[JUnit Basics 4](#_Toc503469016)

[What is Junit? 4](#_Toc503469017)

[What is Unit Testing? 4](#_Toc503469018)

[Why you need JUnit testing 4](#_Toc503469019)

[Features and advantages of JUnit5 5](#_Toc503469020)

[How to Download and Install JUnit in Eclipse 6](#_Toc503469021)

[PART 1) Install Java 6](#_Toc503469022)

[PART 2) Download JUnit 6](#_Toc503469023)

[PART 3) JUnit Environment Setup 9](#_Toc503469024)

[PART 4) Install JUnit jar file in eclipse 12](#_Toc503469025)

[PART 5) Verifying whether required jar file for JUnit is in my build path 13](#_Toc503469026)

[PART 6) Verify JUnit setup 14](#_Toc503469027)

[JUnit Test Cases @Before @BeforeClass Annotation 18](#_Toc503469028)

[What is Test fixture 18](#_Toc503469029)

[Setup and Teardown 20](#_Toc503469030)

[Once-only setup 22](#_Toc503469031)

[Once-only tear down 22](#_Toc503469032)

[JUnit Test Suites 23](#_Toc503469033)

[Junit Test Runner 23](#_Toc503469034)

[First JUnit program 24](#_Toc503469035)

[What is JUnit Annotations? 27](#_Toc503469036)

[JUnit Assert Class 28](#_Toc503469037)

[JUnit Test Cases Class 29](#_Toc503469038)

[JUnit TestResult Class 30](#_Toc503469039)

[JUnit TestResult Class 31](#_Toc503469040)

[JUnit Test Suite Class 32](#_Toc503469041)

[JUnit Annotations 33](#_Toc503469042)

[JUnit Annotations Example 35](#_Toc503469043)

[What is Junit Assert? 39](#_Toc503469044)

[JUnit Assert methods 39](#_Toc503469045)

[Boolean 39](#_Toc503469046)

[Null object 39](#_Toc503469047)

[Identical 39](#_Toc503469048)

[Assert Equals 40](#_Toc503469049)

[Assert Array Equals 40](#_Toc503469050)

[Fail Message 40](#_Toc503469051)

[JUnit assertEquals 41](#_Toc503469052)

[Floating point assertions 41](#_Toc503469053)

[JUnit Assert Example 42](#_Toc503469054)

[Create JUnit Test Suite with Example: @RunWith @SuiteClasses 46](#_Toc503469055)

[Steps to create Test Suite and Test Runner 46](#_Toc503469056)

[JUnit Test Suite Example 48](#_Toc503469057)

[JUnit @Ignore Test Annotation 53](#_Toc503469058)

[What is JUnit @Ignore test annotation 53](#_Toc503469059)

[Junit Test Example - Ignore 53](#_Toc503469060)

[Creating a simple test class without ignoring a test 53](#_Toc503469061)

[Ignore a test method using @Ignore annotation 56](#_Toc503469062)

[Using @ ignore annotation with Condition 57](#_Toc503469063)

[Ignore all test methods using @Ignore annotation. 58](#_Toc503469064)

[JUnit Expected Exception Test: @Test(expected) 61](#_Toc503469065)

[Expected Parameter 61](#_Toc503469066)

[Example using @test(expected) 61](#_Toc503469067)

[JUnit ErrorCollector @Rule with Example 66](#_Toc503469068)

[Why use Error Collector? 66](#_Toc503469069)

[What is @Rule in jUnit? 67](#_Toc503469070)

[Example using ErrorCollector 67](#_Toc503469071)

[Benefits of JUnit ErrorCollector 69](#_Toc503469072)

[JUnit Parameterized Test 71](#_Toc503469073)

[What is Parameterized Test in Junit? 71](#_Toc503469074)

[Steps to create a Parameterized JUnit test 72](#_Toc503469075)

[JUnit in Parallel 77](#_Toc503469076)

[Technologies Used 77](#_Toc503469077)

[Project Setup 77](#_Toc503469078)

[JUnit Parallel Test 80](#_Toc503469079)

[XML Configurations 80](#_Toc503469080)

[Test Classes 81](#_Toc503469081)

[JUnit Parallel Tests 82](#_Toc503469082)

[Using the surefire plugin 84](#_Toc503469083)

[Conclusion 84](#_Toc503469084)

[TestNG Vs JUnit 85](#_Toc503469085)

[JUnit4 and TestNG Feature comparison 85](#_Toc503469086)

[Annotations 86](#_Toc503469087)

[Suite Test 88](#_Toc503469088)

[Using JUnit4 88](#_Toc503469089)

[Using TestNG 88](#_Toc503469090)

[Ignore Test 89](#_Toc503469091)

[Using JUnit4 89](#_Toc503469092)

[Using TestNG 89](#_Toc503469093)

[Exception Test 90](#_Toc503469094)

[Using JUnit4 90](#_Toc503469095)

[Using TestNG 90](#_Toc503469096)

[Timeout 91](#_Toc503469097)

[Using JUnit4 91](#_Toc503469098)

[Using TestNG 91](#_Toc503469099)

[Parameterized Test 92](#_Toc503469100)

[Using JUnit4 92](#_Toc503469101)

[Using TestNG 93](#_Toc503469102)

# JUnit Basics

## What is Junit?

* JUnit is an open source Unit[Testing](https://www.guru99.com/software-testing.html)Framework for JAVA. It is useful for[Java](https://www.guru99.com/java-tutorial.html)Developers to write and run repeatable tests.
* Erich Gamma and Kent Beck initially develop it. It is an instance of xUnit architecture. As the name implies, it is used for [Unit Testing](https://www.guru99.com/unit-testing-guide.html) of a small chunk of code.
* Developers who are following test-driven methodology must write and execute unit test first before any code.
* Once you are done with code, you should execute all tests, and it should pass. Every time any code is added, you need to re-execute all test cases and makes sure nothing is broken.

## What is Unit Testing?

Unit Testing is used to verify a small chunk of code by creating a path, function or a method. The term "unit" exist earlier than the object-oriented era. It is basically a natural abstraction of an object oriented system i.e. a Java class or object (its instantiated form).

Unit Testing and its importance can be understood by below-mentioned points:

* Unit Testing is used to identify defects early in software development cycle.
* Unit Testing will compel to read our own code. i.e. a developer starts spending more time in reading than writing.
* Defects in the design of code affect the development system. A successful code breeds the confidence of developer.

## Why you need JUnit testing

* It finds bugs early in the code, which makes our code more reliable.
* JUnit is useful for developers, who work in a test-driven environment.
* Unit testing forces a developer to read code more than writing.
* You develop more readable, reliable and bug-free code which builds confidence during development.

## Features and advantages of JUnit5

JUnit has added many new features in JUnit4. You can understand it easily by comparing JUnit 3.x and JUnit 4.x.

Below is quick comparison between JUnit4.x and JUnit 3.x -

* All the old assert statements are same as before.
* Most of the things are easier in JUnit4 as..
  + With JUnit 4 you are more capable of identifying exception. You can define expected exception as a parameter while using @test annotation.
  + Parameterized test is introduced, which enables us to use parameters.
  + JUnit4 still can execute JUnit3 tests.
* JUnit 4 can be used with java5 or higher version.
* While using JUnit4, you are not required to extend **JUnit.framework.TestCase**. You can just create a simple java class.
* You need to use annotations in spite of special method name as before.
  + Instead of using **setup**method, you need to use **@before**annotation.
  + Instead of using **teardown** method, put **@after** annotation.
  + Instead of using **testxxxx**before method name, use **@test** annotation.

# How to Download and Install JUnit in Eclipse

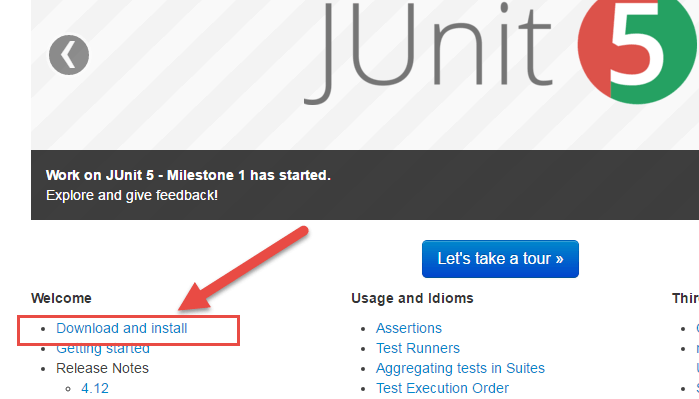
Installing Junit is a 6 part process. It is explained in detailed below-

## PART 1) Install Java

JUnit is a[Testing](https://www.guru99.com/software-testing.html)framework used to test[Java](https://www.guru99.com/java-tutorial.html)based application. So before installing JUnit, you need to configure or verify java development kit (JDK) in your machine.

## PART 2) Download JUnit

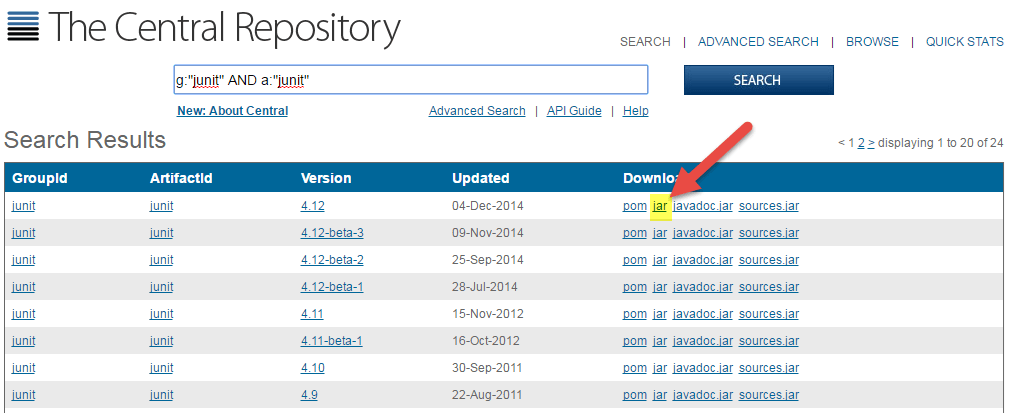
**Step 1)**Visit <http://junit.org/junit4/> and click Download and Install

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo1.png)

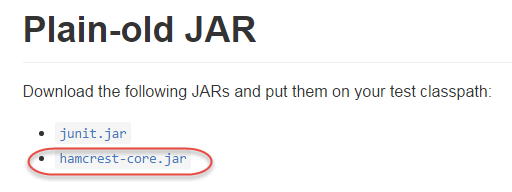
**Step 2)**Click junit.jar

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo2.png)

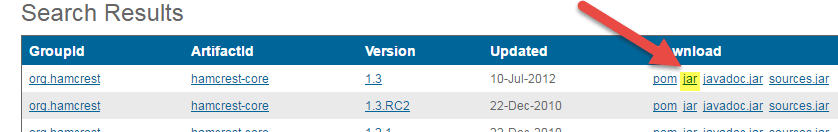
**Step 3)**In the central repository you are shown all versions of Junit that can be downloaded. Usually, you will select the latest version. Click on jar link to download Junit version 4.12 as shown below

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo3.png)

**Step 4)**Visit <https://github.com/junit-team/junit4/wiki/Download-and-Install> again. Click hamcrest-core.jar

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo4.png)

**Step 5)**Download the Jar

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo5.png)

For JUnit installation, you need JUnit jars, and you can download the desired version of JUnit jar file from JUnit official youbsite [http://www.junit.org](http://www.junit.org/)

Here is the jars list:

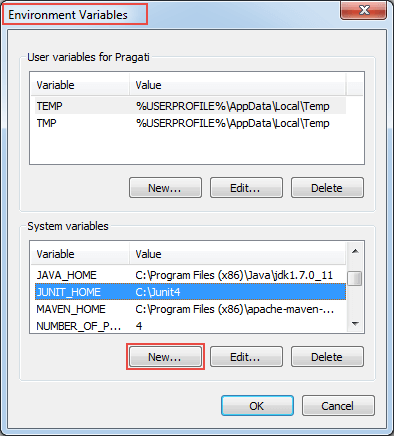
* JUnit.jar
* hamcrest-core.jar

## PART 3) JUnit Environment Setup

**Step 1)** You need to set **JUNIT\_HOME** environment variable to point out the base location where you have placed JUnit Jars.

For example, if you have created a JUnit folder in c: drive and placed jars there, then for environment settings you need to open control panel ->advanced ->environment variable.

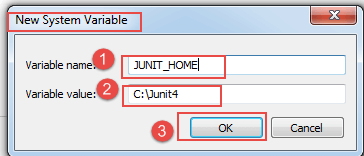
1. Under environment window clicks on "new" button.

