InterviewBit

# JUnit Interview Questions

To view the live version of the page, [click here.](#)

# Contents

## JUnit Interview Questions for Freshers

## JUnit Interview Questions for Experienced

(.....Continued)

## JUnit Mockito Interview Questions

## JUnit Mockito Interview Questions (.....Continued)

# Let's get Started

## Introduction

Testing is an important part of the software development process as it helps in resolving bugs before the software goes to production. Unit testing is a type of testing that tests individual entities at a time. JUnit is an open-source, Java-based and one of the most popular unit testing frameworks and is supported by most popular IDEs like Eclipse, IntelliJ etc. It is used to write and run unit tests by providing annotations for identifying the test methods and expected results. This framework allows writing codes faster, increases the quality of the source code and helps in identifying bugs in the source code at a very early stage.

In this article, we will see the most commonly asked JUnit interview questions for freshers and experienced professionals.

## JUnit Interview Questions for Freshers

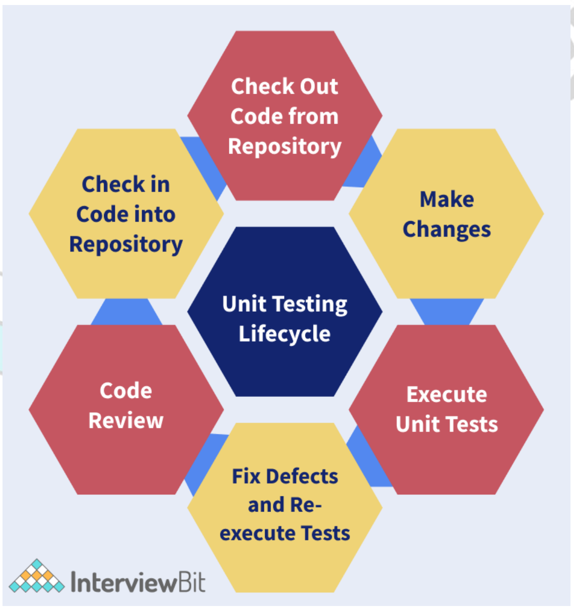
### 1. What is JUnit?

JUnit is an open-source, Java-based unit testing framework that plays a crucial role in achieving the culture of TDD (Test Driven Development). The TDD culture lays strong emphasis on setting up the test data for testing a logic that would be implemented once the testing is successful. JUnit helps to increase the software stability as it helps in identifying the bug in the code logic at an early stage without requiring the software to go to production. This helps in reducing the time required to debug any issues later on.

### 2. What is Unit Testing?

Unit testing is a software testing strategy that tests single entities like methods or classes at a time. This helps to ensure that the product quality is met as per the business requirements. It also helps in reducing the technical debt of the application by helping developers discover issues in the code logic due to any changes. It also gives insights into how the code logic implemented could impact future changes.

The lifecycle of the unit testing process is as shown in the image below:


Check Out Code from Repository
Make Changes
Execute Unit Tests
Fix Defects and Re-execute Tests
Code Review
Check in Code into Repository
Unit Testing Lifecycle
InterviewBit

## 3. Why do we use JUnit? Who uses JUnit more - Developers or Testers?

JUnit is used more often by developers for implementing unit test cases for the functionalities they have developed. However, these days testers also use this framework for performing unit testing of the applications.

JUnit is used due to the following reasons:

- Helps in automating test cases.
- Helps in reducing defects in the code logic whenever the logic changes.
- Helps in reducing the cost of testing as the bugs are identified, captured and addressed at early phases in the software development.
- Helps to identify the gaps in coding and gives a chance to refactor the code.

## 4. What are the features of JUnit?

Following are the **features of JUnit**:

- JUnit is an open-source framework.
- Supports automated testing of test suites.
- Provides annotations for identifying the test methods.
- Provides assertions to test expected results or exceptions of the methods under test.
- Provides a platform for running test cases automatically and checking their results and giving feedback.

## 5. Is it mandatory to write test cases for every logic?

No, it is not mandatory. However, test cases can be written for the logic which can be reasonably broken and tested independently.

