# JUnit Mockito Interview Questions

September 1, 2019 Saurabh Oza Developer Interview Questions, Automation Testing, JUnit, Mockito, Testing, Unit Testing 0

Table of Contents



# JUnit Mockito Testing Interview Questions


**JUnit Mockito Testing Interview Questions Only Fullstack**

# 1. What is Unit Testing?

Unit testing simply verifies that individual units of code (mostly functions) work independently as expected. Usually, you write the test cases yourself to cover the code you wrote. Unit tests verify that the component you wrote works fine when we ran it independently.
A unit test is a piece of code written by a developer that executes a specific functionality in the code to be tested and asserts a certain behavior or state.

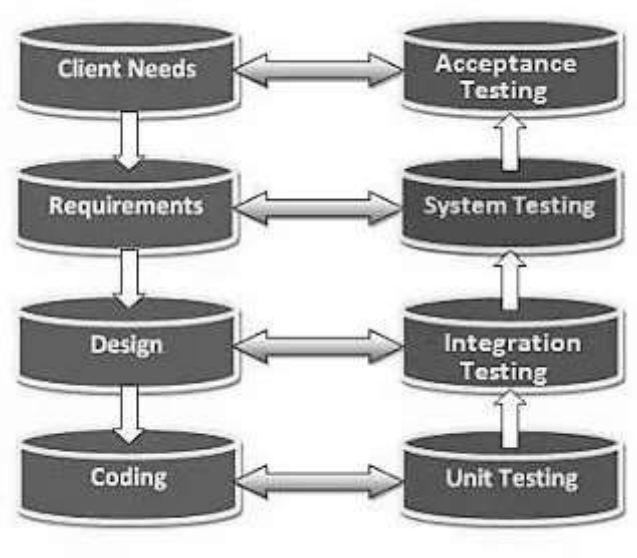
The percentage of code which is tested by unit tests is typically called test coverage.

A unit test targets a small unit of code, e.g., a method or a class. External dependencies should be removed from unit tests, e.g., by replacing the dependency with a test implementation or a (mock) object created by a test framework.

Unit tests are not suitable for testing complex user interface or component interaction. For this, you should develop integration tests.

### When is it performed?

Unit Testing is the first level of software testing and is performed prior to Integration Testing.




### Who performs it?

It is normally performed by software developers themselves or their peers.

### How to perform it?

Almost always, the process of unit-testing is built into an IDE (or through extensions) such that it executes the tests with every compile. A number of frameworks exist for assisting the creation of unit tests (and indeed mock objects), often named fooUnit (cf. jUnit, xUnit, nUnit). These frameworks provide a formalized way to create tests.

As a process, test-driven development (TDD) is often the motivation for unit testing (but unit testing does not require TDD) which supposes that the tests are a part of the spec definition, and therefore requires that they are written first, with code only written to “solve” these tests.

**Reference –** https://www.onlyfullstack.com/what-is-unit-testing/

## 2. What is the difference in between @Before, @After, @BeforeClass and @AfterClass?

### Annotations used in Junit

| Annotation | Description |
|---|---|
| **@Before** | Executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class). |
| **@After** | Executed after each test. It is used to clean up the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures. |
| **@BeforeClass** | Executed once, before the start of all tests. It is used to perform time intensive activities, for exa |

| | |
|---|---|
| | to connect to a database. Methods marked with this annotation need to be defined as `static` to work with JUnit. |
| **@AfterClass** | Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit. |

