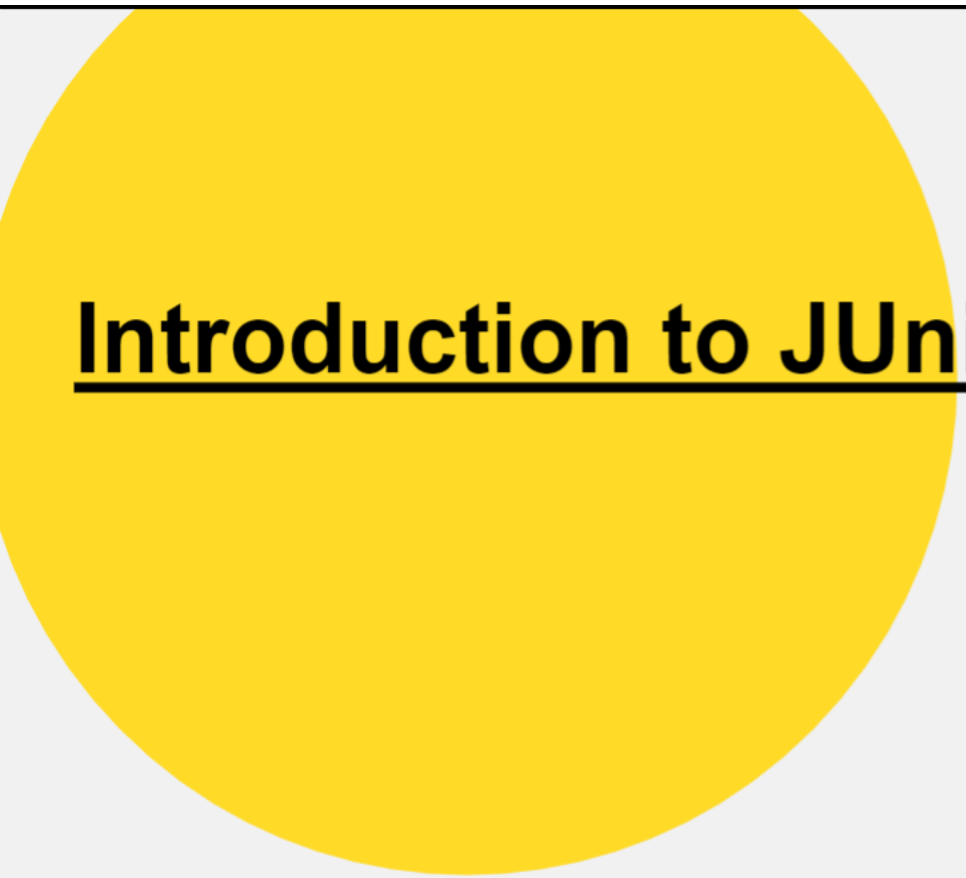
COCSIT

JUNIT

Introduction to JUnit





JUNIT

- JUnit is an open source testing framework for Java
- JUnit test cases are Java classes that contain one or more unit test methods
- It is a simple framework for creating automated unit tests

JUNIT(Contd).

- These tests are grouped into test suites
  - JUnit tests are pass/fail tests explicitly designed to run without human intervention
  - JUnit can be integrated with several IDEs, including Eclipse
    - The JUnit distribution can be downloaded as a single jar file from *http://www.junit.org*
    - It has to be kept in the classpath of the application to be tested

Junit - an open source testing framework

JUnit has these advantages:

- it's simple to use
- it can test a single class at a time, or a suite of tests can be created for a group of classes
- it *greatly* increases your confidence in the correctness of your code
- it often improves the design of the class you are testing - since you spend more time thinking about how an object is actually *used*, instead of its implementation, defects in its interface become more obvious.

