# How to Download and Install JUnit in Eclipse

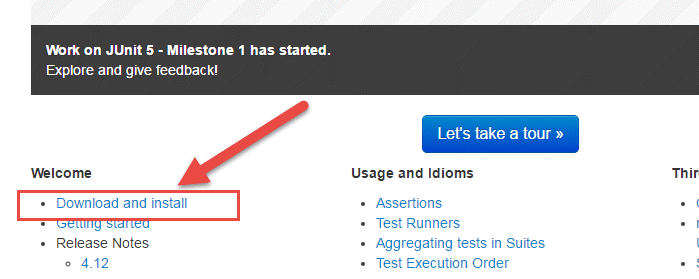
Installing Junit is a 6 part process. It is explained in detailed below-

## PART 1) Install Java

JUnit is a Testing framework used to test Java based application. So before installing JUnit, you need to configure or verify java development kit (JDK) in your machine.

## PART 2) Download JUnit

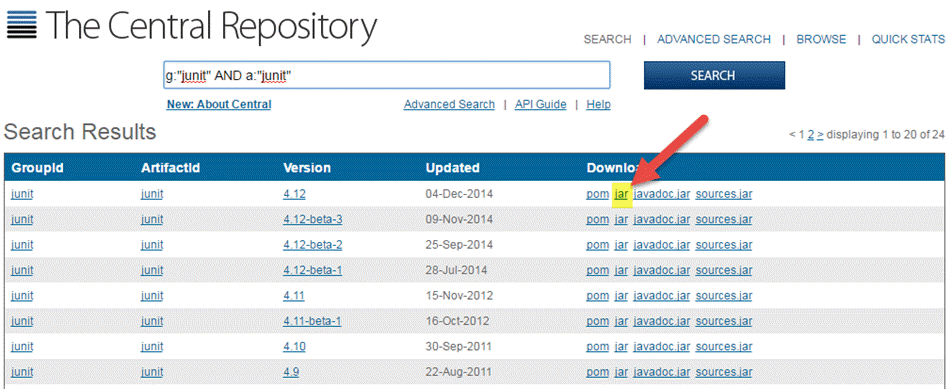
**Step 1)**Visit <https://junit.org/junit4/> and click Download and Install

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

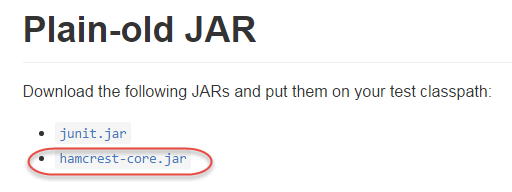
**Step 2)**Click junit.jar

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

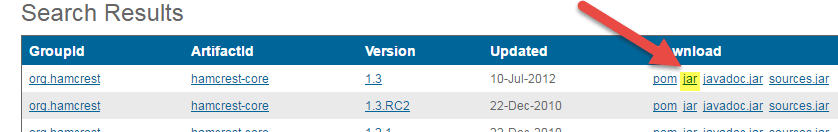
**Step 3)**In the central repository you are shown all versions of Junit that can be downloaded. Usually, you will select the latest version. Click on jar link to download Junit version 4.12 as shown below

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

**Step 5)**Download the Jar

[](https://www.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](https://junit.org/junit5/)

Here is the jars list:

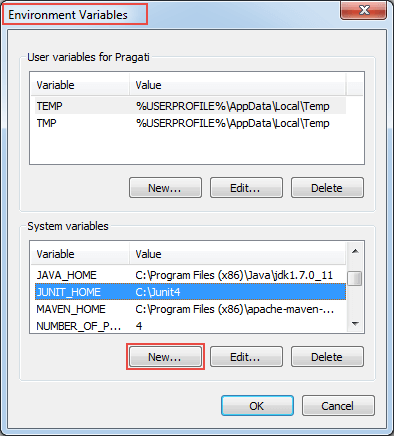
* JUnit.jar
* hamcrest-core.jar

## 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.

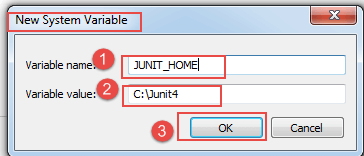
1. Under environment window clicks on "new" button.

