JUnit 5

Prepared By - Nirmala Kumar Sahu

JUnit 5

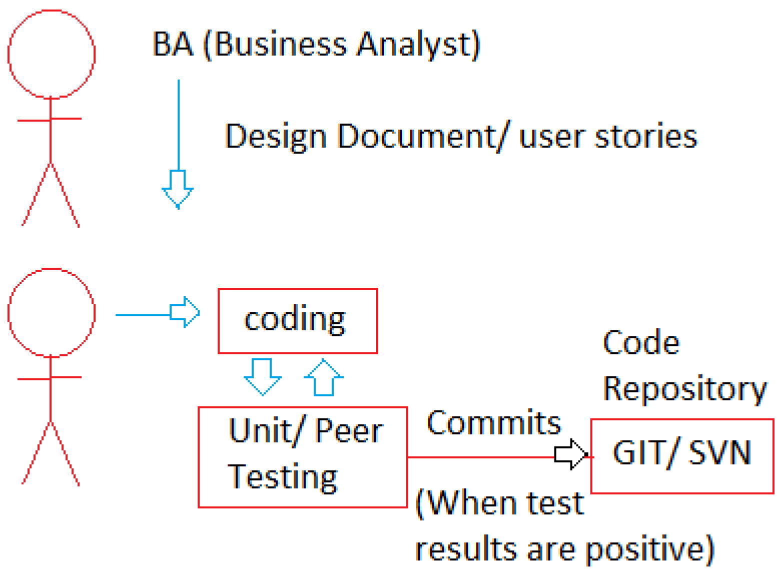
Pre-requisite: Core Java + Eclipse IDE knowledge + (Advance java basics)

JUnit

Unit Testing: The test done by programmer on own piece of code is called unit testing.
Peer Testing: The unit testing done on 1 programmer's code/ task by his colleague programmer is called Peer testing.

Note: Testing = matching expected results with actual results.
if matched then test result is positive (Test succeeded).
if not matched then test result is negative (Test failed).

Development --> Unit/ Peer Testing should be done continuously by programmer until test results are positive.



Testing done developers:

- Unit testing
- Peer testing
- Integration testing
  [Under monitoring of TL] (talks about modules/ applications integration]

Testing done Testers (QA Team): (They test whole project)

- Performance Test
- Navigation Test
- User-Experience Test
- System testing
- Load Testing

- Functional Testing
- Sanity Testing

  and etc.

Unit testing can be done in two ways:

a. Manual Unit Testing
b. Automated Unit Testing

Limitations of Manual Unit Testing:

- No Productivity (takes time).
- Writing test report manually is complex process (Excel sheet report).
- Presenting Test plans/ test cases to TL or superior is complex process.
- Test Regression (repeating the tests) is very complex.
- It is not industry standard.

- To overcome these problems, take the support Unit testing automation tools like JUnit, HttpUnit (for web applications), Mockito, TestNG and etc. for HttpUnit, Mockito and TestNG, JUnit is the base tool (Java based Unit Testing tools).

In Test results we can see:

a. success: expected results are matched actual results.
b. failures: expected results are not matched actual results.
c. errors: Unanticipated/ unexpected exception has come while testing the code.

Q. What is the difference between failure and error?

Ans. failure: actual code has given result but not matching expected result.
error: actual code has not given result rather it has thrown exception.

While working with Junit we can see 3 main components:

1. Service class/ Main class (Class to be tested) (1 or more).
2. Test case class (The class that contains test methods) (1 or more).
3. Test Suite: Allows to combine multiple test case classes to generated the test report (0 or 1) (optional).

Note: We can run each Test case class manually to generate Test report. But, if want get test report of all the classes together then take the support Test suite class.

- Eclipse IDE gives built-in Support for JUnit (i.e. eclipse gives JUnit libraries as built-in libraries)

To add Junit Libraries eclipse Java Project:

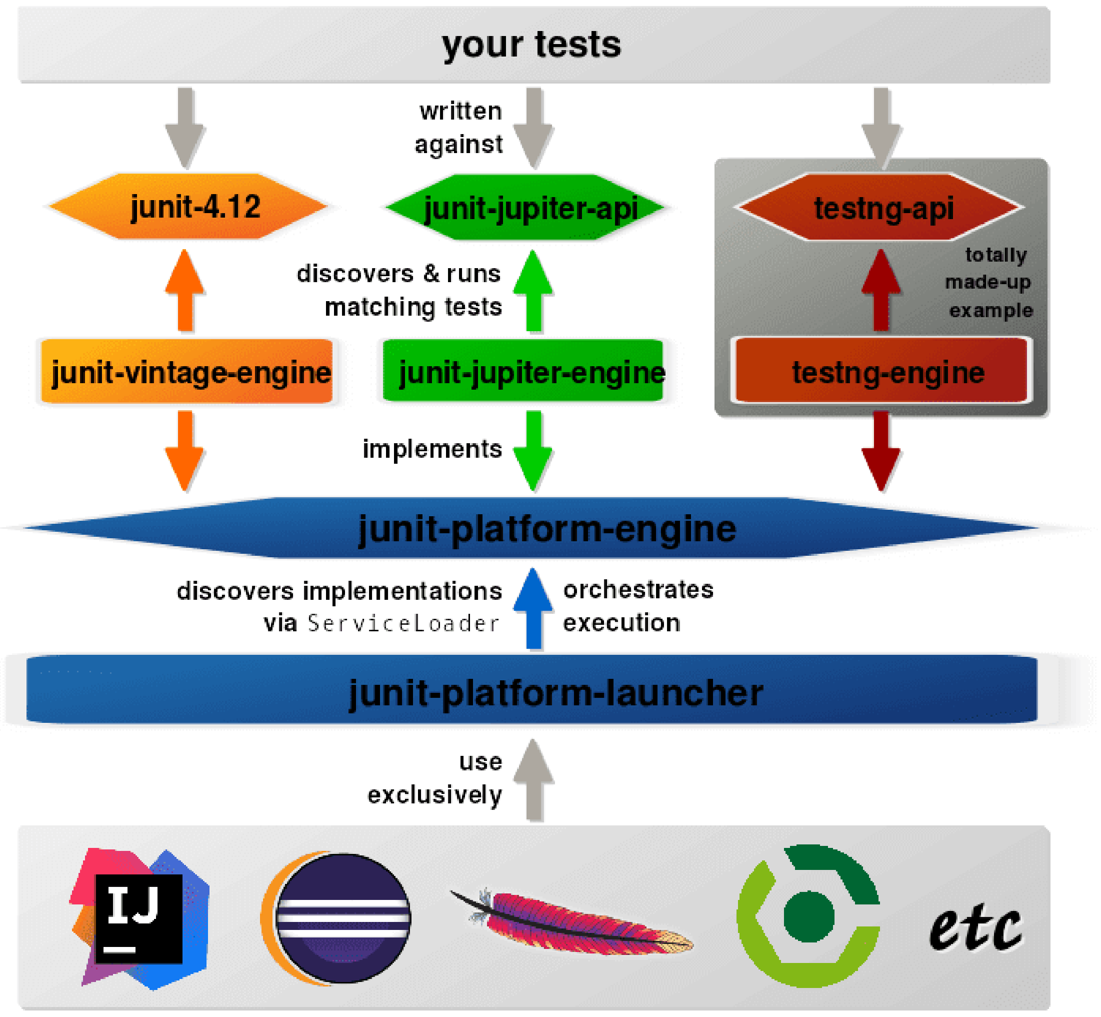
right click on project --> BUILDPATH --> configuration BUILDPATH --> libraries tab --> classpath --> add Libraries JUnit --> select JUnit5.

