1. Lifecycle Methods

**Step 1.1:** Writing code for @BeforeAll annotation, @BeforeEach annotation, @AfterAll annotation, and @AfterEach annotation:

Below is a sample program that includes all the annotations mentioned above. These annotations play a very specific role in the test execution order.

* @BeforeAll is used to signal that the annotated method must be executed before all the tests in the current test class.
* @BeforeEach is used to signal that the annotated method must be executed before each @Test method in the current class.
* @AfterAll is used to signal that the annotated method must be executed after all the tests in the current test class.
* @AfterEach is used to signal that the annotated method must be executed after each @Test method in the current class.

**package com.simplilearn.unittest;**

**import org.junit.jupiter.api.AfterAll;**

**import org.junit.jupiter.api.AfterEach;**

**import org.junit.jupiter.api.BeforeAll;**

**import org.junit.jupiter.api.BeforeEach;**

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

**import org.junit.jupiter.api.Test;**

**@DisplayName("The standard annotations for Junit")**

**public class JunitSdAnnotations {**

**// Junit Fixture**

**@BeforeEach**

**public void setUp() {**

**System.out.println("--- Before each is executed. ---");**

**}**

**@AfterEach**

**public void cleanUp() {**

**System.out.println("--- After each is executed. ---");**

**}**

**@BeforeAll**

**public static void setUpAll() {**

**System.out.println("--- Before All test is executed. ---");**

**}**

**@AfterAll**

**public static void cleanUpAll() {**

**System.out.println("--- After All test is executed. ---");**

**}**

**@Test**

**@DisplayName("Test One")**

**public void testOne() {**

**System.out.println("--- Test One is executed. ---");**

**}**

**@Test**

**@DisplayName("Test Two")**

**public void testTwo() {**

**System.out.println("--- Test Two is executed. ---");**

**}**

**@Test**

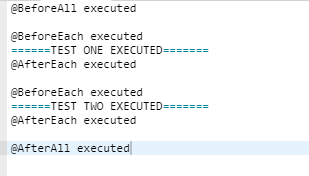
**@DisplayName("Test Three")**

**public void testThree() {**

**System.out.println("--- Test Three is executed. ---");**

**}**

**}**



2. Assertions

**Steps 2.1**: Explaining Assertion Methods:

**Boolean:** **If you want to test the boolean conditions (true or false), you can use the following assert methods:**

assertTrue(condition)

assertFalse(condition)

Here, the condition is a boolean value.

### Identical: If you want to check the initial value of an object/variable, you have the following methods:

assertNull(object)

assertNotNull(object)

Here, the object is a[Java](https://www.guru99.com/java-tutorial.html)object, for e.g**.** assertNull(actual);

### Null object: If you want to check whether the objects are identical (i.e. comparing two references to the same java object) or different, follow the below methods:

assertSame(expected, actual), It will return true if expected == actual

assertNotSame(expected, actual)

### Assert Equals: If you want to test the equality of two objects, you have the following methods:

assertEquals(expected, actual)

It will return true if:expected.equals( actual )returns true.

### Assert Array Equals :

assertArrayEquals(expected, actual)

The above method must be used if the arrays have the same length for each valid value for **i** as shown below:

assertEquals(expected[i],actual[i])

assertArrayEquals(expected[i],actual[i])

### Fail Message:

If you want to throw any assertion error, you have fail() that always results in a fail verdict.

Fail(message);

You can have the assertion method with an additional stringparameter as the first parameter. This string will be appended in the failure message if the assertion fails. E.g**.**fail( message )can be written as:

assertEquals (message, expected, actual)

## JUnit assertEquals

 assertEquals(a , b)which relies on theequals()method of the Object class.

* If a and b are primitives such as byte, int, Boolean, etc. then the following will be done for assertEquals (a, b):

a and b will be converted to their equivalent wrapper object type (Byte, Integer**,**Boolean, etc.), and then a. equals( b ) will be evaluated.

For Example: Consider that the below-mentioned strings have the same values, let's test it using assertTrue.

String obj1="Junit";

String obj2="Junit";

assertEquals (obj1 , obj2);

The above assert statement will return true as obj1.equals(obj2) returns true.

## Floating point assertions

When you want to compare the floating-point types (e.g. **double**or**float**), you need an additional required parameter **delta** to avoid problems with round-off errors while doing floating point comparisons.

The assertion evaluates as given below:

* + Math .abs( expected – actual ) <= delta

For example:

assertEquals( aDoubleValue, anotherDoubleValue, 0.001 )

**Steps 2.2:** Writing code for Assertions

package com.simplilearn.unittest;

import org.junit.jupiter.api.Test;

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

import org.junit.jupiter.api.DisplayName;

public class AssertionTest {

String userName = "Sasidhar";

int age = 30;

String email = null;

Object actual = new Object();

Object referance = actual;

int[] aNumbers = new int[] { 10, 20, 30 };

int[] eNumbers = new int[] { 10, 20, 30 };

@Test

@DisplayName("Should match expected and actual")

public void test1() {

assertEquals(userName, "Sasidhar");

assertEquals(age, 30);

}

@Test

@DisplayName("Should be true")

void shouldBeTrue() {

assertTrue(true);

assertTrue(age > 20);

}

@Test

@DisplayName("Should be false")

void shouldBeFalse() {

assertFalse(false);

assertFalse(age < 20);

}

@Test

@DisplayName("Should be null")

void shouldBeNull() {

assertNull(email);

}

@Test

@DisplayName("Should be not null")

void shouldBeNotNull() {

assertNotNull(userName);

}

@Test

@DisplayName("Should refer to the same object")

void shouldReferToSameObject() {

assertSame(referance, actual);

}

@Test

@DisplayName("Should contain the same integers array element")

void shouldContainSameIntegers() {

assertArrayEquals(aNumbers, eNumbers);

}

@Test

@DisplayName("Should assert mulple Assertions")

public void lambdaExpressionTest3() {

int[] numbers = { 0, 1, 20, 3, 40 };

assertAll("Numbers", () -> assertEquals(numbers[0], 0));

assertAll("Numbers", () -> assertEquals(numbers[2], 20));

assertAll("Numbers", () -> assertEquals(numbers[2], 20), () -> assertEquals(numbers[4], 40));

assertAll("Numbers", () -> assertEquals(numbers[2], 20), () -> assertEquals(numbers[4], 40),

() -> assertEquals(numbers[3], 3));

}

@Test

@DisplayName("Should throw the correct exception")

void shouldThrowCorrectException() {

assertThrows(NullPointerException.class, () -> {

throw new NullPointerException();

});

}

@Test

@DisplayName("Should not throw an exception")

void shouldNotThrowException() {

assertDoesNotThrow(()-> {

// any logic -> should not throw exception

});

}

}

* Next, create a java class file Calculator.java named  in

package com.simplilearn.unittest;

public interface Calculator {

// abstract method

int add(int num1 , int num2);

// default method

default int multiply(int num1 , int num2) {

return num1 \* num2;

}

}

Compile the Test Case and Test Runner classes



3. Disabling Tests

**Step 3.1:** Writing code to demonstrate @ignore annotation

Sometimes our code is not completely ready while running a test case. As a result, the test case fails. The **@Ignore** annotation helps in this scenario.

* A test method annotated with @Ignore will not be executed.
* If a test class is annotated with @Ignore, then none of its test methods will be executed.

**Junit Test Example - Ignore**

We can use @Ignore annotation to ignore a test or a group of tests.

Let's understand it using simple examples and in the scenarios given below:

1. Creating a simple test class without ignoring a test
2. Ignore a test method using @Ignore annotation
3. Ignore a test method using @Ignore annotation with proper reason
4. Ignore all test methods using @Ignore annotation

## Creating a simple test class with ignoring a test method using @Ignore annotation

Let's write a program to disable a test. For this, you need to use @Ignore in the method you want to skip.

Let's do it for testJUnitMessage() of JunitTestExample.java

**DisableTest.Java**

package com.simplilearn.unittest;

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Disabled;

import org.junit.jupiter.api.DisplayName;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.condition.DisabledOnJre;

import org.junit.jupiter.api.condition.DisabledOnOs;

import org.junit.jupiter.api.condition.JRE;

import org.junit.jupiter.api.condition.OS;

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

@DisplayName("Test Age calculator")

// @Disabled

public class DisableTest {

AgeCalculator ageCalculator;

@BeforeEach

public void setUp() {

ageCalculator = new AgeCalculator();

}

@AfterEach

public void cleanUp() {

if(ageCalculator!=null)

ageCalculator = null;

}

@Test

@Disabled

@DisplayName("Should return valid age for +ve year")

public void test1() {

int eResult = 33;

int aResult = ageCalculator.calculateAge(1990);

assertEquals(eResult, aResult);

// assertEquals(28, ageCalculator.calculateAge(1995));

}

@Test

@DisabledOnOs(value = OS.WINDOWS)

@DisplayName("Should return 0 age for -ve year")

public void test2() {

assertEquals(0, ageCalculator.calculateAge(-1995));

}

@Test

@DisabledOnJre(value = JRE.JAVA\_17)

@DisplayName("Should return 0 age for 0 year")

public void test3() {

assertEquals(0, ageCalculator.calculateAge(0));

}

@Test

@DisabledOnJre({JRE.JAVA\_17, JRE.JAVA\_8})

@DisplayName("Should return 0 age for +ve future year")

public void test4() {

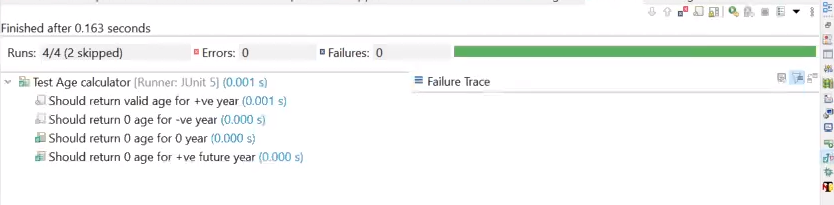
assertEquals(0, ageCalculator.calculateAge(2050));

}

}

**Output:**

Let's execute and verify the output of the above example.