[](https://www.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:

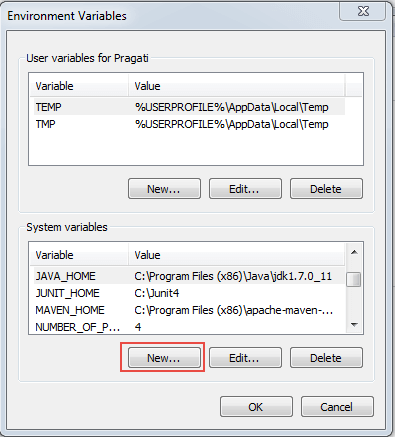
1. Provide variable name as "JUNIT\_HOME".
2. Provide JUnit value as JUnit path where you have copied JUnit jar files.
3. Click on OK.

[](https://www.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.

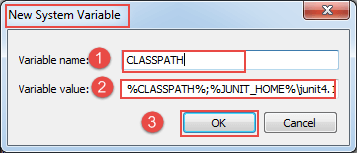
**Step 3)**After creating JUNIT\_HOME, create another variable with the name CLASSPATH. Again go to Environment Variables and follow the below steps.

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

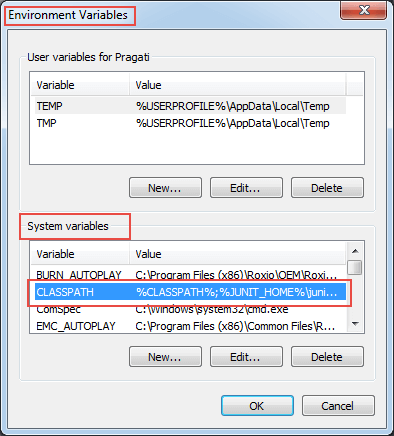
[](https://www.guru99.com/images/junit/051716_0555_HowtoDownlo8.png)

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

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

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

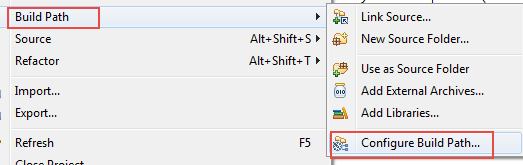
**Step 5)** Once you click on the 'OK' button, you can verify that a new environment variable named "CLASSPATH" can be seen under system variable. See below

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

## PART 4) Install JUnit jar file in eclipse

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

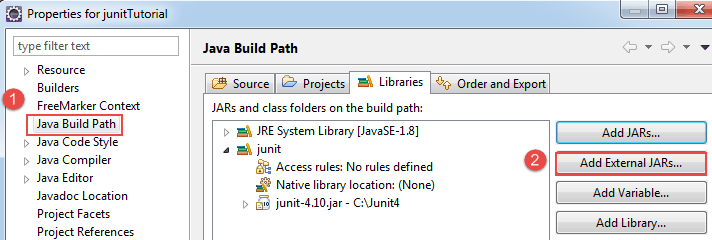
1. Click on "build path" and then
2. Click on "Configure build path".

[](https://www.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 your downloaded JUnit.jar file with eclipse.

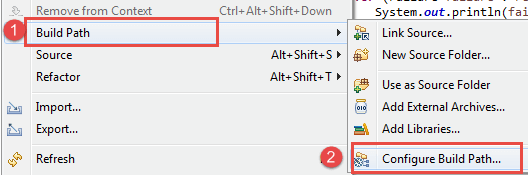
After adding a JUnit.jar file, click on 'OK' button to close java build path window.

[](https://www.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:

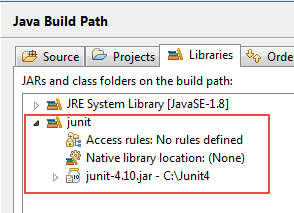
1. Right click on project -> Build Path
2. Click on "Configure build path".

[](https://www.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://www.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.

package klu.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.

package klu.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());

}

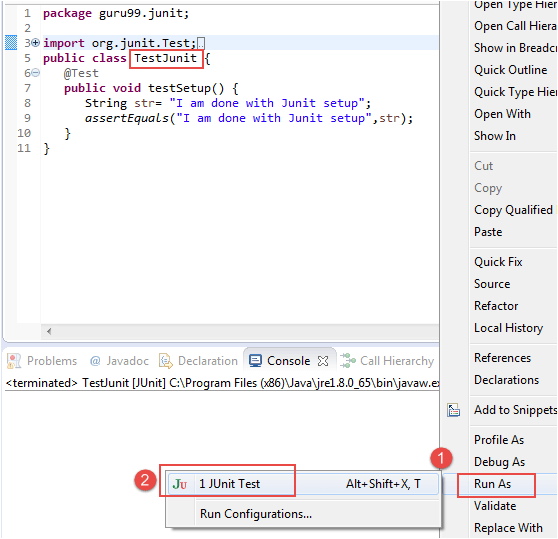
System.out.println("Result=="+result.wasSuccessful());

