - Activity.onStart()

```
val intentFilter = IntentFilter()  
intentFilter.addCategory("ACTION_CUSTOM")  
registerReceiver(receiver, intentFilter)
```

## Unregister - Activity.onStop

```
unregisterReceiver(receiver)
```



# BroadcastReceiver.kt

- onReceive must finish in 10 seconds, otherwise ANR
- For longer tasks run service

```
class ExampleReceiver: BroadcastReceiver() {  
  
    override fun onReceive(context: Context, intent: Intent) {  
  
    }  
}
```

# BroadcastReceiver - security

- It is possible to limit who can send broadcast by permissions
- It is possible to protect receiver when it is registered statically and dynamically
- Possible to set permission when sending broadcast

# Broadcast receivers limitations

- Android Nougat API-24
  - Not possible to register for connectivity changes in manifest
- Android Oreo API-26
  - Not possible to register receiver for implicit broadcast in manifest
- <https://developer.android.com/guide/components/broadcast-exceptions>
  - ACTION\_BOOT\_COMPLETED
  - ACTION\_LOCALE\_CHANGED
  - `android.intent.action.TIME_SET`
  - SMS\_RECEIVED\_ACTION
  - ...

# Local broadcasts

```
val lbManager =  
    LocalBroadcastManager.getInstance(this@SplashScreenActivity)  
lbManager.registerReceiver(receiver, intentFilter)  
lbManager.unregisterReceiver(receiver)  
lbManager.sendBroadcast(intent)  
lbManager.sendBroadcastSync(intent)
```

# Scheduling, delayed start

Timer

Handler

AlarmManager

JobScheduler

GCMNetworkManager

WorkManager

# Timer and TimerTask

- Timer allows to run TimerTask in defined time or repeatedly
- Creates new thread where it runs
  - One thread per timer
- For updating UI needs to call `runOnUiThread()`
- Not recommended to use -> Use Handler instead
- Timer can schedule multiple TimerTask
- TimerTask is not reusable

# Timer and TimerTask

```
val delay = 10000L
val period = 10000L
val timer = Timer()

val myTimerTask = object: TimerTask() {

    override fun run() {
        doSomeStuff()
    }
}

timer.schedule(myTimerTask, delay) // run task after delay
```

# Handler

- Possible to run on background or UI thread
- Possible for scheduling or delaying start of some “task”
- In case of device sleep handler doesn’t run
- Messages
  - `sendMessageAtTime(Message msg, long uptimeMillis)`
  - `sendMessageDelayed(Message msg, long delayMillis)`
- Runnable
  - `postAtTime(Runnable r, long uptimeMillis)`
  - `postDelayed(Runnable r, long delayMillis)`
- Good for task with high frequency (more than one in few minutes)
- Tight with application component



# Handler - repeating

```
private fun handlerRepeat() {  
    val runnable = object: Runnable {  
        override fun run() {  
            updateUI()  
            handler.postDelayed(this, 5000L)  
        }  
    }  
    handler.postDelayed(runnable, 5000L)  
}
```

# Alarm manager

- Perform time-based operations outside the application lifecycle
- Fire intents at specified time
- In conjunction with broadcast receivers start services
- Operate outside of your application, trigger events or actions even app is not running or device is asleep
- Minimize app resource requirements
- Action is specified by `PendingIntent`
- Many API changes
  - Added some new method
  - Some method changed behaviour from exact -> inexact
  - READ the documentation carefully

# Alarm manager - tips

- For synchronization consider to use WorkManager
- For repeating sync add some spread when it is syncing
  - Imagine 1M+ of devices trying to download something from your server at the same time
- Use `setInexactRepeating` if it is possible to group alarms from multiple apps => Reduces battery drain
- Alarms are cancelled on reboot, reschedule alarms when device boots

# Alarm manager - alarm type

- ELAPSED\_REALTIME
  - ELAPSED\_REALTIME\_WAKEUP
  - RTC
  - RTC\_WAKEUP
- 
- Clock types
    - Elapsed - time since system boot
      - Use when there is no dependency on timezone
    - Real time clock - time since epoch
      - Use when you need to consider timezone/locale
  - Wake up
    - wakeup - ensure alarm will fire at the scheduled time
    - non wakeup - alarm are fired when device awakes

