**Mockito Interview Questions**

Contents

[Mockito 2](#_Toc10889)

[1. What is Mockito and why is it used in unit testing? 2](#_Toc12182)

[2. Explain the difference between a mock and a spy in Mockito. 3](#_Toc16336)

[3. How do you create a mock object using Mockito? 6](#_Toc9901)

[4. What is the purpose of the @InjectMocks annotation? 8](#_Toc2662)

[5. How do you use the when-thenReturn syntax in Mockito? 11](#_Toc14719)

[6. How can you verify interactions with mock objects using Mockito? 13](#_Toc29659)

[7. Describe how you would throw an exception from a mock method using Mockito. 17](#_Toc10880)

[8. Can you explain the purpose and usage of argument matchers in Mockito? 19](#_Toc24404)

[9. What are the limitations of using Mockito for unit testing? 21](#_Toc22719)

[10. What is an ArgumentCaptor ? 23](#_Toc32312)

[11. How do you use Mockito in combination with JUnit for writing unit tests? 25](#_Toc3155)

[12. How do you verify the order of method calls on a mock object? 29](#_Toc2916)

[13. How do you reset a mock object in Mockito? 31](#_Toc25400)

[14. How do you test a void method using Mockito? 33](#_Toc10516)

[15. How do you create custom annotations for JUnit — especially in JUnit 5 — to bundle or reuse common testing configurations, annotations, or behavior? 36](#_Toc18987)

[16. How does @Mock works internally in Mockito: how does the framework handles mock creation under the hood? 37](#_Toc16083)

[Step-by-step Mechanism 38](#_Toc2370)

[Sources in Mockito Codebase 39](#_Toc24628)

# Mockito

### What is Mockito and why is it used in unit testing?

Discuss how Mockito is a popular mocking framework for Java used to create mock objects for testing purposes.

Mockito is a popular open-source mocking framework for Java, designed to simplify the process of creating and using mock objects in unit tests. Here’s why it’s widely used and what makes it so handy:

**What is Mockito?**

Mockito enables you to create mock objects that simulate the behavior of real objects in a controlled way. These mock objects help isolate the unit of code you're testing by replacing actual dependencies with mocks, allowing you to focus solely on the behavior of the code under test.

**Why Use Mockito in Unit Testing?**

1. **Isolation**: Mockito helps isolate the unit of code being tested by mocking dependencies. This ensures that tests are focused and only fail due to issues within the unit itself, not because of external dependencies.
2. **Control and Flexibility**: With Mockito, you have fine-grained control over how mocks behave. You can specify return values, throw exceptions, and verify interactions with the mock objects.
3. **Simplicity**: Mockito’s API is straightforward and intuitive. It integrates seamlessly with other testing frameworks like JUnit, making it easy to set up and use.
4. **Readability**: Tests written with Mockito are often more readable and maintainable. The syntax is clear, and the intention of each mock behavior is explicit.
5. **Verification**: Mockito allows you to verify that certain interactions with mocks occurred, ensuring that the unit under test behaves as expected.

**Example Usage:**

Creating a mock object and using it in a test:

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

import static org.mockito.Mockito.\*;

import static org.junit.jupiter.api.Assertions.\*;

class MyServiceTest {

@Test

void testMyService() {

// Create a mock object

MyDependency mockDependency = Mockito.mock(MyDependency.class);

// Define behavior for the mock object

when(mockDependency.someMethod()).thenReturn("mocked value");

// Use the mock object in the service

MyService myService = new MyService(mockDependency);

String result = myService.performAction();

// Verify interactions and assert results

assertEquals("mocked value", result);

verify(mockDependency).someMethod();

}

}

In this example:

* **Mocking**: Mockito.mock(MyDependency.class) creates a mock object.
* **Stubbing**: when(mockDependency.someMethod()).thenReturn("mocked value") defines the behavior of the mock.
* **Verification**: verify(mockDependency).someMethod() checks that the method was called.

Mockito streamlines testing by letting you mock dependencies easily, ensuring that your unit tests remain focused, clear, and effective.

### Explain the difference between a mock and a spy in Mockito.

Describe how mocks are proxy objects that simulate the behavior of real objects, while spies wrap real objects and can stub or verify methods selectively.

**Mocks:**

* **Definition**: Mocks are proxy objects that simulate the behavior of real objects. They are used to create dummy objects that imitate real objects' behavior without invoking any actual logic.
* **Usage**: Use mocks when you want to isolate the class under test by providing controlled behavior for its dependencies.
* **Behavior**: You need to explicitly define the behavior of the mock using methods like when(...).thenReturn(...).
* **Interactions**: You can verify interactions with mocks to check if specific methods were called with expected parameters.