}

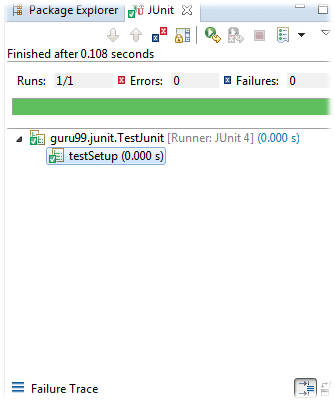
}

**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://www.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://www.guru99.com/images/junit/051716_0555_HowtoDownlo18.png)

**JUnit Test Cases @Before @BeforeClass Annotation**

JUnit is the most popular unit Testing framework in Java. It is explicitly recommended for Unit Testing. JUnit does not require server for testing web application, which makes the testing process fast.

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.

public class OutputFileTest {

private File output;

output = new File(...);

output.delete();

public void testFile1(){

//Code to verify Test Case 1

}

output.delete();

output = new File(...);

public void testFile2(){

//Code to verify Test Case 2

}

output.delete();

}

**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

public class OutputFileTest

{

private File output;

@Before public void createOutputFile()

{

output = new File(...);

}

@After public void deleteOutputFile()

{

output.delete();

}

@Test public void testFile1()

{

// code for test case objective

}

@Test public void testFile2()

{

// code for test case objective

}

}

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

A test fixture is a context where a JUnit Test Case 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 JUnit annotations are discussed below-

### Setup

**@Before** annotation in JUnit is used on a method containing Java 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** in JUnit 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 JUnit @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 test fixture**

public class OutputFileTest

{

private File output;

@Before public void createOutputFile()

{

output = new File(...);

}

@After public void deleteOutputFile()

{

output.delete();

}

@Test public void testFile1()

{

// code for test case objective

}

@Test public void testFile2()

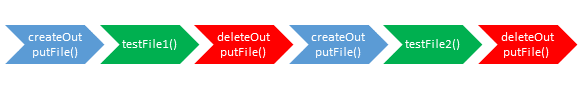
{

// code for test case objective

}

}

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

[](https://www.guru99.com/images/junit/051716_0608_JUnitTestfr1.png)

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 annotation **@BeforeClass** in JUnit.

@BeforeClass public static void Method\_Name() {

// class setup code here

}

### 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

@AfterClass public static void Method\_Name()

{

// class cleanup code here

}

## 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.

public class Test {

public static void main(String[] args) {

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

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

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

}

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

}

}

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**

package klu.JUnit;

import static org.JUnit.Assert.\*;

import org.JUnit.Test;

public class MyFirstClassTest {

@Test

public void myFirstMethod(){

String str= "JUnit is working fine";

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

}

}

**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's runclasses() 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 :

package klu.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(MyFirstClassTest.class);

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

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

}

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

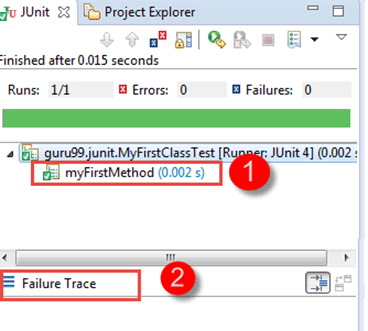
}

}

**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://www.guru99.com/images/junit/051716_0608_JUnitTestfr2.png)

# JUnit Annotations Tutorial with Example

## What is JUnit Annotations?

**JUNIT ANNOTATIONS** is a special form of syntactic meta-data that can be added to Java source code for better code readability and structure. Variables, parameters, packages, methods and classes can be annotated. 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 Junit5, one can easily learn and implement a JUnit test. Below is the list of important and frequently used annotations:

| **S.No.** | **Annotations** | **Description** |
| --- | --- | --- |
| 1. | @Test | This annotation is a replacement of org.junit.TestCase which indicates that public void method to which it is attached can be executed as a test Case. |
| 2. | @Before | This annotation is used if you want to execute some statement such as preconditions before each test case. |
| 3. | @BeforeClass | This annotation is used if you want to execute some statements before all the test cases for e.g. test connection must be executed before all the test cases. |
| 4. | @After | This annotation can be used if you want to execute some statements after each Test Case for e.g resetting variables, deleting temporary files ,variables, etc. |
| 5. | @AfterClass | This annotation can be used if you want to execute some statements after all test cases for e.g. Releasing resources after executing all test cases. |
| 6. | @Ignores | This annotation can be used if you want to ignore some statements during test execution for e.g. disabling some test cases during test execution. |
| 7. | @Test(timeout=500) | This annotation can be used if you want to set some timeout during test execution for e.g. if you are working under some SLA (Service level agreement), and tests need to be completed within some specified time. |
| 8. | @Test(expected=IllegalArgumentException.class) | This annotation can be used if you want to handle some exception during test execution. For, e.g., if you want to check whether a particular method is throwing specified exception or not. |

