# JUnit Examples

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 which is collectively known as xUnit. JUnit is suitable for only unit testing, not for integration test. Unit testing is testing small units of codes like methods, or a class, whereas integration testing involves complete component or integration between multiple components.

- See more at: <http://java2novice.com/junit-examples/#sthash.OJBn5oKT.dpuf>

# Program: Simple JUnit test using @Test annotation.

We are using Junit 4 for examples shown here. Here is a simple example.

@org.junit.Test annotation: The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case. To run the method, JUnit first constructs a fresh instance of the class then invokes the annotated method. Any exceptions thrown by the test will be reported by JUnit as a failure. If no exceptions are thrown, the test is assumed to have succeeded.

assertEquals(String message, Object expected, Object actual): Asserts that two objects are equal. If they are not, an AssertionError is thrown with the given message. If expected and actual are null, they are considered equal.

In Eclipse, run MyEvenOddTest class as JUnit Test.

# Program: List of JUnit annotations.

Annotations are introduced in JUnit4. Here are the list of annotations and its descriptions

Reference: org.junit java docs

@Test: The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case. To run the method, JUnit first constructs a fresh instance of the class then invokes the annotated method. Any exceptions thrown by the test will be reported by JUnit as a failure. If no exceptions are thrown, the test is assumed to have succeeded.

public class MyTestClass {

@Test

public void myTestMethod() {

/\*\*

\* Use Assert methods to call your methods to be tested.

\* A simple test to check whether the given list is empty or not.

\*/

org.junit.Assert.assertTrue( new ArrayList().isEmpty() );

}

}

@Test (expected = Exception.class): Sometimes we need to test the exception to be thrown by the test. @Test annotation provides a parameter called 'expected', declares that a test method should throw an exception. If it doesn't throw an exception or if it throws a different exception than the one declared, the test fails.

public class MyTestClass {

@Test(expected=IOException.class)

public void myTestMethod() {

/\*\*

\* this test performs some IO operations, sometimes we may not

\* get access to the resources, then the method should through

\* declared exception.

\*/

....

....

}

}

@Test(timeout=100): Somethimes we need to mesure the performance interms of time. The @Test annotations provides an optional parameter called 'timeout', which causes a test to fail if it takes longer than a specified amount of clock time (measured in milliseconds).

public class MyTestClass {

@Test@Test(timeout=100)

public void myTestMethod() {

/\*\*

\* The IO operation has to be done with in 100 milli seconds. If not,

\* the test should fail.

\*/

....

....

}

}

@Before: When writing tests, it is common to find that several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before the Test method. The @Before methods of super classes will be run before those of the current class.

public class MyTestClass {

List<String> testList;

@Before

public void initialize() {

testList = new ArrayList<String>();

}

@Test

public void myTestMethod() {

/\*\*

\* Use Assert methods to call your methods to be tested.

\* A simple test to check whether the given list is empty or not.

\*/

org.junit.Assert.assertTrue( testList.isEmpty() );

}

}

@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. All @After methods are guaranteed to run even if a Before or Test method throws an exception. The @After methods declared in superclasses will be run after those of the current class.

public class MyTestClass {

OutputStream stream;

@Before

public void initialize() {

/\*\*

\* Open OutputStream, and use this stream for tests.

\*/

stream = new FileOutputStream(...);

}

@Test

public void myTestMethod() {

/\*\*

\* Now use OutputStream object to perform tests

\*/

...

...

}

@After

public void closeOutputStream() {

/\*\*

\* Close output stream here

\*/

try{

if(stream != null) stream.close();

} catch(Exception ex){

}

}

}

@BeforeClass: Sometimes several tests need to share computationally expensive setup (like logging into a database). While this can compromise the independence of tests, sometimes it is a necessary optimization. Annotating a public static void no-arg method with @BeforeClass causes it to be run once before any of the test methods in the class. The @BeforeClass methods of superclasses will be run before those the current class.

The annotations @BeforeClass and @Before are same in functionality. The only difference is the method annotated with @BeforeClass will be called once per test class based, and the method annotated with @Before will be called once per test based.

public class MyTestClass {

@BeforeClass

public void initGlobalResources() {

/\*\*

\* This method will be called only once per test class.

\*/

}

@Before

public void initializeResources() {

/\*\*

\* This method will be called before calling every test.

\*/

}

@Test

public void myTestMethod1() {

/\*\*

\* initializeResources() method will be called before calling this method

\*/

}

@Test

public void myTestMethod2() {

/\*\*

\* initializeResources() method will be called before calling this method

\*/

}

}

@AfterClass: If you allocate expensive external resources in a BeforeClass method you need to release them after all the tests in the class have run. Annotating a public static void method with @AfterClass causes that method to be run after all the tests in the class have been run. All @AfterClass methods are guaranteed to run even if a BeforeClass method throws an exception. The @AfterClass methods declared in superclasses will be run after those of the current class.

The annotations @AfterClass and @After are same in functionality. The only difference is the method annotated with @AfterClass will be called once per test class based, and the method annotated with @After will be called once per test based.

public class MyTestClass {

@BeforeClass

public void initGlobalResources() {

/\*\*

\* This method will be called only once per test class. It will be called

\* before executing test.

\*/

}

@Test

public void myTestMethod1() {

// write your test code here...

...

...

}

@BeforeClass

public void closeGlobalResources() {

/\*\*

\* This method will be called only once per test class. It will be called

\* after executing test.

