2.1 Install Artifactory

This section will guide you:

* To understand Artifactory
* To install Artifactory
* To configure Artifactory with Jenkins

**Development Environment:**

* Jenkins
* Artifactory

This guide has three sub-sections, namely:

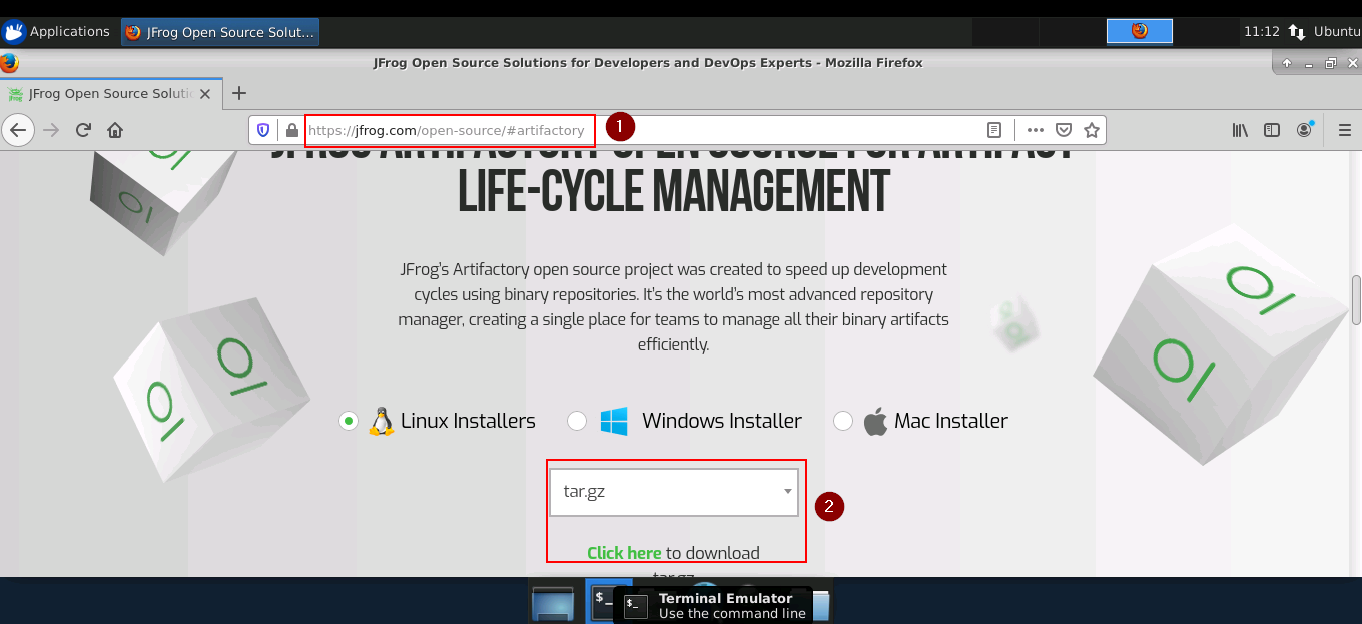
2.1.1 Installing Artifactory

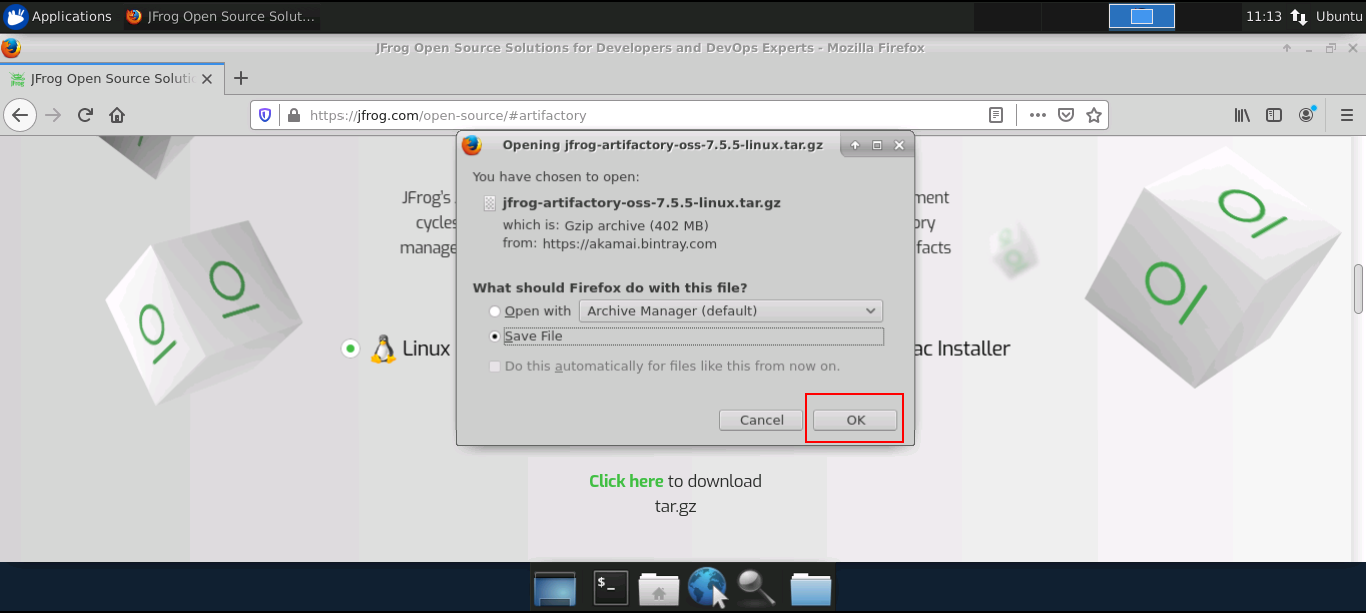
2.1.2 Setting up Artifactory with Jenkins

2.1.3 Pushing the code into GitHub repositories

**Step 2.1.1:** Installing Artifactory

* Go to this link: [https://jfrog.com/open-source/#artifactory](https://jfrog.com/open-source/" \l "artifactory).
* Download the tar.gz file as shown in the following screenshot

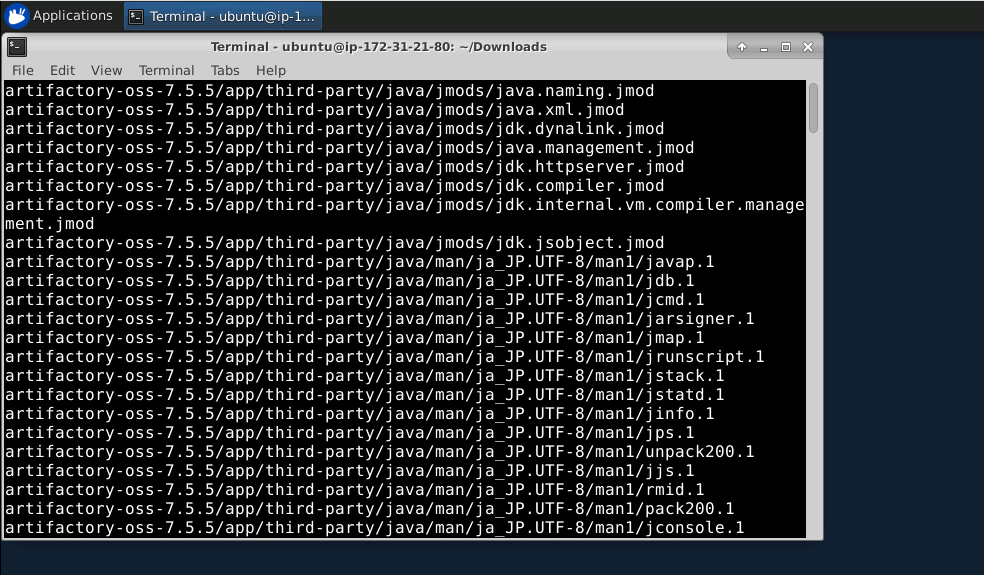




* Use the following command to extract the tar file.

*cd Downloads*

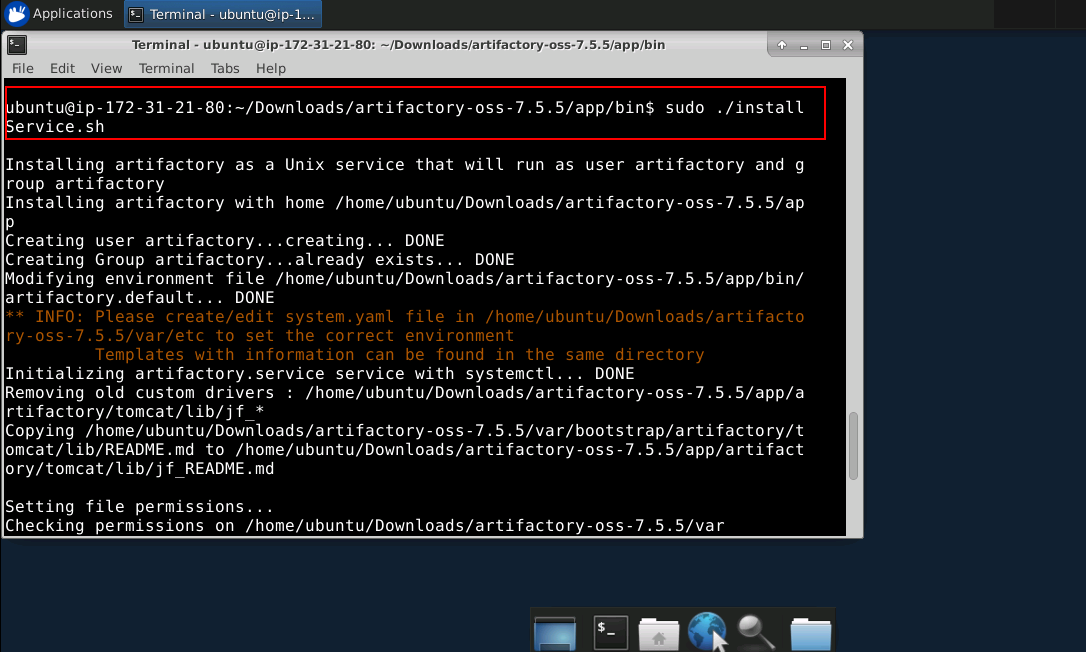
*tar -xvf jfrog-artifactory-oss-7.5.5-linux.tar.gz*



* The tar file will be extracted in a directory named artifactory-oss-7.5.5
* Open the command prompt and go to the bin of the Artifactory folder and run the following commands:

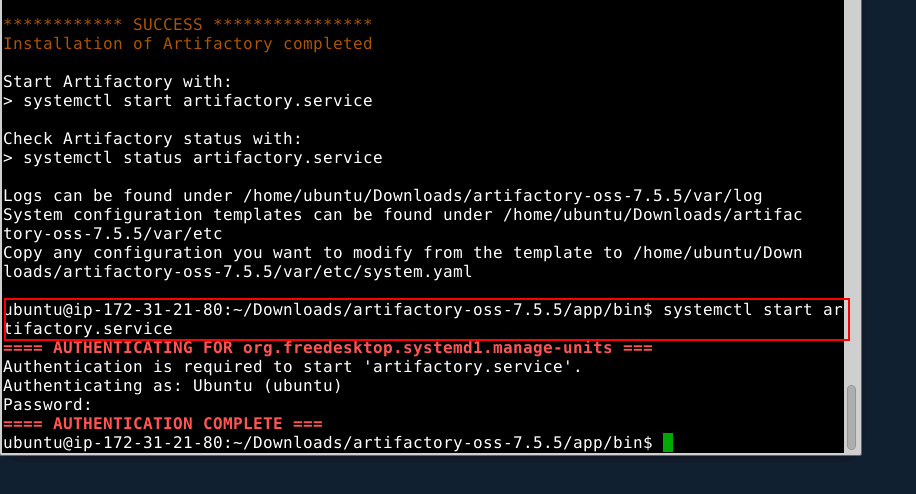
*cd Downloads/artifactory-oss-7.5.5/artifactory/app/bin*

