



# Android Development

Firebase again



# Firestore revisited

# Firestore rules

- Firestore Realtime Database Rules determine who has
  - read and write access to your database
  - how your data is structured
  - what indexes exist.
- These rules are enforced automatically at all times.
  - Every read and write request will only be completed if your rules allow it.
  - This is to protect your database from abuse until you have time to customize your rules or set up authentication.

# Firestore rules

- Firestore Database Rules have a JavaScript-like syntax and come in four types:
  - `.read`
    - Describes if and when data is allowed to be read by users.
  - `.write`
    - Describes if and when data is allowed to be written.
  - `.validate`
    - Defines what a correctly formatted value will look like, whether it has child attributes, and the data type.
  - `.indexOn`
    - Specifies a child to index to support ordering and querying.

# Firestore rules

- Rule examples:

// These rules require authentication

```
{  
  "rules": {  
    ".read": "auth != null",  
    ".write": "auth != null"  
  }  
}
```

// These rules grant access to a node matching the authenticated  
// user's ID from the Firebase auth token

```
{  
  "rules": {  
    "users": {  
      "$uid": {  
        ".read": "$uid === auth.uid",  
        ".write": "$uid === auth.uid"  
      }  
    }  
  }  
}
```

# Firestore rules

- What can we use when we want to validate data
  - `now`
    - The current time in milliseconds since Linux epoch.
    - This works particularly well for validating timestamps created with the SDK's `firebase.database.ServerValue.TIMESTAMP`.
  - `root`
    - A `RuleDataSnapshot` representing the root path in the Firestore database as it exists before the attempted operation.
  - `newData`
    - A `RuleDataSnapshot` representing the data as it would exist after the attempted operation.
    - It includes the new data being written and existing data.
  - `data`
    - A `RuleDataSnapshot` representing the data as it existed before the attempted operation.
  - `$` variables
    - A wildcard path used to represent ids and dynamic child keys.
  - `auth`
    - Represents an authenticated user's token payload.

# Firestore example

```
{
  "rules": {
    "foo": {
      // /foo is readable by the world
      ".read": true,

      // /foo is writable by the world
      ".write": true,

      // data written to /foo must be a string less than 100 characters
      ".validate": "newData.isString() && newData.val().length < 100"
    }
  }
}
```

# Firestore rules

- When we validate data we also can use:
  - `hasChildren('children_name')`
  - `isString()`
  - `isNumber()`
  - `isBoolean()`
  - `val().matches(regex)`



# Firestore rules

- We can create rules for disabling certain write operations such as update:

```
// we can write as long as old data or new data  
// does not exist in other words, if this is a  
// delete or a create, but not an update  
".write": "!data.exists() || !newData.exists()"
```

- We also can use any existing data in the rules:

```
".write": "root.child('allow_writes').val() === true &&  
!data.parent().child('readOnly').exists() &&  
newData.child('foo').exists()"
```

# Firestore rules

- Rules Are Not Filters
  - Rules are applied in an atomic manner.
  - That means that a read or write operation is failed immediately if there isn't a rule at that location or at a parent location that grants access.
  - Even if every affected child path is accessible, reading at the parent location will fail completely.

# Firestore rules

- Rules Are Not Filters
  - Consider this structure:

```
{  
  "rules": {  
    "records": {  
      "rec1": {  
        ".read": true  
      },  
      "rec2": {  
        ".read": false  
      }  
    }  
  }  
}
```

# Firestore rules

- We can use queries for our read operations.
  - In this case we need to add an index to the data node.
  - The index will be incorporated in the query.

```
{  
  "rules": {  
    "scores": {  
      ".indexOn": ".value"  
    }  
  }  
}
```

# Firestore login

## My Firebase App



Sign in with Google



Sign in with Facebook



Sign in with Twitter



Sign in with email



Sign in with phone

# Firestore login

- Firestore login allows:
  - Multiple Providers
    - sign-in flows for email, phone authentication, Google Sign-In, Facebook Login, Twitter Login, Apple login, Phone number, ...
  - Account Management
    - flows to handle account management tasks, such as account creation and password resets.
  - Account Linking
    - flows to safely link user accounts across identity providers.
  - Custom Themes
    - Customize the look of FirestoreUI to match your app.
    - Also, because FirestoreUI is open source, you can fork the project and customize it exactly to your needs.
  - Smart Lock for Passwords
    - automatic integration with [Smart Lock for Passwords](#) for fast cross-device sign-in.

