**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 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/).

**Why BDD Framework?**

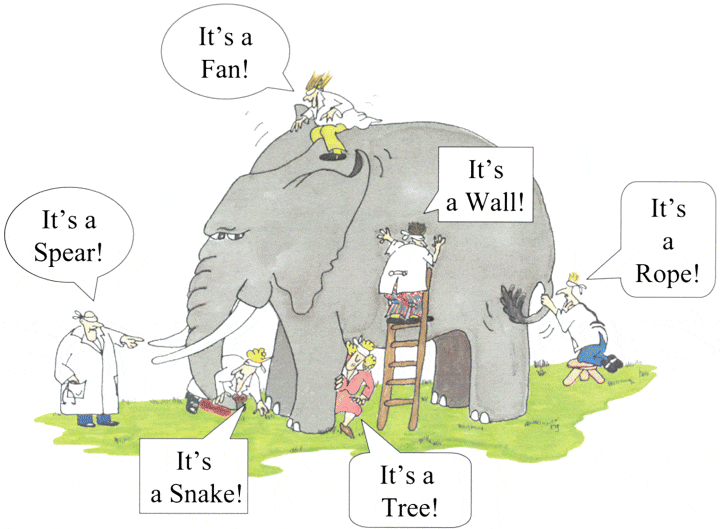
Let's assume there is a requirement from a client for an E-Commerce website to increase the sales of the product with implementing some new features on the website. The only challenge of the development team is to convert the client idea into something that actually delivers the benefits to client.

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***



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***

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

***Gherkin in BDD***

***1. What is Gherkin?***

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

Let's start with some details

**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 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***

package cucumberTest;

import org.junit.runner.RunWith;

import cucumber.api.CucumberOptions;

import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)

@CucumberOptions(

features = "Feature"

,glue={"stepDefinition"}

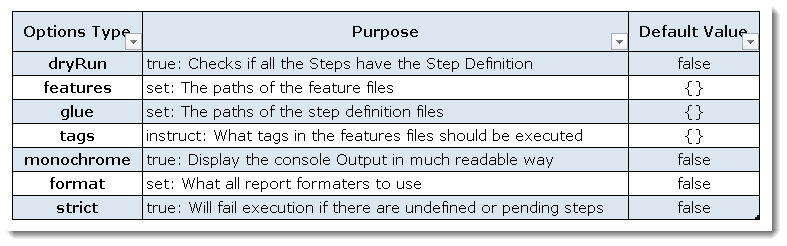
)

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***

package cucumberTest;

import org.junit.runner.RunWith;

import cucumber.api.CucumberOptions;

import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)

@CucumberOptions(

features = "Feature"

,glue={"stepDefinition"}

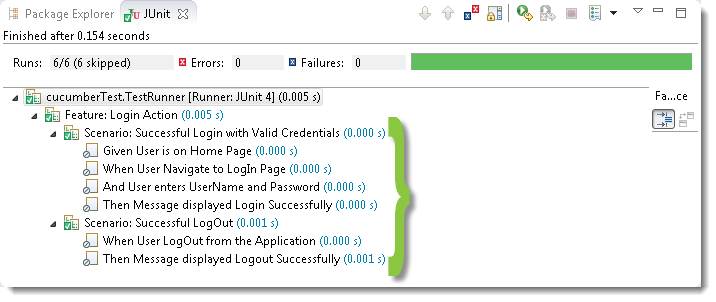
,dryRun = true

)

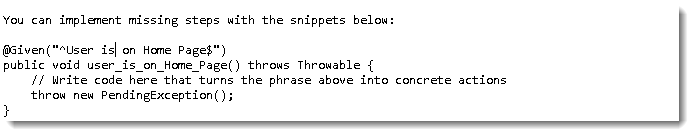
public class TestRunner {

}

*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***

package cucumberTest;

import org.junit.runner.RunWith;

import cucumber.api.CucumberOptions;

import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)

@CucumberOptions(

features = "Feature"

,glue={"stepDefinition"}

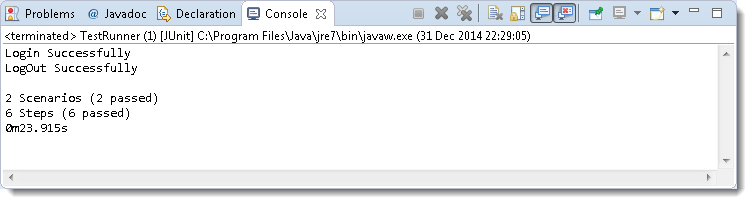
,monochrome = false

)

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:



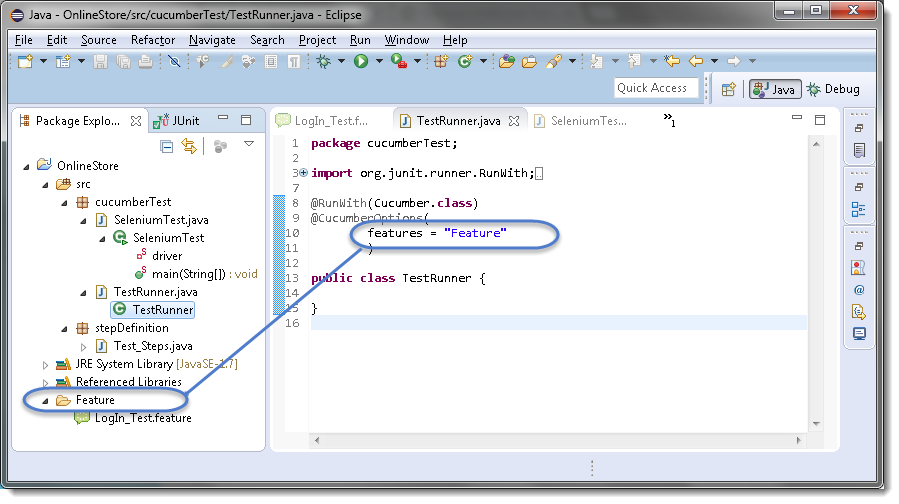
***Features***

***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"***



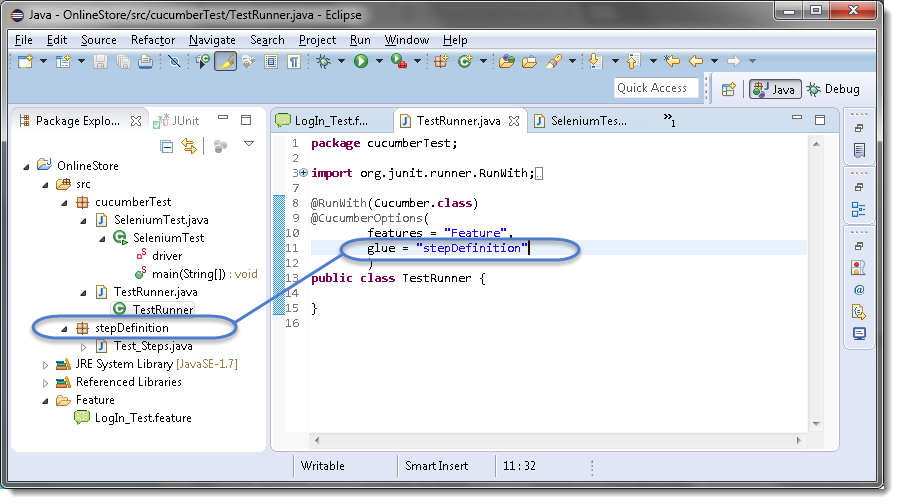
***Glue***

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"}***

**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 Annotations***](https://toolsqa.com/testng/testng-annotations/), 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***

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***

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***

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");

}

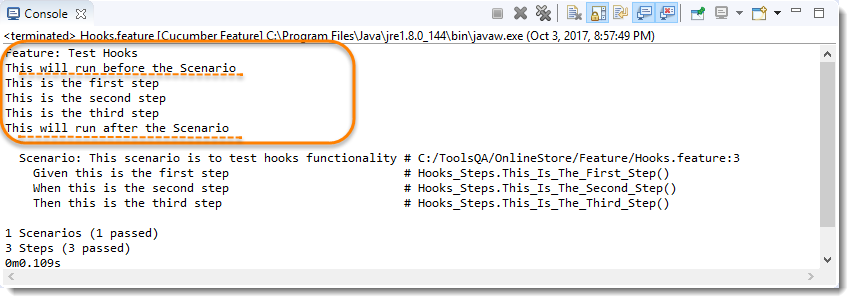
}

***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.

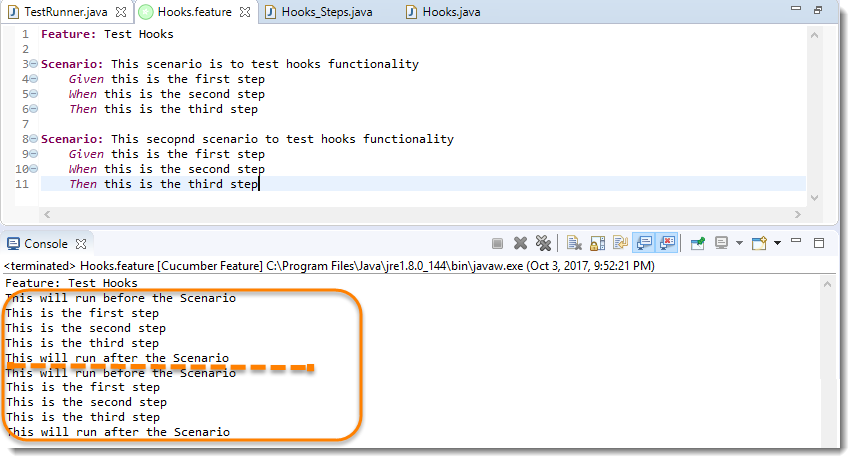
***Output***



No need for explanation, it is self-explanatory :)

***Test Hooks with Multiple Scenarios***

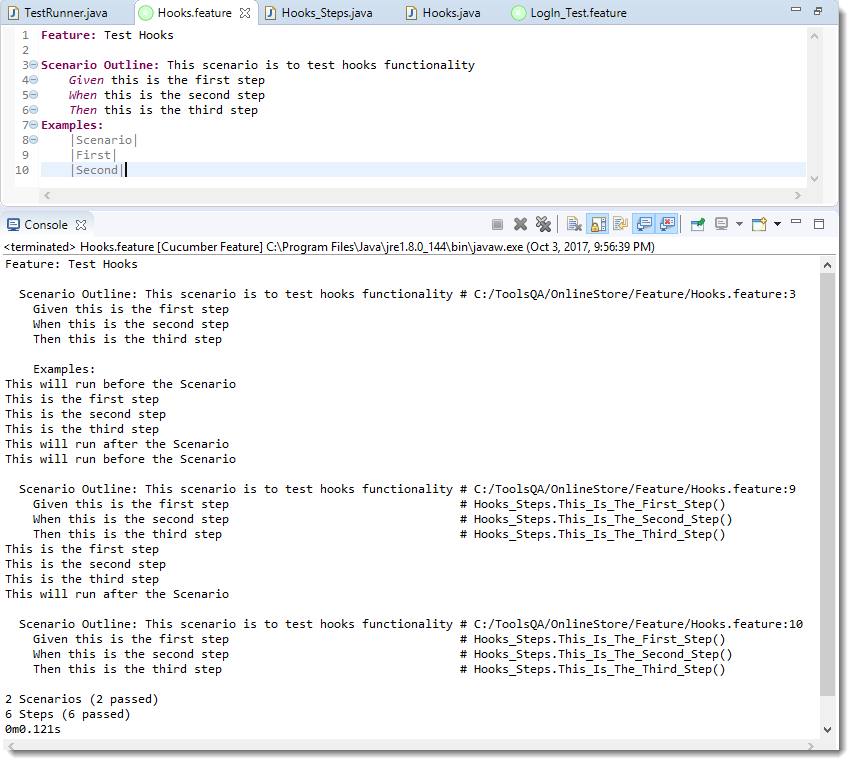
I just wanted to show you the reaction of Hooks with the multiple scenarios. Let's just add one more Test Scenario in the feature file and run the feature again.



