**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 till 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 allow to improve 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

If we were to summarize this as phases in development process we can write as

**Phase 1 (Requirement Definition)**

We will take a simple example of a calculator application and we will define the requirements based on the basic features of calculator. For further simplicity we will condense the calculator application to a simple java class named

*public class Calculator{*

*}*

In phase 1 application requirements are gathered and defined. Taking the example of a simple calculator we can say that in iteration 1 we would like to implement

*1. The ability to add two numbers*

*2. The ability to subtract two numbers*

*3. The ability to multiply two numbers*

***Requirement 1:*** *Calculator should have the ability to add two numbers.*

***Test 1:*** *Given two numbers positive numbers (10 and 20) calculator should be able to add the two numbers and give us correct result (30)*

***Test 2:*** *Given two negative numbers (-10 and -20) calculator should be able to add the two numbers and give us correct result (-30)*

This list of tests will go on and also for each requirement. In phase 1 all we have to do is to write tests for all the requirements. At this point in time, in the Calculator application we will just have a class called Calculator.



|  |  |
| --- | --- |
| 1  2  3  4  5 | package source;    public class Calculator {    } |

We will write all our tests against this class. Here is how our Test 1 will look like. We will put all our Adding tests in a class called ***AddingNumbersTests***



|  |  |
| --- | --- |
| 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 AppTests;    import org.junit.Assert;  import org.junit.Test;    import source.Calculator;    public class AddingNumbersTests {    private Calculator myCalculator = new Calculator();    @Test  public void addTwoPositiveNumbers()  {  int expectedResult = 30;  int ActuaResult = myCalculator.Add(10, 20);  Assert.assertEquals("The the sum of two positive numbers is correct" , expectedResult, ActuaResult);  }    @Test  public void addTwoNegativeNumbers()  {  int expectedResult = -30;  int ActuaResult = myCalculator.Add(-10, -20);  Assert.assertEquals("The the sum of two negative numbers is correct" , expectedResult, ActuaResult);  }  } |

Now the very first thing that will come to our mind is that Calculator class doesn’t have any methods and in our tests we have used a method named Add() on calculator class. This will give us compilation error.  
Well, that’s the whole point of writing the tests first. This will force us to add only the code that’s necessary. Ignoring the compilation error, lets just move on to the next step.

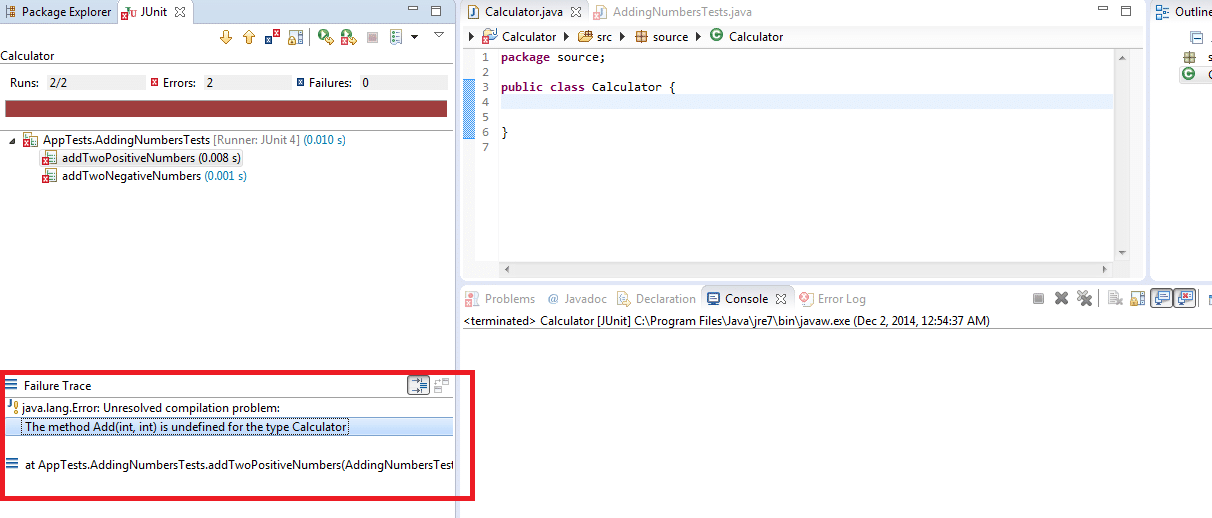
**Phase 2: Executing Tests**

In this phase we will simply run our tests. Lets do it one by one

**Attempt 1**: When we run our tests for the first time we will get this error message

*java.lang.Error: Unresolved compilation problem:*   
*The method Add(int, int) is undefined for the type Calculator*

This error clearly states that Add method is not present in the Calculator class. In details you can see in this screenshot

[](http://toolsqa.com/wp-content/uploads/2014/12/JunitFailure.png)<img class="alignnone wp-image-22249 size-full" src="https://toolsqa.com/wp-content/uploads/2014/12/JunitFailure.png" alt="What is Test Driven Development (TDD)" width="1212" height="518"/>

**Phase 3: Adding/Refactoring code**

After the test failure in the previous step, we will take a logical action and we will simply add a method called Add in our Calculator class and make it return 0 for the time being. Now our Calculator class will look something like this



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | package source;    public class Calculator {  public int Add(int number1, int number2)  {  return 0;  }  } |

With this change we will move to the next step that is rerun our tests. Which is nothing but Phase 2 mentioned earlier. Lets see what is the test result that we get this time.

***Results from the two tests is***

*java.lang.AssertionError: The the sum of two positive numbers is incorrect expected:<30> but was:<0>*  
*at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main(RemoteTestRunner.java:192)*

And

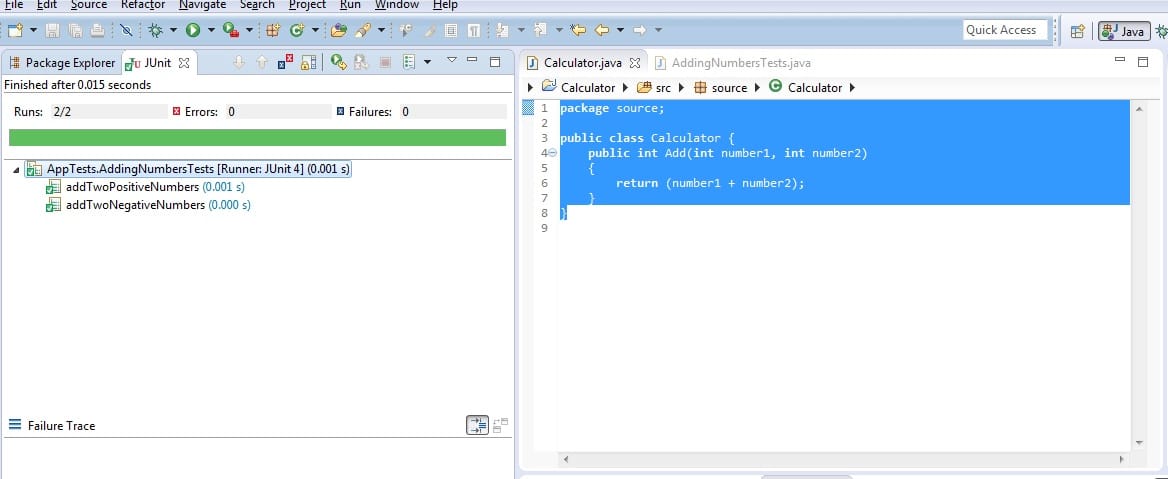
*java.lang.AssertionError: The the sum of two negative numbers is incorrect expected:<-30> but was:<0>*  
*at org.junit.Assert.fail(Assert.java:88)*

Now with this test failure we conclude that addition of two positive and negative numbers is not happening properly. Based on the test failure we will add just enough code that these two tests pass. As we do this we move to the next phase which is Phase 3. This phase is already describer earlier. I will just show you how the code of our Calculator will look like after this phase



