**Junit and Mockito**

**─**

Contents

[**1. Introduction to testing** 2](#_Toc190936314)

[# Unit testing 3](#_Toc190936315)

[**2. Junit** 4](#_Toc190936316)

[# setting up Junit 4](#_Toc190936317)

[# First Test Case 4](#_Toc190936318)

[# Assertions 5](#_Toc190936319)

[# Important Annotations 7](#_Toc190936320)

[# Test Fixtures 8](#_Toc190936321)

[# Parameterized Test 10](#_Toc190936322)

[**3. Mockito** 11](#_Toc190936323)

[# Setting up Mockito 12](#_Toc190936324)

[# Why Mockito? 13](#_Toc190936325)

[# Mock vs Stub vs Spy 15](#_Toc190936326)

[# Core Mockito Annotations 18](#_Toc190936327)

[# Mockito Behavior Control 20](#_Toc190936328)

[# Arguments Matchers 25](#_Toc190936329)

[# Verify Execution 28](#_Toc190936330)

[# Handling Private Methods in Mockito 32](#_Toc190936331)

[# Mocking static methods 33](#_Toc190936332)

[# Object initiation inside a method 34](#_Toc190936333)

[# Best Mockito Practices 35](#_Toc190936334)

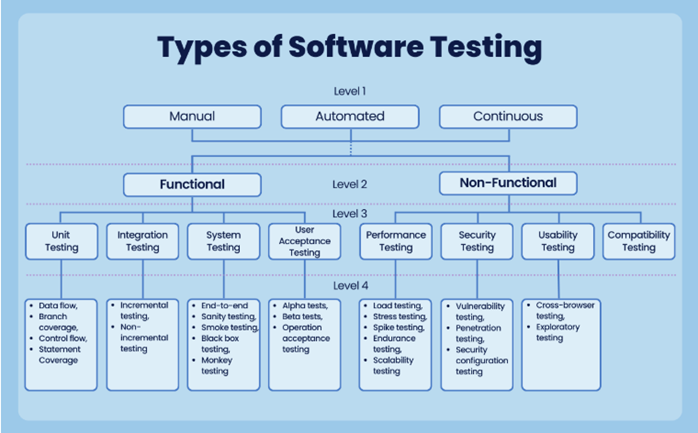
[**4. Interview Questions** 37](#_Toc190936335)

## **1. Introduction to testing**

Software testing is the process of evaluating a software application to ensure it works as expected and meets user requirements.

Different types of testing:

1. **Unit testing**: Test individual components or functions in isolation to ensure they work as expected.
2. **Integration Testing**: Validate the interaction between integrated components or systems to detect interface defects.
3. **Smoke Testing**: A basic test to check basic functionalities of an application after a new build is deployed.
4. **Load Testing**: Tests the system’s performance under expected load conditions to identify bottlenecks.
5. **Stress Testing:** Assess how system behave under extreme conditions beyond its operational capacity.
6. **Regression Testing:** Confirms new code changes do not adversely affect existing functionality.
7. **Sanity Testing:** A focused subset of regression testing to verify that specific functionality work after changes or bug fixes.
8. **Alpha testing:** Conducted internally by developers or testers to identify bugs before releasing the product to external users.
9. **Beta Testing:** Final testing phase conducted by real users in a real environment to gather feedback and identify any remaining issues.
10. **User Acceptance Testing (UAT):** Conducted by end-users to validate whether the system meets their needs and requirements.



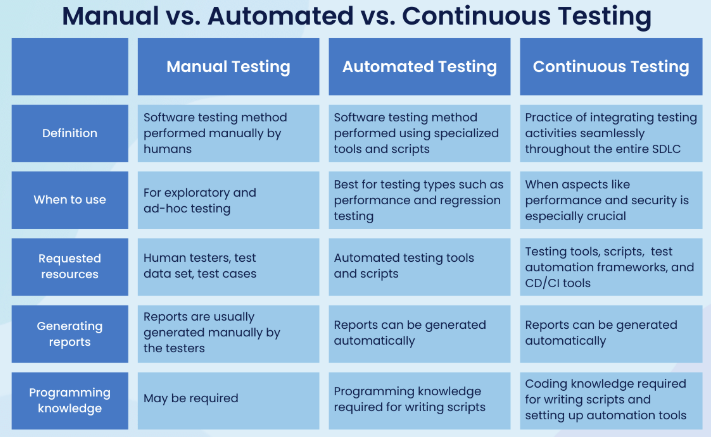
Functional and non-functional testing -

Functional testing verifies that the software behaves according to the requirements, while non-functional testing evaluates aspects like performance, usability and reliability.

**Test coverage:** Test coverage measures the amount of code that is tested by unit tests, it helps to identify parts of codebase that lack testing.

**Code smell:** Code smell is a hint that something might be wrong in the code (eg duplicated code, long methods). Unit testing can help identify these issues by ensuring that tests fails when code changes introduce bugs.

**TDD (Test Driven Development):**  is a software development approach in which test cases for each functionality are created and tested first and if test fails, the new code is written in order to pass the test and making code simple and bug-free. In simple terms, in TDD test is written first and then to pass the test, code is written accordingly.



### # Unit testing

Unit Testing is a software testing technique where individual units or components of a software application are tested in isolation to ensure they function as expected. A "unit" refers to the smallest testable part of an application, such as a method or function.

**Why is Unit Testing Important?**

* Detects issues early in the development lifecycle.
* Simplifies debugging by isolating problems.
* Facilitates code changes and refactoring without introducing new bugs.
* Provides documentation of how individual components are intended to work.

## **2. Junit**

JUnit is a popular testing framework for Java that provides annotations and tools for writing unit tests. It allows developers to automate testing and integrates seamlessly with build tools like Maven and Gradle.

### # setting up Junit

1. **For Maven Projects**: Add the following dependency to the pom.xml file.

<dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter</artifactId>  
 <version>5.10.0</version>  
 <scope>test</scope>  
</dependency>

1. **For Gradle Projects**: Add the following to the dependencies block in the build.gradle file.

testImplementation 'org.junit.jupiter:junit-jupiter:5.10.0'

1. **For Standalone Setup**:  
   Download the JUnit JAR files from the [JUnit website](https://junit.org/) and add them to your project’s classpath.

### # First Test Case

Class to test

package org.example;  
  
public class Calculator {  
 public int multiply(int a, int b) {  
 return a \* b;  
 }  
}

Test class

import org.junit.Test;  
import static org.junit.Assert.*assertEquals*;  
  
public class CalculatorTest {  
 @Test  
 public void testMultiply(){  
 Calculator calc = new Calculator();  
 *assertEquals*(8,calc.multiply(2,4));  
 }  
}

A typical JUnit test case consists of the following components:

1. **Test Method**: A method annotated with @Test that contains the logic to test a specific function.
2. **Assertions**: Used to validate expected outcomes.
3. **Setup Code**: Initializes the environment or objects required for the test.
4. **Teardown Code**: Cleans up resources after a test (if needed).

