**Test-Driven Development**

**TDD** is an iterative development process. Each iteration starts with a set of tests written for a new piece of functionality. These tests are supposed to fail during the start of iteration as there will be no application code corresponding to the tests. In the next phase of the iteration, Application code is written with an intention to pass all the tests written earlier in the iteration. Once the application code is ready tests are run.

Any failures in the test run are marked and more Application code is written/re-factored to make these tests pass. Once application code is added/re-factored the tests are run again. This cycle keeps on happening until all the tests pass. Once all the tests pass we can be sure that all the features for which tests were written have been developed

The following sequence of steps is generally followed:

* Add a test
* Run all tests and see if the new one fails
* Write some code
* Run tests
* Refactor code
* Repeat

**Behavior Driven Development**

Behavior Driven testing is an extension of TDD. Like in TDD in BDD also we write tests first and the add application code. The major difference that we get to see here are

* Tests are written in plain descriptive English type grammar
* Tests are explained as behavior of application and are more user-focused
* Using examples to clarify requirements

This difference brings in the need to have a language that can define, in an understandable format.

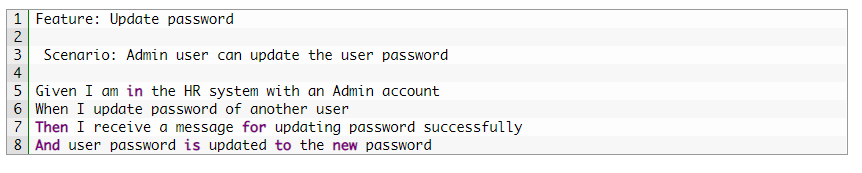
Features of BDD

1. Shifting from thinking in “tests” to thinking in “behavior”
2. Collaboration between Business stakeholders, Business Analysts, QA Team and developers
3. Ubiquitous language, it is easy to describe
4. Driven by Business Value
5. Extends Test-Driven Development (TDD) by utilizing natural language that non-technical stakeholders can understand
6. BDD frameworks such as Cucumber or JBehave are an enabler, acting a “bridge” between Business & Technical Language

### **What is Cucumber?**

***Cucumber****is a testing framework which supports****Behavior Driven Development (BDD).****It lets us define application behavior in plain meaningful English text using a simple grammar defined by a language called****Gherkin****. Cucumber itself is written in****Ruby****, but it can be used to “test” code written in*Ruby*or other languages including but not limited to*Java*,*C#*and*Python.

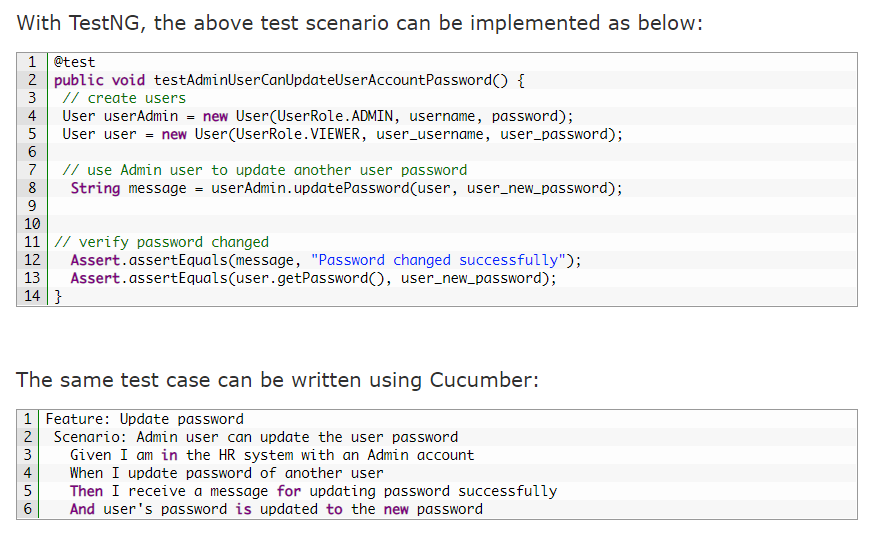
[**Cucumber**](http://toolsqa.com/cucumber/cucumber-tutorial/) is a tool used to run automated acceptance tests created in a BDD format. One of its most outstanding features of the tool is the ability to carry out plain-text functional descriptions (written in the language called [***Gherkin***](http://toolsqa.com/cucumber/gherkin/)) as automated tests. Let’s take a look at the below example:



The automation test with TestNG may be difficult for most manual testers and BAs to catch up with. Moreover, it is impossible to use this test again for AT. As a result, based on these flaws mentioned before, this can not be considered as a suitable method.

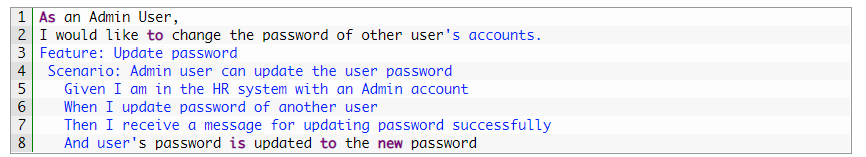
In contrast, the automation test using Cucumber is created in a business domain language or in natural language, which can be easily made out by all members of the software project team. Communication is crucial for any development team, especially in the Agile team. There are usually many continuous chats, discussions, or even arguments happening among developers and testers to figure out what the correct behavior of a feature is.

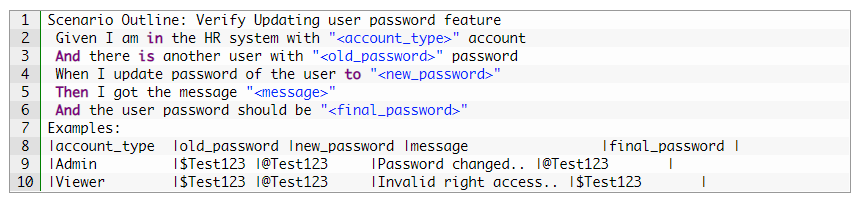
By using Cucumber, the same feature specification is now used for developing by developers, for testing by testers. It is considered to be a powerful tool because it can help lower the risk for misunderstanding as well as the communication breakdown.



## Cucumber is an Automated Acceptance Testing Tool

The acceptance test typically is carried out by BAs/customers to make sure that the development team has built specific features. Typical activity in this testing stage is verifying the system against the original requirements with specific, real data from production. Cucumber testing not only follows the requirements as its test scenarios but also helps BAs or Product Manager to adjust test data easily. Here is a demonstration with a little adjustment:

**

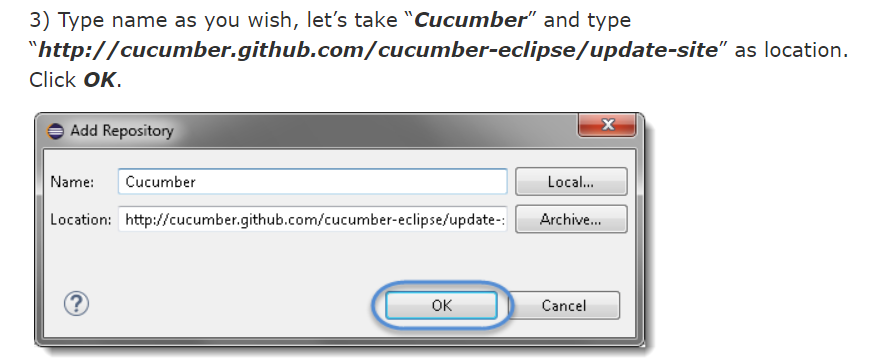
**

**Installation**

1.Install Java/Install Eclipse

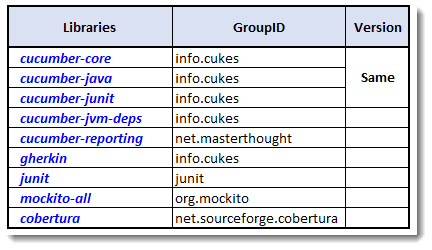
**2.Cucumber Eclipse Plugin** helps eclipse to understand the basic Gherkin syntax and it works like a syntax highlighter. It highlights all the main syntax in the feature file which makes it more readable and clear. It also enables the run of the feature file independently, without the help of JUnit.

Launch the Eclipse IDE and from Help menu, click “***Install New Software***”.



3. Do not forget to add all the dependencies for all the below mentioned jars required for Cucumber set up:cucumber-core

1. cucumber-java
2. cucumber-junit
3. cucumber-jvm-deps
4. cucumber-reporting
5. gherkin
6. junit
7. mockito-all
8. cobertura -to find whether the code lacks test its



info.cukes:cucumber-jvm:pom:1.2.5:compile

info.cukes:cucumber-junit:jar:1.2.5:test

info.cukes:cucumber-core:jar:1.2.5:test

info.cukes:cucumber-java:jar:1.2.5:test

info.cukes:cucumber-picocontainer:jar:1.2.5:compile

info.cukes:cucumber-testng:jar:1.2.5:compile

## Make sure you give the path right for Features. For features you need to specify the folder path and for step definition you can specify test.StepDefinition

## Junit was failing asking for Hamcrest class not found . Install Hamcrest jar for Junit

## While installing cucumber picocontainer was failing so removed cucumber pico container and installed just pico container from org.picocontainer and worked fine

## 

## Assert.assertEquals Depreciated

## You're using junit.framework.Assert instead of org.junit.Assert.

## What is Mocking?

Mocking is a way to test the functionality of a class in isolation. Mocking does not require a database connection or properties file read or file server read to test a functionality. Mock objects do the mocking of the real service. A mock object returns a dummy data corresponding to some dummy input passed to it.

## Mockito

Mockito facilitates creating mock objects seamlessly. It uses Java Reflection in order to create mock objects for a given interface. Mock objects are nothing but proxy for actual implementations.

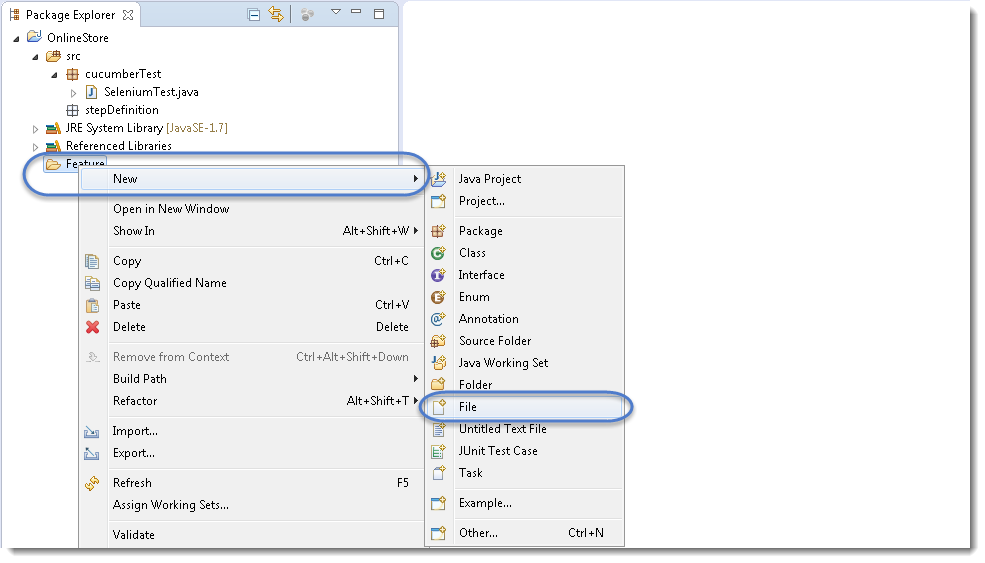
Consider a case of Stock Service which returns the price details of a stock. During development, the actual stock service cannot be used to get real-time data. So we need a dummy implementation of the stock service. Mockito can do the same very easily, as its name suggests.

Benefits of Mockito

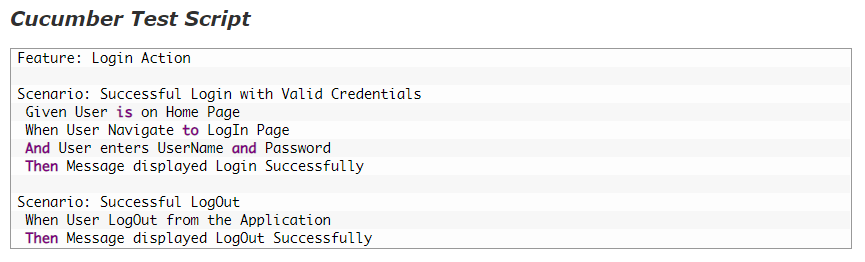
* **No Handwriting** − No need to write mock objects on your own.
* **Refactoring Safe** − Renaming interface method names or reordering parameters will not break the test code as Mocks are created at runtime.
* **Return value support** − Supports return values.
* **Exception support** − Supports exceptions.
* **Order check support** − Supports check on order of method calls.
* **Annotation support** − Supports creating mocks using annotation.

 4.Selenium WebDriver

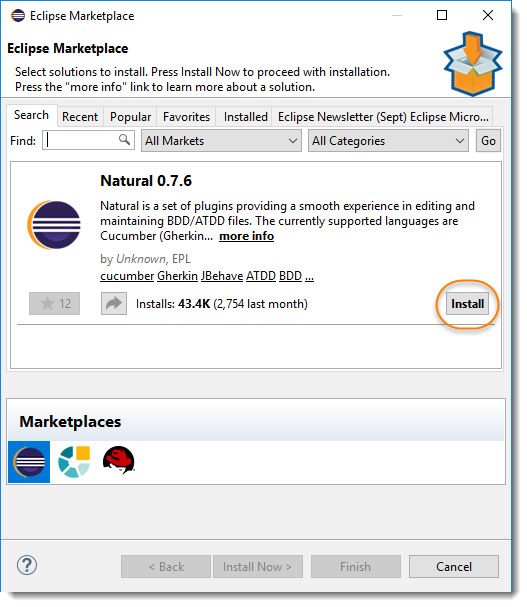
A ***Feature File*** is an entry point to the Cucumber tests. This is a file where you will describe your tests in Descriptive language (Like English). It is an essential part of Cucumber, as it serves as an automation test script as well as live documents. A feature file can contain a scenario or can contain many scenarios in a single feature file but it usually contains a list of scenarios. Let’s create one such file.