## Test Class

```
package com.onlyfullstack.unittesting.service;

import com.onlyfullstack.unittesting.bean.Employee;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import static org.junit.Assert.*;

/**
 * This class contains Unit Test cases of {@link EmployeeService}
 */
public class EmployeeServiceTest {

    private EmployeeService employeeService = new EmployeeService();

    private Employee employee = null;

    @BeforeClass
    public static void beforeClass() {
        System.out.println("Executing Before Class");
    }

    @Before
    public void before() {
        System.out.println("Executing Before");
        employee = new Employee();
        employee.setSalary(1000000.0);
    }

    @AfterClass
    public static void afterClass() {
        System.out.println("Executing After Class");
    }

    @After
    public void after() {
        System.out.println("Executing After");
    }

    @Test
    public void isValidEmployee_withNullEmployee() {
        System.out.println("Entered in isValidEmployee_withNullEmployee");
        assertEquals(new Double(0.0), employeeService.calculateTax(null));
        System.out.println("Exited from isValidEmployee_withNullEmployee");
```

```
    }

    @Test
    public void isValidEmployee_withNegativeSalary() {
        System.out.println("Entered in isValidEmployee_withNegativeSalary");
        employee.setSalary(-2.0);
        assertEquals(new Double(0.0), employeeService.calculateTax(null));
        System.out.println("Exited from isValidEmployee_withNegativeSalary");
    }
}
```

Let's run the test cases with Eclipse run with Junit test option.

**Output**

Executing Before Class
Executing Before
Entered in isValidEmployee_withNegativeSalary
Exited from isValidEmployee_withNegativeSalary
Executing After
Executing Before
Entered in isValidEmployee_withNullEmployee
Exited from isValidEmployee_withNullEmployee
Executing After
Executing After Class

**@Before and @After are called for every test case.**

**@BeforeClass and @AfterClass are called once**

**https://www.onlyfullstack.com/part-3-annotations-used-in-junit/**

# 3. What are Assert Methods in JUnit?

This class provides a set of assertion methods, useful for writing tests. Only failed assertions are recorded. We will be writing the test cases for below class:

**https://github.com/onlyfullstack/unit-testing-and-integration-testing-with-spring-boot/blob/master/unit-testing/src/main/java/com/onlyfullstack/unittesting/service/EmployeeService.java**

## assertEquals()

The assertEquals() method compares two objects for equality, using their equals() method.

```
@Test
public void assertEquals_example() {
    Employee employeeNew = new Employee();
    employee.setSalary(1000000.0);
    assertEquals("EMPLOYEE OBJECT", employee, employeeNew);
}
```

If the two objects are equal according to there implementation of their equals() method, the assertEquals() method will return normally. Otherwise, the assertEquals() method will throw an exception, and the test will stop there.

## assertTrue() + assertFalse()

The assertTrue() and assertFalse() methods tests a single variable to see if its value is either true or false. Here is a simple example:

```
@Test
public void assertTrue_assertFalse_example() {

    assertTrue("VALID EMPLOYEE OBJECT",
employeeService.isValidEmployee(employee));
    assertFalse("INVALID EMPLOYEE OBJECT",
employeeService.isValidEmployee(null));
}
```

If the isValidEmployee() method returns true, the assertTrue() method will return normally. Otherwise, an exception will be thrown, and the test will stop there.
If the isValidEmployee() method returns false, the assertFalse() method will return normally. Otherwise, an exception will be thrown, and the test will stop there.

## assertNull() + assertNotNull()

The assertNull() and assertNotNull() methods test a single variable to see if it is null or not null. Here is an example:

```
@Test
public void assertNull_assertNotNull_example() {

    assertNotNull(employeeService.getEmployeeFromId(1)); // in EmployeeService we
have a map with

// single entry of key as 1 so here we will get employee object
    assertNull(employeeService.getEmployeeFromId(2)); // We will get null as
response
}
```

If the employeeService.getEmployeeFromId(2) returns null, the assertNull() method will return normally. If a non-null value is returned, the assertNull() method will throw an exception, and the test will be aborted here.
The assertNotNull() method works oppositely of the assertNull() method, throwing an exception if a null value is passed to it, and returning normally if a non-null value is passed to it.

