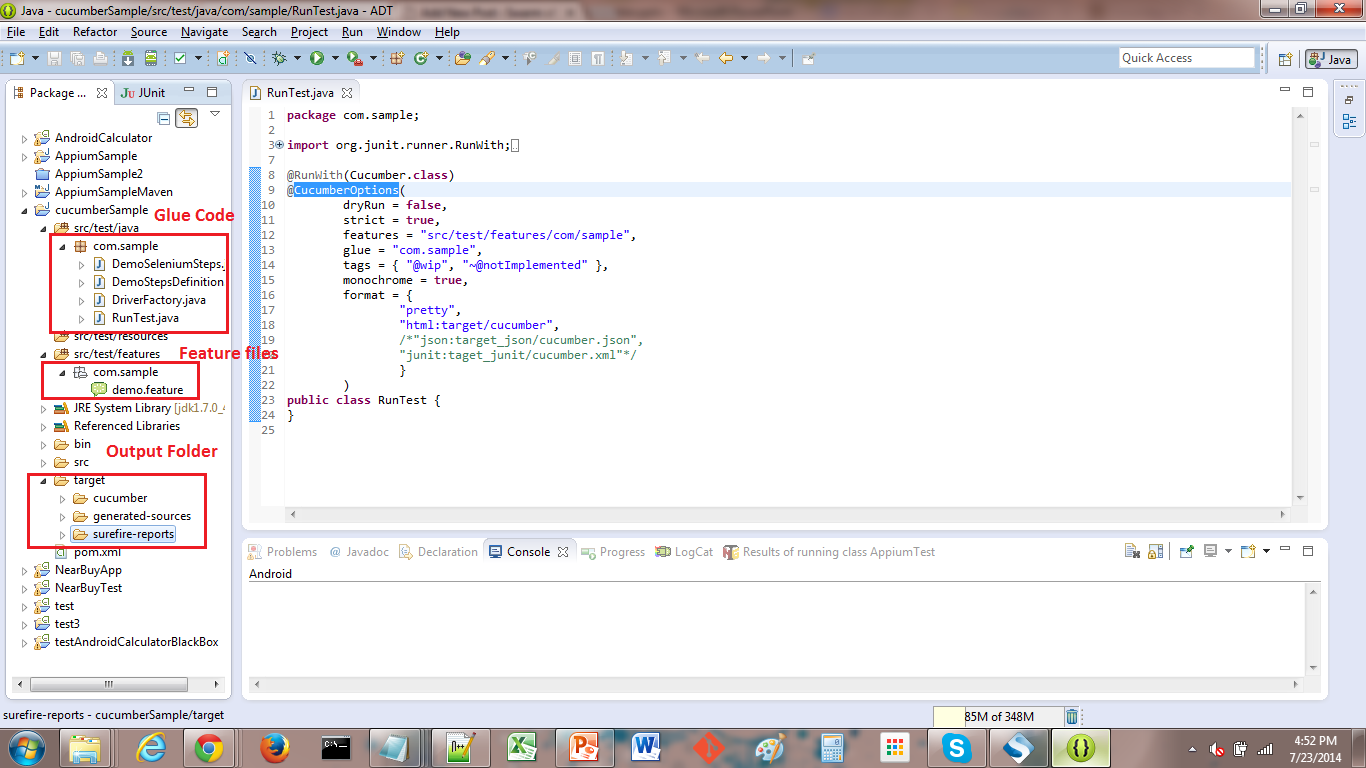
**Cucumber-JVM: All you need to know about @CucumberOptions**

When we talk about Cucumber JVM for Behavior Driven development, we often talk about Feature files, Scenarios, Background and Glue code(step definitions). There is no iota of doubt that you can’t implement BDD until you know the concepts mentioned above but other area that is really important and is very useful in day to day BDD life is @CucumberOptions.

Let’s start by understanding the structure of the cucumber project i am going to use in examples. Here is the snapshot for project structure.



To simplify the learning process, i have divided the content in 3 parts:

* Usage of @CucumberOptions
* How to use it ( code part)
* Different options available

***Usage***: @CucumberOptions annotation provides the same options as the cucumber jvm command line. for example: we can specify the path to feature files, path to step definitions, if we want to run the execution in dry mode or not etc.

           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.

***How to use it:*** Create one empty class with the @RunWith(Cucumber.class) annotation.  The options to be used are defined with the @CucumberOptions.