*sudo ./installService.sh*



* Now, run the command:

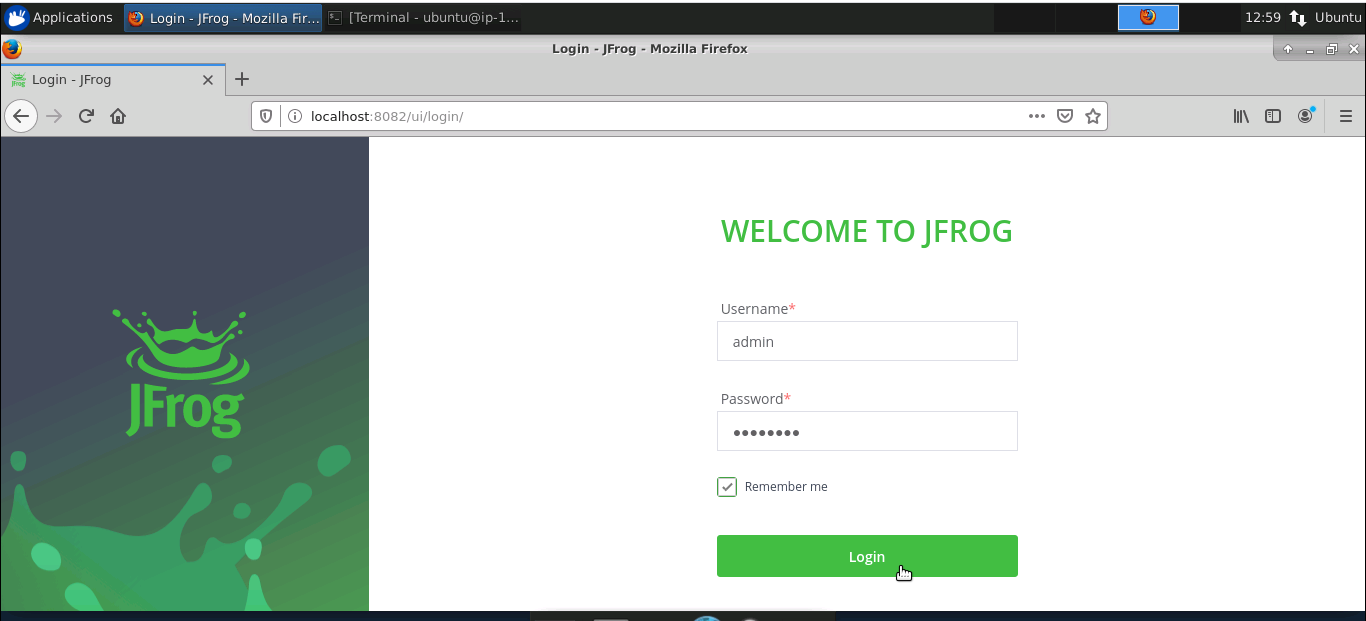
*systemctl start artifactory.service*



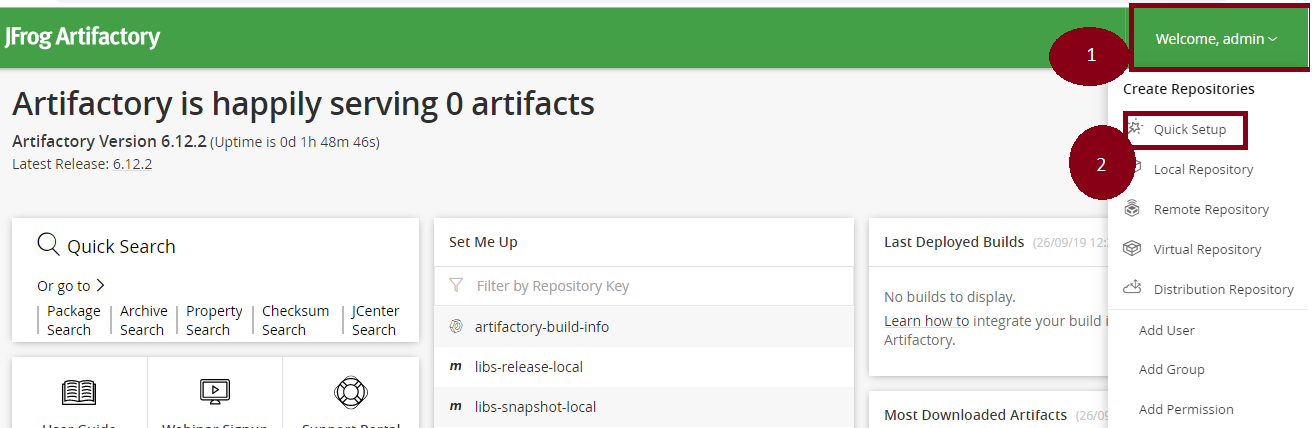
* Now, go to the browser type- localhost:8081.
* Enter the following details:

ID : admin

Password: password



* Click on ‘Welcome Admin.’
* Click on ‘Quick Setup.’
* Create a maven repository.

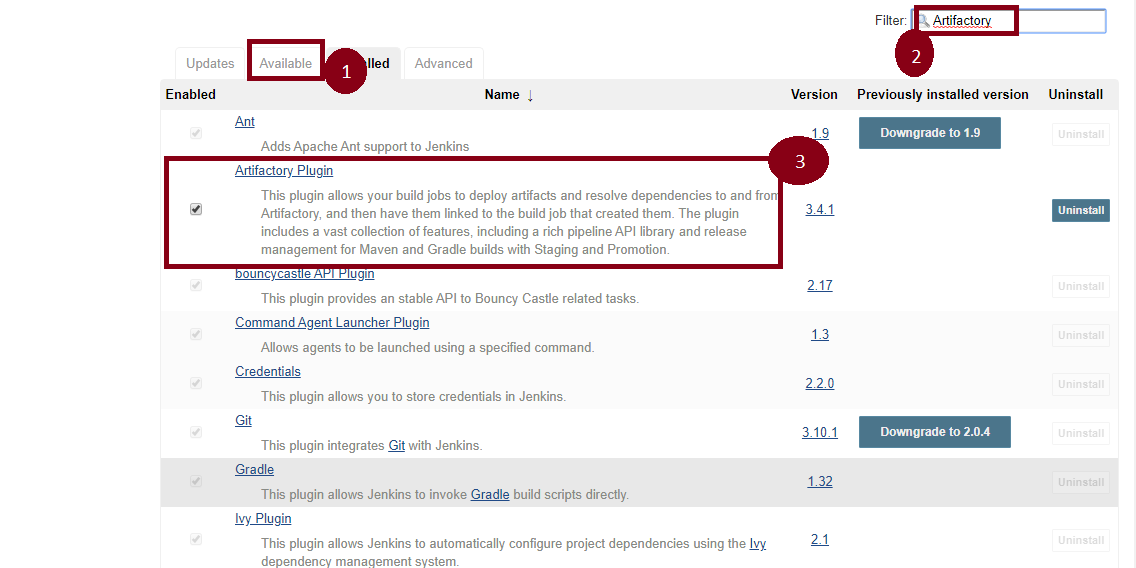


**Step 2.1.2:** Setting up Artifactory with Jenkins

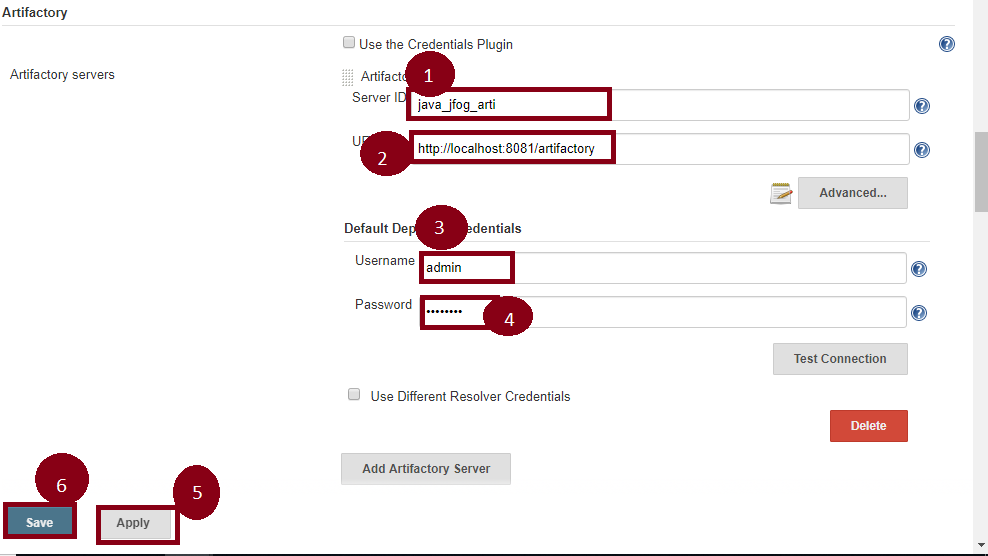
* Jenkins is already installed in your practice lab. Refer to QA to QE lab guide for Phase 2 for more information.
* Open command line and run the following command:

*sudo less /var/lib/jenkins/secrets/initialAdminPassword*

* Copy the password displayed in the command line
* Go to localhost:8080 in the browser and paste the password in the given field
* Click on install suggested plugins
* Go to ‘Manage Jenkins.’
* Click on ‘Manage Plugin.’
* Select the available tab.
* Search for Artifactory.

****

* Again, go to ‘Manage Jenkins.’
* Go to ‘Configure System.’
* There is an option Artifactory.
* Select ‘Add Artifactory.’
* Pass the artifactory localhost URL.
* Now, pass the credentials.
* Click on ‘Apply’ and ‘Save.’



**Step 2.1.3:** Pushing the Code to GitHub Repositories

Open your command prompt and navigate to the folder where you have created your files.

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

2.2 Build and Configure CI/CD Pipeline with Maven Project

This section will guide you:

* To understand CI/CD Pipeline
* To set up a Pipeline
* To write Jenkins file
* To build a CI/CD Pipeline with Maven

**Development Environment:**

* Jenkins

This guide has three sub-sections, namely:

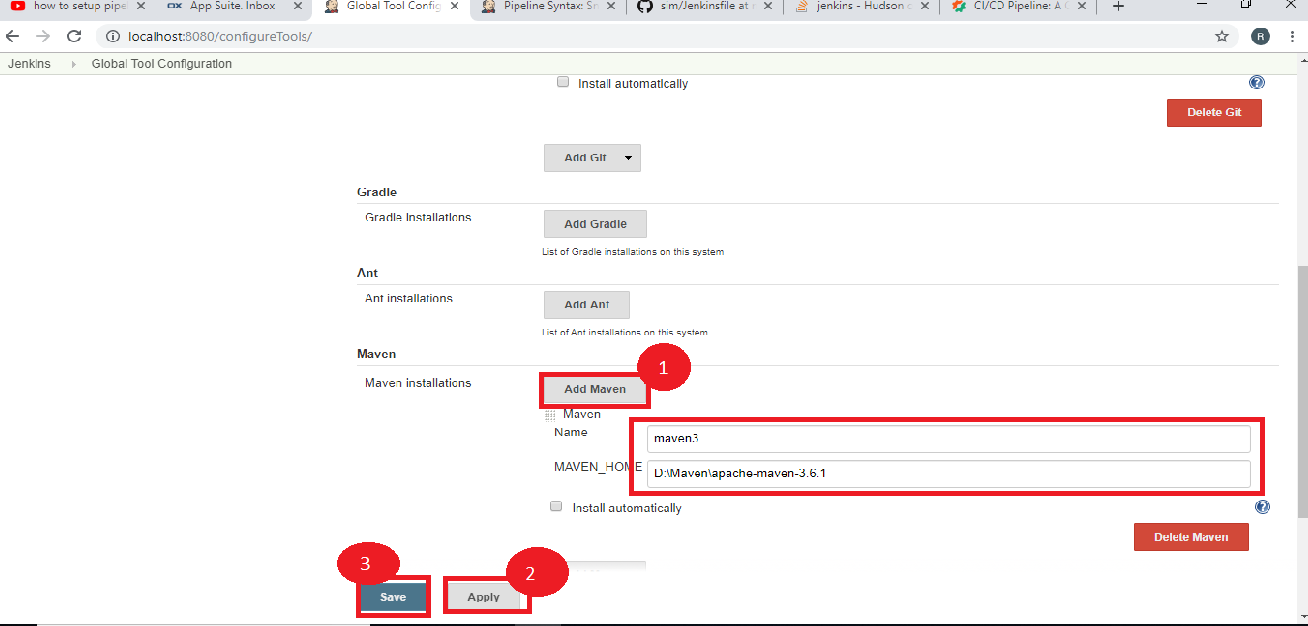
2.2.1 Setting up a Pipeline

2.2.2 Writing a Jenkins file

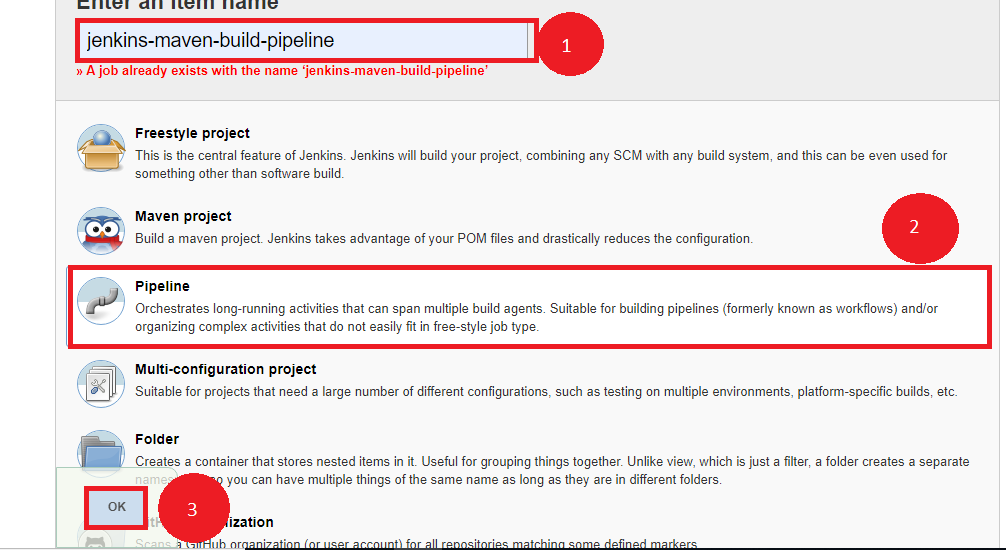
2.2.3 Pushing the code into GitHub repositories

**Step 2.2.1:** Setting up a Pipeline

* Jenkins is already set up in your practice lab. Refer to the lab guide for phase 2 for more information.
* Open Jenkins- locahhost:8080.
* Click on **Manage Jenkins.**
* Select **Global Tool Configuration.**
* Set the path of **Java and Maven.**
* Then, **Apply and Save.**



* Select a **New Item.**
* Name the job: **Jenkins-maven-build-pipeline.**
* Select the **Pipeline.**
* Now, click on **OK.**



**Step 2.2.2:** Writing a Jenkins file

* Create a job that needs to be configured.
* Click on **Pipeline.**
* Select the definition as- **Pipeline script from SCM.**
* Pass the GitHub URL.
* Now, we need to write the Jenkins file inside that git file.

node{

stage('SCM Checkout'){

git 'https://github.com/Autamation/sim.git'

}

stage('Compile-Package'){

def mvnHome = tool name: 'maven3', type: 'maven'

bat "${mvnHome}/bin/mvn package"

bat 'mvn package'

}

}

* Now, click on ‘Apply’ and ‘Save.’
* Click on **Build Now.**

**Step 2.2.3:** Pushing the code to GitHub repositories

Open your command prompt and navigate to the folder where you have created your files.

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

2.3 Build and Configure CI/CD Pipeline with Selenium WebDriver

This section will guide you to:

* Integrate Selenium WebDriver with Jenkins

**Development Environment:**

* Jenkins
* Selenium jars

This guide has three subsections, namely:

2.3.1 Forking the git repository

2.3.2 Creating a Jenkins pipeline job

2.3.3 Pushing the code into GitHub repositories

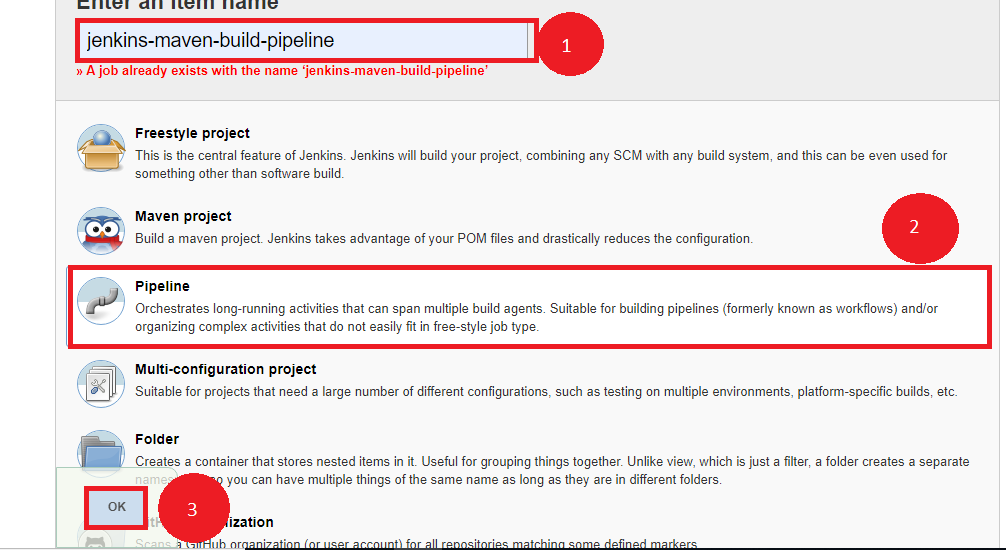
**Step 2.3.1:** Foking the git repository

* Fork the following repository

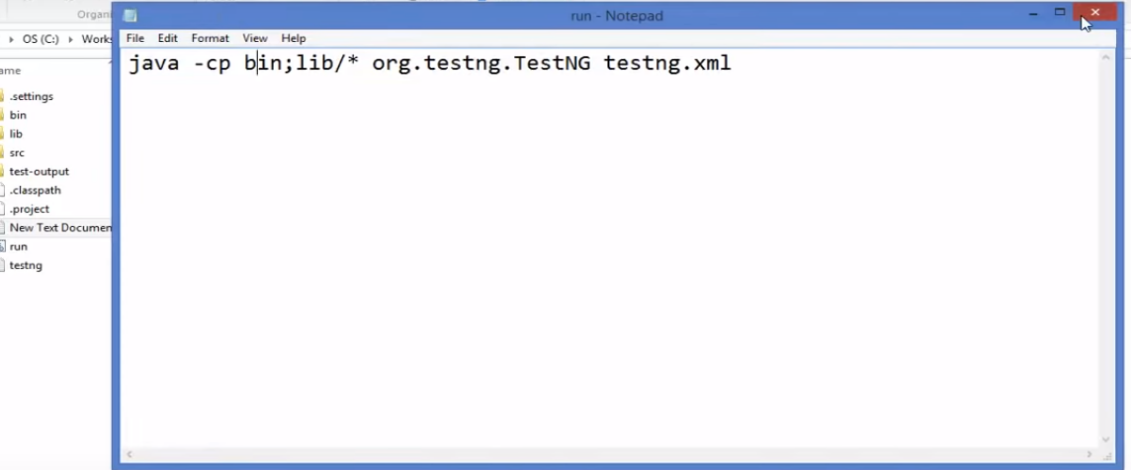
<https://github.com/canindit75/JenkinsDemo>

**Step 2.3.2:** Create a Jenkins pipeline job

* Java 1.8 is already installed in your practice lab. Refer to QA to QE lab guide for Phase 1 for more information.
* Jenkins.war file is already present in your practice lab in cd /usr/share/jenkins directory.
* Go to jenkins.war location Start the Jenkins by using command on command prompt:**java -jar jenkins.war.**
* Open browser and type **localhost:8080.**
* Enter the password.
* Create a job.
* Pass a name.
* Select **Pipeline.**
* Click on Ok.



* Create a text file name it **run.sh** in your lab and keep the below given code in it.



* Give executable permission to **run.sh** using the commands below:

**chmod 755 run.sh**

**chmod 777 run.sh**

* Push **run.sh in your repository** under master branch.

**git push <reponame> master**

**git status**

* Go to Jenkins pipeline job.
* Write a groovy script in the pipeline.

node {

def mvnHome

stage('Preparation') { // for display purposes

// Get some code from a GitHub repository

git 'https://github.com/jglick/simple-maven-project-with-tests.git'

// Get the Maven tool.

// \*\* NOTE: This 'M3' Maven tool must be configured

// \*\* in the global configuration.

mvnHome = tool 'maven3'

}

stage('Build') {

// Run the maven build

withEnv(["MVN\_HOME=$mvnHome"]) {

**if** (isUnix()) {

sh '"$MVN\_HOME/bin/mvn" -Dmaven.test.failure.ignore clean package'

} **else** {

sh ‘"%MVN\_HOME%\bin\mvn" -Dmaven.test.failure.ignore clean package’

}

}

}

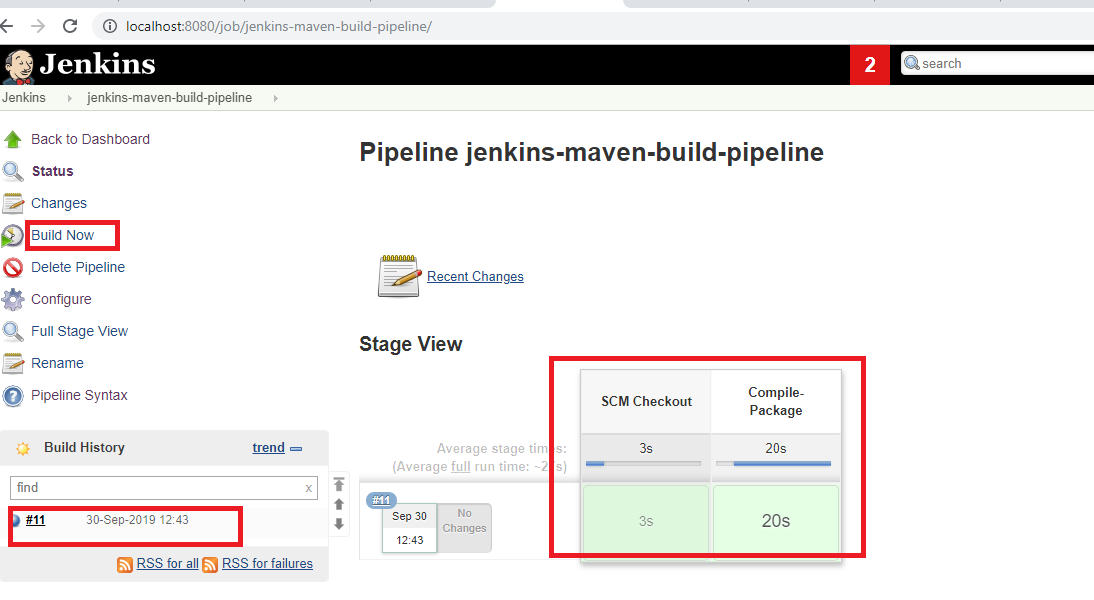
stage('Results') {

junit '\*\*/target/surefire-reports/TEST-\*.xml'

archiveArtifacts 'target/\*.jar'

}}

* Click on Apply and Save.
* Click on Build now.

