JADE Unit Testing

Version 2018

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

JADE Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of JADE Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2019 JADE Software Corporation Limited.

All rights reserved.

JADE is a trademark of JADE Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

Contents

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

JADE Unit Testing

Introduction

Unit testing is an important tool that is used to identify defects in a system during development. The key idea is to develop tests for a unit of code (which can be a fragment, a method, or a module) as the code is being written.

The JADE unit testing framework enables unit tests to be executed automatically and for the results of the tests to be captured. By doing so, the test provides a contract of the minimum functionality that a unit of code must satisfy.

In addition to providing a contract of functionality, unit tests provide the following additional benefits.

- Enables refactoring without regression (that is, ensuring that the module still works correctly)
- Eliminates uncertainty in the units
- Enables a bottom-up testing style approach
- Documents the functionality provided by the unit and how to use it

JadeTestCase Class

Unit tests are added as subclasses of the **JadeTestCase** class in RootSchema.

The **JadeTestCase** class is an abstract class that provides common functionality for unit tests as well as providing the **JadeTestListenerIF** interface that allows callback methods to report on the progress of a set of test cases.

The following methods are defined in the **JadeTestCase** class. By using these methods in test cases, you can define the expected behavior of your tested module.

Method	Code Example
assert	`// Sets the message to be displayed upon test failure` `assert("The unit test failed!");`

assertEquals	`// Compares the first parameter to the second,`
	`// and fails the test if they are not equal.`
	`assertEquals(2, (1 + 1));`

assertEqualsMsg	`// Compares the first parameter to the second,and fails the`
	`// test if they are not equal, displaying the given message.`
	`assertEqualsMsg("one plus one was not two", 2, (1 + 1));`

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

Method	Code Example

assertFalse	`// Asserts that a given boolean evaluates to false. If the`
	`// condition evaluates to true, the test fails.`
	`assertFalse(1 = 2);`

assertFalseMsg	`// Asserts that a given boolean evaluates to false. If the`
	`// condition evaluates to true, the test fails, displaying`
	`// the given message.`
	`assertFalseMsg("One equals two was true", 1 = 2);`

assertNotNull	`// Asserts that an object exists. If this is not the case,`
	`// the test fails.`
	`assertNotNull(myObject);`

assertNotNullMsg	`// Asserts that an object exists. If this is not the case,`
	`// the test fails, and the given message is displayed.`
	`assertNotNullMsg("Object doesn't exist.", myObject);`

assertNull	`// Asserts that an object doesn't exist. If it does exist,`
	`// the test fails.`
	`assertNull(myObject);`

assertNullMsg	`// Asserts that an object doesn't exist. If it does exist,`
	`// the test fails, and the given message is displayed.`
	`assertNullMsg("Expected object to not exist", myObject);`

assertTrue	`// Asserts that a condition is true. If the condition is`
	`// false, the test fails.`
	`assertTrue(1 + 1 = 2);`

assertTrueMsg	`// Asserts that a condition is true. If the condition is`
	`// false, the test fails, and the given message is displayed.`
	`assertTrueMsg("One plus one should equal two", 1 + 1 = 2);`

expectedException	`// Registers an exception that the test should produce.`
	`// The test will fail if no exception is raised.`
	`expectedException(1090); // Null object reference`

info	`// Outputs the given message without failing the test` `info("The test is going fine so far.");`

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

Writing a Test Case

When writing a unit test, you must first create a subclass of the **JadeTestCase** class and add the unit tests of methods in that subclass. Unit tests have the **unitTest** method option, shown in the following example.

```
exampleTest() unitTest;

vars
    calc : Calculator;
begin
    calc := create Calculator() transient;
    assertEquals(2, calc.add(1, 1));

epilog
    delete calc;
end;
```

Note When using the **unitTest** method option, the method cannot have any parameters or a return type.

In addition to the **unitTest** method option, you can use the following method options to establish pre- and post-conditions of unit tests.

Method Option	The method will be run…
unitTestBefore	Before every method of the class that uses the **unitTest** method option. It is used to enforce pre-conditions that may be impacted during test execution. The **unitTestBefore** method option should be defined in one method only in each class.
unitTestAfter	After every method of the class that uses the **unitTest** method option. It is used to enforce post-conditions that may be impacted during test execution.
unitTestBeforeClass	Once before the first **unitTest** method of a class is run. It is used to enforce pre-conditions that are required specifically for the tests of a class and that are unlikely to be impacted during test execution.
unitTestAfterClass	Once after the last **unitTest** method of a class is run. It is used to enforce post-conditions that are required at the end of the tests of a class but do not need to be maintained between tests within the class.
unitTestBeforeAll	Once before the first **unitTest** method of the first class is run. It is used to enforce pre-conditions that are relevant to all test classes and that are unlikely to be impacted during test execution. The method in which this option is specified must be defined directly in the **JadeTestCase** class.
unitTestAfterAll	Once after the last **unitTest** method of the last class is run. It is used to enforce post-conditions that are required at the end of the entire set of tests across all test classes and that do not need to be maintained between tests or test classes. The method in which this option is specified must be defined directly in the **JadeTestCase** class.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

Note Specify each of the **unitTestBefore**, **unitTestAfter**, **unitTestBeforeClass**, and **unitTestAfterClass** method options in this table in one method only in each class.

The **unitTestBefore** and **unitTestAfter** method options are defined only on the base **JadeTestCase** class, and there can be one instance only of these method options in a schema.

Running a Test Case

You can run unit test methods from any of the following.

- The Class Browser, by selecting:
  - A unit test class or method and then pressing F9.
  - The **Unit Test** command from the Jade menu.

  If you selected a single test method, the test runner runs only that test. If you selected a test class, the test runner runs only the tests of that class.
- The Schema Browser, by selecting:
  - A schema that contains at least one unit test class and then pressing F9.
  - The **Unit Test** command from the Jade menu.

  The Unit Test Runner form is then opened, displaying all of the tests of that schema. The tests are not run until you select one or more tests and then click the **Run** button.
- Calling the **JadeTestRunner** class **runTests** method from your code, passing the collection of test classes in to that method as a parameter.

Note As it is not common practice to run JADE tests from code, you should do so only if you specifically need more control over how the results of the tests are presented.

