**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.

**Benefits of TDD**

1. Unit test proves that the code actually works
2. Can drive the design of the program
3. Refactoring allows improving the design of the code
4. Low-Level regression test suite
5. Test first reduce the cost of the bugs

**Drawbacks of TDD**

1. Developer can consider it as a waste of time
2. The test can be targeted on verification of classes and methods and not on what the code really should do
3. Test become part of the maintenance overhead of a project
4. Rewrite the test when requirements change

In the last section, we discussed what TDD is. We discussed how TDD is a test centered development process in which we start writing tests firsts. Initially, these tests fails but as we add more application code these tests pass. This helps us in many ways

* We write the application code based on the tests. This gives a test first environment for development and the generated application code turns out to be bug-free.
* With each iteration, we write tests and as a result, with each iteration, we get an automated regression pack. This turns out to be very helpful because with every iteration we can be sure that earlier features are working.
* These tests serve as documentation of application behavior and reference for future iterations.

## 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

BDD is popular and can be utilised for ***Unit level*** test cases and for***UI level*** test cases. Tools like ***RSpec***(for Ruby) or in .NET something like ***MSpec***or ***SpecUnit*** is popular for Unit Testing following BDD approach.  Alternatively, you can write BDD-style specifications about***UI interactions***. Assuming you’re building a web application, you’ll probably use a browser automation library like ***WatiR/WatiN or Selenium***, and script it either using one of the frameworks I just mentioned, or a given/when/then tool such as ***Cucumber (for Ruby)*** or***SpecFlow (for .NET)***.

## BDD Tools Cucumber & SpecFlow

### ***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.

### ***What is SpecFlow?***

***SpecFlow*** is inspired by Cucumber framework in the Ruby on Rails world. Cucumber uses plain English in the Gherkin format to express user stories. Once the user stories and their expectations are written, the Cucumber gem is used to execute those stores. ***SpecFlow brings the same concept to the .NET world*** and allows the developer to express the feature in plain English language. It also allows to write specification in human-readable[***Gherkin format***](https://toolsqa.com/cucumber/gherkin-business-driven-development-bdd-language/).

## Why BDD Framework?

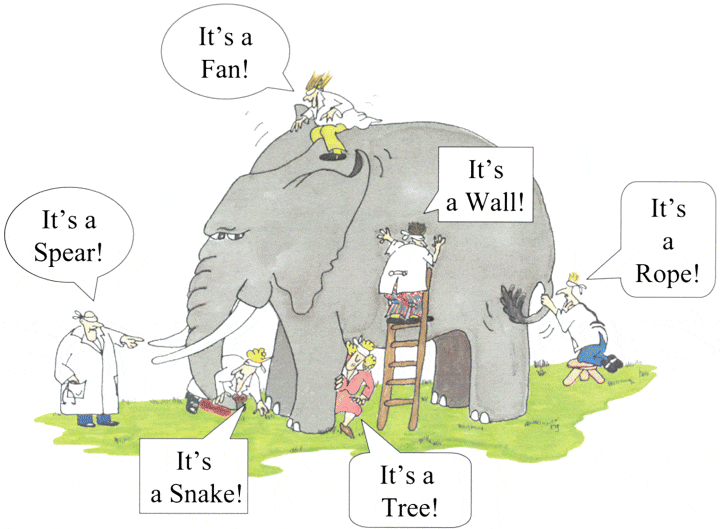
Let’s assume there is a requirement from a client for an E-Commerce website to increase the sales of the product with implementing some new features on the website. The only challenge of the development team is to convert the client idea into something that actually delivers the benefits to client.

The original idea is awesome. But the only challenge here is that the person who is developing the idea is not the same person who has this idea. If the person who has the idea happens to be a talented software developer, then we might be in luck: the idea could be turned into working software without ever needing to be explained to anyone else. Now the idea needs to be communicated and has to travel from Business Owners(Client) to the development teams or many other people.

Most software projects involve teams of several people working collaboratively together, so high-quality communication is critical to their success. As you probably know, good communication isn’t just about eloquently describing your ideas to others; you also need to solicit feedback to ensure you’ve been understood correctly. This is why agile software teams have learned to work in small increments, using the software that’s built incrementally as the feedback that says to the stakeholders “Is this what you mean?”

Below image is the example of what clients have in their mind and communicated to the team of developers and how developers understands it and work on it.

### **Wrong Perception**

[](https://toolsqa.com/wp-content/uploads/2014/12/Blind-Men-And-The-Elephant.gif)  
With the help of Gherkin language cucumber helps facilitate the discovery and use of a ubiquitous language within the team. Tests written in cucumber directly interact with the development code, but the tests are written in a language that is quite easy to understand by the business stakeholders. Cucumber test removes many misunderstandings long before they create any ambiguities in to the code.

### Example of a Cucumber/SpecFlow/BDD Test:

The main feature of the Cucumber is that it focuses on Acceptance testing. It made it easy for anyone in the team to read and write test and with this feature it brings business users into the test process, helping teams to explore and understand requirements.

**Feature: Sign up**

**Sign up should be quick and friendly.**

**Scenario: Successful sign-up**  
**New users should get a confirmation email and be greeted**  
**personally by the site once signed in.**

**Given I have chosen to sign up**  
**When I sign up with valid details**  
**Then I should receive a confirmation email**  
**And I should see a personalized greeting message**

**Scenario: Duplicate email**

**Where someone tries to create an account for an email address**  
**that already exists.**

**Given I have chosen to sign up**  
**But I enter an email address that has already registered**  
**Then I should be told that the email is already registered**  
**And I should be offered the option to recover my password**

## What is Gherkin – *****BDD***** Language?

Before diving into Gherkin, it is necessary to understand the importance and need of a common language across different domains of project. By different domains I  By different domains I mean **Clients, Developers, Testers, Business analysts** and the ***Managerial*** team. Let’s start by talking about usual problems of a development project first and then we will move to a solution, while doing so we will come across the need for a common language.

Assume you are a part of a technical team (Developer and Tester) and you have a task of collaborating with the business team (Business owners and Business analysts). You have to come up with the requirements of your project, these requirements will be what your development team will be implementing and test team will be testing. Also, that you have to make a small search feature on your E-Commerce platform. This feature will allow users to search for a product on your website.

As we all might have faced in our experience that requirement given by business team are very crude and basic. For example, in this scenario we may get the following requirements:

**3.    Functional Requirements**

***3.1    Search Functionality***

3.1.1     User should be able to search for a product

3.1.2    Only the products related to search string should be displayed.

### Questions raised from the above requirements

As we can see these requirements are good and useful but are not accurate. They describe a broad behavior of the system but do not specify concrete behavior of the system. Let me illustrate it by dissecting the first requirement, first requirement says that user should be able to search for a product but it fails to specify following

– What is the maximum searchable length of search string?

– What should be the search results if user searches for an invalid product?

– What are the valid characters that can be used to search?

**\***and similarly a few more detailed behavior of the application.

Usually in a project we end up asking above questions with the business team and we get replies, most of the replies reach the project documentation but the unfortunate ones are lost in emails and telephonic conversations. Also these replies are open to interpretation, for example:

***Question to Business Owner :*** What should be the search results if user searches for an invalid product?

***Reply from Business Owner :*** Invalid product searches should show following text on the search page: ***No product found***

### Answers of the Questions result in to more Doubts and Interpretation

We get the answers of the questions asked from the Business team but it opens for interpretation or doubts in following ways:

– Definition of invalid product is ambiguous and different team members will interpret it in different ways. One may consider that an invalid product is one which is not present in the inventory and other team member might consider an invalid product to be one which is a spelling mistake.

– The answer by the business team says that “No product found” text should be displayed on the page. Does it says that a new search option should be present for the user? or may be related/similar search options should be displayed for the user?

These are exact points where error is introduced in the system. Also, if we analyze the second doubt we would see that user Business team would love to have a new search option and related/similar searches option presented to the user. However, they were not able to think of this scenario when the question was asked. As a result what happened in the above example is

1. Business team and the technical teams are communicating at two different levels, business team being vague and technical team trying to be precise.
2. Ambiguity being introduced in the system, here by the definition of “invalid product”.
3. Not enough insight being given to the Business team, so that they could have come up with new scenarios.
4. Some details of project being lost in emails and telephonic conversations.

### How to Improve the Requirement?

Now let’s improve the first requirement given by the business team and try to make it more precise:

“When a user searches, without spelling mistake, for a product name present in the inventory. All the products with similar name should be displayed“

“When a user searches, without spelling mistake, for a product name present in the inventory. Search results should be displayed with exact matches first and then similar matches“

Here we can see that how clear the requirements have become and with these clear requirements we are able to think more about the system. For eg. In the case of second requirement, after reading it we may think of other scenarios like:

* What should happen when there no exact and similar matches?
* Should the user be given an error message?
* Or the user is given a message stating when the product is expected to arrive in inventory.

### What have we achieved here?

We have forced the client to think in terms of details. With this improved thinking Business teams are coming with more refined requirements. This in turn with reduces the ambiguity in the project and will make developers and testers life easy by reducing the number of incorrect implementations. Also, you can see that each requirement now documents one exact behavior of the application. This means that it can be considered as a requirement document in itself.

### What’s the conclusion?

Well, with the above example or exercise we can conclude the followings:

1. Different teams in the project need a common language to express requirements. This language should be simple enough to be understood by Business team members and should be explicit enough to remove most of the ambiguities for developers and testers.
2. This language should open up the thinking of team members to come up with more scenarios. As you express more details you try to visualize the system more and hence you end up making more user scenarios.
3. This language should be good enough to be used as project documentation.

To answer these problems ***Gherkin*** was created. Gherkin is a simple, lightweight and structured language which uses regular spoken language to describe requirements and scenarios. By regular spoken language we mean English, French and around 30 more languages.

## Example of Gherkin

As Gherkin is a structured language it follows some syntax let us first see a simple scenario described in gherkin.

**Feature: Search feature for users**  
**This feature is very important because it will allow users to filter products**

**Scenario: When a user searches, without spelling mistake, for a product name present in inventory. All the products with similar name should be displayed**

**Given User is on the main page of www.myshopingsite.com**  
**When User searches for laptops**  
**Then search page should be updated with the lists of laptops**

Gherkin contains a set of keywords which define different premise of the scenario. As we can see above the colored parts are the keywords. We will discuss about the gherkin test structure in details later but the key points to note are:

* – The test is written in plain English which is common to all the domains of your project team.
* – This test is structured that makes it capable of being read in an automated way. There by creating automation tests at the same time while describing the scenario.

## Cucumber and its outstanding features

As yet, there have been many successful Agile software projects. Thanks to the Behavior-Driven Development (BDD) method using the Cucumber tool. So, what is Cucumber?

A cucumber is a tool used to run automated acceptance tests created in a BDD format. One of its extraordinary features of the tool is the ability to carry out plain-text functional descriptions (written in the language called [***Gherkin***](https://toolsqa.com/cucumber/gherkin/)) as automated tests.

Let’s take a look at the below example:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Feature: Update password     Scenario: Admin user can update the user password    Given I am in the HR system with an Admin account  When I update password of another user  Then I receive a message for updating password successfully  And user password is updated to the new password |

This amazing feature of ***[Behavior-Driven Development (BDD)](https://toolsqa.com/cucumber/behavior-driven-development/" \t "_blank)*** approach with the advantages as below:

* Writing BDD tests in an omnipresent language, a language whose structure is built around the domain model and widely used by all team members comprising of developers, testers, BAs, and customers.
* Connecting technical with non-technical members of a software team.
* Allowing direct interaction with the developer’s code, but we write BDD tests in a language that can also be made out by business stakeholders.
* Last but not least, acceptance tests can execute automatically, while business stakeholders manually perform it.

### ***1. Cucumber helps improve communication***

Cucumber assists in improving communication between technical and non-technical members of the same project. Let’s have a look at the requirement below and its automation tests:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | As an Admin User,  I would like to change the password of other user's accounts.  Feature: Update password   Scenario: Admin user can update the user password     Given I am in the HR system with an Admin account     When I update password of another user     Then I receive a message for updating password successfully     And user's password is updated to the new password |

With TestNG, the above test scenario can implement as below:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @test  public void testAdminUserCanUpdateUserAccountPassword() {   // create users   User userAdmin = new User(UserRole.ADMIN, username, password);   User user = new User(UserRole.VIEWER, user\_username, user\_password);     // use Admin user to update another user password    String message = userAdmin.updatePassword(user, user\_new\_password);      // verify password changed    Assert.assertEquals(message, "Password changed successfully");    Assert.assertEquals(user.getPassword(), user\_new\_password);  } |

We can write the same test case using Cucumber:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Feature: Update password   Scenario: Admin user can update the user password     Given I am in the HR system with an Admin account     When I update password of another user     Then I receive a message for updating password successfully     And user's password is updated to the new password |

Both automation test scripts above execute well to complete the test automatically. But do all testers of your team make out these tests? Is there a possibility of other business analysts and other stakeholders use these tests again at the acceptance testing (AT) stage?

The automation test with TestNG may be tough for most manual testers and BAs to catch up. Moreover, it is not possible 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, we develop/ create the automation test using Cucumber in a business domain language or in natural language, which all members of the software project team can easily make out. Communication is vital for any development team, especially in the Agile team. There are usually lot of 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 developers can develop the same feature specification now for testing by testers. It is a powerful tool because it can help lower the risk of misunderstanding as well as the communication breakdown.

### ***2. Cucumber is an Automated Acceptance Testing Tool***

The acceptance test is generally carried out by BAs/customers to make sure that the development team has built specific features. A common 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 quickly. Here is a demonstration with a little adjustment:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | As an Admin User,  I would like to change the password of other user's accounts.  Feature: Update password   Scenario: Admin user can update the user password     Given I am in the HR system with an Admin account     When I update password of another user     Then I receive a message for updating password successfully     And user's password is updated to the new password |

We write the automation test in the Cucumber framework:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Scenario Outline: Verify Updating user password feature   Given I am in the HR system with "<account\_type>" account   And there is another user with "<old\_password>" password   When I update password of the user to "<new\_password>"   Then I got the message "<message>"   And the user password should be "<final\_password>"  Examples:  |account\_type  |old\_password |new\_password |message                |final\_password |  |Admin         |$Test123 |@Test123     |Password changed.. |@Test123       |  |Viewer        |$Test123 |@Test123     |Invalid right access.. |$Test123       | |

### ***3. All testers can take part in automation test with Cucumber BDD***

In addition to improving communication within the members of the same testing team, Cucumber also helps leverage tester’s skills efficiently. The expertise gap always exists in every organization. In other words, some testers have great technical expertise in programming utilizing automated testing, while others are performing manual testing with limited programming skills in the same team. Thanks to Cucumber, all testers, irrespective of their skill levels, can participate in the process of performing automation tests.

Let’s take a look at the above example:

* Any tester who knows the business logic and workflow can write feature files, add more scenarios, and test datasets.
* Moreover, any tester who has a basic knowledge of programming and know-how to create objects, access properties, call methods, can generate step definitions.
* Any tester with a higher programming skill level can take part in the process of making a framework, define data source connection, and so on.

There are still a few important issues when implementing Cucumber:

Cucumber helps run test scenarios mentioned in a plain text file using business domain knowledge. Thus, the usage of languages and the perception of the one who creates the test might directly influence the test scenarios, leading to the risk of misunderstanding. We should present the Test scenarios clearly, and their implementation should perform accurately for each step. For example, when you want to verify the Search feature on Google, the test should be:



|  |  |
| --- | --- |
| 1  2  3  4 | Scenario: performing a search on google  Given I am on "www.google.com" site  When I search for "Cucumber and BDD"  Then ... |

We incorporate the below steps to have the following test:



|  |  |
| --- | --- |
| 1  2  3 | Scenario: performing a search on google  When I search for "Cucumber and BDD"  Then ... |

The stages of the Cucumber tool perform in an ordinary language. One can use them again in various test scenarios. It helps reduce the effort to create tests. However, maintaining the test to be both readable and reusable is a big challenge. If the test is written at a very high level for any stakeholders to make out, we can reuse a few steps (bold). Both the above scripts are right; however, the second one is not apparent. Because it does too much more than expected: opening Google’s website and searching with the specified text. Say, you want to extend the test to search more texts, you may repeat the above step. Consequently, we open the Google site twice. If you do not follow the requirement stringently, the Cucumber testing tool will cause misunderstanding sooner or later and be so difficult to maintain when being extended.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Feature: Update password     Scenario: Admin user can update the user password     Given I am in the HR system with an Admin account     When I update password of another user     Then I receive a message for updating password successfully     And user's password is updated to the new password     Scenario: Viewer user cannot update the user password     Given I am in the HR system with a Viewer account      When I update password of another user      Then I receive a message for not able to update the user password      And user's password remains the same |

Contrarily, if the test is generic and if we can reuse to verify updating the user’s Last Name, non-technical stakeholders will have difficulty in catching up. Additionally, they can not perform Acceptance Tests.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Scenario: Admin user can update user password:   Given I am in the "$System.HR\_Page" with "admin@test.com" username  and "$Test123" password   And there is another user in "$System.HR\_Page" with "user@test.com"  username and "$Test123" password   When I update "$UserTemplate.Password" of "user@test.com" user to"@Test123"   And I save the response message as "response\_message"   Then "$response\_message" should be "Password changed successfully"   And the  "user@test.com" user's "$UserTemplate.Password" should be"@Test123" |

During the testing process, you have to adjust test scenarios regularly. We do it until they reach an entirely acceptable balance where all members can understand and reuse.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Scenario: Verify Updating user password feature   Given I am in the HR system with "Admin" account   And there is another user with "$Test123" password   When I update password of the user to "@Test123"   Then I got the message "Password changed successfully."   And the user password should be "@Test123" |

Or with some more test data:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Scenario Outline: Verify Updating user password feature   Given I am in the HR system with "<account\_type>" account   And there is another user with "<old\_password>" password   When I update password of the user to "<new\_password>"   Then I got the message "<message>"   And the user password should be "<final\_password>"    Examples:  |account\_type |old\_password |new\_password |message           |final\_password |  |Admin        |$Test123 |@Test123     |Password changed.. |@Test123  |  |Viewer       |$Test123 |@Test123     |Invalid right access.. |$Test123  | |

## Important notes for the testing team who wants to get started with Cucumber

* Consider automation tests as essential as a real project. The code should follow coding practice, convention, etc.
* One should also consider an appropriate editor tool. This editor should help debug and edit feature files in standard text format. Aptana (free editor), RubyMine (commercial editor), and ***Katalon Studio*** are suitable options that completely support BDD-based Cucumber.
* Last but not least, make feature files an actual “communication” layer where you can store received test data and format test data. It does not contain Domain business logic.

Cucumber is one of the most powerful tools. It offers us the real communication layer on top of a robust testing framework. The tool can help run automation tests on a wide-ranging testing needs from the backend to the frontend. Moreover, Cucumber creates deep connections among members of the testing team, which we hardly find in other testing frameworks. With many years of automation testing experience, I recommend that Cucumber for Web UI and Web service testing should implement in a way that it helps in successful Agile software project operation.

## How to Install Java?

It is not difficult to install java on any platform whether it is Windows, Mac, etc. But there are a few steps to follow:

1. **Download Java**
2. **Install Java**
3. ***Set Java Environment Path***
4. **Verify Java Installation**

The following are steps to***Download & Install Java on Windows***and these steps also remain the same for any platform other than windows.

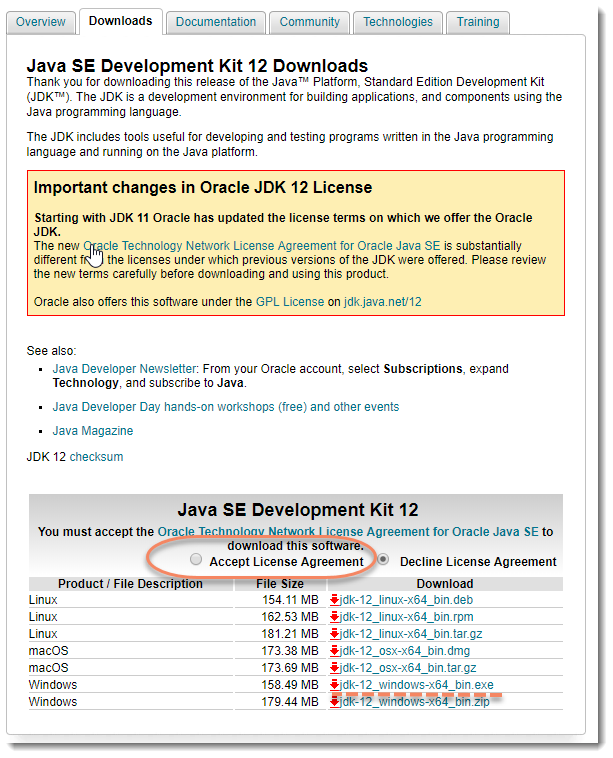
### ***Steps to Download Java Development Kit (JDK)***

To install Java, you first need to download the installer program from Oracle. Visit the ***Download Java*** page: [***https://www.oracle.com/technetwork/java/javase/downloads/index.html***](https://www.oracle.com/technetwork/java/javase/downloads/index.html)***.***Click on ***Download*** button.

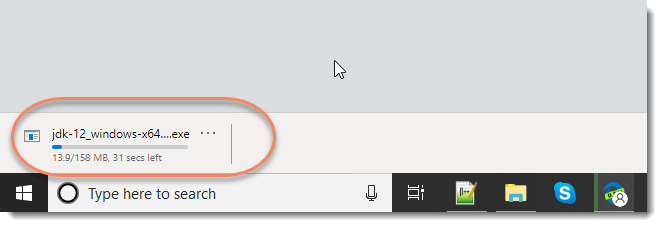


***Note***: As on date: 15th Apr’19 the latest Java version is 12.

2) ***Accept***the License Agreement.  Choose the correct OS corresponding to the specific JDK. (Windows, Mac, Linux, etc.)

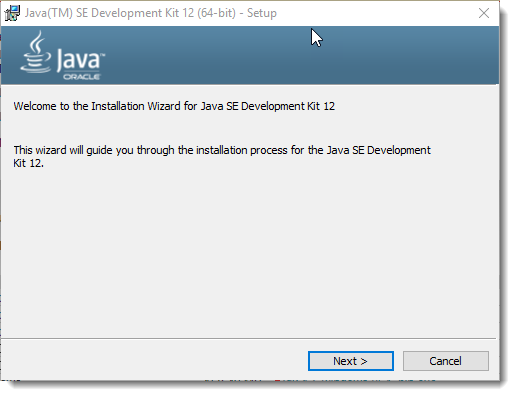


3) The above step will start the downloading of the JDK exe automatically. This can be noticed at the left bottom side of the browser window.

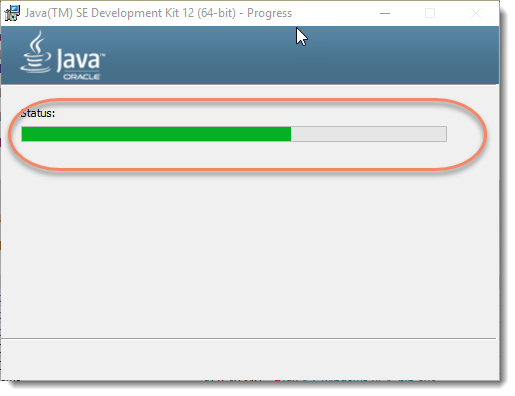


### ***Steps to Install Java***

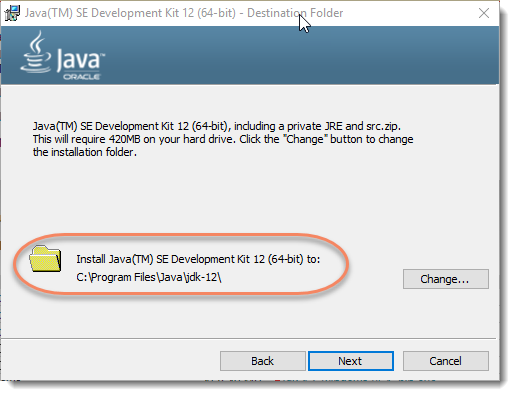
1) Once the download is complete, ***double click*** the file to begin the installation of ***JDK***. This will start the installation process. The installation process starts. Click the **Next** button to continue the installation.