Organizing test cases -

1. **Single Responsibility**: Each test method should test one specific functionality.
2. **Naming Conventions**:
   * Use descriptive names for test methods, e.g., testAdditionWithPositiveNumbers.
3. **Separation of Concerns**:
   * Separate the setup, execution, and validation logic for clarity.
4. **No order :** Junit don’t guarantee any order of execution of test cases.

### # Assertions

Assertions are used to validate the output of a test case. They ensure the tested code meets the expected criteria.

1. **assertEquals(expected, actual)**: Verifies if the actual value matches the expected value. With double values can have 3rd parameter delta, which defined amount of difference in decimal we can tolerate. For opposite case, we have assertNotEquals() too.

Assertions.assertEquals(10, result);

1. **assertNotNull(object)**: Validates that an object is not null. Similarly assertNull() also exists.
2. **assertThrows(exception, executable)**: Ensures an exception is thrown.

Assertions.assertThrows(IllegalArgumentException.class, () -> {  
someMethodThatThrowsException();  
});

1. **assertTrue(condition)**:Validates that a condition is true. Similarly assertFalse() exists.
2. **assertArrayEquals()**: When we want to verify value of array is equal to expected one.

Sample codes -

1. Using assetArrayEquals()

//Test case to verify Arrays.sort method  
@Test  
public void testSortArray() {  
 //Working code  
 int[] input = {5, 3, 1, 4, 2};  
 Arrays.*sort*(input);  
  
 int[] expected = {1, 2, 3, 4, 5};  
 Assertions.*assertArrayEquals*(expected, input);  
}

1. Palindrome testing

Main code

public class StringUtils {  
 public boolean isPalindrome(String input) {  
 String reversed = new StringBuilder(input).reverse().toString();  
 return input.equals(reversed);  
 }  
}

Test class

public class StringUtilsTest {  
 @Test  
 public void testIsPalindrome() {  
 StringUtils stringUtils = new StringUtils();  
 Assertions.*assertTrue*(stringUtils.isPalindrome("madam"));  
 Assertions.*assertFalse*(stringUtils.isPalindrome("hello"));  
 }  
}

### # Important Annotations

1. **@Disabled:**

* Temporarily disables a test.
* Useful for tests that are incomplete or failing.

@Disabled("Feature under development")  
@Test  
void incompleteTest() {  
 Assertions.fail("Not implemented yet");  
}

1. **@DisplayName**

* Customizes the display name of the test.

@DisplayName("Addition Test")  
@Test  
void testAddition() {  
 Assertions.assertEquals(4, 2 + 2);  
}

1. **@Tag**

* Tag tests for filtering during execution

@Tag("fast")  
@Test  
void fastTest() {  
 Assertions.assertTrue(true);  
}

### # Test Fixtures

Test fixtures are a set of methods used to set up the necessary environment and resources for testing and to clean them up afterward. They ensure that:

1. Each test starts in a known state.
2. Resources such as files, databases, or connections are properly handled.
3. Code duplication in test setup and teardown is minimized.

For example, if multiple tests rely on a database connection, a test fixture can establish the connection once and clean it up afterward.

Annotations in Test fixtures -

1. **@BeforeEach**

* Executes **before each test method**.
* Used to initialize or set up objects and data.
* Ensures every test starts in a clean state.

1. **@AfterEach**

* Executes **after each test method**.
* Used to clean up resources or reset configurations.

1. **@BeforeAll**

* Executes **once before all test methods** in the class.
* Used for time-consuming setup like database connections.
* Must be **static**.

1. **@AfterAll**

* Executes **once after all test methods** in the class.
* Used to release resources shared across tests.
* Must be **static**.

Code Example:

import org.junit.jupiter.api.\*;  
  
class CalculatorTest {  
 private Calculator calculator;  
  
 @BeforeAll  
 static void initializeSuite() {  
 System.*out*.println("Suite setup: Establishing shared resources.");  
 }  
  
 @BeforeEach  
 void setup() {  
 calculator = new Calculator();  
 System.*out*.println("Test setup: Initializing calculator.");  
 }  
  
 @Test  
 void testAddition() {  
 System.*out*.println("Running testAddition.");  
 Assertions.assertEquals(5, calculator.add(2, 3), "Addition test failed.");  
 }  
  
 @Test  
 void testSubtraction() {  
 System.*out*.println("Running testSubtraction.");  
 Assertions.assertEquals(1, calculator.subtract(3, 2), "Subtraction test failed.");  
 }  
  
 @AfterEach  
 void tearDown() {  
 System.*out*.println("Test teardown: Cleaning test-specific resources.");  
 }  
  
 @AfterAll  
 static void finalizeSuite() {  
 System.*out*.println("Suite teardown: Releasing shared resources.");  
 }  
}  
  
class Calculator {  
 int add(int a, int b) {  
 return a + b;  
 }  
  
 int subtract(int a, int b) {  
 return a - b;  
 }  
}

### # Parameterized Test

Parameterized tests in JUnit allow you to run the same test logic repeatedly with different sets of input data. Instead of duplicating the same test multiple times with varying inputs, you define a single test method that accepts parameters.

**1. @ParameterizedTest**

* Marks a test method as parameterized.
* Works in conjunction with other data source annotations like @ValueSource, @CsvSource, etc.

**2. @ValueSource**

* Provides a fixed array of values to the test method.
* Supports primitive types (int, long, double), String, and Clas

class ParameterizedExample {  
 @ParameterizedTest  
 @ValueSource(ints = {1, 2, 3, 4, 5})  
 void testIsEven(int number) {  
 System.*out*.println("Testing with number: " + number);  
 Assertions.assertTrue(number > 0, "Number should be positive");  
 }

@ParameterizedTest  
@ValueSource(strings = {"hello", "world", "junit"})  
void testStringLength(String word) {  
 Assertions.assertTrue(word.length() > 0, "String should not be empty");  
}

}

**3. @CsvSource**

* Supplies a list of comma-separated values.
* Each line represents a set of inputs.

@ParameterizedTest  
@CsvSource({  
 "1, 1",  
 "2, 4",  
 "3, 9"  
})  
void testSquare(int number, int expectedSquare) {  
 Assertions.assertEquals(expectedSquare, number \* number, "Square calculation failed");  
}

## **3. Mockito**

Mockito is a mocking framework for Java that allows developers to create mock objects for testing. It helps in writing unit tests by simulating dependencies instead of using real implementations.

**Challenges in Unit Testing Without Mockito**

* **Dependency Issues**: When testing a class, it may depend on external services (databases, APIs, file systems). Testing with real dependencies can be slow and unreliable.
* **Unpredictability**: External dependencies might return different results, making tests non-deterministic.
* **Difficult to Test Edge Cases**: Some real-world scenarios (like network failures) are hard to replicate.
* **Slow Test Execution**: Tests relying on real services (e.g., databases) take time to execute.

**Code Examples:**

**Without Mockito**

//If the UserRepository connects to a database, running tests may be slow and require a working database.  
  
class UserService {  
 private UserRepository userRepository;  
  
 public UserService(UserRepository userRepository) {  
 this.userRepository = userRepository;  
 }  
  
 public User getUserById(int id) {  
 return userRepository.findById(id); // This interacts with a real database!  
 }  
}

We here only want to do unit testing not integration testing, so we don’t really need to communicate to real database, for this purpose we can use dummy object of UserRepository and depict it’s functionality.

**Test with Mockito**

class UserServiceTest {  
 @Test  
 void testGetUserById() {  
 // Create a mock for UserRepository  
 UserRepository mockRepository = *mock*(UserRepository.class);  
  
 // Define behavior: when findById(1) is called, return a fake user  
 *when*(mockRepository.findById(1)).thenReturn(new User(1, "John Doe"));  
  
 UserService userService = new UserService(mockRepository);  
 User user = userService.getUserById(1);  
  
 *assertEquals*("John Doe", user.getName());

// Test passes without using a real database!  
 }  
}

Mockito is essential for unit testing as it helps:

✅ Write faster and more reliable tests.

✅ Isolate dependencies for better test accuracy.

✅ Simulate real-world scenarios like API failures or payment errors.

### # Setting up Mockito

For Maven:

<dependencies>  
 <!-- Mockito Core -->  
 <dependency>  
 <groupId>org.mockito</groupId>  
 <artifactId>mockito-core</artifactId>  
 <version>5.6.0</version>  
 <scope>test</scope>  
 </dependency>  
  
 <!-- JUnit 5 Integration -->  
 <dependency>  
 <groupId>org.mockito</groupId>  
 <artifactId>mockito-junit-jupiter</artifactId>  
 <version>5.6.0</version>  
 <scope>test</scope>  
 </dependency>  
</dependencies>

For gradle

dependencies {  
 testImplementation 'org.mockito:mockito-core:5.6.0'  
 testImplementation 'org.mockito:mockito-junit-jupiter:5.6.0'  
}

For spring project it defaults comes, so we don’t need to add that dependency for spring project.

### # Why Mockito?

Let’s consider normal case where Business Layer contact with DataService.

interface DataService{  
 int[] getAllData();  
}  
  
public class BusinessLayer {

private DataService dataService;  
  
 public BusinessLayer(DataService dataService) {  
 this.dataService = dataService;  
 }  
  
 public int getSumOfValues() {  
 int[] arr = dataService.getAllData();  
 int total = 0;  
 for (int ele : arr) {  
 total += ele;  
 }  
 return total;  
 }

}

First testing by implementing DataService interface using real objects.

We are here using interface not class, as we want all types of different results from DataService to test all scenarios, but if we directly make it a class it will return same type of results.

//using Real class

class BusinessLayerTest {  
  
 @Test  
 void test(){  
 DataServiceClass dataServiceObject1 = new DataServiceClass();  
 BusinessLayer bs = new BusinessLayer(dataServiceObject1);  
 *assertEquals*(42,bs.getSumOfValues());  
 }  
  
 @Test  
 void test2(){  
 DataServiceClass2 dataServiceObject2 = new DataServiceClass2();  
 BusinessLayer bs2 = new BusinessLayer(dataServiceObject2);  
 *assertEquals*(8,bs2.getSumOfValues());  
 }  
  
  
}  
class DataServiceClass implements DataService{  
 @Override  
 public int[] getAllData() {  
 return new int[]{12,14,16};  
 }  
}  
  
class DataServiceClass2 implements DataService{  
 @Override  
 public int[] getAllData() {  
 return new int[]{3,5};  
 }  
}

In above example, just to get different data for testing, we are required to create a new class. That will make our code complex and slow. Therefore to solve this issue we will use Mock.

//using mock

class BusinessLayerTest2 {  
  
 @Test  
 void test() {  
 DataService dataServiceMock = *mock*(DataService.class);  
 BusinessLayer bs = new BusinessLayer(dataServiceMock);  
 *when*(dataServiceMock.getAllData()).thenReturn(new int[]{12,14,16});  
 *assertEquals*(42, bs.getSumOfValues());  
 //Case 2  
 *when*(dataServiceMock.getAllData()).thenReturn(new int[]{3,5});  
 *assertEquals*(8, bs.getSumOfValues());  
 }  
  
}

### # Mock vs Stub vs Spy

**Mock**

A mock is a fully fake object where all methods return default values unless explicitly stubbed (using Mockito.when()). It is used to verify interactions rather than actual behavior.

**When to Use Mocks?**

* When you don’t want to call real methods.
* When you only care about method calls (not the actual behavior).
* When the dependency is slow (like a database or API call).

import static org.mockito.Mockito.\*;  
 import static org.junit.jupiter.api.Assertions.\*;  
  
@ExtendWith(MockitoExtension.class) // Enable Mockito  
class UserServiceTest {  
  
 @Mock  
 private UserRepository userRepository; // Fake object  
  
 @InjectMocks  
 private UserService userService; // Inject fake repository  
  
 @Test  
 void testFindUserById() {  
 // Define behavior for mock  
 *when*(userRepository.findById(1)).thenReturn(new User(1, "John Doe"));  
  
 // Call the method  
 User user = userService.getUserById(1);  
  
 // Verify result  
 *assertEquals*("John Doe", user.getName());  
  
 // Verify interaction  
 *verify*(userRepository, *times*(1)).findById(1);  
 }  
}

📌 **Key Points:**

* @Mock creates a **fake object** of UserRepository.
* It runs the method, ignores the body of method(like interface)
* when(...).thenReturn(...) defines a **mocked response**.
* verify(...) ensures the method was **called exactly once**.

**Stub**

A stub is a controlled object that returns predefined values when specific methods are called. Unlike mocks, stubs don’t track interactions.

**When to Use Stubs?**

* When you need fixed return values from dependencies.
* When the dependency is slow (like an external API).
* When the return value does not change based on execution.

class UserServiceTest {  
  
 @Test  
 void testStub() {  
 // Create a stub using mock()  
 UserRepository userRepositoryStub = *mock*(UserRepository.class);  
  
 // Define behavior  
 *when*(userRepositoryStub.findById(1)).thenReturn(new User(1, "John Doe"));  
  
 // Use the stub in the service  
 UserService userService = new UserService(userRepositoryStub);  
 User user = userService.getUserById(1);  
  
 // Verify result  
 *assertEquals*("John Doe", user.getName());  
 }  
}

📌 **Key Points**:

* mock(UserRepository.class) creates a stub object.
* when(...).thenReturn(...) predefines specific responses.

**Spy**

A spy is a real object where only selected methods are mocked, while others execute normally.

**When to Use Spies?**

* When you need partial mocking (some real behavior, some faked).
* When you want to track interactions on real objects.
* When calling the real method would cause side effects.

