VISVESVARAYA TECHNOLOGICAL UNIVERSITY

JNANA SANGAMA, BELAGAVI-590018, KARNATAKA

ASSIGNMENT ON SIMPLE CALCULATOR PROGRAM USING JUNIT TESTING TOOL.

Information Science and Engineering

Submitted by

SRINIVASA V	**- 1BI17IS048**
SYED AFFANULLA	**- 1BI17IS051**
DADAKHALANDAR B	**- 1BI18IS403**

Under the guidance of

Mrs. VEDASREE T K

Assistant Professor

Dept. of ISE, BIT

BANGALORE INSTITUTE OF TECHNOLOGY

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

K.R. Road, V.V. Puram, Bengaluru – 560004

2019-2020

1. INTRODUCTION

Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects), and verifying that the software product is fit for use.

It is also defined as an activity to check whether the actual results match the expected results and to ensure that the software system is Defect free. It involves execution of a software component or system component to evaluate one or more properties of interest. Software testing also helps to identify errors, gaps or missing requirements in contrary to the actual requirements. It can be either done manually or using automated tools.

Software testing involves the execution of a software component or system component to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test:

- meets the requirements that guided its design and development,
- responds correctly to all kinds of inputs,
- performs its functions within an acceptable time,
- it is sufficientlyusable,
- can be installed and run in its intended environments, and
- achieves the general result its stakeholders desire.

Software testing can be divided into two steps**:**

1. Verification: it refers to the set of tasks that ensure that software correctly implements a specific function.

2. Validation: it refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

2. TYPES OF SOFTWARE TESTING

Typically Testing is classified into three categories.

- Functional Testing
- Non-Functional Testing or Performance Testing
- Maintenance (Regression and Maintenance)

It is referred to in the table 2.1 given below:

Testing Category	**Types of Testing**
Functional Testing	Unit Testing
	Integration Testing
	Smoke
	UAT(User Acceptance Testing)
	Globalization
	Localization
	Interoperability
	So on
Non-Functional Testing	Performance
	Endurance
	Load
	Volume
	Usability
	Scalability
	So on
Maintenance	Regression
	Maintenance

Table 2.1: Types of Software Testing

3. LEVELS OF SOFTWARE TESTING

Broadly speaking, there are at least three levels of testing: unit testing, integration testing, and system testing. This may be in the form of operational acceptance testing or be simple end-user testing, testing to ensure the software meets functional expectations. Tests are frequently grouped into one of these levels by where they are added in the software development process.

Unit testing

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors. Unit testing is a software development process that involves a synchronized application of a broad spectrum of defect prevention and detection strategies in order to reduce software development risks, time, and costs. It is performed by the software developer or engineer during the construction phase of the software development life cycle. Unit testing aims to eliminate construction errors before code is promoted to additional testing; this strategy is intended to increase the quality of the resulting software as well as the efficiency of the overall development process.

Integration testing

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Software components may be integrated in an iterative way or all together ("big bang"). Normally the former is considered a better practice since it allows interface issues to be located more quickly and fixed. Integration tests usually involves a lot of code, and produce traces that are larger than those produced by unit tests. This has an impact on the ease of localizing the fault when an integration test fails. To overcome this issue, it has been proposed to automatically cut the large tests in smaller pieces to improve fault localization.

System testing

System testing tests a completely integrated system to verify that the system meets its requirements. For example, a system test might involve testing a login interface, then creating and editing an entry, plus sending or printing results, followed by summary processing or deletion (or archiving) of entries, then logoff.

4. SOFTWARE TESTING TOOL – ECLIPSE JUnit

JUnit

Figure 4.1: ECLIPSE JUnit Logo

JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks which is collectively known as xUnit that originated with SUnit.

JUnit is linked as a JAR at compile-time; the framework resides under package `junit.framework` for JUnit 3.8 and earlier, and under package `org.junit` for JUnit 4 and later.

JUnit is also a Regression Testing Framework used by developers to implement unit testing in Java, and accelerate programming speed and increase the quality of code. JUnit Framework can be easily integrated with either of the following −

- Eclipse
- Ant
- Maven
- Intellij

5. Features of JUnit Test Framework

JUnit test framework provides the following important features −

- Fixtures
- Test suites
- Test runners
- JUnit classes

Fixtures is a fixed state of a set of objects used as a baseline for running tests. The purpose of a test fixture is to ensure that there is a well-known and fixed environment in which tests are run so that results are repeatable. It includes −

- setUp() method, which runs before every test invocation.
- tearDown() method, which runs after every test method.

A test suite bundles a few unit test cases and runs them together. In JUnit, both @RunWith and @Suite annotation are used to run the suite test. Given below is an example that uses TestJunit1 & TestJunit2 test classes.

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

        //JUnit Suite Test
        @RunWith(Suite.class)

@Suite.SuiteClasses({
          TestJunit1.class ,TestJunit2.class
})

public class JunitTestSuite {
}
```

Test runner is used for executing the test cases. Here is an example that assumes the test class TestJunit already exists.

```
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());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

JUnit classes are important classes, used in writing and testing JUnits. Some of the important classes are −

- Assert − Contains a set of assert methods.
- TestCase − Contains a test case that defines the fixture to run multiple tests.
- TestResult − Contains methods to collect the results of executing a test case.

6. ANNOTATIONS OF JUNIT TESTING

The Junit 4.x framework is annotation based, so let's see the annotations that can be used while writing the test cases.

@Test annotation specifies that method is the test method.

@Test(timeout=1000) annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

@BeforeClass annotation specifies that method will be invoked only once, before starting all the tests.

@Before annotation specifies that method will be invoked before each test.

@After annotation specifies that method will be invoked after each test.

