# Android lecture 5

Services, release process, libraries

# Services

# Services

- Long running operation in background
- Doesn’t depend on 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

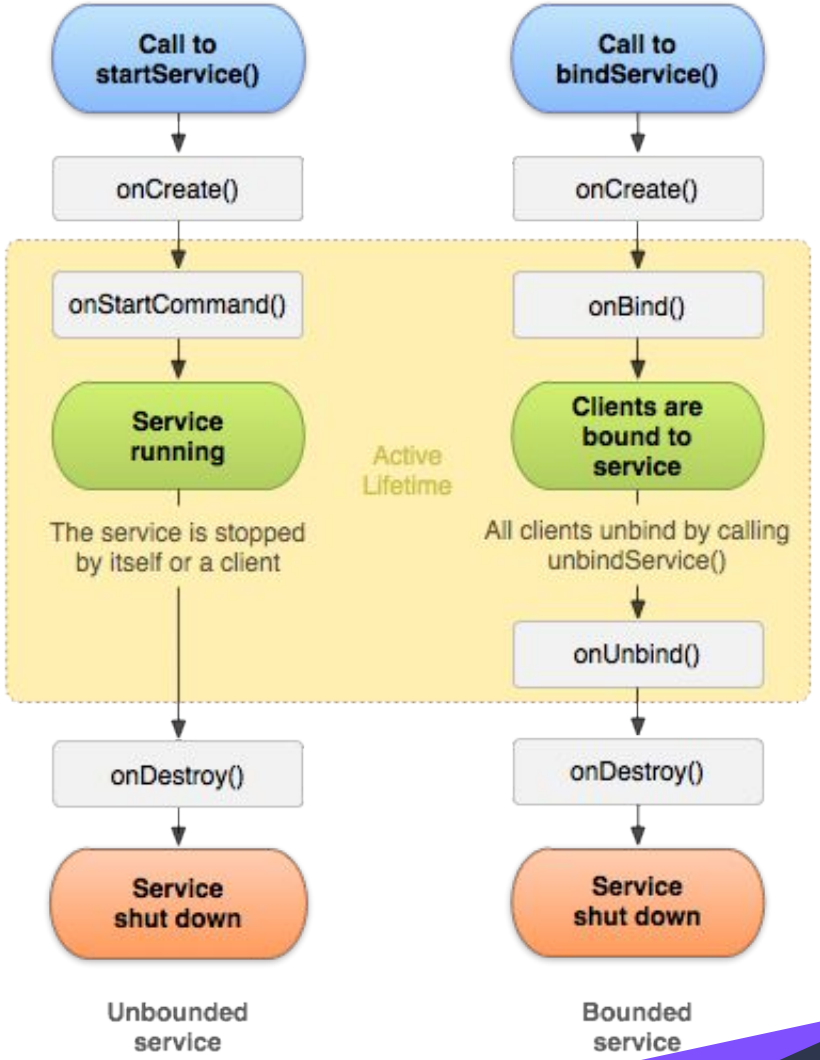
# Service connection

- 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

# Service lifecycle



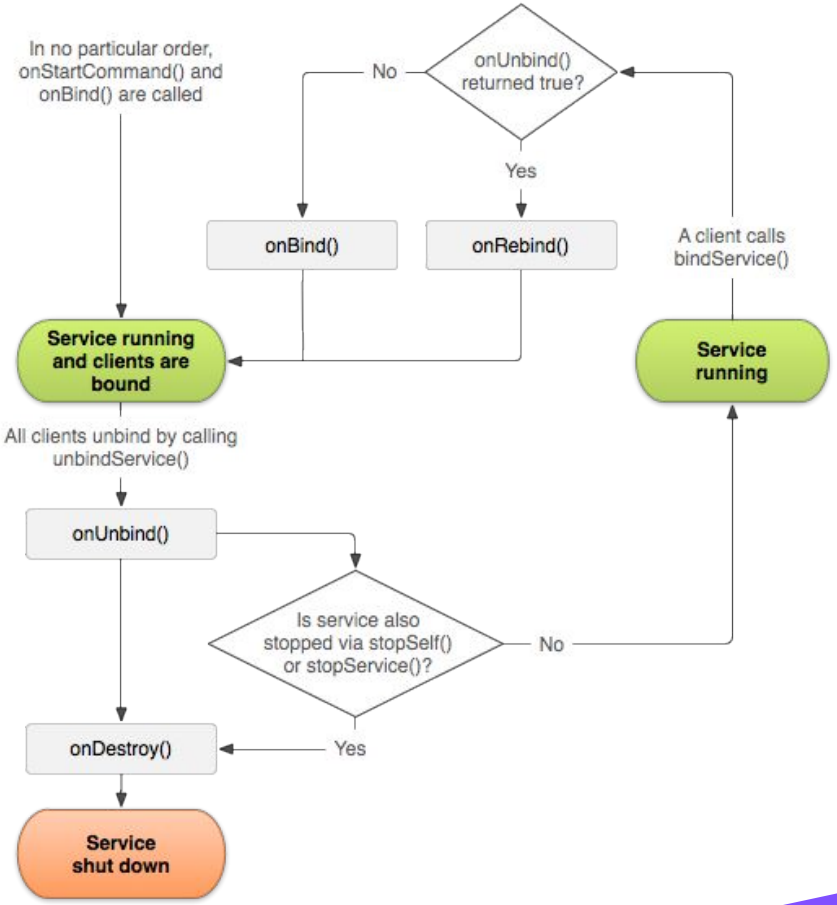
# Service lifecycle

- 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 heading
- 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

avast

# Application release

# Gather materials for release

- Cryptographic keys
  - Apps in store are digitally signed, by developer certificate
  - Identify app developer
  - Self-signed certificate is enough
  - Needs to end after 22 October 2033
  - Protect your private key and password, it is not possible to update application if it is lost
- End-user license agreement (EULA)
  - User should know if you gather some data
  - Not required but recommended
  - GDPR

# Gather material for release

- Application icon
  - Visible in launcher, settings or other applications
  - High-res assets for google play store listing
- Misc. materials
  - Promo and marketing materials
  - Promotional text and graphic for store listing
- Changelog

# Before release

- Delete unused parts (sources, resources, assets)
- Package name
  - https://play.google.com/store/apps/details?id=com.avast.android.mobilesecurity
- Turn off debug features
  - Delete Log.* calls
  - Remove android:debuggable flag  from AndroidManifest
  - Remove Debug or Trace calls
  - If you using WebView ensure that debug is disabled, otherwise is possible to inject js code
