# JUnit Testing - Mockito and coverage

[JUnit](http://junit.org/junit5/)is one of the most popular unit-testing frameworks in the Java ecosystem. The JUnit 5 version contains a number of exciting innovations, with **the goal of supporting new features in Java 8 and above**, as well as enabling many different styles of testing.

Mockito, on the other hand, is a popular mocking framework that allows developers to create mock objects in order to test their code in isolation. In this presentation, we'll be exploring the capabilities of both JUnit 5 and Mockito, and how they can be used together to create effective unit tests.

Mockito is an open-source test automation framework that internally uses Java Reflection API to create mock objects. Mock objects are dummy objects used for actual implementation. The main purpose of using a dummy object is to simplify the development of a test by mocking external dependencies and using them in the code.

**Test-driven development (TDD)**

Itis a software development process that interweaves coding, testing, and design. It is a test-first approach that aims to improve the quality of your applications. Test-driven development is defined by the following lifecycle:

1. Add a test.
2. Run all of your tests and observe the new test failing.
3. Implement the code.
4. Run all of your tests and observe the new test succeeding.
5. Refactor the code

Diagram

Description automatically generated

## Benefits of Mockito

There are multiple ways to mock an object. Here are some listed below.

* **No handwriting:** In Mockito, there is no requirement for writing your mock objects.
* **Safe refactoring:** While renaming the method name of an interface or interchanging the parameters do not change the test code, as mock objects are created at runtime.
* **Exception support:** It supports the exception. In Mockito, the stack trace is used to find the cause of the exception.
* **Annotation support:** It creates mock objects using annotations like @Mock.
* **Order support:** It provides a check on the order of the method calls.

Shape

Description automatically generated

## Mockito - Junit Installation

When you use Mockito in your unit tests, you will need to download the jar file and place it in a path that your build system can find. Mockito is available in two versions: mockito-core (which contains only the core of Mockito, and mockito-all (which contains all modules).

The preferred way of installing Mockito is to declare a dependency on mockito-core with a build system of choice. The second best way is to download the artifacts using a manual approach and add them to the classpath. You can also add dependencies to your existing Maven

Add the following dependencies in your pom.xml:

<!-- https://mvnrepository.com/artifact/org.mockito/mockito-junit-jupiter -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-junit-jupiter</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

</dependency>

## Junit Mockito Setup - Annotations

**@Test:** This annotation is used to mark a method as a test method. JUnit 5 will automatically detect all methods marked with @Test and execute them as part of the test suite.

**@BeforeAll and @AfterAll:** These annotations are used to mark methods that should be executed once before or after all the test methods in the test class, respectively. These methods can be used to set up or tear down any resources that are needed for the test suite.

**@BeforeEach and @AfterEach:** These annotations are used to mark methods that should be executed before or after each test method in the test class, respectively. These methods can be used to set up or tear down any resources that are needed for each individual test.

**@Disabled:** This annotation is used to mark a test method as disabled, meaning it will not be executed as part of the test suite. This can be useful if a particular test is failing and needs to be temporarily disabled until the issue can be resolved.

**@Mock:** This annotation is used in conjunction with Mockito to create mock objects for testing. When a class is annotated with @Mock, Mockito will create a mock object of that class that can be used in the test methods.

**@InjectMocks:** This annotation is also used in conjunction with Mockito, but is applied to a class rather than a method. It tells Mockito to inject any mock objects that have been created into the fields of the annotated class.

**@ExtendWith or @RunWith:** This annotation is used to register JUnit 5 extensions, which are classes that can add additional functionality to the testing framework. For example, the Mockito extension can be registered using this annotation to allow the use of Mockito in JUnit 5 tests.

**@DisplayName** – defines a custom display name for a test class or a test method

**@Nested** – denotes that the annotated class is a nested, non-static test class

**@Tag** – declares tags for filtering tests

**@Spy** annotation is used to create a real object and spy on that real object. This would help to call all the object methods while still tracking every interaction that is being mocked.

**@Captor** annotation is used to create an Argument Captor instance to capture method argument values for further assertions.

**@InjectMocks** annotation is used to mock a class with all its dependencies. This is quite useful to test the behavior completely.

JUnit 5 also provides a number of built-in assertions that can be used to test various aspects of your code, such as:

1. **void assertEquals(boolean expected,boolean actual)**: checks that two primitives/objects are equal. It is overloaded.
2. **void assertTrue(boolean condition)**: checks that a condition is true.
3. **void assertFalse(boolean condition)**: checks that a condition is false.
4. **void assertNull(Object obj)**: checks that object is null.
5. **void assertNotNull(Object obj)**: checks that object is not null.

## Naming conventions in Junit

There are no naming conventions that you should follow in Junit. But it is to start function names with test or end with test for clarity.