@AfterClass annotation specifies that method will be invoked only once, after finishing all the tests.

7. METHODS OF ASSERT CLASS

The common methods of Assert class are as follows:

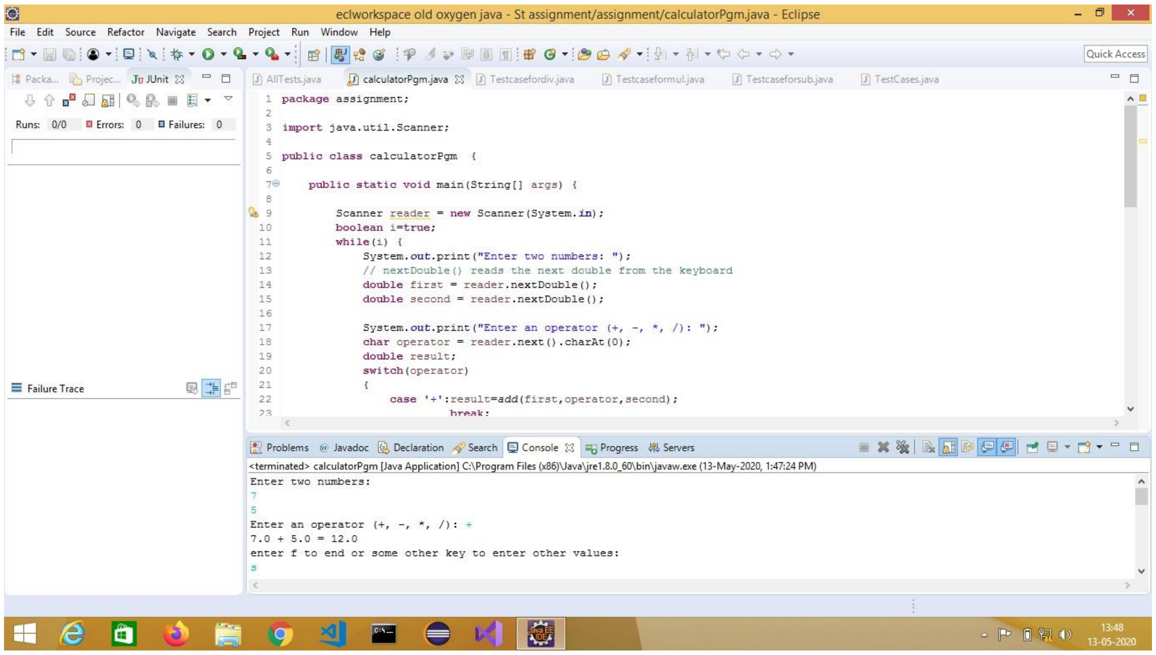
1. **void assertEquals(boolean expected,boolean actual)**: checks that two primitives/objects are equal. It is overloaded.

2. **void assertTrue(boolean condition)**: checks that a condition is true.
3. **void assertFalse(boolean condition)**: checks that a condition is false.
4. **void assertNull(Object obj)**: checks that object is null.
5. **void assertNotNull(Object obj)**: checks that object is not null.

8. TESTING THE CALCULATOR PROGRAM USING JUNIT

There are various steps involved in testing a program. It must be noted that junit is in-built in the eclipse platform so, care must be taken to have a code which runs on JVM. Here, a simple calculator program (Java code) is first executed. After its compiled and run, the next step is to test the code using junit testing tool. It must be noted that junit is a manual testing tool, so a code needs to be written to test the program. So, below are some self-explanatory snapshots of how actually the program is tested.

So here the calculator java code is being executed with a public class calculator Program. Figure 8.1 given below shows the sample program code for the simple calculator.