**Example of Creating and Using a Mock**:

java

Copy

import static org.mockito.Mockito.\*;

public class MockExample {

@Test

void testWithMock() {

// Create a mock object

List<String> mockedList = mock(List.class);

// Define behavior

when(mockedList.size()).thenReturn(5);

// Use the mock

assertEquals(5, mockedList.size());

// Verify interactions

verify(mockedList).size();

}

}

**Spies:**

* **Definition**: Spies wrap real objects and allow you to stub or verify specific methods selectively. They enable partial mocking by calling actual methods unless you explicitly stub them.
* **Usage**: Use spies when you want to retain the real object’s behavior but also want to control certain methods.
* **Behavior**: By default, spies call the actual methods of the real object. You can selectively override methods using doReturn(...).when(...).
* **Interactions**: You can verify interactions with spies similarly to mocks.

**Example of Creating and Using a Spy**:

java

Copy

import static org.mockito.Mockito.\*;

public class SpyExample {

@Test

void testWithSpy() {

// Create a real object

List<String> realList = new ArrayList<>();

realList.add("one");

// Wrap the real object with a spy

List<String> spyList = spy(realList);

// Verify initial behavior

assertEquals(1, spyList.size());

// Override behavior

doReturn(5).when(spyList).size();

// Use the spy

assertEquals(5, spyList.size());

// Verify interactions

verify(spyList).size();

}

}

**Key Differences:**

* **Mocks**: Proxy objects; require explicit behavior definition; do not invoke real methods.
* **Spies**: Wrap real objects; retain actual behavior unless overridden; can invoke real methods and selectively stub.

Mocks are great for completely isolating the unit under test, while spies are useful for partial mocking where some actual behavior is retained. Understanding when to use each helps in writing effective and maintainable unit tests.

### How do you create a mock object using Mockito?

Explain using the @Mock annotation and the Mockito.mock() method.

**Step-by-Step Guide to Creating a Mock Object:**

**1. Add Mockito Dependency:** First, ensure you have the Mockito dependency in your pom.xml if you’re using Maven:

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.12.4</version>

<scope>test</scope>

</dependency>

**2. Create a Mock Object:** You can create a mock object using the Mockito.mock() method or the @Mock annotation. Here are both approaches:

**Using**Mockito.mock()**:**

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import java.util.List;

public class MockExample {

@Test

void testMockObject() {

// Create a mock object

List<String> mockedList = mock(List.class);

// Define behavior for the mock

when(mockedList.size()).thenReturn(5);

// Use the mock object

System.out.println(mockedList.size()); // Prints: 5

// Verify interactions with the mock

verify(mockedList).size();

}

}

**Using**@Mock**Annotation:**

First, use the @ExtendWith(MockitoExtension.class) annotation to initialize the mocks automatically:

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.junit.jupiter.MockitoExtension;

import static org.mockito.Mockito.\*;

import org.mockito.Mock;

import org.junit.jupiter.api.Test;

@ExtendWith(MockitoExtension.class)

public class MockAnnotationExample {

// Create a mock using the @Mock annotation

@Mock

List<String> mockedList;

@Test

void testMockAnnotation() {

// Define behavior for the mock

when(mockedList.size()).thenReturn(10);

// Use the mock object

System.out.println(mockedList.size()); // Prints: 10

// Verify interactions with the mock

verify(mockedList).size();

}

}

**Explanation:**

1. **Create Mock**:
   * Mockito.mock(Class<T> classToMock): This static method creates a mock instance of the specified class.
   * @Mock: This annotation creates and injects the mock objects. You need to enable Mockito annotations using the @ExtendWith(MockitoExtension.class) annotation.
2. **Define Behavior**:
   * when-thenReturn: Use when(mock.method()).thenReturn(value) to specify the behavior of the mock object.
3. **Use and Verify**:
   * You can use the mock object as you would use a real instance.
   * verify: Use verify(mock).method() to confirm that certain methods were called on the mock object.

### What is the purpose of the @InjectMocks annotation?

Discuss how @InjectMocks is used to inject mocked dependencies into the object being tested.

The @InjectMocks annotation in Mockito is used to automatically inject mock dependencies into the object being tested. It helps ensure that the class under test is provided with its required dependencies without needing to manually set them up.

**Purpose of @InjectMocks:**

* **Automatic Injection**: Simplifies the creation and injection of mock dependencies into the class being tested.
* **Dependency Management**: Ensures that all necessary dependencies are provided, making the setup process smoother and less error-prone.
* **Code Cleanliness**: Reduces boilerplate code by handling the injection automatically, keeping the test code clean and focused on testing logic rather than setup.

**How @InjectMocks Works:**

1. **Annotation Placement**:
   * Place @InjectMocks on the field where the tested object is declared.
   * Ensure that mock dependencies are annotated with @Mock or created using Mockito.mock().
2. **Initialization**:
   * Use @ExtendWith(MockitoExtension.class) in JUnit 5 to automatically initialize mocks and inject them.