## 6. What are some of the important annotations provided by JUnit?

Some of the annotations provided by JUnit are as follows:

- `@Test` : This annotation over a public method of void return type can be run as a test case. This is a replacement of the `org.junit.TestCase` annotation.
- `@Before` : This is used when we want to execute the preconditions or any initialisation based statements before running every test case.
- `@BeforeClass` : This is used when we want to execute statements before all test cases. The statements may include test connections, common setup initialisation etc.
- `@After` : This is used when we want to execute statements after each test case. The statements can be resetting the variables, deleting extra memory used etc.
- `@AfterClass` : This is used when certain statements are required to be executed after all the test cases of the class are run. Releasing resource connections post-execution of test cases is one such example.
- `@Ignores` : This is used when some statements are required to be ignored during the execution of test cases.
- `@Test(timeout=x)` : This is used when some timeout during the execution of test cases is to be set. The value of x is an integer that represents the time within which the tests have to be completed.
- `@Test(expected=NullPointerException.class)` : This is used when some exception thrown by the target method needs to be asserted.

## 7. How will you write a simple JUnit test case?

Let us understand how to write a simple JUnit Test Case using an example. Consider a Calculator class that has a method that adds 2 numbers:

```java
public class Calculator {
    public int add(int num1, int num2) {
        return num1 + num2;
    }
}
```

Let us write a test case in JUnit 5 to this method under a class named CalculatorTest.

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.Test;
public class CalculatorTest {
   Calculator calcObject;
   @BeforeEach
   void setUp() {
       calcObject = new Calculator();
   }
   @Test
   @DisplayName("Add 2 numbers")
   void addTest() {
       assertEquals(15, calcObject.add(10, 5));
   }
   @RepeatedTest(5)
   @DisplayName("Adding a number with zero to return the same number")
   void testAddWithZero() {
       assertEquals(15, calcObject.add(0, 15));
   }
}
```

From the above example, we can see that:

- @BeforeEach annotated method runs before each test case. In JUnit 4, this annotation was `@Before` .
- @Test annotation indicates that the method is a test method.
- @DisplayName is used for defining the test name displayed to the user.
- assertEquals() method is used for validating whether the expected and actual values are equal.
- @RepeatedTest annotation indicates that the test method will be run 5 times.

## 8. What will happen if the return type of the JUnit method is String?

The test case execution will fail because the JUnit methods are designed to return void.

## 9. What is the importance of @Test annotation?

@Test annotation is used for marking the method as a test method.

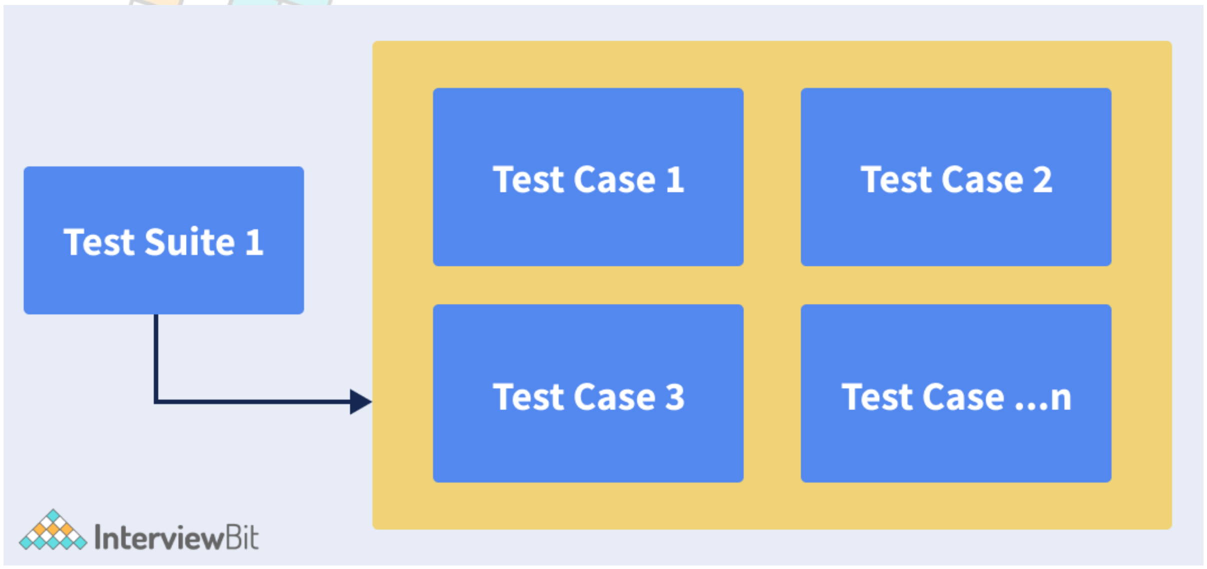
## 10. What is a JUnit fixture?

Fixture represents a fixed state of object sets used as a baseline for running test methods. This is to ensure there is a fixed and well-known environment where the results of the test methods are repeatable when run multiple times. The fixture has the following 2 methods:

- `setUp()` This runs before every test case is run.
- `tearDown()` This method is run after every test is run.

## 11. What is a test suite?

A test suite is a bundle of multiple unit test cases which can be run together. The following image represents how to test suite looks like:



We can use @RunWith and @Suite annotations over the test class for running the test suite.

## 12. What is mocking and stubbing?

Mocking is a phenomenon where an object mimics a real object. Whereas, stubbing are the codes responsible for taking place of another component. Mockito, EasyMock are some of the mocking frameworks in Java.

## 13. What are the JUnit Assert Methods?

Assert methods are utility methods that support assert conditions in test cases. They belong to the Assert class in JUnit 4 and the Assertions class in JUnit 5. It is recommended to import the assert methods statically to the test class for avoiding using the class as a prefix to the method. Let us consider an example of the Multiplier class:

```
public class Multiplier{
   public int multiply(int num1, int num2){
       return num1 * num2;
   }
}
```

Following are some of the assert methods:

- `assertEquals()` : This method compares 2 objects for equality by making use of the equals() method of the object. This is shown in the test case that multiplies 2 methods and checks for the expected and actual value below.

```
@Test
@DisplayName("Multiplying 2 numbers")
public void multiplyTest() {
    assertEquals(50, multiplier.multiply(10, 5));
}
```

When two objects are found to be equal based on the equals() method implementation of the object, then assertEquals() returns normally. Else, an exception will be thrown and the test will stop its execution.

- `assertTrue()` : This method tests whether the value of a variable is true.

```
@Test
public void checkNumTest() {
    int num1 = 20;
    assertTrue("Number is not equal to 0", num1!=0);
}
```

If the assertion fails, then an exception will be thrown and the test execution will be stopped.

- `assertFalse()` : This method tests whether the value of a variable is false.

```
@Test
public void checkNumTest() {
    int num1 = -20;
    assertFalse("Number is not greater than 0",num1>0);
}
```

If the assertion fails, then an exception will be thrown and the test execution will be stopped.

- `assertNull()` : This method tests if a variable is null. If null, it returns normally else an exception will be thrown and the test stops execution.

```
@Test
public void checkNullTest() {
    int num1 = null;
    assertNull("Number is null",num1);
}
```

- `assertNotNull()` : This method tests whether a variable is not null. If it is not null, then the test returns normally else an exception will be thrown and the test stops its execution.

```
@Test
public void checkNotNullTest() {
    int num1 = null;
    assertNotNull("Number is null",num1);
}
```

- `assertSame()` : This method checks if two references of the object are pointing to the same object.

```
   @Test
   public void checkAssertSameTest() {
        Object num1 = new Object();
        Object num2 = new Object();
        assertSame(num1, num2);
   }
```

If the object references are pointing to the same object, the test runs normally else, it will throw an exception and the test execution is aborted.

- `assertNotSame()` : This method checks if two references of an object are not pointing to the same object.

```
@Test
public void checkAssertSameTest() {
     Object num1 = new Object();
     Object num2 = new Object();
     assertSame(num1, num2);
}
```

If the object references are not pointing to the same object, the test runs normally else, it will throw an exception and the test execution is aborted.

- `assertThat()` : This method compares the object to org.hamcrest.Matcher for checking if it matches whatever the Matcher class requires for the object to match. Consider the following code.

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.hasItems;
import org.junit.jupiter.api.Test;

public class MultiplierTest {
    @Test
    public void assertThatTest() {
        assertThat(
          Arrays.asList(1,2,3,4),
          hasItems(2,3));
    }
}
```

In this code, we will be asserting that the list has some items specified in the hamcrest's hasItems() method. If the `assertThat` is successful, i.e if the list indeed has items specified, then the test runs normally. Else, the test throws an exception and the test stops executing.