JUnit with Eclipse

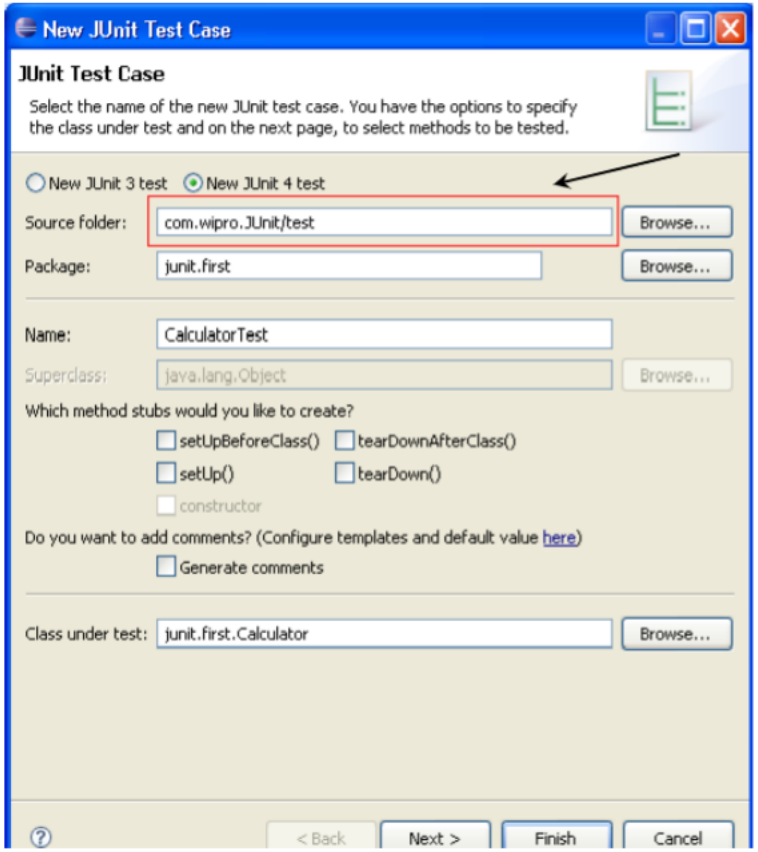



JUnit with Eclipse

1. Create a new Project com.wipro.JUnit
2. Add JUnit.jar to the Classpath
3. Right click the Project and create a new Source folder called 'test'
4. Create a new Java class called Calculator in a package junit.first
5. Add 2 methods add and sub to the Calculator class which does addition and subtraction of 2 numbers respectively

```
package junit.first;
public class Calculator {
public int add(int x,int y)
{ return x+y; }
public int sub(int x,int y)
{ return x-y; } }
```

JUnit with Eclipse (Contd.).



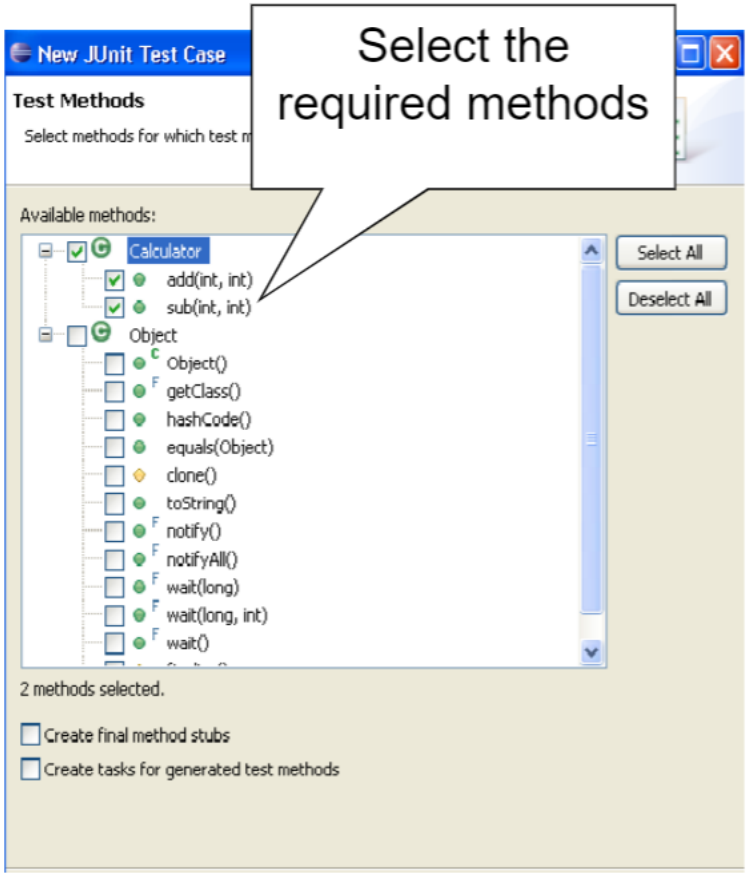
- Right click on the Calculator class in the Package Explorer and