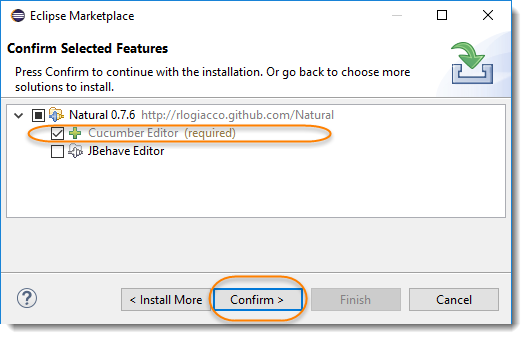






Install Natural plugin Editor

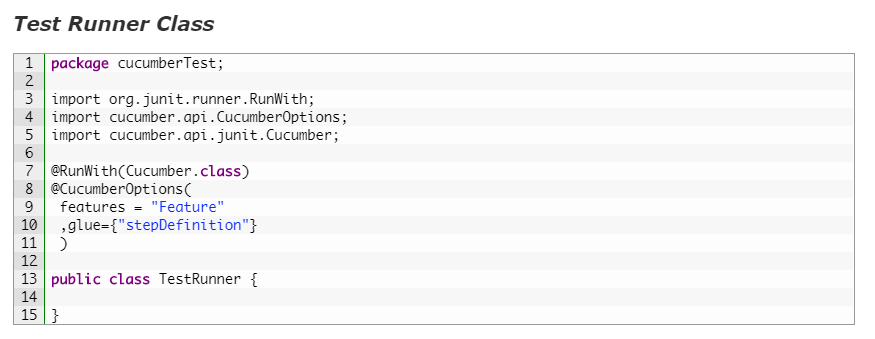




## JUnit Test Runner Class

As Cucumber uses Junit we need to have a ***Test Runner class***. This class will use the Junit annotation **@RunWith(),**which tells JUnit what is the test runner class. It more like a starting point for Junit to start executing your tests. In the src folder create a class called **TestRunner.**

Create a new ***Class*** file in the ‘***cucumberTest***‘ package and name it as ‘***TestRunner***‘



### Import Statements

First import statement ‘***org.junit.runner.RunWith***‘ imports  @RunWith annotation from the Junit class. @RunWith annotation tells JUnit that tests should run using ***Cucumber class*** present in ‘***Cucumber.api.junit***‘ package.

Second import statement ‘***cucumber.api.CucumberOptions***‘ imports the ***@CucumberOptions*** annotation. This annotation tells Cucumber a lot of things like where to look for feature files, what reporting system to use and some other things also. But as of now in the above test, we have just told it for the Feature file folder.

**Plugin:***plugin Option is used to specify different formatting options for the output reports. Various options that can be used as for-matters are:*  
**Note –***Format option is deprecated . Use Plugin in place of that.*

***A-******Pretty****:*Prints the *Gherkin* source with additional colors and stack traces for errors.

@RunWith(Cucumber.class)

@CucumberOptions(plugin = {"pretty"}, strict = false)

public class RunYoursTest

{

// This class will be empty

}

***B-******HTML****:*This will generate a HTML report at the location mentioned in the for-matter itself.

@RunWith(Cucumber.class)

@CucumberOptions(plugin ={"pretty" , "html:Folder\_Name"})

public class RunYoursTest

{

// This class will be empty

}

**C-** **JSON**: This report contains all the information from the gherkin source in JSON Format. This report is meant to be post-processed into another visual format by 3rd party tools such as Cucumber Jenkins.

@RunWith(Cucumber.class)

@CucumberOptions(plugin = {"pretty" ,

"json:Folder\_Name/cucumber.json"})

public class RunYoursTest

{

// This class will be empty

*}*

***D-******JUnit****:* This report generates XML files just like Apache Ant’s JUnit report task. This XML format is understood by most Continuous Integration servers, who will use it to generate visual reports.

@RunWith(Cucumber.class)

@CucumberOptions(plugin = {"pretty" ,

"junit:Folder\_Name/cucumber.xml"})

public class RunYoursTest

{

// This class will be empty

}

**We can also use these option together.**

@RunWith(Cucumber.class)

@CucumberOptions(plugin = {"pretty" ,"html:Folder\_Name" ,

"json:Folder\_Name/cucumber.json" ,

"junit:Folder\_Name/cucumber.xml"})

public class RunYoursTest

{

// This class will be empty

}

**DryRun:** This option can either set as true or false (default value is false). If it is set as true, it means that Cucumber will only checks that every Step mentioned in the Feature File have corresponding code written in Step Definition file or not. So in case any of the function is missed in the Step Definition for any Step in Feature File, it will give us the message. So If you writing scenarios first and then implementing step definitions then add dryRun = true.

@RunWith(Cucumber.class)

@CucumberOptions(plugin = {"pretty"}, dryRun = true)

public class RunYoursTest

{

// This class will be empty

}

**Strict:** if strict option is set to false then at execution time if cucumber encounters any undefined/pending steps then cucumber does not fail the execution and undefined steps are skipped and BUILD is SUCCESSFUL.

@RunWith(Cucumber.class)

@CucumberOptions(plugin = {"pretty"}, strict = false)

public class RunYoursTest

{

// This class will be empty

}

and if Strict option is set to true then at execution time if cucumber encounters any undefined/pending steps then cucumber does fails the execution and undefined steps are marked as fail and BUILD is FAILURE. This is what the Console output looks like:

@RunWith(Cucumber.class)

@CucumberOptions(plugin = {"pretty"}, strict = true)

public class RunYoursTest

{

// This class will be empty

}

**Monochrome**: This option can either set as true or false (default value is false). If it is set as true, it means that the console output for the Cucumber test are much more readable. And if it is set as false, then the console output is not as readable as it should be. For practice just add the code ‘monochrome = true‘ in TestRunner class.

@RunWith(Cucumber.class)

@CucumberOptions(plugin = {"pretty"}, strict = true,

monochrome = true)

public class RunYoursTest

{

// This class will be empty

}

**Features:** Features Options helps Cucumber to locate the Feature file in the project folder structure .All we need to do is to specify the folder path and Cucumber will automatically find all the ‘.features‘ extension files in the folder.  
It can be defined like:

@CucumberOptions(

features = “src/test/features“

)

**Glue:**It is almost the same think as Features Option but the only difference is that it helps Cucumber to locate theStep Definition file. Whenever Cucumber encounters a Step, it looks for a Step Definition inside all the files present in the folder mentioned in Glue Option.  
It can be defined like-

@RunWith(Cucumber.class)

@CucumberOptions(

features = “Feature“

glue = “stepDefinition“

)

Or

@CucumberOptions(

features = “src/test/features“

glue = “src/test/stepDeinition“

)

Cucumber can exclude scenarios with a particular tag by inserting the tilde character before the tag.  
For the following command will run all Scenarios without the SoapUI tag.

@RunWith(Cucumber.class)

@CucumberOptions(

features = “src/test/features“,

tags ={“~@SoapUI“},... )

**Logical OR**

Separate a list of tags by commas for a Logical OR tag expression.

**@CucumberOptions(**

**features = “src/test/features“,**

**tags ={“@SoapUI,@Functional"},... )**

**Logical AND**

Specifying multiple tag arguments creates a logical AND between each tag expression.

@RunWith(Cucumber.class)

@CucumberOptions(

features = “src/test/features“,

tags ={“@SoapUI","@Functional"},... )

**AND OR NOT**

Specifying multiple tag arguments creates a logical AND between each tag expression.

**@CucumberOptions(**

**features = “src/test/features“,**

**tags ={“@SoapUI,@Functional","~@Regression"},... )**

Even from the IDE, there are a couple of ways to run these feature files.

* *Click on the****Run****button on eclipse and you have your test run*
* *Right Click on****TestRunner****class and Click****Run As****>****JUnit Test Application***
* ***Precondition****to the test, which represent with (****Given****) keyword*
* ***Test step****execution, which represent with (****When****) keyword*
* ***Verification****of the output with expected result, which represent with (****Then****)*

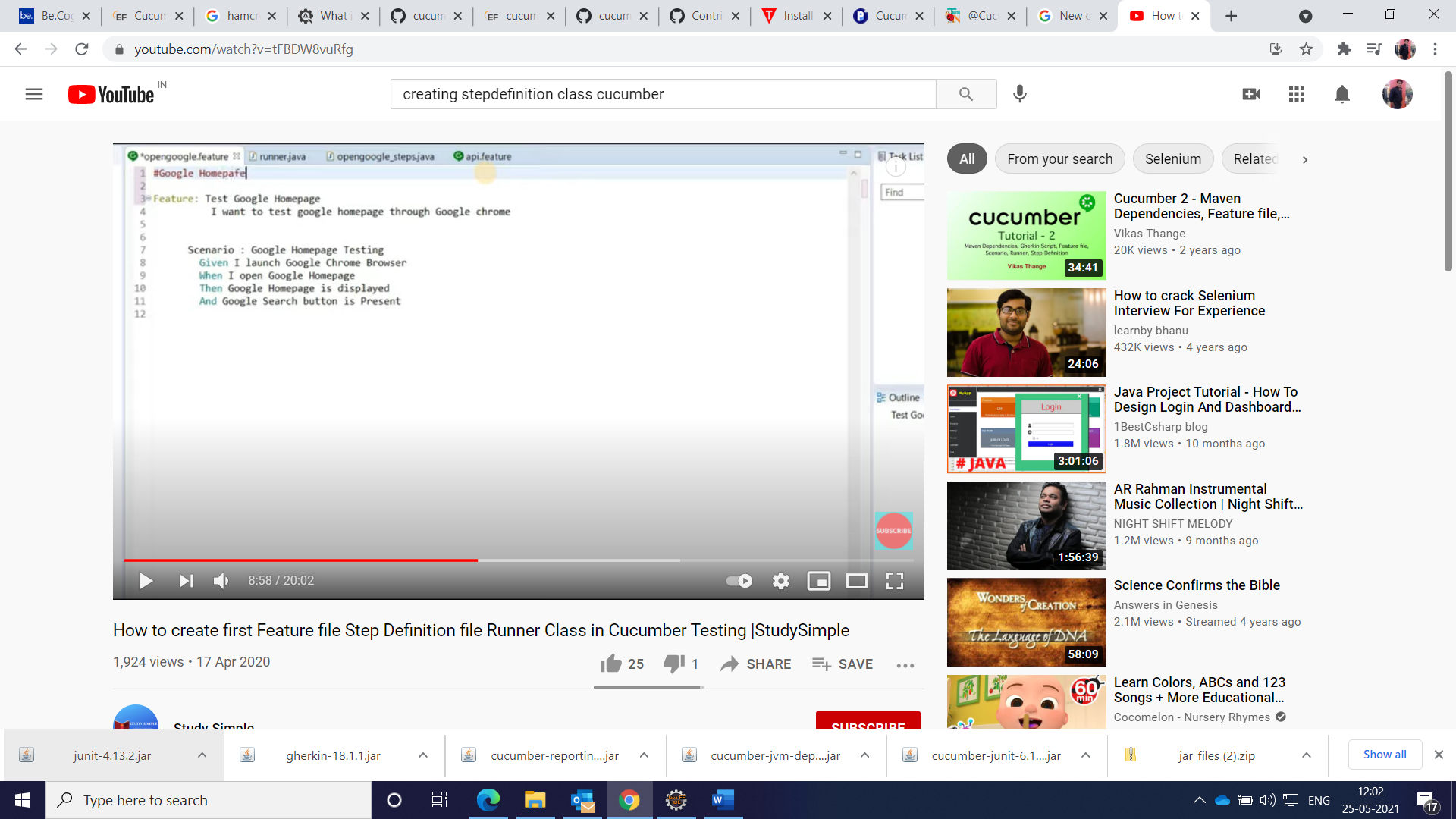
***And*** keyword is used to add conditions to your steps. Let’s look at it by modifying our example a little

***Scenario***: Successful Login with Valid Credentials  
***Given***User is on Home Page  
***And***LogIn Link displayed  
***When***User Navigate to LogIn Page  
***And***User enters UserName and Password  
***Then***Message displayed Login Successfully  
***And***LogOut Link displayed

## But Keyword

***But***keyword is used to add negative type comments. It is not a hard & fast rule to use but only for negative conditions. It makes sense to use But when you will try to add a condition which is opposite to the premise your test is trying to set. Take a look at the example below:

***Scenario***: Unsuccessful Login with InValid Credentials  
***Given***User is on Home Page  
***When***User Navigate to LogIn Page  
***And***User enters UserName and Password  
***But***The user credentials are wrong  
***Then***Message displayed Wrong UserName & Password



### Given

Given steps are used to describe the initial context of the system - the scene of the scenario. It is typically something that happened in the past.

When Cucumber executes a Given step, it will configure the system to be in a well-defined state, such as creating and configuring objects or adding data to a test database.

The purpose of Given steps is to **put the system in a known state** before the user (or external system) starts interacting with the system (in the When steps). Avoid talking about user interaction in Given’s. If you were creating use cases, Given’s would be your preconditions.

It’s okay to have several Given steps (use And or But for number 2 and upwards to make it more readable).

Examples:

* Mickey and Minnie have started a game
* I am logged in
* Joe has a balance of £42

### When

When steps are used to describe an event, or an action. This can be a person interacting with the system, or it can be an event triggered by another system.

It’s strongly recommended you only have a single When step per Scenario. If you feel compelled to add more, it’s usually a sign that you should split the scenario up into multiple scenarios.

Examples:

* Guess a word
* Invite a friend
* Withdraw money

### Then

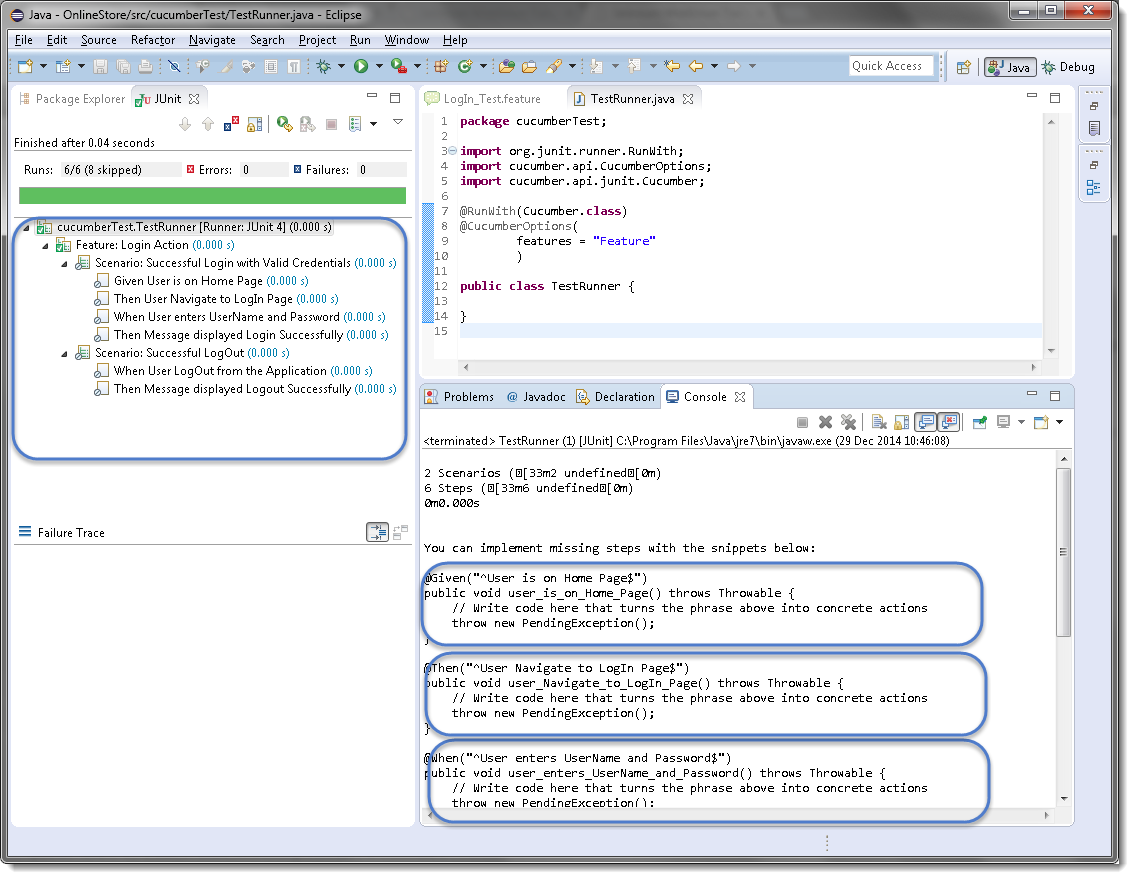
Then steps are used to describe an expected outcome, or result.