Text

Description automatically generated

## How to Write JUnit Test

@RunWith(MockitoJUnitRunner.**class**)

**public** **class** MatildaUserServiceTest {

@InjectMocks

**private** MatildaUserService userService;

@Mock

**private** MatildaUserRepository userRepository;

// mock the needed classes

@BeforeEach

**public** **void** setUp() {

MockitoAnnotations.*openMocks*(**this**);

List<MatildaUserEntity> userEntityList = **new** ArrayList<>();

MatildaUserEntity userEntity1 = **new** MatildaUserEntity();

userEntity1.setId("1");

userEntity1.setName("User 1");

userEntity1.setEmail("user1@test.com");

MatildaUserEntity userEntity2 = **new** MatildaUserEntity();

userEntity2.setId("2");

userEntity2.setName("User 2");

userEntity2.setEmail("user2@test.com");

userEntityList.add(userEntity1);

userEntityList.add(userEntity2);

}  
//data used to mock

@Test

**public** **void** testGetUsersList() **throws** Exception {

List<MatildaUserEntity> entityList = **new** ArrayList<>();

entityList.add(testUserEntity);

*when*(userRepository.findAll()).thenReturn(entityList);

List<MatildaUser> userList = userService.getUsersList();

*assertEquals*(1, userList.size());

*assertEquals*(testUser.getName(), userList.get(0).getName());

*assertEquals*(testUser.getEmail(), userList.get(0).getEmail());

}

@Test

**public** **void** testGetUsersListWithEmptyList() {

*when*(userRepository.findAll()).thenReturn(**new** ArrayList<>());

*assertThrows*(Exception.**class**, () -> userService.getUsersList());

}

@Test

**void** validateUserDuplicateName\_withUniqueName\_returnsTrue()

**throws** NoSuchMethodException, InvocationTargetException, IllegalAccessException {

MatildaUserEntity matildaUserEntity = **new** MatildaUserEntity();

matildaUserEntity.setId("1");

matildaUserEntity.setName("TestUser");

List<MatildaUserEntity> accountEntityList = Collections.*emptyList*();

*when*(userRepository.findAllByNameIgnoreCaseAndBaseAndIdNot(*anyString*(), *anyString*()))

.thenReturn(accountEntityList);

Method method = MatildaUserService.**class**.getDeclaredMethod("validateUserDuplicateName", MatildaUserEntity.**class**);

method.setAccessible(**true**);

**boolean** result = (**boolean**) method.invoke(userService, matildaUserEntity);

*assertTrue*(result);

}

//test the methods

## Mocking

The mocking technique is not only used in Java but also used in any object-oriented programming language. There are many frameworks available in Java for mocking, but Mockito is the most popular framework among them.

To mock objects, you need to understand the three key concepts of mocking, i.e., stub, fake, and mock. Some of the unit tests involve only stubs, whereas some involve fake and mocks.

The brief description of the mocking concepts is given below:

1. **Stub:** Stub objects hold predefined data and provide it to answer the calls during testing. They are referred to as a dummy object with a minimum number of methods required for a test. It also provides methods to verify other methods used to access the internal state of a stub, when necessary. Stub object is generally used for **state verification**.
2. **Fake:** Fake are the objects that contain working implementations but are different from the production one. Mostly it takes shortcuts and also contains the simplified version of the production code.
3. **Mock:** Mock objects act as a dummy or clone of the real object in testing. They are generally created by an open-source library or a mocking framework like Mockito, EasyMock, etc. Mock objects are typically used for **behavior verification**.

## Mocking private-protected methods

In general, private and protected methods are not directly accessible from outside the class and cannot be called directly in JUnit 5 or Mockito test cases.

**Refactor the code:** One way to test private or protected methods is to refactor the code and extract the logic in the method into a new public method. This new public method can then be called directly from the test case. While this may not always be feasible, it is a good practice to write code that is testable and modular.

**Use reflection:** Another way to test private or protected methods is to use Java reflection to access the method. By using reflection, you can bypass the accessibility restrictions and invoke the method directly. However, this approach can be fragile and may break if the method signature or implementation changes.

**Test through public methods**: A third approach is to test the functionality of the private or protected method indirectly, by testing the public methods that call it. This approach requires the public method to be tested thoroughly to ensure that it correctly handles all possible scenarios and calls the private or protected method correctly.

**Use package-private access:** If the test class is in the same package as the class containing the private or protected method, it may be possible to call the method using package-private access. Package-private access means that the method is accessible within the same package, but not outside it. This approach can be a useful compromise between accessibility and encapsulation, but may not be appropriate for all scenarios.

It's important to note that while these approaches can be used to test private and protected methods, they may not always be the best solution. It's often better to focus on testing the public API of the class and ensuring that it behaves correctly in all scenarios, rather than testing the implementation details of individual methods.

