UNIT TESTING(JUNIT AND MOCKITO)

by

Irtifa Haider

Computer Science and Engineering Department
Jahangirnagar University
Savar, Dhaka-1342, Bangladesh.

October 21, 2024

Contents

Chapter 1

Unit Testing

1.1 What is Unit Testing

Unit testing involves testing small, isolated units of code, like a method or a class, to ensure they behave as expected. In Android, unit tests can be written using **JUnit**, which provides a framework to write and run tests, and **Mockito**, which helps in mocking external dependencies that your code interacts with (e.g., Android system components like `SharedPreferences`).
For example, if you have a class that performs some math operations, you'll write tests to ensure that each method (like `add()`, `subtract()`) behaves as expected.

1.2 Why Unit Testing is Important

- **Early Bug Detection**: It allows bugs to be caught early in the development process, saving time and effort in the long run.
- **Confidence in Code Changes**: Having a suite of unit tests ensures that future changes in code won't break existing functionality.
- **Code Quality**: Unit testing encourages writing modular, clean, and testable code.

1.3 Tools Used for Unit Testing in Android

In this guide, we will use:

- **JUnit**: A framework for writing and running unit tests in Java.
- **Mockito**: A popular framework for creating mock objects, which are used to simulate the behavior of real objects during testing.

1.4 Setup JUnit and MOckito Dependencies

Follow the images and read the captions for instructions:

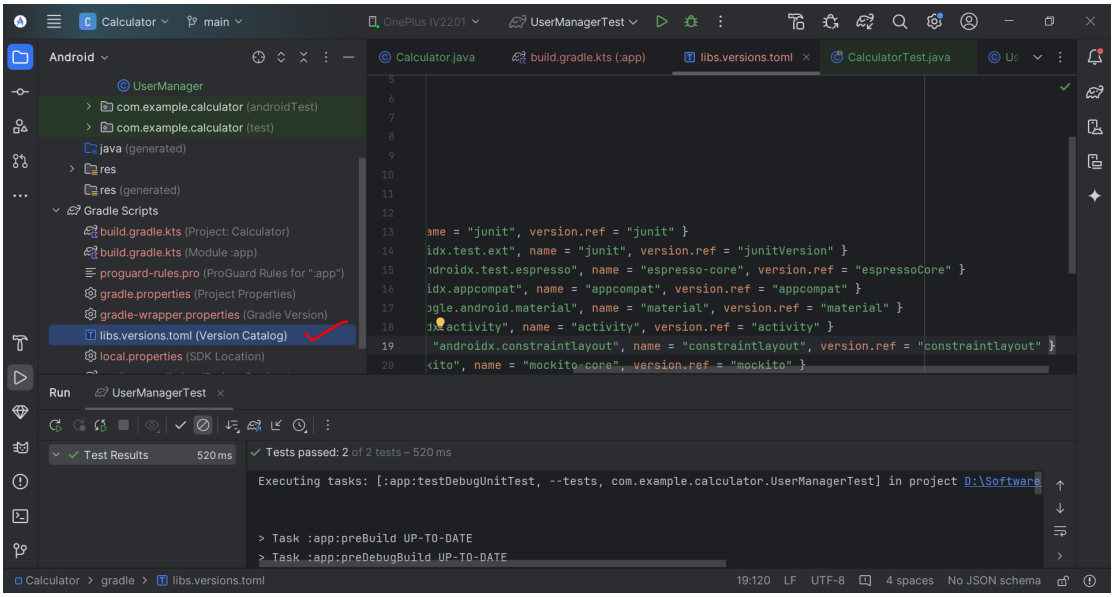

Figure 1.1: go to **libs.versions.toml** file

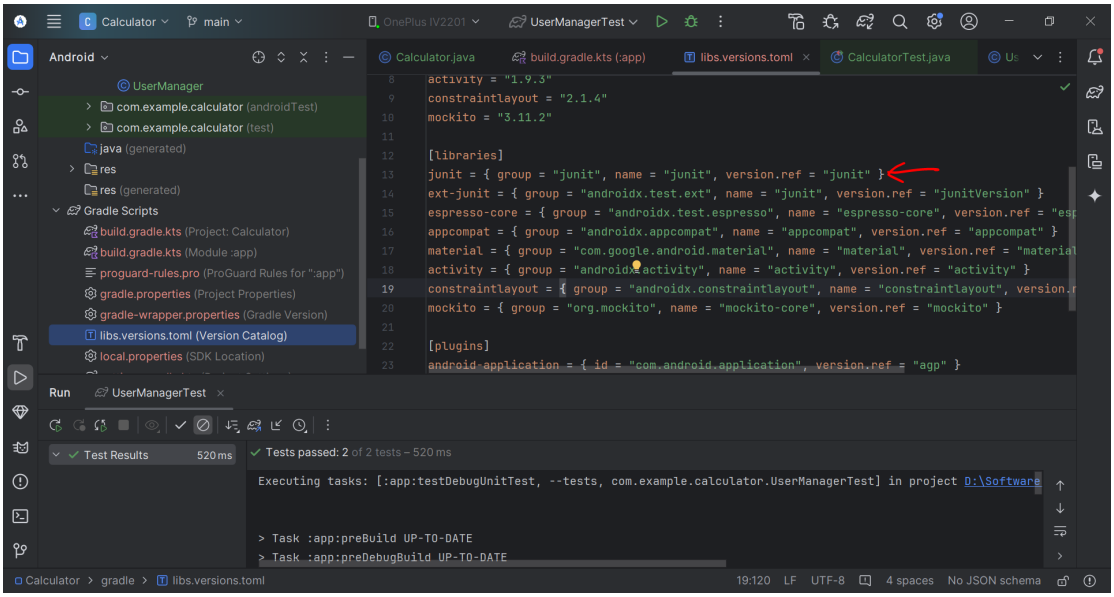

Figure 1.2: Add JUnit to your libs.versions.toml under [versions] and [libraries]