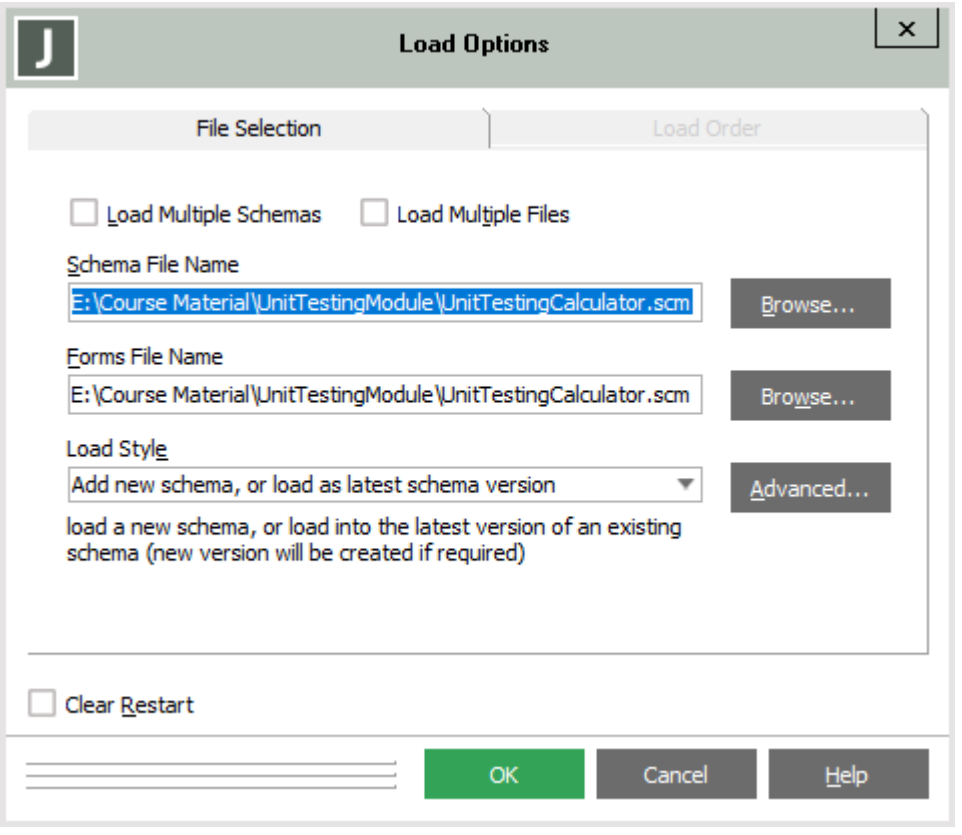
Exercise 1 – Loading the UnitTestingCalculator Schema

In this exercise, you will load the **UnitTestingCalculator** schema and then locate the **JadeTestCase** class.

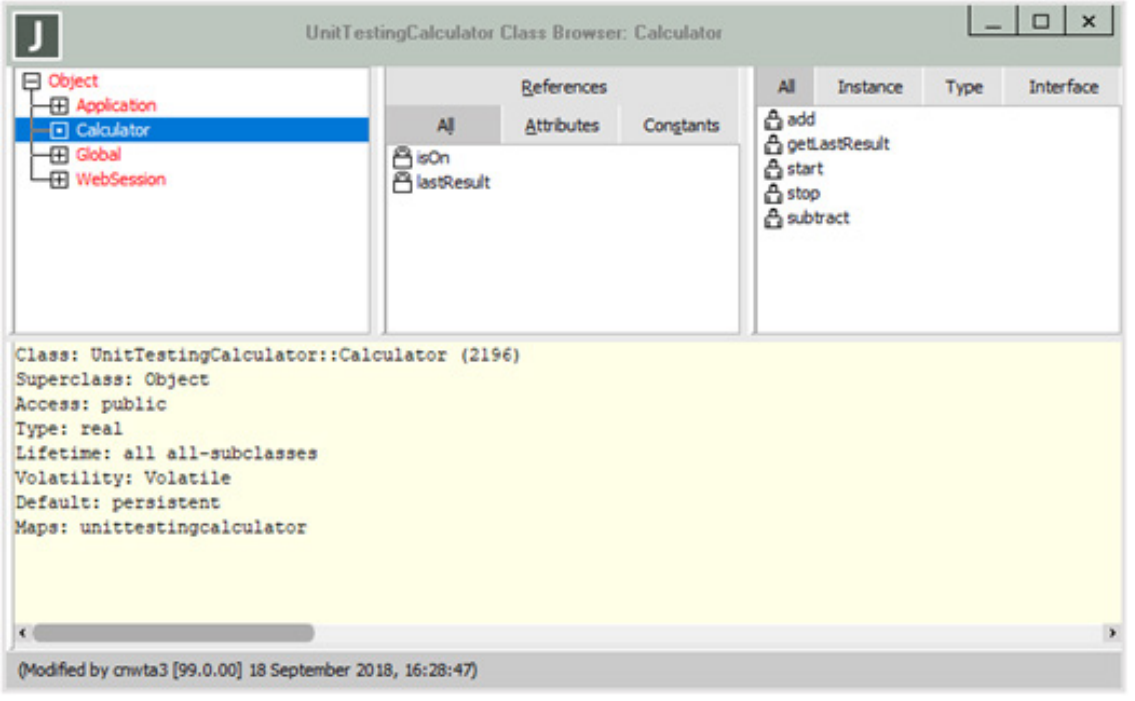
1. Right-click on **RootSchema** in the Schema Browser and then select the **Load** command.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

2. Click the **Browse** button at the right of the **Schema File Name** text box and then locate and select the **UnitTestingCalculator.scm** file from the provided USB.