**Example:**

**Service Class (Class under test)**:

public class UserService {

private final UserRepository userRepository;

// Constructor injection of the dependency

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

public User findUserById(Long id) {

return userRepository.findById(id).orElse(null);

}

}

**Test Class**:

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.junit.jupiter.MockitoExtension;

import static org.mockito.Mockito.\*;

import static org.junit.jupiter.api.Assertions.\*;

@ExtendWith(MockitoExtension.class)

public class UserServiceTest {

// Mock the dependency

@Mock

private UserRepository userRepository;

// Inject the mock into the class under test

@InjectMocks

private UserService userService;

@Test

void testFindUserById() {

User user = new User();

user.setId(1L);

// Define behavior for the mock

when(userRepository.findById(1L)).thenReturn(Optional.of(user));

// Test the method

User result = userService.findUserById(1L);

// Assert the result

assertEquals(1L, result.getId());

// Verify the interaction with the mock

verify(userRepository).findById(1L);

}

}

**Explanation:**

* @Mock: Creates a mock instance of UserRepository.
* @InjectMocks: Injects the mock UserRepository into the UserService instance automatically.
* @ExtendWith(MockitoExtension.class): Ensures that Mockito annotations are processed and mocks are initialized.

### How do you use the when-thenReturn syntax in Mockito?

Demonstrate how to set up mock behavior using Mockito.when() and thenReturn() for defining return values.

The when-thenReturn syntax in Mockito is used to define the behavior of mock objects. This is particularly useful for setting up expectations and return values for methods in your mocks.

**Step-by-Step Example:**

**1. Add Mockito Dependency:** First, make sure you have Mockito in your project by adding it to your pom.xml:

xml

Copy

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.12.4</version>

<scope>test</scope>

</dependency>

**2. Create a Mock Object and Use**when-thenReturn**:** Let's create a simple example using a list.

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import java.util.List;

import static org.junit.jupiter.api.Assertions.\*;

public class WhenThenReturnExample {

@Test

void testWhenThenReturn() {

// Create a mock object of List

List<String> mockedList = mock(List.class);

// Define behavior: when size() is called, return 10

when(mockedList.size()).thenReturn(10);

// Use the mock object

int size = mockedList.size();

// Assert the behavior

assertEquals(10, size);

// Verify the interaction

verify(mockedList).size();

}

}

**Explanation:**

1. **Create a Mock Object**:

List<String> mockedList = mock(List.class);

* + mock(List.class) creates a mock object of the List interface.

1. **Define Behavior**:

when(mockedList.size()).thenReturn(10);

* + when(mockedList.size()) sets up the expectation.
  + thenReturn(10) specifies that when size() is called, the mock should return 10.

1. **Use the Mock Object**:

int size = mockedList.size();

* + Calling size() on the mock object returns 10 as defined.

1. **Assert the Behavior**:

assertEquals(10, size);

* + Asserts that the returned value is as expected.

1. **Verify the Interaction**:

verify(mockedList).size();

* + Verifies that the size() method was called on the mock.

**More Complex Example with Parameters:**

You can also define behavior for methods with parameters:

java

Copy

@Test

void testWhenThenReturnWithParameters() {

// Create a mock object of List

List<String> mockedList = mock(List.class);

// Define behavior: when get(0) is called, return "Mockito"

when(mockedList.get(0)).thenReturn("Mockito");

// Use the mock object

String element = mockedList.get(0);

// Assert the behavior

assertEquals("Mockito", element);

// Verify the interaction

verify(mockedList).get(0);

}

In this case:

* when(mockedList.get(0)).thenReturn("Mockito") sets up the expectation for the get(0) method to return "Mockito".

**Conclusion:**

The when-thenReturn syntax is key to setting up mock behavior in Mockito, ensuring your tests focus on the logic under test rather than the behavior of dependencies.

### How can you verify interactions with mock objects using Mockito?

Explain using Mockito.verify() to check that specific methods were called with expected arguments.

Verifying interactions with mock objects using Mockito is straightforward and powerful. Here’s a detailed explanation, along with a practical example:

**Purpose of Verification:**

Verification ensures that specific methods on mock objects are called with expected arguments. This helps confirm that the code under test interacts with its dependencies as intended.

**Using Mockito.verify():**

**1. Setup the Mock and Class Under Test:**

Start by creating a mock object and the class under test that depends on this mock:

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import java.util.List;

public class VerifyExample {

@Test

void testVerifyInteractions() {

// Create a mock object

List<String> mockedList = mock(List.class);

// Use the mock object in some code

mockedList.add("one");

mockedList.clear();

// Verify that specific methods were called on the mock with expected arguments

verify(mockedList).add("one"); // Verifies that add("one") was called

verify(mockedList).clear(); // Verifies that clear() was called

}

}

**Explanation:**

1. **Creating a Mock Object:**

List<String> mockedList = mock(List.class);

* + mock(List.class) creates a mock object of the List interface.

1. **Using the Mock Object:**

mockedList.add("one");

mockedList.clear();

* + These calls simulate interactions with the mock object.

1. **Verifying Interactions:**

verify(mockedList).add("one");

verify(mockedList).clear();

* + verify(mockedList).add("one"): Confirms that the add method was called with the argument "one".
  + verify(mockedList).clear(): Confirms that the clear method was called.