  select New->JUnitTestCase

- select "New JUnit4 test“

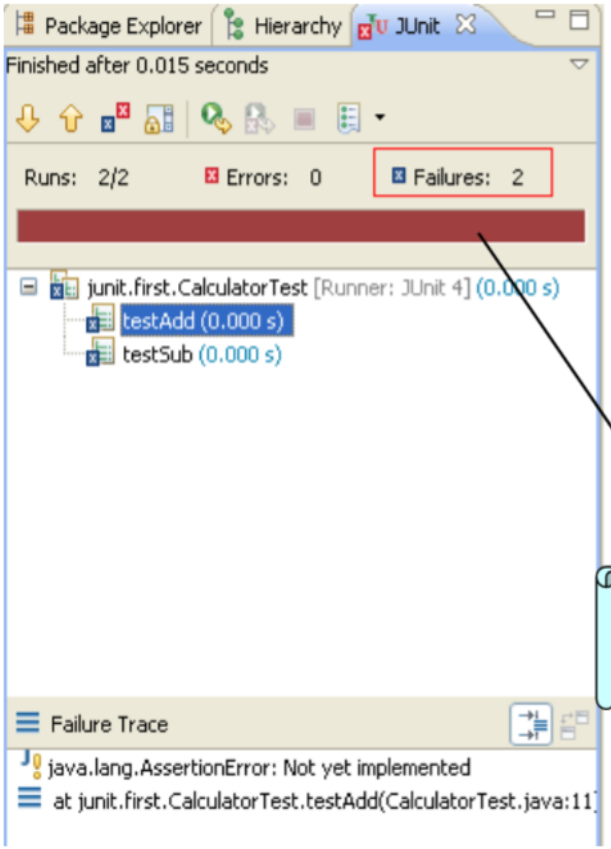
- set the source folder to "test“ –
  the test class gets created here

JUnit with Eclipse (Contd.).



- Press "Next" and select the methods you want to test

JUnit with Eclipse (Contd.).



- Right click on CalculatorTest class and select Run-As → JUnit Test
- The results of the test will be displayed in JUnit view
- This is because the testAdd and testSub are not implemented correctly

Brown color indicates failure

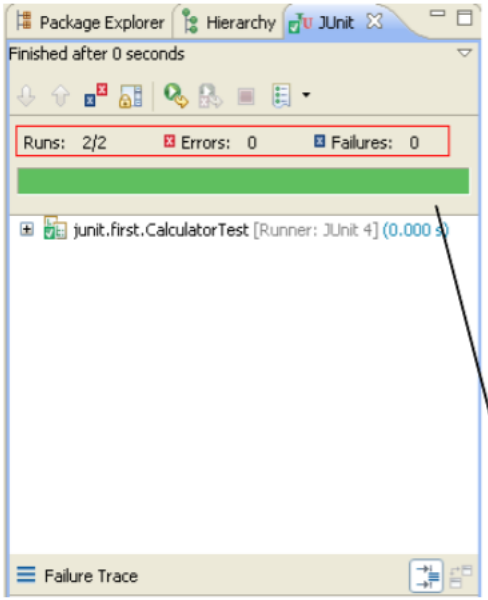
How to write a JUnit test method?