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo6.png)

When you click on new button in environment variables, it will open another window

**Step 2)**A "New System Variable" window will open:

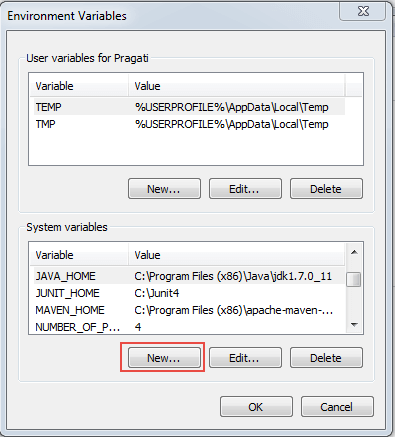
1. Provide variable name as "JUNIT\_HOME".
2. Provide JUnit value as JUnit path where you have copied JUnit jar files.
3. Click on OK.

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo7.png)

When you click on OK, it will create a new system variable with the given name and value. Which you can verify in environment variable window as shown in step 1 image.

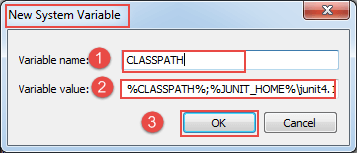
**Step 3)**After creating JUNIT\_HOME, create another variable with the name CLASSPATH. Again go to Environment Variables and follow the below steps.

1. Click on "new" button. When you click on new in environment variables, it will open another window.

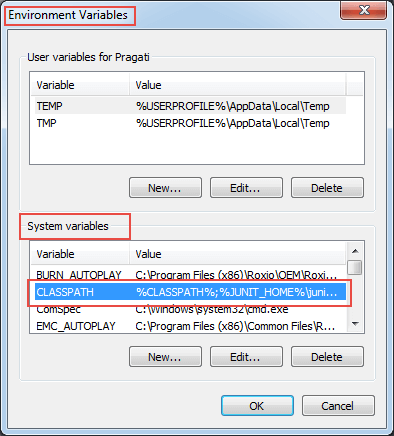
[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo8.png)

**Step 4)**In this step, point out JUNIT\_HOME to [JUnit.jar](http://bit.ly/My9IXz) which is placed in JUnit folder as given below:

1. Variable Name: JUNIT\_HOME
2. Variable Value: %CLASSPATH%;%JUNIT\_HOME%\JUnit4.10.jar;.;
3. Click on the OK button.

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo9.png)

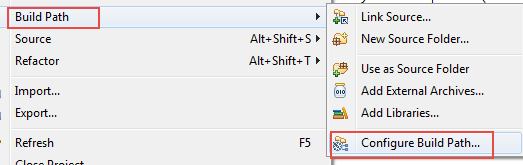
**Step 5)** Once you click on the 'OK' button, you can verify that a new environment variable named "CLASSPATH" can be seen under system variable. See below

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo10.png)

## PART 4) Install JUnit jar file in eclipse

**Step 1)**Right click on project:

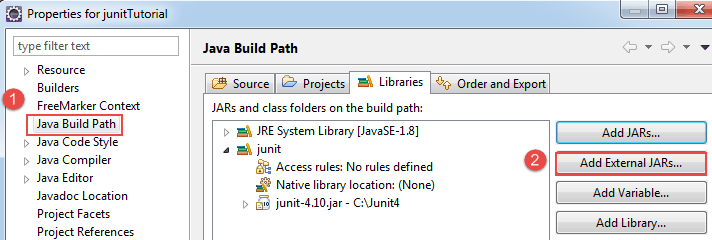
1. Click on "build path" and then
2. Click on "Configure build path".

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo11.png)

**Step 2)**In this step,

1. Go to java build path window as shown in below figure
2. Now click on "Add External JARs" button to add yyour downloaded JUnit.jar file with eclipse.

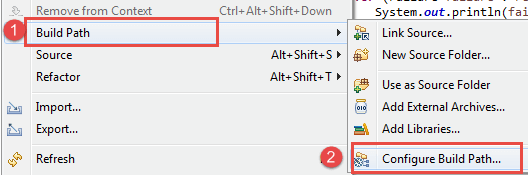
After adding a JUnit.jar file, click on 'OK' button to close java build path window.

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo12.png)

## PART 5) Verifying whether required jar file for JUnit is in my build path

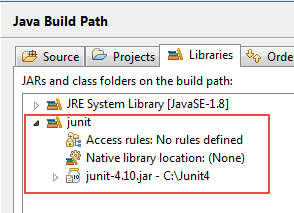
In order to verify JUnit jar file in eclipse, you need to follow below-mentioned steps:

1. Right click on project -> Build Path
2. Click on "Configure build path".

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo13.png)

**Step 2)** Java build path window will appear as shown below.In that window, go to Libraries tab to see all jar files. In jar file tree view, you need to look for the jar file name which is starting with JUnit.

Once you expand JUnit libraries, you can see java libraries as shown below:

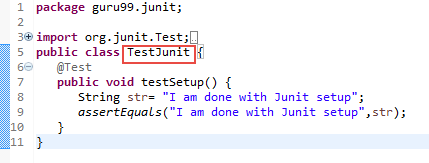
[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo14.png)

Now you are ready to use JUnit with eclipse.

## PART 6) Verify JUnit setup

You can create a simple JUnit test to verify JUnit setup. See below test class:

**Step 1)** Create a java class named TestJUnit.java and provide a simple assert statement.

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo15.png)

package guru99.junit;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class TestJunit {

@Test

public void testSetup() {

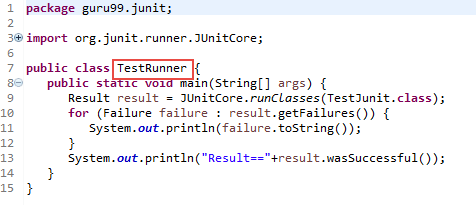
String str= "I am done with Junit setup";

assertEquals("I am done with Junit setup",str);

}

}

**Step 2)** Create a Test Runner class to execute above test.

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo16.png)

package guru99.junit;

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());

}

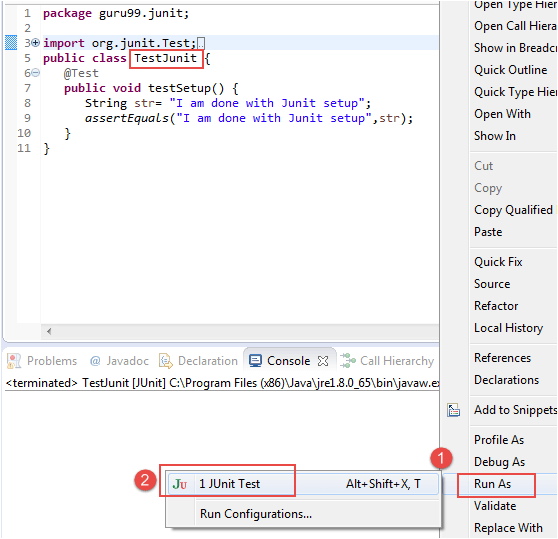
System.out.println("Result=="+result.wasSuccessful());

}

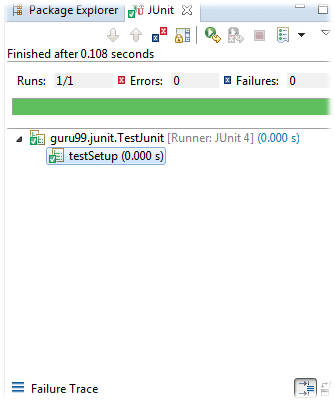
}

**Step 3)**Toexecute the test, follow below steps:

1. Right click on TestRunner.java and click on "Run As" as shown below
2. Another window will be open once you click on "Run As", click on "1 JUnit Test" as shown below:

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo17.png)

**Step 4)**Here is the output or result for your test. If it is successfully executed, it will show a green check mark in front of it.

[](https://cdn.guru99.com/images/junit/051716_0555_HowtoDownlo18.png)

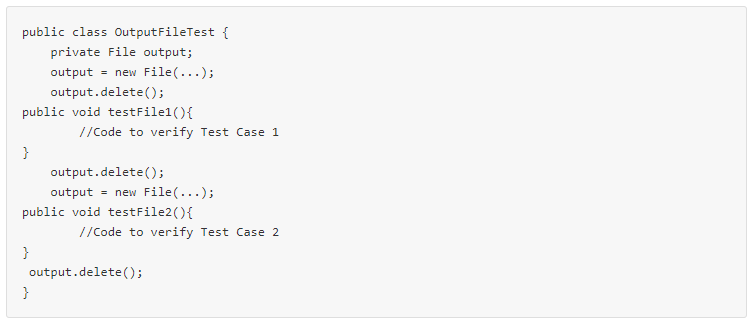
Click to download the above code for your testing purposes.

# JUnit Test Cases @Before @BeforeClass Annotation

JUnit framework also allows quick and easy generation of test cases and test data. The **org.Junit** package consist of many interfaces and classes for JUnit Testing such as Test, Assert, After, Before, etc.

## What is Test fixture

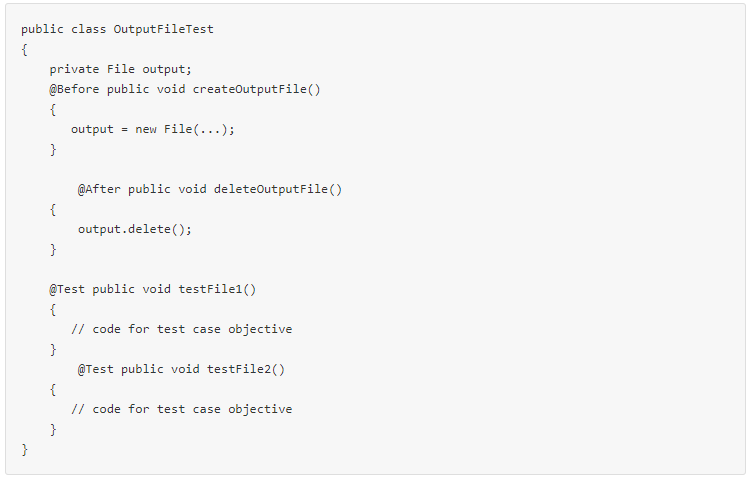
Before we understand what a test fixture is, let's study the code below .This code is designed to execute two test cases on a simple file.



Few issues here

* The code is not readable
* The code is not easy to maintain.
* When the test suite is complex the code could contain logical issues.

Compare the same code using JUnit



The code far more readable and maintainable. The above code structure is a **Text fixture.**

A test fixture is a context where a[Test Case](https://www.guru99.com/test-case.html)runs. Typically, test fixtures include:

* Objects or resources that are available for any test case.
* Activities required that makes these objects/resources available.
* These activities are
  1. allocation (**setup**)
  2. de-allocation (**teardown**).

## Setup and Teardown

* Usually, there are some repeated tasks that must be done prior to each test case. **Example:** create a database connection.
* Likewise, at the end of each test case, there may be some repeated tasks. **Example:** to clean up once test execution is over.
* JUnit provides annotations that help in setup and teardown. It ensures that resources are released, and the test system is in a ready state for next test case.

These annotations are discussed below-

**Setup**

**@Before** annotation 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 (regardless of the verdict)**

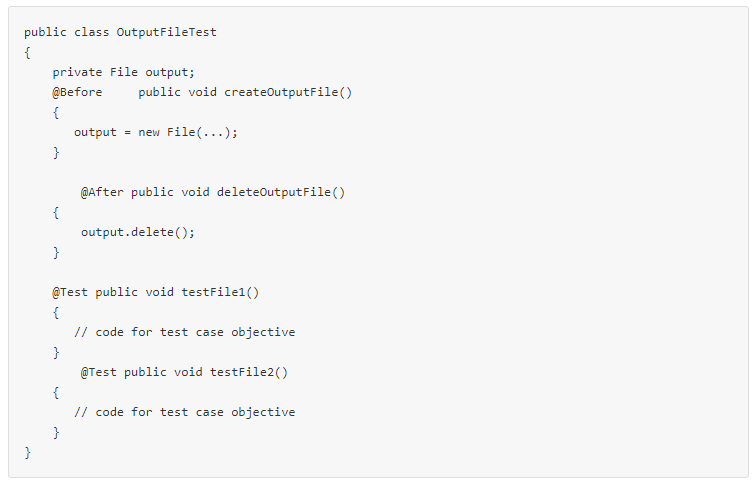
**@After** annotation is used on a method containing java code to run after each test case. These methods will run even if any exceptions are thrown in the test case or in the case of assertion failures.

**Note:**

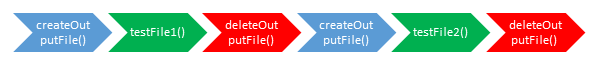
* It is allowed to have any number of annotations listed above.
* All the methods annotated with **@Before** will run before each test case, but they may run in any order.
* You can inherit @Before and @After methods from a super class, Execution is as follows: It is a standard execution process in JUnit.