***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***

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

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

***Step Definitions***

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");

}

}

1. 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***

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.*

1. Run the feature file and observe the output.

***Output***

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***

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***

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

**Cucumber Tags**

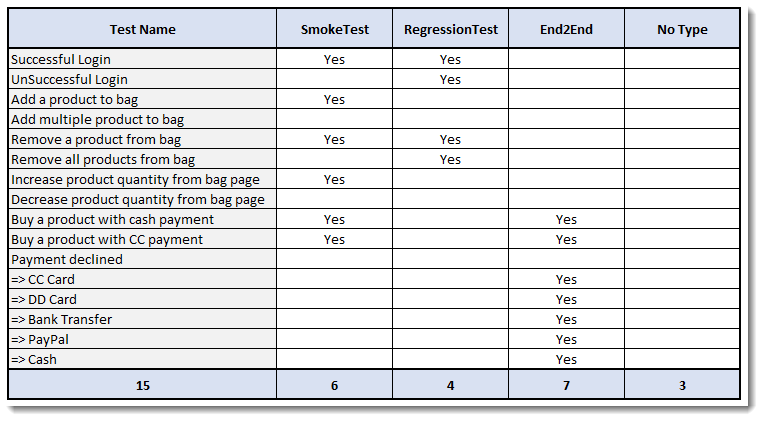
Let's say you have got many different feature files that cover all the different functionality of the application. Now there can be a certain situation in the project where you like to execute just a ***SmokeTests*** or ***End2EndTests*** or may be ***RegressionTests***. One approach is that you start 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***

Scenario: Increa@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

@SmokeTestse 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|

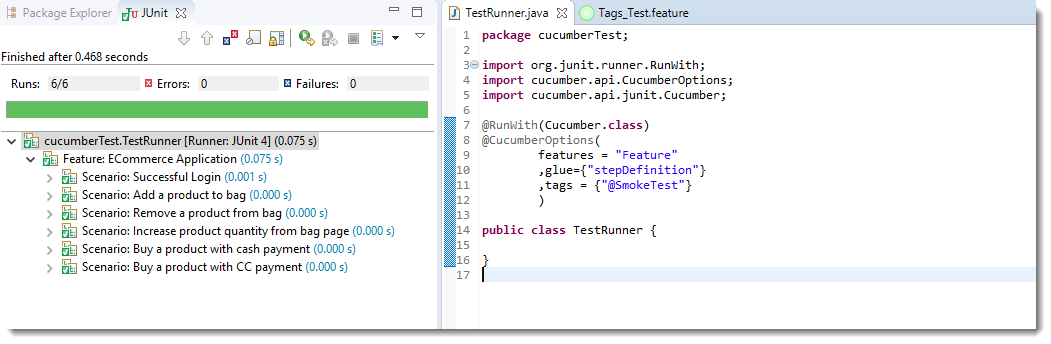
|DD Card|

|Bank Transfer|

|PayPal|

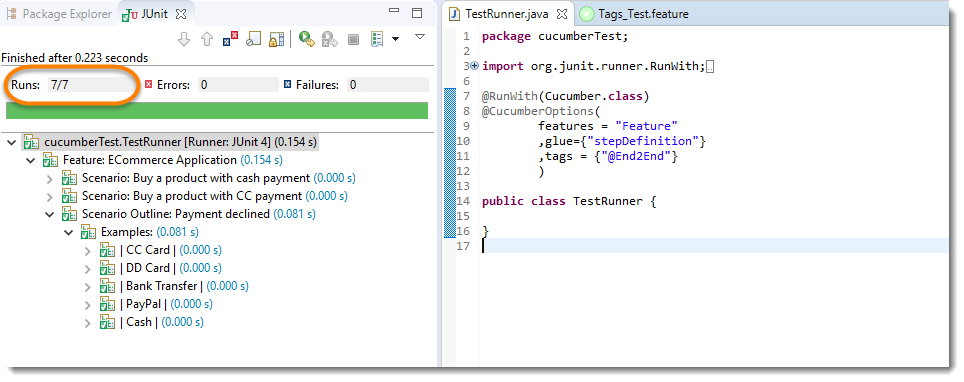
|Cash|

***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*.

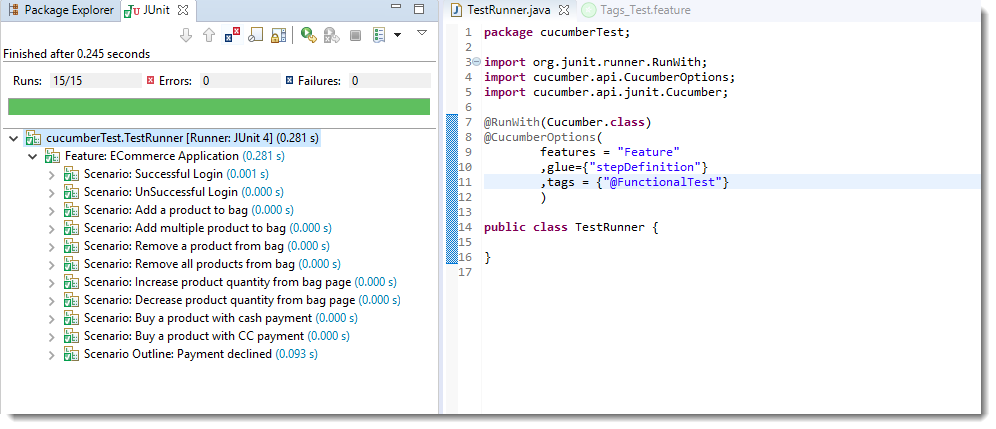
***Execute all tests tagged as @End2End***



***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.



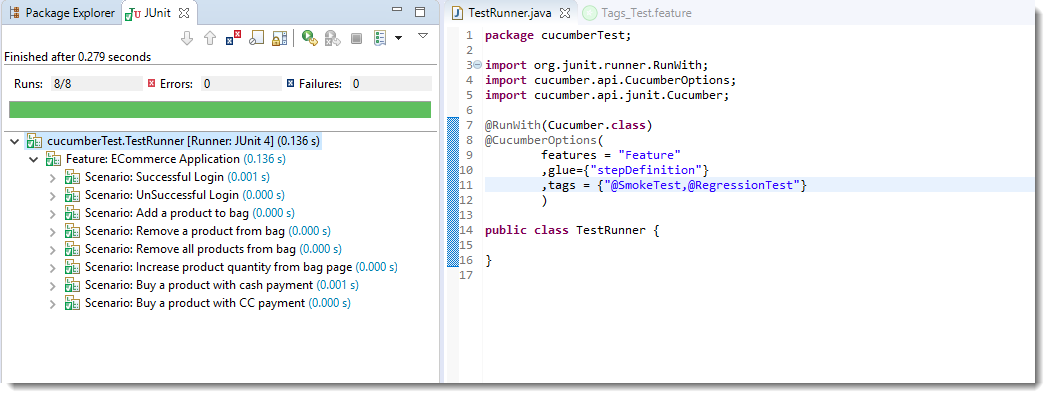
***Note****: All the test exists in the feature file are executed.*