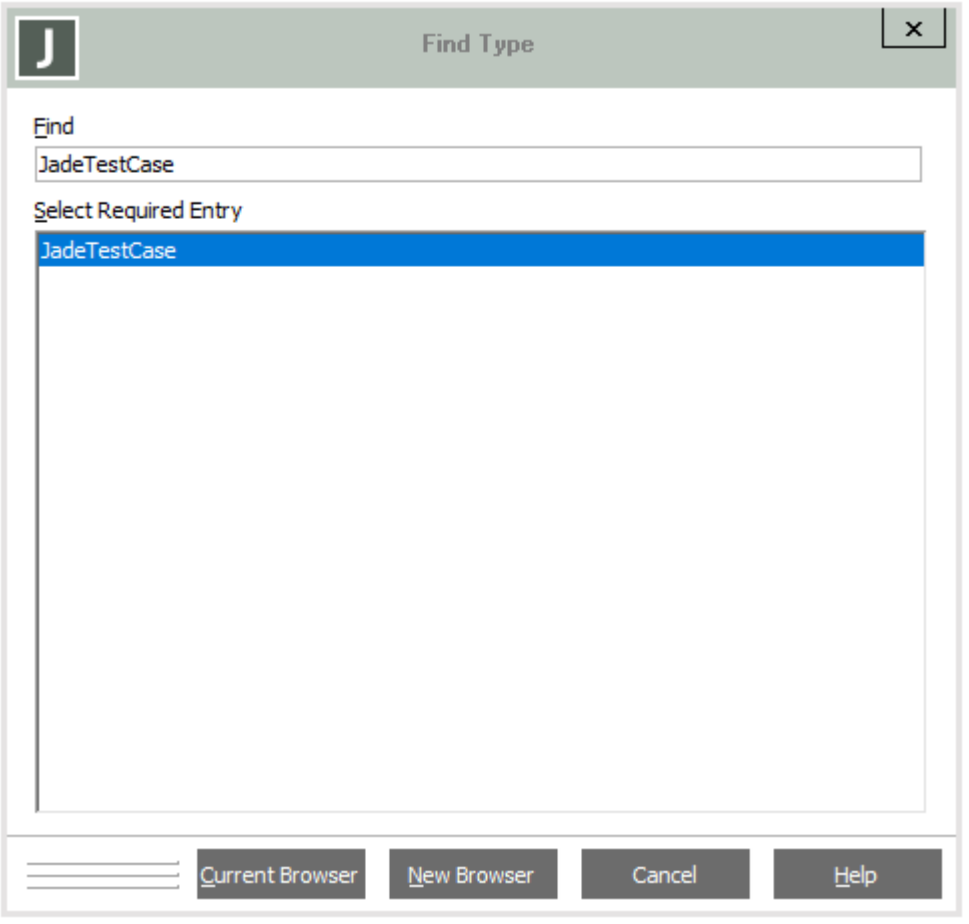


3. Open the **UnitTestingCalculator** schema in the Class Browser and then select the **Calculator** class so that the properties and methods shown in the following image are displayed.



Copyright 2019, JADE Software Corporation Ltd. All rights reserved

4. Press F4 to display the Find Type dialog and then search for **JadeTestCase**.



5. Click the **Current Browser** button, to display the **JadeTestCase** class in the Class Browser.

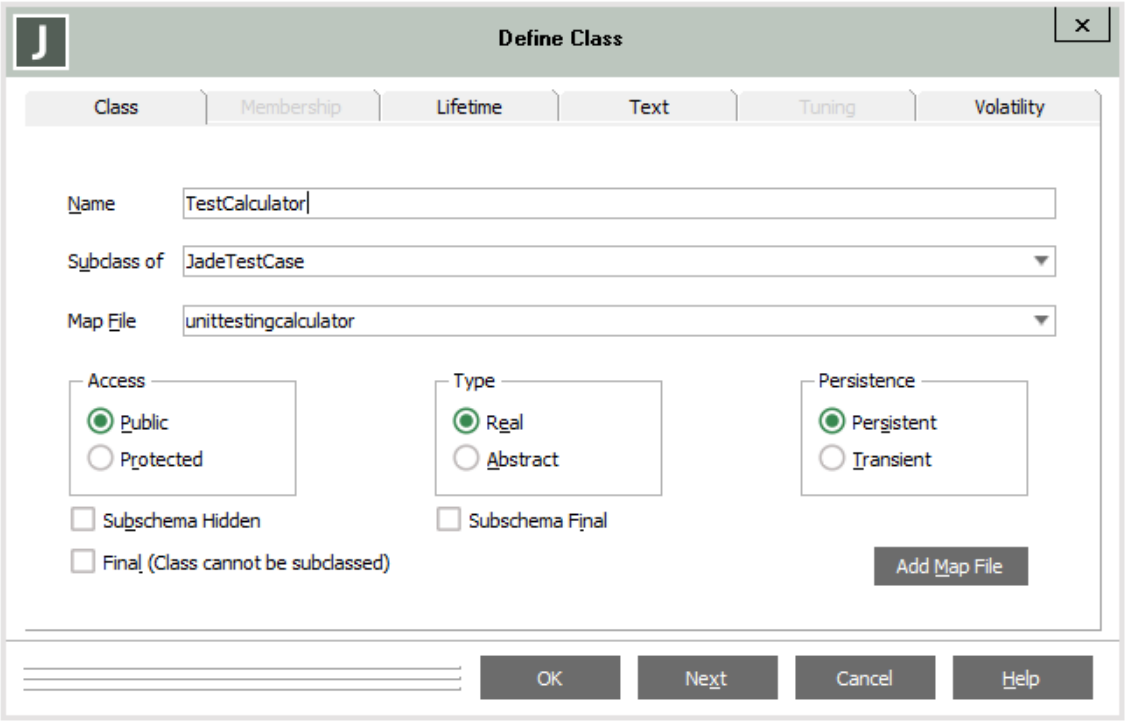
Exercise 2 – Writing a Test Case

In this exercise, you will write a unit test for the **Calculator** class **add** method.

1. Right-click on the **JadeTestCase** class and then select the **Add** command.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

In the Define Class dialog, specify **TestCalculator** as the class name and then click the **OK** button.



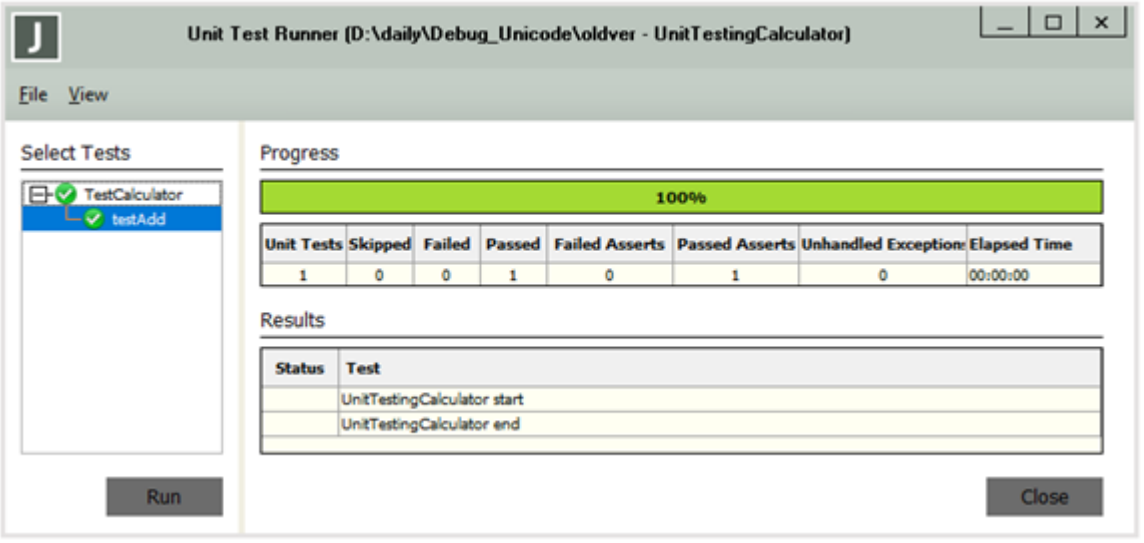
2. Add an attribute called **calc** of type **Calculator** to the class.
3. Add a new method called **testAdd**, and check the **Updating** and **Unit Test** check boxes.
4. Code the method, as follows.

```
testAdd() unitTest, updating;

vars
    actual      : String;
    expected    : String;

begin
    self.calc   := create Calculator() transient;
    self.calc.start();

    actual      := self.calc.add(1, 1);
    expected    := "2";

    assertEquals(expected, actual);

epilog
    self.calc.stop();
    delete self.calc;
end;
```

