# JUnit

JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks collectively known as xUnit that originated with JUnit.

## Features

* JUnit is an open source framework which is used for writing & running tests.
* Provides Annotation to identify the test methods.
* Provides Assertions for testing expected results.
* Provides Test runners for running tests.
* JUnit tests allow you to write code faster which increasing quality
* JUnit is elegantly simple. It is less complex & 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 test is going fine 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 the another part of code (method) works as expected. To achieve those desired results quickly, test framework is required .JUnit is perfect unit test framework for java programming language.

A formal written unit test case is characterized by a known input and by 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 postcondition.

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.

## What is 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 followings:

* Eclipse
* Ant
* Maven

## Features

JUnit test framework provides 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 suite:

Test suite means bundle a few unit test cases and run it together. In JUnit, both **@RunWith** and **@Suite** annotation are used to run the suite test. Here is an example which 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 runner:

Test runner is used for executing the test cases. Here is an example which assumes TestJunit test class 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 classesare important classes which are used in writing and testing JUnits. Some of the important classes are

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

# JUnit API

## Important API's of JUnit:

The most important package in JUnit is **junit.framework** which contain all the core classes. Some of the important class are

|  |  |  |
| --- | --- | --- |
| **Serial 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:

|  |  |
| --- | --- |
| **S.N.** | **Methods & Description** |
| 1 | **void assertEquals(boolean expected, boolean actual)** Check that two primitives/Objects are equal |
| 2 | **void assertFalse(boolean condition)** Check that a condition is false |
| 3 | **void assertNotNull(Object object)** Check that an object isn't null. |
| 4 | **void assertNull(Object object)** Check that an object is null |
| 5 | **void assertTrue(boolean condition)** Check 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

|  |  |
| --- | --- |
| **S.N.** | **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

|  |  |
| --- | --- |
| **S.N.** | **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 name TestJunit3.java in **C:\ > JUNIT\_WORKSPACE**

import org.junit.Test;

import junit.framework.AssertionFailedError;

import junit.framework.TestResult;

public class TestJunit3 extends TestResult {

## 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

|  |  |
| --- | --- |
| **S.N.** | **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. |

## Annotation:

Annotations are like meta-tags that you can add to you code and apply them to methods or in class. These annotation in JUnit gives us information about test methods , which methods are going to run before & after test methods, which methods run before & after all the methods, which methods or class will be ignore during execution.

List of annotations and their meaning in JUnit :

|  |  |
| --- | --- |
| **S.N.** | **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. |

Create a java class file name JunitAnnotation.java in **C:\ > JUNIT\_WORKSPACE** to test annotation

import org.junit.After;

import org.junit.AfterClass;

import org.junit.Before;

import org.junit.BeforeClass;

import org.junit.Ignore;

import org.junit.Test;

public class JunitAnnotation {

//execute before class

@BeforeClass

public static void beforeClass() {

System.out.println("in before class");

}

//execute after class

@AfterClass

public static void afterClass() {

System.out.println("in after class");

}

//execute before test

@Before

public void before() {

System.out.println("in before");

}

//execute after test

@After

public void after() {

System.out.println("in after");

}

//test case

@Test

public void test() {

System.out.println("in test");

}

//test case ignore and will not execute

@Ignore

public void ignoreTest() {

System.out.println("in ignore test");

}

}

Next, let's create a java class file name **TestRunner.java** in **C:\ > JUNIT\_WORKSPACE** to execute annotaions

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(JunitAnnotation.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 JunitAnnotation.java TestRunner.java

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

C:\JUNIT\_WORKSPACE>java TestRunner

Verify the output.

in before class

in before

in test

in after

in after class

true

# JUnit - Time Test

Junit provides a handy option of Timeout. If a test case takes more time than specified number of milliseconds then Junit will automatically mark it as failed. The **timeout** parameter is used along with @Test annotation. Now let's see @Test(timeout) in action.

# JUnit - Exceptions Test

Junit provides a option of tracing the Exception handling of code. You can test the code whether code throws desired exception or not. The **expected** parameter is used along with @Test annotation. Now let's see*@Test(expected)* in action.

@Test(expected = ArithmeticException.class)

public void testPrintMessage() {

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

messageUtil.printMessage();

}

# JUnit - Parameterized Test

Junit 4 has introduced a new feature **Parameterized tests**.Parameterized tests allow developer to run the same test over and over again using different values. There are five steps, that you need to follow to create**Parameterized tests**.

* 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 tests case(s) using the instance variables as the source of the test data.

The test case will be invoked once per each row of data. Let's see Parameterized tests in action.

## Create a Class

* Create a java class to be tested say PrimeNumberChecker.java in **C:\ > JUNIT\_WORKSPACE**.

public class PrimeNumberChecker {

public Boolean validate(final Integer primeNumber) {

for (int i = 2; i < (primeNumber / 2); i++) {

if (primeNumber % i == 0) {

return false;

}

}

return true;

}

}

## Create Parameterized Test Case Class

* Create a java test class say PrimeNumberCheckerTest.java.

