**JUnit**

# Overview

Testing is the process of checking the functionality of an application to ensure it runs as per requirements. Unit testing comes into picture at the developers’ level; it is the testing of single entity (class or method). Unit testing plays a critical role in helping a software company deliver quality products to its customers.

Unit testing can be done in two ways − manual testing and automated testing.

|  |  |
| --- | --- |
| **Manual Testing** | **Automated Testing** |
| Executing a test cases manually without any tool support is known as manual testing. | Taking tool support and executing the test cases by using an automation tool is known as automation testing. |
| **Time-consuming and tedious** − Since test cases are executed by human resources, it is very slow and tedious. | **Fast** − Automation runs test cases significantly faster than human resources. |
| **Huge investment in human resources** − As test cases need to be executed manually, more testers are required in manual testing. | **Less investment in human resources** − Test cases are executed using automation tools, so less number of testers are required in automation testing. |
| **Less reliable** − Manual testing is less reliable, as it has to account for human errors. | **More reliable** − Automation tests are precise and reliable. |
| **Non-programmable** − No programming can be done to write sophisticated tests to fetch hidden information. | **Programmable** − Testers can program sophisticated tests to bring out hidden information. |

## 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. This approach is like "test a little, code a little, test a little, code a little." It increases the productivity of the programmer and the stability of program code, which in turn reduces the stress on the programmer and the time spent on debugging.

## 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 increase 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.

## 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.

A formal written unit test case is characterized by a known input and an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post-condition.

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.

# Test Framework

JUnit is a **Regression Testing Framework** used by developers to implement unit testing in Java, and accelerate programming speed and increase the quality of code. JUnit Framework can be easily integrated with either of the following −

* Eclipse
* Ant
* Maven

## Features of JUnit Test Framework

JUnit test framework provides the following important features −

* Fixtures
* Test suites
* Test runners
* JUnit classes

### Fixtures

**Fixtures** is a fixed state of a set of objects used as a baseline for running tests. The purpose of a test fixture is to ensure that there is a well-known and fixed environment in which tests are run so that results are repeatable. It includes −

* setUp() method, which runs before every test invocation.
* tearDown() method, which runs after every test method.

## 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.

## Test Runners

Test runner is used for executing the test cases. Here is an example that assumes the test class **TestJunit** already exists.

## JUnit Classes

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

* **Assert** − Contains a set of assert methods.
* **TestCase** − Contains a test case that defines the fixture to run multiple tests.
* **TestResult** − Contains methods to collect the results of executing a test case.

# JUnit – API

The most important package in JUnit is **junit.framework**, which contains all the core classes. Some of the important classes are as follows –

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Class Name** | **Functionality** |
| 1 | Assert | A set of assert methods. |
| 2 | TestCase | A test case defines the fixture to run multiple tests. |
| 3 | TestResult | A TestResult collects the results of executing a test case. |
| 4 | TestSuite | A TestSuite is a composite of tests. |

## Assert Class

* Following is the declaration for **org.junit.Assert** class −
* public class Assert extends java.lang.Object
* This class provides a set of assertion methods useful for writing tests. Only failed assertions are recorded. Some of the important methods of Assert class are as follows –

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **void assertEquals(boolean expected, boolean actual)** |
| Checks that two primitives/objects are equal. |
| 2 | **void assertFalse(boolean condition)** |
| Checks that a condition is false. |
| 3 | **void assertNotNull(Object object)** |
| Checks that an object isn't null. |
| 4 | **void assertNull(Object object)** |
| Checks that an object is null. |
| 5 | **void assertTrue(boolean condition)** |
| Checks that a condition is true. |
| 6 | **void fail()** |
| Fails a test with no message. |

## TestCase Class

Following is the declaration for **org.junit.TestCaset** class −

public abstract class TestCase extends Assert implements Test

A test case defines the fixture to run multiple tests. Some of the important methods of **TestCase** class are as follows −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 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 Class

Following is the declaration for **org.junit.TestResult** class −

public class TestResult extends Object

A TestResult collects the results of executing a test case. It is an instance of the Collecting Parameter pattern. The test framework distinguishes between failures and errors. A failure is anticipated and checked for with assertions. Errors are unanticipated problems like an ArrayIndexOutOfBoundsException. Some of the important methods of **TestResult** class are as follows −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 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 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. |

## TestSuite Class

Following is the declaration for **org.junit.TestSuite** class:

public class TestSuite extends Object implements Test

A TestSuite is a Composite of tests. It runs a collection of test cases. Some of the important methods of **TestSuite** class are as follows –

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 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. |

# Using Assertion

## Assertion

All the assertions are in the Assert class.

public class Assert extends java.lang.Object

This class provides a set of assertion methods, useful for writing tests. Only failed assertions are recorded. Some of the important methods of Assert class are as follows −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **void assertEquals(boolean expected, boolean actual)** |
| Checks that two primitives/objects are equal. |
| 2 | **void assertTrue(boolean expected, boolean actual)** |
| Checks that a condition is true. |
| 3 | **void assertFalse(boolean condition)** |
| Checks that a condition is false. |
| 4 | **void assertNotNull(Object object)** |
| Checks that an object isn't null. |
| 5 | **void assertNull(Object object)** |
| Checks that an object is null. |
| 6 | **void assertSame(boolean condition)** |
| The assertSame() method tests if two object references point to the same object. |
| 7 | **void assertNotSame(boolean condition)** |
| The assertNotSame() method tests if two object references do not point to the same object. |
| 8 | **void assertArrayEquals(expectedArray, resultArray);** |
| The assertArrayEquals() method will test whether two arrays are equal to each other. |

## Annotation

Annotations are like meta-tags that you can add to your code, and apply them to methods or in class. These annotations in JUnit provide the following information about test methods −

* which methods are going to run before and after test methods.
* which methods run before and after all the methods, and.
* which methods or classes will be ignored during the execution.

The following table provides a list of annotations and their meaning in JUnit –

|  |  |
| --- | --- |
| **Sr.No.** | **Annotation & Description** |
| 1 | **@Test** |
| The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case. |
| 2 | **@Before** |
| Several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before each Test method. |
| 3 | **@After** |
| If you allocate external resources in a Before method, you need to release them after the test runs. Annotating a public void method with @After causes that method to be run after the Test method. |
| 4 | **@BeforeClass** |
| Annotating a public static void method with @BeforeClass causes it to be run once before any of the test methods in the class. |
| 5 | **@AfterClass** |
| This will perform the method after all tests have finished. This can be used to perform clean-up activities. |
| 6 | **@Ignore** |
| The Ignore annotation is used to ignore the test and that test will not be executed. |
| 7 | **@Test(timeout=1000)** |
| annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second). |

# Execution Procedure

The execution procedure is as follows −

* First of all, the beforeClass() method executes only once.
* The afterClass() method executes only once.
* The before() method executes for each test case, but before executing the test case.
* The after() method executes for each test case, but after the execution of test case.
* In between before() and after(), each test case executes.