# Steps of adding Firebase login

1. [Add Firebase to your Android project.](#)
2. Add the dependencies for FirebaseUI to your app-level build.gradle file.
  1. If you want to support sign-in with Facebook or Twitter, also include the Facebook and Twitter SDKs
3. If you haven't yet connected your app to your Firebase project, do so from the [Firebase console](#)
4. In the [Firebase console](#), open the **Authentication** section and enable the sign-in methods you want to support. Some sign-in methods require additional information, usually available in the service's developer console.
5. If you support Google Sign-in and haven't yet specified your app's SHA-1 fingerprint, do so from the [Settings page](#) of the Firebase console. See [Authenticating Your Client](#) for details on how to get your app's SHA-1 fingerprint.
6. If you support sign-in with Facebook or Twitter, add string resources to strings.xml that specify the identifying information required by each provider

# Add sign in intent

```
// Choose authentication providers
val providers = arrayListOf(
    AuthUI.IdpConfig.EmailBuilder().build(),
    AuthUI.IdpConfig.PhoneBuilder().build(),
    AuthUI.IdpConfig.GoogleBuilder().build(),
    AuthUI.IdpConfig.FacebookBuilder().build(),
    AuthUI.IdpConfig.TwitterBuilder().build())

// Create and launch sign-in intent
startActivityForResult(
    AuthUI.getInstance()
        .createSignInIntentBuilder()
        .setAvailableProviders(providers)
        .build(),
    RC_SIGN_IN)
```



# Result of sign in

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data:
Intent?) {
    super.onActivityResult(requestCode, resultCode, data)

    if (requestCode == RC_SIGN_IN) {
        val response = IdpResponse.fromResultIntent(data)