# AlarmManager - important changes

- API < 19 (KITKAT) - set\* methods behave like exact time
- API >= 19
  - All old methods are inexact now
  - New API for setting exact alarm
    - setExact
  - Added new API for specify windows, when it should be delivered
    - setWindow
- API 21
  - Added methods setAlarmClock and getNextAlarmClock
  - system can show information about alarm
- API 23
  - Added methods setExactAndAllowWhileIdle and setAndAllowWhileIdle
- API 24
  - Added direct callback versions of set and setExact and setWindow

# AlarmManager - usage

```
val alarmManager = getSystemService(Context.ALARM_SERVICE) as AlarmManager

val intent = Intent("AlarmAction")
val pendingIntent = PendingIntent.getBroadcast(applicationContext, ALARM_REQUEST_CODE,
                                              intent,
                                              PendingIntent.FLAG_UPDATE_CURRENT)

alarmManager.set(AlarmManager.RTC,
                System.currentTimeMillis() + TimeUnit.HOURS.toMillis(1L),
                pendingIntent)
```

- AlarmType
- Time
  - Depending on the alarm type it is timestamp or time since device boots
- PendingIntent
  - PendingIntent which specify action which should happen

# Alarm manager - sleeping device

- Alarm manager can wake devices, when it asleep BUT
- pending intent is able to start activity/service or send broadcast
- BUT it is not guaranteed by system to start service/activity before device fall asleep again
- only `BroadcastReceiver.onReceive` is guaranteed to keep device awake
  - If you start activity/service in receiver, there is no guarantee that activity/service will start before the wake lock is released

# Wake locks

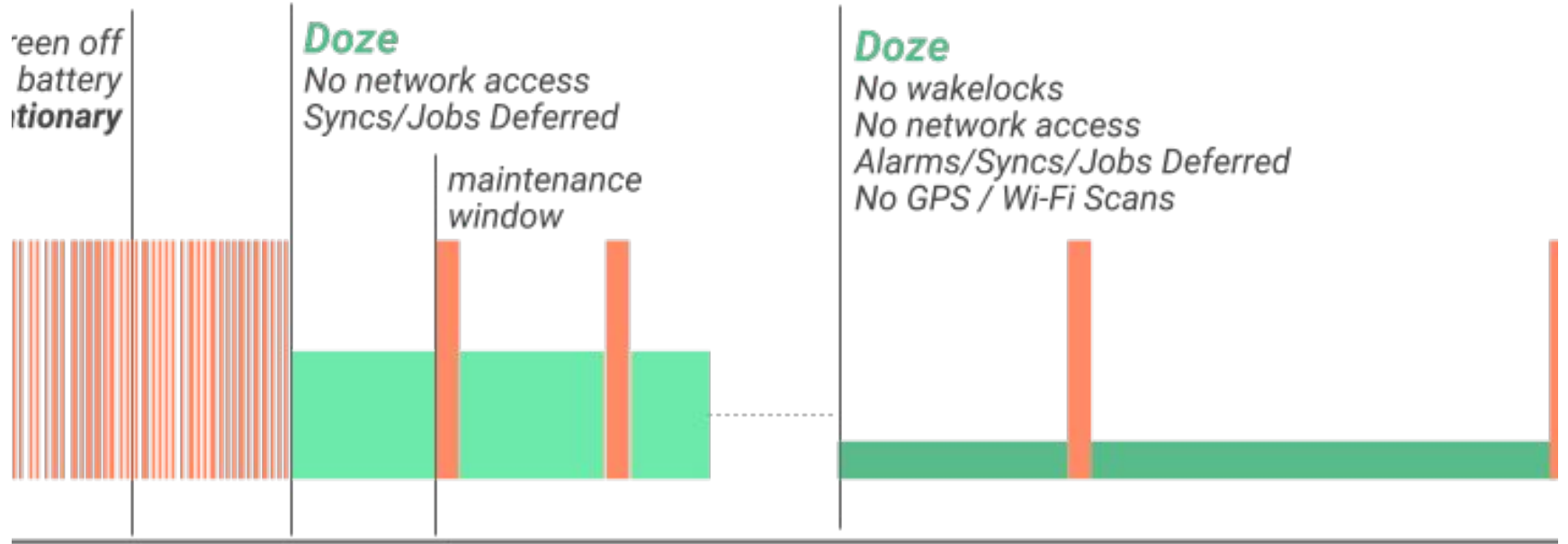
- Prevent device from sleep
- Requires permission `android.permission.WAKE_LOCK`
- Multiple levels
  - `PARTIAL_WAKE_LOCK`
    - CPU is running, screen and keyboard backlight allowed to go off
  - `FULL_WAKE_LOCK`
    - Screen and keyboard on full brightness
    - Released when user press power button
  - `SCREEN_DIM_WAKE_LOCK`
    - Screen is on, but can be dimmed, keyboard backlight allowed to go off
    - Released when user press power button
  - `SCREEN_BRIGHT_WAKE_LOCK`
    - Screen on full brightness, keyboard backlight allowed to go off
    - Released when user press power button



