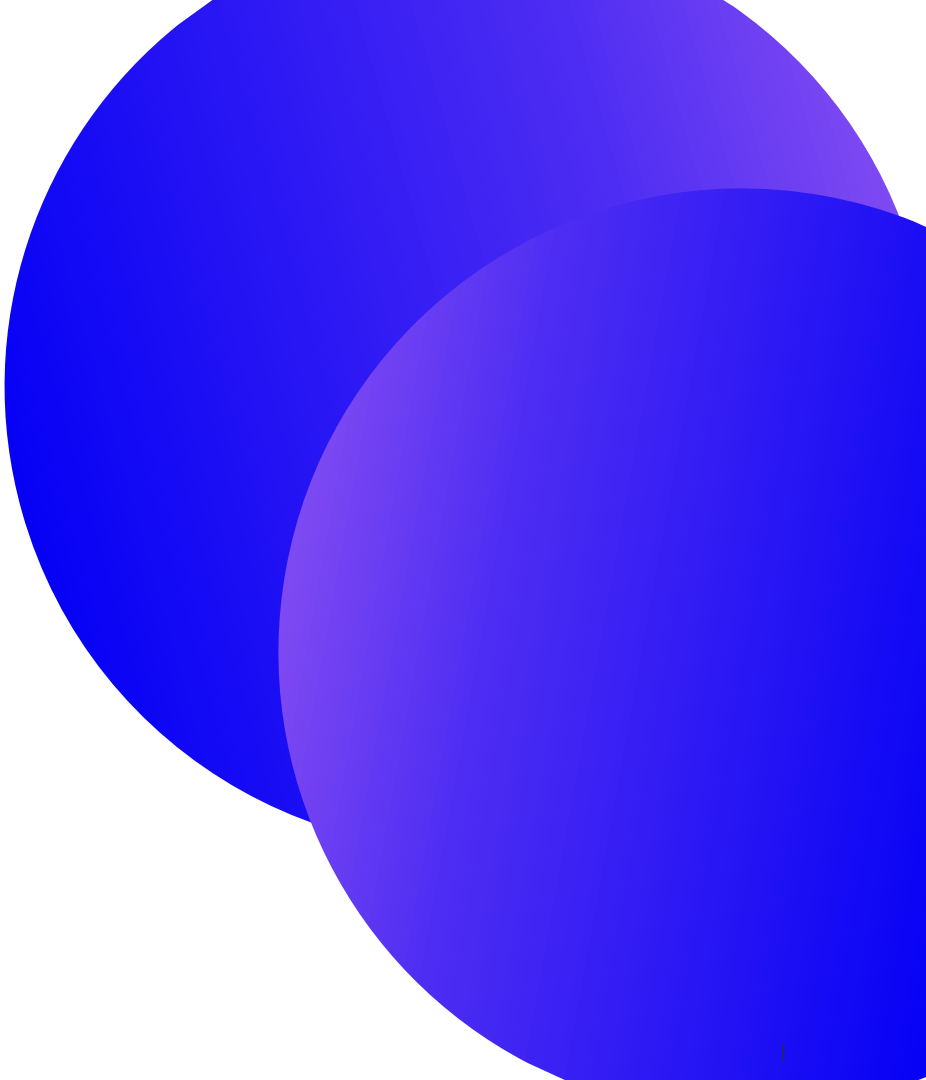
Persistence, Background processing, Adapters, Broadcasts

Android lecture 4

2023

Lukáš Prokop
Simona Kurňavová

Persistence



Persisting data - files

- Standard Java API for file operations

Internal storage

- Always available
- For private data
- Removed with application uninstall
  - https://medium.com/inloopx/samsung-tablets-are-not-removing-application-files-after-uninstall-45cc22ace56a
- Cache

Internal storage

- **`Context.getFilesDir()`**
  - File representing internal directory for your app
- **`Context.openFileOutput(filename: String, mode: Int)`**
  - Filename - name of file
  - Mode - specify access to file
    - `MODE_PRIVATE` - accessible by apps with same UID
    - `MODE_APPEND` - append data instead of erasing file
    - `MODE_WORLD_READABLE` - Deprecated API 17, SecurityException API 24
    - `MODE_WORLD_WRITEABLE` - Deprecated API 17, SecurityException API 24
- **`Context.openFileInput(filename: String)`**
  - Filename - name of file

Internal storage - cache

- `Context.getCacheDir()`
  - File representing internal directory for app temporary files
  - System can delete these files, when is running low on storage
  - 3rd party cleaner apps often clear cache
  - Delete these files when are not longer needed
  - Presence of these files should not affect your application
    - It can just slow down app, need to download some resources

Internal storage - sharing data

- **Data can be shared via `FileProvider`**
  - Allows to specify shared directories
  - Implicit intent to pick specific files

External storage

- External storage != SD Card
- Not always available
- World readable
- Uninstall remove files in `Context.getExternalFilesDir()`
- Lot of API changes between android versions
- Often modified by vendors

External storage

- **Requires permissions**
  - `android.permission.WRITE_EXTERNAL_STORAGE`
  - `android.permission.READ_EXTERNAL_STORAGE`
  - Since API 19 permissions are not needed for private files
- **Developer responsibility to check if the external storage is available**
- **Public files**
  - Available to the other apps and user
  - Downloaded files
- **Private files**
  - Files to be deleted with app uninstall
  - Accessible to other, but no value for them
  - Temp downloaded files, ringtones, ….

External storage

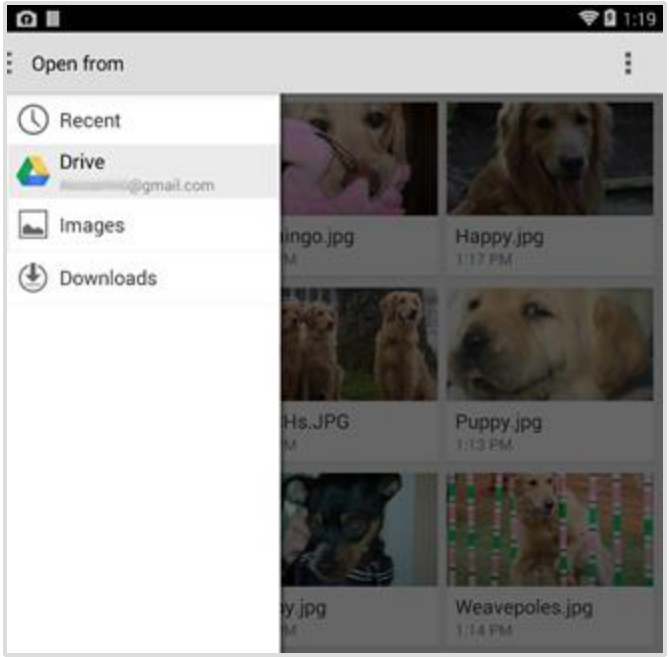
- **`Environment.getExternalStoragePublicDirectory(type: String): File`**
  - Type - type of files to access `Environment.DIRECTORY_*`
  - File representing top-level shared/external directory for files of particular type
  - Multi user devices - access only to current user
- **`Environment.getExternalFilesDir(type: String): File`**
  - Type - type of files to access `Environment.DIRECTORY_*`
  - `File` representing where app place internal files
  - Files are deleted after app uninstall
- **`Environment.isExternalStorageEmulated(): Boolean`**
- **`Environment.isExternalStorageRemovable(): Boolean`**
- **`Environment.getExternalStorageState(): String`**