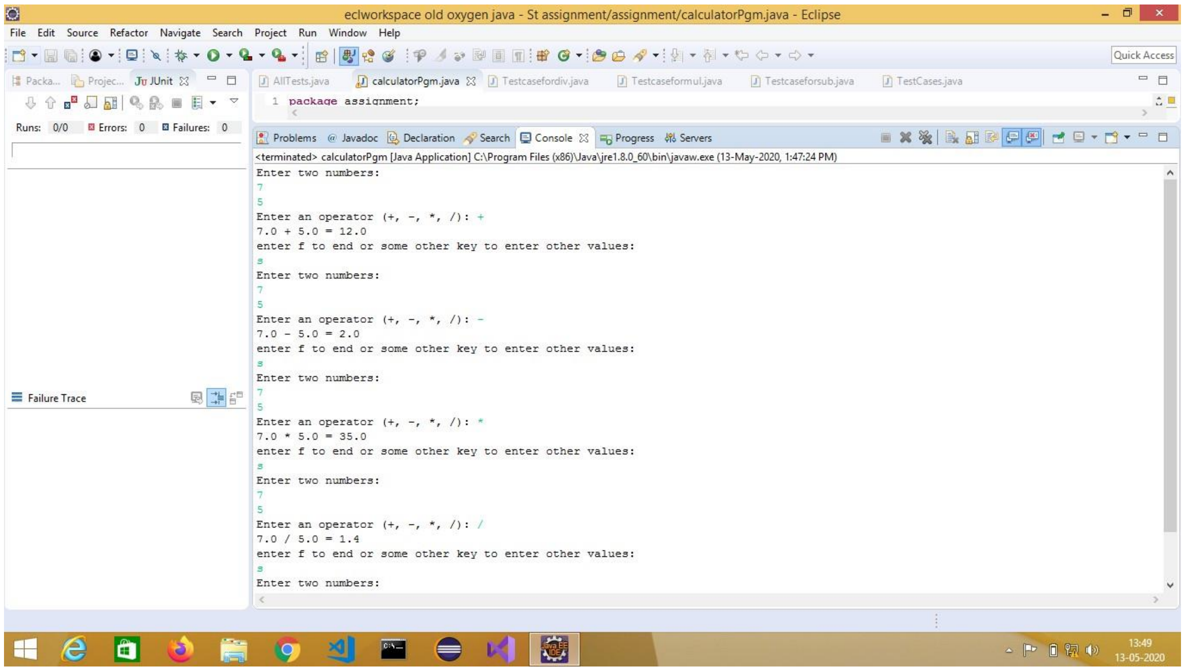

eclworkspace old oxygen java - St assignment/assignment/calculatorPgm.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Packa... Projec... JUnit
Runs: 0/0 Errors: 0 Failures: 0
Failure Trace
AllTests.java calculatorPgm.java Testcasefordiv.java Testcaseformul.java Testcaseforsub.java TestCases.java
Problems Javadoc Declaration Search Console Progress Servers
13:48
13-05-2020

```java
package assignment;

import java.util.Scanner;

public class calculatorPgm  {

    public static void main(String[] args) {

        Scanner reader = new Scanner(System.in);
        boolean i=true;
        while(i) {
            System.out.print("Enter two numbers: ");
            // nextDouble() reads the next double from the keyboard
            double first = reader.nextDouble();
            double second = reader.nextDouble();

            System.out.print("Enter an operator (+, -, *, /): ");
            char operator = reader.next().charAt(0);
            double result;
            switch(operator)
            {
                case '+':result=add(first,operator,second);
                    break;
```

```
<terminated> calculatorPgm [Java Application] C:\Program Files (x86)\Java\jre1.8.0_60\bin\javaw.exe (13-May-2020, 1:47:24 PM)
Enter two numbers:
7
5
Enter an operator (+, -, *, /): +
7.0 + 5.0 = 12.0
enter f to end or some other key to enter other values:
s
```

Figure 8.1: Simple Calculator Program

Sample output of the code for various operations like addition, subtraction, multiplication and division, is being displayed in the console output. It is been shown in 2 figures that is Figure 8.2 and Figure 8.3 which displays the output of the program in the console view shown below respectively.

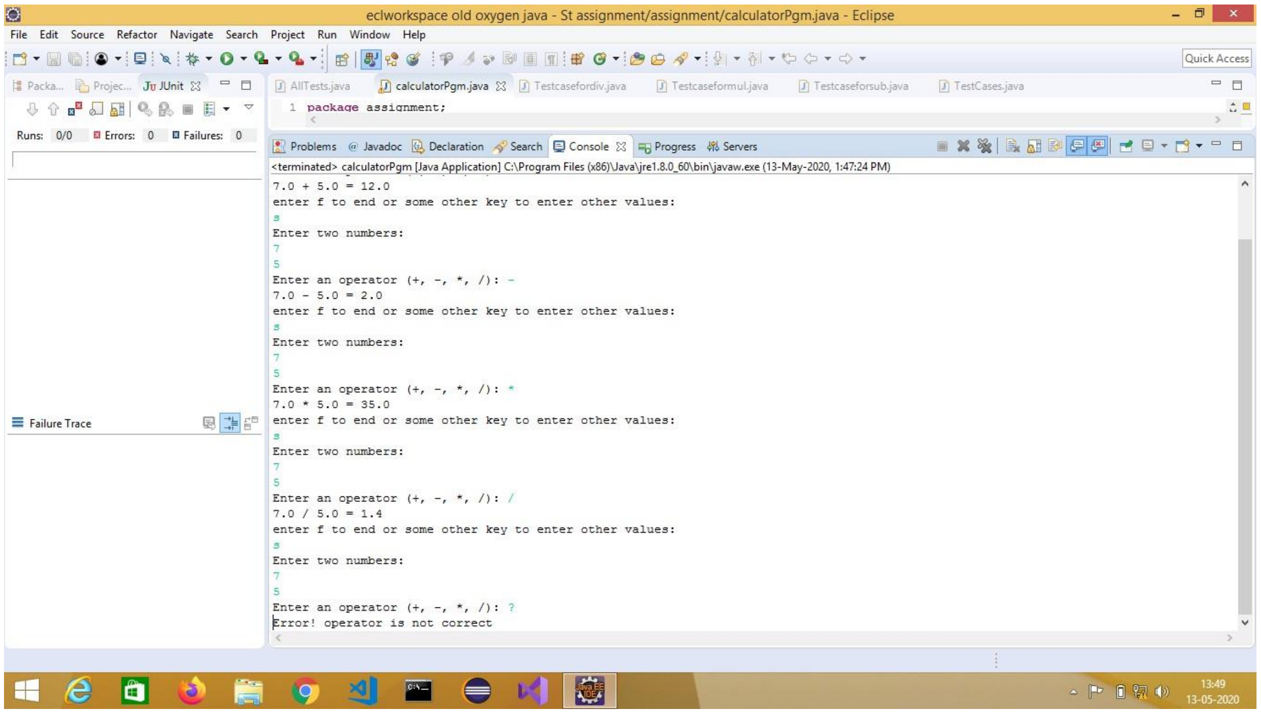


eclworkspace old oxygen java - St assignment/assignment/calculatorPgm.java - Eclipse

```
package assignment;
```

```
<terminated> calculatorPgm [Java Application] C:\Program Files (x86)\Java\jre1.8.0_60\bin\javaw.exe (13-May-2020, 1:47:24 PM)
Enter two numbers:
7
5
Enter an operator (+, -, *, /): +
7.0 + 5.0 = 12.0
enter f to end or some other key to enter other values:
s
Enter two numbers:
7
5
Enter an operator (+, -, *, /): -
7.0 - 5.0 = 2.0
enter f to end or some other key to enter other values:
s
Enter two numbers:
7
5
Enter an operator (+, -, *, /): *
7.0 * 5.0 = 35.0
enter f to end or some other key to enter other values:
s
Enter two numbers:
7
5
Enter an operator (+, -, *, /): /
7.0 / 5.0 = 1.4
enter f to end or some other key to enter other values:
s
Enter two numbers:
```