1. Execute the @Before methods in the superclass
2. Execute the @Before methods in this class
3. Execute a @Test method in this class
4. Execute the @After methods in this class
5. Execute the @After methods in the superclass

**Example: Creating a class with file as a text fixture**



In the above example the chain of execution will be as follows-

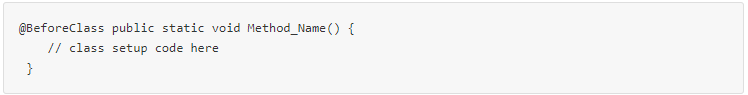


1. createOutputFile()
2. testFile1()
3. deleteOutputFile()
4. createOutputFile()
5. testFile2()
6. deleteOutputFile()

**Assumption:** testFile1() runs before testFile2()– which is not guaranteed.

#### Once-only setup

* It is possible to run a method only once for the entire test class before any of the tests are executed, and prior to any **@Before** method(s).
* "Once only setup" are useful for starting servers, opening communications, etc. It's time-consuming to close and re-open resources for each test.
* This can be done using the **@BeforeClass** annotation



#### Once-only tear down

* Similar to once only setup , a once-only cleanup method is also available. It runs after all test case methods and **@After**annotations have been executed.
* It is useful for stopping servers, closing communication links, etc.
* This can be done using the **@AfterClass** annotation



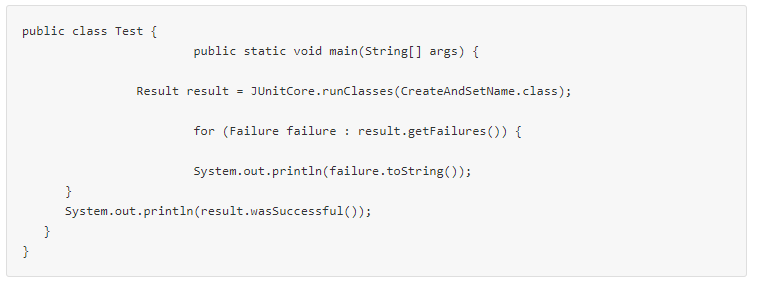
## JUnit Test Suites

If we want to execute multiple tests in a specified order, it can be done by combining all the tests in one place. This place is called as the test suites. More details on how to execute test suites and how it is used in JUnit will be covered in this [tutorial](https://www.guru99.com/create-junit-test-suite.html).

#### Junit Test Runner

JUnit provides a tool for execution of your test cases.

* **JUnitCore** class is used to execute these tests.
* A method called **runClasses** provided by **org.junit.runner.JUnitCore,** is used to run one or several test classes.
* Return type of this method is the **Result** object (**org.junit.runner.Result**), which is used to access information about the tests. See following code example for more clarity.



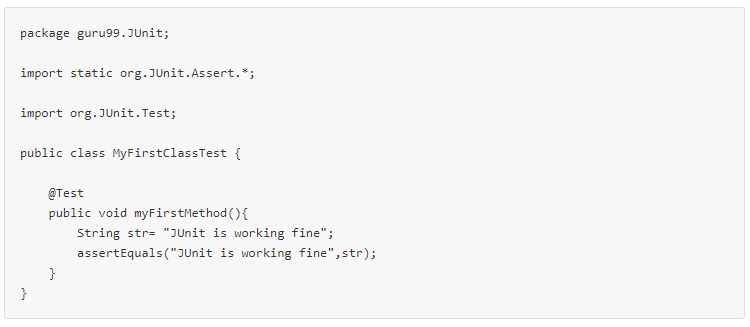
In above code "result" object is processed to get failures and successful outcomes of test cases we are executing.

#### First JUnit program

Fair knowledge of SDLC, java programming, and basics of software testing process helps in understanding JUnit program.

Let's understand Unit Testing using a live example. We need to create a test class with a test method annotated with **@Test** as given below:

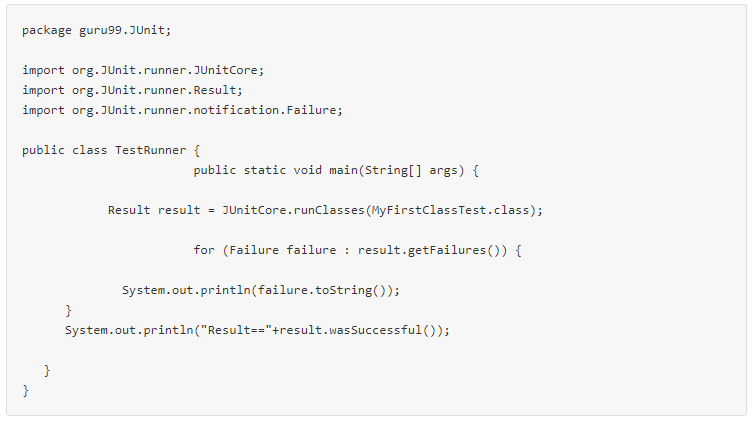
**MyFirstClassTest.java**



**TestRunner.java**

To execute our test method (above) ,we need to create a test runner. In the test runner we have to add test class as a parameter in JUnitCore'srunclasses() method . It will return the test result, based on whether the test is passed or failed.

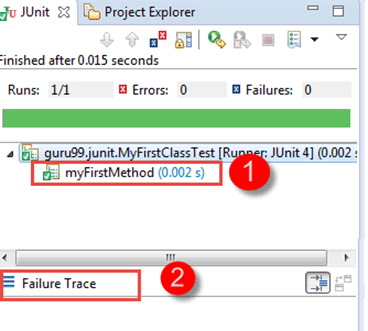
For more details on this see the code below :



**Output**

Once **TestRunner.java** executes our test methods we get output as failed or passed. Please find below output explanation:

1. In this example, after executing **MyFirstClassTest.java** , test is passed and result is in green.
2. If it would have failed it should have shown the result as Red and failure can be observed in failure trace. See below JUnit gui :

[](https://cdn.guru99.com/images/junit/051716_0608_JUnitTestfr2.png)

**Summary**:

* JUnit is a framework which supports several annotations to identify a method which contains a test.
* JUnit provides an annotation called **@Test,** which tells the JUnit that the public void method in which it is used can run as a test case.
* A test fixture is a context where a test case runs
* To execute multiple tests in a specified order, it can be done by combining all the tests in one place. This place is called as the test suites.
* JUnit provides a tool for execution of the tests where we can run our test cases referred as Test Runner.

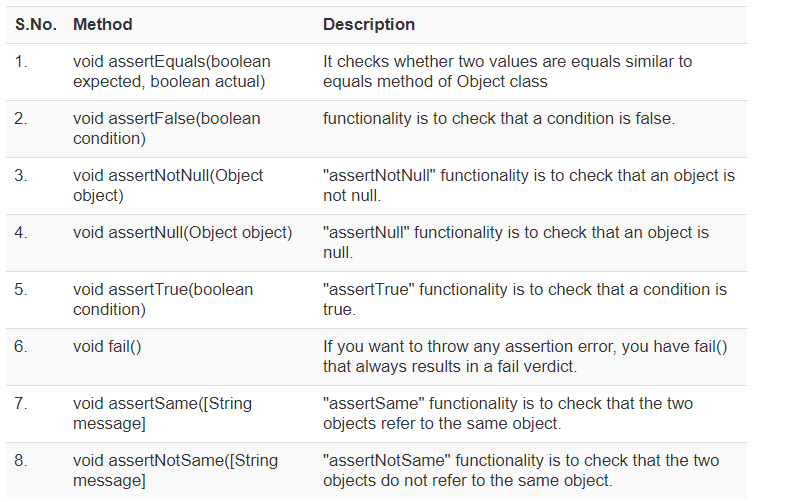
# What is JUnit Annotations?

* Annotation is a special form of syntactic meta-data that can be added to[Java](https://www.guru99.com/java-tutorial.html)source code for better code readability and structure. Variables, parameters, packages, methods and classes can be annotated.
* Some of the JUnit annotations which can be useful are :
* Before
* After
* BeforeClass
* AfterClass
* Ignore
* RunWith
* Test
* Before learn Junit Annotations in detail see below classes which are very useful while writing a test case:

1. org.junit.Assert
2. org.junit.TestCase
3. org.junit.TestResult
4. org.junit.TestSuite

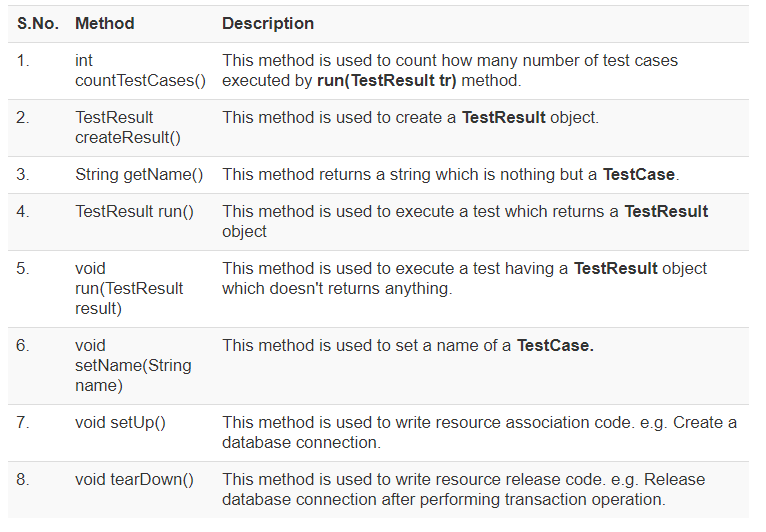
## JUnit Assert Class

* This class provides a bunch of assertion methods useful in writing a test case. If all assert statements are passed, test results are successful. If any assert statement fails, test results are failed.
* As you seen earlier, below table describes important Assert methods and description:



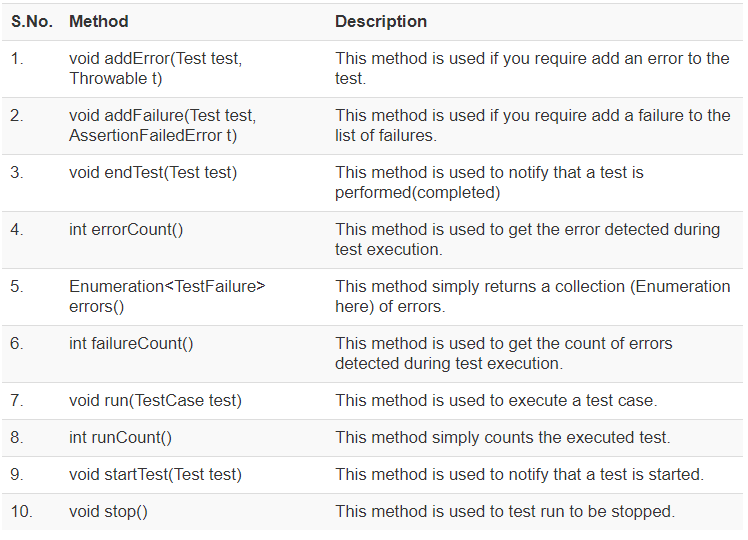
## JUnit Test Cases Class

* To run multiple test, TestCase class is available in **org.junit.TestCase** packages. Annotation @Test tells JUnit that this public void method (Test Case here) to which it is attached can be run as a test case.
* Below table shows some important methods available in **org.junit.TestCase**class:



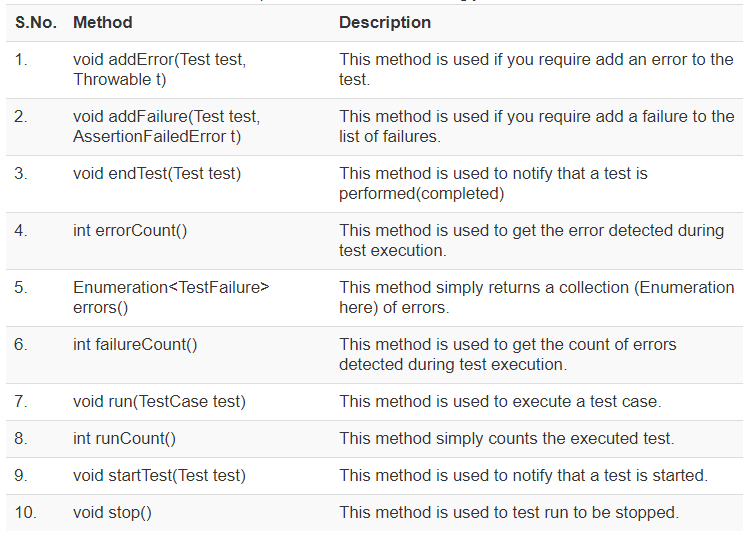
## JUnit TestResult Class

* When you execute a test, it returns a result (in the form of TestResult object). This TestResult object can be used to analyse the resultant object. This test result can be either failure or successful. See below table for important methods used in org.junit.TestResult class:



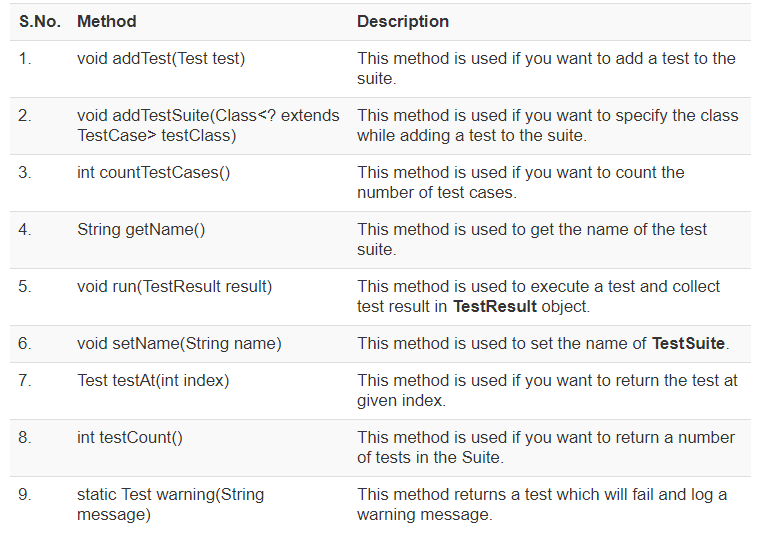
## JUnit TestResult Class

* When you execute a test, it returns a result (in the form of **TestResult**object). This TestResult object can be used to analyse the resultant object. This test result can be either failure or successful.
* See below table for important methods used in org.junit.TestResult class:



## JUnit Test Suite Class

* If you want to execute multiple tests in a specified order, it can be done by combining all the tests in one place. This place is called as the test suites.
* See below table for important methods used in **org.junit.TestSuite** class:

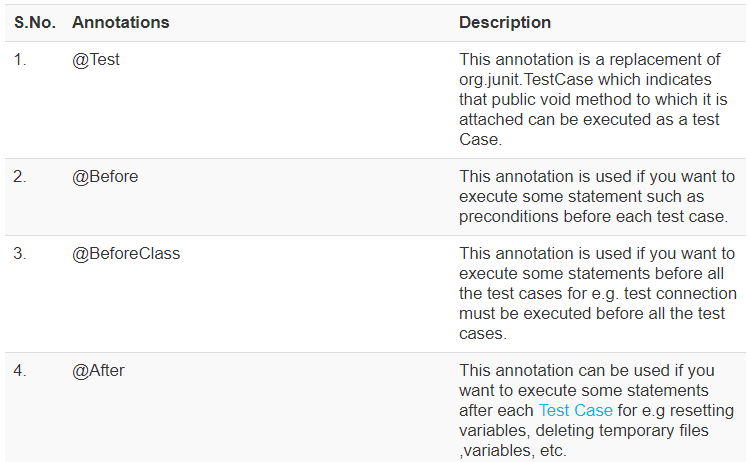


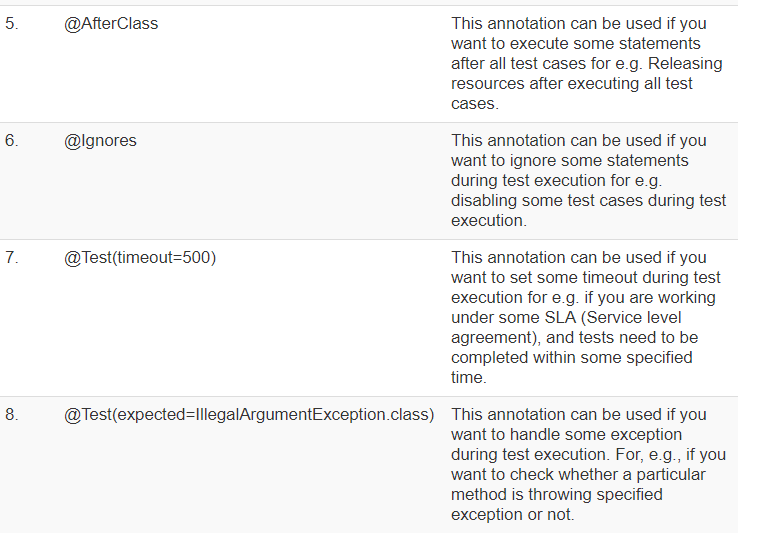
## JUnit Annotations

* Annotations were introduced in Junit4, which makes Java code more readable and simple. This is the big difference between Junit3 and Junit4 that Junit4 is annotation based.
* With the knowledge of annotations in Junit4, one can easily learn and implement a JUnit test.
* Below is the list of important and frequently used annotations:

1. @Before
2. @BeforeClass
3. @After
4. @AfterClass
5. @Test
6. @Ignore
7. @Test(timeout=500)
8. @Test(expected=IllegalArgumentException.class)

* See below table to understand more on annotation :





## JUnit Annotations Example

Let's create a class covering important JUnit annotations with simple print statements and execute it with a test runner class:

**Step 1)**Consider below java class having various methods which are attached to above-listed annotations:

**JunitAnnotationsExample.java**

package guru99.junit;

import static org.junit.Assert.assertEquals;

import static org.junit.Assert.assertFalse;

import java.util.ArrayList;

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 JunitAnnotationsExample {

private ArrayList<String> list;

@BeforeClass

public static void m1() {

System.out.println("Using @BeforeClass , executed before all test cases ");

}

@Before

public void m2() {

list = new ArrayList<String>();

System.out.println("Using @Before annotations ,executed before each test cases ");

}

@AfterClass

public static void m3() {

System.out.println("Using @AfterClass ,executed after all test cases");

}

@After

public void m4() {

list.clear();

System.out.println("Using @After ,executed after each test cases");

}

@Test

public void m5() {

list.add("test");

assertFalse(list.isEmpty());

assertEquals(1, list.size());

}

@Ignore

public void m6() {

System.out.println("Using @Ignore , this execution is ignored");

}

@Test(timeout = 10)

public void m7() {

System.out.println("Using @Test(timeout),it can be used to enforce timeout in JUnit4 test case");

}

@Test(expected = NoSuchMethodException.class)

public void m8() {

System.out.println("Using @Test(expected) ,it will check for specified exception during its execution");

}

}

**Step 2)** let's create a test runner class to execute above test:

**TestRunner.java**

package guru99.junit;

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(JunitAnnotationsExample.class);

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

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

}

System.out.println("Result=="+result.wasSuccessful());

}

}

**Expected Result**

* All the test cases will be executed one by one, and all print statement can be seen on a console.
* As discussed in above table @Before, @BeforeClass [ method m1() and m2() ] will be executed before each and before all test cases respectively.
* In the same way @after,@afterClass (method m3() and m4()) will be executed after each and after all test cases respectively. @ignore (method m6())will be treated as ignoring the test.

Let's analyse test cases used in above java class in detail:

1. Consider method m5() as given below :

@Test

public void m5() {

list.add("test");

assertFalse(list.isEmpty());

assertEquals(1, list.size());

}

In above method as you are adding a string in the variable "list" so

* **list.isEmpty()** will return false.
* **assertFalse(list.isEmpty())** must return true.
* As a result, the test case will **pass**.

As you have added only one string in the list, so the size is one.

* **list.size()** must return int value as "1" .
* So **assertEquals(1, list.size())** must return true.
* As a result, the test case will **pass**.

1. Consider method m7() as given below :

@Test(timeout = 10)

public void m7() {

System.out.println("Using @Test(timeout),it can be used to enforce timeout in JUnit4 test case");

}

As discussed above**@Test(timeout = 10)**annotation is used to enforce timeout in the test case.

1. Consider method m7() as given below :

@Test(expected = NoSuchMethodException.class)

public void m8() {

System.out.println("Using @Test(expected) ,it will check for specified exception during its execution");

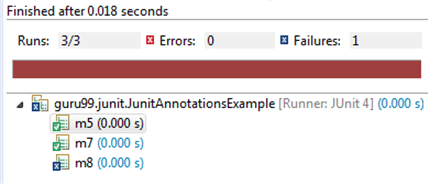
}

As discussed above **@Test(expected)**will check for specified exception during its execution so method m8() will throw "No Such Method Exception." As a result, the test will be executed with an exception.

As all test cases are passed, this results in a successful test execution.

**Actual Result**

As there are three test cases in above example, all test cases will be executed one by one. See output below**:**

[](https://cdn.guru99.com/images/junit/052416_0549_JUnitAnnota1.png)

**See below print statements which can be seen on console:**

* Using @BeforeClass , executed before all test cases
* Using @Before annotations, executed before each test cases
* Using @After, executed after each test cases
* Using @Before annotations, executed before each test cases
* Using @Test(timeout),it can be used to enforce timeout in JUnit4 test case
* Using @After, executed after each test cases
* Using @Before annotations, executed before each test cases
* Using @Test(expected) ,it will check for specified exception during its execution
* Using @After, executed after each test cases
* Using @AfterClass, executed after all test cases

# What is Junit Assert?

* Assert is a method useful in determining Pass or Fail status of a test case, The assert methods are provided by the class org.junit.Assert which extends java.lang.Object class.
* There are various types of assertions like Boolean, Null, Identical etc.
* Junit provides a class named Assert, which provides a bunch of assertion methods useful in writing test cases and to detect test failure

The assert methods are provided by the class**org.junit.Assert** which extends**java.lang.Object**class.

## JUnit Assert methods

#### Boolean

If you want to test the boolean conditions (true or false), you can use following assert methods

1. **assertTrue(condition)**
2. **assertFalse(condition)**

Here the condition is a boolean value.

#### Null object

If you want to check the initial value of an object/variable, you have the following methods:

1. **assertNull(object)**
2. **assertNotNull(object)**

Here object is[Java](https://www.guru99.com/java-tutorial.html)object **e.g.** assertNull(actual);

#### Identical

If you want to check whether the objects are identical (i.e. comparing two references to the same java object), or different.

1. **assertSame(expected, actual),** It will return true if **expected == actual**
2. **assertNotSame(expected, actual)**

#### Assert Equals

If you want to test equality of two objects, you have the following methods

* **assertEquals(expected, actual)**

It will return true if:**expected.equals( actual )**returns true.

#### Assert Array Equals

If you want to test equality of arrays, you have the following methods as given below:

* **assertArrayEquals(expected, actual)**

Above method must be used if arrays have the same length, for each valid value for **i**, you can check it as given below:

* **assertEquals(expected[i],actual[i])**
* **assertArrayEquals(expected[i],actual[i])**

#### Fail Message

If you want to throw any assertion error, you have **fail()** that always results in a fail verdict.

* **Fail(message);**

You can have assertion method with an additional **String**parameter as the first parameter. This string will be appended in the failure message if the assertion fails. E.g. **fail( message )**can be written as

* **assertEquals( message, expected, actual)**

## JUnit assertEquals

You have **assertEquals(a,b)**which relies on the**equals()**method of the Object class.

* Here it will be evaluated as **a.equals( b ).**
* Here the class under test is used to determine a suitable equality relation.
* If a class does not override the **equals()** method of **Object**class, itwill get the default behaviour of **equals()**method, i.e. object identity.

If **a** and **b** are primitives such as **byte**, **int**, **boolean**, etc. then the following will be done for assertEquals(a,b) :

**a** and **b** will be converted to their equivalent wrapper object type (**Byte,Integer**, **Boolean**, etc.), and then **a.equals( b )** will be evaluated.

For Example: Consider below-mentioned strings having same values, let's test it using assertTrue

String obj1="Junit";

String obj2="Junit";

assertEquals(obj1,obj2);

Above assert statement will return true as obj1.equals(obj2) returns true.

## Floating point assertions

When you want to compare floating point types (e.g. **double**or**float**), you need an additional required parameter **delta** to avoid problems with round-off errors while doing floating point comparisons.

The assertion evaluates as given below:

* **Math.abs( expected – actual ) <= delta**

For example:

**assertEquals( aDoubleValue, anotherDoubleValue, 0.001 )**

## JUnit Assert Example

Below example demonstrates how to assert a condition using JUnit assert methods.

Let's create a simple test class named **Junit4AssertionTest.java**and a test runner class **TestRunner.java**.

You will create few variables and important assert statements in JUnit.

