Unit Testing in Android

Anuj Middha
ADG Delhi

Niharika Arora
Senior Software Engineer,1mg

Agenda

- Introduction to App Architecture and Unit Tests
- Tools and Frameworks
- Writing Testable Code
- Demo

Good/Clean Code Base?

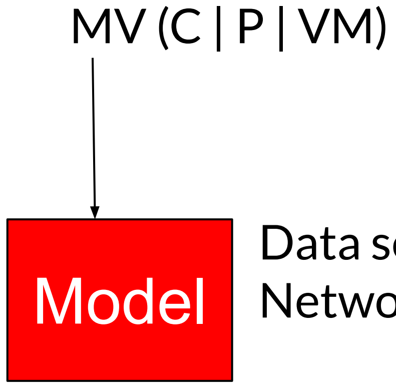
What is that?Why should we have good/clean code base?

Scalable | Stable | Testable | Modular

Application Architecture

Why do I care?

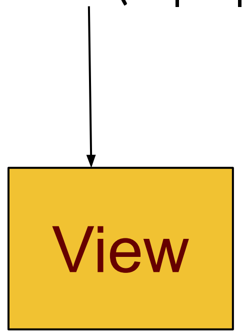
MV* Patterns



Data source of the application
Network layer, database operations

MV* Patterns

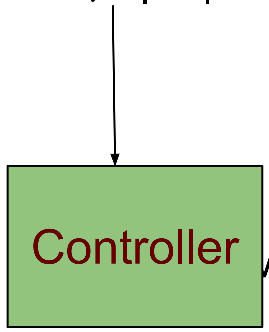
MV (C | P | VM)



Responsible for displaying data

MV* Patterns

MV (C | P | VM)

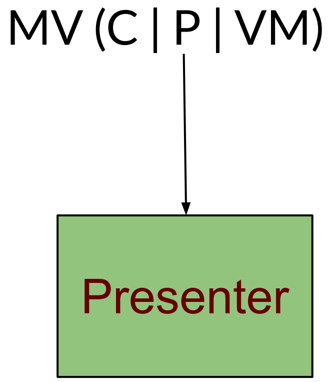


Controller manipulates, edit, uses data model

w it to users via View.

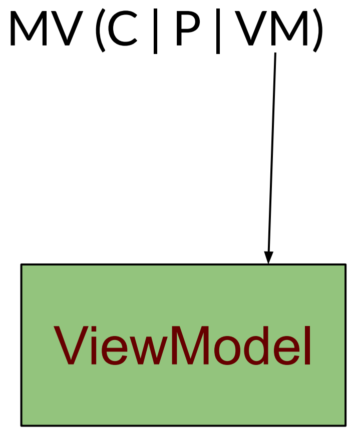
In Android, Activity/Fragments can act as both View and Controller

MV* Patterns



Presenter is a simple java class that do not contain any UI components, it just manipulates data from model and display in on View.

MV* Patterns



They provide data and functionality to be used by views. They are what define the structure and behavior of the actual application you are building

**** Many View can be mapped to one View-Model**

MVC vs MVP vs MVVM

Which one to follow?

Activity/Fragment/View should be Business logic free

Unit Test

“ A software testing method by which individual units of code, are tested to determine whether they are fit for use.

The smallest testable part of an application (Classes and Methods).

Anatomy of a unit test

- **Arrange** all necessary preconditions and inputs.
- **Act** on the object or method under test.
- **Assert** that the expected results have occurred.

Benefits of a unit test

- Find problems early.
- Facilitate refactoring.
- Simplify integration.
- Document code usage.

Types of Android unit test

- Instrumented unit test
- Local unit test

Instrumented unit test

- Runs on device or emulator
- Actually affects the device

Local unit test

- Runs on JVM
- No need for device or emulator
- Faster than instrumented unit test

Android Unit Testing Tools & Framework

- JUnit
- Mockito
- PowerMock
- Robolectric
- Espresso
- UI Automator

JUnit

Gradle - testImplementation 'junit:junit:4.12'

JUnit Annotations

@Test

@Before

@After

@BeforeClass

@AfterClass

@Ignore

JUnit statement assertions

assertFalse(condition)

assertEquals(expected, actual, tolerance)

assertNull(object)

assertNotNull(object)

assertSame(expected, actual)

Mockito

“ Objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

Mocking reasons

Reason - Your Object have external dependencies

Mockito is a Java framework allowing the creation of test mock objects in automated unit tests

dependencies { testImplementation "org.mockito:mockito-core:2.11.0" }

Mockito features

- Mocking
- Stubbing
- Argument matchers
- Verifying number of invocations
- Verifying order of invocations

Mockito limitations

- Cannot mock final classes
- Cannot mock static methods
- Cannot mock final methods
- Cannot mock equals(), hashCode()

PowerMock

PowerMock is a framework that extends other mock libraries such as Mockito with more powerful capabilities.

Enable mocking of static methods, constructors, final classes and methods, private methods, removal of static initializers and more.