# Alarm manager - sleeping device, solution

- Acquire your wake lock during `BroadcastReceiver.onReceive` and before starting service
- Start service
- When service finish its job release the wake lock
  - It is really important to release wake lock, it disables turning off CPU

# Doze mode



# Doze mode

- Since API 21 (Lollipop)
- Restrict app access to network and cpu intensive services
- Defers jobs, sync and alarms

# Doze mode

- Network access is suspended.
- The system ignores wake locks.
- Standard `AlarmManager` alarms (including `setExact()` and `setWindow()`) are deferred to the next maintenance window.
  - If you need to set alarms that fire while in Doze, use `setAndAllowWhileIdle()` or `setExactAndAllowWhileIdle()`.
  - Alarms set with `setAlarmClock()` continue to fire normally — the system exits Doze shortly before those alarms fire.
- The system does not perform Wi-Fi scans.
- The system does not allow sync adapters to run.
- The system does not allow `JobScheduler` to run.

# Job Scheduler

- Not for exact time schedule
- Possible to specify connectivity, charging, idle conditions
- System batch “jobs”
- Since API 21
- Battery efficient
- Job parameters defined in `JobInfo`
  - Backoff policy
  - Periodic
  - Delay triggers
  - Deadline
  - Persistency
  - Network type
  - Charging
  - Idle

# Job Scheduler

```
val jobScheduler =  
    getSystemService(Context.JOB_SCHEDULER_SERVICE) as JobScheduler  
    val componentName = ComponentName(this, MyJob::class.java)  
  
    jobScheduler.schedule(JobInfo.Builder(1, componentName)  
        .setBackoffCriteria(TimeUnit.MINUTES.toMillis(5L),  
JobInfo.BACKOFF_POLICY_EXPONENTIAL)  
        .setPersisted(true)  
        .setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED)  
        .setRequiresCharging(true)  
        .build())
```