## assertSame() and assertNotSame()

The assertSame() and assertNotSame() methods tests if two object references point to the same object or not. It is not enough that the two objects pointed to are equals according to their equals() methods. It must be exactly the same object pointed to.
Here is a simple example:

```
@Test
public void assertSame_assertNoSame_example() {

    assertSame(employeeService.getEmployeeFromId(1),
employeeService.getEmployeeFromId(1));
    assertNotSame(employee, employeeService.getEmployeeFromId(1)); // We will get
null as response
}
```

If the two references point to the same object, the assertSame() method will return normally. Otherwise, an exception will be thrown and the test will stop here.
The assertNotSame() method works oppositely of the assertSame() method. If the two objects do not point to the same object, the assertNotSame() method will return normally. Otherwise, an exception is thrown and the test stops here.

### assertThat()

The assertThat() method compares an object to an org.hamcrest.Matcher to see if the given object matches whatever the Matcher requires it to match.
Please find below example where on the
1st assertThat – its taking the  employeeService.getEmployeeFromId(1) and comparing it with employeeService.getEmployeeFromId(1) with the help of is Matcher.
2nd assertThat is taking employeeService.getEmployeeFromId(1) and checking if the object is not null with is(CoreMatchers.notNullValue()) Matcher.
The matcher is a big topic and we will cover them separately.

```
@Test
public void assertThat_example() {

    assertThat(employeeService.getEmployeeFromId(1),
is(employeeService.getEmployeeFromId(1)));
    assertThat(employeeService.getEmployeeFromId(1),
is(CoreMatchers.notNullValue()));

}
```

## 4. What are Hamcrest Matcher?

Hamcrest is a framework for writing matcher objects allowing ‘match’ rules to be defined declaratively. There are a number of situations where matchers are invaluable, such as

validation, or data filtering, but it is in the area of writing flexible tests that matchers are most commonly used. This tutorial shows you how to use Hamcrest for unit testing.

To use Hamcrest matchers in JUnit you use the assertThat statement followed by one or several matchers.

Hamcrest is typically viewed as a third generation matcher framework.

**The first generation** used assert(logical statement) but such tests were not easily readable.
**The second generation** introduced special methods for assertions, e.g., assertEquals(). This approach leads to lots of assert methods.
Hamcrest uses assertThat method with a matcher expression to determine if the test was successful.
See **Wiki on Hamcrest** for more details.
**Reference – https://www.onlyfullstack.com/complete-guide-for-hamcrest-matchers/**

## 5. How do you assert that a certain exception is thrown in JUnit 4 tests?

Let's write some business logic which will throw an exception.

```
package com.onlyfullstack.unittesting.service;

import org.apache.commons.lang3.StringUtils;

/**
 * This class contains the business logic to throw an exception
 */
public final class ExceptionHandling {

    public String convertIntoUpperCase(String input) {
        if (StringUtils.isEmpty(input)) {
            throw new IllegalArgumentException("Empty value is passed.");
        }
        return input.toUpperCase();
    }
}
```

The convertIntoUpperCase() method will throw an IllegalArgumentException if an empty string is passed to the method.

**There are 3 ways to assert a certain exception in Junit. Let's write the unit test cases for it.**

### 1. try-catch idiom

This idiom is one of the most popular ones because it was used already in JUnit 3. This approach is a common pattern. The test will fail when no exception is thrown and the exception itself is verified in a catch clause.

```
@Test
public void convertIntoUpperCase_withInvalidInput_tryCatchIdiom() {
    try {
        exceptionHandling.convertIntoUpperCase("");
        fail("It should throw IllegalArgumentException");
    } catch (IllegalArgumentException e) {
        Assertions.assertThat(e)
                .isInstanceOf(IllegalArgumentException.class)
                .hasMessage("Empty value is passed.");
    }
}
```

## 2. @Test expected annotation

In this approach, we specify the expected exception in @Test as below
**@Test(expected = IllegalArgumentException.class)**

When the exception wasn't thrown you will get the following message:
java.lang.AssertionError: Expected exception: java.lang.IllegalArgumentException

With this approach, you need to be careful though. Sometimes it is tempting to expect general Exception, RuntimeException or even a Throwable. And this is considered as a bad practice because your code may throw an exception in other places than you actually expected and your test will still pass!