PowerMock

```
testImplementation 'org.powermock:powermock-module-junit4:1.6.4'

testImplementation 'org.powermock:powermock-module-junit4-rule:1.6.4'
testImplementation 'org.powermock:powermock-api-mockito:1.6.4'
testImplementation 'org.powermock:powermock-classloading-xstream:1.6.4'
```

Robolectric

Unit Testing framework which allows Android application to be tested on JVM without an emulator or device.

Robolectric provides implementation of Android SDKs by rewriting Android core libraries using shadow classes

Robolectric

Gradle -

testImplementation “org.robolectric:robolectric:latestVersion”

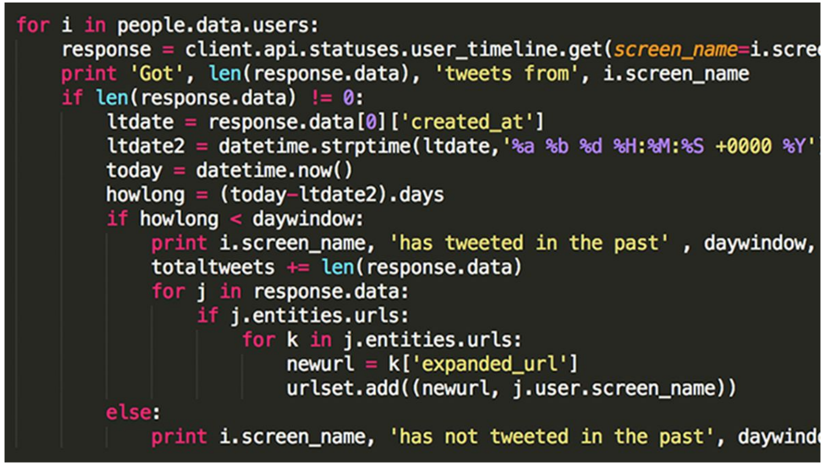
Robolectric handles inflation of views, resource loading, and lots of other stuff that’s implemented in native C code on Android devices.

**Robolectric is not an integration test framework, i.e., you cannot not test the interaction of Android components with it.

Writing Testable Code

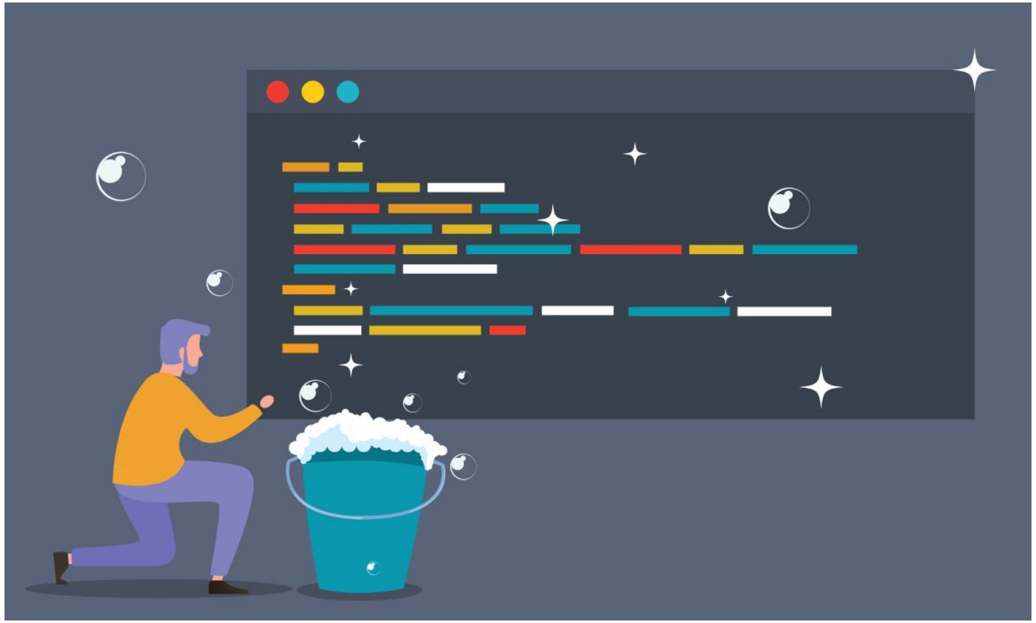
Or how not to lose your mind coding

Writing Tests can be hard!