In this example, you will execute our test class using TestRunner.java

**Step 1)** Lets create a class covering all important assert statement methods in junit:

**Junit4AssertionTest.java**

package guru99.junit;

import static org.junit.Assert.\*;

import org.junit.Test;

public class Junit4AssertionTest {

@Test

public void testAssert(){

//Variable declaration

String string1="Junit";

String string2="Junit";

String string3="test";

String string4="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(string3, string4);

assertNotSame(string1, string3);

assertNotNull(string1);

assertNull(string5);

assertTrue(variable1<variable2);

assertArrayEquals(airethematicArrary1, airethematicArrary2);

}

}

**Step 2)** You need to create a test runner class to execute above class:

**TestRunner.java**

package guru99.junit;

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(Junit4AssertionTest.class);

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

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

}

System.out.println("Result=="+result.wasSuccessful());

}

}

**Step 3)** Lets analyse expected output step by step:

Consider all assert statements one by one:

1. **assertEquals(string1,string2);**

Now compare string1=" Junit" with string2=" Junit" with equals method of object class. Replacing assertEquals method from java.lang.Object.equals() method :

string1.equals(string2)=> returns true

So assertEquals(string1,string2) will return **true**.

1. **assertSame(string3, string4);**

"assertSame()" functionality is to check that the two objects refer to the same object.

Since string3="test" and string4="test" means both string3 and string4 are of the same type so assertSame(string3, string4) will return **true**.

1. **assertNotSame(string1, string3);**

"assertNotSame()" functionality is to check that the two objects do not refer to the same object.

Since string1="Junit" and string3="test" means both string1 and string3 are of different types, so assertNotSame(string1, string3) will return **true**.

1. **assertNotNull(string1);**

"assertNotNull()" functionality is to check that an object is not null.

Since string1= "Junit" which is a non-null value so assertNotNull(string1) will return **true**.

1. **assertNull(string5);**

"assertNull()" functionality is to check that an object is null.

Since string5= null which is a null value so assertNull(string5) will return **true**.

1. **assertTrue(variable1<variable2);**

"assertTrue()" functionality is to check that a condition is true.

Since variable1=1 and variable2=2, which shows that variable1<variable2 condition is true so assertTrue(variable1<variable2) will return **true**.

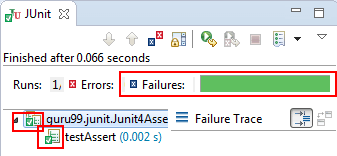
1. **assertArrayEquals(airethematicArrary1, airethematicArrary2);**

"assertArrayEquals()" functionality is to check that the expected array and the resulted array are equal. The type of Array might be int, long, short, char, byte or java.lang.Object.

Since airethematicArrary1 = { 1, 2, 3 } and airethematicArrary2 = { 1, 2, 3 } which shows both the arrays are equal so assertArrayEquals(airethematicArrary1, airethematicArrary2) will return **true**

Since all seven assert statements of **Junit4AssertionTest.java** class returns true, therefore when you execute the test assert class, it will return a successful test. (see the output below)

**Step 4)** Right click on Junit4AssertionTest.java and click on runAs->JUnit. You will see the output as given below:

[](https://cdn.guru99.com/images/junit/Junit-Assert.png)

Above output shows a successful test result as expected.

# Create JUnit Test Suite with Example: @RunWith @SuiteClasses

In Junit, test suite allows us to aggregate all test cases from multiple classes in one place and run it together.

To run the suite test, you need to annotate a class using below-mentioned annotations:

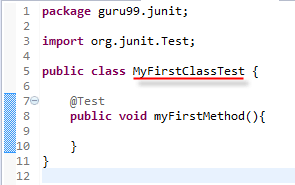
1. @Runwith(Suite.class)
2. @SuiteClasses(test1.class,test2.class……) or

@Suite.SuiteClasses ({test1.class, test2.class……})

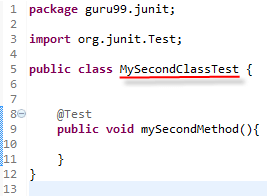
With above annotations, all the test classes in the suite will start executing one by one.

#### Steps to create Test Suite and Test Runner

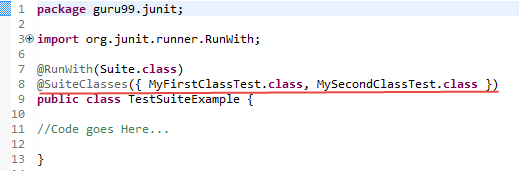
**Step 1)** Create a simple test class (e.g. MyFirstClassTest) and add a method annotated with @test.

[](https://cdn.guru99.com/images/junit/052416_0732_CreateJUnit1.png)

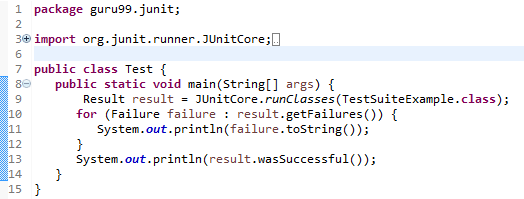
**Step 2)** Create another test class to add (e.g. MySecondClassTest) and create a method annotated with @test.

[](https://cdn.guru99.com/images/junit/052416_0732_CreateJUnit2.png)

**Step 3)** To create a testSuite you need to first annotate the class with @RunWith(Suite.class) and @SuiteClasses(class1.class2…..).

[](https://cdn.guru99.com/images/junit/052416_0732_CreateJUnit3.png)

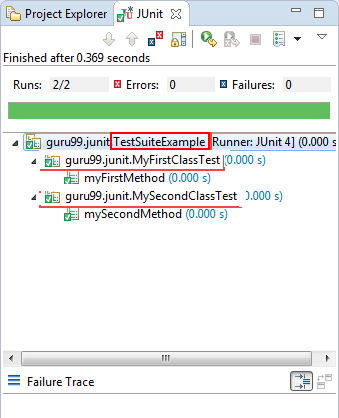
**Step 4)** Create a Test Runner class to run our test suite as given below;

[](https://cdn.guru99.com/images/junit/052416_0732_CreateJUnit4.png)

**Code Explanation:**

* **Code Line 8:**Declaring the main method of the class test which will run our JUnit test.
* **Code Line 9:**Executing test cases using JunitCore.runclasses which takes the testclass name as a parameter (In the example above, you are using TestSuiteExample.class shown in step 3).
* **Code Line 11:**Processing the result using for loop and printing out failed result.
* **Code Line 13:**Printing out the successful result.

**Output:**Here is the output which shows successful test with no failure trace as given below:

[](https://cdn.guru99.com/images/junit/052416_0732_CreateJUnit5.png)

#### JUnit Test Suite Example

Consider a more complex example

**JunitTest.java**

JunitTest.java is a simple class annotated with **@RunWith** and **@Suite** annotations. You can list out number of .classes in the suite as parameters as given below:

package guru99.junit;

import org.junit.runner.RunWith;

import org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({

SuiteTest1.class,

SuiteTest2.class,

})

public class JunitTest {

// This class remains empty, it is used only as a holder for the above annotations

}

**SuiteTest1.java**

**SuiteTest1.java** is a test class having a test method to print out a message as given below. You will use this class as a suite in above mentioned class.

package guru99.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class SuiteTest1 {

public String message = "Saurabh";

JUnitMessagejunitMessage = new JUnitMessage(message);

@Test(expected = ArithmeticException.class)

public void testJUnitMessage() {

System.out.println("Junit Message is printing ");

junitMessage.printMessage();

}

@Test

public void testJUnitHiMessage() {

message = "Hi!" + message;

System.out.println("Junit Hi Message is printing ");

assertEquals(message, junitMessage.printHiMessage());

System.out.println("Suite Test 2 is successful " + message);

}

}

**SuiteTest2.java**

**SuiteTest2.java** is another test class similar to **SuiteTest1.java** having a test method to print out a message as given below. You will use this class as suite in **JunitTest.java**.

package guru99.junit;

import org.junit.Assert;

import org.junit.Test;

public class SuiteTest2 {

/\*\* Test of setName() method, of class Value \*/

@Test

public void createAndSetName() {

Value v1 = new Value();

v1.setName("Y");

String expected = "Y";

String actual = v1.getName();

Assert.assertEquals(expected, actual);

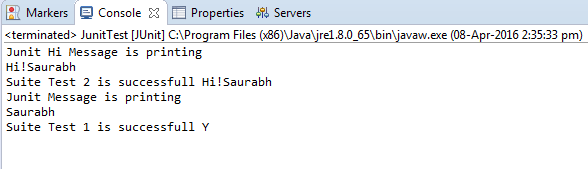
System.out.println("Suite Test 1 is successful " + actual);

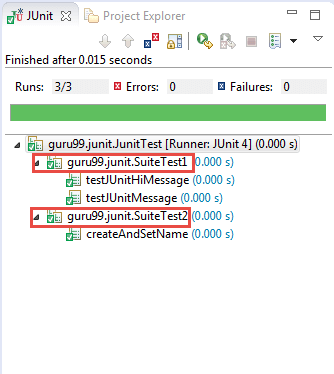
}

}

**Output**

After executing **JunitTest**.java which contains a suite having **test1.java** and **test2.java**, you will get below output:

[](https://cdn.guru99.com/images/junit/052416_0732_CreateJUnit6.png)

[](https://cdn.guru99.com/images/junit/052416_0732_CreateJUnit7.png)

**Summary**

In this tutorial, you have learned basics of test harness and test suites in details with an example.

* Test harness or automation[Testing](https://www.guru99.com/software-testing.html)is a software or a collection of software, which allows a user to test data with multiple inputs and control the execution
* Test harness actually enables a test framework that does all the work of executing tests using a test library and generating a test report
* In Junit, test suite allows us to aggregate all test cases of many classes in one place and run it together.

# JUnit @Ignore Test Annotation

Sometimes you may require not to execute a method/code or[Test Case](https://www.guru99.com/test-case.html)because coding is not done fully. For that particular test, JUnit provides **@Ignore** annotation to skip the test.

## What is JUnit @Ignore test annotation

* The @Ignore test annotation is used to ignore particular tests or group of tests in order to skip the build failure.
* **@Ignore** annotation can be used in two scenarios as given below:

1. If you want to ignore a test method, use @Ignore along with @Test annotation.
2. If you want to ignore all the tests of class, use @Ignore annotation at the class level.

* You can provide the reason for disabling a test in the optional parameter provided by @Ignore annotation.
* It will help other developers working on the same piece of code, to understand "why a particular test is disabled?" When the issue of that particular test is fixed, you can simply enable it by removing **@Ignore annotation**.

#### Junit Test Example - Ignore

As discussed in above definition, you can use @Ignore annotation to ignore a test or group of the test.

Let's understand it using simple example and in below given scenarios:

1. Creating a simple test class without ignoring a test.
2. Ignore a test method using @Ignore annotation.
3. Ignore a test method using @Ignore annotation with proper reason.
4. Ignore all test method using @Ignore annotation.

#### Creating a simple test class without ignoring a test

Let's create a simple[Java](https://www.guru99.com/java-tutorial.html)class which prints two types of message.

* First method prints a simple message and
* The second method prints a "hi" message

**JUnitMessage.java**

package guru99.junit;

public class JUnitMessage {

private String message;

public JUnitMessage(String message) {

this.message = message;

}

public String printMessage(){

System.out.println(message);

return message;

}

public String printHiMessage(){

message="Hi!"+ message;

System.out.println(message);

return message;

}

}

**JunitTestExample.java**

Let's create a JUnit test class to test JUnitMessage.java.

In this JUnit test class,

* First test, named "testJUnitMessage()" tests "printMessage()" method of above class.
* Similarly the second test, named "testJUnitHiMessage" tests "testJUnitHiMessage" of above class.

package guru99.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class JunitTestExample {

public String message = "Guru99";

JUnitMessagejunitMessage = new JUnitMessage(message);

@Test

public void testJUnitMessage() {

System.out.println("Junit Message is printing");

assertEquals(message, junitMessage.printMessage());

}

@Test

public void testJUnitHiMessage() {

message="Hi!" +message;

System.out.println("Junit Hi Message is printing ");

assertEquals(message, junitMessage.printHiMessage());

}

}

**TestRunner.java**

Let's create a test runner class to execute JunitTestExample.java

package guru99.junit;

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(JunitTestExample.class);

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

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

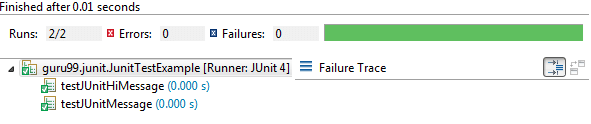
}

System.out.println("Result=="+result.wasSuccessful());

}

}

**Output:**

[](https://cdn.guru99.com/images/junit/052416_0743_JUnitIgnore1.png)

**Print statement on console:**

Junit Hi Message is printing

Hi!Guru99

Junit Message is printing

Guru99

#### Ignore a test method using @Ignore annotation

Let's create ignore test to disable a test in above example. For this, you need to use @Ignore in the method, you want to skip.

