Mobile Application Development

Mobile Application Testing

- Mobile Mindset
- Mobile Platforms and Application Development fundamentals
- Introduction to Android Operating System
- Android Interface Design Concepts
- Main Components of Android Application
- Sensors and Media Handling in Android Applications
- Data Handling in Android Applications
- Android Web Services
- Android Application Testing and security aspects
- Kotlin Language to develop Android Mobile Apps

Learning Outcomes of the Lecture

At the end of this lecture students should be able to:

- Identify the purpose of Mobile Application Testing
- Understand the Testing Types for Mobile Application
- Write test cased for Android Mobile Application

Purpose of Software Testing

- What is a software test?
  - A piece of software which executes another piece of software.
  - Validates if the code results as expected.
  - Software unit tests help the developer to verify that the logic of piece of the program is correct.

Android App Testing

- Android app testing is a complex task due to the existence of multiple device manufacturers, device models, Android OS versions, screen sizes, and network conditions.

✓ Testing on Real Android Devices

Testing against a wide selection of devices from various manufacturers with different screen resolutions and Android OS versions.

✓ Immediate new Android version support

Supported for newly release devices and Android versions

Android App Testing

✓Test complex scenarios and custom UI elements

- Testing coverage integrations with device components, peripherals, and system apps such as camera, audio, GPS, Google now, Google Assistant or Google Maps.
- Automate customized actions and UI elements such as sliders, pickers, tables, gestures, etc.

✓ Test performance to ensure a great user experience

- Able to catch performance issues before deployment.

To-Do before Mobile Testing for first release

1. Research on OS and Devices
2. Test Bed
3. Test plan
4. Automation Tools
5. Testing techniques or methods

Best Practices in Android App Testing

- ✓Device Selection
- ✓Beta Testing of the Application
- ✓Connectivity
- ✓Manual or Automated Testing

Testing Types for Mobile Apps

1. Functional Testing
   - ✓ Checks whether the application is working based on the requirements.
   - ✓ The flow of use cases and various business rules are tested.

Eg:

To validate whether – all required mandatory fields are working as required.

- the device is able to perform required multitasking requirements.

- navigation between relevant modules in the app as per the requirement.

- the user receives appropriate error messages like “network error, try again after some time”, etc..

Testing Types for Mobile Apps

2. Android UI Testing
   - ✓ User-centric testing of the application is done under this.
   - ✓ Normally performed by manual users.

Eg: testing – visibility of text in various screens of the app
- interactive messages
- alignment of data
- look and feel of the app for different screens
- whether the buttons are in required size and suitable to big fingers.
- whether the icons are natural and consistent with the app.
- size of fields, etc..

Testing Types for Mobile Apps

3. Compatibility Testing

✓Performed to ensure that the app is fit across all the devices because they have different size, resolution, screen, version and hardware.

✓So, mostly done in form of two matrices

1. OS vs. App
2. Device model vs. App

Eg: To ensure – the UI of the app is as per the screen size o the device and no text/control is partially invisible or inaccessible.

- text is readable for all users.

- call/alarm functionality is enabled whenever the app is running.

Testing Types for Mobile Apps

4. Interface Testing / Integration Testing

✓ This is done after all the modules of the app are completely developed.

✓ Includes a complete end-to-end testing of the app, interaction with other apps like Maps and social apps, usage of microphone to enter text, usage of camera to scan a barcode or to take a picture, etc.

Testing Types for Mobile Apps

5. Network Testing
   - ✓Mainly done to verify the response time in which the activity is performed like refreshing data after sync or loading data after login.
   - ✓This is an in-house testing.
   - ✓Done for both strong WiFi connection and the mobile data network.
   - ✓Request/response to/from the service is tested for various conditions.
   - ✓App should talk to the immediate service to carry out the process.

Testing Types for Mobile Apps

6. Performance Testing

- ✓Performance of the app under some conditions are checked.
- ✓Tested from both application end and the app server end.
- ✓Conditions- Low memory in the device
  - - The battery in extremely at a low level.
  - - Poor/Bad network reception.

Testing Types for Mobile Apps

7. Installation Testing

- ✓This is done to ensure that the installation of the app is going smoothly without ending up in errors or partial installation etc.
- ✓ Upgrade and uninstallation testing are carried out as part of this testing.

Testing Types for Mobile Apps

8. Security Testing

✓Testing for the data flow for encryption and decryption mechanism is tested under this.