Executing this class as any JUnit test class will run all features found on the classpath in the same package as this class.Name of the class could be anything and most common name for this class is RunCukeTest.java

Note: @Cucumber.Options is deprecated from version 1.1.5

let’s look at the sample code for the RunCukeTest.java structure:

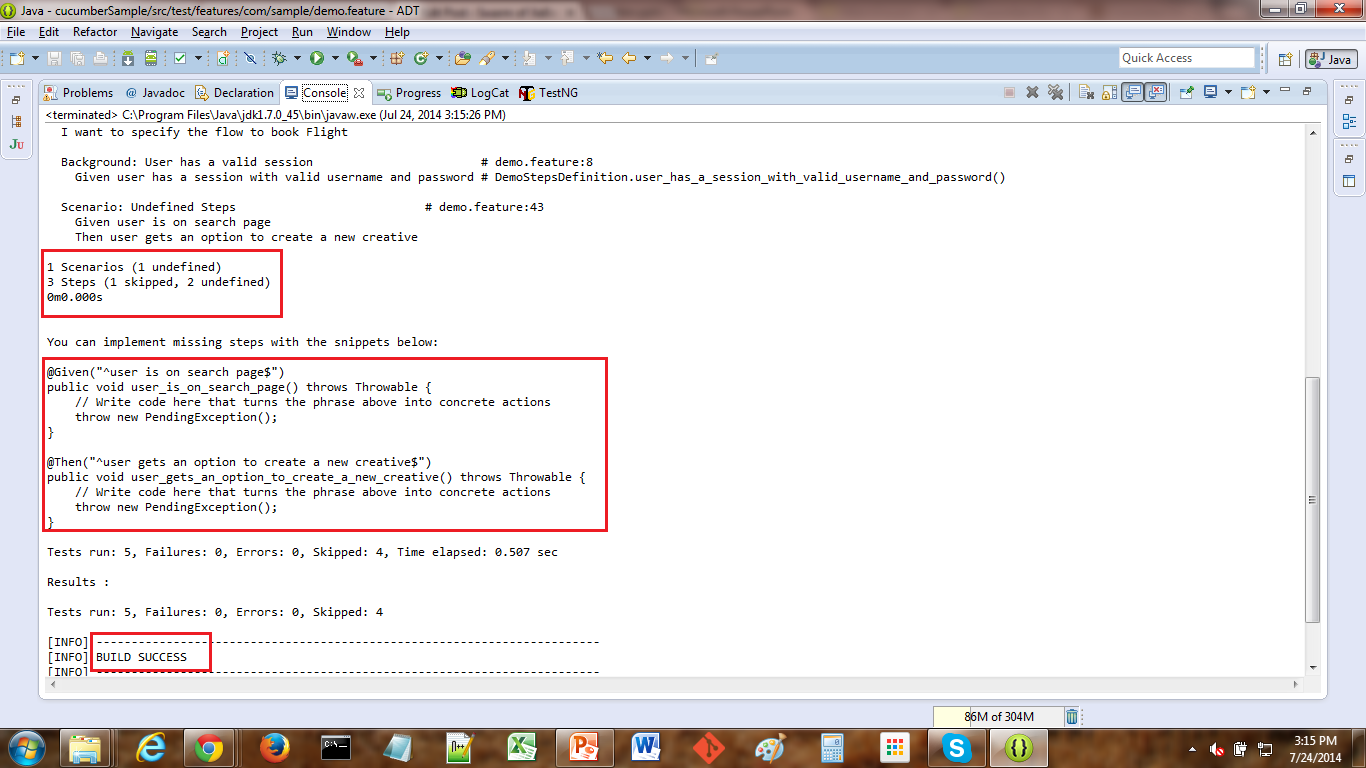
package com.sample;  
import org.junit.runner.RunWith;  
import cucumber.api.CucumberOptions;  
import cucumber.api.junit.Cucumber;  
@RunWith(Cucumber.class)  
@CucumberOptions(  
//your cucumber options goes here  
}  
)  
public class RunCukeTest {  
// This class will be empty  
}