Figure 8.2: Simple Calculator Program Sample Output



eclworkspace old oxygen java - St assignment/assignment/calculatorPgm.java - Eclipse

```
package assignment;
```

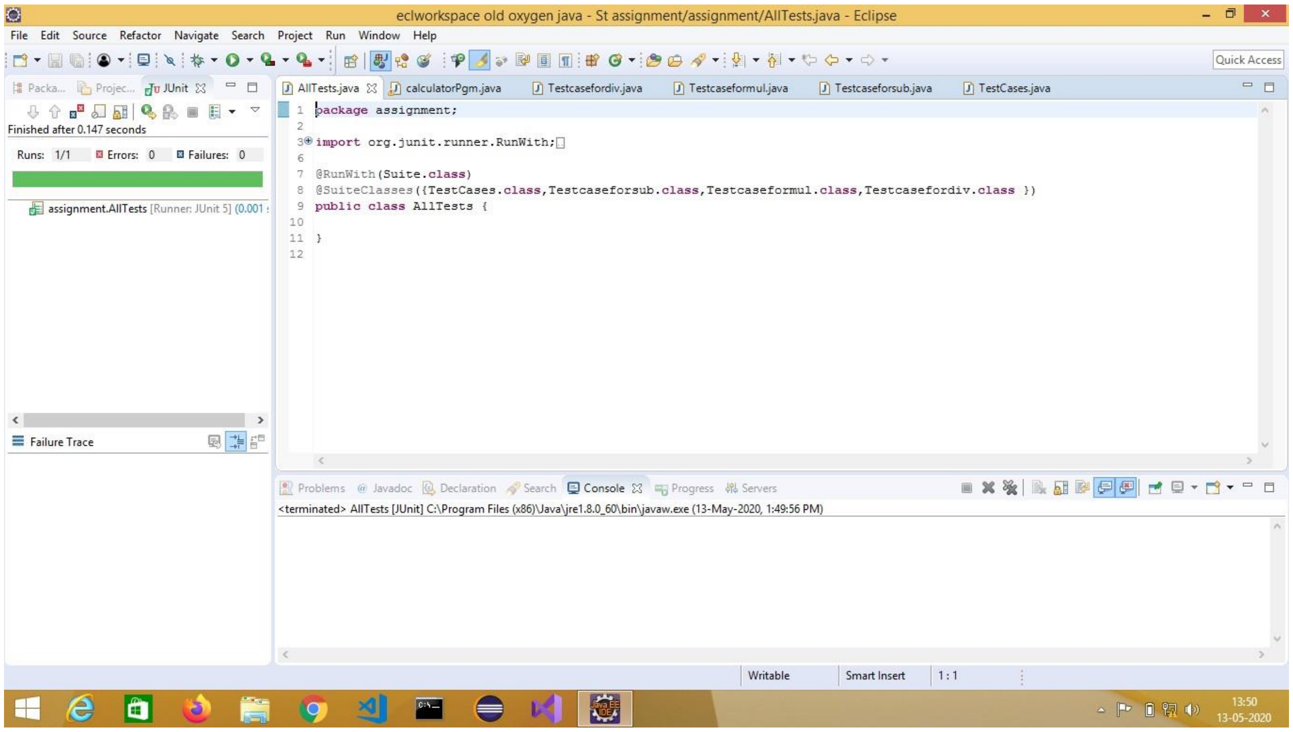
```
<terminated> calculatorPgm [Java Application] C:\Program Files (x86)\Java\jre1.8.0_60\bin\javaw.exe (13-May-2020, 1:47:24 PM)
7.0 + 5.0 = 12.0
enter f to end or some other key to enter other values:
s
Enter two numbers:
7
5
Enter an operator (+, -, *, /): -
7.0 - 5.0 = 2.0
enter f to end or some other key to enter other values:
s
Enter two numbers:
7
5
Enter an operator (+, -, *, /): *
7.0 * 5.0 = 35.0
enter f to end or some other key to enter other values:
s
Enter two numbers:
7
5
Enter an operator (+, -, *, /): /
7.0 / 5.0 = 1.4
enter f to end or some other key to enter other values:
s
Enter two numbers:
7
5
Enter an operator (+, -, *, /): ?
Error! operator is not correct
```

Figure 8.3: Simple Calculator Program Sample Output

The calculator java code is a user-friendly code wherein the is first asked to enter the two numbers upon which the mathematical operation has to be performed. Then the user is asked to enter symbol of operations i.e. + or – or * or /. Based on the operation and the numbers provided by the user the respective results are printed in the console output.

Based on the above sample inputs, test cases are created with the help of the junit testing tool. So, these further steps and processes of testing are described in the snapshots below.