@ExtendWith(MockitoExtension.class) // Enable Mockito  
class UserServiceTest {  
  
 @Spy // Creates a real object but allows selective mocking  
 private UserRepository userRepository = new UserRepository();  
  
 @InjectMocks  
 private UserService userService;  
  
 @Test  
 void testSpy() {  
 // Mock only one method  
 *doReturn*(new User(1, "John Doe")).when(userRepository).findById(1);  
  
 // Call method  
 User user = userService.getUserById(1);  
  
 // Verify the result  
 *assertEquals*("John Doe", user.getName());  
  
 // Verify that the real object’s method was NOT called  
 *verify*(userRepository, *times*(1)).findById(1);  
 }  
}

**📌 Key Points**:

* @Spy creates a real object, not a fake one.
* doReturn(...).when(...) prevents calling the real method.

| **Feature** | **Mock** | **Stub** | **Spy** |
| --- | --- | --- | --- |
| Calls Real Methods? | ❌ No | ❌ No | ✅ Yes (unless stubbed) |
| Tracks Method Calls? | ✅ Yes | ❌ No | ✅ Yes |
| Used for Verifying Interactions? | ✅ Yes | ❌ No | ✅ Yes |
| Used for Predefined Values? | ✅ Yes | ✅ Yes | ✅ Yes |
| Allows Partial Mocking? | ❌ No | ❌ No | ✅ Yes |

| **Use Case** | **Use Mock?** | **Use Stub?** | **Use Spy?** |
| --- | --- | --- | --- |
| Replace a slow database/API call? | ✅ Yes | ✅ Yes | ❌ No |
| Verify method calls and arguments? | ✅ Yes | ❌ No | ✅ Yes |
| Need a fixed return value? | ❌ No | ✅ Yes | ✅ Yes |
| Need real behavior with some stubbing? | ❌ No | ❌ No | ✅ Yes |
| Testing complex logic without dependency interference? | ✅ Yes | ✅ Yes | ❌ No |

### # Core Mockito Annotations

**1. @Mock :** use to create a fully fake object where all methods return default values unless explicitly stubbed. Used when you need to verify method calls without calling real methods.

**2. @ExtendWith(MockitoExtension.class):** Applies Mockito to whole junit class.

**3. @InjectMocks:** automatically injects mock or spy object into a class under test, avoids manual dependency injection in tests.

@ExtendWith(MockitoExtension.class) // Enable Mockito  
class UserServiceTest {  
  
 @Mock // Creates a mock UserRepository  
 private UserRepository userRepository;  
  
 @InjectMocks // Injects the mock into UserService  
 private UserService userService;  
  
 @Test  
 void testFindUserById() {  
 // Define behavior for mock  
 *when*(userRepository.findById(1)).thenReturn(new User(1, "John Doe"));  
  
 // Call the method  
 User user = userService.getUserById(1);  
  
 // Verify result  
 *assertEquals*("John Doe", user.getName());  
  
 // Verify interaction  
 *verify*(userRepository, *times*(1)).findById(1);  
 }  
}

in place of injecting like:

@Mock  
DataService dataServiceMock;

BusinessLayer bs = new BusinessLayer(dataServiceMock);

Can directly inject the mock in BusinessLayer using @InjectMocks

@Mock  
private DataService dataServiceMock;  
  
@InjectMocks //inject mock of dataservice in businessLayer  
BusinessLayer bs;

**4. @Spy :** Is used to partially mock a real object, calls real methods unless explicitly stubbed.

@ExtendWith(MockitoExtension.class) // Enable Mockito  
class UserServiceTest {  
  
 @Spy // Creates a real object but allows selective mocking  
 private UserRepository userRepository = new UserRepository();  
  
 @InjectMocks  
 private UserService userService;  
  
 @Test  
 void testSpy() {  
 // Mock only one method  
 doReturn(new User(1, "John Doe")).when(userRepository).findById(1);  
  
 // Call method  
 User user = userService.getUserById(1);  
  
 // Verify the result  
 *assertEquals*("John Doe", user.getName());  
  
 // Verify that the real object’s method was NOT called  
 verify(userRepository, times(1)).findById(1);  
 }  
}

**📌 Key Points:**

* @Spy creates a real object, unlike @Mock.
* Only specific methods are stubbed using doReturn().
* Useful when part of the logic should execute normally.

### # Mockito Behavior Control

Mockito allows us to control how a mock behaves when a method is called. This is crucial because:

* We need to define expected responses for test cases.
* We need to simulate edge cases, errors, and exceptions.
* We need to avoid real dependencies and isolate business logic.

1. **when().thenReturn()**

* when().thenReturn() is used to specify the return value of a mocked method.
* Works only with mocked (not spied) objects.
* Useful for simulating database calls, API responses, or repository methods.

Main class

class Calc{  
 public int modifyNumsAndAdd(int a,int b){  
 a = a+b;  
 b = 2\*b;  
 return a+b;  
 }  
}  
  
public class LearnMockito {  
 private Calc calc;  
  
 public LearnMockito(Calc calc) {  
 this.calc = calc;  
 }  
  
 public int addNums(int a, int b){  
 if(a%2==1){  
 a-=1;  
 }  
 return calc.modifyNumsAndAdd(a,b);  
 }  
}

Test class

In place of creating the object in method, can even use injectMock too.

@InjectMocks  
private LearnMockito lm;

@ExtendWith(MockitoExtension.class)  
class LearnMockitoTest {  
  
 @Mock  
 private Calc calc;  
  
 @Test  
 void testAddNums() {  
 // Define mock behavior  
 *when*(calc.modifyNumsAndAdd(2,3)).thenReturn(11);  
  
 // Create the object  
 LearnMockito lm = new LearnMockito(calc);  
  
 // Verify result  
 *assertEquals*(11, lm.addNums(2,3));  
 }  
}

Eg2: Mocking list class

class ListMockTest {  
  
 @Test  
 void testMockList() {  
 List<String> mockList = mock(List.class);  
  
 when(mockList.get(0)).thenReturn("Hello");  
 when(mockList.size()).thenReturn(5);  
  
 assertEquals("Hello", mockList.get(0));  
 assertEquals(5, mockList.size());  
 }  
}

**📌 Key Points:**

* when(repository.findById(1)).thenReturn(...) defines expected behavior.
* The real method is never called, and a fake result is returned.
* Used when we need to mock dependencies returning values.

1. **when().thenThrow()**

* when().thenThrow() makes a mock method throw an exception when called.
* Useful for testing error handling and exception scenarios.

Main class

class Calc {  
 public int modifyNumsAndAdd(int a, int b) {  
 if(a<0 || b<0){  
 throw new IllegalArgumentException("Argument can't be less than 0");  
 }  
 a = a + b;  
 b = 2 \* b;  
 return a + b;  
 }  
}  
  
public class LearnMockito {  
 private Calc calc;  
 public LearnMockito(Calc calc) {  
 this.calc = calc;  
 }  
 public int addNums(int a, int b) throws IllegalArgumentException {  
 try {  
 return calc.modifyNumsAndAdd(a, b);  
 } catch (Exception e) {  
 throw new IllegalArgumentException(e);  
 }  
 }  
}

Test class

@ExtendWith(MockitoExtension.class)  
class LearnMockitoTest {  
  
 @Mock  
 private Calc calc;  
 @InjectMocks  
 private LearnMockito lm;  
  
 @Test  
 void testAddNums() throws Exception {  
 // Define mock behavior  
 *when*(calc.modifyNumsAndAdd(-1,3)).thenThrow(new IllegalArgumentException("Argument can't be less than 0"));  
  
 // Call method and expect an exception  
 Exception exception = *assertThrows*(IllegalArgumentException.class, () -> {  
 lm.addNums(-1,3);  
 });  
  
 *assertEquals*("java.lang.IllegalArgumentException: Argument can't be less than 0", exception.getMessage());  
 }  
}

1. **doReturn().when()**

* doReturn().when() is used when working with spies (@Spy).
* Works even when the method is final or already called.