4. Assumptions

**Steps 4.1:** Importing a Junit Assumption

* JUnit Assumptions class provides a useful collection of assumption methods. To import them in our test class, write the commands given below.

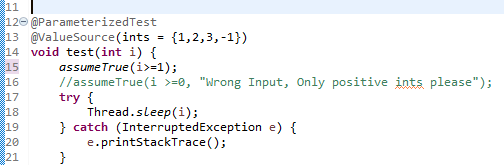
import static org.junit.jupiter.api.Assumptions.\*;

import static org.junit.Assume.assumeTrue;

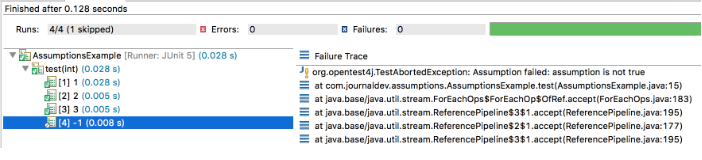
**Steps 4.2:** Writing a code to demonstrate the types of Assumptions

* assumeTrue()

We can use assumeTrue() to skip the test if the input number is negative. Below is the updated code:

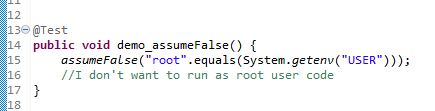


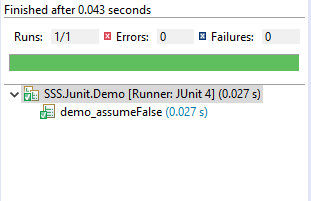
* JUnit will skip when the input number is negative. When we run the test, we will get the following output:



* assumeFalse()

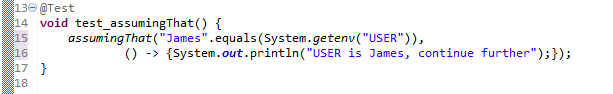
assumeFalse() validates the given assumption to false and if the assumption is false then the test proceeds, otherwise, the test execution is aborted. It works just opposite to assumeTrue().





* assumingThat()

This method executes the supplied Executable if the assumption is valid. If the assumption is invalid, this method does nothing. We can use this for logging or notifications when our assumptions are valid.



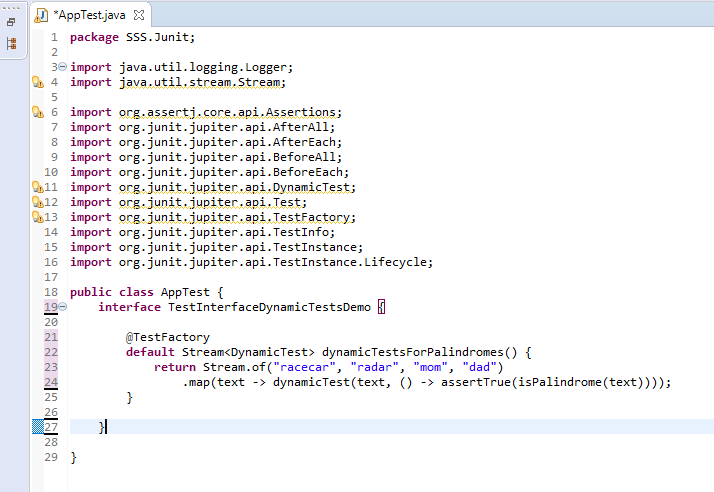
5.Test Interfaces and Default Methods

**Step 5.1:** Writing a code to demonstrate interfaces and default methods:

This section will guide you to write code with the annotations given below:

* Test
* RepeatedTest
* ParameterizedTest
* TestFactory
* TestTemplate
* BeforeEach
* AfterEach
* BeforeAll
* AfterAll

**Step 5.2:** Writing a code to demonstrate TestFactory

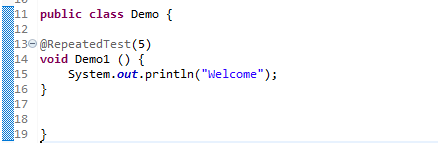


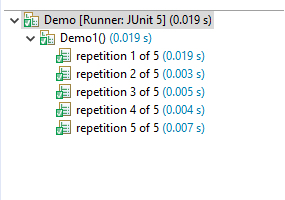
6. Repeating Tests

**Step 6.1:** Writing a code to demonstrate the execution of repeated tests

* + 1. **Repeated Tests Example**

With the below example, we will use @RepeatedTest annotation (introduced in JUnit 5). This is more convenient to write the Junit test that we want to repeat several times.



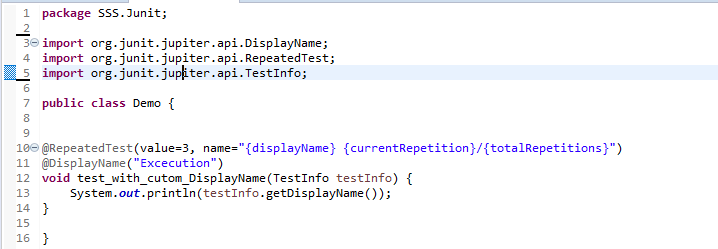


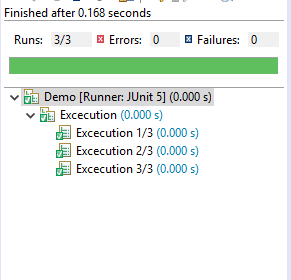


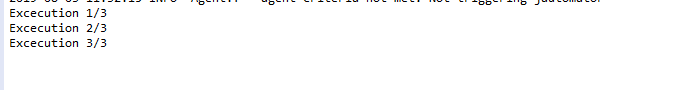
**6.1.2 @RepeatedTest DisplayName**

We will be using @DisplayName annotation to declare a custom display name for the annotated test class or test method.

**DisplayName value**

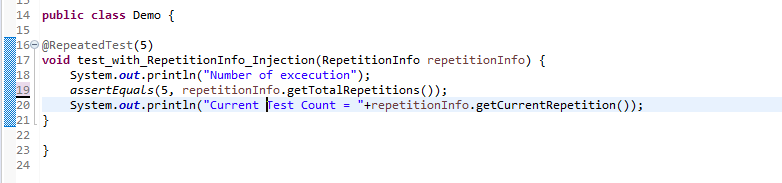




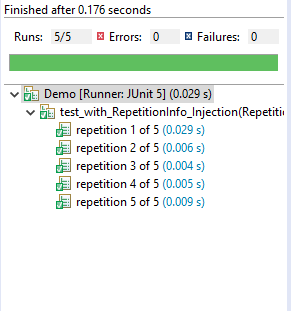


## 6.1.3 JUnit @RepetitionInfo

When we injected TestInfo into our test method, JUnit Jupiter provides @RepetitionInfo annotation that we can inject into our test method.







If we have many such methods, then we can move it to @BeforeEach or @AfterEach methods too.

@BeforeEach

**void** setUp(RepetitionInfo repetitionInfo, TestInfo testInfo) {

System.***out***.println("Method = "+testInfo.getTestMethod().get().getName()+", Execution Count = "+repetitionInfo.getCurrentRepetition());

}

The above mentioned @BeforeEach method will throw an error if all the test methods are not annotated with @RepeatedTest.

org.junit.jupiter.api.extension.ParameterResolutionException: No ParameterResolver registered for parameter

[org.junit.jupiter.api.RepetitionInfo arg0] in executable

[void com.journaldev.repeatedtests.RepeatedTestExample.setUp

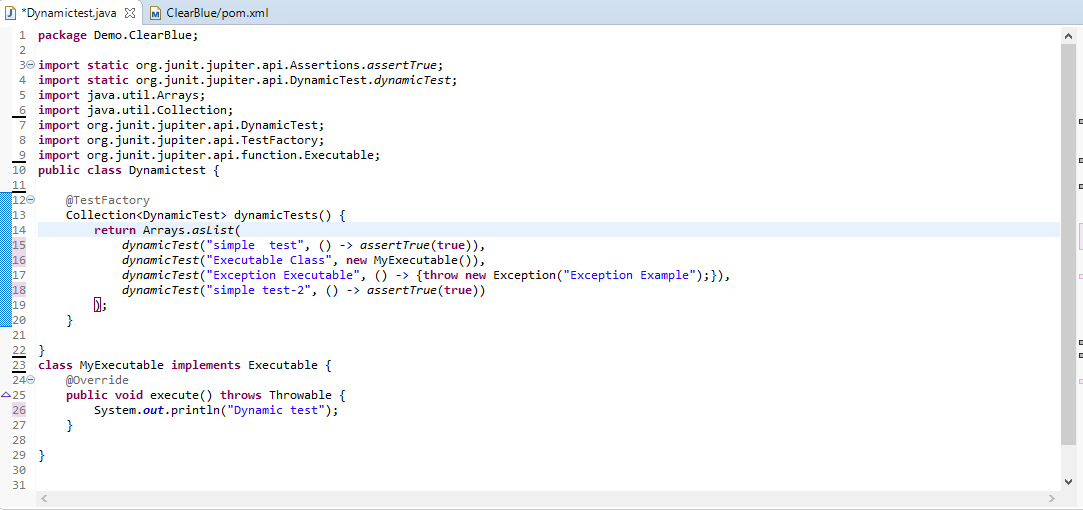
(org.junit.jupiter.api.RepetitionInfo,org.junit.jupiter.api.TestInfo)]