- All the test methods should be marked with the JUnit annotation - @org.junit.Test
- All the JUnit test methods should be "public" methods
- The return type of the JUnit test method must be “void"
- The test method need not start with the test keyword
- Here is a simple JUnit test method:

```
@Test
public void testAdd()
{
    Calculator c=new Calculator();
    assertEquals("Result",5,c.add(2,3));
}
```

JUnit with Eclipse

- Now let's provide implementation to the code and run the test again



Green color indicates pass

```
package junit.first;
import static org.junit.Assert.*;
import org.junit.Test;
public class CalculatorTest {
@Test
public void testAdd() {
Calculator c=new Calculator();
assertEquals(5,c.add(2,3));
}
@Test
public void testSub() {
Calculator c=new Calculator();
assertEquals(20,c.sub(100,80));
} }
```

JUnit with Eclipse(Contd.).

- Unit tests are implemented as classes with test methods. Each test method usually tests a single method of the target class. Sometimes, a test method can test more than one method in the target class, and sometimes, if the method to test is big, you split the test into multiple test methods.

- The unit test class is an ordinary class, with two methods, tesAdd() and testSub. Notice how this method is annotated with the JUnit annotation @Test. This is done to signal to the unit test runner, that this is method represents a unit test, that should be executed. Methods that are not annotated with @Test are not executed by the test runner.

JUnit with Eclipse(Contd.).

- Inside the testAdd() method an instance of Calculator is created. Then it's add() method is called with two integer values.

- Finally, the assertEquals() method is called. It is this method that does the actual testing. In this method we compare the output of the called method (add()) with the expected output.

- If the two values are equal, nothing happens. The assertEquals() method returns normally. If the two values are not equal, an exception is thrown, and the test method stops executing here.

- The assertEquals() method is a statically imported method, which normally resides in the org.junit.Assert class. Notice the static import of this class at the top of MyUnitTest. Using the static import of the method is shorter than writing Assert.assertEquals().

Quiz

- What is meant by **import static org.junit.Assert.*;**?
- What is the content of the JAR file? How you will add a Jar file to your project?
- What is meant by SDLC? What are the phases of SDLC?
- Testing is done by Testing team; Then, why a programmer like you should worry about testing?

Assert methods and Annotations




Assert methods with JUnit

- **assertArrayEquals()**
  - Used to test if two arrays are equal to each other

```
int[] expectedArray = {100,200,300};
int[] resultArray =  myClass.getTheIntArray();
assertArrayEquals(expectedArray, resultArray);
```

- **assertEquals()**
  - It compares two objects for their equality

```
String result = myClass.concat("Hello", "World");
assertEquals("HelloWorld", result);
assertEquals("Reason for failure","HelloWorld",result);
```

Will get printed if the test will fail

Note: All assert methods are static methods, hence one has to use static import
import static org.junit.Assert.*;

Assert methods with JUnit

- **assertArrayEquals()**

Used to test if two arrays are equal to each other. If the arrays are equal, the assertArrayEquals() will proceed without errors. If the arrays are not equal, an exception will be thrown, and the test aborted. Any test code after the assertArrayEquals() will not be executed.

- **assertEquals**

The assertEquals() method can compare any two objects to each other. If the two objects compared are not same, then an AssertionError will be thrown.

Assert methods with JUnit

- The new assertEquals methods use Autoboxing, and hence all the assertEquals(primitive, primitive) methods will be tested as assertEquals(Object, Object).
- This may lead to some interesting results. For example autoboxing will convert all numbers to the Integer class, so an Integer(10) may not be equal to Long(10).
- This has to be considered when writing tests for arithmetic methods.
- For example,

The following Calc class and it's corresponding test CalcTest will give you an error.

```
public class Calc {
 public long add(int a, int b) {
  return a+b;
 }
}
```

Assert methods with Junit(Contd.).