External storage - SD card

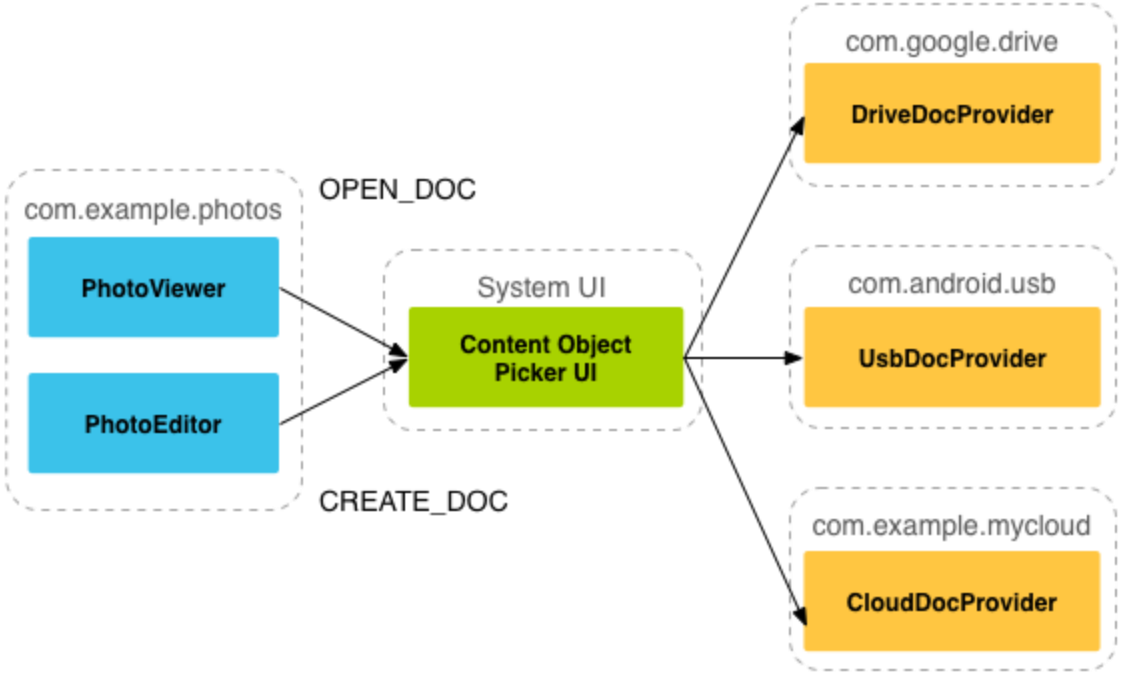
- **< API-19 guess where the sdcard is mounted**
- **= API-19 not possible write shared data on sd card, when primary external storage is available**
  - Or using storage access framework, but access is granted per file
- **>=API-21 Storage access framework allows to grant access for directories**
  - New APIs for accessing media folders on SD card
  - Context.getExternalMediaDirs(): Array<File>

Storage access framework

- Let user pick a file
- Allows to plug-in custom service (cloud services like Dropbox, Google drive, …)
- Since API 19



Storage access framework



Scoped storage - Android 10 (API-29)

- **Restrict access to files on external storage**
- **App has access to app-specific directory**
- **Files are in collections - no permission needed for contribution**
  - Pictures
  - Videos
  - Music/Audio
  - Download
- **Modify/Delete files created by other app requires explicit user consent**

Scoped storage - Android 10 (API-29)

- Access to documents, downloaded file use storage access framework
- Read/write outside of the collections requires storage access framework
- Photo location metadata permission

Scoped storage - Android 10 (API-29)

- Possible to opt-out by AndroidManifest.xml flag

Scoped storage - Android 11 (API-30)

- **Files API for files accessible through MediaStore API**
  - 3rd party libraries, C/C++ code
- **API for bulk options over media files**
- **Special app access for selected use case**
  - Antivirus
  - Backup&Restore
  - File managers
  - ...
  - Manually reviewed by google
  - User have to explicitly grant the access in Android settings
- **Managed by TargetSdk**

Scoped storage

https://developer.android.com/training/data-storage/use-cases

SharedPreferences

- **Key value storage**
- **Backed by XML**
- **`Context.getSharedPreferences(name: String, mode: Int)`**
  - Name - name of file with preferences
  - Mode - operating mode
    - `MODE_PRIVATE` - only apps with same UID have access
    - `MODE_WORLD_READABLE` - API 17 Deprecated, API 24 SecurityException
    - `MODE_WORLD_WRITEABLE` - API 17 Deprecated, API 24 SecurityException
- **`Activity.getPreferences(int mode)`**
  - Preferences associated with activity
- **`PreferenceManager.getDefaultSharedPreferences (Context)`**
  - Default preferences used by Preference framework

SharedPreferences

```
val sharedPrefs = getSharedPreferences("preferences", Context.MODE_PRIVATE)
val intVal = sharedPrefs.getInt("int_key", 42)
val stringVal = sharedPrefs.getString("string_key", "Default")
```

```
val editor = sharedPrefs.edit()
editor.putString("string_key", "new value")
editor.commit() //Synchronous
editor.apply()  // Async
```

SharedPreferences

- **Editor.commit()**
  - Notifies about result
  - Synchronous operation, waits until changes are written to disk
- **Editor.apply()**
  - Async variant
  - Atomically stores values
  - ANRs bugs (fsync() on main thread)
- **If multiple editors modifying preferences at she same time, last calling apply() wins**
- **Debugging - rooted device or flipper/stetho**

http://facebook.github.io/stetho/ - old, not maintained
https://fbflipper.com/ - new, multiplatform

Data store

- Shared preferences replacement
- Shared preferences async API has some design flaws https://engineering.avast.io/how-we-fought-with-anr-rate-in-android-vitals/