        if (resultCode == Activity.RESULT_OK) {
            // Successfully signed in
            val user = FirebaseAuth.getInstance().currentUser
            // ...
        } else {
            // Sign in failed. If response is null the user canceled the
            // sign-in flow using the back button. Otherwise check
            // response.getError().getErrorCode() and handle the error.
            // ...
        }
    }
}
```

# Sign out

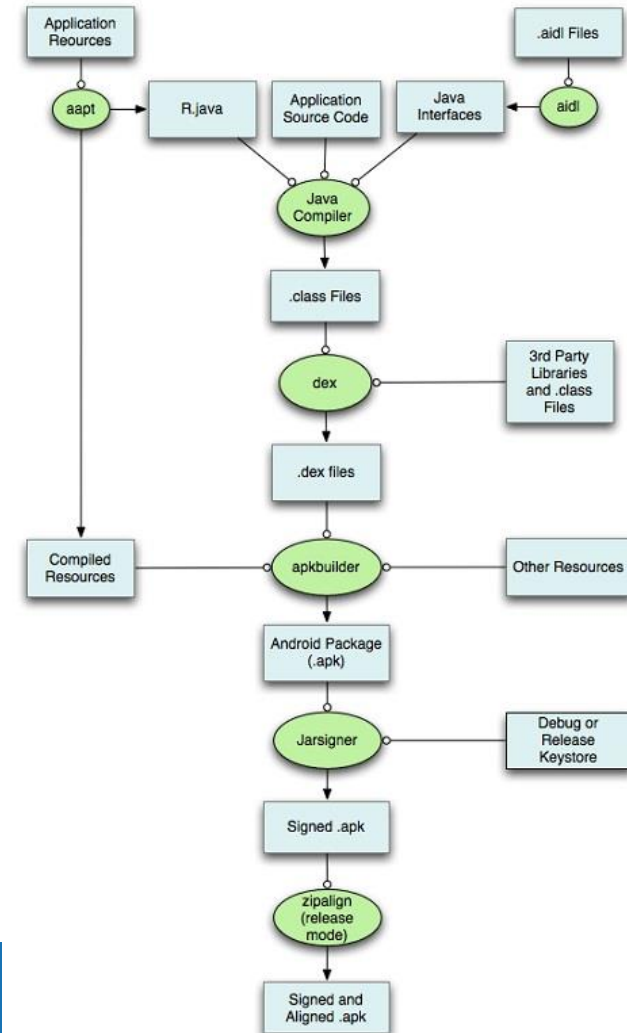
```
AuthUI.getInstance()  
    .delete(this)  
    .addOnCompleteListener {  
        // ...  
    }
```



# Android build process

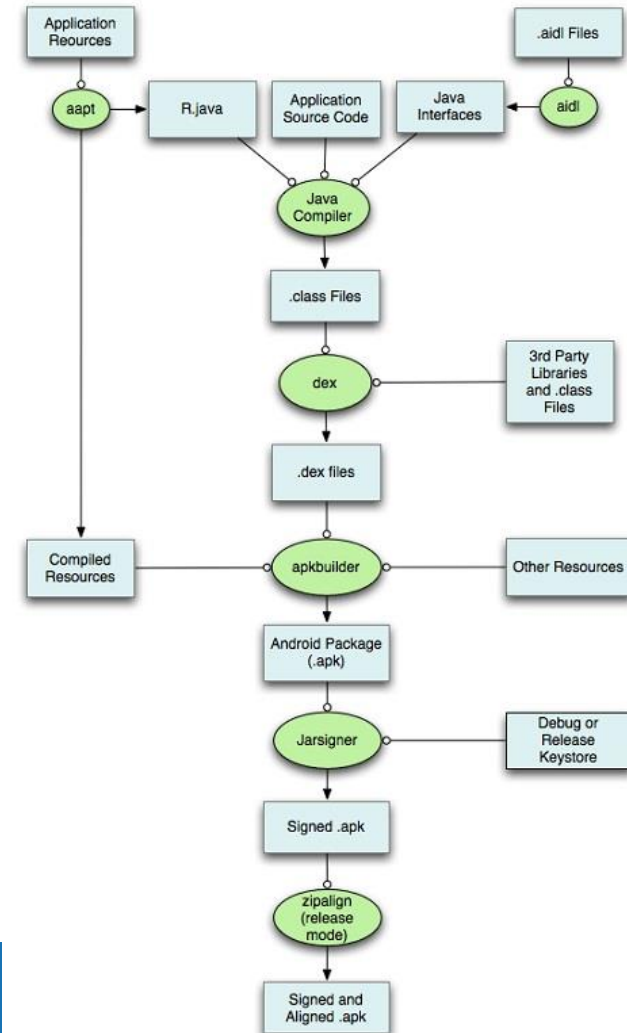
# Android build process in details

- AIDL
  - Android Interface Definition Language
  - Inter process Communication
- Java -> Class -> Dex
- APK
  - Android Application Package

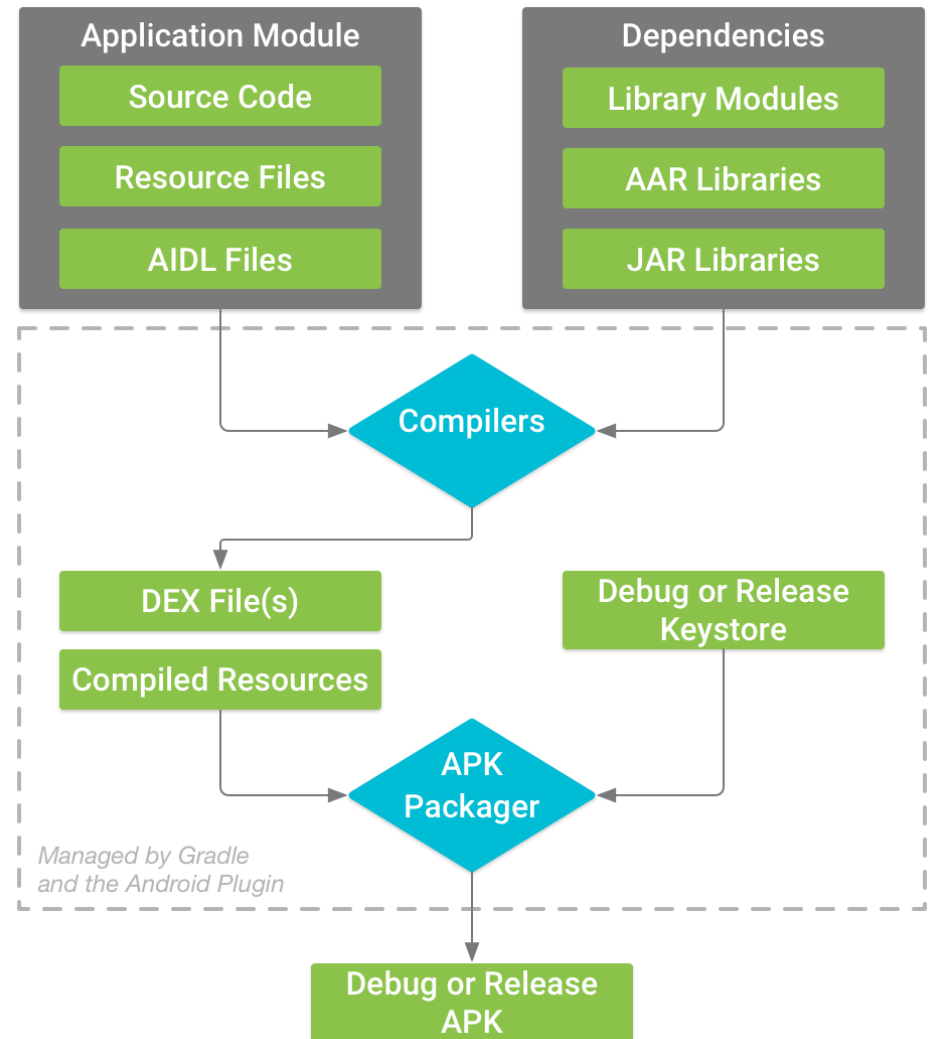


# Android build process in details

- Zipalign
  - ensure that all uncompressed data starts with a particular alignment relative to the start of the file
  - The benefit is a reduction in the amount of RAM consumed when running the application.
- ProGuard - for .class files
  - Shrinks its size
  - Optimizes
  - Obfuscate
  - And checks it
    - Removes unused code



# Android build process in details

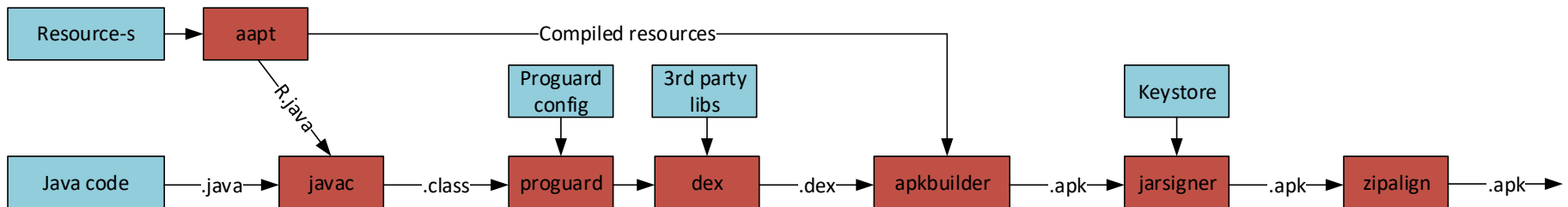


# Android project

- As you already aware
  - Android applications are developed in Java/Kotlin language
  - Other languages can be used as well
- The Java (Kotlin) build process and steps are going to be investigated
  - These steps are performed by the IDE, however you should understand the details

# Build process

- R.java is generated based on the resources (aapt - Android Asset Packaging Tool)
- R.java and other .java files are compiled to java byte code (.class files)
- .class files are obfuscated by Proguard