5. Run the method, by pressing F9.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

The Unit Test Runner form displays the results of the unit test, which should have passed.



Exercise 3 – Test Case Failure

In this exercise, you will write a test case for the test case for the faulty **subtract** method of the **Calculator** class.

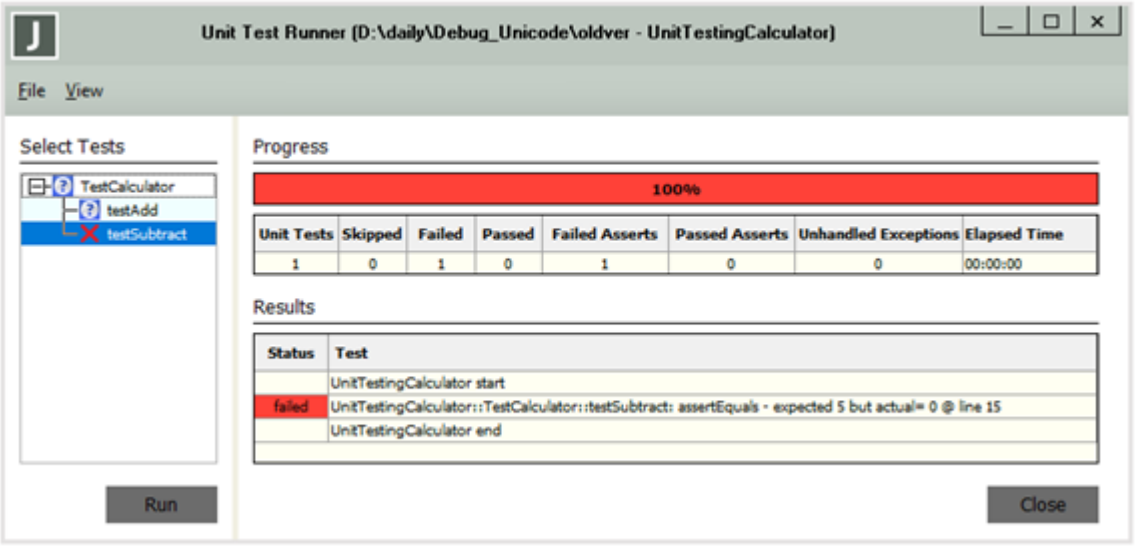
1. Add a **testSubtract** method to your **TestCalculator** class.
2. Code the method, as follows.

```
vars
    actual      : String;
    expected    : String;

begin
    self.calc   := create Calculator() transient;
    self.calc.start();

    actual      := self.calc.subtract(8, 3);
    expected    := "5";

    assertEquals(expected, actual);

epilog
    self.calc.stop();
    delete self.calc;
end;
```

3. Run the method, by pressing F9.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

The Unit Test Runner form then displays the results of the unit test, which should have failed.



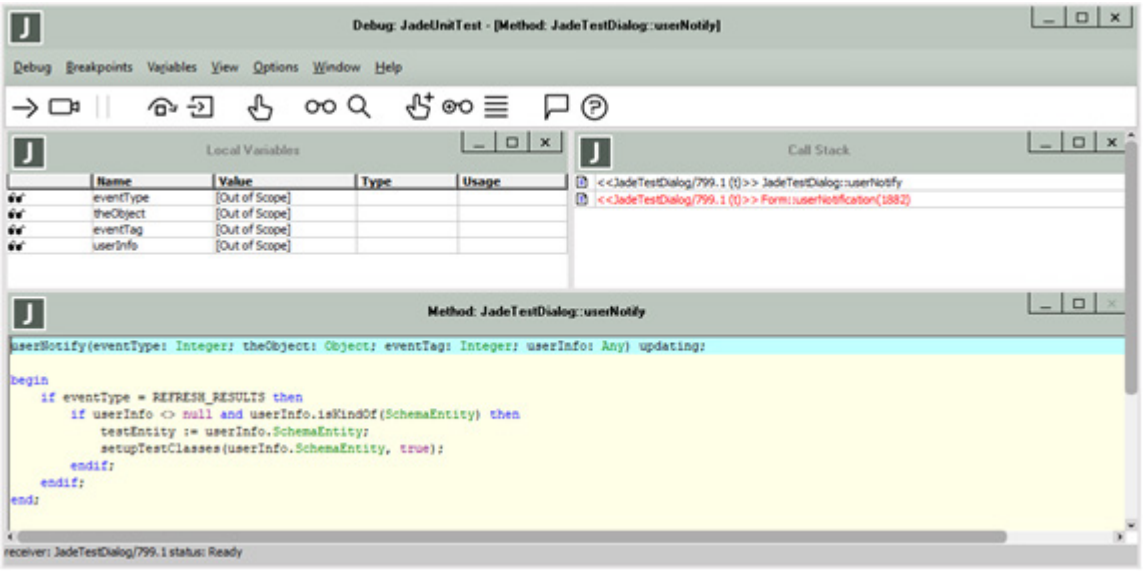
4. Looking at the **Results** section, we see that the unit test expected a result of **5** but the **testSubtract** method returned zero (**0**).

5. Use the F5 key to add a breakpoint to the **testSubtract** method, as follows.

```
testSubtract() updating, unitTest;

vars
    actual      : String;
    expected    : String;

begin
    self.calc   := create Calculator() transient;
    self.calc.start();

    actual      := self.calc.subtract(8, 3);
    expected    := "5";

    assertEquals(expected, actual);

epilog
    self.calc.stop();
    delete self.calc;
end;
```

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

6. Run the test again, but this time using Shift+F9 to run it in debug mode.



7. Click the **Continue execution** button at the far left of the toolbar (or press F9) to skip to the breakpoint, and then step into the code using the F7 shortcut key until you have found the fault.