Data store

Feature	SharedPreferences	Preferences DataStore	Proto DataStore
Async API	✅ (only for reading changed values, via listener)	✅ (via `Flow`)	✅ (via `Flow`)
Synchronous API	✅ (but not safe to call on UI thread)	❌	❌
Safe to call on UI thread	❌*	✅ (work is moved to `Dispatchers.IO` under the hood)	✅ (work is moved to `Dispatchers.IO` under the hood)
Can signal errors	❌	✅	✅
Safe from runtime exceptions	❌**	✅	✅
Has a transactional API with strong consistency guarantees	❌	✅	✅
Handles data migration	❌	✅ (from SharedPreferences)	✅ (from SharedPreferences)
Type safety	❌	❌	✅ with Protocol Buffers

https://android-developers.googleblog.com/2020/09/prefer-storing-data-with-jetpack.html

Data store

- Completely asynchronous approach
- RxJava/Kotlin coroutines API
- Google protocol buffer type safe API
- API for migration from shared preferences

Exercise

6. Count app launches
7. Prefill login with last used one

Oh wait I/O operations needs to happen on backround thread

“Some people, when confronted with a problem, think, “I know, I’ll use threads,” and then they have two problems.”

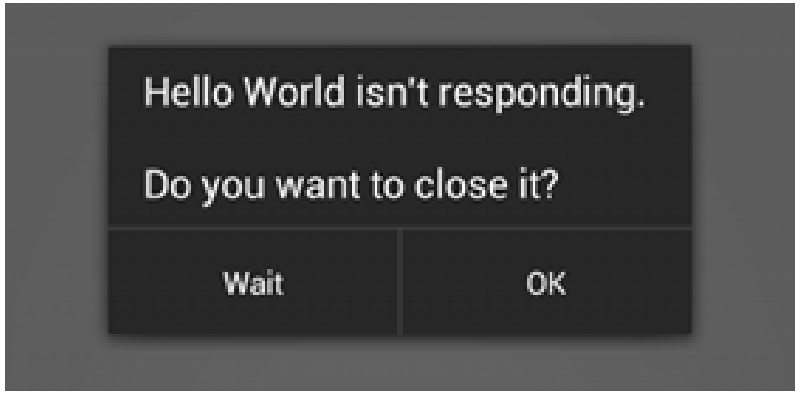
Background processing

https://nedbatchelder.com/blog/201204/two_problems.html

Background processing

- Threads
- Handler
- ~~AsyncTask~~ - Deprecation in Android 11 (API-30)
- ~~Loader~~ deprecated Android 9 (API-28)
- Kotlin coroutines
- RxJava

Motivation



Keep your application responsive

Background processing

- **Avoid long running operations on Main/UI thread**
  - Files, database, network
- **Most component runs on Main thread by default**
- **5 second to ANR (10s BroadcastReceiver)**

Background processing

- Main thread = UI thread
- Never block UI thread

Background processing - issues

- Activities can be restarted
- Memory leaks
- Crashes

Thread

- **java.lang.Thread**

```
Thread() {
    override fun run() {
        // Long running operation
            . . .
    }
}.start()
```

- **Standard java thread**
- **Simple way how to offload work to the background**
- **UI can’t be updated from background**

Handler

- **android.os.Handler**
- **Sends and processes messages**
- **Instance is bound to thread/message queue of the thread creating it**
  - Scheduling messages and Runnables to be executed at some point in future
  - Enqueue an action to be performed on different thread

Handler

Receiving message on UI thread

- Overriding `handleMessage(Message)`

```
val handler = object : Handler() {
    override fun handleMessage(msg: Message?) {
        Txt_username_value.text =
                msg.data.getString("data_key")
    }
}
```

Send message from background

- Obtain message is more effective than create new instance
- Requires reference to handler

```
val message = handler.obtainMessage()
message.arg1 = 1001
handler.sendMessage(message)
```

Looper and Handler

- **Looper**
  - Class that runs a message loop for a thread
  - UI thread has its own Looper
    - Looper.getMainLooper()
    - Keeps UI thread alive
  - Other threads by default do not have Looper associated with them
- **Handler**
  - Provides interaction with the message loop

HandlerThread

- Holds a queue of task
- Other task can push task to it
- The thread processes its queue, one task after another
- When queue is empty, it blocks until something appears

Async task - DEPRECATED

- **android.os.AsyncTask**
- **Simplify running code on background**
- **`AsyncTask<Params, Progress, Result>`**
  - Params - The type of the parameters sent to the task upon execution
  - Progress - type of progress unit published during background operation
  - Result - type of result of background operation

AsyncTask - methods

- **`onPreExecute()`**
  - UI thread, before executing, show progress bar
- **`doInBackground(Params…)`**
  - Background thread
  - `publishProgress(Progress…)`
  - Returns Result
- **`onProgressUpdate(Progress…)`**
  - UI thread
  - For updating progress, params are values passed in `publishProgress`
- **`onPostExecute(Result)`**
  - UI thread
  - Returned value from `doInBackground` is passed as parameter

AsyncTask - canceling

- `cancel(boolean)` - Cancel execution of task
- `isCancelled()` - call often in `doInBackground` to stop background processing as quick as possible
- `onCancelled(Result)` - called instead of `onPostExecute()` in case task was cancelled