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

import java.util.Arrays;

import java.util.Collection;

import org.junit.Test;

import org.junit.Before;

import org.junit.runners.Parameterized;

import org.junit.runners.Parameterized.Parameters;

import org.junit.runner.RunWith;

import static org.junit.Assert.assertEquals;

@RunWith(Parameterized.class)

public class PrimeNumberCheckerTest {

private Integer inputNumber;

private Boolean expectedResult;

private PrimeNumberChecker primeNumberChecker;

@Before

public void initialize() {

primeNumberChecker = new PrimeNumberChecker();

}

// Each parameter should be placed as an argument here

// Every time runner triggers, it will pass the arguments

// from parameters we defined in primeNumbers() method

public PrimeNumberCheckerTest(Integer inputNumber,

Boolean expectedResult) {

this.inputNumber = inputNumber;

this.expectedResult = expectedResult;

}

@Parameterized.Parameters

public static Collection primeNumbers() {

return Arrays.asList(new Object[][] {

{ 2, true },

{ 6, false },

{ 19, true },

{ 22, false },

{ 23, true }

});

}

// This test will run 4 times since we have 5 parameters defined

@Test

public void testPrimeNumberChecker() {

System.out.println("Parameterized Number is : " + inputNumber);

assertEquals(expectedResult,

primeNumberChecker.validate(inputNumber));

}

}

## Create Test Runner Class

Create a java class file name TestRunner.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 TestRunner {

public static void main(String[] args) {

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

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

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

}

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

}

}

# JUnit Extensions

Following are the JUnit extensions

* Cactus
* JWebUnit
* XMLUnit
* MockObject

## Cactus:

Cactus is a simple test framework for unit testing server-side java code (Servlets, EJBs, Tag Libs, Filters). The intent of Cactus is to lower the cost of writing tests for server-side code. It uses JUnit and extends it. Cactus implements an in-container strategy, meaning that tests are executed inside the container.

Cactus Ecosystem is made of several components:

* **Cactus Framework** is the heart of Cactus. It is the engine that provides the API to write Cactus tests.
* **Cactus Integration Modules** are front ends and frameworks that provide easy ways of using the Cactus Framework (Ant scripts, Eclipse plugin, Maven plugin).

Here is the code example how cactus can be used.

import org.apache.cactus.\*;

import junit.framework.\*;

public class TestSampleServlet extends ServletTestCase {

@Test

public void testServlet() {

// Initialize class to test

SampleServlet servlet = new SampleServlet();

// Set a variable in session as the doSomething()

// method that we are testing

session.setAttribute("name", "value");

// Call the method to test, passing an

// HttpServletRequest object (for example)

String result = servlet.doSomething(request);

// Perform verification that test was successful

assertEquals("something", result);

assertEquals("otherValue", session.getAttribute("otherName"));

}

}

## JWebUnit:

JWebUnit is a Java-based testing framework for web applications. It wraps existing testing frameworks such as HtmlUnit and Selenium with a unified, simple testing interface to allow you to quickly test the correctness of your web applications.

JWebUnit provides a high-level Java API for navigating a web application combined with a set of assertions to verify the application's correctness. This includes navigation via links, form entry and submission, validation of table contents, and other typical business web application features.

The simple navigation methods and ready-to-use assertions allow for more rapid test creation than using only JUnit or HtmlUnit. And if you want to switch from HtmlUnit to other plugins such as Selenium (available soon), there is no need to rewrite your tests.

Here is the sample code

import junit.framework.TestCase;

import net.sourceforge.jwebunit.WebTester;

public class ExampleWebTestCase extends TestCase {

private WebTester tester;

public ExampleWebTestCase(String name) {

super(name);

tester = new WebTester();

}

//set base url

public void setUp() throws Exception {

getTestContext().setBaseUrl("http://myserver:8080/myapp");

}

// test base info

@Test

public void testInfoPage() {

beginAt("/info.html");

}

}

## XMLUnit:

XMLUnit provides a single JUnit extension class, XMLTestCase, and a set of supporting classes that allow assertions to be made about:

* The differences between two pieces of XML (via Diff and DetailedDiff classes)
* The validity of a piece of XML (via Validator class)
* The outcome of transforming a piece of XML using XSLT (via Transform class)
* The evaluation of an XPath expression on a piece of XML (via classes implementing the XpathEngine interface)
* Individual nodes in a piece of XML that are exposed by DOM Traversal (via NodeTest class)

Lets say we have two pieces of XML that we wish to compare and assert that they are equal. We could write a simple test class like this:

import org.custommonkey.xmlunit.XMLTestCase;

public class MyXMLTestCase extends XMLTestCase {

// this test method compare two pieces of the XML

@Test

public void testForXMLEquality() throws Exception {

String myControlXML = "<msg><uuid>0x00435A8C</uuid></msg>";

String myTestXML = "<msg><localId>2376</localId></msg>";

assertXMLEqual("Comparing test xml to control xml",

myControlXML, myTestXML);

}

}

## MockObject:

In a unit test, mock objects can simulate the behavior of complex, real (non-mock) objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test.

The common coding style for testing with mock objects is to:

* Create instances of mock objects
* Set state and expectations in the mock objects
* Invoke domain code with mock objects as parameters
* Verify consistency in the mock objects

Below is the example of MockObject using Jmock.

import org.jmock.Mockery;

import org.jmock.Expectations;

class PubTest extends TestCase {

Mockery context = new Mockery();

public void testSubReceivesMessage() {

// set up

final Sub sub = context.mock(Sub.class);

Pub pub = new Pub();

pub.add(sub);

final String message = "message";

// expectations

context.checking(new Expectations() {

oneOf (sub).receive(message);

});

// execute

pub.publish(message);

// verify

context.assertIsSatisfied();

}

}