Junit

* JUnit is a unit testing framework for Java programming language
* JUnit has been important in the development of test-driven development
* 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.

# Features of Junit

* Open source tool
* Provide annotations to identify test methods
* Provides assertions for validations
* Easy, fast and simple
* Runs automatically and provides the report

Note - Set the **JAVA\_HOME** environment variable to point to the base directory location where Java is installed on your machine.

Set the **JUNIT\_HOME** environment variable to point to the base directory location where JUNIT jar is stored on your machine

https://github.com/junit-team/junit4/wiki/Download-and-Install

# Junit Framework

Junit framework provides with the following features:

* Fixtures - Fixtures contain the setup() and teardown() methods
* Test Suites – Are bundle of test cases that needs to be executed
* Test Runners – The suites that should be executed is mentioned in runner class
* Junit Classes – Asserts and Test Result contain in Junit classes

# Fixtures

Test Fixtures used methods setUp() and teardown()

## setUp() method

Usually, there are some repeated tasks that must be done prior to each test case.

**@Before** annotation in JUnit is used on a method containing[Java](https://www.guru99.com/java-tutorial.html)code to run before each test case. i.e it runs before each test execution.

## TearDown() Method

Likewise, at the end of each test case, there may be some repeated tasks. **Example:** to clean up once test execution is over.

**@After** annotation is used on a method containing java code to run after each test case.

# Junit Annotations

Junit annotations are used for better readability, structure, and execution.

|  |  |
| --- | --- |
| Annotations | Description |
| @Test | All test methods are marked with these annotations |
| @Before | The method with this annotation will be executed before each test method |
| @After | The method with this annotation will be executed after each test method |
| @BeforeClass | This annotation is used if you want to execute some statements before all the test cases |
| @AfterClass | This annotation is used if you want to execute some statements after all the test cases |
| @Ignores | This annotation can be used if you want to ignore some statements during test execution |
| @Test(timeout = 500) | The test method will fail if it takes more than 500ms to execute |

# Creating a Test Runner

* Right click on src
* New 🡪 other 🡪 Java 🡪 Junit 🡪 Junit Test Suite
* Select the tests to be included
* Test runner is created as below:

package demo;

import org.junit.runner.RunWith;

import org.junit.runners.Suite;

import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)

@SuiteClasses({ TestingJUnit.class })

public class AllTests {

}

# Junit Assert Classes

|  |  |
| --- | --- |
| Method | Description |
| assertEquals(boolean expected, boolean actual) | It checks whether two values are equals |
| assertFalse(boolean condition) | functionality is to check that a condition is false. |
| assertNotNull | Method to check that an object is not null. |
| assertNull | Method to check that an object is null. |
| assertTrue(boolean condition) | Method to check that a condition is true. |
| fail() | If you want to throw any assertion error, you have fail(). |
| assertSame([String message] | Method to check that the two objects refer to the same object. |
| assertNotSame([String message] | Method to check that the two objects do not refer to the same object. |

@Test

**public** **void** testJUnit()

{

//Variable declaration

String string1="Junit";

String string2="Junit";

String string3= **new** String("test");

String string5=**null**;

**int** variable1=1;

**int** variable2=2;

**int**[] airethematicArrary1 = { 1, 2, 3 };

**int**[] airethematicArrary2 = { 1, 2, 3 };

//Assert statements

*assertEquals*(string1,string2); *assertSame*(string1,string2);

*assertNotSame*(string1, string3);

*assertNotNull*(string1);

*assertNull*(string5);

*assertTrue*(variable1<variable2);

*assertArrayEquals*(airethematicArrary1, airethematicArrary2);

}

# Performing Unit testing

Let us take a simple method to add two numbers.

Now, in order to test this code, we will create a Junit test class

We can add a method to test this code  
@Test

Public void twoPlusTwoIsFour()

{

SimpleCalculator cal = new SimpleCalculator()

assertEquals(4,cal.addNumbers(2,2);

}

But this test will work if developer has by mistake is multiplying numbers instead of addition

# Junit Expected Exception test

JUnit provides the facility to trace the exception and also to check whether the code is throwing expected exception or not.

Junit provides an easy and readable way for exception testing, you can use the Optional parameter (**expected**) of **@test** annotation.

While[Testing](https://www.guru99.com/software-testing.html)exceptions, you need to ensure that the exception class you are providing in that optional parameter of **@test annotation** is the same.

Example: dividing by zero throws an Arithmetic exception

@Test(expected = IllegalArgumentException.**class**)

**public** **void** testJUnit2()

{

Test2 test = **new** Test2();

System.***out***.println("Test case 2");

test.takeNumber(-1);

}

Write code for Calculating grade as below:

* If number is less than 0, throw exception “IllegalArgumentException”
* If number is less than 60, then F
* If number is less than 70, then D
* If number is less than 80, then C
* If number is less than 90, then B
* If greater than 90 then A

Add test methods to test different grades.

# Parameterized test

Parameterized test is to execute the same test over and over again using different values. It helps developer to save time in executing same test which differs only in their inputs and expected results.

Using Parameterized test, one can set up a test method that retrieves data from some data source.

Say we want to test addNumbers method with 10 different values. We can parameterize the test instead of copy pasting the code 10 times.

* Use the @RunWith(Parameterized.**class**) for Test class
* Declare private variables to store the data.

**private** **int** number1;

**private** **int** number2;

**private** **int** expectedResult;

**private** Test2 test2 = **new** Test2();

* Add a constructor to assign values of data to instance variables

**public** JUnitTest2 (**int** n1, **int** n2, **int** result) {

**this**.number1 = n1;

**this**.number2 = n2;

**this**.expectedResult = result;

}

* Create a method to take array of test data values

@Parameterized.Parameters

**public** **static** List input() {

Object newArray[][] = **new** Object[][] {

{1,2,3},{2,2,4},{3,4,8}

};

List<Object[]> aa = Arrays.*asList*(newArray);

**return** aa;

}

* Create a test method with assertions

@Test

**public** **void** testAddition() {

*assertEquals*(expectedResult, test2.addNumbers(number1, number2));

}

-----------------------Code-----------------------------

@RunWith(Parameterized.**class**)

**public** **class** JUnitTest2 {

**private** **int** number1;

**private** **int** number2;

**private** **int** expectedResult;

**private** Test2 test2 = **new** Test2();

**public** JUnitTest2 (**int** n1, **int** n2, **int** result) {

**this**.number1 = n1;

**this**.number2 = n2;

**this**.expectedResult = result;

}

@Parameterized.Parameters

**public** **static** List input() {

Object newArray[][] = **new** Object[][] {

{1,2,3},{2,2,4},{3,4,8}

};

List<Object[]> aa = Arrays.*asList*(newArray);

**return** aa;

}

@Test

**public** **void** testAddition() {

*assertEquals*(expectedResult, test2.addNumbers(number1, number2));

}

}