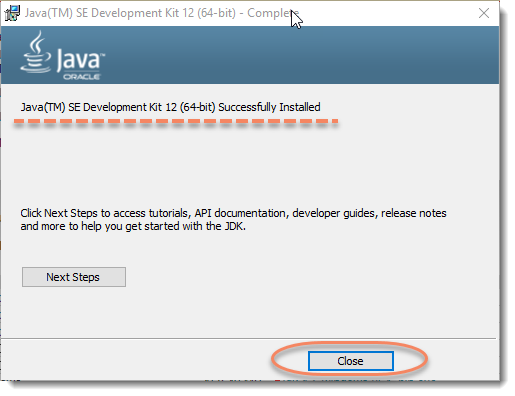
2) It starts preparing for the installation and that will take a few minutes to complete.



3) Next window will ask you where or on which location you would like to install Java on your system. You can choose to change where you want to keep your folder but it’s best to stick with the default options. Click **Next** to continue.

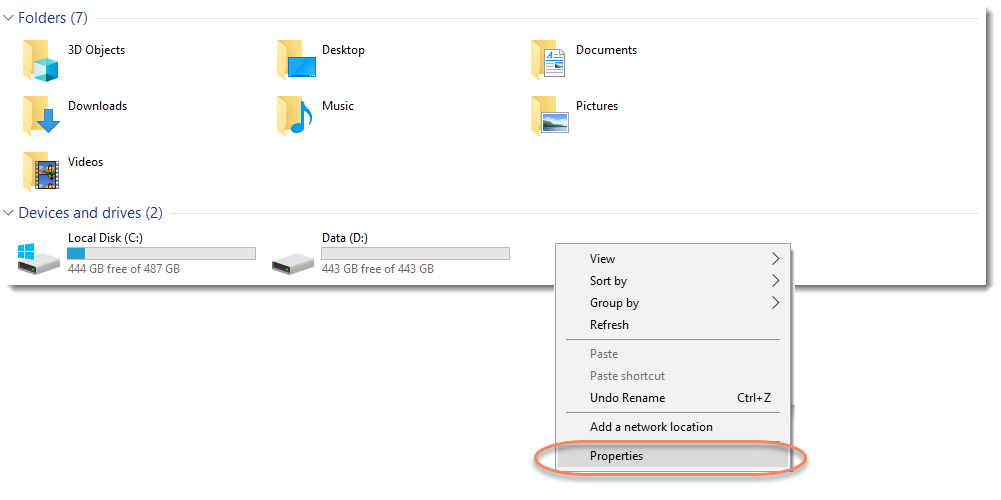


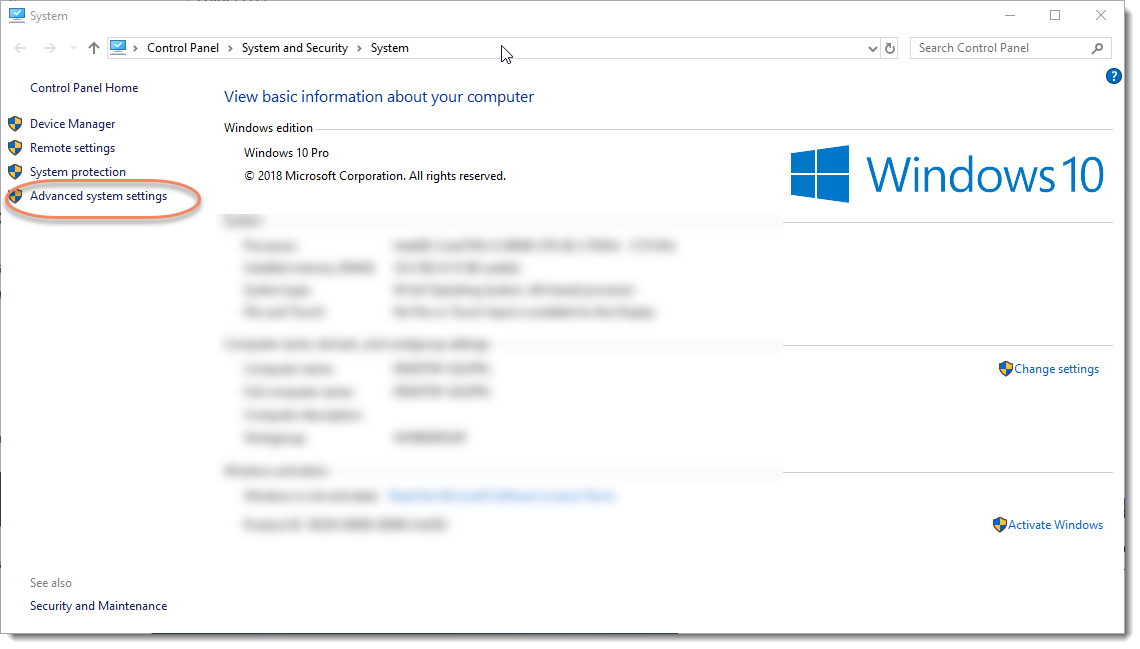
4) Let the installation finish. A few brief dialogs confirm the last steps of the installation process; click **Close** on the last dialog. This will complete the Java installation process.

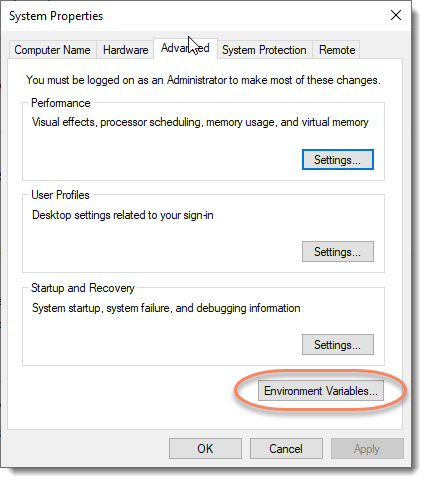


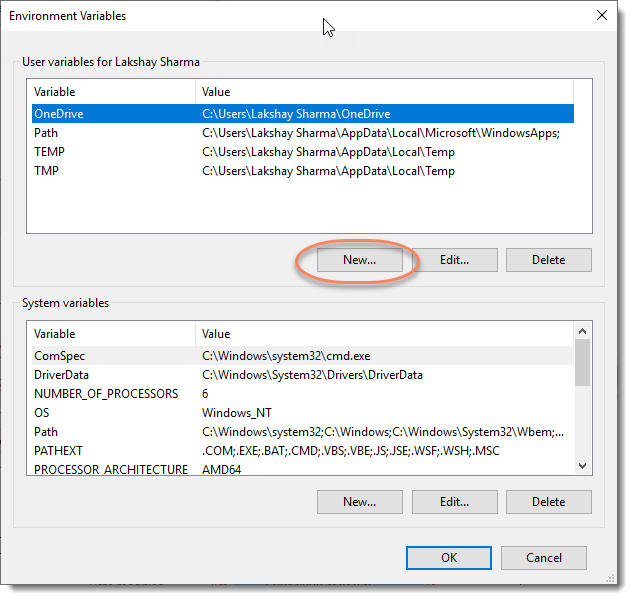
### ***Steps to set the Java Environment Path***

The ***Path Variable*** gives the location of executables like ***javac***, ***java,***etc. JDK’s programs (such as Java compiler javac.exe and Java runtime java.exe) reside in the sub-directory ***bin*** of the JDK installed directory. You need to include JDK’s bin in the path to run the JDK programs. Let’s see how to do it.

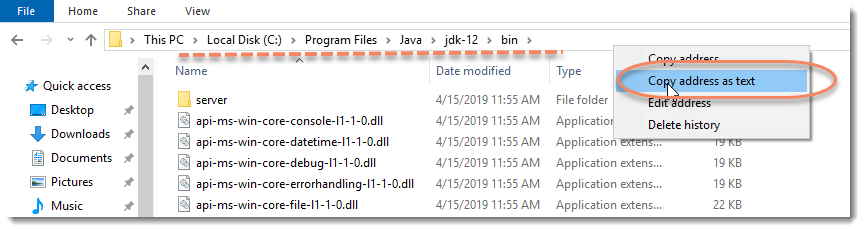
1) ***Right Click*** on the ***My Computer*** and select the ***Properties***.  


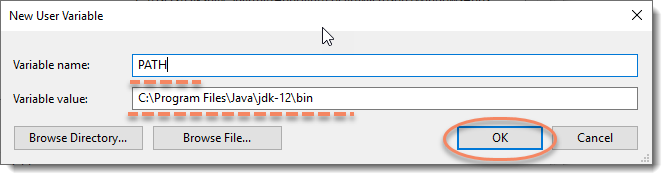
2) Click ***Advanced system settings*** on the left pane.  


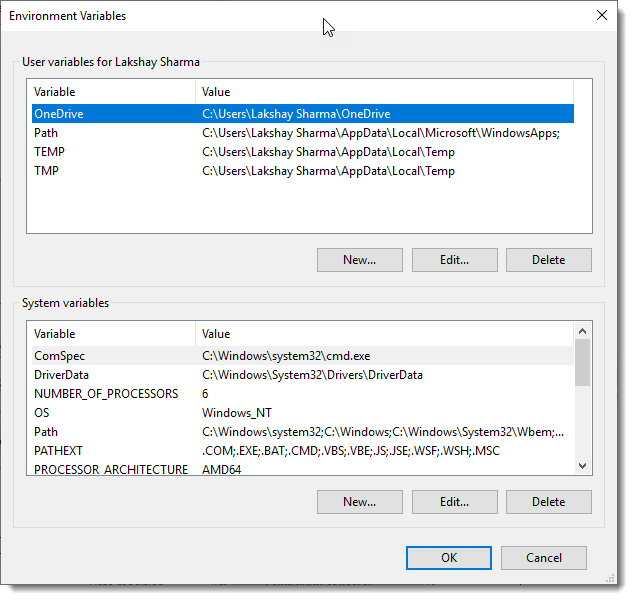
3) Switch to ***Advanced*** tab and click ***Environment Variables*** button.  


4) Click on the ***New***button of ***User variables***.   


***Note***: In case the path variable is already there, you can simply edit the same. For Newer Windows 10: You shall see a TABLE listing all the existing PATH entries (if not, go to the next step). Just add the new one for JDK as well. Steps to get the bin path are mentioned below.

5) Right-click on the address bar and ***Copy address as text***of ***bin*** folder where the JDK is installed.  


6) Type ***PATH***in the ***Variable Name***. And then paste the path of bin folder in ***Variable Value.*** Click on ***OK*** button.  


7) Click on ***OK*** button.  


### ***Steps to Verify Java Installation***

Go to ***Command Prompt*** and type ***java -version*** If you see a screen like below, Java is installed.

### Verify-Java-Installation

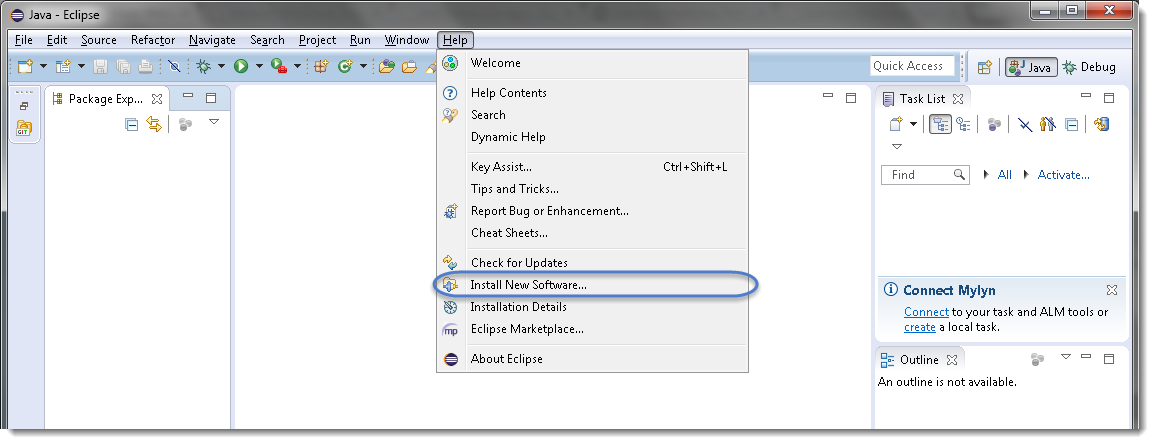
***Cucumber Eclipse Plugin*** helps eclipse to understand the basic Gherkin syntax and it works like a syntax highlighter. It highlights all the main syntax in the feature file which makes it more readable and clear. It also enables the run of the feature file independently, without the help of JUnit. We will talk about it later, but as of now please go ahead and install this plugin.

## Install Cucumber Eclipse Plugin

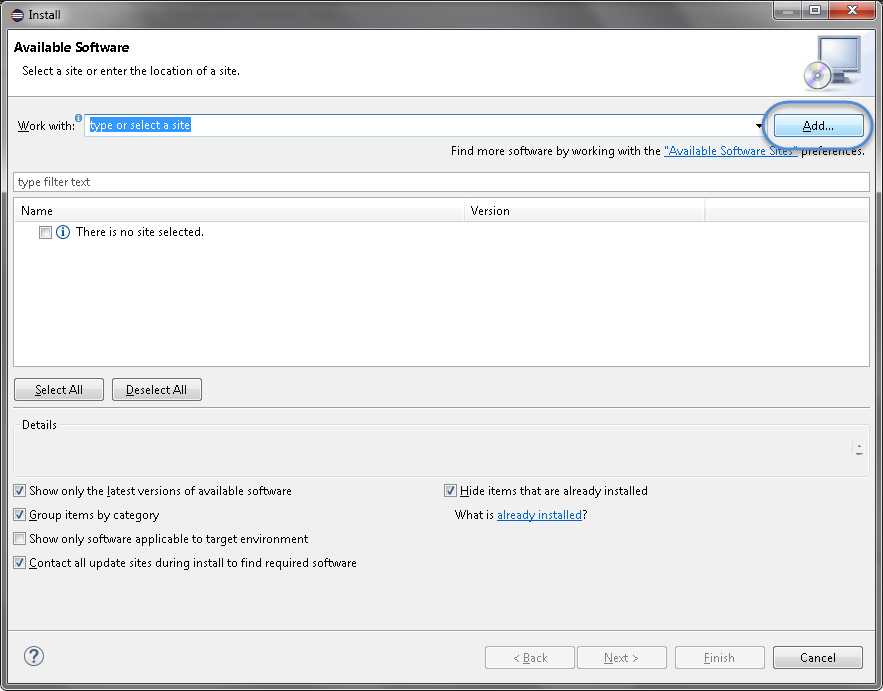
It is easy to install ***Cucumber Eclipse Plugin***, as it comes as a plugin for ***Eclipse IDE***. Prerequisite for installing this plugin is your Internet connection should be up & running during installation of this plugin and Eclipse IDE should be installed in your computer. Please see [***Download and Install Eclipse***](https://toolsqa.com/selenium-webdriver/download-and-start-eclipse/) to setup Eclipse to you system.

### Steps to follow:

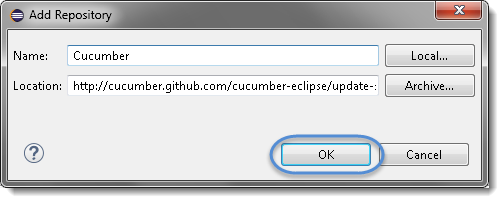
1) Launch the Eclipse IDE and from Help menu, click “***Install New Software***”.



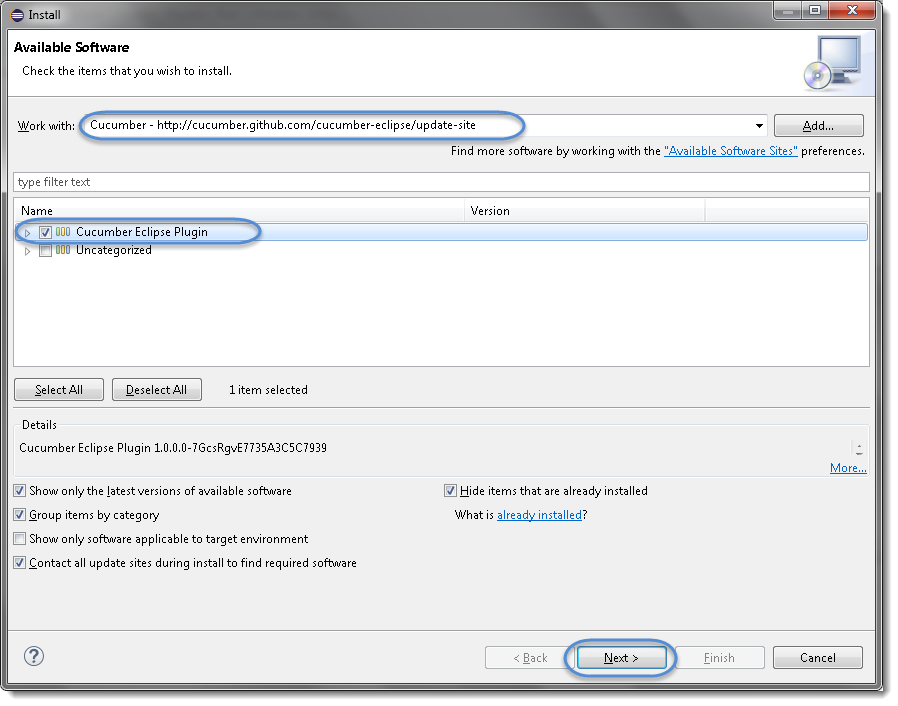
2) You will see a dialog window, click “***Add***” button.



3) Type name as you wish, let’s take “***Cucumber***” and type “***http://cucumber.github.com/cucumber-eclipse/update-site***” as location. Click ***OK***.

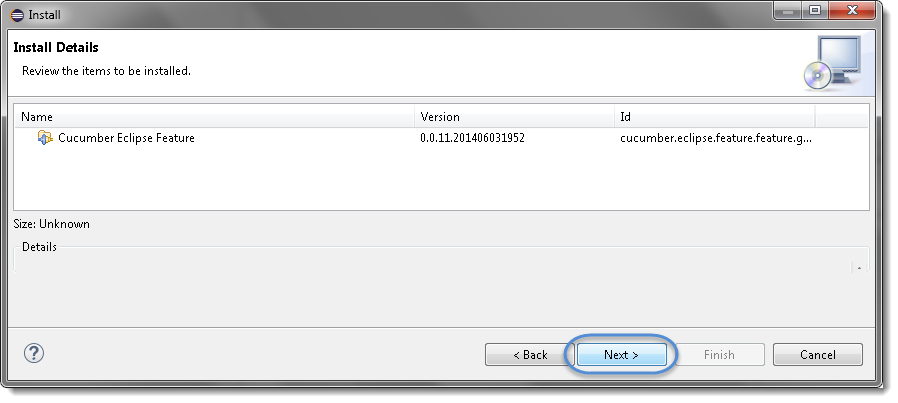


4) You come back to the previous window but this time you must see ***Cucumber Eclipse Plugin*** option in the available software list. Just ***Check*** the box and press “***Next***” button.

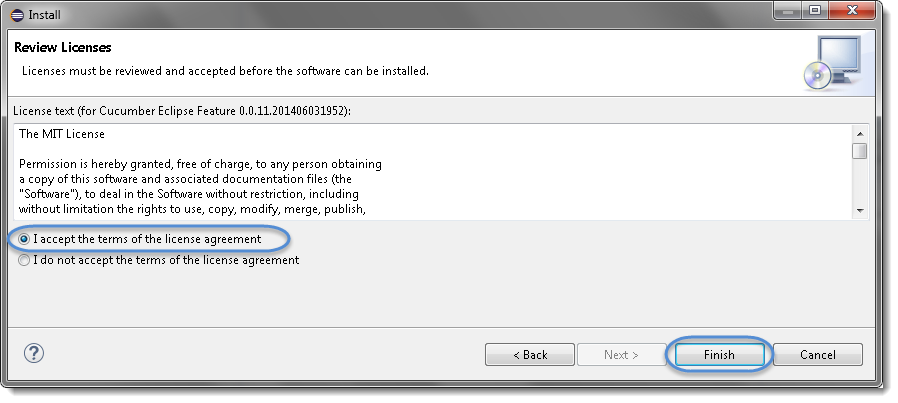


***Note:*** If running behind a proxy server and you get a ‘HTTP Proxy Authentication Required’ error you may need to contact a system administrator to set up your proxy server settings.

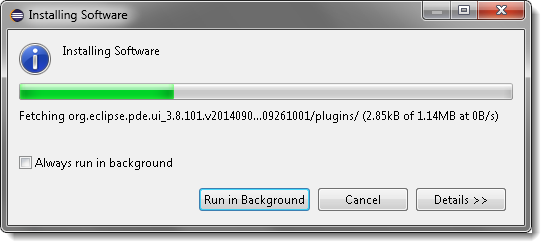
5) Click on ***Next***.



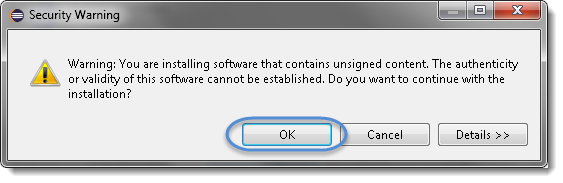
6) Click “***I accept the terms of the license agreement***” then click ***Finish***.



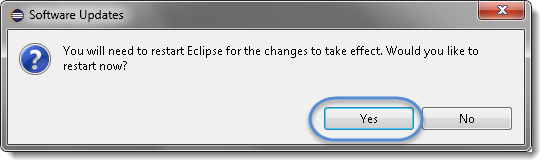
7) Let it install, it will take few seconds to complete.



8) You may or may not encounter a Security warning, if in case you do just click ***OK***.



9) You are all done now, just click **Yes**.



ucumber functionality is abstracted out in jars and following Cucumber jars/modules are required to run a *Cucumber* test with *Java* in *Eclipse*.

