Lab 06 – Using Assertions and Exceptions

Objectives:

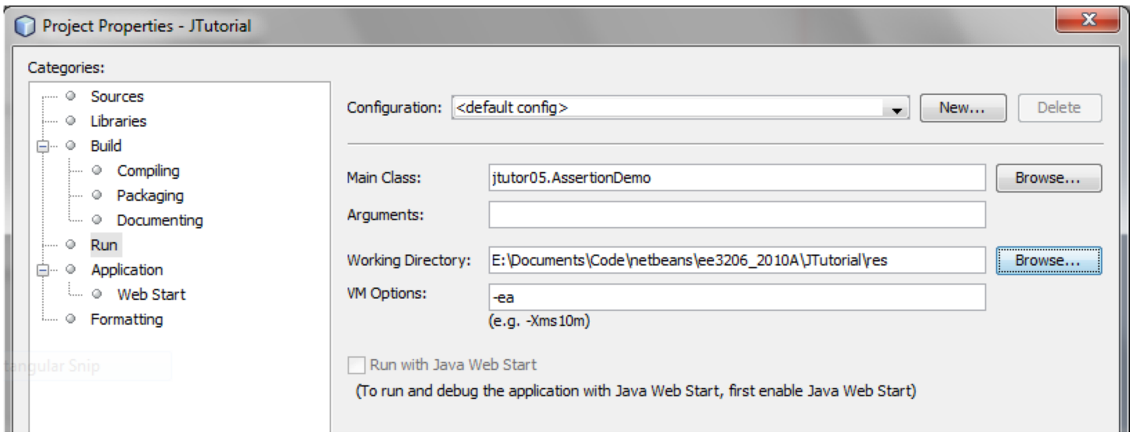
- Understand the uses of assertions and exceptions
- Learn to use assertions of different forms and enable assertion at runtime
- Learn to report exceptions and recover from exceptions

Finding bugs in programs is not an easy task, especially when they are subtle bugs. The process of finding bugs is not exciting, and you may have gone through your program wasting your time trying to find bugs that should not have existed in the first place. Such bugs may exist because you did not understand the specifications correctly. And as Barry Boehm, the father of software economics put it: If it costs $1 to find and fix a requirement-based problem during the requirements definition process, it can cost $5 to repair it during design, $10 during coding, $20 during unit testing, and as much as $200 after delivery of the system. Thus, finding bugs early in the software life cycle pays off.

1. An assertion is a statement in the Java programming language that enables you to test your assumptions about your program. The following program uses it to assure the precondition. Try to execute this program, and you should see it outputting **400** and **-100**.

```
class AssertionDemo {

    private static int balance = 900;

    public static void main(String args[]) {
        System.out.println("Acc balance: " + withdrawMoney(500));
        System.out.println("Acc balance: " + withdrawMoney(500));
    }

    public static double withdrawMoney(double amount) {
        // precondition: the balance must be enough for the withdraw amount
        assert balance >= amount;    // when balance is enough
        balance -= amount;
        return balance;
    }
}
```

By default assertions are disabled at runtime, you can enable it by adding the **–ea** option. In NetBeans, you need to put the option in [Project Properties > Run > VM Options]. Now run the program again. You should see the ***AssertionError*** and its ***stack trace***. This runtime option is useful as one can simply turn off the assertion and leave the testing code in program without runtime penalty.


Project Properties - JTutorial

Configuration: <default config>

Main Class: jtutor05.AssertionDemo

Arguments:

Working Directory: E:\Documents\Code\netbeans\ee3206_2010A\JTutorial\res

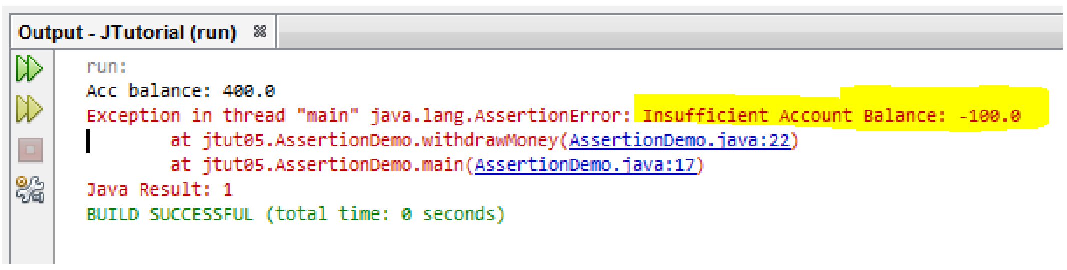
VM Options: -ea

(e.g. -Xms10m)