- Java byte codes and 3rd party libraries are compiled (dexed) to ART/Dalvik executables (.dex files)
- Compiled resources and .dex files are gathered by the apk builder to and .apk file
- Apk file is signed digitally (jarsigner) and data is aligned (zipalign)
  - Then it is ready to submit to the play store





# Let's investigate an apk file

- BKK Info application
- The .apk file is a zip file
  - Unzip it and discover its content
    - Resources
    - Values
  - Source code?
    - classes.dex

# Let's investigate an apk file

- Apktool
  - `java -jar apktool_2.4.1.jar d bkkinfo.apk`
  - Manifest
  - Source code
    - Smali
- Decompile java byte code
  - dex2jar
  - jd-gui

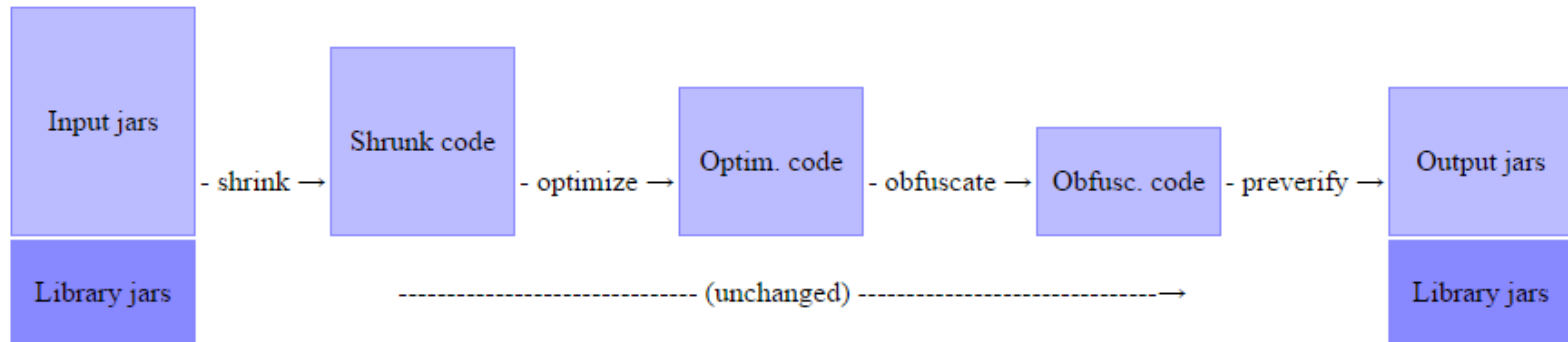
# ProGuard

- It is easy to decompile the java byte code
  - Everything is there
    - Variable names
    - Functions
  - Easy to copy
- Code obfuscation
  - Names are changed to „hard to read by human”
- To compact the code
  - Shorter code – smaller apk, less storage required

# ProGuard

- Eliminating dead code
  - E.g. unused modules
- Optimizing the code
- It has to be configured, unless it might remove useful codes
  - Reflection
  - proguard-rules.pro
- Proguard is integrated to the build process

# ProGuard



# ProGuard

- Some of the parameters
  - keep [,modifier,...] class\_specification
  - keepclassmembers [,modifier,...]  
class\_specification
  - ...
  - dontshrink
  - ...
  - dontoptimize
  - ...
  - dontobfuscate



Pázmány Péter Catholic University  
Faculty of Information Technology and Bionics



# Gradle

- To automate the build process
  - To perform the entire process as seen in the slide
- Project is described by
  - build.gradle
- Based on Groovy-n
  - DSL (domain specific language)
- General, its functionality can be augmented with plugins
  - Java plugin
  - Android plugin
- Wrapper
  - Downloads the entire Gradle distribution
  - gradlew



# Gradle dependency management