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
 public class CalcTest {
 @Test
 public void testAdd() {
  assertEquals(5, new Calc().add(2, 3));
 }}
```

- You will end up with the following error.

  java.lang.AssertionError: expected:<5> but was:<5>

- This is due to autoboxing. By default all the integers are cast to Integer, but we were expecting long here. Hence the error.
- In order to overcome this problem, it is better if you type cast the first parameter in the **assertEquals** to the appropriate return type for the tested method as follows

```
assertEquals((long)5, new Calc().add(2, 3));
```

Assert methods with JUnit (Contd.).

- **assertTrue() , assertFalse()**
  - Used to test whether a method returns true or false

```
assertTrue (testClass.isSafe());
assertFalse(testClass.isSafe());
```

- **assertNull(),assertNotNull()**
  - Used to test a variable to see if it is null or not null

```
assertNull(testClass.getObject());
assertNotNull(testClass.getObject());
```

- **assertSame() and assertNotSame()**
  - Used to test if two object references point to the same object or not

```
String s1="Hello";
String s2="Hello";
assertSame(s1,s2); ->true
```

Assert methods with JUnit (Contd.).

- **assertTrue() , assertFalse()**
  - If the isSafe() method returns true, the assertTrue() method will return normally. Else an exception will be thrown, and the test will stop there.
  - If the isSafe() method returns false, the assertFalse() method will return normally. Else an exception will be thrown, and the test will stop there.
- **assertNull(),assertNotNull()**
  - If the testClass.getObject() returns null, the assertNull() method will return normally, else the assertNull() method will throw an exception, and the test will be stopped.
  - The assertNotNull() method works oppositely of the assertNull() method. It throws an exception if a null value is passed to it, and returns normally if a non-null value is passed to it.
- **assertSame(),assertNotSame()**
  - Used to check if two object references point to the same object or not.

Annotations

- Fixtures
  - The set of common resources or data that you need to run one or more tests
- @Before
  - It is used to call the annotated function before running each of the tests
- @After
  - It is used to call the annotated function after each test method

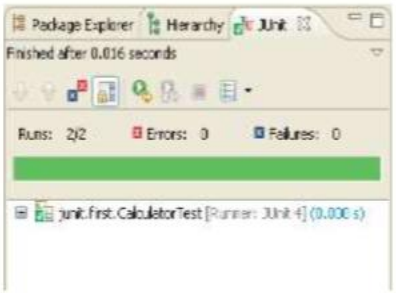
```
public class CalculatorTest {
Calculator c=null;

@Before
public void before()
{
System.out.println("Before Test");
 c=new Calculator();
}
@After
public void after()
{
 System.out.println("After Test");
}

@Test
public void testAdd() {
System.out.println("Add function");
assertEquals("Result",5,c.add(2,3));
}
@Test
public void testSub() {
System.out.println("Sub function");
assertEquals("Result",20,c.sub(100,80))

} }
```

O/P :
Before Test
Add function
After Test
Before Test
Sub function
After Test



Annotations(Contd.).

- Let's consider the case in which each of the tests that you design needs a common set of objects. One approach can be to create those objects in each of the methods. Alternatively, the JUnit framework provides two special methods, setUp() and tearDown(), to initialize and clean up any common objects. This avoids duplicating the test code necessary to do the common setup and cleanup tasks. These are together referred to as *fixtures*. The framework calls the setup() before and tearDown() after each test method—thereby ensuring that there are no side effects from one test run to the next.

- In Junit 4.x the @Before annotation does the role of the setUp() method and the @After annotation performs the role of the tearDown() method of JUnit 3.x

Annotations (Contd.).

- @BeforeClass
  - The annotated method will run before executing any of the test method
  - The method has to be static
- @AfterClass
  - The annotated method will run after executing all the test methods
  - The method has to be static

O/P :
Before Test
Add function
Sub function
After Test

