



## CST-239 Activity 7 Guide

### Contents

Part 1: Basic Unit Tests.....	1
Part 2: Parameterized Unit Tests.....	4
Part 3: Advanced Unit Tests .....	7

### Part 1: Basic Unit Tests

#### Overview

##### Goal and Directions:

In this activity you will setup Eclipse to run JUnit in a project for a simple calculator application. You will then write unit tests to validate the functionality of the calculator.

Reference the following tutorials to support this activity:

1. <http://www.junit.org/>
2. <https://www.tutorialspoint.com/junit/index.htm>
3. <http://www.vogella.com/tutorials/JUnit/article.html>
4. <https://junit.org/junit5/docs/current/user-guide/>

The following are the tasks you need to complete for this activity:

#### Execution

- 1) Create a new Java project named *topic7-1a*.
- 2) Create a new Java class named *Calculator* (with a *main()*) in the *junit* package:
  - a. Add a public method named *add(int a, int b)* that returns an int result.
  - b. Add a public method named *subtract(int a, int b)* that returns an int result.
  - c. Add a public method named *multiply(int a, int b)* that returns an int result.
  - d. Add a public method named *divide(int a, int b)* that returns an int result.
  - e. Using *main()* run and test your class to ensure that the simply calculator operations work as expected.



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







    public int subtract(int a, int b)
    {
        return a - b;
    }

    public int multiply(int a, int b)
    {
        return a * b;
    }

    public int divide(int a, int b)
    {
        return a / b;
    }

    public static void main(String[] args)
    {
        Calculator calc = new Calculator();
        System.out.println("Adding 1 + 2 is " + calc.add(1, 2));
        System.out.println("Subtracting 2 from 1 is " + calc.subtract(2, 1));
        System.out.println("Multiplication of 10 and 2 is " + calc.multiply(10, 2));
        System.out.println("Division of 10 by 2 is " + calc.divide(10, 2));
    }
}
```

## Create JUnit Test Cases:

-  Create a new package named *test*. Right click on the *Calculator* class and select the New->JUnit Test Case menu options.
-  Select JUnit 4, place the new class in the *test* package, and leave the defaults and click the Next button.
  - i. NOTE: because of bugs in Eclipse JUnit 5 will not be used in this activity.
-  Select the *add()*, *subtract()*, *multiply()* and *divide()* methods from the Calculator class.
-  Click the Finish button (and if prompted add JUnit to your build path).
-  Inspect the generated *CalculatorTest* class.
-  Add one or more test conditions to fully validate the Calculator addition function:
  - i. `Calculator calc = new Calculator();`
  - ii. `Assert.assertEquals(3, calc.add(2, 1));`












```
public class CalculatorTest
{
    @Test
    public void testAdd()
    {
        Calculator calc = new Calculator();
        Assert.assertEquals(3, calc.add(2, 1));
        Assert.assertEquals(7, calc.add(5, 2));
    }

    @Test
    public void testSubtract()
    {
        Calculator calc = new Calculator();
        Assert.assertEquals(1, calc.subtract(2, 1));
        Assert.assertEquals(3, calc.subtract(5, 2));
    }

    @Test
    public void testMultiply()
    {
        Calculator calc = new Calculator();
        Assert.assertEquals(2, calc.multiply(2, 1));
        Assert.assertEquals(10, calc.multiply(5, 2));
    }

    @Test
    public void testDivide()
    {
        Calculator calc = new Calculator();
        Assert.assertEquals(2, calc.divide(2, 1));
        Assert.assertEquals(2, calc.divide(5, 2));
    }
}
```

-  From Eclipse, click the Run As menu and select the JUnit Test option. Validate that the add test case passes.
  -  Complete writing test conditions for the remaining Calculator functions.
  -  From Eclipse, click the Run menu and validate that all test cases pass.
  -  Emulate a bug in the Calculator by changing the add() method to add the 2 numbers and subtract 1 result in the correct value.
  -  From Eclipse, click the Run menu and validate that all test cases pass. Add test case that will cause a failure. Run the unit tests again. Double click on the JUnit error to view the test case that failed. Undo the code change from the previous step. Take a screenshot of the JUnit results screen.
-  4) Create a JUnit Test Suite:
-  a. Create a new project named *topic7-1b*. Copy all the code from *topic7-1a* to the new project. Run the test cases in the new *topic7-1b* project to ensure the new project is working properly.
  -  b. Right click on the *test* package and select the New->Other->Java->JUnit->JUnit Test Suite menu options. Click the Next button. Under the 'Test classes to include' select the *CalculatorTest* class. Name your Test Suite class *AllTests* and put in the *test* package.
  -  c. Click the Finish button.