The [step definition](https://cucumber.io/docs/cucumber/step-definitions) of a Then step should use an assertion to compare the actual outcome (what the system actually does) to the expected outcome (what the step says the system is supposed to do).

An outcome should be on an **observable** output. That is, something that comes out of the system (report, user interface, message), and not a behaviour deeply buried inside the system (like a record in a database).

Examples:

* See that the guessed word was wrong
* Receive an invitation
* Card should be swallowed



## \* Keyword

This keyword is very special. This keyword defies the whole purpose of having Given, When, Then and all the other keywords. Basically Cucumber doesn’t care about what Keyword you use to define test steps, all it cares about what code it needs to execute for each step. That code is called a ***step definition*** and we will discuss about it in the next section. At this time just remember that all the keywords can be replaced by the***\* keyword*** and your test will just work fine. Let’s see with example, we had this test earlier:

***Using \* Keyword***

***Feature***: LogIn Action Test  
Description: This feature will test a LogIn and LogOut functionality

***Scenario***: Successful Login with Valid Credentials  
***\****User is on Home Page  
***\****User Navigate to LogIn Page  
***\****User enters UserName and Password  
***\****Message displayed Login Successfully

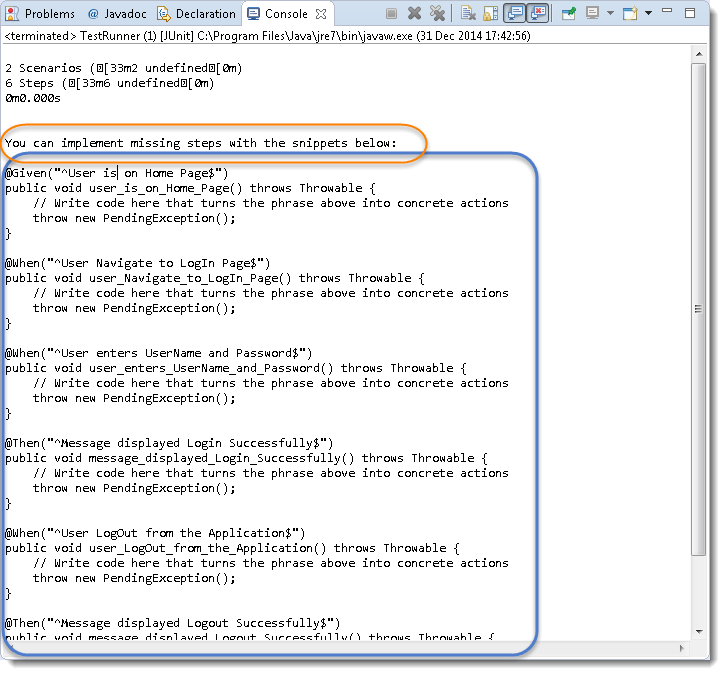
## What is Step Definition?

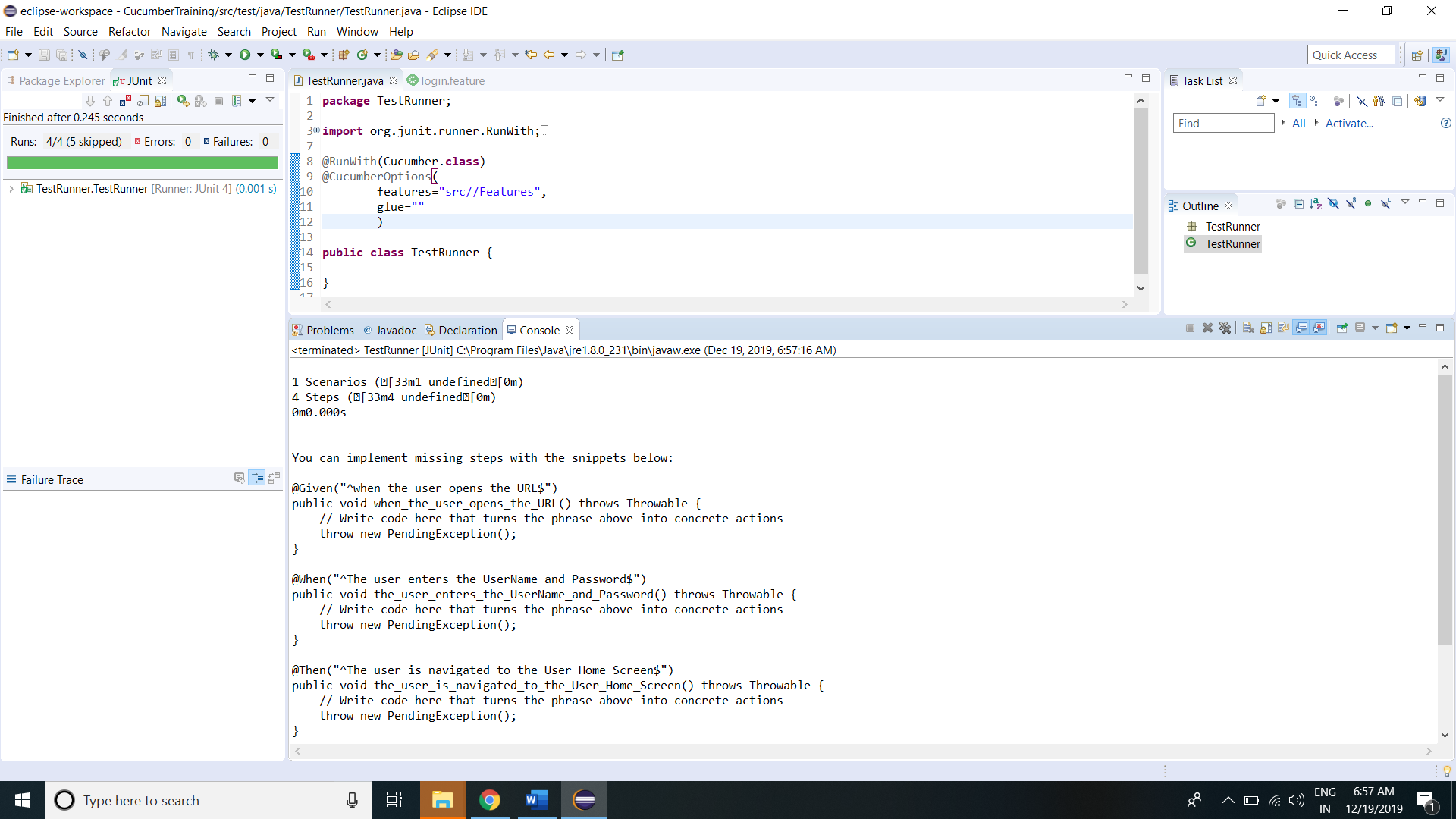
A Step Definition is a small piece of code with a pattern attached to it or in other words a Step Definition is a java method in a class with an annotation above it. An annotation followed by the pattern is used to link the Step Definition to all the matching Steps, and the code is what Cucumber will execute when it sees a Gherkin Step. Cucumber finds the Step Definition file with the help of the Glue code in ***Cucumber Options***. We will cover different Cucumber Options in the next chapter.

## Add a Step Definition file

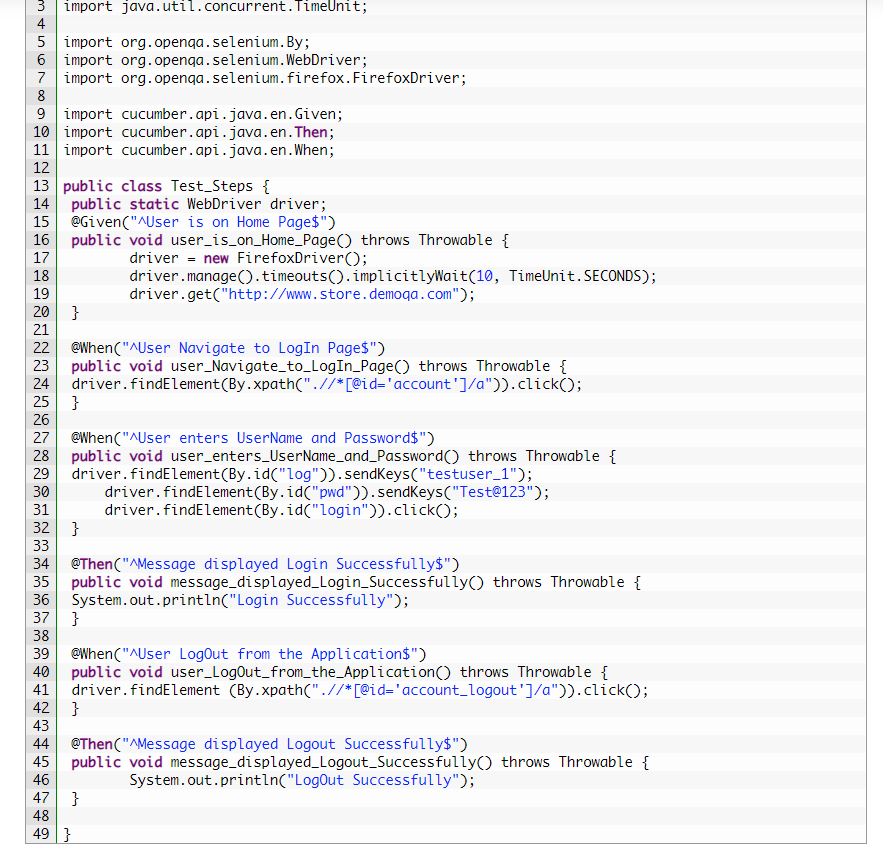
1) Create a new ***Class*** file in the ‘***stepDefinition***‘ package and name it as ‘***Test\_Steps***‘, by right click on the Package and select New > Class. Do not check the option for ‘***public static void main***‘ and click on ***Finish*** button.

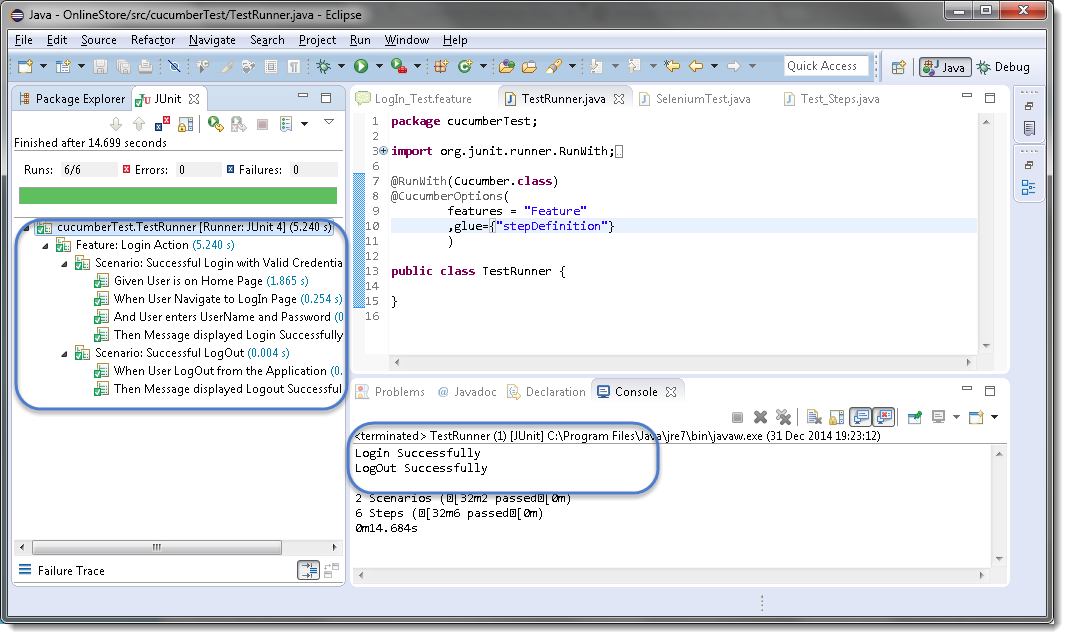
2) Take a look at the message in the console window. This message was displayed, when we ran the ***Test\_Runner*** class.