```
repositories {  
    mavenCentral()  
    maven {  
        url 'https://reponse.hu/'  
    }  
}
```

```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.core:core-ktx:1.0.2'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'  
}
```

- Configuration for dependencies
  - compile: required to compile the main project
  - testCompile: required to compile the tests
- gradlew dependencies

# Gradle Android plugin

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 29
    buildToolsVersion "29.0.3"

    defaultConfig {
        applicationId "hu.ppke.itk.android.demoapp"
        minSdkVersion 21
        targetSdkVersion 29
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
}
```

# Product flavors

- ```
android {  
    defaultConfig { ... }  
    signingConfigs { ... }  
    buildTypes { ... }  
    productFlavors {  
        demo {  
            applicationId "com.buildsystemexample.app.demo"  
            versionName "1.0-demo"  
        }  
        full {  
            applicationId "com.buildsystemexample.app.full"  
            versionName "1.0-full"  
        }  
    }  
}
```

...

# Product flavors

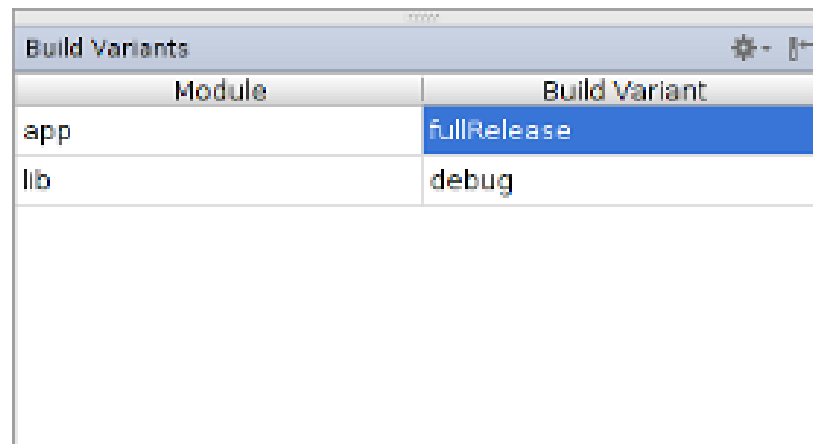
- `src/demo/java`
- `src/demo/res`
- `src/demo/res/layout`
- `src/demo/res/values`
- ...

# Build types

- ```
android {  
    defaultConfig { ... }  
    signingConfigs { ... }  
    buildTypes { ... }  
    productFlavors { ... }  
    buildTypes {  
        release {  
            minifyEnabled false  
            proguardFiles getDefaultProguardFile(  
                'proguard-android.txt'), 'proguard-rules.pro',  
            }  
        debug {  
            debuggable true  
        }  
    }  
}
```

# Build variants

- Variant = flavor + build type
- All variants: cross product of flavor and build types

A screenshot of the 'Build Variants' window in an IDE. The window has a title bar 'Build Variants' with a gear icon and a minus sign. It contains a table with two columns: 'Module' and 'Build Variant'. The 'app' module is associated with the 'fullRelease' build variant, which is highlighted in blue. The 'lib' module is associated with the 'debug' build variant.