One of the drawback of this approach is you can't assert for the exception message.

```
@Test(expected = IllegalArgumentException.class)
public void convertIntoUpperCase_withInvalidInput_testExpected() {
    exceptionHandling.convertIntoUpperCase("");
}
```

## 3. Junit @Rule

The same example can be created using ExceptedException rule. The rule must be a public field marked with @Rule annotation.

```
@Test
public void convertIntoUpperCase_withInvalidInput_ExpectedExceptionRule() {
    exception.expect(IllegalArgumentException.class);
    exception.expectMessage("Empty value is passed.");
    exceptionHandling.convertIntoUpperCase("");
}
```

I find the above code more readable hence I prefer to use this approach.

When the exception isn't thrown you will get the following message: java.lang.AssertionError: Expected test to throw (an instance of java.lang.IllegalArgumentException and exception with the message "Empty value is passed."). Pretty nice.

But not all exceptions I check with the above approach. Sometimes I need to check only the type of the exception thrown and then I use @Test annotation.

## 6. What is a Mock Object?

In **object-oriented programming**, mock objects are simulated objects that mimic the behaviour of real objects in controlled ways. A programmer typically creates a mock object to test the behaviour of some other object, in much the same way that a car designer uses a **crash test dummy** to **simulate** the dynamic behaviour of a human in vehicle impacts.

## 7. When should I mock?

A unit test should test a single code path through a single method. When the execution of a method passes outside of that method, into another object, and back again, you have a dependency.

When you test that code path with the actual dependency, you are not unit testing; you are integration testing. While that's good and necessary, it isn't unit testing.

## 8. How to enable Mockito Annotations?

we need to enable the Mockito to use its annotation and functionality. There are 2 ways to enable the Mockito framework for our JUnit class.

### 1. MockitoJUnitRunner

We'll need to annotate the JUnit test with a runner – MockitoJUnitRunner as in the following example

```
@RunWith(MockitoJUnitRunner.class)
public class TestClass {
    ...
}
```

### 2. MockitoAnnotations.initMocks()

Alternatively, we can enable these annotations programmatically as well, by invoking MockitoAnnotations.initMocks() as in the following example:

```
@Before
public void init() {
     MockitoAnnotations.initMocks(this);
}
```

**Reference – https://www.onlyfullstack.com/what-is-mock-object-what-is-mockito/**

## 9. How to mock methods with Mockito?1. when/then

```
when(repository.saveCusomer(any())).thenReturn(true);
```

Here,

**when**: when is a static method of the Mockito class which takes an object and its method which needs to be mocked

**any()**: It's a matcher which specifies that the method may get a call with any parameter.

**thenRetun:** What value do you want to return when this method is called with the specified parameters.

```
package com.onlyfullstack.unittesting.service;

import com.onlyfullstack.unittesting.bean.Customer;
import com.onlyfullstack.unittesting.repository.CustomerRepository;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;
import static org.mockito.Mockito.any;
```

```
import static org.mockito.Mockito.eq;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

/**
 * This class contains usage of Mockito
 */
@RunWith(MockitoJUnitRunner.class)
public class CustomerServiceTest {

    @Mock
    CustomerRepository repository;

    @InjectMocks
    CustomerService customerService;

    @Test
    public void saveCustomer_withValidCustomer() {
        Customer customer = new Customer(6, "QQQ", "Mumbai");
        when(repository.saveCusomer(any())).thenReturn(true);
        Boolean save = customerService.saveCustomer(customer);
        assertThat(true, is(save));
    }

}
```

We have seen how to use when/thenRetun pattern, lets explore other patterns:

### 2. when/thenThrow

This method will throw the specified exception when the mocked method is called.

```
@Test(expected = IllegalStateException.class)
public void saveCustomer_withValidCustomer_when_thenThrow() {
    Customer customer = new Customer(6, "QQQ", "Mumbai");
    when(repository.saveCusomer(any())).thenThrow(new IllegalStateException());
    customerService.saveCustomer(customer);
}
```