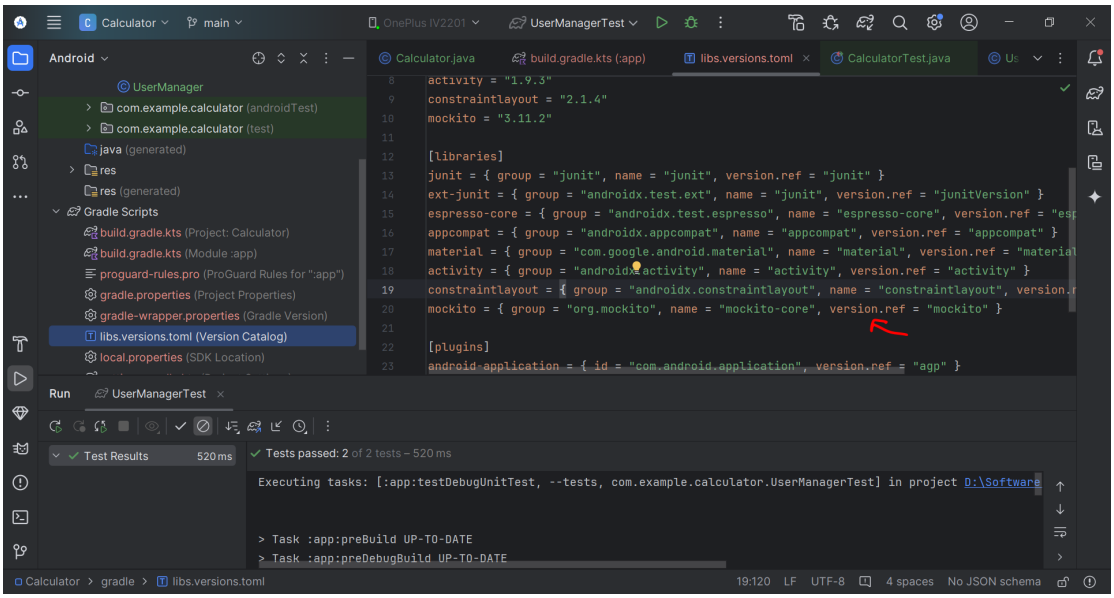

Figure 1.3: Add Mockito to your libs.versions.toml under [versions] and [libraries]

1.5 Conclusion

Mocking is a powerful technique for isolating your code during testing. By using **Mockito**, you can create ”fake” versions of your dependencies, allowing you to focus on testing your code logic without involving real components like Android’s system, databases, or networks. This makes your tests faster, more reliable, and easier to write.

By following this guide, you now have a basic understanding of how to use Mockito in Android unit tests to mock dependencies like `SharedPreferences`. Keep practicing, and soon, mocking will become second nature in your testing!

1.6 Best Practices for Writing Unit Tests

- **Test One Thing at a Time**: Each test should focus on testing one method or one behavior.
- **Use Meaningful Test Names**: The names of your test methods should clearly indicate what is being tested.
- **Keep Tests Independent**: Each test should be able to run independently of other tests. Avoid shared state between tests.
- **Test Edge Cases**: Test edge cases, such as passing negative numbers or zero, to ensure your code handles these scenarios correctly.

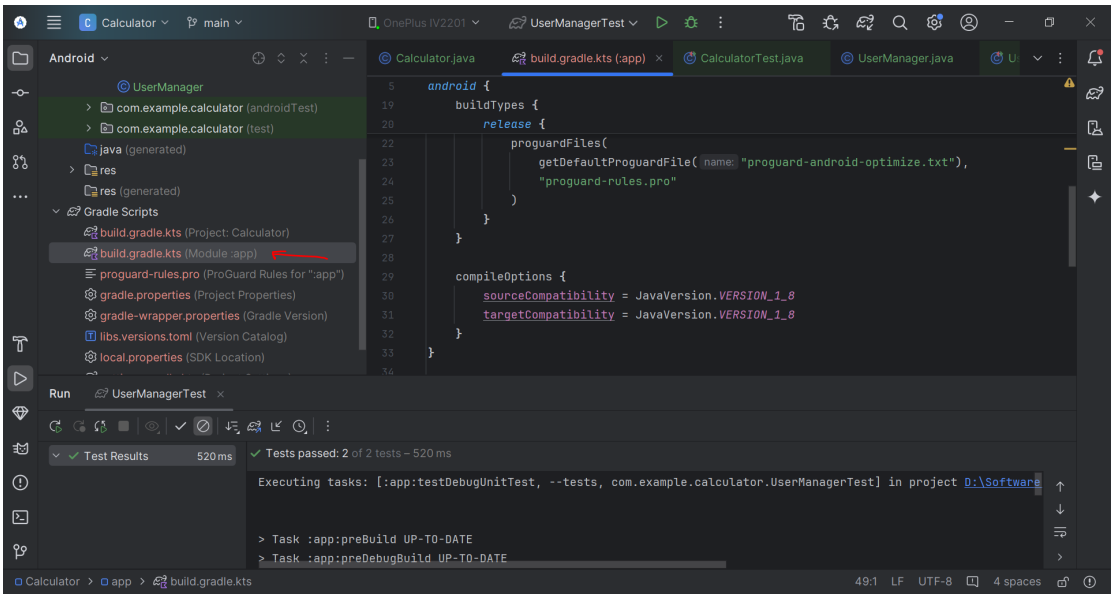

Figure 1.4: Now, update your **build.gradle** file

1.7 Conclusion

In this guide, we walked through the basics of writing unit tests using **JUnit**. Here's a summary of what we covered:

- How to set up JUnit in an Android project.
- Writing and running simple JUnit tests.
- Understanding the structure of a unit test.