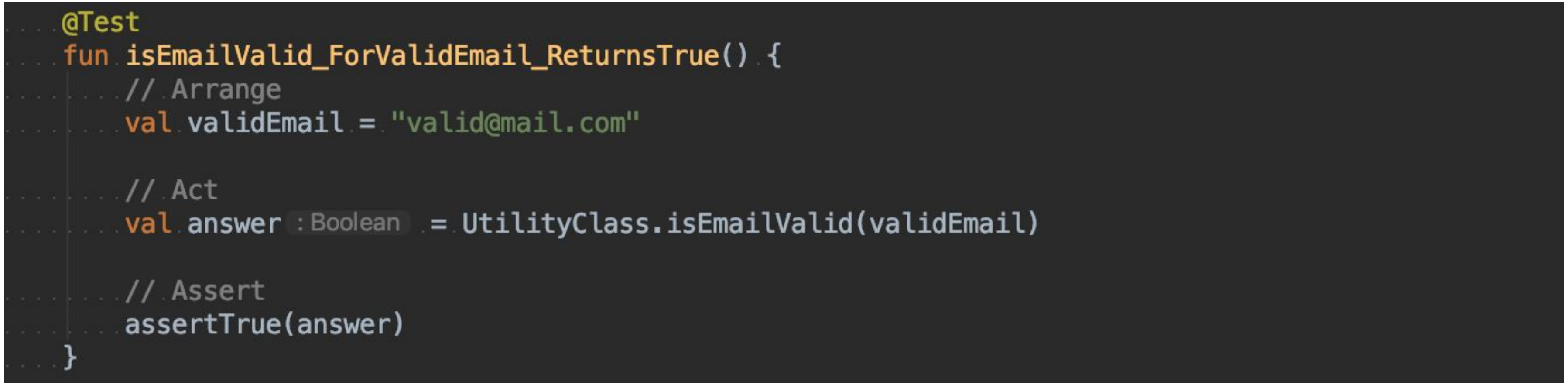
```
for i in people.data.users:
    response = client.api.statuses.user_timeline.get(screen_name=i.scre
    print 'Got', len(response.data), 'tweets from', i.screen_name
    if len(response.data) != 0:
        ltdate = response.data[0]['created_at']
        ltdate2 = datetime.strptime(ltdate,'%a %b %d %H:%M:%S +0000 %Y'
        today = datetime.now()
        howlong = (today-ltdate2).days
        if howlong < daywindow:
            print i.screen_name, 'has tweeted in the past' , daywindow,
            totaltweets += len(response.data)
            for j in response.data:
                if j.entities.urls:
                    for k in j.entities.urls:
                        newurl = k['expanded_url']
                        urlset.add((newurl, j.user.screen_name))
        else:
            print i.screen_name, 'has not tweeted in the past', daywind
```

When done right, results in a clean, easy to maintain codebase



Good Unit Test

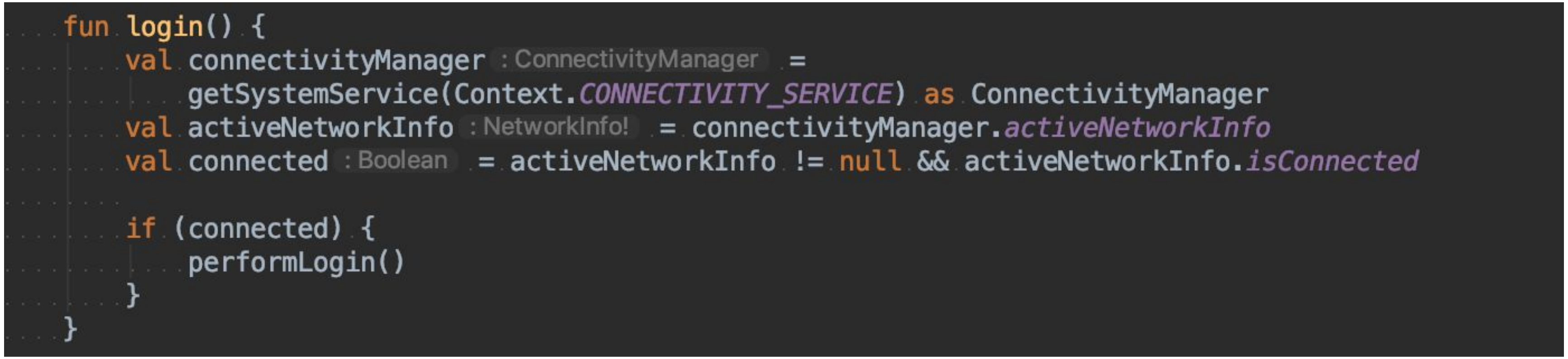
- Easy to write
- Readable
- Reliable
- Fast
- Truly Unit