An all test cases class is created along with importing of the in-built junit class.


eclworkspace old oxygen java - St assignment/assignment/AllTests.java - Eclipse

Finished after 0.147 seconds

Runs: 1/1 Errors: 0 Failures: 0

assignment.AllTests [Runner: JUnit 5] (0.001 s)

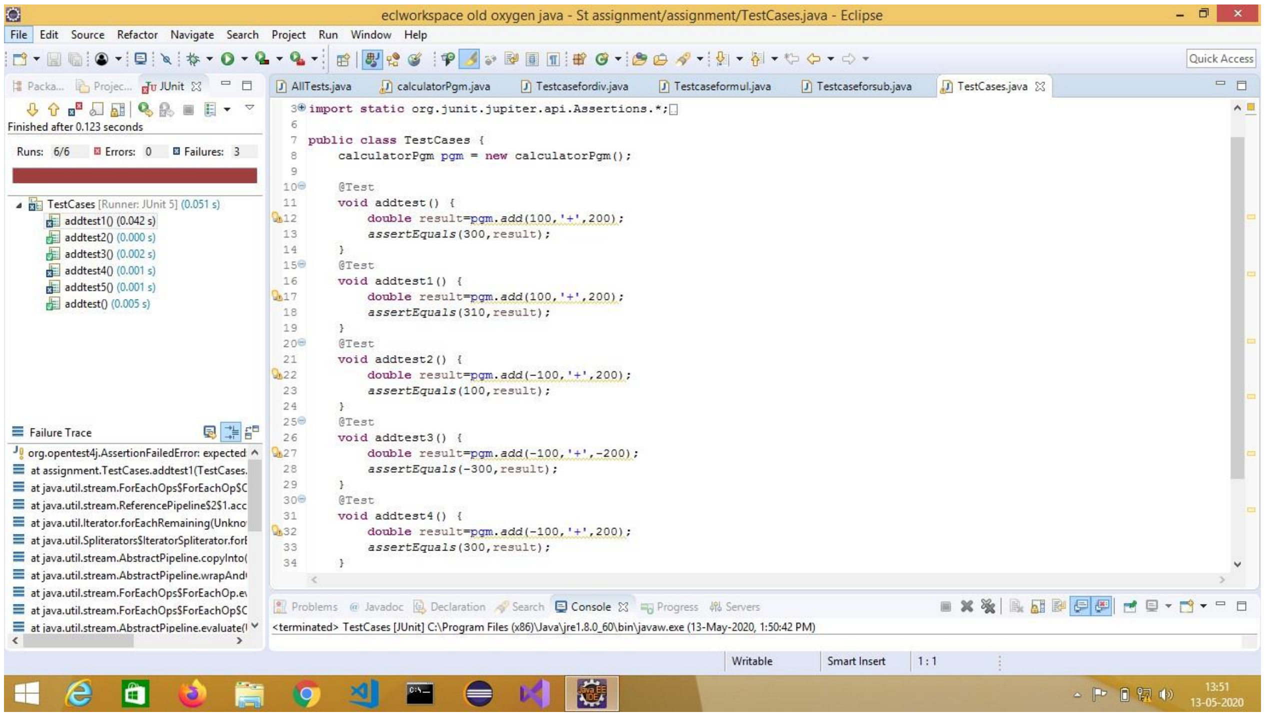
```java
package assignment;

import org.junit.runner.RunWith;

@RunWith(Suite.class)
@SuiteClasses({TestCases.class,Testcaseforsub.class,Testcaseformul.class,Testcasefordiv.class })
public class AllTests {

}
```

```
<terminated> AllTests [JUnit] C:\Program Files (x86)\Java\jre1.8.0_60\bin\javaw.exe (13-May-2020, 1:49:56 PM)
```

Figure 8.4: All Test Cases Class

Through this class AllTests individual test cases of each operation i.e. addition, subtraction, multiplication and division are accessed and executed. This is been shown in the Figure 8.4: All Test Cases Class shown above.

The test cases of addition operation are run, with manual sample inputs testing the operation. The Figure 8.5: Test Cases of Addition Operation in the figure below represents the same.


eclworkspace old oxygen java - St assignment/assignment/TestCases.java - Eclipse

```
Finished after 0.123 seconds
Runs: 6/6    Errors: 0    Failures: 3

TestCases [Runner: JUnit 5] (0.051 s)
    addtest1() (0.042 s)
    addtest2() (0.000 s)
    addtest3() (0.002 s)
    addtest4() (0.001 s)
    addtest5() (0.001 s)
    addtest() (0.005 s)

Failure Trace
org.opentest4j.AssertionFailedError: expected
at assignment.TestCases.addtest1(TestCases.
at java.util.stream.ForEachOps$ForEachOp$C
at java.util.stream.ReferencePipeline$2$1.acc
at java.util.Iterator.forEachRemaining(Unkno
at java.util.Spliterators$IteratorSpliterator.for
at java.util.stream.AbstractPipeline.copyInto(
at java.util.stream.AbstractPipeline.wrapAnd
at java.util.stream.ForEachOps$ForEachOp.ev
at java.util.stream.ForEachOps$ForEachOp$C
at java.util.stream.AbstractPipeline.evaluate(
```