***Different options:***

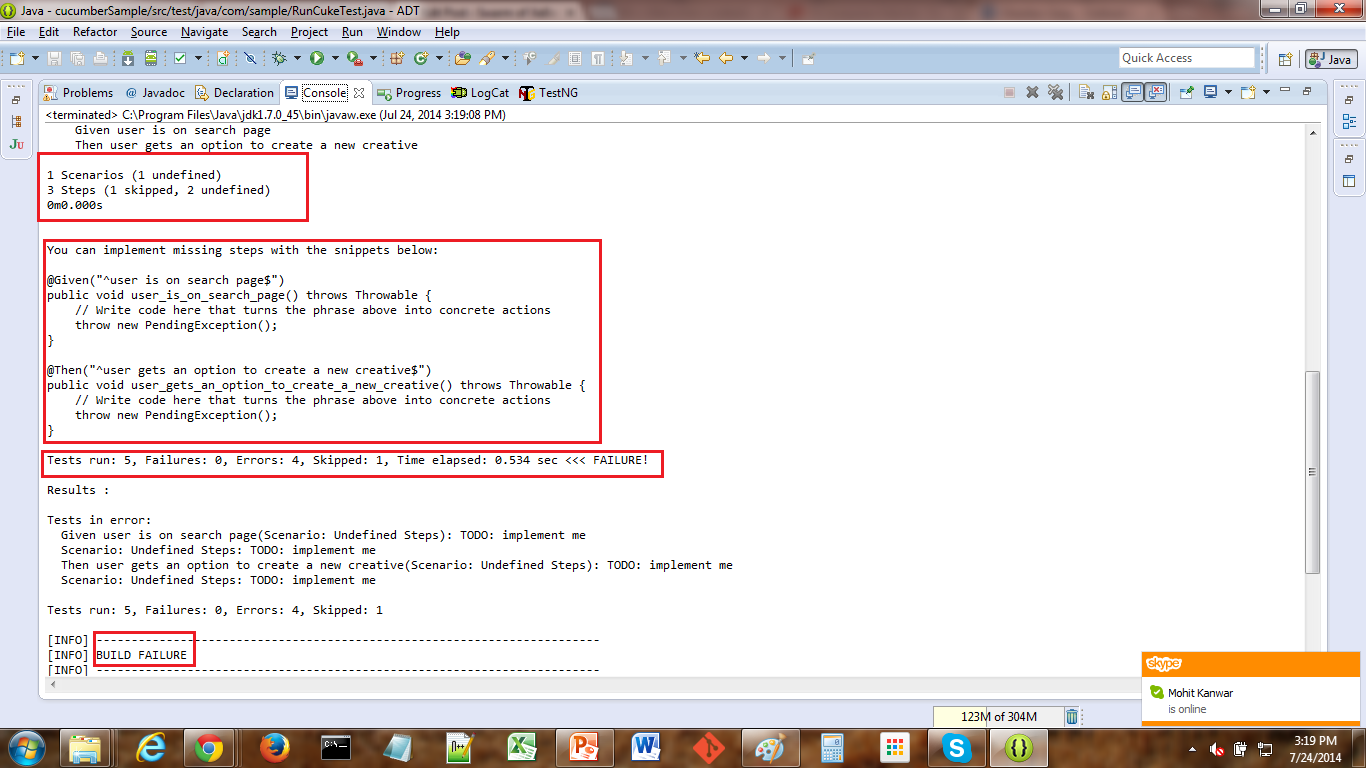
|  |  |  |
| --- | --- | --- |
| Element | Purpose | Default |
| dryRun | true (Skip execution of glue code) | FALSE |
| strict | true (will fail execution if there are undefined or pending steps) | FALSE |
| features | the paths to the feature(s) | {} |
| glue | where to look for glue code (stepdefs and hooks) | {} |
| tags | what tags in the features should be executed | {} |
| monochrome | whether or not to use monochrome output | FALSE |
| format | what formatter(s) to use | {} |

Here is the code snippet which shows hows to write these options in code format  
  
package com.sample;  
import org.junit.runner.RunWith;  
import cucumber.api.CucumberOptions;  
import cucumber.api.junit.Cucumber;  
@RunWith(Cucumber.class)  
@CucumberOptions(  
dryRun = false,  
strict = true,  
features = "src/test/features/com/sample",  
glue = "com.sample",  
tags = { "~@wip", "@executeThis" },  
monochrome = true,  
format = {  
"pretty",  
"html:target/cucumber",  
"json:target\_json/cucumber.json",  
"junit:taget\_junit/cucumber.xml"  
}  
)  
public class RunCukeTest {  
}  
  
Although the table and code above summaries things very clearly but still lets look at each option one by one to understand it better.

* ***dryRun:***if dryRun option is set to true then cucumber only checks if all the steps have their corresponding step definitions defined or not. The code mentioned in the Step definitions is not executed. This is used just to validate if we have defined a step definition for each step or not.
* ***Strict:***if strict option is set to false then at execution time if cucumber encounters any undefined/pending steps then cucumber does not fail the execution and undefined steps are skipped and BUILD is **SUCCESSFUL**. This is what the Console output looks like:

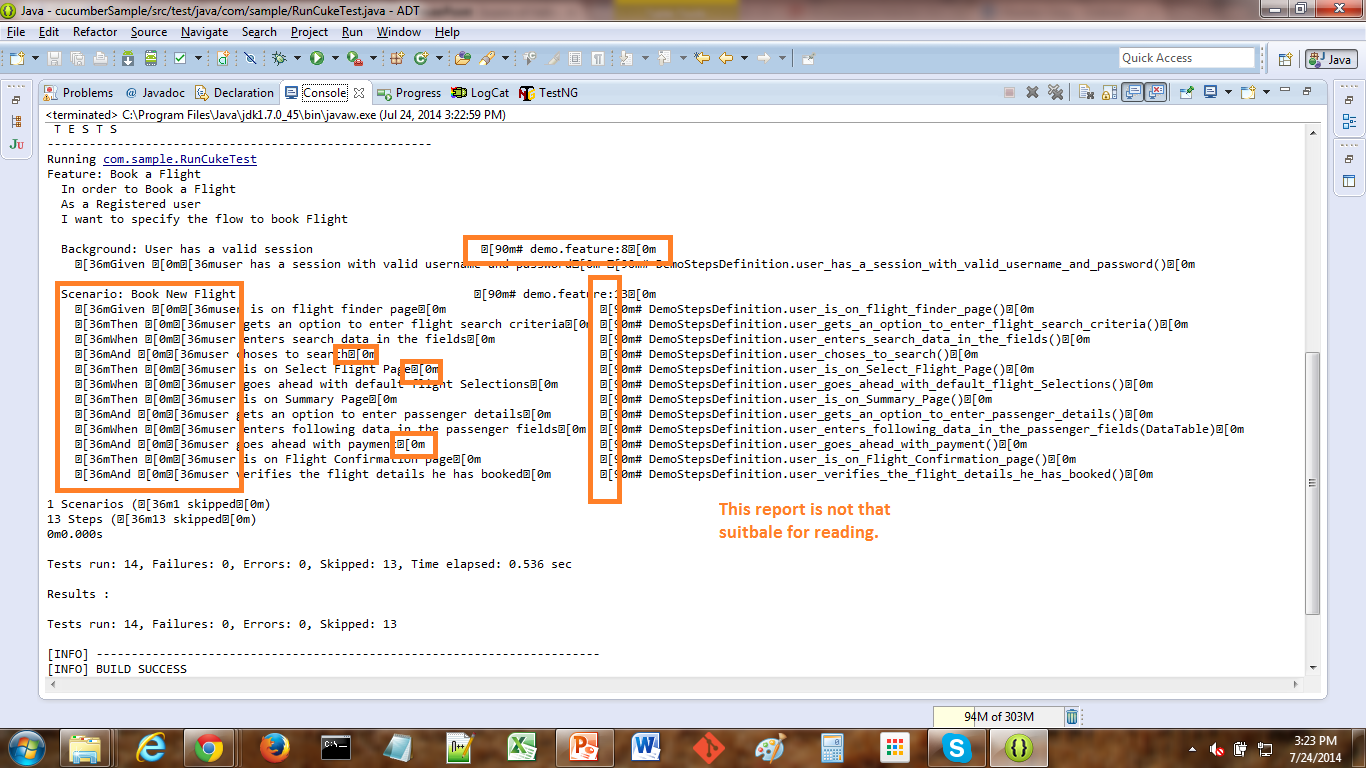


and if Strict option is set to true then at execution time if cucumber encounters any undefined/pending steps then cucumber does fails the execution and undefined steps are marked as fail and BUILD is **FAILURE**. This is what the Console output looks like:

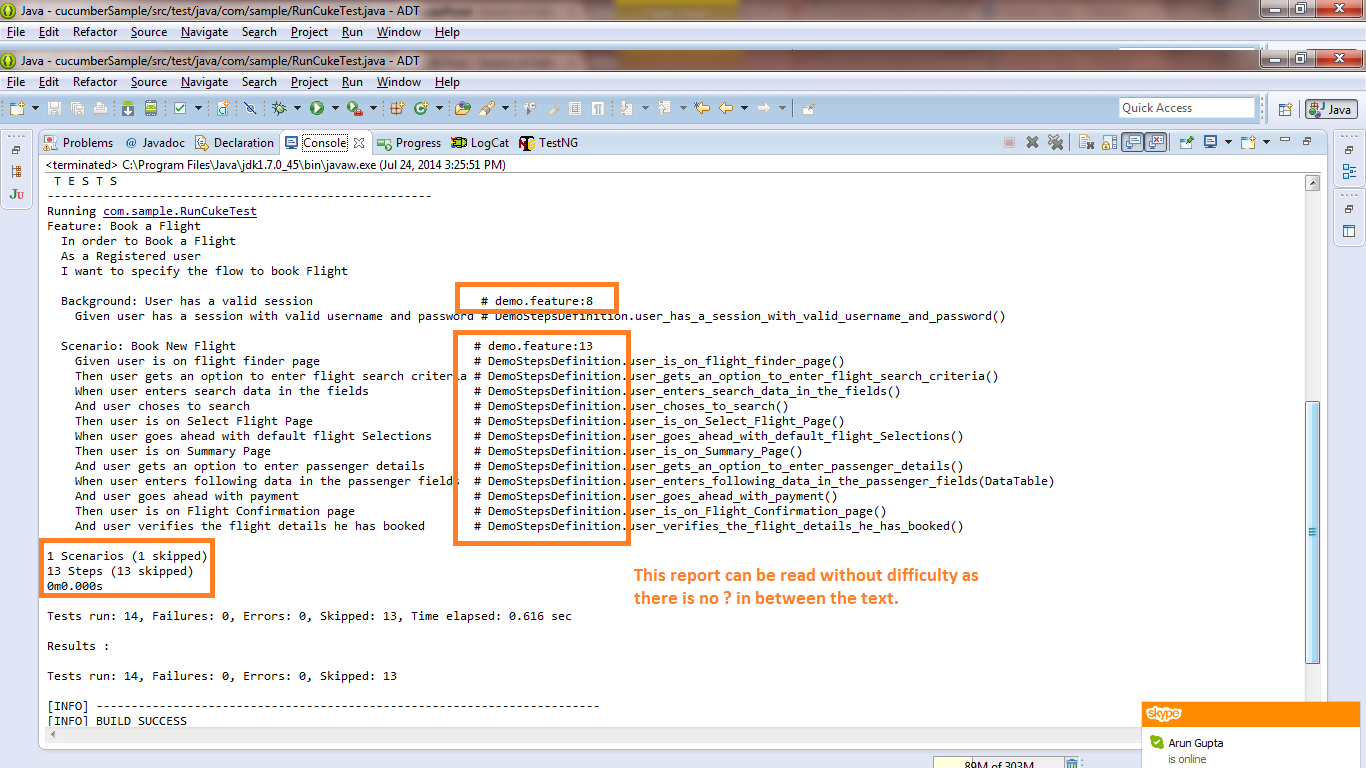


* ***Monochrome:***if monochrome option is set to False, then the console output is not as readable as it should be. may be the output images will make this more clear.

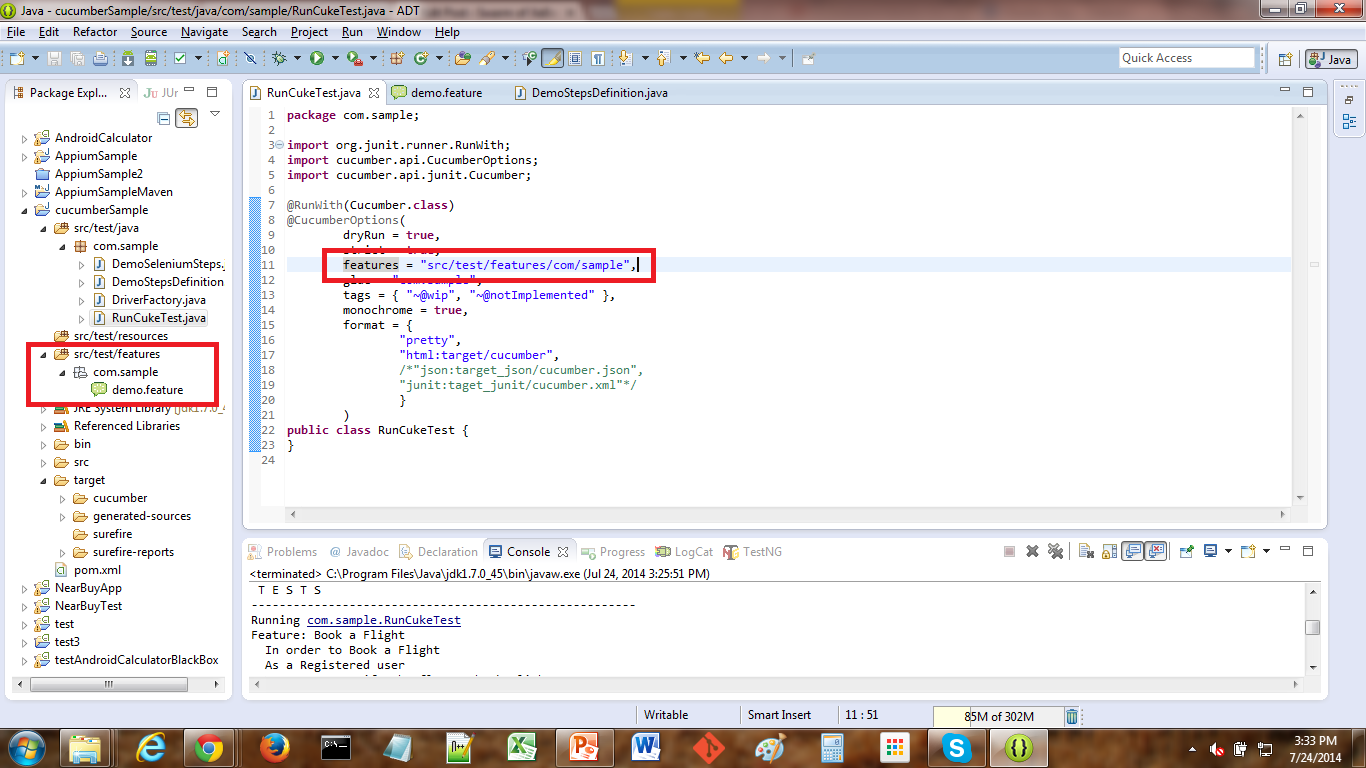
Output When monochrome option is set to false is shown in below screenshot.