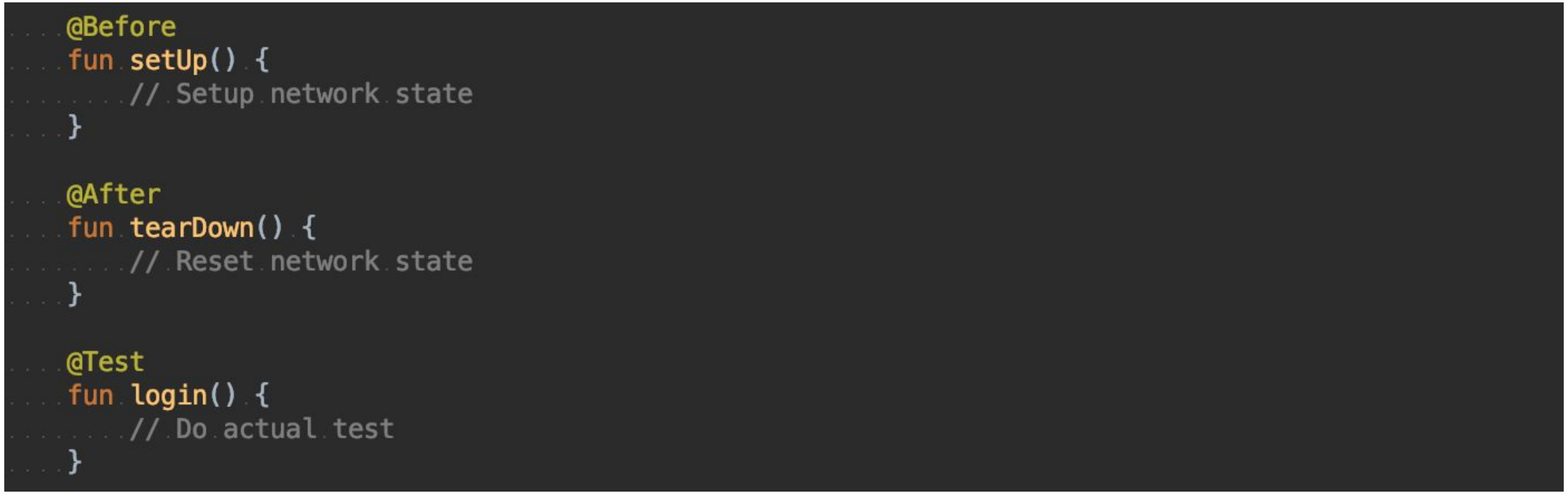
```
@Test
fun isEmailValid_ForValidEmail_ReturnsTrue() {
    // Arrange
    val validEmail = "valid@mail.com"

    // Act
    val answer : Boolean = UtilityClass.isEmailValid(validEmail)

    // Assert
    assertTrue(answer)
}
```

Testable Code

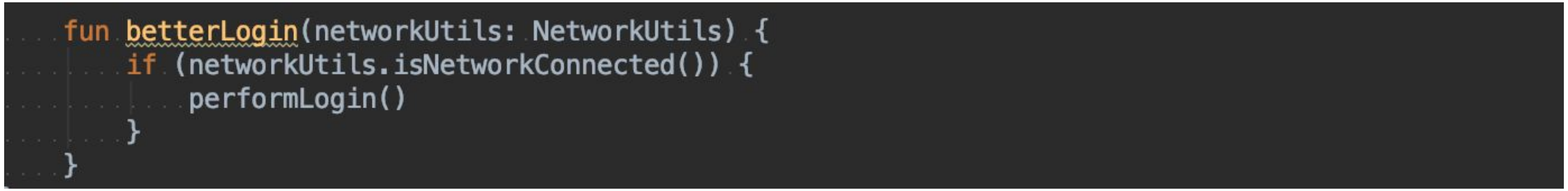
Deterministic



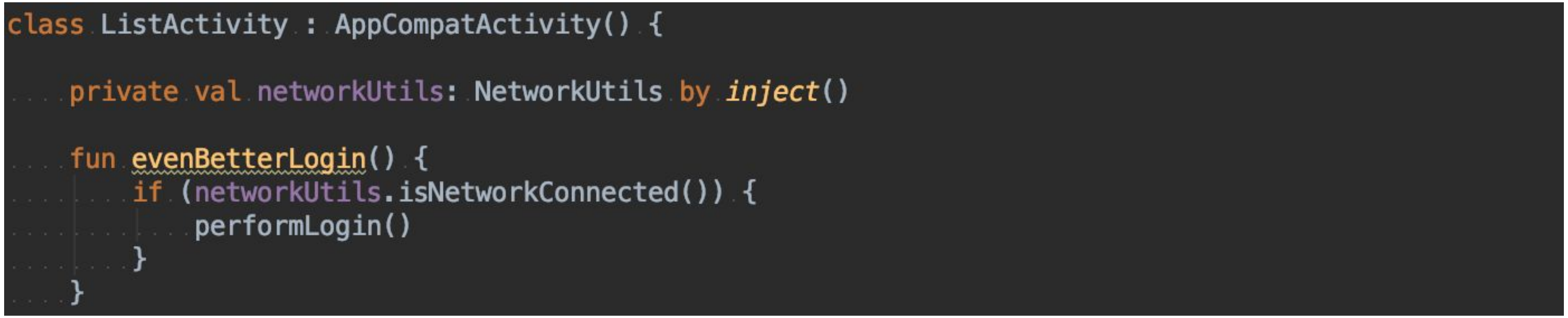
```
fun login() {
    val connectivityManager : ConnectivityManager =
        getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
    val activeNetworkInfo : NetworkInfo! = connectivityManager.activeNetworkInfo
    val connected : Boolean = activeNetworkInfo != null && activeNetworkInfo.isConnected

    if (connected) {
        performLogin()
    }
}
```



```
    @Before
    fun setUp() {
        // Setup network state
    }

    @After
    fun tearDown() {
        // Reset network state
    }

    @Test
    fun login() {
        // Do actual test
    }
```