Note: It is recommended to use Junit with Maven/Gradle Project because it gives the support built-in Decompiled to see the source code and other advantages.

JUnit5 contains 3 runtime libraries:

a. JUnit Jupiter: JUnit5 libraries
b. JUnit Vintage: JUnit 3/ 4 libraries (for backward compatibility)
c. JUnit Integrations: To allow JUnit integration with Test Ng, Mockito and etc.

JUnit 5 Architecture Diagram:



JUnit-platform-engine: This engine is responsible to load test case classes, to create object for them and to call test methods.
JUnit-platform-launcher: Helps different IDEs, Tools like maven and etc. to search and get JUnit.

Junit5 Jupiter API gives:

a. Annotations
b. Assertions API

we use them in the development of Test Case classes and test methods.

$\rightarrow$ gives Assertions.assertXxx() methods (static methods) to match actual results with expected result and generate test report.

Annotations:

@Test
@DisplayName
@BeforeEach
@AfterEach
@BeforeAll
@AfterAll
@Tag
@ParameterizedTest
@ValueSource
@NullSource
@EmptySource
@NullAndEmptySource
@TestMethodOrder
@Order and etc.

Static Methods of Assertions class:

assertTrue
assertSame
assertNull
assertNotSame
assertNotEquals
assertNotNull
assertralse
assertEquals
assertArrayEquals
assertAll
assertThrows and etc.

Generally, The Testcase class name starts or ends with Test word and all test methods generally begins with "test" word.
e.g.
BankService (main class)
    p float calcSimpleIntrest(-, -)
    p float getBalance(-)

→ TestBankService or BankServiceTest (Test case class name)
        p v testcalcSimpleIntrest()
        p v testGetBalnace()
            we write multiple forms of these test methods to test main method/ service method in multiple angels

Note: In Test Case classes, for each business method/ service method we need to write variety of test methods not quantity test method.

Login App Code:
Possible test methods/ test plans
a. Test with Valid credentials
b. Test with Invalid credentials
c. Test with No credentials

Sum logic of two numbers:
Possible test methods/ test plans
- test with positives
- test with negatives
- test with mixed values
- test with zeros
- test with floating points
- test with chars/ strings

First Example Application:
Step 1: Create Maven Project in eclipse IDE as standalone Project by taking maven-archetype-quickstart as the Project archetype and change java version to 13.

[open pom.xml and change java version to 13, Right click on the Project - maven update the project].

File --> maven project --> next --> select maven-archetype-quickstart -> next -->

group Id: nit
artifact Id: JUnitTestProject1
package: com.nt.service --> next --> finish.

Step 2: Add JUnit5 Jupiter jar in pom.xml as dependent by collecting from mvnrepository.com in pom.xml under <dependencies> tag.

- junit-jupiter-api.5.7.0.jar
- junit-jupiter-params.5.7.0.jar [for some work with some advance topic]
- junit-jupiter-engine.5.7.0.jar

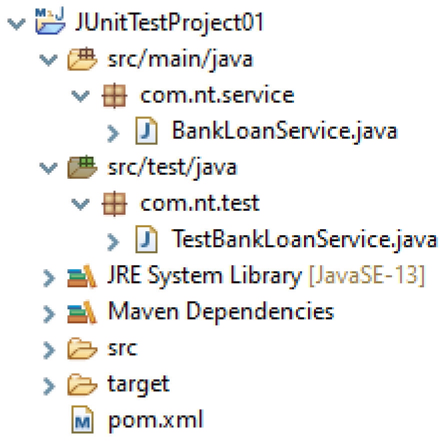
Step 3: Develop main class or service class in com.nt.service package of src/main/java folder.

src/main/java: To place main source code
src/test/java: To place unit testing code (test case/ test suit classes)

Step 4: Develop Testcase class with Test Methods in src/test/java folder having package com.nt.test.

Step 5: Then right click on the test class – Run As – JUnit Test

Directory Structure of JUnitTestProject01:



```
JUnitTestProject01
  src/main/java
    com.nt.service
      BankLoanService.java
  src/test/java
    com.nt.test
      TestBankLoanService.java
  JRE System Library [JavaSE-13]
  Maven Dependencies
  src
  target
  pom.xml
```

- Develop the above directory Structure and package, class, XML file and add the jar dependencies in pom.xml file then use the following code with in their respective file.

pom.xml

```
        <dependencies>
                <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
```

```xml
                <dependency>
                        <groupId>org.junit.jupiter</groupId>
                        <artifactId>junit-jupiter-api</artifactId>
                        <version>5.7.0</version>
                        <scope>test</scope>
                </dependency>
                <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-
jupiter-params -->
                <dependency>
                        <groupId>org.junit.jupiter</groupId>
                        <artifactId>junit-jupiter-params</artifactId>
                        <version>5.7.0</version>
                        <scope>test</scope>
                </dependency>
                <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-
jupiter-engine -->
                <dependency>
                        <groupId>org.junit.jupiter</groupId>
                        <artifactId>junit-jupiter-engine</artifactId>
                        <version>5.7.0</version>
                        <scope>test</scope>
                </dependency>
        </dependencies>
```

BankLoanService.java

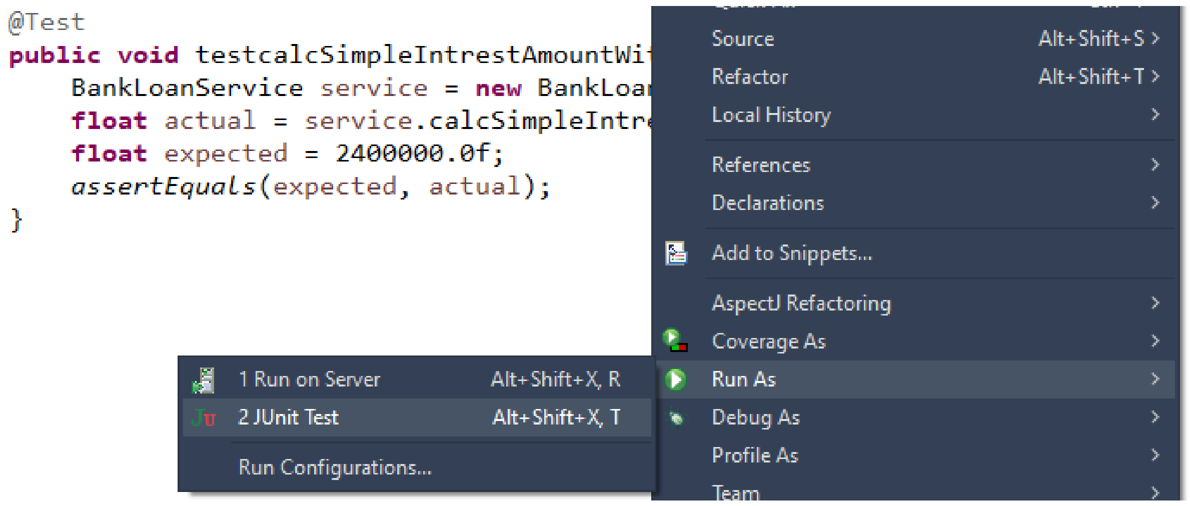
```java
package com.nt.service;

public class BankLoanService {
        public float calcSimpleIntrestAmount(float pAmount, float rate, float
time) {
        System.out.println("BankLoanService.calcSimpleIntrestAmount()");
                return pAmount*rate*time/100.0f;
        }
}
```