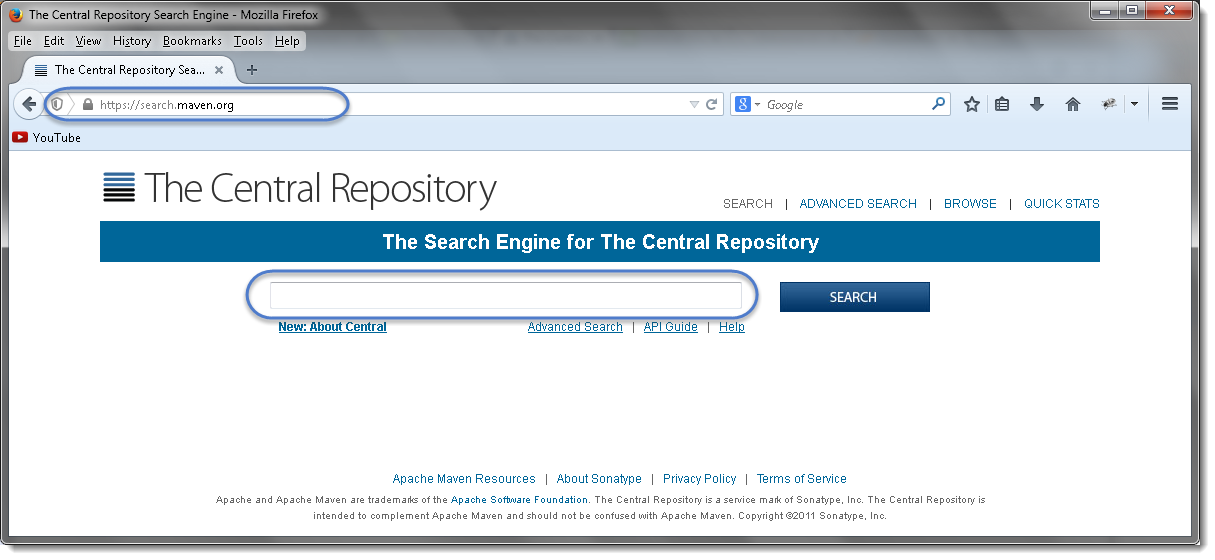
1. ***cucumber-core***
2. ***cucumber-java***
3. ***cucumber-junit***
4. ***cucumber-jvm-deps***
5. ***cucumber-reporting***
6. ***gherkin***
7. ***junit***
8. ***mockito-all***
9. ***cobertura***

There are many options to download Cucumber jars for Eclipse, can follow any of these below:

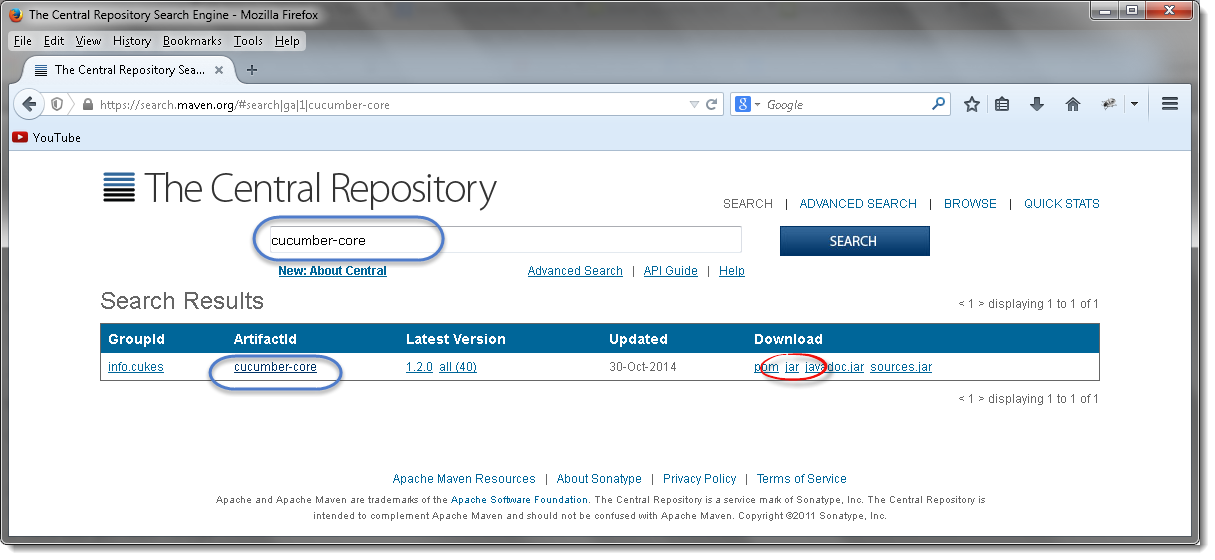
* ***Download Cucumber Jars from Online Maven Repository***
* ***Download Cucumber Jars from oss.sonatype.org***
* ***Download Cucumber Jars from Maven dependencies (Easiest)***

**Option 1 : Download Cucumber Jars from Online Maven Repository**

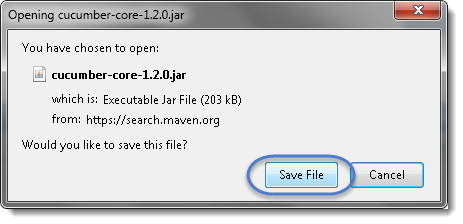
1) Go to [***https://search.maven.org***](https://search.maven.org/)*.*



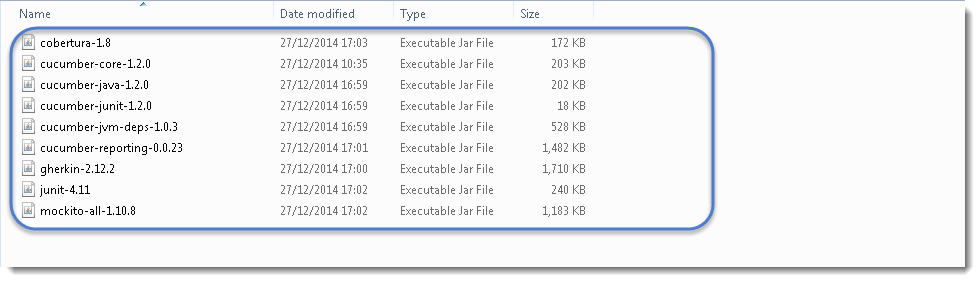
2) Search for ***cucumber-core***in the Central Maven Repository. It will return the *Cucumber Core* jars. Click on the ***jar***.



3) It will display a pop up and ask you to save the *cucumber core jar* file.

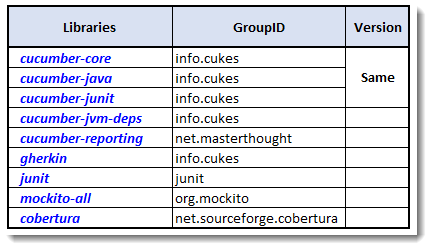


4) Just like this, one by one search for every other jar file mentioned above and downloads these to your drive.



***Note****that****core****,****java****and****junit****files all need to be the****same file version****e.g. 1.2.0*

Also, one most important thing to know is to download the correct Jars. As when you search for these libraries, you will get many options to download. So below chart will help you to install the right libraries from GroupID.

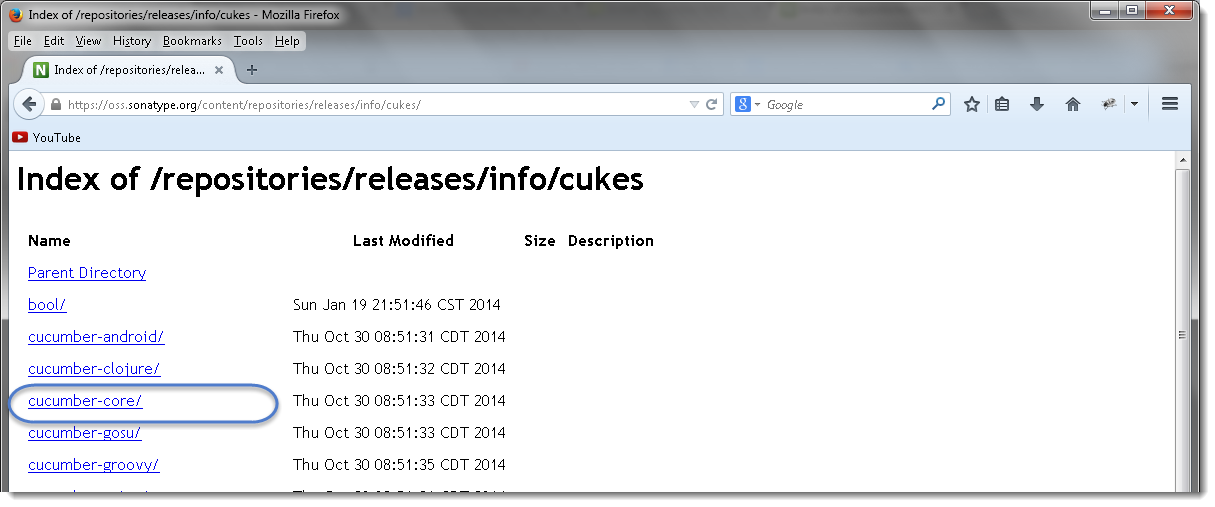


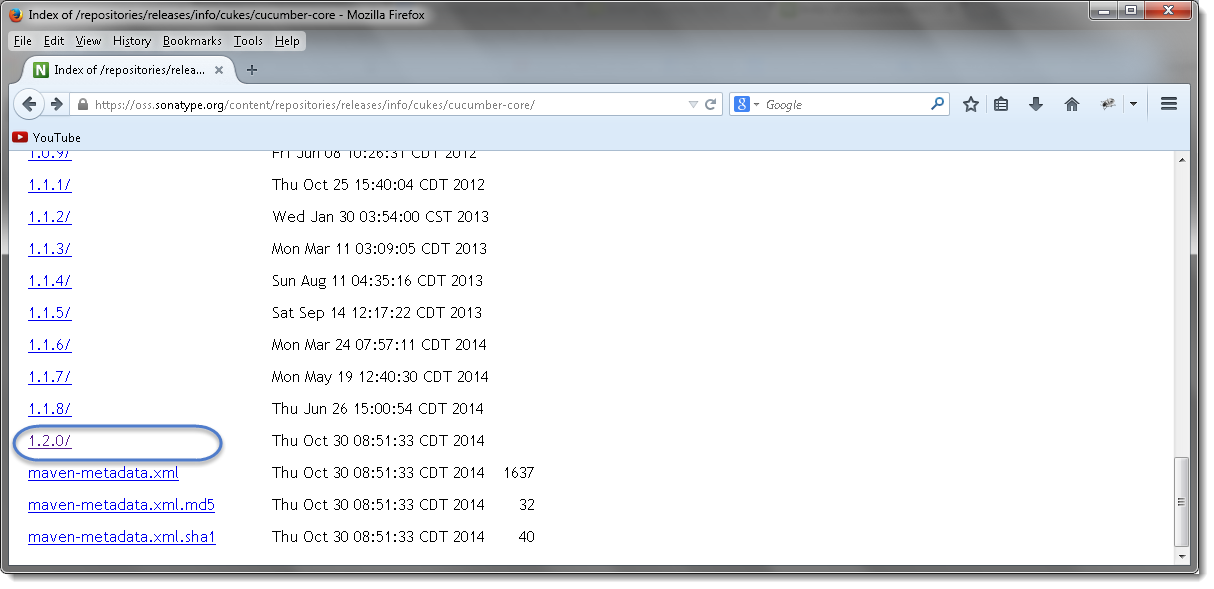
***I got below versions on Oct’17 for Cucumber***

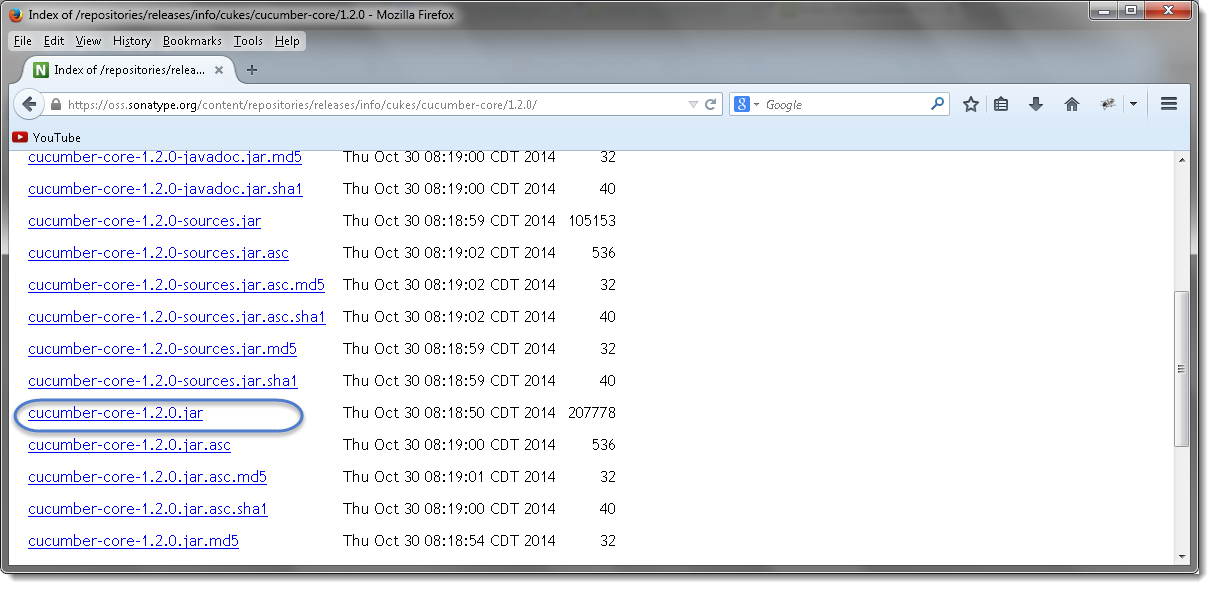
* *cobertura-2.1.1*
* *cucumber-core-1.2.5*
* *cucumber-java-1.2.5*
* *cucumber-junit-1.2.5*
* *cucumber-jvm-deps-1.0.5*
* *cucumber-reporting-3.10.0*
* *gherkin-2.12.2*
* *junit-4.12*
* *mockito-all-2.0.2-beta*

**Option 2 : Download Cucumber Jars from oss.sonatype.org**

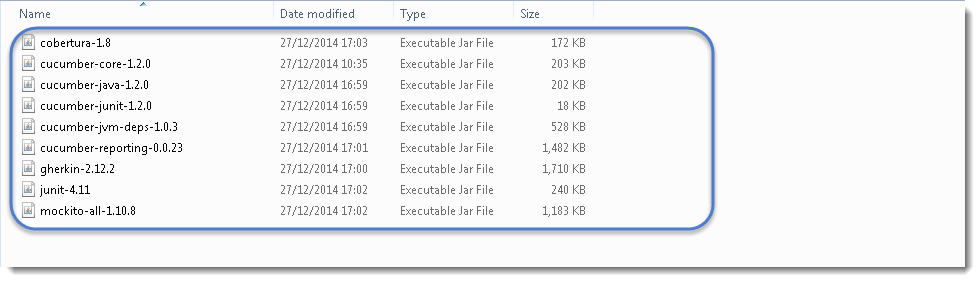
1) Go to [***https://oss.sonatype.org/content/repositories/releases/info/cukes/***](https://oss.sonatype.org/content/repositories/releases/info/cukes/)***.***Here also all the cucumber jars are available. Start with ***cucumber-core****.*



2) Click on the latest version at the bottom of the page, which is ***version 1.2.0*** as of now by Dec’14.  


3) Once clicked on the version, it will display the all types of jar available to download. Click on ***cucumber-core-1.2.0.jar*** .  


4) Just like this, downloads all of these to your drive and every other jar is available on the same page.



***Note****that****core****,****java****and****junit****files all need to be the****same file version****e.g. 1.2.0*

**Option 3 : Download Cucumber Jars from Maven dependencies**

This is the most common and effective way of setting up cucumber with eclipse. But this is bit tricky for the people who do not have much experience with Maven and may end up in wasting a lot of time. There are few prerequisites for setting up cucumber in eclipse.

1. [***Install Maven in Eclipse IDE***](https://toolsqa.com/java/maven/how-to-install-maven-eclipse-ide/)
2. [***Create a New Maven Project in Eclipse***](https://toolsqa.com/java/maven/create-new-maven-project-eclipse/)

Once *Maven* is installed on eclipse and a *Maven* project is created, the next step is to add cucumber dependencies on the project. I have written a nice tutorial on *Maven* and how to add dependencies in to *Maven* project. Please follow the ***Step 4 : Add Dependencies to the Maven Repository*** at [***this article***](https://toolsqa.com/java/maven/configure-selenium-continuous-integration-maven/). The way we have added Selenium dependencies in the article the same way add the below mentioned dependencies in to the *Maven POM*.

Do not forget to add all the dependencies for all the below mentioned jars required for Cucumber set up:

1. ***cucumber-core***
2. ***cucumber-java***
3. ***cucumber-junit***
4. ***cucumber-jvm-deps***
5. ***cucumber-reporting***
6. ***gherkin***
7. ***junit***
8. ***mockito-all***
9. ***cobertura***

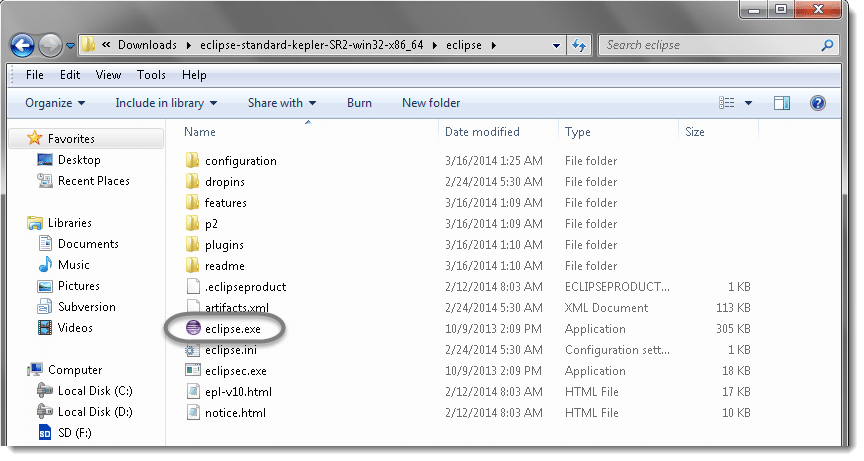
***For example****, Open the****pom.xml****file and Copy the following inside the****dependencies****tag.*



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-java</artifactId>  <version>1.1.5</version>  <scope>test</scope>  </dependency>    <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-jvm</artifactId>  <version>1.1.5</version>  <type>pom</type>  </dependency>    <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-junit</artifactId>  <version>1.1.5</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>net.sourceforge.cobertura</groupId>  <artifactId>cobertura</artifactId>  <version>2.1.1</version>  </dependency>  <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-jvm-deps</artifactId>  <version>1.0.5</version>  </dependency>  <dependency>  <groupId>net.masterthought</groupId>  <artifactId>cucumber-reporting</artifactId>  <version>1.0.0</version>  </dependency>  <dependency>  <groupId>info.cukes</groupId>  <artifactId>gherkin</artifactId>  <version>2.12.2</version>  </dependency>  <dependency>  <groupId>org.mockito</groupId>  <artifactId>mockito-all</artifactId>  <version>2.0.2-beta</version>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>3.8.1</version>  <scope>test</scope>  </dependency> |

### Step 1: Select WorkSpace on Eclipse start up

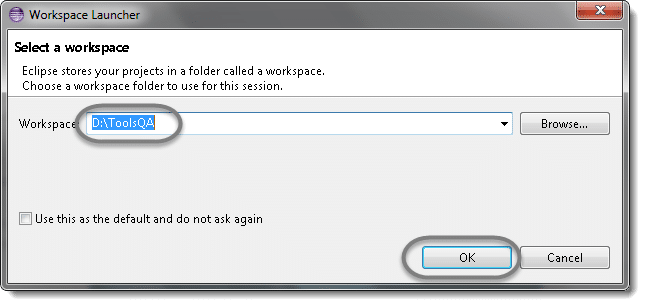
a) ***Double click*** on ‘eclipse.exe‘ to start eclipse. First time when you start eclipse, it will ask you to select your workspace where your work will be stored as shown in below image.

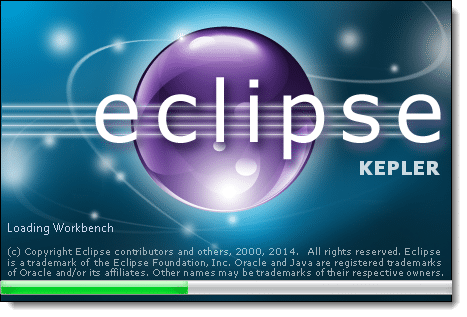


b) Create a “***working directory***” for all of your projects. Think of it like “***My Documents***” in the Windows operating system. It’s a folder that contains a lot of your documents, but there’s nothing to prevent you from creating another folder called “My Other Documents” (for instance) to house other documents.

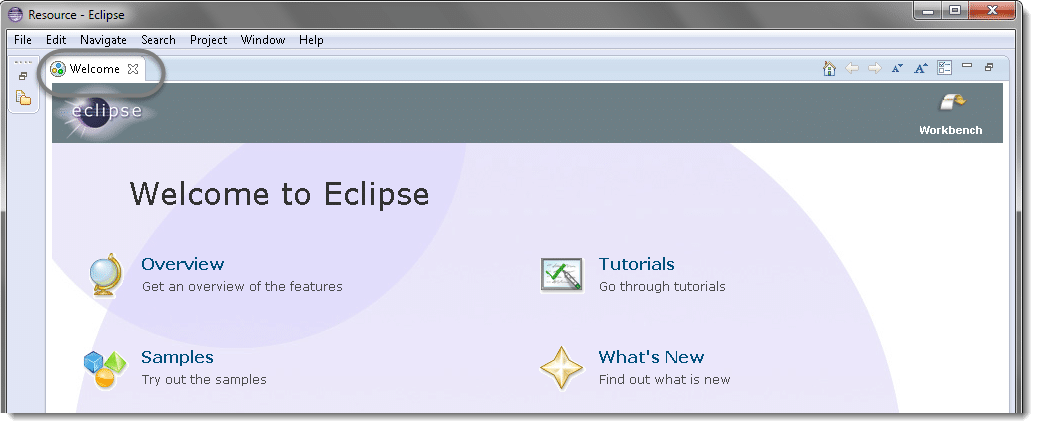
Typically you only need one workspace, and you can think of it as your “My Documents” for Java code. If you wanted to, you could have more than one, but chances are you won’t have a use for more. I like to choose my own workplace location and will place all my ***ToolsQA*** tutorial projects under it.

You can change it later on from ‘***Switch Workspace***‘ under ‘***File***‘ menu of eclipse. After selecting the workspace folder, Eclipse will be open.





c) You may see the window like this, this is the ***Welcome*** window for Eclipse. You may ***close*** this window.

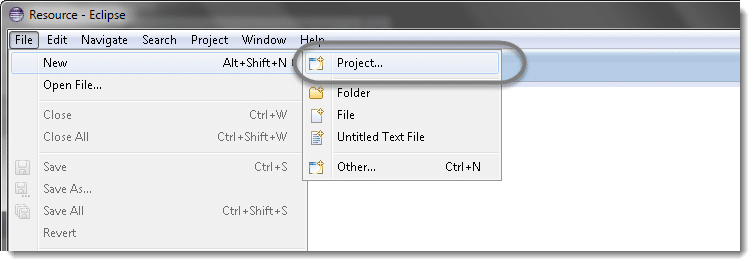


### Step 2: Create a New Project

**Projects:** A collection of related code. Generally speaking, each project encompasses one independent program. Each programming assignment you do will typically require its own project.