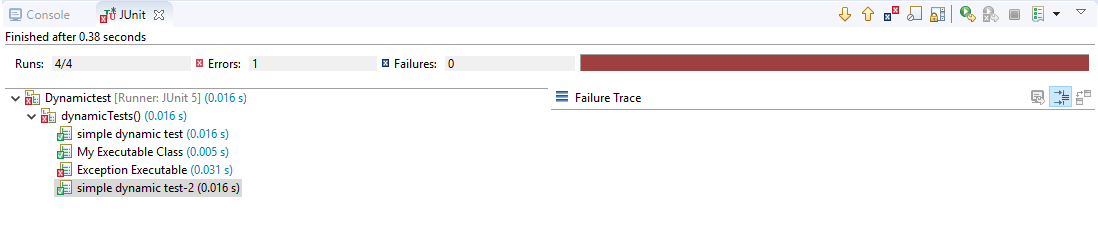
7. Dynamic Tests

**Step 7.1:** Writing a code to import Dynamic Test libraries

* import org.junit.jupiter.api.DynamicTest;
* import org.junit.jupiter.api.TestFactory;
* import org.junit.jupiter.api.function.Executable;

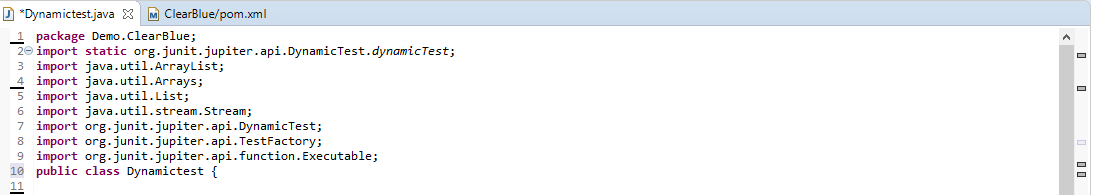
**Step 7.2:**  Writing a code to demonstrate JUnit @TestFactory



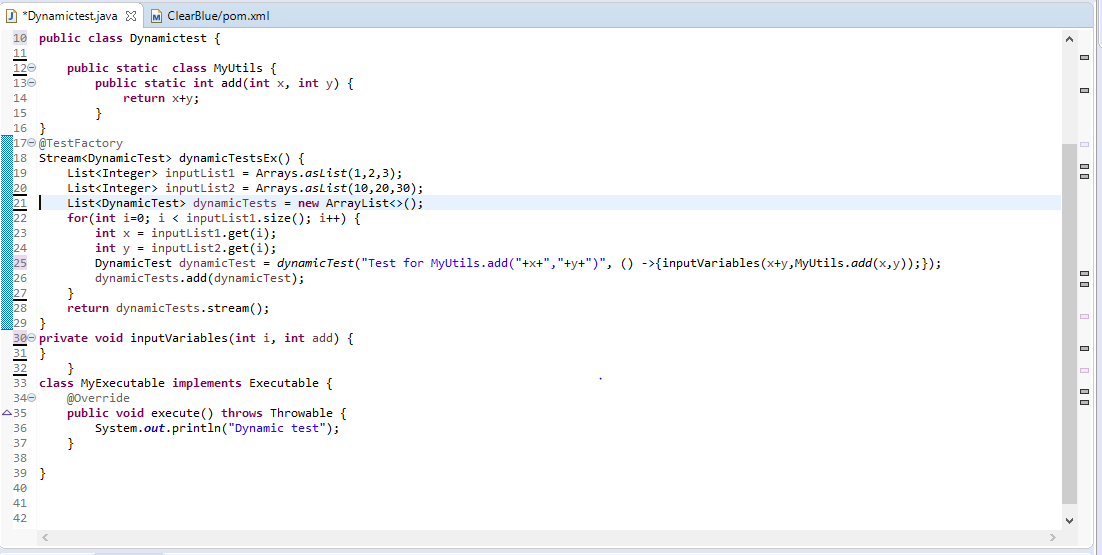


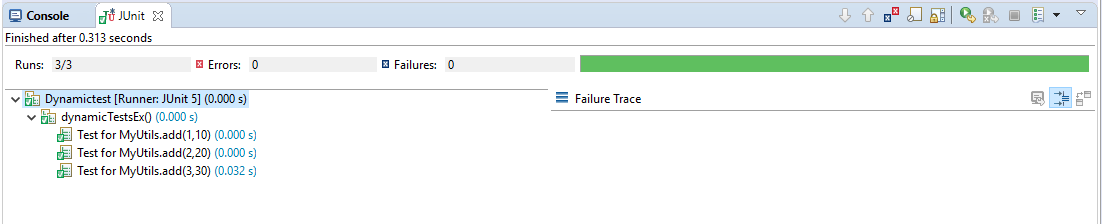
**Step 7.3:** Writing a code to demonstrate the execution of Dynamic Tests

* **Screenshot 1:**



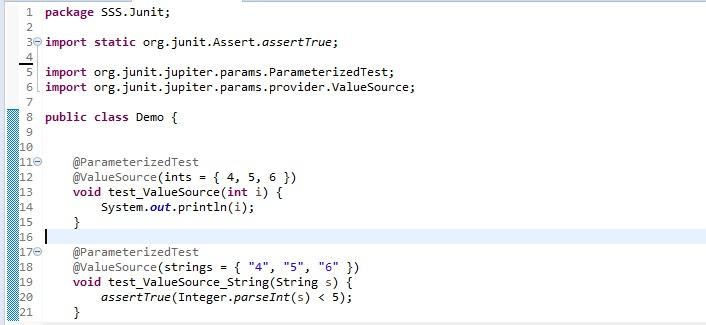
* **Screenshot 2:**



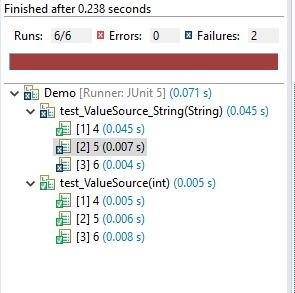


8. Parameterized Tests

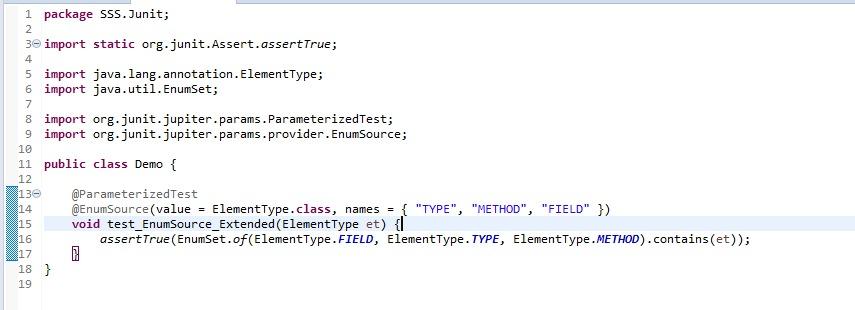
**Step 8.1:** Writing a code to demonstrate Parameterized Test with @ValueSource

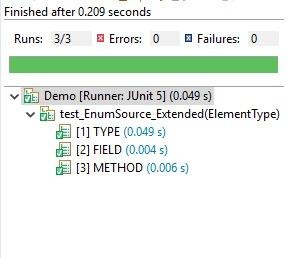




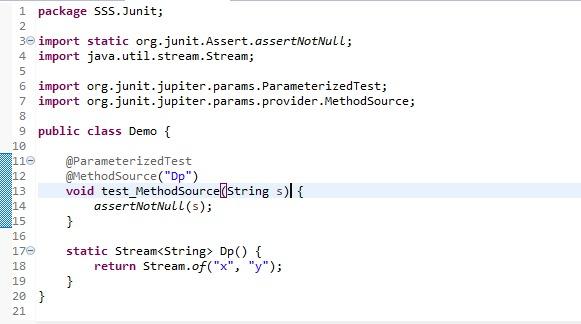


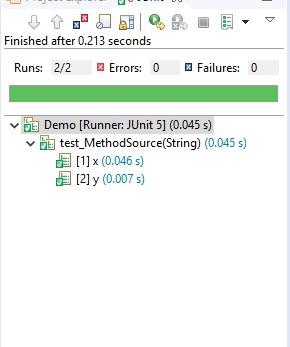
**Step 8.2:** Writing a code to demonstrate @ParameterizedTest with @EnumSource



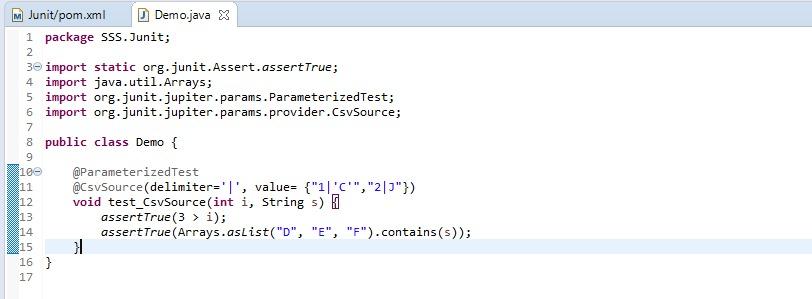


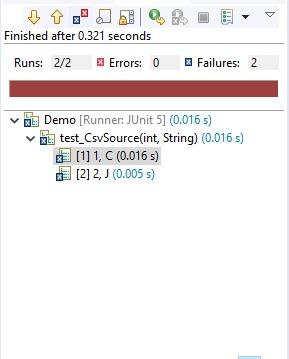
**Step 8.3:** Writing a code to demonstrate@ParameterizedTest with @MethodSource