Module	Build Variant
app	fullRelease
lib	debug

- `gradle assembleDemoDebug`
- `gradle assembleFullRelease`



Pázmány Péter Catholic University  
Faculty of Information Technology and Bionics

# Testing

# Test – Validating software

- Objectives of software testing
  - Discover errors of the system
  - Ensure that the system works properly in real scenarios
- Validation
  - Determine whether the software works as desired
    - The correct software is built
- Verification
  - The software meets the specifications
    - The software is built correctly.
- The malfunctions and errors have to be discovered



# Tests

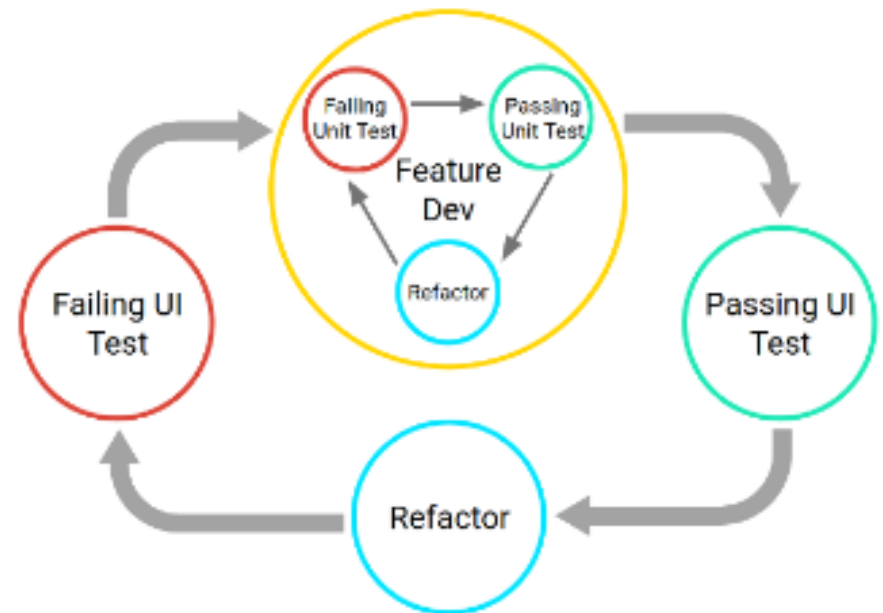
- Testing functionality
  - What does the program do?
  - The software is considered as a black box, test cases are written based on the specifications
  - The actual implementation is not taken into considerations
  - Tests can be designed in early stages of the software process
- Structural test
  - How the program does it?
  - Tests are written based on the structure of the program, and implementation
  - Testers analyze the code to ensure that all of the instructions are evaluated once
    - All data/instruction path cannot be tested due to the complexity of the code
- Testing non functional parts
  - How good the program is?
  - The efficiency, reliability is tested
- Regression testing
  - What went wrong during error corrections?

# Load tests

- Programs should be tested with larger load (than designed)
  - The load is increased step-by-step until system failure
- Tasks
  - To tests the system under extremal conditions
  - Data loss or service loss is not allowed due to overload
    - System have to be designed in order to guarantee this requirement
  - Errors can be discovered which do not occur in normal cases
- Load tests are important in distributed systems.