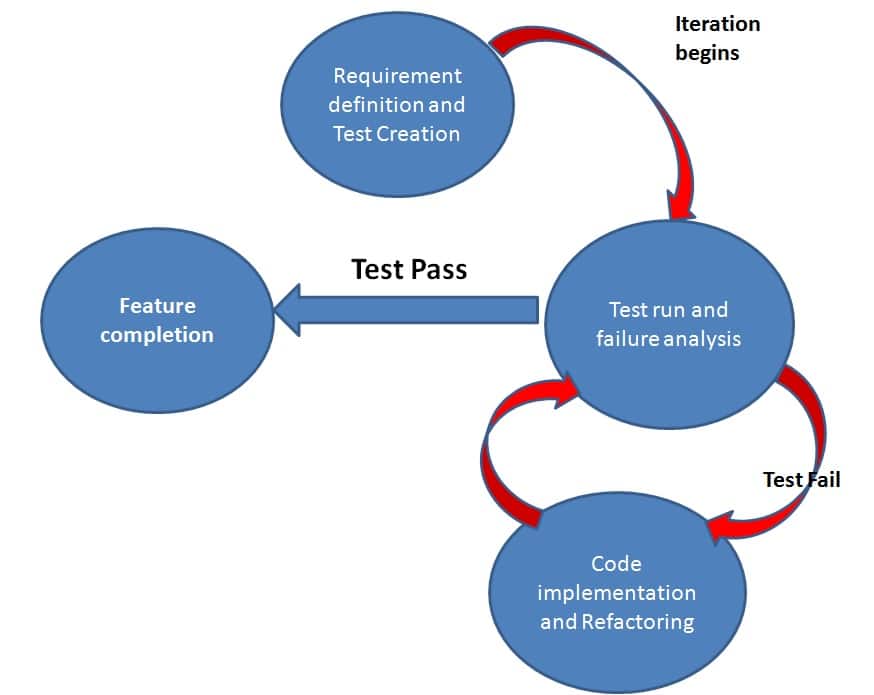
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | package source;    public class Calculator {  public int Add(int number1, int number2)  {  return (number1 + number2);  }  } |

Now we will run our tests Phase 2. The test results after this change will make all our tests pass. Once all the tests will pass we will conclude that our Iteration has been completed. Tests results:

[](http://toolsqa.com/wp-content/uploads/2014/12/PassedJunit.jpg)<img class="alignnone size-full wp-image-22250" src="https://toolsqa.com/wp-content/uploads/2014/12/PassedJunit.jpg" alt="PassedJunit" width="1168" height="479"/>

Once all the tests pass it signals the end of iteration. If there are more features that needs to be implemented in your product, product will go through the same phases again but this time with new feature set and more tests.

Summarizing it in a figure can be done like this

[](http://toolsqa.com/wp-content/uploads/2014/12/Cycle.jpg)

With this understanding of TDD we will move to BDD. Which will form the basis of understanding Gherkin and eventually Cucumber

**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 which 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***.

**Why BDD Framework?**

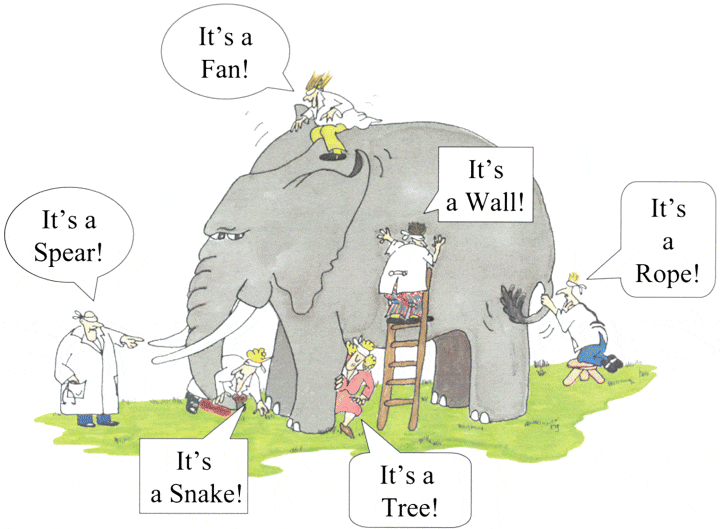
Let’s assume there is a requirement from a client for an E-Commerce website to increase the sales of the product with implementing some new features on the website. The only challenge of the development team is to convert the client idea in to something that actually delivers the benefits to client.

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**

[](http://toolsqa.com/wp-content/uploads/2014/12/Blind-Men-And-The-Elephant.gif)<img class="alignnone size-full wp-image-22317" src="https://toolsqa.com/wp-content/uploads/2014/12/Blind-Men-And-The-Elephant.gif" alt="Preception" width="720" height="529"/>  
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 in to 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***

Now after a look on the above example code anybody can understand the working of the test and what it is intend to do. It gives an unexpected powerful impact by enabling people to visualize the system before it has been built. Any of the business user would read and understand the test and able to give you the feedback that whether it reflects their understanding of what the system should do, and it can even leads to thinking of other scenarios that needs to be consider too.

**Gherkin**

***Gherkin – BDD*** ***Language (Business Driven Development)***. We will try to answer these questions in details

1. ***What is Gherkin?***
2. ***What is the use of Gherkin?***

Let’s start with some details

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

Before diving in to 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 with 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 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 Agile software projects succeeded thanks to the Behavior-Driven Development (BDD) method using Cucumber tool. So, what is Cucumber?
* ***Cucumber*** is a tool used to run automated acceptance tests created in a BDD format. One of its most outstanding features of the tool is the ability to carry out plain-text functional descriptions (*written in the language called* ***Gherkin***) 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 incredible feature of ***Behavior-Driven Development (BDD)*** approach with the advantages as below:
* *– Writing BDD tests in an omnipresent language, a language structured around the domain model and widely used by all team members comprising of developers, testers, BAs, and customers.   
       – Connecting technical with nontechnical members of a software team.  
       – Allowing direct interaction with the developer’s code, but BDD tests are written in a language which can also be made out by business stakeholders.  
       – Last but not least, acceptance tests can be executed automatically, while it is performed manually by business stakeholders.*

* **Cucumber helps improve communication**
* Cucumber helps improve communication between technical and non-technical members in the same project. Let’s have a look at the below requirement 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 be implemented 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);  } |

* The same test case can be written 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 perform well to complete the test automatically. But do all testers of your team make out these tests? Can other business analysts and other stakeholders use these tests again at the acceptance testing (AT) stage?
* The automation test with TestNG may be difficult for most manual testers and BAs to catch up with. Moreover, it is impossible to use this test again for AT. As a result, based on these flaws mentioned before, this can not be considered as a suitable method.
* In contrast, the automation test using Cucumber is created in a business domain language or in natural language, which can be easily made out by all members of the software project team. Communication is crucial for any development team, especially in the Agile team. There are usually many continuous chats, discussions, or even arguments happening among developers and testers to figure out what the correct behavior of a feature is. By using Cucumber, the same feature specification is now used for developing by developers, for testing by testers. It is considered to be a powerful tool because it can help lower the risk for misunderstanding as well as the communication breakdown.

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



|  |  |
| --- | --- |
| 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 |

* The automation test is written in 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       | |

* **All testers can take part in automation test with Cucumber BDD**
* In addition to improving communication among members of the same testing team, Cucumber also helps leverage tester’s skills efficiently. Expertise gap always exists in every organization. In other words, some testers have high 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, no matter what their skill levels are, can participate in the process of performing automation tests.
* Let’s take a look at the above example:  
  *– Any tester who is aware of the business logic and workflow can write feature files, add more scenarios, and test datasets.  
       – 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 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 potential issues when implementing Cucumber:
* Cucumber helps run test scenarios specified 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. Test scenarios should be presented clearly, and their implementation should perform accurately for each step. For instance, 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 ... |

* These steps may be incorporated 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 are performed in an ordinary language. They can be used again in various test scenarios. This 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; few steps (bold) can be reused:Both the above scripts are correct; 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. Imagine if you want to extend the test to search more texts, you may repeat the above step, and the Google site is consequently opened twice. If you do not strictly follow the requirement, 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 |

* In contrast, if the test is generic and can be reused, i.e., verifying updating user’s Last Name, non-technical stakeholders will have difficulty in catching up with and performing 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 until they reach entirely an acceptable balance where all members can understand and reuse.