****

****

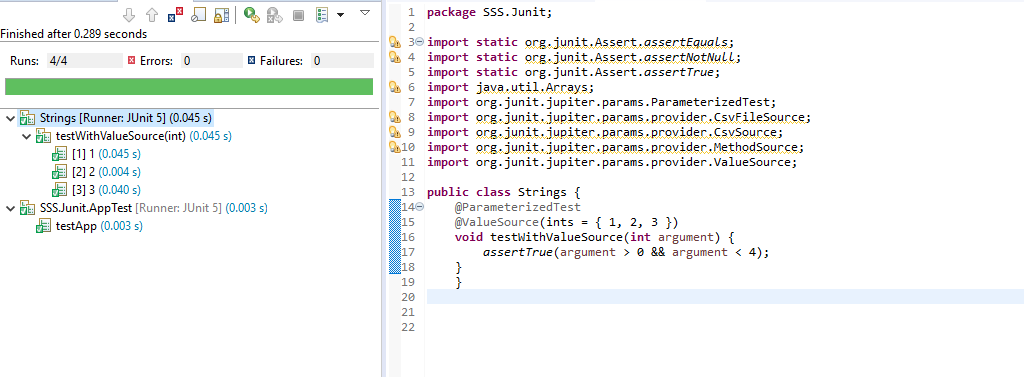
**Step 8.4:** Writing a code to demonstrate@ParameterizedTest with @CsvSource



****

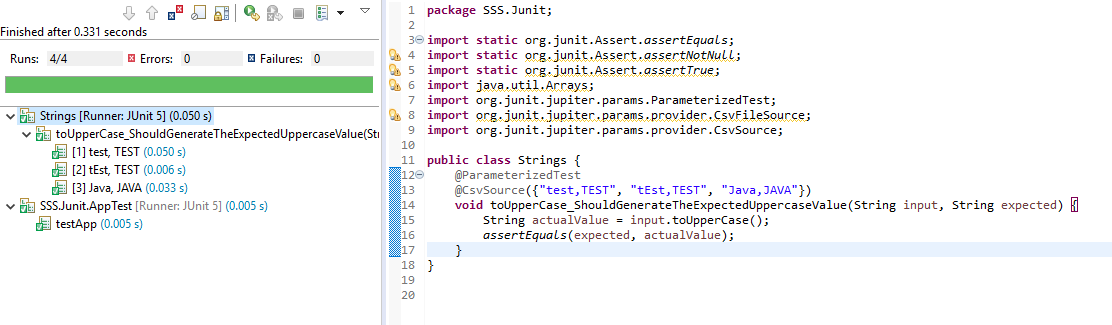
9. Argument Sources

**Step 9.1:**  Writing a code to demonstrate @ValueSource annotation

With the @ValueSource annotation, we can pass an array of literal values to the test method. 

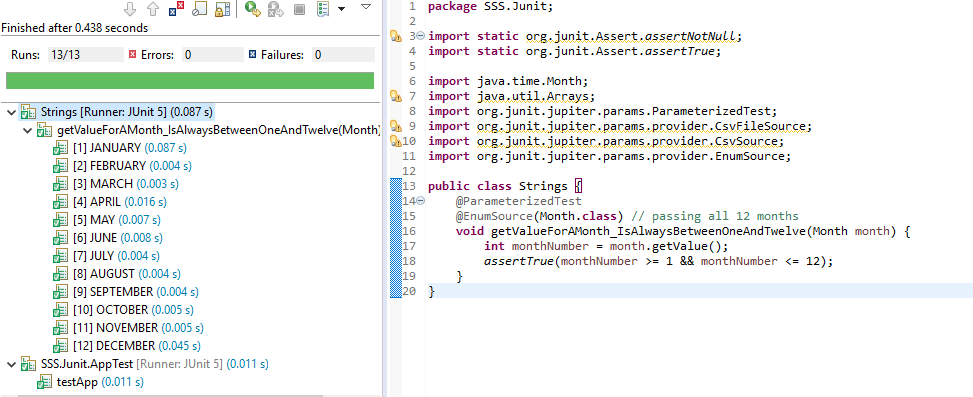
**Step 9.2:** Writing a code to demonstrate @CsvSource annotation

Suppose we’re going to make sure that the toUpperCase() method from String generates the expected uppercase value. @ValueSource won’t be enough.



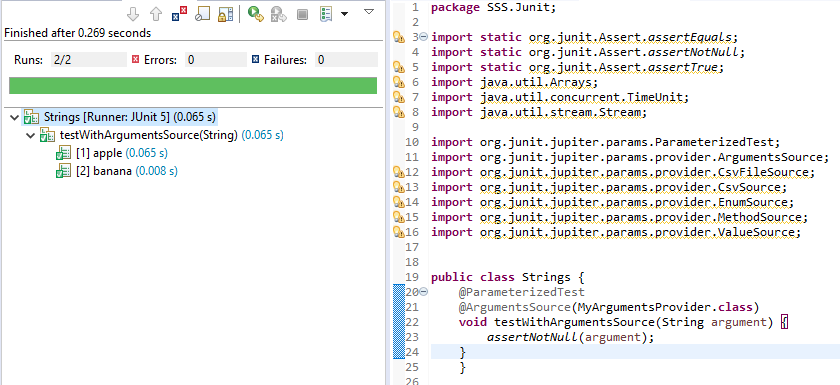
**Step 9.3:** Writing a code to demonstrate@EnumSource annotation

In order to run a test with different values from an enumeration, we can use the @EnumSource annotation.



**Step 9.4:** Writing a code to demonstrate@ArgumentsSource annotation

@ArgumentsSource can be used to specify a custom, reusable ArgumentsProvider. Note that an implementation of ArgumentsProvider must be declared as either a top-level class or as a static nested class.



10. Argument Conversion

**Step 10.1:** Writing a code to demonstrate Implicit Conversion

To support use cases like @CsvSource, JUnit Jupiter provides a number of built-in implicit type converters. The conversion process depends on the declared type of each method parameter.

Create a class with test with ToStringArgumentConverter.class

@Override

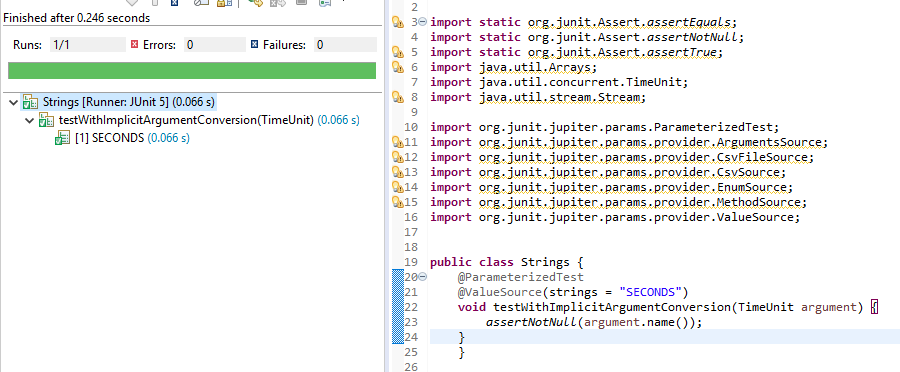
protected Object convert(Object source, Class<?> targetType) {

assertEquals(String.class, targetType, "Can only convert to String");

return String.valueOf(source);

}

}



**Step 10.2:** Writing a code to demonstrate Explicit Conversion

Instead of relying on implicit argument conversion, you may explicitly specify an ArgumentConverter to use for a certain parameter using the @ConvertWith annotation as shown in the following example. Note that an implementation of ArgumentConverter must be declared as either a top-level class or a static nested class.

Create a class” ToStringArgumentConverter.class” with below code

public class ToStringArgumentConverter extends SimpleArgumentConverter

{

@Override

protected Object convert(Object source, Class<?> targetType)

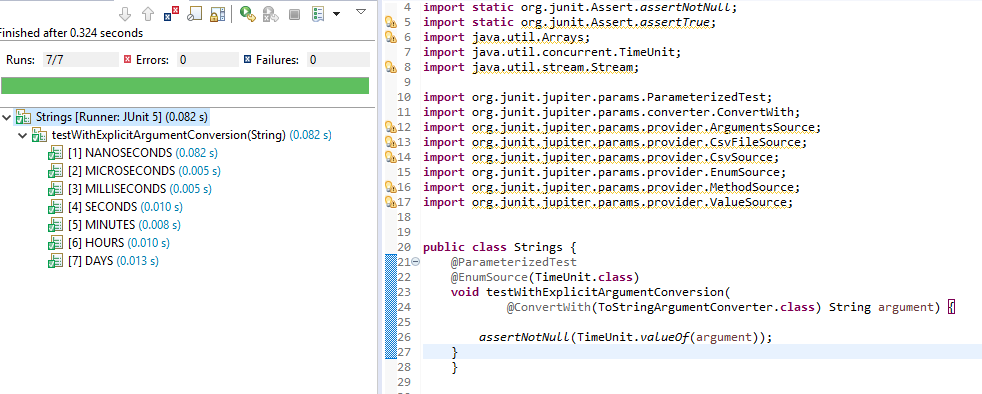
{

assertEquals(String.class, targetType, "Can only convert to String");

return String.valueOf(source);