**Advanced Verification:**

You can also verify the number of times a method was called and more complex interactions:

**1. Verify Method Call Count:**

verify(mockedList, times(1)).add("one"); // Verifies add("one") was called exactly once

verify(mockedList, never()).remove("two"); // Verifies remove("two") was never called

verify(mockedList, atLeastOnce()).clear(); // Verifies clear() was called at least once

1. **Verify with Argument Matchers:**

verify(mockedList).add(anyString()); // Verifies add() was called with any String argument

verify(mockedList).add(argThat(arg -> arg.length() > 2)); // Verifies add() was called with a String longer than 2 characters

**Example with Class Under Test:**

Here’s a more complete example with a service class that uses a repository:

public class UserService {

private final UserRepository userRepository;

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

public User createUser(String name) {

User user = new User(name);

userRepository.save(user);

return user;

}

}

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

public class UserServiceTest {

@Test

void testCreateUser() {

// Create a mock UserRepository

UserRepository mockRepository = mock(UserRepository.class);

// Create a UserService with the mock repository

UserService userService = new UserService(mockRepository);

// Call the method under test

userService.createUser("John");

// Verify that the save method was called with the expected User

verify(mockRepository).save(any(User.class));

}

}

In this example:

* **Mock**: UserRepository is mocked.
* **Interaction**: createUser calls the save method on the mock.
* **Verification**: verify(mockRepository).save(any(User.class)) confirms that save was called with any User object.

Verification ensures your code behaves as expected by confirming that interactions with dependencies are correct.

### Describe how you would throw an exception from a mock method using Mockito.

Show how to use when-thenThrow to simulate exceptions from mock methods.

ou can use the when-thenThrow syntax to simulate exceptions. Here’s how it works:

**Step-by-Step Example:**

**1. Add Mockito Dependency:** Ensure you have the Mockito dependency in your pom.xml:

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.12.4</version>

<scope>test</scope>

</dependency>

**2. Set Up the Mock and Define Behavior:** Use when-thenThrow to specify that a mock method should throw an exception.

Copy

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.\*;

public class WhenThenThrowExample {

@Test

void testWhenThenThrow() {

// Create a mock object

List<String> mockedList = mock(List.class);

// Define behavior: when get(0) is called, throw an IndexOutOfBoundsException

when(mockedList.get(0)).thenThrow(new IndexOutOfBoundsException("Index out of bounds"));

// Use the mock object and catch the exception

Exception exception = assertThrows(IndexOutOfBoundsException.class, () -> {

mockedList.get(0);

});

// Assert the exception message

assertEquals("Index out of bounds", exception.getMessage());

// Verify the interaction

verify(mockedList).get(0);

}

}

**Explanation:**

1. **Create a Mock Object:**

List<String> mockedList = mock(List.class);

* + mock(List.class) creates a mock object of the List interface.

1. **Define Exception Behavior:**

when(mockedList.get(0)).thenThrow(new IndexOutOfBoundsException("Index out of bounds"));

* + when(mockedList.get(0)): Sets up the expectation that get(0) will be called.
  + thenThrow(new IndexOutOfBoundsException("Index out of bounds")): Specifies that when get(0) is called, it should throw an IndexOutOfBoundsException with the specified message.

1. **Use the Mock and Catch the Exception:**

Exception exception = assertThrows(IndexOutOfBoundsException.class, () -> {

mockedList.get(0);

});

* + assertThrows(IndexOutOfBoundsException.class, () -> { mockedList.get(0); }): Executes the code that is expected to throw the exception and captures it.

1. **Assert the Exception Message:**

assertEquals("Index out of bounds", exception.getMessage());

* + Confirms that the exception message matches the expected message.

1. **Verify the Interaction:**

verify(mockedList).get(0);

* + Verifies that the get(0) method was called on the mock object.

**Conclusion:**

Using when-thenThrow, you can effectively simulate exceptions from mock methods, allowing you to test how your code handles various error scenarios. This is crucial for ensuring your application is robust and can gracefully handle unexpected conditions.

### Can you explain the purpose and usage of argument matchers in Mockito?

Discuss how to use Mockito.any(), Mockito.eq(), and other matchers for flexible argument matching.

Argument matchers in Mockito are used to make your tests more flexible by allowing you to specify criteria for method arguments rather than requiring exact values. This is particularly useful when the exact arguments might vary, but you still want to verify interactions or define behavior for mocks.

**Purpose of Argument Matchers:**

1. **Flexibility**: Allow you to verify method calls with varying arguments.
2. **Simplification**: Make it easier to write tests when the exact argument values are not crucial.
3. **Readability**: Improve the readability of your tests by focusing on the behavior rather than the specific values.

**Common Argument Matchers:**

**1.**Mockito.any()**:**

* Matches any value of the specified type.

import static org.mockito.Mockito.\*;

@Test

void testAnyMatcher() {

List<String> mockedList = mock(List.class);

when(mockedList.get(anyInt())).thenReturn("element");

assertEquals("element", mockedList.get(0));

assertEquals("element", mockedList.get(5));

verify(mockedList, times(2)).get(anyInt());

}

**2.**Mockito.eq()**:**

* Matches an exact value. Useful when combined with other matchers.