**doReturn() Vs thenReturn()**

| **Feature** | **when().thenReturn()** | **doReturn().when()** |
| --- | --- | --- |
| Works with **mocks**? | ✅ Yes | ✅ Yes |
| Works with **spies**? | ⚠️ **Yes, but problematic** | ✅ **Yes, without issues** |
| **Calls the actual method?** | ✅ **Yes** (problematic for spies) | ❌ **No** (safe for spies) |
| Use case | Standard mocking | Avoiding real method calls in spies |

**Why doReturn().when() is Better for Spies?**

* With when().thenReturn() on spies, the real method gets called when setting up the stub. This can cause issues if the method has side effects (e.g., modifies state, interacts with DB).(check by seeing coverage, actual code is not covered with doReturn())
* With doReturn().when(), the real method is never called, ensuring better control.

Main Class

class Calc {  
 public int modifyNumsAndAdd(int a, int b) {  
 a = a + b;  
 b = 2 \* b;  
 return a + b;  
 }  
}  
  
public class LearnMockito {  
 private Calc calc;  
  
 public LearnMockito(Calc calc) {  
 this.calc = calc;  
 }  
  
 public int addNums(int a, int b) throws IllegalArgumentException {  
 if (a % 2 == 1) {  
 a -= 1;  
 }  
 return calc.modifyNumsAndAdd(a, b);  
 }  
}

Test Class

@ExtendWith(MockitoExtension.class)  
class LearnMockitoTest {  
  
 @Spy  
 private Calc calc;  
 @InjectMocks  
 private LearnMockito lm;  
  
 @Test  
 void testAddNums() throws Exception {  
  
 *doReturn*(13).when(calc).modifyNumsAndAdd(2,3);  
 *assertEquals*(13,lm.addNums(2,3));  
  
 }  
}

📌 **When to Use Which?**

* Use when().thenReturn() for mocks (no real method execution).
* Use doReturn().when() for spies to avoid real method calls.

1. **doThrow().when()**

Used for **spy.** If we use when().thenTrow() with spies, real method will get executed and code will fail. The junit will fail because of failure in actual code.

So with spies we will use doThrow(), which don’t really calls function in actual and runs junit correctly.

@Spy  
private Calc calc;  
@InjectMocks  
private LearnMockito lm;  
  
@Test  
void testAddNums() throws Exception {  
  
 *doThrow*(new IllegalArgumentException("Argument can't be less than 0")).when(calc).modifyNumsAndAdd(-1,3);  
   
 // below will fail because fo actual code failure  
 //when(calc.modifyNumsAndAdd(-1,3)).thenThrow(new IllegalArgumentException("Argument can't be less than 0"));

1. **doNothing().when()**

In Mockito, methods that return void cannot be stubbed using when().thenReturn(), because when() requires a return type. Instead, we use doNothing(), doThrow(), or doAnswer() to control their behavior.

For Main class

class Calc {  
 public void temp() {  
 System.*out*.println("Hello");  
 }  
 public int modifyNumsAndAdd(int a, int b) {  
 return a + b;  
 }  
}  
  
public class LearnMockito {  
 private Calc calc;  
  
 public LearnMockito(Calc calc) {  
 this.calc = calc;  
 }  
 public int addNums(int a, int b) {  
 calc.temp();  
 return calc.modifyNumsAndAdd(a, b);  
 }  
}

If test is like this:

@ExtendWith(MockitoExtension.class)  
public class LearnMockitoTest {  
  
 @Spy  
 private Calc calc;  
 @InjectMocks  
 private LearnMockito lm;  
  
 @Test  
 void testAddNums() throws Exception {  
 *when*(calc.modifyNumsAndAdd(*anyInt*(),*anyInt*())).thenReturn(6);  
 *assertEquals*(6,lm.addNums(7,3));  
 }  
}

Then with test : Hello will get printed in console. So if we want that temp method don’t get called in real, we can use doNothing()

@Test  
void testAddNums() throws Exception {  
 *when*(calc.modifyNumsAndAdd(*anyInt*(),*anyInt*())).thenReturn(6);  
 *doNothing*().when(calc).temp();  
 *assertEquals*(6,lm.addNums(7,3));  
}

### # Arguments Matchers

When testing, sometimes we don’t care about the exact arguments passed to a method but only that the method is called correctly. Matchers help us:

* Avoid hardcoding values.
* Generalize test cases.
* Improve flexibility when input values are unknown.

| **Matcher** | **Description** | **Example** |
| --- | --- | --- |
| any() | Matches any object of any type. | when(mockObj.method(any())).thenReturn("value") |
| anyInt() | Matches any int value. | when(mockObj.method(anyInt())).thenReturn(100) |
| anyString() | Matches any String value. | when(mockObj.method(anyString())).thenReturn("mocked") |
| anyDouble() | Matches any double value. | when(mockObj.method(anyDouble())).thenReturn(5.5) |
| anyList() | Matches any List. | when(mockObj.method(anyList()))  .thenReturn(Collections.emptyList()) |
| eq(value) | Matches exactly the given value. | when(mockObj.method(eq(10))).thenReturn(200) |
| isNull() | Matches null values. | when(mockObj.method(isNull())).thenReturn("null value") |
| notNull() | Matches any **non-null** value. | when(mockObj.method(notNull())).thenReturn("non-null") |

Eg: **using anyString()**

public class Greet { //Main class  
 public String greetUser(String name){  
 String greet = "Hello " + name;  
 return greet;  
 }  
}

class GreetTest { //Test class  
 @Test  
 public void testGreetUser(){  
 Greet greet = *mock*(Greet.class);  
 *when*(greet.greetUser(*anyString*())).thenReturn("Hello World");  
 *assertEquals*("Hello World", greet.greetUser("Alex"));  
 *assertEquals*("Hello World", greet.greetUser("Brad"));  
 }  
}

Eg: **using eq()**

public class Calculator {  
 public int multiply(int a,int b){  
 return a\*b;  
 }  
}

class CalculatorTest {  
 @Test  
 public void testMultiply(){  
 Calculator mockCalc = *mock*(Calculator.class);  
 // Stubbing using eq()  
 *when*(mockCalc.multiply(*eq*(10), *anyInt*())).thenReturn(5);  
  
 System.*out*.println(mockCalc.multiply(10, 2)); // ✅ Output: 5  
 System.*out*.println(mockCalc.multiply(10, 5)); // ✅ Output: 5  
 System.*out*.println(mockCalc.multiply(20, 5)); // ❌ Default return (0), as eq(10) was specified, don’t return 100, the real result from that(as real method is not called in mock)   
 }  
}

* eq(10): Only works when first argument is exactly 10.
* anyInt(): Allows any integer.