Eg: -To validate whether the app is not permitting an attacker to access sensitive content or functionality without proper authentication.

- To validate the app has a strong password protection system.

- To prevent from insecure data storage in the keyboard cache of the applications.

Testing Types for Mobile Apps

9. Field Testing

- ✓Done specifically for the mobile data network.
- ✓Doing only after the whole app is developed.
- ✓Verify the behavior of the app when the phone has 2G or 3G connection.
- ✓This testing verifies if the app is crashing under slow network connection or if it is taking too long to load the information.

Testing Types for Mobile Apps

10. Interrupt Testing (Offline Scenario Verification)

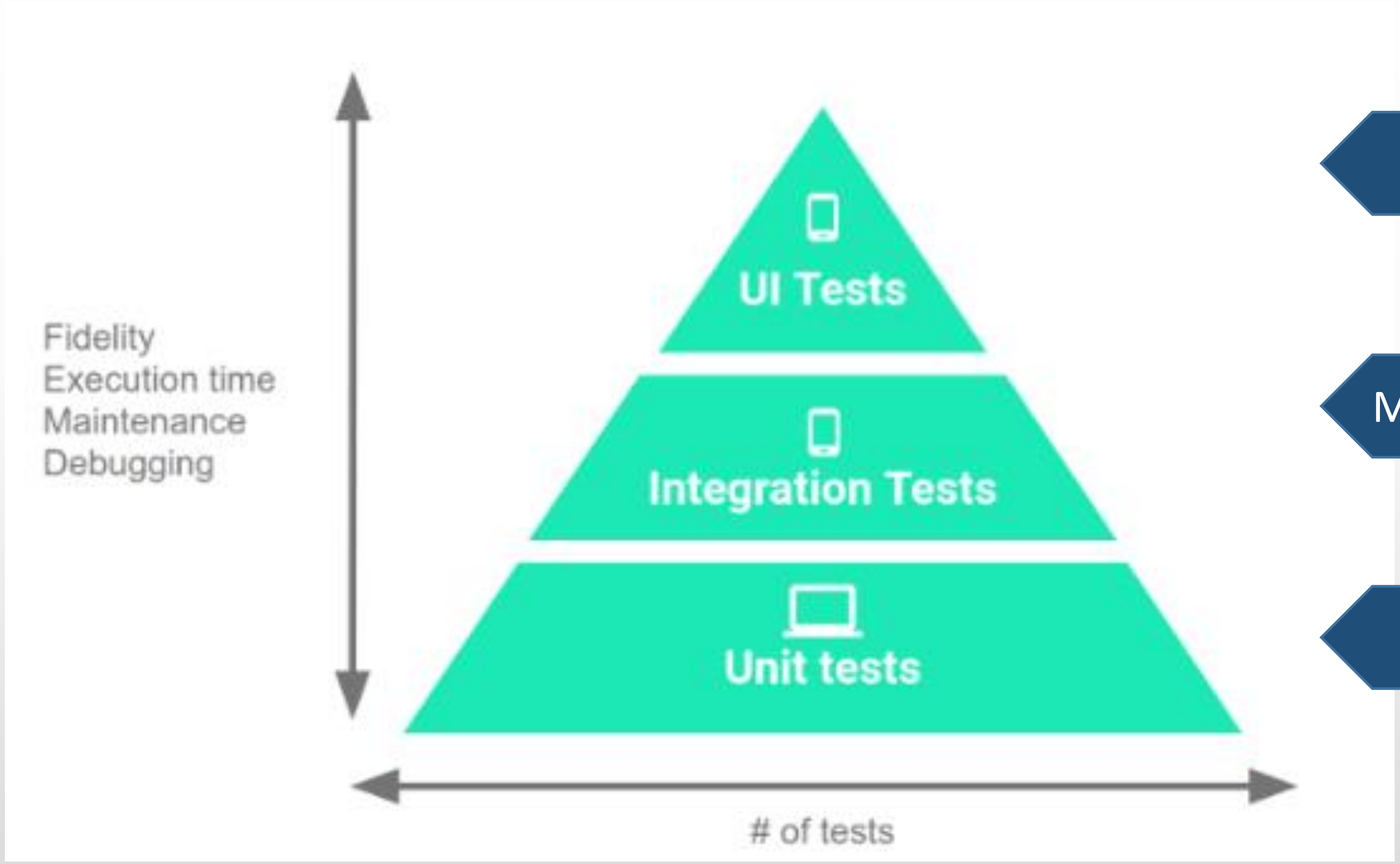
- ✓Offline conditions – Condition where the communication breaks in the middle

  Eg:
  - Data cable removal during data transfer process.
  - Network outage during the transaction posting phase.
  - Network recovery after an outage.
  - Battery removal or power ON/Off when it is in the transactional phase.