## 14. What is the importance of @RunWith annotation?

@RunWith annotation lets us run the JUnit tests with other custom JUnit Runners like SpringJUnit4ClassRunner, MockitoJUnitRunner etc. We can also do parameterized testing in JUnit by making use of @RunWith(Parameterized.class).

### 15. How does JUnit help in achieving tests isolation?

For calling test methods, JUnit creates separate individual instances of the test class. For example, if the test class contains 10 tests, then JUnit creates 10 instances of the class to execute the test methods. In this way, every test is isolated from the other.

## JUnit Interview Questions for Experienced

### 16. What are the best practices for writing Unit Test Cases?

A standard unit test case comprises a known input and an expected output. These two things need to be known before we run the test case. A known input is tested for a precondition and an expected output is tested by postcondition. Following are the best practices for writing unit test cases:

- For every method, we need to have at least two unit test cases - a positive test case and a negative test case.
- If there are sub-requirements for a requirement, then those sub-requirements should have their own positive and negative test cases.
- Each test case should be independent of other test cases. If we make a chain of unit test cases, then it would not be possible for finding the root cause of the test case failures.
- Mock all the external services that are used by the modules under test. This is necessary because we do not want to unnecessarily debug our modules under test due to the failures of the external systems.
- Configuration settings need not be tested as they won't be part of any unit code. Even if we want to inspect the configuration, then test whether the loading code is working or not.
- The unit test cases should be named consistently and clearly. The names of the test cases should be dependent on the operations that the test case would test.

### 17. What are the differences between JUnit 4 and JUnit 5?

| Category | JUnit 4 | JUnit 5 |
|---|---|---|
| Annotation for executing method before all test methods run in the current class | @BeforeClass | @BeforeAll |
| Annotation for executing method after all test methods run in the current class | @AfterClass | @AfterAll |
| Annotation for executing method before each test case | @Before | @BeforeEach |
| Annotation for executing method after each test method | @After | @AfterEach |

## 18. What are the differences between JUnit and TestNG?

| JUnit | TestNG |
|---|---|
| Open-source unit testing framework for writing test cases. | TestNG is similar to JUnit but with extended functionalities. |
| JUnit does not support advanced annotations. | This supports advanced and special annotations too. |
| JUnit does not support parallel testing. | TestNG supports multiple threads to run parallelly. |
| Group test is not supported. | TestNG is supported in TestNG. |
| The naming convention for annotations like Before, After, Expected are confusing. | The naming convention for understanding annotations like BeforeMethod, AfterMethod, ExpectedException is easy to understand based on their functionality. |
| Cannot rerun failed cases. | Can rerun failed tests. |

## 19. How to ignore tests in JUnit?

We need to ignore test cases when we have certain functionalities that are not certain or they are under development. To wait for the completion of the code, we can avoid running these test cases. In JUnit 4, we can achieve this by using @Ignore annotation over the test methods. In JUnit 5, we can do it using @Disabled annotation over the test methods.

## 20. What is the purpose of @Before and @After annotations in JUnit 4?

These are the annotations present in JUnit 4. Methods annotated with @Before will be called and run before the execution of each test case. Methods annotated with @After will be called and executed after the execution of each test case. If we have 5 test cases in a JUnit class, then the methods annotated with these two would be run 5 times. In JUnit 5, the annotations @Before and @After are renamed to @BeforeEach and @AfterEach for making it more readable.

## 21. How can we test protected methods?

For testing protected methods, the test class should be declared in the same package as that of the target class.

## 22. Why can't we use System.out.println() for testing and debugging?

If we use `System.out.println()` for debugging, then we would have to do a manual scan of the outputs every time program is executed to ensure that the output printed is the expected output. Also, in the longer run, writing JUnit tests will be easier and will take lesser time. Writing test cases will be like an initial investment and then, later on, we can later automatically test them on the classes.