Let's do it for testJUnitMessage() of JunitTestExample.java

**JunitTestExample.java**

package guru99.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Ignore;

import org.junit.Test;

public class JunitTestExample {

public String message = "Guru99";

JUnitMessagejunitMessage = new JUnitMessage(message);

@Ignore

@Test

public void testJUnitMessage() {

System.out.println("Junit Message is printing ");

assertEquals(message, junitMessage.printMessage());

}

@Test

public void testJUnitHiMessage() {

message="Hi!" +message;

System.out.println("Junit Hi Message is printing ");

assertEquals(message, junitMessage.printHiMessage());

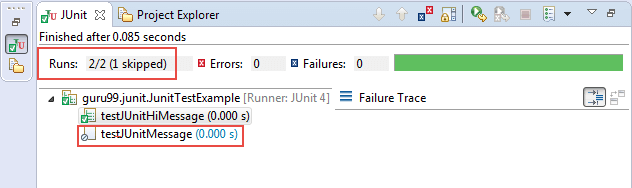
}

}

**Output:**

Let's execute and verify the output of above example.

Below output shows that one test is skipped (disabled), see as marked below:

[](https://cdn.guru99.com/images/junit/052416_0743_JUnitIgnore2.png)

**Print statement on console:**

Junit Hi Message is printing

Hi!Guru99

#### Using @ ignore annotation with Condition

Let's take the example of how to ignore a test and define the reason for ignoring along with it. As discussed above, to provide a reason you have one optional parameter in @Ignore annotation where you can provide the reason statement.

**JunitTestExample.java**

package guru99.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Ignore;

import org.junit.Test;

public class JunitTestExample {

public String message = "Guru99";

JUnitMessagejunitMessage = new JUnitMessage(message);

@Ignore("not yet ready , Please ignore.")

@Test

public void testJUnitMessage() {

System.out.println("Junit Message is printing ");

assertEquals(message, junitMessage.printMessage());

}

@Test

public void testJUnitHiMessage() {

message="Hi!" +message;

System.out.println("Junit Hi Message is printing ");

assertEquals(message, junitMessage.printHiMessage());

}

}

**Output:**

Same as above.

#### Ignore all test methods using @Ignore annotation.

As discussed above to ignore all the tests in class, you need to use @Ignore annotation at the class level.

Let's modify above example to understand how to ignore all the tests:

package guru99.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Ignore;

import org.junit.Test;

@Ignore

public class JunitTestExample {

public String message = "Guru99";

JUnitMessagejunitMessage = new JUnitMessage(message);

@Test

public void testJUnitMessage() {

System.out.println("Junit Message is printing ");

assertEquals(message, junitMessage.printMessage());

}

@Test

public void testJUnitHiMessage() {

message="Hi!" +message;

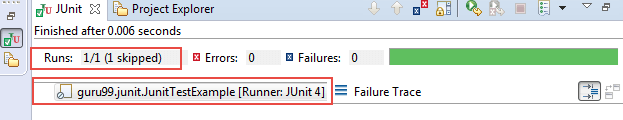
System.out.println("Junit Hi Message is printing ");

assertEquals(message, junitMessage.printHiMessage());

}

}

**Output :**

[](https://cdn.guru99.com/images/junit/052416_0743_JUnitIgnore3.png)

**Print statement on console:**

As both the tests skipped by using @Ignore at class level so no statement would be printed on the console.

**Summary:**

In this tutorial, you learned how to ignore a single test, group of test or all tests by using @Ignore annotation.

**@Ignore** annotation can be used in two scenarios as given below:

1. If you want to ignore a test method, use @Ignore along with @Test annotation.
2. If you want to ignore all the tests of class, use @Ignore annotation at the class level.

You also learned how to provide a statement to make understand other developer, why a particular test is disabled.

# JUnit Expected Exception Test: @Test(expected)

* JUnit provides the facility to trace the exception and also to check whether the code is throwing expected exception or not.
* Junit4 provides an easy and readable way for exception testing, you can use
  + Optional parameter (expected) of @test annotation and
  + To trace the information ,"fail()" can be used
* While Testing exception, you need to ensure that exception class you are providing in that optional parameter of **@test annotation** is the same. This is because you are expecting an exception from the method you are Unit Testing, otherwise our JUnit test would fail.

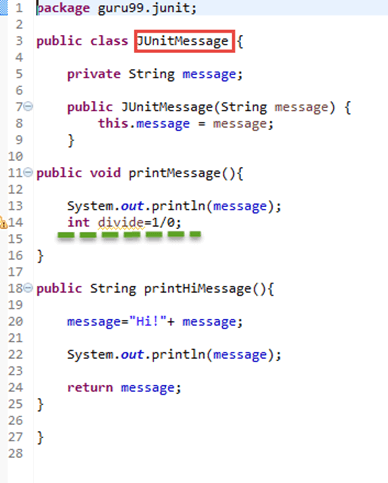
#### Expected Parameter

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

By using "expected" parameter, you can specify the exception name our test may throw. In above example, you are using "**IllegalArgumentException"**which will be thrown by the test if a developer uses an argument which is not permitted.

#### Example using @test(expected)

Let's understand exception testing by creating a Java class with a method throwing an exception. You will handle it and test it in a test class. Consider JUnitMessage.java having a method which simply do a mathematical operation based on input received by the user. If any illegal argument would be entered, it will throw "ArithmeticException**"**. See below:

[](https://cdn.guru99.com/images/junit/052416_1148_JUnitExcept1.png)

package guru99.junit;

public class JUnitMessage{

private String message;

public JUnitMessage(String message) {

this.message = message;

}

public void printMessage(){

System.out.println(message);

int divide=1/0;

}

public String printHiMessage(){

message="Hi!" + message;

System.out.println(message);

return message;

}

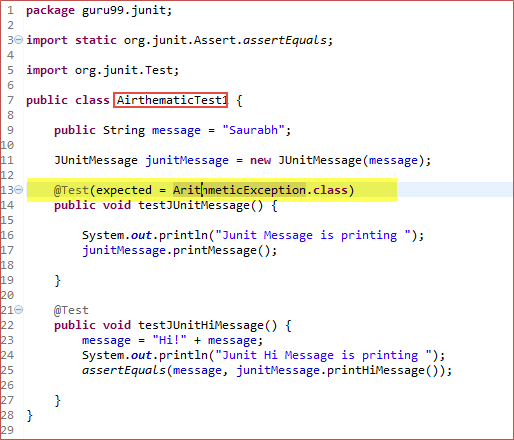
}

**Code Explanation:**

* **Code Line 7:**Creating a parameterized constructor with field initialization.
* **Code Line 11-14:**Creating a method for mathematical operation.
* **Code Line 18:**Creating another method to print a message.
* **Code Line 20:**Creating a new string to print a message.
* **Code Line 21:**Printing new message created in line 20.

Let's create a test class for above java class to verify exception.

See below test class to unit test exception (ArithmeticException here) throwing from above java class:

**AirthematicTest.java**[](https://cdn.guru99.com/images/junit/052416_1148_JUnitExcept2.png)

package guru99.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class AirthematicTest {

public String message = "Saurabh";

JUnitMessagejunitMessage = new JUnitMessage(message);

@Test(expected = ArithmeticException.class)

public void testJUnitMessage(){

System.out.println("Junit Message is printing ");

junitMessage.printMessage();

}

@Test

public void testJUnitHiMessage(){

message="Hi!" + message;

System.out.println("Junit Message is printing ");

assertEquals(message, junitMessage.printMessage());

}

}

**Code Explanation:**

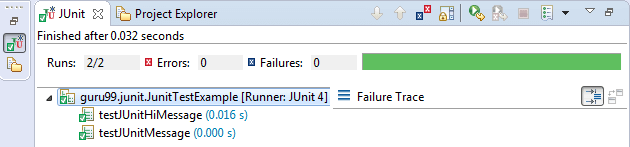
* **Code Line 13:**Using @Test annotation to create our test. As you execute above classes method, it will invoke a mathematical operation. Here Arithmetic Exception is expected, so you are listing it out as a parameter in @Test.
* **Code Line 17:**Invoking printMessage() JUnitMessage.java
* **Code Line 18:**Creating another method to print HI message.

If you execute this test class, the test method is executed with each defined parameter. In the above example, the test method is executed five times.

Let's execute it and verify the result. See below the test runner class to execute **JunitTestExample.java**

**Output:**

Here is the output which shows successful test with no failure trace as given below:

[](https://cdn.guru99.com/images/junit/052416_1148_JUnitExcept3.png)

**Summary:**

* Exception testing is a special feature introduced in JUnit4. In this tutorial, you have learned how to test exception in JUnit using @test(excepted)
* Junit provides the facility to trace the exception and also to check whether the code is throwing exception or not
* For exception testing, you can use
  + Optional parameter (expected) of @test annotation and
  + To trace the information ,"fail()" can be used

# JUnit ErrorCollector @Rule with Example

* In a normal scenario, whenever you identify any error during test execution, you would stop the test, fix the error and re-run the test.
* But JUnit has a slightly different approach. With JUnit error collector, you can still continue with the test execution even after an issue is found or test fails. Error collector collects all error objects and reports it only once after the test execution is over.

#### Why use Error Collector?

* While writing a test script, you want to execute all the tests even if any line of code fails due to network failure, assertion failure, or any other reason. In that situation, you can still continue executing test script using a special feature provided by JUnit known as "error collector."
* For this, JUnit uses **@Rule annotation** which is used to create an object of error collector. Once the object for error collector is created, you can easily add all the errors into the object using method **addError (Throwable error).**As you know, that **Throwable** is the super class of **Exception** and **Error** class in Java. When you add errors in this way, these errors will be logged in JUnit test result .
* The benefit of adding all errors in an Error Collector is that you can verify all the errors at once. Also, if the script fails in the middle, it can still continue executing it

**Note**: In the case of using simple assert or try/catch block , using error collector method won't be possible.

**Sample code**

To understand more on Error Collector, see below code example which demonstrates how to create an Error Collector object and add all the errors in that object to track the issue :

package guru99.junit;

import org.junit.Rule;

import org.junit.Test;

import org.junit.rules.ErrorCollector;

public class ErrorCollectorExample {

@Rule

public ErrorCollector collector = new ErrorCollector();

@Test

public void example() {

collector.addError(new Throwable("There is an error in first line"));

collector.addError(new Throwable("There is an error in second line"));

collector.checkThat(getResults(),

not(containsString("here is an error")));

// all lines of code will execute and at the end a combined failure will

be logged in.

}

}

#### What is @Rule in jUnit?

JUnit provides special kind of handling of tests,[Test Case](https://www.guru99.com/test-case.html)or test suite by using **@rule annotation**. Using @rule, you can easily add or redefine the behaviour of the test.

There are several built-in rules provided by JUnit API that a tester can use, or even you can write our own rule.

See below line of code, which shows how to use @rule annotation along with Error Collector:

@Rule

public ErrorCollector collector= new ErrorCollector();

#### Example using ErrorCollector

To understand error collector, let's create a class and a rule to collect all the errors. You will add all the errors using addError(throwable) here.

See below code which simply creates a rule which is nothing but creating "Error Collector object." Which is further used to add all the errors in order to report the issue at the end:

**ErrorCollectorExample.java**

package guru99.junit;

import org.junit.Assert;

import org.junit.Rule;

import org.junit.Test;

import org.junit.rules.ErrorCollector;

public class ErrorCollectorExample {

@Rule

public ErrorCollector collector = new ErrorCollector();

@Test

public void example() {

collector.addError(new Throwable("There is an error in first line"));

collector.addError(new Throwable("There is an error in second line"));

System.out.println("Hello");

try {

Assert.assertTrue("A " == "B");

} catch (Throwable t) {

collector.addError(t);

}

System.out.println("World!!!!");

}

}

**TestRunner.java**

Let's add above test class in a test runner and execute it to collect all errors. See below code:

package guru99.junit;

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(ErrorCollectorExample.class);

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

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

}

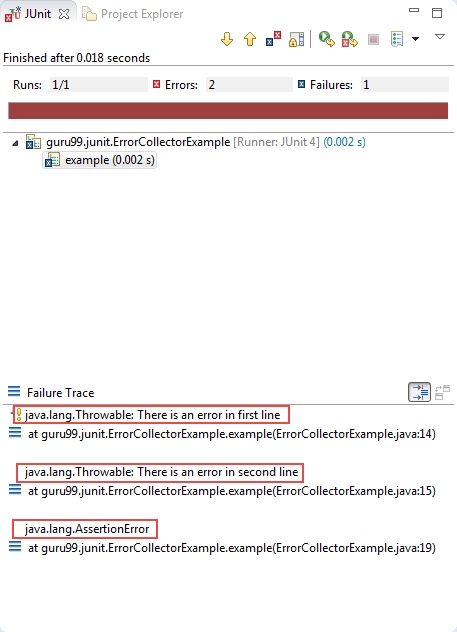
System.out.println("Result=="+result.wasSuccessful());

}

}

**Output:**

See the failure trace which traces all the errors in one place:

[](https://cdn.guru99.com/images/junit/052416_1156_JUnitErrorC1.png)

#### Benefits of JUnit ErrorCollector

You can use JUnit assertion for functional or GUI validation e.g.