Mobile Testing tools

- Kobiton
- TestProject
- Squish By FroLogic
- TestingBot
- Apptim
- Headspin
- Appium (iOS/Android Testing tool)
- Selendroid
- MonkeyRunner
- Calabash
- KIF
- Testroid
- Robotium - Android
- Robo-electric - Android

Writing Tests



Large Tests
10%

Medium Tests
20%

Small Tests
70%

Testing Pyramid

Write Small Tests

- Highly focused to Unit tests.

Local Unit tests	Instrumented Unit tests
• Use AndroidX Test APIs	• Can be done on a physical device or emulator
• Can be used Robolectric for tests that always run on a JVM-powered development machine	• AndroidX test makes use of following threads. - **Main thread** (UI thread/ activity thread) -> occur UI interactions and activity lifecycle events - **Instrumentation thread** -> most of the tests are run under this. When the test suit begins, the AndroidJUnitTest class starts this thread.
• Roboelectric supports: - Component lifecycles - Event loops - All resources	

Write Medium Tests

- Validate the collaboration and interaction of a group of units.

Eg:

- ✓Interactions between a view and view model (testing a Fragment object, validating a layout XML, evaluation a data binding logic of a ViewModel object)
- ✓Testing od app's repository layer – verify different data sources and data access objects interact as expected.

- Use methods from the Espresso Intents library.

Write Large Tests

- Validate end-to-end workflows that guide users through multiple modules and features.

Configuring the environment in Android Studio

✓Organize test directories based on execution environment

Android Studio contains two directories to locate tests.

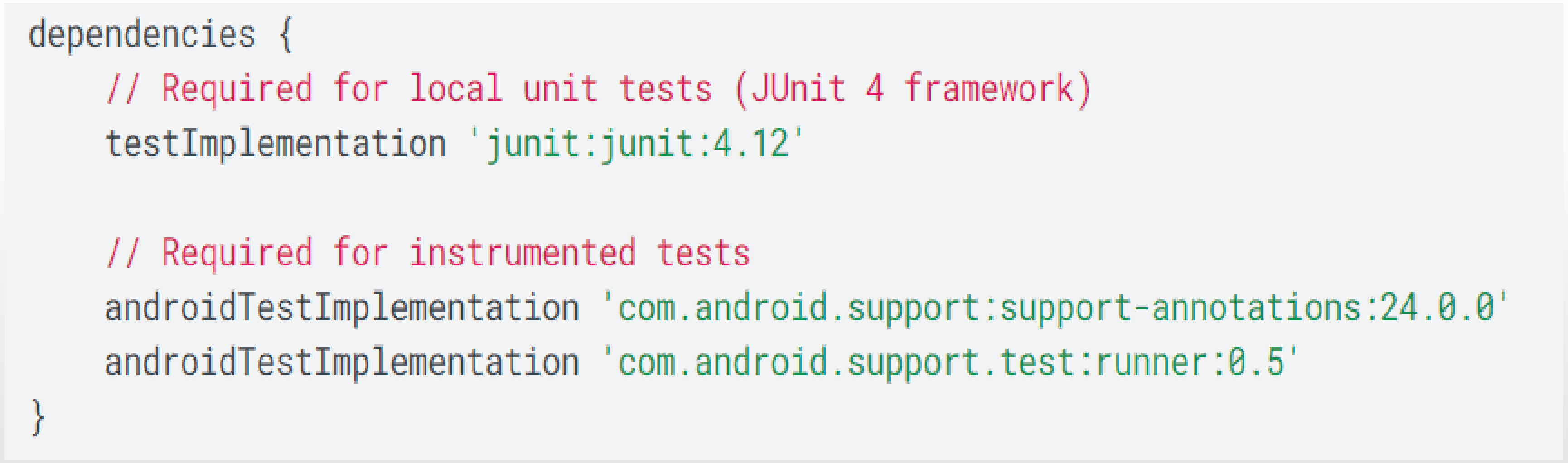
1. androidTest – Contains the tests that run on real or virtual devices.

   - Tests include integration tests, end-to-end tests and other tests where JVM alone cannot validate the app's functionality.