## 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 klu.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 klu.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 m8() 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");

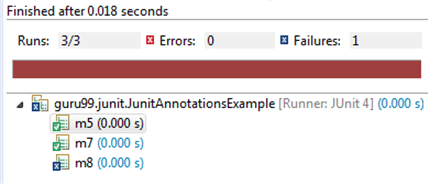
}

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://www.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

## 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:

| **S.No.** | **Method** | **Description** |
| --- | --- | --- |
| 1. | void assertEquals(boolean expected, boolean actual) | It checks whether two values are equals similar to equals method of Object class |
| 2. | void assertFalse(boolean condition) | functionality is to check that a condition is false. |
| 3. | void assertNotNull(Object object) | "assertNotNull" functionality is to check that an object is not null. |
| 4. | void assertNull(Object object) | "assertNull" functionality is to check that an object is null. |
| 5. | void assertTrue(boolean condition) | "assertTrue" functionality is to check that a condition is true. |
| 6. | void fail() | If you want to throw any assertion error, you have fail() that always results in a fail verdict. |
| 7. | void assertSame([String message] | "assertSame" functionality is to check that the two objects refer to the same object. |
| 8. | void assertNotSame([String message] | "assertNotSame" functionality is to check that the two objects do not refer to the same object. |

## 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:

| **S.No.** | **Method** | **Description** |
| --- | --- | --- |
| 1. | int countTestCases() | This method is used to count how many number of test cases executed by **run(TestResult tr)** method. |
| 2. | TestResult createResult() | This method is used to create a **TestResult** object. |
| 3. | String getName() | This method returns a string which is nothing but a **TestCase**. |
| 4. | TestResult run() | This method is used to execute a test which returns a **TestResult** object |
| 5. | void run(TestResult result) | This method is used to execute a test having a **TestResult** object which doesn't returns anything. |
| 6. | void setName(String name) | This method is used to set a name of a **TestCase.** |
| 7. | void setUp() | This method is used to write resource association code. e.g. Create a database connection. |
| 8. | void tearDown() | This method is used to write resource release code. e.g. Release database connection after performing transaction operation. |

## 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:

| **S.No.** | **Method** | **Description** |
| --- | --- | --- |
| 1. | void addError(Test test, Throwable t) | This method is used if you require add an error to the test. |
| 2. | void addFailure(Test test, AssertionFailedError t) | This method is used if you require add a failure to the list of failures. |
| 3. | void endTest(Test test) | This method is used to notify that a test is performed(completed) |
| 4. | int errorCount() | This method is used to get the error detected during test execution. |
| 5. | Enumeration<TestFailure> errors() | This method simply returns a collection (Enumeration here) of errors. |
| 6. | int failureCount() | This method is used to get the count of errors detected during test execution. |
| 7. | void run(TestCase test) | This method is used to execute a test case. |
| 8. | int runCount() | This method simply counts the executed test. |
| 9. | void startTest(Test test) | This method is used to notify that a test is started. |
| 10. | void stop() | This method is used to test run to be stopped. |

## 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:

| **S.No.** | **Method** | **Description** |
| --- | --- | --- |
| 1. | void addTest(Test test) | This method is used if you want to add a test to the suite. |
| 2. | void addTestSuite(Class<? extends TestCase> testClass) | This method is used if you want to specify the class while adding a test to the suite. |
| 3. | int countTestCases() | This method is used if you want to count the number of test cases. |
| 4. | String getName() | This method is used to get the name of the test suite. |
| 5. | void run(TestResult result) | This method is used to execute a test and collect test result in **TestResult** object. |
| 6. | void setName(String name) | This method is used to set the name of **TestSuite**. |
| 7. | Test testAt(int index) | This method is used if you want to return the test at given index. |
| 8. | int testCount() | This method is used if you want to return a number of tests in the Suite. |
| 9. | static Test warning(String message) | This method returns a test which will fail and log a warning message. |

# Junit Assert & AssertEquals with Example

## 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
* Null object
* Identical
* Assert Equals
* Assert Array Equals
* Fail Message
* JUnit assertEquals
* Floating point assertions
* JUnit Assert Example

## 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 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 klu.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 klu.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**.

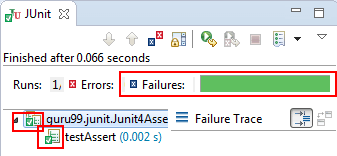
1. **assertArrayEquals(airethematicArrary1, airethematicArrary2);**

"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://www.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.

**package** klu.junit;

**import** org.junit.Test;

**public** **class** MyFirstClassTest {

@Test

**public** **void** myFirstMethod() {

}

}

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

**package** klu.junit;

**import** org.junit.Test;

**public** **class** MySecondClassTest {

@Test

**public** **void** mySecondMethod() {

}

}

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

**package** klu.junit;

**import** org.junit.runner.RunWith;

**import** org.junit.runners.Suite;

**import** org.junit.runners.Suite.SuiteClasses;

**public** **class** TestSuiteExample {

@RunWith(Suite.**class**)

@SuiteClasses(MyFirstClassTest.**class**,MySecondClassTest.**class**)

// Code here

}

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

**package** klu.junit;

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** Test{

**public** **static** **void** main(String[] args) {

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

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

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

}

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

}

}

**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:

**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 klu.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 klu.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class SuiteTest1 {

public String message = "Saurabh";

JUnitMessage junitMessage = 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 klu.junit;

import org.junit.Assert;

import org.junit.Test;

public class SuiteTest2 {

@Test

public void createAndSetName() {

String expected = "Y";

String actual = "Y";

Assert.assertEquals(expected, actual);

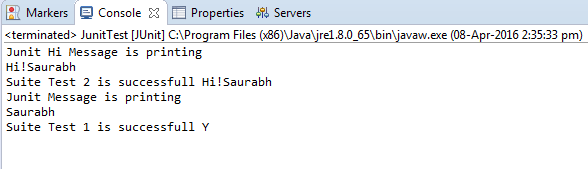
System.out.println("Suite Test 1 is successful " + actual);

}

}

**Output**

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

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

# JUnit @Ignore Test Annotation with Example

Sometimes you may require not to execute a method/code or Test Case 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 class which prints two types of message.

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

**JUnitMessage.java**

package klu.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 klu.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class JunitTestExample {

public String message = "Klu";

JUnitMessage junitMessage = 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 klu.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());

}

}

**Print statement on console:**

Junit Hi Message is printing

Hi!Klu

Junit Message is printing

Klu

## 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 klu.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Ignore;

import org.junit.Test;

public class JunitTestExample {

public String message = "Klu";

JUnitMessage junitMessage = 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());

} }

**Print statement on console:**

Junit Hi Message is printing

Hi!Klu

## 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 klu.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Ignore;

import org.junit.Test;

public class JunitTestExample {

public String message = "Klu";

JUnitMessage junitMessage = 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 klu.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Ignore;

import org.junit.Test;

@Ignore

public class JunitTestExample {

public String message = "Klu";

JUnitMessage junitMessage = 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());

}

}

**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.

**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:

package klu.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**

package klu.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class AirthematicTest {

public String message = "Saurabh";

JUnitMessage junitMessage = 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**

# 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 klu.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 klu.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 klu.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());

}

}

## 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.

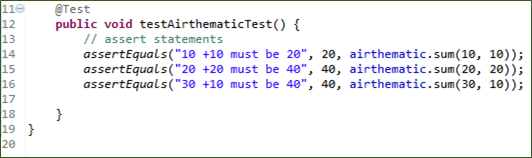
**JUnit Parameterized Test with Example using @Parameters**

**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://www.guru99.com/images/junit/052416_1228_JUnitParame1.png)

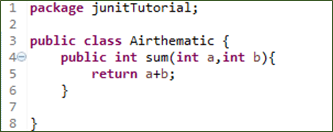
The approach above leads to lot of redundancy.

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.