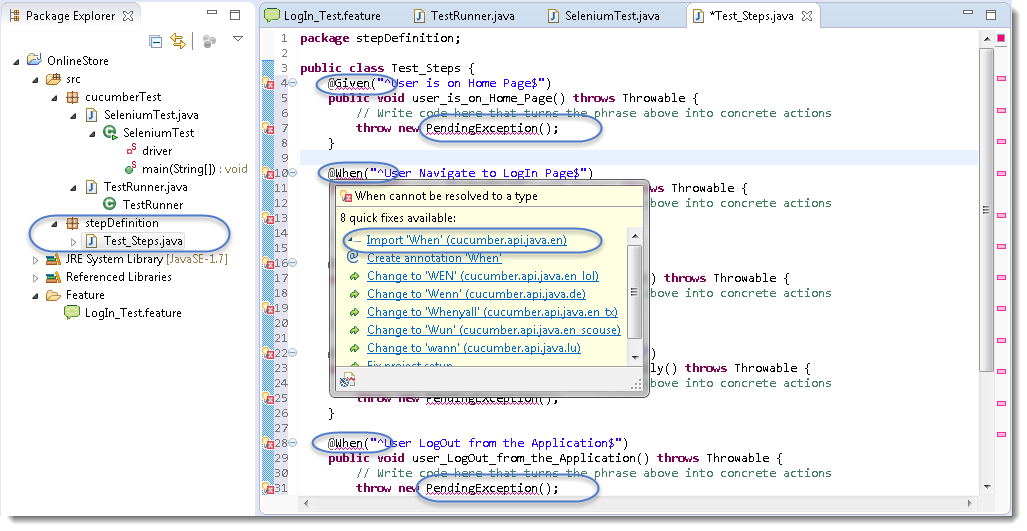




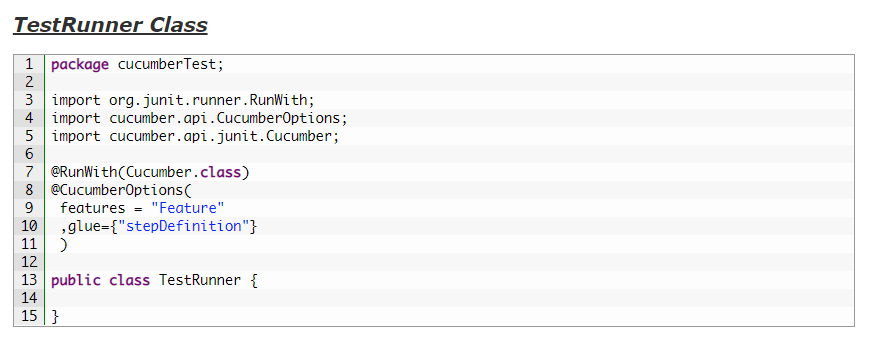
3) Notice, the eclipse console window says ‘***You can implement missing steps with the snippets below:***‘. It is very easy to implement all the steps, all you need to do is to copy the complete text marked in a blue box and paste it into the above created ***Test\_Steps*** class.

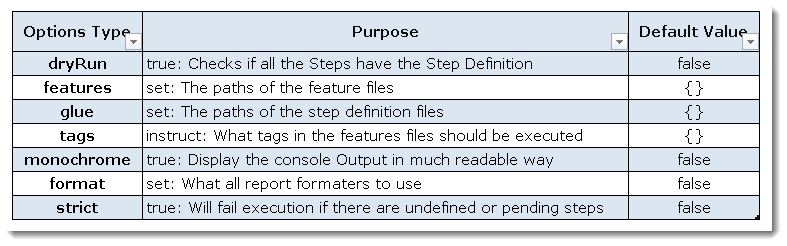






## What is Cucumber Options?

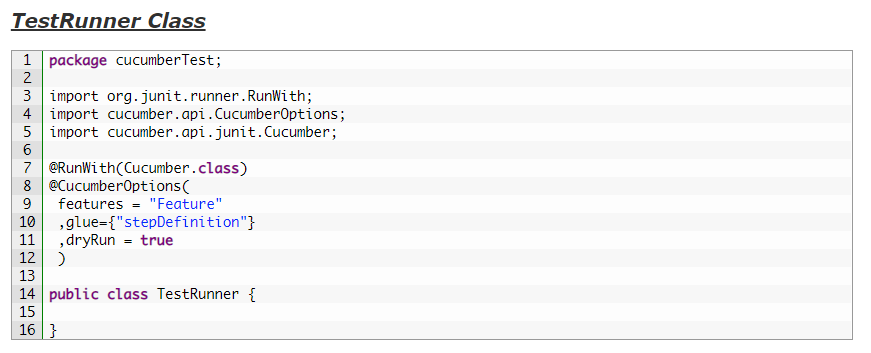


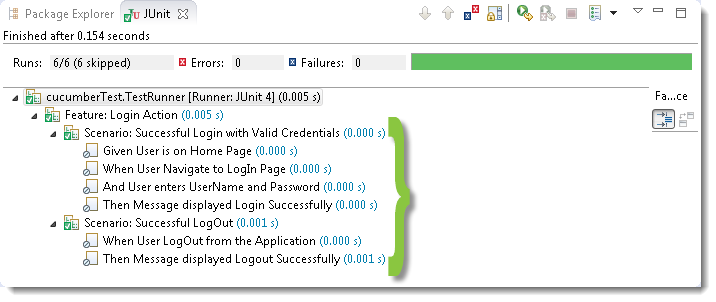


### ***Dry Run***

***dryRun*** option can either set as ***true*** or ***false***. If it is set as true, it means that Cucumber will only check that every Step mentioned in the Feature File has corresponding code written in Step Definition file or not. So in case any of the functions are missed in the Step Definition for any Step in Feature File, it will give us the message. For practice just add the code ‘***dryRun = true***‘ in ***TestRunner*** class:

Now give it a run by Right Click on ***TestRunner*** class and Click ***Run As*** > ***JUnit Test.***Cucumber will run the script and the result will be shown in the left-hand side project explorer window in JUnit tab.





### ***Monochrome***

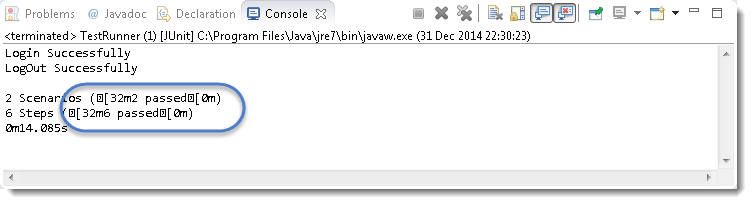
This option can either set as ***true*** or ***false***. If it is set as true, it means that the console output for the Cucumber test are much more readable. And if it is set as false, then the console output is not as readable as it should be. For practice just add the code ‘***monochrome = true***‘ in ***TestRunner*** class:



Now give it a run by Right Click on ***TestRunner*** class and Click ***Run As***  > ***JUnit Test.***Cucumber will run the script and Console Output will display like this:



This time change the value from true to false and run the ***TestRunner*** class again. This time the Console Output will look like this:



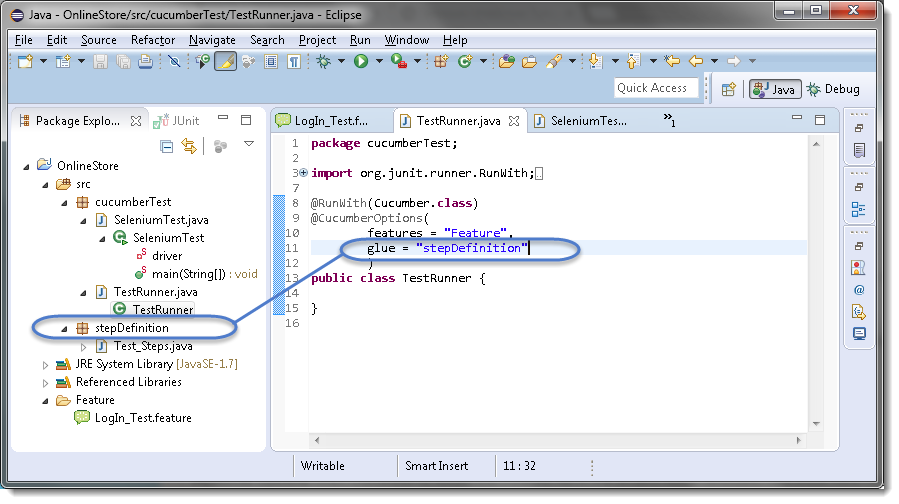
### ***Glue***

It is almost the same think as Features Option but the only difference is that it helps Cucumber to locate the ***Step Definition file.*** Whenever Cucumber encounters a Step, it looks for a Step Definition inside all the files present in the folder mentioned in **Glue Option**. It can be specified like:

***glue = “stepDefinition“***

Or if the Step Definition file is in the deep folder structure

***glue = “src/test/stepDeinition“***



### ***Format***

**Format Option** is used to specify different formatting options for the output reports. Various options that can be used as for-matters are:

***Pretty:***Prints the Gherkin source with additional colors and stack traces for errors. Use below code:

***format = {“pretty“}***

***HTML:***This will generate a HTML report at the location mentioned in the for-matter itself. Use below code:

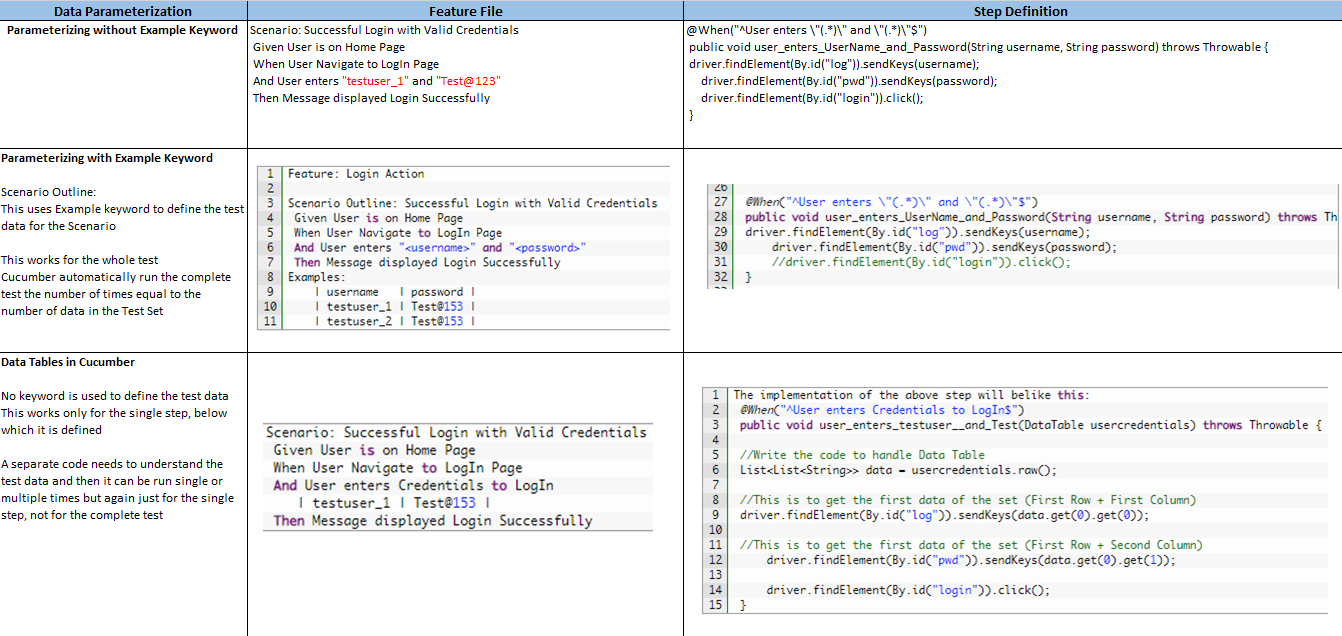
***format = {“html:Folder\_Name“}***

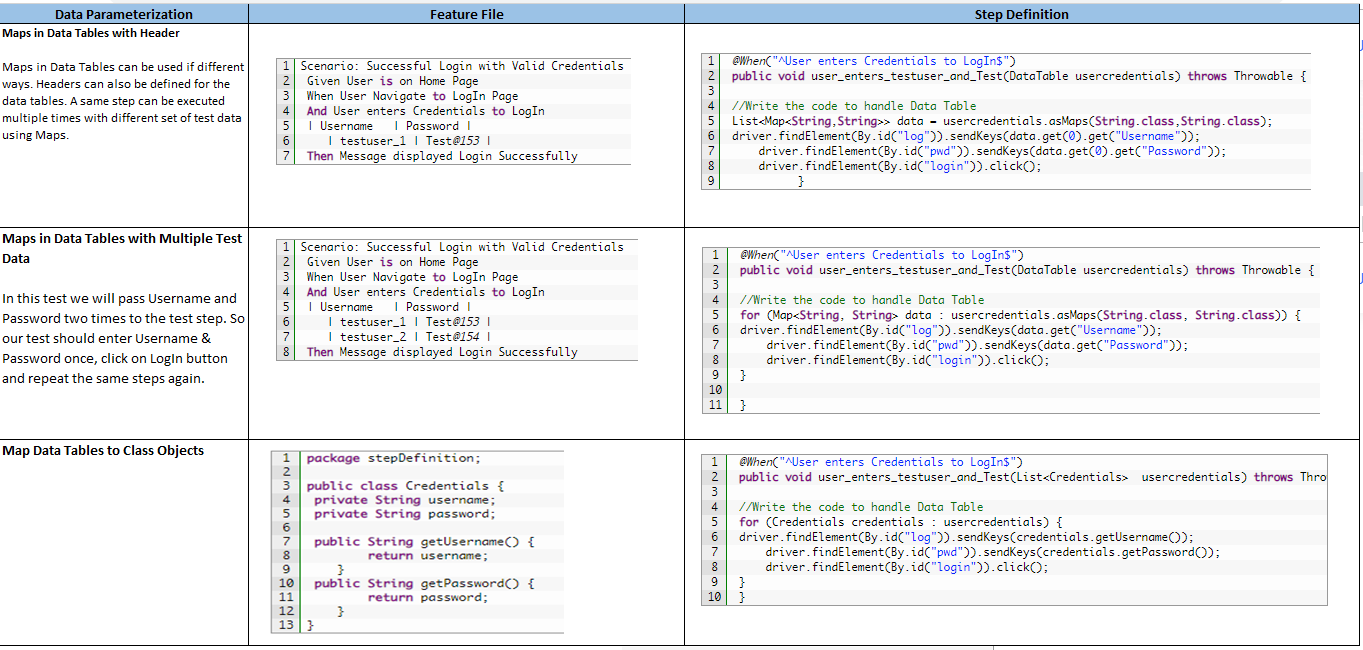
***JSON:***This report contains all the information from the gherkin source in JSON Format. This report is meant to be post-processed into another visual format by 3rd party tools such as Cucumber Jenkins. Use the below code:

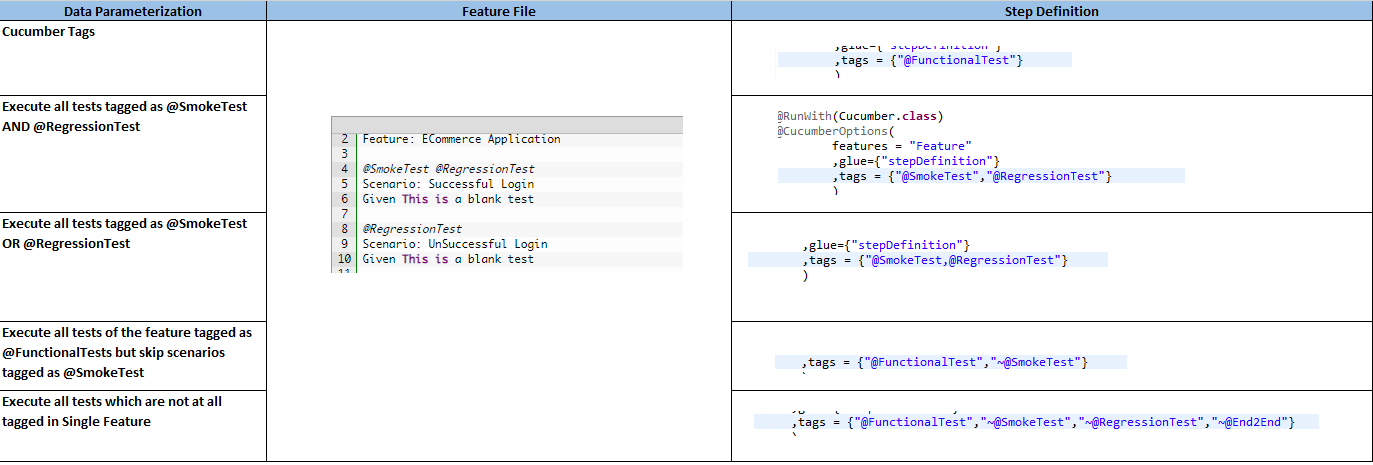
***format = {“json:Folder\_Name/cucumber.json“}***

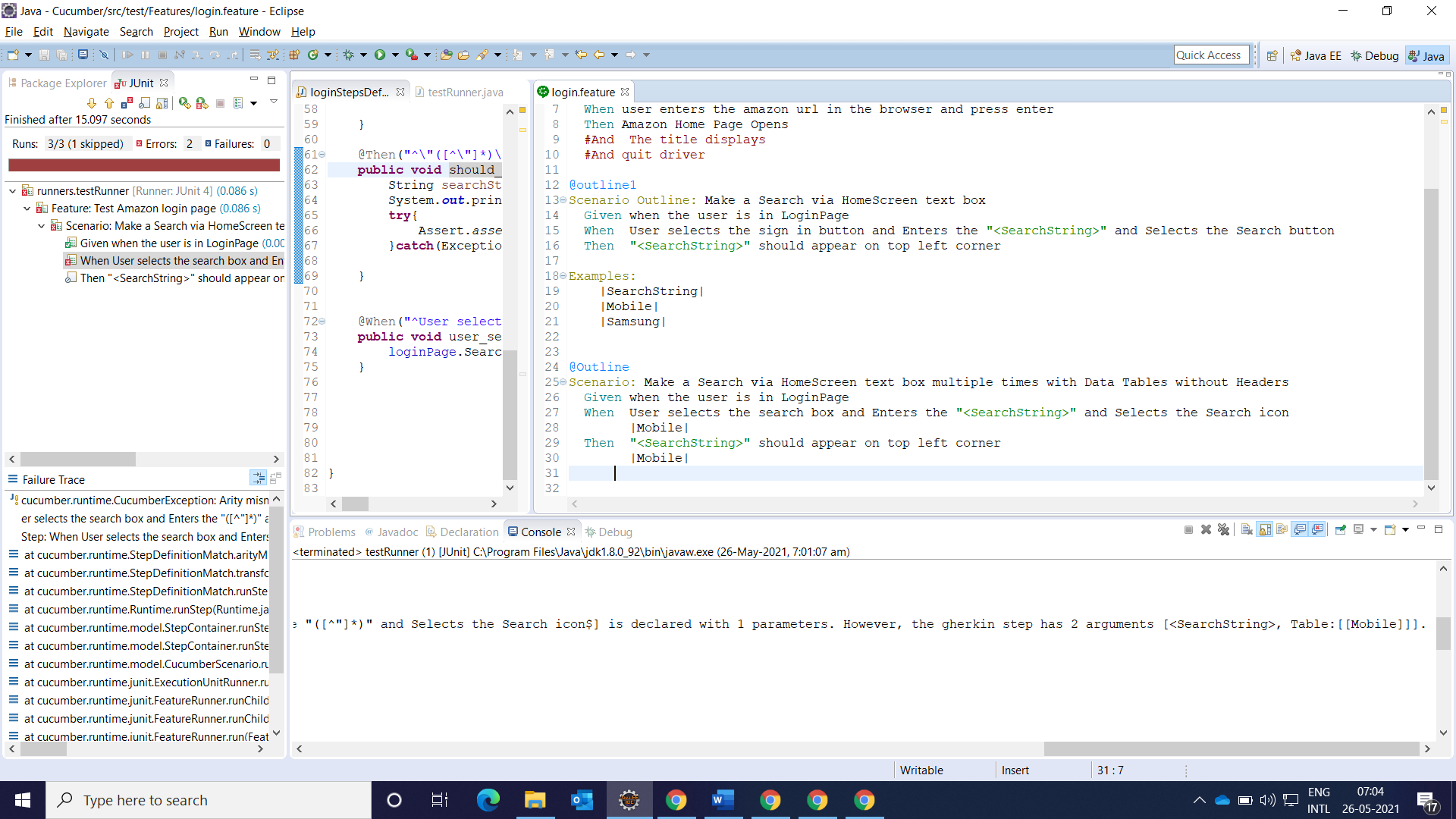
***JUnit:*** This report generates XML files just like Apache Ant’s JUnit report task. This XML format is understood by most Continuous Integration servers, who will use it to generate visual reports. use the below code:

***format = { “junit:Folder\_Name/cucumber.xml“}***

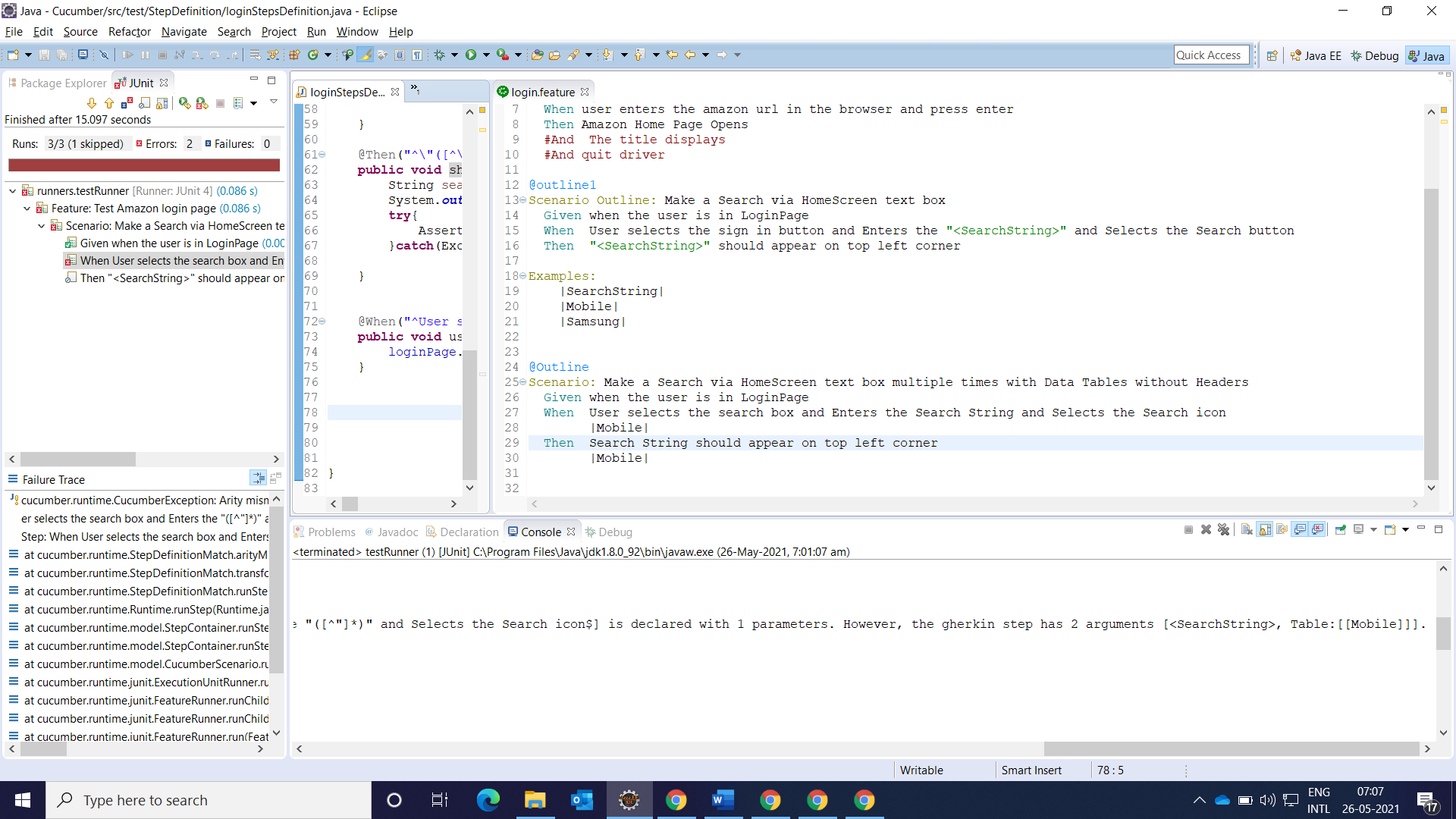




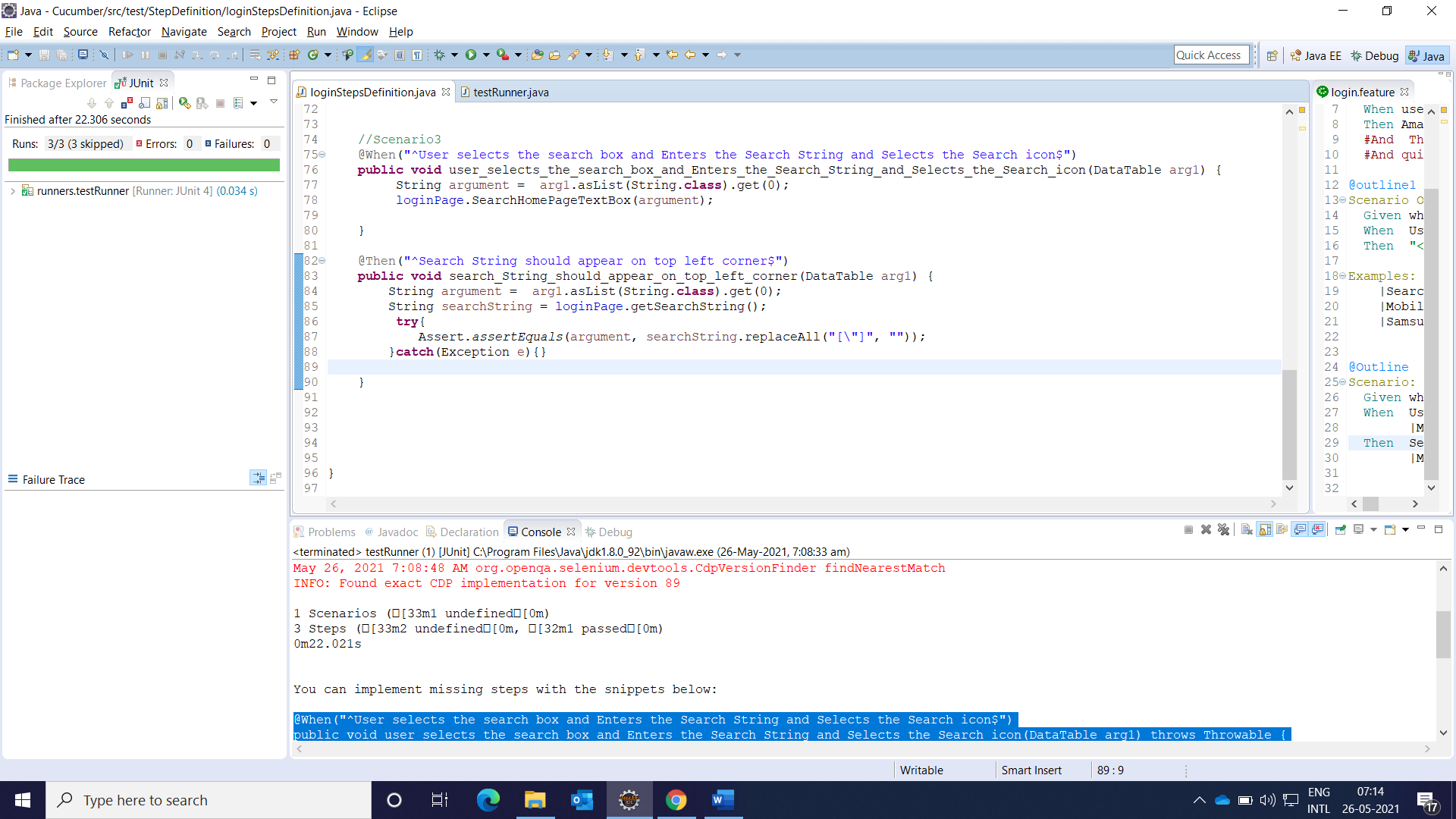




In the above when step, <Search String> defines an example argument and |Mobile| defines Data Table argument so, two argumemts. When you use data tables do not give string<> in angle quotes