**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***

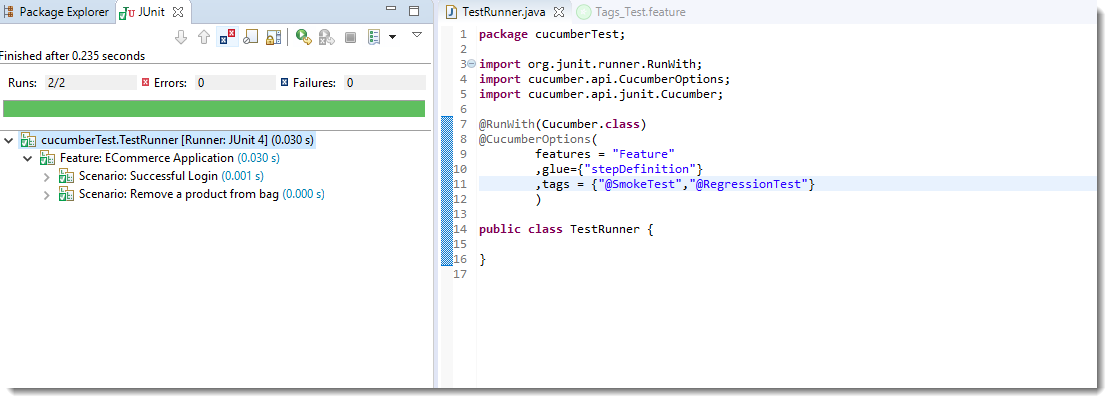
Tags that are ***comma-separated*** are ORed.



***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

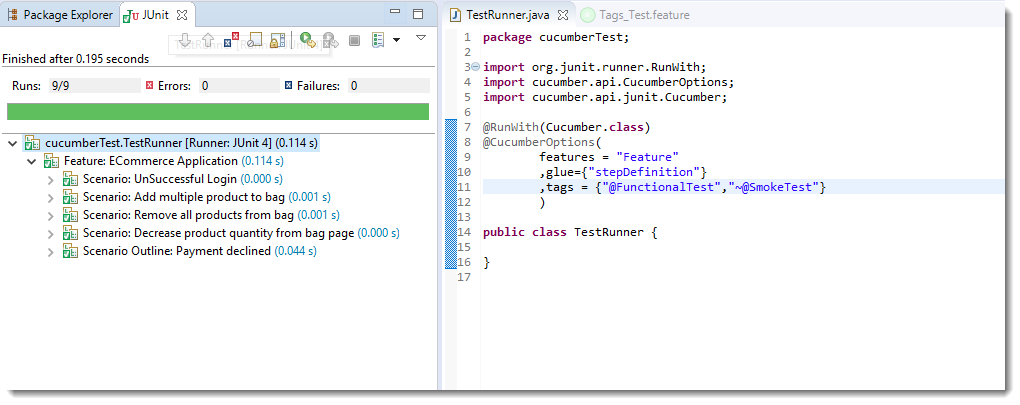


***Note****: There are only two scenarios in our feature file which have both tags together*.

**How to Ignore Cucumber Tests**

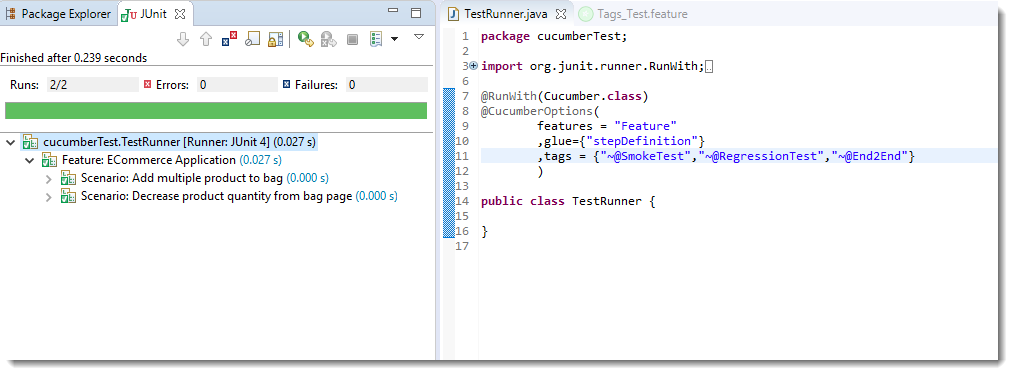
This is again a good feature of Cucumber Tags that you can even skip tests in the group execution. Special Character ~ is used to skip the tags. This also works both for *Scenarios* and *Features*. And this can also works in conjunction with AND or OR.