@Test

void testEqMatcher() {

List<String> mockedList = mock(List.class);

mockedList.add("one");

verify(mockedList).add(eq("one"));

}

**3.**Mockito.argThat()**:**

* Matches arguments based on a custom condition.

@Test

void testArgThatMatcher() {

List<String> mockedList = mock(List.class);

mockedList.add("Mockito");

verify(mockedList).add(argThat(arg -> arg.length() == 7));

}

**4.**Mockito.isA()**:**

* Matches if the argument is an instance of the specified class.

@Test

void testIsAMatcher() {

List<String> mockedList = mock(List.class);

when(mockedList.contains(isA(String.class))).thenReturn(true);

assertTrue(mockedList.contains("test"));

}

**Combining Matchers:**

You can combine matchers to handle more complex scenarios:

@Test

void testCombinedMatchers() {

Map<String, String> mockedMap = mock(Map.class);

when(mockedMap.put(anyString(), eq("value"))).thenReturn("previousValue");

assertEquals("previousValue", mockedMap.put("key", "value"));

verify(mockedMap).put(anyString(), eq("value"));

}

**Key Points:**

* **Matchers**: Provide flexibility in verifying interactions and defining behaviors.
* **Readability**: Make tests clearer and more focused on behavior rather than specific values.
* **Combination**: Combine matchers for complex scenarios, enhancing the expressiveness of your tests.

### What are the limitations of using Mockito for unit testing?

Reflect on potential limitations like difficulty in mocking static methods, constructors, or final classes, and when it's better to use other testing approaches.

Mockito is a powerful tool for unit testing, but like any tool, it comes with its own set of limitations. Here's a detailed look at some of those limitations and when you might need to consider other testing approaches:

**Limitations of Mockito:**

1. **Mocking Static Methods**:
   * **Challenge**: By default, Mockito cannot mock static methods.
   * **Workaround**: Use tools like PowerMockito, but they can add complexity and are generally less recommended due to design and maintainability concerns.
2. **Mocking Constructors**:
   * **Challenge**: Mockito does not support mocking constructors. It can only mock methods on existing objects.
   * **Workaround**: Refactor code to use dependency injection or factory patterns, avoiding the need to mock constructors.
3. **Mocking Final Classes and Methods**:
   * **Challenge**: By default, Mockito cannot mock final classes or methods.
   * **Workaround**: Use tools like PowerMockito or the mock-maker-inline extension for Mockito 2.1+ to mock final methods, but consider the design implications of making classes or methods final.
4. **Partial Mocks**:
   * **Challenge**: While Mockito can create partial mocks using spies, this can lead to complex and brittle tests.
   * **Workaround**: Use partial mocks sparingly and prefer to test real instances with well-defined mocks for dependencies.
5. **Complex Verifications**:
   * **Challenge**: Complex verification logic can make tests harder to read and maintain.
   * **Workaround**: Keep verifications simple and clear. Use argument captors and custom argument matchers to simplify complex verifications.
6. **Limited Integration with Legacy Code**:
   * **Challenge**: Legacy code with poor design (tight coupling, lack of interfaces) can be difficult to mock effectively.
   * **Workaround**: Gradually refactor legacy code to improve testability, introducing interfaces and decoupling dependencies.

**When to Use Other Testing Approaches:**

1. **Integration Testing**:
   * **When**: When you need to test the interaction between multiple components or systems.
   * **Tool**: Use tools like Spring Boot Test for testing Spring applications end-to-end.
2. **Functional Testing**:
   * **When**: To validate the functionality of the application from the user's perspective.
   * **Tool**: Use tools like Selenium or Cypress for web applications.
3. **Load and Performance Testing**:
   * **When**: To ensure the application can handle expected loads and performs well under stress.
   * **Tool**: Use tools like JMeter or Gatling for load testing.
4. **Behavior-Driven Development (BDD)**:
   * **When**: To write tests in a more natural language and involve non-technical stakeholders.
   * **Tool**: Use frameworks like Cucumber for defining feature files and step definitions.

**Conclusion:**

Mockito is invaluable for unit testing and mocking dependencies, but it's essential to recognize its limitations and know when to employ other testing strategies for a comprehensive testing approach. Proper design and architecture choices can significantly enhance the effectiveness of Mockito and other testing tools.

### What is an ArgumentCaptor ?

An ArgumentCaptor in Mockito is a powerful feature that captures argument values for further assertions. It’s particularly useful when you want to verify what arguments were passed to a mock’s method during a test. Here’s how it works:

**Step-by-Step Example:**

**1. Add Mockito Dependency:** Make sure you have the Mockito dependency in your pom.xml:

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.12.4</version>

<scope>test</scope>

</dependency>

**2. Set Up the Mock and ArgumentCaptor:** Use ArgumentCaptor to capture the arguments passed to a mock method.

import static org.mockito.Mockito.\*;

import static org.mockito.ArgumentCaptor.\*;

import org.junit.jupiter.api.Test;

import org.mockito.ArgumentCaptor;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class ArgumentCaptorExample {

@Test

void testArgumentCaptor() {

// Create a mock object

List<String> mockedList = mock(List.class);

// Use the mock object

mockedList.add("Mockito");

// Create an ArgumentCaptor

ArgumentCaptor<String> captor = ArgumentCaptor.forClass(String.class);

// Verify and capture the argument

verify(mockedList).add(captor.capture());

// Assert the captured value

assertEquals("Mockito", captor.getValue());

}

}

**Explanation:**

1. **Create a Mock Object:**

List<String> mockedList = mock(List.class);

* + Creates a mock object of the List interface.

