What is JUnit?

**Result** result = **JUnitCore.runClasses(JunitTestSuite.class);**

for (Failure failure : **result.getFailures()**) {

System.out.println(failure.toString());

}

System.out.println(**result.wasSuccessful());**

**@Test**

**@RunWith(Suite.class)**

**@Suite.SuiteClasses({**

**TestJunit1.class,**

**TestJunit2.class**

**})**

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 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.

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 JUnit Setup

Create a java class file name TestJunit in **C:\>JUNIT\_WORKSPACE**

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class TestJunit {

@Test

public void testAdd() {

String str = "Junit is working fine";

assertEquals("Junit is working fine",str);

}

}

Create a java class file name TestRunner in **C:\>JUNIT\_WORKSPACE** to execute test case(s).

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.class);

for (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

System.out.println(result.wasSuccessful());

}

}

## Step 7: Verify the Result

Compile the classes using **javac** compiler as follows −

C:\JUNIT\_WORKSPACE>javac TestJunit.java TestRunner.java

Now run the Test Runner to see the result as follows −

C:\JUNIT\_WORKSPACE>java TestRunner

Verify the output.

true

# JUnit - 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.

Let's check one example −

import junit.framework.\*;

public class JavaTest extends TestCase {

protected int value1, value2;

// assigning the values

protected void setUp(){

value1 = 3;

value2 = 3;

}

// test method to add two values

public void testAdd(){

double result = value1 + value2;

assertTrue(result == 6);

}

}

## 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. Given below is an example that uses TestJunit1 & TestJunit2 test classes.

import org.junit.runner.RunWith;

import org.junit.runners.Suite;

//JUnit Suite Test

@RunWith(Suite.class)

@Suite.SuiteClasses({

TestJunit1.class ,TestJunit2.class

})

public class JunitTestSuite {

}

import org.junit.Test;

import org.junit.Ignore;

import static org.junit.Assert.assertEquals;

public class TestJunit1 {

String message = "Robert";

MessageUtil messageUtil = new MessageUtil(message);

@Test

public void testPrintMessage() {

System.out.println("Inside testPrintMessage()");

assertEquals(message, messageUtil.printMessage());

}

}

import org.junit.Test;

import org.junit.Ignore;

import static org.junit.Assert.assertEquals;

public class TestJunit2 {

String message = "Robert";

MessageUtil messageUtil = new MessageUtil(message);

@Test

public void testSalutationMessage() {

System.out.println("Inside testSalutationMessage()");

message = "Hi!" + "Robert";

assertEquals(message,messageUtil.salutationMessage());

}

}

## Test Runners

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

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.class);

for (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

System.out.println(result.wasSuccessful());

}

}

## 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.

/\*

\* This class prints the given message on console.

\*/

public class MessageUtil {

private String message;

//Constructor

//@param message to be printed

public MessageUtil(String message){

this.message = message;

}

// prints the message

public String printMessage(){

System.out.println(message);

return message;

}

}

# 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. |

Let's use some of the above-mentioned methods in an example. Create a java class file named TestJunit1.java in **C:\>JUNIT\_WORKSPACE**.

import org.junit.Test;

import static org.junit.Assert.\*;

public class TestJunit1 {

@Test

public void testAdd() {

//test data

int num = 5;

String temp = null;

String str = "Junit is working fine";

//check for equality

assertEquals("Junit is working fine", str);

//check for false condition

assertFalse(num > 6);

//check for not null value

assertNotNull(str);

}

}

Next, create a java class file named **TestRunner1.java** in C:\>JUNIT\_WORKSPACE to execute test case(s).

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner1 {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit1.class);

for (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

System.out.println(result.wasSuccessful());

}

}

Compile the test case and Test Runner classes using javac.

C:\JUNIT\_WORKSPACE>javac TestJunit1.java TestRunner1.java

Now run the Test Runner, which will run the test case defined in the provided Test Case class.

C:\JUNIT\_WORKSPACE>java TestRunner1

Verify the output.

true

## TestCase Class

Following is the declaration for **org.junit.TestCase** 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. |

Let's use some of the above-mentioned methods in an example. Create a java class file named **TestJunit2.java** in C:\>JUNIT\_WORKSPACE.

import junit.framework.TestCase;

import org.junit.Before;

import org.junit.Test;

public class TestJunit2 extends TestCase {

protected double fValue1;

protected double fValue2;

@Before

public void setUp() {

fValue1 = 2.0;

fValue2 = 3.0;

}

@Test

public void testAdd() {

//count the number of test cases

System.out.println("No of Test Case = "+ this.countTestCases());

//test getName

String name = this.getName();

System.out.println("Test Case Name = "+ name);

//test setName

this.setName("testNewAdd");

String newName = this.getName();

System.out.println("Updated Test Case Name = "+ newName);

}

//tearDown used to close the connection or clean up activities

public void tearDown( ) {

}

}

Next, create a java class file named **TestRunner2.java** in C:\>JUNIT\_WORKSPACE to execute test case(s).

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner2 {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit2.class);

for (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

System.out.println(result.wasSuccessful());

}

}

Compile the test case and Test Runner classes using javac.

C:\JUNIT\_WORKSPACE>javac TestJunit2.java TestRunner2.java

Now run the Test Runner, which will run the test case defined in the provided Test Case class.

C:\JUNIT\_WORKSPACE>java TestRunner2

Verify the output.

No of Test Case = 1

Test Case Name = testAdd

Updated Test Case Name = testNewAdd

true

## 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. |

Create a java class file named **TestJunit3.java** in C:\>JUNIT\_WORKSPACE.

import org.junit.Test;

import junit.framework.AssertionFailedError;

import junit.framework.TestResult;

public class TestJunit3 extends TestResult {

// add the error

public synchronized void addError(Test test, Throwable t) {

super.addError((junit.framework.Test) test, t);

}

// add the failure

public synchronized void addFailure(Test test, AssertionFailedError t) {

super.addFailure((junit.framework.Test) test, t);

}

@Test

public void testAdd() {

// add any test

}

// Marks that the test run should stop.

public synchronized void stop() {

//stop the test here

}

}

Next, create a java class file named **TestRunner3.java** in C:\>JUNIT\_WORKSPACE to execute test case(s).

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner3 {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit3.class);

for (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

System.out.println(result.wasSuccessful());

}

}

Compile the test case and Test Runner classes using javac.

C:\JUNIT\_WORKSPACE>javac TestJunit3.java TestRunner3.java

Now run the Test Runner, which will run the test case defined in the provided Test Case class.

C:\JUNIT\_WORKSPACE>java TestRunner3

Verify the output.

true

## 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. |

Create a java class file named **JunitTestSuite.java** in C:\>JUNIT\_WORKSPACE to create Test suite.

import junit.framework.\*;

public class JunitTestSuite {

public static void main(String[] a) {

// add the test's in the suite

TestSuite suite = new TestSuite(TestJunit1.class, TestJunit2.class, TestJunit3.class );

TestResult result = new TestResult();

suite.run(result);

System.out.println("Number of test cases = " + result.runCount());

}

}

Compile the Test suite classes using javac.

C:\JUNIT\_WORKSPACE>javac JunitTestSuite.java

Now run the Test Suite.

C:\JUNIT\_WORKSPACE>java JunitTestSuite

Verify the output.

No of Test Case = 1

Test Case Name = testAdd

Updated Test Case Name = testNewAdd

Number of test cases = 3

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. |