```
@RunWith(Suite.class)
@SuiteClasses({ CalculatorTest.class })
public class AllTests
{
}
```

- 4. Run the Test Suite from Eclipse by clicking the Run As menu and select the JUnit Test option. Validate that all test cases still pass.
  - 5. Take a screenshot of the JUnit results screen.
- 5) Complete a writeup that answers the following questions for testing the calculator:
  - 2. How many test cases do you write? What error conditions would you need to test? Can you think of a very important test case to validate division?
  - 3. What is White and Black box testing?
  - 4. When would you use a Test Suite?

### Deliverables:

The following needs to be submitted as this part of the Activity:

- 2. Answers to testing questions.
- 3. All screenshots of the results of running the JUnit Tests.

## Part 2: Parameterized Unit Tests

### Overview

#### Goal and Directions:

In this activity you will develop dynamic data driven unit tests to further test the functionality of the calculator by using JUnit Parameterized Unit Tests. The following are the tasks you need to complete for this activity:

### Execution

- 1) Create a Parameterized JUnit Test Case:
  - 2. Create a new project named *topic7-2*. Copy all the code from *topic7-1b* to the new project. Run the test cases in the new *topic7-2* project to ensure the new project is working properly.
  - 3. Modify the *CalculatorTest* class to support test parameters:
    1. Add `@RunWith(Parameterized.class)` to the *CalculatorTest* class
    2. Add class member variables as test parameters:

```
enum Type {ADD, SUBTRACT, MULTYPLY, DIVIDE};
@Parameter(0)
public Type type;
@Parameter(1)
```



```
public int a1;  
@Parameter(2)  
public int a2;  
@Parameter(3)  
public int result;
```

iii. Add test data:

```
@Parameters  
public static Collection<Object[]> data()  
{  
    Object[][] data = new Object[][] {  
        {Type.ADD, 2, 1, 3}, {Type.ADD, 5, 2, 7},  
        {Type.SUBTRACT, 2, 1, 1}, {Type.SUBTRACT, 5, 2, 3},  
        {Type.MULTIPLY, 2, 1, 2}, {Type.MULTIPLY, 5, 2, 10},  
        {Type.DIVIDE, 2, 1, 2}, {Type.DIVIDE, 5, 2, 2}  
    };  
    return Arrays.asList(data);  
}
```

iv. Update the test cases:

```
Assume.assumeTrue(type == Type.ADD); // Change for each test case  
Calculator calc = new Calculator();  
Assert.assertEquals(result, calc.add(a1, a2)); // Change for each test case
```



```
@RunWith(Parameterized.class)
public class CalculatorTest
{
    enum Type {ADD, SUBTRACT, MULTIPLY, DIVIDE};

    @Parameter(0)
    public Type type;
    @Parameter(1)
    public int a1;
    @Parameter(2)
    public int a2;
    @Parameter(3)
    public int result;

    @Parameters
    public static Collection<Object[]> data()
    {
        Object[][] data = new Object[][] {
            {Type.ADD, 2, 1, 3}, {Type.ADD, 5, 2, 7},
            {Type.SUBTRACT, 2, 1, 1}, {Type.SUBTRACT, 5, 2, 3},
            {Type.MULTIPLY, 2, 1, 2}, {Type.MULTIPLY, 5, 2, 10},
            {Type.DIVIDE, 2, 1, 2}, {Type.DIVIDE, 5, 2, 2}
        };

        return Arrays.asList(data);
    }

    @Test
    public void testAdd()
    {
        Assume.assumeTrue(type == Type.ADD);
        Calculator calc = new Calculator();
        Assert.assertEquals(result, calc.add(a1, a2));
    }

    @Test
    public void testSubtract()
    {
        Assume.assumeTrue(type == Type.SUBTRACT);
        Calculator calc = new Calculator();
        Assert.assertEquals(result, calc.subtract(a1, a2));
    }

    @Test
    public void testMultiply()
    {
        Assume.assumeTrue(type == Type.MULTIPLY);
        Calculator calc = new Calculator();
        Assert.assertEquals(result, calc.multiply(a1, a2));
    }