## Methods of Mockito

**Mockito mock() method**

* It is used to create mock objects of a given class or interface. Mockito contains five mock() methods with different arguments. When we didn't assign anything to mocks, they will return default values.

**Mockito when() method**

* It enables stubbing methods. It should be used when we want to mock to return specific values when particular methods are called. In simple terms, "When the XYZ() method is called, then return ABC." It is mostly used when there is some condition to execute.

**Syntax**: <T> when(T methodCall)

**Mockito verify() method**

* The verify() method is used to check whether some specified methods are called or not. In simple terms, it validates the certain behavior that happened once in a test. It is used at the bottom of the testing code to assure that the defined methods are called.
* There are two types of verify() methods available in the Mockito class, which are given below:
* verify() method with VerificationMode: It verifies some behavior happened at least once, exact number of times, or never.  
  **Syntax:** <T> verify(T mock, VerificationMode mode)
* verify() method: It verifies certain behaviour happened once.

**Syntax**: <T> verify(T mock)

**Mockito verifyNoMoreInteractions() method**

* It is used to check that any of the given mocks have any unverified interactions. We can use this method after verifying all the mock, to make sure that nothing else was invoked on the mocks. It also detects the unverified invocations that occur before the test method, for example, in setup(), @Before method, or the constructor. It is an optional method, and we don't need to use it in every test.

**Mockito doThrow() method**

* It is used when to stub a void method to throw an exception. It creates a new exception instance for each method invocation.

**Mockito doCallRealMethod() method**

* It is used when we want to call the real implementation of a method. In other words, it is used to create partial mocks of an object. It is used in rare situations, such as to call the real methods. It is similar to the spy() method, and the only difference is that it results in complex code.

**Mockito doAnswer() method**

* It is used when we want to stub a void method with a generic Answer type.

**Mockito doNothing() method**

* It is used for setting void methods to do nothing. The doNothing() method is used in rare situations. By default, the void methods on mock instances do nothing, i.e., no task is performed.

**Mockito doReturn() method**

* It is used on those rare occasions when we cannot use Mockito.when(object). The Mockito.when(object) method is always suggested for stubbing because it is argument type-safe and more readable as compare to the doReturn() method.

**Mockito times() method**

* It is used to verify the exact number of method invocations, which means it declares how many times a method is invoked.

**Mockito never() method**

* It is used to verify that the interaction did not happen.

**Mockito atLeastOnce() method**

* It is used to verify the invocation at-least-once that means the method should be invoked at least once.

**Mockito atLeast() method**

* It is used to verify the invocation at least x number of times. For eg., given atLeast(3) means the method will invoke a minimum of three times.

**Mockito atMost() method**

* It is used to verify the invocation at most x number of times. For eg., given atMost(3) means the method will invoke a maximum of three times.

**Mockito calls() method**

* It allows a non-greedy verification in order. It can only be used with the inOrder() verification method. For eg., inOrder.verify(mock, calls(3)).xyzMethod("...");

**Mockito only() method**

* It checks that the given method was the only invoked method.

**Mockito timeout() method**

* It allows Mockito to perform verification with a timeout. It instructs a verify to wait for a specific period of time for a particular interaction rather than to fail immediately. It may be useful for testing in existing situations.

**Mockito after() method**

* It allows Mockito to verify over a given period of time. We have already discussed that the after() method differs from the timeout() method.

## Coverage

Coverage is a tool for measuring code coverage for programs. In Junit coverage refers to the measurement of how much of the code in a project is executed by tests.

Code coverage is usually expressed as a percentage, and it indicates how many lines of code, branches, functions, and statements are executed by the tests. This information can be used to identify untested code or code that is only partially tested, which can help improve the overall quality of the application.

## Coverage Installation

* Go to IDE top header **Help->Eclipse Market Place** it will open the pop up for Eclipse Market Place.
* Type text as **“Code Coverage”** in search text box and click “Go” button. You will get below screen

Graphical user interface, text, application, email

Description automatically generated

* You will get list of plugins select the plugin “EclEmma Code Coverage” click on Install button for this plugin.
* You will get next screen where need to accept terms and conditions and click continue.
* After installing this code coverage plugin, It will ask to restart your IDE.
* After restart your IDE, go to the top **Windows->Show View->Others** then search for **“Coverage**” as below then select option click on Open button.

Graphical user interface, text, application, Word

Description automatically generated

* You will see one additional tab “Coverage” on your console view section as below.



* Also get “Coverage As” option as below when right click on your application. Now you can run you Junit.

Graphical user interface, application

Description automatically generated

* Once you run your application Junit by using “Coverage As” option you will see the results on “Coverage” tab as below.

Table

Description automatically generated

A picture containing text

Description automatically generated