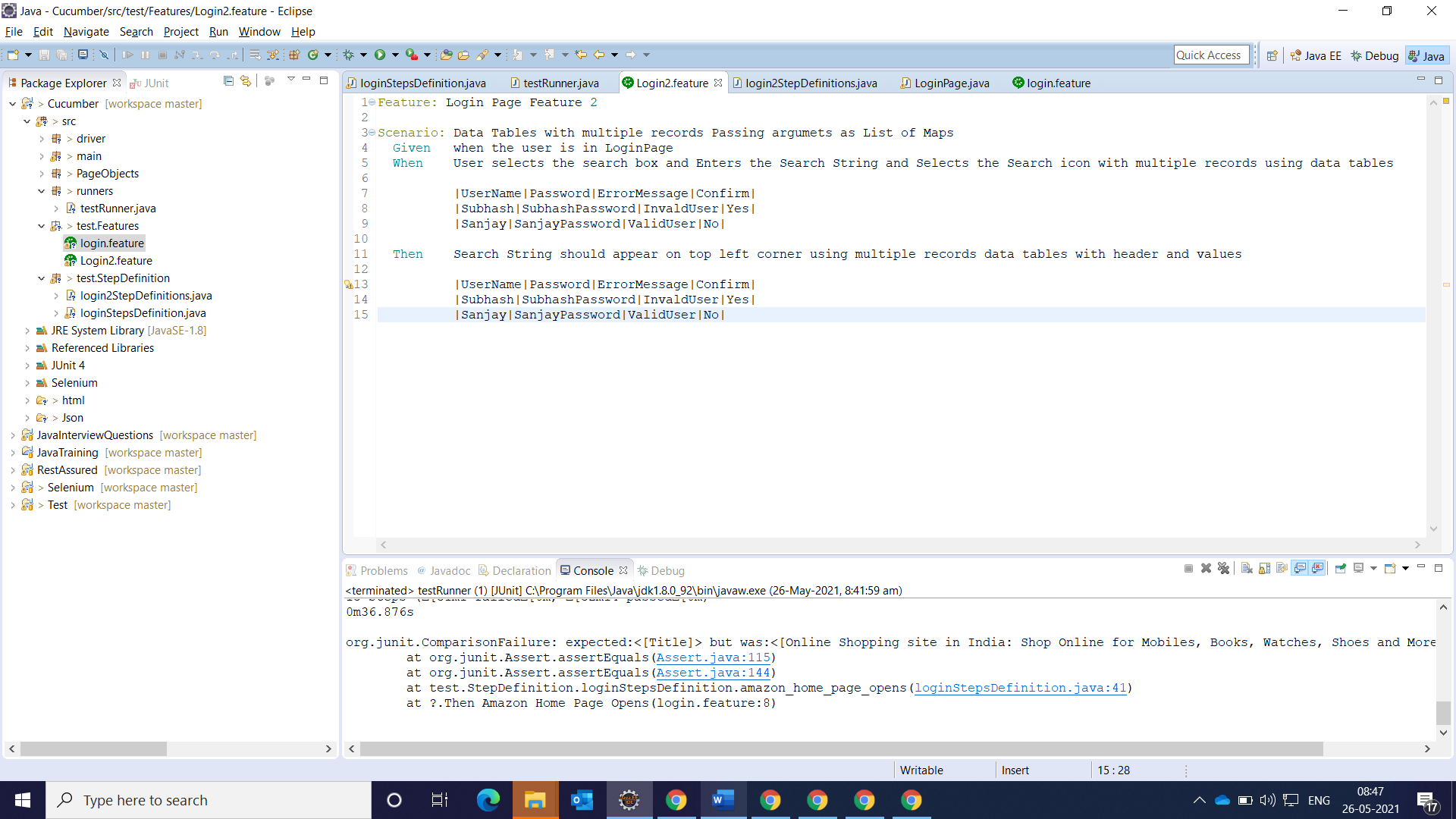


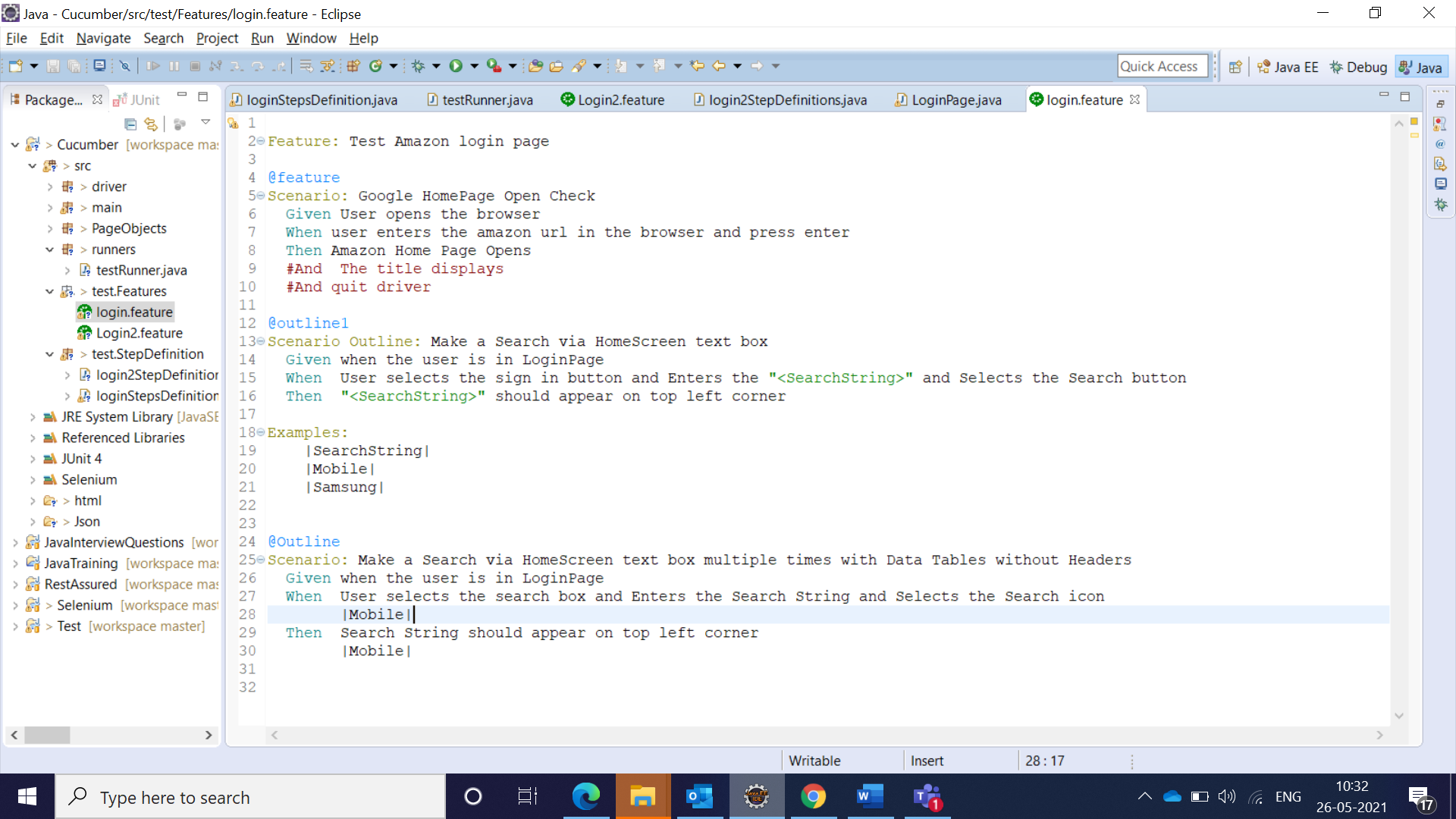
DataTable as Argument

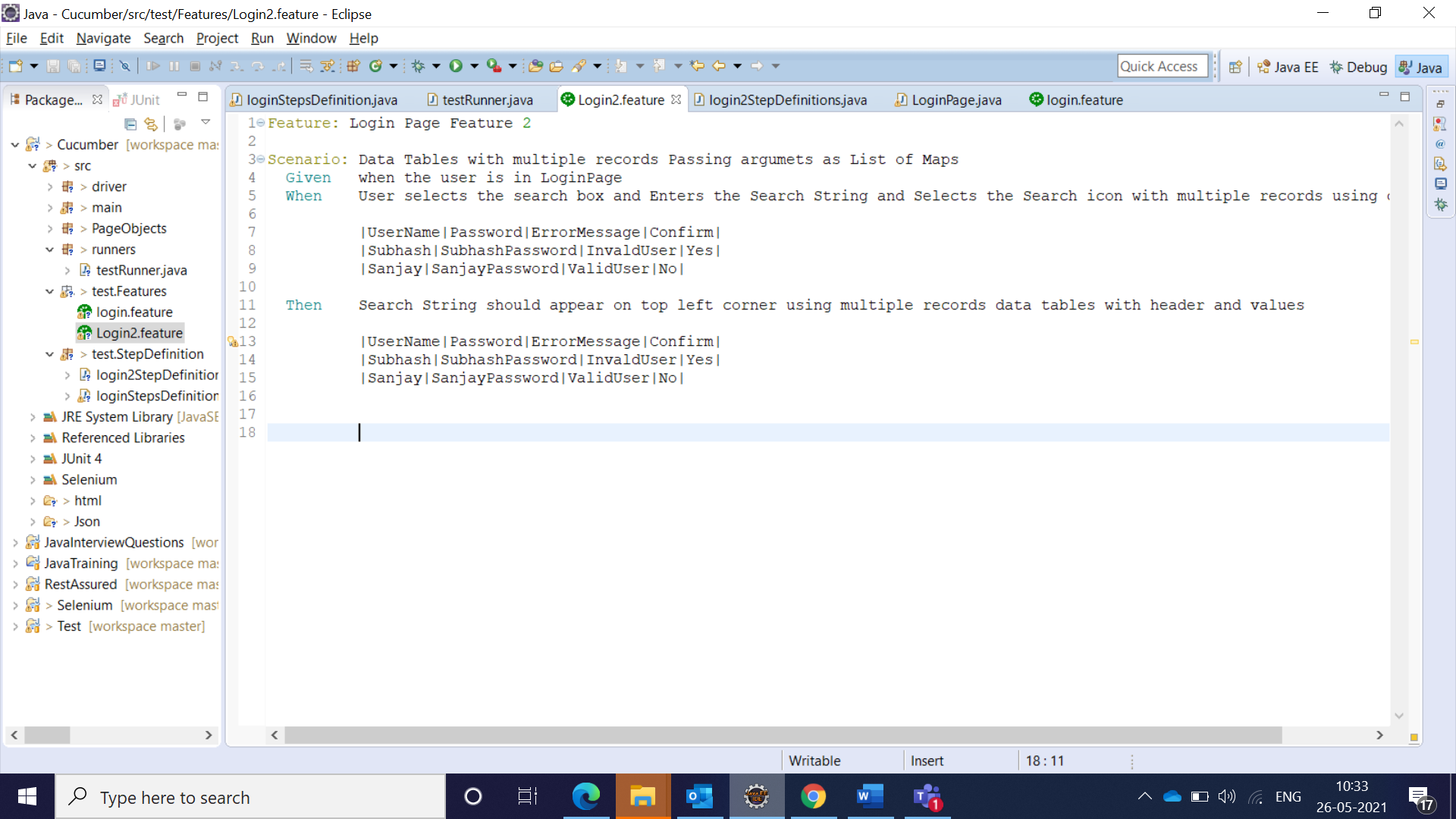


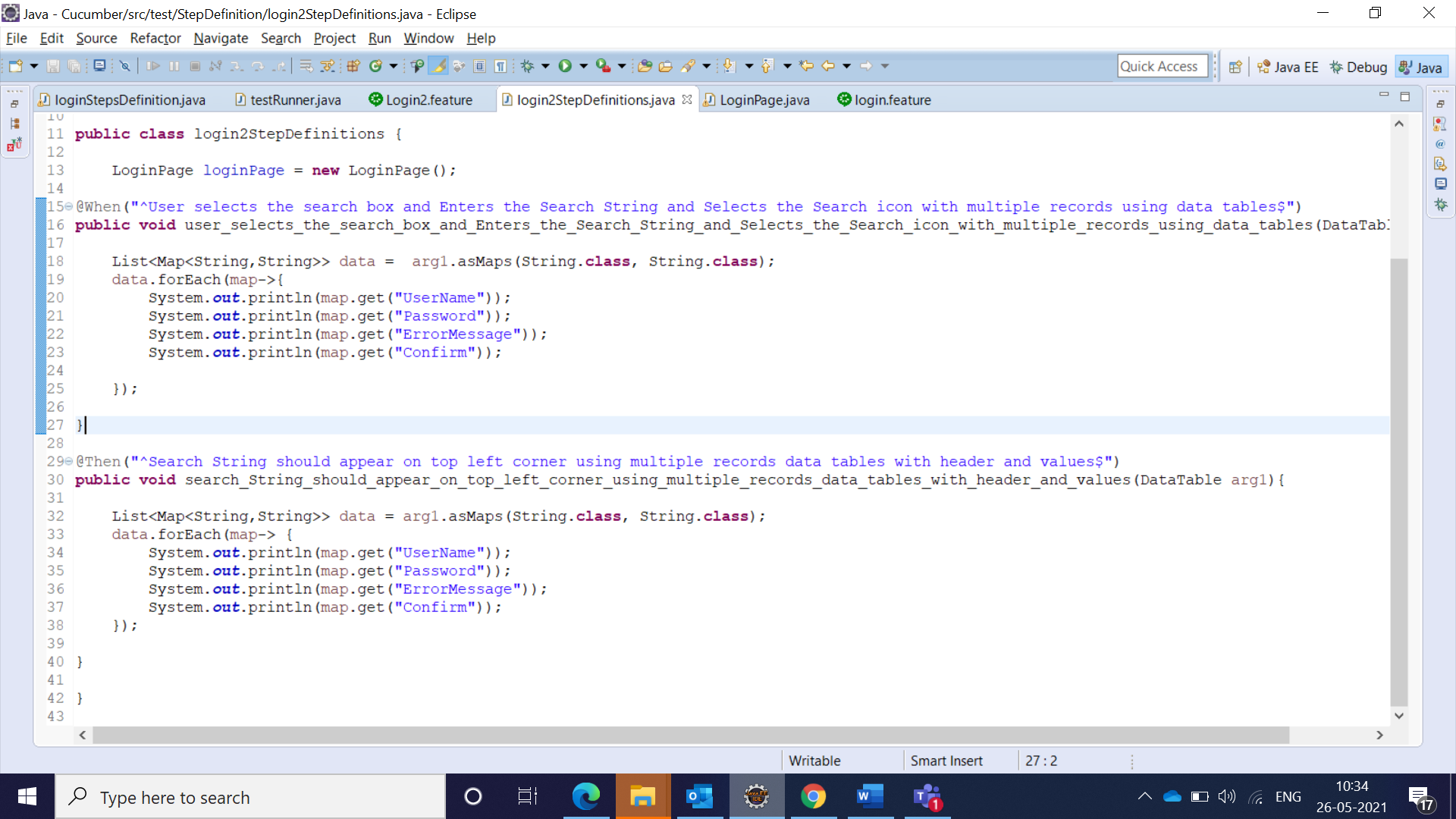
Data Tabes with multiple arguments and multiple records with headers. The data table would be a list of Maps

List<Map<String,String>> data = arg1.asMaps(String.class,String.class);









**Cucumber Hooks**

Cucumber supports ***hooks***, which are blocks of code that run ***before*** or ***after*** each scenario. You can define them anywhere in your project or step definition layers, using the methods ***@Before*** and ***@After***.