1. **Use the Mock Object:**

mockedList.add("Mockito");

* + Simulates the interaction with the mock object.

1. **Create an ArgumentCaptor:**

ArgumentCaptor<String> captor = ArgumentCaptor.forClass(String.class);

* + Initializes an ArgumentCaptor for the String type.

1. **Verify and Capture the Argument:**

verify(mockedList).add(captor.capture());

* + Verifies that the add method was called and captures the argument passed to it.

1. **Assert the Captured Value:**

assertEquals("Mockito", captor.getValue());

* + Asserts that the captured value is "Mockito".

**Advanced Usage:**

You can capture multiple arguments and handle more complex scenarios:

@Test

void testMultipleArgumentCaptor() {

// Create a mock object

List<String> mockedList = mock(List.class);

// Use the mock object

mockedList.add("Mockito");

mockedList.add("JUnit");

// Create an ArgumentCaptor

ArgumentCaptor<String> captor = ArgumentCaptor.forClass(String.class);

// Verify and capture the arguments

verify(mockedList, times(2)).add(captor.capture());

// Assert the captured values

assertEquals("Mockito", captor.getAllValues().get(0));

assertEquals("JUnit", captor.getAllValues().get(1));

}

In this advanced example:

* verify(mockedList, times(2)).add(captor.capture()): Captures arguments for multiple calls to the add method.
* captor.getAllValues(): Retrieves a list of all captured values.

### How do you use Mockito in combination with JUnit for writing unit tests?

Provide an example of integrating Mockito with JUnit, setting up mocks, stubbing methods, and verifying interactions.

Combining Mockito with JUnit is a common and powerful approach for writing unit tests in Java. Here’s a step-by-step guide to integrating Mockito with JUnit, setting up mocks, stubbing methods, and verifying interactions.

**Step-by-Step Example:**

**1. Add Dependencies:**

First, ensure you have the necessary dependencies in your pom.xml for both JUnit and Mockito:

<dependency>

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

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

<version>5.8.2</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.12.4</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

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

<version>3.12.4</version>

<scope>test</scope>

</dependency>

**2. Create the Class to Test:**

Here’s a simple service class that uses a repository:

public class UserService {

private final UserRepository userRepository;

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

public User findUserById(Long id) {

return userRepository.findById(id).orElse(null);

}

public User createUser(String name) {

User user = new User(name);

userRepository.save(user);

return user;

}

}

**3. Set Up the Test Class:**

Create the test class, set up mocks, and use @ExtendWith(MockitoExtension.class) to enable Mockito.

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.junit.jupiter.MockitoExtension;

import static org.mockito.Mockito.\*;

import static org.junit.jupiter.api.Assertions.\*;

@ExtendWith(MockitoExtension.class)

public class UserServiceTest {

@Mock

private UserRepository userRepository;

@InjectMocks

private UserService userService;

@Test

void testFindUserById() {

User user = new User(1L, "John");

// Stub the findById method

when(userRepository.findById(1L)).thenReturn(Optional.of(user));

// Call the method under test

User foundUser = userService.findUserById(1L);

// Assert the result

assertNotNull(foundUser);

assertEquals(1L, foundUser.getId());

assertEquals("John", foundUser.getName());

// Verify the interaction with the mock

verify(userRepository).findById(1L);

}

@Test

void testCreateUser() {

User user = new User("Jane");

// Call the method under test

userService.createUser("Jane");

// Verify the interaction with the mock

verify(userRepository).save(user);

}

}

**Explanation:**

1. **Annotations**:
   * @ExtendWith(MockitoExtension.class): Enables Mockito’s annotation processing, initializing mocks and injecting them where needed.
   * @Mock: Creates a mock instance of UserRepository.
   * @InjectMocks: Injects the mock UserRepository into the UserService instance automatically.
2. **Stubbing Methods**:

when(userRepository.findById(1L)).thenReturn(Optional.of(user));

* + Sets up the behavior for the findById method to return a specific user when called with 1L.

1. **Calling Methods and Assertions**:
   * Calls the method under test and uses assertions to verify the results.
   * assertNotNull(foundUser): Asserts that the user is found.
   * assertEquals("John", foundUser.getName()): Checks that the user’s name is as expected.
2. **Verifying Interactions**:

verify(userRepository).findById(1L);

* + Verifies that the findById method was called on the mock with the expected argument 1L.