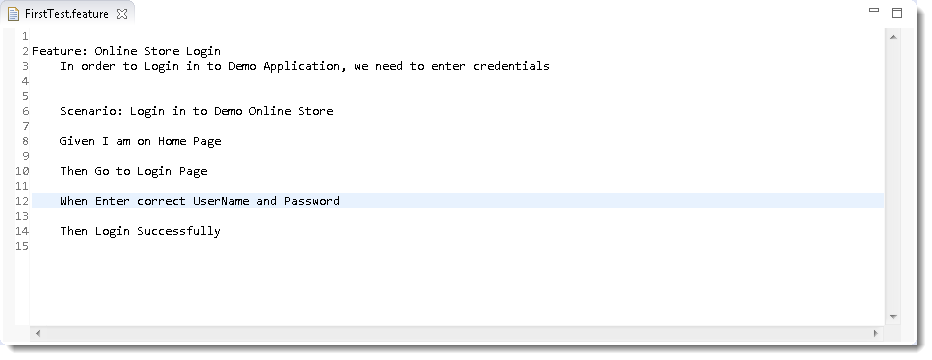
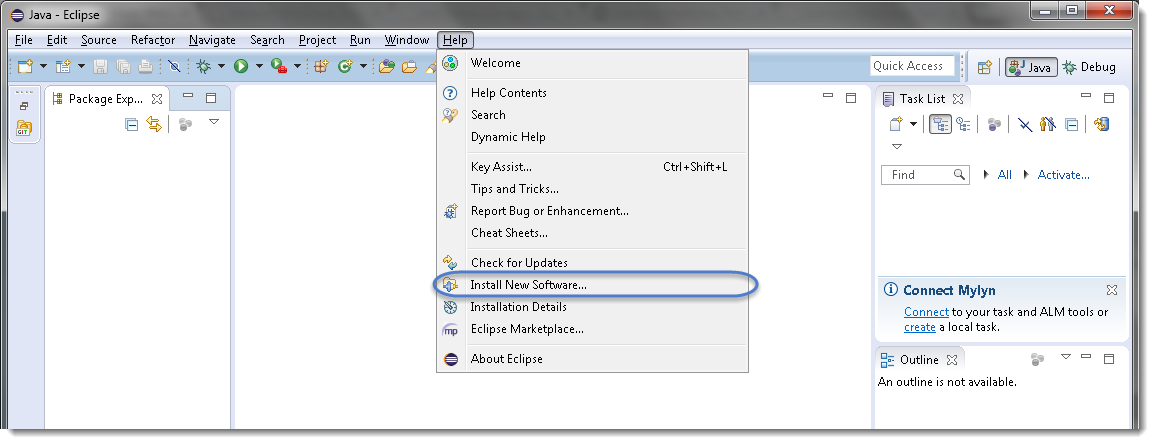
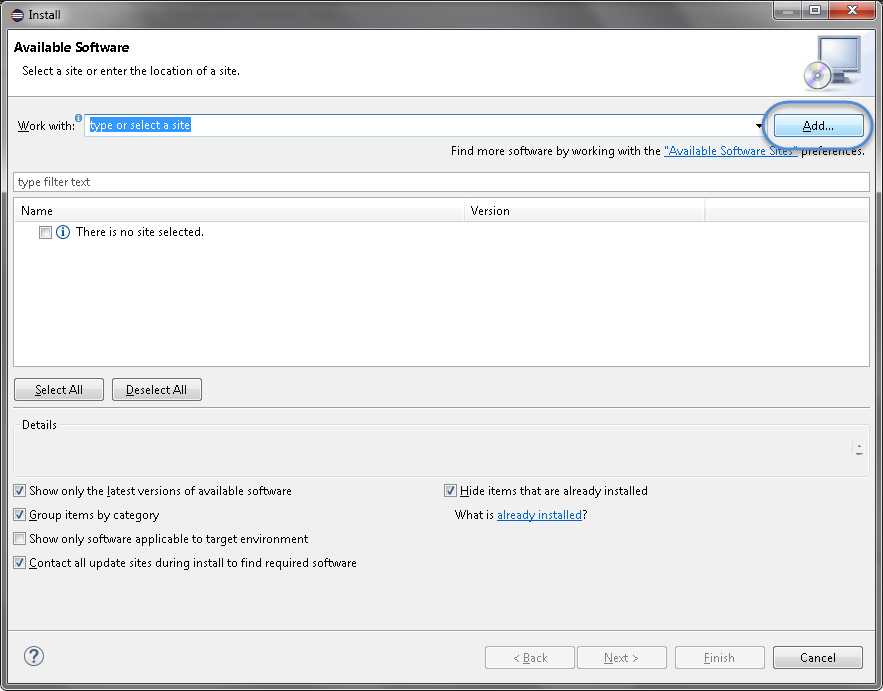


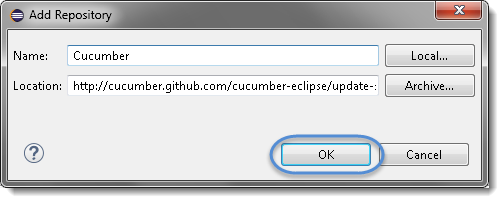
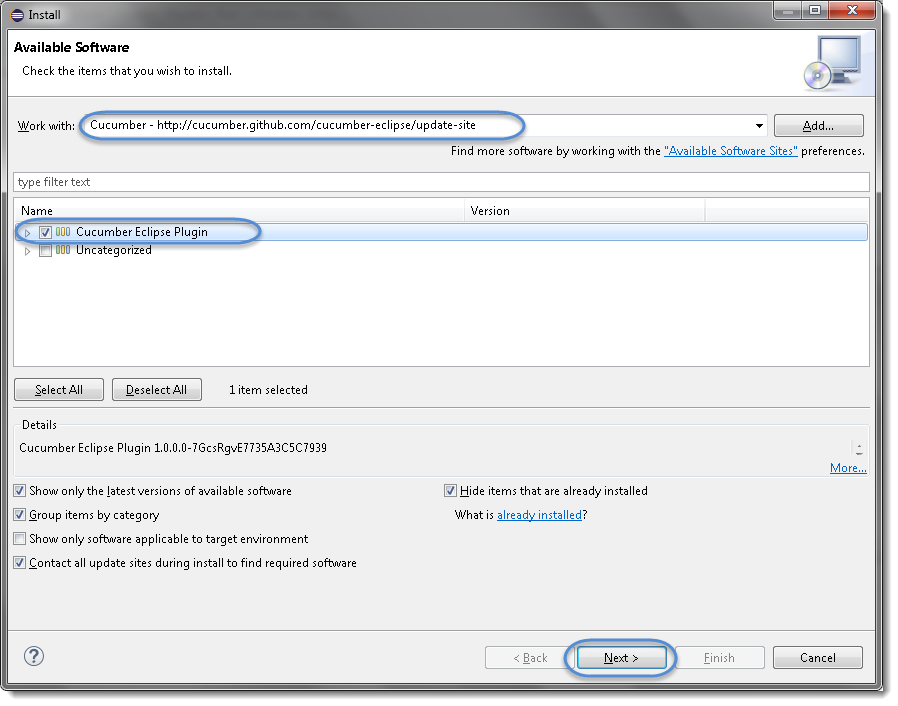
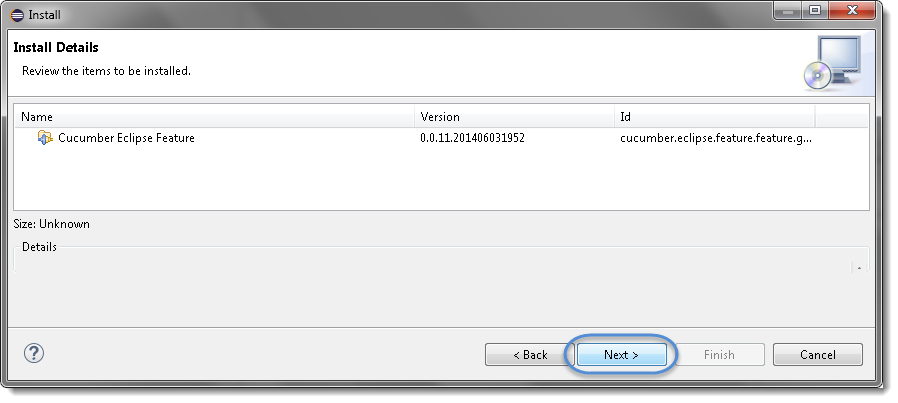
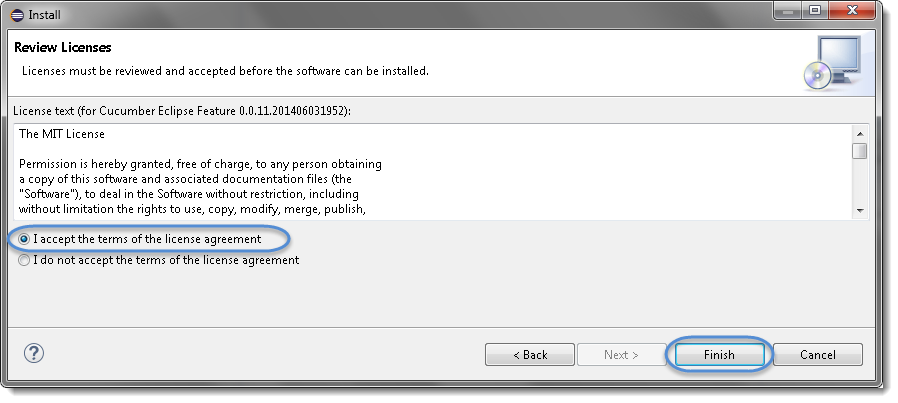
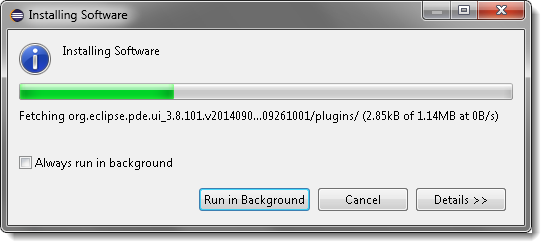
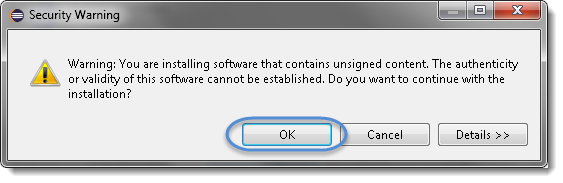
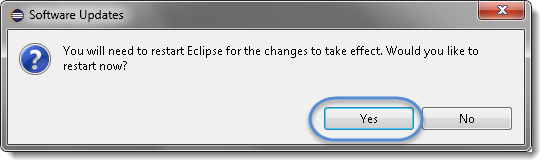
|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Scenario: Verify Updating user password feature   Given I am in the HR system with "Admin" account   And there is another user with "$Test123" password   When I update password of the user to "@Test123"   Then I got the message "Password changed successfully."   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.*  
  *– An appropriate editor tool should be considered. 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 which 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. Domain business logic is not contained.*
* All things considered, Cucumber is one of the most powerful tools to offer 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 backend to frontend. Moreover, Cucumber creates deep connections among members of the testing team which is hardly found in other testing frameworks. With many years of automation testing experience, I recommend that Cucumber for Web UI and Web service testing should be implemented to help Agile software projects to be operated successfully.
* ***Cucumber Eclipse Plugin*** as it is not the main cucumber plugin for running the test as BDD. Even this is not a mandatory plugin but it is quite handy. At start when you write the feature file for the cucumber test, the code for the feature file looks like this:
* <img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_Eclipse\_Plugin\_910.png" alt="Cucumber\_Eclipse\_Plugin\_910"/>
* ***Note:*** *None of the keyword in feature file is highlighted, all the text is plain.*
* ***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*** to setup Eclipse to you system.
* **Steps to follow:**
* 1) Launch the *Eclipse IDE* and from Help menu, click “***Install New Software***”.
* <img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_Eclipse\_Plugin\_1.png" alt="Cucumber\_Eclipse\_Plugin\_1"/>
* 2) You will see a dialog window, click “***Add***” button.
* <img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_Eclipse\_Plugin\_2.png" alt="Cucumber\_Eclipse\_Plugin\_2"/>