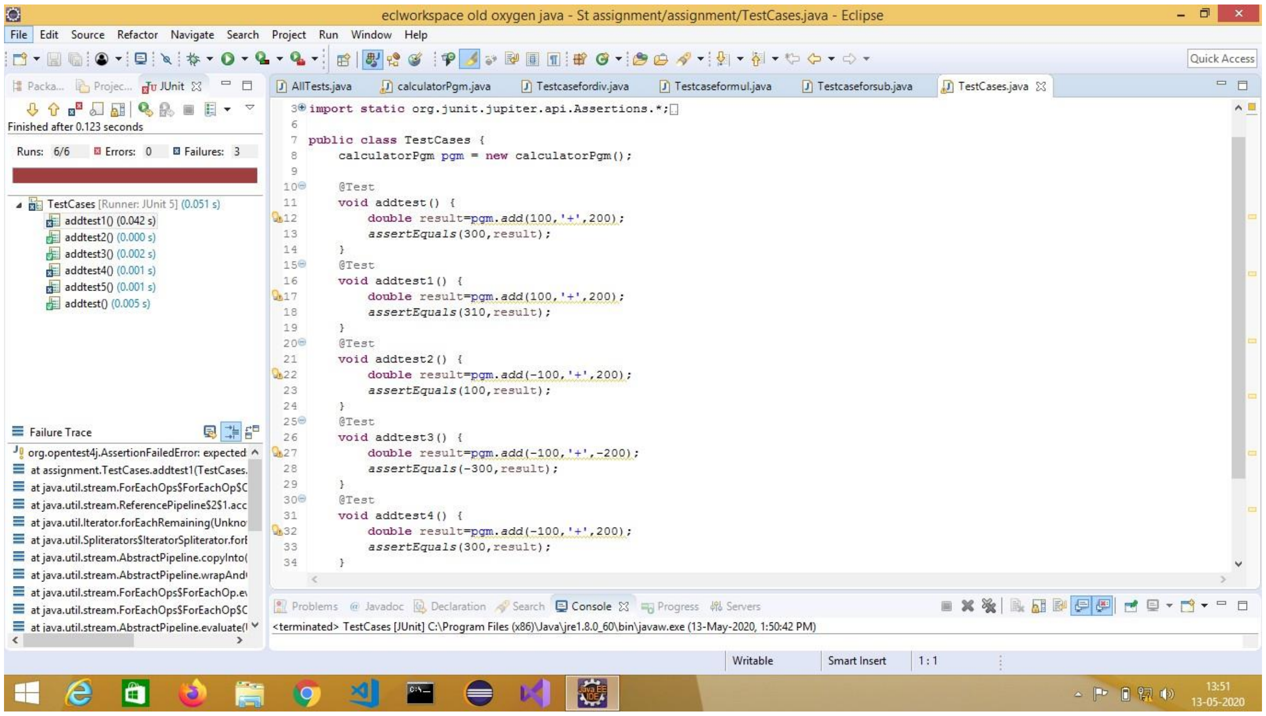
```java
import static org.junit.jupiter.api.Assertions.*;

public class TestCases {
    calculatorPgm pgm = new calculatorPgm();

    @Test
    void addtest() {
        double result=pgm.add(100,'+',200);
        assertEquals(300,result);
    }
    @Test
    void addtest1() {
        double result=pgm.add(100,'+',200);
        assertEquals(310,result);
    }
    @Test
    void addtest2() {
        double result=pgm.add(-100,'+',200);
        assertEquals(100,result);
    }
    @Test
    void addtest3() {
        double result=pgm.add(-100,'+',-200);
        assertEquals(-300,result);
    }
    @Test
    void addtest4() {
        double result=pgm.add(-100,'+',200);
        assertEquals(300,result);
    }
```

```
<terminated> TestCases [JUnit] C:\Program Files (x86)\Java\jre1.8.0_60\bin\javaw.exe (13-May-2020, 1:50:42 PM)
```

Figure 8.5: Test Cases of Addition Operation

The results of the individual tests are displayed on the left-hand side, column section, where In a tick mark denotes success of a test and an x denotes the failure of the test. The test cases of subtraction operation are run with manual sample inputs testing the operation.

The Figure 8.6: Test Cases of Subtraction Operation in the figure below represents the same.



eclworkspace old oxygen java - St assignment/assignment/TestCases.java - Eclipse

Finished after 0.123 seconds

Runs: 6/6 Errors: 0 Failures: 3

```
TestCases [Runner: JUnit 5] (0.051 s)
    addtest1() (0.042 s)
    addtest2() (0.000 s)
    addtest3() (0.002 s)
    addtest4() (0.001 s)
    addtest5() (0.001 s)
    addtest() (0.005 s)
```

