# The Android Prep Lab (Kotlin)

## Part - 9 of 10
## Testing Android Apps

1

# What is Unit Testing in Android?

Unit testing is a software testing method where individual components of an application are tested in isolation. In Android, unit tests focus on testing business logic, utility functions, and ViewModel logic using frameworks like JUnit and Mockito.

# What are the different types of testing in Android?

Android testing is broadly categorized into:

- **Unit Testing**: Tests individual functions/methods (e.g., using JUnit, Mockito)
- **UI Testing**: Tests user interface interactions (e.g., using Espresso, UIAutomator)
- **Integration Testing**: Ensures that multiple modules work together correctly.
- **End-to-End Testing**: Validates the entire app flow as a user would experience.

2

# What is JUnit, and how is it used in Android?

JUnit is a Java-based testing framework used for unit testing. In Android, JUnit (along with JUnit 4 or JUnit 5) is used for writing test cases to validate the correctness of logic in ViewModels, repositories, and other classes.

**Example:**

```
@RunWith(JUnit4::class)
class ExampleUnitTest {
    @Test
    fun addition_isCorrect() {
        assertEquals(4, 2 + 2)
    }
}
```

3

# What is Mockito, and why is it used?

Mockito is a mocking framework used to simulate dependencies in unit tests. It helps in testing classes without actually depending on real implementations like databases or network calls.

**Example:**

```
@RunWith(MockitoJUnitRunner::class)
class UserRepositoryTest {
    @Mock
    private lateinit var apiService: ApiService

    @Test
    fun testGetUserDetails() {
        `when`(apiService.getUser("123")).thenReturn(User("123", "John"))
        assertEquals("John", apiService.getUser("123").name)
    }
}
```

4

# What is Espresso, and how is it used for UI Testing?

Espresso is a UI testing framework that allows writing concise and reliable Android UI tests. It ensures smooth interaction with UI components and handles synchronization automatically.

## Example:

```
@Test
fun testButtonClick() {
    onView(withId(R.id.button)).perform(click())
    onView(withId(R.id.textView)).check(matches(withText("Clicked!")))
}
```

5

# How do you perform dependency injection in tests?

To replace dependencies in tests, use Dagger's @Inject annotation or Hilt's @TestInstallIn to provide test-specific implementations of dependencies.

# What is Robolectric, and how is it different from Espresso?

Robolectric is a framework for running unit tests in a JVM environment instead of an emulator or device. It is used for testing Android framework-dependent logic without requiring an actual Android runtime.

Espresso, on the other hand, is designed for UI testing and runs tests on real devices/emulators.

6

# How do you handle asynchronous operations in testing?

For testing coroutines, use runBlockingTest (Kotlin Coroutines Test) or TestDispatcher.

Example:

```
@Test
fun testAsyncCall() = runTest {
    val result = asyncFunction()
    assertEquals("Success", result)
}
```

7

# How do you set up test cases for ViewModels?

Mock dependencies using Mockito or use real instances with dependency injection.

Example:

```
@Test
fun `test ViewModel live data`() {
    val viewModel = MyViewModel()
    viewModel.loadData()
    assertEquals("Data Loaded", viewModel.liveData.value)
}
```

8

# What are best practices for writing Android tests?

- Keep tests independent and isolated.
- Use dependency injection to provide test doubles.
- Write descriptive test names.
- Run tests frequently to ensure stability.
- Use @Before and @After annotations to set up and clean up resources.

https://www.linkedin.com/in/pavankreddy92/

# Follow for More