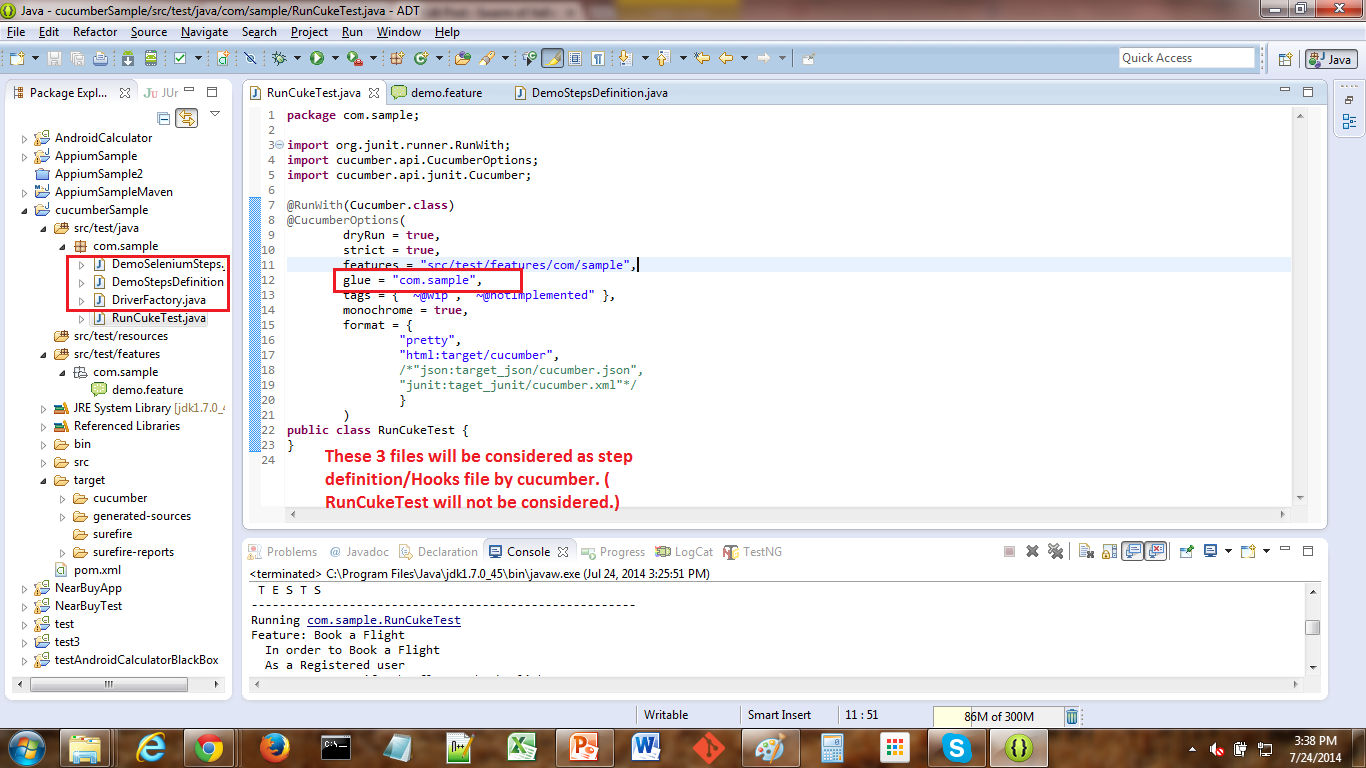
Output When monochrome option is set to true is shown in below screenshot.



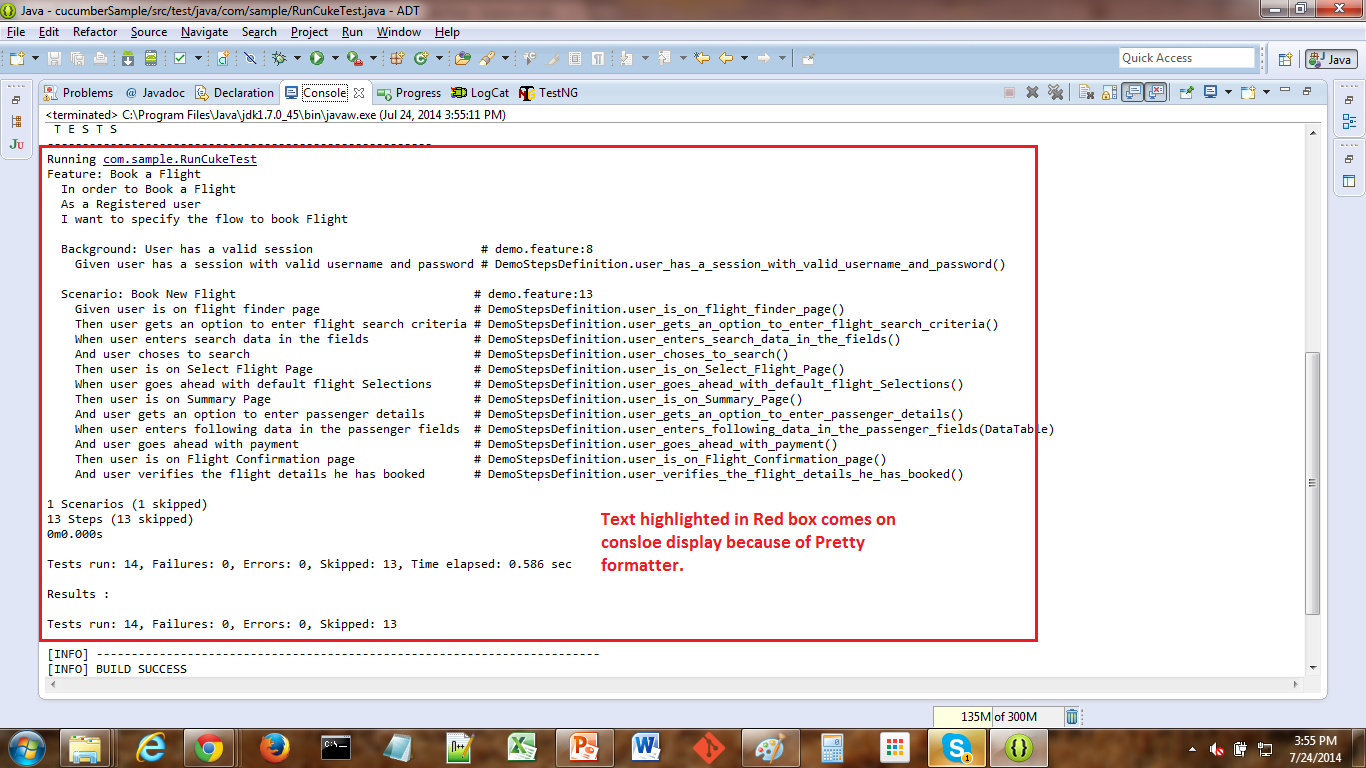
* ***Features:***features option is to specify the path to feature files. when cucumber starts execution, Cucumber looks for .feature files at the path/folder mentioned in features option. Whichever files are with .feature extension ( if there are no compilation errors) at the path/folder mentioned in features, are executed. below snapshot will make it clear on what path to define keeping in mind the project structure:



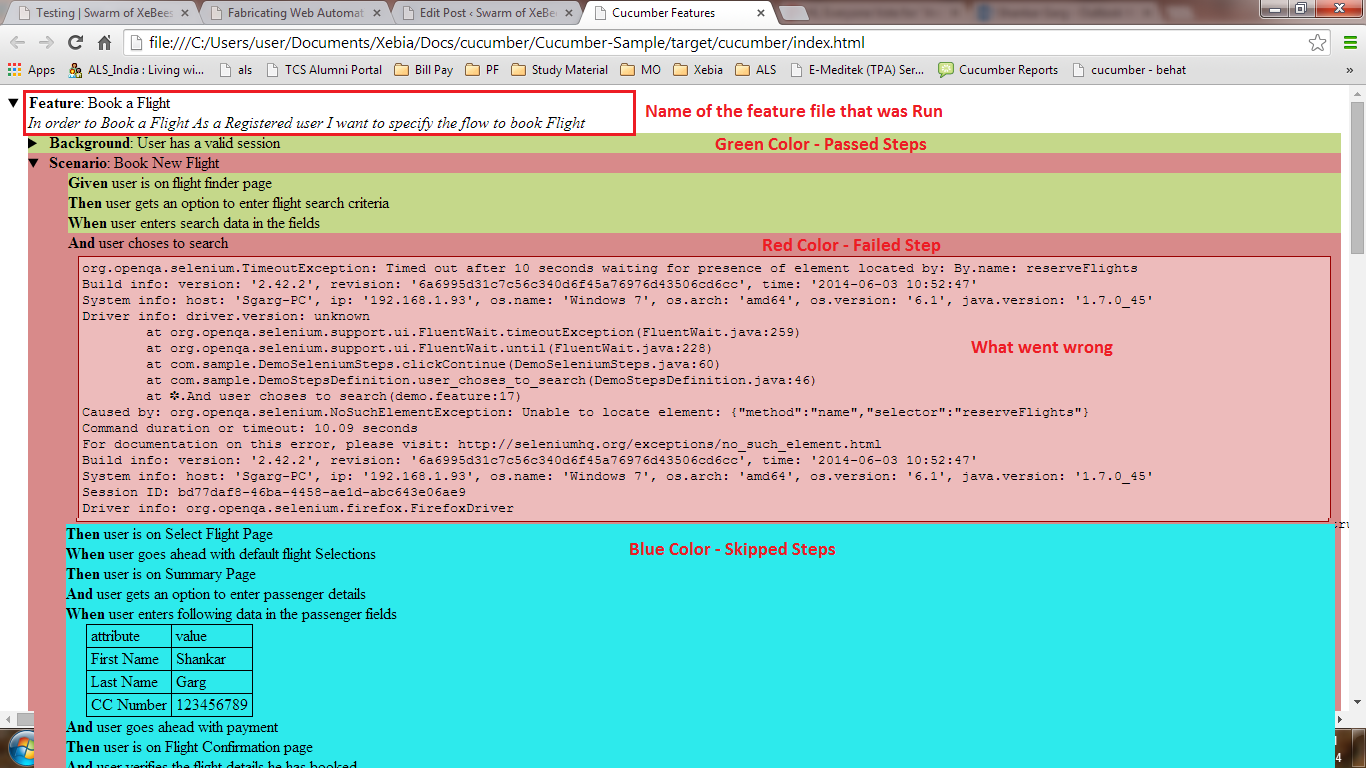
* ***Glue:***glue option is to specify where the step definitions and glue code is present. Whenever cucumber encounters a step, the cucumber looks for a step definition inside all the files present in the folder mentioned in GLUE option. This also holds true for Hooks. below snapshot will make it clear on what path to define keeping in mind the project structure:



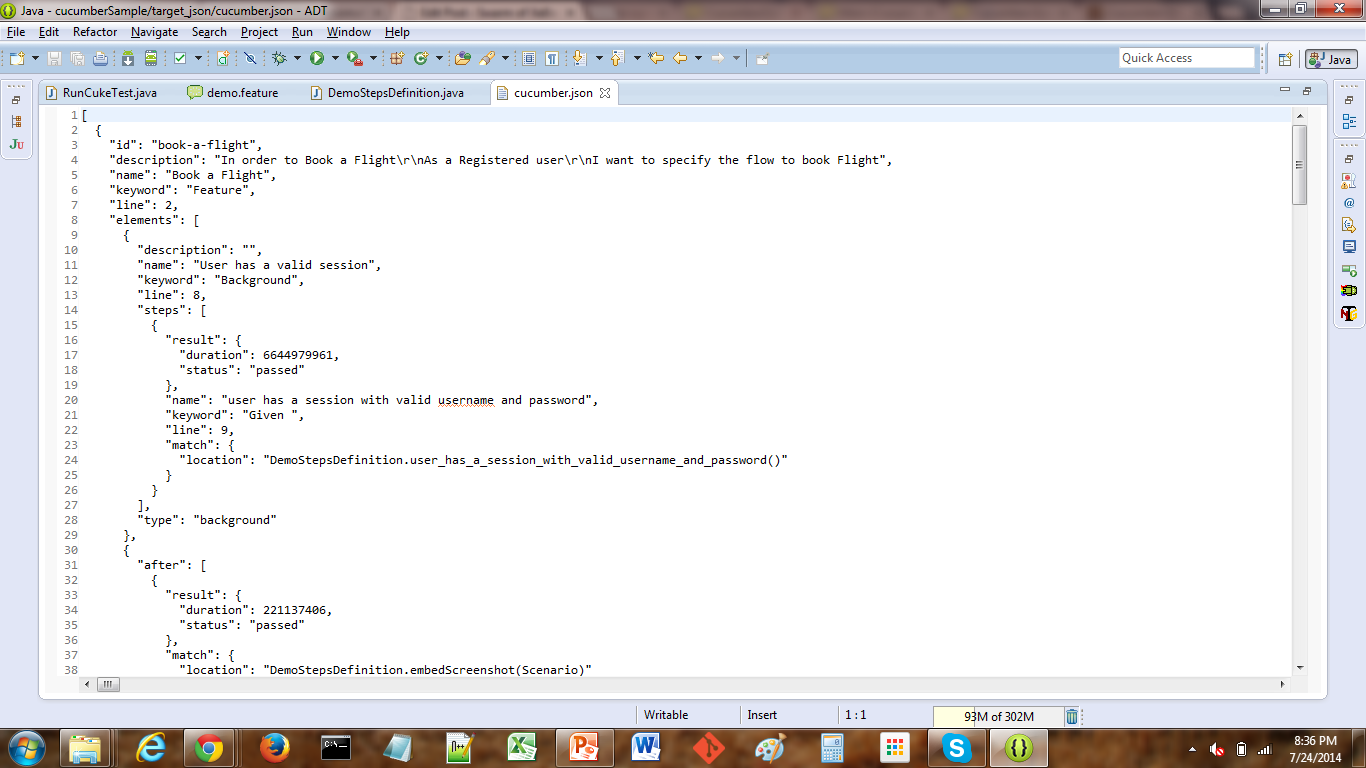
* ***Tags:***what tags in the features should be executed or for that matter what tags should not be executed. for example: as per our example, whichever feature file or scenario would be tagged with @execueThis would be executed and whichever is tagged with @wip would not be executed. because of the "~" mentioned before @wip Tag. "~" in front of any Tag tells cucumber to skip scenario/feature tagged with that Tag.
* ***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 snippet for pretty:  
    
  format = { "pretty"}



* ***HTML:***This will generate a HTML report at the location mentioned in the for-matter itself. use below code snippet:  
    
  format = { "html:target/cucumber"}
* below snapshot displays the kind of report that we will see.



* ***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 snippet:  
    
  format = {  
  "json:target\_json/cucumber.json",  
  }
* below snapshot shows the kind of report we are going to see:



* ***Junit:***This report generates XML files just like Apache Ant’s junitreport task. This XML format is understood by most Continuous Integration servers, who will use it to generate visual reports. use the below code snippet:  
  format = { "junit:taget\_junit/cucumber.xml"

below snapshot shows the kind of report we are going to see:

