Lab I: JUnit: Automated Testing

Suite Testing and Testing Using TestNG, Parametrized Tests and Theories in JUnit

Software Testing and QA (COE891)

Week 2 and Week 3

1 Introduction to JUnit Testing

1.1 Objectives

- Introducing the concept of unit testing within the Java framework of JUnit.
- By the end of this lab, the students should be able to design and set up JUnit tests for the programs that they write or are given.

1.2 Test-Driven Development

Test-Driven Development (TDD) is a software development process which includes test-first development. It means that the developer first writes a fully automated test case before writing the production code to fulfil that test and refactoring. The steps are as follows:

1. Add a test.
2. Run all the tests and see if any new test fails.
3. Update the code to make it pass the new tests.
4. Run the test again and if they fail then refactor again and repeat.

1.3 Lab Setup

Before beginning this lab, you should have the Eclipse Java IDE and Java Runtime Environment installed on your computer.

1.4 JUnit Test Tool

JUnit is a Java library to help you perform unit testing. Unit testing is the process of examining a small "unit" of software (usually a single class) to verify that it meets its expectations or specification.

A unit test targets some other class under test; for example, the class ArrayIntListTest might be targeting the ArrayIntList as its class under test. A unit test generally consists of various testing methods that each interact with the class under test in some specific way to make sure it works as expected.

JUnit is not part of the standard Java class libraries, but it does come included with Eclipse or if you are not using Eclipse, JUnit can be downloaded for free from the JUnit web site which will be further provided. JUnit is distributed as a "JAR" which is a compressed archive containing Java .class files.

1.5 Implementing Java Test-Driven Development

1.5.1 Naming

We use common conventions in naming a test class. Let's start with the name of a class which is being tested and assume the name of that class is "Student". In that case, the name of the test class should be "StudentTest". We have to append "Test" to it. The same naming convention is used in the case of methods. If there is a method "DisplayStudentAddress()", then the name of the method in testing should be "testDisplayStudentAddress()".

Naming in Production	Naming in Testing
Student	StudentTest
DisplayStudentAddress()	testDisplayStudentAddress()

Production Code:

```
public class Student {
 public String displayStudentName(String firstName, String lastName) {
  return firstName + lastName;}
}
```

Testing Code:

```
Import org.junit.Test;
Import static org.junit.Assert.*;
Public class StudentTest {
 @Test
 Public void testDisplayStudentName() {
  Student student = new Student();
  String studentName = student.displayStudentName("Anshuman", "Nain");
  assertEquals("AnshumanNain", studentName);}
}
```

1.5.2 Packages

We do not use the same package for production code and testing code. The best practice is to use different source directories, "src/main/java" for production and "src/test/java" for testing.

1.5.3 Structure and Annotations

The annotation @Test (JUnit4) tells JUnit to execute the *testDisplayStudentAddress()* method as a test method and report whether it passes or fails. As long as all assertions (if any) pass and no exceptions are thrown, the test is considered to pass. Thus, our test code should follow AAA (Arrange Act Assert) pattern. Now build the project and run.

1.6 Unit Testing With JUnit: Prerequisites

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.

- Installing JDK: Java Development Kit
- Installing Eclipse IDE: Eclipse IDE

Download JUnit:

1. Visit JUnit4 Website and click Download and Install
2. Click junit.jar
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 latest version.

4. Visit hamcrest-core again. Click hamcrest-core.jar

5. Download the Jar file.

JUnit Environment Setup:

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

2. A "New System Variable" window will open:

   Provide variable name as "JUNIT_HOME". Provide JUnit value as JUnit path where you have copied JUnit jar files. Click on OK.

   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.

3. After creating JUNIT_HOME, create another variable with the name CLASSPATH.

   Again go to Environment Variables and follow the below steps. Click on "new" button. When you click on new in environment variables, it will open another window.

4. In this step, point out JUNIT_HOME to JUnit.jar which is placed in JUnit folder as given below:

   Variable Name: CLASSPATH
   Variable Value: %CLASSPATH%;%JUNIT_HOME%
   JUnit4.10.jar;.;
   Click on the OK button.

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

Install JUnit Jar File in Eclipse:

1. Right click on project: Click on "build path" and then click on "Configure build path".

2. In this step, go to java build path window. 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.

To verify JUnit, create a java class named TestJUnit.java and provide a simple assert statement:

```
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); }
}
```

Then, create a Test Runner class to execute above test:

```
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());
```

```
10          }
11          System.out.println("Result=="+result.wasSuccessful()); }
12  }
```

Finally, to execute the test, right click on TestJunit.java and click on “Run As”. Another window will be open once you click on “Run As”, click on “1 JUnit Test”. If it is successfully executed, it will show a green check mark in front of it.

1.7 Creating A JUnit Test Case in Eclipse

To use JUnit you must create a separate .java file in your project that will test one of your existing classes. In the Package Explorer area on the left side of the Eclipse window, right click the class you want to test and click New → JUnit Test Case.

A dialog box will pop up to help you create your test case. Make sure that the option at the top is set to use JUnit 4. Click Next.

You will see a set of checkboxes to indicate which methods you want to test. Eclipse will help you by creating "stub" test methods that you can fill in (you can always add more later manually). Choose the methods to test and click Finish.

At this point Eclipse will ask whether you want it to automatically attach the JUnit library to your project. Yes, you do. Select "Perform the following action: Add JUnit 4 library to the build path" and press OK (If you forget to do add JUnit to your project, you can later add it to your project manually by clicking the top Project menu, then Properties, then Java Build Path, then click Add Library..., and choose JUnit 4 from the list).

When you are done, you should have a nice new JUnit test case file. We suggest that you change the second import statement at the top to state the following:

```
1  import static org.junit.Assert.*;
2  import org.junit.*;
3  import java.util.*;
```

1.8 Writing Tests

Each unit test method in your JUnit test case file should test a particular small aspect of the behavior of the "class under test." For example, an ArrayIntListTest (a test class for array list of integers) might have one testing method to see whether elements can be added to the list and then retrieved. Another test might check to make sure that the list’s size is correct after various manipulations, and so on. Each testing method should be short and should test only one specific aspect of the class under test.

JUnit testing methods utilize assertions, which are statements that check whether a given condition is true or false. If the condition is false, the test method fails. If all assertions’ conditions in the test method are true, the test method passes. You use assertions to state things that you expect to always be true, such as `assertEquals(3, list.size());` if you expect the array list to contain exactly 3 elements at that point in the code. JUnit provides the following assertion methods:

Notice that when using comparisons like `assertEquals`, expected values are written as the left (first) argument, and the actual calls to the list should be written on the right (second argument). This is so that if a test fails, JUnit will give the right error message such as, "expected 4 but found 0".

A well-written test method chooses the various assertion method that is most appropriate for each check. Using the most appropriate assertion method helps JUnit provide better error messages when a test case fails. You might think that writing unit tests is not useful. After all, we can just look at the code of methods like add or `isEmpty` to see whether they work, but it is easy to have bugs, and JUnit will catch them better than our own eyes.

Even if we already know that the code works, unit testing can still prove useful. Sometimes we introduce a bug when adding new features or changing existing code; something that used to work is now broken. This is called a regression. If we have JUnit tests over the old code, we can make sure that they still pass and avoid costly regressions. Let’s write the logic to find the maximum number for an array:

Function & Arguments	Description
assertTrue(test) assertTrue("message", test)	Fails if the given boolean test is not true.
assertFalse(test) assertFalse("message", test)	Fails if the given boolean test is not false.
assertEquals(expectedValue, value) assertEquals("message", expectedValue, value)	Fails if the given two values are not equal.
assertNotEquals(value1, value2) assertNotEquals("message", value1, value2)	Fails if the given two values are equal.
assertNull(value) assertNull("message", value)	Fails if the given value is not null.
assertNotNull(value) assertNotNull("message", value)	Fails if the given value is null.
assertSame(expectedValue, value) assertSame("message", expectedValue, value) assertNotSame(value1, value2) assertNotSame("message", value1, value2)	Identical to assertEquals and assertNotEquals, respectively (== operator for objects and an object is only == to itself).
fail() fail("message")	This test method will fail.

```
public class Calculation {
    public static int findMax(int arr[]){
        int max=0;
        for(int i=1;i<arr.length;i++){
            if(max<arr[i])
                max=arr[i];
        }
        return max; }
}
```

Here, we are using JUnit 4, so there is no need to inherit TestCase class. The main testing code is written in the `testFindMax()` method. But we can also perform some task before and after each test, as you can see in the given program.

```
import static org.junit.Assert.*;
import com.javatpoint.logic.*;
import org.junit.Test;
public class CalculationTest {
    @Test
    public void testFindMax(){
        assertEquals(4,Calculation.findMax(new int[]{1,3,4,2})); }
}
```

To run this example, right click on CalculationTest class → Run As → 1Junit Test.
Let's see the output displayed in eclipse IDE:

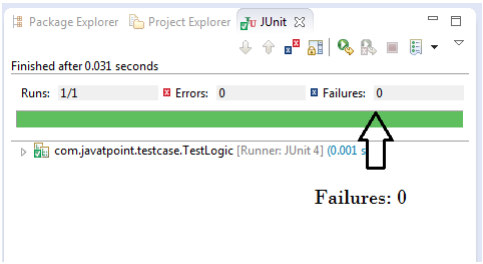


Figure 1: A sample test running

Also, for some assertions, you can open a dialog to compare the expected and the actual value. Right-click on the test result, and select Compare Result or double-click on the line.

1.9 Running Your Test Case

Once you have written one or two test methods, run your JUnit test case. There are two ways to do this. One way is to click the Run button in the top toolbar (it looks like a green "Play" symbol). A menu will drop down; choose to run the class as a JUnit Test. The other way is to right-click your JUnit test case class and choose Run As → JUnit Test.

A new pane will appear showing the test results for each method. You should see a green bar if all of the tests passed, or a red bar if any of the tests failed. If any tests fail, you can view the details about the failure by clicking on the failed test's name/icon and looking at the details in the pane below.

Getting a red failure bar is not bad. It means that you have found a potential bug to be fixed. Finding and fixing bugs is a good thing. Making a red bar become a green bar (by fixing the code and then re-running the test program) can be very rewarding.

The list of important annotations are as follows:

Annotation	**Description**
Initialization	
@Before	Run before each @Test method.
@BeforeClass	Run once before all @Test methods in a class.
Cleanup	
@After	Run after each @Test method.
@AfterClass	Run once after all @Test methods in a class.
Test	
@Test	Defines a JUnit test case method.
@Test(timeout=X)	Defines a JUnit test case method that runs until X milliseconds.
@Test(expected=ExcepName.class)	Defines a JUnit test case method that throws an exception *ExcepName*.
@Ignore	JUnit will ignore this method.

If you want to put this same timeout for multiple test cases, it is not necessary to add timeout attribute within each and every @Test annotation you have specified in your script. In such case, you can use @Rule annotation to define a global timeout that would be applicable for each @Test method annotation of your script.

```
1 @Rule
2 public Timeout globalTimeout = Timeout.seconds(10); // 10 seconds max per test method
```

1.9.1 Attributes Used With JUnit Annotations

These test annotations in JUnit have multiple attributes that can be used for our test method:

1. timeout: To specify timeout for each test case, timeout attribute is specified within @Test annotation. Timeout time is specified in milliseconds. For example, suppose, you know your particular test case takes around 15-20 seconds to get executed and you want your test to fail if it takes more than 20 seconds to get executed, then you can use this attribute as @Test(timeout=20000) for your particular test case. If an execution fails then the progress bar would get red instead of green.

```
@Test(timeout=10000)
```

2. expected: This is a special feature introduced by JUnit 4 which provides the facility to trace an exception that was expected from the execution of a particular code. There are different kinds of exceptions that are probably expected while running a code, such as, `NoSuchMethodException`, `ArithmeticException`, `IndexOutOfBoundsExceptions`, etc. For example, you expect occurrence of exception from your code when a particular number is divided by zero, in this case, you would

be using `ArithmeticException`. If the excepted exception does not occur then the test execution will fail.

```
1  @Test(expected=ArithmeticException.class)
```

Now, let's learn another example:
Here in this Test Driven Development example, we will define a class password. For this class, we will try to satisfy following condition. A condition for password acceptance is that the password should be between 5 to 10 characters. First in this TDD example, we write the code that fulfills all the above requirements.

```
package Prac;

public class PasswordValidator {
 public boolean isValid(String Password)
 {
     if (Password.length()>=5 && Password.length()<=10)
     {
         return true;
     }
     else
         return false;
 }
}
```

This is main condition checking length of password. If meets return true otherwise false.

Scenario 1: To run the test, we create class PasswordValidator();

```
package Prac;

import org.testng.Assert;
import org.testng.annotations.Test;

public class TestPassword {
  @Test
  public void TestPasswordLength() {
     PasswordValidator pv = new PasswordValidator();
     Assert.assertEquals(true, pv.isValid("Abc123"));
  }
}
```

Needed for TestNG

We can not run test because this class is not created yet

This is main validation test

We will run above class TestPassword(). Output is PASSED as shown below:

```
<terminated> TestPassword [TestNG] C:\Program Files\Java\jre1.8.0_77\bin\javaw.exe (Jul 25, 2016, 2:10:22 PM)
[TestNG] Running:
  C:\Users\kanchan\AppData\Local\Temp\testng-eclipse--571370159\testng-customsuite.xml

PASSED: TestPasswordLength

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================

[TestNG] Time taken by org.testng.reporters.EmailableReporter2@1b40d5f0: 202 ms
[TestNG] Time taken by org.testng.reporters.XMLReporter@28f67ac7: 63 ms
[TestNG] Time taken by org.testng.reporters.jq.Main@546a03af: 78 ms
[TestNG] Time taken by org.testng.reporters.JUnitReportReporter@5a01ccaa: 2 ms
[TestNG] Time taken by [FailedReporter passed=0 failed=0 skipped=0]: 1 ms
[TestNG] Time taken by org.testng.reporters.SuiteHTMLReporter@2b80d80f: 10 ms
```

Result of test as Passed

Scenario 2: Here we can see in method TestPasswordLength() there is no need of creating an instance of class PasswordValidator. Instance means creating an object of class to refer the members (variables/methods) of that class.

```java
package Prac;

import org.testng.Assert;

public class TestPassword {
  @Test
  public void TestPasswordLength() {

      PasswordValidator pv = new PasswordValidator();

      Assert.assertEquals(true, pv.isValid("Abc123"));

  }
}
```

We will remove it.

We will remove `nPasswordValidator pv = new PasswordValidator()` from the code. We can call the `isValid()` method directly by `PasswordValidator.IsValid("Abc123")`. So we Refactor (change code) as below:

```java
package Prac;

import org.testng.Assert;
import org.testng.annotations.Test;

public class TestPassword {
  @Test
  public void TestPasswordLength() {

      Assert.assertEquals(true, PasswordValidator.isValid("Abc123"));

  }
}
```

Re factor code as there is no need of creating instance of class PasswordValidator().

Scenario 3: After refactoring the output shows failed status (see image below) this is because we have removed the instance. So there is no reference to non –static method `isValid()`.

```
[TestNG] Running:
  C:\Users\kanchan\AppData\Local\Temp\testng-eclipse--157192639\testng-customsuite.xml

FAILED: TestPasswordLength
java.lang.Error: Unresolved compilation problem:
        Cannot make a static reference to the non-static method isValid(String) from the type PasswordValidator

        at Prac.TestPassword.TestPasswordLength(TestPassword.java:10)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at java.lang.reflect.Method.invoke(Unknown Source)
        at org.testng.internal.MethodInvocationHelper.invokeMethod(MethodI
        at org.testng.internal.Invoker.invokeMethod(Invoker.java:639)
        at org.testng.internal.Invoker.invokeTestMethod(Invoker.java:821)
```

If we removed instance creation statement compiler will give error. As we do not create instance it becomes non static method and there is no any reference to this method. Test results in Fail. To remove this error we have to make isValid() method of class PasswordValidator as static.

So we need to change this method by adding "static" word before Boolean as public static boolean `isValid(String password)`. Refactoring Class `PasswordValidator()` to remove above error to pass the test.

```java
package Prac;

public class PasswordValidator {
 public static boolean isValid(String Password)
 {
     if (Password.length()>=5 && Password.length()<=10)
     {
         return true;
     }
     else
         return false;
 }
}
```

Re factor : Added static word to pass test.

After making changes to class PassValidator() if we run the test then the output will be PASSED as shown below:

```
<terminated> TestPassword [TestNG] C:\Program Files\Java\jre1.8.0_77\bin\javaw.exe (Jul 25, 2016, 3:02:16 PM)
[TestNG] Running:
  C:\Users\kanchan\AppData\Local\Temp\testng-eclipse--1385484104\testng-customsuite.xml

PASSED: TestPasswordLength

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================

[TestNG] Time taken by org.testng.reporters.EmailableReporter2@1b40d5f0: 19 ms
[TestNG] Time taken by org.testng.reporters.XMLReporter@28f67ac7: 10 ms
[TestNG] Time taken by org.testng.reporters.jq.Main@546a03af: 34 ms
```

Test results passed as we changed code in class PasswordValidator().

2 Test Suite and Advanced JUnit

2.1 Objectives

- Performing unit testing within JUnit using its advanced features.
- Performing JUnit test suite for testing different Java programs concurrently.

2.2 Writing Unit Tests: A Quick Review

Now that we have learnt about JUnit testing and setup, we will move on to the actual construction and execution of these tests. To best illustrate more advanced aspects of JUnits, we want to review some basic concepts. In the JUnit test example image below, we have a simple method (top) that converts Fahrenheit to Celsius, and the JUnit (bottom) associated with our method. We will discuss the following sections in detail below:

```java
public class Conversion {
    public double tempConversion (double temperature, String unit) {
        if (unit.equals("F"))
        return (temperature - 32) * (5.0/9.0);
        else
        return (temperature * (9.0/5.0)) + 32; }
    }
```

```java
// Sections 1 and 2
import static org.junit.Assert.assertEquals;
import org.junit.*;
// Section 3
public class ConversionTest {
    // Section 4
    @Test
    // Section 5
    public void testTempCoversion() throws Throwable {
        // Section 6: Given
        Conversion underTest = new Conversion();

        // Section 7: When
        double temperature = 80.0d;
        String unit = "";
        // Section 8
        double result = underTest.tempConversion(temperature, unit);

        // Section 9: Then - assertions for result of method tempConversion
        assertEquals(176.0d, result, 0.0); }
    }
```

Sections 1 and 2: These are imports for the JUnit libraries needed to leverage the testing framework. The imported libraries can be specified down to a particular functionality of JUnit, but are commonly imported with asterisks to have access to all functionality.

Section 3: This has the start of our test class, the important thing to take note of here is the naming convention for the class, which follows ClassNameTest.

Section 4: Here, we see our first JUnit-specific syntax, an annotation. Annotations are extremely important when creating JUnits. This is how JUnit knows what to do with the processing section of code. In our example case, we have an @Test annotation, which tells JUnit that the public void method to which it is attached can be run as a test case.

There are many other annotations, but the more common are @Before (which runs some statement/precondition before @Test, public void), @After (which runs some statement after @Test, public void e.g. resetting variables, deleting temporary files, variables, etc.), and @Ignore (which ignores some statement during test execution — note that @BeforeClass and @AfterClass are used for running statements before and after all test cases, public static void, respectively).

Section 5: The take away here is again naming convention. Note the structure, testMethodName.

Section 6: Here we construct a new instance of our class object. This is necessary so we can call the method we are testing on something. Without this object instance, we cannot test the method.

Section 7: Variables associated with the method need to be established, so here we declare variables corresponding to our method. These should be given meaningful values (note: if a parameter is an object, one can instantiate it, or mock it), so that our test has meaning.

Section 8: This variable declaration could be argued as optional, but it's worthwhile for the sake of organization and readability. We assign the results of our method being tested to this variable, using it as needed for assertions and such.

Section 9: The assert methods (which are part of the org.junit.Assert class) are used in determining pass/fail status of test cases. Only failed assertions are recorded. Like with annotations, there are many assert options. In our example JUnit above, we use assertEquals(expected, actual, delta). This takes in the expected outcome, which the user defines, the actual, which is the result of the method being called, and the delta, which allows for implementing an allowed deviation between expected and actual values. The purpose of an assertion is validation. Although not required to run your JUnit, failing to add assertions arguably defeats the purpose of your test. Without assertions, you have no verification and at most a smoke test, which gives feedback only when test errors out.

2.3 JUnit Suite Test: Creating Test Suites

Suite test means creating and running more than one test from two or more separated (but maybe related) classes simultaneously. You can create a test suite via Eclipse. For this, select the test classes which should be included in suite in the Package Explorer view, right-click on them and select New → Other... → Java → JUnit → JUnit Test Suite.

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, we 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,...)
3. @SelectPackages specifies the names of packages to select when running a test suite via @RunWith(JUnitPlatform.class).
4. Filtering Test Classes with @IncludeClassNamePatterns and @ExcludeClassNamePatterns

Note that @SelectPackages causes all its sub-packages to be scanned as well for test classes. If we want to exclude any specific sub-package, or include any package, then you may use @IncludePackages and @ExcludePackages annotations.

As simple JUnit tests for greeting and maximum number for an array using @Test annotation, we are using two various classes with different logics:

Writing a program for Greeting:

```
public class Greeting {
    public String sayHello(){
        return "Hello!"; }
}
```

Writing a program for findMax:

```
public class Calculation {
    public static int findMax(int arr[]){
        int max=0;
        for(int i=1;i<arr.length;i++){
            if(max<arr[i])
                max=arr[i];
```

```
        }
        return max; }
}
```

Writing testcases:

1) Writing Testcases for Greetings class:

```
import static org.junit.Assert.*;
import com.javatpoint.logic.*;
import org.junit.Test;

public class GreetingTest {
    Greeting g;
    @Before
    public void init(){
        g = new Greeting(); }
    @Test
    public void testSayHello(){
        assertEquals("Hello!", g.sayHello()); }
}
```

2) Writing testcases for Calculation class:

```
import static org.junit.Assert.*;
import com.javatpoint.logic.*;
import org.junit.Test;

public class CalculationTest {
    @Test
    public void testFindMax(){
        assertEquals(4, Calculation.findMax(new int[]{1,3,4,2})); }
}
```

2.3.1 Creating Test Suite Class

Right click on com.logic.testcases go to New and select 'Others'. Then, select JUnit Test Suite and click next. Then, select the number of test classes and click on finish. The Test Suite class is created called *ArbitraryName*.java.

TestSuite class is an empty class. The empty class for the test suite is just a holder for the program to compile:

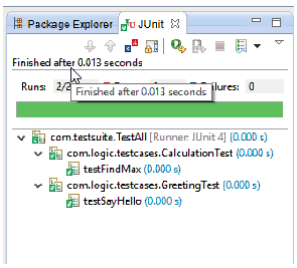


Figure 2: Corresponding test suite class

To run this example, right click on *ArbitraryName*.java → Run As → 1Junit Test.

3 JUnit Parameterized Tests and Theories

3.1 Objectives

- Running the same test multiple times using different input values.
- Making tests generalized.

3.2 Introduction

Parameterized tests in JUnit helps remove boiler plate test code and that saves time while writing test code. This is particularly useful during Enterprise Application Development with the Spring Framework. However, a common complaint is that when a parameterized test fails, it is very hard to see the parameters which caused it to fail. By properly naming the `@Parameters` annotation and great unit testing support that modern IDEs provide, such complains are quickly failing to hold grounds. Although Theories are less commonly used, they are powerful instruments in any programmers test toolkit. Theories not only makes your tests more expressive but it makes your test data become more independent of the code under testing. This will improve the quality of your code, since you are more likely to hit edge cases, which you may have previously overlooked.

3.3 Parameterized Tests

While testing, we usually want to execute a series of tests which differ only by input values and expected results. JUnit supports this functionality through parameterized tests. Parameterized tests allow a developer to run 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. There are five steps that you need to follow to create a parameterized test:

1. Annotate test class with `@RunWith(Parameterized.class)`.
2. Create a public static method annotated with `@Parameters` that returns a Collection of Objects (as Array) as test data set.
3. Create a public constructor that takes in what is equivalent to one "row" of test data.
4. Create an instance variable for each "column" of test data.
5. Create your test case(s) using the instance variables as the source of the test data.

One benefit to using parameters is that if one set of data fails, execution will just move to the next set of data instead of stopping the whole test.

As an example[1], if you are testing a method that validates email IDs, you should test it with different email ID formats to check whether the validations are getting correctly done. But testing each email ID format separately, will result in duplicate or boilerplate code. It is better to abstract the email ID test into a single test method and provide it a list of all input values and expected results. To see how parameterized test works, we will start with a class with two methods which we will put under test.

EmailIdUtility.java

```
1  import java.util.regex.Matcher;
2  import java.util.regex.Pattern;
3
4  public class EmailIdUtility {
5      public static String createEmailID(String firstPart,String secondPart){
6          String generatedId = firstPart+"."+secondPart+"@testdomain.com";
7          return generatedId;
8      }
9      public static boolean isValid(String email){
10         String regex = "^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@((\\[[0-9]{1,3}\\.[0-9]{1,3}
11         \\.[0-9]{1,3}\\.[0-9]{1,3}\\])|(([a-zA-Z\\-0-9]+\\.)+[a-zA-Z]{2,}))";
12         Pattern pattern = Pattern.compile(regex);
```

[1]The example is adopted from: https://dzone.com/

```
        Matcher m = pattern.matcher(email);
        return m.matches();
    }
}
```

The `EmailIdUtility` class above has two utility methods. The `createEmailID()` method accepts two String parameters and generates an email ID in a specific format. The format is simple: if you pass `mark` and `doe` as parameters to this method, it returns `mark.doe@testdomain.com`. The second `isValid()` method accepts an email ID as a String, uses regular expression to validate its format, and returns the validation result.

We will first test the `isValid()` method with a parameterized test. JUnit runs a parameterized test with a special runner, Parameterized and we need to declare it with the `@RuntWith` annotation. In a parameterized test class, we declare instance variables corresponding to the number of inputs to the test and the output. As the `isValid()` method under test takes a single String parameter and returns a boolean, we declare two corresponding variables. For a parameterized test, we need to provide a constructor, which will initialize the variables.

EmailIdValidatorTest.java

```
@RunWith(value = Parameterized.class)
public class EmailIdValidatorTest {
    private String emailId;
    private boolean expected;

    public EmailIdValidatorTest(String emailId, boolean expected) {
        this.emailId = emailId;
        this.expected = expected; }
```

We also need to provide a public static method annotated with `@Parameters` annotation. This method will be used by the test runner to feed data into our tests.

```
@Parameterized.Parameters(name= "{index}: isValid({0})={1}")
public static Iterable<Object[]> data() {
    return Arrays.asList(new Object[][]{
                    {"mary@testdomain.com", true},
                    {"mary.smith@testdomain.com", true},
                    {"mary_smith123@testdomain.com", true},
                    {"mary@testdomaindotcom", false},
                    {"mary-smith@testdomain", false},
                    {"testdomain.com", false} } ); }
```

The `@Parameters` annotated method above returns a collection of test data elements (which in turn are stored in an array). Test data elements are the different variations of the data, including the input as well as expected output needed by the test. The number of test data elements in each array must be the same with the number of parameters we declared in the constructor.

When the test runs, the runner instantiates the test class once for each set of parameters, passing the parameters to the constructor that we wrote. The constructor then initializes the instance variables we declared.

Notice the optional name attribute we wrote in the `@Parameters` annotation to identify the parameters being used in the test run. This attribute contains placeholders that are replaced at run time.

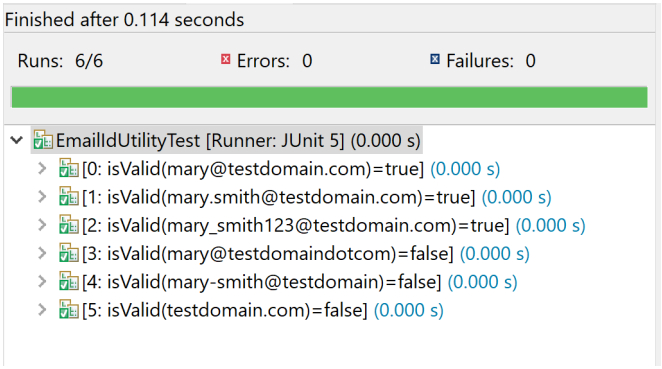
- {index}: The current parameter index, starting from 0.
- {0}, {1}, ...: The first, second, and so on, parameter value.

As an example, for the parameter {`mary@testdomain.com`, true}, then {0} = `mary@testdomain.com` and {1} = true. Finally, we write the test method annotated with `@Test`. The complete code of the parameterized test is as follows:

EmailIdValidatorTest.java

```java
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import static org.hamcrest.CoreMatchers.*;
import java.util.Arrays;
import static org.junit.Assert.*;

@RunWith(value = Parameterized.class)
public class EmailIdValidatorTest {

    private String emailId;
    private boolean expected;

    public EmailIdValidatorTest(String emailId, boolean expected) {
        this.emailId = emailId;
        this.expected = expected; }

    @Parameterized.Parameters(name= "{index}: isValid({0})={1}")
    public static Iterable<Object[]> data() {
        return Arrays.asList(new Object[][]{
                        {"mary@testdomain.com", true},
                        {"mary.smith@testdomain.com", true},
                        {"mary_smith123@testdomain.com", true},
                        {"mary@testdomaindotcom", false},
                        {"mary-smith@testdomain", false},
                        {"testdomain.com", false} } ); }
    @Test
    public void testIsValidEmailId() throws Exception {
        boolean actual= EmailIdUtility.isValid(emailId);
        assertEquals(expected, actual);
        //or: assertThat(actual, is(equalTo(expected)));
        }
}
```

The output on running the parameterized test is as follows:



3.4 JUnit Theories

In a parameterized test, the test data elements are statically defined and the programmer is responsible for figuring out what data is needed for a particular range of tests. Thus, we want to make tests more generalized. Instead of testing for specific values, you might require to test for some wider range of acceptable input values. For this scenario, JUnit provides theories.

A theory is a special test method that a special JUnit runner (Theories) executes. To use the runner, annotate your test class with the `@RunWith(Theories.class)` annotation. The Theories runner executes a theory against several data inputs called data points. A theory is annotated with `@Theory`, but unlike normal `@Test` methods, a `@Theory` method has parameters. In order to fill these parameters with values, the Theories runner uses values of the data points having the same type.

3.5 Parameterized Tests vs. Theories

- Class annotated with `@RunWith (Parameterized.class)` versus `@RunWith(Theories.class)`.

- Test inputs are retrieved from a static method returning Collection and annotated with `@Parameters` versus static fields annotated with `@DataPoints` or `@DataPoint`.
- Inputs are passed to the constructor (mandatory) and used by the test method versus inputs are directly passed to the test method.
- Test method is annotated with `@Test` and does not take arguments versus Test method is annotated with `@Theory` and may take arguments.

There are two types of data points. You use them through the following two annotations:

- `@DataPoint`: Annotates a field or method as a single data point. The value of the field or that the method returns will be used as a potential parameter for theories having the same type.
- `@DataPoints`: Annotates an array or iterable-type field or method as a full array of data points.

The values in the array or iterable will be used as potential parameters for theories having the same type. Use this annotation to avoid single data point fields cluttering your code. Note that all data point fields and methods must be declared as **`public`** and **`static`**.

```
1  @DataPoint
2  public static String name="mary";
3
4  @DataPoints
5  public static String[] names() {
6      return new String[]{"first","second","abc","123"}; }
```

In the code example above, we annotated a String field with the `@DataPoint` annotation and a `names()` method that returns a `String[]` with the `@DataPoints` annotation.

3.5.1 Creating a JUnit Threory

Recall the `createEmailID()` method that we wrote earlier, "The `createEmailID()` method accepts two String parameters and generates an email ID in a specific format." A test theory that we can establish is "Provided stringA and stringB passed to `createEmailID()` are non-null, it will return an email ID containing both stringA and stringB." This is how we can represent the theory.

```
1  @Theory
2  public void testCreateEmailID(String firstPart, String secondPart) throws Exception {
3   String actual= EmailIdUtility.createEmailID(firstPart,secondPart);
4   assertThat(actual, is(allOf(containsString(firstPart), containsString(secondPart))));
5  }
```

The `testCreateEmailID()` theory we wrote accepts two String parameters. At run time, the Theories runner will call `testCreateEmailID()` passing every possible combination of the data points we defined of type String. For example (mary,mary), (mary,first), (mary,second), and so on.

3.5.2 Assumptions

It is very common for theories NOT to be valid for certain cases. You can exclude these from a test using assumptions, which basically means "do not run this test if these conditions do not apply". In our theory, an assumption is that the parameters passed to the `createEmailID()` method under testing are non-null values.

If an assumption fails, the data point is silently ignored. Programmatically, we need to use `import static org.junit.Assume.*;`, and then add assumptions to theories through one of the many methods of the Assume class as follows:

Function & Arguments	Description
assumeTrue(boolean b) assumeTrue("message", boolean b)	If an expression evaluating to false, the test will halt and be ignored.
assumeFalse(boolean b) assumeFalse("message", boolean b)	If an expression evaluating to true, the test will halt and be ignored.
assumeThat(actual, matcher) assumeThat("message", actual, matcher)	To assume that actual satisfies the condition specified by matcher.
assumeNotNull(Object... objects)	If one or more null elements in objects, the test will halt and be ignored.
assumeNoException(Throwable e) assumeNoException("message", Throwable e)	Attempts to halt the test and ignore it if Throwable e is not null.

Here is our modified theory with assumptions:

```
@Theory
public void testCreateEmailID(String firstPart, String secondPart) throws Exception {
    assumeNotNull(firstPart, secondPart);
    assumeThat(firstPart, notNullValue());
    assumeThat(secondPart, notNullValue());
    String actual= EmailIdUtility.createEmailID(firstPart,secondPart);
     assertThat(actual, is(allOf(containsString(firstPart),  containsString(secondPart))));
}
```

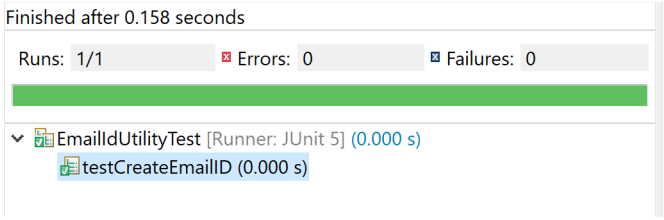
In the code above, we used `assumeNotNull` because we assume that the parameters passed to `createEmailID()` are non-null values. Therefore, even if a null data point exists and the test runner passes it to our theory, the assumption will fail and the data point will be ignored. The two `assumeThat` we wrote together performs exactly the same function as `assumeNotNull`. We have included them only for demonstrating the usage of `assumeThat`, which you can see is very similar to `assertThat`. The following is the complete code using a theory to test the `createEmailID()` method.

EmailIDCreatorTest.java

```
import org.junit.Test;
import org.junit.experimental.theories.DataPoint;
import org.junit.experimental.theories.DataPoints;
import org.junit.experimental.theories.Theories;
import org.junit.experimental.theories.Theory;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import java.util.Arrays;

import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.CoreMatchers.containsString;
import static org.junit.Assert.*;
import static org.junit.Assume.assumeNotNull;
import static org.junit.Assume.assumeThat;

@RunWith(Theories.class)
public class EmailIDCreatorTest {

    @DataPoints
    public static String[] names() {
        return new String[]{"first","second","abc","123",null}; }
    @DataPoint
    public static String name="mary";
    /*Generated Email ID returned by EmailIdUtility.createEmailID must
    contain first part and second part passed to it*/
    @Theory
    public void testCreateEmailID(String firstPart, String secondPart) throws Exception {
        System.out.println(String.format("Testing with %s and %s", firstPart, secondPart));
        assumeNotNull(firstPart, secondPart);
        /*Same assumptions as assumeNotNull(). Added only to demonstrate
        usage of assertThat*/
```

```
32          assumeThat(firstPart, notNullValue());
33          assumeThat(secondPart, notNullValue());
34          String actual= EmailIdUtility.createEmailID(firstPart,secondPart);
35          System.out.println(String.format("Actual: %s \n", actual));
36          assertThat(actual, is(allOf(containsString(firstPart), containsString(secondPart)))); }
37  }
```

In the test class above, we have included `null` as a data point in the return statement of Line 21 for our assumptions and couple of `System.out.println()` statements to trace how parameters are passed to theories at run time.




```
1   Testing with mary and mary
2   Actual: mary.mary@testdomain.com
3
4   Testing with mary and first
5   Actual: mary.first@testdomain.com
6
7   Testing with mary and second
8   Actual: mary.second@testdomain.com
9
10  Testing with mary and abc
11  Actual: mary.abc@testdomain.com
12
13  Testing with mary and 123
14  Actual: mary.123@testdomain.com
15
16  Testing with mary and null
17  Testing with first and mary
18  Actual: first.mary@testdomain.com
19
20  Testing with first and first
21  Actual: first.first@testdomain.com
22
23  Testing with first and second
24  Actual: first.second@testdomain.com
25
26  Testing with first and abc
27  Actual: first.abc@testdomain.com
28
29  Testing with first and 123
30  Actual: first.123@testdomain.com
31
32  Testing with first and null
33  Testing with second and mary
34  Actual: second.mary@testdomain.com
35
36  Testing with second and first
37  Actual: second.first@testdomain.com
38
39  Testing with second and second
40  Actual: second.second@testdomain.com
41
42  Testing with second and abc
43  Actual: second.abc@testdomain.com
44
45  Testing with second and 123
46  Actual: second.123@testdomain.com
47
48  Testing with second and null
49  Testing with abc and mary
```

```
Actual: abc.mary@testdomain.com

Testing with abc and first
Actual: abc.first@testdomain.com

Testing with abc and second
Actual: abc.second@testdomain.com

Testing with abc and abc
Actual: abc.abc@testdomain.com

Testing with abc and 123
Actual: abc.123@testdomain.com

Testing with abc and null
Testing with 123 and mary
Actual: 123.mary@testdomain.com

Testing with 123 and first
Actual: 123.first@testdomain.com

Testing with 123 and second
Actual: 123.second@testdomain.com

Testing with 123 and abc
Actual: 123.abc@testdomain.com

Testing with 123 and 123
Actual: 123.123@testdomain.com

Testing with 123 and null
Testing with null and mary
Testing with null and first
Testing with null and second
Testing with null and abc
Testing with null and 123
Testing with null and null
```

In the output above, notice that whenever a null value is being passed to the theory, the remaining part of the theory after `assumeNotNull` does not execute.

4 Assignments

Note that you may consider using other annotations such as `@Before`, `@BeforeClass`, `@After`, and `@AfterClass` whenever they are needed.

Q1: In this question, you will first implement a class *ArrayMult* which carries out point-wise multiplication on two arrays of non-negative integers, and second, implement a JUnit test *ArrayMultTest* to check that your code does what it is supposed to do. *ArrayMult* should define a method `mult()` with the following declaration: `public int[] mult(int[] array1, int[] array2) {}`

The method should be defined so that *array1* and *array2* can be of different lengths. Let *minlen* be the length of the shortest array (or of both arrays if they are the same size), and let *longArray* be whichever is the longest of the two. Then the following properties hold of the output array *outArray*:

- The length of *outArray* is the length of *longArray*.
- For all $i <$ *minlen*, `outArray[i] = array1[i] * array2[i]`.
- For all $j >=$ *minlen*, `outArray[j] = longArray[j]`.

Implement the *ArrayMult* class with method `mult()` as specified, and implement *ArrayMultTest* class. Submit both of these Java files.