Exercise 4 – unitTestBefore and unitTestAfter Method Options

You may have noticed that there is a significant amount of redundant code in the **testAdd** and **testSubtract** methods. In this exercise, you will refactor this code into **unitTestBefore** and **unitTestAfter** methods that will automatically run for each unit test.

1. Add a **setUp** method to your **TestCalculator** class, and code it as follows.

```
setUp() updating, unitTestBefore;

begin
    self.calc   := create Calculator() transient;
    self.calc.start();
end;
```

2. Add a **tearDown** method to your **TestCalculator** class, and code it as follows.

```
tearDown() updating, unitTestAfter;

begin
    self.calc.stop();
    delete self.calc;
end;
```

Note The **unitTestBefore** option in the method signature specifies that the method will automatically run *before* each unit test. The u**nitTestAfter** method option specifies that the method will automatically run *after* each unit test.

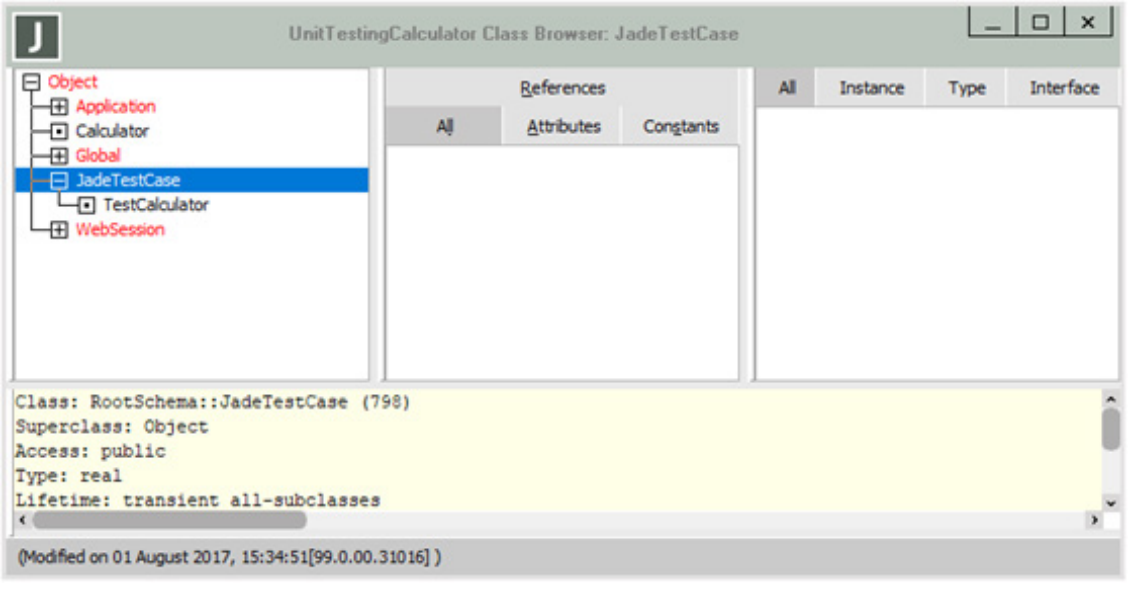
Copyright 2019, JADE Software Corporation Ltd. All rights reserved

3. Change the **testAdd** and **testSubtract** code to remove the redundant code that has now been factored out to your new **setUp** and **tearDown** methods, as shown in the following method examples.

```
testAdd() unitTest, updating;

vars
    actual      : String;
    expected    : String;

begin
    actual      := self.calc.add(1, 1);
    expected    := "2";

    assertEquals(expected, actual);
end;
```

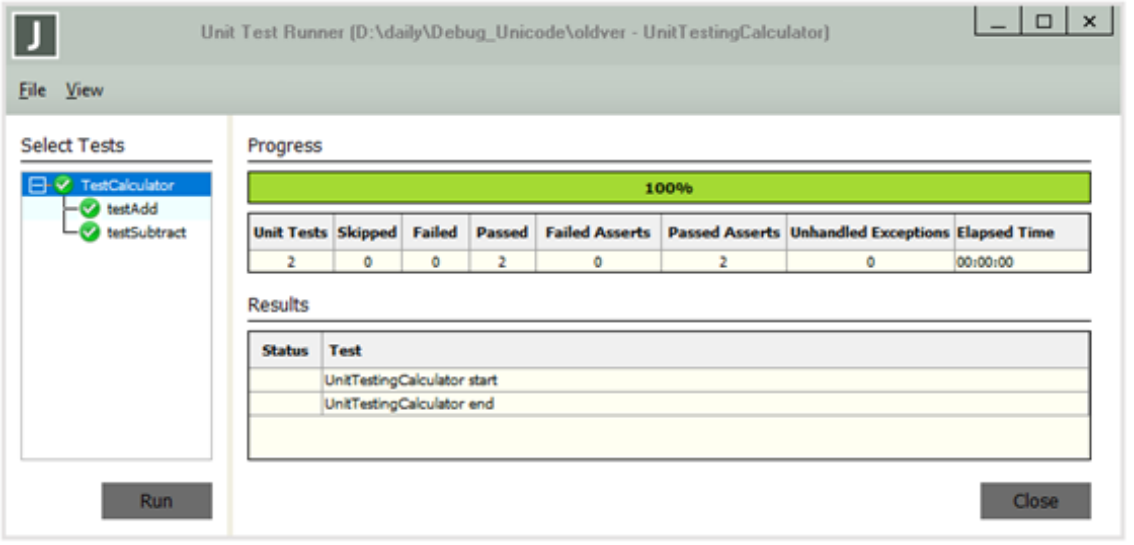
```
testSubtract() updating, unitTest;

vars
    actual      : String;
    expected    : String;

begin
    actual      := self.calc.subtract(8, 3);
    expected    := "5";

    assertEquals(expected, actual);
end;
```

4. Run all of the unit tests by selecting the **JadeTestCase** class in the Class Browser and then pressing F9.



Copyright 2019, JADE Software Corporation Ltd. All rights reserved

The Unit Test Runner form is then displayed.



Tip You can achieve a similar result by selecting the class in the Class Browser and then pressing F9.