By following these steps, you'll be able to write and run unit tests in your Android project, ensuring that your code is reliable and behaves as expected.

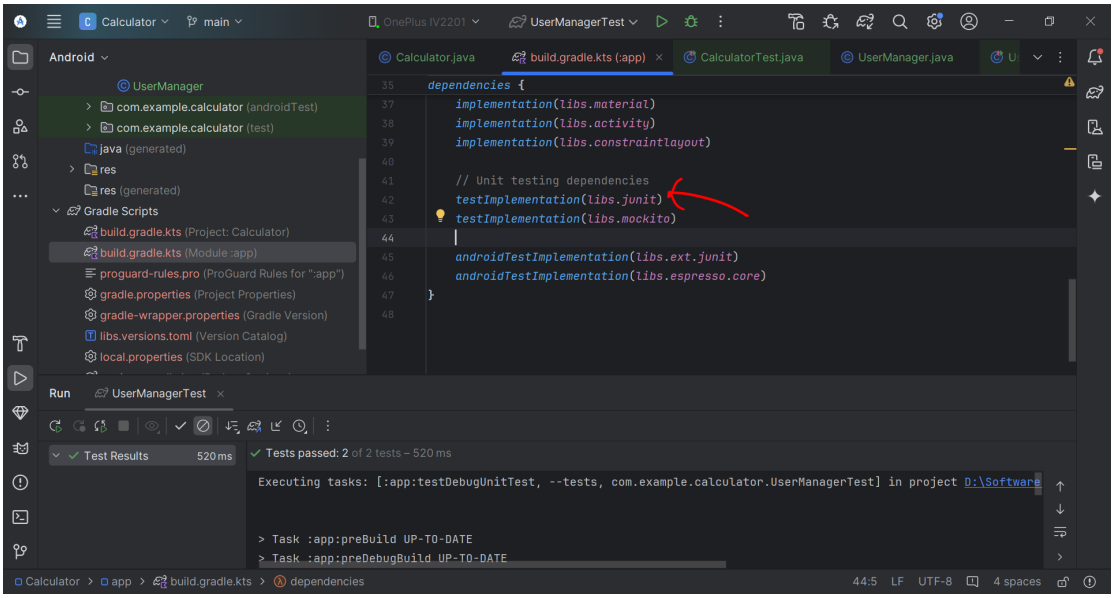

Figure 1.5: The dependencies section with unit testing libraries added for JUnit and Mockito file

Chapter 2

JUnit

2.1 What is JUnit?

JUnit is a popular testing framework for Java that is used for writing and running unit tests.

2.2 Why is JUnit?

- As we are building LivSmart project using Java in Android Studio, JUnit is already integrated into the environment, simplifying the setup process.
- JUnit tests are lightweight and fast because they run on the JVM and don't need an emulator or actual Android device for basic logic testing. As a result, you get instant feedback on whether your methods are working as intended.
- JUnit works seamlessly with Mockito, a powerful mocking library. In the later section, I presented Mockito.
- JUnit tests can be easily integrated into a Continuous Integration (CI) pipeline.
- We will be able to adopt TDD and get fast feedback during development, improving the quality of code.

2.3 Demonstration of Unit Testing with JUnit

Let's start with a simple example. We'll write unit tests for a basic `Calculator` class that has two methods: `add()` and `subtract()`.

2.3.1 Step 1: Create the `Calculator` Class

First, create the `Calculator` class inside the `app/src/main/java/com/example/calcula` folder:

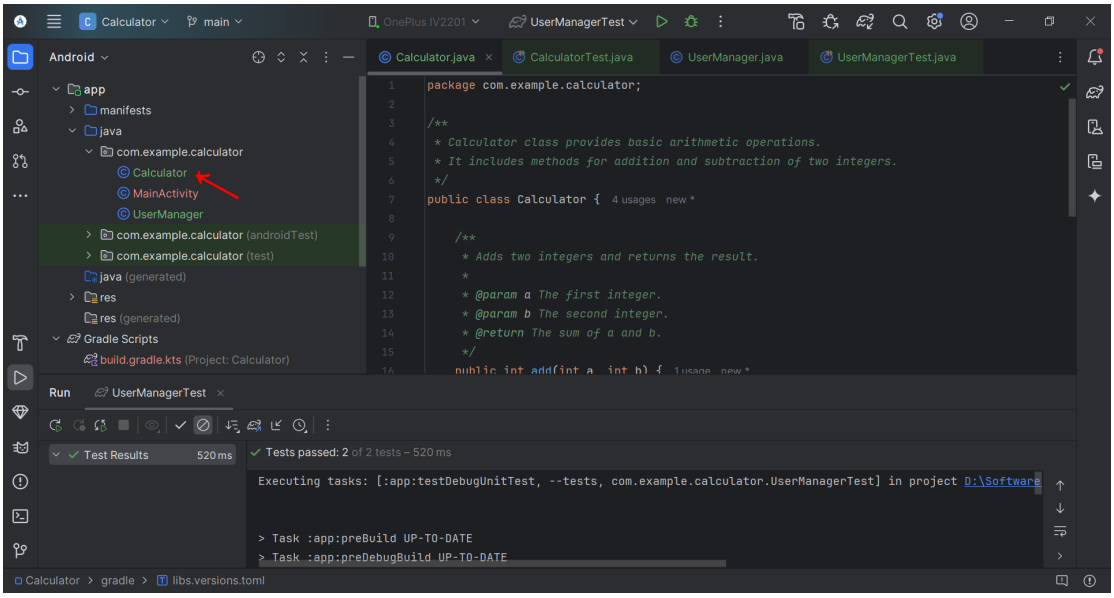

Figure 2.1: Create the **Calculator** class

2.3.2 Step 2: Create a Test Class

Now, we'll write a test class for the `Calculator` class using JUnit. Create a new test file inside the `app/src/test/java/com/example/calculator/` folder and name it `CalculatorTest1`.

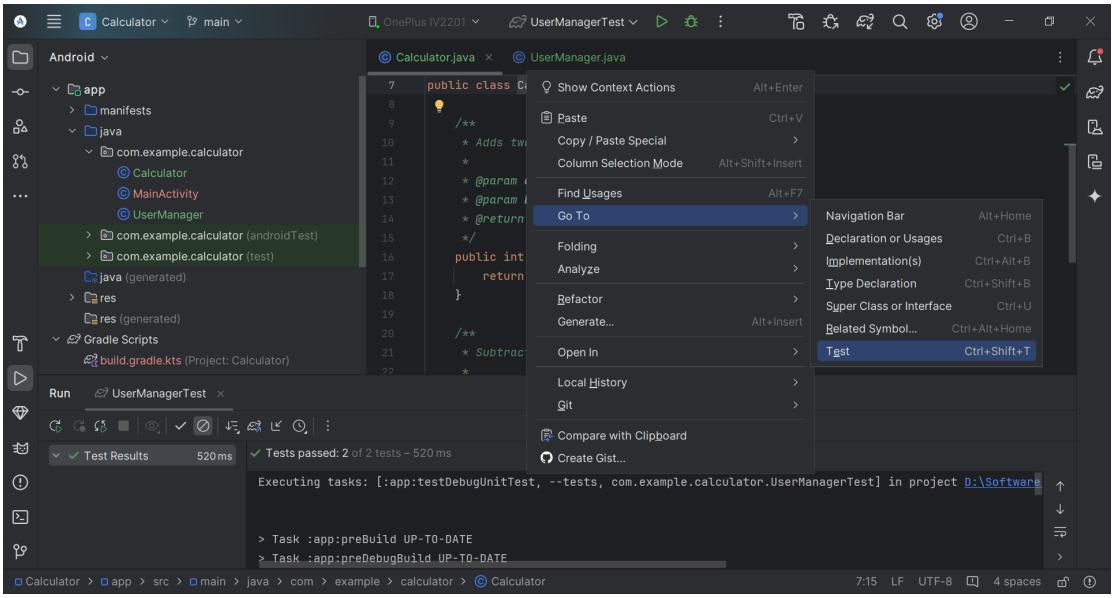

Figure 2.2: The context menu in Android Studio for the Calculator class, highlighting the option to create or view tests for the class

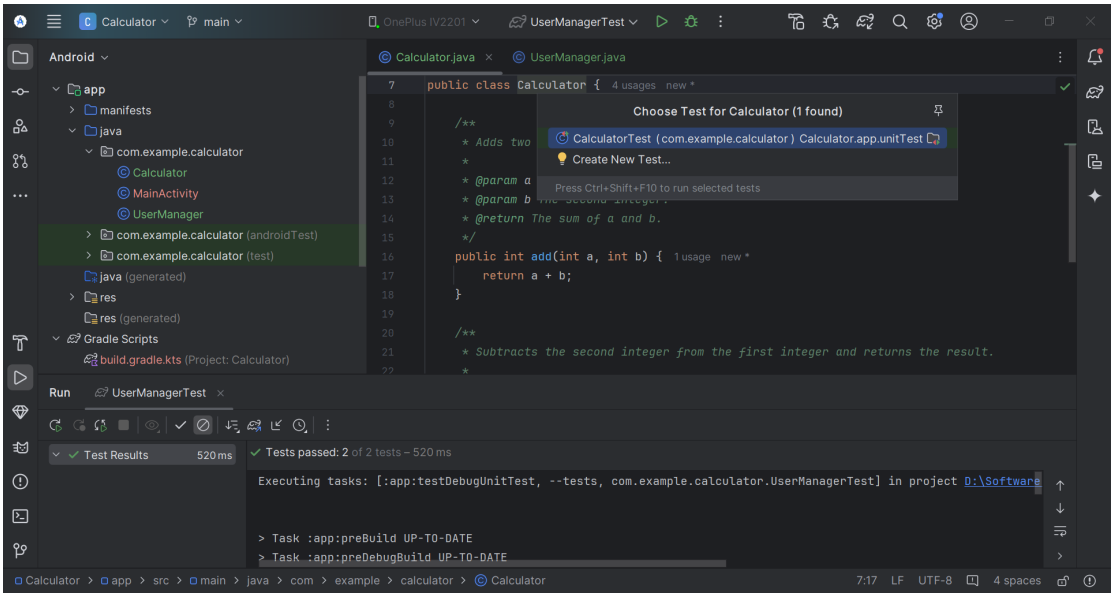

Figure 2.3: A pop-up in Android Studio for selecting or creating a test for the Calculator class