```
public class CalculatorTest {
static Calculator c=null;
@BeforeClass
public static void before()
{
System.out.println("Before Test");
c=new Calculator();
}

@AfterClass
public  static void after()
{
System.out.println("After Test");
}

@Test
public void testAdd() {
System.out.println("Add function");
assertEquals("Result",5,c.add(2,3));
}
@Test
public void testSub() {
System.out.println("Sub function");
assertEquals("Result",20,c.sub(100,80));
}
}
```

Annotations (Contd.).

- **@Ignore**
  - Used for test cases you wanted to ignore
  - A String parameter can be added to define the reason for ignoring

```
@Ignore("Not Ready to Run")
@Test
public void testComuteTax() { }
```

- **@Test**
  - Used to identify that a method is a test method

Annotations (Contd.).

Two optional parameters are supported by Test Annotation.

- The first optional parameter 'expected' is used to declare that a test method should throw an exception. If it doesn't throw an exception or if it throws a different exception than the one declared the test fails.
- For example, the following test succeeds:

```
@Test(expected=IndexOutOfBoundsException.class)
 public void checkOutOfBounds()
{
new ArrayList<String>().get(1);
 }
```

- The second optional parameter, '**timeout',**causes a test to fail if it takes longer than a specified amoun t of clock time (measured in milliseconds). The following test fails:

```
@Test(timeout=1000)
 public void infinityCheck()
{
 while(true);
}
```

Annotations (Contd.).

- **Timeout**
  - It defines a timeout period in miliseconds with “timeout” parameter
  - The test fails when the timeout period exceeds.

```
@Test (timeout = 1000)
 public void testinfinity() {
 while (true)
 ;
 }
```

Quiz

- From tester point of view, What is the use of @Ignore annotation?
- From tester point of view, What is the use of

```
@Test (timeout = 1000)
```

Test Suite




Test Suite

- Test Suite is a Convenient way to group together tests that are related
- Used to bundle a few unit test cases and run it together
- Annotations used for this
  - @RunWith
    - Used to invoke the class which is annotated to run the tests in that class
  - @Suite
    - Allows you to manually build a suite containing tests from many classes

User Defined Class 1

```
package junit.first;

public class Stringmanip {
String datum;

    public Stringmanip(String datum) {
        this.datum = datum;
    }

    public String upperCase() {
        return datum.toUpperCase();
    }

}
```

Test Case for User Defined Class 1

```
package junit.first;
import junit.first.Stringmanip.*;
import java.util.*;
import org.junit.Test;
import org.junit.runners.*;
import org.junit.runner.RunWith;
import static org.junit.Assert.*;
        @RunWith(Parameterized.class)
   public class StringmanipTest2
    {
     // Fields
      private String datum;
      private String expected;
      public StringmanipTest2(String datum, String expected)       {
                this.datum = datum;
                this.expected = expected;
       }
```

Test Case for User Defined Class 1 (Contd.).

```
@Parameters
  public static Collection<Object[]> generateData()
{                    Object[][] data = new  Object[][]      {
                { "Smita", "SMITA" },
                      { "smita", "SMITA" },
                      { "SMitA", "SMITA" }
        };
   return Arrays.asList(data);
}

  @Test
  public void testUpperCase()
  {
     Stringmanip s = new Stringmanip(this.datum);
    String actualResult = s.upperCase();
    assertEquals(actualResult, this.expected);
  }
}
```

In this example, the parameter generator returns a List of arrays.

Each row has two elements:
{ input_data, expected_output }.
These data are hardcoded into the class, but they could be
generated in any way you like.

User Defined Class 2

```
package junit.first;

public class Calc {

     public int add( int v1, int v2)     {
          return v1+v2;
     }

     public int sub( int v1, int v2)     {
          return v1-v2;
     }

// You can add more functions here as needed..

}
```

Test Case for User Defined Class 2

```
package junit.first;
import static org.junit.Assert.*;
import org.junit.Test;
public class CalcTest {
      Calc c = new Calc();

      @Test
      public void testAdd() {
      assertEquals(5, c.add(10,-5));
      assertEquals(5, c.add(10,-5));
      assertEquals(5, c.add(20,-15));
      assertEquals(5, c.add(0,5));
      }
```

Test Case for User Defined Class 2

```
@Test
    public void testSub() {
        assertEquals(5, c.sub(10,5));
        assertEquals(95, c.sub(100,5));
        assertEquals(5, c.sub(20,15));
        assertEquals(5, c.sub(10,5));
    }
}
```

Test Suite

- In JUnit, both **@RunWith** and **@Suite** annotation are used to run the suite test.
- When a class is annotated with @RunWith,
  JUnit will invoke the class it references to run the tests in that class.
- Using Suite as a runner allows you to manually build a suite containing tests from many classes.

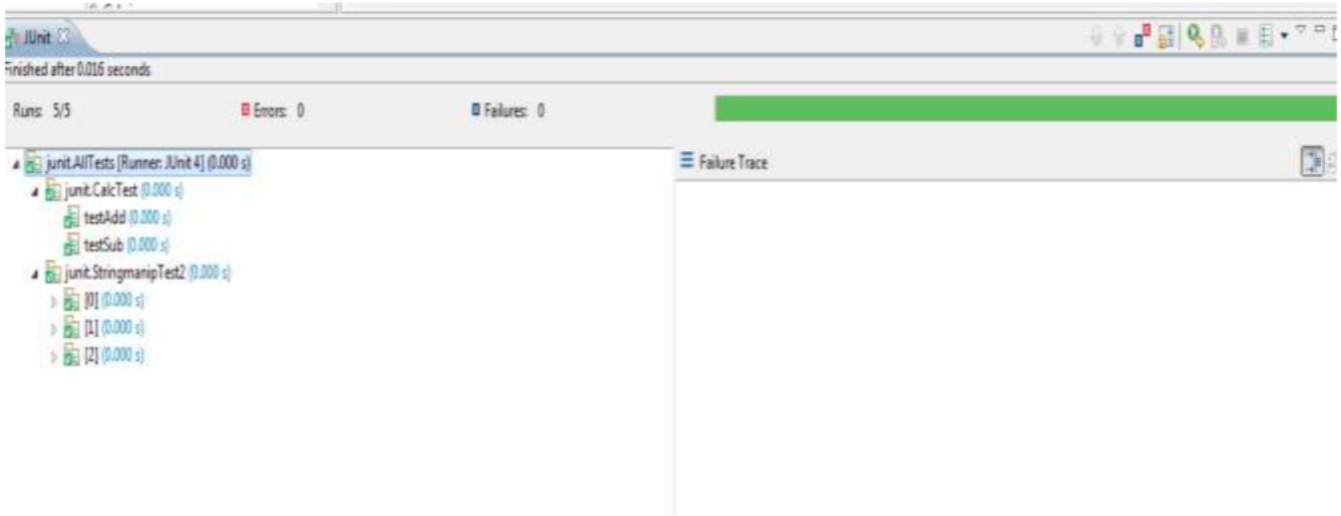
```
@RunWith(Suite.class)
@Suite.SuiteClasses({
            CalcTest.class,
            StringmanipTest2.class
    })