### How do you verify the order of method calls on a mock object?

Verifying the order of method calls on a mock object in Mockito is straightforward and useful for ensuring that your code adheres to the expected call sequence. You can use the InOrder class to achieve this. Here's how you do it:

**Example:**

Let's assume we have a simple service that interacts with a repository and we want to verify the order of these interactions.

**1. Add Mockito Dependency:** Ensure you have the Mockito dependency in your pom.xml:

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.12.4</version>

<scope>test</scope>

</dependency>

**2. Set Up the Classes to Test:**

**Service Class:**

public class OrderService {

private final OrderRepository orderRepository;

public OrderService(OrderRepository orderRepository) {

this.orderRepository = orderRepository;

}

public void processOrder() {

orderRepository.validateOrder();

orderRepository.saveOrder();

}

}

**Repository Interface:**

public interface OrderRepository {

void validateOrder();

void saveOrder();

}

**3. Write the Test with**InOrder**:** Use Mockito's InOrder to verify the order of method calls.

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.InOrder;

public class OrderServiceTest {

@Test

void testOrderOfMethodCalls() {

// Create a mock OrderRepository

OrderRepository mockRepository = mock(OrderRepository.class);

// Create OrderService with the mock repository

OrderService orderService = new OrderService(mockRepository);

// Call the method under test

orderService.processOrder();

// Verify the order of method calls

InOrder inOrder = inOrder(mockRepository);

inOrder.verify(mockRepository).validateOrder();

inOrder.verify(mockRepository).saveOrder();

}

}

**Explanation:**

1. **Creating Mocks:**

OrderRepository mockRepository = mock(OrderRepository.class);

* + Creates a mock object of the OrderRepository interface.

1. **Calling Method:**

orderService.processOrder();

* + Calls the processOrder method which internally calls validateOrder and saveOrder.

1. **Verifying Order:**

InOrder inOrder = inOrder(mockRepository);

inOrder.verify(mockRepository).validateOrder();

inOrder.verify(mockRepository).saveOrder();

* + InOrder inOrder = inOrder(mockRepository): Creates an InOrder verifier for the mock repository.
  + inOrder.verify(mockRepository).validateOrder(): Verifies that validateOrder was called first.
  + inOrder.verify(mockRepository).saveOrder(): Verifies that saveOrder was called after validateOrder.

Using InOrder, you can easily ensure that your mock methods are called in the expected sequence, which is crucial for testing workflows and business logic that depend on the order of operations.

### How do you reset a mock object in Mockito?

Resetting a mock object in Mockito can be useful when you need to clear any stubbing and interaction history to start fresh. This is typically done using the reset method.

**Step-by-Step Example:**

**1. Add Mockito Dependency:** Ensure you have the Mockito dependency in your pom.xml:

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.12.4</version>

<scope>test</scope>

</dependency>

**2. Use**reset**to Clear Mock:** Here's an example of how to reset a mock object:

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import java.util.List;

import static org.junit.jupiter.api.Assertions.\*;

public class ResetMockExample {

@Test

void testResetMock() {

// Create a mock object

List<String> mockedList = mock(List.class);

// Define behavior

when(mockedList.size()).thenReturn(10);

assertEquals(10, mockedList.size());

// Verify interaction

verify(mockedList).size();

// Reset the mock

reset(mockedList);

// After reset, mock has no behavior defined and no interactions recorded

when(mockedList.size()).thenReturn(20);

assertEquals(20, mockedList.size());

// Verify interaction post reset

verify(mockedList).size();

}

}

**Explanation:**

1. **Create a Mock Object:**

List<String> mockedList = mock(List.class);

* + Creates a mock object of the List interface.

1. **Define Behavior and Verify Interactions:**

when(mockedList.size()).thenReturn(10);

assertEquals(10, mockedList.size());

verify(mockedList).size();

* + Stubs the size method to return 10 and verifies the interaction.

1. **Reset the Mock:**

reset(mockedList);

* + Resets the mock, clearing all stubbing and recorded interactions.

1. **Define New Behavior and Verify New Interactions:**

when(mockedList.size()).thenReturn(20);

assertEquals(20, mockedList.size());

verify(mockedList).size();

* + After reset, redefines the behavior and verifies new interactions.

**Key Points:**

* **Use Case**: Resetting is useful when you want to reuse a mock in multiple test scenarios without creating a new mock instance each time.
* **Caution**: Use reset sparingly. Excessive use of reset might indicate that the test setup could be improved for clarity and separation of concerns.

By using reset, you can keep your tests clean and focused, especially when reusing mocks across different test methods or scenarios.

### How do you test a void method using Mockito?

Testing a void method in Mockito is typically done by verifying that certain interactions occurred as expected. You can't directly assert a return value because void methods don't return anything. Instead, you focus on verifying that the method caused the correct side effects or interactions.