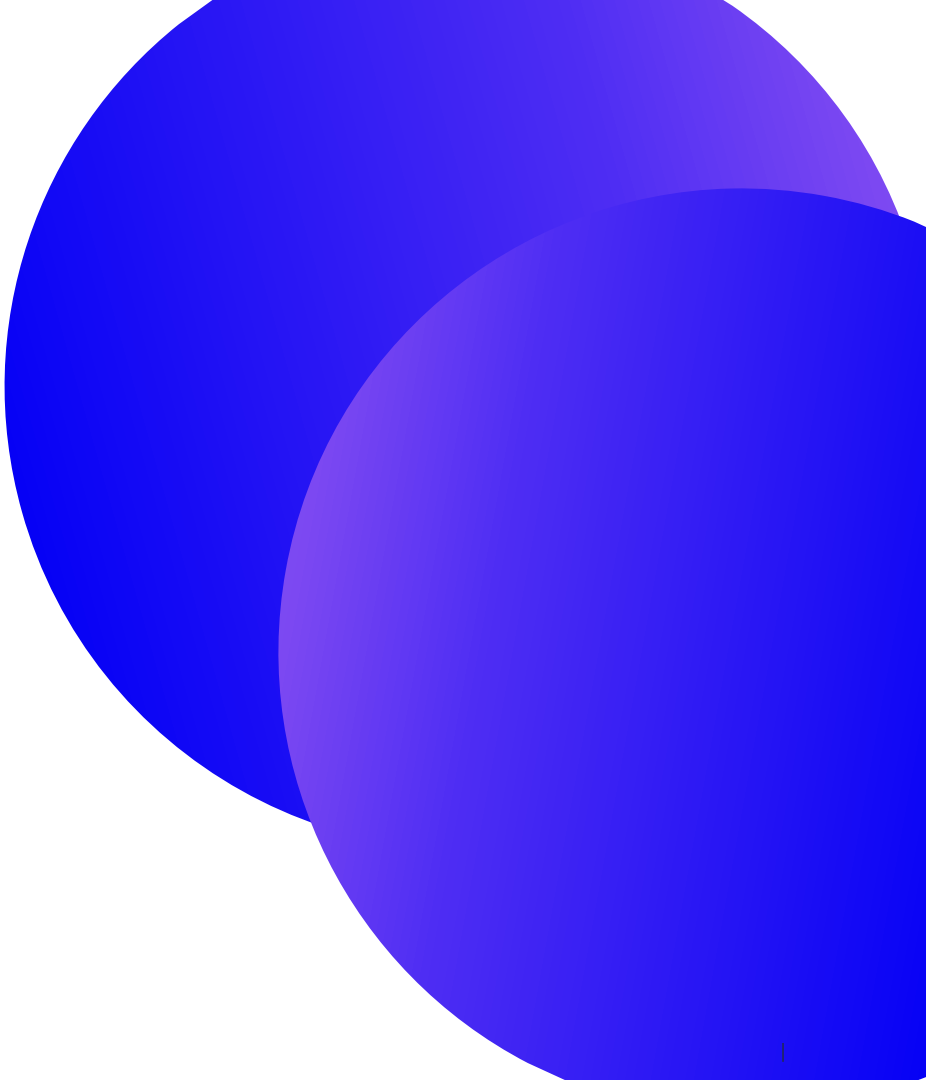
Memory leaks

- Activity runs AsyncTask which takes long time, meanwhile configuration change happens
- Anonymous and non-static inner class still keeps reference to Activity => Activity can’t be garbage collected => activity leaks

Memory leaks - Solutions

- **Disable configuration changes in manifest**
  - Don’t do this, it just hides another bugs
- ~~**Retain activity instance**~~
  - ~~Using `onRetainNonConfigurationInstance()` and `getLastNonConfigurationInstance()` deprecated~~
- **WeakReference to activity/fragment or views**
- **Task as static inner class**
- ~~**TaskFragment** deprecated~~
  - Fragment without UI and called `setRetainInstance(true)`
- ~~**AsyncTaskLoader**~~
- **ViewModel + LiveData**

Kotlin coroutines intro



Kotlin coroutines

- Added in Kotlin 1.3
- **Lightweight thread**
- Creating a new coroutine is cheap and efficient (unlike creation of thread)
- **Uses suspending functions**
- **Jetpack integration:** many libraries provide support of coroutines: custom scopes, extension functions...

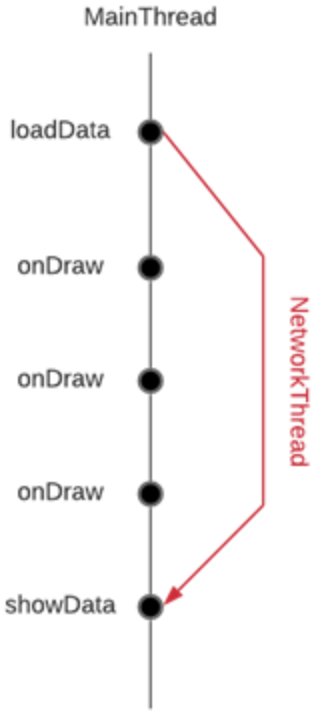
Suspend function

- **Function which is able to suspend it's execution without blocking thread**

```
suspend fun loadData() { delay(10_000) }
```

- **Suspend lambda**
- **Suspend/resume**
  - Internally pass Continuation object as callback
  - Uses finite state machine under the hood
- **Can be called only from other suspend function or in coroutine created by coroutine builder**

- **suspend does not mean to run function on background**



Suspend function

Suspend functions should be main-safe

CoroutineDispatcher

- All coroutines run in dispatcher
- Dispatcher is responsible for managing the execution of coroutines on a thread / set of threads
- Coroutines can suspend themselves
- Knows how to resume suspended coroutines
- Part of coroutine context

CoroutineDispatcher

Dispatchers.Main	**Dispatchers.IO**	**Dispatchers.Default**
Main thread, interact with UI, light work	Disk and network IO off the main thread	CPU intensive work
• Call suspend functions • Call UI functions • Updating LiveData	• Database • R/W files • Networking	• Sorting list • Parsing JSON • DiffUtils

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

CoroutineDispatcher

```
override suspend fun getUser(username: String): User? = withContext(Dispatchers.IO) {
    return@withContext GithubServiceFactory.githubService.getUser(username).body()
}
```

Coroutine scope

- **Keep track of all coroutines running inside**
- **Not possible to start coroutine outside of some scope**
- **If scope cancels, coroutines cancels**
- **GlobalScope - lifetime of whole application**
- **`ViewModel.viewModelScope` - extension property**
  - Cancel coroutines started by current view model when it is cleared

```
class UserViewModel: ViewModel() {

    fun fetchData(username: String) {
        viewModelScope.launch {
            state.value = LoadInProgress
            fetchDataSuspend(username)
        }
    }
}
```

Coroutine scope

Avoid leaking coroutines

Coroutine scope builders

- **Creates new coroutine scope inside current one**
- **Cancellation is propagated from parent to children's**
- **For parallel work decomposition**
- **`coroutineScope` vs. `supervisorScope`**
  - coroutineScope - cancels if any of its children fail
  - supervisorScope - still run if some children fail
  - Suspends until coroutines complete

Coroutines - starting

- **launch**
  - Fire and forget - do not return result to caller
  - Usually bridge from regular function into coroutines
  - Return Job for cancellation
  - Do not block current thread
- **async/await**
  - Start computation asynchronously
  - Creates coroutine and return it’s future result (Deferred)
  - Await - wait until coroutine finishes and return result to the caller
  - Thrown exceptions are not signaled until await is called
- **runBlocking**
  - Blocks until coroutine finishes
  - Handy for initial refactoring

Exercise

1. Count app launches
2. Prefill login with last used one

Storage - continue

Database -SQLite

- Full-featured SQL
- Single-file database
- Source code is just 1 file
- Small footprint
- ACID transactions
- Well documented
- Supports most of the SQL92 standard