TestBankLoanService.java

```java
package com.nt.test;
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
import com.nt.service.BankLoanService;
```

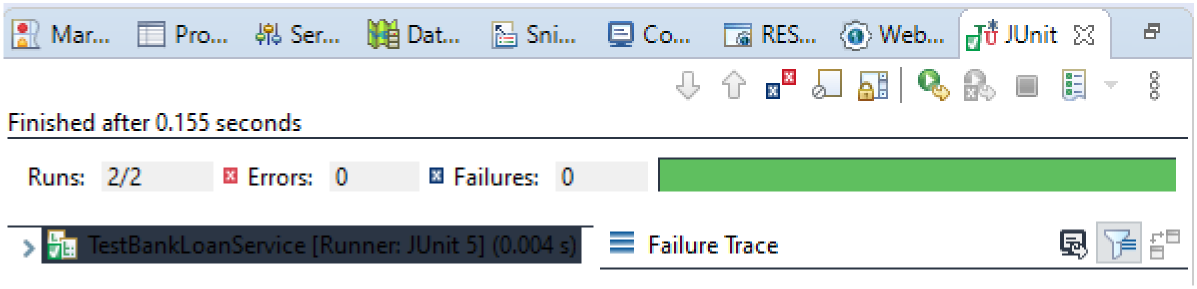
```
public class TestBankLoanService {
        @Test
        public void testcalcSimpleIntrestAmountWithSmallNumber() {
                BankLoanService service = new BankLoanService();
                float actual = service.calcSimpleIntrestAmount(100000, 2, 12);
                float expected = 24000.0f;
                assertEquals(expected, actual);
        }
        @Test
        public void testcalcSimpleIntrestAmountWithBigNumber() {
                BankLoanService service = new BankLoanService();
                float actual = service.calcSimpleIntrestAmount(10000000, 2,
12);
                float expected = 2400000.0f;
                assertEquals(expected, actual);
        }
}
```

For Run the Test class:



```
@Test
public void testcalcSimpleIntrestAmountWi
    BankLoanService service = new BankLoa
    float actual = service.calcSimpleIntre
    float expected = 2400000.0f;
    assertEquals(expected, actual);
}
```

Right click on the Test class then Run As then click JUnit Test. After that go to JUnit console to check the success and all.



```
Finished after 0.155 seconds

Runs: 2/2    Errors: 0    Failures: 0

TestBankLoanService [Runner: JUnit 5] (0.004 s)    Failure Trace
```

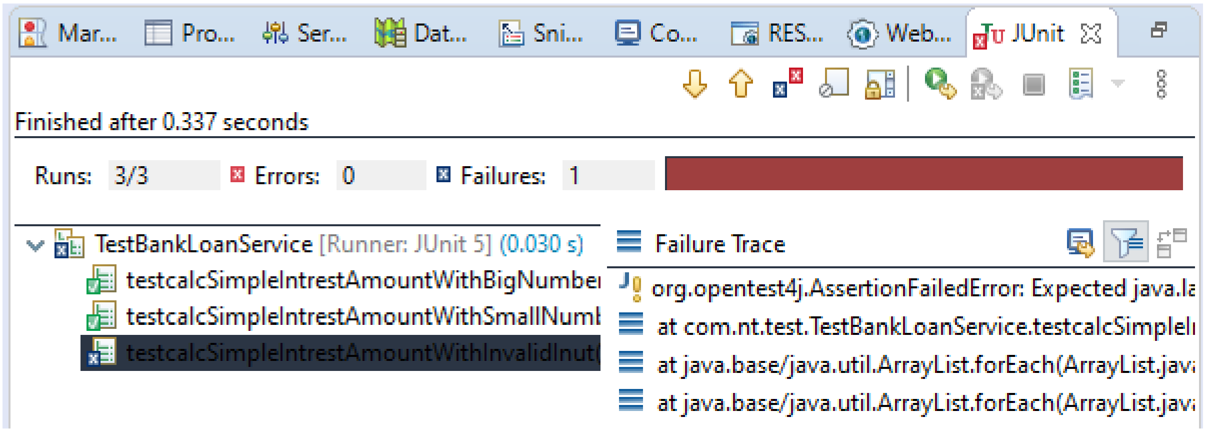
@Test: To make the method of Testcase class as the Test method
assertEquals()/ assertNotEquals(): To check whether expected result is equal or not with actual result and to generate test report.
assertThrows(): To check expected exception has come or not.

BankLoanService.java

```
	public float calcSimpleIntrestAmount(float pAmount, float rate, float time) {
	System.out.println("BankLoanService.calcSimpleIntrestAmount()");
		if (pAmount<=0||rate<=0||time<=0)
			throw new IllegalArgumentException("Invalid inputs");
		return pAmount*rate*time/100.0f;
	}
```

TestBankLoanService.java

```
	@Test
	public void testcalcSimpleIntrestAmountWithInvalidInut() {
		BankLoanService service = new BankLoanService();
		assertThrows(ArithmeticException.class, ()->{
			service.calcSimpleIntrestAmount(0, 0, 0);
		});
	}
```



```
Finished after 0.337 seconds
Runs: 3/3    Errors: 0    Failures: 1

TestBankLoanService [Runner: JUnit 5] (0.030 s)
    testcalcSimpleIntrestAmountWithBigNumbe
    testcalcSimpleIntrestAmountWithSmallNum
    testcalcSimpleIntrestAmountWithInvalidInut

Failure Trace
org.opentest4j.AssertionFailedError: Expected java.la
at com.nt.test.TestBankLoanService.testcalcSimpleI
at java.base/java.util.ArrayList.forEach(ArrayList.jav
at java.base/java.util.ArrayList.forEach(ArrayList.jav
```

You will get error if read the Failure Trace you can get the problem and followed by solution according that you can solve the problem.

- If you want to write your own message.