* 3) Type name as you wish, let’s take “***Cucumber***” and type “***http://cucumber.github.com/cucumber-eclipse/update-site***” as location. Click ***OK***.
* <img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_Eclipse\_Plugin\_3.png" alt="Cucumber\_Eclipse\_Plugin\_3"/>
* 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.
* <img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_Eclipse\_Plugin\_4.png" alt="Cucumber\_Eclipse\_Plugin\_4"/>
* ***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***.
* <img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_Eclipse\_Plugin\_5.png" alt="Cucumber\_Eclipse\_Plugin\_5"/>
* 6) Click “***I accept the terms of the license agreement***” then click ***Finish***.
* <img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_Eclipse\_Plugin\_6.png" alt="Cucumber\_Eclipse\_Plugin\_6"/>
* 7) Let it install, it will take few seconds to complete.
* <img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_Eclipse\_Plugin\_7.png" alt="Cucumber\_Eclipse\_Plugin\_7"/>
* 8) You may or may not encounter a Security warning, if in case you do just click ***OK***.
* <img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_Eclipse\_Plugin\_8.png" alt="Cucumber\_Eclipse\_Plugin\_8"/>
* 9) You are all done now, just Click **Yes**.
* 

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

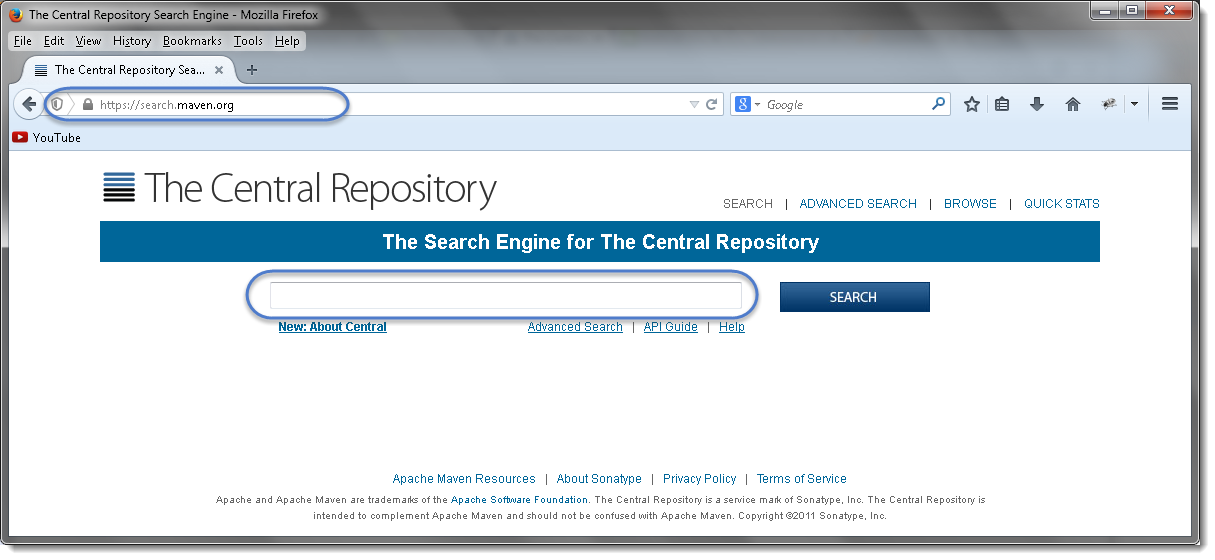
1. ***cucumber-core***
2. ***cucumber-java***
3. ***cucumber-junit***
4. ***cucumber-jvm-deps***
5. ***cucumber-reporting***
6. ***gherkin***
7. ***junit***
8. ***mockito-all***
9. ***cobertura***

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

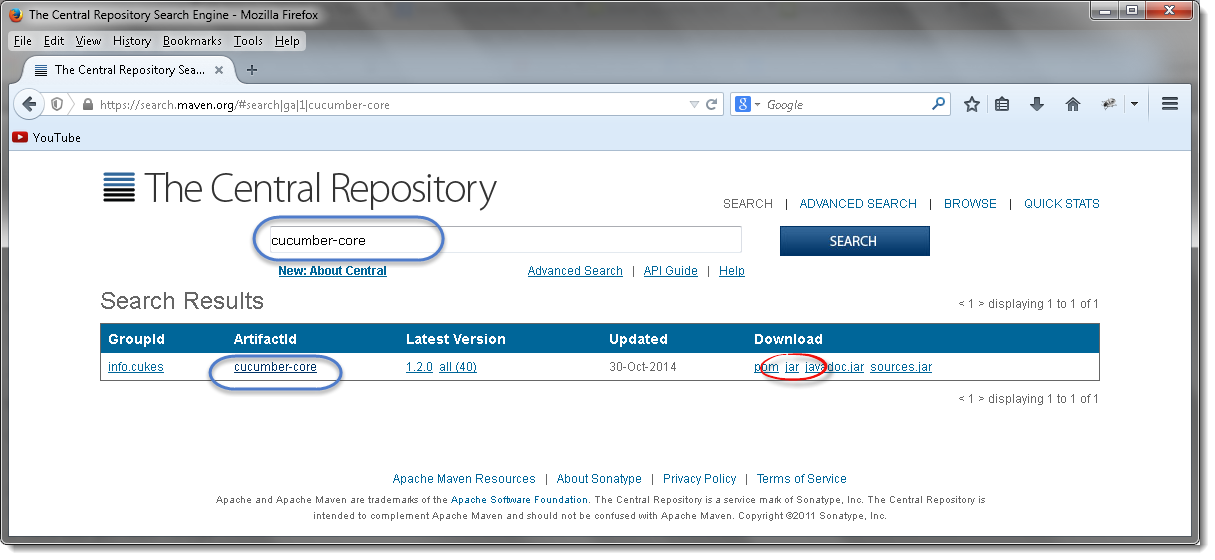
* ***Download Cucumber Jars from Online Maven Repository***
* ***Download Cucumber Jars from oss.sonatype.org***
* ***Download Cucumber Jars from Maven dependencies (Easiest)***