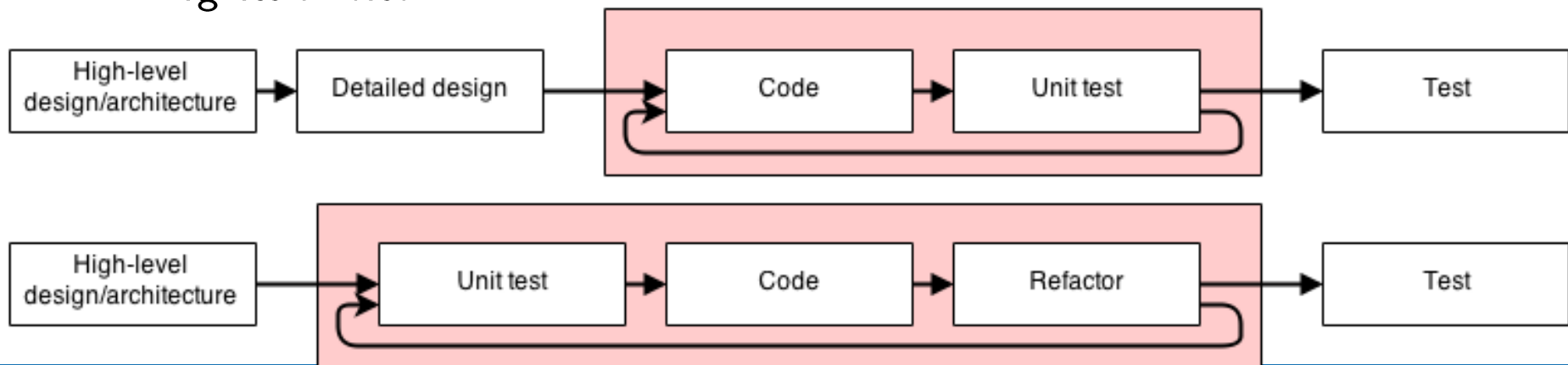
# Designing tests

- Optimal case
  - For each program unit a test case have to be designed
  - Connections between program units also must be tested
- Tests have to be designed in parallel of the process of program designing



# Software lifecycle – TDD

- Agile method
  - It is popular in modern software development methods
- A red (failing) test is written first
  - Something new, which is not implemented
- Next, the new code is implemented to turn the test green
  - The function is implemented well
- Refactoring to check the correctness of the program
  - Regression test



# Test methods in Android

- Instrumentation test
  - Tests on the device or emulator
- UI tests
  - Tests for the User Interface only
- Monkey tests
  - Random interactions with the application UI without any previous knowledge
- Unit tests
  - Tests for a specific part of the application without the rest
- Instrumented Unit Test
  - Testing the app on the device with Unit tests 😊. This has great support in Android Studio

# Instrumentation tests

- On emulator of physical devices (slow)
- In this case the lifecycle can be controlled manually
  - It can be tested how an activity responds to an intent
  - It can be tested whether a value entered into a text field remains there after orientation change
- Components can be tested isolated
  - Additional components can be mocked
- Based on JUnit



# Espresso



# Espresso

- Use Espresso to write concise, beautiful, and reliable Android UI tests.

@Test

```
fun greeterSaysHello() {  
    onView(withId(R.id.name_field)).perform(typeText("Steve"))  
    onView(withId(R.id.greet_button)).perform(click())  
    onView(withText("Hello Steve!")).check(matches(isDisplayed()))  
}
```



# Espresso

- The core API is small, predictable, and easy to learn and yet remains open for customization.
- Espresso tests state expectations, interactions, and assertions clearly without the distraction of boilerplate content, custom infrastructure, or messy implementation details getting in the way.
- Espresso lets you leave your waits, syncs, sleeps, and polls behind while it manipulates and asserts on the application UI when it is at rest.

# Espresso

- Packages

- espresso-core - Contains core and basic View matchers, actions, and assertions.
- espresso-web - Contains resources for WebView support.
- espresso-idling-resource - Espresso's mechanism for synchronization with background jobs.
- espresso-contrib - External contributions that contain DatePicker, RecyclerView and Drawer actions, accessibility checks, and CountingIdlingResource.
- espresso-intents - Extension to validate and stub intents for hermetic testing.
- espresso-remote - Location of Espresso's multi-process functionality.

# UiAutomator

- UI tests executed on the device
  - Another application can be opened as well
- `uiautomatorviewer` – Represents the actual state of the UI hierarchy
- `UiAutomatorTestCase`
- API is divided into five parts
  - `UiDevice` – represents a device, for example `pressHome`
  - `UiSelector` – to find different elements
  - `UiObject` – represent a GUI element to perform actions (such as click)
  - `UiCollection` – set of elements, selected with `UiSelector`
  - `UiScrollable` – represents scrollable elements
- <https://developer.android.com/training/testing/ui-automator>

# Appium

- Server-client architecture
- Server is based on node.js
- WebDriver clients can send command through JSON Wire Protocol
- Client can be on arbitrary language, independent on the tested application
  - Thus it can be reused
    - Ruby, Python, Java, JavaScript, PHP, C#, Objective-C, Clojure, Perl
- For native, web based, and hybrid applications as well

# UI/Application Exerciser Monkey

- The Monkey is a command-line tool that you can run on any emulator instance or on a device.
  - It sends a pseudo-random stream of user events into the system, which acts as a stress test on the application software you are developing.
- The Monkey includes a number of options, but they break down into four primary categories:
  - Basic configuration options, such as setting the number of events to attempt.
  - Operational constraints, such as restricting the test to a single package.
  - Event types and frequencies.
  - Debugging options.