TestBankLoanService.java

```
	@Test
	public void testcalcSimpleIntrestAmountWithBigNumber() {
		BankLoanService service = new BankLoanService();
		float actual = service.calcSimpleIntrestAmount(10000000, 2, 
12);
		float expected = 2400000.0f;
		assertEquals(expected, actual, "may results not matching");
	}

	@Test
	public void testcalcSimpleIntrestAmountWithInvalidInut() {
		BankLoanService service = new BankLoanService();
		assertThrows(ArithmeticException.class, ()->{
			service.calcSimpleIntrestAmount(0, 0, 0);
		},  "may results not matching");
	}
```

Assert that expected and actual are equal within the given non-negative delta. Delta value is the different that is allowed in the results.

TestBankLoanService.java

```
	@Test
	public void testcalcSimpleIntrestAmountWithBigNumber() {
		BankLoanService service = new BankLoanService();
		float actual = service.calcSimpleIntrestAmount(10000000, 2, 
12);
		float expected = 2400000.12f;
		assertEquals(expected, actual, 0.5, "may results not 
matching");
	}
```

assertTimeout(): To check whether business method execution is completed in the specified time or not.

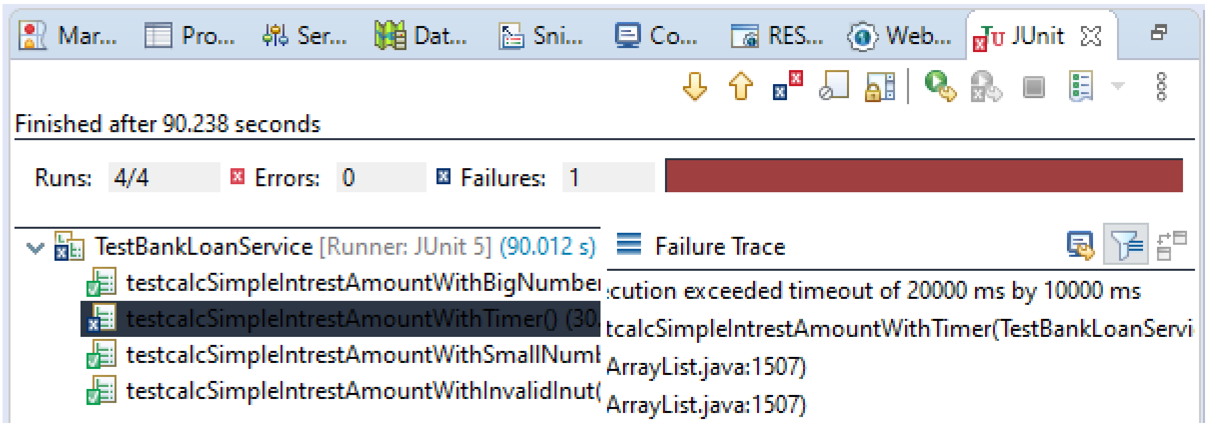
BankLoanService.java

```
	public float calcSimpleIntrestAmount(float pAmount, float rate, float 
time) {
	System.out.println("BankLoanService.calcSimpleIntrestAmount()");
		if (pAmount <=0 || rate <=0 || time <=0)
			throw new IllegalArgumentException("Invalid inputs");
```

```
                try {
                        Thread.sleep(30000);
                }
                catch (Exception e) {
                        e.printStackTrace();
                }
                return pAmount * rate * time / 100.0f;
        }
```

TestBankLoanService.java

```
        @Test
        public void testcalcSimpleIntrestAmountWithTimer() {
                BankLoanService service = new BankLoanService();
                assertTimeout(Duration.ofMillis(20000), ()->{
                        service.calcSimpleIntrestAmount(100000, 2, 12);
                });
        }
```



```
Finished after 90.238 seconds

Runs: 4/4    Errors: 0    Failures: 1

TestBankLoanService [Runner: JUnit 5] (90.012 s)      Failure Trace
    testcalcSimpleIntrestAmountWithBigNumbe           :cution exceeded timeout of 20000 ms by 10000 ms
    testcalcSimpleIntrestAmountWithTimer() (30        tcalcSimpleIntrestAmountWithTimer(TestBankLoanServi
    testcalcSimpleIntrestAmountWithSmallNumt          ArrayList.java:1507)
    testcalcSimpleIntrestAmountWithInvalidInut(       ArrayList.java:1507)
```

@BeforeEach: To place common logic that should execute before each Test method execution.
@AfterEach: To place common logic that should execute after the each Test method execution.
TestBankLoanService.java

```
        private BankLoanService service;

        @BeforeEach
        public void setUp() {
                service = new BankLoanService();
        }
```

```
            @Test
        public void testcalcSimpleIntrestAmountWithSmallNumber() {
        System.out.println("TestBankLoanService.testcalcSimpleIntrestAmou
ntWithSmallNumber()");
                float actual = service.calcSimpleIntrestAmount(100000, 2, 12);
                float expected = 24000.0f;
                assertEquals(expected, actual,  "may results not matching");
        }

        @Test
        public void testcalcSimpleIntrestAmountWithBigNumber() {
        System.out.println("TestBankLoanService.testcalcSimpleIntrestAmou
ntWithBigNumber()");
                float actual = service.calcSimpleIntrestAmount(10000000, 2,
12);
                float expected = 2400000.12f;
                assertEquals(expected, actual, 0.5, "may results not
matching");
        }

        @AfterEach
        public void clear() {
                service = null;
        }
```

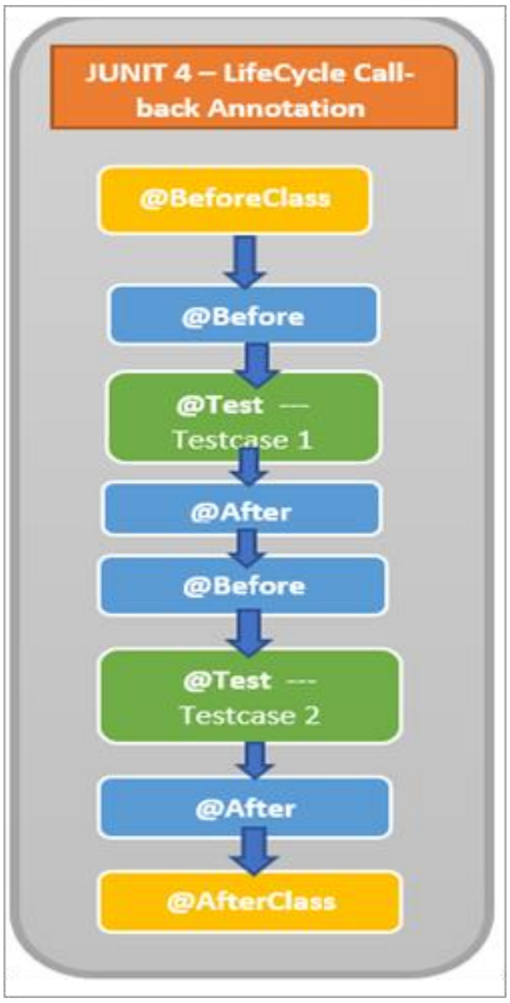
@BeforeAll: To write common logic only for 1 time for all test methods.
@AfterAll: To place cleanup logic for all test methods.

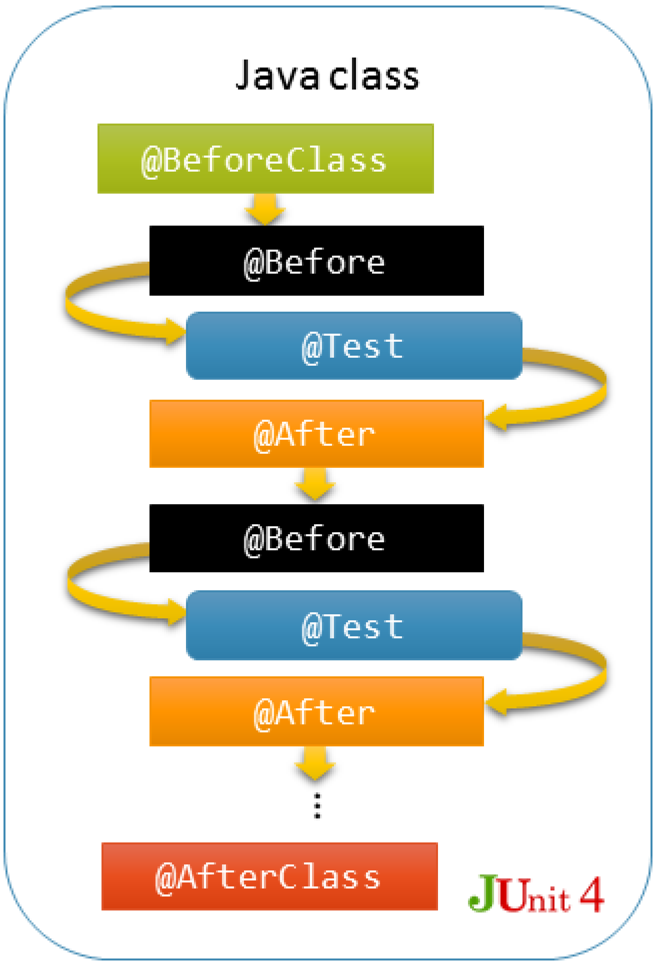
- These methods must be taken as static methods

TestBankLoanService.java

```
        private static BankLoanService service;

        @BeforeAll
        public static void setUpOnce() {
                service = new BankLoanService();
        }

        @AfterAll
        public static void clearOnce() {
                service=null;
        }
```

JUnit Life cycle call back Annotation:

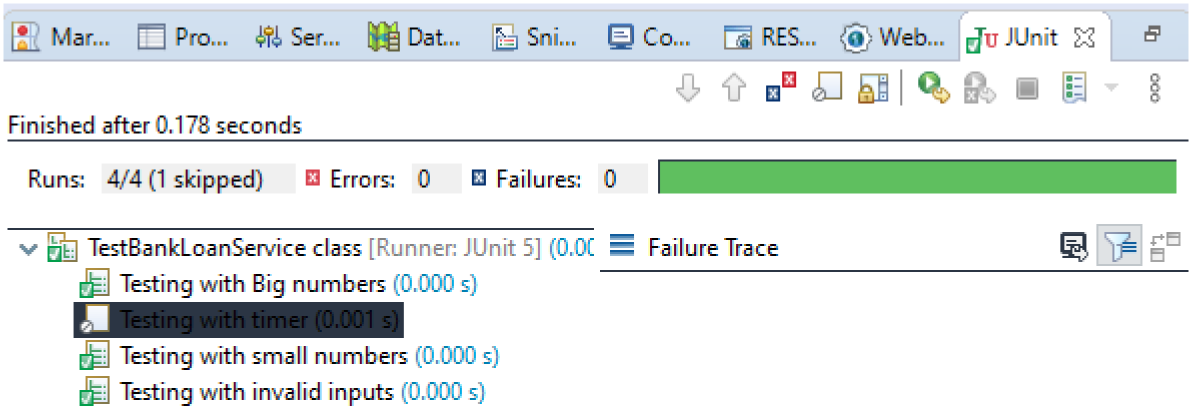




@Disabled: Marks test method as skipped/ disabled/ ignored test method
@DisplayName: To give programmer choice non-technical names to Test case class and test methods.

TestBankLoanService.java

```
@DisplayName("TestBankLoanService class")
public class TestBankLoanService {
        @Test
        @DisplayName("Testing with small numbers")
        public void testcalcSimpleIntrestAmountWithSmallNumber() {
        System.out.println("TestBankLoanService.testcalcSimpleIntrestAmou
ntWithSmallNumber()");
                float actual = service.calcSimpleIntrestAmount(100000, 2, 12);
                float expected = 24000.0f;
                assertEquals(expected, actual,  "may results not matching");
        }
        @Test
        @Disabled
        @DisplayName("Testing with timer")
        public void testcalcSimpleIntrestAmountWithTimer() {
        System.out.println("TestBankLoanService.testcalcSimpleIntrestAmou
ntWithTimer()");
                assertTimeout(Duration.ofMillis(20000), ()->{
                        service.calcSimpleIntrestAmount(100000, 2, 12);
                });
        }
```



```
Finished after 0.178 seconds
Runs: 4/4 (1 skipped)   Errors: 0   Failures: 0

TestBankLoanService class [Runner: JUnit 5] (0.00
    Testing with Big numbers (0.000 s)
    Testing with timer (0.001 s)
    Testing with small numbers (0.000 s)
    Testing with invalid inputs (0.000 s)
```

@TestMethodOrder: Useful to `specify` execution order of test methods with different possibilities .

TestBankLoanService.java

```
@DisplayName("TestBankLoanService class")
//@TestMethodOrder(value = MethodName.class)
//@TestMethodOrder(value = MethodOrderer.DisplayName.class)
//@TestMethodOrder(value = OrderAnnotation.class)
@TestMethodOrder(value = MethodOrderer.Random.class)
public class TestBankLoanService {
```

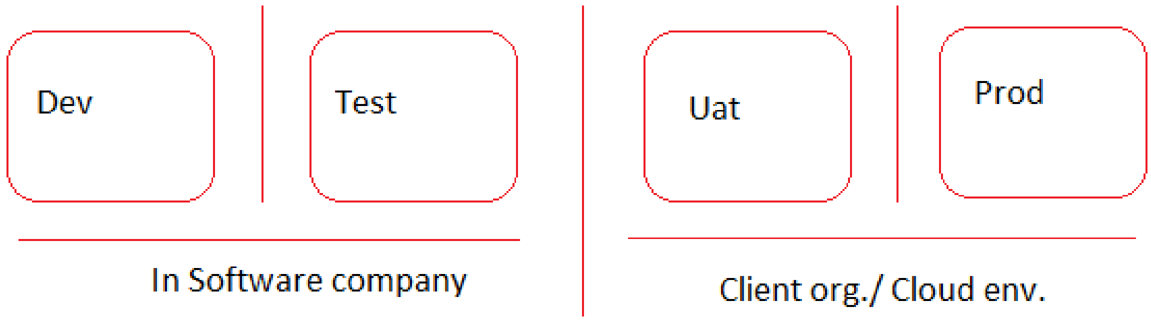
MethodOrderer (I):

It is having multiple inner classes implementing same MethodOrderer (I) they are

a. MethodName
b. DisplayName (gives ambiguity with @DisplayName, so specify MethodOrderer.DisplayName.class)
c. OrderAnnotation (best, we should add @Order(n) on top of test methods while using this option. n-> priority number high value indicates low priority and low value indicate high priority)

d. Random (default but gives ambiguity, so specify MethodOrderer.Random.class)
e. AlphaNumeric (deprecated)

In real Scenarios we need to execute the application/ project in 4 environments:



@Tag: Useful to mark test methods to execute only in certain environment like "dev", "test", "uat", "prod" and etc. So, that we can write separate test methods for "dev", "test" environment based light weight setup like using MySQL, Tomcat server and etc. and similarly we can write separate test methods for "uat", "prod" environment based heavy weight production ready setup like using Oracle, WebLogic , Wildfly and etc.