## 23. How can we run JUnit from the command window?

The first step is to set the CLASSPATH with the library path. Then invoke the JUnit runner by running the command:

```
java org.junit.runner.JUnitCore
```

To run the test class via commandline, we would first have to compile the class. This can be done by running:

```
javac -d target -cp target:junit-platform-console-standalone-1.7.2.jar src/test/java/cc
```

Then we can run the compiled test class using the console launcher of JUnit by running the following command:

```
java -jar junit-platform-console-standalone-1.7.2.jar --class-path target --select-clas
```

## 24. How do you assert exceptions thrown in JUnit 4 tests?

Consider an example to see how to write test cases for asserting exceptions.

```
import org.apache.commons.lang3.StringUtils;
public final class ExceptionDemo {
   public String convertToUpperCase(String input) {
       if (StringUtils.isEmpty(input)) {
           throw new IllegalArgumentException("Input empty, cannot convert to upper cas
       }
       return input.toUpperCase();
   }
}
```

The method `convertToUpperCase()` defined in the `ExceptionDemo` class would throw an `IllegalArgumentException` if an empty string is passed as an input to the method.
Let us write unit test cases for asserting the exception thrown in the code. In JUnit, there are 3 different ways of achieving this and they are as follows:

**try-catch idiom:** We can use the try-catch block for asserting exception as shown below:

```
@Test
public void convertToUpperCaseWithTryCatchTest{
   try {
       exceptionDemo.convertToUpperCase("");
       fail("It should throw IllegalArgumentException");
   } catch (IllegalArgumentException e) {
       Assertions.assertThat(e)
               .isInstanceOf(IllegalArgumentException.class)
               .hasMessage("Input empty, cannot convert to upper case!!");
   }
}
```

### @Test expected annotation:

Here we give the expected exception as value to the expected parameter of the @Test annotation as: `@Test(expected = IllegalArgumentException.class)`

When no exception is thrown by the target method under test, then we will get the below message:

```
java.lang.AssertionError: Expected exception: java.lang.IllegalArgumentException
```

Care has to be taken here while defining the expected Exception. We should not expect general Exception, RuntimeException or even a Throwable type of exceptions as that is a bad practice. If we do so, then there are chances that the code can throw an exception in other places and the test case can pass.

Here we cannot assert the message in the exception. The code snippet of the test method would be:

```
@Test(expected = IllegalArgumentException.class)
public void convertToUpperCaseExpectedTest() {
    exceptionDemo.convertToUpperCase("");
}
```

### Junit @Rule:

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

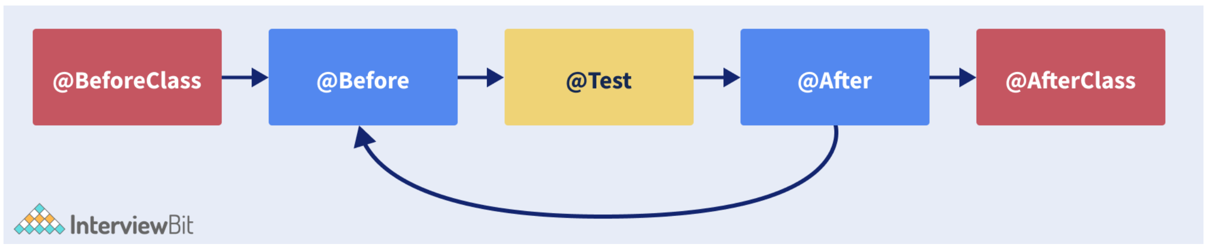
```java
@Rule
public ExpectedException exceptionRule = ExpectedException.none();

@Test
public void convertToUpperCaseRuleTest() {
    exceptionRule.expect(IllegalArgumentException.class);
    exceptionRule.expectMessage("Input empty, cannot convert to upper case!!");
    exceptionDemo.convertToUpperCase("");
}
```

In this approach, when the exception is not thrown, we will get:

```
java.lang.AssertionError: Expected test to throw (an instance of java.lang.IllegalArgum
```