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

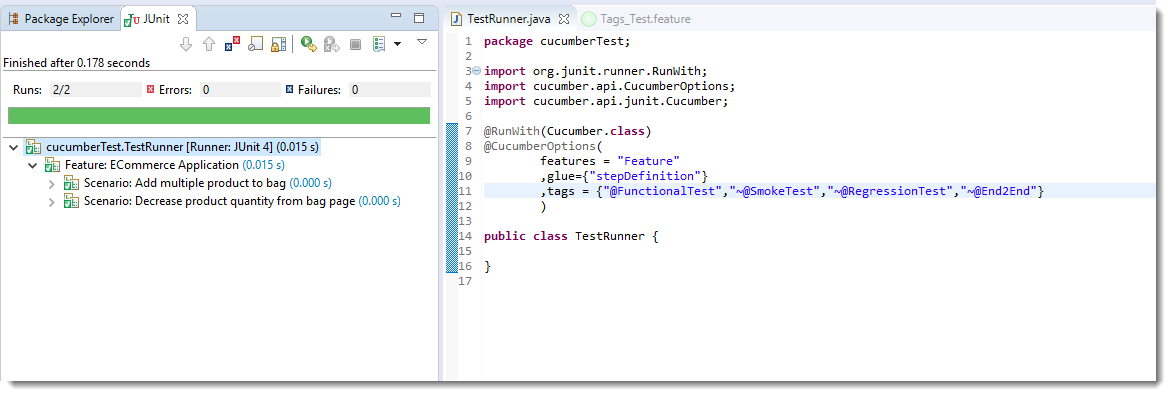


***Note****: This is AND condition, which means all the scenarios tagged as @FunctionalTest but not @SmokeTest. So total tests are 15 and smoke tests are 6, so it ran just 9 tests.*

***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.

**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***

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***

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***

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***

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***

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***

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.

**Background in Cucumber**

Let's start with a simple exercise to build the understanding of Background usage in Cucumber test. If we create a feature file of the scenario we explained above, this is how it will look like:

***Feature File***

Feature: Test Background Feature

Description: The purpose of this feature is to test the Background keyword

Background: User is Logged In

Given I navigate to the login page

When I submit username and password

Then I should be logged in

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

Given User search for Lenovo Laptop

When Add the first laptop that appears in the search result to the basket

Then User basket should display with added item

Scenario: Navigate to a product and add the same to the User basket

Given User navigate for Lenovo Laptop

When Add the laptop to the basket

Then User basket should display with added item

In the above example, we have two different scenarios where a user is adding a product from search and directly from the product page. But the common step is to logIn to website for both the scenario. *This is why we create another Scenario for LogIn but named it as Background rather than a Scenario*. So that it executes for both the Scenarios

***Step Definitions***

package stepDefinition;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

public class BackGround\_Steps {

@Given("^I navigate to the login page$")

public void i\_navigate\_to\_the\_login\_page() throws Throwable {

System.out.println("I am at the LogIn Page");

}

@When("^I submit username and password$")

public void i\_submit\_username\_and\_password() throws Throwable {

System.out.println("I Submit my Username and Password");

}

@Then("^I should be logged in$")

public void i\_should\_be\_logged\_in() throws Throwable {

System.out.println("I am logged on to the website");

}

@Given("^User search for Lenovo Laptop$")

public void user\_searched\_for\_Lenovo\_Laptop() throws Throwable {

System.out.println("User searched for Lenovo Laptop");

}

@When("^Add the first laptop that appears in the search result to the basket$")

public void add\_the\_first\_laptop\_that\_appears\_in\_the\_search\_result\_to\_the\_basket() throws Throwable {

System.out.println("First search result added to bag");

}

@Then("^User basket should display with added item$")

public void user\_basket\_should\_display\_with\_item() throws Throwable {

System.out.println("Bag is now contains the added product");

}

@Given("^User navigate for Lenovo Laptop$")

public void user\_navigate\_for\_Lenovo\_Laptop() throws Throwable {

System.out.println("User navigated for Lenovo Laptop");

}

@When("^Add the laptop to the basket$")

public void add\_the\_laptop\_to\_the\_basket() throws Throwable {

System.out.println("Laptop added to the basket");

}

}

Just printing the appropriate logs in the steps. But we would bring the advance usage of the same in our next series of ***Designing Automation Framework with Cucumber.***

***Output***

Feature: Test Background Feature

Description: The purpose of this feature is to test the Background keyword

I am at the LogIn Page

I Submit my Username and Password

I am logged on to the website

User searched for Lenovo Laptop

First search result added to bag

Bag is now contains the added product

I am at the LogIn Page

I Submit my Username and Password

I am logged on to the website

User navigated for Lenovo Laptop

Laptop added to the basket

Bag is now contains the added product

***Note****: Hope you noticed the impact. The background ran two times in the feature before each scenario.*

***Background with Hooks***

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

To get it straight, let's assign a task to the Before & After Hook in the same test.

* *@Before: Print the starting logs*
* *@Before: Start browser and Clear the cookies*
* *@After:  Close the browser*
* *@After: Print the closing logs*

***Hooks File***

package utilities;

import cucumber.api.java.After;

import cucumber.api.java.Before;

public class Hooks {

@Before(order=1)

public void beforeScenario(){

System.out.println("Start the browser and Clear the cookies");

}

@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("Log out the user and close the browser");

}

}

**Page Object Model**

Page Object Model, also known as POM, is a [design pattern in Selenium](https://www.browserstack.com/guide/design-patterns-in-selenium) that creates an object repository for storing all web elements. It helps reduce code duplication and improves test case maintenance.

In Page Object Model, consider each web page of an application as a class file. Each class file will contain only corresponding web page elements. Using these elements, testers can perform operations on the website under test.

**Read More:** [Design Patterns in Automation Framework](https://www.browserstack.com/guide/design-patterns-in-automation-framework)

#### **Advantages of Page Object Model**

* **Easy Maintenance**: POM is useful when there is a change in a UI element or a change in action. An example would be: a drop-down menu is changed to a radio button. In this case, POM helps to identify the page or screen to be modified. As every screen will have different Java files, this identification is necessary to make changes in the right files. This makes test cases easy to maintain and reduces errors.
* [**Code Reusability**](https://www.browserstack.com/guide/importance-of-code-reusability): As already discussed, all screens are independent. By using POM, one can use the test code for one screen, and reuse it in another test case. There is no need to rewrite code, thus saving time and effort.
* **Readability and Reliability of Scripts**: When all screens have independent java files, one can quickly identify actions performed on a particular screen by navigating through the java file. If a change must be made to a specific code section, it can be efficiently done without affecting other files.

#### **Implementing POM in Selenium Project**

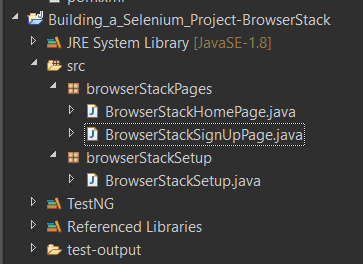
As already discussed, each java class will contain a corresponding page file. This tutorial will create 2-page files.