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

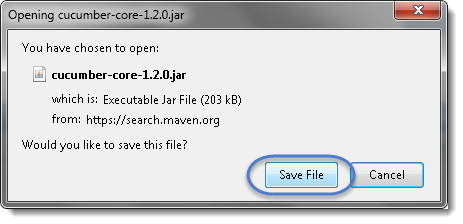
1) Go to ***https://search.maven.org****.*

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_1.png" alt="Download Cucumber JVM for Eclipse" width="1215" height="560"/>

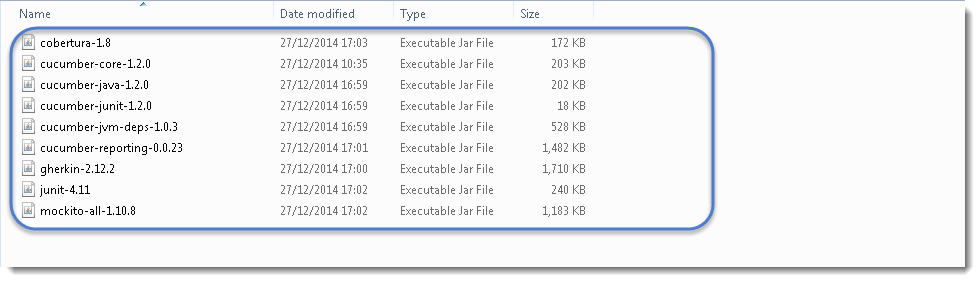
2) Search for ***cucumber-core***in the Central Maven Repository. It will return the *Cucumber Core* jars. Click on the ***jar***.

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_2.png" alt="Cucumber\_2"/>

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

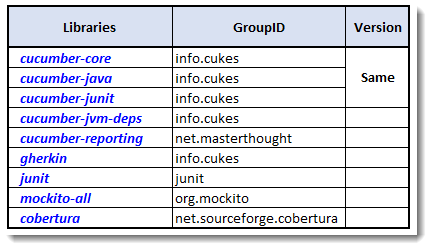
<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_3.png" alt="Cucumber\_3"/>

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

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_7.png" alt="Cucumber\_7"/>

***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.

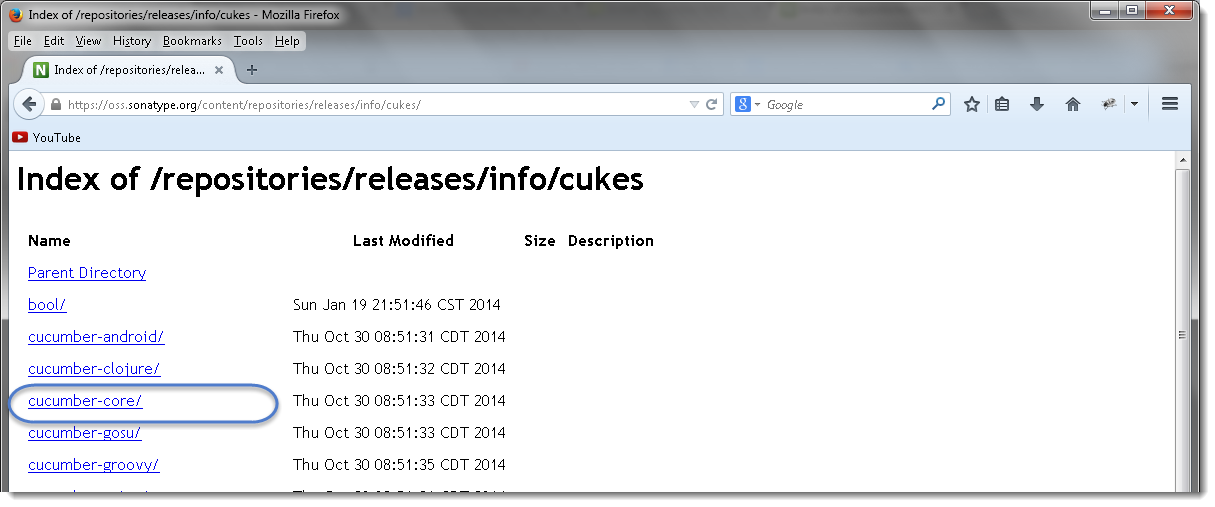
<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber-Libraries.png" alt="Cucumber Libraries"/>

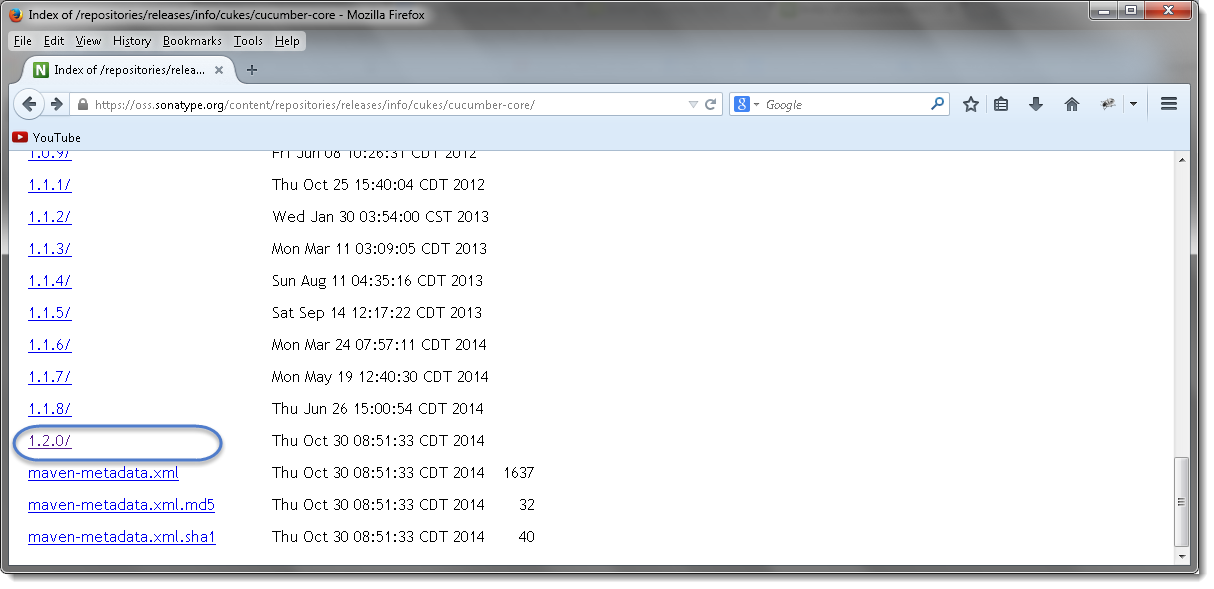
***I got below versions on Oct’17 for Cucumber***

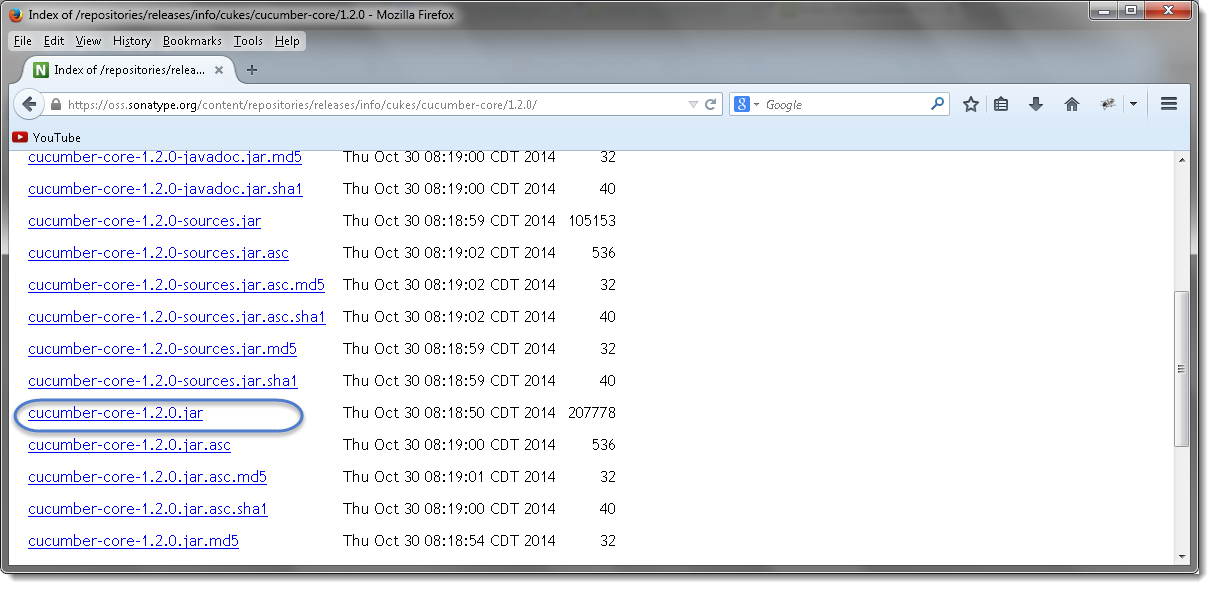
* *cobertura-2.1.1*
* *cucumber-core-1.2.5*
* *cucumber-java-1.2.5*
* *cucumber-junit-1.2.5*
* *cucumber-jvm-deps-1.0.5*
* *cucumber-reporting-3.10.0*
* *gherkin-2.12.2*
* *junit-4.12*
* *mockito-all-2.0.2-beta*