Q2: We have a Triangle class that contains a *calculateArea* method. We want to test this method; the mathematical equation is INCORRECT as Heron's formula. Consider the following instructions and answer the questions.

```
1  import static org.junit.Assert.assertTrue;
2  import org.junit.Test;
3  public class Triangle {
4      public int side1, side2, side3;
5      public Triangle(int side1, int side2, int side3) {
6          this.side1 = side1;
7          this.side2 = side2;
8          this.side3 = side3; }
9      public double calculateArea () {
10         //Heron's Formula for area of a triangle
11         double s = (side1 + side2 + side3) * 0.5;
12         System.out.println("\t s=" + s);
13         double result = Math.sqrt(s * (side1 - s) * (side2 - s) * (side3 - s));
14         System.out.println("\t result=" + result);
15         return result; }
16 }
```

- Create a JUnit Test Case which should contain initialization function to be run before test methods for three triangles $t1$, $t2$, and $t3$ with the sides $(3, 4, 5)$, $(5, 4, 3)$, and $(8, 5, 5)$, respectively.
- Use your understanding of JUnit to debug and fix the issues regarding our implementation of Heron's formula. Feel free to look up the original formula as part of your investigations.
- Write at least three test methods for calculating the area of $t1$, $t2$, and $t3$.
- Write one test method to verify whether or not the area of $t1$ and $t2$ are the same.
- Make sure you have both "positive" and "negative" test cases: the ones that show the code serves its intended usage, and others that show it cannot be misused.
- What will happen if the user creates a triangle with `new Triangle(3,4,100)`? Is this something you should or can test in JUnit? How might you do so? It is possible to add any normal bit of code to your testing class.
- Submit the modified program and its testing program.

Q3: Consider the following single class containing a static method that main function can call. The method is trying to utilize a regular expression to check the phone numbers to be valid. Follow the instructions below and answer the questions:

```
import java.util.*;
public class RE {
    public static boolean checkPhoneNumber(String s) {
        return s.matches("(\d{3}) \d{3} - \d{4}"); }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a phone number: ");
        String input = sc.nextLine();
        boolean wasPhoneNum = checkPhoneNumber(input);
        System.out.println("\nThat was"+(wasPhoneNum? "" : "n't")+" a phone number."); }
}
```

1. Debug and fix the compilation error. It relates to the issues of representing the regular expressions inside a Java String. You can easily search for "regular expressions in Java programming" to learn more.
2. What does "\d" mean to Java when you are creating a String?
3. The task is to verify whether a phone number is valid. Test the current regular expression with the following inputs as the "VALID" inputs: (123)123 – 1234 and (123) 456 – 7890.
    - Why are not these working? What is wrong with our regex? Test another case: 123 123 – 1234
4. As you may see, parentheses may not be considered. They are special symbols. Modify the regular expression (escape the parentheses), and test your program again with these inputs: (123)123 – 1234 and (123) 456 – 7890.
5. Again, fix the regular expression to allow white space anywhere except between adjacent number characters.
6. Write the testing program using at least three different test cases in separated test methods.

Q4: Create and submit a test suit class with its test programs to be run in a suit mode from the programs of the Questions 2 and 3.

Q5: Consider the following Java program for computing Fibonacci numbers based on a specific index:

```
public class Fibonacci {
    public static int compute(int n) {
        int result = 0;
        if (n <= 1) {
            result = n;
        } else {
            result = compute(n - 1) + compute(n - 2); }
        return result; }
}
```

Write a testing program for the above Fibonacci program to test the computation of the first 10 Fibonacci numbers (index from 0 to 9) using **parameterized testing with constructor**.

Q6: Write a Java program containing a class named `PrimeNumberChecker` for checking if an integer number is a prime number and returning a boolean result. Then, write a testing program for the above PrimeNumberChecker program to test if integer numbers {2, 6, 19, 22, 23} are prime numbers or not, using **parameterized testing with constructor**.

Q7: Consider the following mathematical theory: For all $a, b > 0$, the following is true: $a + b > a$ and $a + b > b$. This statement holds for every element (or combination of elements) set. Using **JUnit Theories**:

1. Write a testing program to check the truth of this mathematical statement for every pair (every possible combination) of a and b from the value set `val={1, 2, 307, 400567}`.
2. Add another theory to be covered in your testing program in which the commutative feature of every possible pair of the integers a and b should be true: $a + b = b + a$. Use the same value set *val* as in Task 1 for this statement as well.
3. Consider a new value set `newval={0, -1, -10, -1234, 1, 10, 6789}` as your data points for input values in your testing program for every possible pair of a and b. What will be your results, and why? Demonstrate and justify your achieved results.
4. Considering the value set *newval*, rectify your previous testing program using **Assumption** to be able to check the truth of the aforementioned mathematical theories over every possible pair of a and b from the value set *newval*.
5. Consider another value set:

```
1       {0, -1, -10, -1234, 1, 10, 6789, Integer.MAX_VALUE, Integer.MIN_VALUE}
```

   as your data points for input values in your testing program for every possible pair of a and b. What will be your results, and why? Demonstrate and justify your achieved results.

Q8: Using **JUnit Theories**, rewrite the testing programs you have already provided for Question 5 and 6.

5 Submission and Marking Instructions

5.1 Submissions

Once all required tasks are completed, you should submit your lab assignment. Follow the instructions below for a valid submission:

- After checking the accuracy and completeness of your assignment tasks, you should export and submit the FULL Eclipse project/folder as a ZIP file containing packages and source code files (for implementation and coding tasks/questions).

  - Individual files (e.g. java or class files) will NOT be accepted as a valid submission since your submitted package or Java project should be run completely and successfully by itself.

- You MUST submit a single PDF file containing required description or explanation for each of the assignment tasks/questions separately including any required justification, graphs, diagrams, test requirements/cases, calculation, pictures/snapshots from tools or IDE, test results, and so forth.

- The **submission deadline** for this lab (Lab I) is the corresponding lab session in **Week 4**.

- The lab demo and questioning-answering will be held during the lab sessions of the corresponding submission weeks.

5.2 Marking Scheme

- This lab (Lab I) constitutes 4% of your entire grade for this course.

- All assigned tasks/questions in each lab have the same grade.

- The grade for each lab is constituted from 50% for the lab submissions, 10% for the lab attendance, and 40% for demo and questioning-answering during the lab sessions.

- Note that all the labs constitute 25% of your entire grade for this course.