## 25. What are the differences between @Before, @After, @BeforeClass and @AfterClass?

The order of execution of these annotations is as shown below:



| @Before | @After | @BeforeClass | @AfterClass |
|---|---|---|---|
| Methods annotated with @Before are executed before each test case. | Methods annotated with @After are executed after each test case. | Methods annotated with @BeforeClass are executed once before the start of all the test cases. | Methods annotated with @AfterClass are executed once all the tests have completed their execution. |
| This is used for preparing the test environment like to read inputs, initialise the variables etc. | This is used for cleaning up the test environment like deleting the temporary data, restoring the value of variables to default. This helps in saving memory. | This is used for performing time-intensive set-up activities like connecting to any resource that is common across all tests. Methods annotated with this API needs to be defined as static methods for working with JUnit. | This is used for performing time-intensive cleanup activities like disconnecting from a resource that is common across all tests. Methods annotated with this annotation has to be defined as static methods for it to work with JUnit. |

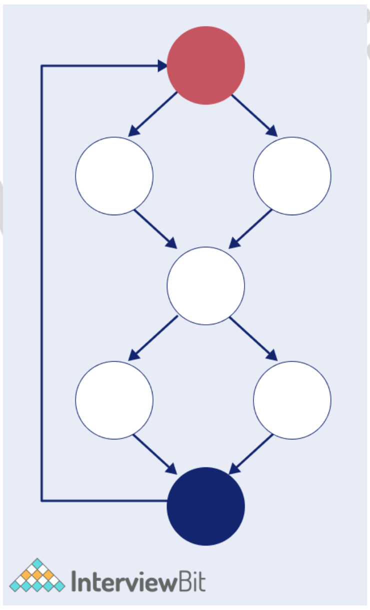
## 26. What are Hamcrest Matchers?

Hamcrest is a JUnit framework used to write matcher objects that provides ‘match’ rules definition in a declarative manner. They are used in UI validation, data filtering and for writing flexible unit tests. Hamcrest matches are used in the assertThat() method in JUnit. We can use one or more matchers within this assert method. Some of the matcher methods are as follows:

| Matcher | Description | Example |
|---|---|---|
| **allOf()** | This calculates the logical conjunction of multiple matchers and the object under consideration should match all of the matcher given. | `assertThat("InterviewBit",allOf(sta`<br>--> All conditions should match for a |
| **anyOf()** | This is used for calculating the logical disjunction of multiple matchers. This means that the object under consideration matches ANY of the specified matchers. | `assertThat( "InterviewBit",anyOf(co`<br>-->Any of the conditions should mat |
| **describedAs()** | This adds a description to the matcher. | `assertThat("Friday", describedAs("F` |

## 27. What is the relationship between the cyclomatic complexity of code and unit tests?

The code cyclomatic complexity is calculated based on the number of decision points of the code that contributes to different execution paths as shown in the image below:



This is why, higher the cyclomatic complexity, the more difficult is to attain the required code coverage.

## 28. What is the keyboard shortcut for running the Junit test cases in eclipse IDE?

We can press Alt+Shift+X for running the test. We can also right-click on the project, select Run As and then select JUnit Test. If the test method needs to be rerun, then we can press Ctrl+F11.

## 29. Define code coverage. What are the different types of code coverages?

The extent to which the source code is unit tested is called coverage. There are three types of coverage techniques, they are:

- **Statement coverage**: This ensures that each statement/line in the source code is executed and tested.
- **Decision coverage**: This ensures that every decision point that results in true or false is executed and run.
- **Path coverage**: This ensures that every possible route from a given point is run and tested.

## 30. Is it possible to test the Java method for a timeout in JUnit?

Yes, it is possible. @Test annotation provides a timeout parameter that takes value in milliseconds in JUnit that is used for instructing JUnit to pass or fail the test method based on execution time. For instance, if a test method has Test annotation as `@Test(timeout=30)` , then it means that if the execution does not complete in 30ms, then the test case will be failed. This helps in verifying the SLA of the Java method.

## 31. What are some of the best practices to be followed while writing code for making it more testable?

Following are some of the best practices that can be followed while coding to make the code more testable:

- Make use of interfaces as a wrapper to implementation classes. This helps testers to replace the class with mocks or stubs to test the module.
- Use dependency injection wherever needed instead of creating. new objects. This makes it easy for testing individual modules and also supply dependency in the test configuration.
- Avoid using static methods as that makes it difficult to test because they cannot be called polymorphically.