Run with Java Web Start

(To run and debug the application with Java Web Start, first enable Java Web Start)

2. In many cases, you may want to know more details about the AssertionError being thrown. You can therefore add a message to the assertion statement. Modify the assertion statement above such that it reports the net balance (balance - amount) when it is thrown.



```
run:
Acc balance: 400.0
Exception in thread "main" java.lang.AssertionError: Insufficient Account Balance: -100.0
        at jtut05.AssertionDemo.withdrawMoney(AssertionDemo.java:22)
        at jtut05.AssertionDemo.main(AssertionDemo.java:17)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

3. The four exceptions below are some common types of exception. Find their causes from the Java API and determine if these are checked or unchecked exception.
   - ArithmeticException
   - ArrayIndexOutOfBoundsException
   - ClassCastException
   - NullPointerException

4. Regarding the following try-catch pattern, point out the potential problem if any.

```
try {
    // do something
} finally {
    // do something
}
```

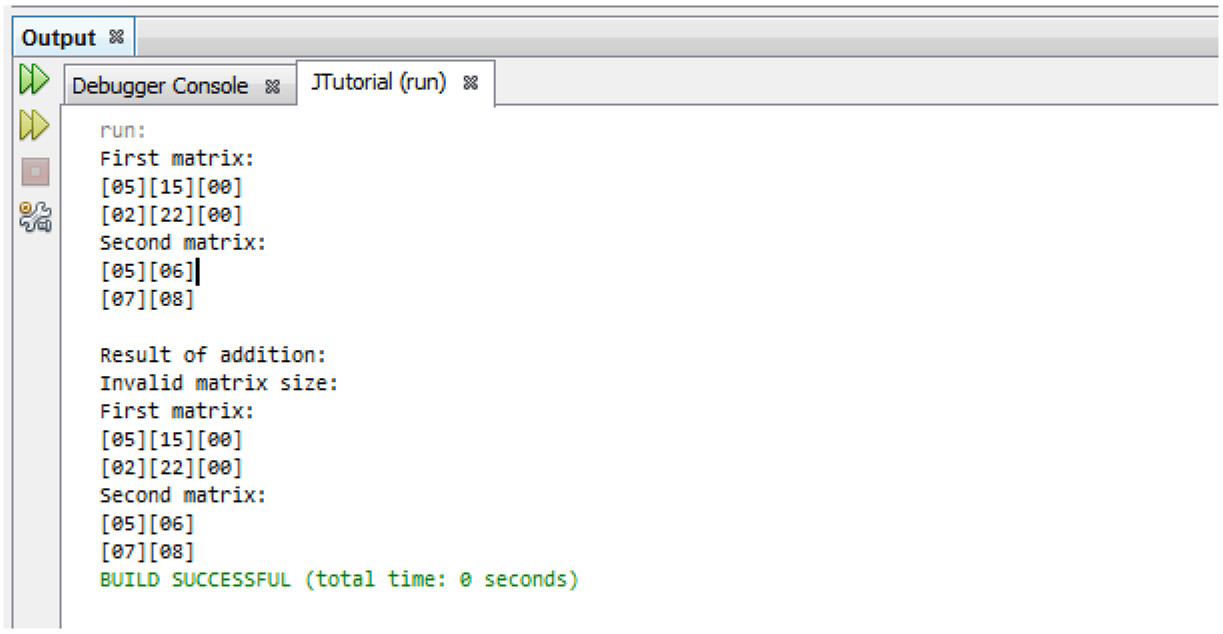
```
try {
     // do something
} catch (Exception e) {
     // do something
} catch (ArithmeticException a) {
     // do something
}
```

5. Do you know you can define your custom exception class to cater for your application need?

In lab 3, we have created a Matrix class with print, add and multiply methods. The original multiply method returns a null value when the two matrices' dimensions are incorrect. A better alternative is to throw an exception to report this error, and we can define our own exception class to store the error information (i.e. the matrices).

Revise the multiply method such that it throws a custom exception ***DimensionMismatchException*** on fail. You need to design this custom exception class and use it to store the two mismatched matrices for reference. The test program and the corresponding output are given below.

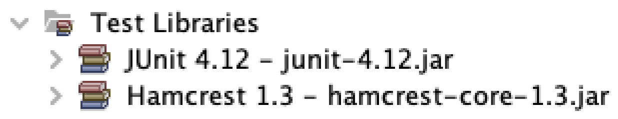
```
public class TestMatrix {

    public static void main(String[] args) {
        // initialize both matrices
        Matrix m1 = new Matrix(new int[][]{{5,15,0},{2,22,0}});
        Matrix m2 = new Matrix(new int[][]{{5,6},{7,8}});