- Clean up your directories, checks if libraries doesn’t include some unnecessary files to your *.apk (*.proto, java manifests, …)

# Configure application for release

- Review
  - permissions - remove unnecessary
  - App icon and label
  - Version code and version name
  - Used URLs test vs. production backend
- Check compatibility
  - Support of multiple screens
  - Tablet mode

# Build application for release

- Signing
  - Manually
    - Using Keytool and Jarsigner from JDK
  - Configure sign options in gradle
  - Android studio
  - Sign scheme v2 https://source.android.com/security/apksigning/v2
  - Sign scheme v3 https://source.android.com/security/apksigning/v3
- Obfuscation
  - Use proguard to obfuscate your code, it is really easy to decompile application and find how it works, endpoints
- Consider
  - Who has access to your sign key
  - Signing server

# Prepare external servers and resources

- Ensure that backend is running
- Check if app is switched to prod environment

# Testing your application for release

- Regression test
- Test new features
- If it is possible test it on multiple devices, android versions
- Test multiple languages
- Check if it is looks good with RTL languages
- Check lint, if it doesn’t contain some several issues

# Publishing

- Google play store
  - Registration cost 25 USD
  - Reporting about installs
  - Crashes
    - If user sends it
  - Cloud test lab
    - Run monkey tests on multiple devices before releasing
  - Alpha/Beta groups
  - Distribution specific domain - internal apps
  - Staged rollout
  - API - import crash reports to bug tracker, upload new APK, ...