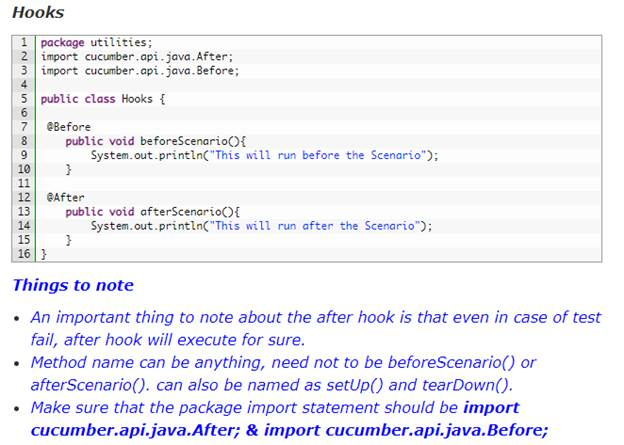
Hooks can be used to perform background tasks that are not part of business functionality. Such tasks could be:

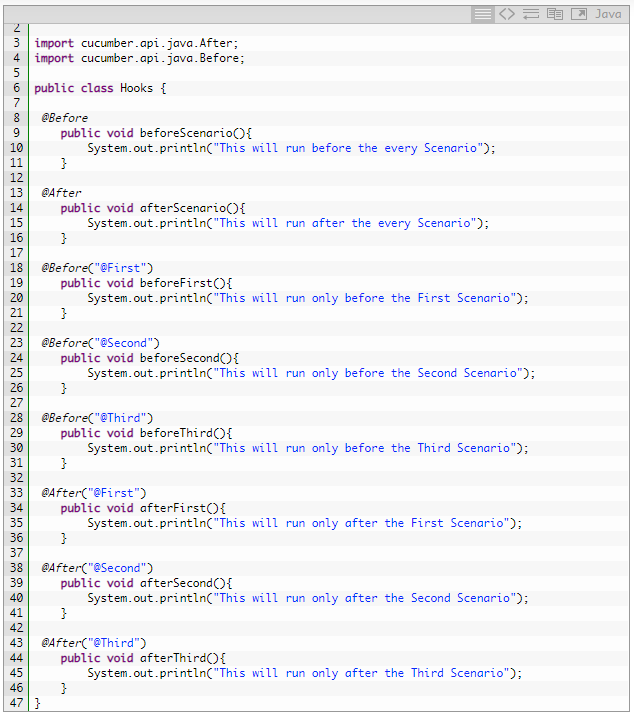
* Starting up a browser
* Setting or clearing cookies
* Connecting to a database
* Checking the state of the system
* Monitoring

A use case for monitoring would be to update a dashboard with the test progress in real-time.

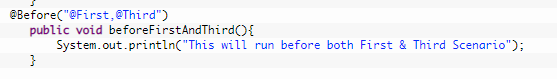
Create a hooksStepDefinitionClass in the StepDefinition Folder. Cucumber automatically runs the before and after each scenario



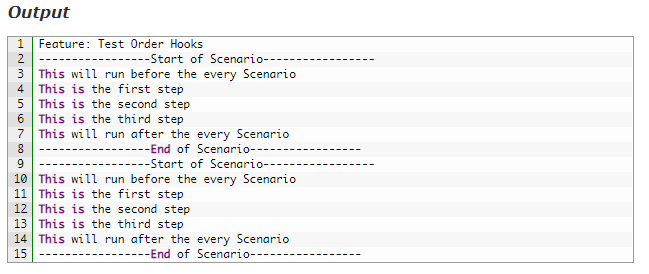




### **Common Tagged Hooks for Multiple Scenarios**

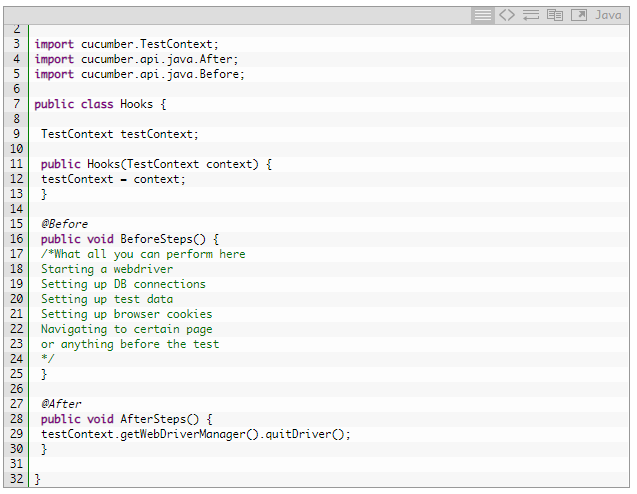




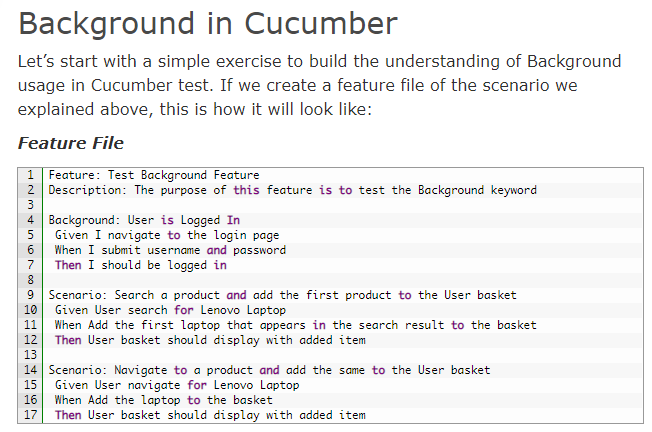


# **How to use Hooks in Selenium Cucumber Framework**

1. Create a ***New Class*** file and name it as ***Hooks***by right click on the ***stepDefinitions*** package select ***New >> Class***.



***Background in Cucumber***is used to define a step or series of steps that are common to all the tests in the feature file. It allows you to add some context to the scenarios for a feature where it is defined



### **Background with Hooks**

This is so interesting to see the working of Background with Hooks. The background is run before each of your scenarios but after any of your[**@Before hook**](http://toolsqa.com/cucumber/cucumber-hooks/).

# **Handle Ajax call Using JavaScriptExecutor in Selenium?**

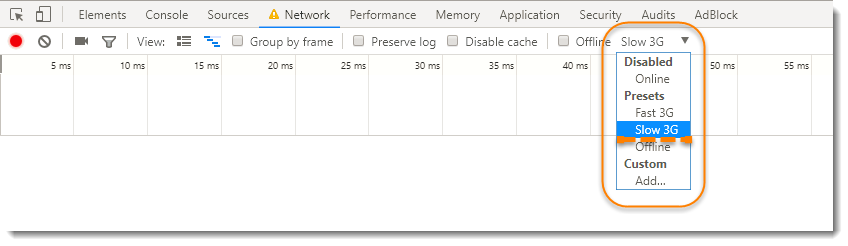
AJAX is a technique to do an XMLHttpRequest (out of band Http request) from a web page to the server and send/retrieve data to be used on the web page. AJAX stands for Asynchronous Javascript And XML.

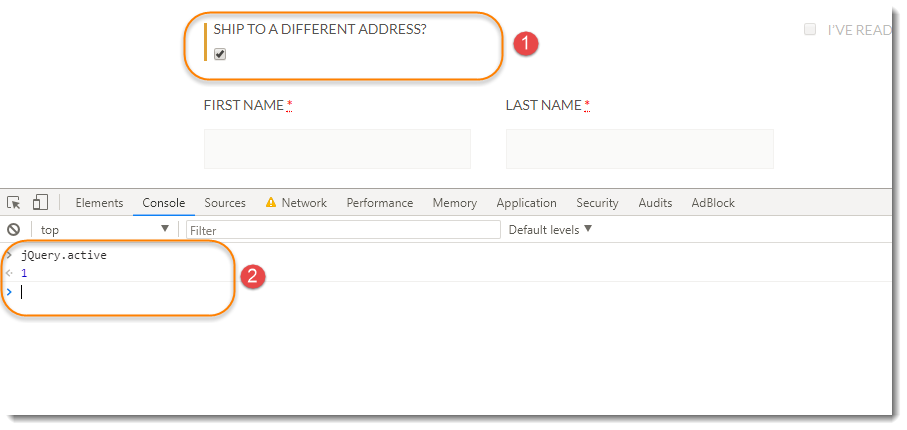
Means: Ajax is way for the client-side browser to communicate with the server (for example: retrieve data from a database) without having to perform a page refresh.

### **What is JQuery?**

**JQuery** (website) is a javascript framework that makes working with the DOM easier by building lots of high level functionality that can be used to search and interact with the DOM. **Part of the functionality of jQuery implements a high-level interface to do AJAX requests**.

* JQuery is a lightweight client side scripting library while AJAX is a combination of technologies used to provide asynchronous data transfer
* JQuery and AJAX are often used in conjunction with each other
* JQuery is primarily used to modify data on the screen dynamically and it uses AJAX to retrieve data that it needs without changing the current state of the displayed page





Here is a sample code to showcase the handling of AJAX controls using Selenium Webdriver. You can integrate it in your test execution class.

Boolean isJqueryCallDone = (Boolean)((JavascriptExecutor) driver).executeScript(“return jQuery.active==0“);

Java 8 introduced **‘Function Interface’** which takes one argument and returns one object. See it’s signature below:

## [Interface Function<T,R>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)

* **Type Parameters:**

T – the type of the input to the function

R – the type of the result of the function

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.util.function.Function;    public class HigherOrderFunc {        public static void main(String args[]) {            Function <Integer, Integer> inc = e -> e + 1;          doSum(5, inc);        }      public static void doSum(int value, Function <Integer, Integer> func) {          System.out.println(func.apply(value));      }  } |

**Let’s understand the code:**

First, create a function which takes one integer and return an integer after adding 1 to it.

* Function <Integer, Integer> inc = e -> e + 1;

Pass that function into another method (sum) with another integer on the side (value)

doSum(5, inc);

public static void doSum(int value, Function <Integer, Integer> func)

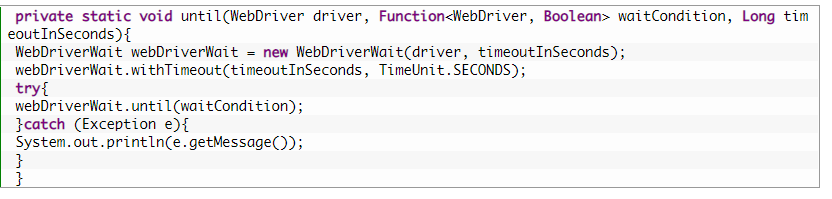
Now, here is the magic of a default ***‘apply’*** method of ***‘function interface’*** which operates the logical operation specified in step 1 with the value being passed. So, here in this case, 5 is the value and operation is value -> value+1 i.e 5 -> 5+1

func.apply(value)

I hope you understand now, what will come back as an output. That would be 6.

This is how, higher order function works. This is simplest example, I’ve provided you to make you clear enough on the working of functions passing into other functions.

As the above script would return either ***True*** or ***False***. But we need to run this code till the time either we get true or the specified time is over. To do that we need to have the ***Selenium WebDriver Wait***, which would provide us ***untill***method.



public static void untilJqueryIsDone(WebDriver driver, Long timeoutInSeconds){

until(driver, (d) ->

{

Boolean isJqueryCallDone = (Boolean)((JavascriptExecutor) driver).executeScript("return jQuery.active==0");

if (!isJqueryCallDone) System.out.println("JQuery call is in Progress");

return isJqueryCallDone;

}, timeoutInSeconds);

}

# Sharing Test Context between Cucumber Step Definitions

### ***What is PicoContainer?***

It’s a tiny library written by ***Paul Hammant*** in 2003-2004. It is pretty awesome because it’s so little and simple:

* *PicoContainer doesn’t require any configuration*
* *PicoContainer doesn’t require your classes to use any APIs such as the horrible @Inject – just use constructors*
* *PicoContainer really only has a single feature – it instantiates objects*

Simply hand it some classes and it will instantiate each one, correctly wired together via their constructors. That’s it. Cucumber scans your classes with step definitions in them, passes them to *PicoContainer*, then asks it to create new instances for every scenario.

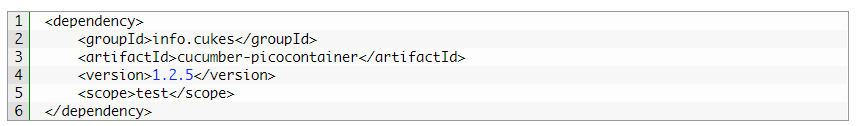
**How to Sharing Test Context between Cucumber Step Definitions using PicoContainer**

We will be performing below steps to share data state across steps:

1. ***Add PicoContainer to the Project***
2. ***Create a Test Context class which will hold all the objects state***
3. ***Divide the Steps class into multiple steps classes with logical separation***
4. ***Write Constructor to share Test Context***

### ***Step 1: Add PicoContainer Library to the Maven Project***

This is really simple, as we have been using Maven Project, all we need to do is to **add the dependencies in to the project POM file**. Dependencies information can be taken from [***Maven Repository – Cucumber PicoContainer***](https://mvnrepository.com/artifact/info.cukes/cucumber-picocontainer)***.***



### ***Step 2: Create a Test Context class***

Be wise to create this class logically. Just think what all information your Steps file are using and put that information in to this class. In our case our steps file is just using the below information:

* ***PageObjects*** : Provided by ***PageObjectManager***
* ***WebDriver***: Provided by ***WebDriverManager***
* ***Properties***: Provided by ***FileReaderManager***

So, the case is simple. We just need the above objects in our Test Context class. But if you re-look at the objects, you would release that our ***FileReaderManager is already a Singleton Class*** and to use it we don’t need to create an instance of it. It creates its instance by itself. So no need to add FileReaderManager to TestContext class, as this class can be referred directly statically like FileReaderManager.getInstance().

1. Create a ***New Package*** and name it as ***cucumber***, by right click on the ***src/test/java*** and select ***New >> Package***. We will keep all the Cucumber Helper classes in the same package moving forward.

2. Create a ***New Class*** file and name it as ***TestContext*** by right click on the above-created package and select **New >> Class**.