1. assertEquals(String message, Object expected, Object actual) which compare that two objects are equals.
2. Similarly, assertTrue(Boolean condition) asserts that a condition is true.

Using assertion, validation test becomes easy. But one major issue is that test execution will stop even if a single assertion fails.

Test continuity and recovery handling is crucial to test automation success. Error Collector is the best way to handle such kind of scenarios.

**Summary**:

* Junit error collector allows a test to continue even after the first issue is found and test fails at the end
* Error collector collects all error objects and reports it only, after all, the test execution over
* The benefit of adding all errors in an Error Collector is that you can verify all the errors at once
* Error collector simply adds errors using method addError(throwable err) provided by ErrorCollector.java.

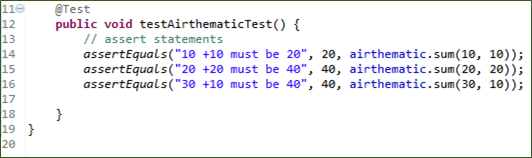
# JUnit Parameterized Test

#### What is Parameterized Test in Junit?

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.

Consider a simple test to sum different numbers. The code may look like -

[](https://cdn.guru99.com/images/junit/052416_1228_JUnitParame1.png)

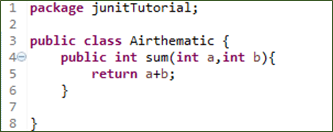
The approach above leads to lot of redundancy.

We need a simple approach and. Using parameterized test you can just add a method to input 10 data inputs and your test will run 10 times automatically.

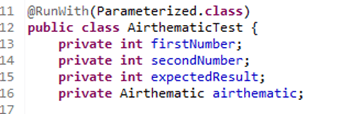
#### Steps to create a Parameterized JUnit test

Following code shows an example for a parameterized test. It tests sum() method of the Arithmetic class :

**Step 1)**Create a class. In this example, we are going to input two numbers by using sum (int,int) method which will return the sum of given numbers

[](https://cdn.guru99.com/images/junit/052416_1228_JUnitParame2.png)

**Step 2)** Create a parameterized test class

[](https://cdn.guru99.com/images/junit/052416_1228_JUnitParame3.png)

Code Explanation

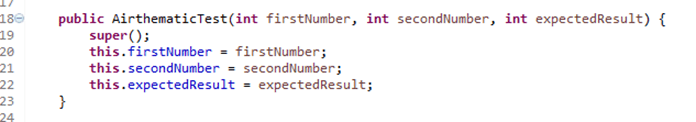
* **Code Line 11:** Annotate your test class using @runWith(Parameterized.class).
* **Code Line 13:** Declaring the variable 'firstNumber' as private and type as int.
* **Code Line 14:** Declaring the variable 'secondNumber'as private and type as int.
* **Code Line 15:** Declaring the variable 'expectedResult'as private and type as int.
* **Code Line 16:** Declaring the variable 'airthematic'as private and type as Airthematic.

**@RunWith(class\_name.class):** @RunWith annotation is used to specify its runner class name. If we don't specify any type as a parameter, the runtime will choose BlockJunit4ClassRunner by default.

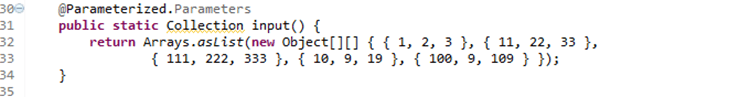
This class is responsible for tests to run with a new test instance. It is responsible for invoking JUnit lifecycle methods such as setup(associate resources) and teardown(release resources).

To parameterize you need to annotate using @RunWith and pass required .class to be tested

**Step 3)** Create a constructor that stores the test data. It stores 3 variables

[](https://cdn.guru99.com/images/junit/052416_1228_JUnitParame4.png)

**Step 4)** Create a static method that generates and returns test data.

[](https://cdn.guru99.com/images/junit/052416_1228_JUnitParame5.png)

**Code Line 32,33:** Creating a two-dimensional array (providing input parameters for addition). Using asList method we convert the data into a List type. Since, the return type of method input is collection.

**Code Line 30:** Using @Parameters annotation to create a set of input data to run our test.

The static method identified by @Parameters annotation returns a Collection where each entry in the Collection will be the input data for one iteration of the test.

Consider the elemenent

{1,2,3}

Here

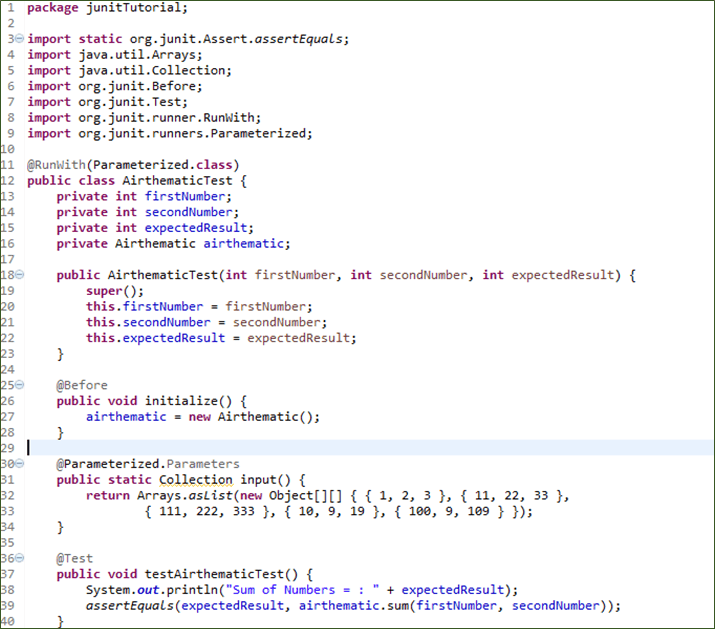
firstNumber =1

secondNumber=2

expectedResult=3

Here each array element will be passed to the constructor, one at a time as the class is instantiated multiple times.

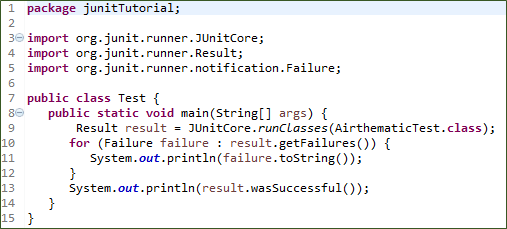
**Step 5)** The complete code

[](https://cdn.guru99.com/images/junit/052416_1228_JUnitParame6.png)

Code Explanation:

* Code Line 25: Using @Before annotation to setup the resources (Airthematic.class here). The @Before annotation is used here to run before each test case. It contains precondition of the test.
* Code Line 36: Using @Test annotation to create our test.
* Code Line 39: Creating an assert statement to check whether our sum is equivalent to what we expected.

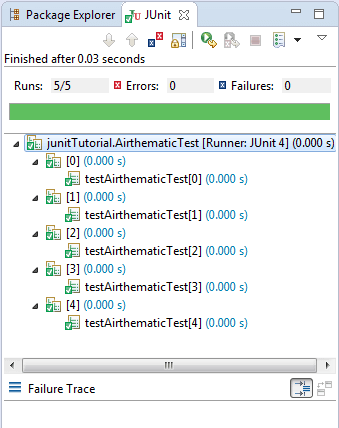
**Step 6)** Create a test runner class to run parameterized test:

[](https://cdn.guru99.com/images/junit/052416_1228_JUnitParame7.png)

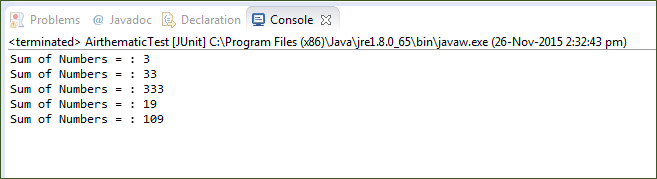
Code Explanation:

* **Code Line 8:** Declaring the main method of the class test which will run our JUnit test.
* **Code Line 9:** Executing test cases using JunitCore.runclasses, it will take the test class name as a parameter (In our example we are using Airthematic.class).
* **Code Line 11:** Processing the result using for loop and printing out failed result.
* **Code Line 13:** Printing out the successful result.

**Output:**Here is the output which shows successful test with no failure trace as given below:

[](https://cdn.guru99.com/images/junit/052416_1228_JUnitParame8.png)

See the result on console,which shows addition of two numbers :-

[](https://cdn.guru99.com/images/junit/052416_1228_JUnitParame9.png)

**Summary**:

* Parameterized test enables developer to execute the same test over and over again using different values.
* Important annotations to be used during parameterization
* @RunWith
* @Parameters

# JUnit in Parallel

* To speed up running tests, parallel execution is commonly used strategy. When programmatically runnning JUnit tests via JunitCore, the code have to incorporate with the parallel execution logic as well.
* JUnit provides an experimental class ParallelComputer. It has various ways where we can run our test cases in parallel.
* There is another way i.e. to use a surefire plugin of Maven that helps us to run the test cases in parallel. We will also see how we can achieve it. But for this example, we will concentrate on the use of the class ParallelComputer of JUnit.

#### Technologies Used

We will be using the following technologies while building an example. Here we are taking the latest version of each technology.

* **Java 1.8** – We will be using the latest version of Java. There is no specific feature of Java 1.8 that has been used.
* **JUnit 4.12** – This is the main testing framework that we will be using.
* **Maven** – This is the build and dependency tool for this example.
* **Eclipse** – IDE for writing the code.

For links to the latest versions and their websites visit the [**reference**](https://examples.javacodegeeks.com/core-java/junit/junit-run-tests-in-parallel/#references) section of the tutorial. Now, we have defined the technologies to be used. Let’s start the setup of our example.

#### Project Setup

As you already know, we are using the Eclipse. So start by opening the Eclipse.  
Click on the **File -> New -> Maven Project**.  
You will see the first screen as shown here. Simply check the first checkbox and click on the **Next** button.

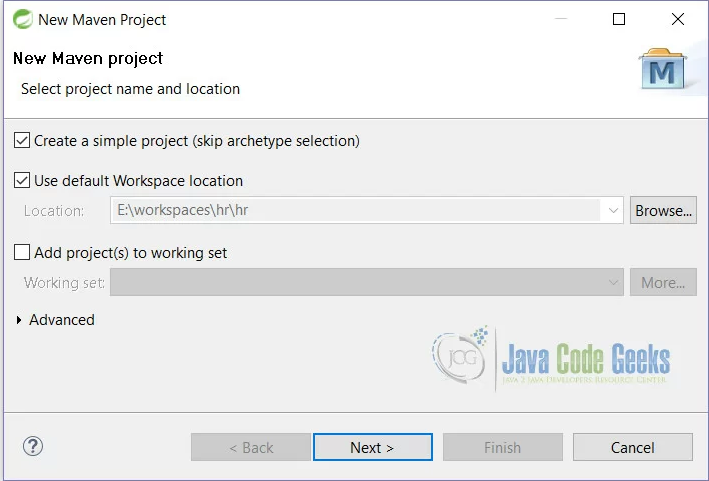


Figure 1: JUnit Parallel Test Setup 1

On this screen, fill all the details as shown here and click on the **Finish** button.

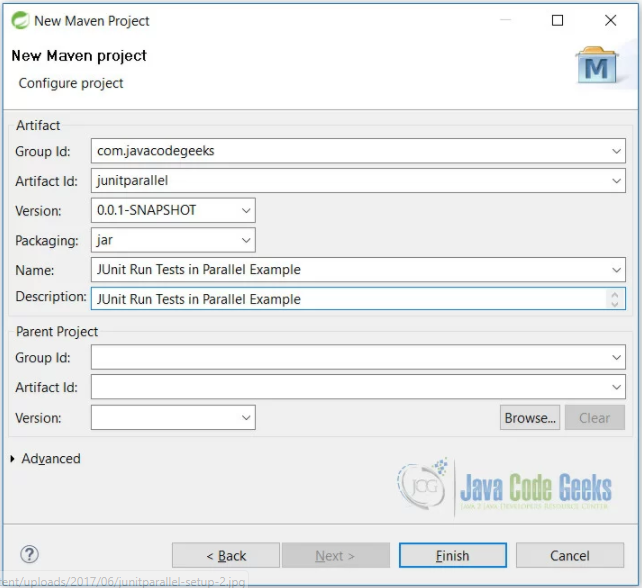


Figure 2: JUnit Parallel Test Setup 2

With the click of the Finish button, we are ready with the blank Maven project. To continue we have to do some changes in pom.xml file. See the next section.

#### JUnit Parallel Test

Now when we are ready with the blank Maven project, we need to do some changes the pom.xml file.