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

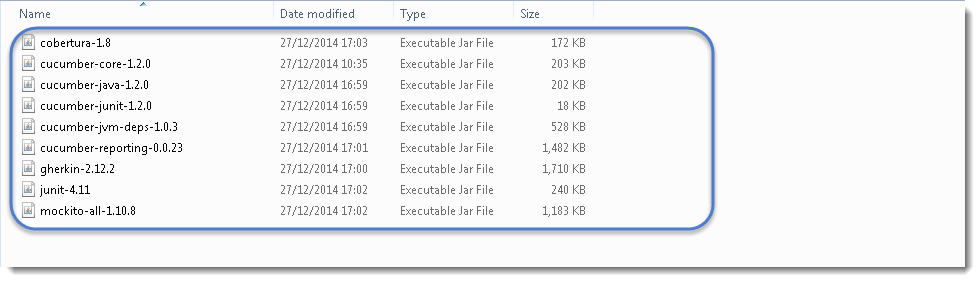
1) Go to ***https://oss.sonatype.org/content/repositories/releases/info/cukes/ .***Here also all the cucumber jars are available. Start with ***cucumber-core****.*

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_4.png" alt="Cucumber\_4"/>

2) Click on the latest version at the bottom of the page, which is ***version 1.2.0*** as of now by Dec’14.  
<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_5.png" alt="Cucumber\_5"/>

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*** .  
<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_6.png" alt="Cucumber\_6"/>

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

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_7.png" alt="Cucumber\_7"/>

***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***
2. ***Create a New Maven Project in 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***. 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.*

YAML



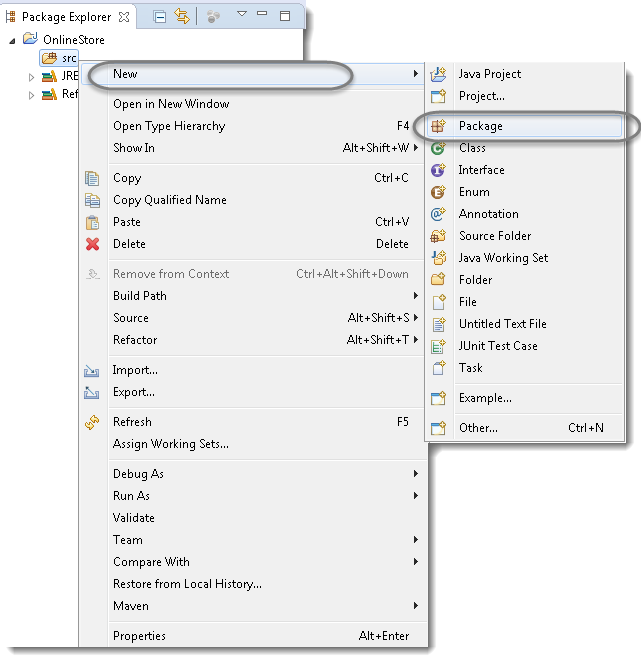
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-java</artifactId>  <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> |

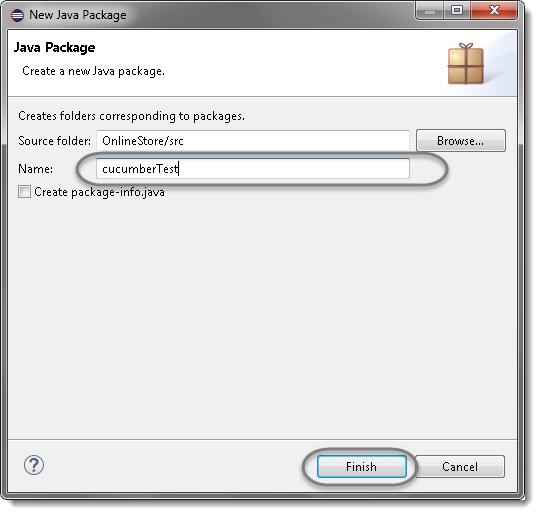
1. ***Download & Install Java***
2. ***Download and Install Eclipse***
3. ***Install Cucumber Eclipse Plug-in***
4. ***Download Cucumber***
5. ***Download Selenium WebDriver Client***
6. ***Configure Eclipse with Selenium & Cucumber***

***Create Folder Structure***

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

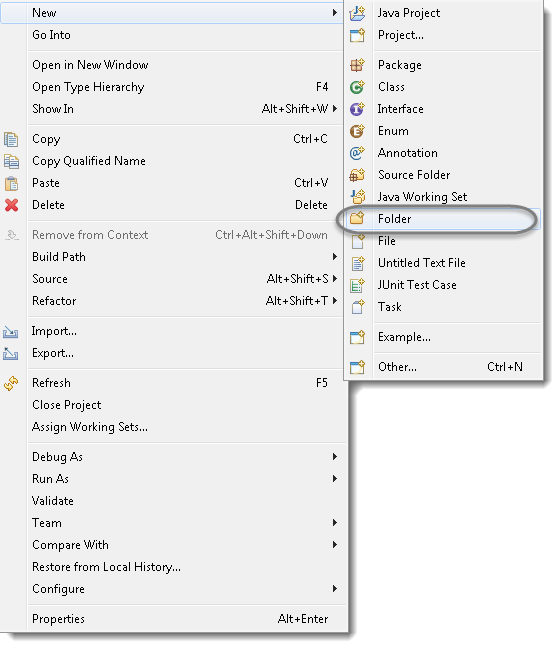
1) Create a new ***Package*** by *right click* on the ‘***src***‘ folder and select *New > Package*.

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/First\_Cucumber\_Test\_1.png" alt="First\_Cucumber\_Test\_1"/>

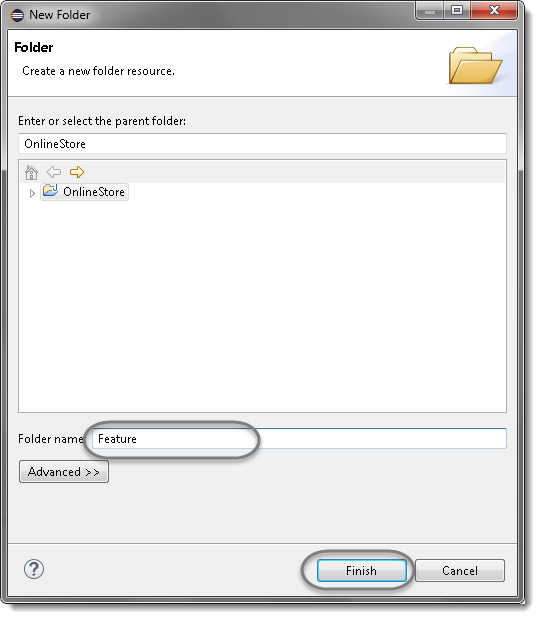
2) Name it as ‘***cucumberTest***’ and click on *Finish* button.  
<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/First\_Cucumber\_Test\_2.png" alt="First\_Cucumber\_Test\_2"/>

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*.

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/First\_Cucumber\_Test\_3.png" alt="First\_Cucumber\_Test\_3"/>

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

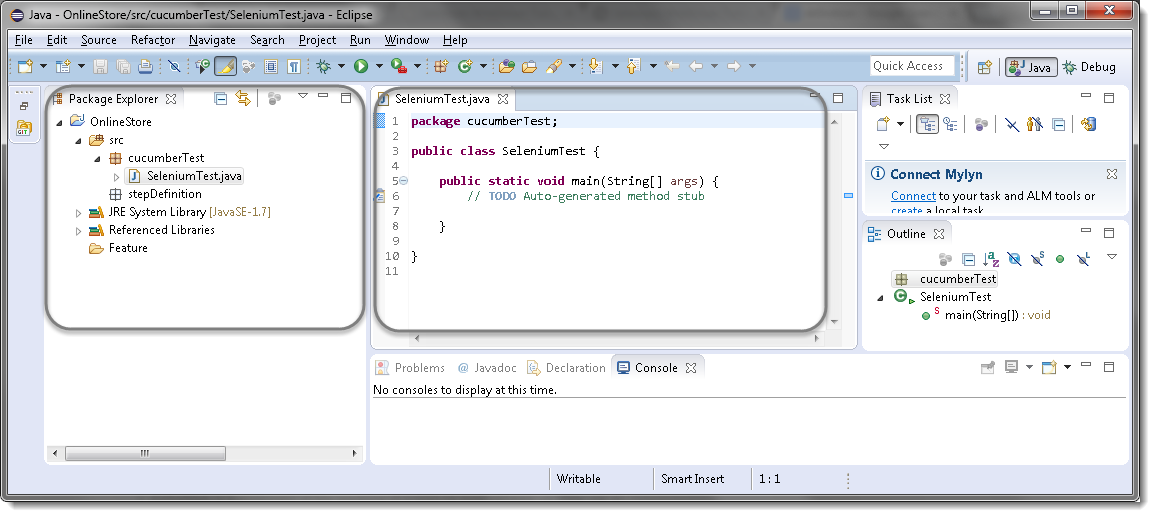
<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/First\_Cucumber\_Test\_4.png" alt="First\_Cucumber\_Test\_4"/>

**Selenium Java Test**

Lets first write a simple ***Selenium Test script*** for ***LogIn*** functionality and then convert that script in to *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:

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/First\_Cucumber\_Test\_5.png" alt="First\_Cucumber\_Test\_5"/>

***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("http://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();    }    } |

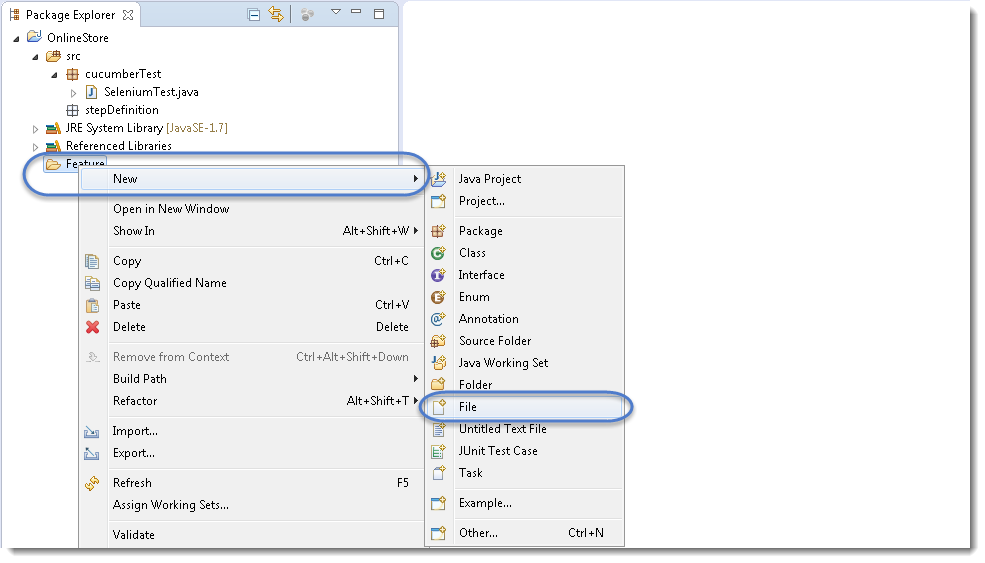
***Note****: If the Selenium version is less than 3.0, above test will work for you. If the version is above 3.0, in that case please look at the chapter* ***How to Use Gecko Driver in Selenium 3***

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.***

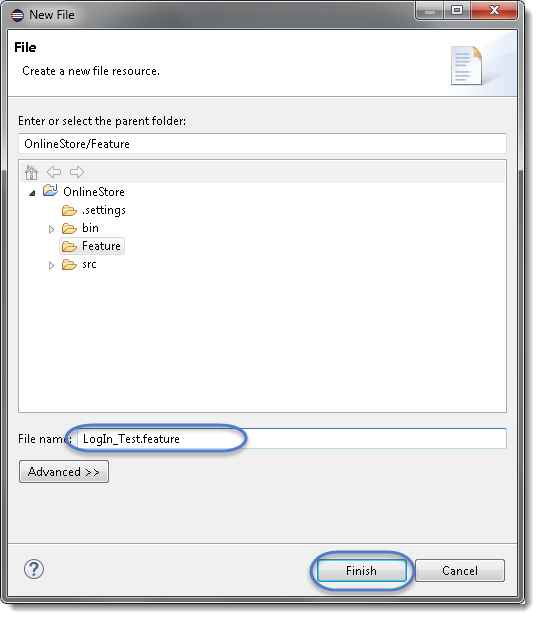
**What is Cucumber Feature File?**

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***

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/First\_Cucumber\_Test\_6.png" alt="Cucumber Feature File" width="985" height="563"/>

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.

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/First\_Cucumber\_Test\_7.png" alt="First\_Cucumber\_Test\_7"/>

***Note****: In case you get a pop up from Eclipse which suggest you to install the better Editor for BDD files, please go ahead and install that. At the botttom 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 details in next chapter.*

***Keywords***

Now moving forward we have just defined a test. You will notice colored part 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 following tutorials. But to start off we can quickly explain some of the keywords in one line. Note this is not 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 expected result of the test****.*

***Gherkin***

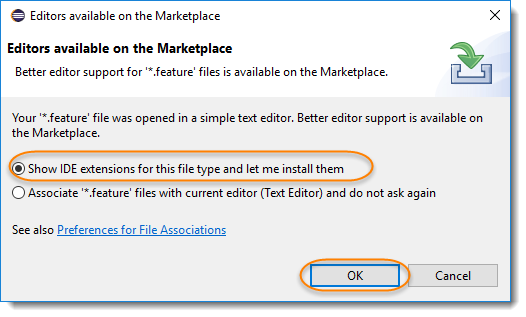
A language above is called ***Gherkin*** and it implements the principles of ***Business readable domain specific language(BRDSL)***. Domain specific language gives you the ability to describe your application behavior without getting into details of implementation. What does that mean? If we go back to our tutorial in ***TDD*** 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*** 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 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.

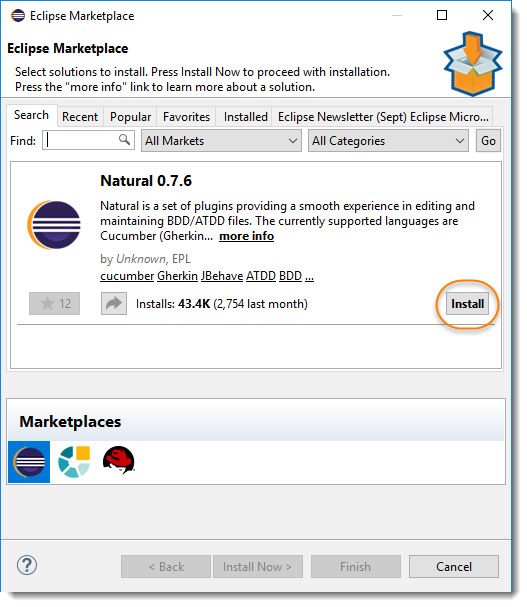
***Steps to install the Natural Eclipse Editor for Gherkin***

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.

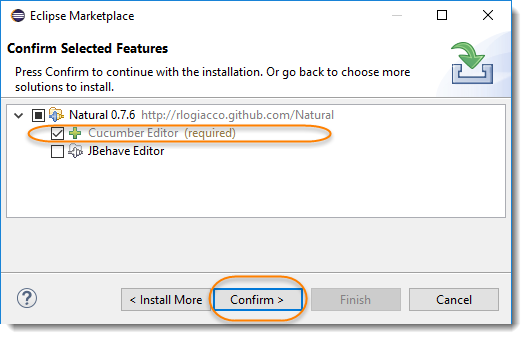
1)Just select the first option of ***Show IDE extensions*** if it is not pre-selected and click OK.

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_Feature\_SetUp-1.png" alt="Cucumber\_Feature\_SetUp 4"/>

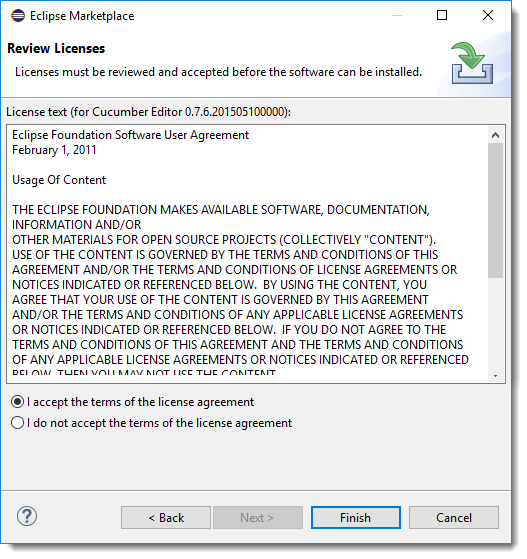
2) Natural is the name of the plugin, so this can also be found ***Eclipse Marketplace***. Just click Install.