        System.out.println("First matrix:");
        m1.print();

        System.out.println("Second matrix:");
        m2.print();

        System.out.println("");
        System.out.println("Result of addition:");

        try {
            m1.multiply(m2).print();                         // exception may occur
        } catch(DimensionMismatchException ex) {
            System.out.println("Invalid matrix size: ");
            System.out.println("First matrix:");
            ex.getFirstMatrix().print();                    // get and print the first matrix
            System.out.println("Second matrix:");
            ex.getSecondMatrix().print();                   // get and print the second matrix
        }
    }
}
```



```
run:
First matrix:
[05][15][00]
[02][22][00]
Second matrix:
[05][06]
[07][08]

Result of addition:
Invalid matrix size:
First matrix:
[05][15][00]
[02][22][00]
Second matrix:
[05][06]
[07][08]
BUILD SUCCESSFUL (total time: 0 seconds)
```

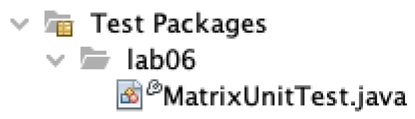
6. Software unit tests help the developer to verify that the logic of a piece of the program is correct. Running tests automatically helps to identify software regressions introduced by changes in the source code. Having a high test coverage of your code allows you to continue developing features without having to perform lots of manual tests.

   JUnit is a popular test framework which uses **annotations** to identify methods that specify a test. To use it in our project, we need to add the following JARs to our **Test Libraries (right-click and choose Add Library)**.



Test Libraries
JUnit 4.12 – junit-4.12.jar
Hamcrest 1.3 – hamcrest-core-1.3.jar

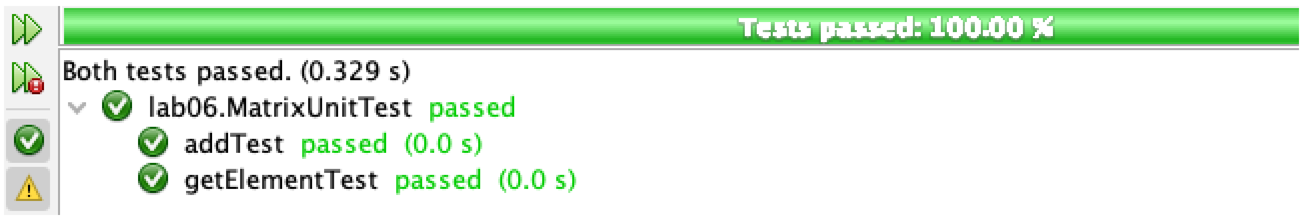
   Instead of putting all the test cases in the main method, we can set up a test class (***MatrixUnitTest.java***) under the **Test Packages** for the Matrix as follows:



Test Packages
lab06
MatrixUnitTest.java

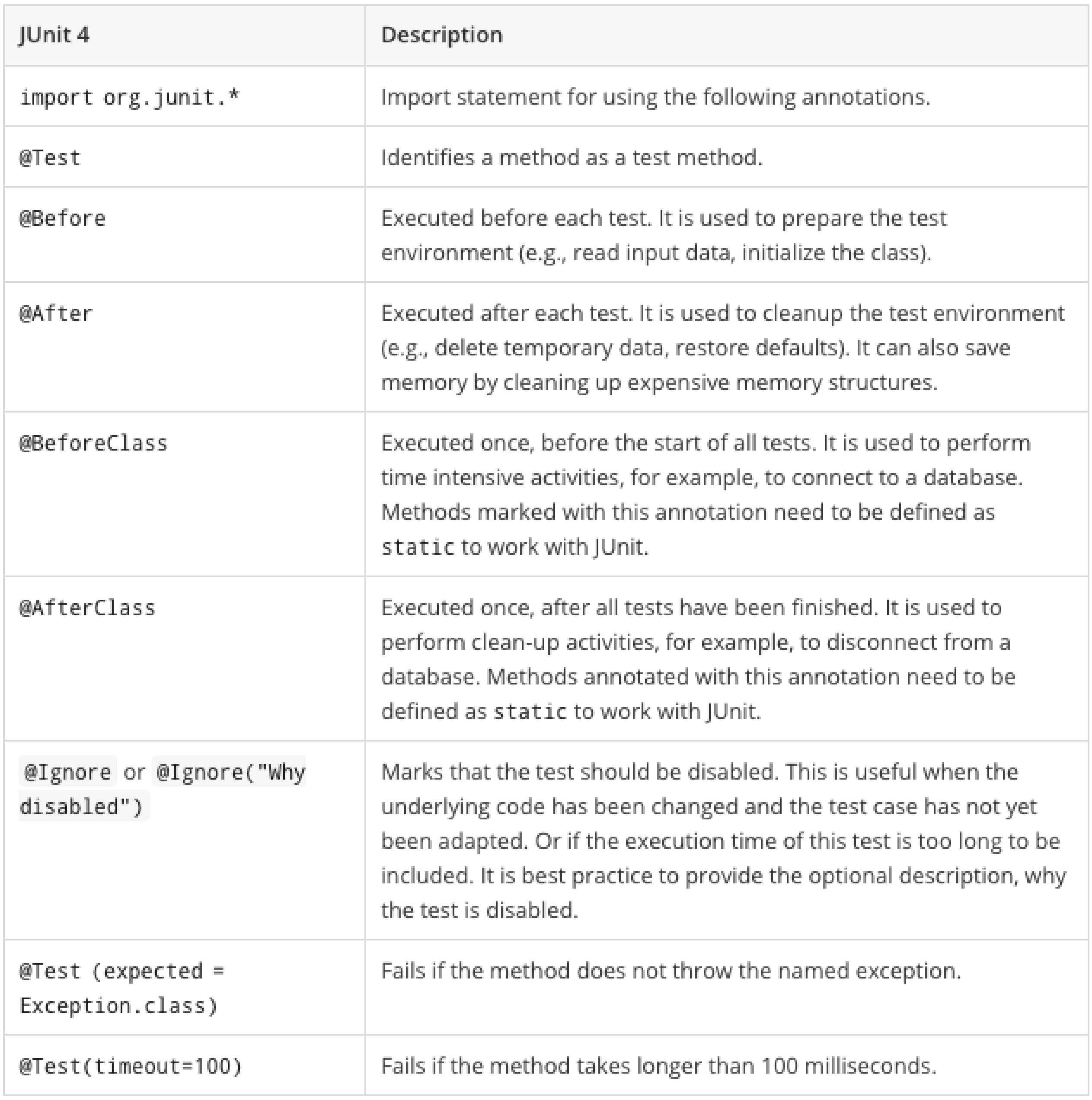
```
import static junit.framework.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class MatrixUnitTest {