# Job Scheduler

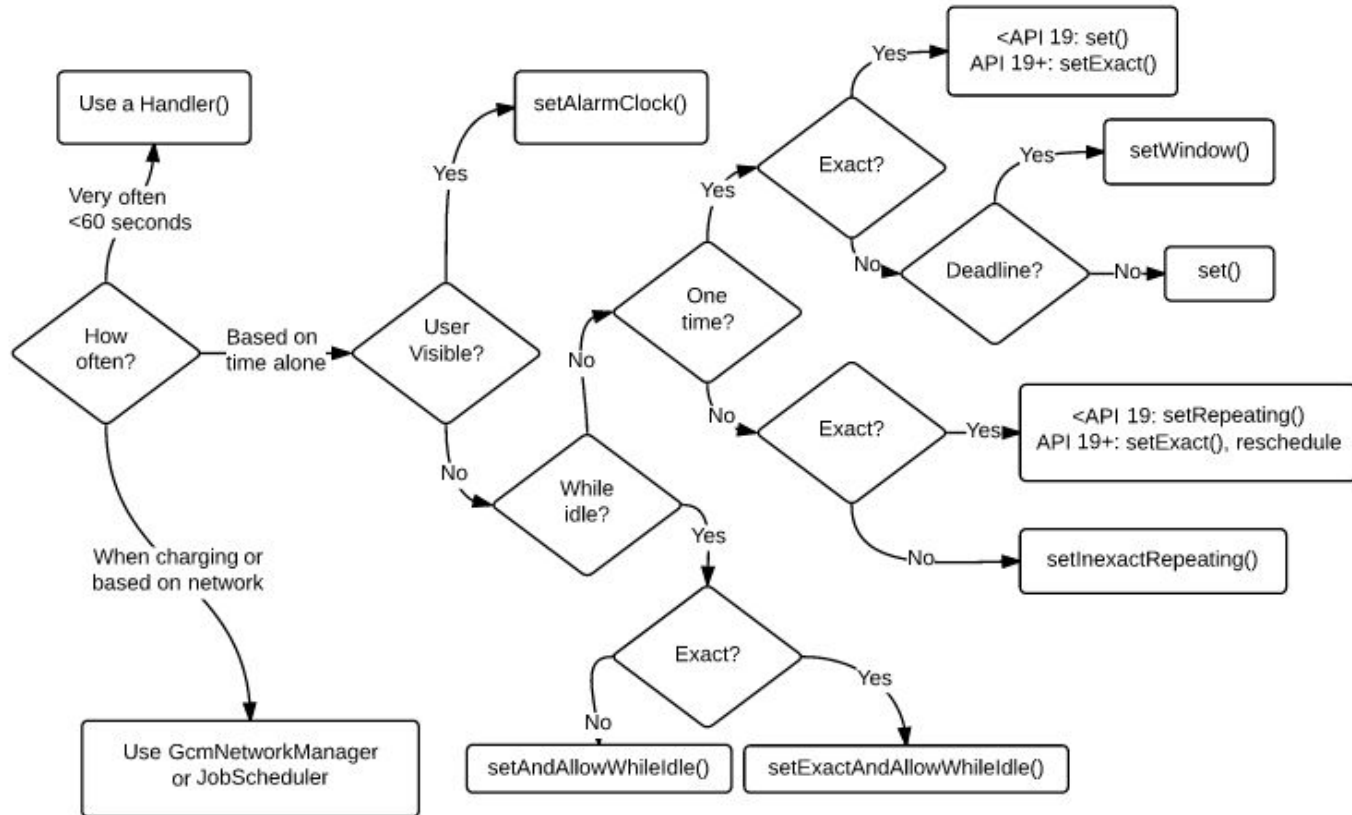
```
class MyJob: JobService() {  
    override fun onStopJob(params: JobParameters?): Boolean {  
        // Do the job  
        jobFinished(params, false)  
  
        return false // no more work to do with this job service  
    }  
  
    override fun onStartJob(params: JobParameters?): Boolean {  
        // do some stuff  
  
        jobFinished(params, false)  
        return false // no more work to do with this job service  
    }  
}
```

# Firebase JobDispatcher

- Part of firebase
- Similar functionality and API as JobScheduler
- Uses JobScheduler on API > 21



# How to decide what to use



OR

# Android-job & workmanager library

~~<http://evernote.github.io/android-job/>~~

Replaced by

<https://developer.android.com/topic/libraries/architecture/workmanager/>

# WorkManager

- Backward compatible up to API 14
- Use JobScheduler on devices with API 23+
- Combination of BroadcastReceiver + AlarmManager API 14-22
- Work constraints
  - Network
  - Charging status
- One-off or periodic
- Monitor and manage scheduled tasks
- Chain tasks
- Ensure execution even if app or device restarts
- Adheres to doze mode

# Work requests

```
// Create a Constraints object that defines when the task should run
val constraints = Constraints.Builder()
    .setRequiresDeviceIdle(true)
    .setRequiresCharging(true)
    .build()

// ...then create a OneTimeWorkRequest that uses those constraints
val compressionWork = OneTimeWorkRequestBuilder<CompressWorker>()
    .setConstraints(constraints)
    .build()
```

# Workers

```
class UploadWorker(appContext: Context, workerParams: WorkerParameters)
    : Worker(appContext, workerParams) {

    override fun doWork(): Result {

        // Get the input
        val imageUriInput = getInputData().getString(Constants.KEY_IMAGE_URI)
        // Do the work
        val response = uploadFile(imageUriInput)

        // Create the output of the work
        val outputData = workDataOf(Constants.KEY_IMAGE_URL to response.imageUrl)

        // Return the output
        return Result.success(outputData)
    }
}
```

# Services

# Services

- Long running operation in background
- Not bound with UI
- Can expose API for other applications
- By default runs on UI thread



# Services

- Types:
  - Started
  - Bound
- Visibility:
  - Background
    - Limited since Oreo (API  $\geq$  26)
  - Foreground

# Started service

- Independent from caller
- Do not return result to caller

# Started service - starting

- Started by calling  
    `Context#startService()`
- Override  
    `Service#onStartCommand()`

