It is an open-source testing framework for java programmers. The java programmer can create test cases and test his/her own code.It is one of the unit testing framework.

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.

Test Suites: A test suite bundles a few unit test cases and runs them together. In JUnit, both @RunWith and @Suite annotation are used to run the suite test.

1.void addTest(Test test): Adds a test to the suite.

2.void addTestSuite(Class<? extends TestCase> testClass): Adds the tests from the given class to the suite.

3.int countTestCases(): Counts the number of test cases that will be run by this test.

4.String getName(): Returns the name of the suite.

5.void run(TestResult result): Runs the tests and collects their result in a TestResult.

6.void setName(String name): Sets the name of the suite.

7.Test testAt(int index): Returns the test at the given index.

8.int testCount(): Returns the number of tests in this suite.

9.static Test warning(String message): Returns a test which will fail and log a warning message.

Test Runners: Test runner is used for executing the test cases.

Current version is junit 4. To perform unit testing, we need to create test cases. The unit test case is a code which ensures that the program logic works as expected.

The org.junit package contains many interfaces and classes for junit testing such as Assert, Test, Before, After etc.

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.

JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails.

Types of unit testing:

There are two ways to perform unit testing:

1) manual testing : If you execute the test cases manually without any tool support, it is known as manual testing. It is time consuming and less reliable.

2) Automated Testing: If you execute the test cases by tool support, it is known as automated testing. It is fast and more reliable.

Annotations for Junit testing:

The Junit 4.x framework is annotation based, so let's see the annotations that can be used while writing the test cases.

@Test annotation specifies that method is the test method.

@Test(timeout=1000) annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

@BeforeClass annotation specifies that method will be invoked only once, before starting all the tests.

@Before annotation specifies that method will be invoked before each test.

@After annotation specifies that method will be invoked after each test.

@AfterClass annotation specifies that method will be invoked only once, after finishing all the tests.

JUnit Classes: JUnit classes are important classes, used in writing and testing JUnits. Some of the important classes are −

TestCase − Contains a test case that defines the fixture to run multiple tests.

1.int countTestCases(): Counts the number of test cases executed by run(TestResult result).

2.TestResult createResult(): Creates a default TestResult object.

3.String getName(): Gets the name of a TestCase.

4.TestResult run(): A convenience method to run this test, collecting the results with a default TestResult object.

5.void run(TestResult result): Runs the test case and collects the results in TestResult.

6.void setName(String name): Sets the name of a TestCase.

7.void setUp(): Sets up the fixture, for example, open a network connection.

8.void tearDown(): Tears down the fixture, for example, close a network connection.

9.String toString(): Returns a string representation of the test case.

TestResult − Contains methods to collect the results of executing a test case.

1.void addError(Test test, Throwable t): Adds an error to the list of errors.

2.void addFailure(Test test, AssertionFailedError t): Adds a failure to the list of failures.

3.void endTest(Test test): Informs the result that a test was completed.

4.int errorCount(): Gets the number of detected errors.

5.Enumeration<TestFailure> errors(): Returns an Enumeration for the errors.

6.int failureCount(): Gets the number of detected failures.

7.void run(TestCase test): Runs a TestCase.

8.int runCount(): Gets the number of run tests.

9.void startTest(Test test): Informs the result that a test will be started.

10.void stop(): Marks that the test run should stop.

Assert class: The org.junit.Assert class provides methods to assert the program logic.

Methods of Assert class:

void assertEquals(boolean expected,boolean actual): checks that two primitives/objects are equal. It is overloaded.

void assertTrue(boolean condition): checks that a condition is true.

void assertFalse(boolean condition): checks that a condition is false.

void assertNull(Object obj): checks that object is null.

void assertNotNull(Object obj): checks that object is not null.

Mockito is a mocking framework, JAVA-based library that is used for effective unit testing of JAVA applications. Mockito is used to mock interfaces so that a dummy functionality can be added to a mock interface that can be used in unit testing. Mockito facilitates creating mock objects seamlessly. It uses Java Reflection in order to create mock objects for a given interface. Mock objects are nothing but proxy for actual implementations.

Mockito adds a functionality to a mock object using the methods when().

Mockito can ensure whether a mock method is being called with reequired arguments or not. It is done using the verify() method.

Mockito provides Inorder class which takes care of the order of method calls that the mock is going to make in due course of its action.

Mockito provides the capability to a reset a mock so that it can be reused later.eg.//reset mock -> reset(calcService);

Behavior Driven Development is a style of writing tests uses given, when and then format as test methods. Mockito provides special methods to do so.

//Given

given(calcService.add(20.0,10.0)).willReturn(30.0);

//when

double result = calcService.add(20.0,10.0);

//then

Assert.assertEquals(result,30.0,0);

Mockito provides a special Timeout option to test if a method is called within stipulated time frame.//passes when add() is called within 100 ms.

verify(calcService,timeout(100)).add(20.0,10.0);

Mockito provides the following additional methods to vary the expected call counts.

atLeast (int min) − expects min calls.

atLeastOnce () − expects at least one call.

atMost (int max) − expects max calls.

Mockito provides the capability to a mock to throw exceptions, so exception handling can be tested.

Mockito provides a Answer interface which allows stubbing with generic interface.

Mockito provides option to create spy on real objects. When spy is called, then actual method of real object is called. eg.//create a spy on actual object - >calcService = spy(calculator);

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.