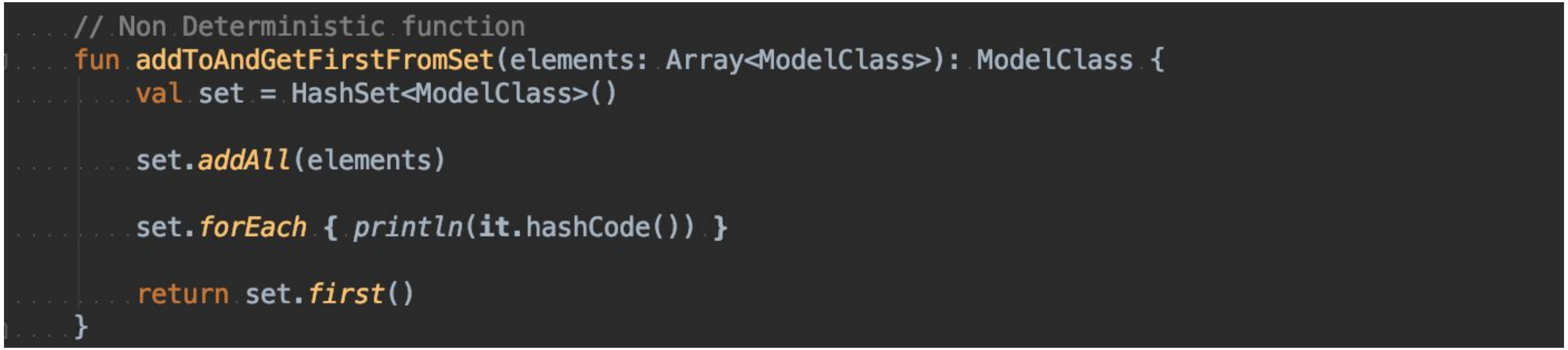


```
    fun betterLogin(networkUtils: NetworkUtils) {
        if (networkUtils.isNetworkConnected()) {
            performLogin()
        }
    }
```



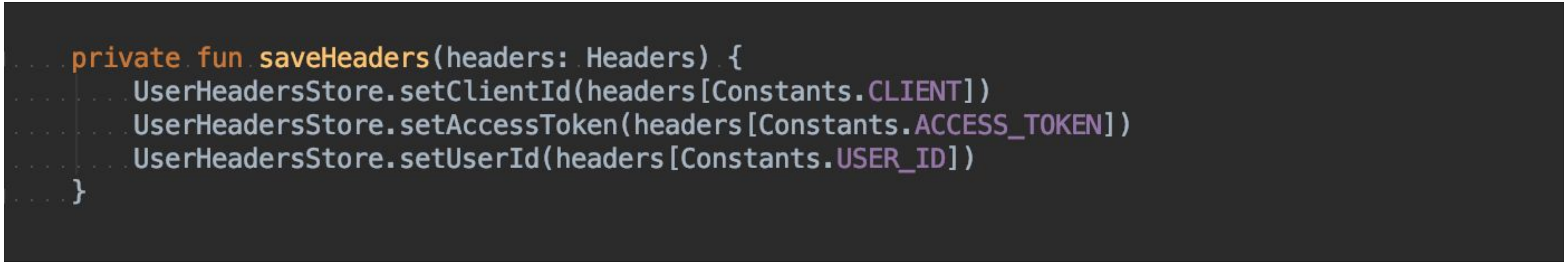
```kotlin
class ListActivity : AppCompatActivity() {

    private val networkUtils: NetworkUtils by inject()

    fun evenBetterLogin() {
        if (networkUtils.isNetworkConnected()) {
            performLogin()
        }
    }
```

Can you guess why this code is non deterministic?