public class AllTests
{
}
```

Note >>
Test Classes
which are defined in previous pages are included here…

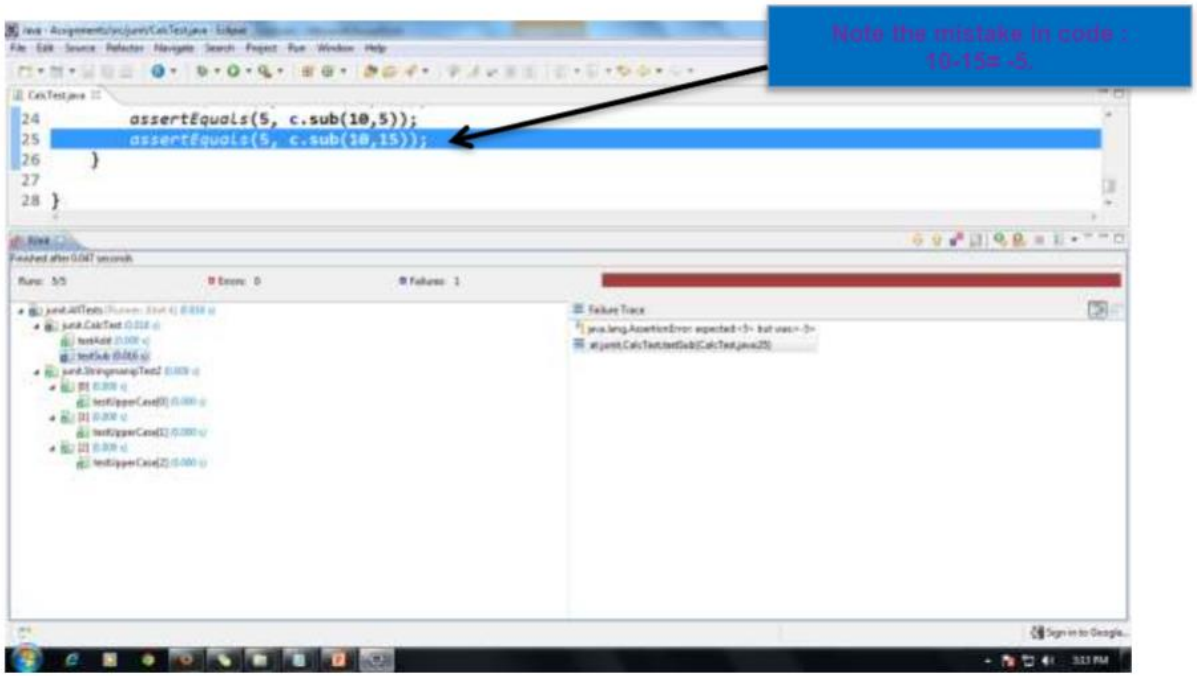
Test Suite(Contd.).

- When all the test cases are executed successfully, it shows **green color** signal as shown below.



Test Suite(Contd.).

- When any one test cases fails, it shows **Brown color** signal as shown below.



Quiz

1. Which of the following annotations has to be used before each of the test method?
   a. @Before
   b. @BeforeClass
   c. @After
   d. None of the above

None of the above

2. Which of the following are true?
   a. All assert methods are static methods
   b. The JUnit test methods can be private
   c. The JUnit test methods should start with the test keyword
   d. All of the above true

All assert methods are static methods


Introduction to Mockito

Mockito

- Mockito is open source
- It is a mocking framework for testing Java Applications using Java – based library
- Mocking implementation of interfaces or classes enables the isolation during unit testing
- Side effects of other classes are controlled by mocking
  - Full service class implementation is not complete
  - DB connection to actual db is not yet up but test needs to be done on other behaviours

Mocking

- Mocking is creation of mock objects during unit testing

- Mock objects are used in situations like
  - To just record the interactions with the systems and validate it
  - To fill dummy parameter list
  - Used to connect with local sample internal database instead of actual database etc

Simple steps to test the code using mockito

- Mock away external dependencies by creating mock objects
- Inserting the mock codes into the test
- Execute the test
- Validate the code

Sample mock object

- Let us consider EmployeeDB class. In order to mock it we can use

```
empDB = mock(EmployeeDB.class);
```

- Or by using annotation

```
@Mock
EmployeeDB empDB;
```

- In both the cases we don't instantiate the object instance, its been mocked
- We can mock the behaviors with When Then Return statement

```
when(empDB.insert(empObj)).thenReturn(1);
when(empDB.insert(deptObj)).thenReturn(0);
```

For Further Reading and full demo visit:

https://www.tutorialspoint.com/mockito/mockito_junit_integration.htm

Thank You