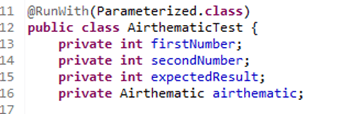
**Steps to create a Parameterized JUnit test**

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://www.guru99.com/images/junit/052416_1228_JUnitParame2.png)

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

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

Code Explanation

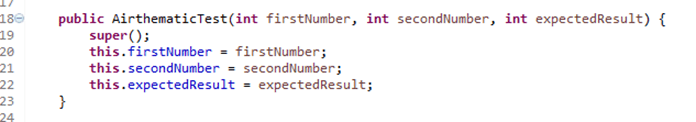
* **Code Line 11:**Annotate your test class using @runWith(Parameterized.class).
* **Code Line 13:**Declaring the variable 'firstNumber' as private and type as int.
* **Code Line 14:**Declaring the variable 'secondNumber'as private and type as int.
* **Code Line 15:**Declaring the variable 'expectedResult'as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

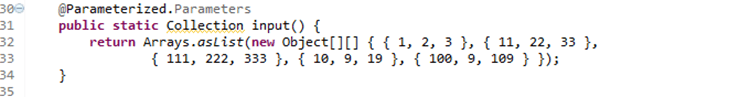
This class is responsible for tests to run with a new test instance. It is responsible for invoking JUnit lifecycle methods such as setup(associate resources) and teardown(release resources).

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://www.guru99.com/images/junit/052416_1228_JUnitParame4.png)

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

[](https://www.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

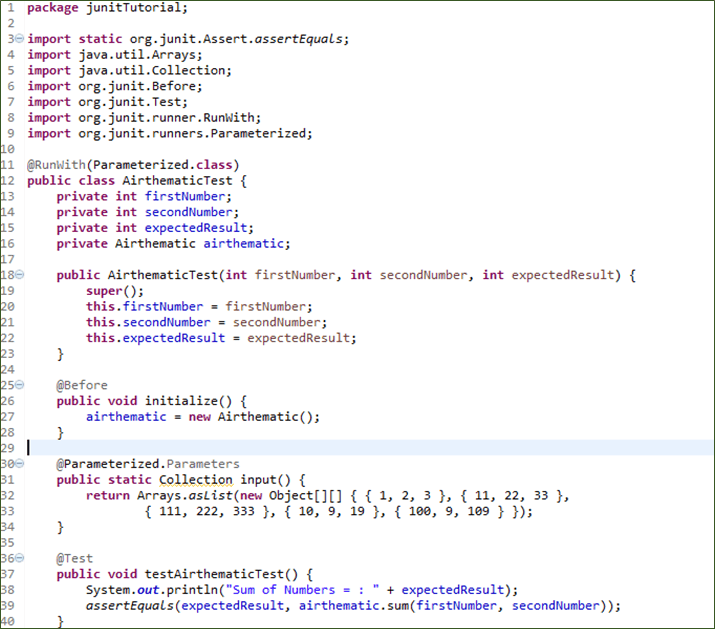
firstNumber =1

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.

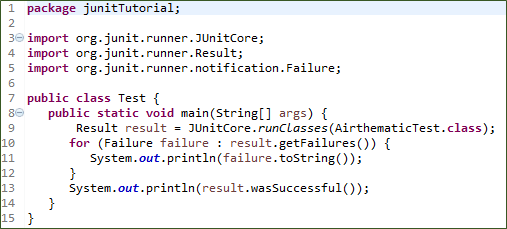
**Step 5)** The complete code

[](https://www.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:

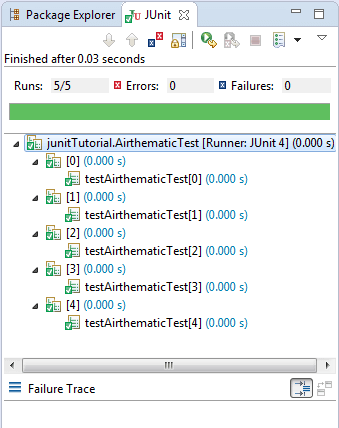
[](https://www.guru99.com/images/junit/052416_1228_JUnitParame7.png)

**Code Explanation:**

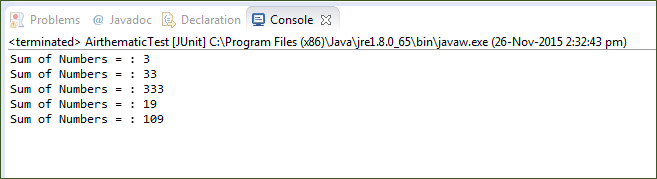
* **Code Line 8:**Declaring the main method of the class test which will run our JUnit test.
* **Code Line 9:**Executing test cases using JunitCore.runclasses, it will take the testclass name as a parameter (In our example we are using Airthematic.class).
* **Code Line 11:**Processing the result using 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://www.guru99.com/images/junit/052416_1228_JUnitParame8.png)

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

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