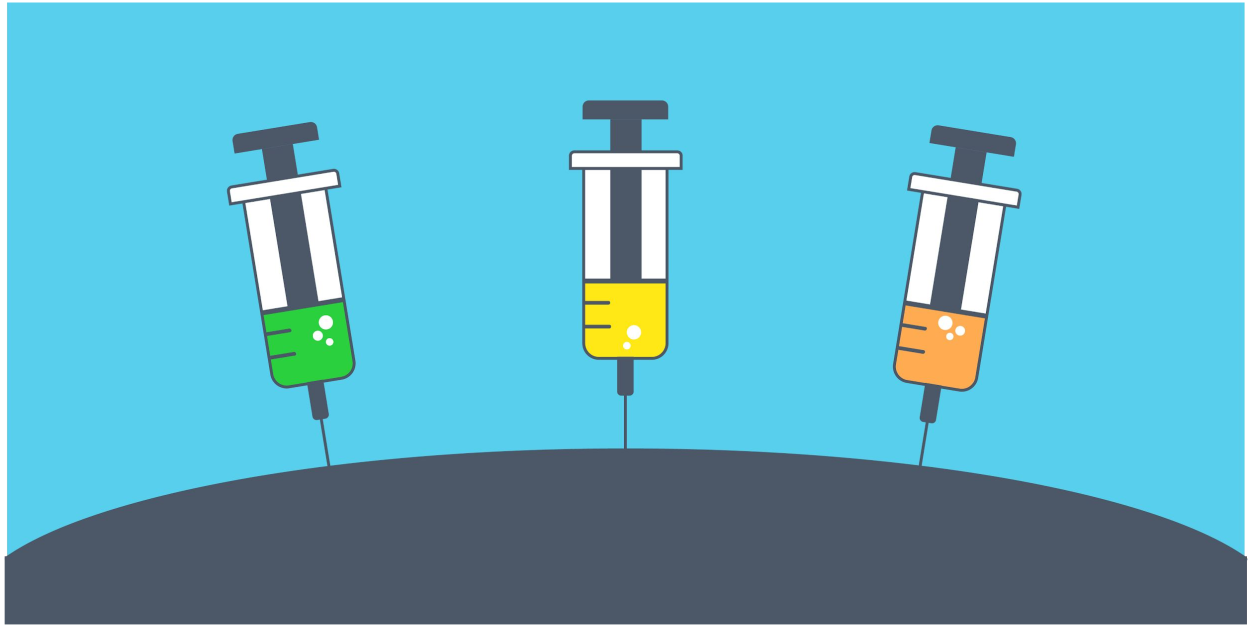
```
// Non Deterministic function
fun addToAndGetFirstFromSet(elements: Array<ModelClass>): ModelClass {
    val set = HashSet<ModelClass>()

    set.addAll(elements)

    set.forEach { println(it.hashCode()) }

    return set.first()
}
```

Side Effects

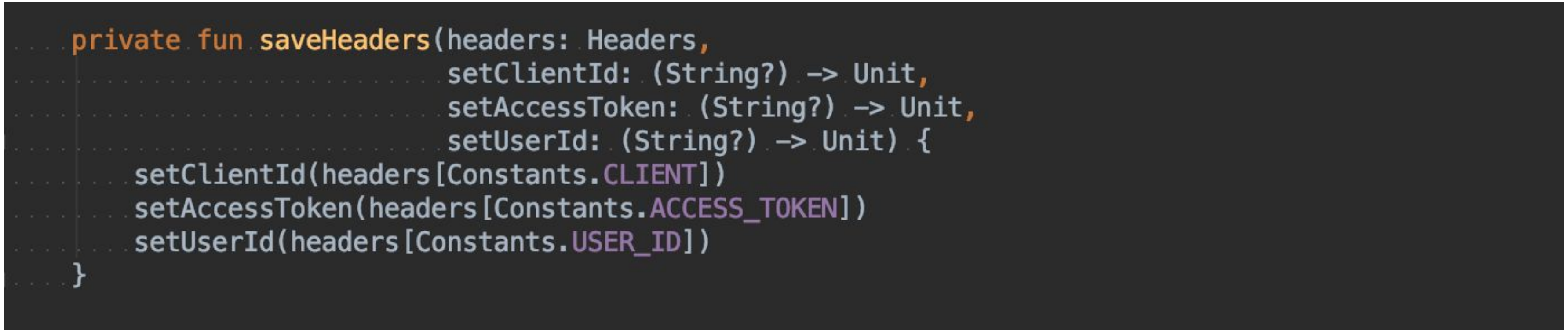


```
    private fun saveHeaders(headers: Headers) {
        UserHeadersStore.setClientId(headers[Constants.CLIENT])
        UserHeadersStore.setAccessToken(headers[Constants.ACCESS_TOKEN])
        UserHeadersStore.setUserId(headers[Constants.USER_ID])
    }
```