### 3. When/thenAnswer

**Reference –** https://www.onlyfullstack.com/how-to-mock-void-methods-with-mockito/

## 10. How to mock void methods with Mockito?

Let's add a method which internally calls the void method of another component.

```
public Customer updateCustomer(Customer customer) {
    if (customer == null || customer.getId() < 0) {
        throw new IllegalArgumentException("Invalid Customer details passed.");
    }
    repository.updateCustomer(customer);
    return customer;
}
```

In the above example, updateCustomer is calling the void method of the repository object. Let's see how to mock this method to write a unit test for updateCustomer() of EmployeeService.

### Three ways to mock the void method:

### 1. doNothing/when

If you don't want to check for params and just skip the actual execution then we can use doNothing which will not do anything when this method is called during the test case.

```
@Test
public void updateCustomer_doNothing_when() {
    Customer customer = new Customer(6, "QQQ", "Mumbai");
    doNothing().when(repository).updateCustomer(any(Customer.class));
    customerService.updateCustomer(customer);
}
```

### 2. doAnswer/when

We have a updateCustomer method in CustomerService class which calls a void method updateCustomer of CustomerRepository. Now we need to mock this void method and make sure the params passed to this method is as per the expectation.

```
@Test
public void updateCustomer_doAnswer_when() {
    Customer customer = new Customer(6, "QQQ", "Mumbai");
    doAnswer((arguments) -> {
        System.out.println("Inside doAnswer block");
        assertEquals(customer, arguments.getArgument(0));
        return null;
    }).when(repository).updateCustomer(any(Customer.class));
    customerService.updateCustomer(customer);
}
```

### 3. doThrow/when

When we want to throw an exception from the void method or normal method then we can use doThrow/when pattern.

```
@Test(expected = Exception.class)
public void updateCustomer_doNothing_when() {
    Customer customer = new Customer(6, "QQQ", "Mumbai");
    doThrow(new Exception("Database connection issue"))

            .when(repository).updateCustomer(any(Customer.class));
    customerService.updateCustomer(customer);
}
```

## 11. How to verify the mocks?

Mockito Verify methods are used to check that certain behaviour happened. We can use Mockito verify methods at the end of the testing method code to make sure that specified methods are called.

**Let's look at some of the Mockito verify method variations and examples.**

### Simple verify method:

```
@Test
public void saveCustomer_withValidCustomer_when_thenReturn() {
    Customer customer = new Customer(6, "QQQ", "Mumbai");
    when(repository.saveCusomer(any())).thenReturn(true);
    Boolean save = customerService.saveCustomer(customer);
    assertThat(true, is(save));
    verify(repository, times(1)).saveCusomer(eq(customer));
}
```

### Variations in verify method

Below are the variations of verify method which we can use when we want any type of parameters or a specific type of parameters or exact parameter.

```
verify(repository).saveCusomer(any());
verify(repository).saveCusomer(any(Customer.class));
verify(repository).saveCusomer(ArgumentMatchers.any(Customer.class));
verify(repository).saveCusomer(eq(customer));
```

### Verify with the number of times

Mockito verify() method is overloaded, the second one is **verify(T mock, VerificationMode mode)**. We can use it to verify for the invocation count.

```
verify(repository, times(1)). saveCusomer (); //same as normal verify method

verify(repository, atLeastOnce()).saveCusomer (); // must be called at least once

verify(repository, atMost(2)). saveCusomer(); // must be called at most 2 times

verify(repository, atLeast(1)). saveCusomer(); // must be called at least once

verify(repository, never()).getCusomer(); // must never be called
```

## 12. How to check the Order of Invocation with verify?

We can use InOrder to verify the order of invocation. We can skip any method to verify, but the methods being verified must be invoked in the same order.

```
// A. Single mock whose methods must be invoked in a particular order
List singleMock = mock(List.class);

//using a single mock
singleMock.add("was added first");
singleMock.add("was added second");

//create an inOrder verifier for a single mock
InOrder inOrder = inOrder(singleMock);

//following will make sure that add is first called with "was added first,
```

```
//then with "was added second"
inOrder.verify(singleMock).add("was added first");
inOrder.verify(singleMock).add("was added second");

// B. Multiple mocks that must be used in a particular order
List firstMock = mock(List.class);
List secondMock = mock(List.class);
```

```
//using mocks
firstMock.add("was called first");
secondMock.add("was called second");

//create inOrder object passing any mocks that need to be verified in order
InOrder inOrder = inOrder(firstMock, secondMock);

//following will make sure that firstMock was called before secondMock
inOrder.verify(firstMock).add("was called first");
inOrder.verify(secondMock).add("was called second");

// Oh, and A + B can be mixed together at will
```

# 13. What is difference between @Mock and @Spy?

### 1. Object declaration

**Mock –** We don't need to instantiate the mock List as the @Mock will create and instantiate the list for us.

**Spy-** We need to instantiate the list object as the @Spy will use the real object's method if we don't mock them.

```
/*
 We dont need to instantiate the mock List as the @Mock will create and
instantiate the list for us
 */
@Mock
private List<String> mockedList;

/*
We need to instantiate the list object as the @Spy will use the real objects
method if we dont mock them
 */
@Spy
private List<String> spyList = new ArrayList();
```

### 2. When the methods are not mocked

**Mock –** If we don't mock the methods of @Mock object and try to call them then it will not do anything.

**Spy –** add method is not mocked so the spyList will execute the default behaviour of the add method and it will add a new String into the list.

```
@RunWith(MockitoJUnitRunner.class)
public final class MockVsSpy {

    /*
     We dont need to instantiate the mock List as the @Mock will create and
instantiate the list for us
     */
    @Mock
    private List<String> mockedList;

    /*
    We need to instantiate the list object as the @Spy will use the real objects
method if we dont mock them
     */
    @Spy
    private List<String> spyList = new ArrayList();

    @Test
    public void testMockList_checkDefaultBehaviour_whenMethodIsNotMocked() {
        /*If we dont mock the methods of @Mock object and try to call them
        then it will not do anything.*/

        mockedList.add("test"); // add the String into list which will not do
anything
        assertNull(mockedList.get(0)); // As the String was not added into the
list it will return null value
    }

    @Test
    public void testSpyList_checkDefaultBehaviour_whenMethodIsNotMocked() {
        /* add method is not mocked so the spyList will execute
        * the default behaviour of the add method and it will add a new String
into list*/
        spyList.add("test");
        assertEquals("test", spyList.get(0));
    }
}
```

## 3. When the method is mocked

**Mock –** Mock will execute as per the above example and it will not add or get the element from the mockedList.

**Spy –** add method is not mocked so the spyList will execute the default behaviour of the add method and it will add a new String into the list.

```
@Test
public void testMockList_whenMethodIsMocked() {
    /*If we dont mock the methods of @Mock object and try to call them
    then it will not do anything.*/
    when(mockedList.size()).thenReturn(10);
```

```
    mockedList.add("One");
    assertNull(mockedList.get(0)); // Again the execution of add and get methods
will not have any impact on mocked object

    assertEquals(10, mockedList.size()); // As the String was not added into the
list it will return null value
}

@Test
public void testSpyList_whenMethodIsMocked() {
    /* add method is not mocked so the spyList will execute
     * the default behaviour of the add method and it will add a new String into
list*/
    when(spyList.size()).thenReturn(10);
    spyList.add("One");
    assertEquals("One", spyList.get(0));

    assertEquals(10, spyList.size()); // size method will return 10 as we have
mocked its implementation
}
```

## Related Interview Questions

### Java 8 Lambda Iterview Questions

### Java 8 Programming Interview Questions

### Java Interview Questions For Senior Full Stack Developer

INTERVIEW QUESTIONS JUNIT MOCKITO UNIT TESTING


« PREVIOUS
Java Interview Questions for Senior Developers

NEXT »
Angular Interview Questions – Part 1