}

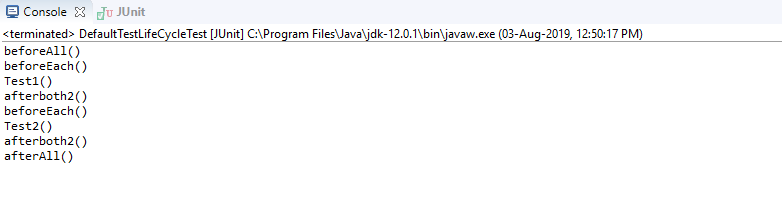
}



11. Extension Points

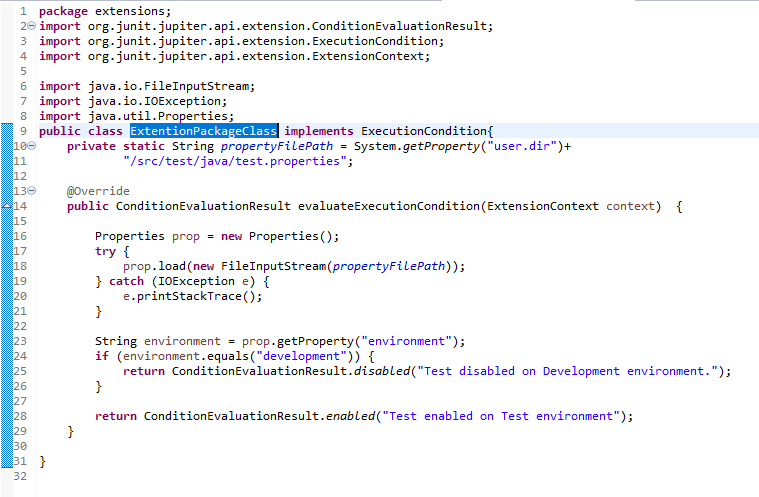
**Steps 11.1:** Writing a code to demonstrate Life Cycle Call Back

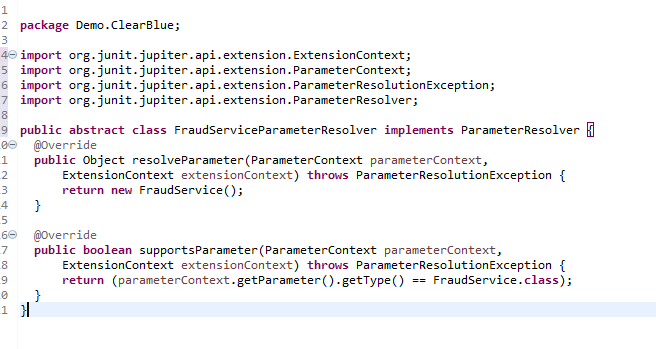


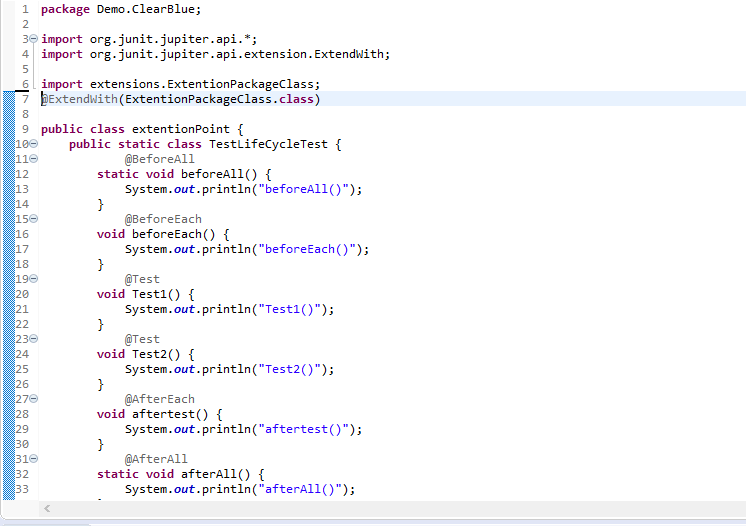


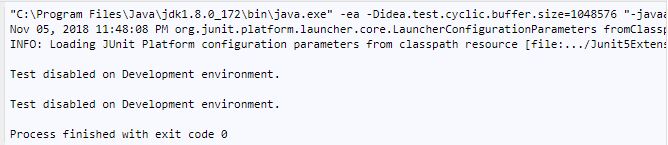
**Steps 11.2:** Writing code to demonstrate Conditional Test Execution

* Create the extension class and extend them in the base class.
* Create an extension as shown below:



* Now 



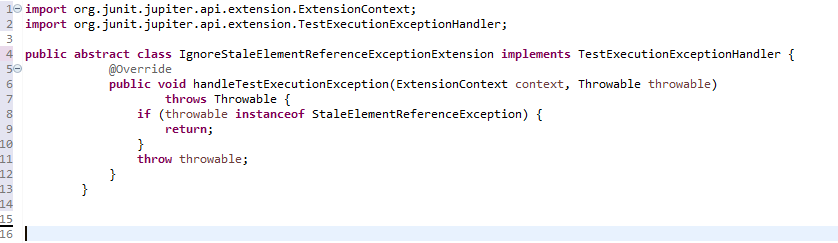


* We should register this with @ExtendWith(FraudServiceParameterResolver.class)

annotation on top of our test classes.

**Steps 11.3:** Writing a code to demonstrate Exception Handling Extension

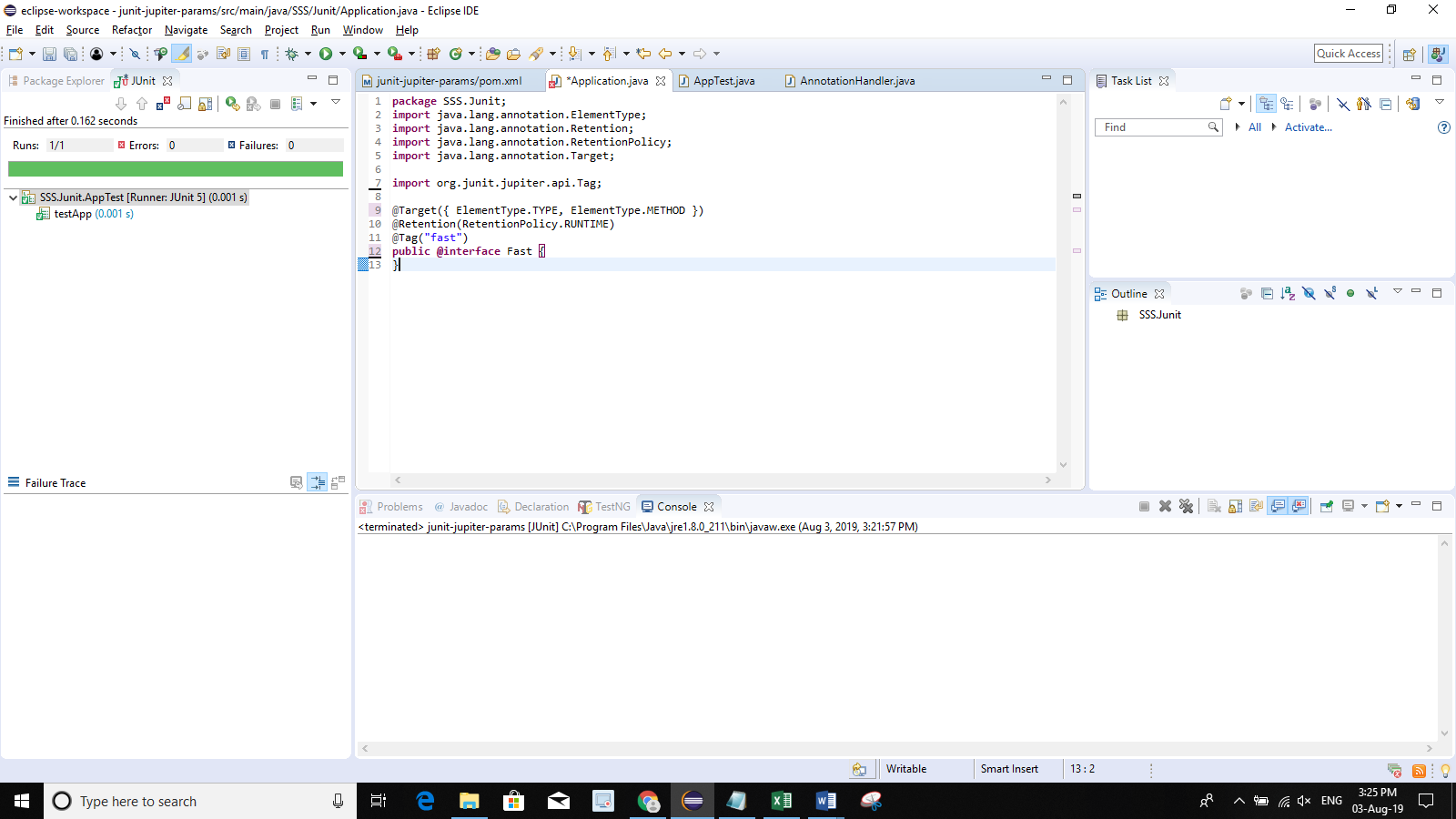
* Create an extension class as shown below:



12. Meta-Annotations

**Step 12.1:** Writing a code to demonstrate Meta-Annotations and Composed Annotations

* JUnit Jupiter annotations can be used as meta-annotations. That means that you can define your own composed annotation that will automatically inherit the semantics of its meta-annotations.
* Instead of copying and pasting @Tag("fast") throughout your code base (see Tagging and Filtering), you can create a custom composed annotation named @Fast as follows. @Fast can then be used as a drop-in replacement for @Tag("fast").



* The following @Test method demonstrates the usage of the @Fast annotation.