    Matrix matA;
    Matrix matB;

    @Before
    public void setUp() {
        matA = new Matrix(new int[][]{{1,2,3},{4,5,6}});
        matB = new Matrix(new int[][]{{1,1,1},{2,2,2}});
    }

    @Test
    public void getElementTest() {
        int[][] expected = new int[][]{{1,2,3},{4,5,6}};
        for(int i=0; i<expected.length; i++)
            for(int j=0; j< expected[0].length; j++)
                assertEquals(expected[i][j], matA.getElement(i, j));
    }

    @Test
    public void addTest() {
        int[][] expected = new int[][]{{2,3,4},{6,7,8}};
        matA.add(matB);
        for(int i=0; i<expected.length; i++)
            for(int j=0; j< expected[0].length; j++)
                assertEquals(expected[i][j], matA.getElement(i, j));
    }

}
```

   Run the test class and you should see a test report as follows.



Tests passed: 100.00 %
Both tests passed. (0.329 s)
lab06.MatrixUnitTest passed
addTest passed (0.0 s)
getElementTest passed (0.0 s)

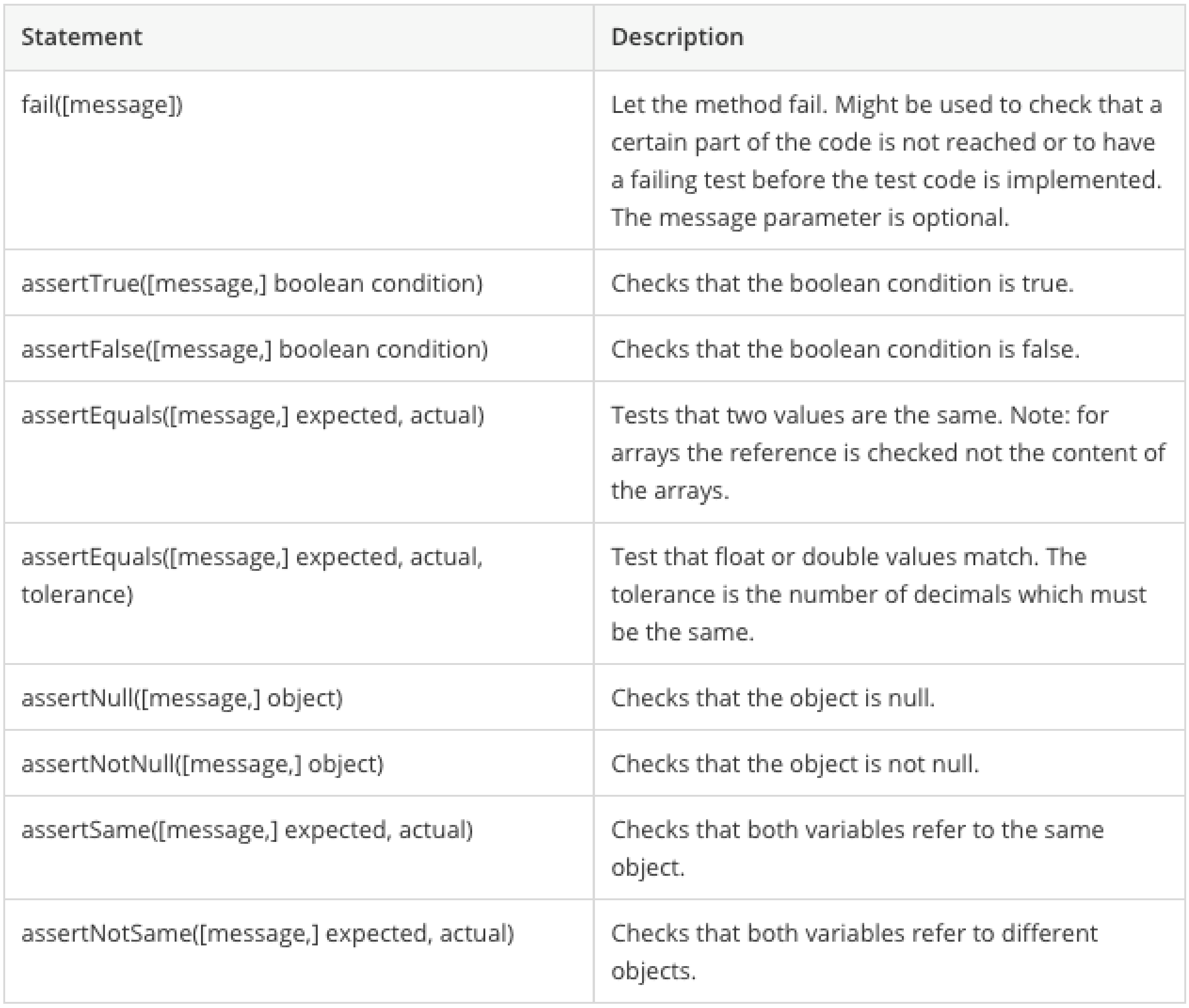
JUnit uses annotations to mark methods as test methods and to configure them. The following table gives an overview of the most important annotations in JUnit for the 4.x and 5.x versions. All these annotations can be used on methods.



JUnit 4	Description
`import org.junit.*`	Import statement for using the following annotations.
`@Test`	Identifies a method as a test method.
`@Before`	Executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
`@After`	Executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
`@BeforeClass`	Executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as `static` to work with JUnit.
`@AfterClass`	Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.
`@Ignore` or `@Ignore("Why disabled")`	Marks that the test should be disabled. 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. It is best practice to provide the optional description, why the test is disabled.
`@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.

JUnit provides static methods to test for certain conditions via the ***Assert*** class. These assert statements typically start with “assert...”. They allow you to specify the error message, the expected and the actual result. An assertion method compares the actual value returned by a test to the expected value. It throws an ***AssertionException*** if the comparison fails.

The following table gives an overview of these methods. Parameters in [ ] brackets are optional and of type String.



Statement	Description
fail([message])	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional.
assertTrue([message,] boolean condition)	Checks that the boolean condition is true.
assertFalse([message,] boolean condition)	Checks that the boolean condition is false.
assertEquals([message,] expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([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([message,] expected, actual)	Checks that both variables refer to the same object.
assertNotSame([message,] expected, actual)	Checks that both variables refer to different objects.

Now, let’s complete the unit test by adding two more test cases for the multiply() method <u>on success</u> and <u>on fail</u> respectively.

- END -