#### XML Configurations

**pom.xml**

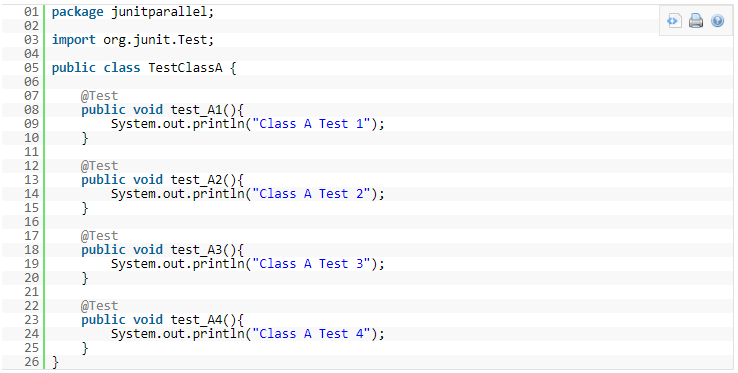
****

**Line 4:** will fetch the JUnit jar from maven repository.  
**Line 18, 19:** will simply make the project compile with a Java 1.8 version.

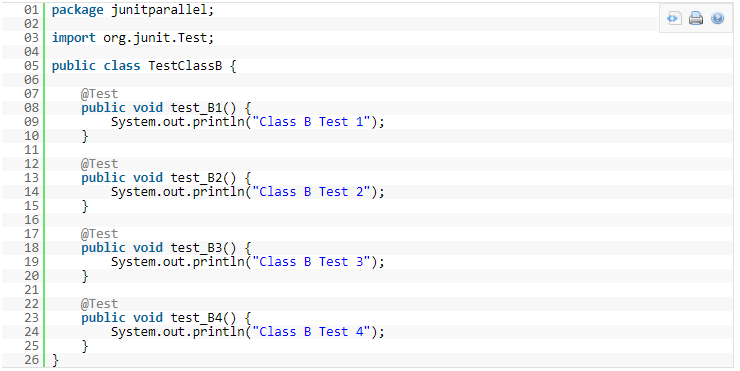
#### Test Classes

First, we will create 2 classes which have test cases that we will run.

**TestClassA.java:**

**

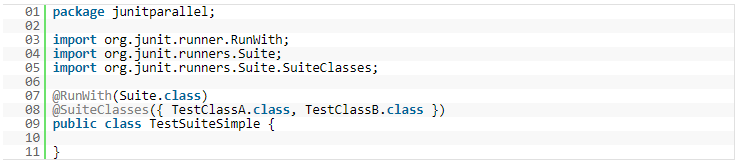
**TestClassB.java :**

**

These are simple classes with methods having print statements. We will see further that when we run these classes without parallel functionality, then they will run one by one.

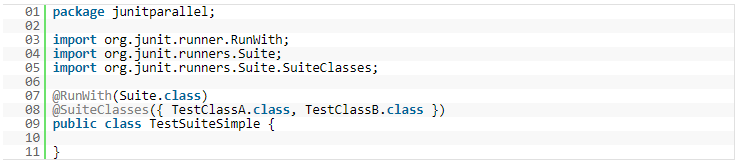
Let’s create a test suite to run the above test cases.

**TestSuiteSimple.java :**



So, once you run the above class you will see the following output in the console.

**Output :**



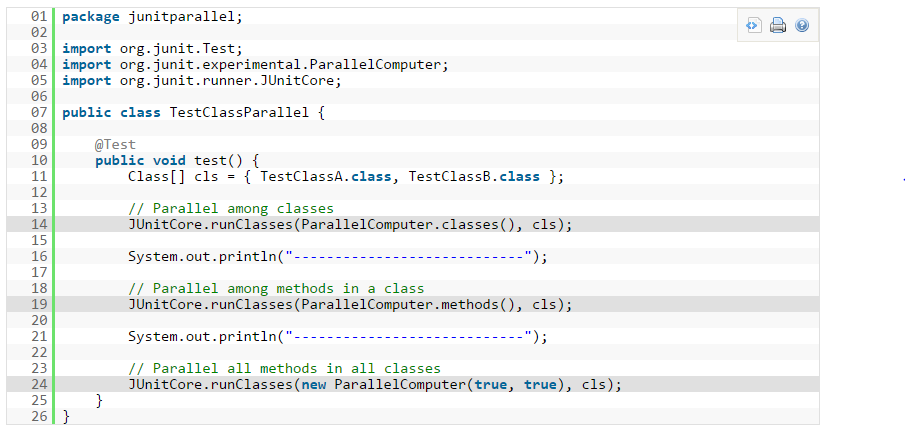
As you can see, all test cases have been executed one by one. All test cases of TestClassA were executed first and then the test cases of the TestClassB.

Let’s see the parallel case now.

#### JUnit Parallel Tests

Now, we will create a class that will run the test cases parallel.  
Here we will be using the experimental ParallelComputer class of the JUnit. We will see that ParallelComputer has different ways of running.

**TestClassParallel.java**



**Line 14:** will run the classes in parallel while running test cases.  
**Line 19:** will run the methods of same classes parallel.  
**Line 24:** will run all methods in all classes in parallel. Here the first parameter will let us know that we need to run classes in parallel, while the second parameter specifies the methods to run in parallel.

Once we run the above class we will see the following output.

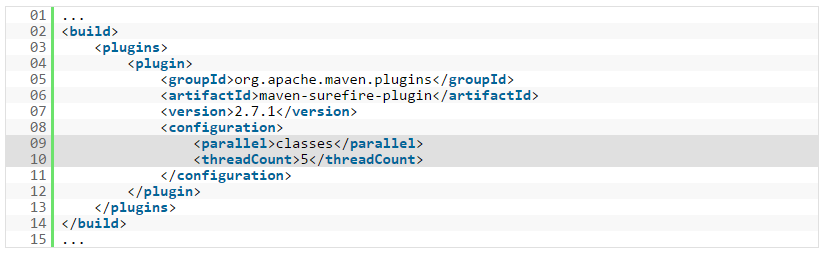
**Output**



From the above output, we can see how both classes run in parallel.

#### Using the surefire plugin

* If you do not want to go with the ParallelComputer class then you can use the surefire plugin of maven.
* Simply put the below line in the pom.xml file.



**Line 9:** specifies that we want to run our classes parallel while testing. We can also specify the methods instead of classes if we want to run methods parallel and not classes. See below:



**Line 10:** specifies the number of threads to be generated while running our test cases. This facility is not available when we use ParalleComputer class.

This way you do not have to handle anything and is automatically handled by the surefire plugin.

#### Conclusion

There are 2 ways to run our test cases in parallel.

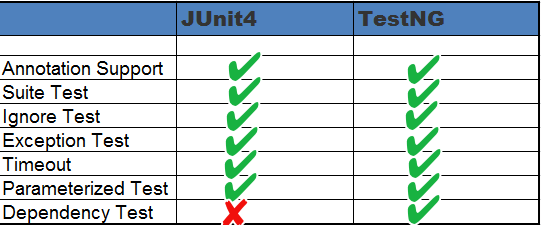
1. To use JUnit ParalelComputer class
2. Use of surefire plugin of Maven

**TestNG Vs JUnit**

* Both TestNGand Junit are[Testing](https://www.guru99.com/software-testing.html)framework used for [Unit Testing](https://www.guru99.com/unit-testing-guide.html). TestNG is similar to JUnit. Few more functionalities are added to it that makes TestNG more powerful than JUnit.
* TestNG is a testing framework inspired by JUnit and NUnit.

#### JUnit4 and TestNG Feature comparison

* Both TestNG and JUnit4 looks similar, except one or two features. Let's have a comparison between the two to quickly decide, which technology is more favourable for Unit Testing. Below table highlights the features supported by both:

[](https://cdn.guru99.com/images/junit/052416_1236_JUnitVsTest1.png)

#### Annotations

* Both JUnit and TestNG uses annotations and almost all the annotations looks similar.
* TestNG uses @BeforeMethod ,@AfterMethod similar to @Before ,@After in JUnit4.
* Both TestNG and Junit4 uses @Test(timeout = 1000) for timeout .Check the table below for more details-

|  |  |  |  |
| --- | --- | --- | --- |
| S.N. | Description | TestNG | JUnit 4 |
| 1 | Test annotation | @Test | @Test |
| 2 | Executes before the first test method is invoked in the current class | @BeforeClass | @BeforeClass |
| 3 | Executes after all the test methods in the current class | @AfterClass | @AfterClass |
| 4 | Executes before each test method | @BeforeMethod | @Before |
| 5 | Executes after each test method | @AfterMethod | @After |
| 6 | annotation to ignore a test | @Test(enable=false) | @ignore |
| 7 | annotation for exception | @Test(expectedExceptions = ArithmeticException.class) | @Test(expected = ArithmeticException.class) |
| 8 | timeout | @Test(timeout = 1000) | @Test(timeout = 1000) |
| 9 | Executes before all tests in the suite | @BeforeSuite | n/a |
| 10 | Executes after all tests in the suite | @AfterSuite | n/a |
| 11 | Executes before a test runs | @BeforeTest | n/a |
| 12 | Executes after a test runs | @AfterTest | n/a |
| 13 | Executes before the first test method is invoked that belongs to any of these groups is invoked | @BeforeGroups | n/a |
| 14 | run after the last test method that belongs to any of the groups here | @AfterGroups | n/a |

## Suite Test

Suites are used to execute multiple tests together. Suites can be created using both TestNG and JUnit4. However, suites are more powerful in TestNG as it uses very different method for execution of tests. Let's understand it using code snippet as given below:

#### Using JUnit4

Below class describes use of suite while working with JUnit4:

package guru99.junit;

import org.junit.runner.RunWith;

import org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({

SuiteTest1.class,

SuiteTest2.class,

})

public class JunitTest {

// This class remains empty,it is used only as a holder for the above annotations

}

#### Using TestNG

TestNG uses xml to bundle all tests at one place.Below xml describes use of suite while working with TestNG:

<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >

<suite name="My test suite">

<test name="testing">

<classes>

<class name="com.guru99.SuiteTest1" />

<class name="com.guru99.SuiteTest2" />

</classes>

</test>

</suite>

## Ignore Test

Using both we can skip a test .Let's see it using code example as given below:

#### Using JUnit4

Below code snippet describes use of @ignore annotation while working with JUnit4:

@Ignore

public void method1()

{

System.out.println("Using @Ignore , this execution is ignored");

}

#### Using TestNG

Below code snippet describes use of @Test(enabled=false) annotation while working with TestNG:

@Test(enabled=false)

public void TestWithException()

{

System.out.println("Method should be ignored as it's not ready yet");

}

## Exception Test

Exception testing is available both in TestNG and JUnit4. It is used to check, which exception is thrown from the test?

#### Using JUnit4

Below code snippet describes use of exception test while working with JUnit4:

@Test(expected = ArithmeticException.class)

public void divideByZero()

{

Inti = 1/0;

}

#### Using TestNG

Below code snippet describes use of exception test while working with TestNG:

@Test(expectedExceptions = ArithmeticException.class)

public void divideByZero()

{

Inti = 1/0;

}

## Timeout

This feature is implemented both in TestNg and JUnit4.Timeout is used to terminate a test which takes longer than specified time (in milliseconds).

#### Using JUnit4

Below code snippet describes use of timeout test while working with JUnit4:

@Test(timeout = 1000)

public void method1()

{

while (true);

}

#### Using TestNG

Below code snippet describes use of timeout test while working with TestNG:

@Test(timeOut = 1000)

public void method1()

{

while (true);

}

## Parameterized Test

JUnit provides an easier and readable approach to test known as Parameterized test. Both TestNG and JUnit supports parameterized test but differ in the way they define the parameter value. Let see this one by one.

#### Using JUnit4

The "@RunWith" and "@Parameter" annotations are used to provide parameter value for the unit test. The annotation @Parameters have to return List[] .This parameter will be passed into the class constructor as an argument.

@RunWith(value = Parameterized.class)

public class JunitTest{

privateint number;

public JunitTest6(int number)

{

this.number = number;

}

@Parameters

public static Collection<Object[]> data()

{

Object[][] data = new Object[][] { { 1 }, { 2 }, { 3 }, { 4 } };

returnArrays.asList(data);

}

@Test

public void parameterTest()

{

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

}

}

#### Using TestNG

In TestNG, XML file or "@DataProvider" is used to provide a parameter for testing.

Here @Parameters annotation declared in the method, needs a parameter for testing. The data used as the parameter will provide in TestNG's XML configuration files. By doing this, we can reuse a single[Test Case](https://www.guru99.com/test-case.html)with different data sets, and we can get different results.

public class Test1 {

@Test

@Parameters(value="number")

public void parameterTest(int number)

{

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

}

}

See below xml file to be used for above class:

<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >

<suite name="My test suite">

<test name="testing">

<parameter name="number" value="2"/>

<classes>

<class name="com.guru99.Test1" />

</classes>

</test>

</suite>