## 32. Why does JUnit report only the first failure in a single attempt?

JUnit is usually designed in a way that it deals with smaller tests and is capable of running each assessment with a boundary of separate analysis. Due to this, it reports only the first failure on each test case attempt.

## 33. How can we do testing for private methods?

It is generally not required to test private methods directly. Since they are private, it is assumed that they are called from public methods. If these methods are working as expected, then by extension it is considered that the private methods are working correctly. However, we can explore how the private methods are tested using reflection.

Let us consider we have a public class `Item` with private method `getItemName()` as shown below:

```
public class Item {
   private String getItemName(String itemName) {
          //Some logic
          if(itemName.equals("computer")) {
          return itemName;
          }
         return "laptop";
   }
}
```

Consider we have a test class ItemTest that tests this method as shown below:

```
import static org.junit.Assert.assertEquals;
import java.lang.reflect.Method;
import org.junit.BeforeClass;
import org.junit.Test;
public class ItemTest {
   public static Item item;

   @BeforeClass
   public static void beforeClass() {
       item = new Item();
   }

   @Test
   public void testGetItemNameSuccess() throws Exception {
       String expectedName = "computer";
       Method method = Item.class.getDeclaredMethod("getItemName", String.class);
       method.setAccessible(true);
       String actualName = (String) method.invoke(item, actualName);
       assertEquals(expectedName, actualName);
   }
}
```

In the above code, we see that we can test the private method by using `getDeclaredMethod` method of the Method class that belongs to reflection strategy. Since we know that the private methods cant be invoked directly, we can use the Method class object's invoke() method and send the input parameters. Then as usual, we can compare the results with the expected and the actual methods.

### 34. How can you test a generics class?

Generics allows creating classes, interfaces or methods that operate with different data types at a time. In order to test Generic entities, we can test it for one or two datatypes for the logic since the datatype allocation are evaluated at compilation time for type-correctness.

## JUnit Mockito Interview Questions

### 35. Why do we need mocking in unit testing?

Mocking is the process of creating and using mock objects that simulates the behaviour of real objects and is used to isolate the behaviour of the module under test from its external dependencies or services. These are particularly useful in unit testing and help in making unit test cases repeatable and predictable. Mocking is required in the following cases:

- Whenever the module under test has dependencies that are not fully implemented. For example, if we have a module that calls a REST API that is still in progress, then to test our module, it is advised to mock the API call and perform unit testing of our module.
- Whenever the module under test updates the system states. For example, whenever the module involves database calls that creates or updates or delete data from the system, it is very much important to mock those objects.
- Even if we have DB calls that just retrieves the data, it is advised to mock those because of the risk of DB availability.

### 36. What is Mockito? What are some of its advantages?

Mockito is an open-source, Java-based, mocking framework that allows the creation of test objects that simulate the behaviour (mock) of real-world objects. This helps in achieving test-driven or behaviour-driven development. The framework allows developers to verify system behaviours without establishing expectations. Mockito framework attempts to eliminate expect-run-verify development patterns by removing external specifications and dependencies. Some of the advantages of Mockito are:

- Mocks are created at runtime, hence reordering method input parameters or renaming interface methods will not break test code.
- Mockito supports returning of values.
- It supports exception simulation
- It provides a check on the order of method calls.
- It helps in creating mock objects using annotation.

## 37. When and why should we use spy?

Spy is a partial mock object supported by the Mockito framework. Following are the features of spy objects:

- Whenever mock is not set up, interaction on spy results in real method calls. Spy objects allow verifying interactions such as whether the method was called and how many times it was called, etc.
- They provide flexibility for setting up partial mocks. For instance, if an object has 2 methods, and we want one method to be mocked and the other to be called actually, then we can use spies.

The main difference between mock and spy is that in mock, we create a complete fake object whereas, in spy, we have a hybrid of a real object and fake/stubbed methods.

## 38. What is the difference between thenReturn and doReturn?

thenReturn and `doReturn` are used for setting up stubs and are provided by the Mockito framework. They are generally used along with when clause as --> `when-thenReturn` and `doReturn-when` . `when-thenReturn` is used in most cases due to better readability.