\*/

}

}

@Ignore: Sometimes you want to temporarily disable a test or a group of tests. Methods annotated with Test that are also annotated with @Ignore will not be executed as tests. Also, you can annotate a class containing test methods with @Ignore and none of the containing tests will be executed. Native JUnit 4 test runners should report the number of ignored tests along with the number of tests that ran and the number of tests that failed.

You can also use @Ignore annotation at class level.

public class MyTestClass {

@Ignore

@Test

public void myTestMethod() {

/\*\*

\* This test will be ignored.

\*/

org.junit.Assert.assertTrue( new ArrayList().isEmpty() );

}

}

<< Previous Program | Next Program >>

- See more at: http://java2novice.com/junit-examples/junit-annotations/#sthash.ABfaamqA.dpuf

package com.java2novice.junit.samples;

public class MyEvenOdd {

public boolean isEvenNumber(int number){

boolean result = false;

if(number%2 == 0){

result = true;

}

return result;

}

}

package com.java2novice.junit.tests;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import com.java2novice.junit.samples.MyEvenOdd;

public class MyEvenOddTest {

@Test

public void testEvenOddNumber(){

MyEvenOdd meo = new MyEvenOdd();

assertEquals("10 is a even number", true, meo.isEvenNumber(10));

}

}

# Program: Assertion method Assert.assertArrayEquals() example.

Java Class: org.junit.Assert

Assert class provides a set of assertion methods useful for writing tests.

assertArrayEquals() method checks that two object arrays are equal or not. If they are not, it throws an AssertionError with the given message. Incase if expected input and actual inputs are null, then they are considered to be equal. It checks whether both arrays are having same number of elements or not, and all elements should be same. It compares based on the order. If mismatch in order results in failure.

- See more at: http://java2novice.com/junit-examples/assert-array-equals/#sthash.R9vDFpt7.dpuf

package com.java2novice.junit.tests;

import org.junit.Test;

import static org.junit.Assert.\*;

public class MyAssertArrayEqualsTest {

@Test

public void myTestMethod(){

/\*\*

\* we are demonstrating the usage of assertArrayEquals()

\* method here, so we are preparing input data here itself.

\* In real scenario, we will consider the methods returned

\* value which suppose to be test, as a input.

\*/

//assume that the below array represents expected result

String[] expectedOutput = {"apple", "mango", "grape"};

//assuem that the below array is returned from the method

//to be tested.

String[] methodOutput = {"apple", "mango", "grape"};

assertArrayEquals(expectedOutput, methodOutput);

}

- See more at: <http://java2novice.com/junit-examples/assert-array-equals/#sthash.R9vDFpt7.dpuf>

# 9999

Program: How to do JUnit test for comapring two list of user defined objects?

Java Class: org.junit.Assert

Assert class provides a set of assertion methods useful for writing tests.

assertArrayEquals() method checks that two object arrays are equal or not. If they are not, it throws an AssertionError with the given message. Incase if expected input and actual inputs are null, then they are considered to be equal. It checks whether both arrays are having same number of elements or not, and all elements should be same. It compares based on the order. If mismatch in order results in failure. Incase of user defined objects, the objects should override equals() and hashCode() methods. Then assertArrayEquals() method called equals and hashCode methods to check the equality on each object.

- See more at: <http://java2novice.com/junit-examples/test-list-objects/#sthash.z3jmOCn7.dpuf>

package com.java2novice.junit.tests;

import static org.junit.Assert.\*;

import org.junit.Before;

import org.junit.Test;

import com.java2novice.junit.samples.Employee;

public class MyArrayObjectEqualsTest {

Object[] expectedEmps = new Object[3];

@Before

public void initInputs(){

expectedEmps[0] = new Employee(1, "Nats", 15000);

expectedEmps[1] = new Employee(2, "Kalid", 25000);

expectedEmps[2] = new Employee(3, "Krish", 5000);

}

@Test

public void compareEmployees(){

/\*\*

\* convert List of objects to array of objects

\*/

Object[] testOutput = Employee.getEmpList().toArray();

assertArrayEquals(expectedEmps, testOutput);

}

}

- See more at: <http://java2novice.com/junit-examples/test-list-objects/#sthash.z3jmOCn7.dpuf>

package com.java2novice.junit.samples;

import java.util.ArrayList;

import java.util.List;

public class Employee {

private String name;

private int empId;

private int salary;

public Employee(int id, String name, int sal){

this.empId = id;

this.name = name;

this.salary = sal;

}

public boolean equals(Object obj){

Employee emp = (Employee) obj;

boolean status = false;

if(this.name.equalsIgnoreCase(emp.name)

&& this.empId == emp.empId

&& this.salary == emp.salary){

status = true;

}

return status;

}

public static List<Employee> getEmpList(){

List<Employee> emps = new ArrayList<Employee>();

emps.add(new Employee(1, "Nats", 15000));

emps.add(new Employee(2, "Kalid", 25000));

emps.add(new Employee(3, "Krish", 5000));

return emps;

}

public int hashCode(){

return this.empId;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getEmpId() {

return empId;

}

public void setEmpId(int empId) {

this.empId = empId;

}

public int getSalary() {

return salary;

}

public void setSalary(int salary) {

this.salary = salary;

}

}

- See more at: http://java2novice.com/junit-examples/test-list-objects/#sthash.z3jmOCn7.dpuf