2. test – Contains the tests that run on the local machine such as unit tests.

Setting dependencies

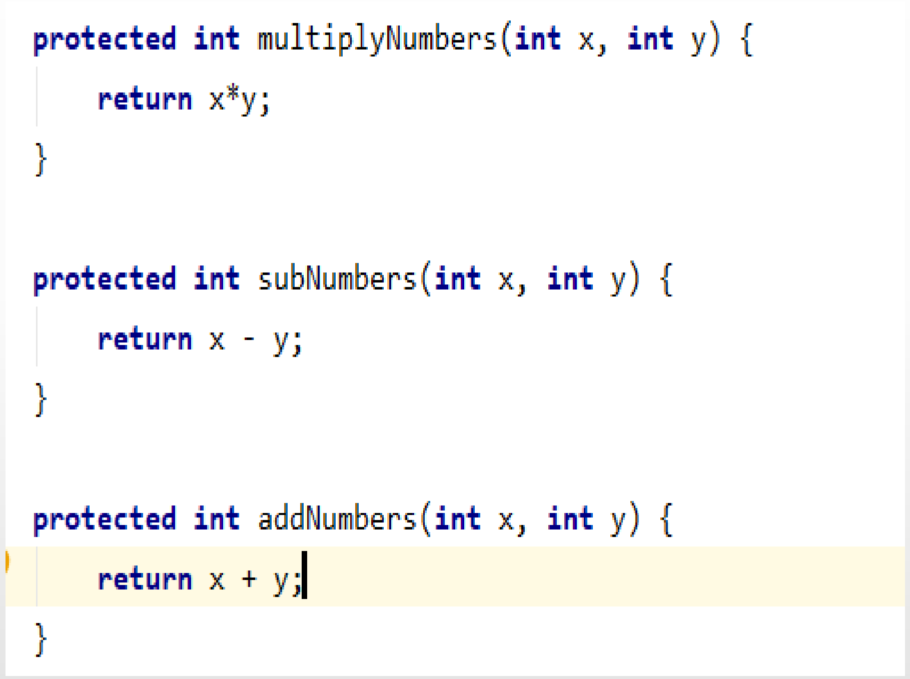
✓Specify the test library dependencies in the app module’s *build.gradle* file.



```
dependencies {
    // Required for local unit tests (JUnit 4 framework)
    testImplementation 'junit:junit:4.12'

    // Required for instrumented tests
    androidTestImplementation 'com.android.support:support-annotations:24.0.0'
    androidTestImplementation 'com.android.support.test:runner:0.5'
}
```