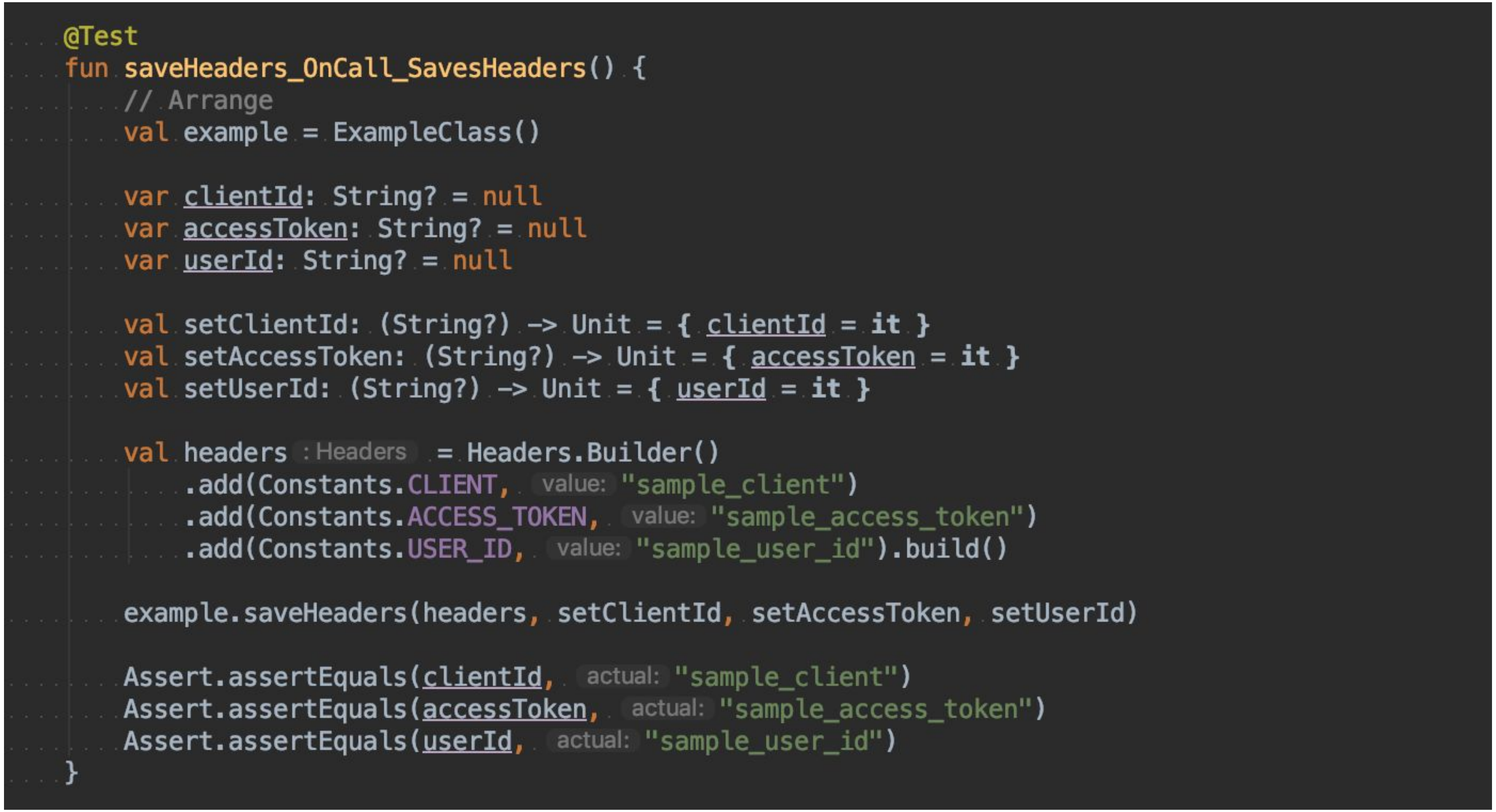
Solution?



Higher order functions



```
    private fun saveHeaders(headers: Headers,
                            setClientId: (String?) -> Unit,
                            setAccessToken: (String?) -> Unit,
                            setUserId: (String?) -> Unit) {
        setClientId(headers[Constants.CLIENT])
        setAccessToken(headers[Constants.ACCESS_TOKEN])
        setUserId(headers[Constants.USER_ID])
    }
```



```kotlin
@Test
fun saveHeaders_OnCall_SavesHeaders() {
    // Arrange
    val example = ExampleClass()

    var clientId: String? = null
    var accessToken: String? = null
    var userId: String? = null

    val setClientId: (String?) -> Unit = { clientId = it }
    val setAccessToken: (String?) -> Unit = { accessToken = it }
    val setUserId: (String?) -> Unit = { userId = it }

    val headers : Headers  = Headers.Builder()
        .add(Constants.CLIENT,  value: "sample_client")
        .add(Constants.ACCESS_TOKEN,  value: "sample_access_token")
        .add(Constants.USER_ID,  value: "sample_user_id").build()

    example.saveHeaders(headers, setClientId, setAccessToken, setUserId)

    Assert.assertEquals(clientId,  actual: "sample_client")
    Assert.assertEquals(accessToken,  actual: "sample_access_token")
    Assert.assertEquals(userId,  actual: "sample_user_id")
}
```

Rule of Thumb

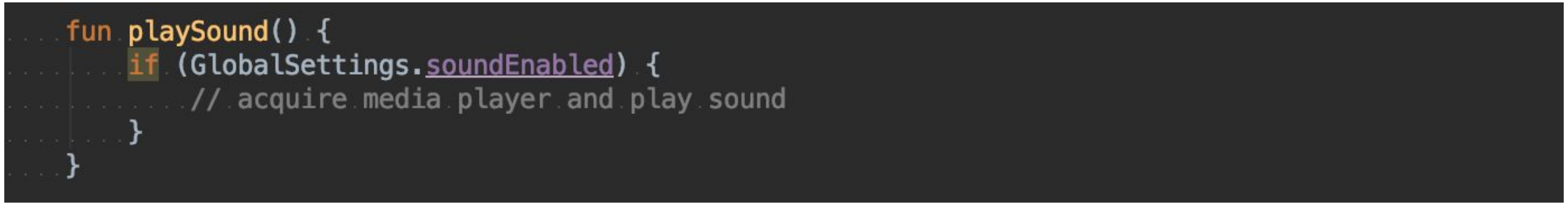
- Write deterministic code
- Minimize side effects
- In essence, write pure functions

Can impurity really be removed?