# Publishing

- Email, Web page
  - Untrusted sources - security risk
  - Manual Updates
- 3rd party distribution
  - Amazon

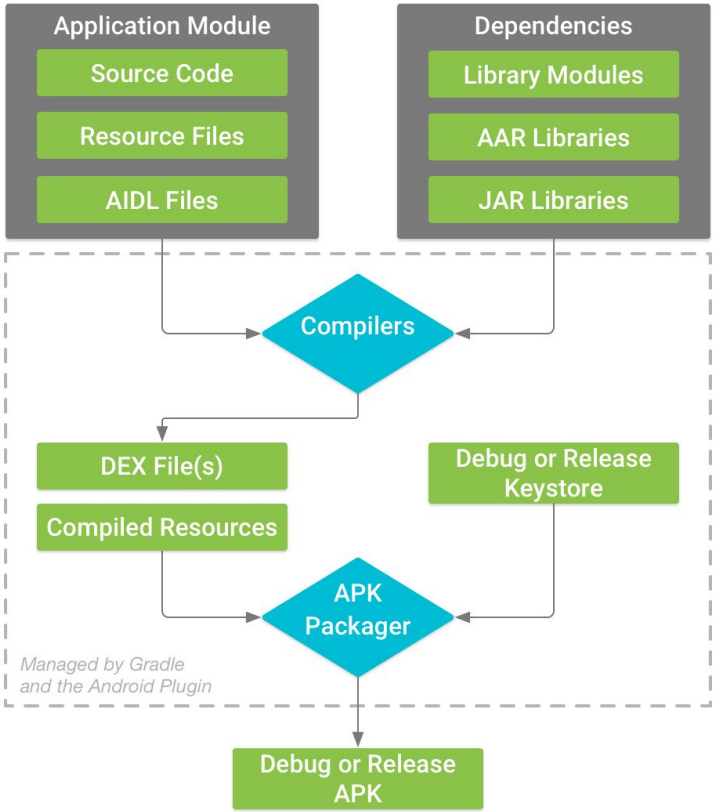
# After publish

- Monitor new crashes
- New permission slows down spreading between users
- Not good idea publish app before weekend or vacation