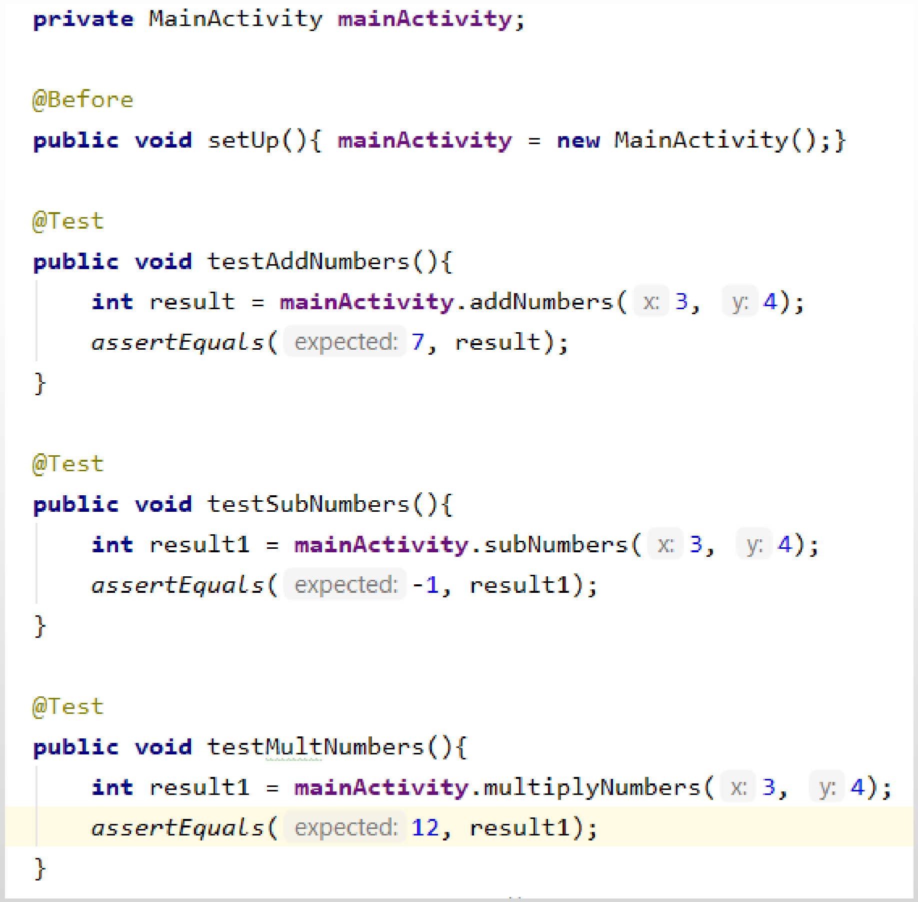
Sample Calculator for local Unit tests

MainActivity.java



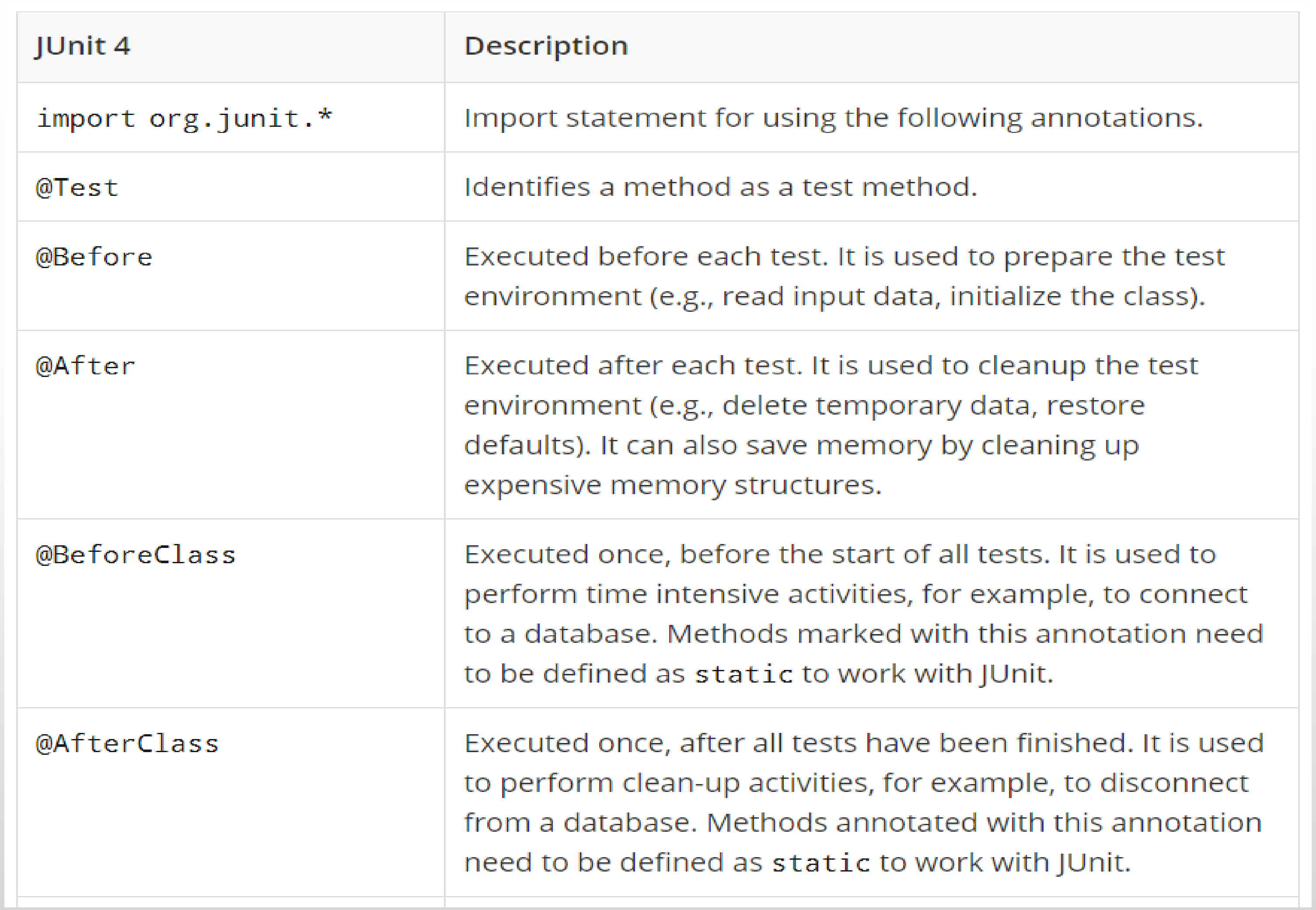
```java
protected int multiplyNumbers(int x, int y) {
    return x*y;
}

protected int subNumbers(int x, int y) {
    return x - y;
}

protected int addNumbers(int x, int y) {
    return x + y;
}
```