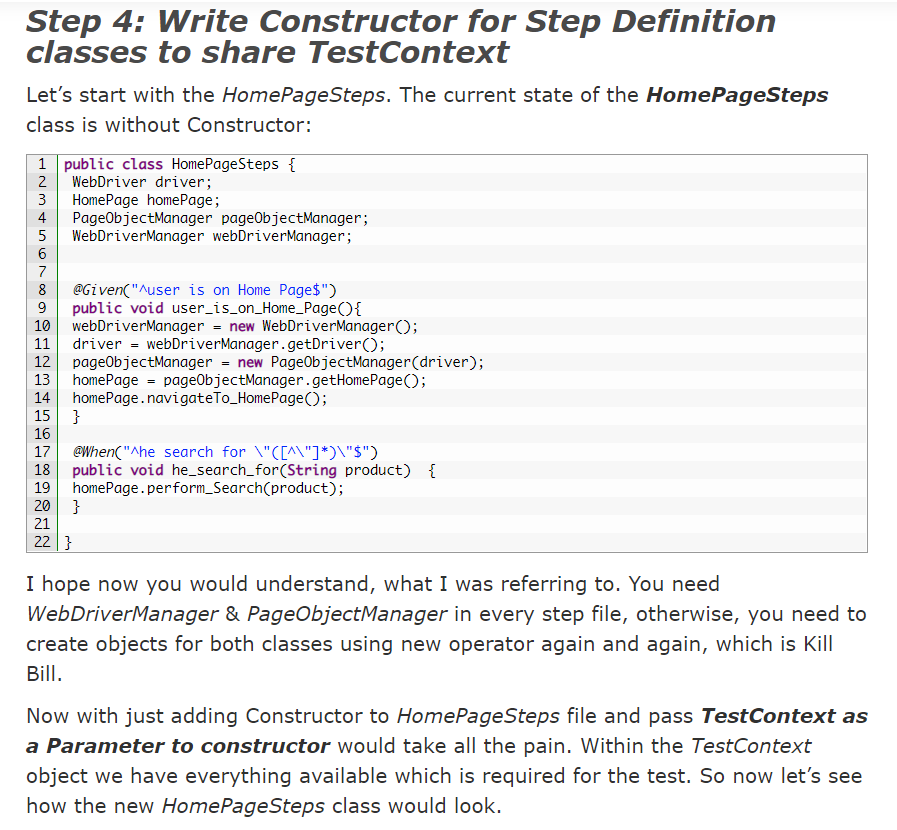
### ***Step 3: Divide the Steps file***

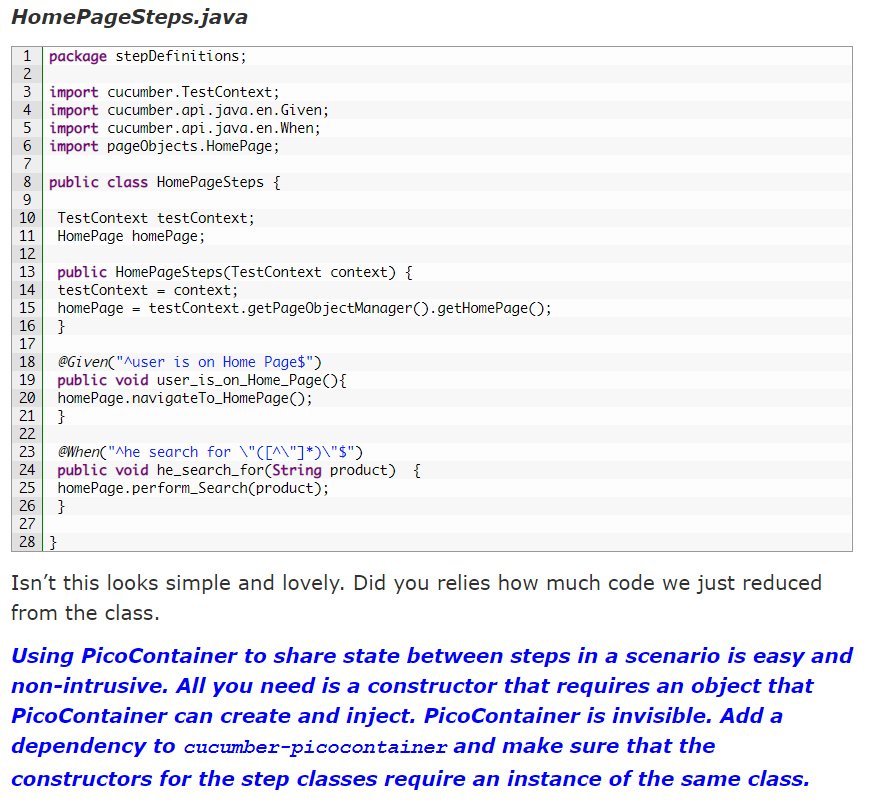
I would just divide the steps file as I did the separations between the Page Objects. For every different page, we have a separate PageObject class. So it makes sense to have a separate step definition class for every page as well. May be your application is different and you won’t get to agree with my approach. In that case who cares 🙂 Divide your steps accordingly.

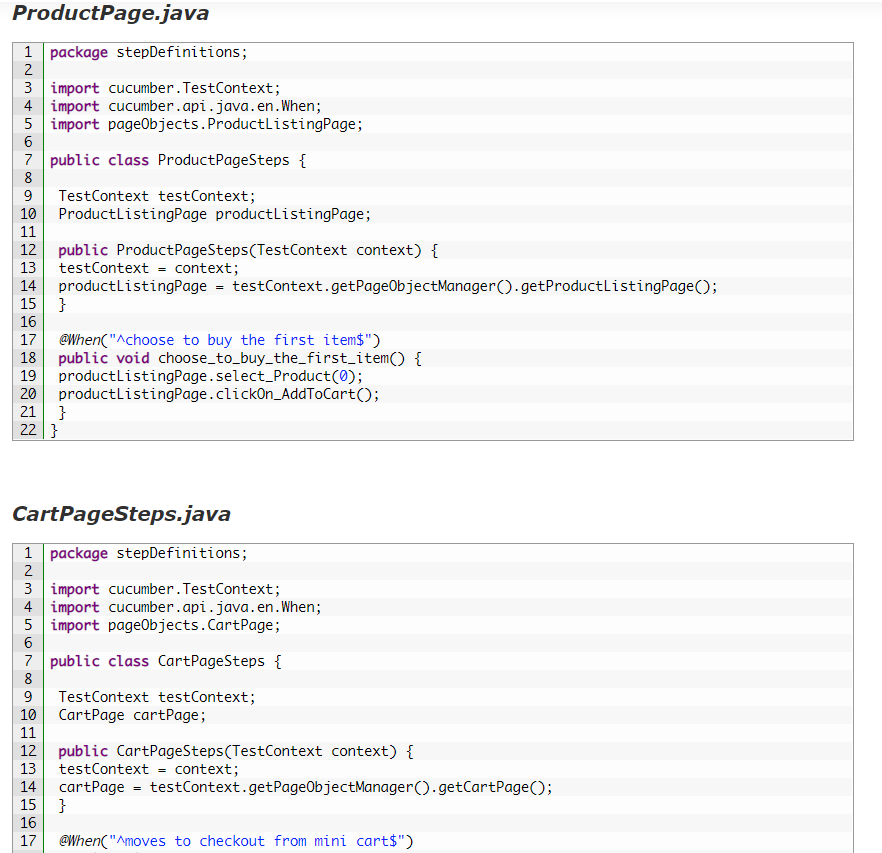
Create four ***New Classes*** in the ***stepDefinitions*** package with the following names:

* HomePageSteps
* ProductPageSteps
* CartPageSteps
* CheckoutPageSteps

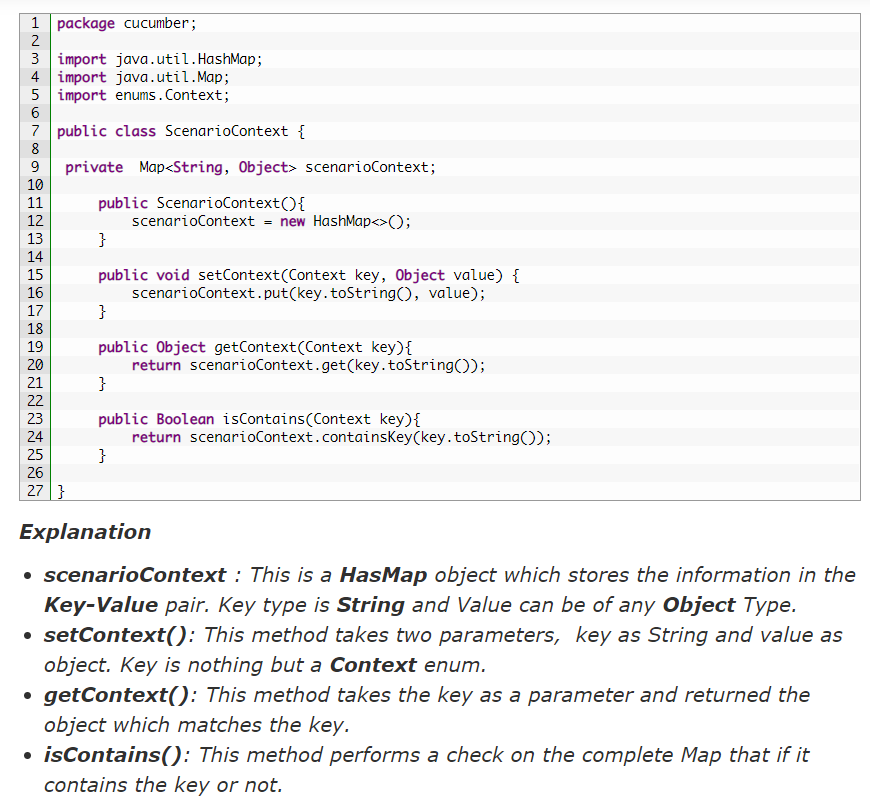
I am sure you won’t need my help to do this separation. Just start copying pasting information from steps class into the above-created classes accordingly. You would get the code in the below step.

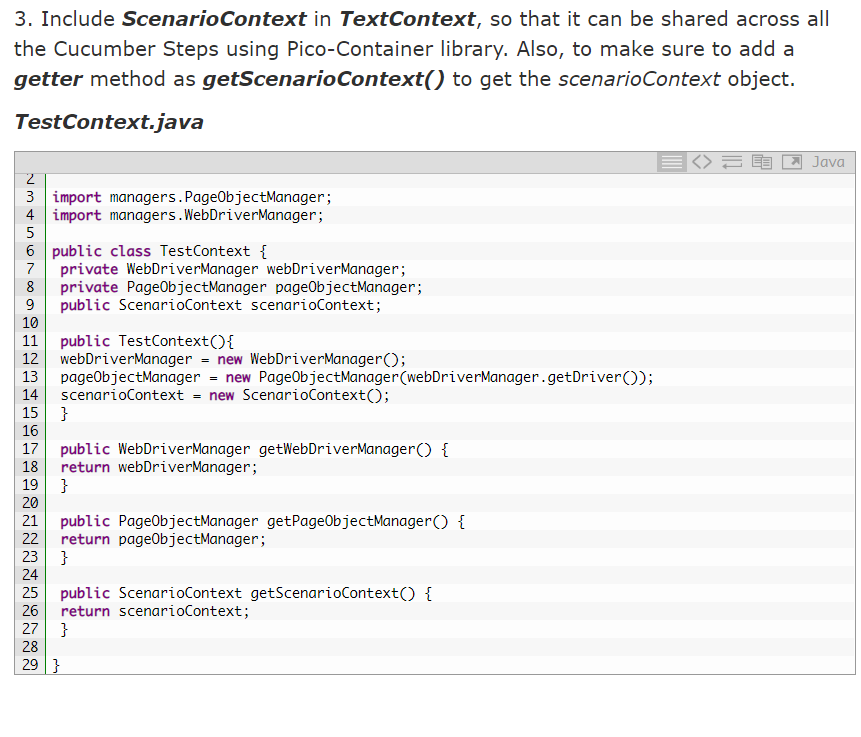


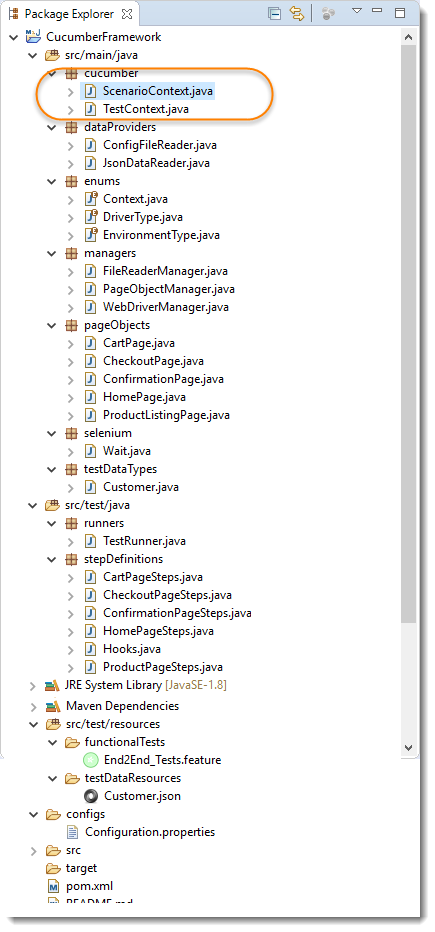




# Share data between steps in Cucumber using Scenario Context

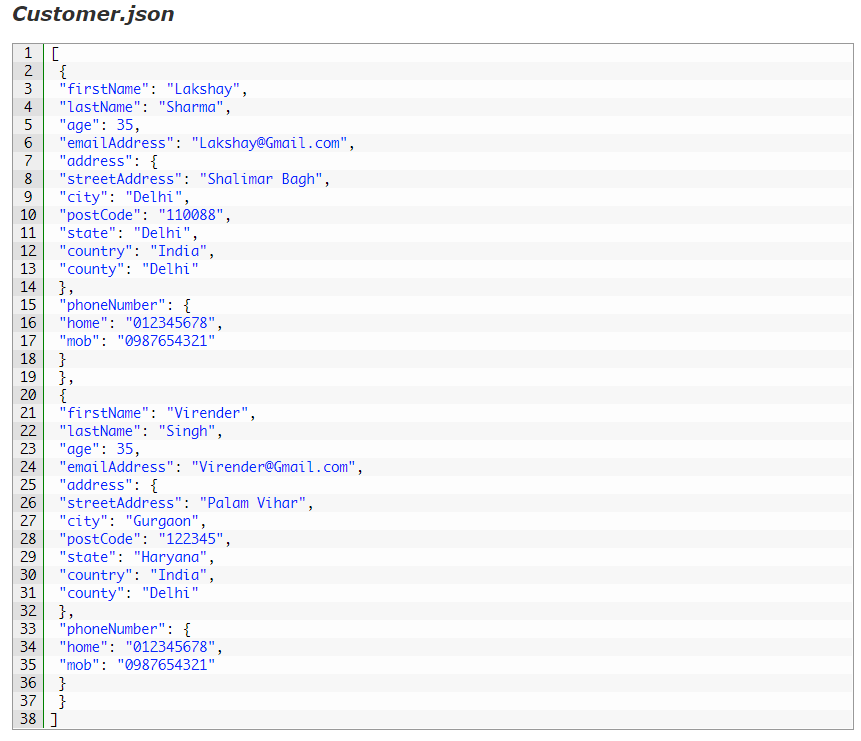








# Data Driven Testing using Json with Cucumber





***Explanation:***

1. ***getCustomerData()****: This is a private method, which has the logic implemented to read the Customer Json and save it to the class instance variable. You should be creating more methods like this if you have more test data files like getPaymentOptions(), getProducts() etc.*
2. ***JsonDataReader()****: Here the responsibility of the constructor is to call getCustomerData() method only.*
3. ***getCustomerByName()****: This just filter the information and return the specific customer to the test. Same method can also be written as :*
4. 