@Fast

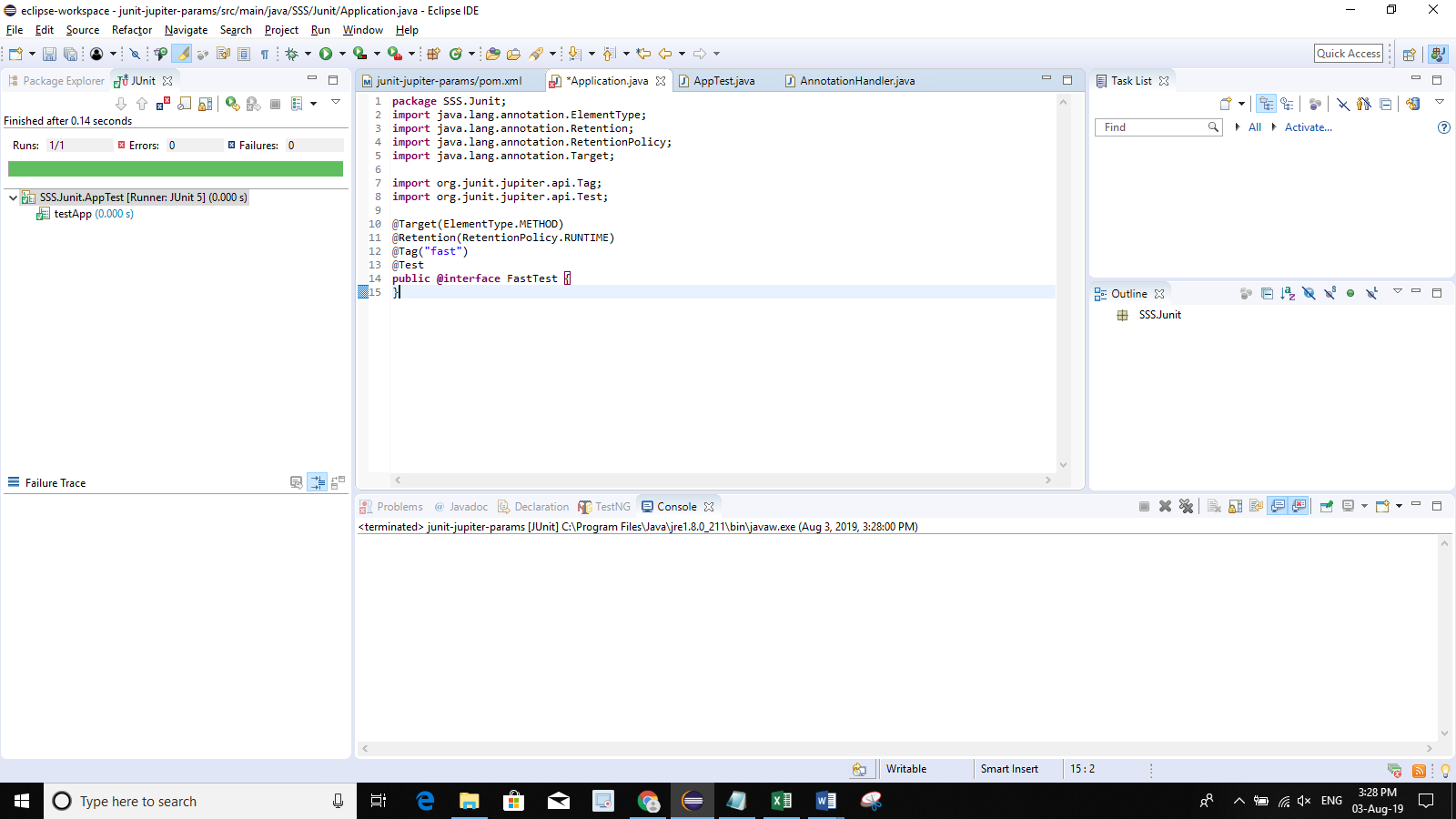
@Test

void myFastTest() {

// ...

}

* + You can even take that one step further by introducing a custom @FastTest annotation that can be used as a drop-in replacement for @Tag("fast") and @Test.



* + JUnit automatically recognizes the following as a @Test method that is tagged with "fast."

@FastTest

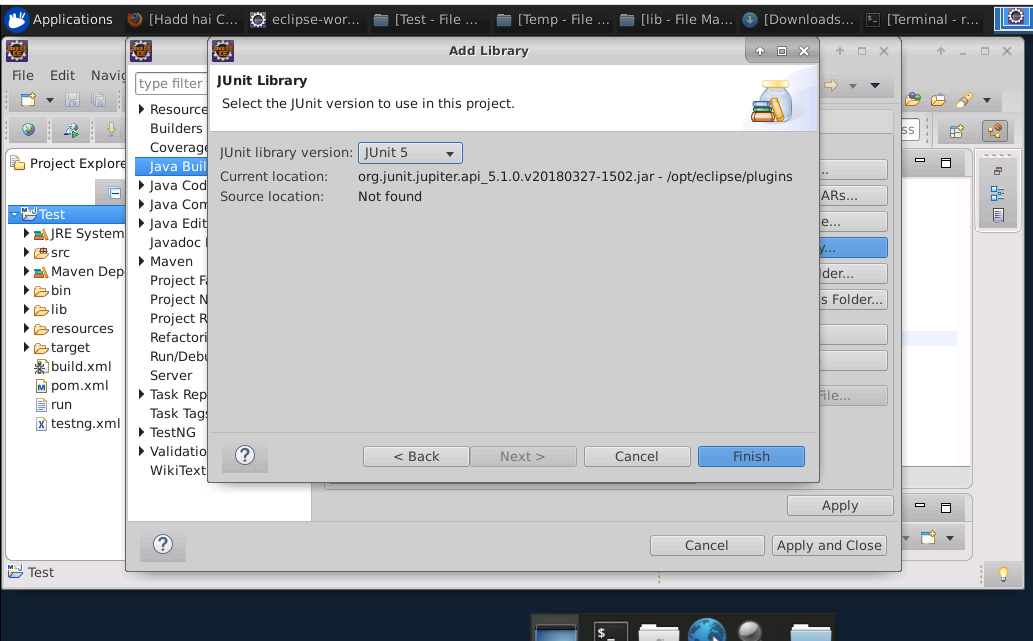
void myFastTest() {

// ...

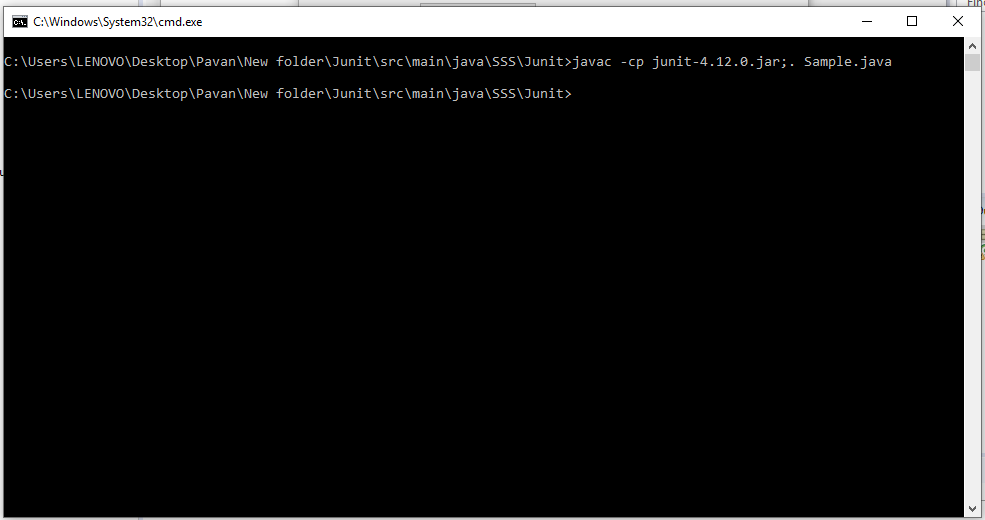
}

13. Running Test from the Console

**Step 13.1**: Configuring the Junit-4 version from the eclipse build path to add libraries



1. Make sure there is no error in the script.
2. Open the command prompt and set the path folder where you saved the JavaScript file.
3. To compile the Java file, write the command given below:



1. Run the Junit java file in the command prompt.

Type the command: *javac -cp junit-4.12.jar;. UserDAOTest.java*

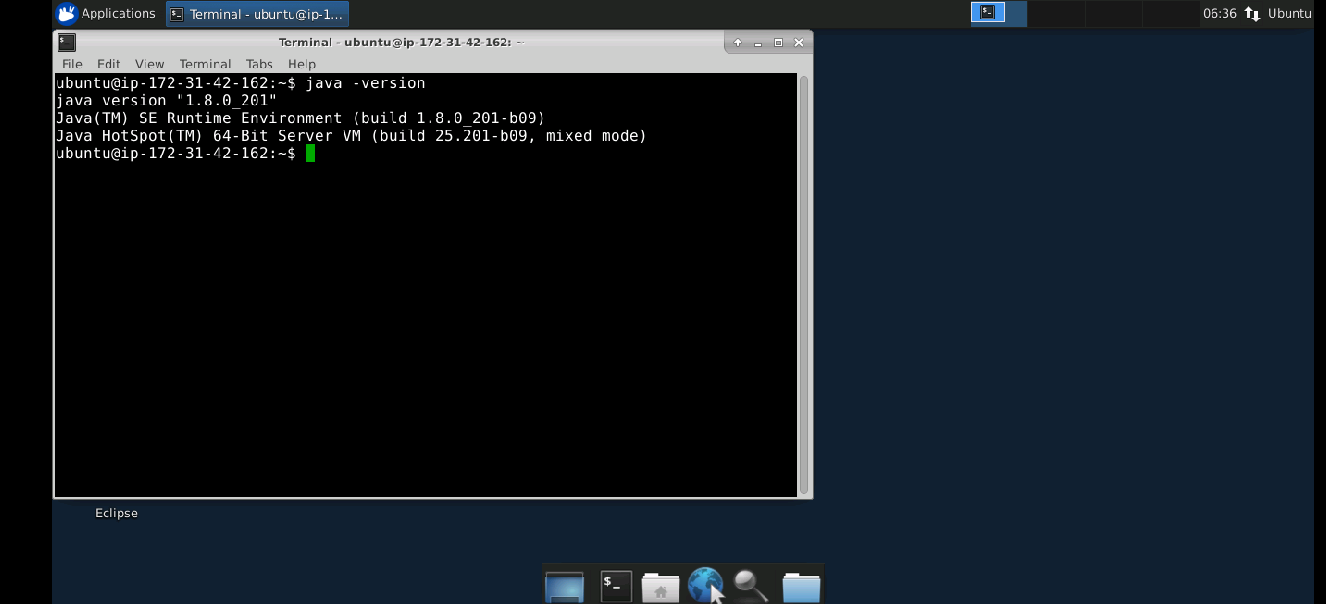
Type the command: *java -cp junit-4.12.jar;hamcrest-core-1.3.jar;. org.junit.runner.JUnitCore UserDAOTest JUnit version 4.12*