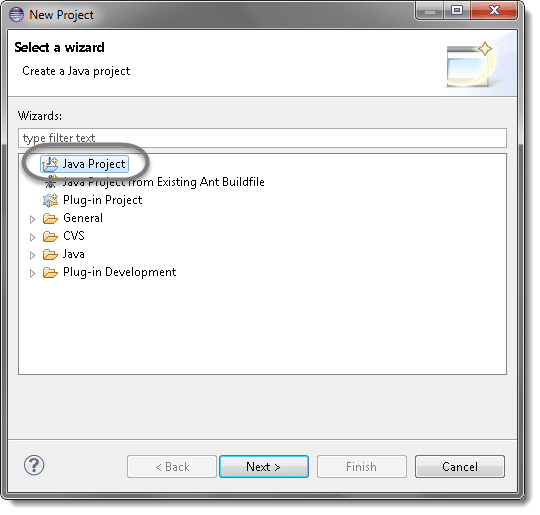
Once you’ve established your workspace, you’ll want to create a project and begin writing code. In Eclipse, projects are the next-smallest functional unit after workspaces, but where you might have only one workspace, you will usually have several projects inside one workspace.

a) Create new Java Project from ***File*** > ***New*** > ***Java Project*** .

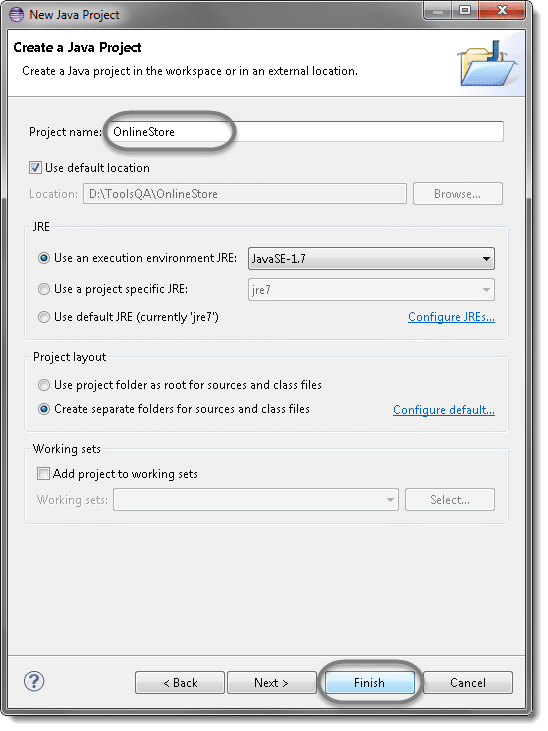


***Note:*** You may see a direct option for Java Project, if you are using the latest version of Eclipse like LUNA.

b) Select **Java Project**and click**Next**.



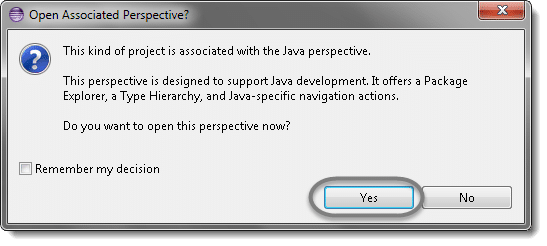
c) Give your Project name ‘**OnlineStore**‘ as shown in below given figures. Click on **Finish** button.



**Note:**I am naming this as ***OnlineStore***, as [***Toolsqa.com***](https://toolsqa.com/) gives you the opportunity to practice automation on [***Live demo site***](https://store.demoqa.com/). This demo website is a complete ecommerce website where a customer can register and purchase electronic items.

***Website :***[***www.store.demoqa.com***](https://www.store.demoqa.com/)

d) You may or may not see this message but if in case you get any, check ‘***Remember my decision’*** and  click on ***Yes***.



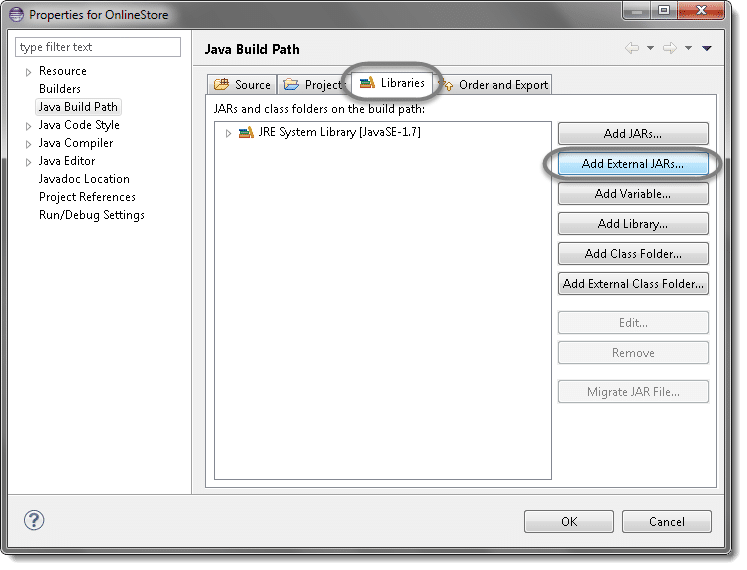
Now your new created project ‘***OnlineStore***‘ will display in eclipse project explorer.

### Step 3: Add External Jars to Java build path

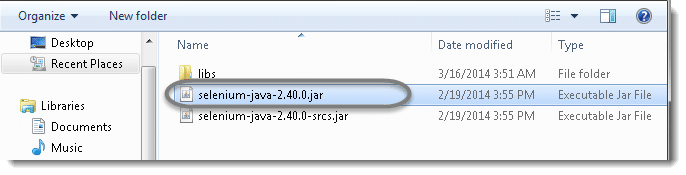
We are almost ready to write the first cucumber selenium test but before that we need to associate all the Selenium & Cucumber jars to the newly created project.

***Add Selenium Jars***

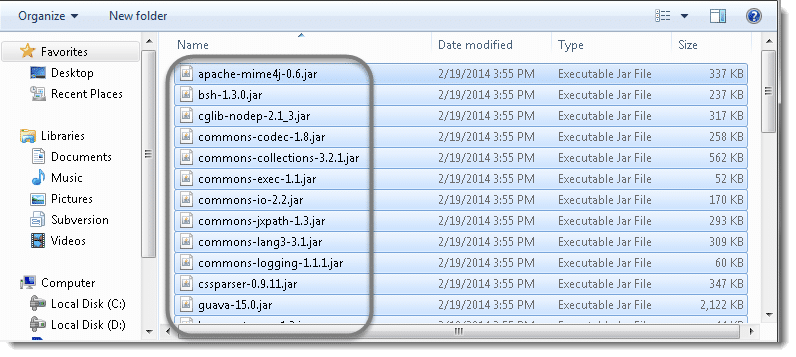
a) Let’s first add Selenium Jars. Right click on Project ‘**OnlineStore’ > Select Properties > Java build path.**Thennavigate to ***Libraries*** tab and click ***Add External JARs***.



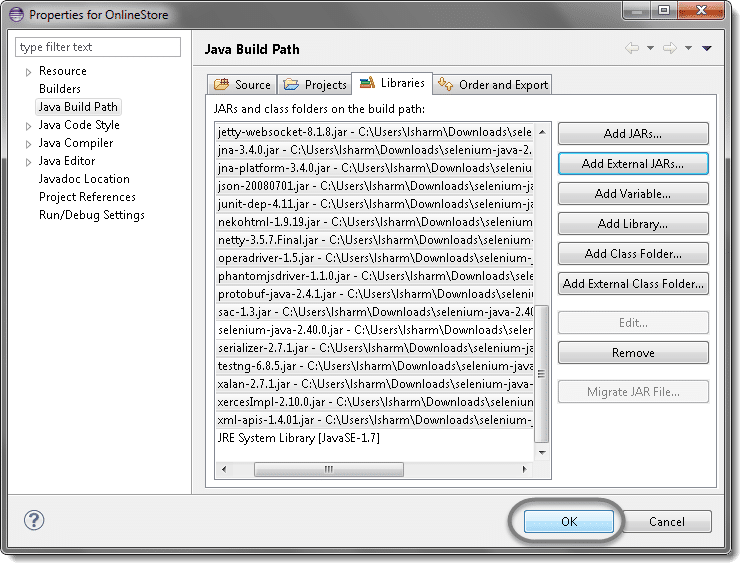
b) Add Selenium Java jar, you may add the source file too.



c) Add all the jars from the **libs** folder as well.

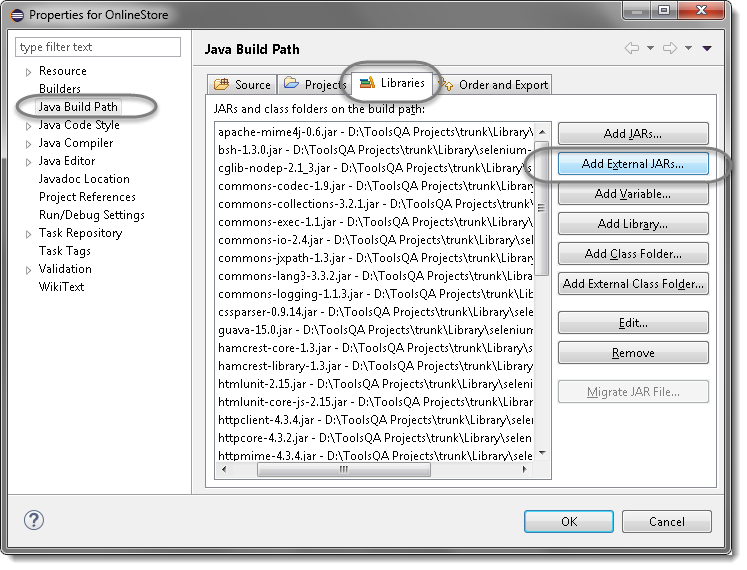


d) Click **OK**.

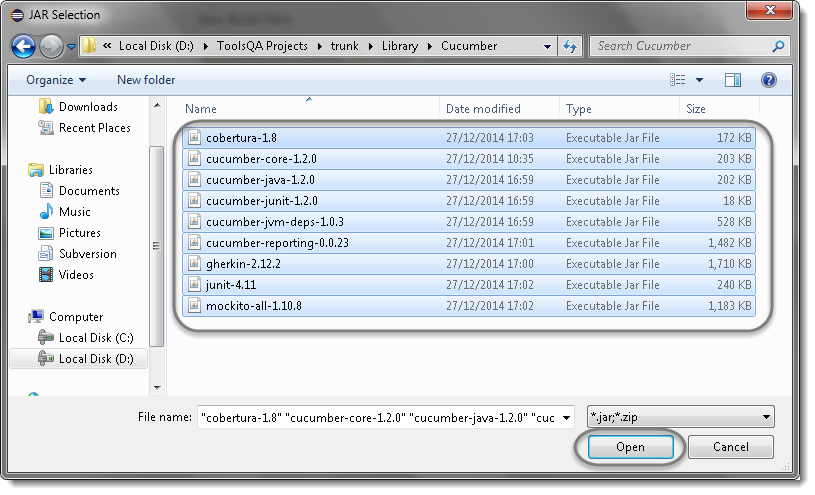


***Add Cucumber Jars***

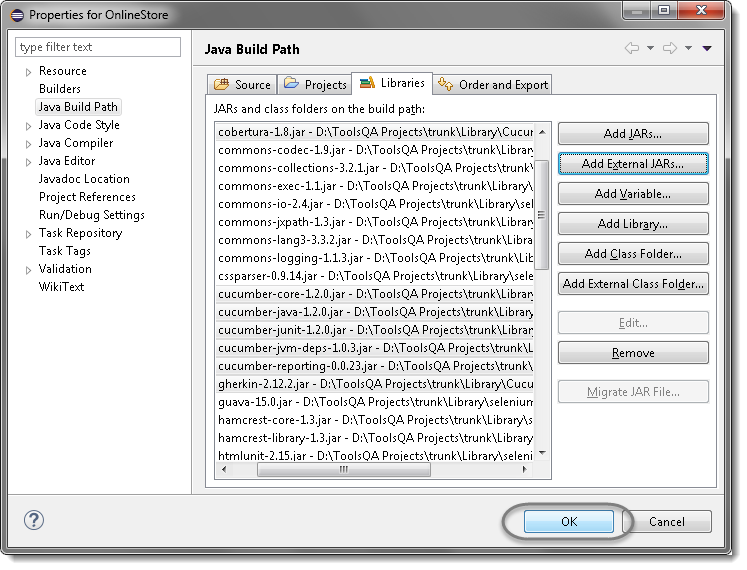
a) Right click on Project ‘**OnlineStore’ > Select Properties > Java build path.**Thennavigate to ***Libraries*** tab and click ***Add External JARs***.



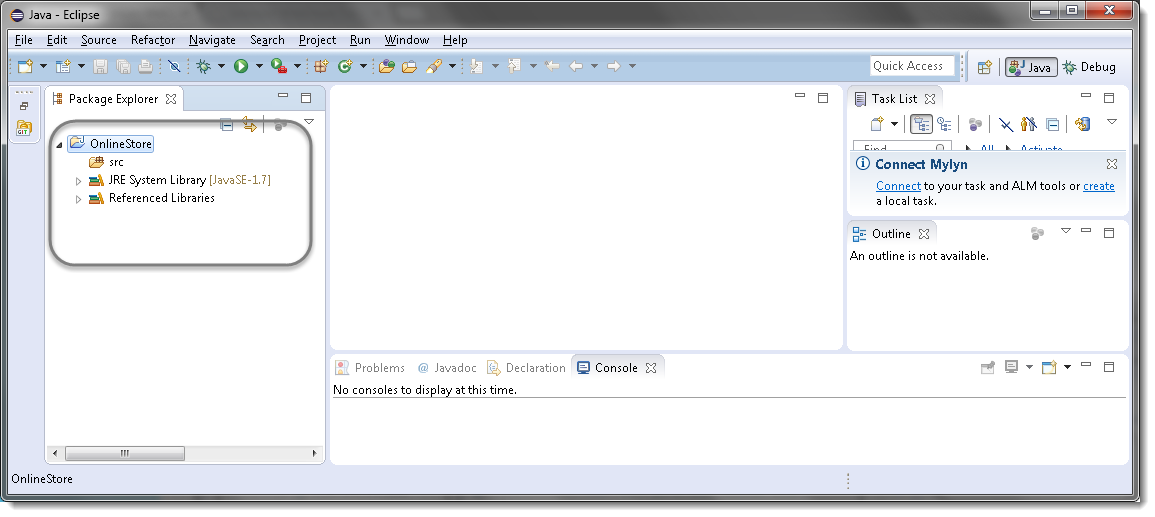
b) Add all the **Cucumber’s Jars** to the project and click **Open**.



c) Click on **Ok**.



d) This is how the project looks like in Eclipse as of now.



That’s all about configuration of Cucumber & Selenium WebDriver with Eclipse. Now you are ready to write your test script in eclipse.

Till now we have understood what ***Cucumber*** is and what development model it follows. Now let’s try to create first Cucumber Selenium Java test and I assume that you have some basic understanding of ***Selenium WebDriver*** and its basic commands.

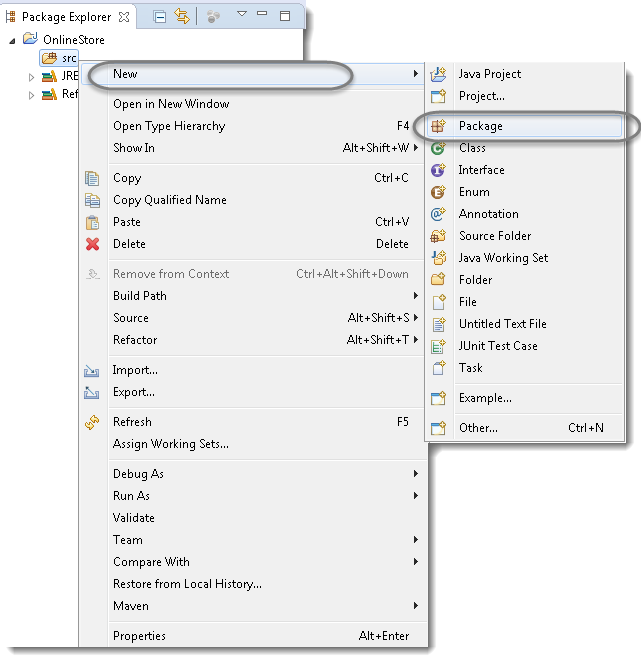
I hope you have been following the complete tutorial and I expect that by now you have completed the following steps, which are the prerequisite for writing Cucumber Selenium test:

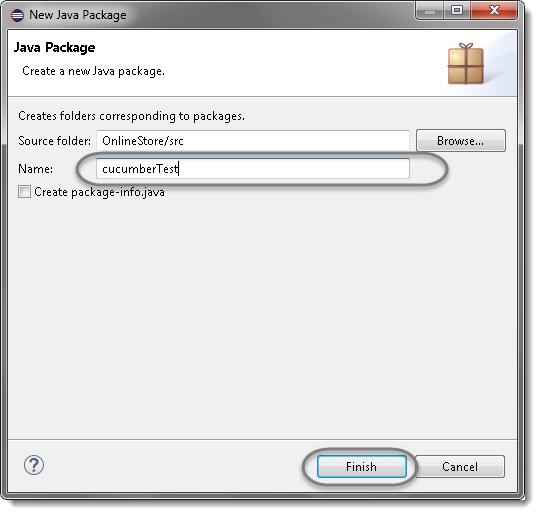
1. [***Download & Install Java***](https://toolsqa.com/selenium-webdriver/download-and-install-java/)
2. [***Download and Install Eclipse***](https://toolsqa.com/selenium-webdriver/download-and-start-eclipse/)
3. [***Install Cucumber Eclipse Plug-in***](https://toolsqa.com/cucumber/install-cucumber-eclipse-plugin/)
4. [***Download Cucumber***](https://toolsqa.com/cucumber/download-cucumber-eclipse/)
5. [***Download Selenium WebDriver Client***](https://toolsqa.com/selenium-webdriver/download-webdriver-java-client/)
6. [***Configure Eclipse with Selenium & Cucumber***](https://toolsqa.com/cucumber/configure-eclipse-cucumber/)

### Create Folder Structure

Before moving head for writing the first script, let’s create a nice folder structure of the project.

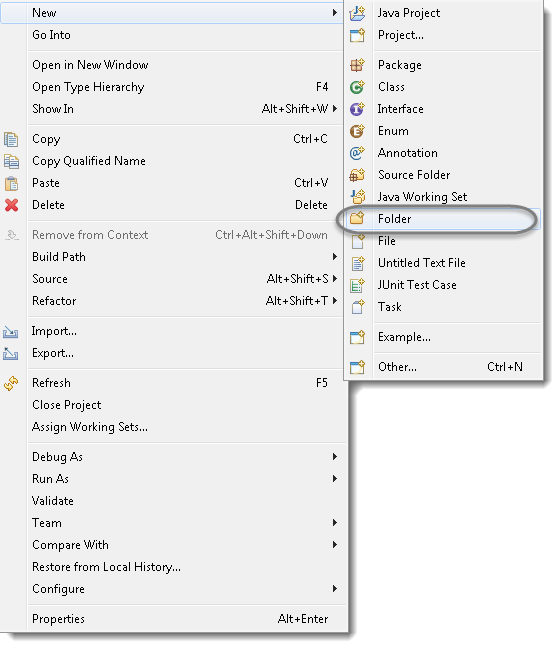
1) Create a new ***Package*** by right click on the ‘***src***‘ folder and select New > Package.



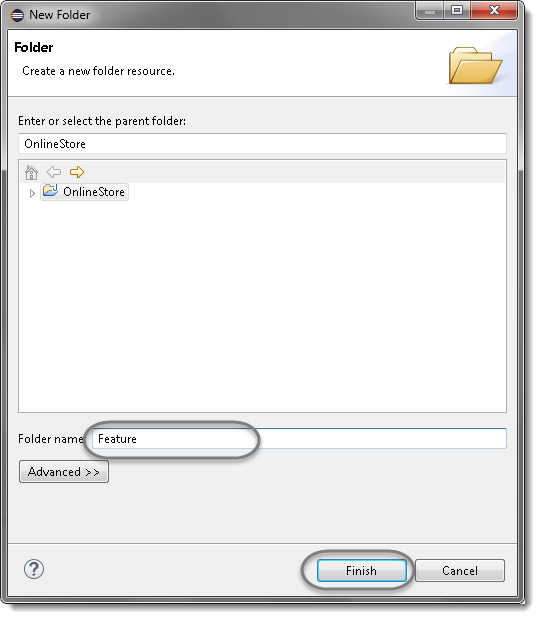
2) Name it as ‘***cucumberTest***’ and click on Finish button.  


3) Create another ***Package*** and name it as ‘***stepDefinition***’, by right click on the ‘***src***‘ folder and select New > Package.

4) Create a new ***Folder*** this time by right click on the project ‘***OnlineStore***‘ and select New > Folder.



5) Name it as ‘***Feature***’ and click on Finish button.

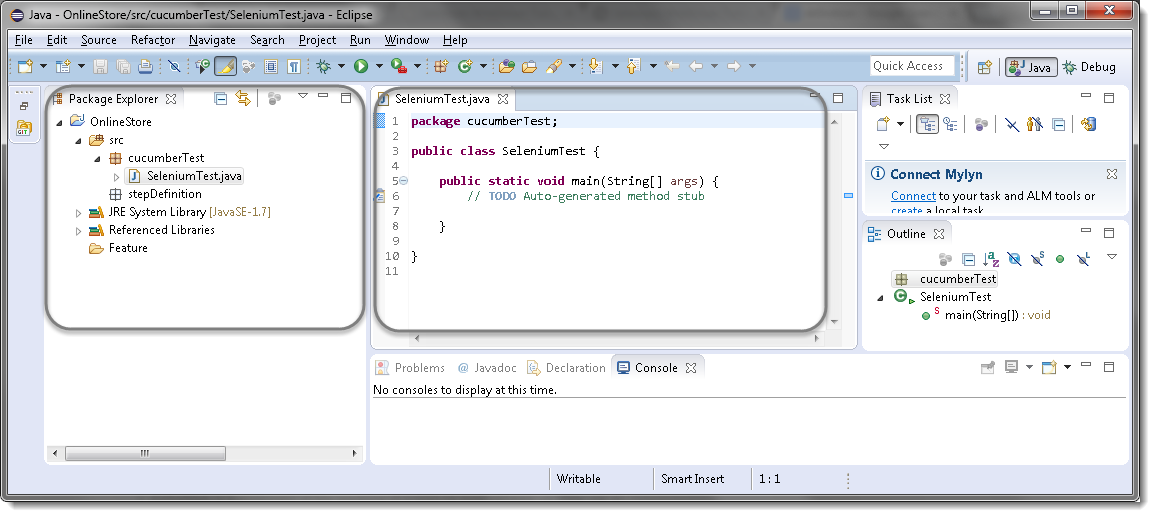


## Selenium Java Test

Lets first write a simple ***Selenium Test script*** for ***LogIn*** functionality and then convert that script into Cucumber script to understand it better.

1) Create a new ***Class*** file in the ‘***cucumberTest***‘ package and name it as ‘***SeleniumTest***‘, by right click on the Package and select**New > Class**. Check the option ‘***public static void main***‘ and click on ***Finish*** button.

Now the Eclipse Window must look like this:



### Selenium Test Script

Now write a simple script performing the following steps in Selenium.