SQLite on Android

- **Foreign keys disabled by default**
- **Internal storage**
- **Collation**
  - BINARY - SQLite default
  - LOCALIZED - changes with system locale
  - UNICODE - Unicode collation algorithm
- **Thread safe**
- **Create/upgrade on background thread**
- **Take care about opening/closing from different threads**
- **Use `BaseColumn._ID` for primary keys, some components rely on it**
- **Stetho tool for debugging**

Database

- **`android.database.sqlite.SQLiteOpenHelper`**
  - Database creation
  - Version management
  - Sqlite configuration
    - Enable write ahead log
    - Enable support for foreign keys
- **`android.database.sqlite.SQLiteDatabase`**
  - Exposes methods to manage a SQLite databases
  - CRUD methods
  - Manage transactions

SQLiteOpenHelper

- `onCreate(db: SQLiteDatabase)`
  - Called when the database is created for the first time
- `onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int)`
  - Upgrade logic
- `getReadableDatabase/getWriteableDatabase`
  - creates/open database
- `close()`
  - Close open database object

SQLiteDatabase

- `insert(table: String, nullColumnHack: String, values: ContentValues)`
  - Table - name of table
  - nullColumnHack - optional, allows to insert empty row
  - Values - inserted values
  - Returns ID of newly inserted row
- `long insertOrThrow`
- `long insertWithOnConflict`

SQLiteDatabase

- `query(boolean distinct,`
  `table: String,`
  `columns: Array<String>,`
  `selection: String,`
  `selectionArgs: Array<String>,`
  `groupBy: String,`
  `having: String,`
  `orderBy: String,`
  `limit: String): Cursor`
  - `Selection` - WHERE clausule, values replaced by ?
  - `selectionArgs` - values to replace ? in selection
- **Multiple variants of query, with different possibilities**
- `rawQuery(sql: String, selectionArgs: Array<String>): Cursor`
- **Close returned cursors**

SQLiteDatabase

- `update(table: String,`
  `values: ContentValues,`
  `whereClause: String,`
  `whereArgs: Array<String>): Int`

SQLiteDatabase

- ```
  delete(table: String,
         values: ContentValues,
         whereClause: String,
         whereArgs: Array<String>): Int
  ```
  ```

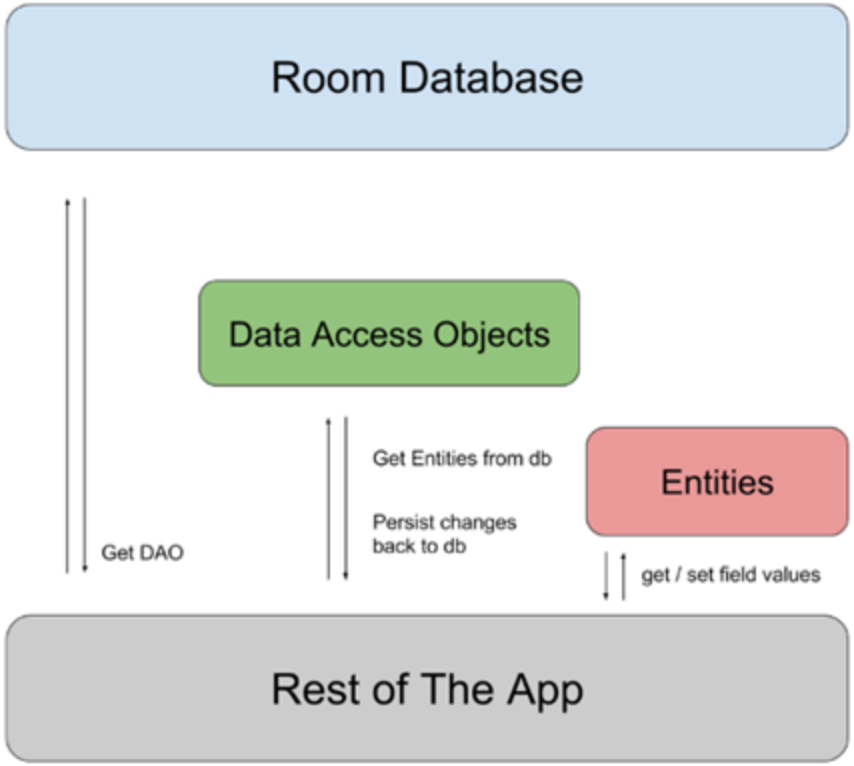
SQLiteDatabase

- Every CRUD operation is a transaction
- For inserting more rows in one time use transactions
- `beginTransaction()`
- `endTransaction()`
- `setTransactionSuccessful()`

Room

- Part of the Android Jetpack
- Abstraction over SQLite
- Compile time validation of SQL queries
- Full integration with other Architecture components (LiveData, LifecycleObserver)
- RxJava bindings

Room


Room Database
Data Access Objects
Get DAO
Get Entities from db
Persist changes back to db
Entities
get / set field values
Rest of The App

Room - entities

- Represents a table

```
@Entity
data class Car(
    @PrimaryKey val id: Int,
    @ColumnInfo(name = "manufacturer") val manufacturer: String?,
    @ColumnInfo(name = "model") val model: String?,
    @ColumnInfo(name = "nubmer_of_wheels") val numberOfWheels: String?
)
```

Room - DAO

- Defines operations on top of entities

```
@Dao
interface CarDao {
    @Query("SELECT * FROM car")
    fun getAll(): List<Car>

    @Query("SELECT * FROM car WHERE id IN (:carIds)")
    fun loadAllByIds(carIds: IntArray): List<Car>

    @Query("SELECT * FROM car WHERE manufacturer LIKE :manufacturer AND " +
           "model LIKE :model LIMIT 1")
    fun findByModel(manufacturer: String, model: String): Car

    @Insert
    fun insertAll(vararg cars: Car)

    @Delete
    fun delete(car: Car)
}
```

Room database

- Defines database

```
@Database(entities = arrayOf(Car::class), version = 1)
abstract class CarDatabase : RoomDatabase() {
    abstract fun carDao(): CarDao
}


val db = Room.databaseBuilder(
            applicationContext,
            CarDatabase::class.java, "car-database"
        )
    .addMigrations(...)
    .build()
```

ContentProvider

- **Access to structured set of data**
- **Define data security**
  - Via permissions
    - Global
    - Read/Write permissions
    - For single URI
- **Connects data from one process to code running in another process**
- **ContentResolver for access data**

ContentProvider

- **Used by system aps**
  - SMS
  - Contacts
  - Calendar
- **Allows to share data between apps**
- **Data specified via Uri**
- **Allows to use CursorLoader**

ContentProvider

- **Can be backed up by different data sources**
  - SQLite database
  - Network
  - Files
  - …

ContentProvider

- **Initializes early**
  - In priority order
- **Application component start order**
  - Content resolvers
  - Application
  - Invoked component by intent
- https://firebase.googleblog.com/2016/12/how-does-firebase-initialize-on-android.html

ContentProvider - implementation

- **Design data storage**
- **Design content URIs**
  - `content://com.example.app.provider/table1`
  - `content://com.example.app.provider/table2/dataset1`
  - `content://com.example.app.provider/table3/#`
- **Define UriMatcher**
  - Translates Uris to number constant
- **Extend ContentProvider class**
  - query(), insert(), update(), delete()
  - getType()
  - onCreate() - fast operations, postpone db creation
- **Register provider in manifest**

ContentResolver

- `context.getContentResolver()`
- CRUD operations similar params as SQLiteDatabase
- Specify data by URI

Adapter views

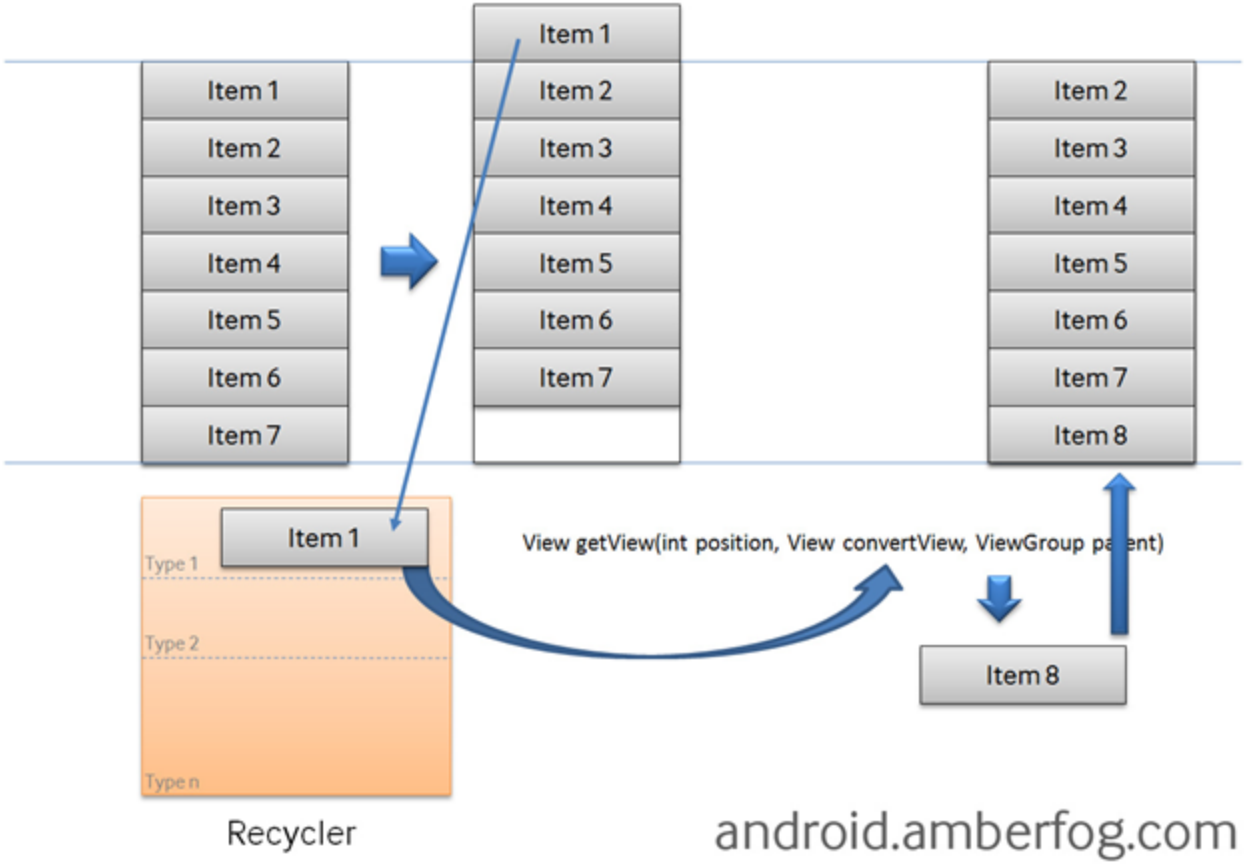
Adapter views

- **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

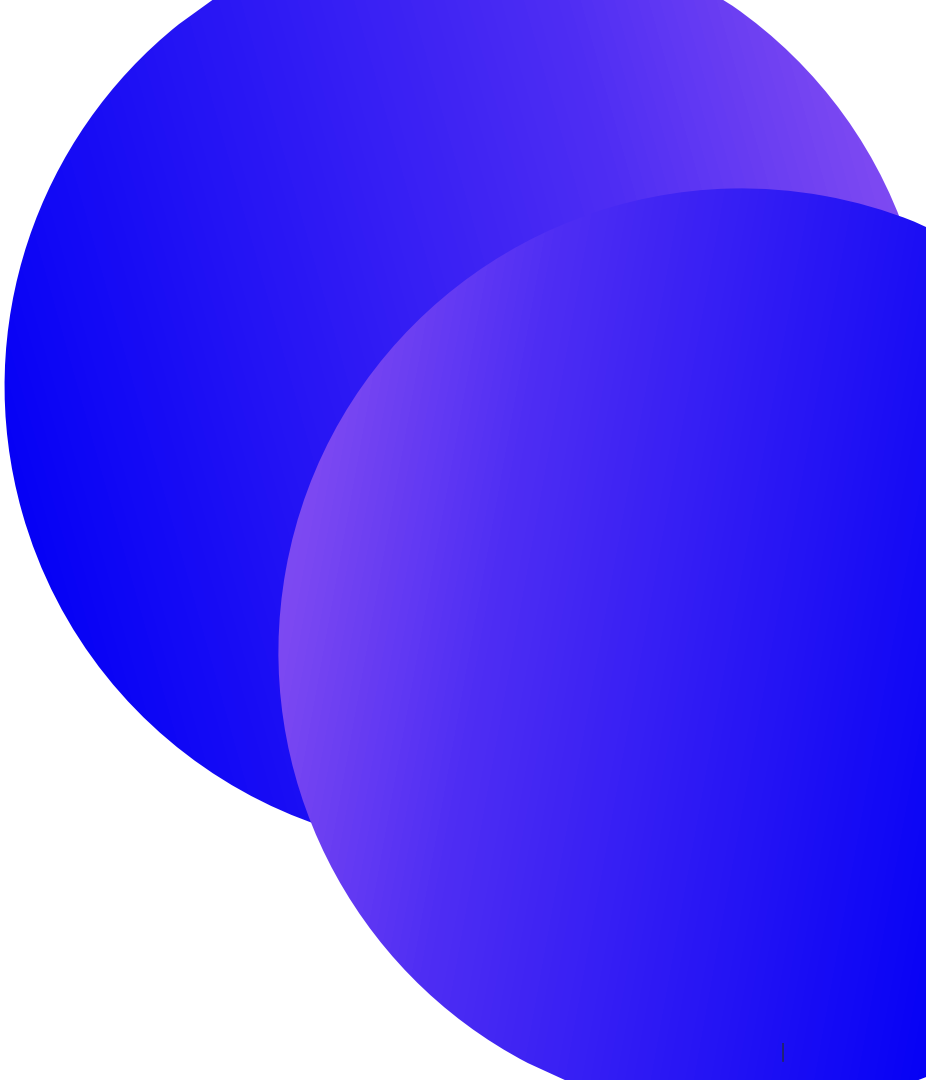
RecyclerView

- 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 - deprecated

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

Scheduling, delayed start

- Handler
- AlarmManager
- JobScheduler
- GCMNetworkManager
- WorkManager

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

Hander - 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

Wake locks

- **Example for obtaining wake lock:**

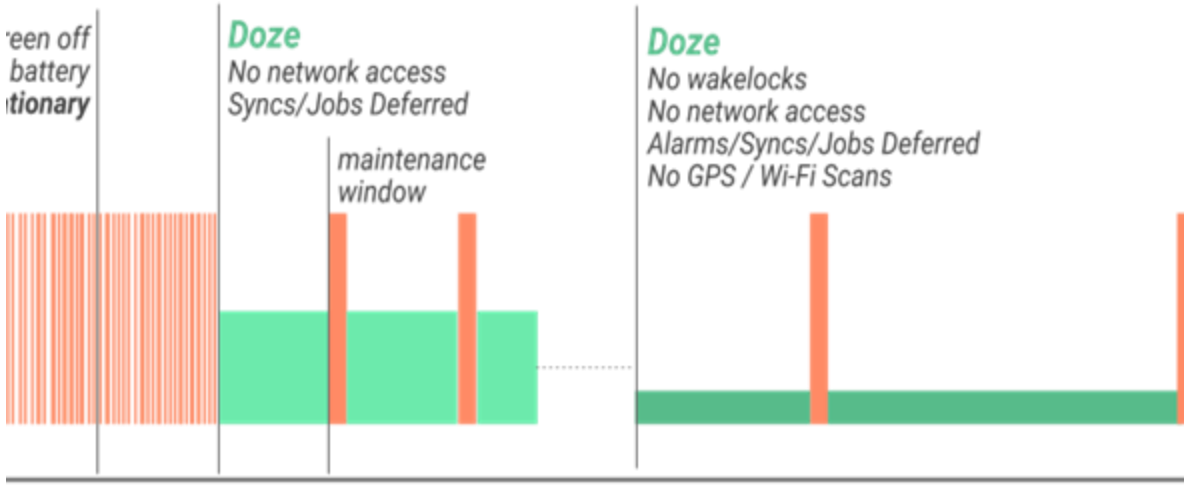
```
val wakeLock: PowerManager.WakeLock =
        (getSystemService(Context.POWER_SERVICE) as PowerManager).run {
            newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "MyApp::MyWakelockTag").apply {
                acquire()
            }
        }
…
wakeLock.release()
```

Alarm manager - sleeping device, solution

- **Acquire your wake lock during `BroadcastReciver.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

- **Maintanance window –** periodical exits of "Doze mode" and allows apps to complete their deffered tasks
- Over time, system schedules maintanance windows less frequently



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**
- **JobInfo** — abstract class that hides the task that needs to be done and the conditions under which the tasks will get executed.
- **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) // Wait time after job failed
        .setPersisted(true) // Survives reboots
        .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
    }

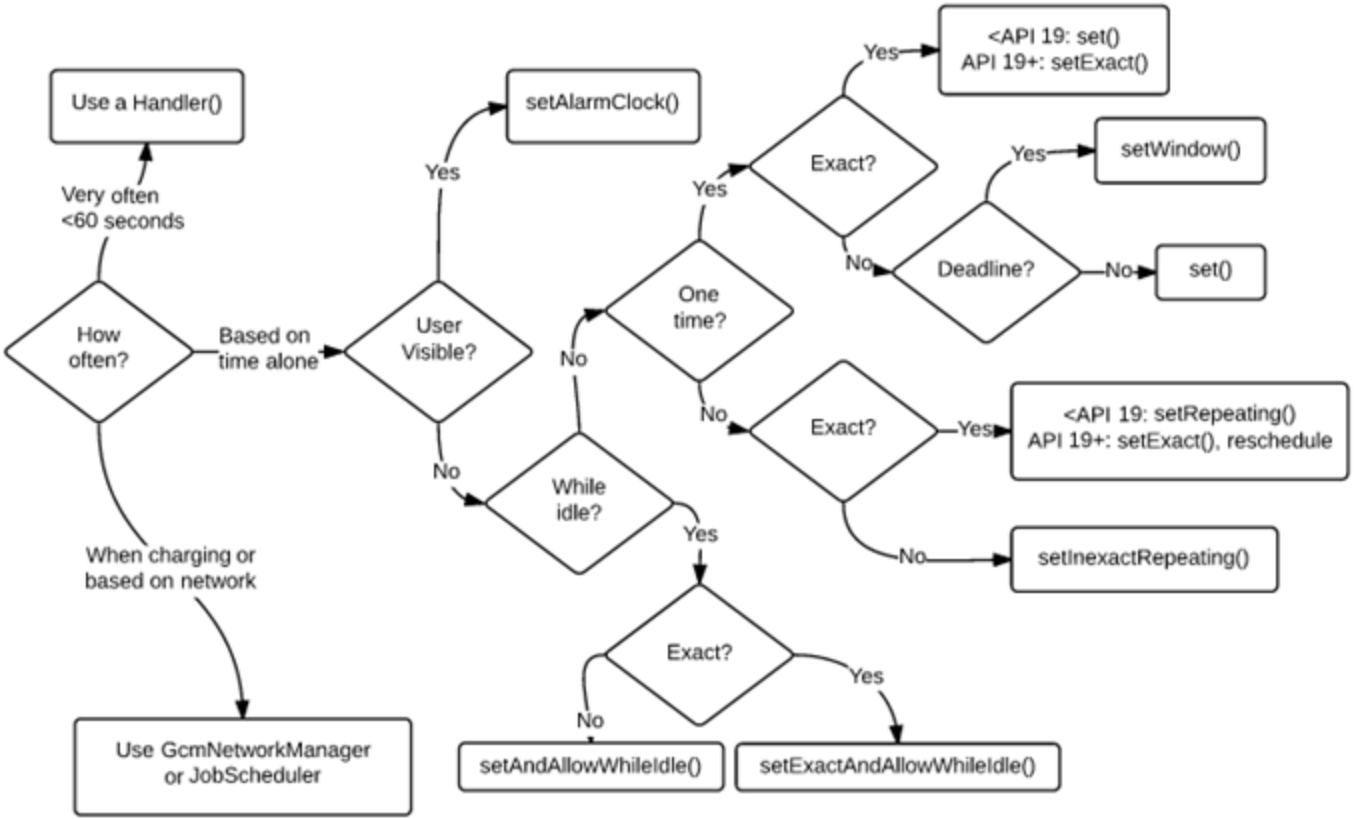
    override fun onStartJob(params: JobParameters?): Boolean {
        // do some stuff

        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**
- **Uses:**
  - **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()
```

https://developer.android.com/topic/libraries/architecture/workmanager/how-to/define-work

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 >= 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

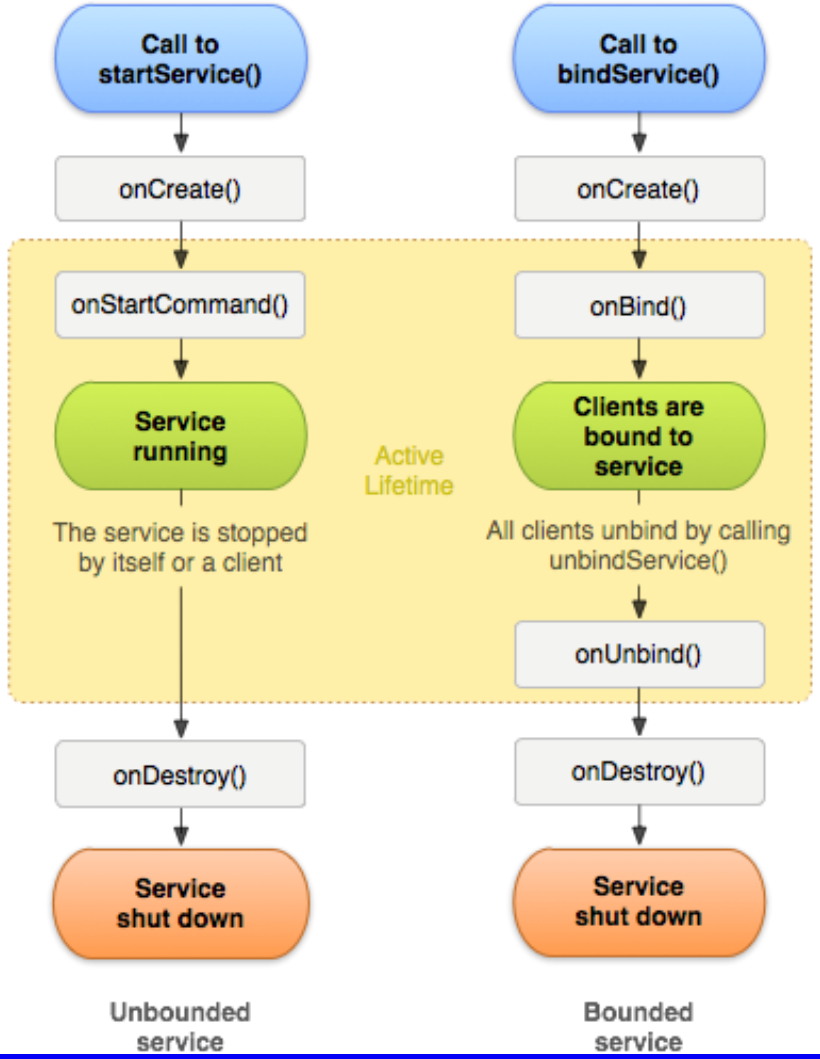
Service connection

- **Define callbacks for service binding**
- **`fun onBindingDied(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



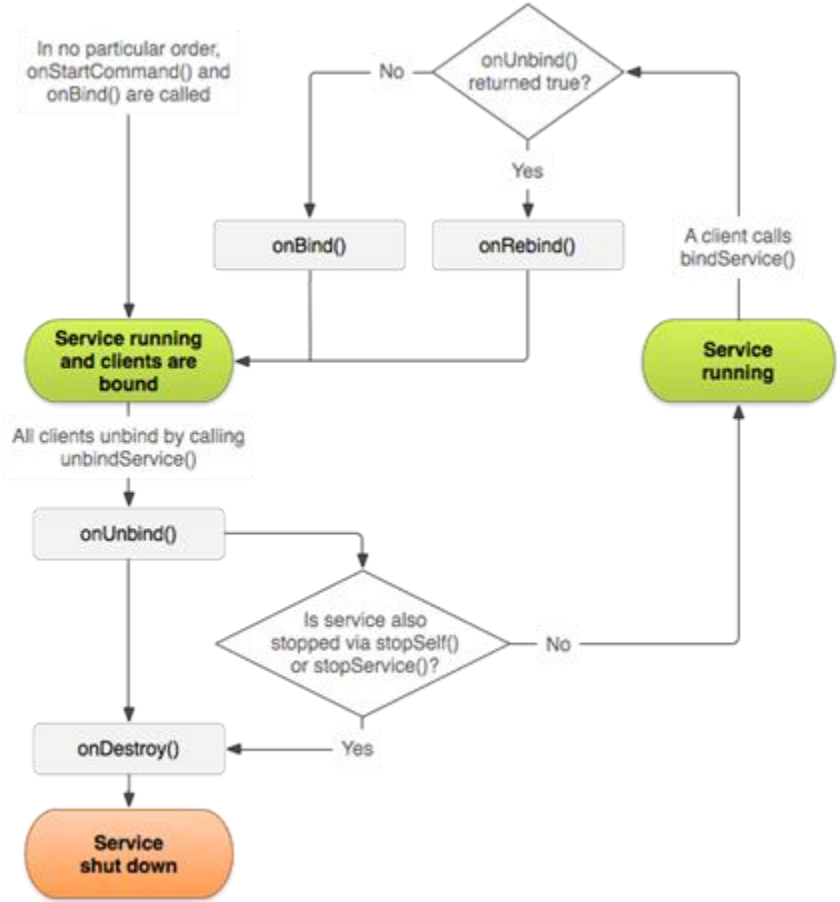
Service lifecycle

- **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 on`HandleIntent(Intent)` for processing requests, runs on worker thread

JobIntentService

- Replacement of IntentService
- Part of support library
- Uses JobScheduler
- Requires WAKE_LOCK permission