14. Running Tests with Gradle

**Step 14.1:** Installing Gradle from the official website

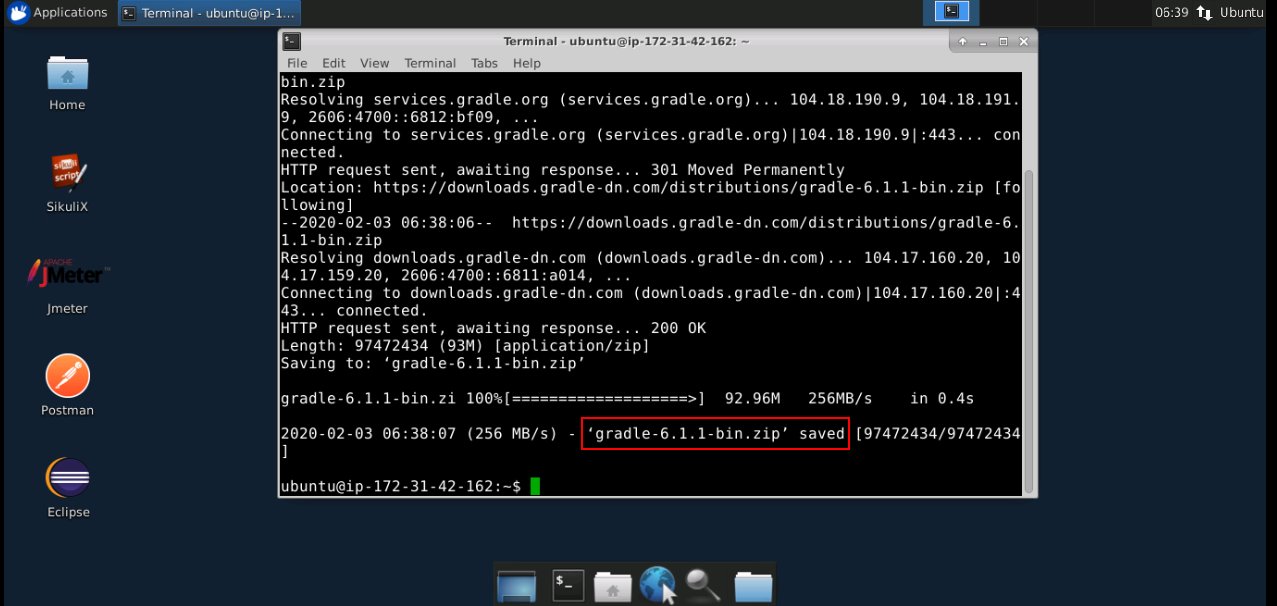
* Gradle requires Java development kit in order to work. Java has already been installed in your practice lab. To verify the installation, run the following command:

*java -version*



* Download Gradle using the following command:

*wget* [*https://services.gradle.org/distributions/gradle-6.1.1-bin.zip*](https://services.gradle.org/distributions/gradle-6.1.1-bin.zip)



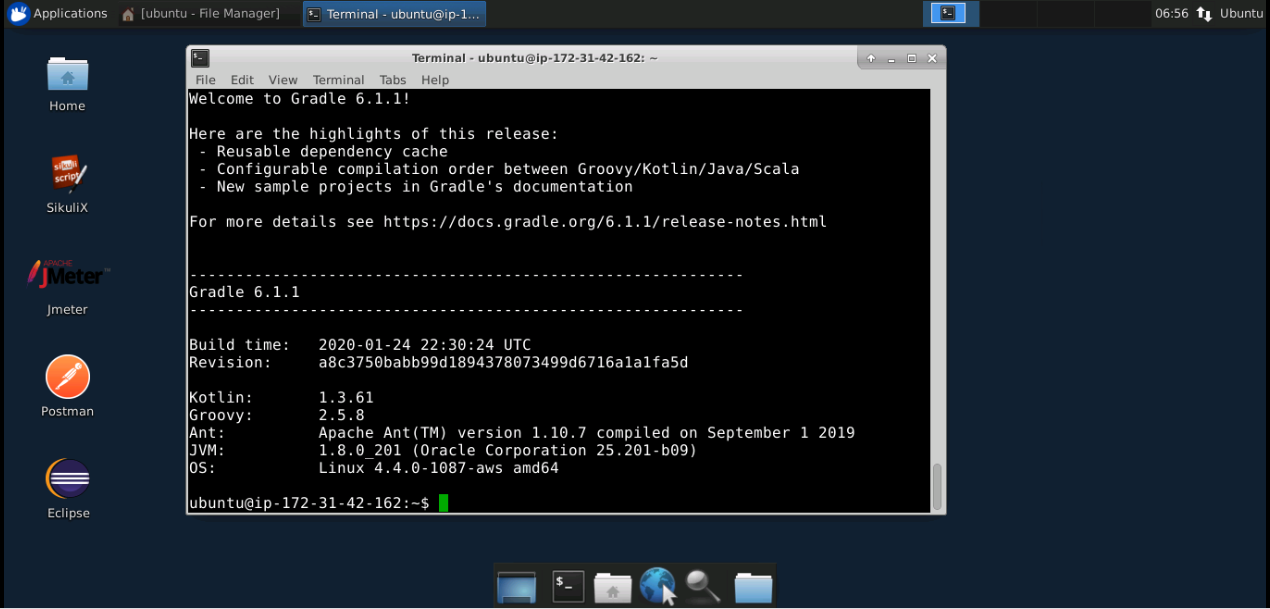
* After downloading, create a directory for Gradle installation and unzip the downloaded file in the new directory.
* Configure the PATH environment variable using the following command:

*export PATH=$PATH:gradle/gradle-6.1.1/bin*

**Step 14.2:** Checking the Installed version

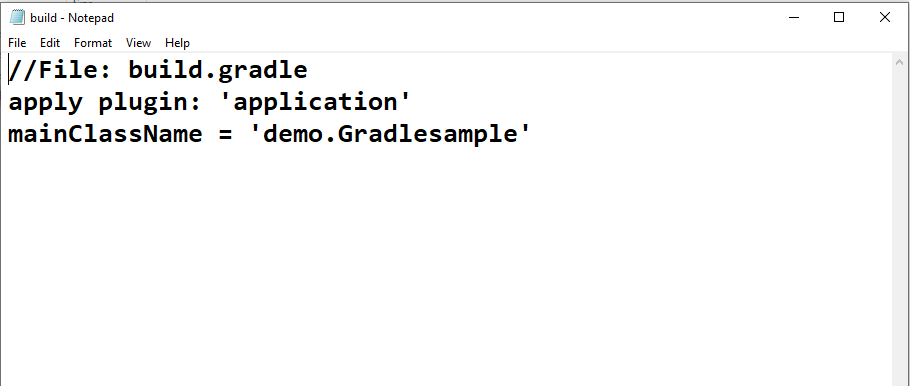
* Verify the installation using the following command:

*gradle -v*



**Step 14.3:** Scripting the Gradle built program

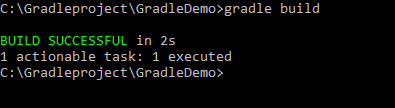
* Create a new file, type the script given below, and save it as “build.gradle” outside the Gradle folder.



* Go to the command prompt and type the following:
  + gradle tasks



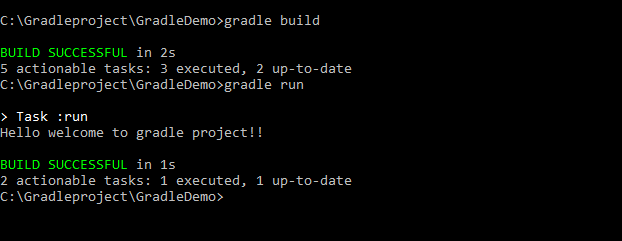
* + gradle build

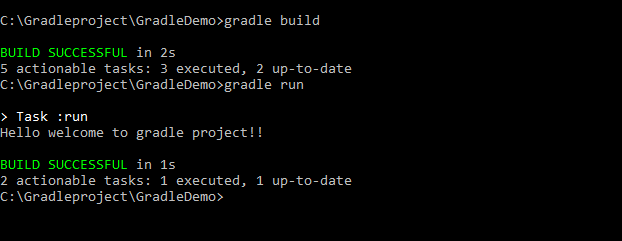


* That will automatically generate the build folder file.

**Step 14.4:** Checking Final Gradle Output

* Run the Java program using the gradle command line prompt.
* Type gradle build
* Output: “Hello welcome to gradle project !!”





15. Running Tests with Maven

**Step 15.1:** Writing a code to demonstrate Maven-surefire-plugin

Below is the Surefile Plugin. This Plugin (code below) needs to be added to the Page Object Model “POM” file in xml format.

* Add the below Surefile to POM

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.22.0</version>

</plugin>

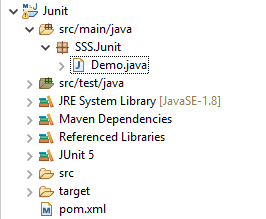
</plugins>

</build>

**Step 15.2:** Explaining the structure of a Maven project

* Below is a simple Java project which will guide us on how to run the unit test classes in the Maven project. This is the directory structure for creating a Maven project which essentially includes the Junit test cases, JRE system library, maven dependencies, Junit 5 library, and the POM XML file.

