JUnit

What is JUnit

JUnit is a unit testing framework for Java programming language. It plays a crucial role test-driven development, and is a family of unit testing frameworks collectively known as xUnit.

JUnit promotes the idea of "first testing then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented

Features of JUnit

* JUnit is an open source framework, which is used for writing and running tests.
* Provides annotations to identify test methods.
* Provides assertions for testing expected results.
* Provides test runners for running tests.
* JUnit tests allow you to write codes faster, which increases quality.
* JUnit is elegantly simple. It is less complex and takes less time.
* JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
* JUnit tests can be organized into test suites containing test cases and even other test suites.

What is a Unit Test Case

A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected. To achieve the desired results quickly, a test framework is required. JUnit is a perfect unit test framework for Java programming language.

There must be at least two unit test cases for each requirement − one positive test and one negative test. If a requirement has sub-requirements, each sub-requirement must have at least two test cases as positive and negative.

JUnit Annotations

**JUnit Annotations** is a special form of syntactic meta-data that can be added to Java source code for better code readability and structure

@Test - This annotation is a replacement of org.junit.TestCase which indicates that public void method to which it is attached can be executed as a test Case.

@Before - This annotation is used if you want to execute some statement such as preconditions before each test case.

@BeforeClass - This annotation is used if you want to execute some statements before all the test cases for e.g. test connection must be executed before all the test cases.

@After - This annotation can be used if you want to execute some statements after each Test Case for e.g resetting variables, deleting temporary files ,variables, etc.

@AfterClass - This annotation can be used if you want to execute some statements after all test cases for e.g. Releasing resources after executing all test cases.

Testing exceptions in junit

we can simply use the expected attribute of the **@Test annotation** to declare that we expect an exception to be thrown anywhere in the annotated test method.

You can **use assertThrows()** , which allows you to test multiple exceptions within the same test.

Parameterized Test

 Parameterized tests allow a developer to run the same test over and over again using different values. There are five steps that you need to follow to create a parameterized test.

* Annotate test class with @RunWith(Parameterized.class).
* Create a public static method annotated with @Parameters that returns a Collection of Objects (as Array) as test data set.
* Create a public constructor that takes in what is equivalent to one "row" of test data.
* Create an instance variable for each "column" of test data.
* Create your test case(s) using the instance variables as the source of the test data.

Mockito

What is Mocking

Mocking is a way to test the functionality of a class in isolation. Mocking does not require a database connection or properties file read or file server read to test a functionality. Mock objects do the mocking of the real service. A mock object returns a dummy data corresponding to some dummy input passed to it.

Benefits of Mockito

* **No Handwriting** − No need to write mock objects on your own.
* **Refactoring Safe** − Renaming interface method names or reordering parameters will not break the test code as Mocks are created at runtime.
* **Return value support** − Supports return values.
* **Exception support** − Supports exceptions.
* **Order check support** − Supports check on order of method calls.
* **Annotation support** − Supports creating mocks using annotation.

Verify calls on Mocks

Mockito verify() method can be used to test number of method invocations too. We can test exact number of times, at least once, at least, at most number of invocation times for a mocked method

Mockito – Spying

Mockito provides option to create spy on real objects. When spy is called, then actual method of real object is called.

## Syntax

//create a spy on actual object

calcService = spy(calculator);

//perform operation on real object

//test the add functionality

Assert.assertEquals(mathApplication.add(20.0, 10.0),30.0,0);

[Mockito JUnit Rules](https://www.javatpoint.com/mockito-junit-rules)

Mockito JUnit Rule **helps keeping tests clean**. It initializes mocks, validates usage and detects incorrect stubbing. Make sure to configure your rule with strictness(Strictness) which automatically detects stubbing argument mismatches and is planned to be the default in Mockito

Difference between JUnit and Mockito

**JUnit** is a framework that helps with writing and running your unit tests. **Mockito** (or any other mocking tool) is a framework that you specifically use to efficiently write certain kind of tests. One core aspect in unit testing is the fact that you want to isolate your "class under test" from anything else **in the** world.

JUnit is the Java library used to write tests (offers support for running tests and different extra helpers - like setup and teardown methods, test sets etc.). Mockito is a library that enables writing tests using the mocking approach. JUnit is used to test API's in source code.

Hamcrest Matchers

Hamcrest is **a popular framework that help us to create the matcher objects**. It is used for writing software tests and also performs unit testing in Java programming language. Hamcrest is mainly used with other unit testing frameworks like JUnit, jMockit, Mockito, etc.

Mockito Annotations

**@Mock:** It is used to mock the objects that helps in minimizing the repetitive mock objects. It makes the test code and verification error easier to read as parameter names (field names) are used to identify the mocks.

**@RunWith:** It is a class-level annotation. It is used to keep the test clean and improves debugging. It also detects the unused stubs available in the test and initialize mocks annotated with @Mock annotation

**@InjectMocks:** It marks a field or parameter on which the injection should be performed. It allows shorthand mock and spy injections and minimizes the repetitive mocks and spy injection. In Mockito, the mocks are injected either by setter injection, constructor injection, and property injection.l

**@Captor:** It allows the creation of a field-level argument captor. It is used with the Mockito's verify() method to get the values passed when a method is called

**@Spy -** It allows the creation of partially mock objects. In other words, it allows shorthand wrapping of the field instances in a spy object.

PowerMock-Mockito

PowerMock is an open-source Java framework used for creating a mock object in unit testing.

The PowerMock framework provides a class called **PowerMockito** used to create mock objects and initiates verification and expectation. The PowerMockito provides the functionality to work with the Java reflection API.