Code Coverage

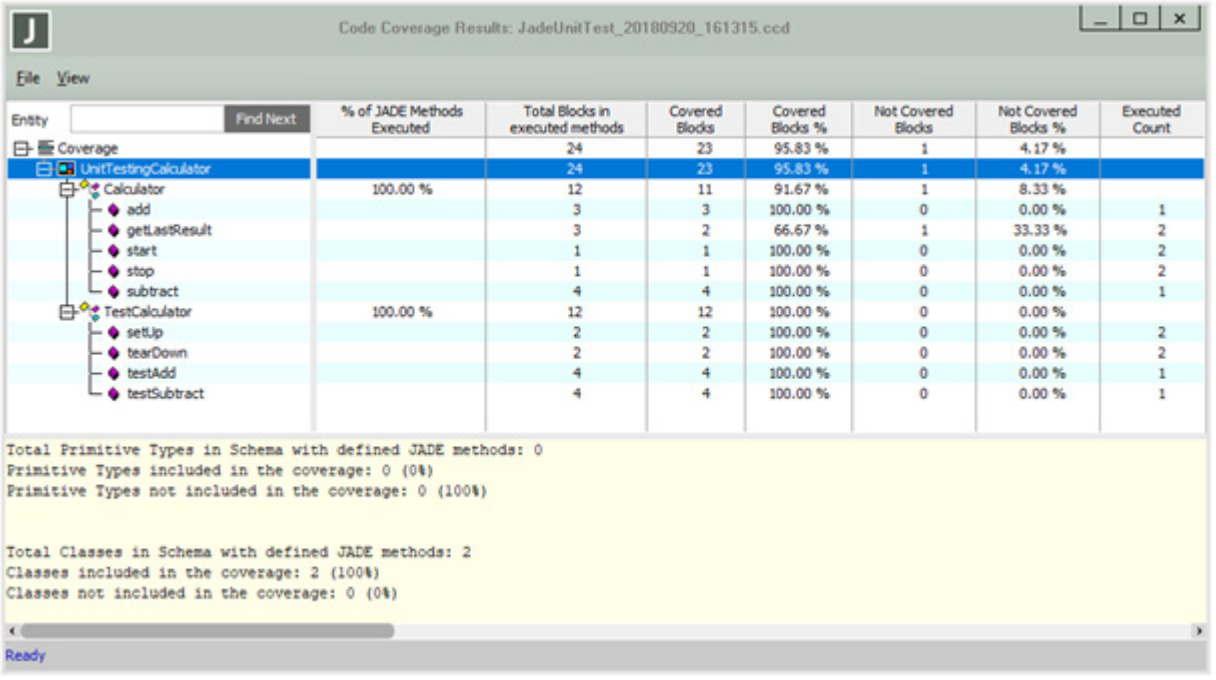
Code coverage is a measure used in software engineering to describe the degree to which the system's source code has been executed. It is a useful measure to assure the quality of a set of tests, as opposed to directly reflecting the quality of the system under test.

To monitor code coverage during unit tests, select the **Code Coverage** command in File menu of the Unit Test Runner form.

After running the unit tests for which you want to record code coverage, select the **View Code Coverage** command in File menu of the Unit Test Runner form.

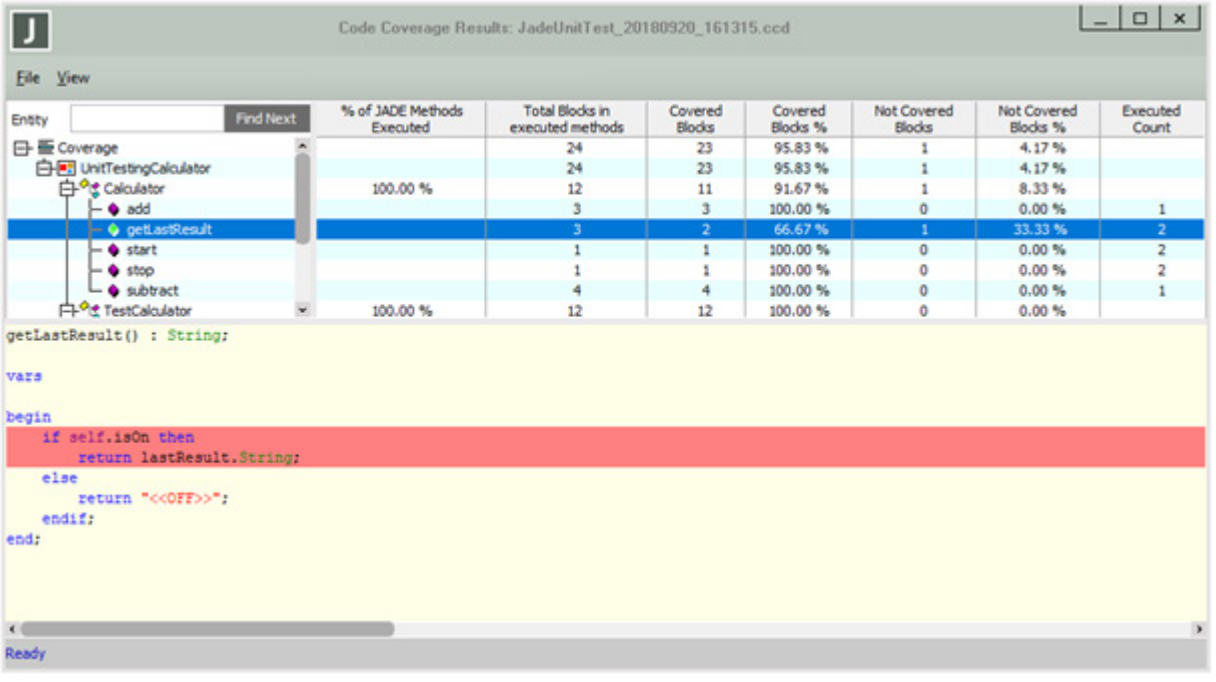
Copyright 2019, JADE Software Corporation Ltd. All rights reserved

The code coverage results are then displayed on a new form; that is, the Code Coverage Results Browser.



The Code Coverage Results Browser displays the number of JADE methods that were executed, and within those methods, the percentage of code blocks that were executed.

When selecting a method, the Code Coverage Results Browser displays the method code with the executed code highlighted in red.



By using the code coverage information, you can identify areas of code that have not been tested by a specific tests suite. This enables you to focus new tests on code that has not yet been executed by any test.