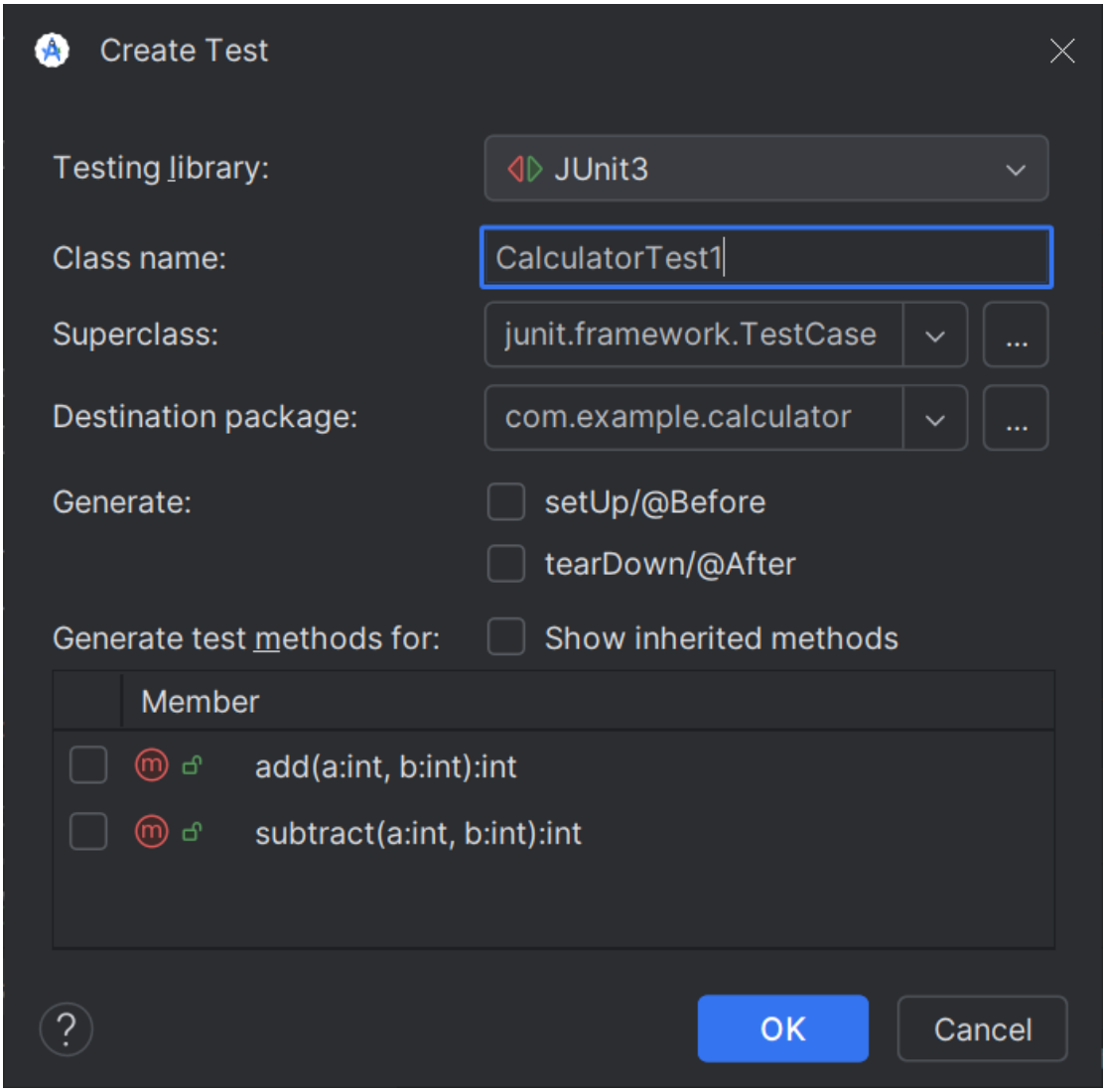

Figure 2.4: A dialog box in Android Studio for creating a new test class named CalculatorTest1

2.3.3 Step 3: Run the Test

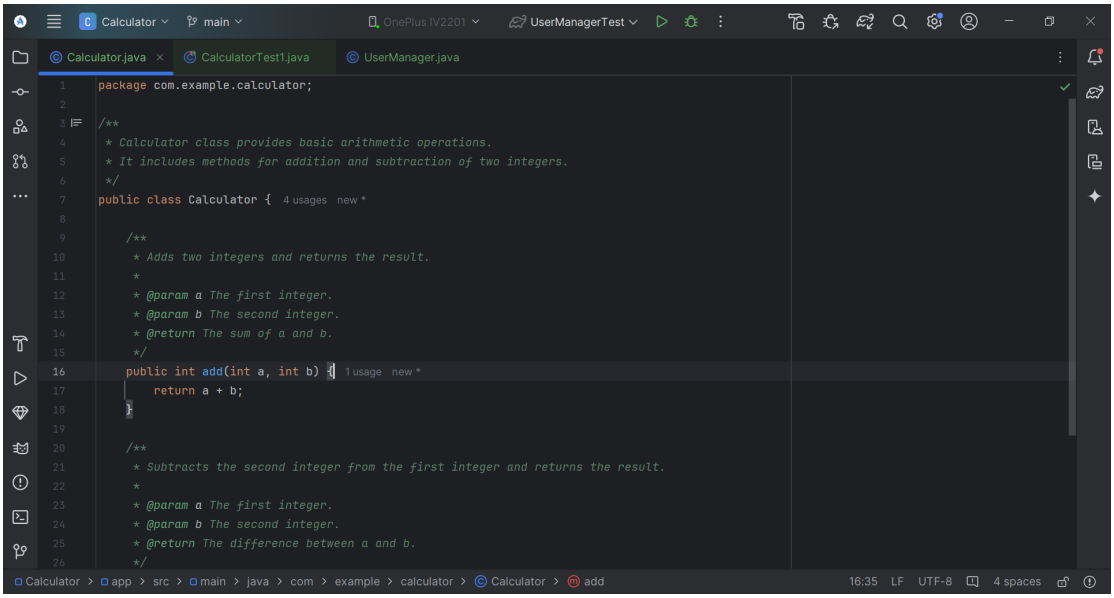

Figure 2.5: The Calculator class in Android Studio showing the add method that adds two integers and returns the result.

Once the test class is ready, you can run the tests by following these steps:

1. Right-click on the `CalculatorTest.java` file.
2. Select **Run 'CalculatorTest'**.

JUnit will run the tests and display the results in the **Run** window in Android Studio.

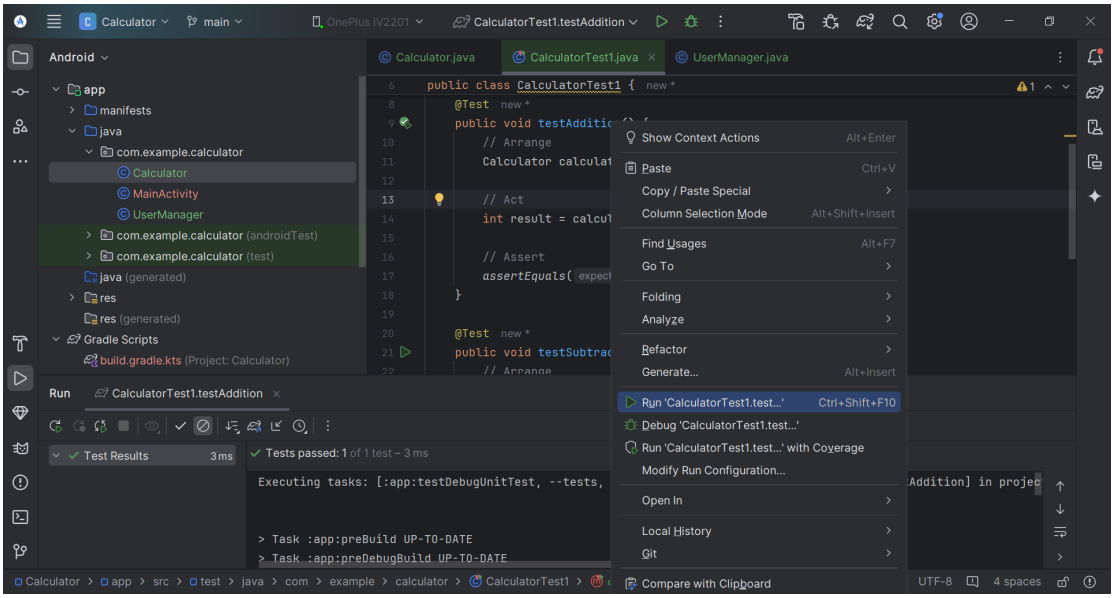

Figure 2.6: A test method for testing the addition functionality of the Calculator class in Android Studio.

2.4 Understanding the Test

Here's what happens in the test above:

- **Test Case**: Each method annotated with `@Test` is a test case.
- **Arrange**: We create an instance of the class that we are testing.
- **Act**: We call the method we want to test (e.g., `add()` or `subtract()`).
- **Assert**: We verify that the result is correct using the `assertEquals()` method.

Chapter 3

Mockito

3.1 The Problem: External Dependencies in Unit Tests

When you test a class, it often interacts with other components like databases, network calls, or Android-specific components like `SharedPreferences` or `Context`. These external components are difficult to use in tests because:

- They may depend on real Android functionality.
- They might slow down your tests (e.g., network calls, file storage).
- They make it hard to **isolate** the class you're trying to test.

3.2 Solution: Mocking with Mockito

Mocking is the process of creating "fake" versions of these dependencies so you can test your class without involving real components like databases or Android components.

Mockito is a library that makes mocking easy. It allows you to simulate the behavior of external components, giving you control over what they return when called.

3.3 Why Use Mockito?

Here's why mocking and Mockito are useful:

- **Test Classes in Isolation**: By mocking dependencies, you only focus on the class you're testing, ignoring external systems.
- **Faster and More Reliable Tests**: You don't need to interact with the real Android system or wait for network/database responses.
- **Control Behavior**: You can control what the mocked dependencies return, making it easy to test different scenarios.
- **Simulate Errors**: You can simulate failures or exceptions without needing a real database or API.

3.4 Demonestration of Unit Testing in Android Studio using Mockito

Let's say you have a class `UserManager` that interacts with `SharedPreferences` to save and retrieve a user's name. You want to test this class, but you don't want to use the real Android `SharedPreferences` because:

- It depends on Android.
- It interacts with the file system, making your tests slow.

Here's where Mockito comes in! You can create a mock version of `SharedPreferences` so that you don't need the real Android system to run your tests.

3.4.1 Step 1: Create the `UserManager` Class

This class saves and retrieves a username using `SharedPreferences`.