Failure Trace

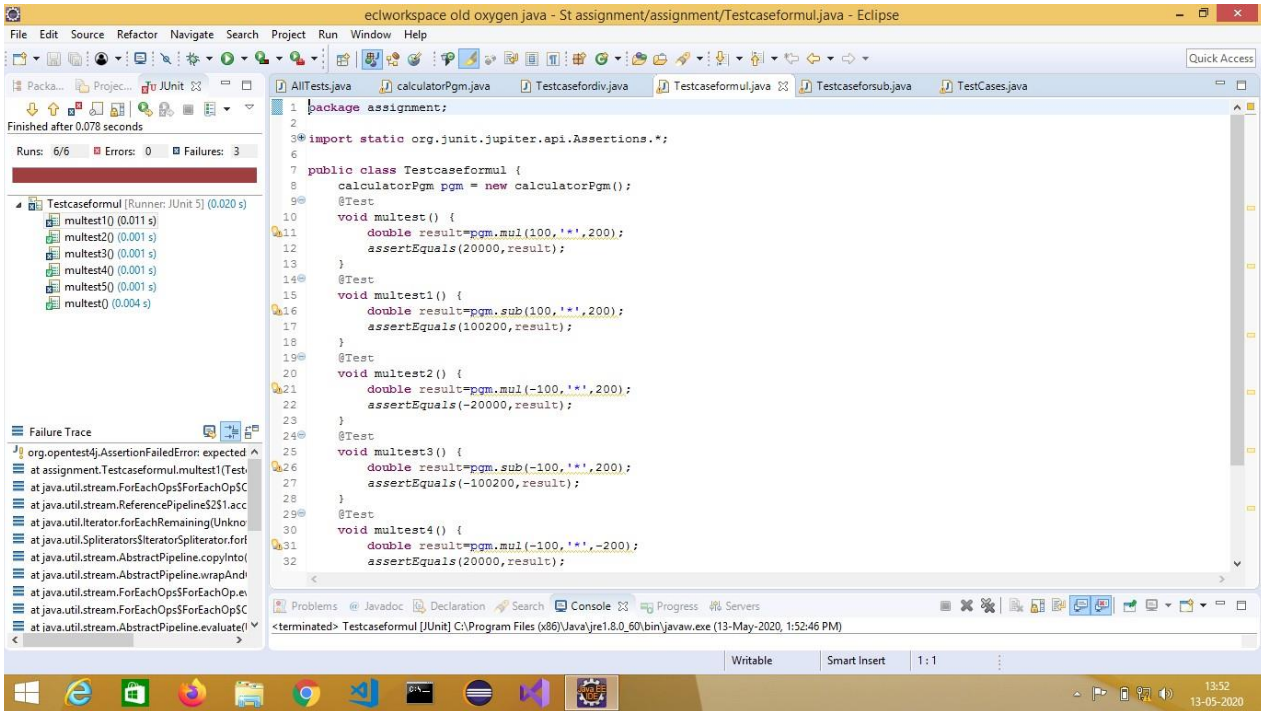
```
org.opentest4j.AssertionFailedError: expected
at assignment.TestCases.addtest1(TestCases.
at java.util.stream.ForEachOps$ForEachOp$C
at java.util.stream.ReferencePipeline$2$1.acc
at java.util.Iterator.forEachRemaining(Unkno
at java.util.Spliterators$IteratorSpliterator.for
at java.util.stream.AbstractPipeline.copyInto(
at java.util.stream.AbstractPipeline.wrapAnd
at java.util.stream.ForEachOps$ForEachOp.ev
at java.util.stream.ForEachOps$ForEachOp$C
at java.util.stream.AbstractPipeline.evaluate(
```

```java
import static org.junit.jupiter.api.Assertions.*;

public class TestCases {
    calculatorPgm pgm = new calculatorPgm();

    @Test
    void addtest() {
        double result=pgm.add(100,'+',200);
        assertEquals(300,result);
    }
    @Test
    void addtest1() {
        double result=pgm.add(100,'+',200);
        assertEquals(310,result);
    }
    @Test
    void addtest2() {
        double result=pgm.add(-100,'+',200);
        assertEquals(100,result);
    }
    @Test
    void addtest3() {
        double result=pgm.add(-100,'+',-200);
        assertEquals(-300,result);
    }
    @Test
    void addtest4() {
        double result=pgm.add(-100,'+',200);
        assertEquals(300,result);
    }
```

```
<terminated> TestCases [JUnit] C:\Program Files (x86)\Java\jre1.8.0_60\bin\javaw.exe (13-May-2020, 1:50:42 PM)
```

Figure 8.6: Test Cases of Subtraction Operation

The test cases of multiplication operation are run, with manual sample inputs testing the operation.



eclworkspace old oxygen java - St assignment/assignment/Testcaseformul.java - Eclipse

Finished after 0.078 seconds

Runs: 6/6 Errors: 0 Failures: 3

```
Testcaseformul [Runner: JUnit 5] (0.020 s)
    multest1() (0.011 s)
    multest2() (0.001 s)
    multest3() (0.001 s)
    multest4() (0.001 s)
    multest5() (0.001 s)
    multest() (0.004 s)
```

Failure Trace

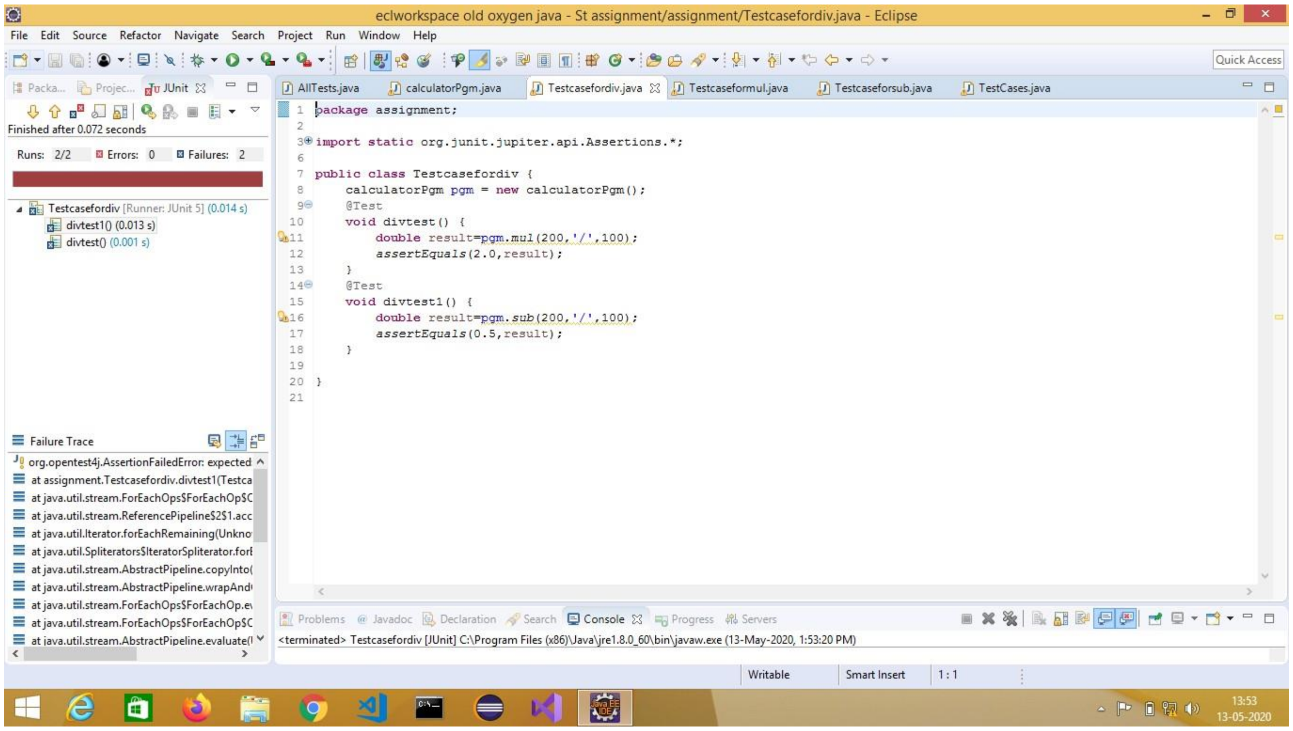
```
org.opentest4j.AssertionFailedError: expected
at assignment.Testcaseformul.multest1(Test
at java.util.stream.ForEachOps$ForEachOp$C
at java.util.stream.ReferencePipeline$2$1.acc
at java.util.Iterator.forEachRemaining(Unkno
at java.util.Spliterators$IteratorSpliterator.for
at java.util.stream.AbstractPipeline.copyInto(
at java.util.stream.AbstractPipeline.wrapAnd
at java.util.stream.ForEachOps$ForEachOp.ev
at java.util.stream.ForEachOps$ForEachOp$C
at java.util.stream.AbstractPipeline.evaluate(
```

```java
package assignment;

import static org.junit.jupiter.api.Assertions.*;

public class Testcaseformul {
    calculatorPgm pgm = new calculatorPgm();
    @Test
    void multest() {
        double result=pgm.mul(100,'*',200);
        assertEquals(20000,result);
    }
    @Test
    void multest1() {
        double result=pgm.sub(100,'*',200);
        assertEquals(100200,result);
    }
    @Test
    void multest2() {
        double result=pgm.mul(-100,'*',200);
        assertEquals(-20000,result);
    }
    @Test
    void multest3() {
        double result=pgm.sub(-100,'*',200);
        assertEquals(-100200,result);
    }
    @Test
    void multest4() {
        double result=pgm.mul(-100,'*',-200);
        assertEquals(20000,result);
```

```
<terminated> Testcaseformul [JUnit] C:\Program Files (x86)\Java\jre1.8.0_60\bin\javaw.exe (13-May-2020, 1:52:46 PM)
```