MainActivityTest.java



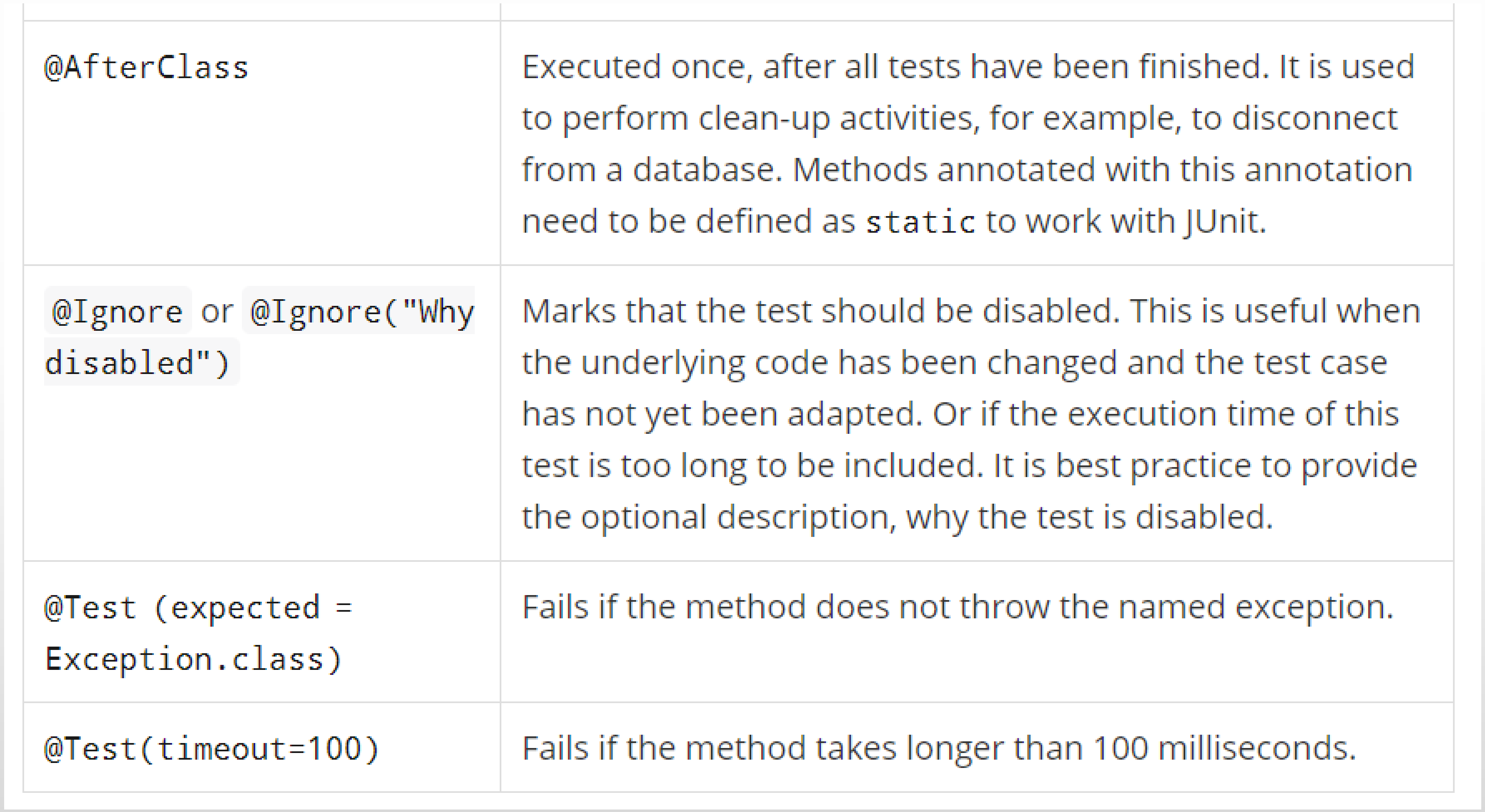
```java
private MainActivity mainActivity;

@Before
public void setUp(){ mainActivity = new MainActivity();}

@Test
public void testAddNumbers(){
    int result = mainActivity.addNumbers( x: 3,  y: 4);
    assertEquals( expected: 7, result);
}

@Test
public void testSubNumbers(){
    int result1 = mainActivity.subNumbers( x: 3,  y: 4);
    assertEquals( expected: -1, result1);
}

@Test
public void testMultNumbers(){
    int result1 = mainActivity.multiplyNumbers( x: 3,  y: 4);
    assertEquals( expected: 12, result1);
}
```