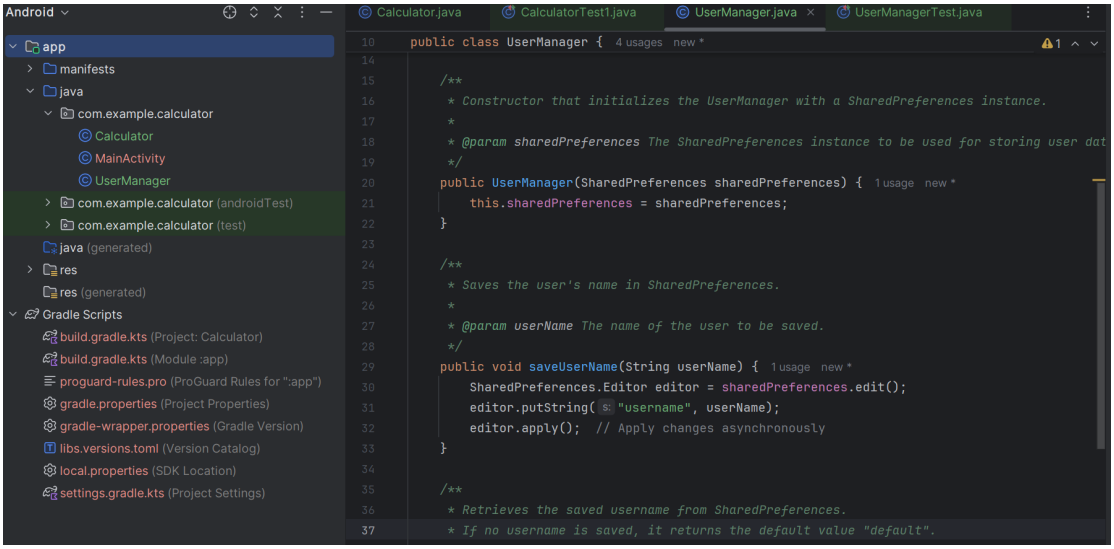

Figure 3.1: Create the **UserManager** class

3.4.2 Step 2: Write the Unit Test with Mockito

Now, let's write a unit test where we **mock** `SharedPreferences` using Mockito.

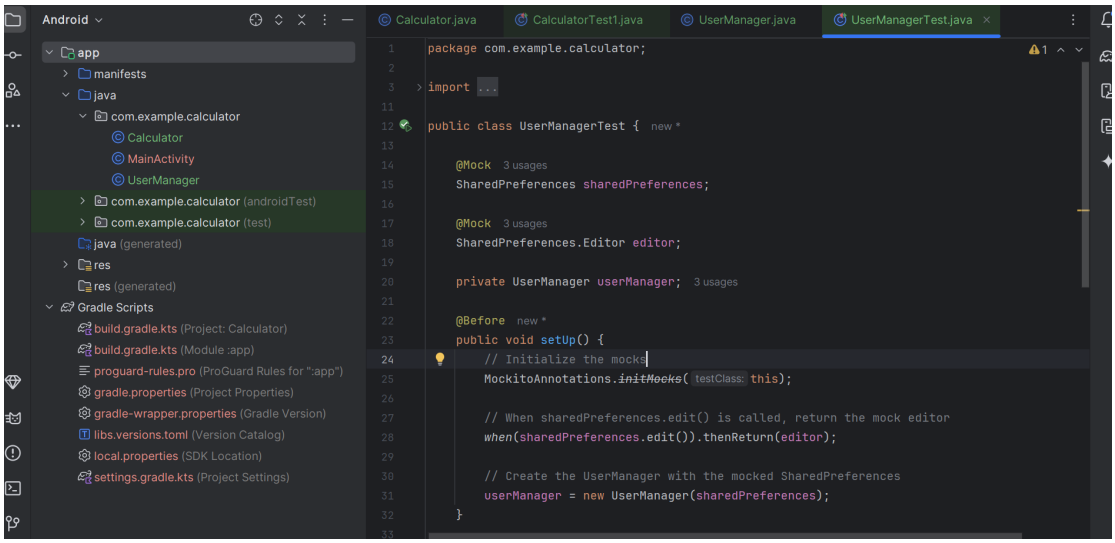

Figure 3.2: Create the **UserManageTest** class

3.4.3 Step 2: Run the Unit Test with Mockito

Now, let's run the test where using Mockito.

3.5 Understanding the Test

Here's what happens in the test above:

- **Mocks**:
  - `@Mock SharedPreferences sharedPreferences`: We mock `SharedPrefer` so that we don't interact with the real Android system.
  - `@Mock SharedPreferences.Editor editor`: We mock the `Editor` that `SharedPreferences.edit()` returns.
- **Mock Initialization**: `MockitoAnnotations.initMocks(this)`: Initializes the mocks (i.e., `sharedPreferences` and `editor`).
- **Mock Behavior**:

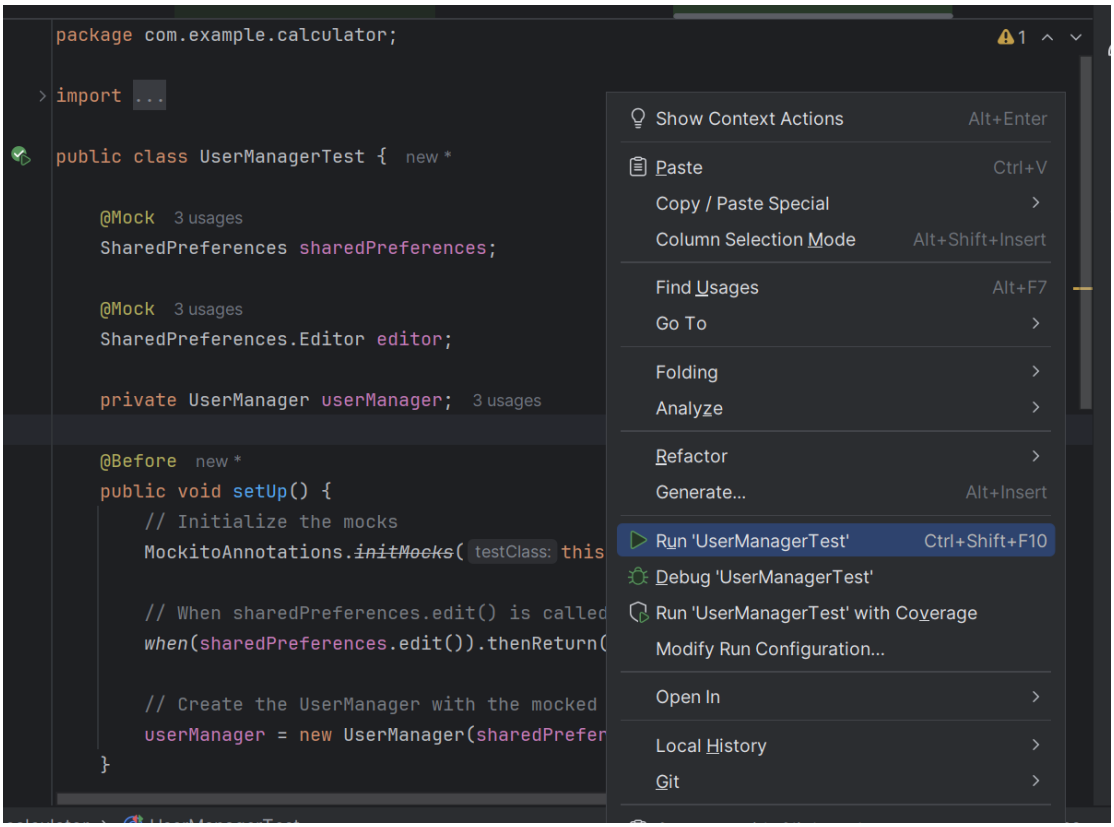

Figure 3.3: Run the **UserManageTest** class

  - `when(sharedPreferences.edit()).thenReturn(editor);`: This tells Mockito, "Whenever `edit()` is called on `SharedPreferences`, return the mock `editor`."
  - `when(sharedPreferences.getString("username", "default")).then` This tells Mockito to return "John" when `getString()` is called on `SharedPreferen`

- **Verification**:
  - `verify(editor).putString("username", "John");`: This verifies that the `putString()` method was called on the mock editor with the right arguments.
  - `verify(editor).apply();`: This ensures that `apply()` was called to commit the changes.

3.6 When to Use Mocking

- When your class has **dependencies** (like databases, network calls, or Android components) that are not easy to include in unit tests.

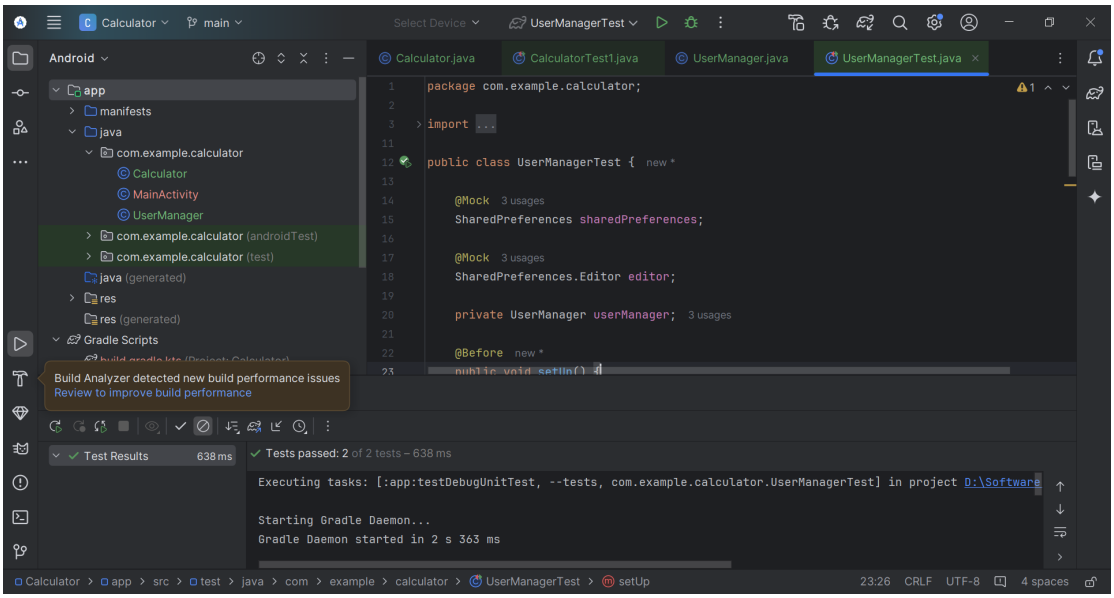

Figure 3.4: Test Passed

- When you want to **test in isolation**, focusing on just the class under test and ignoring the details of how its dependencies work.
- When you want to simulate different behaviors, like errors or exceptions, without needing to recreate those scenarios with real dependencies.

Chapter 4

Conclusion

By combining JUnit for structuring and running tests and Mockito for mocking dependencies, we can ensure that our application's core logic is thoroughly tested, minimizing bugs and reducing development costs. These tools together foster a test-driven development (TDD) approach, encourage modular code and ensure high-quality, stable code throughout the development lifecycle.