* BrowserStackHomePage
* BrowserStackSignUpPage

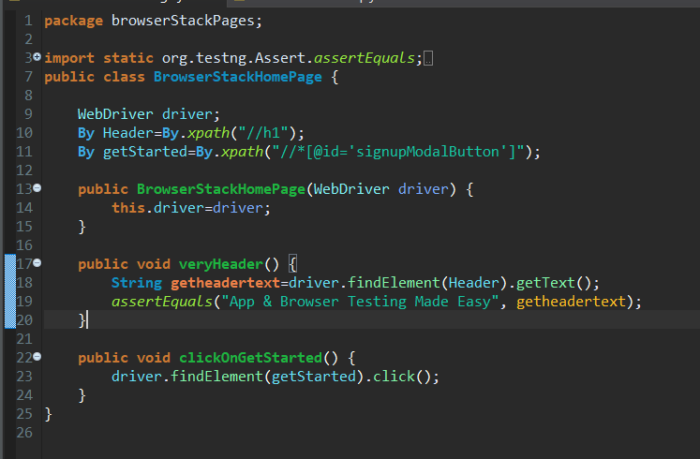
Each of these files will contain UI elements or Objects which are present on these screens. It will also contain the operations to be performed on these elements.

**Also Read:** [How to Build and Execute Selenium Projects](https://www.browserstack.com/guide/build-and-execute-selenium-projects)

#### **Sample Project Structure for POM**



**BrowserStackHomePage Java File**

  
**Explanation of Code**

* **Code Line-10 to 11:** Identifying elements present on BrowserStack Home Page such as **header** and **Get Started** button
* **Code Line-17 to 24:** Performing actions on identified objects on BrowserStack Home Page

**Code Snippet**

package browserStackPages;

import static org.testng.Assert.assertEquals;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

public class BrowserStackHomePage {

WebDriver driver;

By Header=By.xpath("//h1");

By getStarted=By.xpath("//\*[@id='signupModalButton']");

public BrowserStackHomePage(WebDriver driver) {

this.driver=driver;

}

public void veryHeader() {

String getheadertext=driver.findElement(Header).getText();

assertEquals("App & Browser Testing Made Easy", getheadertext);

}

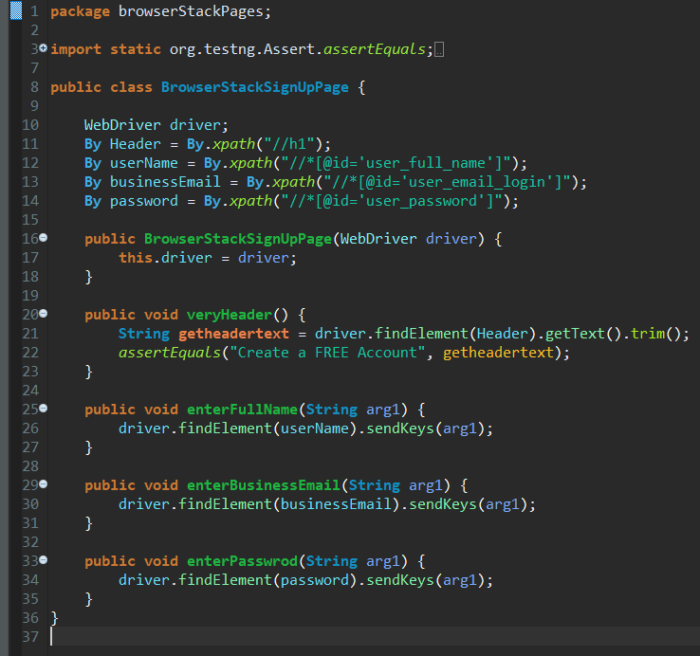
public void clickOnGetStarted() {

driver.findElement(getStarted).click();

}

}

**BrowserStackSignUpPage Java File**



**Explanation of Code**

* **Code Line-10 to 14:** Identifying elements present on [BrowserStack SignUp Page](https://www.browserstack.com/users/sign_up" \o "Sign Up on BrowserStack" \t "_blank) such as **header** and **Get Started** button
* **Code Line-20 to 35:** Performing actions on identified objects on the BrowserStack SignUp Page

**Code Snippet**

package browserStackPages;

import static org.testng.Assert.assertEquals;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

public class BrowserStackSignUpPage {

WebDriver driver;

By Header = By.xpath("//h1");

By userName = By.xpath("//\*[@id='user\_full\_name']");

By businessEmail = By.xpath("//\*[@id='user\_email\_login']");

By password = By.xpath("//\*[@id='user\_password']");

public BrowserStackSignUpPage(WebDriver driver) {

this.driver = driver;

}

public void veryHeader() {

String getheadertext = driver.findElement(Header).getText().trim();

assertEquals("Create a FREE Account", getheadertext);

}

public void enterFullName(String arg1) {

driver.findElement(userName).sendKeys(arg1);

}

public void enterBusinessEmail(String arg1) {

driver.findElement(businessEmail).sendKeys(arg1);

}

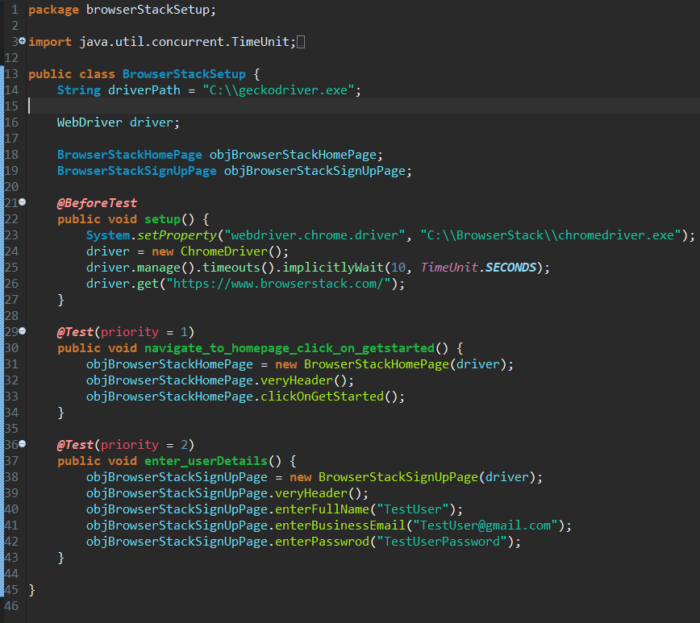
public void enterPasswrod(String arg1) {

driver.findElement(password).sendKeys(arg1);