Here's a step-by-step example to illustrate how to test a void method using Mockito:

**Step-by-Step Example:**

**1. Add Mockito Dependency:** Ensure you have the Mockito dependency in your pom.xml:

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.12.4</version>

<scope>test</scope>

</dependency>

**2. Create the Class to Test:**

**Service Class with a Void Method:**

public class OrderService {

private final OrderRepository orderRepository;

public OrderService(OrderRepository orderRepository) {

this.orderRepository = orderRepository;

}

public void processOrder(Order order) {

if (order.isValid()) {

orderRepository.save(order);

} else {

orderRepository.reject(order);

}

}

}

**Repository Interface:**

public interface OrderRepository {

void save(Order order);

void reject(Order order);

}

**3. Write the Test for the Void Method:** Use Mockito to verify that the void method caused the expected interactions.

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

public class OrderServiceTest {

@Test

void testProcessOrder() {

// Create a mock OrderRepository

OrderRepository mockRepository = mock(OrderRepository.class);

// Create an instance of OrderService with the mock repository

OrderService orderService = new OrderService(mockRepository);

// Create a valid Order

Order validOrder = mock(Order.class);

when(validOrder.isValid()).thenReturn(true);

// Call the void method

orderService.processOrder(validOrder);

// Verify the interaction with the mock repository

verify(mockRepository).save(validOrder);

// Create an invalid Order

Order invalidOrder = mock(Order.class);

when(invalidOrder.isValid()).thenReturn(false);

// Call the void method

orderService.processOrder(invalidOrder);

// Verify the interaction with the mock repository

verify(mockRepository).reject(invalidOrder);

}

}

**Explanation:**

1. **Creating Mocks:**

OrderRepository mockRepository = mock(OrderRepository.class);

* + Creates a mock object of the OrderRepository interface.

1. **Stubbing Methods:**

when(validOrder.isValid()).thenReturn(true);

when(invalidOrder.isValid()).thenReturn(false);

* + Defines the behavior for the isValid method of the Order mock.

1. **Calling the Void Method:**

orderService.processOrder(validOrder);

orderService.processOrder(invalidOrder);

* + Calls the processOrder method with both a valid and an invalid order.

1. **Verifying Interactions:**

verify(mockRepository).save(validOrder);

verify(mockRepository).reject(invalidOrder);

* + Verifies that the save method was called for the valid order and the reject method was called for the invalid order.

By verifying the interactions, you can ensure that the void method behaves as expected, even though it doesn't return a value.

### How do you create custom annotations for JUnit — especially in JUnit 5 — to bundle or reuse common testing configurations, annotations, or behavior?

Suppose you often write this:

@ExtendWith(MockitoExtension.class)

@TestInstance(Lifecycle.PER\_CLASS)

@Tag("unit")

You can create a custom annotation to encapsulate all of that:

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@ExtendWith(MockitoExtension.class)

@TestInstance(TestInstance.Lifecycle.PER\_CLASS)

@Tag("unit")

public @interface UnitTest {

}

Then use it like this:

@UnitTest

class MyServiceTest {

// test code

}

Use Cases

Bundle @ExtendWith, @Tag, @TestInstance, etc.

Create annotations like @UnitTest, @IntegrationTest, @DatabaseTest

Improve readability and DRY up repeated annotations

Limitation (JUnit 4)

JUnit 4 has limited support for custom meta-annotations. You'd typically need to use custom Runners or Rules instead.

### How does @Mock works internally in Mockito: how does the framework handles mock creation under the hood?

**Internals of @Mock in Mockito (JUnit 5 / MockitoExtension)**

When you write:

@Mock

MyDependency dependency;

Here's roughly what happens inside Mockito:

### Step-by-step Mechanism

#### 1. **Annotation Processing**

Mockito uses reflection to scan fields annotated with @Mock.

This is done via the MockitoExtension (in JUnit 5) or MockitoJUnitRunner (in JUnit 4).

#### 2. **Field Interception**

When Mockito sees @Mock, it uses the MockAnnotationProcessor (an internal class).

It locates the field and calls:

Mockito.mock(field.getType());

Which creates a proxy.

#### 3. **Dynamic Proxy Generation**

For interfaces: Mockito uses the JDK Proxy API.

For classes: Mockito uses **Byte Buddy**, a bytecode manipulation library, to subclass the original class and override methods with stubs.

#### 4. **Injection (if using @InjectMocks)**

Fields annotated with @InjectMocks get their dependencies injected.

Mockito uses reflection to:

Find constructors or fields

Inject mock objects into them

This happens via the internal ConstructorInjection or FieldInjection strategies.

Key Internal Classes

MockAnnotationProcessor

MockUtil

MockMaker

InlineMockMaker or SubclassByteBuddyMockMaker (depending on config)

MockCreationSettings

Example Internals (Simplified)

Field field = ... // the @Mock field

Class<?> clazz = field.getType();

Object mock = Mockito.mock(clazz);

field.setAccessible(true);

field.set(testInstance, mock);

### Sources in Mockito Codebase

org.mockito.internal.configuration.InjectingAnnotationEngine

org.mockito.internal.creation.bytebuddy.SubclassByteBuddyMockMaker

org.mockito.internal.util.MockUtil

So in short: @Mock is syntactic sugar processed by Mockito’s extension mechanism, which uses reflection + bytecode generation (via Byte Buddy) to generate test doubles dynamically at runtime.