****

**Step 2.3.3:** Pushing the code to GitHub repositories

Open your command prompt and navigate to the folder where you have created your files.

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

2.4 Selenium with Jenkins

This section will guide you:

* To Integrate Selenium WebDriver with Jenkins

**Development Environment:**

* Jenkins
* Selenium jars

This guide has four sub-sections, namely:

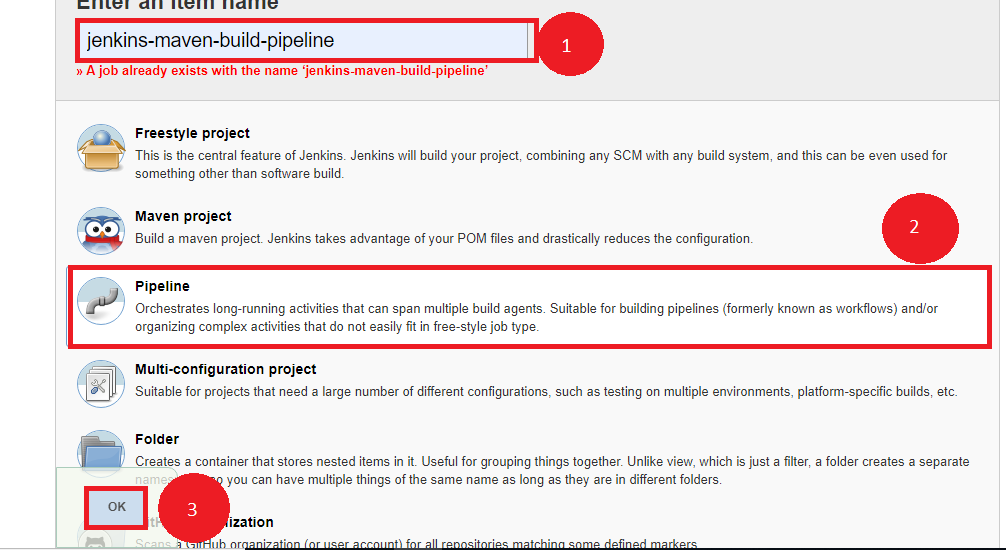
2.4.1 Creating a Jenkins Pipeline job

2.4.2 Integrating Selenium Webdriver with Jenkins

2.4.3 Pushing the code into GitHub repositories

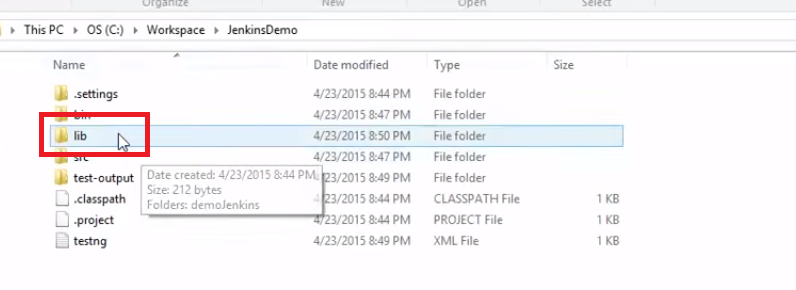
**Step 2.4.1:** Creating a Jenkins Pipeline job

* Java 1.8 is already installed in your practice lab.
* Jenkins.war file is already present in your practice lab in directory /usr/share/jenkins.
* Go to the jenkins.war location. Now, start Jenkins by using the following command on command prompt: **java -jar jenkins.war.**
* Open your browser and type- **localhost:8080.**
* Enter the password.
* Create a job.
* Enter the name.
* Select the **Pipeline.**
* Now, click on OK.

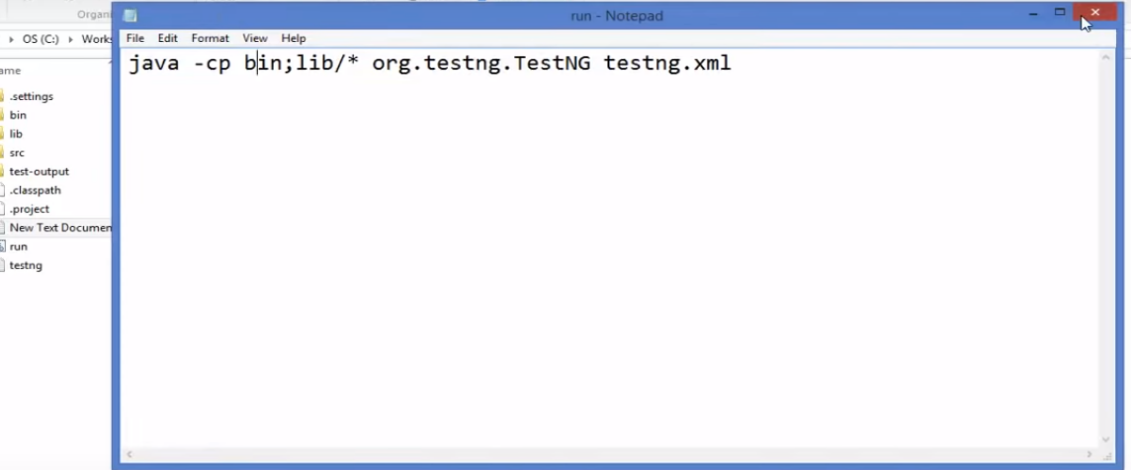


**Step 2.4.2:** Integrating Selenium WebDriver with Jenkins

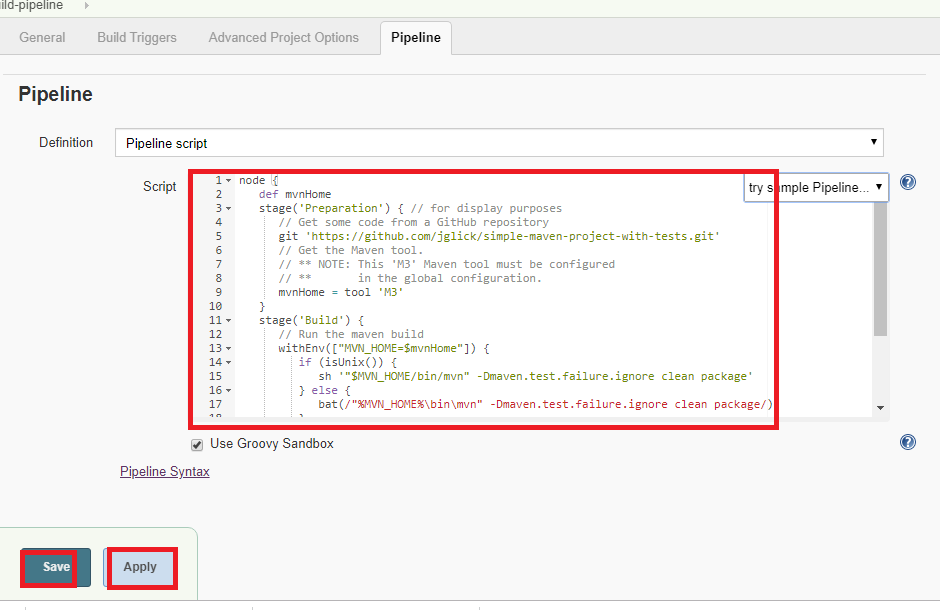
* Go to the Selenium script location.
* Add all the Selenium libraries.



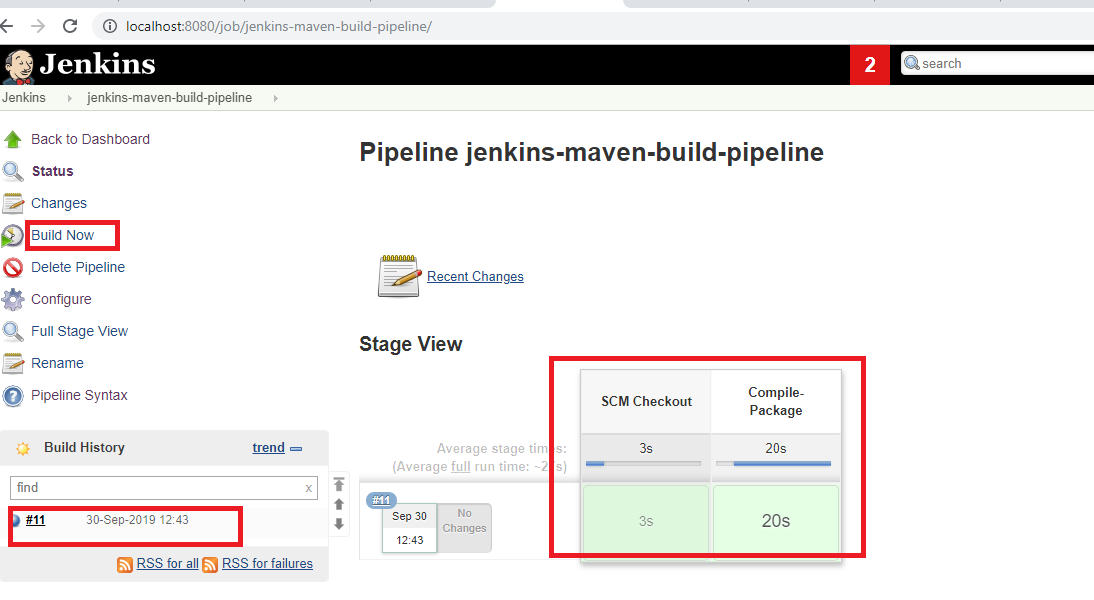
* Create a text file **run.bat** and keep it within double quotation marks.



* Go to the Jenkins Pipeline job.
* Pass the Selenium script location.
* Write a groovy script in Pipeline.
* Click on Apply and then Save.



* Now, click on **Build Now.**

****

**Step 2.4.3:** Pushing the code to GitHub repositories

Open your command prompt and navigate to the folder where you have created your files.

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

3.1 TDD with TestNG

This section will guide you:

* To understand TDD
* To perform TDD test with TestNG

**Development Environment:**

* Eclipse IDE for Enterprise Java Developers Version Oxygen.3a Release (4.7.3a)
* JavaDevelopment Kit Version 8
* Selenium Standalone Server Version 3.141.59

This guide has mainly three sub-sections, namely:

3.1.1 Performing a TDD test

3.1.2 Running the code

3.1.3 Pushing the code to your GitHub repositories

**Step 3.1.1:** Performing a TDD test:

* To perform a TDD test, follow the below steps:

1. Add the test.
2. Execute the test and see if the new one fails.
3. Write the code.
4. Execute the test.
5. Refactor the code.
6. Now, repeat the steps mentioned above.

Now, let’s look at the above steps in detail:

1. Firstly, write the code that will be based on the requirements in Eclipse. It should look something like:

**package** test.testing;

**import** org.testng.Assert;

**import** org.testng.annotations.Test;

**public** **class** AddNumbers {

**@Test**

**public** void addIntegerNumbers()

{

Calculator myCalculator = **new** Calculator();

int expected= 30;

int actual= myCalculator.add(10,20);

Assert.assertEquals(actual, expected);

}

}

1. Now, if we execute our test for the first time, we should get the below error:

FAILED: addIntegerNumbers

java.lang.Error: Unresolved compilation problems:

Calculator cannot be resolved to a type

Calculator cannot be resolved to a type

1. Write the code shown below to resolve the above error in Eclipse. It will look something like:

**package** test.testing;

**public** **class** Calculator {

**public** int add(int number1, int number2)

{

**return** 0;

}

}

1. Now, execute our test:

FAILED: addIntegerNumbers

java.lang.AssertionError: expected [30] but found [0]

1. Refactor the code in Eclipse. It will look like:

**package** test.testing;

**public** **class** Calculator {

**public** int add(int number1, int number2)

{

**return** (number1+number2);

}

}

1. Now, if execute our test again, it will show the below message:

PASSED: addIntegerNumbers

**Step 3.1.2:** Running the code:

* Run the code through Eclipse.

**Step 3.1.3:** Pushing the code to your GitHub repositories:

Open your command prompt and navigate to the folder where you have created your files:

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

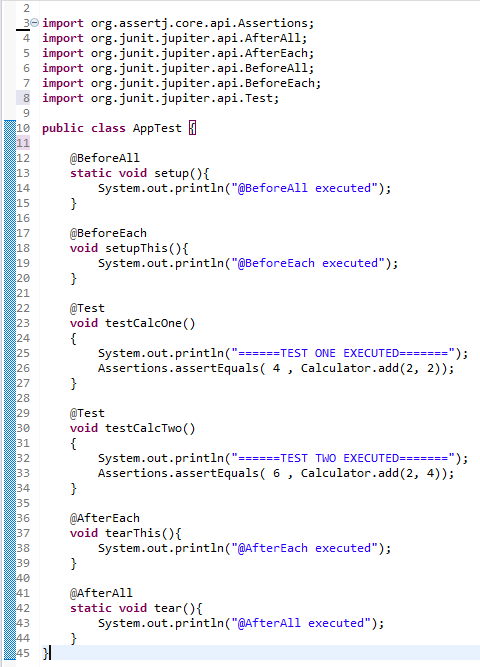
4.1 Lifecycle Methods

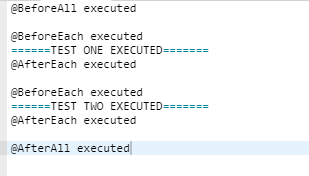
This guide has two sub-sections, namely:

* + 1. Writing code for @BeforeAll annotation, @BeforeEach annotation, @AfterAll annotation, and @AfterEach annotation
    2. Pushing the code to your GitHub repositories

**Step 4.1.1:** 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.
* 



**Step 4.1.2:** Pushing the code to your GitHub repositories:

Open your command prompt and navigate to the folder where you have created your files:

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

4.2 Assertions

This section will guide you:

* To understand Assertions
* To understand Assertion Methods

This guide has three sub-sections, namely:

* + 1. Explaining Assertion Methods
    2. Writing code for Assertions
    3. Pushing the code to your GitHub repositories

**Steps 4.2.1**: 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 4.2.2:** Writing code for Assertions

Let's use some of the above-mentioned methods in an example. Create a java class file named **TestAssertions.java** in /home/ubuntu/Desktop/JUNIT\_WORKSPACE/.

import org.junit.Test;

import static org.junit.Assert.\*;

public class TestAssertions {

@Test

public void testAssertions() {

//test data

String str1 = new String ("abc");

String str2 = new String ("abc");

String str3 = null;

String str4 = "abc";

String str5 = "abc";

int val1 = 5;

int val2 = 6;

String[] expectedArray = {"one", "two", "three"};

String[] resultArray = {"one", "two", "three"};

//Check that two objects are equal

assertEquals(str1, str2);

//Check that a condition is true

assertTrue (val1 < val2);

//Check that a condition is false

assertFalse(val1 > val2);

//Check that an object isn't null

assertNotNull(str1);

//Check that an object is null

assertNull(str3);

//Check if two object references point to the same object

assertSame(str4,str5);

//Check if two object references not point to the same object

assertNotSame(str1,str3);

//Check whether two arrays are equal to each other.

assertArrayEquals(expectedArray, resultArray);

}

}

Next, create a java class file named **TestRunner.java** in /home/ubuntu/Desktop/JUNIT\_WORKSPACE/ to execute the test case(s).

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner2 {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(TestAssertions.class);

for (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

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

}

}

Compile the Test Case and Test Runner classes using javac.

/home/ubuntu/Desktop/JUNIT\_WORKSPACE/>javac TestAssertions.java TestRunner.java

Now run the Runner, which will run the test case defined in the provided Test Case Class.

/home/ubuntu/Desktop/JUNIT\_WORKSPACE/>java TestRunner

Verify the output.

true

**Step 4.2.3:** Pushing the code to GitHub repositories:

Open your command prompt and navigate to the folder where you have created your files:

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

4.3 Disabling Tests

This section will guide you:

1. To disable a test method
2. Example of @ignore annotation
3. To use @ignore annotation

This guide has two sub-sections, namely:

* + 1. Writing code to demonstrate @ignore annotation
    2. Pushing the code to your GitHub repositories

**Step 4.3.1:** 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

**JunitTestExample.java**

package SSS.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Ignore;

import org.junit.Test;

public class JunitDemoExample {

public String message = "Welcome";

JUnitMessage junitMsg = new JUnitMessage(message);

@Ignore

@Test

public void testJUnitMessage() {

System.out.println("Junit Message is printing ");

assertEquals(message, junitMsg.printMessage());

}

@Test

public void testJUnitHiMessage() {

message="Hi!" +message;

System.out.println("Junit Hi Message is printing ");

assertEquals(message, junitMsg.printHiMessage());

}

}

**Output:**

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

**Print statement on console:**

Junit Hi Message is printing

Hi!Welcome

**Step 4.3.2:** Pushing the code to GitHub repositories:

Open your command prompt and navigate to the folder where you have created your files:

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

4.4 Assumptions

This section will guide you:

* To import the Junit assumptions and its types

This section has mainly three sub-sections, namely:

* + 1. Importing a Junit Assumption
    2. Writing a code to demonstrate the types of Assumptions
    3. Pushing the code to GitHub repositories

**Steps 4.4.1:** 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.

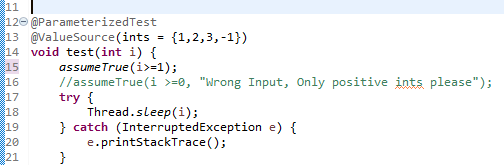
import static org.junit.jupiter.api.Assumptions.\*;

import static org.junit.Assume.assumeTrue;

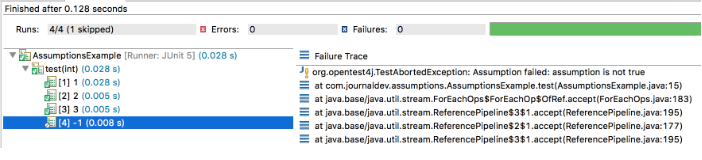
**Steps 4.4.2:** Writing a code to demonstrate the types of Assumptions

* assumeTrue()

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

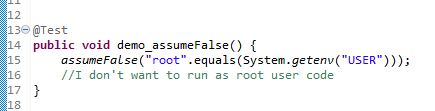


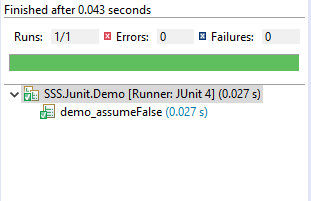
* JUnit will skip when the input number is negative. When we run the test, we will get the following output:



* assumeFalse()

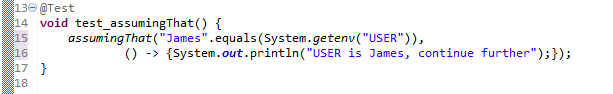
assumeFalse() validates the given assumption to false and if the assumption is false then the test proceeds, otherwise, the test execution is aborted. It works just opposite to assumeTrue().





* 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.



**Step 4.4.3:** Pushing the code to GitHub repositories:

Open your command prompt and navigate to the folder where you have created your files:

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

4.5 Test Interfaces and Default Methods

This section will guide you:

* To understand interfaces and default methods

This section has mainly three sub-sections, namely:

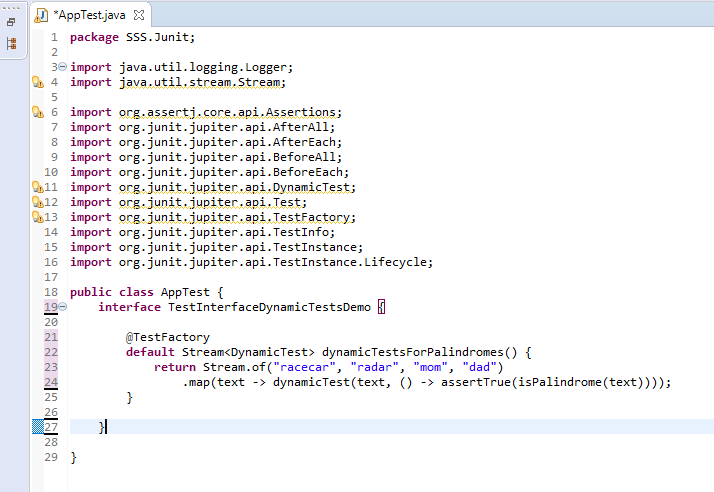
* + 1. Writing a code to demonstrate interfaces and default methods
    2. Writing a code to demonstrate TestFactory
    3. Pushing the code to GitHub repositories

**Step 4.5.1:** 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 4.5.2:** Writing a code to demonstrate TestFactory



**Step 4.5.3:** Pushing the code to GitHub repositories:

Open your command prompt and navigate to the folder where you have created your files:

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

4.6 Repeating Tests

This section will guide you:

1. To understand how the repeated tests are executed using @RepeatedTest annotation
2. To use RepeatedTest @DisplayName annotation
3. To use @RepetitionInfo annotation

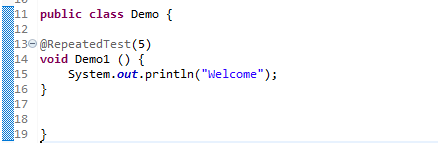
This section has mainly two sub-sections, namely:

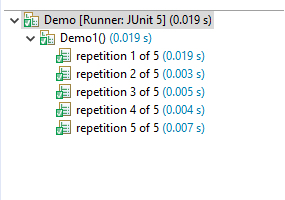
* + 1. Writing a code to demonstrate the execution of repeated tests
    2. Pushing the code to GitHub repositories

**Step 4.6.1:** 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.



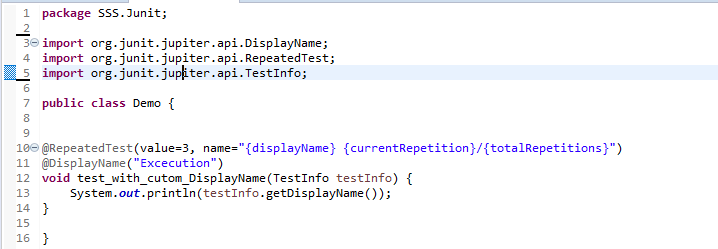


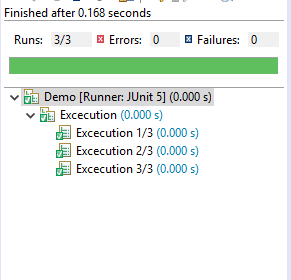


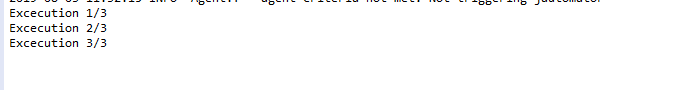
**4.6.1.2 @RepeatedTest DisplayName**

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

**DisplayName value**

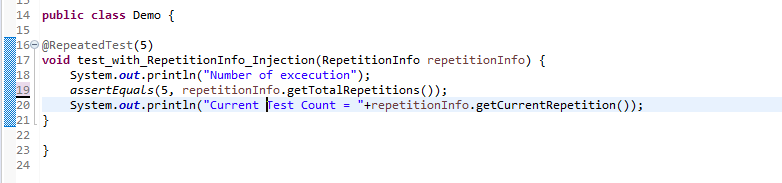




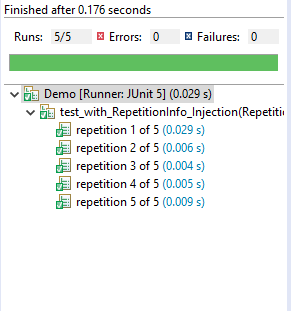


## **4.6.1.3 JUnit @RepetitionInfo**

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)]

**Step 4.6.2:** Pushing the code to GitHub repositories

Open your command prompt and navigate to the folder where you have created your files:

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

4.7 Dynamic Tests

This section will guide you:

* To import Dynamic Test libraries
* To know JUnit @TestFactory Example
* To know Dynamic Test Examples

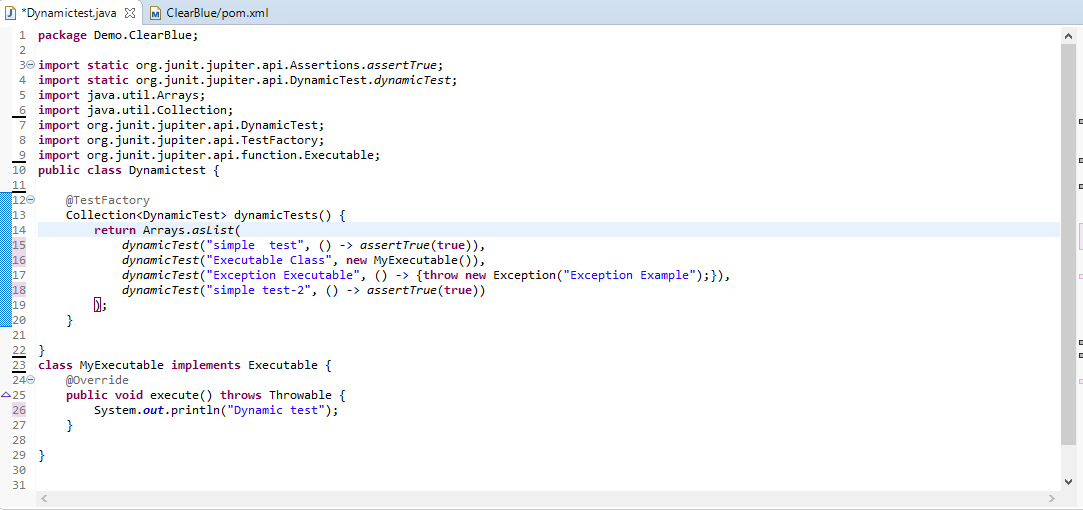
This section has mainly four sub-sections, namely:

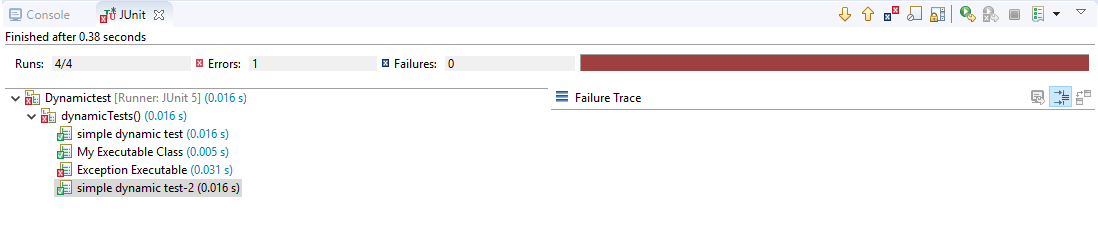
* + 1. Writing a code to import Dynamic Test libraries
    2. Writing a code to demonstrate JUnit @TestFactory
    3. Writing a code to demonstrate the execution of Dynamic Tests
    4. Pushing the code to GitHub repositories

**Step 4.7.1:** 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;

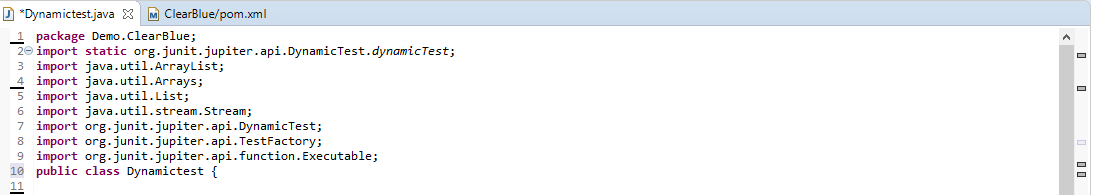
**Step 4.7.2:**  Writing a code to demonstrate JUnit @TestFactory



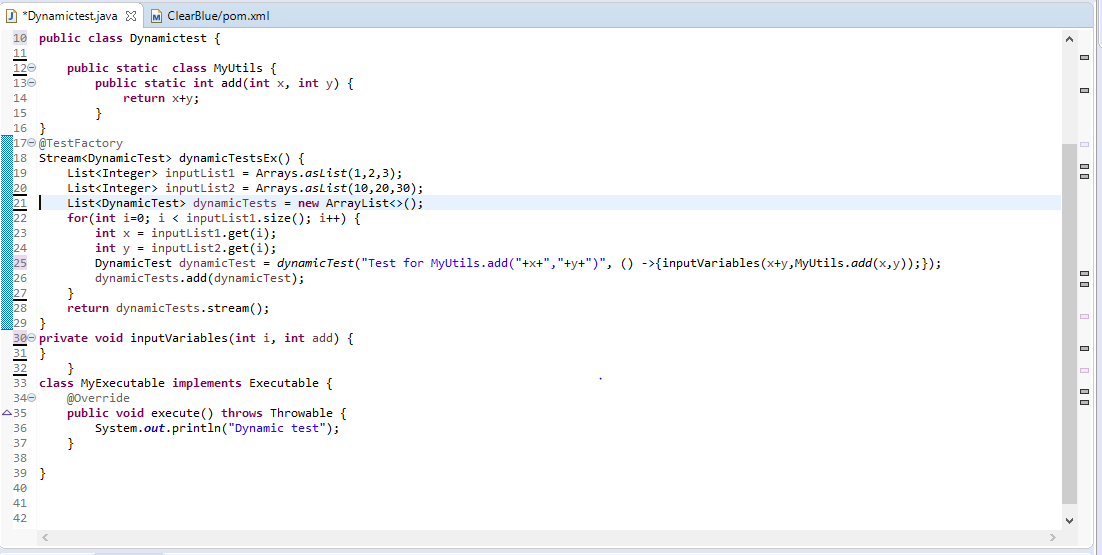


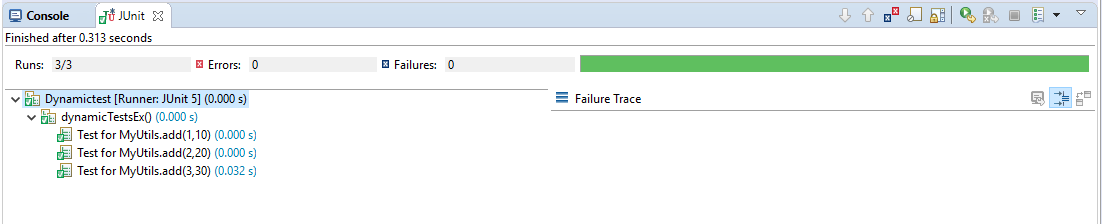
**Step 4.7.3:** Writing a code to demonstrate the execution of Dynamic Tests

* **Screenshot 1:**



* **Screenshot 2:**





**Step 4.7.4:** Pushing the code to GitHub repositories:

Open your command prompt and navigate to the folder where you have created your files:

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

4.8 Parameterized Tests

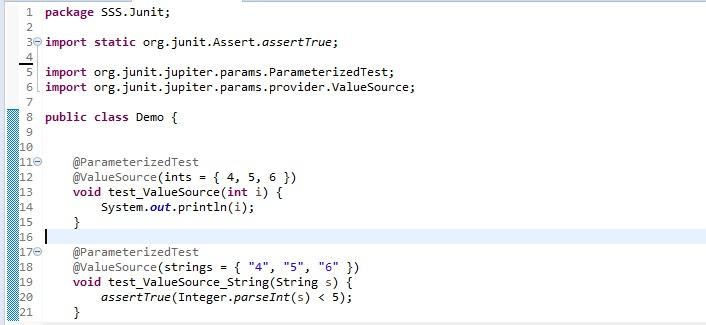
This section will guide you:

* To understand @ParameterizedTest with @ValueSource
* To understand @ParameterizedTest with @EnumSource
* To understand @ParameterizedTest with @MethodSource
* To understand @ParameterizedTest with @CsvSource

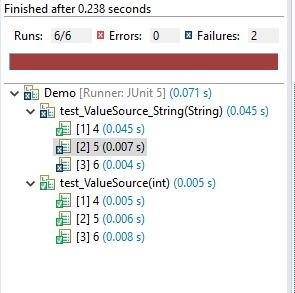
This section has mainly five sub-sections, namely:

* + 1. Writing a code to demonstrate Parameterized Test with @ValueSource
    2. Writing a code to demonstrate @ParameterizedTest with @EnumSource
    3. Writing a code to demonstrate@ParameterizedTest with @MethodSource
    4. Writing a code to demonstrate@ParameterizedTest with @CsvSource
    5. Pushing the code to GitHub repositories

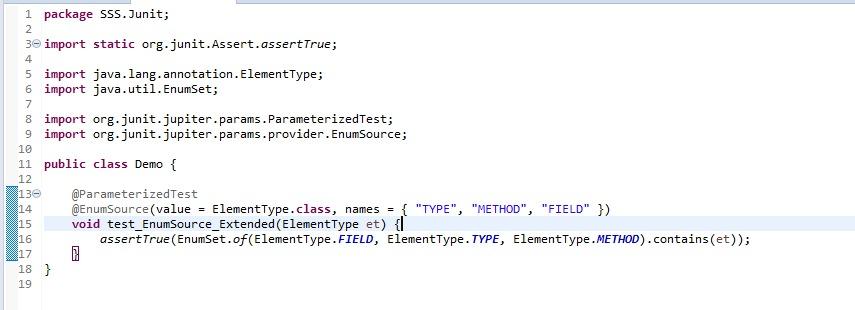
**Step 4.8.1:** Writing a code to demonstrate Parameterized Test with @ValueSource

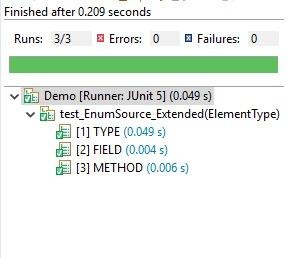




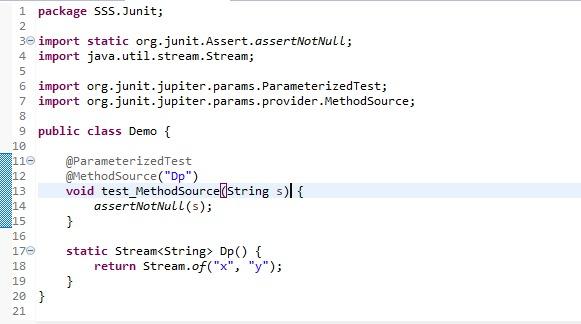


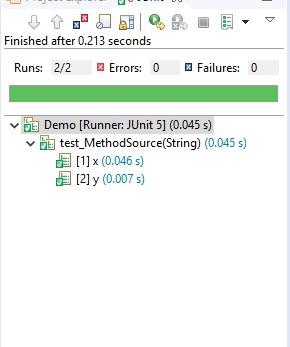
**Step 4.8.2:** Writing a code to demonstrate @ParameterizedTest with @EnumSource



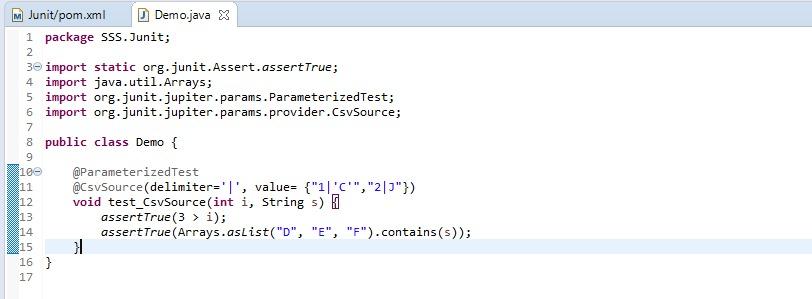


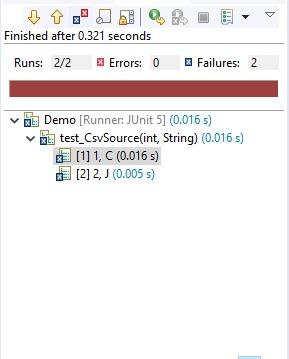
**Step 4.8.3:** Writing a code to demonstrate@ParameterizedTest with @MethodSource