Following are the differences between `thenReturn` and `doReturn` :

- **Type safety**:
  `doReturn` takes Object parameter, unlike `thenReturn` . Hence there is no type check in `doReturn` at compile time. In the case of `thenReturn` , whenever the type mismatches during runtime, the `WrongTypeOfReturnValue` exception is raised. This is why `when-thenReturn` is a better option whenever we know the type.
- **No side-effect:**
  Since `doReturn` does not have type safety, there are no side effects when it is being used. Whenever `thenReturn` is used, if the type mismatch occurs, an exception is thrown which might result in the abortion of test cases.

## 39. What is the main difference between @Mock and @InjectMocks?

@Mock annotation is used for creating mocks whereas @InjectMocks is used for creating class objects. Whenever the actual method body has to be executed of a given class, then we can use @InjectMocks.

## 40. Why can’t we mock static methods in Mockito?

Static methods belong to the class and not to any objects which means that all instances of the class use the same static method. They are more like procedural code and are used in legacy systems. Mock libraries create mocks at runtime utilizing dynamic instance creation using interfaces or inheritance. Since static methods are not linked with any instance, it is not possible to mock them using Mockito. However, we have frameworks like PowerMock that supports static methods.

## 41. How can we mock void methods in Mockito?

Consider we have a method called `updateItem()` which internally calls a void method called `updateItem()` which interacts with the database and updates the object.

```
public Item updateItem(Item item) {
    if (item == null || item.getItemId() < 0) {
        throw new IllegalArgumentException("Invalid Id");
    }
    itemRepository.updateItem(item); //void method that updates database state
    return item;
}
```

We can mock the void method in the following ways:

1. `doNothing-when` : This is used when we do not want to check for the return parameters and skip the actual execution. When this mocked method is called, then it does nothing. The test case would be:

```
@Test
public void updateItemTest() {
    Item item = new Item(2, "Item 1");
    doNothing().when(itemRepository).updateItem(any(Item.class));
    itemService.updateItem(item);
}
```

2. `doAnswer-when` : We can also make use of doAnswer-when whenever we need to perform additional actions like computing return values based on method parameters when a mocked method is called. This is done using the following:

```
@Test
public void updateItemTest() {
    Item item = new Item(2, "Item 2");
    doAnswer((args) -> {
        System.out.println("doAnswer block entered.");
        assertEquals(customer, args.getArgument(0));
        return null;
   }).when(itemRepository).updateItem(any(Item.class));
    itemService.updateItem(item);
}
```

3. `doThrow-when` : If we want to throw an exception from void methods or any normal methods, then we can use doThrow-when as shown below:

```
@Test(expected = Exception.class)
public void updateItemTest() {
    Item item = new Item(3, "Item 3");
    doThrow(new Exception("Item Invalid")).when(itemRepository).updateItem(any(Item.cla
    itemService.updateItem(item);
}
```

## Conclusion

Writing Unit Test Cases has various advantages like design testability, code testability, enhances code maintainability and helps developers enforce object-oriented principles that aids to avoid code smells like long methods, large conditions, long classes etc. It helps to identify any existing logic flaw in the application. Due to these reasons, it has become important for developers to know how to unit test the functionalities developed by them. JUnit is one such popular framework that has robust support for unit testing.

# Links to More Interview Questions

C Interview Questions

Php Interview Questions

C Sharp Interview Questions

Web Api Interview Questions

Hibernate Interview Questions

Node Js Interview Questions

Cpp Interview Questions

Oops Interview Questions

Devops Interview Questions

Machine Learning Interview Questions

Docker Interview Questions

Mysql Interview Questions

Css Interview Questions

Laravel Interview Questions

Asp Net Interview Questions

Django Interview Questions

Dot Net Interview Questions

Kubernetes Interview Questions

Operating System Interview Questions

React Native Interview Questions

Aws Interview Questions

Git Interview Questions

Java 8 Interview Questions

Mongodb Interview Questions

Dbms Interview Questions

Spring Boot Interview Questions

Power Bi Interview Questions

Pl Sql Interview Questions

Tableau Interview Questions

Linux Interview Questions

Ansible Interview Questions

Java Interview Questions

Jenkins Interview Questions