Testing annotations in jUnit4



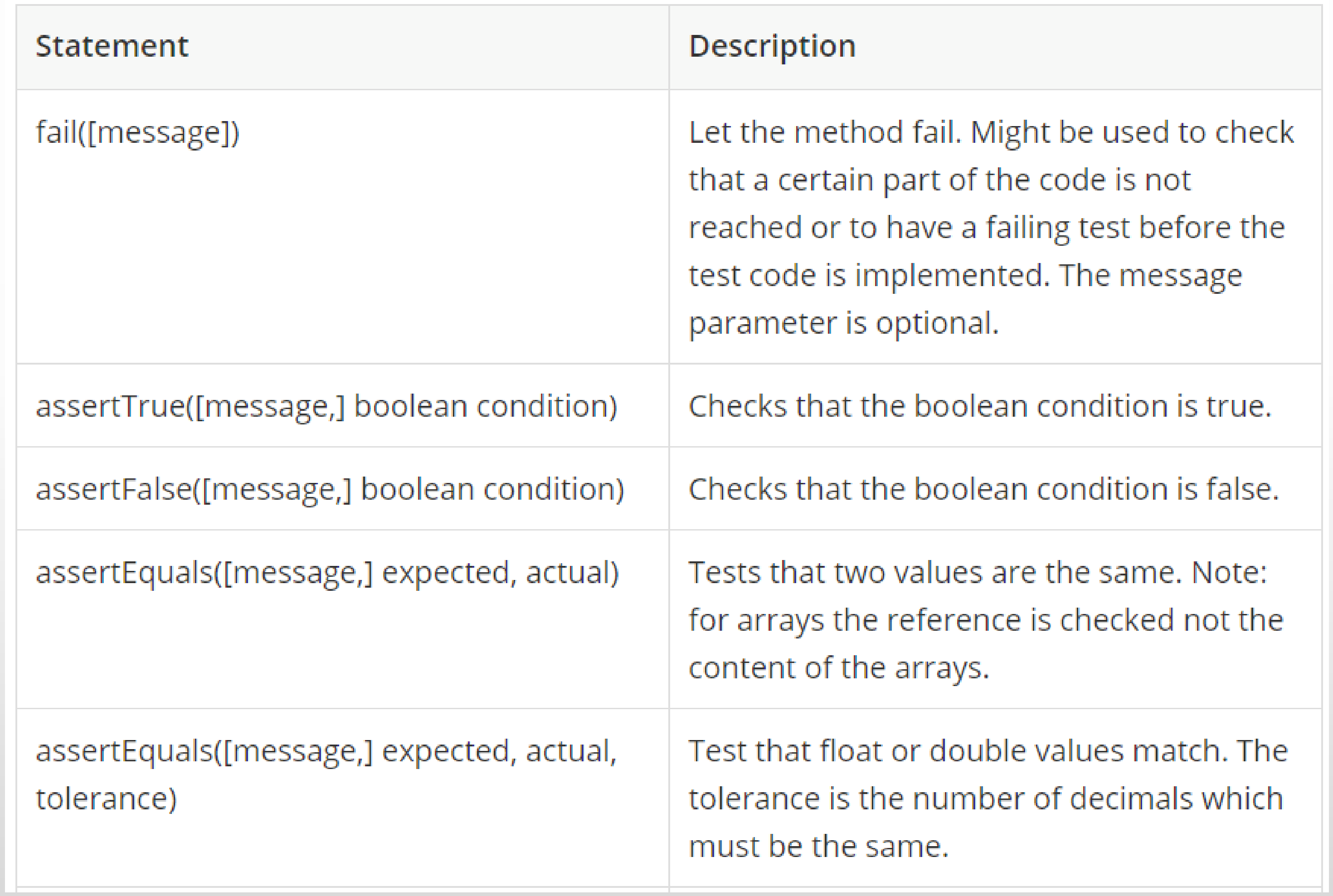
JUnit 4	Description
`import org.junit.*`	Import statement for using the following annotations.
`@Test`	Identifies a method as a test method.
`@Before`	Executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
`@After`	Executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
`@BeforeClass`	Executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as `static` to work with JUnit.
`@AfterClass`	Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.