Eg: **Using isNull() and notNull()**

class OrderService {  
 String placeOrder(String customer, String product) {  
 return "Order placed for " + customer;  
 }  
}

public class OrderServiceTest {  
  
 @Test  
 public void testNullMatchers() {  
 OrderService mockService = *mock*(OrderService.class);  
  
 *when*(mockService.placeOrder(*isNull*(), *anyString*())).thenReturn("Customer required");  
 *when*(mockService.placeOrder(*notNull*(), *anyString*())).thenReturn("Order placed");

// Output: Customer required  
 System.*out*.println(mockService.placeOrder(null, "Laptop"));

// Output: Order placed  
 System.*out*.println(mockService.placeOrder("Alice", "Phone"));

}  
}

📌 **Key points to remember**

**Mixing Matchers and Raw Values (InvalidUseOfMatchersException)**

when(mockObj.method(any(), "specificValue")).thenReturn("error"); // ❌ Error!  
  
//correct ( use eq() )   
when(mockObj.method(any(), eq("specificValue"))).thenReturn("ok");

when(mockObj.method(anyInt(), 10)).thenReturn(50); // ❌ Error!  
  
//correct ( use eq() )  
when(mockObj.method(anyInt(), eq(10))).thenReturn(50);

### # Verify Execution

In Mockito, **verify()** is used to check if a specific method was called on a mock object during the test execution. It helps ensure that the expected interactions happened as intended.

📌 Why Use verify()?

* Ensures that a method was called the expected number of times.
* Helps verify correct arguments were passed.
* Confirms the order of execution.
* Prevents unexpected method calls.
* Detects no interaction with a mock object.

Main class

public class LearnVerify {  
 void process(String user) {  
 System.*out*.println("Processing for " + user);  
 }  
}

Test class

public class LearnVerifyTest {  
 @Test  
 public void testVerify() {  
 // Step 1: Create a mock object  
 LearnVerify mockService = *mock*(LearnVerify.class);  
  
 // Step 2: Call the method  
 mockService.process("Alice");  
  
 // Step 3: Verify that process() was called with "Alice"  
 *verify*(mockService).process("Alice"); // ✅ Test passes  
 }  
}

**Verifying method call count**

| **Method** | **Description** |
| --- | --- |
| times(n) | Verifies that a method was called exactly n times. |
| never() | Ensures the method was **never called**. |
| atLeast(n) | Ensures it was called at least n times. |
| atMost(n) | Ensures it was called at most n times. |

Eg:

public class AccountRepository {  
 public void debit(String from, double amount) {  
 System.*out*.println("Debit");  
 }  
  
 public void credit(String to, double amount) {  
 System.*out*.println("Credit");  
 }  
}

public class BankService {  
 AccountRepository accountRepository;  
  
 public BankService(AccountRepository accountRepository) {  
 this.accountRepository = accountRepository;  
 }  
  
 public void transferMoney(String from, String to, double amount) {  
 accountRepository.debit(from, amount);  
 accountRepository.credit(to, amount);  
 }  
}

Test class

public class BankServiceTest {  
 @Test  
 public void testTransferMoney() {  
 // Create a mock for AccountRepository  
 AccountRepository mockRepo = *mock*(AccountRepository.class);  
  
 // Create BankService with mocked repository  
 BankService bankService = new BankService(mockRepo);  
  
 // Call the method (this doesn't modify real DB)  
 //but calls actual transferMoney()  
 bankService.transferMoney("Alice", "Bob", 100.0);  
  
 // Verify interactions (checking if correct methods were called)  
 *verify*(mockRepo).debit("Alice", 100.0);  
 *verify*(mockRepo).credit("Bob", 100.0);  
 }  
}

**Why Is verify() Useful?**

* verify(mockRepo).debit("Alice", 100.0);

Confirms that debit() was called with correct parameters.

* verify(mockRepo).credit("Bob", 100.0);

Confirms that credit() was called as expected.

* Even though actual logic is not executed, we ensure correct method calls!

**❌ What Happens Without verify()?**

If verify() is missing:

* We can’t confirm if the transfer logic actually called debit() and credit().
* If there’s a bug (e.g., the developer forgot to call credit()), the test won’t catch it.

Eg2: Here no method was called in actual, just mocking of method calls

class OrderService {  
 String placeOrder(String product) {  
 return "Order placed for " + product;  
 }  
}

Test class

public class OrderServiceTest {  
 @Test  
 public void testVerifyTimes() {  
 OrderService mockService = *mock*(OrderService.class);  
  
 mockService.placeOrder("Laptop");  
 mockService.placeOrder("Laptop");  
  
 // Verify it was called exactly 2 times  
 *verify*(mockService, *times*(2)).placeOrder("Laptop");  
  
 // Verify it was never called with "Phone"  
 *verify*(mockService, *never*()).placeOrder("Phone");  
 }  
}

**Verifying Method Call with Specific Arguments**

For above case

public class OrderServiceTest {  
 @Test  
 public void testVerifyTimes() {  
 OrderService mockService = *mock*(OrderService.class);  
  
 mockService.placeOrder("Laptop");  
 mockService.placeOrder("Phone");  
  
 // Verify it was called exactly 2 times  
 *verify*(mockService, *times*(2)).placeOrder(*anyString*());  
 }  
}

**Verifying Method Call Order (inOrder())**

If multiple methods are called, you can verify their execution order.

Main class

class OrderService {  
 String placeOrder(String product) {  
 return "Order placed for " + product;  
 }  
  
 public String verifyPayment(){  
 return "Payment verified";  
 }  
}

Test class

public class OrderServiceTest {  
 @Test  
 public void testVerifyTimes() {  
 OrderService mockService = *mock*(OrderService.class);  
  
 mockService.placeOrder("Laptop");  
 mockService.verifyPayment();  
  
 // Verify execution order  
 InOrder inOrder = *inOrder*(mockService);  
 inOrder.verify(mockService).placeOrder("Laptop");  
 inOrder.verify(mockService).verifyPayment();  
 }  
}

This verifies that placeOrder was called before verifyPayment()

### # Handling Private Methods in Mockito

Private methods are not directly accessible outside their class, making them hard to test. However, there are few ways to test them:

1. Write the test case for public method which is calling the private method and try to run code in real, in this way during execution code will go to private method and will cover it.

class OrderService {  
 String placeOrder(String product) {  
 verifyPayment();  
 return "Order placed for " + product;  
 }  
  
 private String verifyPayment(){  
 return "Payment verified";  
 }  
}

Test class

public class OrderServiceTest {  
 @Test  
 public void testVerifyTimes() {  
 OrderService orderService = new OrderService();  
 orderService.placeOrder("Laptop");;  
 }  
}

2. Using ReflectionUtils

class OrderService {  
 private String verifyPayment(String product){  
 return "Payment verified";  
 }  
}

Test class

public class OrderServiceTest {  
 @Test  
 public void testVerifyTimes() {  
 OrderService orderService = *mock*(OrderService.class);  
 //even works with new OrderSerivce();  
   
 String encrypted = (String) ReflectionTestUtils.*invokeMethod*(orderService, "verifyPayment", "Laptop");  
 *assertEquals*("Payment verified", encrypted);  
 }  
}

### # Mocking static methods

Since Mockito 3.4, we can use mockStatic() to mock static methods.

Main class

public class Utility {  
 public static String getSystemMessage() {  
 return "Hello, World!";  
 }  
}

Test class(method 1)

public class UtilityTest {  
 @Test  
 public void testStaticMethodMocking() {  
 // Mock the static method  
 try (MockedStatic<Utility> mockedStatic = *mockStatic*(Utility.class))

{  
 mockedStatic.when(Utility::*getSystemMessage*)

.thenReturn("Mocked Message");  
 *assertEquals*("Mocked Message", *getSystemMessage*());   
 }  
 }  
}

Test class(method 2)

public class UtilityTest {  
 @Test  
 public void testStaticMethodMocking() {  
 // Mock the static method  
 try (MockedStatic<Utility> mockedStatic = *mockStatic*(Utility.class))

{  
 mockedStatic.when(()->*getSystemMessage*())

.thenReturn("Mocked Message");  
 *assertEquals*("Mocked Message", *getSystemMessage*());   
 }  
 }  
}

* **mockStatic(Utility.class)** creates a mock of static methods.
* Use **mockedStatic.when(...).thenReturn(...)** to override behavior.
* mockStatic() requires **try-with-resources** to properly close the mock.

### # Object initiation inside a method

Consider the below case

public class Utility {  
 public String getMessage() {  
 Temp2 temp= new Temp2();  
 int t= temp.getANumber(2);  
 return "Hello, World!";  
 }  
}  
  
class Temp2 {  
 public int getANumber(int x) {  
 return x \* 10; // Some actual logic  
 }  
}

Since Utility.getMessage() directly instantiates Temp2 inside the method, we cannot inject a mock easily. you will need to use spy() or reflection (though reflection is not recommended due to complexity).In such cases, consider PowerMockito (if allowed) to override object creation inside the method.

Test class

public class UtilityTest {  
 @Test  
 public void testGetMessage() {  
 // Create a spy for Temp2 to partially mock it  
 Temp2 spyTemp2 = *spy*(new Temp2());  
  
 // Stub the getANumber() method to return a fixed value  
 *doReturn*(99).when(spyTemp2).getANumber(2);  
  
 Utility utility = new Utility();  
 String result = utility.getMessage(); // Calls the real method but uses the stubbed value  
  
 // Assert the expected result  
 *assertEquals*("Hello, World!", result);  
 }  
   
}

* Use Mockito.spy(new Temp2()) to create a partial mock of Temp2.
* Stub getANumber(2) to return 99 instead of its real logic using doReturn(99).when(spyTemp2).getANumber(2).
* Call utility.getMessage() normally since we cannot modify Utility.
* The method runs as usual, but getANumber(2) is overridden by the stubbed value.

### # Mockito lenient()

Mockito.lenient() is used to suppress unnecessary stubbing warnings when a test does not strictly require a particular stubbed behavior.

* By default, Mockito throws warnings when a stubbed method is never used.
* If some stubs are needed for setup but not always called, lenient() prevents warnings.
* Useful when multiple test cases share common stubbing, but not all test cases use all stubs.

Eg: for Main class

class Calc {  
 public int setup() {  
 System.*out*.println("Object created");  
 return 1;  
 }  
  
 public int modifyNumsAndAdd(int a, int b) {  
 return a + b;  
 }  
}  
  
public class LearnMockito {  
 private Calc calc;  
  
 public LearnMockito(Calc calc) {  
 this.calc = calc;  
 }  
  
 public int addNums(int a, int b) {  
 return calc.modifyNumsAndAdd(a, b);  
 }  
}

Test Class

@ExtendWith(MockitoExtension.class)  
public class LearnMockitoTest {  
  
 @Mock  
 private Calc calc;  
 @InjectMocks  
 private LearnMockito lm;  
  
 @Test  
 void testAddNums() throws Exception {  
 // Define mock behavior  
 *when*(calc.modifyNumsAndAdd(*anyInt*(),*anyInt*())).thenReturn(6);  
 *when*(calc.setup()).thenReturn(1);  
 *assertEquals*(6,lm.addNums(7,3));  
 }  
}

For above case, junit will fail with given error: 

Cause calc.setup() method was never called in real. So to surpass this exception, we can use lenient().

@Test  
void testAddNums() throws Exception {  
 // Define mock behavior  
 *when*(calc.modifyNumsAndAdd(*anyInt*(),*anyInt*())).thenReturn(6);  
 *lenient*().when(calc.setup()).thenReturn(1);  
 *assertEquals*(6,lm.addNums(7,3));  
}

This test will pass, so if setup method gets called in actual, it will return 1, else if not called, no exception comes for the test.

**When to Use lenient()?**

| **✅ Use lenient() when...** | **❌ Avoid lenient() when...** |
| --- | --- |
| Some stubbed methods **are not used** in a test. | The test **strictly depends on stubbing**. |
| Multiple tests share common stubbing. | You want to detect **unnecessary stubbing**. |
| Avoiding excessive warnings in large test suites. | The test should fail if an expected call is missing. |

### # Best Mockito Practices

**1. Follow the “Given-When-Then” Structure**

Tests should be structured in three parts:

* Given – Set up test data & mock dependencies.
* When – Call the method under test.
* Then – Verify expected behavior.

@Test  
public void testUserService() {  
 // Given  
 *when*(userRepository.findById(1L)).thenReturn(Optional.of(new User("Abc")));  
  
 // When  
 User result = userService.getUserById(1L);  
  
 // Then  
 assertEquals("Alice", result.getName());  
 *verify*(userRepository, *times*(1)).findById(1L);  
}

**2. Avoid Mocking Everything (Only Mock External Dependencies!)**

* Don’t mock value objects (e.g., String, Integer, LocalDate).
* Avoid mocking classes that you own (test real implementations instead).
* Mock only external dependencies like databases, APIs, and external services.

//🚫 Bad Example:  
User user = *mock*(User.class); // Avoid mocking simple objects  
  
//✅ Good Example:  
UserRepository userRepository = *mock*(UserRepository.class);

**3. Use @Mock and @InjectMocks Properly**

Instead of manually creating mock objects, use annotations for better readability and maintainability.

@ExtendWith(MockitoExtension.class)  
class UserServiceTest {  
 @Mock UserRepository userRepository;  
 @InjectMocks UserService userService;  
}

**4. Keep Tests Focused on One Behavior**

Each test should verify only one thing.

@Test  
void testUserCreation() { /\* Test user creation only \*/ }  
@Test  
void testUserDeletion() { /\* Test user deletion only \*/ }

📌 **Key Points**

* ✅ Follow Given-When-Then for test structure.
* ✅ Use @Mock, @InjectMocks, and @Spy properly.
* ✅ Use verify() to check method calls.
* ✅ Mock only external dependencies, not the class under test.
* ✅ Avoid over-mocking and unnecessary stubbing.
* ✅ Make tests independent and repeatable.
* ✅ Close mockStatic() to prevent side effects.

## **4. Interview Questions**

1. **Why BeforeAll and AfterAll methods need to be static?**

@BeforeAll and @AfterAll methods are typically static because they are executed once at the class level, not per test instance. Since JUnit creates a new test class instance for each test method, static ensures these methods can run without needing a specific instance. In JUnit 5, non-static methods can be used if the class is annotated with @TestInstance(Lifecycle.PER\_CLASS).

1. **Why is it recommended to keep test methods void and without parameters in JUnit?**

Test methods in JUnit are kept void and without parameters because they are designed to check specific conditions or behaviors in isolation. A void return type ensures the method focuses on testing and not returning values, while no parameters make the tests simple and independent, running without needing extra inputs or setup during execution.

1. **What does it means that Mocks track the interactions while stubs not?**

* Stubs: Return hardcoded responses but do not verify method calls.
* Mocks: Not only return values but also verify interactions like method calls, arguments passed, and call order.

class StubExample {  
 @Test  
 void testStub() {  
 // Create a stub (mocked object)  
 MyService stubService = *mock*(MyService.class);  
  
 // Define behavior (return a predefined value)  
 *when*(stubService.getData()).thenReturn("Stubbed Response");  
  
 // Call the method  
 String result = stubService.getData();  
  
 // Validate response  
 *assertEquals*("Stubbed Response", result);  
 }  
}

* The stub only returns predefined values ("Stubbed Response").
* No way to check if getData() was called or how many times.

import org.junit.jupiter.api.Test;  
  
class MockExample {  
 @Test  
 void testMock() {  
 // Create a mock (mocked object)  
 MyService mockService = *mock*(MyService.class);  
  
 // Define behavior (return a predefined value)  
 *when*(mockService.getData()).thenReturn("Mocked Response");  
  
 // Call the method multiple times  
 mockService.getData();  
 mockService.getData();  
  
 // Validate response  
 assertEquals("Mocked Response", mockService.getData());  
  
 // Verify how many times the method was called  
 *verify*(mockService, *times*(3)).getData(); // ✅ Tracks interactions  
 }  
}

* Mock can track interactions
* Mock returns predefined values like a stub.
* We can verify getData() was called exactly 2 times.

Both examples **use Mockito** and appear similar at first glance. However, the key difference lies in **how they are used**—**stubs only return values, while mocks also track interactions.**

1. **What is output of this?**

public class Calculator {  
 public int multiply(int a,int b){  
 return a\*b;  
 }  
}

class CalculatorTest {  
 @Test  
 public void testMultiply(){  
 Calculator mockCalc = *mock*(Calculator.class);  
 Calculator spyCalc = *spy*(Calculator.class);  
 System.*out*.println(mockCalc.multiply(10,2));  
 System.*out*.println( spyCalc.multiply(10,2));   
 }  
}

output: 0 , 20

Mock object don’t really call actual method, so if we are not using when().thenReturn() then default 0 is returned. But in spy, real method is called and we get actual output from that method.

1. **How do you mock a method that returns void?**

Use doNothing() or doThrow()

doNothing().when(mockedObject).voidMethod();  
doThrow(new RuntimeException()).when(mockedObject).voidMethod();

1. **Create an API, and test API call with junit and mockito.**

* We have a UserService that calls an external API (https://jsonplaceholder.typicode.com/users).
* The API returns user data.
* We mock the API call using Mockito and test the response handling logic.

Created this service class

@Service  
public class UserService {  
 private final RestTemplate restTemplate;  
  
 public UserService(RestTemplate restTemplate) {  
 this.restTemplate = restTemplate;  
 }  
  
 public Map<String, Object> getUserById(int userId) {  
 String url = "https://jsonplaceholder.typicode.com/users/" + userId; //this is real api  
 return restTemplate.getForObject(url, Map.class);  
 }  
}

Test class

@ExtendWith(MockitoExtension.class)  
public class UserServiceTest {  
  
 @Mock  
 private RestTemplate restTemplate; // Mock API call  
  
 @InjectMocks  
 private UserService userService; // Service being tested  
  
 @Test  
 void testGetUserById() {  
 // 🔹 Step 1: Create mock API response  
 Map<String, Object> mockResponse = new HashMap<>();  
 mockResponse.put("id", 1);  
 mockResponse.put("name", "John Doe");  
 mockResponse.put("email", "john@example.com");  
  
 // 🔹 Step 2: Mock RestTemplate behavior  
 *when*(restTemplate.getForObject("https://jsonplaceholder.typicode.com/users/1", Map.class))  
 .thenReturn(mockResponse);  
  
 // 🔹 Step 3: Call the service method  
 Map<String, Object> response = userService.getUserById(1);  
  
 // 🔹 Step 4: Validate response  
 *assertEquals*(1, response.get("id"));  
 *assertEquals*("John Doe", response.get("name"));  
 *assertEquals*("john@example.com", response.get("email"));  
  
 // 🔹 Step 5: Verify the API was called exactly once  
 *verify*(restTemplate, *times*(1)).getForObject("https://jsonplaceholder.typicode.com/users/1", Map.class);  
 }  
}

1. **What is expected output from this**

Main class

class Calc {  
 public int modifyNumsAndAdd(int a, int b) {  
 if(a<0 || b<0){  
 throw new IllegalArgumentException("Argument can't be less than 0");  
 }  
 a = a + b;  
 b = 2 \* b;  
 return a + b;  
 }  
}  
  
public class LearnMockito {  
 private Calc calc;  
 public LearnMockito(Calc calc) {  
 this.calc = calc;  
 }  
 public int addNums(int a, int b) throws IllegalArgumentException {  
 try {  
 return calc.modifyNumsAndAdd(a, b);  
 } catch (Exception e) {  
 throw new IllegalArgumentException(e);  
 }  
 }  
}

Test Class

@ExtendWith(MockitoExtension.class)  
public class LearnMockitoTest {  
  
 @Spy  
 private Calc calc;  
 @InjectMocks  
 private LearnMockito lm;  
  
 @Test  
 void testAddNums() throws Exception {  
 // Define mock behavior  
 *when*(calc.modifyNumsAndAdd(-1,3)).thenReturn(6);  
  
 *assertEquals*(6,lm.addNums(-1,3)); //a+3b  
 }  
}

Output:

This code will fail, since we are using spy object with when().thenReturn(), so for -1,3 real code of calc runs and throws exception. So for spy if we want that real method not runs, we can use doReturn().

Also if parameters were 1,3, if real code runs the output should be 10. But in when().thenReturn() we gave output as 6. Then in this case we will get output as 6 only. Issue comes in spy only when code has side-effect.