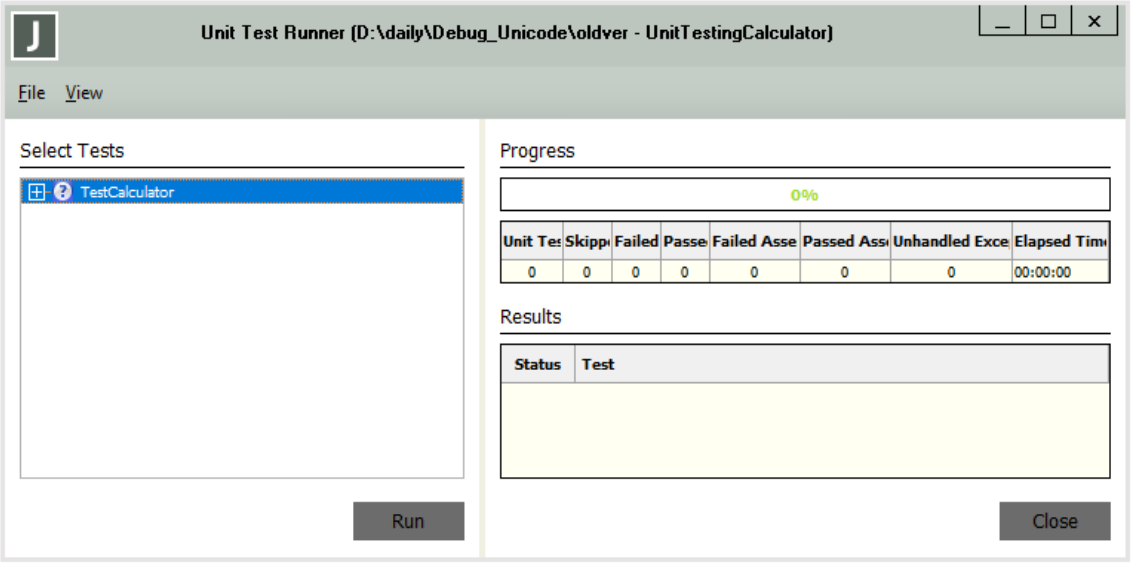
Copyright 2019, JADE Software Corporation Ltd. All rights reserved

Alternatively, if the code coverage is very high, it can provide evidence of how thoroughly the test suite exercises the code, which is one measure of the quality of a set of tests.

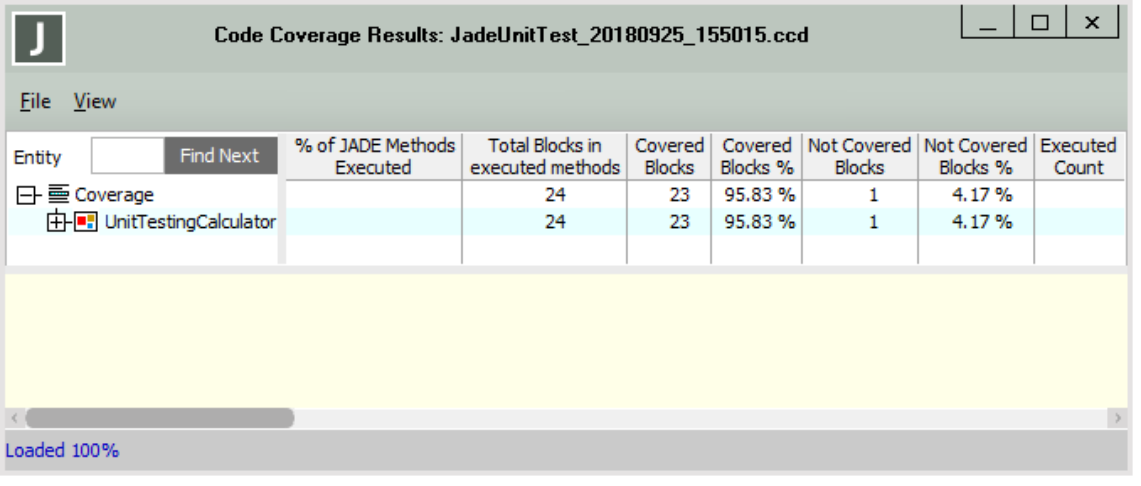
Exercise 5 – Viewing Code Coverage Results

In this exercise, you will use the code coverage functionality to find untested code.

1. Select the **UnitTestingCalculator** class from the Schema Browser and then press F9.



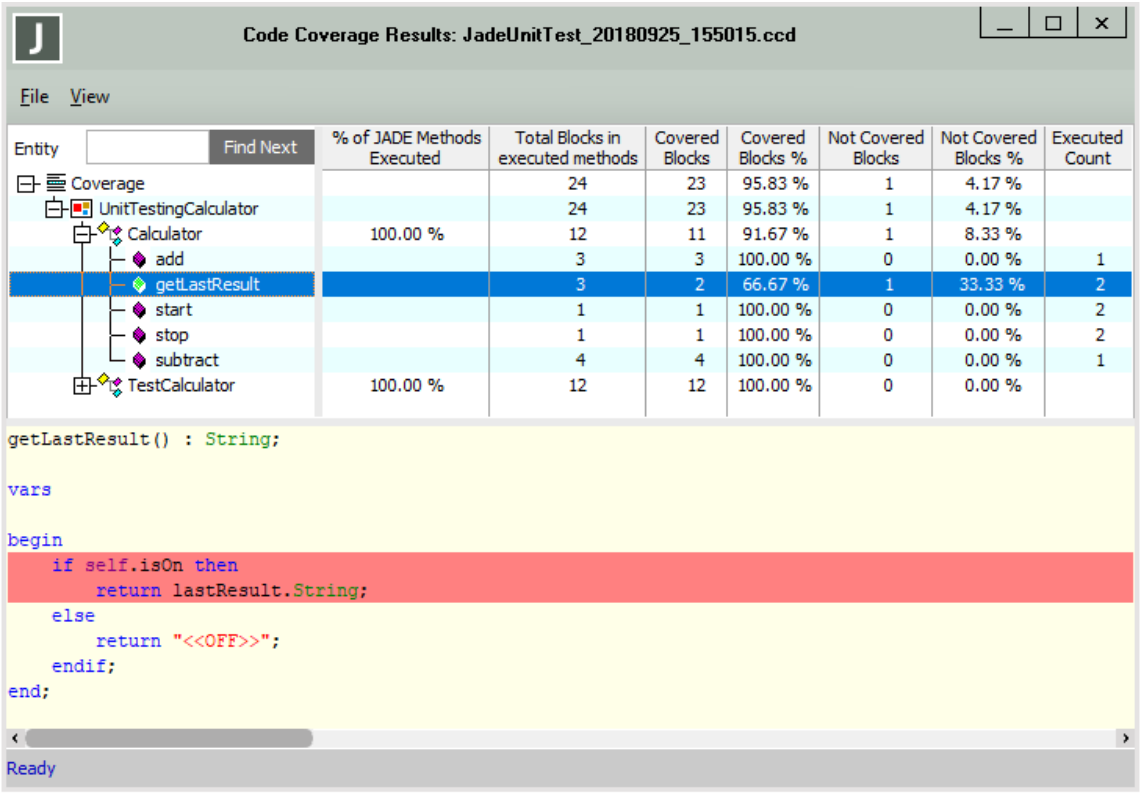
2. Select the **Code Coverage** command from the File menu.
3. Click the **Run** button, to run the tests.
4. When the tests have completed, select the **View Code Coverage** command from the File menu.