****

****

**Step 4.8.4:** Writing a code to demonstrate@ParameterizedTest with @CsvSource



****

**Step 4.8.5:** Pushing the code to GitHub repositories

Open your command prompt and navigate to the folder where you have created your files:

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

4.10 Argument Conversion

This section will guide you:

* To understand Implicit Conversion
* To understand Explicit Conversion

This section has mainly three sub-sections, namely:

* + 1. Writing a code to demonstrate Implicit Conversion
    2. Writing a code to demonstrate Explicit Conversion
    3. Pushing the code to GitHub repositories

**Step 4.10.1:** 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

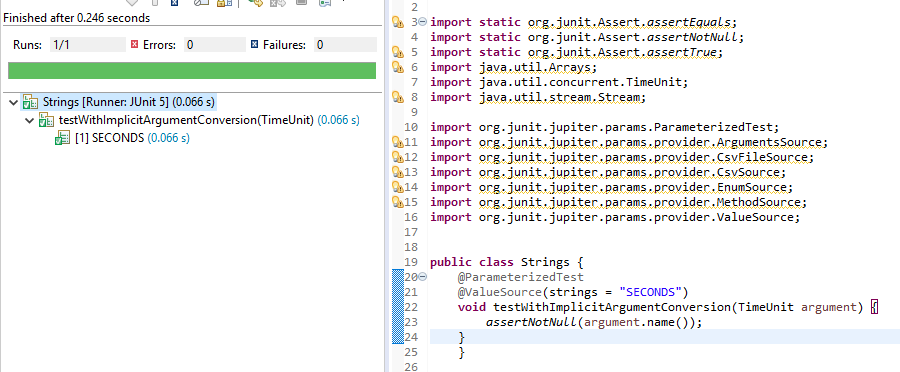
protected Object convert(Object source, Class<?> targetType) {

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

return String.valueOf(source);

}

}



**Step 4.10.2:** 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)

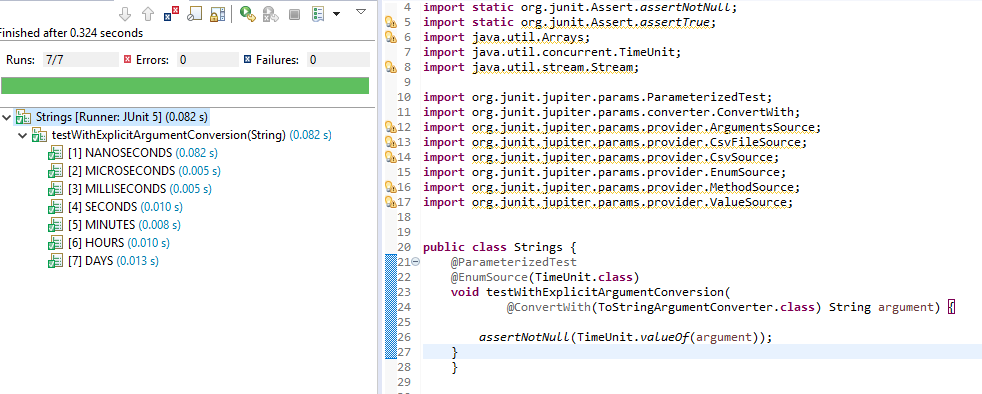
{

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

return String.valueOf(source);

}

}



**Step 4.10.3:** Pushing the code to GitHub repositories

Open your command prompt and navigate to the folder where you have created your files:

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

4.9 Argument Sources

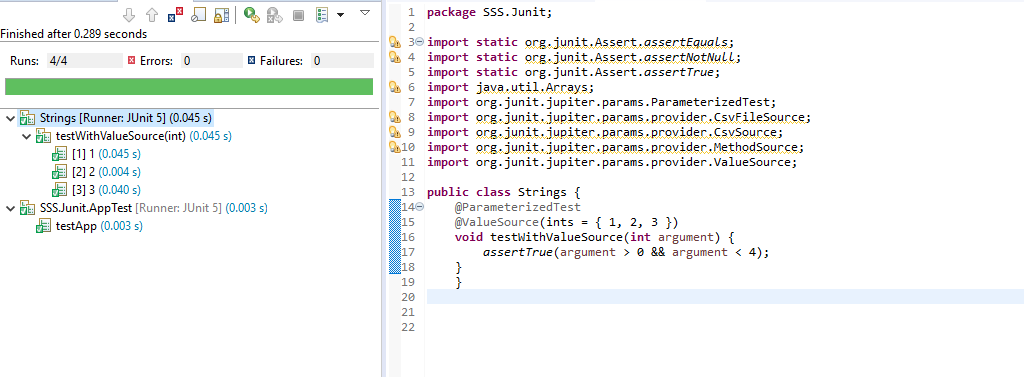
This section will guide you:

* To understand @ValueSource annotation
* To understand @CsvSource annotation
* To understand @EnumSource annotation
* To understand @ArgumentsSource annotation

This section has mainly five sub-sections, namely:

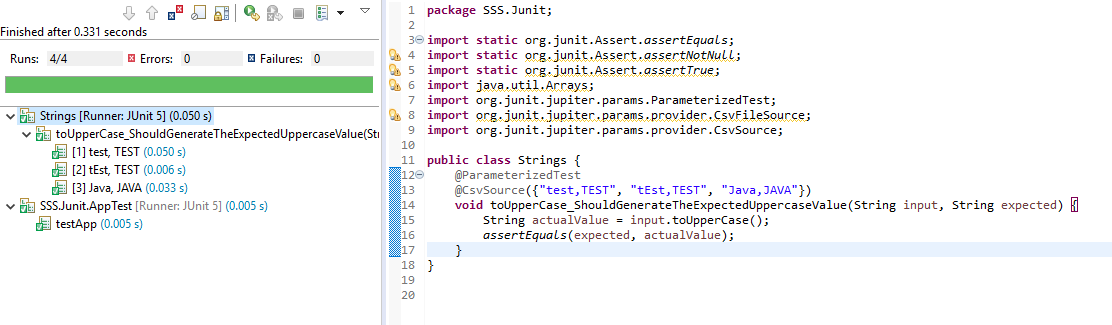
* + 1. Writing a code to demonstrate @ValueSource annotation
    2. Writing a code to demonstrate @CsvSource annotation
    3. Writing a code to demonstrate@EnumSource annotation
    4. Writing a code to demonstrate@ArgumentsSource annotation
    5. Pushing the code to GitHub repositories

**Step 4.9.1:**  Writing a code to demonstrate @ValueSource annotation

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

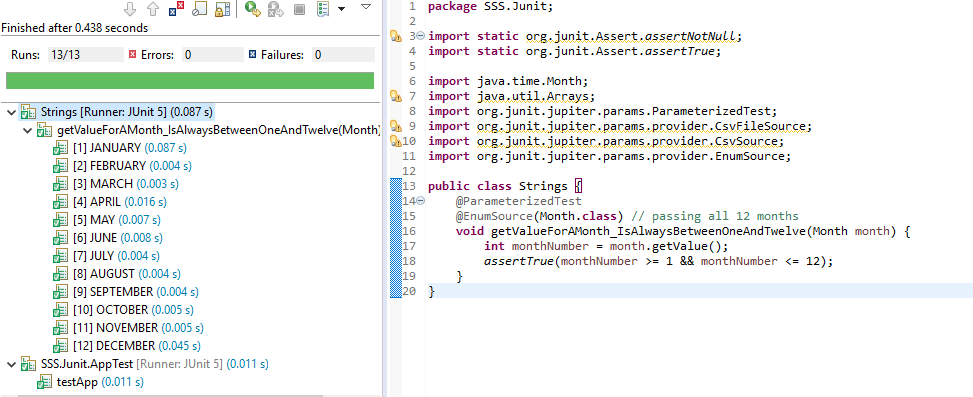
**Step 4.9.2:** Writing a code to demonstrate @CsvSource annotation

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



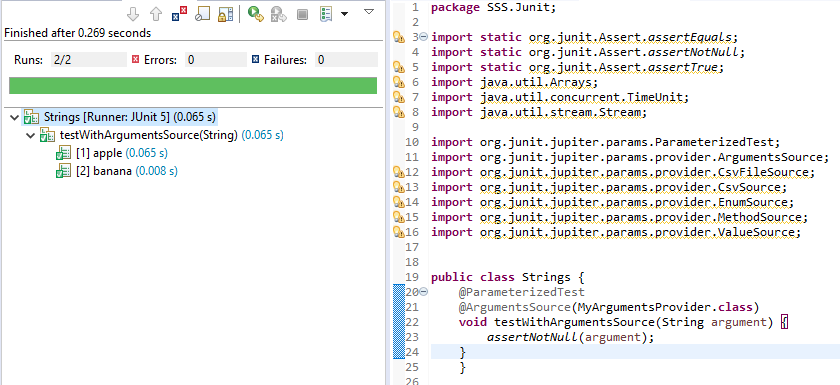
**Step 4.9.3:** Writing a code to demonstrate@EnumSource annotation

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



**Step 4.9.4:** 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.



**Step 4.9.5:** Pushing the code to GitHub repositories

Open your command prompt and navigate to the folder where you have created your files:

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

4.11 Extension Points

This section will guide you:

* To understand Extension Points

This guide has four sub-sections, namely:

* + 1. Writing a code to demonstrate **Life Cycle Call Back**

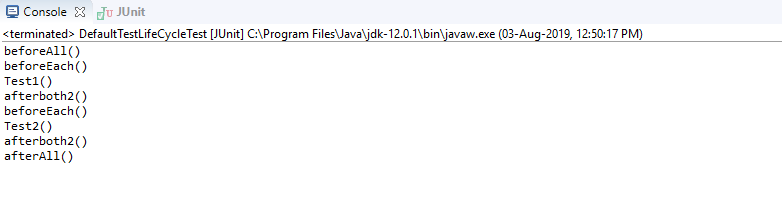
4.11.2 Writing a code to demonstrate **Conditional Test Execution**

4.11.3 Writing a code to demonstrate **Exception Handling Extension**

4.11.4 Pushing the code to GitHub repositories

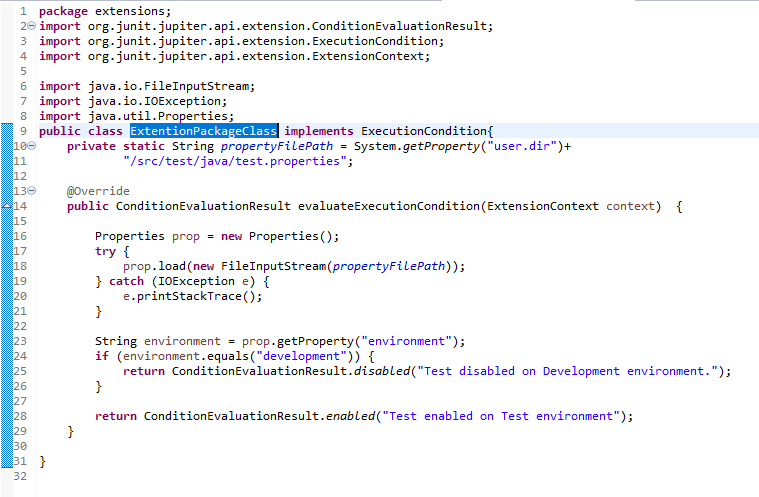
**Steps 4.11.1:** Writing a code to demonstrate Life Cycle Call Back