- As much as possible, extract it out and keep it contained

Red Flags

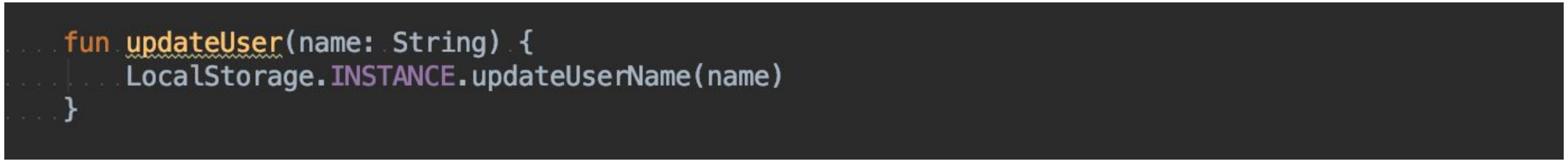
- Static properties and Fields



```
    fun playSound() {
        if (GlobalSettings.soundEnabled) {
            // acquire media player and play sound
        }
    }
```

Red Flags

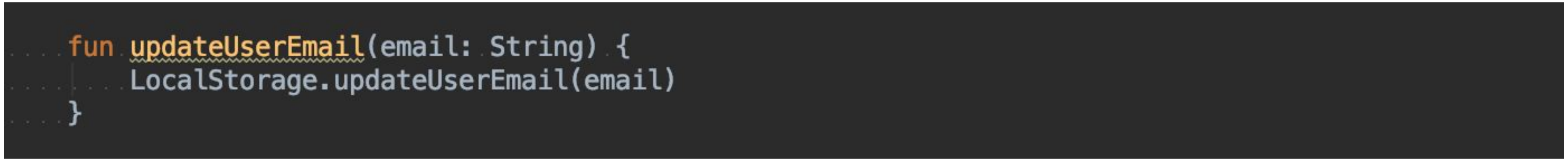
- Singletons



```
    fun updateUser(name: String) {
        LocalStorage.INSTANCE.updateUserName(name)
    }
```

Red Flags

- Static Methods


```
    fun updateUserEmail(email: String) {
        LocalStorage.updateUserEmail(email)
    }
```

What is TDD

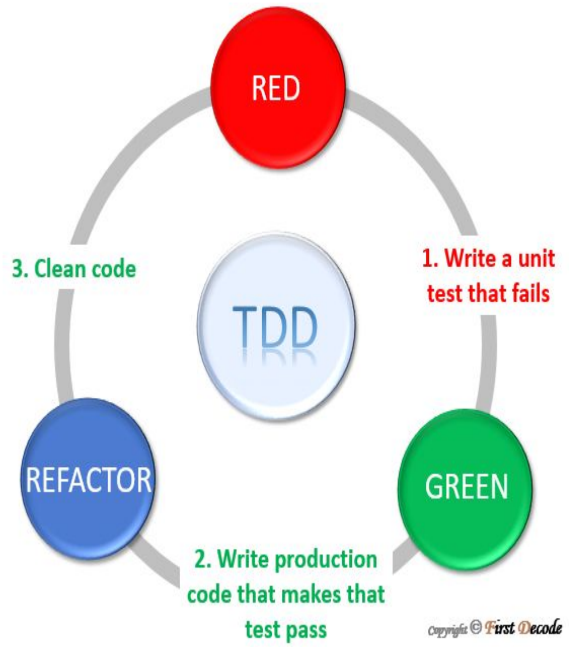
Test-driven development (TDD) is an approach for software development where you write tests first, then use those tests to drive the design and development of your software application.

TDD Cycle

Red – think about what you want to develop

Green – think about how to make your tests pass

Refactor – think about how to improve your existing implementation


RED
1. Write a unit test that fails
TDD
GREEN
2. Write production code that makes that test pass
REFACTOR
3. Clean code
Copyright © First Decode

DEMO

References

Github Link -

https://github.com/Niharika8059/UnitTesting-MVVM-Kotlin-Coroutines-Sample

Medium link -

https://medium.com/1mgofficial/unit-testing-in-mvvm-kotlin-databinding-ba3d4ea08f0e

Resources

https://semaphoreci.com/community/tutorials/stubbing-and-mocking-with-mockito-2-and-junit

https://android.jlelse.eu/better-testing-with-mvvm-ae74d4d872bd

https://medium.com/mindorks/unit-testing-for-viewmodel-19f4d76b20d4

“Writing a test is simple, but writing a code that can be tested is not so simple”

QUESTIONS?