}

}

**BrowserStackSetup Java File**



**Explanation of Code**

* **Code Line-21 to 27:** Setting up browser and website to execute test scripts
* **Code Line-29 to 43:** Initializing driver object to **BrowserStackHomePage** & **BrowserStackSignUpPage** and performing actions on those pages

**Code Snippet**

package browserStackSetup;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

import org.testng.annotations.BeforeTest;

import org.testng.annotations.Test;

import browserStackPages.BrowserStackHomePage;

import browserStackPages.BrowserStackSignUpPage;

public class BrowserStackSetup {

String driverPath = "C:\\geckodriver.exe";

WebDriver driver;

BrowserStackHomePage objBrowserStackHomePage;

BrowserStackSignUpPage objBrowserStackSignUpPage;

@BeforeTest

public void setup() {

System.setProperty("webdriver.chrome.driver", "C:\\BrowserStack\\chromedriver.exe");

driver = new ChromeDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.get("https://www.browserstack.com/");

}

@Test(priority = 1)

public void navigate\_to\_homepage\_click\_on\_getstarted() {

objBrowserStackHomePage = new BrowserStackHomePage(driver);

objBrowserStackHomePage.veryHeader();

objBrowserStackHomePage.clickOnGetStarted();

}

@Test(priority = 2)

public void enter\_userDetails() {

objBrowserStackSignUpPage = new BrowserStackSignUpPage(driver);

objBrowserStackSignUpPage.veryHeader();

objBrowserStackSignUpPage.enterFullName("TestUser");

objBrowserStackSignUpPage.enterBusinessEmail("TestUser@gmail.com");

objBrowserStackSignUpPage.enterPasswrod("TestUserPassword");

}

}

[Run Selenium Tests](https://www.browserstack.com/users/sign_up?ref=guide-page-object-model-in-selenium-mid&product=automate)

### What is Page Factory in Selenium?

Page Factory is a class provided by [Selenium WebDriver](https://www.browserstack.com/guide/selenium-webdriver-tutorial) to support Page Object Design patterns. In Page Factory, testers use **@FindBy** annotation. The **initElements** method is used to initialize web elements.

* **@FindBy**: An annotation used in Page Factory to locate and declare web elements using different locators. Below is an example of declaring an element using **@FindBy**

@FindBy(id="elementId") WebElement element;

Similarly, one can use **@FindBy** with different location strategies to find web elements and perform actions on them. Below are [locators](https://www.browserstack.com/guide/locators-in-selenium) that can be used:

* + ClassName
  + CSS
  + Name
  + [Xpath](https://www.browserstack.com/guide/find-element-by-xpath-in-selenium)
  + TagName
  + [LinkText](https://www.browserstack.com/guide/find-element-by-text-using-selenium)
  + PartialLinkText
* **initElements()**: **initElements** is a static method in Page Factory class. Using the **initElements** method, one can initialize all the web elements located by **@FindBy** annotation.
* **lazy initialization:** **AjaxElementLocatorFactory** is a lazy load concept in Page Factory. This is used to identify web elements only when used in any operation or activity. The timeout of a web element can be assigned to the object class with the help of the**AjaxElementLocatorFactory**.

**Must-Read:** [TestNG Annotations in Selenium Webdriver (with Examples)](https://www.browserstack.com/guide/testng-annotations-in-selenium)

#### **Implementing Page Factory in Selenium Project**

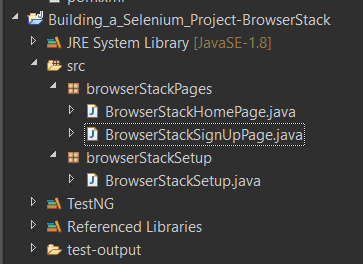
This will try to use the same project used for the POM Model. It will reuse the 2-page files and implement Page Factory.

* BrowserStackHomePage
* BrowserStackSignUpPage

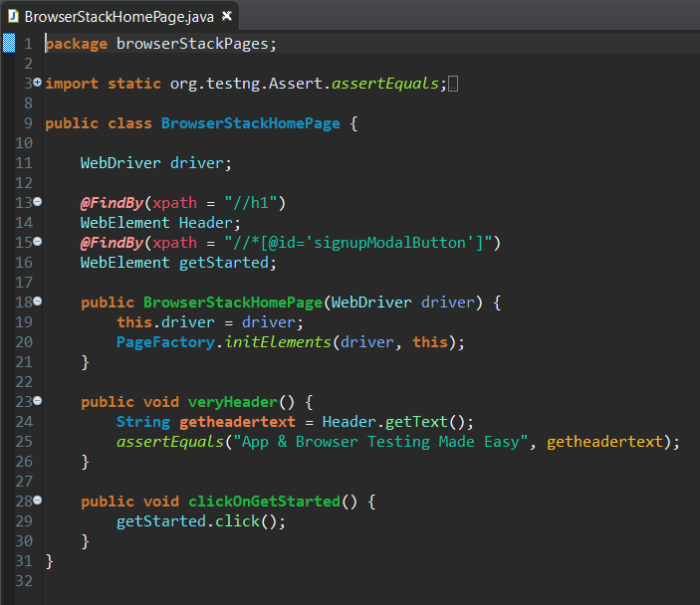
As discussed earlier, each of these files will only contain UI elements or Objects which are present on these screens along with the operations to be performed on these elements.

#### **Sample Project Structure for Page Factory**

The project structure will not be changing as the same project is being used. As already mentioned, Page Factory supports Page Object Model design pattern.