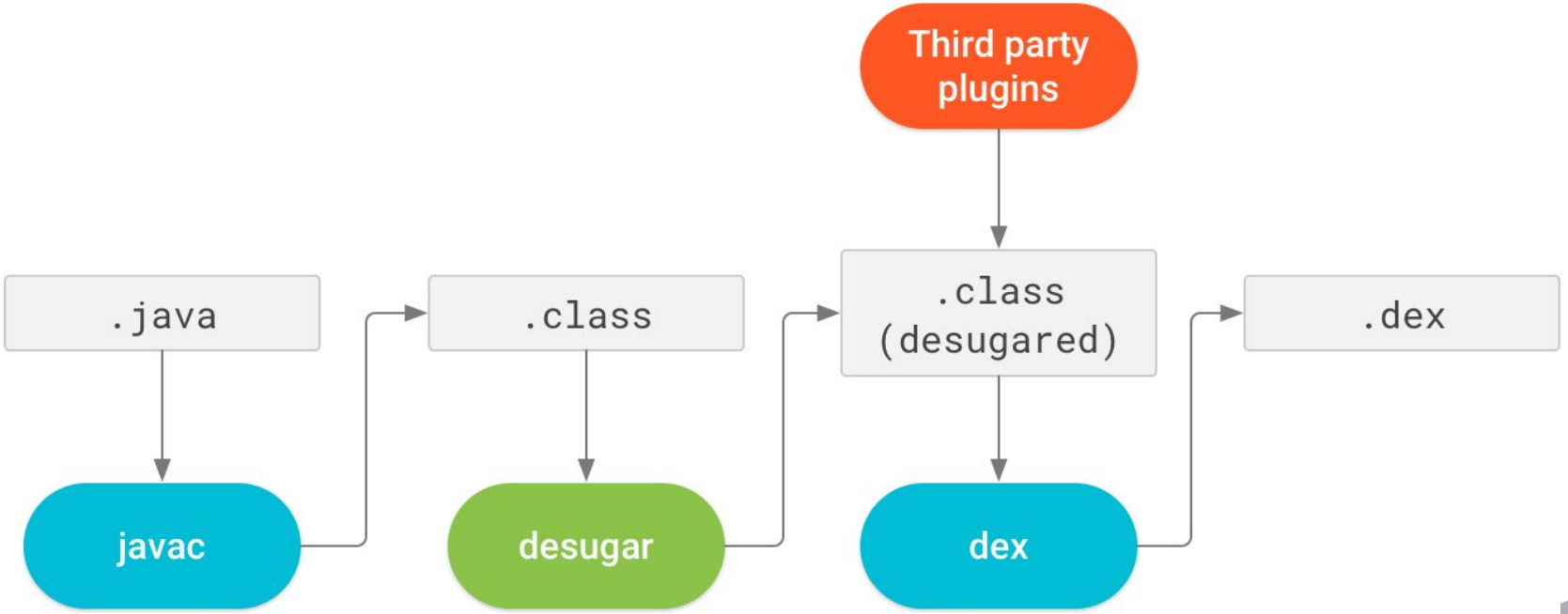
avast

# Build process and APK structure

# Build process



# Support java 8 features (d8)



# Building android apps limitations

- Dex limit 64k methods
  - Shrink unused methods and classes (libraries)
    - You need to specify what is entry point, build dependency graph. Classes which are not part of graph are removed, unused methods as well
    - When you work on library, provide proguard rules together with library
  - Sometimes is better copy some classes from library into application
    - for example guava library
- Google doesn’t allow code side load

# Multidex

- Native support since API-21 for older version support library
- Try to avoid using multidex, it slows down application start
- Splits classes to multiple dex files
- on API>=21 dex files are converted to single .oat file (ART runtime)
- Main dex file loaded when app is started
- Loading of additional dex files is performed during initialization
- Dex files are in app folder
- https://www.blackhat.com/docs/ldn-15/materials/london-15-Welton-Abusing-Android-Apps-And-Gaining-Remote-Code-Execution.pdf

# Apk structure

```
.
├── AndroidManifest.xml- binary XML form of Android manifest
├── classes.dex      - classes compiled to dex
├── META-INF
│   ├── CERT.RSA       - Application certificate
│   ├── CERT.SF        - SHA-1 digest of corresponding lines in the MANIFEST.mf
│   └── MANIFEST.MF    - Manifest file list of files with their SHA-1 hash
├── assets             - assets files
├── res                   - resources
│   ├── drawable-hdpi-v4
│   ├── drawable-mdpi-v4
│   ├── drawable-xhdpi-v4
│   ├── drawable-xxhdpi-v4
│   ├── drawable-xxxhdpi-v4
│   ├── layout
│   ├── menu
│   └── xml
└── resources.arsc        - resources compiled to binary form
```

# Proguard/R8

- Shrink - smaller code, faster build
- Optimize - faster code, removing static conditions
- Obfuscate - make it harder to read

# Decompile apk

- Unzip the apk
- Dex2Jar to convert classes.dex to jar archive
  - https://github.com/pxb1988/dex2jar
- jd-gui to view decompiled classes
  - http://jd.benow.ca/
- Android studio apk analyzer
  - Easy to check resources
  - Compare multiple apk

# Android and SW development - best practices

- Keep strings, dimensions, colors, … in resources
- Create libs for parts used in multiple projects (simplify maintenance, speed-up builds)
- Use git
- Do code reviews
- Write tests