1. Launch the Browser
2. Navigate to Home Page
3. Click on the LogIn link
4. Enter UserName and Password
5. Click on Submit button
6. Print a successful message
7. LogOut from the application
8. Print a successful message
9. Close the Browser

***Selenium Test Script***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | package cucumberTest;    import java.util.concurrent.TimeUnit;    import org.openqa.selenium.By;  import org.openqa.selenium.WebDriver;  import org.openqa.selenium.firefox.FirefoxDriver;    public class SeleniumTest {  private static WebDriver driver = null;  public static void main(String[] args) {  // Create a new instance of the Firefox driver            driver = new FirefoxDriver();            //Put a Implicit wait, this means that any search for elements on the page could take the time the implicit wait is set for before throwing exception            driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);            //Launch the Online Store Website            driver.get("https://www.store.demoqa.com");            // Find the element that's ID attribute is 'account'(My Account)            driver.findElement(By.xpath(".//\*[@id='account']/a")).click();            // Find the element that's ID attribute is 'log' (Username)            // Enter Username on the element found by above desc.            driver.findElement(By.id("log")).sendKeys("testuser\_1");            // Find the element that's ID attribute is 'pwd' (Password)            // Enter Password on the element found by the above desc.            driver.findElement(By.id("pwd")).sendKeys("Test@123");            // Now submit the form. WebDriver will find the form for us from the element            driver.findElement(By.id("login")).click();            // Print a Log In message to the screen            System.out.println("Login Successfully");            // Find the element that's ID attribute is 'account\_logout' (Log Out)            driver.findElement (By.xpath(".//\*[@id='account\_logout']/a")).click();            // Print a Log In message to the screen            System.out.println("LogOut Successfully");            // Close the driver            driver.quit();    }    } |

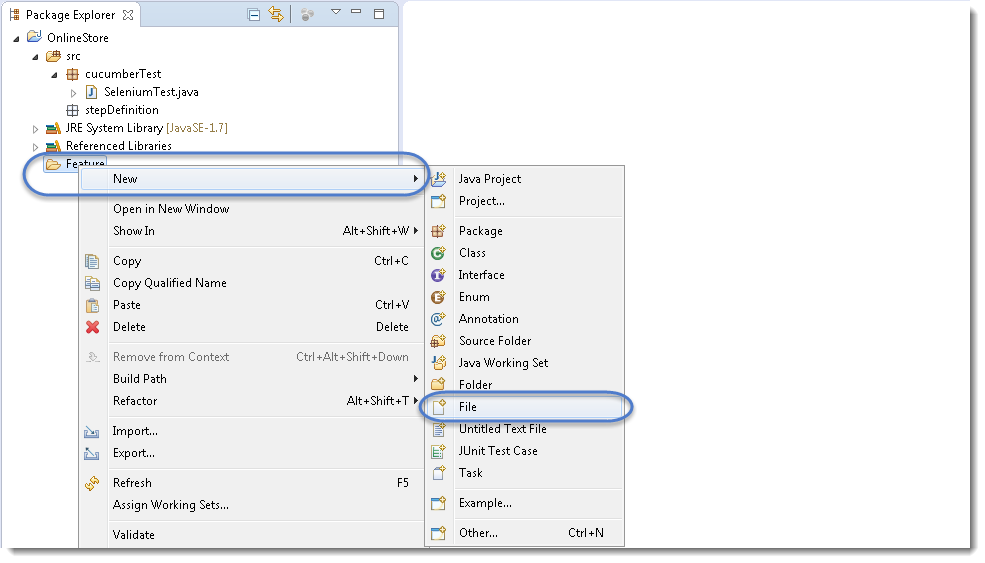
***Note***: If the Selenium version is less than 3.0, above test will work for you. If the version is above 3.0, in that case, please look at the chapter [***How to Use Gecko Driver in Selenium 3***](https://toolsqa.com/selenium-webdriver/how-to-use-geckodriver/)

Now, to start the test just select ***Run*** > ***Run As*** > ***Java Application***Or Right Click on Eclipse code and Click ***Run As***  > ***Java Application.*** After a few seconds, a Mozilla browser will open and you will see that with the help of your script, Selenium will Launch the Online Store demo application, perform ***Sign in.***

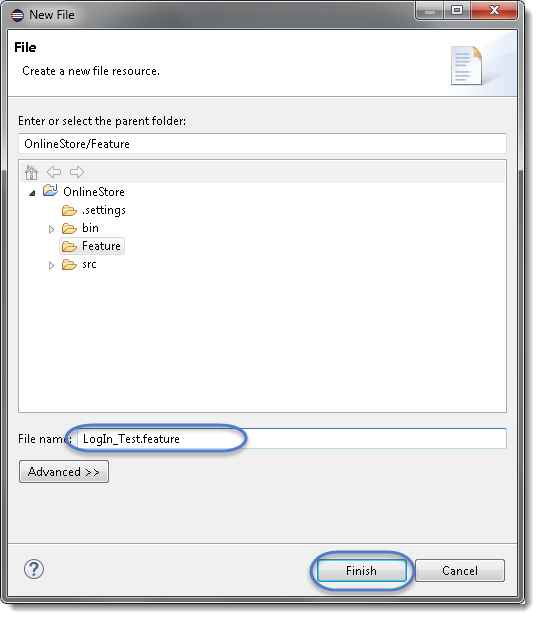
## What is Cucumber Feature File?

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.

1) On the ***Feature*** folder Right-click and select ***New > File***



2) In order for Cucumber to automatically detect the stories (or ***features***, as they’re known in Cucumber), you need to make sure that they carry the ‘***.feature***‘ file extension. For example, in this case, I’ve named my user story ‘***LogIn\_Test.feature***‘. Every ‘.feature‘ file conventionally consists of a single feature.



***Note***: In case you get a pop up from Eclipse which suggests you to install the better Editor for BDD files, please go ahead and install that. At the bottom of the chapter, steps to install the better editor is given.

3) Write the first cucumber script. In BDD terms the scenario would look like the following.

***Cucumber Test Script***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Feature: Login Action    Scenario: Successful Login with Valid Credentials  Given User is on Home Page  When User Navigate to LogIn Page  And User enters UserName and Password  Then Message displayed Login Successfully    Scenario: Successful LogOut  When User LogOut from the Application  Then Message displayed LogOut Successfully |

***Note:***This is a simple test in Cucumber. Don’t worry about the syntax if you don’t understand it. Ideally, you should be able to understand the intent of the test just by reading a test in feature file. We will discuss this in more detail in the next chapter.

### ***Keywords***

Now moving forward we have just defined a test. You will notice colored parts of the tests (**Feature, Scenario, Given, When, And and Then**). These are keywords defined by **Gherkin**. Gherkin has more keywords and we will discuss those in the following tutorials. But to start off we can quickly explain some of the keywords in one line. Note this is not a complete listing of Keywords:

**Feature: Defines what feature you will be testing in the tests below**

**Given: Tells the pre-condition of the test**

**And: Defines additional conditions of the test**

***Then: States the post condition. You can say that it is the expected result of the test***.

### Gherkin

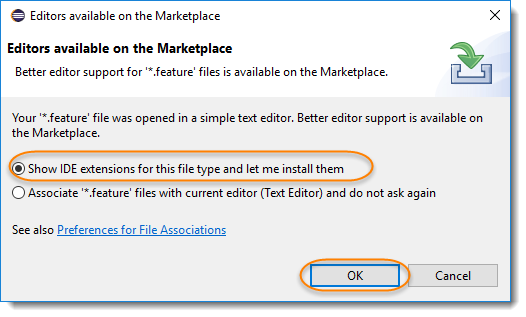
A language above is called ***Gherkin*** and it implements the principles of ***Business readable domain specific language(BRDSL)***. Domain-specific language gives you the ability to describe your application behavior without getting into details of implementation. What does that mean? If we go back to our tutorial in [***TDD***](https://toolsqa.com/cucumber/test-driven-development-tdd/) we saw that we wrote test code before writing any application code. In a way, we described what is the expected behavior of our application in terms of tests. On TDD those tests were pure Java tests, in your case, those might be a C++ or C# tests. But the basic idea is that those are core technical tests.

If we now come back to [***BDD/BRDSL***](https://toolsqa.com/cucumber/behavior-driven-development/) we will see that we are able to describe tests in a more readable format. In the above test, it’s quite clear and evident, just by reading, what test would do. At the same time of being a test it also documents the behavior of an application. This is the true power of BDD/BRDSL and it will become the power of cucumber eventually because cucumber works on the same principles.

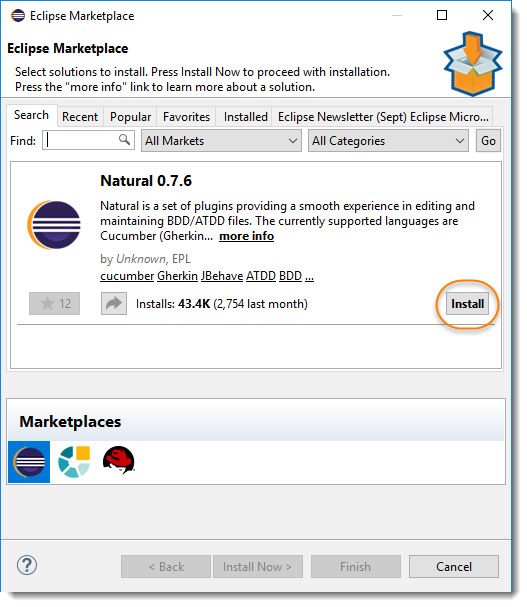
### ***Steps to install the Natural Eclipse Editor for Gherkin***

You get this option automatically when try to create a new file with .feature ext. But if you do not get that one, you can anytime go to Eclipse Marketplace and look for the same to install it.

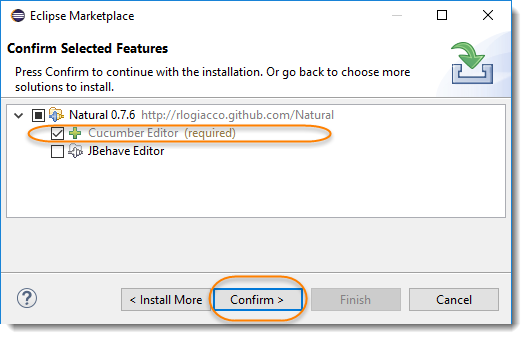
1)Just select the first option of ***Show IDE extensions*** if it is not pre-selected and click OK.



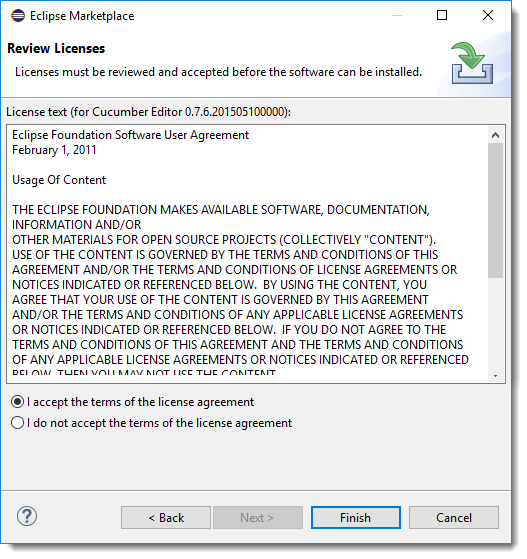
2) Natural is the name of the plugin, so this can also be found [***Eclipse Marketplace***](https://marketplace.eclipse.org/content/natural#group-details). Just click Install.



3) This will give you an option to select, whether you like to use it for Cucumber or JBehave(Another BDD Framework). Go for Cucumber.



4) The last step is to accept the Terms and Conditions.



***Note***: Once done, it may ask you to restart the Eclipse, if not then it is suggested to restart the eclipse after installing any plugins.

## JUnit Test Runner Class

Create a new ***Class*** file in the ‘***cucumberTest***‘ package and name it as ‘***TestRunner***‘, by right click on the Package and select ***New > Class.***This class just needs annotations to understand that cucumber features would be run through it and you can specify feature files to be picked up plus the steps package location. There are bunch of other parameters that it can take, to be discussed later in [***Cucumber Options***](https://toolsqa.com/cucumber/options/).

***Test Runner Class***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package cucumberTest;    import org.junit.runner.RunWith;  import cucumber.api.CucumberOptions;  import cucumber.api.junit.Cucumber;    @RunWith(Cucumber.class)  @CucumberOptions(  features = "Feature"  ,glue={"stepDefinition"}  )    public class TestRunner {    } |

For curious minds, I will explain this code. Note that it is covered in detail in coming tutorials. Consider this as a limited description.

### 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.

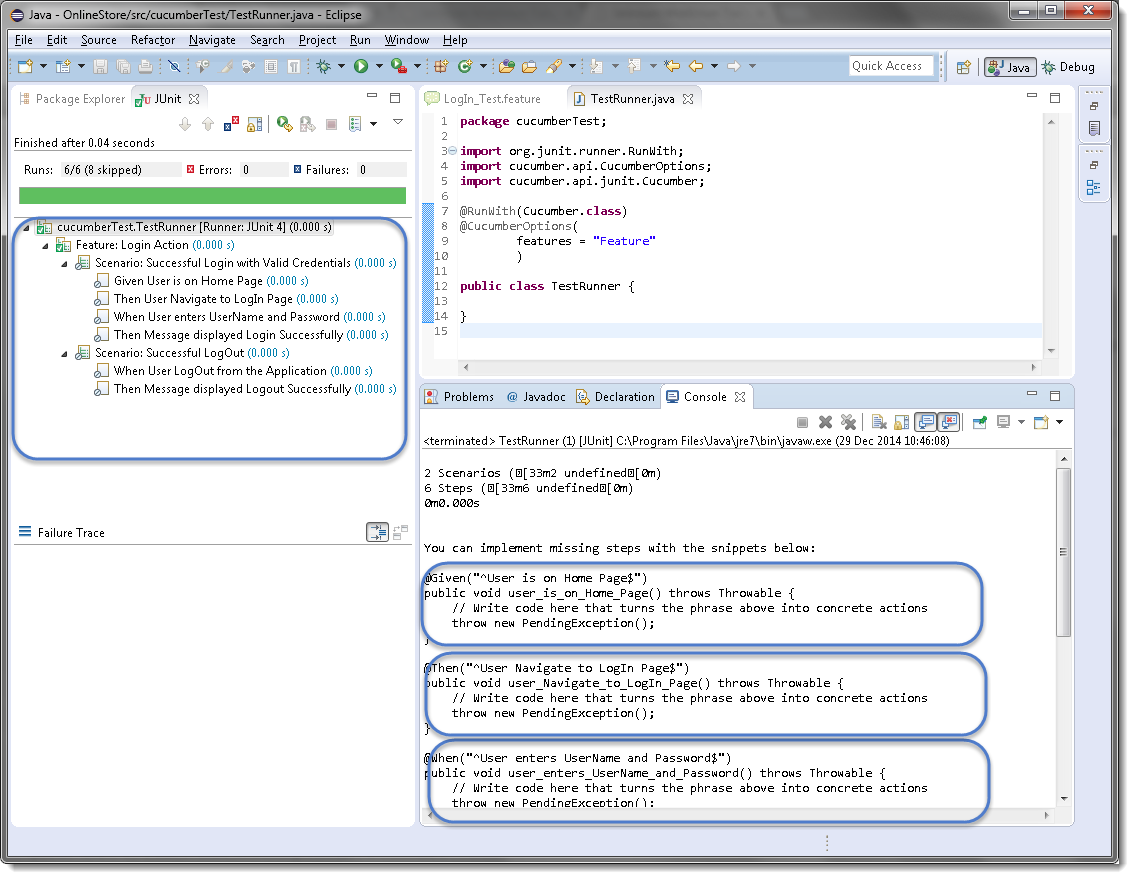
### Run the Cucumber Test

Now we are all set to run the first Cucumber test. There are multiple ways and runners to use when it comes to cucumber feature files. We would try to understand how to run it from the IDE first and then from a command line at a later point.

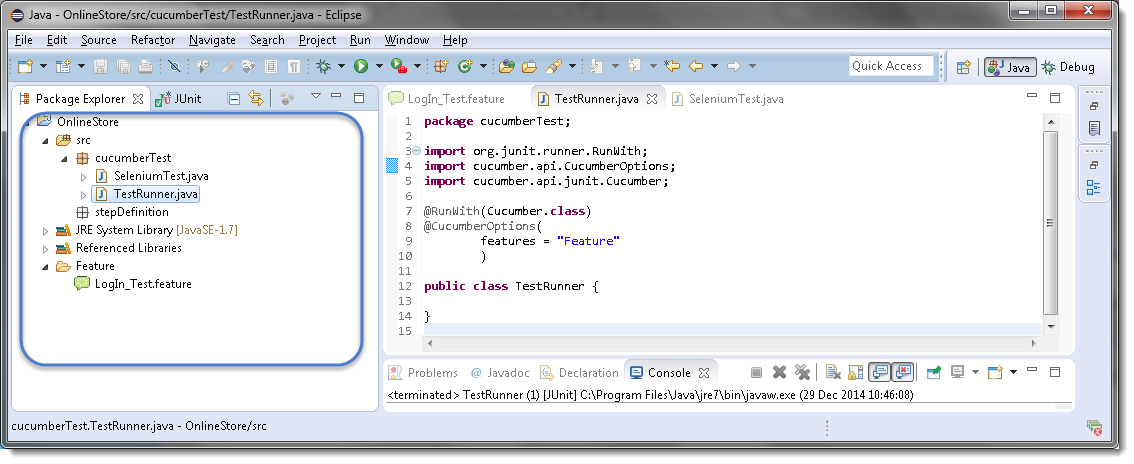
Even from the IDE, there are a couple of ways to run these feature files.

* 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***

You will think where is the java code that will execute for these tests? Well, don’t worry about that at this moment. Let’s just see what we have on the console window. Here is the text that I got on my console. Look how Cucumber has suggested that you should implement these methods so that the Steps mentioned in the Feature file can be traced to Java methods, which can be executed while executing the feature file.



Now your project should look like this in Eclipse IDE:



### Errors on running Cucumber Feature

Exception in thread “main” cucumber.runtime.CucumberException: No backends were found. Please make sure you have a backend module on your CLASSPATH.

***Solution***

Most probably this means that your ***cucumber-java*** version and ***java*** version on your machine is not compatible with each other.  First, check Java Version on your machine by going through this article [***How to check Java/JDK Version Installed on your Machine.***](https://toolsqa.com/java/check-java-version-installed-windows-machine/)

On my machine, I have Java 1.8.0 with ***cucumber-Java8-1.2.5*** and it did not work. When I degraded my cucumber java version to ***cucumber-Java-1.2.5,***it worked fine for me. Just make sure that first, you remove the cucumber-java which did not work for you from ***Project build path >> Libraries*** and then add new. Keeping both may create further issues for you.

Exception in thread “main” java.lang.NoClassDefFoundError: gherkin/formatter/Formatter

***Solution***

This means that the Gherkin version you are using is not compatible with other Cucumber libraries. I tried using the latest ***gherkin3-3.0.0*** but it did not work for me, so I degraded it to ***gherkin-2.12.2***

***Gherkin*** is not necessarily used to write automated tests. *Gherkin* is primarily used to write ***structured***tests which can later be used as project documentation. The property of being *structured* gives us the ability to automate them. This automation is done by ***Cucumber/SpecFlow***. In the [***Gherkin – Business Driven Development***](https://toolsqa.com/cucumber/gherkin-business-driven-development-bdd-language/) we saw a simple Gherkin Keyword test and why *Gherkin* is important to use.

***Note:****Cucumber/SpecFlow understands Gherkin hence we can say that this is a Cucumber/SpecFlow test.*

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

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

You will quickly notice that there are some colored words. These words are *Gherkin keywords* and each keyword holds a meaning. Now we will discuss these keywords one by one. Here is the list of keywords that *Gherkin* supports:

* ***Feature***
* ***Background***
* ***Scenario***
* ***Given***
* ***When***
* ***Then***
* ***And***
* ***But***
* ***\****

**Feature: Keyword**

Each *Gherkin* file begins with a ***Feature*** keyword. *Feature* defines the logical test functionality you will test in this feature file. For e.g, if you are testing a payment gateway your *Feature* will become *Payment Gateway* or if you are testing the *LogIn* functionality then the *Feature* will become *Login*. The idea of having a feature file is to put down a summary of what you will be testing. This will serve as the documentation for your tests as well as a good point to start for a new team member. Note that a feature keyword is present at the starting of the feature file.

***Feature****: LogIn Action Test*

*Or*

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

*Or*

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

Notice that whatever comes after the ***Feature: keyword,***will be considered as the feature description. Feature description can span across multiple lines as shown above in the second example. Everything after *Feature:* till the next Keyword is encountered is considered as feature description.

***Note:****Description is not a keyword of Gherkin.*

Take a look at the example of [***Cucumber Feature***](https://toolsqa.com/cucumber/cucumber-jvm-feature-file/) file and ***[SpecFlow Feature](https://toolsqa.com/specflow/feature-file/)*** file

**Background: Keyword**

***Background***keyword is used to define steps that are common to all the tests in the feature file. For example, to purchase a product, you need to do the following steps:

* *Navigate to Home Page*
* *Click on the LogIn link*
* *Enter UserName and Password*
* *Click on Submit button*

After these steps only you will be able to add a product to your *cart/basket* and able to perform the payment. Now as we are in a feature file where we will be testing only the *Add to Cart* functionality, these tests become common for all tests. So instead of writing them again and again for all tests, we can move it under the background keyword. This is how it will look like:

***Feature****: Add to Cart*  
*This feature will test functionality of adding different products to the User basket from different flow*

***Background:****User is Logged In*

***Scenario****: Search a product and add the first result/product to the User basket*  
***Given****User searched for Lenovo Laptop*  
***When****Add the first laptop that appears in the search result to the basket*  
***Then****User basket should display with 1 item*

Take a look at the example of [***Cucumber Background***](https://toolsqa.com/cucumber/background-in-cucumber/)

**Scenario: Keyword**

Each Feature will contain a number of tests to test the feature. Each test is called a ***Scenario***and is described using the *Scenario:* keyword.

***Scenario****: Search a product and add the first result/product to the User basket*

*Or*

***Scenario****: Successful LogIn with Valid Credentials*

A scenario is equivalent to a test in our regular development process. Each scenario/test can be basically broken down into three parts:

* ***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****)*

**Given Keyword**

***Given***defines a precondition to the test. For e.g. In the shopping website, assume that the *LogIn page* *link* is only present on the Home Page, so the precondition for clicking the*LogIn link* is that the user is at the Home Page. If user is not at the Home Page, user would not be able to enter *Username* & *Password*. This precondition can be expressed in *Gherkin* like this:

***Scenario****: Successful LogIn with Valid Credentials*

***Given****User is on Home Page*  
***When****User Navigate to LogIn Page*

**When Keyword**

***When*** keyword defines the test action that will be executed. By test action we mean the user input action.

***Scenario****: Successful LogIn with Valid Credentials*

***Given****User is on Home Page*  
***When****User Navigate to LogIn Page*

Here user is performing some action using *When* keyword, clicking on the LogIn link. We can see that when defines the action taken by the user. It’s the event that will cause the actual change in state of the application.

**Then Keyword**

***Then***keyword defines the Outcome of previous steps. We can understand it best by looking at the test above and adding a Then step there.

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

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

Here we can see that ***Then***is the outcome of the steps above. The reader of this test would easily be able to relate to *Then* step and would understand that when the above conditions are fulfilled then the *Then* step will be executed.

**And Keyword**

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

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

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

Or

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

***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*

Here you would see that *And* is being used to add more details to the *Given* step, it’s simply adding more conditions. We have just added three conditions. Use it when you have specified more than one condition. *And* is used to add more conditions to *Given*, *When* and *Then* statements.

**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:

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

***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*

Here you can see how adding ***But***has helped define a negative test, in this test we will try to test failure conditions. Where a wrong credentials are a failure condition.

**\* 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:

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

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

***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*

Here we conclude the tutorial on different keywords of Cucumber. I hope you like it. Let’s now jump deep into how to execute these steps with the help of Step Definition file.