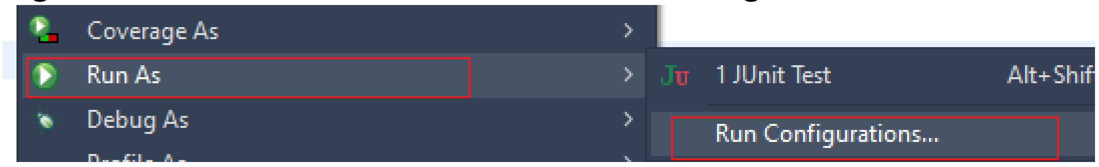
TestBankLoanService.java

```
	@Test
	@DisplayName("Testing with small numbers")
	@Tag("dev")
	public void testcalcSimpleIntrestAmountWithSmallNumber() {
	System.out.println("TestBankLoanService.testcalcSimpleIntrestAmou
ntWithSmallNumber()");
		float actual = service.calcSimpleIntrestAmount(100000, 2, 12);
		float expected = 24000.0f;
		assertEquals(expected, actual,  "may results not matching");
	}
	@Test
	@DisplayName("Testing with Big numbers")
	@Tag("uat")
	public void testcalcSimpleIntrestAmountWithBigNumber() {
	System.out.println("TestBankLoanService.testcalcSimpleIntrestAmou
ntWithBigNumber()");
		float actual = service.calcSimpleIntrestAmount(10000000, 2,
12);
```

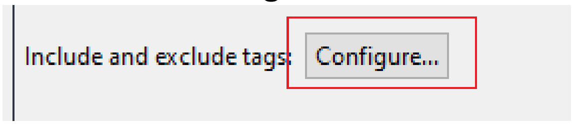
```
                float expected = 2400000.12f;
                assertEquals(expected, actual, 0.5, "may results not
matching");
        }
        @Test
        @DisplayName("Testing with invalid inputs")
        @Tag("uat")
        public void testcalcSimpleIntrestAmountWithInvalidInut() {
        System.out.println("TestBankLoanService.testcalcSimpleIntrestAmou
ntWithInvalidInut()");
                assertThrows(IllegalArgumentException.class, ()->{
                        service.calcSimpleIntrestAmount(0, 0, 0);
                }, "may results not matching");
        }
        @Test
        @Disabled
        @DisplayName("Testing with timer")
        @Tag("dev")
        public void testcalcSimpleIntrestAmountWithTimer() {
        System.out.println("TestBankLoanService.testcalcSimpleIntrestAmou
ntWithTimer()");
                assertTimeout(Duration.ofMillis(20000), ()->{
                        service.calcSimpleIntrestAmount(100000, 2, 12);
                });
        }
```

- While running Test case class, we need to specify tags to include and exclude

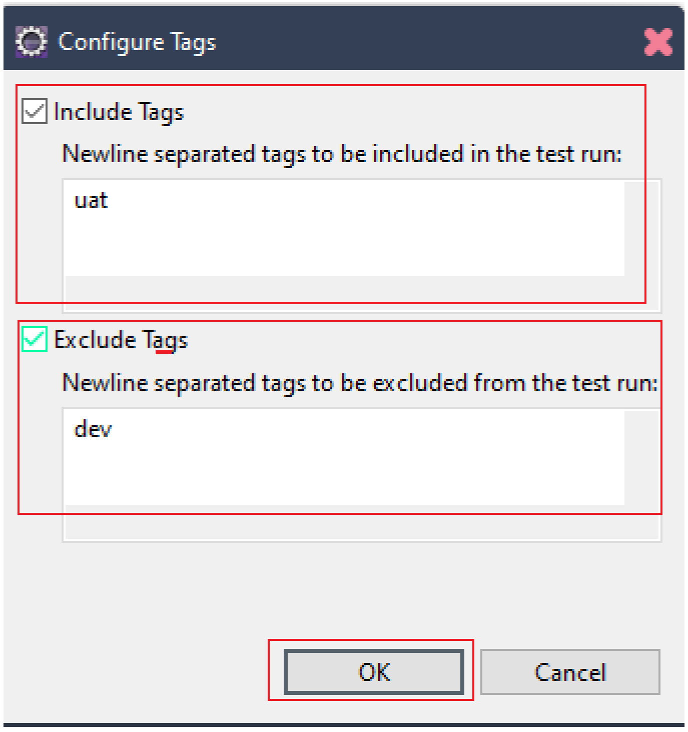
   a. In Eclipse Environment
   Right click Testcase class --> Run As --> Run Configurations -->



   Test tab --> Configure



Include tags: uat
exclude tags: dev --> ok


Configure Tags

☑ Include Tags

Newline separated tags to be included in the test run:

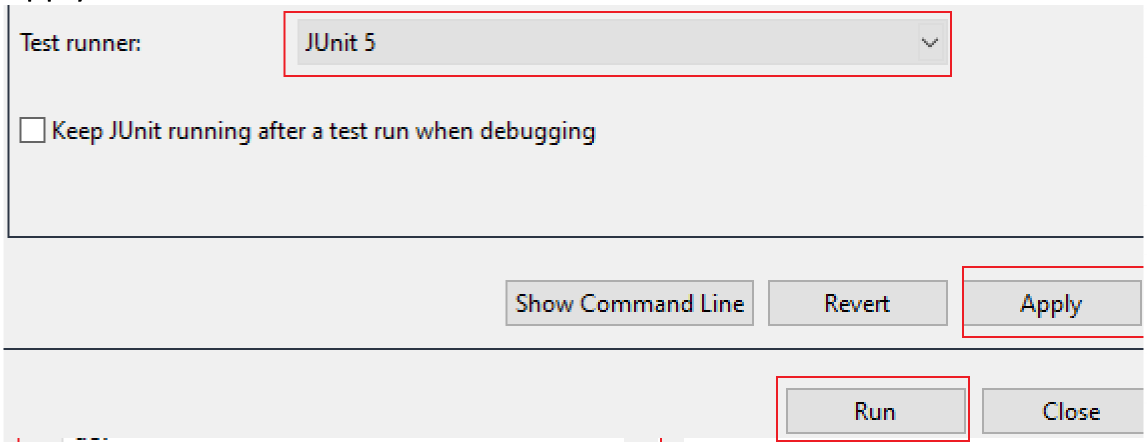
uat

☑ Exclude Tags

Newline separated tags to be excluded from the test run:

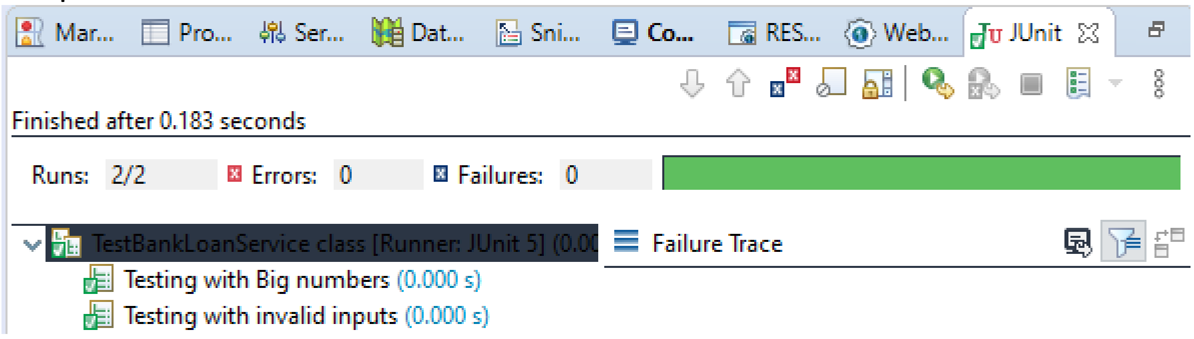
dev

Test runner: JUnit 5
Apply --> Run


Test runner: JUnit 5

☐ Keep JUnit running after a test run when debugging

Output:



Finished after 0.183 seconds

Runs: 2/2 Errors: 0 Failures: 0

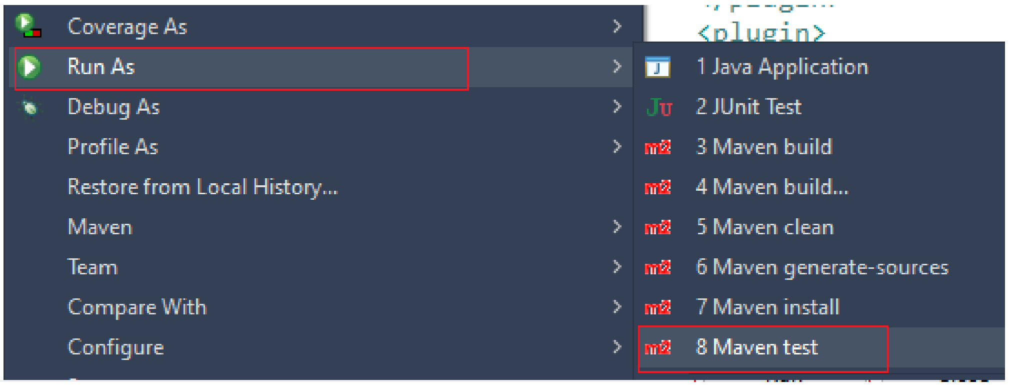
```
TestBankLoanService class [Runner: JUnit 5] (0.00
    Testing with Big numbers (0.000 s)
    Testing with invalid inputs (0.000 s)
```

b. In Maven Environment
   Specify the tag names as shown below under surefire plugin.

pom.xml

```
<plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.0.0-M5</version>
        <configuration>
                <goal>dev</goal>
                <excludedGroups>uat</excludedGroups>
        </configuration>
</plugin>
```

Right click on the Project --> Run As --> Maven test



Note: We can pass TestInfo (I) type parameter in the test method to know more the current test method and its executing environment like tag name, display name, test class name, test method name and etc.

TestBankLoanService.java

```
        @Test
        @DisplayName("Testing with timer")
        @Tag("dev")
        public void testcalcSimpleIntrestAmountWithTimer(TestInfo info) {
        System.out.println("TestBankLoanService.testcalcSimpleIntrestAmou
ntWithTimer()");
                System.out.println(info.getClass()+" "+info.getTags()+"
"+info.getDisplayName()+" "+info.getTestMethod().get().getName()+"
"+info.getTestClass().get().getClass());
                assertTimeout(Duration.ofMillis(20000), ()->{
                        service.calcSimpleIntrestAmount(100000, 2, 12);
                });
        }
```

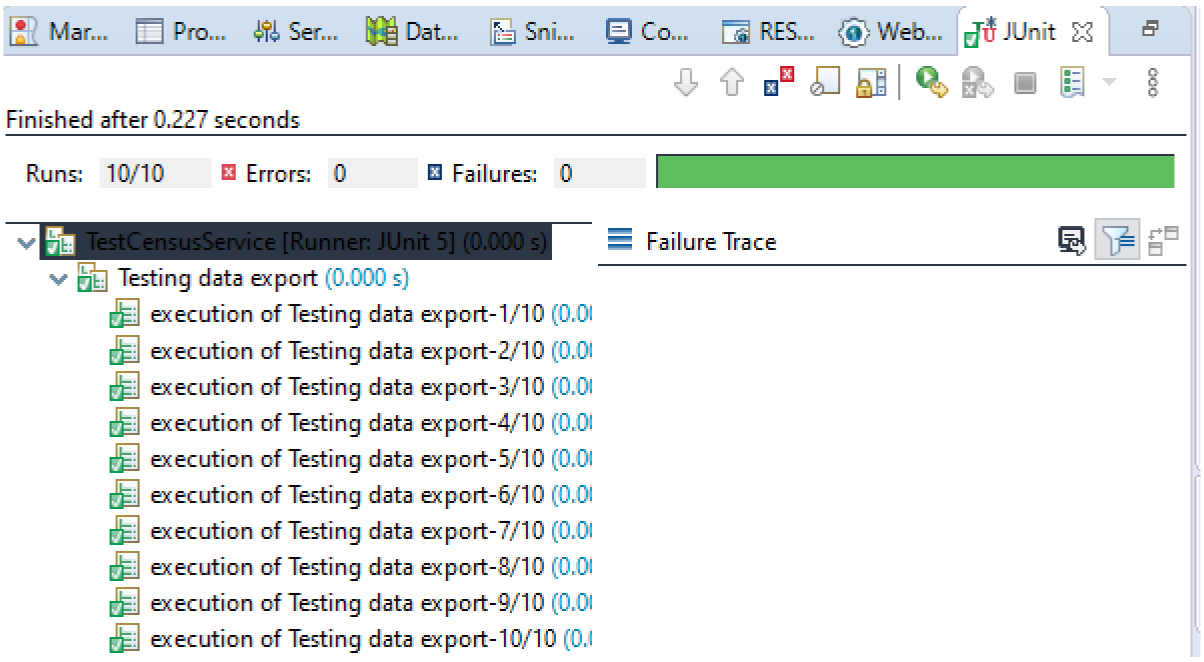
@RepeatedTest: Allows to execute test method repeatedly for multiple times having control on count and name. It is very useful batch processing/ updating related tests.