# Monkey test

- When the Monkey runs, it generates events and sends them to the system.
- It also watches the system under test and looks for three conditions, which it treats specially:
  - If you have constrained the Monkey to run in one or more specific packages, it watches for attempts to navigate to any other packages, and blocks them.
  - If your application crashes or receives any sort of unhandled exception, the Monkey will stop and report the error.
  - If your application generates an application not responding error, the Monkey will stop and report the error.
  - Depending on the verbosity level you have selected, you will also see reports on the progress of the Monkey and the events being generated.

# Monkey test

- UI/Application Exerciser Monkey
  - Random event is sent to the Activity
  - The distribution can be configured
  - `adb shell monkey -p package.name -v 500`
- monkeyrunner
  - Python API
  - Application can be controlled
  - Three main modules
    - MonkeyRunner – to connect to device and start tests
    - MonkeyDevice – to simulate touch or key events
    - MonkeyImage – to save screenshot

# Unit tests – Android built-in

- Executed on JVM – fast
- Stub android.jar
  - Mock is required
  - Final keywords are removed
- Build variant: Unit Tests
- src/test/java
- Test dependencies

```
dependencies {  
    testImplementation 'junit:junit:4.12'  
}
```

- `gradlew test`



# Unit tests – AssertJ Android

- Convenient usage for assertions
  - JUnit Assert:  
`assertEquals(View.GONE, view.getVisibility());`
  - AssertJ:  
`assertThat(view.getVisibility()).isEqualTo(View.GONE);`
  - AssertJ Android:  
`assertThat(view).isGone();`
- Domain-specific language (DSL) for Android
- Asserts for almost all of the Android classes
- <https://github.com/square/assertj-android>

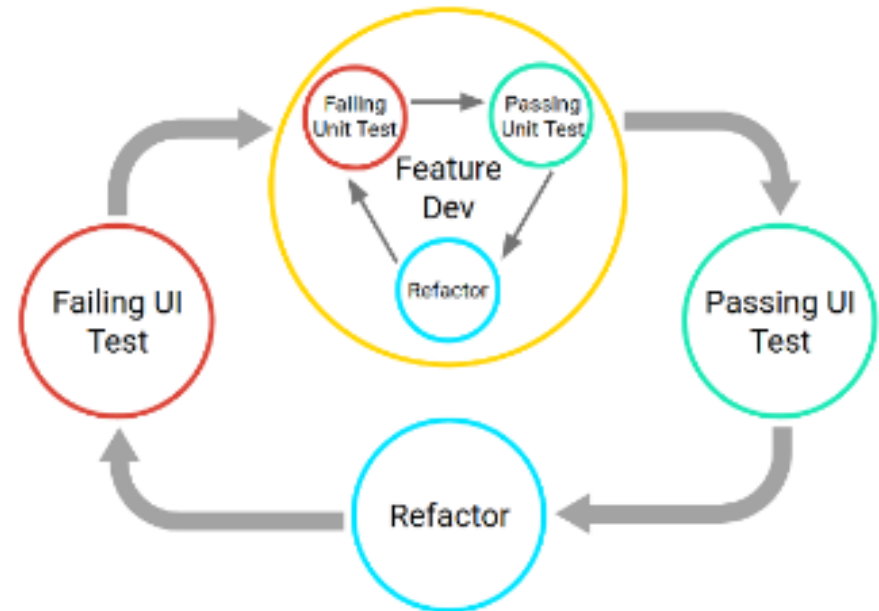
# Unit tests – Robolectric

- Instead on mobile device, it is executed on desktop – extremely fast
- JUnit tests
- Additional project is not required
- RobolectricTestRunner
- API to start Activity

```
@RunWith(RobolectricTestRunner.class)
public class MyActivityTest {
    @Test
    public void clickingButton_shouldChangeMessage() {
        MyActivity activity =
            Robolectric.setupActivity(MyActivity.class);
        activity.button.performClick();
        assertThat(activity.message.getText())
            .isEqualTo("Robolectric Rocks!");
    }
}
```

# Homework

- Create a Simple Demonstration
  - Use test case – code – refactoring approach!
- The demonstration is a Calculator application
  - To add and multiply integer numbers
    - Use buttons to enter numbers and perform calculations
  - Add a delete button
    - Deletes the last character
  - Add a clear button
    - Clears all the entered characters
  - Save the instance state
    - Don't forget the test case for it!
  - Run monkey test on it
    - Save the output into monkeytest\_output.txt and upload with the project!





# Libraries, support libraries

Next week