    @Test
    public void testDivide()
    {
        Assume.assumeTrue(type == Type.DIVIDE);
        Calculator calc = new Calculator();
        Assert.assertEquals(result, calc.divide(a1, a2));
    }
}
```

- 1) Run the Test Suite from Eclipse by clicking the Run As menu and select the JUnit Test option. Validate that all test cases still pass.
- 2) Take a screenshot of the JUnit results screen.
- 3) Complete a writeup that answers the following questions for testing the calculator:
  - a. How did Parameterized Test improve your testing?

## Deliverables:

The following needs to be submitted as this part of the Activity:

- a. Answers to testing questions.



- b. All screenshots of the results of running the JUnit Tests.

## Part 3: Advanced Unit Tests

### Overview

#### Goal and Directions:

In this activity you will explore additional JUnit Assertions that can be used implement more complex tests. The following are the tasks you need to complete for this activity:

#### Execution

- 1) Create a new project named *topic7-3*. Create a new test case file in the *test* package that tests the following assertions. Add test runner class with a *main()* in the *test* package to run the tests from a Java class.
- 2) Create a test class and test runner class that will exercise and use the following assertions:
  - `void assertEquals(boolean expected, boolean actual)` to check that two primitives/objects are equal.
  - `void assertTrue(boolean condition)` to check that a condition is true.
  - `void assertFalse(boolean condition)` to check that a condition is false.
  - `void assertNotNull(Object object)` to check that an object isn't null.
  - `void assertNull(Object object)` to check that an object is null.
  - `void assertEquals(object1, object2)` to check if two object references point to the same object.
  - `void assertNotSame(object1, object2)` to check if two object references do not point to the same object.
  - `void assertEquals(expectedArray, resultArray)` check to whether two arrays are equal to each other.
- 3) An example is shown below:



```
public class TestAssertions
{
    @Test
    public void testAssertions()
    {
        // test data
        String str1 = new String("abc");
        String str2 = new String("abc");
        String str3 = null;
        String str4 = "abc";
        String str5 = "abc";

        int val1 = 5;
        int val2 = 6;

        String[] expectedArray = { "one", "two", "three" };
        String[] resultArray = { "one", "two", "three" };

        // Check that two objects are equal
        Assert.assertEquals(str1, str2);

        // Check that a condition is true
        Assert.assertTrue(val1 < val2);

        // Check that a condition is false
        Assert.assertFalse(val1 > val2);

        // Check that an object isn't null
        Assert.assertNotNull(str1);

        // Check that an object is null
        Assert.assertNull(str3);

        // Check if two object references point to the same object
        Assert.assertSame(str4, str5);

        // Check if two object references not point to the same object
        Assert.assertNotSame(str1, str3);

        // Check whether two arrays are equal to each other.
        Assert.assertArrayEquals(expectedArray, resultArray);
    }
}

import org.junit.runner.JUnit4;

public class TestRunner {
    public static void main(String[] args)
    {
        Result result = JUnit4.runClasses(TestAssertions.class);

        for (Failure failure : result.getFailures())
        {
            System.out.println("A JUnit test failed: " + failure.toString());
        }

        System.out.println("The JUnit Tests " + (result.wasSuccessful() ? "Passed" : "Failed"));
    }
}
```

- 4) Run the tests from Eclipse by running the test runner class as a Java class.
- 5) Take a screenshot of the console output.
- 6) Complete a writeup that answers the following questions for testing code:
  - a. How can you test that an exception is thrown in your code?
  - b. What challenges can you think of that will make testing for **all** error conditions and exceptions in your code even possible?

### Deliverables:

The following needs to be submitted as this part of the Activity:

- a. Answers to testing questions.
- b. All screenshots of the results of running the JUnit Tests.





### **Research Questions**

1. Research Questions: For traditional ground students in a Word document answer the following questions:
  - a. Compare and contrast the difference between JUnit and TestNG unit testing frameworks. Provide a table that outlines 10 similarities and 10 differences.
  - b. Research the concept of Code Coverage. What is code coverage and how is this metric used in software development. Summarize your answer in 300 words.

### **Final Activity Submission**

1. In a Microsoft Word document complete the following for the Activity Report:
  - a. Cover Sheet with the name of this assignment, date, and your name.
  - b. Section with a title that contains all the diagrams, screenshots, and theory of operation writeups.
  - c. Zip file with all code and generated JavaDoc documentation files.
  - d. Section with a title that contains the answers to the Research Questions (traditional ground students only).
2. Submit the Activity Report and zip file of the Code and Documentation to the Learning Management System (LMS).