**BrowserStackHomePage Java File**



**Explanation of Code**

* **Code Line-13 to 17:** Identifying elements present on BrowserStack Home Page such as **header** and **Get Started** button using Page Factory **@FindBy** annotation
* **Code Line-23 to 30:** Performing actions on identified objects on the BrowserStack Home Page

**Code Snippet**

package browserStackPages;

import static org.testng.Assert.assertEquals;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.support.FindBy;

import org.openqa.selenium.support.PageFactory;

public class BrowserStackHomePage {

WebDriver driver;

@FindBy(xpath = "//h1")

WebElement Header;

@FindBy(xpath = "//\*[@id='signupModalButton']")

WebElement getStarted;

public BrowserStackHomePage(WebDriver driver) {

this.driver = driver;

PageFactory.initElements(driver, this);

}

public void veryHeader() {

String getheadertext = Header.getText();

assertEquals("App & Browser Testing Made Easy", getheadertext);

}

public void clickOnGetStarted() {

getStarted.click();

}

}

**BrowserStackSignUpPage Java File**



**Explanation of Code**

* **Code Line-14 to 24:** Identifying elements on [BrowserStack SignUp Page](https://www.browserstack.com/users/sign_up" \o "Sign Up on BrowserStack" \t "_blank), such as the **header** and**Get Started** button, using Page Factory **@FindBy** annotation.
* **Code Line-26 to 46:** Performing actions on identified objects on the BrowserStack SignUp Page

**Code Snippet**

package browserStackPages;

import static org.testng.Assert.assertEquals;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.support.FindBy;

import org.openqa.selenium.support.PageFactory;

public class BrowserStackSignUpPage {

WebDriver driver;

@FindBy(xpath = "//h1")

WebElement Header;

@FindBy(xpath = "//\*[@id='user\_full\_name']")

WebElement userName;

@FindBy(xpath = "//\*[@id='user\_email\_login']")

WebElement businessEmail;

@FindBy(xpath = "//\*[@id='user\_password']")

WebElement password;

public BrowserStackSignUpPage(WebDriver driver) {

this.driver = driver;

PageFactory.initElements(driver, this);

}

public void veryHeader() {

String getheadertext = Header.getText().trim();

assertEquals("Create a FREE Account", getheadertext);

}

public void enterFullName(String arg1) {

userName.sendKeys(arg1);

}

public void enterBusinessEmail(String arg1) {

businessEmail.sendKeys(arg1);

}

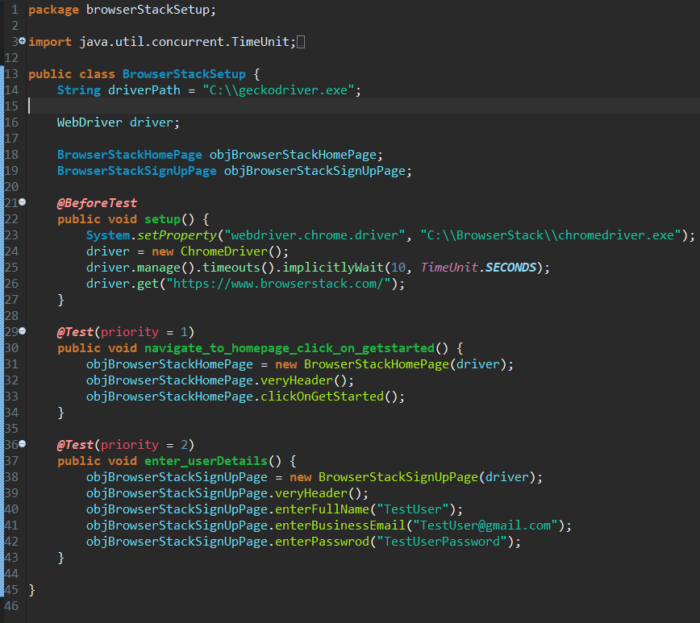
public void enterPasswrod(String arg1) {

password.sendKeys(arg1);

}

}

**BrowserStackSetup Java File**

  
**Explanation of Code**

* **Code Line-21 to 27:** Setting up of browser and website to execute our scripts
* **Code Line-29 to 43:** Initializing driver objects to BrowserStackHomePage & BrowserStackSignUpPage and performing actions on those pages.

**Code Snippet**

package browserStackSetup;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

import org.testng.annotations.BeforeTest;

import org.testng.annotations.Test;

import browserStackPages.BrowserStackHomePage;

import browserStackPages.BrowserStackSignUpPage;

public class BrowserStackSetup {

String driverPath = "C:\\geckodriver.exe";

WebDriver driver;

BrowserStackHomePage objBrowserStackHomePage;

BrowserStackSignUpPage objBrowserStackSignUpPage;

@BeforeTest

public void setup() {

System.setProperty("webdriver.chrome.driver", "C:\\BrowserStack\\chromedriver.exe");

driver = new ChromeDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.get("https://www.browserstack.com/");

}

@Test(priority = 1)

public void navigate\_to\_homepage\_click\_on\_getstarted() {

objBrowserStackHomePage = new BrowserStackHomePage(driver);

objBrowserStackHomePage.veryHeader();

objBrowserStackHomePage.clickOnGetStarted();

}

@Test(priority = 2)

public void enter\_userDetails() {

objBrowserStackSignUpPage = new BrowserStackSignUpPage(driver);

objBrowserStackSignUpPage.veryHeader();

objBrowserStackSignUpPage.enterFullName("TestUser");

objBrowserStackSignUpPage.enterBusinessEmail("TestUser@gmail.com");

objBrowserStackSignUpPage.enterPasswrod("TestUserPassword");

}

}

**Test Result**

### page object model with pagefactory in selenium

[Try Selenium Testing on a Real Device Cloud](https://www.browserstack.com/users/sign_up?ref=guide-page-object-model-in-selenium-bottom&product=automate)

### Difference between Page Object Model & Page Factory in Selenium

|  |  |
| --- | --- |
| **Page Object Model** | **Page Factory** |
| Finding web elements using **By** | Finding web elements using **@FindBy** |
| POM does not provide lazy initialization | Page Factory does provide lazy initialization |
| Page Object Model is a design pattern | PageFactory is a class that implements the Page Object Model design pattern. |
| In POM, one needs to initialize every page object individually | In PageFactory, all page objects are initialized by using the **initElements()** method |

Run the code to test the workings of the Page Object Model (POM) and Page Factory. Since these are important Selenium functions, testers need to be able to use them with ease and accuracy for [Selenium automation](https://www.browserstack.com/selenium-automation). This will help them streamline [automation testing](https://www.browserstack.com/guide/automation-testing-tutorial) efforts and get results quicker.