Testing annotations in jUnit4



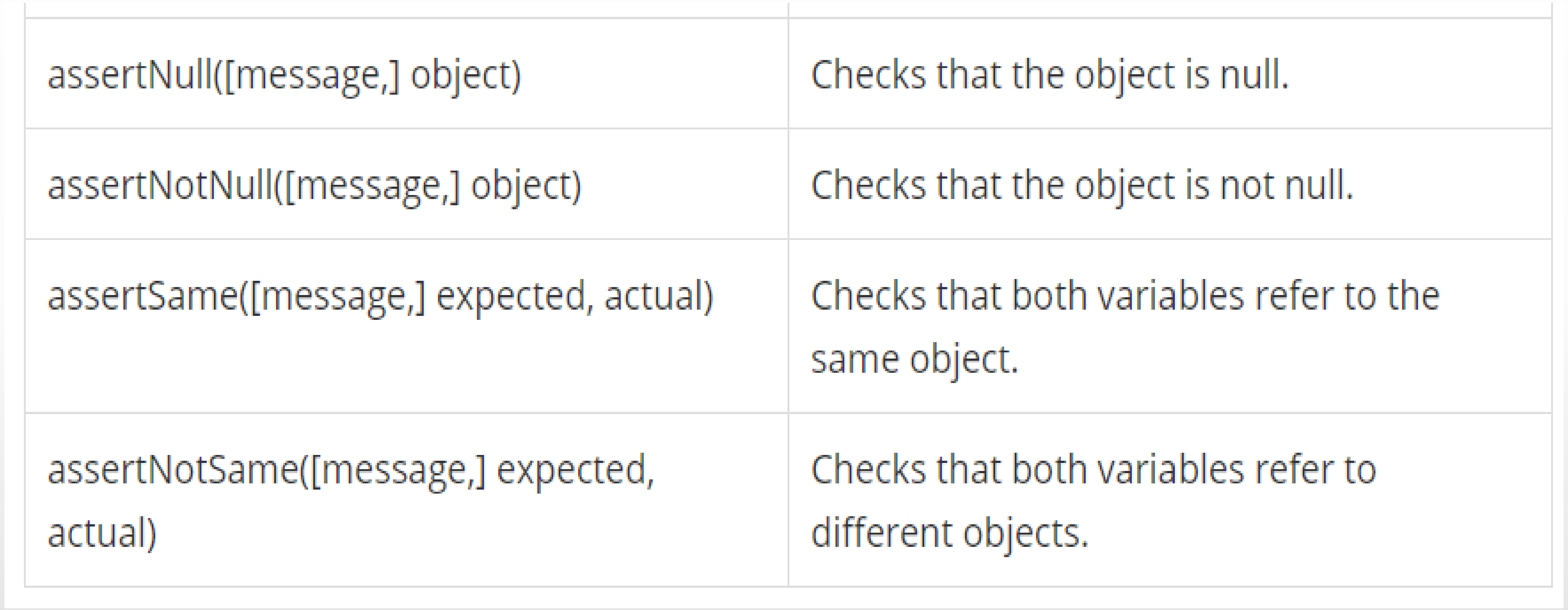
`@AfterClass`	Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.
`@Ignore` or `@Ignore("Why disabled")`	Marks that the test should be disabled. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled.
`@Test (expected = Exception.class)`	Fails if the method does not throw the named exception.
`@Test(timeout=100)`	Fails if the method takes longer than 100 milliseconds.

Methods to assert test results



Statement	Description
fail([message])	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional.
assertTrue([message,] boolean condition)	Checks that the boolean condition is true.
assertFalse([message,] boolean condition)	Checks that the boolean condition is false.
assertEquals([message,] expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([message,] expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.

Methods to assert test results



assertNull([message,] object)	Checks that the object is null.
assertNotNull([message,] object)	Checks that the object is not null.
assertSame([message,] expected, actual)	Checks that both variables refer to the same object.
assertNotSame([message,] expected, actual)	Checks that both variables refer to different objects.