CensusService.java

```
package com.nt.service;

public class CensusService {

	public String exportData() {
		//logics.....
		return "data exported";
	}

}
```

TestCensusService.java

```
package com.nt.test;

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

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.RepeatedTest;

import com.nt.service.CensusService;

public class TestCensusService {

	@RepeatedTest(value = 10, name="execution of {displayName}-
{currentRepetition}/{totalRepetitions}")
	@DisplayName("Testing data export")
	public void testexportData() {
		System.out.println("TestCensusService.testexportData()");
		CensusService service = new CensusService();
		assertEquals("data exported", service.exportData());
	}

}
```



```
Finished after 0.227 seconds

Runs: 10/10    Errors: 0    Failures: 0

TestCensusService [Runner: JUnit 5] (0.000 s)
  Testing data export (0.000 s)
    execution of Testing data export-1/10 (0.0
    execution of Testing data export-2/10 (0.0
    execution of Testing data export-3/10 (0.0
    execution of Testing data export-4/10 (0.0
    execution of Testing data export-5/10 (0.0
    execution of Testing data export-6/10 (0.0
    execution of Testing data export-7/10 (0.0
    execution of Testing data export-8/10 (0.0
    execution of Testing data export-9/10 (0.0
    execution of Testing data export-10/10 (0.
```

@ParameterizedTest: To execute one test method for multiple times with different inputs/ params we can we @ParameterizedTest we can supply inputs by using @ValueSource and other annotations.

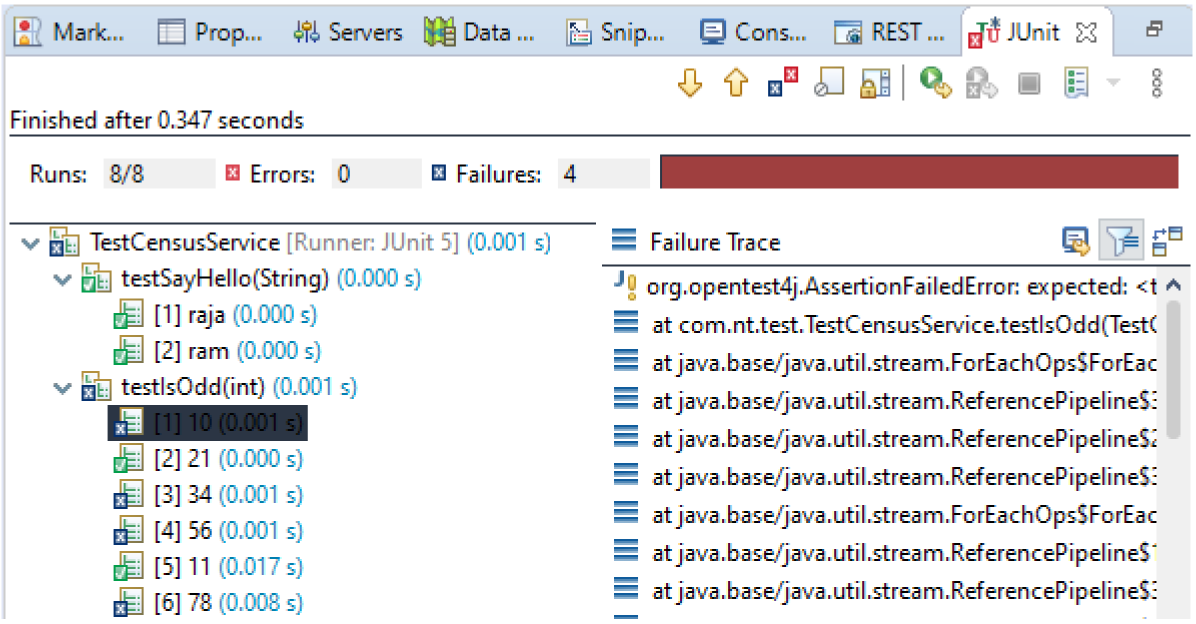
CensusService.java

```
	public boolean isOdd(int no) {
		if (no%2==0)
			return false;
		else
			return true;
	}

	public String sayHello(String user) {
		return "Hello: "+user;
	}
```

TestCensusService.java

```
	@ParameterizedTest
	@ValueSource(ints = {10, 21, 34, 56, 11, 78})
	public void testIsOdd(int n) {
		System.out.println("TestCensusService.testIsOdd()");
		CensusService service = new CensusService();
		assertTrue(service.isOdd(n));
	}
```

```
        @ValueSource(strings =  {"raja", "ram"})
        public void testSayHello(String user) {
                System.out.println("TestCensusService.testSayHello()");
                CensusService service = new CensusService();
                assertEquals("Hello: "+user, service.sayHello(user));
        }
```



```
Finished after 0.347 seconds

Runs:  8/8     Errors:  0     Failures:  4

TestCensusService [Runner: JUnit 5] (0.001 s)
  testSayHello(String) (0.000 s)
    [1] raja (0.000 s)
    [2] ram (0.000 s)
  testIsOdd(int) (0.001 s)
    [1] 10 (0.001 s)
    [2] 21 (0.000 s)
    [3] 34 (0.001 s)
    [4] 56 (0.001 s)
    [5] 11 (0.017 s)
    [6] 78 (0.008 s)

Failure Trace
org.opentest4j.AssertionFailedError: expected: <t
at com.nt.test.TestCensusService.testIsOdd(Test(
at java.base/java.util.stream.ForEachOps$ForEac
at java.base/java.util.stream.ReferencePipeline$3
at java.base/java.util.stream.ReferencePipeline$2
at java.base/java.util.stream.ReferencePipeline$3
at java.base/java.util.stream.ForEachOps$ForEac
at java.base/java.util.stream.ReferencePipeline$1
at java.base/java.util.stream.ReferencePipeline$3
```

assertTrue()/ assertFalse(): Given to checks whether the target business method returning true/ false.
assertNull() /assertNotNull(): Checks whether given object is null (assertNull) or not null (assertNotNull).
assertSame(): Checks whether given two reference variables are pointing to same object or not
Printer.java

```
package com.nt.service;

public class Printer {
        private static Printer INSTANCE = new Printer();

        private Printer() {
        }
        public static Printer getInstance() {
                return INSTANCE;
        }


}
```

TestPrinter.java

```java
package com.nt.test;

import static org.junit.jupiter.api.Assertions.assertSame;
import static org.junit.jupiter.api.Assertions.fail;

import org.junit.jupiter.api.Test;

import com.nt.service.Printer;

public class TestPrinter {

	@Test
	public void testSingletone() {
		Printer p1 = Printer.getInstance();
		Printer p2 = Printer.getInstance();
		/*assertNotNull(p1);
		assertNotNull(p2);*/
		if (p1==null||p2==null)
			fail("p1, p2 references must not be null");
		assertSame(p1, p2);
	}

}
```

Q. What is the difference b/w assertEquals() and assertSame()?
Ans. assertEquals() checks content of given two values (like equals() method). assertSame() checks whether given two references are pointing to same object or not (like ==).

Note: If want to write failure message by writing manual checking then use fail(-) method.

-- The END --