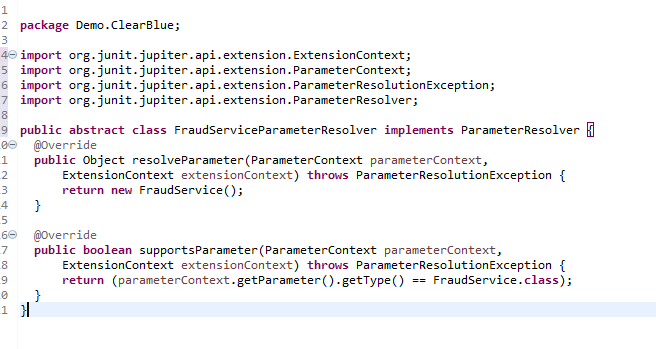
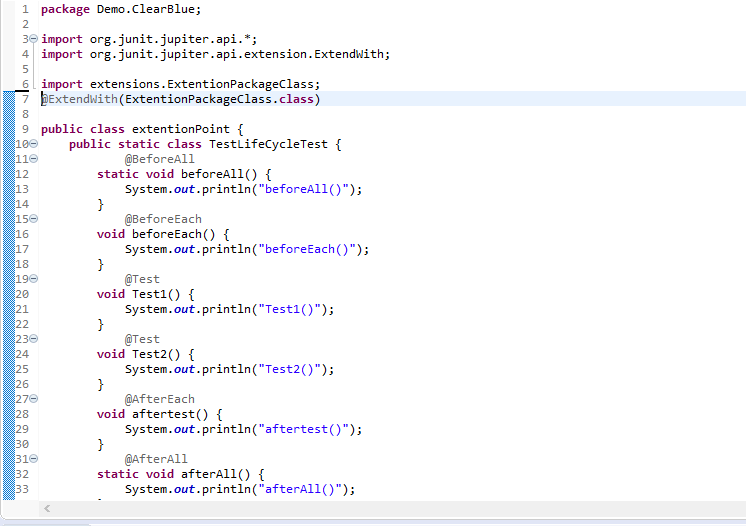


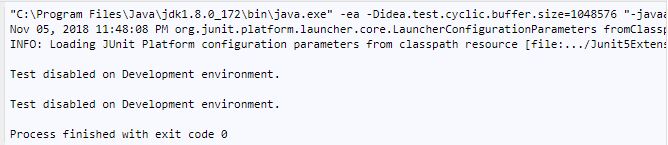
**Steps 4.11.2:** Writing code to demonstrate Conditional Test Execution

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



* Now, execute the below class:

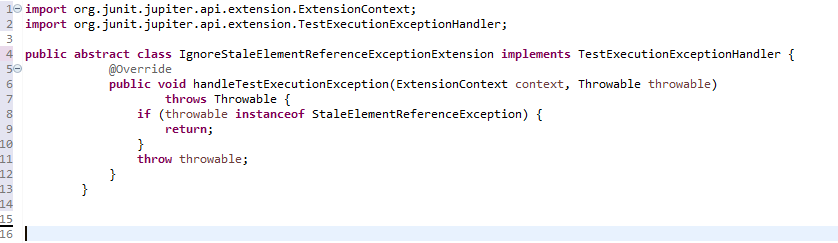


* We should register this with @ExtendWith(FraudServiceParameterResolver.class)

annotation on top of our test classes.

**Steps 4.11.3:** Writing a code to demonstrate Exception Handling Extension

* Create an extension class as shown below:



* We need to implement the **TestExecutionExceptionHandler** interface.
* **StaleElementReference** will ignore all the **StaleElementReference** errors, but it will show other errors.

**Step 4.11.4:** Pushing the code to GitHub repositories

Open your command prompt and navigate to the folder where you have created your files:

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master

4.13 Meta-Annotations

This section will guide you to understand:

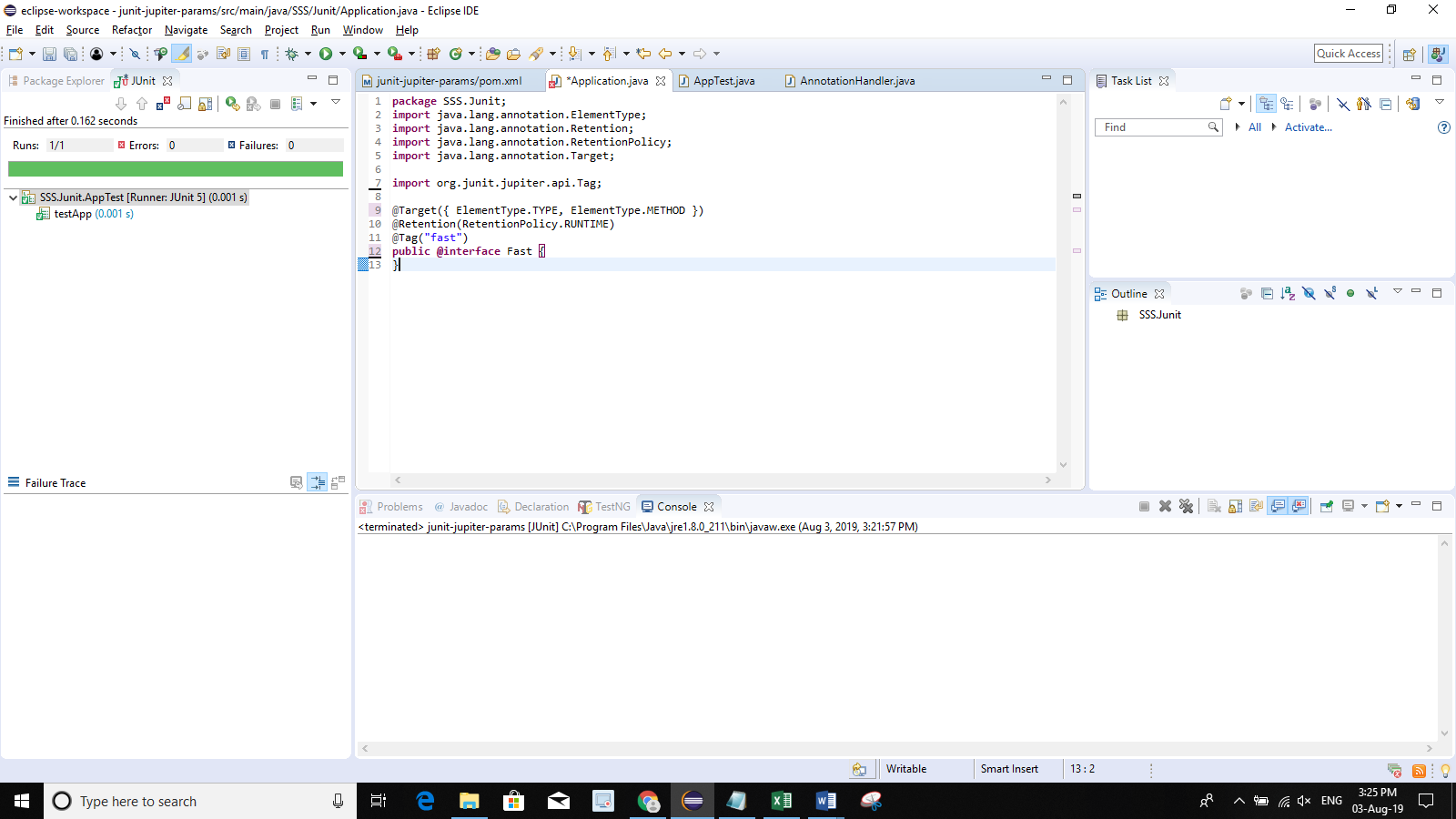
* Meta Annotations

This guide has two subsections, namely:

* + 1. Writing a code to demonstrate Meta-Annotations and Composed Annotations
    2. Pushing the code to GitHub repositories

**Step 4.13.1:** 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

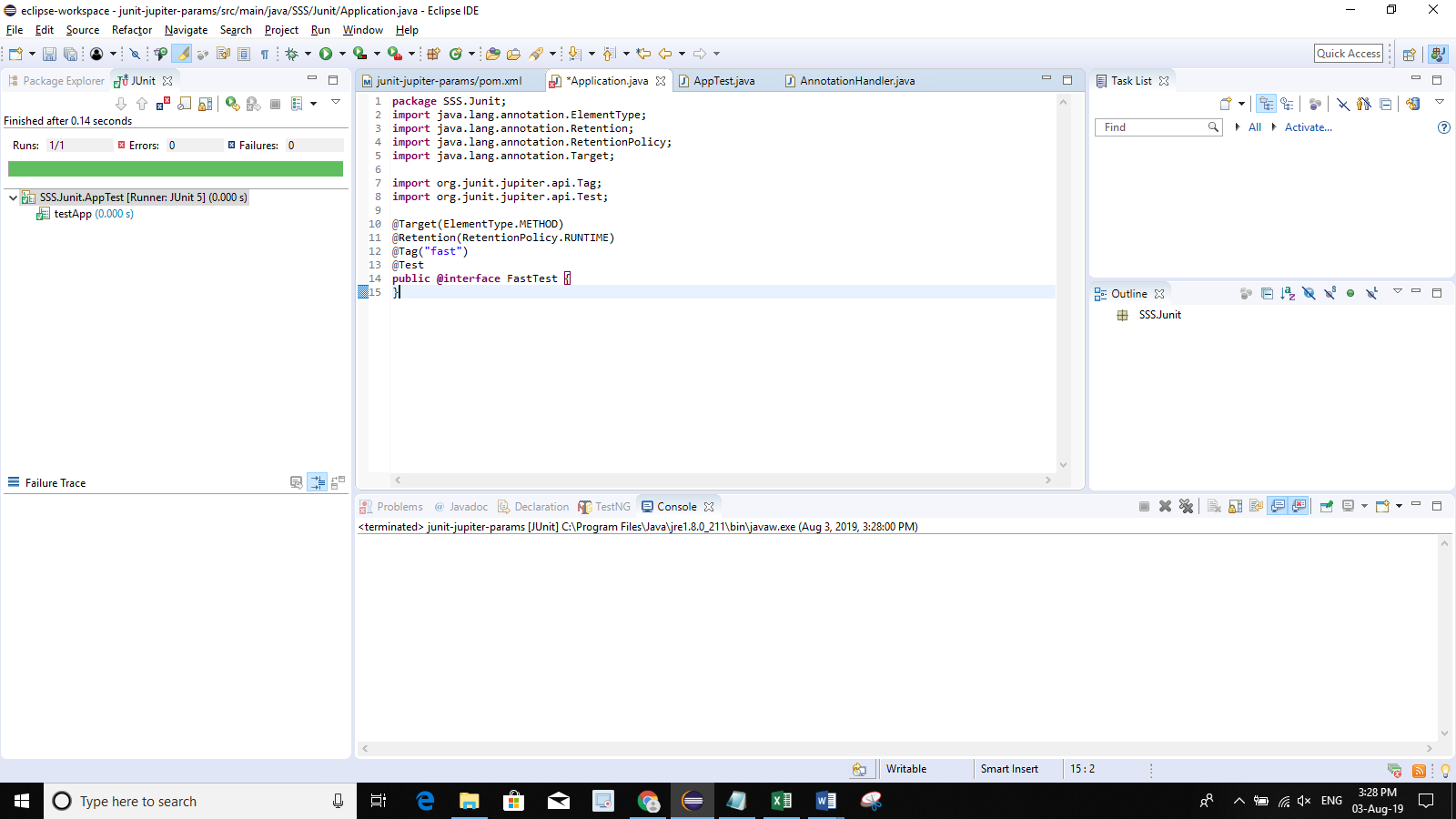
@Test

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() {

// ...

}

**Step 4.13.2:** Pushing the code to GitHub repositories

Open your command prompt and navigate to the folder where you have created your files.

cd <folder path>

Initialize your repository using the following command:

git init

Add all the files to your git repository using the following command:

git add .

Commit the changes using the following command:

git commit . -m “Changes have been committed.”

Push the files to the folder you initially created using the following command:

git push -u origin master