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_Feature\_SetUp-2.png" alt="Cucumber\_Feature\_SetUp 4"/>

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.

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/Cucumber\_Feature\_SetUp-3.png" alt="Cucumber\_Feature\_SetUp 4"/>

4) Last step is to accept the Terms and Conditions.



**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 need *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***.

***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 the curios minds, I will explain this code. Note that it is covered in details 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.

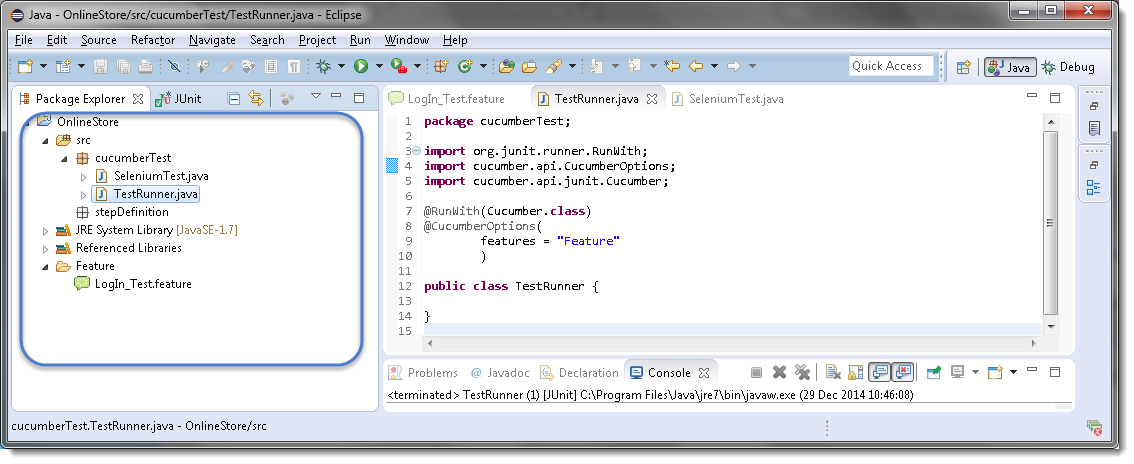
Even from the IDE, there are a couple of ways to run these feature files.

* *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.

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/First\_Cucumber\_Test\_8.png" alt="First\_Cucumber\_Test\_8"/>

Now your project should look like this in Eclipse IDE:

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/First\_Cucumber\_Test\_9.png" alt="First\_Cucumber\_Test\_9"/>

***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.***

*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 than add new. Keeping both may create further issues for you.*

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

***Solution***

*This means that 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***

***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*

***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*** 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 like shown above in 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*** file and ***SpecFlow Feature*** file

**Background: Keyword**

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

**Scenario: Keyword**

Each Feature will contain some 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 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

**What is Cucumber Options ?**

In layman language ***@CucumberOptions*** are like property file 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 few options in the ‘***TestRunner’*** class in the previous chapter.

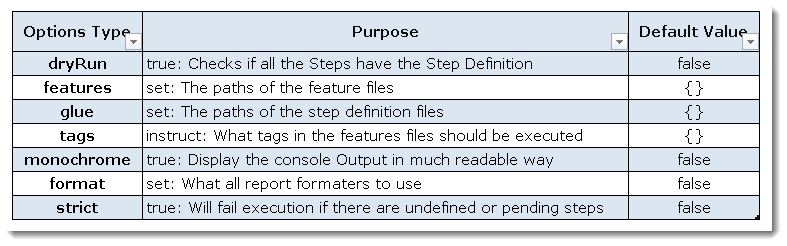
***TestRunner Class***



|  |  |
| --- | --- |
| 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 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:

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/CucumberOptions\_7.png" alt="Cucumber Options" width="787" height="246"/>

***Dry Run***

***dryRun*** option can either set as ***true***or ***false***. If it is set as *true*, it means that *Cucumber* will only checks that every *Step* mentioned in the *Feature File* have corresponding code written in *Step Definition* file or not. So in case any of the function is missed in the *Step Definition* for any *Step* in *Feature File*, it will give us the message. 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.

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/CucumberOptions\_1.png" alt="CucumberOptions\_1"/>

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 have 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:

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/CucumberOptions\_6.png" alt="CucumberOptions\_6"/>

***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***

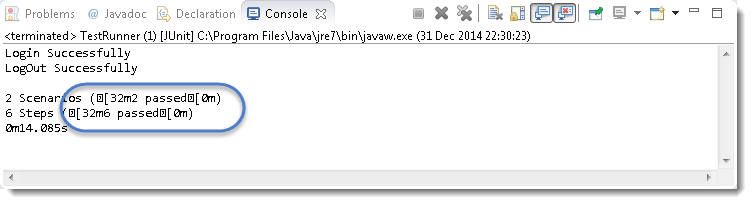


|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | package cucumberTest;    import org.junit.runner.RunWith;  import cucumber.api.CucumberOptions;  import cucumber.api.junit.Cucumber;    @RunWith(Cucumber.class)  @CucumberOptions(  features = "Feature"  ,glue={"stepDefinition"}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

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/CucumberOptions\_2.png" alt="CucumberOptions\_2"/>

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

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/CucumberOptions\_3.png" alt="CucumberOptions\_3"/>

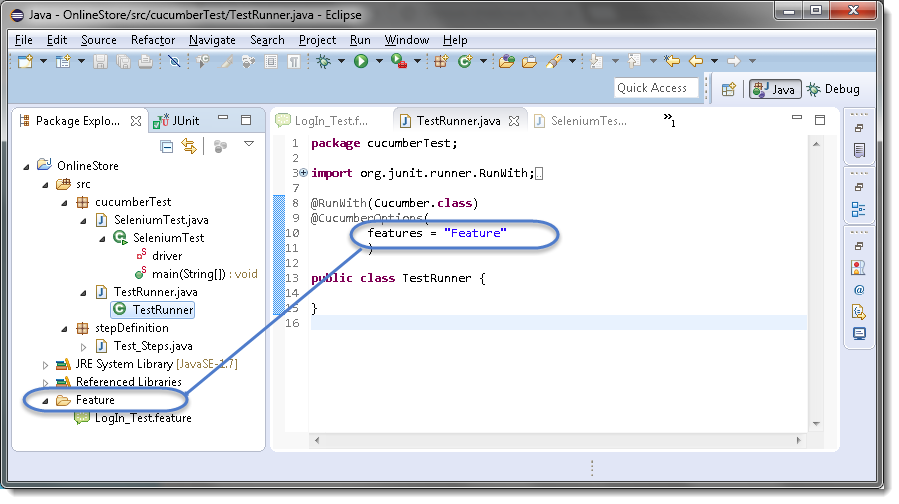
***Features***

***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“***

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/CucumberOptions\_4.png" alt="CucumberOptions\_4"/>

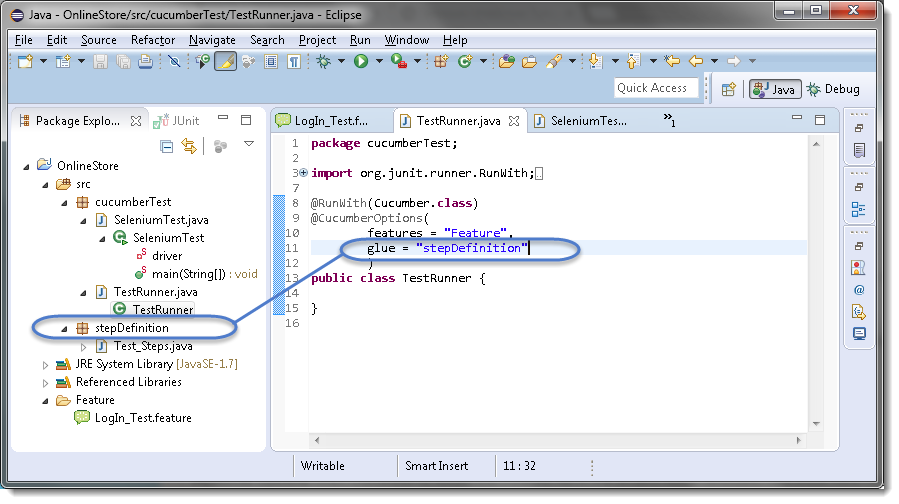
***Glue***

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“***

<img class="ngg-singlepic ngg-none" src="https://toolsqa.com/wp-content/gallery/cucumber/CucumberOptions\_5.png" alt="CucumberOptions\_5"/>

***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 colours 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“}***