# Started service - ending

- Stop by self

`Service#stopSelf()`

- From outside

`Context#stopService()`

# Bound service

- Client server interface for communication
- Lightweight RPC communication

# Bound service - binding

- Component bind to it by calling

```
Context#bindService(service: Intent  
                    conn: ServiceConnection,  
                    flags: Int): Boolean
```

- Override

```
Service#onBind(intent: Intent): IBinder?
```

- Service returns IBinder object for interaction

# Bound services - unbind

- Clients call  
`Context#unbindService(conn: ServiceConnection)`
- System destroys service, when all clients unbond from it

# Service connection

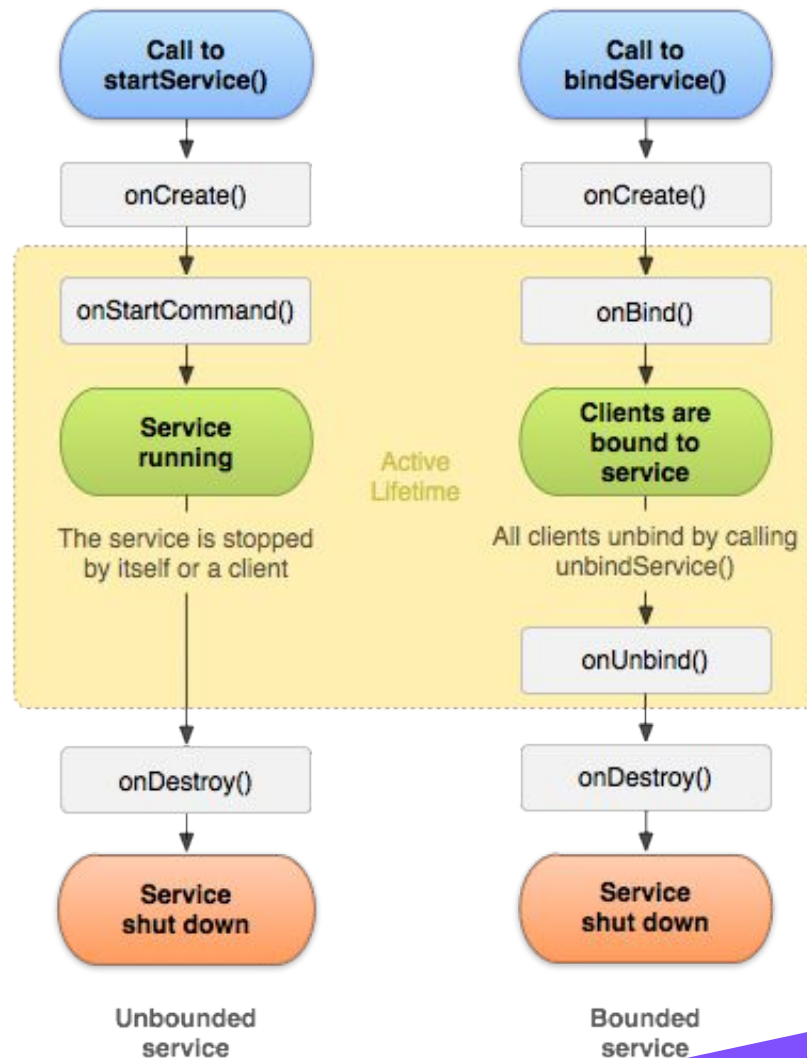
- Define callbacks for service binding
- `fun onBindDied(name: ComponentName)`
  - Binding is dead
  - Can happen during app update
  - Unbind and rebind
- `fun onNullBinding(name: ComponentName)`
  - `Service#onBind` returns null
  - Unbinding is still required
- `fun onServiceConnected(name: ComponentName, service: IBinder)`
  - Connection with the service has been established
- `fun onServiceDisconnected(name: ComponentName)`
  - Connection has been lost
  - Process hosting service crashed or been killed
  - Service connection remain active (`onServiceConnected` can be called again)



# IBinder/Binder

- Remotable object for communication with bounded service
- Can be defined by AIDL

# Service lifecycle



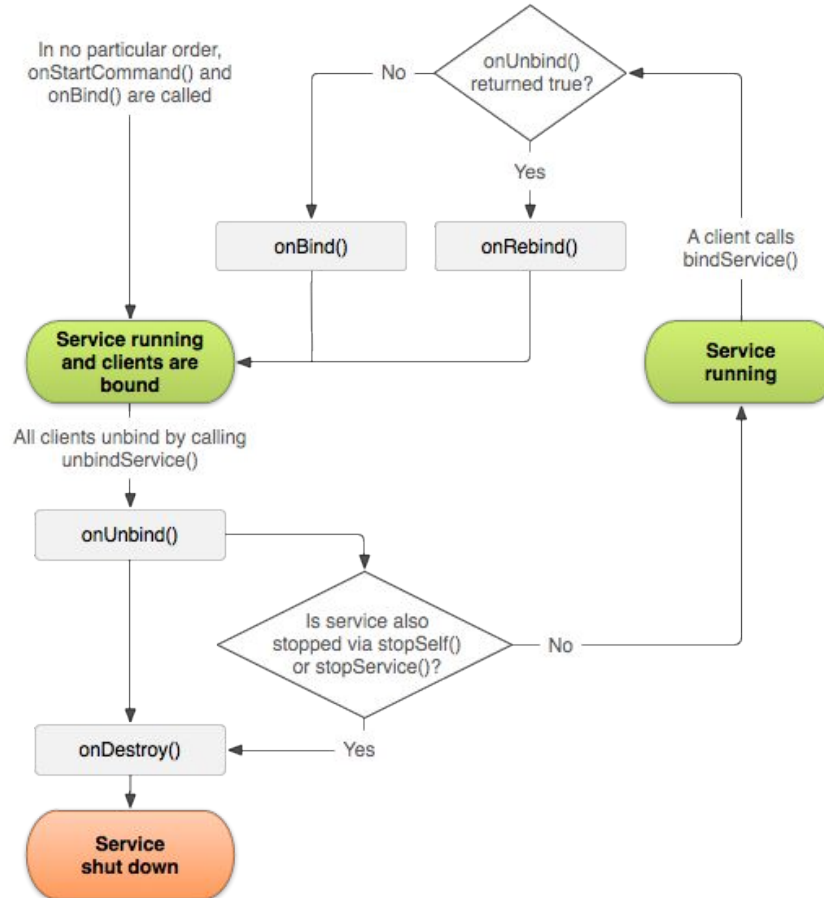
# Service lifecycle

- onCreate()
  - Called when the service is being created (after first call of startService() or bindService())
- onStartCommand()
  - Called when startService() is called, delivers starting intent
  - Returned value specify behaviour when it's killed by system
    - START\_STICKY - don't retain intent, later when system recreate service null intent is delivered (explicitly started/stopped services)
    - START\_NOT\_STICKY - if there is no start intent, take service out of the started state. Service is not recreated.
    - START\_REDELIVER\_INTENT - last delivered intent will be redelivered, pending intent delivered at the point of restart

# Service - lifecycle

- `onBind()`
  - When another component binds to service
  - Returns `Binder` object for communication
- `onUnbind()`
  - When all clients disconnected from interface published by service
  - Returns `true` when `onRebind` should be called when new clients bind to service, otherwise `onBind` will be called
- `onRebind()`
  - Called when new clients are connected, after notification about disconnecting all client in its `onUnbind`
- `onDestroy()`
  - Called by system to notify a `Service` that it is no longer used and is being removed.
  - Cleanup receivers, threads..

# Bound Service lifecycle



# Background service

- On background by default
- Strongly limited since Android Oreo (API 26)
  - Not possible to start background service when app is not on the foreground

# Foreground service

- Service process has higher priority
- User is actively aware of it
- System not likely to kill foreground services
- Requires permanent notification (cannot be dismissed), it is under Ongoing header
- Use `Context#startForegroundService(Intent)`
  - 5s window to make the service foreground
- By calling `Service#startForeground(int, Notification)`
- Remove from foreground `stopForeground()`
- Apps targeting Android 9 (API 28) or higher must define
  - `FOREGROUND_SERVICE` permission (normal permission)

# IntentService

- Subclass of Service
- Uses worker thread to handle requests
- Handle only one request at one time
- Creates work queue
- Stops when it run out of work
- Override `onHandleIntent(Intent)` for processing requests, runs on worker thread



# JobIntentService

- Replacement of IntentService
- Part of support library
- Uses JobScheduler
- Requires WAKE\_LOCK permission



# Thank you Q&A

Feedback is appreciated

[lukas.prokop@avast.com](mailto:lukas.prokop@avast.com)

Please use [mff-android] in subject