5. You will see that 95.83 percent of blocks have been covered.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

Expand the **Calculator** class in the **UnitTestingCalculator** schema and then locate the untested code block.



6. Write a new unit test to cover the situation in which you try to use the calculator while it is off.

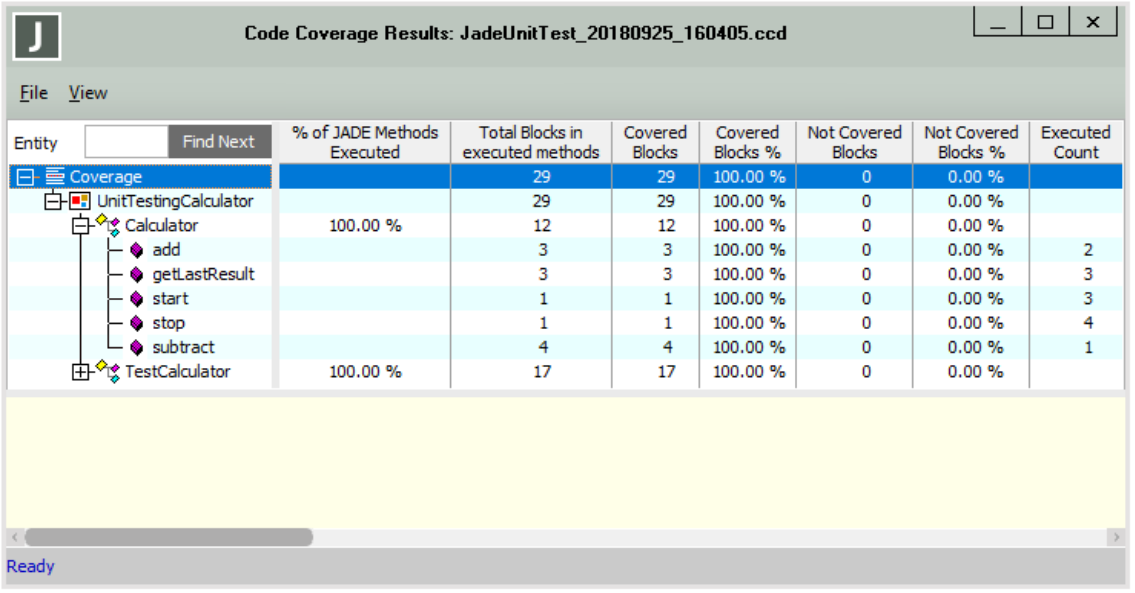
Note While 100 percent code coverage can be a good goal for smaller projects, it can often be impractical in larger systems. It is only one of many measures of code quality.

Copyright 2019, JADE Software Corporation Ltd. All rights reserved

Exercise 6 – Saving Code Coverage Results

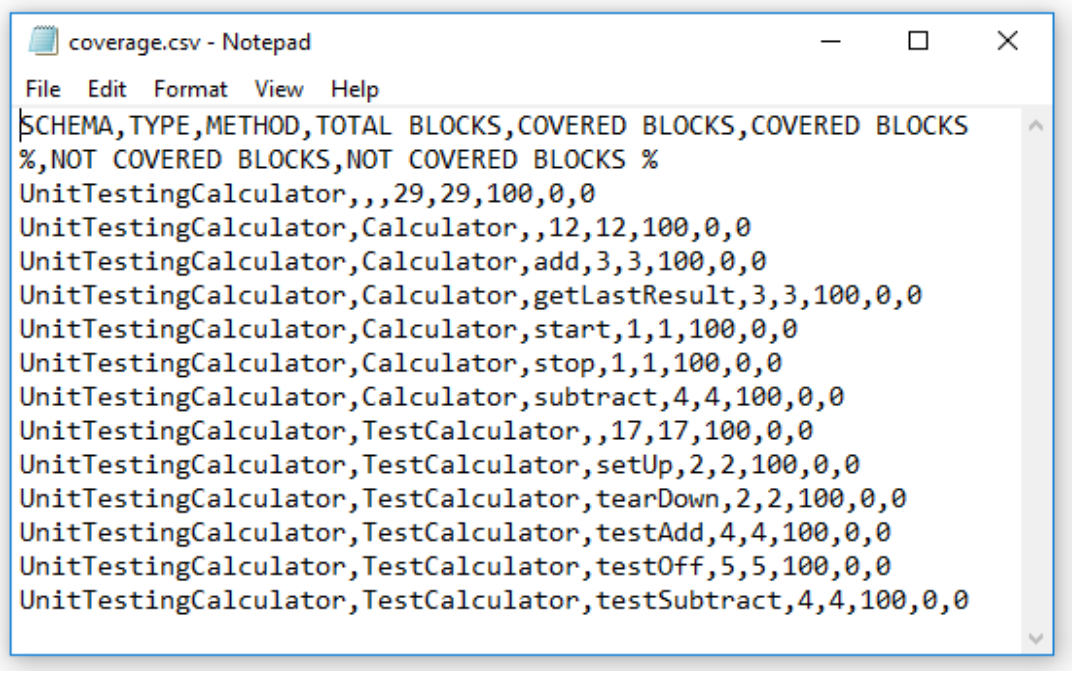
In this exercise, you will save code coverage results to a Comma-Separated Values (.**csv**) file.

1. Open the Code Coverage Results Browser as you did in the previous exercise.



2. Select the **Save As CSV** command from the File menu. A common dialog is then displayed, to enable you to select the folder to which to save the code coverage results. (The Desktop or the Documents folder are good options.)

3. Open the CSV file from the File Explorer, to view its contents.



Copyright 2019, JADE Software Corporation Ltd. All rights reserved