**Directory Structure**



**Step 15.3:** Demonstrating Maven + JUnit5 examples

* Below is the sample POM file with all the dependencies and other annotations:

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<artifactId>junit-jupiter-params</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>Junit</name>

<url>http://maven.apache.org</url>

<properties>

<maven.compiler.source>1.9</maven.compiler.source>

<maven.compiler.target>1.9</maven.compiler.target>

</properties>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.22.0</version>

</plugin>

</plugins>

</build>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

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

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>5.2.0</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>5.2.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-params</artifactId>

<version>5.0.0</version>

<scope>test</scope>

</dependency>

</dependencies>

<groupId>org.junit.jupiter</groupId>

</project>

**Step 15.4:** Demonstrating Java Classes to run the Maven Script

* We have already created a complete Maven project structure above with Java source code. Now we will create different Java classes in the *./src/main/*java/packages/class(s). It also created an example test class in *./src/test/*. In the root folder, there is a *pom.xml* file.
  + - 1. **MagicBuilder.java**

MagicBuilder.java

**package** SSS.Junit;  
public class MagicBuilder {

public static int getLucky() {

return 7;

}

}

Textbuild.java

**package** SSS.Junit;  
  
public class Textbuild {

public static String getHelloWorld(){

return "hello world";

}

public static int getNumber10(){

return 10;

}

}

**15.4.2 Test class for MagicBuilder**

TestMagicBuilder.java

**package** SSS.Junit;  
  
import org.junit.jupiter.api.Test;

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

public class TestMagicBuilder {

@Test

public void testLucky() {

assertEquals(7, MagicBuilder.getLucky());

}

}

* + - 1. **Test class for Textbuild**

pom.xml

**package** SSS.Junit;  
  
import org.junit.jupiter.api.Test;

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

public class Msgbuild {

@Test

public void testHelloWorld() {

assertEquals("hello world", Textbuild.getHelloWorld());

}

@Test

public void testNumber10() {

assertEquals(10, Textbuild.getNumber10());

}

}

## Step 15.5: Executing Maven Test

* While we run the POM and class files with Maven, the below results will be generated at the console:
  + - 1. **Run all test classes**

$ mvn test

[INFO] -------------------------------------------------------

[INFO] T E S T S

[INFO] -------------------------------------------------------

[INFO] Running SSS.Junit.TestMagicBuilder

[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 s - in SSS.Junit.TestMagicBuilder

[INFO] Running SSS.Junit.Msgbuild

[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s - in SSS.Junit.Msgbuild

[INFO]

* + - 1. **Run a single test class Msgbuild**

Terminal

$ mvn -Dtest=Msgbuild test

[INFO] -------------------------------------------------------

[INFO] T E S T S

[INFO] -------------------------------------------------------

[INFO] Running SSS.Junit.Msgbuild

[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 s - in SSS.Junit.Msgbuild

[INFO]

* + - 1. **Run a single test method testHelloWorld() from the test class Msgbuild**

Terminal

$ mvn -Dtest=Msgbuild#testHelloWorld test

[INFO] -------------------------------------------------------

[INFO] T E S T S

[INFO] -------------------------------------------------------

[INFO] Running SSS.Junit.Msgbuild

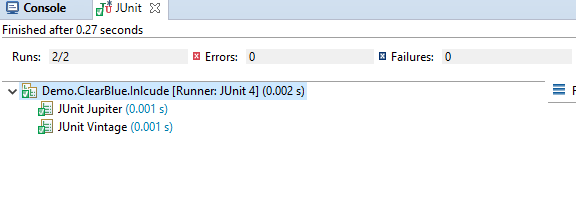
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 s - in SSS.Junit.Msgbuild

[INFO]

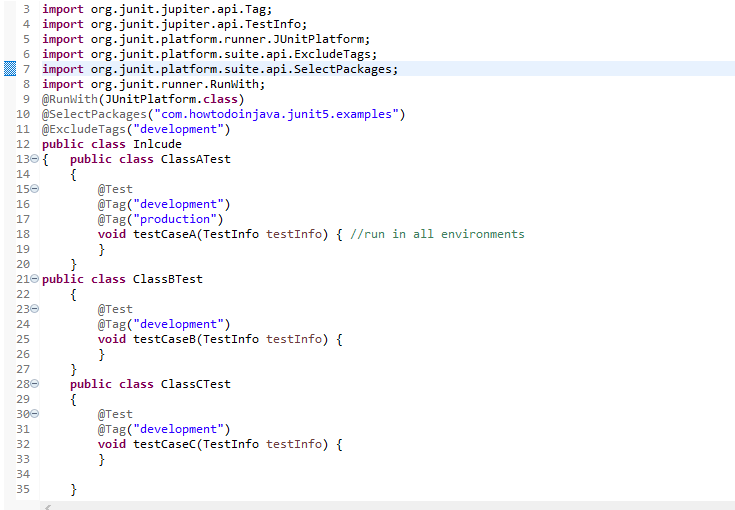
16. Include/Exclude Tests with Tags

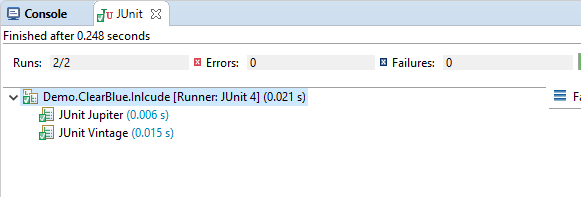
**Step 16.1:** Writing a code to demonstrate Include Tag





**Step 16.2:** Writing a code to demonstrate Exclude Tag





17. Code Coverage

**Step 17.1:** Installing of EclEmma

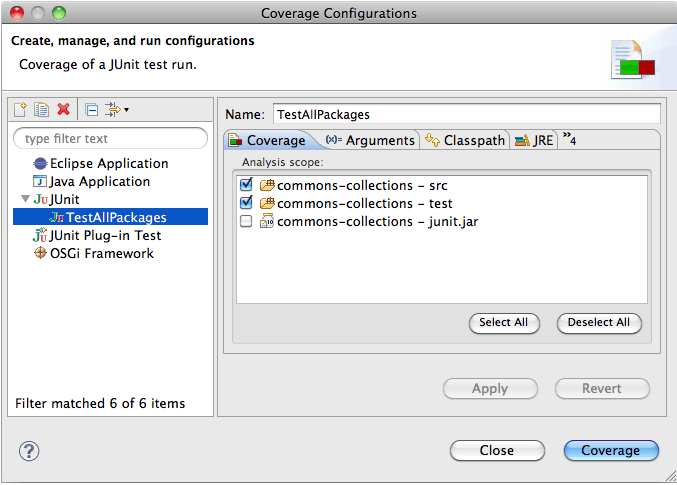
* Install from Eclipse Marketplace Client: EclEmma requires Eclipse 3.8 or higher versions and Java 1.5 or higher versions.
* Follow the steps below or drag and drop the button above into a running Indigo workspace:
* From your Eclipse menu select Help → Eclipse Marketplace
* Search for "EclEmma"
* Hit Install for the entry "EclEmma Java Code Coverage"
* Follow the steps in the installation wizard

**Step 17.2:** Verifying Installed EclEmma tool

* The installation was successful if you can see the coverage launcher in the toolbar of the Java perspective:

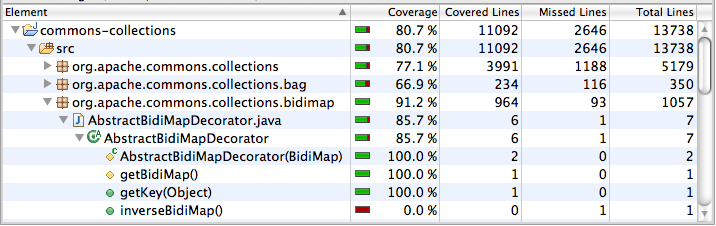


* Note: If the Coverage drop-down toolbar button is not visible in your current workbench perspective, open the **Customize Perspective... dialog** and enable the **Coverage command group** on the Commands tab.



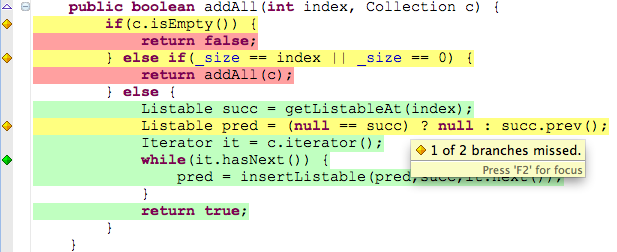
## Step 17.3: Using the Coverage View

* The *Coverage* view automatically appears when a new coverage session is added or can be manually opened from the *Window → Show View* menu in the *Java* category. It shows coverage summaries for the active session.



## Step 17.4: Demonstrating Source Code Annotation

* Line coverage and branch coverage of the active coverage session is also directly displayed in the Java source editors. This works for Java source files contained in the project as well as source codes attached to binary libraries.



* Source lines containing executable code have the following color code:
* **Green** for fully covered lines
* **Yellow** for partly covered lines (some instructions or branches that are missed)
* **Red** for lines that have not been executed at all
* In addition, colored diamonds are shown at the left for the lines containing decision branches. The colors for the diamonds have a similar semantic than the line highlighting the colors:
* **Green** for fully covered branches
* **Yellow** for partly covered branches
* **Red** when no branches in the particular line have been executed
* These default colors can be modified in the *Preferences* dialog (see next section). The source annotations automatically disappear when you start editing a source file or delete the coverage session.

## Step 17.5: Highlighting Preferences

The Eclipse preferences section *General → Appearance → Editors → Text Editors → Annotations* allows to modify the visual representation of coverage highlighting. The corresponding entries are:

* Full Coverage
* Partial Coverage
* No Coverage

## Step 17.6: Demonstrating Coverage Properties

For each Java element (Java project, source folder, package, type, or method) EclEmma provides a *Coverage* property page summarizing all coverage counters:

