**Junit and Mockito**

**─**

Contents

[**1. Introduction to testing** 2](#_Toc190518302)

[# Unit testing 4](#_Toc190518303)

[**2. Junit** 4](#_Toc190518304)

[# setting up Junit 4](#_Toc190518305)

[# First Test Case 5](#_Toc190518306)

[# Assertions 6](#_Toc190518307)

[# Important Annotations 7](#_Toc190518308)

[# Test Fixtures 8](#_Toc190518309)

[# Parameterized Test 10](#_Toc190518310)

[**3. Mockito** 11](#_Toc190518311)

[# Setting up Mockito 13](#_Toc190518312)

[# Why Mockito? 14](#_Toc190518313)

[# Mock vs Stub vs Spy 15](#_Toc190518314)

[# Core Mockito Annotations 18](#_Toc190518315)

[# Mockito Behavior Control 20](#_Toc190518316)

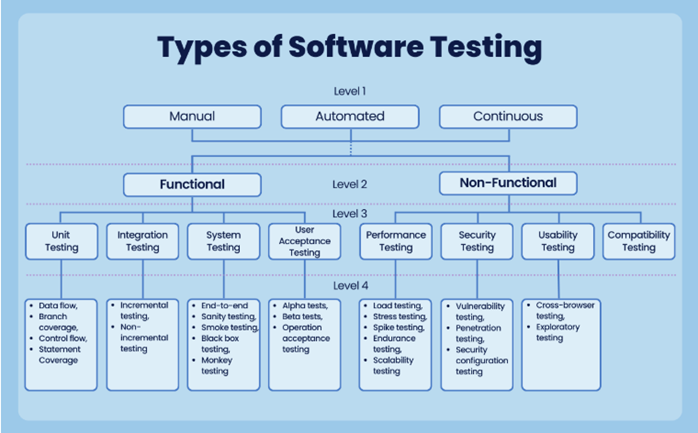
[**4. Interview Questions** 22](#_Toc190518317)

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



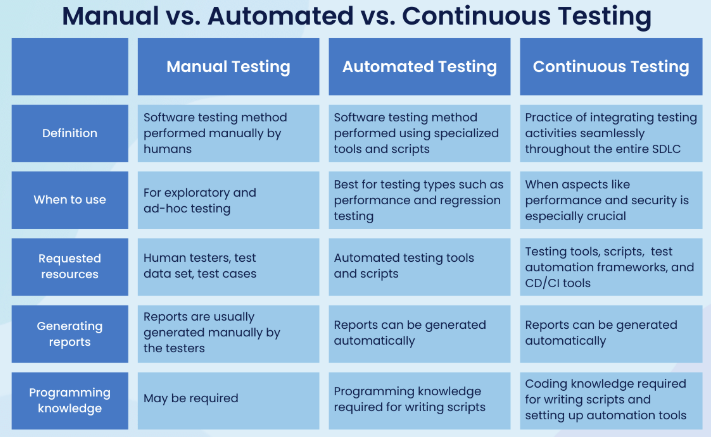
Functional and non-functional testing -

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"));

## **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.**