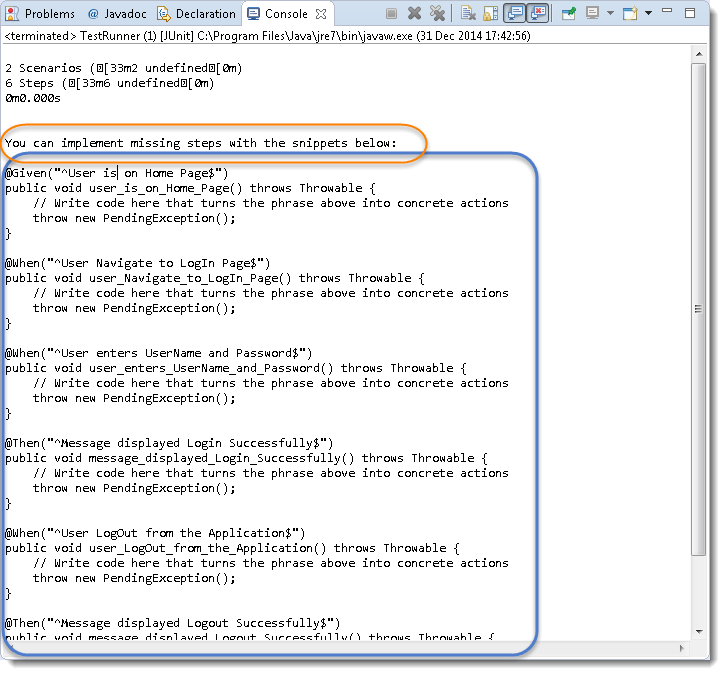
**What is Step Definition?**

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.

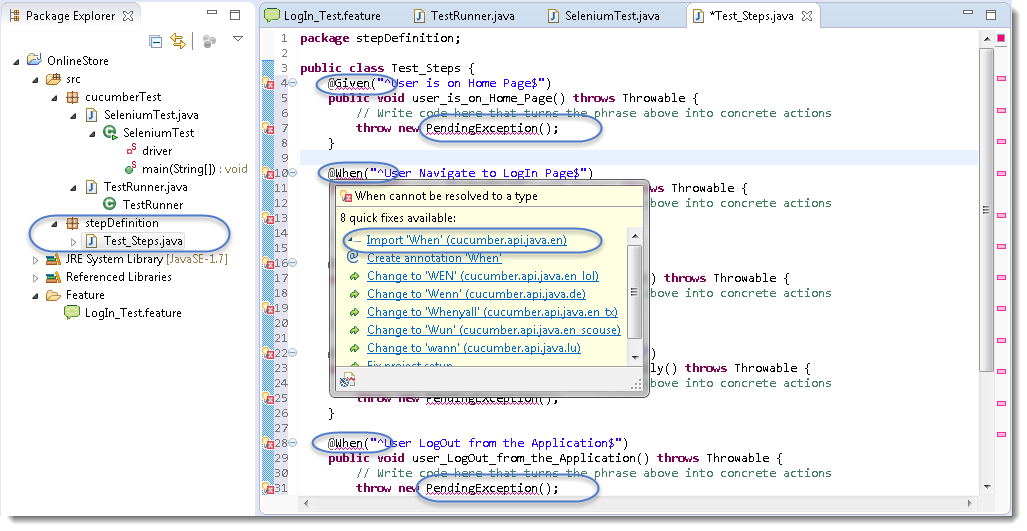
2) Take a look at the message in the console window. This message was displayed, when we ran the ***Test\_Runner*** class.



***Note:****Please go through the chapter of*[***First Cucumber Selenium Test***](https://toolsqa.com/cucumber/first-cucumber-selenium-java-test/)*to understand the above message.*

2) 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.

3) As of now, the test will show many errors on ‘***@***‘ ***annotations***. Mouse hover at the annotations and import the ‘***cucumber.api.java.en***‘ for all the annotations.



**Add Selenium Java code in the Step Definition methods**

1) Now take out the Selenium Java code of the following steps from the ‘***SeleniumTest***‘ and paste it into the first method ‘***@Given(“^User is on Home Page$”)***‘.

* Launch the Browser
* Navigate to Home Page

Method will look like this now:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Given("^User is on Home Page$")  public void user\_is\_on\_Home\_Page() throws Throwable {  driver = new FirefoxDriver();          driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);          driver.get("https://www.store.demoqa.com");  } |

2) Take out the code take out the Selenium Java code of the following steps from the ‘***SeleniumTest***‘ and paste it into the second method ‘***@When(“^User Navigate to LogIn Page$”)***‘.

* Click on the LogIn link

Method will look like this now:



|  |  |
| --- | --- |
| 1  2  3  4 | @When("^User Navigate to LogIn Page$")  public void user\_Navigate\_to\_LogIn\_Page() throws Throwable {  driver.findElement(By.xpath(".//\*[@id='account']/a")).click();  } |

3) Take out the code take out the Selenium Java code of the following steps from the ‘***SeleniumTest***‘ and paste it into the second method ‘***@When(“^User enters UserName and Password$”)***‘.

* Enter UserName and Password
* Click on Submit button

Method will look like this now:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @When("^User enters UserName and Password$")  public void user\_enters\_UserName\_and\_Password() throws Throwable {  driver.findElement(By.id("log")).sendKeys("testuser\_1");      driver.findElement(By.id("pwd")).sendKeys("Test@123");      driver.findElement(By.id("login")).click();  } |

4) Do the same steps for the rest of the methods as well and complete Test\_Steps class will look like this:

***Step Definition: Test\_Steps Class***

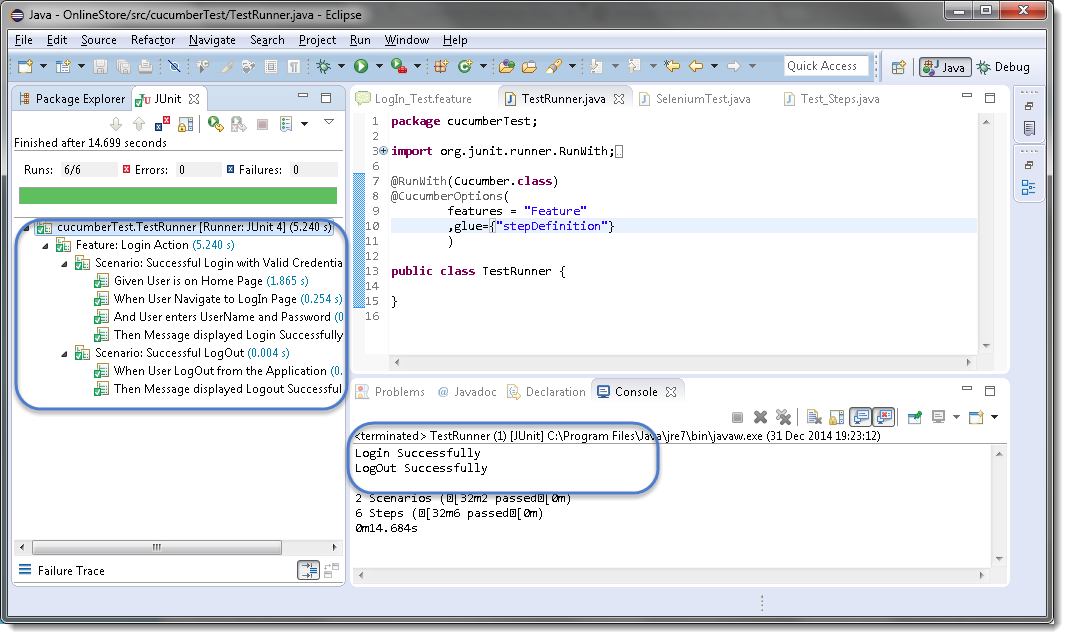


|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | package stepDefinition;    import java.util.concurrent.TimeUnit;    import org.openqa.selenium.By;  import org.openqa.selenium.WebDriver;  import org.openqa.selenium.firefox.FirefoxDriver;    import cucumber.api.java.en.Given;  import cucumber.api.java.en.Then;  import cucumber.api.java.en.When;    public class Test\_Steps {  public static WebDriver driver;  @Given("^User is on Home Page$")  public void user\_is\_on\_Home\_Page() throws Throwable {          driver = new FirefoxDriver();          driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);          driver.get("https://www.store.demoqa.com");  }    @When("^User Navigate to LogIn Page$")  public void user\_Navigate\_to\_LogIn\_Page() throws Throwable {  driver.findElement(By.xpath(".//\*[@id='account']/a")).click();  }    @When("^User enters UserName and Password$")  public void user\_enters\_UserName\_and\_Password() throws Throwable {  driver.findElement(By.id("log")).sendKeys("testuser\_1");      driver.findElement(By.id("pwd")).sendKeys("Test@123");      driver.findElement(By.id("login")).click();  }    @Then("^Message displayed Login Successfully$")  public void message\_displayed\_Login\_Successfully() throws Throwable {  System.out.println("Login Successfully");  }    @When("^User LogOut from the Application$")  public void user\_LogOut\_from\_the\_Application() throws Throwable {  driver.findElement (By.xpath(".//\*[@id='account\_logout']/a")).click();  }    @Then("^Message displayed Logout Successfully$")  public void message\_displayed\_Logout\_Successfully() throws Throwable {          System.out.println("LogOut Successfully");  }    } |

***Note****: Make sure to create your own Username and Password for the test and do not attempt to login with wrong credentials, as you will be blocked for few hours then on demo website.*

**Run the Cucumber Test**

Now we are all set to run the first Cucumber test. *Right Click* on ***TestRunner*** class and Click ***Run As****>****JUnit Test.****Cucumber* will run the script the same way it runs in *Selenium WebDriver* and the result will be shown in the left hand side*project explorer window* in *JUnit* tab.



*Cucumber* starts it’s execution by reading the*feature file steps*. As soon as *Cucumber* reaches the first step for e.g. *Given* the statement of *Scenario*, it looks for the same statement in the *Step Definition* file, the moment it finds the statement, it executes the piece of code written inside the function.

## What is Cucumber Options?

In layman language, ***@CucumberOptions*** are like property files or settings for your test. Basically @CucumberOptions enables us to do all the things that we could have done if we have used cucumber command line. This is very helpful and of utmost importance, if we are using IDE such eclipse only to execute our project. You must have noticed that we set a few options in the ‘***TestRunner’*** class in the previous chapter.

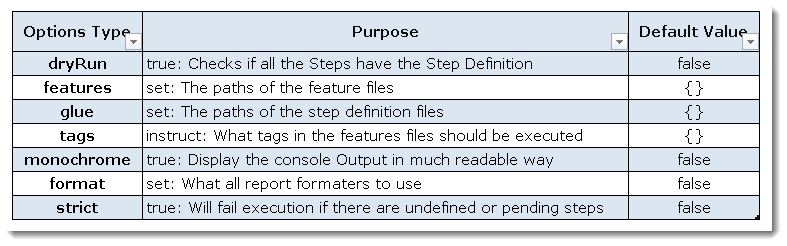
***TestRunner Class***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package cucumberTest;    import org.junit.runner.RunWith;  import cucumber.api.CucumberOptions;  import cucumber.api.junit.Cucumber;    @RunWith(Cucumber.class)  @CucumberOptions(  features = "Feature"  ,glue={"stepDefinition"}  )    public class TestRunner {    } |

So in the above example, we have just set two different Cucumber Options. One is for Feature File and the other is for Step Definition file. We will talk about it in detail now but with this, we can say that @CucumberOptions are used to set some specific properties for the Cucumber test.

Following Main Options are available in Cucumber:



### ***Dry Run***

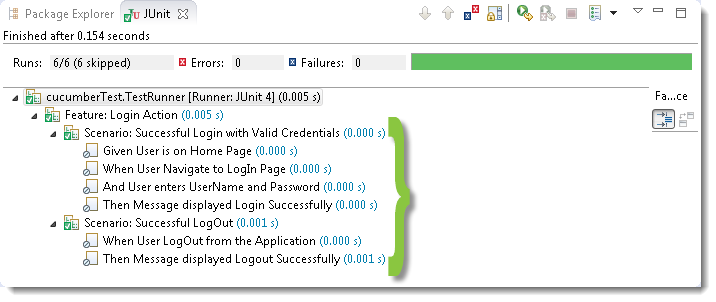
***dryRun*** option can either set as ***true*** or ***false***. If it is set as true, it means that Cucumber will only check that every Step mentioned in the Feature File has corresponding code written in Step Definition file or not. So in case any of the functions are missed in the Step Definition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

***TestRunner Class***

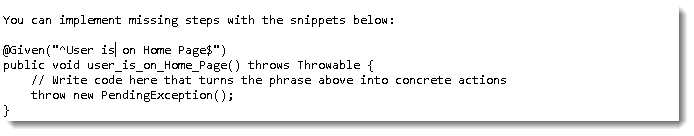


|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | package cucumberTest;    import org.junit.runner.RunWith;  import cucumber.api.CucumberOptions;  import cucumber.api.junit.Cucumber;    @RunWith(Cucumber.class)  @CucumberOptions(  features = "Feature"  ,glue={"stepDefinition"}  ,dryRun = true  )    public class TestRunner {    } |

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.



Take a look at the time duration at the end of the every Steps, it is (***0.000s***). It means none of the Step is executed but still, Cucumber has made sure that every Step has the corresponding method available in the Step Definition file. Give it a try, remove the ‘***@Given(“^User is on Home Page$”)***‘ statement from the ***Test\_Steps*** class and run the ***TestRunner*** class again. You would get the following message:



### ***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:

***TestRunner Class***

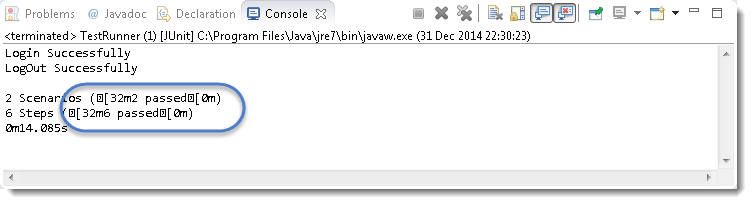


|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | package cucumberTest;    import org.junit.runner.RunWith;  import cucumber.api.CucumberOptions;  import cucumber.api.junit.Cucumber;    @RunWith(Cucumber.class)  @CucumberOptions(  features = "Feature"  ,glue={"stepDefinition"}  ,monochrome = false  )    public class TestRunner {    } |

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:



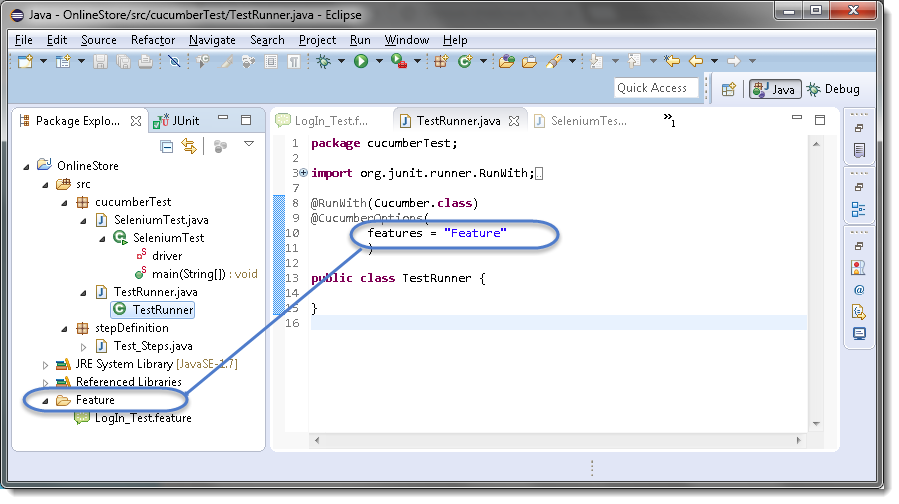
### ***Features***

***Features Options***helps Cucumber to locate the Feature file in the project folder structure. You must have notices that we have been specifying the Feature Option in the ***TestRunner*** class since the first chapter. 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 specified like:

***features = “Feature“***

Or if the Feature file is in the deep folder structure

***features = “src/test/features“***



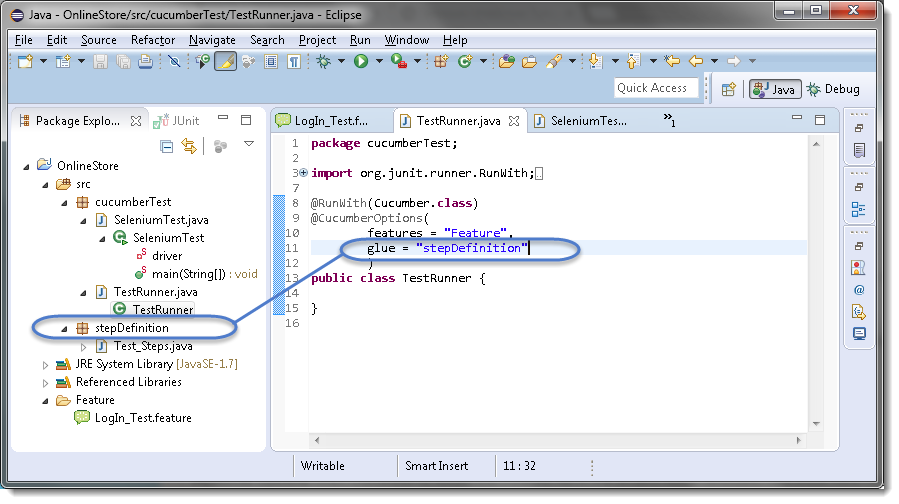
### ***Glue***

It is almost the same think as Features Option but the only difference is that it helps Cucumber to locate 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“}***

Most commercial automated software tools on the market support some sort of ***Data Driven Testing***, which allows to automatically run a test case multiple times with different input and validation values. As Selenium WebDriver is more an automated testing framework than a ready-to-use tool. It takes extra efforts to support data driven testing in automated tests.

This is very often required in any automated test to pass data or to use the same test again with different data set. And the good part is that the ***Cucumber*** inherently supports ***Data Driven Testing using Scenario Outline***. There are different ways to use the data insertion within the Cucumber and outside the Cucumber with external files.

***Data-Driven Testing in Cucumber***

* Parameterization without Example Keyword

***Data-Driven Testing in Cucumber using Scenario Outline***

* Parameterization with Example Keyword
* Parameterization using Tables

***Data-Driven Testing in Cucumber using External Files***

* Parameterization using Excel Files
* Parameterization using Json
* Parameterization using XML

***Scenario Outline*** – This is used to run the same scenario for 2 or more different sets of test data. ***E.g***. In our scenario, if you want to register another user you can data drive the same scenario twice.

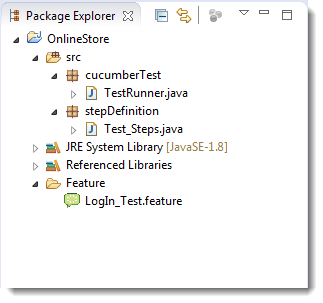
***Examples*** – All scenario outlines have to be followed with the Examples section. This contains the data that has to be passed on to the scenario.

## Data-Driven Testing in Cucumber

In the series of previous chapters, we are following the LogIn scenario. To demonstrate how parametrizing works, I am taking the same scenario again. It is important for you to be on the same page in term of project code, else you may get confused. Let’s take a look at the current state of the project. In case you find it confusing, I would request you to go through the previous tutorial of [***Feature File***](https://toolsqa.com/cucumber/cucumber-jvm-feature-file/)***,***[***Test Runner***](https://toolsqa.com/cucumber/junit-test-runner-class/)***&***[***Step Definition***](https://toolsqa.com/specflow/step-definition-file/).

The project folder structure and code should be in the below state.

***Package Explorer***



***LogIn\_Test.fetaure***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Feature: Login Action    Scenario: Successful Login with Valid Credentials  Given User is on Home Page  When User Navigate to LogIn Page  And User enters UserName and Password  Then Message displayed Login Successfully    Scenario: Successful LogOut  When User LogOut from the Application  Then Message displayed LogOut Successfully |

***Test\_Steps.java***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | package stepDefinition;    import java.util.concurrent.TimeUnit;    import org.openqa.selenium.By;  import org.openqa.selenium.WebDriver;  import org.openqa.selenium.firefox.FirefoxDriver;    import cucumber.api.java.en.Given;  import cucumber.api.java.en.Then;  import cucumber.api.java.en.When;    public class Test\_Steps {  public static WebDriver driver;  @Given("^User is on Home Page$")  public void user\_is\_on\_Home\_Page() throws Throwable {  driver = new FirefoxDriver();      driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);      driver.get("https://www.store.demoqa.com");  }    @When("^User Navigate to LogIn Page$")  public void user\_Navigate\_to\_LogIn\_Page() throws Throwable {  driver.findElement(By.xpath(".//\*[@id='account']/a")).click();  }    @When("^User enters UserName and Password$")  public void user\_enters\_UserName\_and\_Password() throws Throwable {  driver.findElement(By.id("log")).sendKeys("testuser\_1");      driver.findElement(By.id("pwd")).sendKeys("Test@123");      driver.findElement(By.id("login")).click();  }    @Then("^Message displayed Login Successfully$")  public void message\_displayed\_Login\_Successfully() throws Throwable {  System.out.println("Login Successfully");  }    @When("^User LogOut from the Application$")  public void user\_LogOut\_from\_the\_Application() throws Throwable {  driver.findElement (By.xpath(".//\*[@id='account\_logout']/a")).click();  }    @Then("^Message displayed LogOut Successfully$")  public void message\_displayed\_LogOut\_Successfully() throws Throwable {  System.out.println("LogOut Successfully");  }  } |

***TestRunner.java***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package cucumberTest;    import org.junit.runner.RunWith;  import cucumber.api.CucumberOptions;  import cucumber.api.junit.Cucumber;    @RunWith(Cucumber.class)  @CucumberOptions(  features = "Feature"  ,glue={"stepDefinition"}  )    public class TestRunner {    } |

### Parameterizing without Example Keyword

Now the task is to ***Parameterizing the UserName and Password***. Which is quite logical, why would anybody want to hardcode the UserName & Password of the application. As there is a high probability of changing both.

1) Go to the ***Feature File*** and change the statement where passing Username & Password as per below:

***And User enters “testuser\_1” and “Test@123“***

In the above statement, we have passed Username & Password from the Feature File which will feed in to Step Definition of the above statement automatically. Cucumber will do the trick for us. After the above changes, the code will look like this:

***LogIn\_Test.feature***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Feature: Login Action    Scenario: Successful Login with Valid Credentials  Given User is on Home Page  When User Navigate to LogIn Page  And User enters "testuser\_1" and "Test@123"  Then Message displayed Login Successfully    Scenario: Successful LogOut  When User LogOut from the Application  Then Message displayed LogOut Successfully |

2) Changes in the Step Definition file is also required to make it understand the Parameterization of the feature file. So, it is required to update the Test Step in the Step Definition file which is linked with the above-changed Feature file statement. Use the below code:

***@When(“^User enters \”(.\*)\” and \”(.\*)\”$”)***

The same can be achieved by using the below code as well:

***@When(“^User enters \”([^\”]\*)\” and \”([^\”]\*)\”$”)***

With the help of the above statements, Cucumber will understand that the associated Test\_Step is expecting some parameters.

3) Same parameters should also go into the associated Test\_Step. As the Test step is nothing but a simple Java method, syntax to accept the parameter in the Java method is like this:

***public void user\_enters\_UserName\_and\_Password(String username, String password) throws Throwable {***

***}***

4) Now the last step is to feed the parameters in the actual core statements of Selenium WebDriver. Use the below code:

***driver.findElement(By.id(“log“)).sendKeys(username);***  
***driver.findElement(By.id(“pwd“)).sendKeys(password);***  
***driver.findElement(By.id(“login“)).click();***

After making the above changes, the method will look like this:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @When("^User enters \"(.\*)\" and \"(.\*)\"$")  public void user\_enters\_UserName\_and\_Password(String username, String password) throws Throwable {  driver.findElement(By.id("log")).sendKeys(username);      driver.findElement(By.id("pwd")).sendKeys(password);      driver.findElement(By.id("login")).click();  } |

5) Run the test by Right Click on ***TestRunner class*** and Click ***Run As  > JUnit Test*** Application. You would notice that the Cucumber will open the Website in the browser and enter username & password which is passed from the Feature File.

The next chapter is about doing ***Parameterization using Example Keyword in Cucumber***.

In the last chapter of [***Parameterization in Cucumber***](https://toolsqa.com/cucumber/data-driven-testing-in-cucumber/), we learned how to *parameterize* data. But with that trick, only limited functionality can be achieved of Data-Driven. As the test can be run multiple times. But by now that you know the anatomy of a *Data-Driven test*, here’s a trick that simplifies the process of ***Data-Driven testing using Cucumber***. *Cucumber* inherently supports *Data-Driven testing* by the use of the ***Scenario Outline*** and ***Examples*** section. It is with these keywords that *Cucumber* allows for easy *Data-Driven testing* to be completed where no changes need to be made to the Java file. In this tutorial we learn, How to ***Implement a Scenario Outline in Data-Driven testing using Examples Keyword?***

*Example* keyword can only be used with the *Scenario Outline* Keyword.

* ***Scenario Outline****– This is used to run the same scenario for 2 or more different sets of test data. E.g. In our scenario, if you want to register another user you can data drive the same scenario twice.*
* ***Examples****– All scenario outlines have to be followed with the Examples section. This contains the data that has to be passed on to the scenario.*

**Data-Driven Testing Using Examples Keyword**

If you understood the concept of [***Parameterization in Cucumber***](https://toolsqa.com/cucumber/data-driven-testing-in-cucumber/), you would find this one very easy. In this tutorial as well I am taking the same *LogIn test* scenario.

1) Enter the***Example Data*** just below the *LogIn* Scenario of the *Feature* File.

***Examples:***  
***| username  | password  |***  
***| testuser\_1 | Test@153 |***  
***| testuser\_2 | Test@153 |***

***Note****: The table must have a header row corresponding to the variables in the Scenario Outline steps.*

The Examples section is a table where each argument variable represents a column in the table, separated by “***|***”. Each line below the header represents an individual run of the test case with the respective data. As a result, if there are 3 lines below the header in the Examples table, the script will run 3 times with its respective data.

2) Need to update the Statement in the *feature* file, which tells *Cucumber* to enter *username & Password*.

***And User enters <username> and <password>***

*Cucumber* understands the above statement syntax and looks for the ***Examples***Keyword in the test to read the *Test Data*.

***The complete code will look like this***:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Feature: Login Action    Scenario Outline: Successful Login with Valid Credentials  Given User is on Home Page  When User Navigate to LogIn Page  And User enters "<username>" and "<password>"  Then Message displayed Login Successfully  Examples:      | username   | password |      | testuser\_1 | Test@153 |      | testuser\_2 | Test@153 | |

3) There are no changes in ***TestRunner***class.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package cucumberTest;    import org.junit.runner.RunWith;  import cucumber.api.CucumberOptions;  import cucumber.api.junit.Cucumber;    @RunWith(Cucumber.class)  @CucumberOptions(  features = "Feature"  ,glue={"stepDefinition"}  )    public class TestRunner {    } |

4) There are no changes in ***Test\_Steps*** file from the previous chapter.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | package stepDefinition;    import java.util.concurrent.TimeUnit;    import org.openqa.selenium.By;  import org.openqa.selenium.WebDriver;  import org.openqa.selenium.firefox.FirefoxDriver;    import cucumber.api.java.en.Given;  import cucumber.api.java.en.Then;  import cucumber.api.java.en.When;    public class Test\_Steps {  public static WebDriver driver;  @Given("^User is on Home Page$")  public void user\_is\_on\_Home\_Page() throws Throwable {  driver = new FirefoxDriver();      driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);      driver.get("https://www.store.demoqa.com");  }    @When("^User Navigate to LogIn Page$")  public void user\_Navigate\_to\_LogIn\_Page() throws Throwable {  driver.findElement(By.xpath(".//\*[@id='account']/a")).click();  }    @When("^User enters \"(.\*)\" and \"(.\*)\"$")  public void user\_enters\_UserName\_and\_Password(String username, String password) throws Throwable {  driver.findElement(By.id("log")).sendKeys(username);      driver.findElement(By.id("pwd")).sendKeys(password);      //driver.findElement(By.id("login")).click();  }    @Then("^Message displayed Login Successfully$")  public void message\_displayed\_Login\_Successfully() throws Throwable {  System.out.println("Login Successfully");  }    @When("^User LogOut from the Application$")  public void user\_LogOut\_from\_the\_Application() throws Throwable {  driver.findElement (By.xpath(".//\*[@id='account\_logout']/a")).click();  }    @Then("^Message displayed LogOut Successfully$")  public void message\_displayed\_LogOut\_Successfully() throws Throwable {  System.out.println("LogOut Successfully");  }    } |

5) Run the test by *Right Click* on ***TestRunner class*** and Click ***Run As  > JUnit Test*** Application.

This takes the *parameterization* one step further: now our scenario has “***variables***” and they get filled in by the values in each row. To be clear: by defining this, the scenario will run two times, passing in one row at a time. This makes it very easy to define a lot of examples, edge cases, and special outcomes.  Instead of hardcoding the test data, variables are defined in the Examples section and used in the *Scenario Outline* section.

***Data Tables in Cucumber*** are quite interesting and can be used in many ways. DataTables are also used to handle large amounts of data. They are quite powerful but not the most intuitive as you either need to deal with a ***list of maps*** or a ***map of lists***. Most of the people get confused with Data tables & Scenario outline, but these two works completely differently.

## Difference between Scenario Outline & Data Table

***Scenario Outline:***

* This uses Example keyword to define the test data for the Scenario
* This works for the whole test
* Cucumber automatically run the complete test the number of times equal to the number of data in the Test Set

***Test Data:***

* No keyword is used to define the test data
* This works only for the single step, below which it is defined
* A separate code needs to understand the test data and then it can be run single or multiple times but again just for the single step, not for the complete test

As I said above, the Data Tables can be used in many ways because it has provided many different methods to use. Let’s just go through a few most popular methods. I will choose a simple scenario to illustrate the working of the Data Table but we will make effective use of this when we will do **Cucumber Framework** in the next series of this ***Cucumber Tutorial***.

## Data Tables in Cucumber

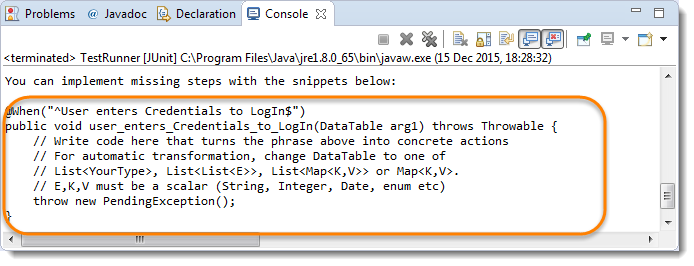
In this example, we will pass the test data using the data table and handle it using***Raw()*** method.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Scenario: Successful Login with Valid Credentials  Given User is on Home Page  When User Navigate to LogIn Page  And User enters Credentials to LogIn      | testuser\_1 | Test@153 |  Then Message displayed Login Successfully |

The complete scenario is same as what we have done earlier. But the only difference is in this, we are not passing parameters in the step line and even we are not using Examples test data. We declared the data under the step only. So we are using Tables as arguments to Steps.

If you run the above scenario without implementing the step, you would get the following error in the Eclipse console window.



Copy the above hint in the Step Definition file and complete the implementation.

***The implementation of the above step will be like this:***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | The implementation of the above step will belike this:  @When("^User enters Credentials to LogIn$")  public void user\_enters\_testuser\_\_and\_Test(DataTable usercredentials) throws Throwable {    //Write the code to handle Data Table  List<List<String>> data = usercredentials.raw();    //This is to get the first data of the set (First Row + First Column)  driver.findElement(By.id("log")).sendKeys(data.get(0).get(0));    //This is to get the first data of the set (First Row + Second Column)      driver.findElement(By.id("pwd")).sendKeys(data.get(0).get(1));        driver.findElement(By.id("login")).click();  } |

### ***The complete test Implementation***

***Test Runner Class***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package cucumberTest;    import org.junit.runner.RunWith;  import cucumber.api.CucumberOptions;  import cucumber.api.junit.Cucumber;    @RunWith(Cucumber.class)  @CucumberOptions(  features = "Feature"  ,glue={"stepDefinition"}  )    public class TestRunner {    } |

***Feature File***



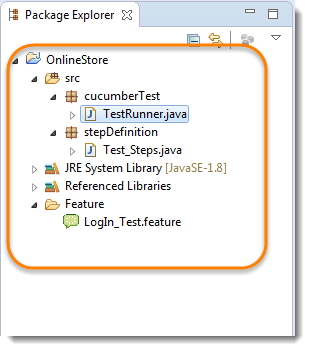
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Feature: Login Action    Scenario: Successful Login with Valid Credentials  Given User is on Home Page  When User Navigate to LogIn Page  And User enters Credentials to LogIn      | testuser\_1 | Test@153 |  Then Message displayed Login Successfully    Scenario: Successful LogOut  When User LogOut from the Application  Then Message displayed LogOut Successfully |

***Step Definition***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | package stepDefinition;    import java.util.List;  import java.util.concurrent.TimeUnit;  import org.openqa.selenium.By;  import org.openqa.selenium.WebDriver;  import org.openqa.selenium.firefox.FirefoxDriver;  import cucumber.api.DataTable;  import cucumber.api.java.en.Given;  import cucumber.api.java.en.Then;  import cucumber.api.java.en.When;    public class Test\_Steps {  public static WebDriver driver;  @Given("^User is on Home Page$")  public void user\_is\_on\_Home\_Page() throws Throwable {  driver = new FirefoxDriver();      driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);      driver.get("https://www.store.demoqa.com");  }    @When("^User Navigate to LogIn Page$")  public void user\_Navigate\_to\_LogIn\_Page() throws Throwable {  driver.findElement(By.xpath(".//\*[@id='account']/a")).click();  }    @When("^User enters Credentials to LogIn$")  public void user\_enters\_testuser\_\_and\_Test(DataTable usercredentials) throws Throwable {  List<List<String>> data = usercredentials.raw();  driver.findElement(By.id("log")).sendKeys(data.get(0).get(0));      driver.findElement(By.id("pwd")).sendKeys(data.get(0).get(1));      driver.findElement(By.id("login")).click();  }    @Then("^Message displayed Login Successfully$")  public void message\_displayed\_Login\_Successfully() throws Throwable {  System.out.println("Login Successfully");  }    @When("^User LogOut from the Application$")  public void user\_LogOut\_from\_the\_Application() throws Throwable {  driver.findElement (By.xpath(".//\*[@id='account\_logout']/a")).click();  }    @Then("^Message displayed LogOut Successfully$")  public void message\_displayed\_LogOut\_Successfully() throws Throwable {  System.out.println("LogOut Successfully");  }  } |

***Project Explorer***



Run the test by Right Click on ***TestRunner class*** and Click ***Run As  > JUnit Test*** Application. you will notice that Cucumber will automatically figure out, what to provide in the Username and Password field.

### ***Maps in Data Tables with Header***

In the previous chapter of [***Data Tables in Cucumber***](https://toolsqa.com/cucumber/data-tables-in-cucumber/)***,*** we pass Username & Password without Header, due to which the test was not much readable. What if there will be many columns. The basic funda of BDD test is to make the Test in Business readable format, so that business users can understand it easily. Setting Header in Test data is not a difficult task in Cucumber. take a look at a below Scenario.

***Feature File Scenario***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Scenario: Successful Login with Valid Credentials  Given User is on Home Page  When User Navigate to LogIn Page  And User enters Credentials to LogIn  | Username   | Password |      | testuser\_1 | Test@153 |  Then Message displayed Login Successfully |

***The implementation of the above step will be like this:***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @When("^User enters Credentials to LogIn$")  public void user\_enters\_testuser\_and\_Test(DataTable usercredentials) throws Throwable {    //Write the code to handle Data Table  List<Map<String,String>> data = usercredentials.asMaps(String.class,String.class);  driver.findElement(By.id("log")).sendKeys(data.get(0).get("Username"));      driver.findElement(By.id("pwd")).sendKeys(data.get(0).get("Password"));      driver.findElement(By.id("login")).click();             } |

### ***Maps in Data Tables with Multiple Test Data***

In this test we will pass Username and Password two times to the test step. So our test should enter Username & Password once, click on LogIn button and repeat the same steps again.

***Feature File Scenario***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Scenario: Successful Login with Valid Credentials  Given User is on Home Page  When User Navigate to LogIn Page  And User enters Credentials to LogIn  | Username   | Password |      | testuser\_1 | Test@153 |      | testuser\_2 | Test@154 |  Then Message displayed Login Successfully |

***The implementation of the above step will be like this:***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @When("^User enters Credentials to LogIn$")  public void user\_enters\_testuser\_and\_Test(DataTable usercredentials) throws Throwable {    //Write the code to handle Data Table  for (Map<String, String> data : usercredentials.asMaps(String.class, String.class)) {  driver.findElement(By.id("log")).sendKeys(data.get("Username"));      driver.findElement(By.id("pwd")).sendKeys(data.get("Password"));      driver.findElement(By.id("login")).click();  }    } |

### ***Map Data Tables to Class Objects***

Luckily there are easier ways to access your data than DataTable. For instance you can create a Class-Object and have Cucumber map the data in a table to a list of these.

***Feature File Scenario***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Scenario: Successful Login with Valid Credentials  Given User is on Home Page  When User Navigate to LogIn Page  And User enters Credentials to LogIn  | Username   | Password |      | testuser\_1 | Test@153 |      | testuser\_2 | Test@154 |  Then Message displayed Login Successfully |

***The implementation of the above step will be like this:***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @When("^User enters Credentials to LogIn$")  public void user\_enters\_testuser\_and\_Test(List<Credentials>  usercredentials) throws Throwable {    //Write the code to handle Data Table  for (Credentials credentials : usercredentials) {  driver.findElement(By.id("log")).sendKeys(credentials.getUsername());      driver.findElement(By.id("pwd")).sendKeys(credentials.getPassword());      driver.findElement(By.id("login")).click();  }  } |

***Class Credentials***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | package stepDefinition;    public class Credentials {  private String username;  private String password;    public String getUsername() {          return username;      }  public String getPassword() {          return password;      }  } |

## What are Cucumber Tags?

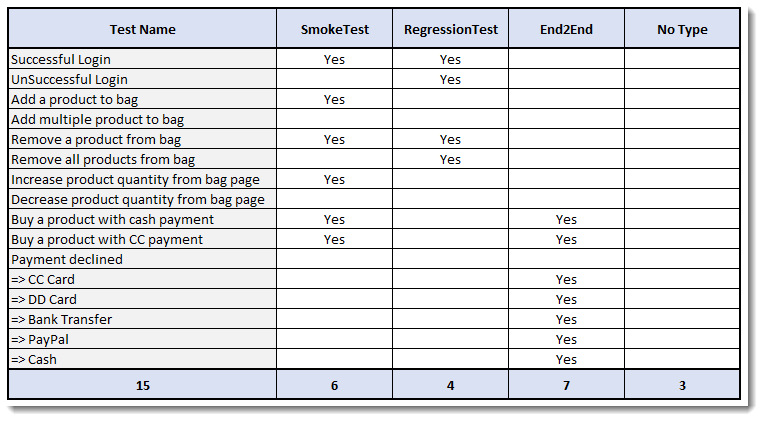
Let’s say you have got many different feature files that cover all the different functionality of the application. Now there can be a certain situation in the project where you like to execute just a ***SmokeTests*** or ***End2EndTests*** or may be ***RegressionTests***. One approach is that you start creating new feature files with the name of the type like ***SmokeTests.features*** or***End2EndTests.feature*** and copy-paste your existing tests in the same. But this would make the project filthy and would require more maintenance in future. So how to manage execution in such cases?

For this, Cucumber has already provided a way to organize your scenario execution by using ***tags*** in feature file. We can define each scenario with a useful tag. Later, in the runner file, we can decide which specific tag (and so as the scenario(s)) we want Cucumber to execute. Tag starts with “***@***”. After “@” you can have any relevant text to define your tag like ***@SmokeTests***just above the scenarios you like to mark. Then to target these tagged scenarios just specify the tags names in the***CucumberOptions***as***tags = {“@SmokeTests”}.***

Tagging not just specifically works with Scenarios, it also works with ***Features***. Means you can also tag your features files. ***Any tag that exists on a Feature will be inherited by Scenario, Scenario Outline or Examples.***

## How to run Cucumber Tests in Groups using Cucumber Tags?

Let’s understand this with an example. Below is an excel sheet containing a list of scenarios of a single feature.



***Things to Note:***

* Few scenarios are part of the Smoke Test, Regression Test, and End2End Test.
* Few scenarios are part of two or more Test Types. For example, the first test is considered as Smoke as well as Regression.
* Few scenarios are not at all tagged
* The last scenario of Payment Declined, it is a single scenario but has five different test data. So this will be considered as five different scenarios.

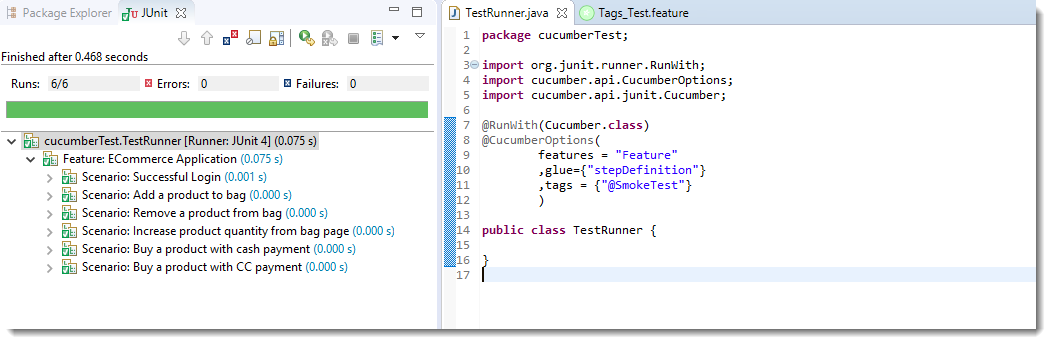
***Feature file will look like this***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | @FunctionalTest  Feature: ECommerce Application    @SmokeTest @RegressionTest  Scenario: Successful Login  Given This is a blank test    @RegressionTest  Scenario: UnSuccessful Login  Given This is a blank test    @SmokeTest  Scenario: Add a product to bag  Given This is a blank test    Scenario: Add multiple product to bag  Given This is a blank test    @SmokeTest @RegressionTest  Scenario: Remove a product from bag  Given This is a blank test    @RegressionTest  Scenario: Remove all products from bag  Given This is a blank test    @SmokeTest  Scenario: Increase product quantity from bag page  Given This is a blank test    Scenario: Decrease product quantity from bag page  Given This is a blank test    @SmokeTest @End2End  Scenario: Buy a product with cash payment  Given This is a blank test    @SmokeTest @End2End  Scenario: Buy a product with CC payment  Given This is a blank test    @End2End  Scenario Outline: Payment declined  Given This is a blank test  Examples:  |PaymentMethod|  |CC Card|  |DD Card|  |Bank Transfer|  |PayPal|  |Cash| |

### ***Running single Cucumber Feature file or single Cucumber Tag***

***Execute all tests tagged as @SmokeTests***

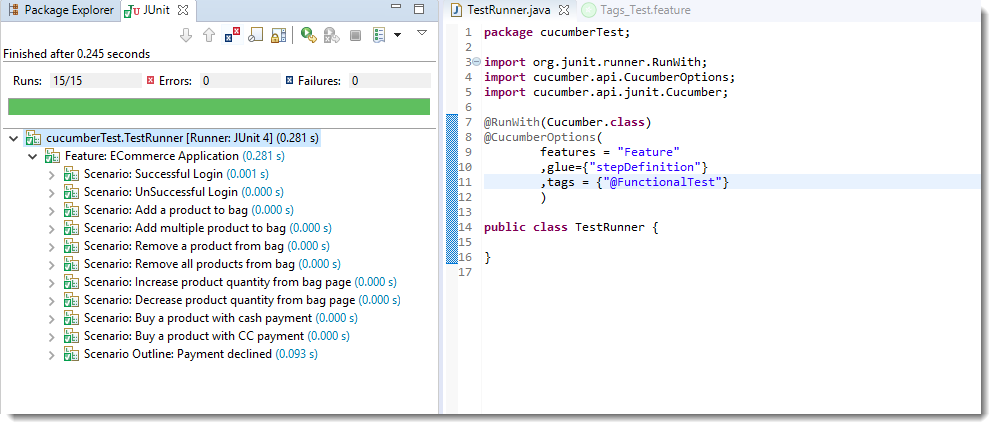
  
***Note***: In the excel sheet and in the feature file paste above if you count the scenarios which are tagged as @SmokeTests, you will find the count is 6 and the same count is also displayed under Junit tab.

***Execute all tests tagged as @End2End***

### Cucumber Group Tags 9

***Note***: A special thing to note here is that, the last scenario ***Payment declined*** has five different data examples. So every example is considered as a separate test. Due to which the total test number is 7.

***Execute all tests of a Feature tagged as @FunctionalTest : Feature Tagging***

Not only tags work with Scenario, tags work with Feature Files as well. Feature files pasted above is also tagged as ***@FunctionTests***. Let’s just see how to executes all the tests in this feature.  


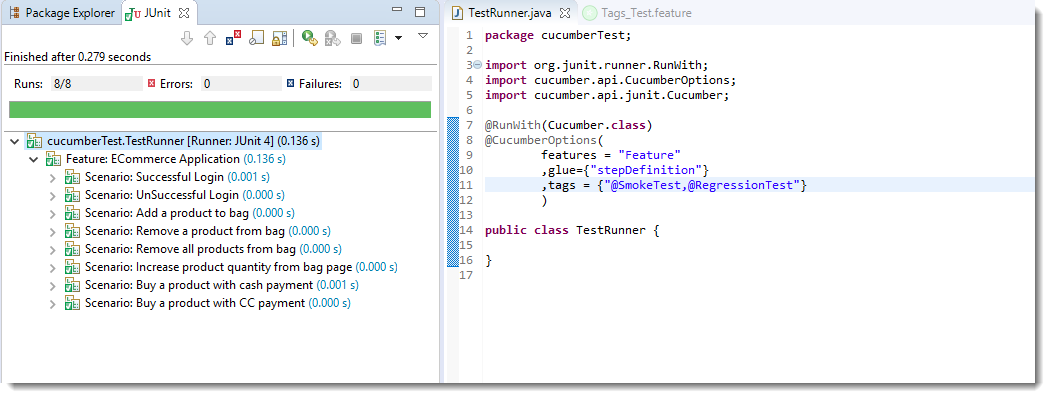
***Note***: All the test exists in the feature file are executed.

## Logically ANDing and ORing Tags

Requirements are complicated, it will not always simple like executing a single tag. It can be complicated like executing scenarios that are tagged either as @SmokeTest or @RegressionTest. It can also be like executing scenarios that are tagged both as @SmokeTest and @RegressionTest. Cucumber tagging gives us the capability to choose what we want with the help of ANDing and ORing.

***Execute all tests tagged as @SmokeTest OR @RegressionTest***

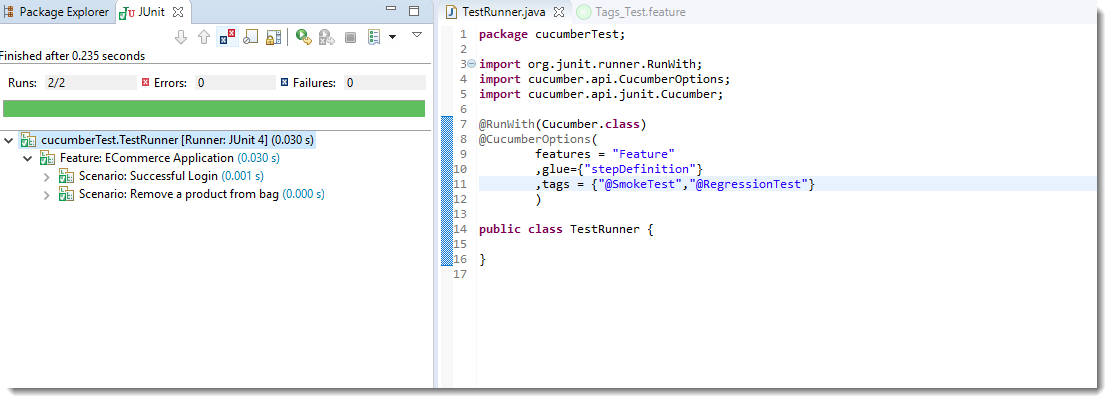
Tags that are ***comma-separated*** are ORed.



***Note***: OR means scenarios that are tagged either as @SmokeTest OR @RegressionTest.

***Execute all tests tagged as @SmokeTest AND @RegressionTest***

Tags which are passed in separate ***quotes*** are ANDed

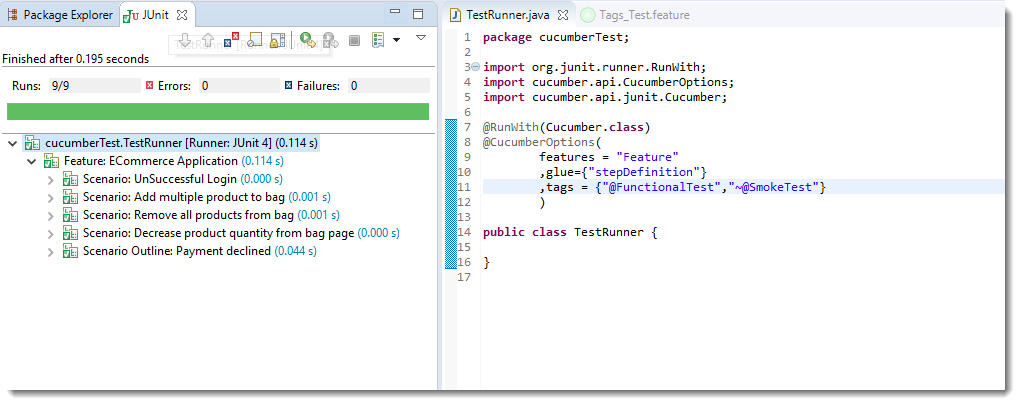


***Note***: There are only two scenarios in our feature file which have both tags together.

## How to Ignore Cucumber Tests

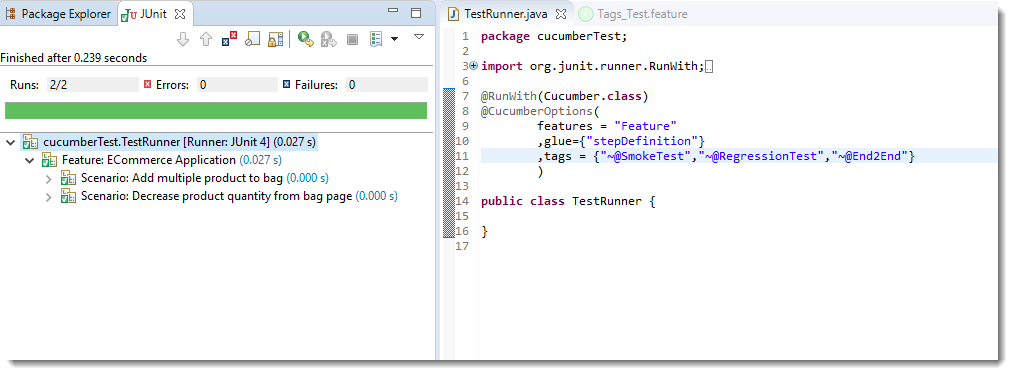
This is again a good feature of Cucumber Tags that you can even skip tests in the group execution. Special Character ***~***is used to skip the tags. This also works both for Scenarios and Features. And this can also works in conjunction with AND or OR.

***Execute all tests of the feature tagged as @FunctionalTests but skip scenarios tagged as @SmokeTest***

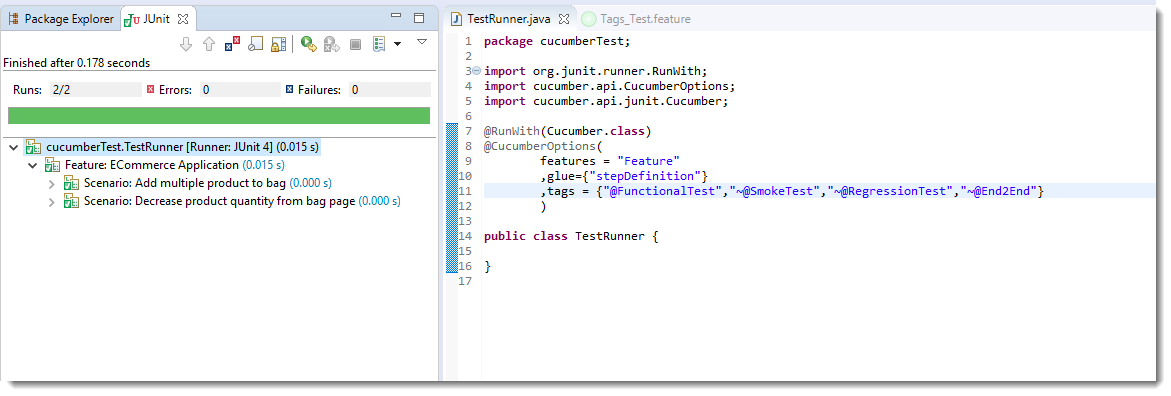


Note: This is AND condition, which means all the scenarios tagged as @FunctionalTest but not @SmokeTest. So total tests are 15 and smoke tests are 6, so it ran just 9 tests.

***Execute all tests which are not at all tagged in all the Features***



***Execute all tests which are not at all tagged in Single Feature***



It is fun to play with tags, especially when you have many features files with multiple scenarios.

## What are Hooks in Cucumber?

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***. ***Cucumber Hooks*** allows us to better manage the code workflow and helps us to reduce the code redundancy. We can say that it is an unseen step, which allows us to perform our scenarios or tests.

## Why Cucumber Hooks?

In the world of testing, you must have encountered the situations where you need to perform the prerequisite steps before testing any test scenario. This prerequisite can be anything from:

* Starting a webdriver
* Setting up DB connections
* Setting up test data
* Setting up browser cookies
* Navigating to certain page
* or anything before the test

In the same way, there are always after steps as well of the tests like:

* Killing the webdriver
* Closing DB connections
* Clearing the test data
* Clearing browser cookies
* Logging out from the application
* Printing reports or logs
* Taking screenshots on error
* or anything after the test

To handle these kinds of situations, cucumber hooks are the best choice to use. Unlike [***TestNG Annotaions***](https://toolsqa.com/selenium-webdriver/testng-annotations-groups-depends/), cucumber supports only two hooks (Before & After) which works at the start and the end of the test scenario. As the name suggests, @before hook gets executed well before any other test scenario, and @after hook gets executed after executing the scenario.

## How to implement Hooks in Cucumber Test

Let’s do some easy and small examples of Cucumber Hooks just to understand the concept. I will bring the intelligent usage of Hooks in my later tutorial series of **Designing Framework with Cucumber**.

### ***Test Hooks with Single Scenario***

***Feature File***



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Feature: Test Hooks    Scenario: This scenario is to test hooks functionality  Given this is the first step  When this is the second step  Then this is the third step |

***Step Definitions***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | package stepDefinition;    import cucumber.api.java.en.Given;  import cucumber.api.java.en.Then;  import cucumber.api.java.en.When;    public class Hooks\_Steps {    @Given("^this is the first step$")  public void This\_Is\_The\_First\_Step(){  System.out.println("This is the first step");  }    @When("^this is the second step$")  public void This\_Is\_The\_Second\_Step(){  System.out.println("This is the second step");  }    @Then("^this is the third step$")  public void This\_Is\_The\_Third\_Step(){  System.out.println("This is the third step");  }    } |

***Note***: There is no logic used in the step definitions. Just printing the step summary log.

***Hooks***



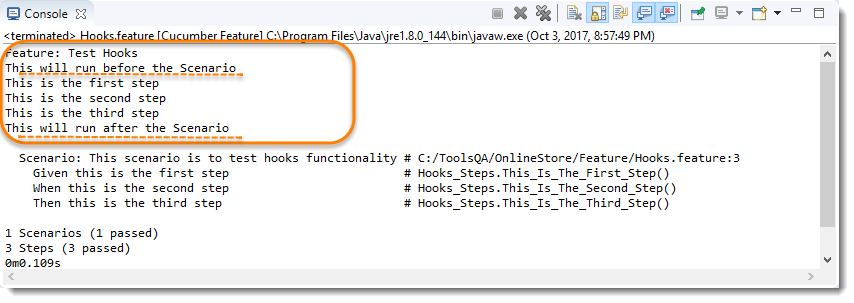
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | package utilities;  import cucumber.api.java.After;  import cucumber.api.java.Before;    public class Hooks {    @Before      public void beforeScenario(){          System.out.println("This will run before the Scenario");      }    @After      public void afterScenario(){          System.out.println("This will run after the Scenario");      }  } |

***Things to note***

* An important thing to note about the after hook is that even in case of test fail, after hook will execute for sure.
* Method name can be anything, need not to be beforeScenario() or afterScenario(). can also be named as setUp() and tearDown().
* Make sure that the package import statement should be ***import cucumber.api.java.After; & import cucumber.api.java.Before;***

Often people mistaken and import Junit Annotations, so be careful with this.

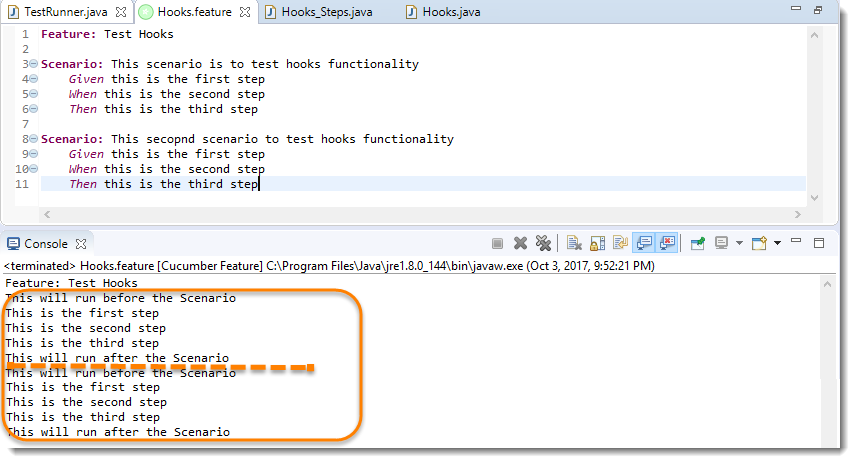
***Output***



No need for explanation, it is self-explanatory

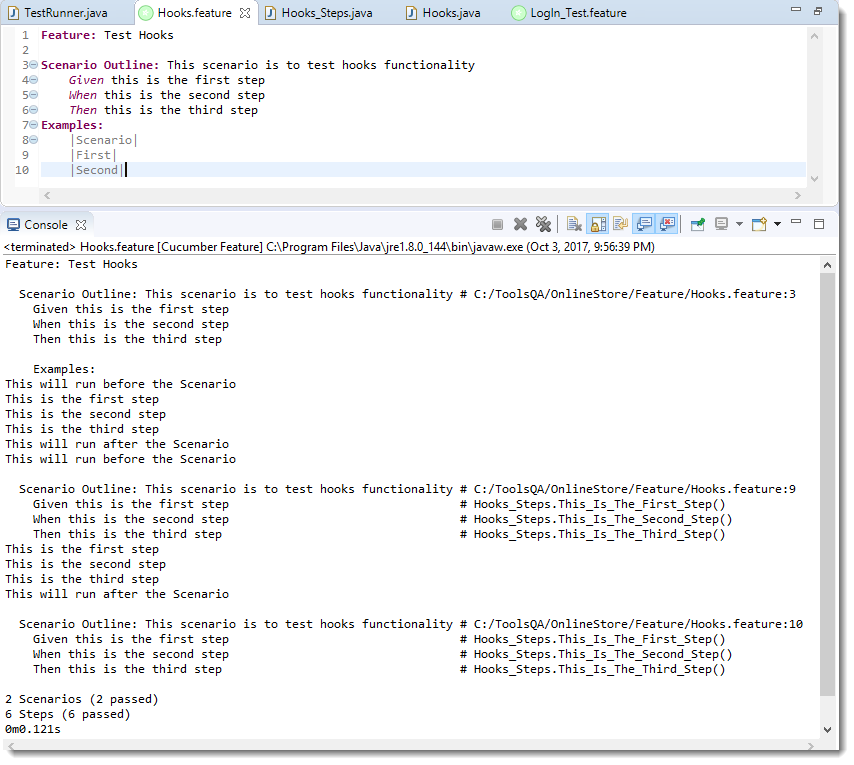
### ***Test Hooks with Multiple Scenarios***

I just wanted to show you the reaction of Hooks with the multiple scenarios. Let’s just add one more Test Scenario in the feature file and run the feature again.



***Note***: Scenario Hooks execute before and after every scenario. In the above example, executed two times for two scenarios.

### ***Test Hooks with Example Scenarios***

Lets take a look when we have Scenario Outline with Examples.  


***Note***: Again, in cucumber, every example is considered as a separate scenario. So the output is the same as the second example above.

## Tagged Hooks in Cucumber

Lets again start with doing a simple exercise to get the concept straight. Just keep three different scenarios in the feature file with the same Given, When & Then steps.

1)-First step is to annotate required scenarios using ***@ + AnyName*** at the top of the Scenario. For this example, I just annotate each scenario with the sequence order of it, like ***@First, @Second & @Third***.

***Feature File***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | Feature: Test Tagged Hooks    @First  Scenario: This is First Scenario  Given this is the first step  When this is the second step  Then this is the third step    @Second  Scenario: This is Second Scenario  Given this is the first step  When this is the second step  Then this is the third step    @Third  Scenario: This is Third Scenario  Given this is the first step  When this is the second step  Then this is the third step |

2) Create a Step definition file and just print the execution order of the steps in the console.

***Step Definitions***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | package stepDefinition;    import cucumber.api.java.en.Given;  import cucumber.api.java.en.Then;  import cucumber.api.java.en.When;    public class Hooks\_Steps {    @Given("^this is the first step$")  public void This\_Is\_The\_First\_Step(){  System.out.println("This is the first step");  }    @When("^this is the second step$")  public void This\_Is\_The\_Second\_Step(){  System.out.println("This is the second step");  }    @Then("^this is the third step$")  public void This\_Is\_The\_Third\_Step(){  System.out.println("This is the third step");  }    } |

3) Define tagged hooks in Hooks class file. Hooks can be used like **@Before(“@TagName”)**. Create before and after hooks for every scenario. I have also added normal before and after hooks, in case you are not aware, please go to the previous chapter of [***Hooks in Cucumber***](https://toolsqa.com/cucumber/cucumber-hooks/).

***Hooks***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | package utilities;    import cucumber.api.java.After;  import cucumber.api.java.Before;    public class Hooks {    @Before      public void beforeScenario(){          System.out.println("This will run before the every Scenario");      }    @After      public void afterScenario(){          System.out.println("This will run after the every Scenario");      }    @Before("@First")      public void beforeFirst(){          System.out.println("This will run only before the First Scenario");      }    @Before("@Second")      public void beforeSecond(){          System.out.println("This will run only before the Second Scenario");      }    @Before("@Third")      public void beforeThird(){          System.out.println("This will run only before the Third Scenario");      }    @After("@First")      public void afterFirst(){          System.out.println("This will run only after the First Scenario");      }    @After("@Second")      public void afterSecond(){          System.out.println("This will run only after the Second Scenario");      }    @After("@Third")      public void afterThird(){          System.out.println("This will run only after the Third Scenario");      }  } |

***Note***: We learned that @Before & @After hooks run before & after every Scenario. But ***@Before(“@First”)*** will run only before the first scenario and likewise other tagged hooks. Again, these tags names can be anything and no need to be first, second and third.

4) Run the feature file and observe the output.

***Output***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | Feature: Test Tagged Hooks  This will run only before the First Scenario  This will run before the every Scenario  This is the first step  This is the second step  This is the third step  This will run after the every Scenario  This will run only after the First Scenario  This will run only before the Second Scenario  This will run before the every Scenario  This is the first step  This is the second step  This is the third step  This will run after the every Scenario  This will run only after the Second Scenario  This will run before the every Scenario  This will run only before the Third Scenario  This is the first step  This is the second step  This is the third step  This will run only after the Third Scenario  This will run after the every Scenario |

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

We can have common tagged hooks for multiple scenarios as well. In the below example, I just combined the ***@Before(“First”)*** and ***@Before(“Third”)*** by ***@Before(“@First, @Third”).***So in this way we do not need to have two different hooks logic.

***Hooks***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | package utilities;    import cucumber.api.java.After;  import cucumber.api.java.Before;    public class Hooks {    @After      public void afterScenario(){          System.out.println("This will run after the every Scenario");      }  @Before      public void beforeScenario(){          System.out.println("This will run before the every Scenario");      }    @Before("@Second")      public void beforeSecond(){          System.out.println("This will run only before the Second Scenario");      }  @Before("@First,@Third")      public void beforeFirstAndThird(){          System.out.println("This will run before both First & Third Scenario");      }    @After("@First")      public void afterFirst(){          System.out.println("This will run only after the First Scenario");      }  @After("@Second")      public void afterSecond(){          System.out.println("This will run only after the Second Scenario");      }  @After("@Third")      public void afterThird(){          System.out.println("This will run only after the Third Scenario");      }    } |

***Output***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | Feature: Test Tagged Hooks  This will run before the every Scenario  This will run before both First & Third Scenario  This is the first step  This is the second step  This is the third step  This will run only after the First Scenario  This will run after the every Scenario    This will run before the every Scenario  This will run only before the Second Scenario  This is the first step  This is the second step  This is the third step  This will run only after the Second Scenario  This will run after the every Scenario    This will run before the every Scenario  This will run before both First & Third Scenario  This is the first step  This is the second step  This is the third step  This will run only after the Third Scenario  This will run after the every Scenario |

## Execution Order of Hooks

Order hooks to run in a particular sequence is easy to do. As we already know the way to specify hooks in cucumber-like putting an annotation just above the scenario. Ordering also works the same way but the only difference is that it required an extra parameter. This extra parameter decides the order of execution of the certain hook.

***For example*** ***@Before***, and if you want to specify the order it will become ***@Before(value = 1)***.

The same goes with any [***Tags***](https://toolsqa.com/cucumber/cucumber-tags/) or [***Hooks***](https://toolsqa.com/cucumber/cucumber-hooks/) available in Cucumber including [***Tagged Hooks***](https://toolsqa.com/cucumber/tagged-hooks-in-cucumber/) as well.

### ***Exercise on Order Hooks***

Let’s take a different approach this time and do an exercise with the multiple hooks without any ordering value. Later we will bring order value and see the difference in output.

***Feature File***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Feature: Test Order Hooks    Scenario: First scenario to test Order Hooks functionality  Given this is the first step  When this is the second step  Then this is the third step    Scenario: Second scenario to test Order Hooks functionality  Given this is the first step  When this is the second step  Then this is the third step |

This is the same plain feature file that we used in previous chapters on Tags, Hooks, and Tagged Hooks.

***Step Definitions***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package stepDefinition;  import cucumber.api.java.en.Given;  import cucumber.api.java.en.Then;  import cucumber.api.java.en.When;    public class Hooks\_Steps {  @Given("^this is the first step$")  public void This\_Is\_The\_First\_Step(){  System.out.println("This is the first step");  }  @When("^this is the second step$")  public void This\_Is\_The\_Second\_Step(){  System.out.println("This is the second step");  }  @Then("^this is the third step$")  public void This\_Is\_The\_Third\_Step(){  System.out.println("This is the third step");  }  } |

Again, steps definitions are also same as previous chapters.

***Hooks***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | package utilities;  import cucumber.api.java.After;  import cucumber.api.java.Before;    public class Hooks {    @Before      public void beforeScenario(){          System.out.println("This will run before the every Scenario");      }  @Before      public void beforeScenarioStart(){          System.out.println("-----------------Start of Scenario-----------------");      }  @After      public void afterScenarioFinish(){          System.out.println("-----------------End of Scenario-----------------");      }  @After      public void afterScenario(){          System.out.println("This will run after the every Scenario");      }    } |

Above we mentioned two before and two after hooks. Execute the feature file as a whole and see the output below.

***Output***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | Feature: Test Order Hooks  -----------------Start of Scenario-----------------  This will run before the every Scenario  This is the first step  This is the second step  This is the third step  -----------------End of Scenario-----------------  This will run after the every Scenario  -----------------Start of Scenario-----------------  This will run before the every Scenario  This is the first step  This is the second step  This is the third step  -----------------End of Scenario-----------------  This will run after the every Scenario |

I would say that I want —–End of Scenario—— to be printed after the This will run after the every Scenario.

### ***How to set the Order or Priority of Cucumber Hooks?***

***The very important thing to note here is:***

* ***@Before(order = int) :*** This runs in increment order, means value 0 would run first and 1 would be after 0.
* ***@After(order = int) :*** This runs in decrements order, means apposite of @Before. Value 1 would run first and 0 would be after 1.

So, as per the logic above the Hooks file will look like below.

***Hooks***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | package utilities;    import cucumber.api.java.After;  import cucumber.api.java.Before;    public class Hooks {    @Before(order=1)      public void beforeScenario(){          System.out.println("This will run before the every Scenario");      }  @Before(order=0)      public void beforeScenarioStart(){          System.out.println("-----------------Start of Scenario-----------------");      }      @After(order=0)      public void afterScenarioFinish(){          System.out.println("-----------------End of Scenario-----------------");      }  @After(order=1)      public void afterScenario(){          System.out.println("This will run after the every Scenario");      }    } |

***Output***



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | Feature: Test Order Hooks  -----------------Start of Scenario-----------------  This will run before the every Scenario  This is the first step  This is the second step  This is the third step  This will run after the every Scenario  -----------------End of Scenario-----------------  -----------------Start of Scenario-----------------  This will run before the every Scenario  This is the first step  This is the second step  This is the third step  This will run after the every Scenario  -----------------End of Scenario----------------- |

Now just play around with the Hooks + Order, also try to figure out how it behaves when you use the Ordering with Tagged Hooks.