# Libraries, gradle plugins, etc...

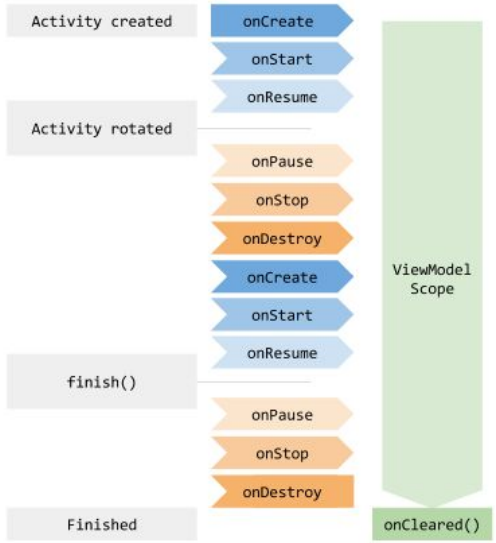
# Android jetpack

- Set of libraries from google
- https://developer.android.com/jetpack
- Groups
  - Foundation
    - AppCompat, Android KTX, Multidex, Test
  - Architecture
    - Data binding, Lifecycles, LiveData, Navigation, Paging, Room, ViewModel, WorkManager
  - Behavior
    - Download manager, Media & playback, Notifications, Permissions, Preferences, Sharing, Slices
  - UI
    - Animations & transitions, Auto, Emoji, Fragment, Layout, Palette, TV, Wear OS

# Android jetpack ViewModel

- Store and manage UI related data
- Scoped to Lifecycle when getting view model instance

- Do not hold reference to Activity/Fragment/View !!!



# Android jetpack - LiveData

- Observable data holder
- Lifecycle aware

# DexCount plugin

- Computes methods count in APK
- Visualize count in nice chart
- https://github.com/KeepSafe/dexcount-gradle-plugin

# Retrofit

- A type-safe HTTP client for Android and Java
- Simplify communication with some API service
- Configurable
  - OkHTTP 3 client - compression, timeouts
  - Supports multiple convertors
    - Gson
    - Jackson
    - Moshi
    - Protobuf
    - Wire
- https://square.github.io/retrofit/

# OkHttp

- An HTTP & HTTP/2 client for Android and Java applications
- Supports sync/async calls
- Supports multiple addresses per URL (Load balancing, failover)
- http://square.github.io/okhttp/

# Dagger

- Dependency injection framework
- Decouple code
- Better testing
- https://dagger.dev/android.html

# Stetho

- Debug tool by Facebook
- Inspect
  - ViewHiearchy
  - Database
  - Shared preferences
  - Network trafic
- http://facebook.github.io/stetho

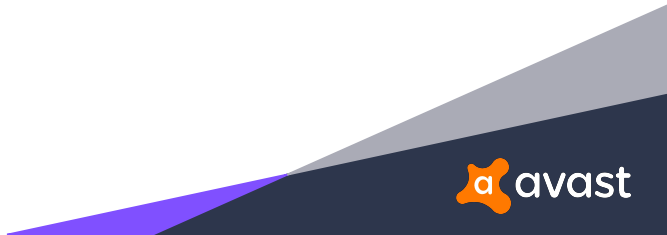
# LeakCannary

- Helps with finding memory leaks
- https://github.com/square/leakcanary


# Kotlin coroutines intro - bonus

# Kotlin coroutines

- Lightway thread
- Uses suspending functions



# Suspend function

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

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

- Suspend lambda
- suspend/resume
- Can be called only from other suspend function or in coroutine created by coroutine builder

# Suspend functions

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

# CorotutineDispatcher

- All coroutines run in dispatcher
- Coroutine can **suspend** themselves
- Knows how to **resume** suspended coroutines

# 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<br>• Call UI functions<br>• Updating LiveData | • Database<br>• R/W files<br>• Networking | • Sorting list<br>• Parsing JSON<br>• 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

# Coroutine scope builders

- Creates new coroutine scope inside current one
- Cancellation is propagated from parent to children's
- `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

# Thank you
# Q&A

Feedback is appreciated

prokop@avast.com

Please use [mff-android] in subject