

# **JUnit Testing**

## **Overview**

1. Unit Testing Basics .....	1
2. JUnit Design Goals .....	2
3. Using JUnit .....	2
3.1. Adding the JUnit library in Eclipse .....	2
3.2. Project Preparation .....	3
3.3. Class Creation .....	4
3.4. Testing the multiply method .....	4
3.5. Create a JUnit Test .....	6
3.6. Run the JUnit Test .....	8
4. JUnit Overview .....	10
4.1. Annotations .....	11
4.2. Assert Statements .....	11
5. References .....	12

## **1. Unit Testing Basics**

- In Java, the standard unit testing framework is known as JUnit. It was created by Erich Gamma and Kent Beck (1995).
- During any software engineering process, developers need to run their code to verify that the software works as intended.
- As the software gets larger and larger, it becomes more likely that introducing a new change will “break” the existing code.
- Testing the whole software manually is very time consuming; doing it by the computer itself would prove beneficial to the company and developers.
- JUnit tests reduce manual testing by checking the codes/classes automatically against predefined answers.

- They make regression testing very easy since predefined test cases can be launched when required without any further changes to the testing codes.

## **2. JUnit Design Goals**

The JUnit team has defined three discrete goals for the framework:

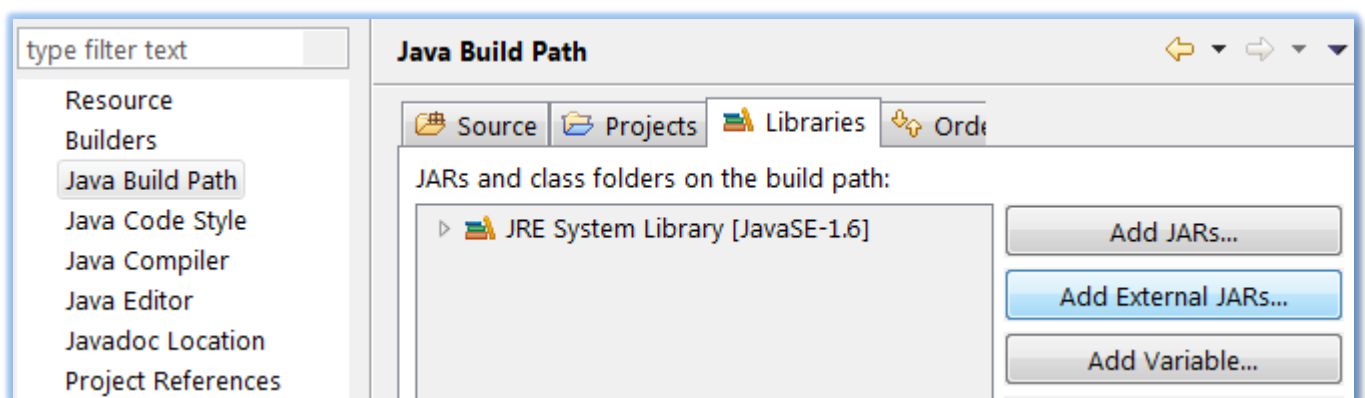
1. The framework must help us write useful tests.
2. The framework must help us create tests that retain their value over time.
3. The framework must help us lower the cost of writing tests by reusing code.

## **3. Using JUnit**

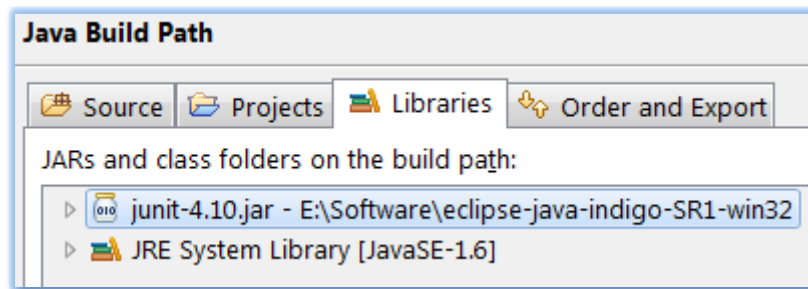
To automate testing for a given project, the **JUnit library** should be included in the project so as to get access to the JUnit framework.

### **3.1. Adding the JUnit library in Eclipse**

1. Download the JUnit library from <http://www.junit.org/>
2. Import the **JUnit.jar**
3. Select “Java Build Path”, choose the “Libraries” tab and click on “Add External JARs”.



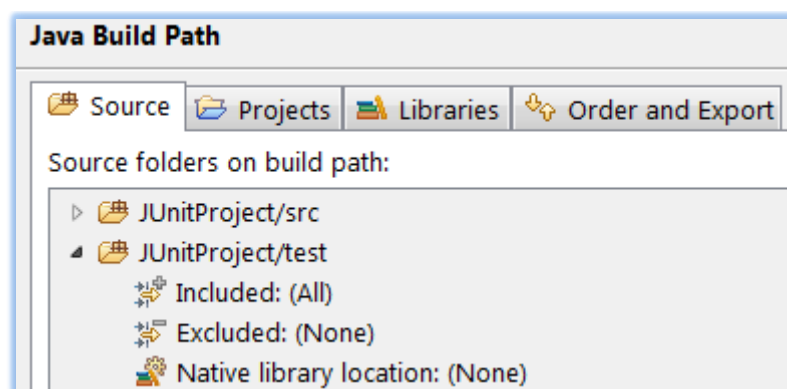
4. After JUnit.jar is successfully imported, the library is listed as shown below. *Note that the JUnit path will depend where you have saved the jar file.*



5. Close the Properties Window.

### 3.2. Project Preparation

- Create a new Java project called “**JUnitProject**”.
- Create a new source folder “**test**” in the project.
  - We want to create the unit tests in a separate folder. The creation of a separate folder for tests is not mandatory. But, it is a good practice to keep the code separated from the regular codes.
- Right-click on the project, select "Properties" and choose the "Java Build Path". Add a new folder “**test**” from the "Source" tab.



### 3.3. Class Creation

Create the following java class in the “src” folder.

```
public class Maths {  
    public int multiply(int x, int y) {  
        return x * y;  
    }  
}
```

The purpose of the **multiply (int, int)** method is to take two integers and return the result as integer.

### 3.4. Testing the multiply method

How to test the multiply method?

- We don't have a user interface to enter a pair of integer values !

#### **Possible Solution:**

We could write a small command-line program that waits for us to type in two integer values and then display the result.

```
import java.util.Scanner;

public class MathsTesting {

    public static void main(String[] args) {
        int firstInteger, secondInteger, result = 0;

        Scanner kb = new Scanner (System.in);
        System.out.println("Enter first integer:");
        firstInteger = kb.nextInt();
        kb.nextLine(); //moving input cursor on a fresh line

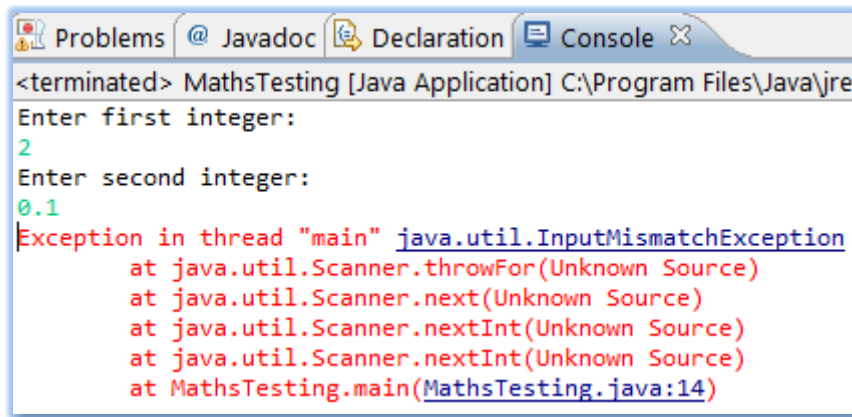
        System.out.println("Enter second integer:");
        secondInteger = kb.nextInt();
        kb.nextLine(); //moving input cursor on a fresh line

        result = new Maths().multiply(firstInteger, secondInteger);
        System.out.println("Results: " + result);
    }
}
```

- The “MathsTesting” program is simple; it creates an instance of Maths, passes it two numbers, and displays the result.
- If the result doesn’t meet our expectations, we print a message on standard output.
- If we compile and run this program, the test is successful.

But, what happens if we change the code so that it fails?

- For example, we can input decimal numbers which will generate errors.
- The conventional way to signal error conditions in Java is to throw an exception using the TRY – CATCH block.



```
<terminated> MathsTesting [Java Application] C:\Program Files\Java\jre
Enter first integer:
2
Enter second integer:
0.1
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Unknown Source)
    at java.util.Scanner.next(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at MathsTesting.main(MathsTesting.java:14)
```

The previous scenario was not automated. What we really needed were running tests in batches and in the end getting a summary of successful/failed tests.

JUnit saves us the trouble of creating manual tests. The JUnit framework supports a different class instance and a class loader instance for each test and reports all errors.

### 3.5. Create a JUnit Test

Our objective is to test the “Maths” class using JUnit.

To create a JUnit test case, follow these steps:

- Right-Click on the “Maths” class → New → JUnit Test Case.
- We change the Source folder to “test”, so that the test class is generated in this folder. *Note that this step is optional.*

☐ New JUnit 3 test ☒ New JUnit 4 test

Source folder: JUnitProject/test

Package:

Name: MathsTest

Superclass: java.lang.Object

- The next step is to choose the **methods** you want to test.

**Test Methods**

Select methods for which test method stubs should be created.

Available methods:

- ☒ ☒ Maths
  - ☒ multiply(int, int)
- ☐ Object
  - ☐ Object()
  - ☐ getClass()

- Click on Finish to complete the test case. *If the JUnit library is not part of your classpath, Eclipse will prompt you to do so.*
- The class is created with the appropriate method stubs generated for us. All we need to do now is write the tests.

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;
import org.junit.Test;

public class MathsTest {

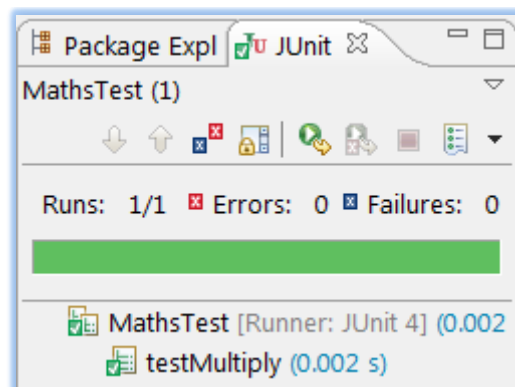
    @Test
    public void testMultiply() {
        Maths myMaths = new Maths();
        assertEquals("Result", 50, myMaths.multiply(10, 5));
    }
}
```

### Exercise 1

Add a subtract method to the “Maths” class. The method takes as parameters two variables of type integer and returns the result. Write the corresponding JUnit test.

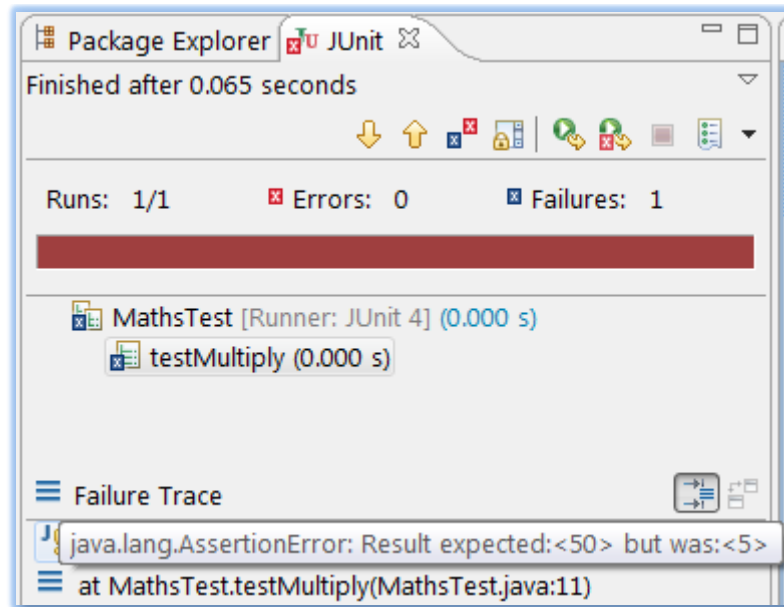
### 3.6 Run the JUnit Test


- Right click on your new test class “MathsTest.java” and select Run-As → JUnit Test.
- The results are as follows:

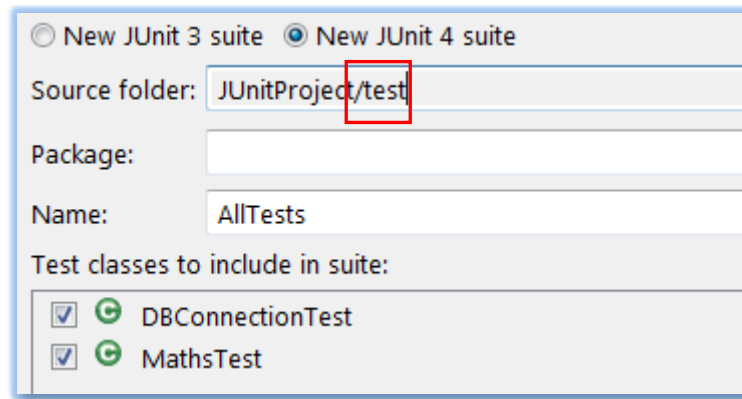




- Tests that fail are indicated with a red bar. For example, if we introduce a bug by changing the arithmetic operator multiply (\*) in the method “multiply(int, int)” to subtraction (-), the following results are obtained:



- The top view displays the name(s) of the test method(s) that failed .
- The bottom view shows a stack trace of the method calls leading to the failure. The **assertEquals** method indicate that the expected value was 50 but the value 5 was obtained.
- If you have several tests you can combine them into a test suite. Running a test suite will execute all tests in that suite.
  - To create a test suite, select your test classes → right click → New → Other → Java → JUnit → JUnit Test Suite.



- The following codes are generated. Any future JUnit tests can be added to @SuiteClasses.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ DBConnectionTest.class, MathsTest.class })
public class AllTests {

}
```

- To run all the tests, right click on “AllTests.java” → Run as → JUnit Test.

## **4. JUnit Overview**

JUnit has many features that make it easy to write and run tests. Some features are:

- JUnit creates a new instance of the test class before invoking each @Test method. This helps provide independence between test methods and avoids unintentional side effects in the test code (instance variable values cannot be used across test methods).

- JUnit annotations to provide resource initialization and reclamation methods: @Before, @BeforeClass, @After, and @AfterClass.
- A variety of assert methods to make it easy to check the results of your tests.

## 4.1. Annotations

The following table gives an overview of the available annotations in JUnit 4.x.

Annotation	Description
@Test public void method()	The annotation @Test identifies that a method is a test method.
@Before public void method()	Will execute the method before each test. This method can prepare the test environment (e.g. read input data, initialize the class).
@After public void method()	Will execute the method after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults).
@BeforeClass public void method()	Will execute the method once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database.
@AfterClass public void method()	Will execute the method once, after all tests have finished. This can be used to perform clean-up activities, for example to disconnect from a database.
@Ignore	Will ignore the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.
@Test (expected = Exception.class)	Fails, if the method does not throw the named exception.
@Test(timeout=100)	Fails, if the method takes longer than 100 milliseconds.

## 4.2. Assert statements

The following table gives an overview of the available assert statements.

Statement	Description
fail(String)	Let the method fail. Might be used to check that a certain part of the code is not reached. Or to have failing test before the test code is implemented.
assertTrue(true) / assertTrue(false)	Will always be true / false. Can be used to predefine a test result, if the test is not yet implemented.
assertTrue([message], boolean condition)	Checks that the boolean condition is true.
assertEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([String message], expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message], object)	Checks that the object is null.
assertNotNull([message], object)	Checks that the object is not null.
assertSame([String], expected, actual)	Checks that both variables refer to the same object.
assertNotSame([String], expected, actual)	Checks that both variables refer to different objects.

## Exercise 2

Consider the following class for connecting to an MS Access database. Write the corresponding JUnit test.

```
import java.sql.DriverManager;
import java.sql.Connection;

class DBConnection{
public boolean DBConnect(){
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    String filename = "c:/database/javadb.mdb";
    String database = "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb, *.accdb)};DBQ=";
                                database+= filename.trim() +
                                ";DriverID=22;READONLY=true}";
    Connection con = DriverManager.getConnection( database , "", "");
    return true;
}
catch (Exception e) {
    System.out.println("Error: " + e);
    return false;
}
}
}
}

```

## 5. References

<http://www.vogella.de/articles/JUnit/article.html>