Figure 8.7: Test Cases of Multiplication Operation

The results of the individual tests are displayed on the left-hand side, column section, wherein a tick mark denotes success of a test and an x denotes the failure of the test. The test cases of division operation are run, with manual sample inputs testing the operation.


eclworkspace old oxygen java - St assignment/assignment/Testcasefordiv.java - Eclipse

Finished after 0.072 seconds

Runs: 2/2 Errors: 0 Failures: 2

```
Testcasefordiv [Runner: JUnit 5] (0.014 s)
    divtest1() (0.013 s)
    divtest() (0.001 s)
```

```java
package assignment;

import static org.junit.jupiter.api.Assertions.*;

public class Testcasefordiv {
    calculatorPgm pgm = new calculatorPgm();
    @Test
    void divtest() {
        double result=pgm.mul(200,'/',100);
        assertEquals(2.0,result);
    }
    @Test
    void divtest1() {
        double result=pgm.sub(200,'/',100);
        assertEquals(0.5,result);
    }

}
```

Failure Trace

```
org.opentest4j.AssertionFailedError: expected
at assignment.Testcasefordiv.divtest1(Testca
at java.util.stream.ForEachOps$ForEachOp$C
at java.util.stream.ReferencePipeline$2$1.acc
at java.util.Iterator.forEachRemaining(Unkno
at java.util.Spliterators$IteratorSpliterator.forE
at java.util.stream.AbstractPipeline.copyInto(
at java.util.stream.AbstractPipeline.wrapAnd(
at java.util.stream.ForEachOps$ForEachOp.ev
at java.util.stream.ForEachOps$ForEachOp$C
at java.util.stream.AbstractPipeline.evaluate(
```

```
<terminated> Testcasefordiv [JUnit] C:\Program Files (x86)\Java\jre1.8.0_60\bin\javaw.exe (13-May-2020, 1:53:20 PM)
```

File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Packa... Projec... JUnit
AllTests.java calculatorPgm.java Testcasefordiv.java Testcaseformul.java Testcaseforsub.java TestCases.java
Problems Javadoc Declaration Search Console Progress Servers
Writable Smart Insert 1 : 1
13:53
13-05-2020

Figure 8.8: Test Cases of Division Operation

The results of the individual tests are displayed on the left-hand side, column section, wherein a tick mark denotes success of a test and an x denotes the failure of the test. The Figure 8.7: Test Cases of Multiplication Operation and Figure 8.8: Test Cases of Division Operation in the figure above represents the same.

9. TEST CASES TABLE AND REPORT

Operation	**No. of TestCases**	**Pass**	**Fail**	**Error**
Add	6	2	3	1
Div	3	1	1	1
Mul	6	3	3	0
Sub	7	3	4	0

Report:

Total Test Cases	: 22
Passed	: 9
Failed	: 11
Error	: 2

10. ADVANTAGES OF JUNIT

- Enables writing test cases while developing the software that helps test early and detect issues.
- Ensure the functionality is performing as expected every time the code is modified by the use of repeatable automated test cases.
- Supported by all IDE including Eclipse, Intellij, Netbeans, RAD etc.
- Integrates with Ant and Maven(CI) that enables execution of test suites or test cases as part of the build process, capturing test result and reporting.
- Accurate, efficient testing process and manual based testing can be done.

11. DISADVANTAGES OF JUNIT

- It cannot do dependency testing.
- It's not suitable for higher level testing i.e. for large test suites.
- Group testing cannot be done in J-Unit.
- Can't create HTML reports of test cases. You need to use ANT to create tests HTML reports.

12. CONCLUSION

Therefore, in conclusion it can be concluded that the junit testing tool can be used to do manual testing in an efficient and robust manner. It also provides the user to set up his own test cases in order to test the given code in user-friendly environment of eclipse.